

**数据库系统实验报告**

|  |  |
| --- | --- |
| 作业名称： | MiniSQL实验报告 |
| 姓 名： | 陈勇帆、汪珉凯 |
| 学 号： |  |
| 电子邮箱： |  |
| 联系电话： |  |
| 指导老师： | 孙建伶 |

2024年6月24日

目录与分工

[#1 DISK AND BUFFER POOL MANAGER（Part1） 3](#_Toc170164969)

[负责人：陈勇帆 3](#_Toc170164970)

[#1 DISK AND BUFFER POOL MANAGER（Part2） 11](#_Toc170164971)

[负责人：汪珉凯 11](#_Toc170164972)

[#2 RECORD MANAGER 21](#_Toc170164973)

[负责人：陈勇帆 21](#_Toc170164974)

[#3 INDEX MANAGER 40](#_Toc170164975)

[负责人：汪珉凯 40](#_Toc170164976)

[#4 CATALOG MANAGER 85](#_Toc170164977)

[负责人：汪珉凯 85](#_Toc170164978)

[#5 PLANNER AND EXECUTOR 100](#_Toc170164979)

[负责人：汪珉凯 100](#_Toc170164980)

[#6 RECOVERY MANAGER 110](#_Toc170164981)

[负责人：陈勇帆 110](#_Toc170164982)

# #1 DISK AND BUFFER POOL MANAGER（Part1）

负责人：陈勇帆

一、实验目的

第一个模块旨在为MiniSQL数据库管理系统设计和实现一个磁盘管理器（Disk Manager）和缓冲池管理器（Buffer Pool Manager）。这些模块负责数据库文件中数据页的分配和回收，以及数据页从磁盘到内存的调度。这部分我和队友一起完成，我负责磁盘管理器的实现，他负责缓冲池管理器的实现。其中，位图页（Bitmap Page）是Disk Manager中的一个关键组件，用于跟踪数据页的分配情况。

二、内容与功能

磁盘管理器部分主要包括位图页的实现和磁盘数据页管理

**（一）位图页**

（1）位图页用于管理一段连续的数据页，通过位图（Bitmap）记录每个数据页的分配状态。位图页由元信息和位图数据两部分组成。元信息包含已分配页的数量（page\_allocated\_）和下一个空闲页的索引（next\_free\_page\_）。位图数据记录数据页的分配情况，每个比特对应一个数据页，0表示空闲，1表示已分配。位图页和其管理的数据页作为整体构成分区（extent）。

（2）结合头文件具体分析其功能：

①类模板 BitmapPage

BitmapPage类模板定义了一个支持不同页大小的位图页。模板参数PageSize表示页的大小，以字节为单位。

②私有成员变量

MAX\_CHARS：表示位图页中用于存储位图数据的最大字节数。计算公式为PageSize - 2 \* sizeof(uint32\_t)，减去的部分是用于存储page\_allocated\_和next\_free\_page\_的空间。

page\_allocated\_：记录当前已分配的页的数量。

next\_free\_page\_：记录下一个空闲页的索引，用于加速查找下一个可分配的页。

bytes[MAX\_CHARS]：用于存储位图数据的数组，每个bit对应一个页的分配状态。

③公共成员函数

GetMaxSupportedSize：返回位图页能记录的最大页数，计算公式为8 \* MAX\_CHARS，其中每个字节可以记录8个页。

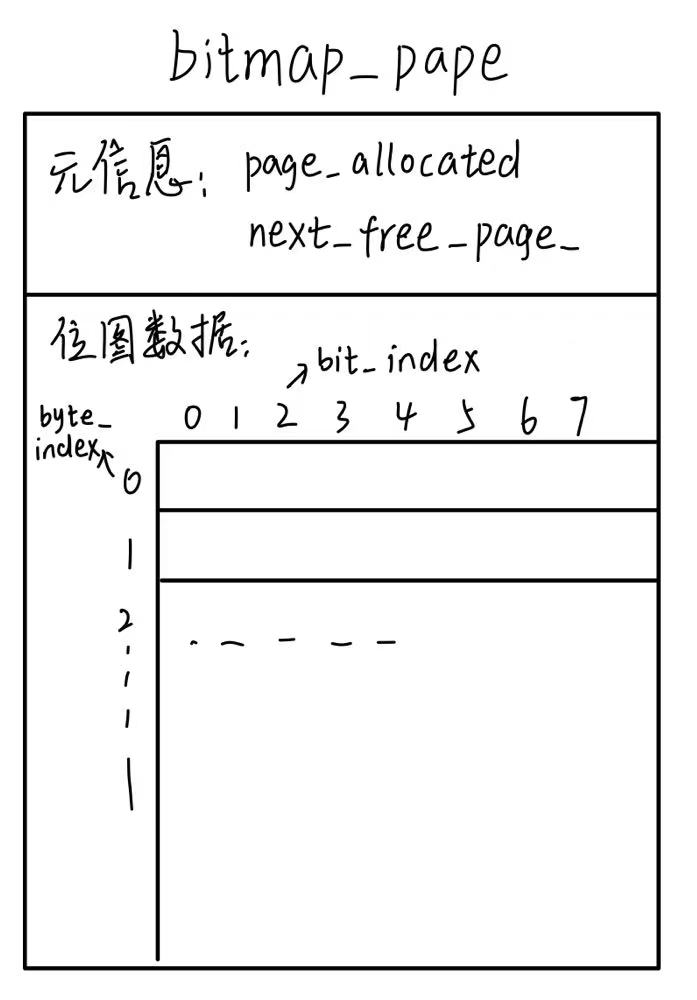
AllocatePage：分配一个空闲页，并通过引用参数page\_offset返回所分配页在位图页管理空间中的偏移量。返回值为true表示分配成功，false表示失败。

DeAllocatePage：回收指定的页。参数page\_offset为页在位图页管理空间中的偏移量。返回值为true表示回收成功，false表示失败。

IsPageFree：检查指定页是否空闲。参数page\_offset为页在位图页管理空间中的偏移量。返回值为true表示该页空闲，false表示已分配。

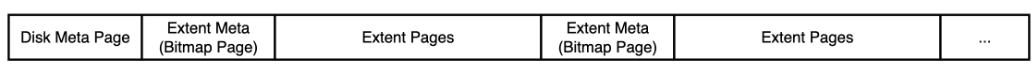
④私有成员函数

IsPageFreeLow：检查位图页中指定字节和比特位置的比特是否为0。参数byte\_index和bit\_index分别表示行索引和列索引。返回值为true表示该bit为0，即页空闲，false表示该bit为1，即页已分配。具体可结合下图理解：



**（二）磁盘数据页管理**

（1）底层模块Disk Manager是负责管理数据页的分配和回收，以及数据的读写。Disk Manager利用额外的元信息页disk meta page来串联多个分区，由其来管理每个分区的位图页来跟踪和管理数据库文件中的数据页。如下图所示：



通过这个机制，Disk Manager能够支持大规模的数据库存储，并确保数据页的有效利用和快速访问。

（2）结合DiskManager类的头文件具体分析：

①私有成员变量

meta\_data\_[PAGE\_SIZE]：用于存储元数据页的缓存。

其他成员与本阶段实验无关，先忽略。

②公共成员函数

explicit DiskManager：构造函数，打开指定的数据库文件。

~DiskManager：析构函数，确保在对象销毁时关闭数据库文件。

ReadPage：读取逻辑页号对应的数据页。

WritePage：写入数据到逻辑页号对应的数据页。

AllocatePage：分配一个空闲的数据页，返回其逻辑页号。

DeAllocatePage：回收指定逻辑页号的数据页。

IsPageFree：检查指定逻辑页号的数据页是否空闲。

void Close：关闭数据库文件。

\*GetMetaData：返回元数据页的缓存地址，用于调试。

这些函数实现了对数据库文件的读写操作、数据页的分配和回收、以及元数据的管理。

③私有成员函数

GetFileSize：获取指定文件的大小。

ReadPhysicalPage：读取物理页号对应的数据页。

WritePhysicalPage：写入数据到物理页号对应的数据页。

MapPageId：将逻辑页号映射为物理页号。映射的具体原理如下图：



可见逻辑页号中不为元数据页和位图页分配页号，具体映射公式：

physical\_page = logical\_page + disk\_meta\_page + num\_of\_bitmap\_page

= logical\_page\_id + 1 + logical\_page\_id / BITMAP\_SIZE + 1；

其中BITMAP\_SIZE指一个尾页图管理的数据页数量。

这些函数实现了具体的辅助功能，包括文件大小获取、物理页读写和逻辑页号到物理页号的映射。

（3）结合disk\_file\_meta\_page.h头文件可以知道磁盘元数据页相关信息：

①成员变量：

num\_allocated\_pages\_：已分配页面的数量，初始值为0。

num\_extents\_：extent的数量，每个extent由一个bitmap和BITMAP\_SIZE个页面组成，初始值为0。

extent\_used\_page\_：一个数组，记录每个extent中已使用页面的数量，是一个变长数组，它的大小由实际分配的内存决定。

②成员函数：

GetExtentNums：返回extent的数量。

GetAllocatedPages：返回已分配页面的数量。

GetExtentUsedPage：返回指定extent中已使用页面的数量。如果请求的extent ID超过了当前extent的数量，返回0。

后续页面分配的代码中需要通过元数据页的信息来加速功能实现。

页面读取的代码已经实现，在这一部分需要实现的主要是有关页面分配和物理、逻辑页号映射的函数。

三、代码实现与分析

**（一）位页图**

① IsPageFree 函数和IsPageFreeLow 函数

前者先对传入的合法的页偏移计算其字节索引和bit索引，然后传入后者判断是否空闲。

template <size\_t PageSize>

    bool BitmapPage<PageSize>::IsPageFree(uint32\_t page\_offset) const {

    if (page\_offset >= MAX\_CHARS \* 8) return false;  // 检查页偏移是否合法

    uint32\_t byte\_index = page\_offset / 8;  // 计算字节索引

    uint8\_t bit\_index = page\_offset % 8;  // 计算bit位置

    return IsPageFreeLow(byte\_index, bit\_index);  // 调用辅助函数检查页是否空闲

    }

    template <size\_t PageSize>

    bool BitmapPage<PageSize>::IsPageFreeLow(uint32\_t byte\_index, uint8\_t bit\_index) const {

return !(bytes[byte\_index] & (1 << bit\_index));

// 检查bit位置是否为0，思路是设置仅对应位置为1的字节，与bytes数组取并，使得取并的结果仅取决于bytes对应bit是否为0，为0表示空闲，返回1（true）

    }

②AllocatePage函数

分配当前最小空闲页并传入page\_offset，并更新得到下一个最小空闲页。

    template <size\_t PageSize>

    bool BitmapPage<PageSize>::AllocatePage(uint32\_t &page\_offset) {

    if (page\_allocated\_ >= MAX\_CHARS \* 8) return false;  // 检查是否还有空闲页

    page\_offset = next\_free\_page\_;  // 传参记录分配页的位置

    uint32\_t byte\_index = next\_free\_page\_ / 8;  // 计算分配页所在的字节索引

    uint8\_t bit\_index = next\_free\_page\_ % 8;  // 计算分配页在字节中的bit位置

    bytes[byte\_index] |= (1 << bit\_index);  // 将对应bit置为1

    page\_allocated\_++;  // 更新已分配页的数量

    // 更新下一个空闲页的索引

for (uint32\_t i = page\_offset; i < MAX\_CHARS \* 8; i++) {

// 从分配的页面开始往后查找下一个最小空闲页

        if (IsPageFree(i)) {

        next\_free\_page\_ = i;

        break;

        }

    }

    return true;

    }

③DeAllocatePage函数

通过位页图置0来释放对应页面，并条件更新下一个最小空闲页

    template <size\_t PageSize>

    bool BitmapPage<PageSize>::DeAllocatePage(uint32\_t page\_offset) {

    if (page\_offset >= MAX\_CHARS \* 8) return false;  // 检查页偏移是否合法

    if (IsPageFree(page\_offset)) return false;  // 如果页已经空闲，返回false

    uint32\_t byte\_index = page\_offset / 8;  // 计算字节索引

    uint8\_t bit\_index = page\_offset % 8;  // 计算bit位置

    bytes[byte\_index] &= ~(1 << bit\_index);  // 将对应bit置为0

    page\_allocated\_--;  // 更新已分配页的数量

    next\_free\_page\_ = (page\_offset < next\_free\_page\_)? page\_offset : next\_free\_page\_;

 // 若释放的页比当前next\_free\_page小，则更新

    return true;

    }

这里尤其需要注意next\_free\_page的处理，倘若每次更新都是从offset=0开始找，固然能得到最小空闲页但可能耗时长，因此这里选择从最小空闲页开始往后更新，同时在deallocatePage函数中保证下个空闲页为最小的，以免错过更小的空闲页造成分配失败。

**（二）磁盘数据页管理**

①IsPageFree函数

通过找到数据页对应的物理页和偏移量，代入bitmap中的该函数得到结果。

    bool DiskManager::IsPageFree(page\_id\_t logical\_page\_id) {

    if(logical\_page\_id > MAX\_VALID\_PAGE\_ID) return false;

    char data[PAGE\_SIZE];

    // 计算bitmap页面的物理页面ID

    page\_id\_t physical\_bitmap\_page\_id = logical\_page\_id + 1 + logical\_page\_id / BITMAP\_SIZE - logical\_page\_id % BITMAP\_SIZE;

    ReadPhysicalPage(physical\_bitmap\_page\_id, data);

    auto\* bitmap = reinterpret\_cast<BitmapPage<PAGE\_SIZE> \*>(data);

    // 检查页面是否空闲

    return bitmap->IsPageFree(logical\_page\_id % BITMAP\_SIZE);

}

②AllocatePage函数

    page\_id\_t DiskManager::AllocatePage() {

    auto meta\_page = reinterpret\_cast<DiskFileMetaPage\*>(meta\_data\_);

    // 检查是否超过最大合法页面ID

    if(meta\_page->GetAllocatedPages() >= MAX\_VALID\_PAGE\_ID) return INVALID\_PAGE\_ID;

    uint32\_t extent\_i;

    // 找到有空闲页面的extent

    for(extent\_i = 0; extent\_i < meta\_page ->GetExtentNums(); extent\_i++){

        if(meta\_page->extent\_used\_page\_[extent\_i]<BITMAP\_SIZE) break;

    }

    // 计算bitmap页面的物理页面ID

    page\_id\_t physical\_bitmap\_page\_id = static\_cast<page\_id\_t>(extent\_i \* (1 + BITMAP\_SIZE)) + 1;

    char data[PAGE\_SIZE];

    ReadPhysicalPage(physical\_bitmap\_page\_id, data);

    auto\* bitmap = reinterpret\_cast<BitmapPage<PAGE\_SIZE> \*>(data);

    uint32\_t page\_offset;

    // 在位图中分配页面

    bitmap->AllocatePage(page\_offset);

    WritePhysicalPage(physical\_bitmap\_page\_id, data);

    // 更新元数据

    meta\_page->num\_allocated\_pages\_++;

    meta\_page->extent\_used\_page\_[extent\_i]++;

    if(extent\_i == meta\_page->GetExtentNums()) meta\_page->num\_extents\_++;

    page\_id\_t logical\_page\_id = extent\_i \* BITMAP\_SIZE + page\_offset;

    return logical\_page\_id;

}

③DeAllocatePage函数

    void DiskManager::DeAllocatePage(page\_id\_t logical\_page\_id) {

    auto meta\_page = reinterpret\_cast<DiskFileMetaPage\*>(meta\_data\_);

    // 计算bitmap页面的物理页面ID

    page\_id\_t physical\_bitmap\_page\_id = logical\_page\_id + 1 + logical\_page\_id / BITMAP\_SIZE - logical\_page\_id % BITMAP\_SIZE;

    char data[PAGE\_SIZE];

    ReadPhysicalPage(physical\_bitmap\_page\_id, data);

    auto\* bitmap = reinterpret\_cast<BitmapPage<PAGE\_SIZE> \*>(data);

    // 在位图中取消分配页面

    bitmap->DeAllocatePage(logical\_page\_id % BITMAP\_SIZE);

    WritePhysicalPage(physical\_bitmap\_page\_id, data);

    // 更新元数据

    meta\_page->num\_allocated\_pages\_--;

    meta\_page->extent\_used\_page\_[logical\_page\_id / BITMAP\_SIZE]--;

}

以上两个函数的逻辑基本和bitmap相同，同样可以用变量’next\_free\_extent’来加快实现，但是原元数据页中未提供此变量，为尽可能减少头文件改动，这里也未做实现。

④：MapPageId函数

    page\_id\_t DiskManager::MapPageId(page\_id\_t logical\_page\_id) {

    // 计算bitmap页面数量

    page\_id\_t bitmap\_num = static\_cast<page\_id\_t>(logical\_page\_id / BITMAP\_SIZE) + 1;

    // 计算物理页面ID

return logical\_page\_id + 1 + bitmap\_num;

// physical\_page = logical\_page + disk\_meta\_page + num\_of\_bitmap\_page;

    }

四、测试代码分析与结果

**（一）代码分析**

这部分代码通过disk\_manager\_test.cpp来测试，分为两部分：

（1）BitMapPageTest 测试：该测试主要用于验证 BitmapPage 类的页面分配和释放功能。具体步骤：

①初始化和页面空闲性检查

初始化一个大小为 512 字节的缓冲区，并将其全部置为零；将缓冲区转换为 BitmapPage<size> 对象；获取 BitmapPage 能记录的最大页面数 (num\_pages)；验证每一个页面初始都是空闲的 (IsPageFree 返回 true)。

②分配页面并检查

分配 num\_pages 数量的页面，并将每次分配的页面偏移量 (ofs) 存入一个集合中，检查每次分配的页面偏移量都是唯一的；当所有页面都分配完后，尝试再次分配页面，验证其失败。

③释放页面并重新分配

释放页面 233，验证释放成功；尝试重新分配一个页面，检查其偏移量是否为 233。

④释放所有页面并再次分配

释放所有已分配的页面，并验证每次释放后的再次释放失败；再次分配所有页面，验证分配成功；尝试再次分配页面，验证其失败。

（2）FreePageAllocationTest 测试：该测试主要用于验证 DiskManager 类的页面分配和释放功能，并检查元数据页的正确性。具体步骤：

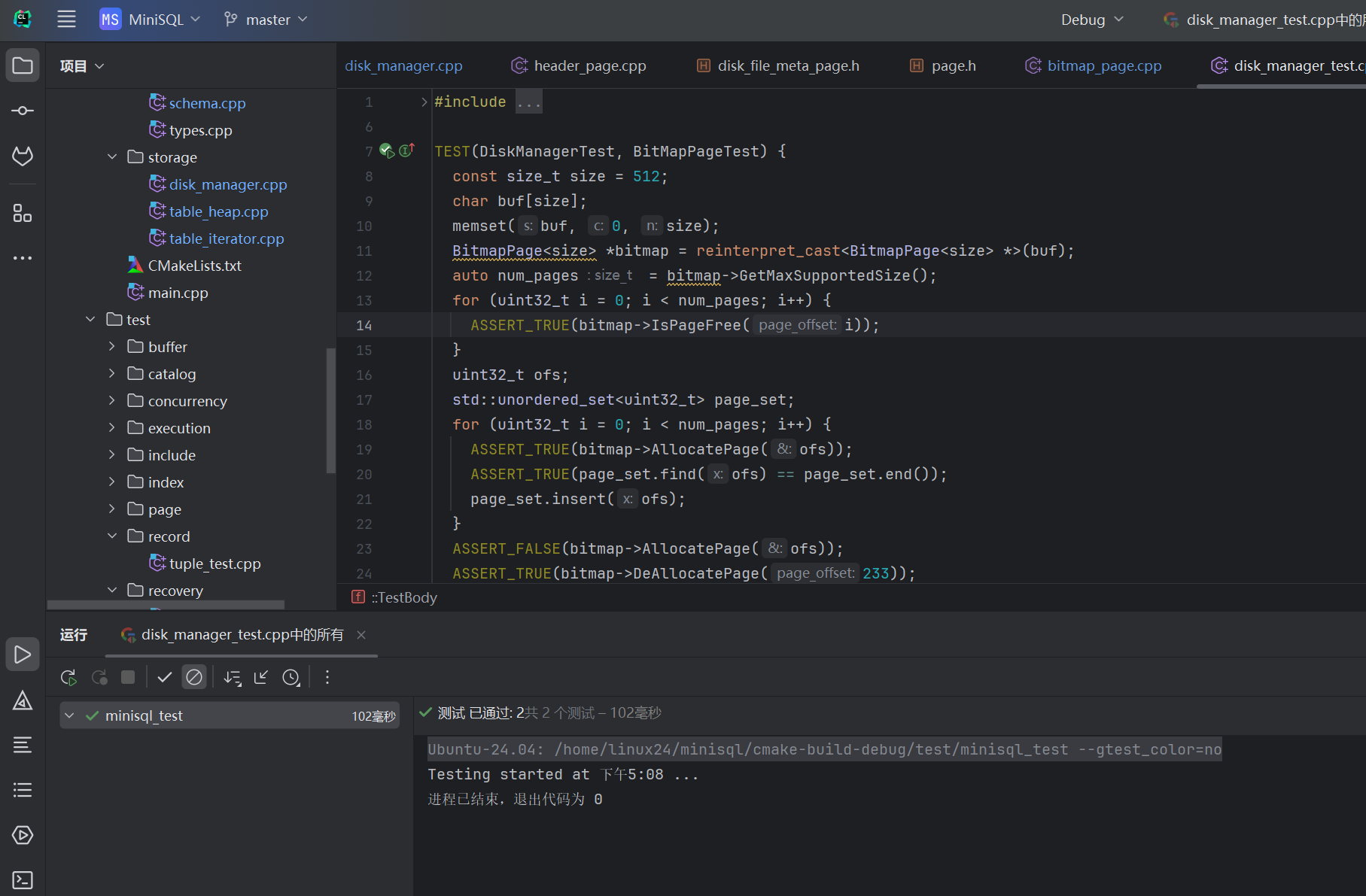
①初始化 DiskManager 并分配页面

初始化一个 DiskManager 对象并创建测试数据库文件；分配 BITMAP\_SIZE \* extent\_nums 数量的页面；检查每次分配的页面ID是否按顺序增加；检查分配页面后元数据页的 num\_extents\_、num\_allocated\_pages\_ 和 extent\_used\_page\_ 是否正确更新。

②释放特定页面并验证元数据页

释放特定的页面，并检查释放后的元数据页；查总的已分配页面数是否减少5；检查每个 extent 中已使用页面数是否正确更新。

**（二）结果：**如下图所示，测试通过。



# #1 DISK AND BUFFER POOL MANAGER（Part2）

负责人：汪珉凯

一、内容与功能

（1）buffer

Buffer Pool Manager是数据库系统中管理内存中数据页的关键组件之一。它的主要任务是在内存中存储从磁盘读取的数据页，并在需要时将数据页转储回磁盘，以及管理内存中数据页的替换和释放。

主要函数功能描述：

1. FetchPage(page\_id):

- 根据逻辑页号 `page\_id` 获取对应的数据页。

- 如果数据页已经在内存中，则返回该数据页的指针。

- 如果数据页不在内存中：

- 如果内存中有空闲页（free\_list\_），则从中分配一个空闲页，并将数据页读取到该空闲页中。

- 如果没有空闲页可用，并且需要替换页面，则使用替换算法（如LRUReplacer）选择一个牺牲页（victim page）替换出去，并将其内容写回磁盘（如果是脏页）。

- 如果无法替换任何页（没有空闲页也无法替换），则返回 nullptr 表示失败。

2. NewPage(&page\_id):

- 分配一个新的数据页，并将其逻辑页号存储在 `page\_id` 中返回。

- 分配的新页应该被固定（pin）在内存中，即其 pin\_count\_ 应该为至少 1。

3. UnpinPage(page\_id, is\_dirty):

- 取消对数据页的固定。

- 如果 `is\_dirty` 为 true，则表示数据页内容已被修改（变为脏页），需要将其标记为脏页，以便在替换时将其内容写回磁盘。

4. FlushPage(page\_id):

- 将指定数据页的内容写回磁盘。

- 不管该数据页是否被固定，都应该将其内容写回磁盘。

5. DeletePage(page\_id):

- 释放一个数据页，使其不再被管理。

- 如果数据页是脏页，应先将其内容写回磁盘，然后再释放。

6. FlushAllPages():

- 将所有数据页的内容写回磁盘。

- 应确保所有数据页都被正确地写回磁盘，即使某些数据页仍被固定。

（2）LRU

LRUReplacer 是一个基于LRU（Least Recently Used，最近最少使用）算法实现的替换器，用于管理 Buffer Pool 中数据页的替换顺序。它跟踪每个数据页的访问情况，并在需要替换数据页时提供一种有效的选择机制。

主要函数功能描述：

LRUReplacer::Victim(\*frame\_id):

当需要替换数据页时，调用此函数选择最近最少被访问的数据页。

如果存在可替换的数据页，将选中的数据页的页帧号存储在 frame\_id 中，并返回 true。

如果没有可替换的数据页（即所有数据页都被固定），则返回 false。

LRUReplacer::Pin(frame\_id):

将指定页帧号 frame\_id 对应的数据页固定（pin），从替换列表 (lru\_list\_) 中移除。

固定的数据页不会在替换时被选择。

LRUReplacer::Unpin(frame\_id):

将指定页帧号 frame\_id 对应的数据页解除固定（unpin），放入替换列表 (lru\_list\_) 中。

当数据页的引用计数减少到零时，需要调用此函数，使得数据页可以在需要时被替换。

LRUReplacer::Size():

返回当前替换器中可替换的数据页数量。

即返回替换列表 (lru\_list\_) 的大小。

二、代码实现与分析

1、buffer pool manager

1. */\*\**
2. \* TODO: Student Implement
3. \*/
4. Page \*BufferPoolManager::FetchPage(page\_id\_t page\_id) {
5. *// 1.     Search the page table for the requested page (P).*
6. *// 1.1    If P exists, pin it and return it immediately.*
7. *// 1.2    If P does not exist, find a replacement page (R) from either the free list or the replacer.*
8. *//        Note that pages are always found from the free list first.*
9. *// 2.     If R is dirty, write it back to the disk.*
10. *// 3.     Delete R from the page table and insert P.*
11. *// 4.     Update P's metadata, read in the page content from disk, and then return a pointer to P.*
12. if(page\_table\_.find(page\_id) != page\_table\_.end()){
13. frame\_id\_t frame\_id = page\_table\_[page\_id];
14. pages\_[frame\_id].pin\_count\_++;
15. replacer\_->Pin(frame\_id);
16. return &pages\_[frame\_id];
17. }*//if page is not in the page table*
18. frame\_id\_t frame\_id;
19. if(!free\_list\_.empty()) {
20. *//find a replacement page (R) from either the free list or the replacer*
21. frame\_id = free\_list\_.back();
22. free\_list\_.pop\_back();
23. }
24. else {
25. *//if the free list is empty, find a replacement page (R) from the replacer*
26. if(!replacer\_->Victim(&frame\_id)){
27. return nullptr;
28. }
29. else {
30. *//if R is dirty, write it back to the disk*
31. if(pages\_[frame\_id].is\_dirty\_) {
32. disk\_manager\_->WritePage(pages\_[frame\_id].page\_id\_, pages\_[frame\_id].GetData());
33. pages\_[frame\_id].is\_dirty\_ = false;
34. }
35. page\_table\_.erase(pages\_[frame\_id].page\_id\_);
36. }
37. }
38. auto result = pages\_[frame\_id].GetData();
39. pages\_[frame\_id].page\_id\_ = page\_id;
40. pages\_[frame\_id].pin\_count\_ = 1;
41. pages\_[frame\_id].is\_dirty\_ = false;
42. page\_table\_[page\_id] = frame\_id;
43. disk\_manager\_->ReadPage(page\_id, result);
44. return &pages\_[frame\_id];
45. }
46. */\*\**
47. \* TODO: Student Implement
48. \*/
49. Page \*BufferPoolManager::NewPage(page\_id\_t &page\_id) {
50. *// both free list and replacer are empty*
51. frame\_id\_t frame\_id0=-1;
52. *// from freelist*
53. if (!free\_list\_.empty()) {
54. *// get the frame\_id of the new page*
55. frame\_id0 = free\_list\_.back();
56. free\_list\_.pop\_back();
57. }
58. *// from replacer*
59. else {
60. if (replacer\_->Victim(&frame\_id0)) {
61. if (pages\_[frame\_id0].is\_dirty\_) {
62. *// if the replacement page is dirty, write it back to the disk*
63. disk\_manager\_->WritePage(pages\_[frame\_id0].page\_id\_, pages\_[frame\_id0].GetData());
64. pages\_[frame\_id0].is\_dirty\_ = false;
65. }
66. *// remove the replacement page from the page table*
67. page\_table\_.erase(pages\_[frame\_id0].page\_id\_);
68. } else {
69. return nullptr;
70. }
71. }
72. *// zero out memory*
73. if(frame\_id0==-1){
74. return nullptr;
75. }
76. page\_id = AllocatePage();*//allocate a new page*
77. pages\_[frame\_id0].page\_id\_ = page\_id;*//set the page id*
78. pages\_[frame\_id0].pin\_count\_ = 1;*//set the pin count*
79. pages\_[frame\_id0].is\_dirty\_ = false;*//set the dirty bit*
80. page\_table\_[page\_id] = frame\_id0;*//add the page to the page table*
81. return &pages\_[frame\_id0];
82. }
83. */\*\**
84. \* TODO: Student Implement
85. \*/
86. bool BufferPoolManager::DeletePage(page\_id\_t page\_id) {
87. *// 0.   Make sure you call DeallocatePage!*
88. *// 1.   Search the page table for the requested page (P).*
89. *// 1.   If P does not exist, return true.*
90. *// 2.   If P exists, but has a non-zero pin-count, return false. Someone is using the page.*
91. *// 3.   Otherwise, P can be deleted. Remove P from the page table, reset its metadata and return it to the free list.*
92. *// if page is not in the page table*
93. if (page\_table\_.find(page\_id) == page\_table\_.end()) {
94. return true;
95. }
96. *// if page exists*
97. else {
98. DeallocatePage(page\_id);
99. frame\_id\_t frame\_id = page\_table\_[page\_id];
100. *// if page has a non-zero pin-count*
101. if (pages\_[frame\_id].pin\_count\_ != 0) {
102. return false;
103. }
104. *// if page can be deleted*
105. else {
106. if (pages\_[frame\_id].is\_dirty\_) {
107. pages\_[frame\_id].is\_dirty\_ = false;
108. disk\_manager\_->WritePage(page\_id, pages\_[frame\_id].GetData());
109. }
110. page\_table\_.erase(page\_id);
111. replacer\_->Pin(frame\_id);
112. free\_list\_.push\_back(frame\_id);
113. return true;
114. }
115. }
116. }
117. */\*\**
118. \* TODO: Student Implement
119. \*/
120. bool BufferPoolManager::UnpinPage(page\_id\_t page\_id, bool is\_dirty) {
121. if (page\_table\_.count(page\_id) == 0) {
122. *// page is not in the page table*
123. return false;
124. }
125. frame\_id\_t frame\_id = page\_table\_[page\_id];
126. if (pages\_[frame\_id].pin\_count\_ == 0) {
127. *// page is not pinned*
128. return false;
129. }
130. *// unpin the page*
131. if (is\_dirty) {
132. pages\_[frame\_id].is\_dirty\_ = true;
133. }
134. pages\_[frame\_id].pin\_count\_--;
135. replacer\_->Unpin(frame\_id);
136. return true;
137. }
138. */\*\**
139. \* TODO: Student Implement
140. \*/
141. bool BufferPoolManager::FlushPage(page\_id\_t page\_id) {
142. *// write the page to the disk*
143. if (page\_table\_.count(page\_id) == 0) {
144. *// page is not in the page table*
145. return false;
146. }
147. else {
148. frame\_id\_t frame\_id = page\_table\_[page\_id];
149. pages\_[frame\_id].is\_dirty\_ = false;
150. disk\_manager\_->WritePage(page\_id, pages\_[frame\_id].GetData());
151. return true;
152. }
153. }

缓冲池管理器(Buffer Pool Manager)的实现代码，用于管理数据库系统中的页面(Page)。以下是对每个函数的详细分析：

1、FetchPage(page\_id\_t page\_id)

功能：根据给定的页面ID获取页面，并进行必要的操作。

实现步骤：

在页面表(page\_table\_)中查找请求的页面(P)。

如果P存在，则增加其引用计数并立即返回。

如果P不存在，则从空闲列表(free\_list\_)或替换器(replacer\_)中选择替换页面(R)。

如果选择的R是脏页面，则将其写回磁盘。

删除页面表中的R，并将P插入页面表。

更新P的元数据，从磁盘读取页面内容，并返回指向P的指针。

2、NewPage(page\_id\_t &page\_id)

功能：创建一个新页面，并将其添加到缓冲池中。

实现步骤：

如果空闲列表不为空，则从空闲列表中获取新页面的帧ID。

否则，从替换器中选择一个页面作为替换页面，并进行必要的处理。

分配新页面的页面ID，设置页面的元数据，将页面添加到页面表中，并返回指向新页面的指针。

3、DeletePage(page\_id\_t page\_id)

功能：删除给定页面ID对应的页面。

实现步骤：

确保调用了DeallocatePage以释放相应的资源。

在页面表中查找请求的页面(P)。

如果P不存在，则返回true表示删除成功。

如果P存在但引用计数不为零，则返回false表示有其他地方在使用该页面。

否则，从页面表中删除P，重置其元数据，并将其返回到空闲列表中。

4、UnpinPage(page\_id\_t page\_id, bool is\_dirty)

功能：解除页面的固定状态。

实现步骤：

检查页面是否在页面表中。

检查页面是否处于固定状态。

如果需要，将页面标记为脏页。

减少页面的引用计数，更新替换器状态，并返回true表示成功解除固定状态。

5、FlushPage(page\_id\_t page\_id)

功能：将页面内容写回磁盘。

实现步骤：

检查页面是否在页面表中。

将页面标记为非脏页。

将页面内容写回磁盘。

这些函数共同实现了对数据库中页面的管理，包括页面的获取、创建、删除、解除固定状态以及将页面内容写回磁盘等功能。

（2）buffer\_pool\_manager

1. */\*\**
2. \* TODO: Student Implement
3. \*/
4. */\**
5. \* @从LRU替换器中选择一个最少被使用的牺牲者，将其帧ID存储在指定的frame\_id指针中
6. \* @param frame\_id 用于存储选定牺牲者的帧ID的指针
7. \* @return true 如果成功选择了牺牲者并将其帧ID存储在frame\_id中
8. \* @return false 如果LRU替换器中没有可用的牺牲者
9. \*/
10. bool LRUReplacer::Victim(frame\_id\_t \*frame\_id) {
11. *// 如果替换器中没有牺牲者，则返回false*
12. if(victims\_.empty())
13. return false;
14. *// 从替换器的末尾获取牺牲者的帧ID*
15. \*frame\_id = victims\_.back();
16. *// 将该帧ID从帧ID到迭代器的映射中删除*
17. frame\_id\_to\_iterator\_[\*frame\_id] = victims\_.end();
18. *// 从替换器中删除选定的牺牲者*
19. victims\_.pop\_back();
20. *// 返回true表示成功选择了牺牲者并将其帧ID存储在frame\_id中*
21. return true;
22. }
23. */\*\**
24. \* TODO: Student Implement
25. \*/
26. void LRUReplacer::Pin(frame\_id\_t frame\_id) {
27. *// 检查缓存中是否存在指定的帧ID*
28. if (frame\_id\_to\_iterator\_[frame\_id]!= victims\_.end()){
29. *// 从替换器中移除该帧ID*
30. victims\_.erase(frame\_id\_to\_iterator\_[frame\_id]);
31. *// 将该帧ID标记为固定，即不可替换*
32. frame\_id\_to\_iterator\_[frame\_id]=victims\_.end();
33. }
34. }
35. */\*\**
36. \* TODO: Student Implement
37. \*/
38. void LRUReplacer::Unpin(frame\_id\_t frame\_id) {
39. *// 如果替换器中已经存在指定的帧ID，则返回*
40. if(victims\_.size()>=num\_pages\_ || frame\_id\_to\_iterator\_[frame\_id]!=victims\_.end()){
41. return;
42. }
43. *// 将该帧ID添加到替换器的开头*
44. victims\_.push\_front(frame\_id);
45. *// 更新帧ID到迭代器的映射*
46. frame\_id\_to\_iterator\_[frame\_id]=victims\_.begin();
47. }
48. */\*\**
49. \* TODO: Student Implement
50. \*/
51. size\_t LRUReplacer::Size() {
52. *// 返回替换器中的牺牲者数量*
53. return victims\_.size();
54. }

1、Victim(frame\_id\_t \*frame\_id)

功能： 从LRU替换器中选择一个最近最少使用的牺牲者。

详细分析：

首先，检查替换器中是否有牺牲者，即是否有帧可供替换。

如果没有可供替换的帧，返回false。

否则，从victims\_（使用了双端队列）的末尾取出一个帧ID，这是因为末尾的帧ID是最近最少使用的。

将这个帧ID从frame\_id\_to\_iterator\_映射中删除，因为这个帧ID将被替换出去。

最后，从victims\_中移除该帧ID并将其存储在frame\_id指向的位置，然后返回true表示成功选择了牺牲者。

2、Pin(frame\_id\_t frame\_id)

功能： 标记一个帧ID为固定，即不可替换。

详细分析：

首先，检查frame\_id\_to\_iterator\_映射中是否存在指定的帧ID。

如果存在，说明这个帧ID在LRU替换器中，因此可以被替换。接下来的步骤是：

从victims\_中移除这个帧ID，通过erase操作，erase接受的是一个迭代器。

将frame\_id\_to\_iterator\_中这个帧ID的映射置为victims\_.end()，表示不再在LRU替换器中。

3、Unpin(frame\_id\_t frame\_id)

功能： 将一个帧ID标记为可替换，即从固定状态恢复为可替换状态。

详细分析：

首先，检查是否需要替换，即LRU替换器是否已满 (victims\_.size() >= num\_pages\_) 或者指定的帧ID已经在LRU替换器中 (frame\_id\_to\_iterator\_[frame\_id] != victims\_.end())。

如果满足上述条件，则直接返回，无需进行替换操作。

否则，将帧ID添加到victims\_的开头 (push\_front)，因为最近访问的帧应该放在队列的最前面，以便于后续被选择为牺牲者。

更新frame\_id\_to\_iterator\_中这个帧ID的映射为victims\_.begin()，表示它现在在LRU替换器中并处于可替换状态。

4、Size()

功能： 返回LRU替换器中当前可替换的帧的数量。

详细分析：

简单地返回victims\_的大小，即当前LRU替换器中有多少个帧是可替换的。

三、测试结果

1、buffer\_pool\_manager



可以看到，本测试样例可以顺利通过。

2、lru



可以看到，本测试样例可以顺利通过。

# #2 RECORD MANAGER

负责人：陈勇帆

一、实验目的

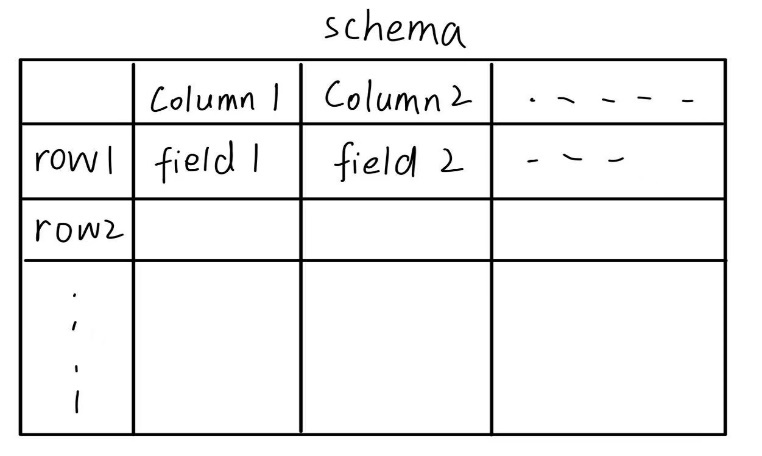
模块二包括两部分。其一是为MiniSQL数据库管理系统中的Record Manager设计和实现记录的序列化和反序列化操作。具体来说，需要完成Row、Schema和Column对象的序列化和反序列化方法，以及计算序列化后的字节流大小的方法，便于将内存中的记录、模式和列对象转换为字节流，以便将它们写入磁盘中，同时也需要能够从磁盘中的字节流中恢复这些对象。其二是通过堆表实现记录管理。

二、内容与功能

在实现记录管理之前，需要实现序列化和反序列化方法。序列化是将内存中的对象转换为字节流的过程，而反序列化则是从字节流中恢复对象的过程。这种操作是为了持久化存储数据，使得数据可以在内外存之间进行转换。

**（一）记录序列化与反序列化操作**

首先通过下图来理解记录的形式与格式：



结合头文件来理解各个块的内容：

（1）Field

①Field类定义中friend声明允许Type类及其派生类访问Field类的私有成员。

②成员变量：

value\_：存储字段的实际值，可以是整数、浮点数或字符指针。

type\_id\_：存储字段的类型。

len\_：存储字段的长度。

is\_null\_：指示字段是否为空。

manage\_data\_：指示字符数据是否由 Field 对象管理内存。

③成员函数：较多，包括了函数构造、序列化、访问和比较等，以下其他模块也是，均不详细说明，主要介绍成员变量。

（2）Row成员变量包括行ID：rid\_和字段列表：vector<Field \*> fields\_

（3）Column

①成员变量

name\_: 列名，类型为 std::string。

type\_: 列的数据类型，类型为 TypeId。

len\_: 列的长度，默认值为 0，对于 CHAR 类型，表示字符串的最大字节长度；否则表示固定大小。

table\_ind\_: 列在表中的索引位置，类型为 uint32\_t。

nullable\_: 指示列是否可以为 NULL，类型为 bool。

unique\_: 指示列是否唯一，类型为 bool。

COLUMN\_MAGIC\_NUM: 静态常量，表示列的魔数，用于序列化和反序列化时验证数据的正确性。

②构造函数

Column(std::string column\_name, TypeId type, uint32\_t index, bool nullable, bool unique): 用于创建一个新列对象，指定列名、类型、索引、是否可为空和是否唯一。

Column(std::string column\_name, TypeId type, uint32\_t length, uint32\_t index, bool nullable, bool unique): 用于创建一个新的 CHAR 类型列对象，指定列名、类型、长度、索引、是否可为空和是否唯一。

Column(const Column \*other): 拷贝构造函数，从另一个列对象拷贝数据。

注意Column根据类型不同有不同的构造函数。

（4）Schema成员变量：

SCHEMA\_MAGIC\_NUM：用于模式对象的魔数，确保数据的正确性。

columns\_：存储列指针的向量。

is\_manage\_：指示模式对象是否管理其列指针的生命周期。

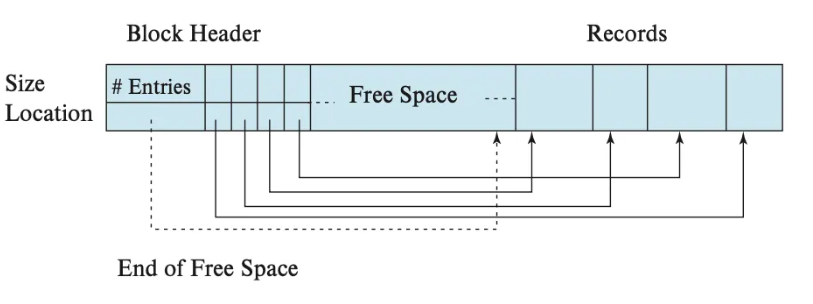
**（二）堆表实现记录管理**

堆表是一种将记录以无序堆形式进行组织的数据结构，不同的数据页（TablePage）之间通过双向链表连接。堆表中的记录通过RowId进行定位。RowId包含两个部分：page\_id和slot\_num，其中page\_id定位记录所在的数据页，slot\_num定位记录在数据页中的具体位置。堆表的数据结构如下：

表头（Table Page Header）：记录页的元数据，包括前后页的页号、空闲空间指针以及记录的偏移和长度。

空闲空间（Free Space）：用于存放尚未使用的空间。

已插入的数据（Inserted Tuples）：实际存储的记录数据。插入的记录在页中自右向左扩展。



堆表迭代器则将堆表作为一种新的数据类型并定义一些基础的运算符号。

再结合头文件了解堆表结构和功能：

（1）table\_heap

①成员变量：

buffer\_pool\_manager\_：BufferPoolManager 类型指针，用于管理数据页的缓冲池。

first\_page\_id\_：page\_id\_t 类型，表示表的第一个数据页的页号。

schema\_：Schema 类型指针，表示表的数据结构和字段信息。

log\_manager\_：LogManager 类型指针，用于记录事务操作的日志。

lock\_manager\_：LockManager 类型指针，用于管理数据的并发访问控制。

②成员函数：

TableHeap(...) 构造函数：两个重载形式，分别用于创建新的表堆和初始化已有的表堆。初始化 first\_page\_id\_ 和相关的管理器对象。

~TableHeap() 析构函数：清理资源，确保数据页和缓冲池的正确释放。

bool InsertTuple(Row &row, Txn \*txn)：插入一条新的数据行到表中。返回 true 表示插入成功，false 表示插入失败（通常是由于数据行过大）。

bool MarkDelete(const RowId &rid, Txn \*txn)：标记指定的数据行为已删除状态。返回 true 表示标记成功，false 表示标记失败（通常是由于未找到要标记的数据行）。

bool UpdateTuple(Row &row, const RowId &rid, Txn \*txn)：更新指定数据行的内容。如果更新后的数据行超出原有容量，执行删除和插入操作。返回 true 表示更新成功，false 表示更新失败。

void ApplyDelete(const RowId &rid, Txn \*txn)：应用删除操作，实际从物理上删除标记为删除的数据行。

void RollbackDelete(const RowId &rid, Txn \*txn)：回滚删除操作，取消标记的删除状态，恢复数据行。

bool GetTuple(Row \*row, Txn \*txn)：获取指定数据行的内容，并存储在 row 中。返回 true 表示获取成功，false 表示获取失败。

RowId GetNextRowID(Row \*row, Txn \*txn)：获取指定数据行的下一条数据行的 RowId。返回下一条数据行的 RowId。

void FreeTableHeap()：销毁整个表堆，释放所有数据页和相关资源。

void DeleteTable(page\_id\_t page\_id = INVALID\_PAGE\_ID)：删除表，释放所有数据页和磁盘存储空间。

TableIterator Begin(Txn \*txn)：返回表的起始迭代器，用于遍历表中的数据行。

TableIterator End()：返回表的结束迭代器。

（2）table\_iterator

①成员变量：

迭代器的成员函数需要自己设计，这里我主要结合构造函数的传参来设计，其中在本实验txn其实没用用到，是置空的。

cur\_table\_heap：TableHeap 类型指针，指向当前迭代器所在的表堆。

cur\_rowid：RowId 类型，表示当前迭代器所在的数据行的 RowId。

txn：Txn 类型指针，表示当前操作的事务对象。

②成员函数：

TableIterator(TableHeap \*table\_heap, RowId rid, Txn\* txn) 构造函数：接受 TableHeap 指针、行 ID 和事务对象作为参数，初始化迭代器。

TableIterator(const TableIterator &other) 拷贝构造函数：拷贝另一个迭代器的状态。

~TableIterator() 析构函数：清理资源，确保没有资源泄漏。

bool operator==(const TableIterator &itr) const：比较两个迭代器是否指向相同位置。

bool operator!=(const TableIterator &itr) const：比较两个迭代器是否不指向相同位置。

const Row &operator\*()：解引用操作符，返回当前迭代器指向位置的数据行。

Row \*operator->()：返回指向当前迭代器指向位置的数据行的指针。

TableIterator &operator=(const TableIterator &itr) noexcept：赋值运算符重载，将一个迭代器的状态赋值给另一个迭代器。

TableIterator &operator++()：前置递增运算符重载，使迭代器向前移动到下一个数据行。

TableIterator operator++(int)：后置递增运算符重载，使迭代器向前移动到下一个数据行，并返回移动前的迭代器状态。

三、代码实现与分析

**（一）记录序列化与反序列化操作**

（1）Row序列化和反序列化：

①SerializeTo方法： 将Row对象序列化为字节流。首先将RowId写入字节流，然后写入字段数量，并在之后的字节流中使用位图标记每个字段是否为null，最后写入每个非null字段的内容。

    uint32\_t Row::SerializeTo(char \*buf, Schema \*schema) const {

    // 断言确保schema不为空

    ASSERT(schema != nullptr, "Invalid schema before serialize.");

    // 断言确保字段数量与schema中的列数匹配

    ASSERT(schema->GetColumnCount() == fields\_.size(), "Fields size do not match schema's column size.");

    uint32\_t field\_num = schema->GetColumnCount();

    uint32\_t ofs = 0;

    // 将RowId写入buf中

    memcpy(buf + ofs, &rid\_, sizeof(rid\_));

    ofs += sizeof(RowId);

    // 将字段数量写入buf中

    memcpy(buf + ofs, &field\_num, sizeof(uint32\_t));

    ofs += sizeof(uint32\_t);

    // 计算位图所需的长度（每8个字段占用一个字节）

    uint32\_t bitmap\_len = (field\_num - 1) / 8 + 1;

    // 位图用于标记字段是否为null，初始化为0

    char\* bitmap = buf + ofs;

    memset(bitmap, 0, bitmap\_len);

    ofs += bitmap\_len;

    // 遍历每个字段，进行序列化操作

    for (uint32\_t i = 0; i < field\_num; i++) {

        Field\* field = fields\_[i];

        if (field->IsNull()) {

        // 如果字段为null，设置对应位图的位为1

        bitmap[i / 8] = bitmap[i / 8] | (1 << (i % 8));

        } else {

        // 否则，将字段内容序列化到buf中，并更新ofs偏移量

        ofs += field->SerializeTo(buf + ofs);

        }

    }

    // 返回序列化后的总长度

    return ofs;

    }

②DeserializeFrom方法： 从字节流中反序列化Row对象。首先读取RowId，然后读取字段数量，并根据位图确定哪些字段为null，然后根据位图中的信息恢复每个字段的内容。

    uint32\_t Row::DeserializeFrom(char \*buf, Schema \*schema) {

    // 断言确保schema不为空

    ASSERT(schema != nullptr, "Invalid schema before serialize.");

    // 断言确保字段为空，防止重复反序列化

    ASSERT(fields\_.empty(), "Non empty field in row.");

    uint32\_t ofs = 0;

    // 从buf中读取RowId

    memcpy(&rid\_, buf + ofs, sizeof(RowId));

    ofs += sizeof(RowId);

    // 从buf中读取字段数量

    uint32\_t field\_num;

    memcpy(&field\_num, buf + ofs, sizeof(uint32\_t));

    ofs += sizeof(uint32\_t);

    // 计算位图所需的长度

    size\_t bitmap\_len = (field\_num - 1) / 8 + 1;

    char\* bitmap = buf + ofs;

    ofs += bitmap\_len;

    // 清空当前字段数组

    fields\_.clear();

    fields\_.resize(field\_num);

    // 根据位图信息反序列化每个字段

    for (uint32\_t i = 0; i < field\_num; i++) {

        if (bitmap[i / 8] & (1 << (i % 8))) {

        // 如果位图对应位为1，说明字段为null

        ofs += Field::DeserializeFrom(buf + ofs, schema->GetColumns()[i]->GetType(), &fields\_[i], true);

        } else {

        // 否则，根据字段类型反序列化字段内容，并更新ofs偏移量

        ofs += Field::DeserializeFrom(buf + ofs, schema->GetColumns()[i]->GetType(), &fields\_[i], false);

        }

    }

    // 返回总的反序列化长度

    return ofs;

}

③GetSerializedSize方法： 计算序列化后的字节流大小。包括RowId、字段数量以及位图所占的字节，以及每个非null字段的大小。

uint32\_t Row::GetSerializedSize(Schema \*schema) const {

    // 断言确保schema不为空

    ASSERT(schema != nullptr, "Invalid schema before serialize.");

    // 断言确保字段数量与schema中的列数匹配

    ASSERT(schema->GetColumnCount() == fields\_.size(), "Fields size do not match schema's column size.");

    uint32\_t size = sizeof(RowId) + sizeof(uint32\_t) + (schema->GetColumnCount() - 1) / 8 + 1;

    // 计算每个非null字段的序列化后大小，并累加到总长度中

    for (uint32\_t i = 0; i < schema->GetColumnCount(); i++) {

        if (!fields\_[i]->IsNull()) {

        size += fields\_[i]->GetSerializedSize();

        }

    }

    // 返回序列化后的总长度

    return size;

    }

（2）Column序列化和反序列化：

①SerializeTo方法： 将Column对象序列化为字节流。依次写入魔数、列名长度和列名、列类型、列长度（如果是CHAR类型）、列索引、是否可为空以及是否唯一。

    uint32\_t Column::SerializeTo(char \*buf) const {

    uint32\_t ofs = 0;

    // 写入列的魔数

    memcpy(buf + ofs, &COLUMN\_MAGIC\_NUM, sizeof(COLUMN\_MAGIC\_NUM));

    ofs += sizeof(COLUMN\_MAGIC\_NUM);

    // 写入列名的长度和列名内容

    size\_t name\_len = name\_.length();

    memcpy(buf + ofs, &name\_len, sizeof(name\_len));

    ofs += sizeof(name\_len);

    memcpy(buf + ofs, name\_.c\_str(), name\_len);

    ofs += name\_len;

    // 写入列的数据类型

    memcpy(buf + ofs, &type\_, sizeof(type\_));

    ofs += sizeof(type\_);

    // 如果是CHAR类型，写入列的长度

    if (type\_ == kTypeChar) {

        memcpy(buf + ofs, &len\_, sizeof(len\_));

        ofs += sizeof(len\_);

    }

    // 写入列的索引、可空性和唯一性

    memcpy(buf + ofs, &table\_ind\_, sizeof(table\_ind\_));

    ofs += sizeof(table\_ind\_);

    memcpy(buf + ofs, &nullable\_, sizeof(nullable\_));

    ofs += sizeof(nullable\_);

    memcpy(buf + ofs, &unique\_, sizeof(unique\_));

    ofs += sizeof(unique\_);

    // 返回序列化后的总长度

    return ofs;

}

②DeserializeFrom方法： 从字节流中反序列化Column对象。首先读取魔数，然后读取列名长度和列名，再读取列类型和其他属性。

uint32\_t Column::DeserializeFrom(char \*buf, Column \*&column) {

    uint32\_t ofs = 0;

    // 读取列的魔数，验证是否为列对象的正确序列化数据

    uint32\_t magic = MACH\_READ\_INT32(buf + ofs);

    ofs += sizeof(magic);

    ASSERT(magic == COLUMN\_MAGIC\_NUM, "Column deserialize wrong, not a column buffer");

    // 读取列名的长度和内容

    size\_t name\_len = MACH\_READ\_FROM(size\_t, buf + ofs);

    ofs += sizeof(size\_t);

    char\* name\_buf = new char[name\_len];

    memcpy(name\_buf, buf + ofs, name\_len);

    std::string name(name\_buf, name\_len / sizeof(char));

    ofs += name\_len;

    // 读取列的数据类型

    TypeId type = MACH\_READ\_FROM(TypeId, buf + ofs);

    ofs += sizeof(TypeId);

    // 如果是CHAR类型，读取列的长度

    uint32\_t len;

    if (type == kTypeChar) {

        len = MACH\_READ\_INT32(buf + ofs);

        ofs += sizeof(uint32\_t);

    }

    // 读取列的索引、可空性和唯一性

    uint32\_t table\_ind = MACH\_READ\_INT32(buf + ofs);

    ofs += sizeof(uint32\_t);

    bool nullable = MACH\_READ\_FROM(bool, buf + ofs);

    ofs += sizeof(bool);

    bool unique = MACH\_READ\_FROM(bool, buf + ofs);

    ofs += sizeof(bool);

    // 根据读取的数据创建新的Column对象，并赋值给column指针

    if (type == kTypeChar) {

        column = new Column(name, type, len, table\_ind, nullable, unique);

    } else {

        column = new Column(name, type, table\_ind, nullable, unique);

    }

    // 返回反序列化后的总长度

    return ofs;

}

③GetSerializedSize方法： 计算序列化后的字节流大小。根据列的类型和属性，确定序列化后的字节流大小。

    uint32\_t Column::GetSerializedSize() const {

    // 计算序列化后的总长度，根据列的数据类型和属性确定具体大小

    if (type\_ == kTypeChar) {

        return sizeof(COLUMN\_MAGIC\_NUM) + sizeof(size\_t) + name\_.length() + sizeof(type\_) + sizeof(len\_) + sizeof(table\_ind\_) + sizeof(nullable\_) + sizeof(unique\_);

    } else {

        return sizeof(COLUMN\_MAGIC\_NUM) + sizeof(size\_t) + name\_.length() + sizeof(type\_) + sizeof(table\_ind\_) + sizeof(nullable\_) + sizeof(unique\_);

    }

    }

（3）Schema序列化和反序列化：

①SerializeTo方法： 将Schema对象序列化为字节流。首先写入魔数，然后写入列数量，并依次将每个列对象序列化为字节流，最后写入是否管理的标志位。

    uint32\_t Schema::SerializeTo(char \*buf) const {

    // 序列化 Schema 对象，包括魔数、列数量、每个列对象和 is\_manage 标志

    uint32\_t ofs = 0;

    // 将魔数写入缓冲区

    memcpy(buf + ofs, &SCHEMA\_MAGIC\_NUM, sizeof(SCHEMA\_MAGIC\_NUM));

    ofs += sizeof(SCHEMA\_MAGIC\_NUM);

    // 写入列的数量

    auto column\_count = static\_cast<uint32\_t>(columns\_.size());

    memcpy(buf + ofs, &column\_count, sizeof(column\_count));

    ofs += sizeof(column\_count);

    // 序列化每个列对象

    for (auto col : columns\_) {

        ofs += col->SerializeTo(buf + ofs);

    }

    // 写入 is\_manage 标志

    memcpy(buf + ofs, &is\_manage\_, sizeof(is\_manage\_));

    ofs += sizeof(is\_manage\_);

    return ofs;

    }

②DeserializeFrom方法： 从字节流中反序列化Schema对象。读取魔数和列数量，然后根据列数量依次反序列化每个列对象。

    uint32\_t Schema::DeserializeFrom(char \*buf, Schema \*&schema) {

    uint32\_t ofs = 0;

    // 读取并验证魔数

    uint32\_t magic = MACH\_READ\_INT32(buf + ofs);

    ofs += sizeof(magic);

    ASSERT(magic == SCHEMA\_MAGIC\_NUM, "schema deserialize wrong, not a schema buffer");

    // 读取列的数量

    uint32\_t column\_count = MACH\_READ\_INT32(buf + ofs);

    ofs += sizeof(uint32\_t);

    // 反序列化每个列对象并加入列向量

    std::vector<Column> columns;

    for (uint32\_t i = 0; i < column\_count; i++) {

        Column col;

        ofs += Column::DeserializeFrom(buf + ofs, col);

        columns.push\_back(col);

    }

    // 读取 is\_manage 标志

    bool is\_manage = MACH\_READ\_FROM(bool, buf + ofs);

    schema = new Schema(columns, is\_manage);

    return ofs;

}

③GetSerializedSize方法： 计算序列化后的字节流大小。根据列的数量和每个列的序列化后大小，确定总的字节流大小。

    uint32\_t Schema::GetSerializedSize() const {

    // 计算 Schema 对象序列化后的总字节大小

    uint32\_t size = sizeof(SCHEMA\_MAGIC\_NUM) + sizeof(uint32\_t) + sizeof(is\_manage\_);

    // 累加每个列对象序列化后的大小

    for (auto col : columns\_) {

        size += col->GetSerializedSize();

    }

    return size;

}

**（二）堆表实现记录管理**

（1）table\_heap.h

在头文件中需要自主实现堆表第一页的创建和初始化

    private:

    /\*\*

     \* create table heap and initialize first page

     \*/

    explicit TableHeap(BufferPoolManager \*buffer\_pool\_manager, Schema \*schema, Txn \*txn, LogManager \*log\_manager,

                        LockManager \*lock\_manager)

        : buffer\_pool\_manager\_(buffer\_pool\_manager),

            schema\_(schema),

            log\_manager\_(log\_manager),

            lock\_manager\_(lock\_manager) {

        page\_id\_t new\_page\_id;

        buffer\_pool\_manager\_->NewPage(new\_page\_id);  //从缓冲池得到空页作为第一页

        first\_page\_id\_ = new\_page\_id;

        auto new\_page = reinterpret\_cast<TablePage \*>(buffer\_pool\_manager\_->FetchPage(new\_page\_id));

        new\_page->WLatch();

        new\_page->Init(new\_page\_id,INVALID\_PAGE\_ID,log\_manager,txn); //初始化第一页

        new\_page->SetNextPageId(INVALID\_PAGE\_ID);

        new\_page->WUnlatch();

        buffer\_pool\_manager\_->UnpinPage(new\_page->GetPageId(), true);

    };

（2）table\_heap.cpp

①InsertTuple函数：将记录插入到堆表中。首先计算记录的大小，检查是否超过单页大小。然后尝试在当前页插入记录，如果当前页空间不足，沿链表寻找下一个页，直到找到能够容纳记录的页。如果所有页均无法容纳记录，则创建新的数据页，并插入记录。

    bool TableHeap::InsertTuple(Row &row, Txn \*txn) {

    // 计算记录的序列化大小

    uint32\_t tuple\_size = row.GetSerializedSize(schema\_);

    // 检查记录大小是否超过页大小

    if (tuple\_size > PAGE\_SIZE - 32) return false;

    // 获取堆表的第一页

    auto page = reinterpret\_cast<TablePage \*>(buffer\_pool\_manager\_->FetchPage(first\_page\_id\_));

    if (page == nullptr) return false;

    // 对页面加写锁

    page->WLatch();

    // 尝试在当前页插入记录

    bool insert\_result = page->InsertTuple(row, schema\_, txn, lock\_manager\_, log\_manager\_);

    // 释放写锁

    page->WUnlatch();

    // 取消页面的pin状态

    buffer\_pool\_manager\_->UnpinPage(page->GetPageId(), true);

    // 如果插入失败，则沿着链表寻找能够容纳记录的页

    while (!insert\_result) {

        page\_id\_t next\_id = page->GetNextPageId();

        if (next\_id != INVALID\_PAGE\_ID) {

        page = reinterpret\_cast<TablePage \*>(buffer\_pool\_manager\_->FetchPage(next\_id));

        page->WLatch();

        insert\_result = page->InsertTuple(row, schema\_, txn, lock\_manager\_, log\_manager\_);

        page->WUnlatch();

        buffer\_pool\_manager\_->UnpinPage(page->GetPageId(), true);

        } else {

        // 创建新的数据页

        page\_id\_t new\_id;

        buffer\_pool\_manager\_->NewPage(new\_id);

        auto new\_page = reinterpret\_cast<TablePage \*>(buffer\_pool\_manager\_->FetchPage(new\_id));

        new\_page->WLatch();

        new\_page->Init(new\_id, page->GetPageId(), log\_manager\_, txn);

        new\_page->SetNextPageId(INVALID\_PAGE\_ID);

        page->WLatch();

        page->SetNextPageId(new\_id);

        page->WUnlatch();

        buffer\_pool\_manager\_->UnpinPage(page->GetPageId(), true);

        insert\_result = new\_page->InsertTuple(row, schema\_, txn, lock\_manager\_, log\_manager\_);

        new\_page->WUnlatch();

        buffer\_pool\_manager\_->UnpinPage(new\_page->GetPageId(), true);

        }

    }

    return insert\_result;

}

②UpdateTuple函数：更新指定RowId的记录。首先定位记录所在页，尝试在该页更新记录。如果更新后的记录大小超过页的空闲空间，则更新失败。

    bool TableHeap::UpdateTuple(Row &row, const RowId &rid, Txn \*txn) {

    // 获取记录所在的旧页

    auto old\_page = reinterpret\_cast<TablePage \*>(buffer\_pool\_manager\_->FetchPage(rid.GetPageId()));

    if (old\_page == nullptr) return false;

    // 对旧页加写锁

    old\_page->WLatch();

    // 创建旧记录的Row对象

    Row old\_row = Row(rid);

    // 尝试在旧页更新记录

    bool update\_result = old\_page->UpdateTuple(row, &old\_row, schema\_, txn, lock\_manager\_, log\_manager\_);

    // 释放写锁

    old\_page->WUnlatch();

    // 取消旧页的pin状态

    buffer\_pool\_manager\_->UnpinPage(old\_page->GetPageId(), true);

    return update\_result;

}

③ApplyDelete函数：物理删除指定RowId的记录。首先定位记录所在页，然后将记录从页中移除。

    void TableHeap::ApplyDelete(const RowId &rid, Txn \*txn) {

    // 获取记录所在的页

    auto page = reinterpret\_cast<TablePage \*>(buffer\_pool\_manager\_->FetchPage(rid.GetPageId()));

    ASSERT(page != nullptr, "page to delete not found");

    // 对页加写锁

    page->WLatch();

    // 从页中删除记录

    page->ApplyDelete(rid, txn, log\_manager\_);

    // 释放写锁

    page->WUnlatch();

    // 取消页的pin状态

    buffer\_pool\_manager\_->UnpinPage(page->GetPageId(), true);

    }

④GetTuple函数：获取指定RowId的记录。首先通过RowId中的page\_id定位数据页，然后通过slot\_num定位具体记录。

    bool TableHeap::GetTuple(Row \*row, Txn \*txn) {

    // 获取记录的RowId

    const RowId &rid = row->GetRowId();

    // 获取记录所在的页

    auto page = reinterpret\_cast<TablePage \*>(buffer\_pool\_manager\_->FetchPage(rid.GetPageId()));

    if (page == nullptr) return false;

    // 对页加读锁

    page->RLatch();

    // 从页中获取记录

    page->GetTuple(row, schema\_, txn, lock\_manager\_);

    // 释放读锁

    page->RUnlatch();

    // 取消页的pin状态

    buffer\_pool\_manager\_->UnpinPage(page->GetPageId(), true);

    return true;

}

⑤Begin和End函数：Begin函数获取堆表的首迭代器，定位到第一条记录的位置。End函数获取堆表的尾迭代器，表示堆表的结束位置。

    TableIterator TableHeap::Begin(Txn \*txn) {

    // 获取堆表的第一页

    auto begin\_page = reinterpret\_cast<TablePage \*>(buffer\_pool\_manager\_->FetchPage(first\_page\_id\_));

    RowId rid;

    // 获取第一页中的第一条记录的RowId

    begin\_page->GetFirstTupleRid(&rid);

    return TableIterator(this, rid, txn);

    }

    TableIterator TableHeap::End() {

    // 返回指向堆表结束位置的迭代器

    return TableIterator(this, INVALID\_ROWID, nullptr);

}

⑥GetNextRowID函数：获取给定记录的下一条记录的RowId。如果当前页内存在下一条记录，则直接返回。如果当前页已经是最后一条记录，则沿着页链表继续查找下一页，直到找到下一条记录或到达链表末尾。这个函数主要是为了方便迭代器更新。

    RowId TableHeap::GetNextRowID(Row \*row, Txn \*txn) {

        // 获取当前记录的RowId

        RowId r\_id = row->GetRowId();

        // 获取当前记录所在的页

        auto page = reinterpret\_cast<TablePage \*>(buffer\_pool\_manager\_->FetchPage(r\_id.GetPageId()));

        // 对页加读锁

        page->RLatch();

        // 用于存储下一条记录的RowId

        RowId n\_id;

        // 在当前页尝试获取下一条记录的RowId

        bool get\_result = page->GetNextTupleRid(r\_id, &n\_id);

        if (get\_result) { // 如果在当前页有下一条记录

            // 释放读锁

            page->RUnlatch();

            // 取消页面的pin状态

            buffer\_pool\_manager\_->UnpinPage(page->GetPageId(), false);

            // 返回下一条记录的RowId

            return n\_id;

        }

        // 释放当前页的读锁

        page->RUnlatch();

        // 取消当前页的pin状态

        buffer\_pool\_manager\_->UnpinPage(page->GetPageId(), false);

        // 获取下一页

        auto next\_page = reinterpret\_cast<TablePage \*>(buffer\_pool\_manager\_->FetchPage(page->GetNextPageId()));

        if (next\_page == nullptr) {

            // 如果没有下一页，返回INVALID\_ROWID表示到达了最后一条记录

            return INVALID\_ROWID;

        }

        // 循环查找，直到找到下一条记录或到达链表末尾

        while (true) {

            // 对页加读锁

            next\_page->RLatch();

            // 尝试获取下一页的第一条记录的RowId

            get\_result = next\_page->GetFirstTupleRid(&r\_id);

            if (get\_result) { // 一旦获取成功就返回

            // 释放读锁

            buffer\_pool\_manager\_->UnpinPage(next\_page->GetPageId(), false);

            next\_page->RUnlatch();

            // 返回下一页的第一条记录的RowId

            return r\_id;

            }

            // 否则继续寻找下一页

            buffer\_pool\_manager\_->UnpinPage(next\_page->GetPageId(), false);

            next\_page->RUnlatch();

            next\_page = reinterpret\_cast<TablePage \*>(buffer\_pool\_manager\_->FetchPage(next\_page->GetNextPageId()));

            if (next\_page == nullptr) {

            // 如果没有更多的页，返回INVALID\_ROWID表示到达了最后一条记录

            return INVALID\_ROWID;

            }

        }

}

（3）table\_iterator.cpp

①构造函数：初始化TableIterator对象，设置指向的TableHeap、当前的RowId以及事务。

    TableIterator::TableIterator(TableHeap \*table\_heap, RowId rid, Txn \*txn) {

        // 初始化当前表堆和行ID

        cur\_table\_heap = table\_heap;

        cur\_rowid = rid;

        this->txn = txn;

}

②拷贝构造函数：创建一个新的TableIterator对象，并将现有TableIterator对象的状态复制到新对象中。

    TableIterator::TableIterator(const TableIterator &other) {

        // 复制构造函数，从另一个TableIterator对象复制成员变量

        cur\_table\_heap = other.cur\_table\_heap;

        cur\_rowid = other.cur\_rowid;

        txn = other.txn;

}

③析构函数不影响其他功能，没有实现。

④比较函数：operator==用于判断两个TableIterator对象是否相等。主要比较它们所指向的TableHeap对象和当前的RowId是否相同。operator!= 用于判断两个TableIterator对象是否不相等。它的实现是调用operator==函数，并取其结果的反值。

    bool TableIterator::operator==(const TableIterator &itr) const {

        // 比较两个迭代器是否相等，主要比较它们指向的表堆和行ID是否相同

        return cur\_table\_heap == itr.cur\_table\_heap && cur\_rowid == itr.cur\_rowid;

        }

        bool TableIterator::operator!=(const TableIterator &itr) const {

        // 比较两个迭代器是否不相等

        return !(\*this == itr);

}

⑤指针函数：operator\*用于获取当前迭代器指向的记录，operator->用于获取当前迭代器指向的记录的指针。他们均创建一个新的Row对象，并从TableHeap中获取当前行的记录。

    const Row &TableIterator::operator\*() {

        ASSERT(cur\_table\_heap, "TableHeap is nullptr.");

        // 创建一个新的Row对象，并设置当前行ID

        Row\* row = new Row(cur\_rowid);

        // 获取当前行的记录

        cur\_table\_heap->GetTuple(row, nullptr);

        ASSERT(row, "Invalid.");

        return \*row;

        }

        Row \*TableIterator::operator->() {

        // 创建一个新的Row对象，并设置当前行ID

        Row\* row = new Row(cur\_rowid);

        // 获取当前行的记录

        cur\_table\_heap->GetTuple(row, nullptr);

        ASSERT(row, "Invalid.");

        return row;

}

⑥赋值运算：operator=用于将一个TableIterator对象赋值给另一个TableIterator对象。它会检查自赋值的情况，并将右侧对象的状态复制到左侧对象中。

    TableIterator &TableIterator::operator=(const TableIterator &itr) noexcept {

        // 确保赋值的对象不是自身

        if (this != &itr) {

            cur\_table\_heap = itr.cur\_table\_heap;

            cur\_rowid = itr.cur\_rowid;

            txn = itr.txn;

        }

        return \*this;

}

⑦加法运算符：operator++（前置++运算符）该函数用于将迭代器前置递增，移动到下一条记录。它首先获取当前记录的Row对象，然后通过TableHeap获取下一条记录的RowId，并更新当前行ID。operator++（后置++运算符）与前置递增类似，只是返回值是递增前的迭代器。

    TableIterator &TableIterator::operator++() {

        ASSERT(cur\_table\_heap, "TableHeap is nullptr.");

        // 获取当前记录的Row对象

        Row cur\_row;

        cur\_row.SetRowId(cur\_rowid);

        cur\_table\_heap->GetTuple(&cur\_row, nullptr);

        // 获取下一条记录的RowId

        cur\_rowid = cur\_table\_heap->GetNextRowID(&cur\_row, nullptr);

        return \*this;

    }

    TableIterator TableIterator::operator++(int) {

        ASSERT(cur\_table\_heap, "TableHeap is nullptr.");

        // 获取当前记录的Row对象

        Row cur\_row;

        cur\_row.SetRowId(cur\_rowid);

        cur\_table\_heap->GetTuple(&cur\_row, nullptr);

        // 获取下一条记录的RowId

        cur\_rowid = cur\_table\_heap->GetNextRowID(&cur\_row, nullptr);

        return TableIterator(cur\_table\_heap, cur\_rowid, nullptr);

    }

四、测试代码分析与结果

**（一）代码分析**

（1）tuple\_test.cpp测试代码主要用于测试 Field 类和 Row 类的序列化/反序列化、比较操作以及表页操作。

首先定义了四个字符指针数组，定义了整数、浮点数、字符和空字段的 Field 数组，用于测试不同类型的字段。然后分为以下两个测试函数：

①FieldSerializeDeserializeTest 测试函数

A、序列化阶段：定义一个 buffer 用于存储序列化数据，并将其初始化为 0；对整数、浮点数和字符字段进行序列化，并将序列化结果存储在 buffer 中。

B、反序列化阶段：使用 DeserializeFrom 方法从 buffer 中读取数据，并与原始字段进行比较；验证反序列化后的字段与原始字段是否相等、比较操作是否正确。

②RowTest 测试函数

A、初始化和插入阶段：创建 TablePage 对象和 Schema 对象；定义列结构和字段数据；初始化 TablePage 并插入一个 Row 对象。

B、检索和验证阶段

获取第一个元组的 RowId 并进行验证；根据 RowId 创建新 Row 对象并从 TablePage 中获取元组数据；验证新获取的 Row 对象中的字段是否与插入时一致。

（2）table\_heap\_test.cpp测试代码用于测试 TableHeap 类的基本功能，包括插入和读取数据。

①初始化测试环境

创建一个 DiskManager 对象用于管理磁盘文件；创建一个 BufferPoolManager 对象用于管理缓冲池；定义将要插入的行数为 10000。

②创建模式（Schema）

定义表的列：一个整型 id 列，一个字符型 name 列（最大长度 64），一个浮点型 account 列；使用这些列创建一个 Schema 对象。

③创建行数据并插入表中

创建一个哈希表 row\_values 用于存储行的值，键是行的 RowId；定义一个变量 size 用于记录插入的行数；创建一个 TableHeap 对象。

④插入数据

使用随机生成的数据创建并插入行：生成随机长度的字符数组 characters；创建一个 Fields 对象，其中包含三个字段（id, name, account）；创建一个 Row 对象并插入到 TableHeap 中；检查行的 RowId 是否已经存在于 row\_values 中，如果存在则打印并断言失败；否则将行的 RowId 和字段存储到 row\_values 中，并增加 size 计数；释放 characters 内存。

⑤验证插入结果：断言插入的行数与预期的行数相同。

⑥读取并验证数据

逐个读取存储在 row\_values 中的行：使用 RowId 创建 Row 对象并从 TableHeap 中获取数据；断言行的字段数与模式中的列数相同；比较行的每个字段与 row\_values 中存储的对应字段，确保数据一致；释放 row\_values 中的字段内存。

最后断言 size 为 0，确保所有行都已被正确处理。

这个测试验证了 TableHeap 的以下功能：

能够正确地插入大量随机数据行。

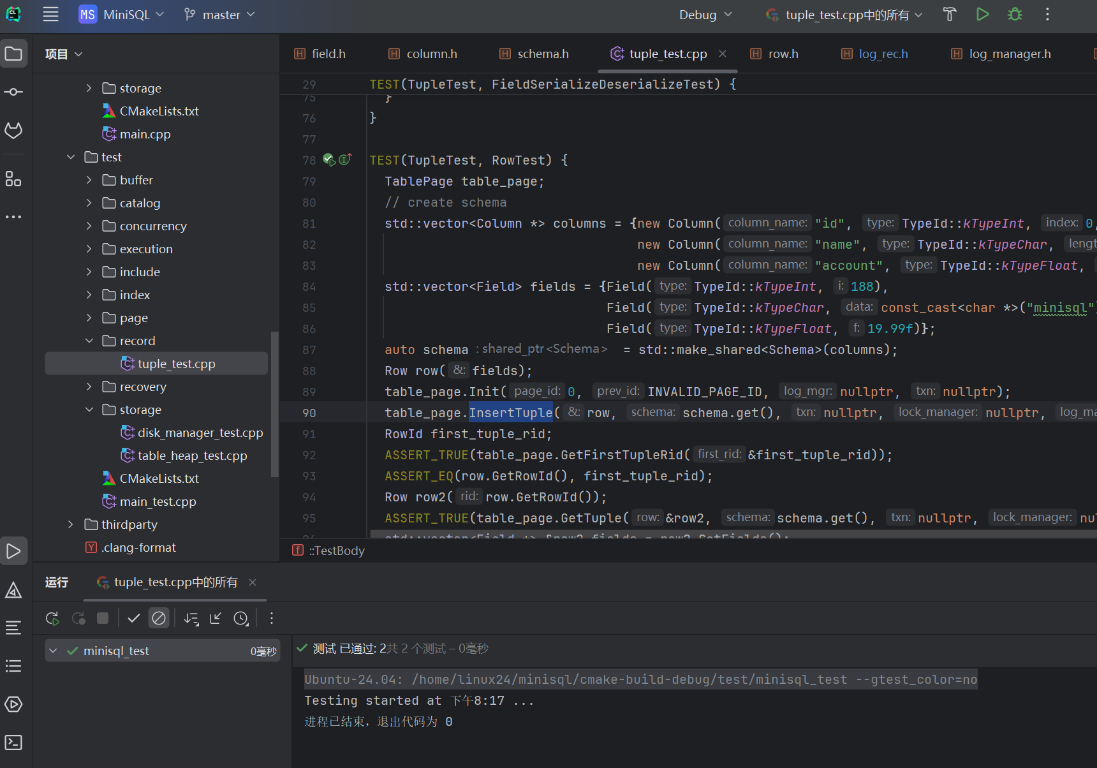
能够根据 RowId 正确地获取并验证插入的数据行。

确保在插入过程中没有重复的 RowId。

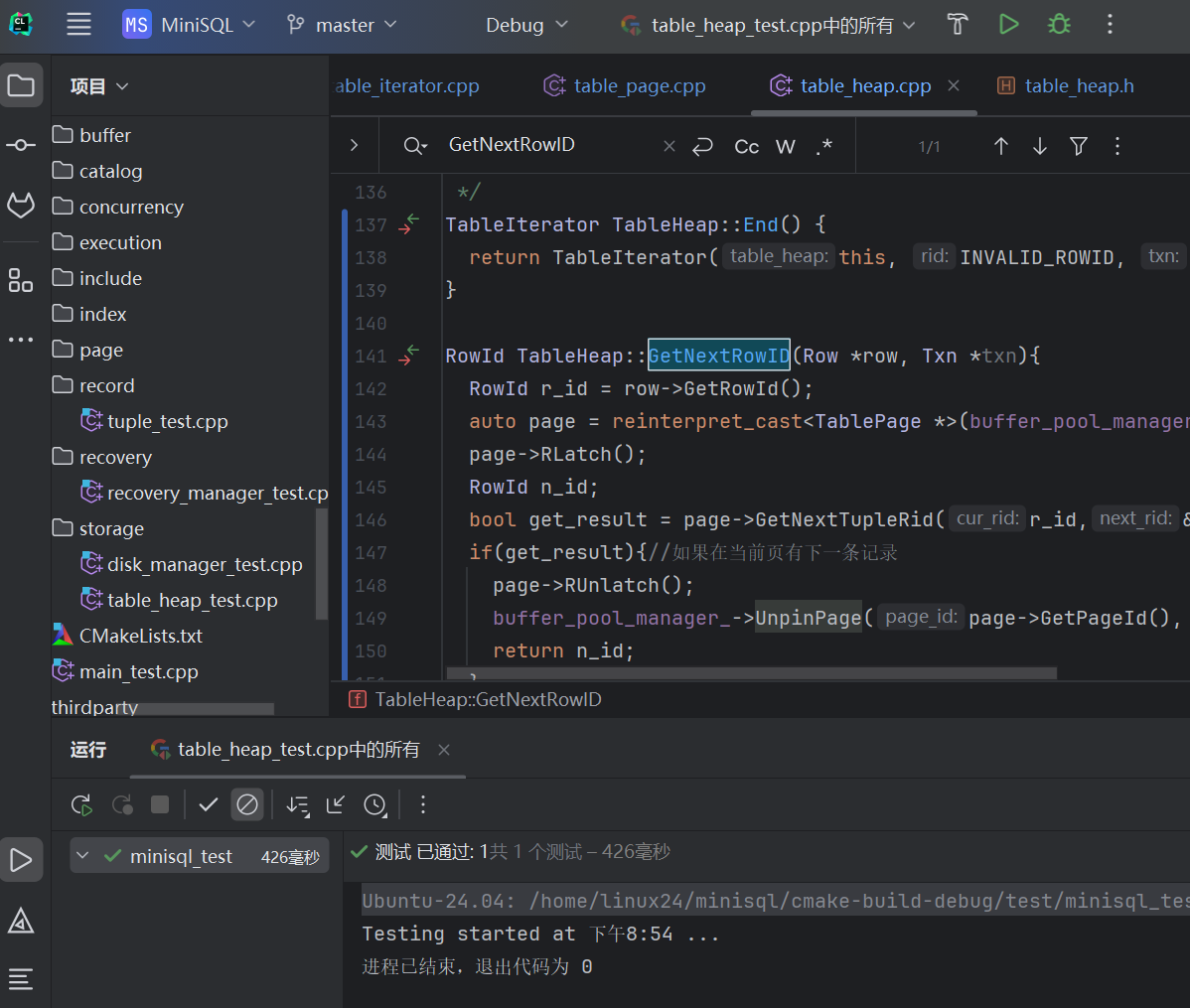
验证了 TableHeap 的基本功能是否按照预期工作。

**（二）结果**

（1）tuple\_test.cpp如下图所示，测试通过。



（2）table\_heap\_test.cpp如下图所示，测试通过：



# #3 INDEX MANAGER

负责人：汪珉凯

一、内容与功能

1、BPlusTreePage

BPlusTreePage类功能描述：

BPlusTreePage是B+树中内部节点（BPlusTreeInternalPage）和叶子节点（BPlusTreeLeafPage）的共同父类，它包含了这两种节点共同需要的数据和方法。

成员变量：

page\_type\_：标记数据页是中间结点还是叶子结点。

key\_size\_：当前索引键的长度。

lsn\_：数据页的日志序列号，用于事务管理，本模块未使用。

size\_：当前节点中存储的键值对数量。

max\_size\_：当前节点最多能够容纳的键值对数量。

parent\_page\_id\_：父节点对应的数据页的page\_id。

page\_id\_：当前节点对应的数据页的page\_id。

方法：

构造函数和析构函数，用于初始化节点和释放资源。

存取方法（getter/setter）用于操作节点的属性。

抽象方法或虚方法，由子类实现具体的操作逻辑，例如节点的分裂、合并等。

2、BPlusTreeInternalPage

BPlusTreeInternalPage类功能描述：

中间节点BPlusTreeInternalPage不存储实际的数据，而是按顺序存储$m$个键和$m+1$个指针（子节点的page\_id）。由于键和指针的数量不相等，第一个键被设置为INVALID，所以顺序查找时从第二个键开始查找。该类还包括对节点半满条件的维护和处理溢出的操作。

特性和操作：

键和指针的存储：使用 std::vector<std::pair<KeyType, page\_id\_t>> keys\_and\_pointers\_; 连续存储键和指针对。

半满条件：节点至少半满，即包含的键值对数量应不少于 (max\_size\_ + 1) / 2。

操作：

插入操作：当插入导致节点溢出时，需要将节点分裂成两个节点。

删除操作：当删除导致节点不满足半满条件时，需要考虑从相邻节点借用或者与相邻节点合并的操作。

分裂操作：将节点按中间键分为左右两部分，左侧的部分留在原节点，右侧的部分成为新节点。

合并操作：将两个相邻的节点合并为一个节点，以减少节点数量。

查找操作：按照键查找子节点的page\_id。

3、BPlusTreeLeafPage

BPlusTreeLeafPage类功能描述：

BPlusTreeLeafPage 类表示 B+ 树中的叶子节点，用于存储实际数据。它按顺序存储了 $m$ 个键和 $m$ 个值。键由一个或多个字段（Field）序列化生成，使用模板参数 KeyType 表示。值存储的是 RowId 的值，使用模板参数 ValueType 表示。与内部节点类似，叶子节点必须遵循键值对数量的约束，并需要处理合并、借用和分裂等操作。

特性和操作：

键值对的存储： 使用 std::vector<std::pair<KeyType, ValueType>> key\_value\_pairs\_; 连续存储键值对。

约束条件： 节点至少半满，即包含的键值对数量应不少于 (max\_size\_ + 1) / 2。

操作：

插入操作： 当插入新的键值对导致节点溢出时，需要将节点分裂成两个节点。

删除操作： 当删除导致节点不满足半满条件时，需要考虑从相邻节点借用或者与相邻节点合并的操作。

分裂操作： 将节点按中间键分为左右两部分，左侧部分留在原节点，右侧部分成为新节点。

合并操作： 将两个相邻的节点合并为一个节点，以减少节点数量。

4、b+ tree index

下面是对于B+树索引的功能化描述：

支持唯一键值（Unique Key）：

B+树索引只允许每个键值对（Key-Value）的键是唯一的。如果尝试插入一个已经存在的键，操作会失败并返回false。

数据结构设计：

B+树由内部节点（Internal Node）和叶子节点（Leaf Node）组成。

内部节点存储键值对的索引和子节点指针，用于导航到适当的叶子节点。

叶子节点存储实际的键值对数据。

操作支持：

创建（Creation）：初始化一个空的B+树。

插入（Insertion）：将新的键值对插入到B+树中。如果键已存在，则插入失败。

删除（Deletion）：从B+树中删除指定键值对。

查找（Search）：根据键查找对应的值。

释放（Release）：释放B+树占用的资源。

根结点变更与持久化：

当进行写操作导致B+树索引的根节点发生变化时，需要调用UpdateRootPageId函数来更新根节点的页面ID，并将此变更持久化到存储介质（例如硬盘）上，以保证数据的持久性和一致性。

节点调整：

Coalesce：当一个节点删除操作后导致节点条目过少时，将其与相邻节点合并。

Redistribute：当节点的条目数过多或过少时，重新分配条目，以保持树的平衡性。

5、b+tree index iterator

以下是对B+树索引迭代器功能的描述：

单向链表组织：

将所有叶子节点组织成一个单向链表，使得可以按照特定方向（例如从小到大）有序遍历叶子节点中的每个键值对。

功能化描述：

Begin()函数：返回指向B+树索引中第一个键值对的迭代器。

End()函数：返回指向B+树索引中最后一个键值对之后的迭代器，通常为nullptr或者特殊标记，表示结束迭代。

遍历操作：

前缀递增运算符（++）：

重载前缀递增运算符++，以实现迭代器的自增操作，将迭代器向前移动到下一个键值对。

解引用运算符（\*）：

重载解引用运算符\*，以允许访问当前迭代器指向的键值对。

相等运算符（==）：

重载相等运算符==，以判断两个迭代器是否指向相同的元素。

不等运算符（!=）：

重载不等运算符!=，以判断两个迭代器是否指向不同的元素。

范围查询支持：

可以利用迭代器实现范围查询，即在给定的键范围内按序访问所有键值对。

实现细节：

迭代器的实现需要考虑B+树结构的特点，如叶子节点的有序性和节点间的链接关系。

使用BPlusTree类中的方法来管理迭代器的移动和访问。

二、代码实现与分析

1、blus\_plus\_tree\_page

1. */\**
2. \* Helper methods to get/set page type
3. \* Page type enum class is defined in b\_plus\_tree\_page.h
4. \*/
5. */\*\**
6. \* TODO: Student Implement
7. \*/
8. bool BPlusTreePage::IsLeafPage() const {
9. if(page\_type\_==IndexPageType::LEAF\_PAGE){
10. return true;
11. }
12. return false;
13. }
14. */\*\**
15. \* TODO: Student Implement
16. \*/
17. bool BPlusTreePage::IsRootPage() const {
18. if(parent\_page\_id\_==INVALID\_PAGE\_ID){
19. return true;
20. }
21. return false;
22. }
23. */\*\**
24. \* TODO: Student Implement
25. \*/
26. void BPlusTreePage::SetPageType(IndexPageType page\_type) {
27. page\_type\_ = page\_type;
28. }
29. int BPlusTreePage::GetKeySize() const {
30. return key\_size\_;
31. }
32. void BPlusTreePage::SetKeySize(int size) {
33. key\_size\_ = size;
34. }
35. */\**
36. \* Helper methods to get/set size (number of key/value pairs stored in that
37. \* page)
38. \*/
39. int BPlusTreePage::GetSize() const {
40. return size\_;
41. }
42. void BPlusTreePage::SetSize(int size) {
43. size\_ = size;
44. }
45. void BPlusTreePage::IncreaseSize(int amount) {
46. size\_ += amount;
47. }
48. */\**
49. \* Helper methods to get/set max size (capacity) of the page
50. \*/
51. */\*\**
52. \* TODO: Student Implement
53. \*/
54. int BPlusTreePage::GetMaxSize() const {
55. return max\_size\_;
56. }
57. */\*\**
58. \* TODO: Student Implement
59. \*/
60. void BPlusTreePage::SetMaxSize(int size) {
61. max\_size\_ = size;
62. }
63. */\**
64. \* Helper method to get min page size
65. \* Generally, min page size == max page size / 2
66. \*/
67. */\*\**
68. \* TODO: Student Implement
69. \*/
70. int BPlusTreePage::GetMinSize() const {
71. return max\_size\_/2;
72. }
73. */\**
74. \* Helper methods to get/set parent page id
75. \*/
76. */\*\**
77. \* TODO: Student Implement
78. \*/
79. page\_id\_t BPlusTreePage::GetParentPageId() const {
80. return parent\_page\_id\_;
81. }
82. void BPlusTreePage::SetParentPageId(page\_id\_t parent\_page\_id) {
83. parent\_page\_id\_ = parent\_page\_id;
84. }
85. */\**
86. \* Helper methods to get/set self page id
87. \*/
88. page\_id\_t BPlusTreePage::GetPageId() const {
89. return page\_id\_;
90. }
91. void BPlusTreePage::SetPageId(page\_id\_t page\_id) {
92. page\_id\_ = page\_id;
93. }
94. */\**
95. \* Helper methods to set lsn
96. \*/
97. void BPlusTreePage::SetLSN(lsn\_t lsn) {
98. lsn\_ = lsn;
99. }

bool BPlusTreePage::IsLeafPage() const

功能： 判断当前页面是否为叶子页面。

实现分析：

如果 page\_type\_ 等于 IndexPageType::LEAF\_PAGE，则返回 true，表示当前页面是叶子页面。

否则，返回 false。

 bool BPlusTreePage::IsRootPage() const

功能： 判断当前页面是否为根页面。

实现分析：

如果 parent\_page\_id\_ 等于 INVALID\_PAGE\_ID，则返回 true，表示当前页面是根页面。

否则，返回 false。

void BPlusTreePage::SetPageType(IndexPageType page\_type)

功能： 设置页面的类型。

实现分析：

将 page\_type\_ 成员变量设置为传入的 page\_type 值。

 int BPlusTreePage::GetMaxSize() const

功能： 获取页面的最大容量。

实现分析：

直接返回 max\_size\_ 成员变量的值，表示页面可以容纳的最大项数。

void BPlusTreePage::SetMaxSize(int size)

功能： 设置页面的最大容量。

实现分析：

将 max\_size\_ 成员变量设置为传入的 size 值。

int BPlusTreePage::GetMinSize() const

功能： 获取页面的最小容量。

实现分析：

根据通常的规则，最小页面大小是 max\_size\_ / 2。

直接返回 max\_size\_ / 2 的计算结果。

page\_id\_t BPlusTreePage::GetParentPageId() const

功能： 获取当前页面的父页面ID。

实现分析：

直接返回 parent\_page\_id\_ 成员变量的值。

int BPlusTreePage::GetKeySize() const 和 void BPlusTreePage::SetKeySize(int size)：用于获取和设置页面中键的大小。

int BPlusTreePage::GetSize() const、void BPlusTreePage::SetSize(int size) 和 void BPlusTreePage::IncreaseSize(int amount)：用于获取、设置和增加页面中项的数量。

page\_id\_t BPlusTreePage::GetPageId() const 和 void BPlusTreePage::SetPageId(page\_id\_t page\_id)：用于获取和设置当前页面的ID。

void BPlusTreePage::SetLSN(lsn\_t lsn)：用于设置页面的LSN（日志序列号），用于事务管理和恢复。

这些函数构成了对B+树页面对象的基本操作和属性访问方法。通过这些方法，可以获取和设置页面的类型、大小、ID、父页面ID等信息，以及在需要时进行相关的增减操作或状态检查。

2、b\_plus\_tree\_internal\_page

1. */\*\**
2. \* TODO: Student Implement
3. \*/
4. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**
5. \* HELPER METHODS AND UTILITIES
6. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
7. */\**
8. \* Init method after creating a new internal page
9. \* Including set page type, set current size, set page id, set parent id and set
10. \* max page size
11. \*/
12. void InternalPage::Init(page\_id\_t page\_id, page\_id\_t parent\_id, int key\_size, int max\_size) {
13. SetPageId(page\_id);
14. SetParentPageId(parent\_id);
15. SetKeySize(key\_size);
16. SetMaxSize(max\_size);
17. SetSize(0);
18. SetPageType(IndexPageType::INTERNAL\_PAGE);
19. }
20. */\**
21. \* Helper method to get/set the key associated with input "index"(a.k.a
22. \* array offset)
23. \*/
24. GenericKey \*InternalPage::KeyAt(int index) {
25. return reinterpret\_cast<GenericKey \*>(pairs\_off + index \* pair\_size + key\_off);
26. }
27. void InternalPage::SetKeyAt(int index, GenericKey \*key) {
28. memcpy(pairs\_off + index \* pair\_size + key\_off, key, GetKeySize());
29. }
30. page\_id\_t InternalPage::ValueAt(int index) const {
31. return \*reinterpret\_cast<const page\_id\_t \*>(pairs\_off + index \* pair\_size + val\_off);
32. }
33. void InternalPage::SetValueAt(int index, page\_id\_t value) {
34. \*reinterpret\_cast<page\_id\_t \*>(pairs\_off + index \* pair\_size + val\_off) = value;
35. }
36. int InternalPage::ValueIndex(const page\_id\_t &value) const {
37. for (int i = 0; i < GetSize(); ++i) {
38. if (ValueAt(i) == value) return i;
39. }
40. return -1;
41. }
42. void \*InternalPage::PairPtrAt(int index) { return KeyAt(index); }
43. void InternalPage::PairCopy(void \*dest, void \*src, int pair\_num) {
44. memcpy(dest, src, pair\_num \* (GetKeySize() + sizeof(page\_id\_t)));
45. }
46. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**
47. \* LOOKUP
48. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
49. */\**
50. \* Find and return the child pointer(page\_id) which points to the child page
51. \* that contains input "key"
52. \* Start the search from the second key(the first key should always be invalid)
53. \* 用了二分查找
54. \*/
55. page\_id\_t InternalPage::Lookup(const GenericKey \*key, const KeyManager &KM) {
56. *// binary search*
57. int left = 1, right = GetSize() - 1;
58. while (left <= right) {
59. int mid = (left + right) / 2;
60. if (KM.CompareKeys(KeyAt(mid), key) < 0) {
61. left = mid + 1;
62. } else if (KM.CompareKeys(KeyAt(mid), key) == 0) {
63. right = mid;
64. } else {
65. right = mid - 1;
66. }
67. }
68. return -1;
69. }
70. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**
71. \* INSERTION
72. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
73. */\**
74. \* Populate new root page with old\_value + new\_key & new\_value
75. \* When the insertion cause overflow from leaf page all the way upto the root
76. \* page, you should create a new root page and populate its elements.
77. \* NOTE: This method is only called within InsertIntoParent()(b\_plus\_tree.cpp)
78. \*/
79. void InternalPage::PopulateNewRoot(const page\_id\_t &old\_value, GenericKey \*new\_key, const page\_id\_t &new\_value) {
80. *// populate new root page*
81. SetSize(2);
82. SetValueAt(0, old\_value);
83. SetValueAt(1, new\_value);
84. SetKeyAt(1, new\_key);
85. }
86. */\**
87. \* Insert new\_key & new\_value pair right after the pair with its value ==
88. \* old\_value
89. \* @return:  new size after insertion
90. \*/
91. int InternalPage::InsertNodeAfter(const page\_id\_t &old\_value, GenericKey \*new\_key, const page\_id\_t &new\_value) {
92. int index = ValueIndex(old\_value);
93. if (index == -1) return GetSize();
94. *// move all pairs after index to right*
95. for (int i = GetSize() - 1; i > index; i--) {
96. SetValueAt(i + 1, ValueAt(i));
97. SetKeyAt(i + 1, KeyAt(i));
98. }
99. *// insert new pair*
100. SetValueAt(index + 1, new\_value);
101. SetKeyAt(index + 1, new\_key);
102. SetSize(GetSize() + 1);
103. return GetSize();
104. }
105. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**
106. \* SPLIT
107. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
108. */\**
109. \* Remove half of key & value pairs from this page to "recipient" page
110. \* buffer\_pool\_manager 是干嘛的？传给CopyNFrom()用于Fetch数据页
111. \*/
112. void InternalPage::MoveHalfTo(InternalPage \*recipient, BufferPoolManager \*buffer\_pool\_manager) {
113. int half = GetSize() / 2;
114. *// copy last half to recipient*
115. recipient->CopyNFrom(PairPtrAt(GetSize()-half),half, buffer\_pool\_manager);
116. *// update size*
117. SetSize(GetSize()-half);
118. }
119. */\* Copy entries into me, starting from {items} and copy {size} entries.*
120. \* Since it is an internal page, for all entries (pages) moved, their parents page now changes to me.
121. \* So I need to 'adopt' them by changing their parent page id, which needs to be persisted with BufferPoolManger
122. \*
123. \*/
124. void InternalPage::CopyNFrom(void \*src, int size, BufferPoolManager \*buffer\_pool\_manager) {
125. PairCopy(PairPtrAt(GetSize()),src,size);*//copy entries*
126. SetSize(GetSize()+size);*//update size*
127. for(int i=1;i<=size;i++) {
128. int page\_id = ValueAt(GetSize()-i);
129. *// fetch page*
130. auto \*page = reinterpret\_cast<BPlusTreePage \*>(buffer\_pool\_manager->FetchPage(page\_id)->GetData());
131. *// update parent page id*
132. page->SetParentPageId(GetPageId());
133. *// unpin page*
134. buffer\_pool\_manager->UnpinPage(page\_id,true);
135. }
136. }
137. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**
138. \* REMOVE
139. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
140. */\**
141. \* Remove the key & value pair in internal page according to input index(a.k.a
142. \* array offset)
143. \* NOTE: store key&value pair continuously after deletion
144. \*/
145. void InternalPage::Remove(int index) {
146. *// remove*
147. for (int i = index; i < GetSize() - 1; i++) {
148. SetValueAt(i, ValueAt(i + 1));
149. SetKeyAt(i, KeyAt(i + 1));
150. }
151. SetSize(GetSize() - 1);
152. }
153. */\**
154. \* Remove the only key & value pair in internal page and return the value
155. \* NOTE: only call this method within AdjustRoot()(in b\_plus\_tree.cpp)
156. \*/
157. page\_id\_t InternalPage::RemoveAndReturnOnlyChild() {
158. if (GetSize() == 1) {
159. auto result = ValueAt(0);
160. Remove(0);
161. return result;
162. }
163. return INVALID\_PAGE\_ID;
164. }
165. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**
166. \* MERGE
167. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
168. */\**
169. \* Remove all of key & value pairs from this page to "recipient" page.
170. \* The middle\_key is the separation key you should get from the parent. You need
171. \* to make sure the middle key is added to the recipient to maintain the invariant.
172. \* You also need to use BufferPoolManager to persist changes to the parent page id for those
173. \* pages that are moved to the recipient
174. \*/
175. void InternalPage::MoveAllTo(InternalPage \*recipient, GenericKey \*middle\_key, BufferPoolManager \*buffer\_pool\_manager) {
176. SetKeyAt(0, middle\_key);*//set middle key*
177. recipient->CopyNFrom(PairPtrAt(0), GetSize(), buffer\_pool\_manager);*//copy all entries*
178. SetSize(0);*//update size*
179. }
180. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**
181. \* REDISTRIBUTE
182. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
183. */\**
184. \* Remove the first key & value pair from this page to tail of "recipient" page.
185. \*
186. \* The middle\_key is the separation key you should get from the parent. You need
187. \* to make sure the middle key is added to the recipient to maintain the invariant.
188. \* You also need to use BufferPoolManager to persist changes to the parent page id for those
189. \* pages that are moved to the recipient
190. \*/
191. void InternalPage::MoveFirstToEndOf(InternalPage \*recipient, GenericKey \*middle\_key,
192. BufferPoolManager \*buffer\_pool\_manager) {
193. SetKeyAt(0, middle\_key);                                             *// set middle key*
194. recipient->CopyLastFrom(KeyAt(0), ValueAt(0), buffer\_pool\_manager);  *// copy first entry*
195. Remove(0);                                                           *// remove first entry*
196. }
197. */\* Append an entry at the end.*
198. \* Since it is an internal page, the moved entry(page)'s parent needs to be updated.
199. \* So I need to 'adopt' it by changing its parent page id, which needs to be persisted with BufferPoolManger
200. \*/
201. void InternalPage::CopyLastFrom(GenericKey \*key, const page\_id\_t value, BufferPoolManager \*buffer\_pool\_manager) {
202. SetValueAt(GetSize(), value);
203. SetKeyAt(GetSize(), key);
204. SetSize(GetSize() + 1);
205. *// fetch page*
206. auto \*page = reinterpret\_cast<BPlusTreePage \*>(buffer\_pool\_manager->FetchPage(value)->GetData());
207. *// update parent page id*
208. page->SetParentPageId(GetPageId());
209. *// unpin page*
210. buffer\_pool\_manager->UnpinPage(value, true);
211. }
212. */\**
213. \* Remove the last key & value pair from this page to head of "recipient" page.
214. \* You need to handle the original dummy key properly, e.g. updating recipient’s array to position the middle\_key at
215. \* the right place. You also need to use BufferPoolManager to persist changes to the parent page id for those pages that
216. \* are moved to the recipient
217. \*/
218. void InternalPage::MoveLastToFrontOf(InternalPage \*recipient, GenericKey \*middle\_key,
219. BufferPoolManager \*buffer\_pool\_manager) {
220. recipient->CopyFirstFrom(ValueAt(GetSize() - 1), buffer\_pool\_manager);  *// copy last entry*
221. recipient->SetKeyAt(1, middle\_key);                                     *// set middle key*
222. Remove(GetSize() - 1);                                                  *// remove last entry*
223. }
224. */\* Append an entry at the beginning.*
225. \* Since it is an internal page, the moved entry(page)'s parent needs to be updated.
226. \* So I need to 'adopt' it by changing its parent page id, which needs to be persisted with BufferPoolManger
227. \*/
228. void InternalPage::CopyFirstFrom(const page\_id\_t value, BufferPoolManager \*buffer\_pool\_manager) {
229. *// copy first entry*
230. PairCopy(PairPtrAt(1), PairPtrAt(0), GetSize());
231. *// fetch page*
232. SetValueAt(0, value);
233. SetSize(GetSize() + 1);
234. *// fetch page*
235. auto \*page = reinterpret\_cast<BPlusTreePage \*>(buffer\_pool\_manager->FetchPage(value)->GetData());
236. *// update parent page id*
237. page->SetParentPageId(GetPageId());
238. *// unpin page*
239. buffer\_pool\_manager->UnpinPage(value, true);
240. }

1. void InternalPage::Init(page\_id\_t page\_id, page\_id\_t parent\_id, int key\_size, int max\_size)

功能： 初始化新创建的内部页。

实现分析：

SetPageId(page\_id)：设置页面ID为传入的 page\_id。

SetParentPageId(parent\_id)：设置父页面ID为传入的 parent\_id。

SetKeySize(key\_size)：设置键的大小。

SetMaxSize(max\_size)：设置最大页面大小。

SetSize(0)：将当前页面的大小设置为0。

SetPageType(IndexPageType::INTERNAL\_PAGE)：设置页面类型为内部页面。

2. GenericKey \*InternalPage::KeyAt(int index)

功能： 获取指定索引位置的键。

实现分析：

使用偏移量计算出指定索引位置的键的内存地址，并将其转换为 GenericKey\* 类型返回。

3. void InternalPage::SetKeyAt(int index, GenericKey \*key)

功能： 设置指定索引位置的键。

实现分析：

使用 memcpy 将传入的 key 复制到指定索引位置的内存地址中。

4. page\_id\_t InternalPage::ValueAt(int index) const

功能： 获取指定索引位置的值（页面ID）。

实现分析：

使用偏移量计算出指定索引位置的值的内存地址，并将其解释为 page\_id\_t 类型后返回。

5. void InternalPage::SetValueAt(int index, page\_id\_t value)

功能： 设置指定索引位置的值（页面ID）。

实现分析：

将传入的 value 直接存储到指定索引位置的内存地址中。

6. int InternalPage::ValueIndex(const page\_id\_t &value) const

功能： 查找给定值（页面ID）在页面中的索引位置。

实现分析：

从索引0开始遍历页面中的值，如果找到与传入的 value 相等的值，则返回该索引位置。

如果循环结束都没有找到相等的值，则返回 -1 表示未找到。

7. void \*InternalPage::PairPtrAt(int index)

功能： 获取指定索引位置的键值对指针。

实现分析：

直接调用 KeyAt(index) 方法返回键的指针。

8. void InternalPage::PairCopy(void \*dest, void \*src, int pair\_num)

功能： 复制键值对。

实现分析：

使用 memcpy 将 src 指向的键值对数据复制到 dest 指向的内存位置。

复制的字节数为 pair\_num \* (GetKeySize() + sizeof(page\_id\_t))，即每对键值对占用的总字节数。

9. page\_id\_t InternalPage::Lookup(const GenericKey \*key, const KeyManager &KM)

功能： 使用二分查找方法查找包含指定键的子页面ID。

实现分析：

初始化二分查找的左右边界，从第二个键开始查找（因为第一个键应该始终无效）。

在 left 到 right 范围内进行二分查找：

计算中间位置 mid。

根据 KeyManager 对象 KM 比较中间位置的键和传入的 key：

如果中间键小于 key，则将查找范围缩小到右半部分。

如果中间键等于 key，则直接返回 mid，表示找到了。

如果中间键大于 key，则将查找范围缩小到左半部分。

如果未找到匹配的键，则返回 -1。

10. void InternalPage::PopulateNewRoot(const page\_id\_t &old\_value, GenericKey \*new\_key, const page\_id\_t &new\_value)

功能： 在创建新根节点时填充数据。

实现分析：

SetSize(2);：将当前页面大小设置为2，表示新根节点有两个子节点。

SetValueAt(0, old\_value);：将索引为0的值设为旧子节点的ID。

SetValueAt(1, new\_value);：将索引为1的值设为新子节点的ID。

SetKeyAt(1, new\_key);：将索引为1的键设为新的分裂键。

11. int InternalPage::InsertNodeAfter(const page\_id\_t &old\_value, GenericKey \*new\_key, const page\_id\_t &new\_value)

功能： 在指定子节点后插入新的键值对。

实现分析：

int index = ValueIndex(old\_value);：查找当前节点中old\_value的索引。

如果未找到old\_value，则返回当前页面大小GetSize()。

否则，将从index+1开始的所有键值对向右移动，为新的键值对腾出空间。

在索引index+1处插入新的键值对new\_key和new\_value。

使用SetSize(GetSize() + 1)更新页面大小，并返回更新后的大小。

12. void InternalPage::MoveHalfTo(InternalPage \*recipient, BufferPoolManager \*buffer\_pool\_manager)

功能： 将当前页面后半部分的条目移动到接收页面。

实现分析：

int half = GetSize() / 2;：计算当前页面大小的一半。

使用CopyNFrom将当前页面后半部分的键值对复制到接收页面。

使用SetSize(GetSize() - half)将当前页面的大小更新为原始大小的一半。

13. void InternalPage::CopyNFrom(void \*src, int size, BufferPoolManager \*buffer\_pool\_manager)

功能： 从源地址复制指定数量的键值对到当前页面的末尾。

实现分析：

PairCopy(PairPtrAt(GetSize()), src, size);：调用PairCopy函数，将源地址src的键值对复制到当前页面的末尾。

使用SetSize(GetSize() + size)更新当前页面的大小。

14. void InternalPage::Remove(int index)

功能： 移除指定索引处的键值对。

实现分析：

从指定索引开始，将所有后续的键值对向左移动一位。

使用SetSize(GetSize() - 1)更新页面的大小。

15. page\_id\_t InternalPage::RemoveAndReturnOnlyChild()

功能： 移除并返回唯一的子节点ID。

实现分析：

如果当前页面大小为1：

使用ValueAt(0)获取索引为0的子节点ID。

使用Remove(0)移除第一个键值对。

返回获取到的子节点ID。

如果当前页面大小不为1，则返回INVALID\_PAGE\_ID。

16. void InternalPage::MoveAllTo(InternalPage \*recipient, GenericKey \*middle\_key, BufferPoolManager \*buffer\_pool\_manager)

功能： 将当前页面的所有键值对移动到接收页面，并设置分裂键。

实现分析：

SetKeyAt(0, middle\_key);：将当前页面的第一个键设置为分裂键。

调用接收页面的CopyNFrom方法，将当前页面的所有键值对复制到接收页面。

使用SetSize(0)将当前页面的大小更新为0。

3、b\_plus\_tree\_leafpage

void LeafPage::Init(page\_id\_t page\_id, page\_id\_t parent\_id, int key\_size, int max\_size) {

  //init page

  SetPageId(page\_id);

  SetPageType(IndexPageType::LEAF\_PAGE);

  SetParentPageId(parent\_id);

  SetSize(0);

  SetKeySize(key\_size);

  SetMaxSize(max\_size);

  SetNextPageId(INVALID\_PAGE\_ID);

}

/\*\*

 \* Helper methods to set/get next page id

 \*/

page\_id\_t LeafPage::GetNextPageId() const {

  return next\_page\_id\_;

}

void LeafPage::SetNextPageId(page\_id\_t next\_page\_id) {

  next\_page\_id\_ = next\_page\_id;

  if (next\_page\_id == 0) {

    LOG(INFO) << "Fatal error";

  }

}

/\*\*

 \* 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

 \* 二分查找

 \*/

int LeafPage::KeyIndex(const GenericKey \*key, const KeyManager &KM) {

  //if size=0

        if (GetSize() == 0) {

        return -1;

        }

  //binary search

  int index = 0;

  int left = 0;

  int right = GetSize() - 1;

  while(left<=right){

    int mid = (left + right) / 2;

    if(KM.CompareKeys(KeyAt(mid), key) < 0){

      left = mid + 1;

    }else if(KM.CompareKeys(KeyAt(mid), key) > 0){

      index = mid;

      right = mid -1;

    }else {

      index = mid;

      break;

    }

  }

  return index;//if not found

}

/\*

 \* Helper method to find and return the key associated with input "index"(a.k.a

 \* array offset)

 \*/

GenericKey \*LeafPage::KeyAt(int index) {

  return reinterpret\_cast<GenericKey \*>(pairs\_off + index \* pair\_size + key\_off);

}

void LeafPage::SetKeyAt(int index, GenericKey \*key) {

  memcpy(pairs\_off + index \* pair\_size + key\_off, key, GetKeySize());

}

RowId LeafPage::ValueAt(int index) const {

  return \*reinterpret\_cast<const RowId \*>(pairs\_off + index \* pair\_size + val\_off);

}

void LeafPage::SetValueAt(int index, RowId value) {

  \*reinterpret\_cast<RowId \*>(pairs\_off + index \* pair\_size + val\_off) = value;

}

void \*LeafPage::PairPtrAt(int index) {

  return KeyAt(index);

}

void LeafPage::PairCopy(void \*dest, void \*src, int pair\_num) {

  memcpy(dest, src, pair\_num \* (GetKeySize() + sizeof(RowId)));

}

/\*

 \* Helper method to find and return the key & value pair associated with input

 \* "index"(a.k.a. array offset)

 \*/

std::pair<GenericKey \*, RowId> LeafPage::GetItem(int index) { return {KeyAt(index), ValueAt(index)}; }

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 \* 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 size=GetSize();

  if(size==0) {

    SetKeyAt(0, key);//set key

    SetValueAt(0, value);//set value

    SetSize(1);//set size

    return 1;

  }

  // insert leaf page

  int index = KeyIndex(key, KM);

  // move the key-value pairs

  PairCopy(PairPtrAt(index + 1), PairPtrAt(index), GetSize() - index);

  // insert key-value pair

  SetKeyAt(index, key);

  SetValueAt(index, value);

  // update size

  IncreaseSize(1);

  return GetSize();

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 \* SPLIT

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

 \* Remove half of key & value pairs from this page to "recipient" page

 \*/

void LeafPage::MoveHalfTo(LeafPage \*recipient) {

  int half = GetSize() / 2;

  recipient->CopyNFrom(PairPtrAt(GetSize() - half), half);

  SetSize(GetSize() - half);

}

/\*

 \* Copy starting from items, and copy {size} number of elements into me.

 \*/

void LeafPage::CopyNFrom(void \*src, int size) {

  PairCopy(PairPtrAt(GetSize()), src, size);

  IncreaseSize(size);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 \* 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) {

  int index = KeyIndex(key, KM);

  if (index < GetSize() && KM.CompareKeys(KeyAt(index), key) == 0) {

    value = ValueAt(index);

    return true;

  }

  return false;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 \* 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) {

  int index = KeyIndex(key, KM);

  if (index < GetSize() && KM.CompareKeys(KeyAt(index), key) == 0) {

    PairCopy(PairPtrAt(index), PairPtrAt(index + 1), GetSize() - index - 1);

    SetSize(GetSize() - 1);

  }

  return GetSize();

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 \* 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(PairPtrAt(0), GetSize());

  recipient->SetNextPageId(GetNextPageId());

  SetSize(0);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 \* REDISTRIBUTE

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

 \* Remove the first key & value pair from this page to "recipient" page.

 \*

 \*/

void LeafPage::MoveFirstToEndOf(LeafPage \*recipient) {

  recipient->CopyLastFrom(KeyAt(0), ValueAt(0));

  PairCopy(PairPtrAt(0), PairPtrAt(1), GetSize() - 1);

  SetSize(GetSize() - 1);

}

/\*

 \* Copy the item into the end of my item list. (Append item to my array)

 \*/

void LeafPage::CopyLastFrom(GenericKey \*key, const RowId value) {

  SetValueAt(GetSize(), value);

  SetKeyAt(GetSize(), key);

  SetSize(GetSize() + 1);

}

/\*

 \* Remove the last key & value pair from this page to "recipient" page.

 \*/

void LeafPage::MoveLastToFrontOf(LeafPage \*recipient) {

  // move the last key-value pair to the front of recipient

  recipient->CopyFirstFrom(KeyAt(GetSize() - 1), ValueAt(GetSize() - 1));

  // update size

  SetSize(GetSize() - 1);

}

/\*

 \* Insert item at the front of my items. Move items accordingly.

 \*

 \*/

void LeafPage::CopyFirstFrom(GenericKey \*key, const RowId value) {

  PairCopy(PairPtrAt(1), PairPtrAt(0), GetSize());

  SetValueAt(0, value);

  SetKeyAt(0, key);

  IncreaseSize(1);

}

Init 方法：这个方法用于初始化 LeafPage 对象，设置提供的 page\_id、parent\_id、key\_size 和 max\_size。它将页面类型设置为 LEAF\_PAGE，将页面大小初始化为零，并设置其他属性如键大小、最大大小和下一个页面的ID。

GetNextPageId：返回叶子页面的 next\_page\_id。

SetNextPageId：设置叶子页面的 next\_page\_id。如果 next\_page\_id 为 0，则记录一个致命错误。

KeyIndex：执行二分查找，找到 key 应该插入的位置或者找到已存在的键在 LeafPage 中的位置。它使用 KeyManager 来比较键。

KeyAt：返回 LeafPage 中指定索引处的键的指针。

SetKeyAt：使用 memcpy 在 LeafPage 中的指定索引处设置键。

ValueAt：返回 LeafPage 中指定索引处的值（RowId 类型）。

SetValueAt：在 LeafPage 中的指定索引处设置值（RowId 类型）。

PairPtrAt：返回 LeafPage 中指定索引处的键的指针。

PairCopy：从 src 复制键值对到 dest。

GetItem：返回指定索引处的键值对（使用 std::pair 包含键和值）。

Insert：按照键的顺序将键值对插入叶子页面。它使用 KeyIndex 找到插入点，必要时移动现有键值对，并插入新的键值对。

MoveHalfTo：将当前叶子页面的一半键值对移动到 recipient 叶子页面。

CopyNFrom：从 src 复制 size 个键值对到当前叶子页面。

bool LeafPage::Lookup(const GenericKey \*key, RowId &value, const KeyManager &KM)

使用 KeyIndex(key, KM) 方法找到键 key 在叶子页面中的索引 index。

如果 index 小于当前页面大小 (GetSize()) 并且通过 KM.CompareKeys(KeyAt(index), key) 比较键是否相等，如果相等，则将对应的值 ValueAt(index) 赋给 value，并返回 true。

如果未找到匹配的键，则返回 false。

int LeafPage::RemoveAndDeleteRecord(const GenericKey \*key, const KeyManager &KM)

使用 KeyIndex(key, KM) 方法找到键 key 在叶子页面中的索引 index。

如果 index 小于当前页面大小 (GetSize()) 并且通过 KM.CompareKeys(KeyAt(index), key) 比较键是否相等，表示找到匹配的键值对。

使用 PairCopy 方法将从 index + 1 开始的键值对向前移动，覆盖掉索引 index 处的键值对。

减小页面大小 (SetSize(GetSize() - 1)).

返回更新后的页面大小。

void LeafPage::MoveAllTo(LeafPage \*recipient)

使用 recipient 的 CopyNFrom 方法，将当前叶子页面从索引 0 到 GetSize() 的所有键值对复制到 recipient 页面。

设置 recipient 页面的下一个页面ID (recipient->SetNextPageId(GetNextPageId()))。

清空当前叶子页面的大小 (SetSize(0)).

void LeafPage::MoveFirstToEndOf(LeafPage \*recipient)

使用 recipient 的 CopyLastFrom 方法，将当前叶子页面的第一个键值对 (KeyAt(0) 和 ValueAt(0)) 复制到 recipient 页面的末尾。

使用 PairCopy 方法将当前叶子页面的从索引 1 开始的所有键值对向前移动，覆盖掉索引 0 处的键值对。

减小当前叶子页面的大小 (SetSize(GetSize() - 1)).

void LeafPage::CopyLastFrom(GenericKey \*key, const RowId value)

使用 SetValueAt(GetSize(), value) 将给定的值插入到当前叶子页面的末尾。

使用 SetKeyAt(GetSize(), key) 将给定的键插入到当前叶子页面的末尾。

增加当前叶子页面的大小 (SetSize(GetSize() + 1)).

void LeafPage::MoveLastToFrontOf(LeafPage \*recipient)

使用 recipient 的 CopyFirstFrom 方法，将当前叶子页面的最后一个键值对 (KeyAt(GetSize() - 1) 和 ValueAt(GetSize() - 1)) 复制到 recipient 页面的开头。

减小当前叶子页面的大小 (SetSize(GetSize() - 1)).

void LeafPage::CopyFirstFrom(GenericKey \*key, const RowId value)

使用 PairCopy 方法将当前叶子页面从索引 0 开始的所有键值对向后移动，空出索引 0 处。

使用 SetValueAt(0, value) 将给定的值插入到当前叶子页面的开头。

使用 SetKeyAt(0, key) 将给定的键插入到当前叶子页面的开头。

增加当前叶子页面的大小 (IncreaseSize(1)).

4、b\_plus\_tree

1. */\*\**
2. \* TODO: Student Implement
3. \*/
4. BPlusTree::BPlusTree(index\_id\_t index\_id, BufferPoolManager \*buffer\_pool\_manager, const KeyManager &KM,
5. int leaf\_max\_size, int internal\_max\_size)
6. : index\_id\_(index\_id),
7. buffer\_pool\_manager\_(buffer\_pool\_manager),
8. processor\_(KM),
9. leaf\_max\_size\_(leaf\_max\_size),
10. internal\_max\_size\_(internal\_max\_size) {
11. *//fetch page which contains root ids of all indexes*
12. Page \*page = buffer\_pool\_manager\_->FetchPage(INDEX\_ROOTS\_PAGE\_ID);
13. if(leaf\_max\_size==UNDEFINED\_SIZE) {
14. leaf\_max\_size\_=(PAGE\_SIZE - LEAF\_PAGE\_HEADER\_SIZE) / (processor\_.GetKeySize() + sizeof(RowId));
15. }
16. if(internal\_max\_size==UNDEFINED\_SIZE) {
17. internal\_max\_size\_=(PAGE\_SIZE - INTERNAL\_PAGE\_HEADER\_SIZE) / (processor\_.GetKeySize() + sizeof(page\_id\_t));
18. }
19. *//get data of the page*
20. IndexRootsPage \*index\_roots\_page = reinterpret\_cast<IndexRootsPage \*>(page->GetData());
21. *//get root page id of this index*
22. auto it = index\_roots\_page->GetRootId(index\_id,&root\_page\_id\_);
23. if(!it){
24. *//if root page id is not found, create a new root page*
25. root\_page\_id\_ = INVALID\_PAGE\_ID;
26. }
27. *//unpin the page*
28. buffer\_pool\_manager\_->UnpinPage(INDEX\_ROOTS\_PAGE\_ID, true);
29. if(root\_page\_id\_!=INVALID\_PAGE\_ID) {
30. buffer\_pool\_manager\_->UnpinPage(root\_page\_id\_, true);
31. }
32. }
33. void BPlusTree::Destroy(page\_id\_t current\_page\_id) {
34. if (IsEmpty()) return;
35. *// fetch the page*
36. Page \*page = buffer\_pool\_manager\_->FetchPage(current\_page\_id);
37. *// get the data of the page*
38. BPlusTreePage \*b\_plus\_tree\_page = reinterpret\_cast<BPlusTreePage \*>(page->GetData());
39. *// if the page is leaf page, delete the page*
40. if (b\_plus\_tree\_page->IsLeafPage()) {
41. buffer\_pool\_manager\_->DeletePage(current\_page\_id);
42. buffer\_pool\_manager\_->UnpinPage(current\_page\_id, false);
43. return;
44. }
45. *// if the page is internal page, recursively delete the children*
46. auto \*internal\_page = reinterpret\_cast<InternalPage \*>(b\_plus\_tree\_page);
47. for (int i = 0; i < internal\_page->GetSize(); i++) {
48. Destroy(internal\_page->ValueAt(i));
49. }
50. *// delete the page*
51. buffer\_pool\_manager\_->DeletePage(current\_page\_id);
52. buffer\_pool\_manager\_->UnpinPage(current\_page\_id, false);
53. }
54. */\**
55. \* Helper function to decide whether current b+tree is empty
56. \*/
57. bool BPlusTree::IsEmpty() const {
58. if (root\_page\_id\_ == INVALID\_PAGE\_ID) {
59. return true;
60. }
61. return false;
62. }
63. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**
64. \* SEARCH
65. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
66. */\**
67. \* Return the only value that associated with input key
68. \* This method is used for point query
69. \* @return : true means key exists
70. \*/
71. bool BPlusTree::GetValue(const GenericKey \*key, std::vector<RowId> &result, Txn \*transaction) {
72. if (IsEmpty()) return false;
73. *// find the leaf page containing the key*
74. Page \*page = FindLeafPage(key, root\_page\_id\_, false);
75. *// get the data of the page*
76. auto \*leaf\_page = reinterpret\_cast<LeafPage \*>(page->GetData());
77. *// find the key in the leaf page*
78. int index = leaf\_page->KeyIndex(key, processor\_);
79. if (index == -1) {
80. *// if the key is not found, unpin the page and return false*
81. buffer\_pool\_manager\_->UnpinPage(leaf\_page->GetPageId(), false);
82. return false;
83. } else {
84. *// if the key is found, get the value and unpin the page*
85. result.push\_back(leaf\_page->ValueAt(index));
86. buffer\_pool\_manager\_->UnpinPage(leaf\_page->GetPageId(), false);
87. return true;
88. }
89. }
90. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**
91. \* INSERTION
92. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
93. */\**
94. \* Insert constant key & value pair into b+ tree
95. \* if current tree is empty, start new tree, update root page id and insert
96. \* entry, otherwise insert into leaf page.
97. \* @return: since we only support unique key, if user try to insert duplicate
98. \* keys return false, otherwise return true.
99. \*/
100. bool BPlusTree::Insert(GenericKey \*key, const RowId &value, Txn \*transaction) {
101. if (IsEmpty()) {
102. StartNewTree(key, value);
103. return true;
104. }
105. return InsertIntoLeaf(key, value, transaction);
106. }
107. */\**
108. \* Insert constant key & value pair into an empty tree
109. \* User needs to first ask for new page from buffer pool manager(NOTICE: throw
110. \* an "out of memory" exception if returned value is nullptr), then update b+
111. \* tree's root page id and insert entry directly into leaf page.
112. \*/
113. void BPlusTree::StartNewTree(GenericKey \*key, const RowId &value) {
114. *// ask for new page from buffer pool manager*
115. page\_id\_t newid;
116. Page \*page = buffer\_pool\_manager\_->NewPage(newid);
117. root\_page\_id\_ = newid;
118. UpdateRootPageId(1);
119. *// get the data of the page*
120. auto \*leaf\_page = reinterpret\_cast<LeafPage \*>(page->GetData());
121. *// insert the key and value into the leaf page*
122. leaf\_page->Init(root\_page\_id\_, INVALID\_PAGE\_ID, processor\_.GetKeySize(), leaf\_max\_size\_);
123. leaf\_page->Insert(key, value, processor\_);
124. *// unpin the page*
125. buffer\_pool\_manager\_->UnpinPage(leaf\_page->GetPageId(), true);
126. }
127. */\**
128. \* Insert constant key & value pair into leaf page
129. \* User needs to first find the right leaf page as insertion target, then look
130. \* through leaf page to see whether insert key exist or not. If exist, return
131. \* immediately, otherwise insert entry. Remember to deal with split if necessary.
132. \* @return: since we only support unique key, if user try to insert duplicate
133. \* keys return false, otherwise return true.
134. \*/
135. bool BPlusTree::InsertIntoLeaf(GenericKey \*key, const RowId &value, Txn \*transaction) {
136. *// find the leaf page containing the key*
137. Page \*page = FindLeafPage(key);
138. *// get the data of the page*
139. auto \*leaf\_page = reinterpret\_cast<LeafPage \*>(page->GetData());
140. *// find the key in the leaf page*
141. RowId ri;
142. if (leaf\_page->Lookup(key,ri, processor\_)) {
143. *// if the key is found, unpin the page and return false*
144. buffer\_pool\_manager\_->UnpinPage(leaf\_page->GetPageId(), false);
145. return false;
146. } else {
147. *// if the key is not found, insert the key and value into the leaf page*
148. leaf\_page->Insert(key, value, processor\_);
149. *// if the leaf page is full, split the leaf page*
150. if (leaf\_page->GetSize() >= leaf\_page->GetMaxSize()) {
151. auto \*newtemp = Split(leaf\_page, transaction);
152. leaf\_page->SetNextPageId(newtemp->GetPageId());
153. InsertIntoParent(leaf\_page, newtemp->KeyAt(0), newtemp, transaction);
154. }
155. *// unpin the page and return true*
156. buffer\_pool\_manager\_->UnpinPage(leaf\_page->GetPageId(), true);
157. return true;
158. }
159. }
160. */\**
161. \* Split input page and return newly created page.
162. \* Using template N to represent either internal page or leaf page.
163. \* User needs to first ask for new page from buffer pool manager(NOTICE: throw
164. \* an "out of memory" exception if returned value is nullptr), then move half
165. \* of key & value pairs from input page to newly created page
166. \*/
167. BPlusTreeInternalPage \*BPlusTree::Split(InternalPage \*node, Txn \*transaction) {
168. *// ask for new page from buffer pool manager*
169. page\_id\_t newid;
170. Page \*page = buffer\_pool\_manager\_->NewPage(newid);
171. *// get the data of the page*
172. auto \*new\_page = reinterpret\_cast<InternalPage \*>(page->GetData());
173. *// move half of key & value pairs from input page to newly created page*
174. new\_page->Init(newid, node->GetParentPageId(), node->GetKeySize(), internal\_max\_size\_);
175. node->MoveHalfTo(new\_page, buffer\_pool\_manager\_);
176. *// unpin the page*
177. buffer\_pool\_manager\_->UnpinPage(newid, true);
178. return new\_page;
179. }
180. BPlusTreeLeafPage \*BPlusTree::Split(LeafPage \*node, Txn \*transaction) {
181. *// ask for new page from buffer pool manager*
182. page\_id\_t newid;
183. Page \*page = buffer\_pool\_manager\_->NewPage(newid);
184. *// get the data of the page*
185. auto \*new\_page = reinterpret\_cast<LeafPage \*>(page->GetData());
186. *// move half of key & value pairs from input page to newly created page*
187. new\_page->Init(newid, node->GetParentPageId(), node->GetKeySize(), leaf\_max\_size\_);
188. node->MoveHalfTo(new\_page);
189. new\_page->SetNextPageId(node->GetNextPageId());
190. *// unpin the page*
191. buffer\_pool\_manager\_->UnpinPage(newid, true);
192. return new\_page;
193. }
194. */\**
195. \* Insert key & value pair into internal page after split
196. \* @param   old\_node      input page from split() method
197. \* @param   key
198. \* @param   new\_node      returned page from split() method
199. \* User needs to first find the parent page of old\_node, parent node must be
200. \* adjusted to take info of new\_node into account. Remember to deal with split
201. \* recursively if necessary.
202. \*/
203. void BPlusTree::InsertIntoParent(BPlusTreePage \*old\_node, GenericKey \*key, BPlusTreePage \*new\_node, Txn \*transaction) {
204. if (old\_node->IsRootPage()) {
205. *// ask for new page from buffer pool manager*
206. auto \*new\_root = reinterpret\_cast<InternalPage \*>(buffer\_pool\_manager\_->NewPage(root\_page\_id\_)->GetData());
207. *// init*
208. new\_root->Init(root\_page\_id\_, INVALID\_PAGE\_ID, processor\_.GetKeySize(), internal\_max\_size\_);
209. *// populate*
210. new\_root->PopulateNewRoot(old\_node->GetPageId(), key, new\_node->GetPageId());
211. *// update parent*
212. old\_node->SetParentPageId(root\_page\_id\_);
213. new\_node->SetParentPageId(root\_page\_id\_);
214. *// unpin*
215. buffer\_pool\_manager\_->UnpinPage(root\_page\_id\_, true);
216. *// update root page id*
217. UpdateRootPageId(0);
218. }
219. else {
220. *// find the parent page of old\_node*
221. Page \*page = buffer\_pool\_manager\_->FetchPage(old\_node->GetParentPageId());
222. *// get the data of the page*
223. auto \*parent = reinterpret\_cast<InternalPage \*>(page->GetData());
224. *// insert key and value into the parent page*
225. int size = parent->InsertNodeAfter(old\_node->GetPageId(), key, new\_node->GetPageId());
226. *// if the parent page is full, split the parent page*
227. if (size >= internal\_max\_size\_) {
228. auto \*newtemp = Split(parent, transaction);
229. InsertIntoParent(parent, newtemp->KeyAt(0), newtemp, transaction);
230. }
231. *// unpin the page*
232. buffer\_pool\_manager\_->UnpinPage(parent->GetPageId(), true);
233. }
234. }
235. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**
236. \* REMOVE
237. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
238. */\**
239. \* Delete key & value pair associated with input key
240. \* If current tree is empty, return immediately.
241. \* If not, User needs to first find the right leaf page as deletion target, then
242. \* delete entry from leaf page. Remember to deal with redistribute or merge if
243. \* necessary.
244. \*/
245. void BPlusTree::Remove(const GenericKey \*key, Txn \*transaction) {
246. *// if the tree is empty, return immediately*
247. if (IsEmpty()) return;
248. *// find the leaf page containing the key*
249. Page \*page = FindLeafPage(key);
250. *// get the data of the page*
251. auto \*leaf\_page = reinterpret\_cast<LeafPage \*>(page->GetData());
252. int deleted\_index = leaf\_page->KeyIndex(key, processor\_);  *// find the key in the leaf page*
253. bool deleted = false;  *// if the key is found, delete the key and value from the leaf page*
254. int new\_size = leaf\_page->RemoveAndDeleteRecord(
255. key, processor\_);  *// if the leaf page is less than min size, coalesce or redistribute*
256. if (new\_size < leaf\_page->GetMinSize()) {
257. *// if the size of leaf page is less than min size, coalesce or redistribute*
258. deleted = CoalesceOrRedistribute<BPlusTree::LeafPage>(leaf\_page, transaction);
259. } else if (deleted\_index == 0) {
260. *// if the first key of leaf page is deleted, update the key in parent page*
261. GenericKey \*first\_key = leaf\_page->KeyAt(0);  *// get the first key of leaf page*
262. *// find the parent page of leaf page*
263. Page \*parent\_page = buffer\_pool\_manager\_->FetchPage(leaf\_page->GetParentPageId());
264. auto \*parent = reinterpret\_cast<InternalPage \*>(parent\_page);
265. *// get the page id of leaf page*
266. page\_id\_t leaf\_id = leaf\_page->GetPageId();
267. if (parent\_page != nullptr) {
268. while ((!parent->IsRootPage()) && parent->ValueAt(0) == leaf\_id) {
269. *// if the first key of leaf page is deleted, update the key in parent page*
270. leaf\_id = parent->GetPageId();
271. buffer\_pool\_manager\_->UnpinPage(parent->GetPageId(), false);
272. parent\_page = buffer\_pool\_manager\_->FetchPage(parent->GetParentPageId());
273. parent = reinterpret\_cast<InternalPage \*>(parent\_page);
274. }
275. if (parent->ValueAt(0) != leaf\_id &&
276. processor\_.CompareKeys(first\_key, parent->KeyAt(parent->ValueIndex(leaf\_id))) != 0) {
277. parent->SetKeyAt(parent->ValueIndex(leaf\_id), first\_key);
278. buffer\_pool\_manager\_->UnpinPage(parent->GetPageId(), true);
279. }
280. }
281. }
282. *// unpin the page*
283. buffer\_pool\_manager\_->UnpinPage(leaf\_page->GetPageId(), true);
284. if (deleted) {
285. buffer\_pool\_manager\_->DeletePage(page->GetPageId());
286. }
287. }
288. */\* todo*
289. \* User needs to first find the sibling of input page. If sibling's size + input
290. \* page's size > page's max size, then redistribute. Otherwise, merge.
291. \* Using template N to represent either internal page or leaf page.
292. \* @return: true means target leaf page should be deleted, false means no
293. \* deletion happens
294. \*/
295. template <typename N>
296. bool BPlusTree::CoalesceOrRedistribute(N \*&node, Txn \*transaction) {
297. bool result = false;
298. if (node->IsRootPage()) {
299. *// if the node is root page*
300. result = AdjustRoot(node);
301. return result;
302. } else if (node->GetSize() >= node->GetMinSize()) {
303. *// if the size of the node is greater than or equal to min size*
304. return false;
305. } else {
306. *// find the parent page of the node*
307. Page \*parent\_page = buffer\_pool\_manager\_->FetchPage(node->GetParentPageId());
308. *// get the data of the page*
309. auto \*parent = reinterpret\_cast<InternalPage \*>(parent\_page->GetData());
310. *// get the index of node*
311. int node\_index = parent->ValueIndex(node->GetPageId());
312. *// get the sibling index*
313. int sibling\_index;
314. if (node\_index == 0) {
315. *// if the node is the first child*
316. sibling\_index = 1;
317. } else {
318. *// if the node is not the first child*
319. sibling\_index = node\_index - 1;
320. }
321. *// get the sibling page*
322. Page \*sibling\_page = buffer\_pool\_manager\_->FetchPage(parent->ValueAt(sibling\_index));
323. *// get the data of the sibling page*
324. auto \*sibling = reinterpret\_cast<N \*>(sibling\_page->GetData());
325. *// if the size of the sibling page + the size of the node is greater than the max size*
326. if (sibling->GetSize() + node->GetSize() > node->GetMaxSize()) {
327. *// redistribute*
328. Redistribute(sibling, node, node\_index);
329. buffer\_pool\_manager\_->UnpinPage(sibling->GetPageId(), true);
330. buffer\_pool\_manager\_->UnpinPage(parent->GetPageId(), true);
331. return false;
332. } else {
333. *// coalesce*
334. result = Coalesce(sibling, node, parent, node\_index);
335. buffer\_pool\_manager\_->UnpinPage(sibling->GetPageId(), true);
336. buffer\_pool\_manager\_->UnpinPage(parent->GetPageId(), true);
337. return result;
338. }
339. }
340. }
341. */\**
342. \* Move all the key & value pairs from one page to its sibling page, and notify
343. \* buffer pool manager to delete this page. Parent page must be adjusted to
344. \* take info of deletion into account. Remember to deal with coalesce or
345. \* redistribute recursively if necessary.
346. \* Using template N to represent either internal page or leaf page.
347. \* @param   neighbor\_node      sibling page of input "node"
348. \* @param   node               input from method coalesceOrRedistribute()
349. \* @param   parent             parent page of input "node"
350. \* @return  true means parent node should be deleted, false means no deletion happened
351. \*/
352. bool BPlusTree::Coalesce(LeafPage \*&neighbor\_node, LeafPage \*&node, InternalPage \*&parent, int index,
353. Txn \*transaction) {
354. int sibling\_index = index - 1;
355. if (index == 0) {  *// if the node is the first child*
356. sibling\_index = 1;
357. }
358. if (index == 0) {
359. *// move all the key & value pairs from one page to its sibling page*
360. neighbor\_node->MoveAllTo(node);
361. node->SetNextPageId(neighbor\_node->GetPageId());
362. parent->Remove(sibling\_index);
363. } else {
364. *// move all the key & value pairs from one page to its sibling page*
365. node->MoveAllTo(neighbor\_node);
366. neighbor\_node->SetNextPageId(node->GetPageId());
367. parent->Remove(index);
368. }
369. return CoalesceOrRedistribute(parent, transaction);
370. }
371. bool BPlusTree::Coalesce(InternalPage \*&neighbor\_node, InternalPage \*&node, InternalPage \*&parent, int index,
372. Txn \*transaction) {
373. int sibling\_index = index - 1;
374. if (index == 0) {  *// if the node is the first child*
375. sibling\_index = 1;
376. }
377. if (index == 0) {
378. *// move all the key & value pairs from one page to its sibling page*
379. neighbor\_node->MoveAllTo(node, parent->KeyAt(sibling\_index), buffer\_pool\_manager\_);
380. parent->Remove(sibling\_index);
381. } else {
382. *// move all the key & value pairs from one page to its sibling page*
383. node->MoveAllTo(neighbor\_node, parent->KeyAt(index),buffer\_pool\_manager\_);
384. parent->Remove(index);
385. }
386. return CoalesceOrRedistribute(parent, transaction);
387. }
388. */\**
389. \* Redistribute key & value pairs from one page to its sibling page. If index ==
390. \* 0, move sibling page's first key & value pair into end of input "node",
391. \* otherwise move sibling page's last key & value pair into head of input
392. \* "node".
393. \* Using template N to represent either internal page or leaf page.
394. \* @param   neighbor\_node      sibling page of input "node"
395. \* @param   node               input from method coalesceOrRedistribute()
396. \*/
397. void BPlusTree::Redistribute(LeafPage \*neighbor\_node, LeafPage \*node, int index) {
398. *// get the parent page of the node*
399. Page \*parent\_page = buffer\_pool\_manager\_->FetchPage(node->GetParentPageId());
400. *// get the data of the page*
401. auto \*parent = reinterpret\_cast<InternalPage \*>(parent\_page->GetData());
402. *// if index == 0, move sibling page's first key & value pair into end of input "node"*
403. if (index == 0) {
404. neighbor\_node->MoveFirstToEndOf(node);
405. *// update parent key*
406. parent->SetKeyAt(1, node->KeyAt(0));
407. } else {
408. *// if index != 0, move sibling page's last key & value pair into head of input "node"*
409. neighbor\_node->MoveLastToFrontOf(node);
410. *// update parent key*
411. parent->SetKeyAt(index, neighbor\_node->KeyAt(0));
412. }
413. *// unpin the page*
414. buffer\_pool\_manager\_->UnpinPage(parent->GetPageId(), true);
415. }
416. void BPlusTree::Redistribute(InternalPage \*neighbor\_node, InternalPage \*node, int index) {
417. *// get the parent page of the node*
418. Page \*parent\_page = buffer\_pool\_manager\_->FetchPage(node->GetParentPageId());
419. *// get the data of the page*
420. auto \*parent = reinterpret\_cast<InternalPage \*>(parent\_page->GetData());
421. *// if index == 0, move sibling page's first key & value pair into end of input "node"*
422. if (index == 0) {
423. neighbor\_node->MoveFirstToEndOf(node, parent->KeyAt(1), buffer\_pool\_manager\_);
424. *// update parent key*
425. parent->SetKeyAt(1, node->KeyAt(0));
426. } else {
427. *// if index != 0, move sibling page's last key & value pair into head of input "node"*
428. neighbor\_node->MoveLastToFrontOf(node, parent->KeyAt(index), buffer\_pool\_manager\_);
429. *// update parent key*
430. parent->SetKeyAt(index, neighbor\_node->KeyAt(0));
431. }
432. }
433. */\**
434. \* Update root page if necessary
435. \* NOTE: size of root page can be less than min size and this method is only
436. \* called within coalesceOrRedistribute() method
437. \* case 1: when you delete the last element in root page, but root page still
438. \* has one last child
439. \* case 2: when you delete the last element in whole b+ tree
440. \* @return : true means root page should be deleted, false means no deletion
441. \* happened
442. \*/
443. bool BPlusTree::AdjustRoot(BPlusTreePage \*old\_root\_node) {
444. *// 2 cases*
445. bool result;
446. if (old\_root\_node->GetSize() > 1) {
447. result = false;
448. } else if (old\_root\_node->IsLeafPage()) {
449. if (old\_root\_node->GetSize() == 1) {
450. *// if the size of root page is 1, no need to delete*
451. result = false;
452. } else {
453. root\_page\_id\_ = INVALID\_PAGE\_ID;
454. UpdateRootPageId(0);
455. result = true;
456. }
457. } else {
458. auto \*root = reinterpret\_cast<InternalPage \*>(old\_root\_node);
459. root\_page\_id\_ = root->RemoveAndReturnOnlyChild();
460. auto \*new\_root = reinterpret\_cast<InternalPage \*>(buffer\_pool\_manager\_->FetchPage(root\_page\_id\_)->GetData());
461. *// update root page id*
462. new\_root->SetParentPageId(INVALID\_PAGE\_ID);
463. buffer\_pool\_manager\_->UnpinPage(root\_page\_id\_, true);
464. UpdateRootPageId(0);
465. result = true;
466. }
467. return result;
468. }
469. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**
470. \* INDEX ITERATOR
471. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
472. */\**
473. \* Input parameter is void, find the left most leaf page first, then construct
474. \* index iterator
475. \* @return : index iterator
476. \*/
477. IndexIterator BPlusTree::Begin() {
478. *// find leaf page*
479. Page \*page = FindLeafPage(nullptr, INVALID\_PAGE\_ID, true);
480. auto \*leaf\_page = reinterpret\_cast<LeafPage \*>(page->GetData());
481. page\_id\_t page\_id = leaf\_page->GetPageId();
482. *// unpin the page*
483. buffer\_pool\_manager\_->UnpinPage(page\_id, false);
484. return IndexIterator(page\_id, buffer\_pool\_manager\_, 0);
485. }
486. */\**
487. \* Input parameter is low key, find the leaf page that contains the input key
488. \* first, then construct index iterator
489. \* @return : index iterator
490. \*/
491. IndexIterator BPlusTree::Begin(const GenericKey \*key) {
492. Page \*page = FindLeafPage(key, INVALID\_PAGE\_ID, false);
493. auto \*leaf\_page = reinterpret\_cast<LeafPage \*>(page->GetData());
494. page\_id\_t page\_id = leaf\_page->GetPageId();
495. *// unpin the page*
496. buffer\_pool\_manager\_->UnpinPage(page\_id, false);
497. return IndexIterator(page\_id, buffer\_pool\_manager\_, leaf\_page->KeyIndex(key, processor\_));
498. }
499. */\**
500. \* Input parameter is void, construct an index iterator representing the end
501. \* of the key/value pair in the leaf node
502. \* @return : index iterator
503. \*/
504. IndexIterator BPlusTree::End() {
505. auto \*page = reinterpret\_cast<BPlusTreePage \*>(buffer\_pool\_manager\_->FetchPage(root\_page\_id\_)->GetData());
506. while (!page->IsLeafPage()) {
507. *// if the page is internal page, get the last child*
508. auto \*internal\_page = reinterpret\_cast<InternalPage \*>(page);  *// get the last child*
509. buffer\_pool\_manager\_->UnpinPage(page->GetPageId(), false);     *// unpin the page*
510. page = reinterpret\_cast<BPlusTreePage \*>(
511. buffer\_pool\_manager\_->FetchPage(internal\_page->ValueAt(internal\_page->GetSize() - 1))->GetData());
512. *// get the data of the page*
513. }
514. buffer\_pool\_manager\_->UnpinPage(page->GetPageId(), false);
515. return IndexIterator(page->GetPageId(), buffer\_pool\_manager\_, page->GetSize() - 1);
516. }
517. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**
518. \* UTILITIES AND DEBUG
519. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
520. */\**
521. \* Find leaf page containing particular key, if leftMost flag == true, find
522. \* the left most leaf page
523. \* Note: the leaf page is pinned, you need to unpin it after use.
524. \*/
525. Page \*BPlusTree::FindLeafPage(const GenericKey \*key, page\_id\_t page\_id, bool leftMost) {
526. if (page\_id == INVALID\_PAGE\_ID) {
527. page\_id = root\_page\_id\_;
528. }
529. *// find the leaf page containing the key*
530. auto \*page = reinterpret\_cast<BPlusTreePage \*>(buffer\_pool\_manager\_->FetchPage(page\_id)->GetData());
531. while (!page->IsLeafPage()) {
532. *// if the page is internal page, get the next page*
533. auto \*internal\_page = reinterpret\_cast<InternalPage \*>(page);
534. page\_id\_t next\_page\_id;
535. if (leftMost) {
536. next\_page\_id = internal\_page->ValueAt(0);
537. } else {
538. next\_page\_id = internal\_page->Lookup(key, processor\_);
539. }
540. buffer\_pool\_manager\_->UnpinPage(page->GetPageId(), false);
541. page = reinterpret\_cast<BPlusTreePage \*>(buffer\_pool\_manager\_->FetchPage(next\_page\_id)->GetData());
542. }
543. return reinterpret\_cast<Page \*>(page);
544. }
545. */\**
546. \* Update/Insert root page id in header page(where page\_id = 0, header\_page is
547. \* defined under include/page/header\_page.h)
548. \* Call this method everytime root page id is changed.
549. \* @parameter: insert\_record      default value is false. When set to true,
550. \* insert a record <index\_name, current\_page\_id> into header page instead of
551. \* updating it.
552. \*/
553. void BPlusTree::UpdateRootPageId(int insert\_record) {
554. auto \*root\_page = reinterpret\_cast<IndexRootsPage \*>(buffer\_pool\_manager\_->FetchPage(INDEX\_ROOTS\_PAGE\_ID)->GetData());
555. if (insert\_record == 0) {
556. root\_page->Update(index\_id\_, root\_page\_id\_);
557. } else {
558. root\_page->Insert(index\_id\_, root\_page\_id\_);
559. }
560. buffer\_pool\_manager\_->UnpinPage(INDEX\_ROOTS\_PAGE\_ID, true);
561. }

构造函数：

成员初始化：初始化B+树的成员变量，包括索引ID index\_id\_、缓冲池管理器 buffer\_pool\_manager\_、关键字处理器 processor\_、叶子页最大大小 leaf\_max\_size\_、内部页最大大小 internal\_max\_size\_。

页加载和根页获取：

通过 buffer\_pool\_manager\_->FetchPage(INDEX\_ROOTS\_PAGE\_ID) 获取包含所有索引根页ID的页面。

如果 leaf\_max\_size 或 internal\_max\_size 未定义（即 UNDEFINED\_SIZE），则根据关键字大小和页结构计算它们的合理值。

从页面中获取该索引的根页ID，并存储在 root\_page\_id\_ 中。

如果未找到根页ID，将 root\_page\_id\_ 设置为 INVALID\_PAGE\_ID，表示需要创建一个新的根页。

页面管理：

最后，解除对根页ID所在页面的固定，并且如果 root\_page\_id\_ 不为无效页ID，则也解除对根页本身的固定，确保页面不再被缓冲池管理器保留

析构函数：

空树检查：如果树为空（通过 IsEmpty() 判断），则直接返回，无需继续销毁操作。

页加载和类型判断：

通过 buffer\_pool\_manager\_->FetchPage(current\_page\_id) 获取当前页的指针。

将页面数据转换为 BPlusTreePage\* 类型，这是B+树页的基类。

叶子页处理：

如果当前页是叶子页（通过 IsLeafPage() 判断），则直接删除该页并解除固定。此处假设 buffer\_pool\_manager\_->DeletePage() 负责删除页的物理存储。

内部页处理：

如果当前页是内部页，则将其转换为 InternalPage\* 类型，并递归地对每个子页面调用 Destroy() 函数，以便逐层删除。

页解固和删除：

最后，再次调用 buffer\_pool\_manager\_->DeletePage() 删除当前页，并调用 buffer\_pool\_manager\_->UnpinPage() 解除固定。

IsEmpty() 函数： 通过检查根页ID是否为无效页ID来确定树是否为空。如果根页ID为无效页ID，则树为空，否则不为空。

GetValue() 函数： 首先调用 IsEmpty() 函数检查树是否为空。然后使用 FindLeafPage() 方法找到包含指定键的叶子页。将叶子页数据转换为 LeafPage 类型，并调用 KeyIndex() 方法查找键的索引。如果找到了键，则将对应的值存入结果向量中，并释放叶子页。如果未找到键，则释放叶子页并返回 false。

Insert() 函数： 首先调用 IsEmpty() 函数检查树是否为空。如果树为空，调用 StartNewTree() 函数创建新树并插入第一个键值对。如果树不为空，则调用 InsertIntoLeaf() 函数将新键值对插入到合适的叶子页中。

StartNewTree() 函数： 通过 buffer\_pool\_manager\_->NewPage(newid) 请求分配新页面。更新根页ID为新页面的ID，并初始化新页面为叶子页。将第一个键值对插入到叶子页中，并将页面标记为脏页后释放。

插入操作函数分析

这个函数负责将新的键值对插入到B+树的叶子节点中：

首先通过查找找到包含指定键的叶子页面。

如果键已存在于叶子页面中，则返回插入失败。

如果键不存在，将键值对插入到叶子页面。

如果叶子页面满了，触发页面分裂操作。

最后将页面解除固定并返回插入成功。

分裂函数分析

这个函数根据节点类型（叶子节点或内部节点）实现节点的分裂：

请求新页面并将其初始化为相应节点类型。

将当前节点的一半键值对移动到新页面。

针对叶子节点，还需设置新页面的下一页ID。

最后将新页面解除固定并返回新页面对象。

InsertIntoParent 函数分析

这个函数用于将新的节点插入到父节点中，如果需要，还会处理根节点的更新和父节点的分裂。

根节点处理：

首先检查 old\_node 是否是根节点 (IsRootPage() 方法)。

如果是根节点，通过缓冲池管理器请求一个新的页面，并将其解释为 InternalPage 类型。

初始化新根节点 (new\_root)，设置其属性并将原根节点和新节点作为子节点插入。

更新原节点和新节点的父节点指针，将根节点的页面解除固定，并更新根页面ID。

非根节点处理：

如果 old\_node 不是根节点，从缓冲池管理器中获取其父节点页面。

将页面数据解释为 InternalPage 类型，并在父节点中插入新的键和节点。

如果父节点满了 (size >= internal\_max\_size\_)，则调用 Split 函数对父节点进行分裂，并递归调用 InsertIntoParent 函数来处理可能的连锁影响。

最后，解除父节点页面的固定。

Remove 函数分析

这个函数用于从B+树中删除指定的键值对，并处理可能的页面合并或重新分配。

空树检查：

首先检查树是否为空，如果是，则立即返回。

找到包含键的叶子页面：

使用 FindLeafPage 函数找到包含指定键的叶子页面。

将页面数据解释为 LeafPage 类型。

删除键处理：

调用 KeyIndex 方法在叶子页面中查找要删除的键的索引。

调用 RemoveAndDeleteRecord 方法删除键和相关的值，返回更新后的叶子页面大小。

页面大小检查：

如果叶子页面的大小小于最小值 (new\_size < leaf\_page->GetMinSize())，则可能需要进行页面合并或重新分配。

调用 CoalesceOrRedistribute 函数来执行合并或重新分配操作。

更新父节点键：

如果删除了叶子页面中的第一个键 (deleted\_index == 0)，则需要更新父节点中对应的键值。

获取叶子页面的第一个键，并找到其父节点。

如果父节点存在且需要更新，则更新父节点中对应位置的键值。

页面解除固定和删除：

解除叶子页面的固定状态，并根据需要删除页面。

CoalesceOrRedistribute 函数

这个函数根据节点当前的大小决定是进行合并还是重新分配操作。

如果节点是根节点 (IsRootPage())，则调用 AdjustRoot 函数来调整根节点的状态。

如果节点的大小大于等于最小大小 (GetSize() >= GetMinSize())，则返回 false，表示无需进一步操作。

否则，需要找到节点的父节点，并获取其兄弟节点来进行合并或重新分配。

在执行合并或重新分配操作之后，函数会根据操作的结果（是否成功合并或重新分配）返回相应的布尔值。

Coalesce 函数（针对叶子节点）

这个函数用于将两个相邻的叶子节点合并为一个节点。

首先确定需要合并的兄弟节点的索引。

如果当前节点是第一个子节点，将所有键值对从当前节点移动到兄弟节点，并设置当前节点的下一个页面ID为兄弟节点的ID。

否则，将所有键值对从兄弟节点移动到当前节点，并设置兄弟节点的下一个页面ID为当前节点的ID。

最后，从父节点中移除相应的兄弟节点索引，并调用 CoalesceOrRedistribute 函数来处理可能的父节点合并或重新分配。

Coalesce 函数（针对内部节点）

这个函数用于将两个相邻的内部节点合并为一个节点。

同样地，首先确定需要合并的兄弟节点的索引。

如果当前节点是第一个子节点，将所有键值对从当前节点移动到兄弟节点，并同时移动父节点中相应位置的键。

否则，将所有键值对从兄弟节点移动到当前节点，并移动父节点中相应位置的键。

最后，从父节点中移除相应的兄弟节点索引，并调用 CoalesceOrRedistribute 函数来处理可能的父节点合并或重新分配。

Redistribute 函数分析

Redistribute 函数用于在B+树的节点中进行重新分配操作，以保持节点大小的平衡。

LeafPage版本的 Redistribute 函数：

首先，通过 node->GetParentPageId() 获取节点的父页面ID，并从缓冲池中获取父页面 (parent\_page)。

将父页面的数据解释为 InternalPage 类型，以便后续操作。

如果 index == 0，表示当前节点是第一个子节点：

调用 neighbor\_node->MoveFirstToEndOf(node)，将兄弟节点的第一个键值对移动到当前节点的末尾。

更新父节点的键值对应位置的键值，使用 parent->SetKeyAt(1, node->KeyAt(0))。

否则，如果 index != 0：

调用 neighbor\_node->MoveLastToFrontOf(node)，将兄弟节点的最后一个键值对移动到当前节点的开头。

更新父节点相应位置的键值，使用 parent->SetKeyAt(index, neighbor\_node->KeyAt(0))。

最后，通过 buffer\_pool\_manager\_->UnpinPage(parent->GetPageId(), true) 将父页面从缓冲池中解锁。

InternalPage 版本的 Redistribute 函数：

类似于 LeafPage 版本，首先获取父页面，并将其数据解释为 InternalPage。

如果 index == 0，表示当前节点是第一个子节点：

调用 neighbor\_node->MoveFirstToEndOf(node, parent->KeyAt(1), buffer\_pool\_manager\_)，将兄弟节点的第一个键值对及其子节点移动到当前节点的末尾。

更新父节点的第一个键值，使用 parent->SetKeyAt(1, node->KeyAt(0))。

否则，如果 index != 0：

调用 neighbor\_node->MoveLastToFrontOf(node, parent->KeyAt(index), buffer\_pool\_manager\_)，将兄弟节点的最后一个键值对及其子节点移动到当前节点的开头。

更新父节点相应位置的键值，使用 parent->SetKeyAt(index, neighbor\_node->KeyAt(0))。

最后，通过 buffer\_pool\_manager\_->UnpinPage(parent->GetPageId(), true) 将父页面从缓冲池中解锁。

AdjustRoot 函数分析

AdjustRoot 函数用于调整B+树的根节点，确保树的正确性和结构。

如果 old\_root\_node 的大小大于1，则根节点不需要调整，直接返回 false。

如果 old\_root\_node 是叶子节点，并且大小为1，说明树中只有一个叶子节点：

返回 false，因为不需要删除根节点。

如果 old\_root\_node 是内部节点，并且大小为1，表示树需要调整：

将根节点的子节点作为新的根节点，并更新相关的树结构信息。

最后，通过 buffer\_pool\_manager\_->FetchPage(root\_page\_id\_) 获取新的根节点页面，解释为 InternalPage 类型，并更新其父页面ID。

将旧的根节点从缓冲池中解锁，并更新根节点ID为无效值。

Begin 函数分析

Begin 函数用于获取B+树中的第一个键的迭代器。

Begin() 函数版本：

调用 FindLeafPage(nullptr, INVALID\_PAGE\_ID, true) 获取树中的第一个叶子页面。

将叶子页面的数据解释为 LeafPage 类型，并获取其页面ID。

通过 buffer\_pool\_manager\_->UnpinPage(page\_id, false) 将叶子页面从缓冲池中解锁。

返回一个 IndexIterator 对象，表示从第一个键开始的迭代器。

Begin(const GenericKey \*key) 函数版本：

调用 FindLeafPage(key, INVALID\_PAGE\_ID, false) 查找包含指定键的叶子页面。

将叶子页面的数据解释为 LeafPage 类型，并获取其页面ID。

通过 buffer\_pool\_manager\_->UnpinPage(page\_id, false) 将叶子页面从缓冲池中解锁。

调用 leaf\_page->KeyIndex(key, processor\_) 获取指定键在叶子页面中的索引位置。

返回一个 IndexIterator 对象，表示从指定键开始的迭代器。

End函数：

返回B+树中最大的键的迭代器。

首先从根节点开始，沿着最右侧的路径找到最后一个叶子页。

获取该叶子页的页号和大小，然后解除固定，返回一个IndexIterator对象，指向该叶子页中最后一个键的位置。

FindLeafPage函数：

根据键值查找包含该键值的叶子页。

如果未指定页号，则从根节点开始查找。

如果当前页不是叶子页，则根据leftMost参数决定获取下一个页的方式（是取第一个子页还是根据键查找）。

最终返回包含该键值的叶子页的指针。

UpdateRootPageId函数：

更新根节点的页号。

首先获取索引根节点页，并根据insert\_record参数决定是更新还是插入该索引的根节点页号。

更新完成后解除固定索引根节点页。

三、测试结果

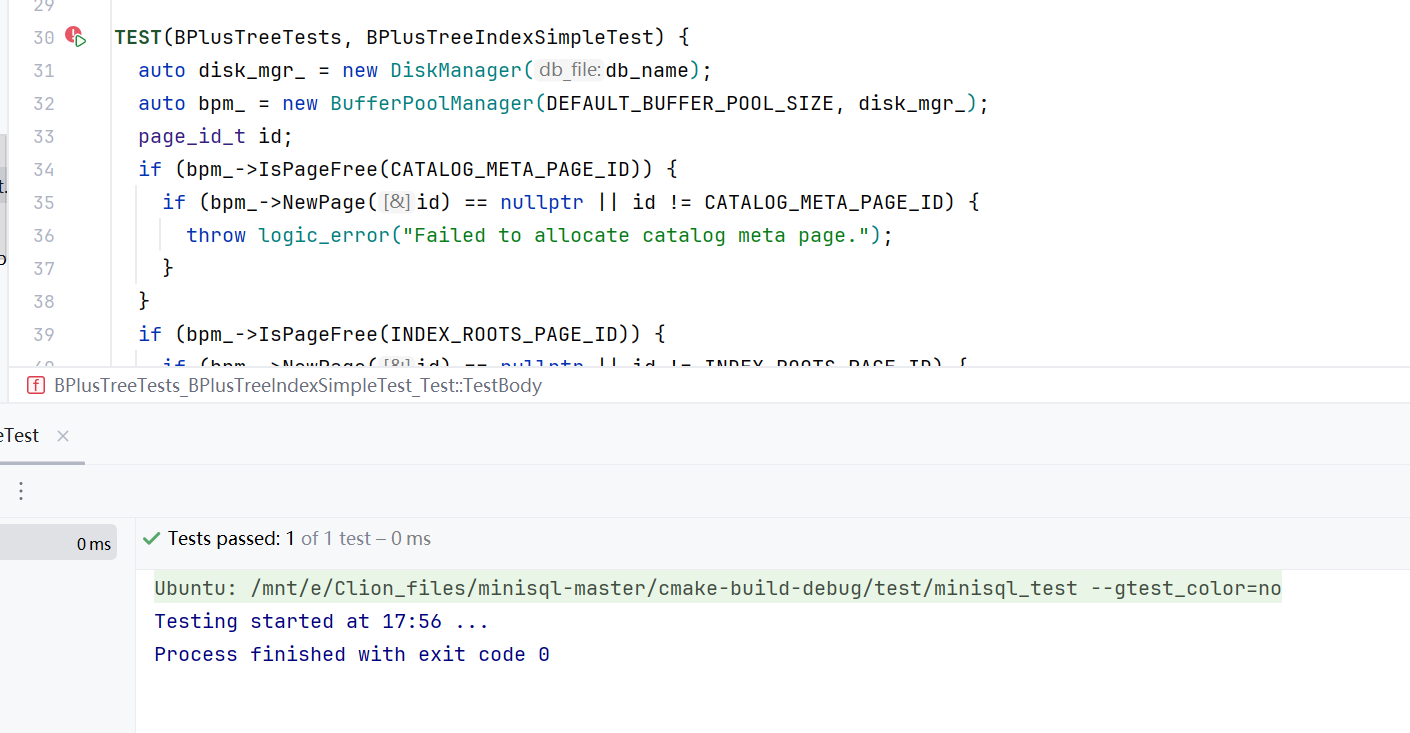
1、index\_root\_page\_test



可以看到测试样例可以顺利通过。

2、b+tree\_index\_test





可以看到只能通过其中的一个测试样例。

3、b+\_tree\_test



可以看到由于先前b+tree\_index\_test模块存在问题，这里的测试样例也会受到影响。

# #4 CATALOG MANAGER

负责人：汪珉凯

一、内容与功能

1、目录元信息

CatalogMeta::GetSerializedSize():

这个函数的作用是计算CatalogMeta对象在序列化为字节流时所需的大小。

CatalogMeta包含了数据库中所有表和索引的元信息，它的序列化大小由以下组成：

一个或多个数据页的信息（每个数据页包含多个表或索引的元信息）。

每个表和索引的元信息，包括表名、列信息、索引名、索引键信息等。

在实现时，需要考虑到每个数据页的大小和其中包含的表和索引数量，以及每个表和索引元信息的具体内容大小。

IndexMetadata::GetSerializedSize():

IndexMetadata表示一个索引的元信息，包括该索引对应的表信息、索引的键信息等。

GetSerializedSize()函数的任务是计算该索引元信息在序列化为字节流时所占的字节数。

这包括索引的名称、键的数量、每个键的类型和长度等信息。

TableMetadata::GetSerializedSize():

TableMetadata表示一个表的元信息，包括表的名称、列信息、索引信息等。

GetSerializedSize()函数的目标是计算表元信息在序列化为字节流时所需的字节数。

这包括表的名称、列数、每列的类型、长度等信息。

IndexInfo::Init(\*index\_meta\_data, \*table\_info, \*buffer\_pool\_manager):

这个函数的作用是初始化一个索引对象，以便将其存储到数据库中。

参数包括预先创建的IndexMetadata对象和从CatalogManager获取的TableInfo对象。

函数根据这些信息创建索引的key\_schema和Index对象。

key\_schema可以通过ShallowCopySchema方法创建，确保它与表的Schema共享存储，以减少内存使用和复制成本。

Index对象的创建可能涉及到内存管理器（buffer\_pool\_manager），以便有效地管理索引数据的存储和检索。

2、表和索引的管理

CatalogManager类负责数据库系统中表和索引的元数据管理。以下是对CatalogManager类的功能描述：

持久化元数据：CatalogManager类需要实现表和索引的元数据的持久化。这意味着当数据库关闭后，这些信息需要被保存下来，并且在数据库重新打开时能够被重新加载。

初始化元数据：在数据库实例首次创建时，CatalogManager需要初始化元数据。这通常涉及到设置默认值或加载预定义的元数据结构。

加载元数据：当数据库实例重新打开时，CatalogManager需要从数据库文件中加载所有的表和索引信息，构建内存中的TableInfo和IndexInfo对象。

提供操作接口：CatalogManager类需要提供一系列的接口供上层模块使用，包括：

CreateTable：创建一个新的数据表。

GetTable：根据表名获取一个表的元数据信息。

GetTables：获取数据库中所有表的列表。

DropTable：删除一个指定的表。

GetTableIndexes：获取指定表的所有索引信息。

CreateIndex：为表创建一个新的索引。

GetIndex：根据索引名获取索引的元数据信息。

DropIndex：删除指定的索引。

内存管理：在实现CatalogManager时，需要注意内存管理问题。例如，Schema类中可能存在深拷贝和浅拷贝的函数，需要在合适的时机使用它们以避免内存泄漏或二次析构的问题。

架构依赖性：CatalogManager在架构设计中具有高度的内聚性，因为它依赖于Disk Manager和Buffer Pool Manager，同时也为Record Manager、Index Manager、Executor甚至Parser提供接口支持。

二、代码实现与分析

1. */\*\**
2. \* TODO: Student Implement
3. \*/
4. *// 返回目录元数据的序列化大小*
5. uint32\_t CatalogMeta::GetSerializedSize() const {
6. *// 计算序列化大小：魔术数大小 + 索引元数据页数组大小 + 表元数据页数组大小*
7. *// 索引元数据页数组大小包括每个索引的索引ID和页号*
8. *// 表元数据页数组大小包括每个表的索引ID和页号*
9. return sizeof(CATALOG\_METADATA\_MAGIC\_NUM)                             *// 魔术数大小*
10. + sizeof(decltype(index\_meta\_pages\_.size()))                    *// 索引元数据页数组大小*
11. + sizeof(decltype(table\_meta\_pages\_.size()))                    *// 表元数据页数组大小*
12. + index\_meta\_pages\_.size() \* (sizeof(index\_id\_t) + sizeof(page\_id\_t))   *// 索引元数据页数组大小乘以每个元素的大小（索引ID和页号）的总和*
13. + table\_meta\_pages\_.size() \*
14. (sizeof(index\_id\_t) + sizeof(page\_id\_t));  *// 表元数据页数组大小乘以每个元素的大小（索引ID和页号）的总和*
15. }
16. CatalogMeta::CatalogMeta() {}
17. */\*\**
18. \* TODO: Student Implement
19. \*/
20. CatalogManager::CatalogManager(BufferPoolManager \*buffer\_pool\_manager, LockManager \*lock\_manager,
21. LogManager \*log\_manager, bool init)
22. : buffer\_pool\_manager\_(buffer\_pool\_manager), lock\_manager\_(lock\_manager), log\_manager\_(log\_manager) {
23. table\_names\_.clear();
24. tables\_.clear();
25. indexes\_.clear();
26. index\_names\_.clear();
27. if (init) {
28. *// init catalog meta*
29. catalog\_meta\_ = CatalogMeta::NewInstance();
30. catalog\_meta\_->index\_meta\_pages\_.clear();
31. catalog\_meta\_->table\_meta\_pages\_.clear();
32. catalog\_meta\_->SerializeTo(buffer\_pool\_manager\_->FetchPage(CATALOG\_META\_PAGE\_ID)->GetData());
33. } else {
34. *// load catalog meta*
35. char \*buf = buffer\_pool\_manager\_->FetchPage(CATALOG\_META\_PAGE\_ID)->GetData();
36. catalog\_meta\_ = CatalogMeta::DeserializeFrom(buf);
37. *// load tables*
38. for (auto iter : catalog\_meta\_->table\_meta\_pages\_) {
39. LoadTable(iter.first, iter.second);
40. }
41. *// load indexes*
42. for (auto iter : catalog\_meta\_->index\_meta\_pages\_) {
43. LoadIndex(iter.first, iter.second);
44. }
45. }
46. *//unpin catalog meta page*
47. buffer\_pool\_manager\_->UnpinPage(CATALOG\_META\_PAGE\_ID, true);
48. }
49. CatalogManager::~CatalogManager() {
50. FlushCatalogMetaPage();
51. delete catalog\_meta\_;
52. for (auto iter : tables\_) {
53. delete iter.second;
54. }
55. for (auto iter : indexes\_) {
56. delete iter.second;
57. }
58. }
59. */\*\**
60. \* TODO: Student Implement
61. \*/
62. dberr\_t CatalogManager::CreateTable(const string &table\_name, TableSchema \*schema,
63. Txn \*txn, TableInfo \*&table\_info) {
64. auto table\_schema = TableSchema::DeepCopySchema(schema);
65. if (table\_names\_.find(table\_name) == table\_names\_.end()) {
66. *// THIS TABLE DOES NOT EXIST*
67. *//create a new table id*
68. table\_id\_t table\_id=next\_table\_id\_;
69. table\_names\_[table\_name]=table\_id;
70. next\_table\_id\_++;
71. *//allocate a new page for table metadata*
72. page\_id\_t table\_page\_id;
73. auto\* table\_page = buffer\_pool\_manager\_->NewPage(table\_page\_id);
74. *//associate the table metadata page with the table id*
75. catalog\_meta\_->table\_meta\_pages\_[table\_id]=table\_page\_id;
76. *//create a new table heap*
77. auto\* table\_heap = TableHeap::Create(buffer\_pool\_manager\_, table\_schema,txn, log\_manager\_, lock\_manager\_);
78. *//create a new table metadata*
79. auto\* table\_metadata = TableMetadata::Create(table\_id, table\_name, table\_heap->GetFirstPageId(), table\_schema);
80. table\_metadata->SerializeTo(table\_page->GetData());
81. *//unpin*
82. buffer\_pool\_manager\_->UnpinPage(table\_page\_id, true);
83. *//create a new table info*
84. table\_info = TableInfo::Create();
85. table\_info->Init(table\_metadata, table\_heap);
86. *//put the table into the tables\_ map*
87. tables\_[table\_id]=table\_info;
88. return DB\_SUCCESS;
89. }
90. return DB\_TABLE\_ALREADY\_EXIST;
91. }
92. */\*\**
93. \* TODO: Student Implement
94. \*/
95. dberr\_t CatalogManager::GetTable(const string &table\_name, TableInfo \*&table\_info) {
96. if(table\_names\_.find(table\_name)==table\_names\_.end()){
97. return DB\_TABLE\_NOT\_EXIST;
98. }
99. table\_info = tables\_[table\_names\_[table\_name]];
100. return DB\_SUCCESS;
101. }
102. */\*\**
103. \* TODO: Student Implement
104. \*/
105. dberr\_t CatalogManager::GetTables(vector<TableInfo \*> &tables) const {
106. for(auto i:tables\_) {
107. tables.push\_back(i.second);
108. }
109. return DB\_SUCCESS;
110. }
111. */\*\**
112. \* TODO: Student Implement
113. \*/
114. dberr\_t CatalogManager::CreateIndex(const std::string &table\_name, const string &index\_name,
115. const std::vector<std::string> &index\_keys, Txn \*txn,
116. IndexInfo \*&index\_info, const string &index\_type) {
117. try {
118. *// Check if the table exists*
119. auto iter\_find\_table = table\_names\_.find(table\_name);
120. if (iter\_find\_table == table\_names\_.end()) {
121. return DB\_TABLE\_NOT\_EXIST;
122. }
123. *// Check if the index already exists for the table*
124. auto iter\_find\_index\_table = index\_names\_.find(table\_name);
125. if (iter\_find\_index\_table != index\_names\_.end()) {
126. auto iter\_find\_index\_name = iter\_find\_index\_table->second.find(index\_name);
127. if (iter\_find\_index\_name != iter\_find\_index\_table->second.end()) {
128. return DB\_INDEX\_ALREADY\_EXIST;
129. }
130. }
131. *// Initialize variables*
132. table\_id\_t table\_id = 0; *// Initialize table ID*
133. TableSchema\* schema = nullptr; *// Initialize table schema pointer*
134. TableInfo\* table\_info = nullptr; *// Initialize table info pointer*
135. *// Index variables*
136. page\_id\_t meta\_page\_id = 0; *// Initialize meta page ID*
137. Page\* meta\_page = nullptr; *// Initialize meta page pointer*
138. index\_id\_t index\_id = 0; *// Initialize index ID*
139. IndexMetadata\* index\_meta = nullptr; *// Initialize index metadata pointer*
140. *// Index key map*
141. std::vector<std::uint32\_t> key\_map{}; *// Initialize index key map*
142. *// Initialize index info using the Create method of index\_info object*
143. index\_info = index\_info->Create();
144. *// Get the next available index ID from catalog metadata*
145. index\_id = catalog\_meta\_->GetNextIndexId();
146. *// Retrieve table ID and schema from the respective containers*
147. table\_id = table\_names\_[table\_name];
148. table\_info = tables\_[table\_id];
149. schema = table\_info->GetSchema();
150. *// Create key map for the columns specified in index\_keys vector*
151. uint32\_t column\_index = 0; *// Initialize column index*
152. for (auto column\_name : index\_keys) {
153. *// Check if the column exists in the table schema*
154. if (schema->GetColumnIndex(column\_name, column\_index) == DB\_COLUMN\_NAME\_NOT\_EXIST) {
155. return DB\_COLUMN\_NAME\_NOT\_EXIST; *// Return error if column does not exist*
156. }
157. key\_map.push\_back(column\_index); *// Add column index to key map vector*
158. }
159. *// Allocate a new page in the buffer pool for storing index metadata*
160. meta\_page = buffer\_pool\_manager\_->NewPage(meta\_page\_id);
161. *// Create index metadata object using the Create method of index\_meta object*
162. index\_meta = IndexMetadata::Create(index\_id, index\_name, table\_id, key\_map);
163. *// Serialize index metadata and store it in the newly allocated meta page*
164. index\_meta->SerializeTo(meta\_page->GetData());
165. *// Initialize index info object with the retrieved index metadata,*
166. *// table info and buffer pool manager*
167. index\_info->Init(index\_meta, table\_info, buffer\_pool\_manager\_);
168. *// Update index information in the index\_names map for the specified table*
169. index\_names\_[table\_name][index\_name] = index\_id;
170. *// Store index info object in the indexes map using the retrieved index ID*
171. indexes\_[index\_id] = index\_info;
172. *// Update the catalog metadata with the page ID of the newly allocated meta page*
173. catalog\_meta\_->index\_meta\_pages\_[index\_id] = meta\_page\_id;
174. *// Fetch the page containing catalog metadata from the buffer pool*
175. Page\* catalog\_meta\_page = buffer\_pool\_manager\_->FetchPage(CATALOG\_META\_PAGE\_ID);
176. *// Retrieve the buffer containing catalog metadata from the fetched page*
177. char\* buf = catalog\_meta\_page->GetData();
178. *// Serialize the catalog metadata and update the buffer containing catalog metadata*
179. catalog\_meta\_->SerializeTo(buf);
180. *// Mark the catalog metadata page in the buffer pool as unpinned and modified*
181. buffer\_pool\_manager\_->UnpinPage(CATALOG\_META\_PAGE\_ID, true);
182. *// Return success status indicating successful creation of index*
183. return DB\_SUCCESS;
184. } catch (exception e) {
185. *// Return failure status indicating failure during index creation*
186. return DB\_FAILED;
187. }
188. }
189. */\*\**
190. \* TODO: Student Implement
191. \*/
192. dberr\_t CatalogManager::GetIndex(const std::string &table\_name, const std::string &index\_name,
193. IndexInfo \*&index\_info) const {
194. if (index\_names\_.find(table\_name) == index\_names\_.end()) {
195. return DB\_TABLE\_NOT\_EXIST;
196. }
197. if (index\_names\_.find(table\_name)->second.find(index\_name) == index\_names\_.find(table\_name)->second.end()) {
198. return DB\_INDEX\_NOT\_FOUND;
199. }
200. index\_info = indexes\_.find(index\_names\_.find(table\_name)->second.find(index\_name)->second)->second;
201. return DB\_SUCCESS;
202. }
203. */\*\**
204. \* TODO: Student Implement
205. \*/
206. dberr\_t CatalogManager::GetTableIndexes(const std::string &table\_name, std::vector<IndexInfo \*> &indexes) const {
207. if (index\_names\_.find(table\_name) == index\_names\_.end()) {
208. return DB\_TABLE\_NOT\_EXIST;
209. }
210. for (auto i : index\_names\_.find(table\_name)->second) {
211. indexes.push\_back(indexes\_.find(i.second)->second);
212. }
213. return DB\_SUCCESS;
214. }
215. */\*\**
216. \* TODO: Student Implement
217. \*/
218. dberr\_t CatalogManager::DropTable(const string &table\_name) {
219. if(table\_names\_.find(table\_name)==table\_names\_.end()){
220. return DB\_TABLE\_NOT\_EXIST;
221. }
222. table\_id\_t table\_id = table\_names\_[table\_name];
223. TableInfo\* table\_info = tables\_[table\_id];
224. tables\_.erase(table\_id);
225. table\_names\_.erase(table\_name);
226. table\_info->~TableInfo();
227. return DB\_SUCCESS;
228. }
229. */\*\**
230. \* TODO: Student Implement
231. \*/
232. dberr\_t CatalogManager::DropIndex(const string &table\_name, const string &index\_name) {
233. if(index\_names\_.find(table\_name)==index\_names\_.end()||index\_names\_.find(table\_name)->second.find(index\_name)==index\_names\_.find(table\_name)->second.end()){
234. return DB\_INDEX\_NOT\_FOUND;
235. }
236. index\_id\_t index\_id = index\_names\_[table\_name][index\_name];
237. indexes\_.erase(index\_id);
238. index\_names\_.erase(table\_name);
239. auto page\_id = catalog\_meta\_->index\_meta\_pages\_[index\_id];
240. catalog\_meta\_->index\_meta\_pages\_.erase(index\_id);
241. buffer\_pool\_manager\_->DeletePage(page\_id);
242. return DB\_SUCCESS;
243. }
244. */\*\**
245. \* TODO: Student Implement
246. \*/
247. dberr\_t CatalogManager::FlushCatalogMetaPage() const {
248. catalog\_meta\_->SerializeTo(buffer\_pool\_manager\_->FetchPage(CATALOG\_META\_PAGE\_ID)->GetData());
249. buffer\_pool\_manager\_->UnpinPage(CATALOG\_META\_PAGE\_ID,false);
250. buffer\_pool\_manager\_->FlushPage(CATALOG\_META\_PAGE\_ID);
251. return DB\_SUCCESS;
252. }
253. */\*\**
254. \* TODO: Student Implement
255. \*/
256. dberr\_t CatalogManager::LoadTable(const table\_id\_t table\_id, const page\_id\_t page\_id) {
257. if(tables\_.find(table\_id)!=tables\_.end()){
258. return DB\_TABLE\_ALREADY\_EXIST;
259. }
260. Page \*page = buffer\_pool\_manager\_->FetchPage(page\_id);*//获取表元数据页*
261. catalog\_meta\_->table\_meta\_pages\_[table\_id] = page\_id;*//更新表元数据页*
262. TableMetadata \*table\_metadata ;*//表元数据*
263. TableMetadata::DeserializeFrom(page->GetData(), table\_metadata);*//反序列化表元数据*
264. table\_names\_[table\_metadata->GetTableName()] = table\_id;*//更新表名*
265. TableInfo\* table\_info =TableInfo::Create();*//创建表信息*
266. TableHeap\* table\_heap = TableHeap::Create(buffer\_pool\_manager\_, page\_id,table\_metadata->GetSchema(), nullptr, nullptr);*//创建表堆*
267. table\_info->Init(table\_metadata, table\_heap);*//初始化表信息*
268. tables\_[table\_id] = table\_info;*//更新表信息*
269. return DB\_SUCCESS;
270. }
271. */\*\**
272. \* TODO: Student Implement
273. \*/
274. dberr\_t CatalogManager::LoadIndex(const index\_id\_t index\_id, const page\_id\_t page\_id) {
275. if(indexes\_.find(index\_id)!=indexes\_.end()){
276. return DB\_INDEX\_ALREADY\_EXIST;
277. }
278. Page \*page = buffer\_pool\_manager\_->FetchPage(page\_id);*//获取索引元数据页*
279. catalog\_meta\_->index\_meta\_pages\_[index\_id] = page\_id;*//更新索引元数据页*
280. IndexMetadata \*index\_metadata ;*//索引元数据*
281. IndexMetadata::DeserializeFrom(page->GetData(), index\_metadata);*//反序列化索引元数据*
282. index\_names\_[tables\_[index\_metadata->GetTableId()]->GetTableName()][index\_metadata->GetIndexName()] = index\_id;*//更新索引id*
283. IndexInfo\* index\_info =IndexInfo::Create();
284. index\_info->Init(index\_metadata,tables\_[index\_metadata->GetTableId()],buffer\_pool\_manager\_);*//初始化索引信息*
285. indexes\_[index\_id] = index\_info;*//更新索引信息*
286. return DB\_SUCCESS;
287. }
288. */\*\**
289. \* TODO: Student Implement
290. \*/
291. dberr\_t CatalogManager::GetTable(const table\_id\_t table\_id, TableInfo \*&table\_info) {
292. if(tables\_.find(table\_id)==tables\_.end()){
293. return DB\_TABLE\_NOT\_EXIST;
294. }
295. table\_info = tables\_[table\_id];
296. return DB\_SUCCESS;
297. }

CatalogMeta::GetSerializedSize()

这个函数计算了CatalogMeta对象序列化后的大小。具体步骤如下：

计算魔术数的大小，这是一个固定值（假设是一个常数）。

计算索引元数据页数组的大小，包括每个索引的索引ID和页号。

计算表元数据页数组的大小，包括每个表的索引ID和页号。

计算方式如下：

魔术数大小：sizeof(CATALOG\_METADATA\_MAGIC\_NUM)

索引元数据页数组大小：sizeof(index\_meta\_pages\_.size())

表元数据页数组大小：sizeof(table\_meta\_pages\_.size())

每个索引元数据页的大小：index\_meta\_pages\_.size() \* (sizeof(index\_id\_t) + sizeof(page\_id\_t))

每个表元数据页的大小：table\_meta\_pages\_.size() \* (sizeof(index\_id\_t) + sizeof(page\_id\_t))

最后将这些大小加总并返回作为对象的序列化后大小。

CatalogManager::CatalogManager()

这是目录管理器的构造函数。根据init参数的不同值，进行初始化或加载操作：

如果 init 为 true，则初始化目录元数据：

创建一个新的 CatalogMeta 实例。

清空相关的数据结构（table\_names\_, tables\_, indexes\_, index\_names\_）。

将目录元数据序列化到数据库的页中。

如果 init 为 false，则加载已有的目录元数据：

从数据库中读取已序列化的目录元数据。

根据读取的元数据加载所有的表和索引信息到内存中。

CatalogManager::CreateTable()

这个函数用于创建新的数据库表：

复制传入的表模式（schema）。

检查表名是否已存在，若不存在则继续：

分配新的表ID，并将表名和表ID关联起来。

为表的元数据分配一个新的页面。

将新的表元数据页与表ID关联。

创建一个新的表堆（TableHeap），用于存储表的数据。

创建新的表元数据（TableMetadata），包括表ID、表名、表堆的第一个页面ID和表模式，然后将其序列化到表的元数据页中。

解锁新分配的页面。

创建新的表信息（TableInfo），并初始化。

将新表信息放入 tables\_ 映射中。

CatalogManager::GetTable()

这个函数根据表名获取表的信息：

检查表名是否存在于 table\_names\_ 映射中，若不存在则返回 DB\_TABLE\_NOT\_EXIST。

若表名存在，通过 table\_names\_ 映射找到表的ID，然后通过 tables\_ 映射获取对应的 TableInfo 对象，并将其返回。

CatalogManager::GetTables()

这个函数获取所有已存在表的信息：

遍历 tables\_ 映射，将所有的 TableInfo 对象放入传入的 tables 向量中。

CatalogManager::CreateIndex(const std::string &table\_name, const string &index\_name, const std::vector[std::string](https://chat.chat826.com/#/std::string) &index\_keys, Txn \*txn, IndexInfo \*&index\_info, const string &index\_type)

表存在性检查：首先检查要创建索引的表是否存在于数据库中，如果不存在则返回相应的错误信息。

索引存在性检查：检查在同一表中是否已经存在同名的索引，如果存在则返回相应的错误信息。

变量初始化：初始化多个变量，用于存储表ID、表架构、meta页面ID、索引ID、索引元数据等信息。

索引信息初始化：通过调用相应对象的方法，初始化用于存储索引信息的对象。

获取表ID和架构信息：根据表名获取表的唯一标识符（ID），并从中获取表的架构信息。

索引键映射处理：将用户指定的列名映射为表中的列索引，确保索引所依赖的列在表中存在。

分配元数据页面：为新创建的索引分配一个页面，用于存储索引的元数据信息。

创建索引元数据：根据用户提供的信息创建索引的元数据对象，包括索引ID、索引名称、表ID以及索引所依赖的列映射。

序列化索引元数据：将索引元数据序列化为可存储的格式，并存储到先前分配的页面中。

初始化索引信息：使用索引元数据、表信息以及相关资源管理器，初始化用于存储索引信息的对象。

更新索引信息：将新创建的索引名称、索引ID等信息更新到数据库系统的相关数据结构中，确保索引在系统中的可访问性。

更新目录元数据：更新数据库系统中的目录元数据，将新创建索引的相关信息存储到永久存储介质中，以确保数据的持久性和一致性。

异常处理：通过捕获可能发生的异常情况，保证系统的稳定性和健壮性，同时返回适当的错误信息以提示用户。

成功返回：如果所有步骤顺利完成，返回成功的状态码，表示索引已经成功创建并可以使用。

GetIndex 函数分析：

首先，它检查给定的表名是否存在于索引名称映射中，如果不存在，则返回表不存在的错误码。

如果表存在，接着它再检查该表中是否存在指定的索引名称，如果不存在，则返回索引不存在的错误码。

如果表和索引都存在，则将对应索引的信息存储到传入的 index\_info 指针中，并返回操作成功的状态码。

GetTableIndexes 函数分析：

这个函数用于获取指定表名下的所有索引信息。

首先，它检查给定的表名是否存在于索引名称映射中，如果不存在，则返回表不存在的错误码。

如果表存在，接着它遍历该表中所有的索引，将每个索引的信息存储到传入的 indexes 向量中。

最后，它返回操作成功的状态码，表示获取表的所有索引信息操作成功完成。

DropTable 函数分析：

这个函数用于删除指定的表及其相关信息。

首先，它检查要删除的表是否存在于表名映射中，如果不存在，则返回表不存在的错误码。

如果表存在，接着它获取该表的 ID，并据此获取表的详细信息。

然后，它从内部数据结构中删除表的相关信息，并释放相应的资源（如表信息的内存空间）。

最后，它返回操作成功的状态码，表示删除表操作成功完成。

DropIndex 函数分析：

这个函数用于删除指定表中的指定索引。

首先，它检查要删除的索引所属的表名是否存在于索引名称映射中，以及指定的索引名是否存在于该表的索引映射中，如果任一条件不满足，则返回索引不存在的错误码。

如果索引存在，接着它获取该索引的 ID，并删除索引信息和索引名称映射中的对应条目。

然后，它从缓冲池管理器中删除索引元数据页，并释放相应的资源。

最后，它返回操作成功的状态码，表示删除索引操作成功完成。

FlushCatalogMetaPage 函数分析：

这个函数用于将目录元数据页的内容刷新到持久存储中。

首先，它将目录元数据序列化到缓冲池管理器获取的目录元数据页中。

然后，它取消固定目录元数据页，并将其刷新到持久存储。

最后，它返回操作成功的状态码，表示刷新目录元数据页操作成功完成。

LoadTable 函数分析：

这个函数用于加载表的元数据并创建对应的数据结构。

首先，它检查要加载的表 ID 是否已存在于表信息映射中，如果存在，则返回表已存在的错误码。

如果表不存在，接着它获取指定页面 ID 的页面，并更新目录元数据页中的表元数据页面信息。

然后，它反序列化表的元数据，获取表的详细信息，并更新表名映射为该表名。

接着，它创建表信息对象和表堆对象，并初始化表信息。

最后，它将表信息存储到表信息映射中，并返回操作成功的状态码，表示加载表操作成功完成。

LoadIndex 函数分析：

这个函数用于加载索引的元数据并创建对应的数据结构。

首先，它检查要加载的索引 ID 是否已存在于索引信息映射中，如果存在，则返回索引已存在的错误码。

如果索引不存在，接着它获取指定页面 ID 的页面，并更新目录元数据页中的索引元数据页面信息。

然后，它反序列化索引的元数据，获取索引的详细信息，包括索引所属的表名和索引名称。

接着，它更新索引名称映射中的索引 ID，并创建索引信息对象，并初始化索引信息。

最后，它将索引信息存储到索引信息映射中，并返回操作成功的状态码，表示加载索引操作成功完成。

CatalogManager::GetTable函数分析：

表是否存在检查：函数首先通过 tables\_（可能是一个映射或者类似结构）来查找是否存在给定 table\_id 的表。这一步是为了确定请求的表是否在数据库中注册。

表信息获取：如果表存在，函数会从 tables\_ 中获取该表的信息，并将其赋值给 table\_info。这里的 table\_info 是一个指向 TableInfo 对象的指针引用，通过这种方式修改 table\_info 的值会影响到调用函数的代码。

返回状态：如果成功找到并获取了表信息，则函数返回成功状态（DB\_SUCCESS），表示操作顺利完成。如果未能找到指定的表，则返回错误码 DB\_TABLE\_NOT\_EXIST，以通知调用方请求的表不存在。

三、测试结果





可以看到，catalog中与index无关的两个测试样例可以顺利通过，另一个则依然存在问题。

# #5 PLANNER AND EXECUTOR

负责人：汪珉凯

一、内容与功能

1.、Iterator Model：

- 这是一种经典的查询执行模型，也称为火山模型。它将整个 SQL 查询构建成一个算子（Operator）树的形式，通过迭代器的方式逐行处理数据。

- 每个算子（如select、insert、update、delete等）都有 `Init()` 和 `Next()` 两个方法。`Init()` 用于初始化算子的内部状态，`Next()` 则是请求下一个数据行。

- 算子之间的耦合性较低，每个算子只需关注自身的逻辑，这使得执行引擎的开发和维护更加简单和直观。

2、功能模块描述：

SeqScanExecutor (顺序扫描执行器):

SeqScanExecutor 用于执行对表的顺序扫描操作，即逐行遍历表格数据来检索符合指定条件的行。

每次调用 Next() 方法时，SeqScanExecutor 返回满足谓词条件的下一行数据及其行标识符（RID）。

如果谓词存在，需要使用谓词表达式的 Evaluate() 方法来判断当前行是否符合条件。

返回的结果是每个匹配行的副本，包括行数据和对应的 RID。

IndexScanExecutor (索引扫描执行器):

IndexScanExecutor 用于利用单列索引来执行表的扫描操作，加速查询性能。

当 Planner 确定查询中某列存在索引且查询条件只涉及单列时，会生成 IndexScanPlanNode。

如果谓词中存在 AND 条件，IndexScanExecutor 会根据索引找出满足所有条件的行。

对于每个满足索引条件的行，再使用谓词表达式的 Evaluate() 方法进一步验证是否符合所有查询条件。

InsertExecutor (插入执行器):

InsertExecutor 负责将新行插入到表中，并更新所有相关的索引，确保数据的一致性和完整性。

在 Parser 阶段解析出的插入值通过 ValueExecutor 生成对应的行数据，随后传递给 InsertExecutor。

插入操作需要检查表中的唯一性约束（unique），确保插入的数据不会导致重复值的存在。

返回空的元组用于统计表中受影响的行数，确保事务操作的正确性。

UpdateExecutor (更新执行器):

UpdateExecutor 用于修改表中已有的行数据，并更新相关的索引以保持数据的一致性。

更新操作首先通过 SeqScanExecutor 找到符合条件的行，并生成新的更新行数据。

更新操作实质上是先删除旧行，然后插入新行的过程，确保事务的原子性和数据的完整性。

返回空的元组用于记录更新操作影响的行数，确保操作结果的正确性。

DeleteExecutor (删除执行器):

DeleteExecutor 负责删除表中符合条件的行，并更新相关的索引信息。

删除操作与更新操作类似，也是通过 SeqScanExecutor 找到符合条件的行，并执行删除操作。

删除操作需要同时更新表格的索引数据，确保删除行后索引与表格数据的一致性。

返回空的元组用于统计删除操作影响的行数，确保数据库操作的正确性和可靠性。

3、技术细节和实现提示：

- 各算子的具体实现需要考虑底层数据结构的优化和性能问题，如使用迭代器、谓词表达式的评估、单线程环境下的安全性等。

- 索引的管理和更新是关键，对于 Insert、Update 和 Delete 操作，需要确保索引与底层表的一致性。

- 各种算子的设计和实现需考虑 SQL 查询语句的各种可能性，以保证数据库系统在不同工作负载下的高效运行和正确性。

通过以上功能性描述，可以深入理解和实现一个基本的数据库查询执行引擎，支持常见的数据操作和索引管理，以提升数据库系统的性能和可靠性。

二、代码实现与分析

1. dberr\_t ExecuteEngine::ExecuteCreateTable(pSyntaxNode ast, ExecuteContext \*context) {
2. *//create a table*
3. #ifdef ENABLE\_EXECUTE\_DEBUG
4. LOG(INFO) << "ExecuteCreateTable" << std::endl;
5. #endif
6. *//check if a database is selected*
7. if(current\_db\_.empty())return DB\_FAILED;
8. *//get the table name*
9. string table\_name = ast->child\_->val\_;
10. *//get the context name*
11. auto context\_name= context->GetCatalog();
12. *//get the first child of the ast*
13. pSyntaxNode node = ast->child\_->next\_->child\_;
14. vector<Column\*> columns;
15. vector<string > primary\_keys\_columns;
16. vector<string > unique\_keys\_columns;
17. *//get the primary keys*
18. while(node) {
19. if(!strcmp(node->val\_,"primary keys") && node->type\_== kNodeColumnList ) {
20. auto primary\_temp=node->child\_;
21. while(primary\_temp) {
22. primary\_keys\_columns.push\_back(primary\_temp->val\_);
23. primary\_temp=primary\_temp->next\_;
24. }
25. }
26. node=node->next\_;
27. }
28. *//get the unique keys*
29. node=ast->child\_->next\_->child\_;
30. while(node){
31. if(!strcmp(node->val\_,"unique keys") && node->type\_== kNodeColumnList ) {
32. auto unique\_temp=node->child\_;
33. while(unique\_temp) {
34. unique\_keys\_columns.push\_back(unique\_temp->val\_);
35. unique\_temp=unique\_temp->next\_;
36. }
37. }
38. node=node->next\_;
39. }
40. int index=0;
41. node=ast->child\_->next\_->child\_;
42. *//iterate*
43. while(node && node->type\_ == kNodeColumnDefinition) {
44. bool unique=(node->val\_ && node->val\_=="unique");
45. *//get the column name*
46. string column\_name = node->child\_->val\_;
47. *//get the column type*
48. string column\_type = node->child\_->next\_->val\_;
49. *//create the object according to the column type*
50. Column\* column;
51. if(column\_type=="int")column = new Column(column\_name,kTypeInt,index,true,unique);
52. if(column\_type=="float")column = new Column(column\_name,kTypeFloat,index,true,unique);
53. if(column\_type=="char") {
54. *// 获取字符列长度*
55. string len=node->child\_->next\_->child\_->val\_;
56. *// 遍历长度字符串的每个字符*
57. for(auto iterator: len)
58. *// 如果字符不是数字，则返回数据库操作失败*
59. if(!isdigit(iterator))
60. return DB\_FAILED;
61. *// 将长度字符串转换为整数*
62. if(stoi(len) < 0)
63. *// 如果长度小于0，则返回数据库操作失败*
64. return DB\_FAILED;
65. *// 创建一个新的列对象，类型为字符型*
66. *// 参数分别为列名、列类型、长度、索引标志、是否为空、是否唯一*
67. column = new Column(column\_name, kTypeChar, stoi(len), index, true, unique);
68. }
69. columns.push\_back(column);
70. index++;
71. node=node->next\_;
72. }
73. }
74. */\*\**
75. \* TODO: Student Implement
76. \*/
77. dberr\_t ExecuteEngine::ExecuteDropTable(pSyntaxNode ast, ExecuteContext \*context) {
78. #ifdef ENABLE\_EXECUTE\_DEBUG
79. LOG(INFO) << "ExecuteDropTable" << std::endl;
80. #endif
81. auto result = context->GetCatalog()->DropTable(ast->child\_->val\_);
82. cout<<ast->child\_->val\_<<" has been dropped"<<endl;
83. return result;
84. }
85. */\*\**
86. \* TODO: Student Implement
87. \*/
88. dberr\_t ExecuteEngine::ExecuteShowIndexes(pSyntaxNode ast, ExecuteContext \*context) {
89. #ifdef ENABLE\_EXECUTE\_DEBUG
90. LOG(INFO) << "ExecuteShowIndexes" << std::endl;
91. #endif
92. vector <TableInfo\*> tables;*//存储表信息*
93. context->GetCatalog()->GetTables(tables);*//获取表信息*
94. for(auto iterator:tables){*//遍历表*
95. vector<IndexInfo\*> indexes;*//存储索引信息*
96. string table\_name = iterator->GetTableName();*//获取表名*
97. context->GetCatalog()->GetTableIndexes(table\_name,indexes);*//获取索引信息*
98. for(auto index: indexes){*//遍历索引*
99. string index\_name = index->GetIndexName();*//获取索引名*
100. std::cout<<table\_name<<"  "<<index\_name<<std::endl;*//输出索引名*
101. }
102. }
103. return DB\_SUCCESS;
104. }
105. */\*\**
106. \* TODO: Student Implement
107. \*/
108. dberr\_t ExecuteEngine::ExecuteCreateIndex(pSyntaxNode ast, ExecuteContext \*context) {
109. #ifdef ENABLE\_EXECUTE\_DEBUG
110. LOG(INFO) << "ExecuteCreateIndex" << std::endl;
111. #endif
112. *//table name*
113. string table\_name = ast->child\_->next\_->val\_;
114. *//index name*
115. string index\_name = ast->child\_->val\_;
116. *//columns*
117. vector<string> columns;
118. for(auto i=ast->child\_->next\_->next\_->child\_;i!=nullptr;i=i->next\_){
119. columns.push\_back(i->val\_);
120. }
121. IndexInfo\* index;
122. auto result=context->GetCatalog()->CreateIndex(table\_name,index\_name,columns,context->GetTransaction(),index,"btree");
123. return result;
124. }
125. */\*\**
126. \* TODO: Student Implement
127. \*/
128. dberr\_t ExecuteEngine::ExecuteDropIndex(pSyntaxNode ast, ExecuteContext \*context) {
129. #ifdef ENABLE\_EXECUTE\_DEBUG
130. LOG(INFO) << "ExecuteDropIndex" << std::endl;
131. #endif
132. *//fetch imformation*
133. vector<TableInfo\*> tables;
134. context->GetCatalog()->GetTables(tables);
135. *//loop through every table*
136. for(auto it:tables) {
137. *//get the table name*
138. string table\_name=it->GetTableName();
139. *//get the indexs*
140. vector<IndexInfo\*> indexs;
141. context->GetCatalog()->GetTableIndexes(table\_name,indexs);
142. *//loop through every index*
143. for(auto index: indexs) {
144. *//get the index name*
145. string index\_name = index->GetIndexName();
146. if(index\_name==ast->child\_->val\_ ){
147. *//drop the index*
148. auto result=context->GetCatalog()->DropIndex(table\_name,index\_name);
149. return result;
150. }
151. }
152. }
153. return DB\_INDEX\_NOT\_FOUND;
154. }
155. dberr\_t ExecuteEngine::ExecuteTrxBegin(pSyntaxNode ast, ExecuteContext \*context) {
156. #ifdef ENABLE\_EXECUTE\_DEBUG
157. LOG(INFO) << "ExecuteTrxBegin" << std::endl;
158. #endif
159. return DB\_FAILED;
160. }
161. dberr\_t ExecuteEngine::ExecuteTrxCommit(pSyntaxNode ast, ExecuteContext \*context) {
162. #ifdef ENABLE\_EXECUTE\_DEBUG
163. LOG(INFO) << "ExecuteTrxCommit" << std::endl;
164. #endif
165. return DB\_FAILED;
166. }
167. dberr\_t ExecuteEngine::ExecuteTrxRollback(pSyntaxNode ast, ExecuteContext \*context) {
168. #ifdef ENABLE\_EXECUTE\_DEBUG
169. LOG(INFO) << "ExecuteTrxRollback" << std::endl;
170. #endif
171. return DB\_FAILED;
172. }
173. */\*\**
174. \* TODO: Student Implement
175. \*/
176. dberr\_t ExecuteEngine::ExecuteExecfile(pSyntaxNode ast, ExecuteContext \*context) {
177. #ifdef ENABLE\_EXECUTE\_DEBUG
178. LOG(INFO) << "ExecuteExecfile" << std::endl;
179. #endif
180. *//get the file name from the ast*
181. string file\_name = ast->child\_->val\_;
182. int MAX\_SIZE = 1024;
183. *//open the file*
184. ifstream infile(file\_name, ios::in);
185. if (!infile) {
186. cout << "File not found" << endl;
187. return DB\_FAILED;
188. }
189. infile>>std::noskipws;*//不跳过空格*
190. while(!infile.eof()){ *//*
191. char buffer[MAX\_SIZE];*//缓冲区*
192. *//initialize the buffer*
193. memset(buffer,0,MAX\_SIZE);
194. int i=0;*//缓冲区的下标*
195. char c;*//字符*
196. *// Read characters from the file until a semicolon is encountered or end of file is reached*
197. while(!infile.eof() && infile.get(c) && c!=';'){*//读取文件中的字符，直到遇到分号或者文件结束*
198. buffer[i++]=c;*//将读取的字符存入缓冲区*
199. }
200. if(infile.eof()) {
201. *// If the end of the file is reached, break the loop*
202. continue;
203. }
204. buffer[i++]=';';*//将分号存入缓冲区*
205. YY\_BUFFER\_STATE buffer\_state = yy\_scan\_string(buffer);*//扫描缓冲区*
206. if(buffer\_state==nullptr) {
207. *// If the buffer state is null, return database operation failed*
208. return DB\_FAILED;
209. }
210. yy\_switch\_to\_buffer(buffer\_state);*//切换到缓冲区*
211. *//initialize the parser*
212. MinisqlParserInit();
213. *//parse the buffer*
214. yyparse();
215. if(MinisqlParserGetError()) {
216. *// If there is an error in the parser, return database operation failed*
217. return DB\_FAILED;
218. }
219. auto result=this->Execute(MinisqlGetParserRootNode());
220. if(result!=DB\_SUCCESS) {
221. *// If the result is not successful, return database operation failed*
222. return DB\_FAILED;
223. }
224. }
225. return DB\_SUCCESS;
226. }
227. */\*\**
228. \* TODO: Student Implement
229. \*/
230. dberr\_t ExecuteEngine::ExecuteQuit(pSyntaxNode ast, ExecuteContext \*context) {
231. #ifdef ENABLE\_EXECUTE\_DEBUG
232. LOG(INFO) << "ExecuteQuit" << std::endl;
233. #endif
234. return DB\_QUIT;
235. }

ExecuteCreateTable 函数分析

这个函数用于执行创建表的操作。首先检查是否已选择数据库，若未选择则返回失败。然后从语法树中获取表名，并获取执行上下文的目录信息。接着从语法树中解析出主键和唯一键的列名，并存储到相应的向量中。然后遍历语法树，解析每个列的定义，包括列名和列类型。根据列类型创建对应的列对象，并根据是否唯一设置属性。最后将每个列对象存储到一个列向量中。

ExecuteDropTable 函数分析

这个函数用于执行删除表的操作。首先从语法树中获取要删除的表名，并调用执行上下文的方法来删除表。在函数的末尾，会输出一条信息表明表已经被成功删除。

ExecuteShowIndexes 函数分析

这个函数用于展示数据库中所有表的索引信息。首先获取执行上下文中的所有表信息，然后遍历每个表。对于每个表，获取其所有索引信息，并输出每个索引的名称以及其所属的表名。

ExecuteCreateIndex 函数分析

这个函数用于执行创建索引的操作。从语法树中获取表名、索引名和索引的列信息。然后调用执行上下文中的方法来创建索引，指定使用的事务和索引类型（在这里是 "btree"）。最终返回操作结果。

ExecuteDropIndex 函数分析

这个函数用于执行删除索引的操作。首先获取执行上下文中的所有表信息，并遍历每个表。对于每个表，获取其所有索引信息，然后遍历每个索引。如果找到要删除的索引名与语法树中指定的索引名相符，则调用执行上下文中的方法删除该索引，并返回操作结果。如果未找到要删除的索引，则返回 DB\_INDEX\_NOT\_FOUND 表示索引未找到。

ExecuteExecfile 函数分析

这个函数的作用是执行一个指定文件中的 SQL 命令。它首先从语法树（ast）中获取文件名，然后尝试打开该文件。如果文件无法打开，函数会输出 "File not found" 并返回 DB\_FAILED 表示操作失败。

如果文件成功打开，函数进入主循环。在循环中，它定义了一个大小为 MAX\_SIZE 的字符数组作为缓冲区，用于逐字符读取文件内容。每次读取字符直到遇到分号 ; 或者文件末尾。读取到的字符存入缓冲区中，并在末尾加上分号，以确保每次解析的是完整的 SQL 命令。

接着，函数使用 yy\_scan\_string 函数扫描该缓冲区，将其作为输入传递给解析器。如果缓冲区扫描失败（返回空指针），则函数返回 DB\_FAILED 表示数据库操作失败。

然后，函数初始化解析器并调用 yyparse() 函数来解析缓冲区中的 SQL 命令。如果解析器返回错误（通过 MinisqlParserGetError() 函数检查），则函数返回 DB\_FAILED。

如果解析成功，函数调用 this->Execute(MinisqlGetParserRootNode()) 执行解析得到的语法树根节点，返回执行结果。如果执行结果不是 DB\_SUCCESS，则函数也返回 DB\_FAILED。

最终，如果文件中还有未处理的内容，循环继续读取并执行，直到文件末尾。如果整个文件处理完毕且没有发生错误，则函数返回 DB\_SUCCESS 表示执行成功。

ExecuteQuit 函数分析

这个函数简单地设置了数据库会话的退出状态。它直接返回 DB\_QUIT，表示程序应该退出执行，通常在解释器或主循环中用来检测是否需要结束会话。

三、测试结果



由于先前index模块仍存在bug，目前我们的executor也会受到影响，测试样例无法通过。

# #6 RECOVERY MANAGER

负责人：陈勇帆

一、实验目的

模块六的目的是实现一个简化版的数据恢复管理器（Recovery Manager）。Recovery Manager 负责管理和维护数据库的恢复过程，包括：定义日志结构；定义检查点（CheckPoint）；执行Redo、Undo等操作，处理插入、删除、更新，事务的开始、提交、回滚等日志，将数据库恢复到宕机之前的状态。

二、内容与功能

在数据库系统中，数据恢复是确保数据一致性和持久性的关键组件。数据恢复主要包括两个阶段：

Redo阶段：重做操作，以确保所有已提交的事务的修改被持久化。

Undo阶段：回滚操作，撤销所有未完成的事务，确保数据的一致性和原子性。

为了简化实现，并避免各模块之间的强耦合，我们将Recovery Manager模块独立出来，并用一个unordered\_map模拟一个KV Database，同时实现一个纯内存的数据恢复模块，不考虑日志的持久化。

（1）日志结构定义

日志结构LogRec定义了各种类型的日志记录，包括插入、删除、更新、事务开始、提交、回滚等。每条日志记录包含日志类型、事务ID、键值对信息、前一个日志的LSN（Log Sequence Number）等。

（2）CheckPoint定义

CheckPoint包含当前数据库的一个完整状态，包括：

检查点的LSN；活跃事务列表；持久化的数据（KV Database）。

（3）Redo和Undo操作

第一阶段：Redo Log恢复：RedoPhase()：从CheckPoint开始，根据不同类型的日志记录对KV Database和活跃事务列表进行修改。Redo操作确保所有已提交的事务的修改被持久化。

数据库启动后，通过Redo Log找到最近一次CheckPoint的位置。从该CheckPoint开始，获取需要重做的日志。解析日志并保存到哈希表中。遍历哈希表中的Redo Log信息，读取相关页进行恢复。在此阶段，所有记录到Redo Log但尚未完成数据刷盘的记录都会被重新写入磁盘。然而，仅依靠Redo Log是不够的，因为需要保证数据库在任何时候都能恢复到一致性的状态。

第二阶段：Undo Log恢复：UndoPhase()：对每个未完成的活跃事务进行回滚，确保数据的一致性和原子性。

根据Undo Log中的信息构造所有未提交事务的链表。判断事务是否需要提交或回滚。

使用MVCC（多版本并发控制）确保事务隔离性，恢复出对于一个事务可见的数据。在我们的实验中，日志在内存中以LogRec的形式表现。为了简化实现，我们直接在内存中定义一个能够用于插入、删除、更新，事务的开始、提交、回滚的日志结构，并实现一个简化的Recovery Manager模块。

这两阶段的操作需要按照log的类型来分类，且redo和undo的具体操作相反，其中注意undo只对活跃事务操作，redo中遇到commit或abort则将事务移出活跃列表。

三、代码实现与分析

**（一）LogRec.h**

（1）LogRec结构修改：

    struct LogRec {

        LogRec() = default;

        // 插入、删除、更新日志构造函数

        LogRec(LogRecType type, txn\_id\_t txn\_id, KeyType key, ValType val, lsn\_t prev\_lsn)

            : type\_(type), lsn\_(next\_lsn\_++), prev\_lsn\_(prev\_lsn), txn\_id\_(txn\_id), key\_(std::move(key)), val\_(val) {

        }

        // 更新日志构造函数（包含旧键、旧值和新键、新值）

        LogRec(LogRecType type, txn\_id\_t txn\_id, KeyType old\_key, ValType old\_val, KeyType new\_key, ValType new\_val, lsn\_t prev\_lsn)

            : type\_(type), lsn\_(next\_lsn\_++), prev\_lsn\_(prev\_lsn), txn\_id\_(txn\_id), old\_key\_(std::move(old\_key)), old\_val\_(old\_val), new\_key\_(std::move(new\_key)), new\_val\_(new\_val) {

        }

        LogRecType type\_{LogRecType::kInvalid};  // 日志类型

        lsn\_t lsn\_{INVALID\_LSN};                 // 日志序列号

        lsn\_t prev\_lsn\_{INVALID\_LSN};            // 前一个日志的LSN

        txn\_id\_t txn\_id\_{INVALID\_TXN\_ID};        // 事务ID

        KeyType key\_{};                          // 键

        ValType val\_{};                          // 值

        KeyType old\_key\_{};                      // 旧键（用于更新日志）

        ValType old\_val\_{};                      // 旧值（用于更新日志）

        KeyType new\_key\_{};                      // 新键（用于更新日志）

        ValType new\_val\_{};                      // 新值（用于更新日志）

        static std::unordered\_map<txn\_id\_t, lsn\_t> prev\_lsn\_map\_;  // 记录每个事务的前一个日志的LSN

        static lsn\_t next\_lsn\_;                                    // 下一个日志的LSN

    };

（2）日志创建函数

    // 创建插入日志

    static LogRecPtr CreateInsertLog(txn\_id\_t txn\_id, KeyType ins\_key, ValType ins\_val) {

        lsn\_t prev\_lsn = LogRec::prev\_lsn\_map\_[txn\_id];  // 获取前一个日志的LSN

        auto log = std::make\_shared<LogRec>(LogRecType::kInsert, txn\_id, std::move(ins\_key), ins\_val, prev\_lsn);

        LogRec::prev\_lsn\_map\_[txn\_id] = log->lsn\_;  // 更新前一个日志的LSN

        return log;

    }

    // 创建删除日志

    static LogRecPtr CreateDeleteLog(txn\_id\_t txn\_id, KeyType del\_key, ValType del\_val) {

        lsn\_t prev\_lsn = LogRec::prev\_lsn\_map\_[txn\_id];  // 获取前一个日志的LSN

        auto log = std::make\_shared<LogRec>(LogRecType::kDelete, txn\_id, std::move(del\_key), del\_val, prev\_lsn);

        LogRec::prev\_lsn\_map\_[txn\_id] = log->lsn\_;  // 更新前一个日志的LSN

        return log;

    }

    // 创建更新日志

    static LogRecPtr CreateUpdateLog(txn\_id\_t txn\_id, KeyType old\_key, ValType old\_val, KeyType new\_key, ValType new\_val) {

        lsn\_t prev\_lsn = LogRec::prev\_lsn\_map\_[txn\_id];  // 获取前一个日志的LSN

        auto log = std::make\_shared<LogRec>(LogRecType::kUpdate, txn\_id, std::move(old\_key), old\_val, std::move(new\_key), new\_val, prev\_lsn);

        LogRec::prev\_lsn\_map\_[txn\_id] = log->lsn\_;  // 更新前一个日志的LSN

        return log;

    }

    // 创建事务开始日志

    static LogRecPtr CreateBeginLog(txn\_id\_t txn\_id) {

        lsn\_t prev\_lsn = INVALID\_LSN;  // 事务开始时没有前一个日志

        auto log = std::make\_shared<LogRec>(LogRecType::kBegin, txn\_id, "", 0, prev\_lsn);

        LogRec::prev\_lsn\_map\_[txn\_id] = log->lsn\_;  // 更新前一个日志的LSN

        return log;

    }

    // 创建事务提交日志

    static LogRecPtr CreateCommitLog(txn\_id\_t txn\_id) {

        lsn\_t prev\_lsn = LogRec::prev\_lsn\_map\_[txn\_id];  // 获取前一个日志的LSN

        auto log = std::make\_shared<LogRec>(LogRecType::kCommit, txn\_id, "", 0, prev\_lsn);

        LogRec::prev\_lsn\_map\_[txn\_id] = log->lsn\_;  // 更新前一个日志的LSN

        return log;

    }

    // 创建事务回滚日志

    static LogRecPtr CreateAbortLog(txn\_id\_t txn\_id) {

        lsn\_t prev\_lsn = LogRec::prev\_lsn\_map\_[txn\_id];  // 获取前一个日志的LSN

        auto log = std::make\_shared<LogRec>(LogRecType::kAbort, txn\_id, "", 0, prev\_lsn);

        LogRec::prev\_lsn\_map\_[txn\_id] = log->lsn\_;  // 更新前一个日志的LSN

        return log;

    }

（二）RecoveryManager

①init函数：

    void RecoveryManager::Init(CheckPoint &last\_checkpoint) {

        persist\_lsn\_ = last\_checkpoint.checkpoint\_lsn\_;  // 恢复检查点的LSN

        active\_txns\_ = last\_checkpoint.active\_txns\_;    // 恢复活跃事务列表

        data\_ = last\_checkpoint.persist\_data\_;          // 恢复持久化数据

}

②redo函数：执行 Redo 操作，根据日志类型修改 KV Database 和活跃事务列表。

    void RedoPhase() {

        for (const auto &log\_pair : log\_recs\_) {

            lsn\_t lsn = log\_pair.first;

            const LogRecPtr &log\_rec = log\_pair.second;

            // 如果日志的LSN小于持久化LSN，则跳过该日志

            if (lsn < persist\_lsn\_) {

            continue;

            }

            // 根据日志类型进行不同操作

            switch (log\_rec->type\_) {

            case LogRecType::kInsert:

                data\_[log\_rec->key\_] = log\_rec->val\_;  // 插入数据

                active\_txns\_[log\_rec->txn\_id\_] = log\_rec->lsn\_;  // 更新活跃事务列表

                break;

            case LogRecType::kDelete:

                data\_.erase(log\_rec->key\_);  // 删除数据

                active\_txns\_[log\_rec->txn\_id\_] = log\_rec->lsn\_;  // 更新活跃事务列表

                break;

            case LogRecType::kUpdate:

                data\_[log\_rec->new\_key\_] = log\_rec->new\_val\_;  // 更新新数据

                data\_[log\_rec->old\_key\_] = log\_rec->old\_val\_;  // 恢复旧数据

                active\_txns\_[log\_rec->txn\_id\_] = log\_rec->lsn\_;  // 更新活跃事务列表

                break;

            case LogRecType::kBegin:

                active\_txns\_[log\_rec->txn\_id\_] = log\_rec->lsn\_;  // 记录事务开始

                break;

            case LogRecType::kCommit:

                active\_txns\_.erase(log\_rec->txn\_id\_);  // 移除已提交的事务

                break;

            case LogRecType::kAbort:

                // 回滚事务

                active\_txns\_.erase(log\_rec->txn\_id\_);  // 移除已回滚的事务

                lsn = log\_rec->prev\_lsn\_;  // 获取前一个日志的LSN

                while (lsn != INVALID\_LSN) {

                const LogRecPtr &log\_rec = log\_recs\_[lsn];

                switch (log\_rec->type\_) {

                    case LogRecType::kInsert:

                    data\_.erase(log\_rec->key\_);  // 回滚插入

                    break;

                    case LogRecType::kDelete:

                    data\_[log\_rec->key\_] = log\_rec->val\_;  // 恢复删除

                    break;

                    case LogRecType::kUpdate:

                    data\_[log\_rec->new\_key\_] = log\_rec->old\_val\_;  // 恢复更新（新值变旧值）

                    break;

                    default:

                    break;

                }

                lsn = log\_rec->prev\_lsn\_;  // 继续回滚前一个日志

                }

                break;

            default:

                break;

            }

        }

}

③undo函数：执行 Undo 操作，对每个未完成的活跃事务进行回滚。

    void UndoPhase() {

        // 遍历所有活跃事务

        for (const auto &txn\_pair : active\_txns\_) {

            txn\_id\_t txn\_id = txn\_pair.first;  // 获取事务ID

            lsn\_t lsn = txn\_pair.second;       // 获取事务的最后一个日志的LSN

            // 回滚该事务的所有日志

            while (lsn != INVALID\_LSN) {

            const LogRecPtr &log\_rec = log\_recs\_[lsn];  // 获取当前LSN对应的日志记录

            // 根据日志类型进行不同的回滚操作

            switch (log\_rec->type\_) {

                case LogRecType::kInsert:

                data\_.erase(log\_rec->key\_);  // 回滚插入操作，删除插入的数据

                break;

                case LogRecType::kDelete:

                data\_[log\_rec->key\_] = log\_rec->val\_;  // 回滚删除操作，恢复删除的数据

                break;

                case LogRecType::kUpdate:

                data\_[log\_rec->new\_key\_] = log\_rec->old\_val\_;  // 回滚更新操作，恢复旧的数据值

                break;

                default:

                break;

            }

            lsn = log\_rec->prev\_lsn\_;  // 获取前一个日志的LSN，继续回滚

            }

        }

        // 清空活跃事务列表

        active\_txns\_.clear();

    }

四、测试代码分析与结果

**（一）代码分析：**

1、日志创建与事务操作模拟

创建不同类型的日志记录，模拟了三个事务 (T0, T1, T2) 的操作，包括开始事务、插入、删除、更新、提交和回滚等操作。验证日志记录的 prev\_lsn\_ 值是否正确，确保日志记录链表正确连接。

2、Checkpoint 创建

创建一个检查点（CheckPoint），记录在某个时刻的数据库状态，包括：

检查点 LSN（checkpoint\_lsn\_）：记录在 d3 的 LSN。

活跃事务（active\_txns\_）：记录事务 T0 和 T1 的最新 LSN。

数据库状态（persist\_data\_）：记录键 A 的值为 2050。

3、添加后续日志记录

创建并添加事务 T1 和 T0 的后续日志记录，包括插入、更新、提交和回滚等操作。

创建并添加事务 T2 的日志记录，包括开始事务、插入和更新操作。

4、恢复管理器初始化与日志追加

初始化 RecoveryManager，使用之前创建的检查点来恢复数据库的初始状态。

将所有日志记录追加到 RecoveryManager 中，准备进行恢复操作。

5、RedoPhase 验证

执行 RedoPhase，重做所有日志记录，从而将数据库状态恢复到最接近事务结束时的状态。

断言数据库状态是否符合预期：

键 A 应为 2000（回滚到旧值）。

键 B 应为 1000（删除被回滚）。

键 C 应为 800（事务 T2 的更新）。

键 D 应为 30000（事务 T2 的插入）。

6、UndoPhase 验证

执行 UndoPhase，撤销所有未提交或回滚的事务操作。

断言数据库状态是否符合预期：

键 A 应为 2000（保持不变）。

键 B 应为 1000（保持不变）。

键 C 应回滚为 600（事务 T2 被撤销）。

键 D 应被删除（事务 T2 被撤销）。

**（二）结果：测试通过，如图：**

