浙江大學

数据库系统实验报告

题　　目：	MiniSQL
授课教师：	孙建伶
助　　教：	聂俊哲/石宇新
姓　　名：	王淳
学　　号：	3220105023
邮　　箱：	zjuheadmaster@zju.edu.cn
联系电话：	13428807817

Contents

1 DISK AND BUFFER POOL MANAGER

1.1 位图页的实现

- 这个部分实现`AllocatePage`,`DeAllocatePage`,`IsPageFree`,`IsPageFreeLow`四个函数，支持对位图页的分配以及删除，还有对其状态的检查。

1.1.1 AllocatePage

```
template<size_t PageSize>
bool BitmapPage<PageSize>::AllocatePage(uint32_t &page_offset) {
  bool IsSuccess = false;
  /*pages allocated are less than the supported size*/
  if(page_allocated_ < GetMaxSupportedSize()){
    this->page_allocated_++;
    /*Find the page to allocate*/
    while(!IsPageFree(this->next_free_page_)&& this->next_free_page_ <
GetMaxSupportedSize()){
      this->next_free_page_++;
    }
    page_offset=this->next_free_page_;

    /*Byte Index*/
    uint32_t byte_index = this->next_free_page_/8;
    /*Bit Index*/
    uint8_t bit_index = this->next_free_page_%8;
    /*mark in the bitmap*/
    uint8_t tmp = 0x01;
    bytes[byte_index] = (bytes[byte_index]|(tmp<<(7-bit_index)));
    /*point to next page*/
    while(!IsPageFree(this->next_free_page_)&&this-
>next_free_page_<GetMaxSupportedSize()) {
      this->next_free_page_++;
    }
    IsSuccess = true;
  }

  return IsSuccess;
}
```

- 在确定位图中的`Byte Index`以及`Bit Index`之前，要先判断是否存在额外的空余页可供处理，若有则找到该页，如果没有则返回`False`，若有，则确定该页的`Byte Index`以及`Byte Index`，更新`Bitmap`后将新开的`Page`的偏移地址传回。

1.1.2 DeAllocatePage

```
template<size_t PageSize>
bool BitmapPage<PageSize>::DeAllocatePage(uint32_t page_offset) {
  /*Byte Index*/
  uint32_t byte_index=page_offset/8;
  /*Bit Index*/
  uint8_t bit_index=page_offset%8;
```

```
  bool IsSuccess=false;
  /*Only deallocated when the page isn't free*/
  if( this->page_allocated_ && !IsPageFree(page_offset)){

    uint8_t tmp=0x01;

    bytes[byte_index]=bytes[byte_index]&(~(tmp<<(7-bit_index)));
    this->page_allocated_--;
    /*update the free page*/
    if(page_offset<this->next_free_page_)
      this->next_free_page_=page_offset;

    IsSuccess=true;
  }

  return IsSuccess;
}
```

- 这里先检查传入要释放的页偏移地址的空间是否被使用，若无，则返回`False`，否则更新`bitmap`以及更新`next_free_page_`指针。

1.1.3 IsPageFree

```
template<size_t PageSize>
bool BitmapPage<PageSize>::IsPageFree(uint32_t page_offset) const {
  /*Byte Index*/
  uint32_t byte_index=page_offset/8;
  /*Bit Index*/
  uint8_t bit_index=page_offset%8;
  return IsPageFreeLow(byte_index, bit_index);
}
```

- 通过检查`bitmap`中对应的位来确定是否为`Free page`.

1.1.4 IsPageFreeLow

```
template <size_t PageSize>
bool BitmapPage<PageSize>::IsPageFreeLow(uint32_t byte_index, uint8_t bit_index) const
{
  uint8_t tmp=0x01;

  if(bytes[byte_index]&(tmp<<(7-bit_index))) return false;
  else return true;
}
```

- 这个函数实现了根据输入的`bit_index`以及`byte_index`来判断是否为空页。

1.2 磁盘数据页管理

- `DiskManager::AllocatePage()`：从磁盘中分配一个空闲页，并返回空闲页的**逻辑页号**；
- `DiskManager::DeAllocatePage(logical_page_id)`：释放磁盘中**逻辑页号**对应的物理页。
- `DiskManager::IsPageFree(logical_page_id)`：判断该**逻辑页号**对应的数据页是否空闲。
- `DiskManager::MapPageId(logical_page_id)`：可根据需要实现。在`DiskManager`类的私有成员中，该函数可以用于将逻辑页号转换成物理页号。

- `GetExtentNums()`: 返回区间数。
- `GetAllocatedPages()`: 返回已分配的页面数。
- `GetExtentUsedPage(uint32_t extent_id)`: 返回指定区间已使用的页面数，如果区间ID超出范围，返回0。
- `num_allocated_pages_`: 已分配页面数。
- `num_extents_`: 区间数，每个区间由一个位图和若干页面组成。
- `extent_used_page_`: 每个区间已使用的页面数。

1.2.1 AllocatePage

`AllocatePage`函数用于在磁盘上分配新的页面，并更新相关的元数据。具体步骤包括更新磁盘文件元数据页、查找非满的区间（extent）、读取位图页、分配新的页面并写回磁盘。

```
page_id_t DiskManager::AllocatePage() {
  /*Read the meta data*/
  DiskFileMetaPage *meta_page = reinterpret_cast<DiskFileMetaPage *>(this->meta_data_);
  /*assign the next page to return*/
  uint32_t NextPage=0;
  bool IsSuccess = false;
  /*New disk*/
  if(!meta_page->GetExtentNums()){
    /*extents*/
    meta_page->num_extents_++;
    /*pages*/
    meta_page->num_allocated_pages_++;
    /*used page*/
    meta_page->extent_used_page_[0] = 1;
    /*read the bitmap data from disk*/
    char Page_Data[PAGE_SIZE];
    ReadPhysicalPage(1,Page_Data);
    BitmapPage<PAGE_SIZE> *Bitmap_page = reinterpret_cast<BitmapPage<PAGE_SIZE> *>(Page_Data);

    IsSuccess = Bitmap_page->AllocatePage(NextPage);
    if(IsSuccess){
      char *Page_Data = reinterpret_cast<char *>(Bitmap_page);
      WritePhysicalPage(1,Page_Data);
    }else{
      std::cout << "AllocatePage Failed  ----- at the begin!" << std::endl;
    }
  }else{
    meta_page->num_allocated_pages_++;
    bool NewOpen = true;
    uint32_t i;
    for (i = 0; i < meta_page->num_extents_; i++){
      if (meta_page->extent_used_page_[i] < BITMAP_SIZE){
        NewOpen = false;
        break;
      }
    }

```

```
    if(NewOpen){
      i = meta_page->num_extents_++;
      meta_page->extent_used_page_[i]++;
    }else{
      meta_page->extent_used_page_[i]++;
    }

    char Page_Data[PAGE_SIZE];
    ReadBitMapPage(i,Page_Data);

    BitmapPage<PAGE_SIZE> *Bitmap_page = reinterpret_cast<BitmapPage<PAGE_SIZE> *>(Page_Data);
    IsSuccess = Bitmap_page->AllocatePage(NextPage);

    if(IsSuccess){
      page_id_t BitMap_page_id=i*(BITMAP_SIZE+1)+1;
      char *Page_Data = reinterpret_cast<char *>(Bitmap_page);
      WritePhysicalPage(BitMap_page_id,Page_Data);
      NextPage += i*BITMAP_SIZE;
    }else{
      std::cout << "AllocatePage Failed  ----- at the middle!" << std::endl;
    }
  }
  return NextPage;
}
```

1.2.2 DeAllocatePage

此函数负责释放给定逻辑页的物理存储空间，并更新相应的元数据。 将 meta_data_ 转换为 DiskFileMetaPage* 类型的指针，以获取磁盘文件的元数据页。 通过 MapPageId 方法将逻辑页号转换为对应的物理页号。 检查元数据页中是否存在已分配的页，如果不存在，则直接返回，无需进行释放操作。 根据逻辑页号计算其所在的扩展（extent）ID，用于确定位图所在的页。 创建一个大小为 PAGE_SIZE 的字符数组 Init_Page_Data，并将其所有元素初始化为 0x00，用于将被释放的物理页清零。 减少已分配页数和扩展已使用页数。 读取位图所在的页，位图记录了每个逻辑页的分配状态。 使用位图页的 DeAllocatePage 方法释放指定逻辑页。 如果释放成功，则将位图页写回物理页，同时将被释放的物理页清零；如果失败，则直接将被释放的物理页清零并返回。

```
void DiskManager::DeAllocatePage(page_id_t logical_page_id) {
    DiskFileMetaPage *meta_page = reinterpret_cast<DiskFileMetaPage *>(this->meta_data_);

    page_id_t Physical_Page_Id = this->MapPageId(logical_page_id);
    if(!meta_page->GetExtentNums()) return ;
    bool IsSuccess = false;
    int extent_id=logical_page_id/BITMAP_SIZE;

    char Init_Page_Data[PAGE_SIZE];
    for (int i = 0; i < PAGE_SIZE; i++) {
        Init_Page_Data[i] = 0x00;
    }
    meta_page->num_allocated_pages_--;
    meta_page->extent_used_page_[extent_id]--;
    char Page_Data[PAGE_SIZE];
    ReadBitMapPage(extent_id,Page_Data);
```

```
    BitmapPage<PAGE_SIZE> *Bitmap_page = reinterpret_cast<BitmapPage<PAGE_SIZE> *>(Page_Data);

    IsSuccess = Bitmap_page->DeAllocatePage(logical_page_id%BITMAP_SIZE);
    if(IsSuccess){
        /*success*/
        page_id_t BitMap_page_id=extent_id*(BITMAP_SIZE+1)+1;
        char *Page_Data = reinterpret_cast<char *>(Bitmap_page);
        WritePhysicalPage(BitMap_page_id, Page_Data);
        /*cover the physical page*/
        WritePhysicalPage(Physical_Page_Id, Init_Page_Data);
    }else{
        /*fail*/
        WritePhysicalPage(Physical_Page_Id, Init_Page_Data);
        return;
    }
}
```

1.2.3 IsPageFree

此函数负责检查给定逻辑页是否空闲（未分配）。创建一个大小为 PAGE_SIZE 的字符数组 Page_Data，用于存储从位图页中读取的数据。 通过 ReadBitMapPage 方法读取包含逻辑页的位图所在的页，并将数据存储到 Page_Data 中。 将 Page_Data 解释为 BitmapPage<PAGE_SIZE>* 类型的指针，以便访问位图页面。 使用位图页的 IsPageFree 方法检查给定的逻辑页是否空闲。 返回检查结果。

```
bool DiskManager::IsPageFree(page_id_t logical_page_id) {
    char Page_Data[PAGE_SIZE];

    ReadBitMapPage(logical_page_id/BITMAP_SIZE, Page_Data);

    BitmapPage<PAGE_SIZE> * bitmap_page = reinterpret_cast<BitmapPage<PAGE_SIZE> *>(Page_Data);
    bool IsSuccess=bitmap_page->IsPageFree(logical_page_id%BITMAP_SIZE);
    return IsSuccess;
}
```

1.2.4 MapPageId

此函数负责将逻辑页号映射到对应的物理页号。将逻辑页号除以 BITMAP_SIZE（位图大小）并加上 2，并将其与逻辑页号相加，以得到物理页号。 返回计算得到的物理页号。

```
page_id_t DiskManager::MapPageId(page_id_t logical_page_id) {
    return logical_page_id/BITMAP_SIZE+2+logical_page_id;
}
```

1.3 LRU替换策略

- Victim 方法获取最近最少使用的页并将其移出缓存。
- Pin 方法将指定页从LRU缓存中移除。
- Unpin 方法将指定页添加到LRU缓存中，如果缓存已满，则移除最近最少使用的页。
- Size 方法获取当前LRU缓存中的页数。

1.3.1 Victim

使用 std::scoped_lock 锁定互斥锁 mutx_ 以保证线程安全。如果 LRU_list 为空，则返回 false。将 LRU_list 的最后一个元素（最近最少使用的页）赋值给 *frame_id。 从 LRU_hash 中移除该页。从 LRU_list 中移除该页。返回 true。

```
bool LRUReplacer::Victim(frame_id_t *frame_id) {
  std::scoped_lock lock{mutx_};
  if (LRU_list.empty()) {
    return false;
  }
  *frame_id = LRU_list.back();
  LRU_hash.erase(*frame_id);
  LRU_list.pop_back();
  return true;
}
```

1.3.2 Pin

此函数将指定页从LRU缓存中移除。使用 std::scoped_lock 锁定互斥锁 mutx_ 以保证线程安全。 检查 frame_id 是否在 LRU_hash 中。如果不在，直接返回。 获取 frame_id 在 LRU_list 中的位置迭代器。从 LRU_list 中移除该位置的页。从 LRU_hash 中移除该页。

```
void LRUReplacer::Pin(frame_id_t frame_id) {
  std::scoped_lock lock{mutx_};
  if (LRU_hash.count(frame_id) == 0) {
    return;
  }
  auto iter = LRU_hash[frame_id];
  LRU_list.erase(iter);
  LRU_hash.erase(frame_id);
}
```

1.3.3 Unpin

此函数将指定页添加到LRU缓存中，如果缓存已满，则移除最近最少使用的页。使用 std::scoped_lock 锁定互斥锁 mutx_ 以保证线程安全。检查 frame_id 是否已经在 LRU_hash 中。如果是，直接返回。 如果 LRU_list 的大小已达到 max_size，则移除 LRU_list 的第一个元素（最近最少使用的页）。将 frame_id 添加到 LRU_list 的前端。 将 frame_id 和其在 LRU_list 中的位置添加到 LRU_hash 中。

```
void LRUReplacer::Unpin(frame_id_t frame_id) {
  std::scoped_lock lock{mutx_};
  if (LRU_hash.count(frame_id) != 0) {
    return;
  }
  if (LRU_list.size() >= max_size) {
    frame_id_t need_del = LRU_list.front();
    LRU_list.pop_front();
    LRU_hash.erase(need_del);
  }
  LRU_list.push_front(frame_id);
  LRU_hash.emplace(frame_id, LRU_list.begin());
}
```

1.4 [BONUS] CLOCK替换策略实现

CLOCKReplacer 类实现了 CLOCK 替换策略，用于管理页的替换顺序。CLOCK 替换策略是一种近似于最近最少使用（LRU）策略的页替换算法，它使用一个环形队列和一个额外的位来模拟页的访问情况。当需要替换页时，CLOCK 替换策略会检查环形队列中的页面，如果页面的参考位为0，则选择该页面进行替换；否则，将参考位设置为0，并继续检查下一个页面。

- `bool Victim(frame_id_t *frame_id)`:选择一个牺牲页进行替换，并将其框架ID存储在 frame_id 指针所指向的位置。如果成功选择了牺牲页，则返回 true；否则，返回 false。
- `void Pin(frame_id_t frame_id)`:标记给定框架ID对应的页为固定状态。
- `void Unpin(frame_id_t frame_id)`:标记给定框架ID对应的页为非固定状态。
- `size_t Size()`:返回 CLOCK 替换器当前存储的页数。

1.4.1 Victim

该方法用于选择一个页进行替换。它遍历时钟队列中的页面，如果找到参考位为0的页面，则选择该页面进行替换，并将其框架ID存储在 frame_id 指针所指向的位置。如果未找到参考位为0的页面，则将所有页面的参考位设置为0，并将页面重新加入队列，直到找到一个可替换的页面或者队列为空。

使用互斥锁 std::mutex 进行加锁，以确保方法的线程安全性。首先检查时钟队列是否为空，如果为空，则返回 false。在一个循环中遍历时钟队列：

- 获取队列的第一个页面（队列头部）。
  - 如果页面的参考位为0，则选择该页面进行替换，将其框架ID存储在 frame_id 指针所指向的位置，并从时钟状态中移除该页面的信息。
  - 如果页面的参考位为1，则将其参考位设置为0，并将该页面移到队列的尾部，以模拟时钟指针的移动。
- 如果遍历完整个时钟队列都未找到参考位为0的页面，则返回 false，表示未能选择可替换的页面。如果成功选择了一个可替换的页面，则将其框架ID存储在 frame_id 指针所指向的位置，并返回 true。如果未能选择可替换的页面（时钟队列为空或所有页面的参考位均为1），则返回 false。

```
// 选择一个页进行替换
bool CLOCKReplacer::Victim(frame_id_t *frame_id) {
std::lock_guard<std::mutex> lock(mutx_);  // 加锁以保证线程安全
if (clock_queue.empty()) {  // 如果时钟列表为空，返回 false
return false;
}

while (!clock_queue.empty()){
frame_id_t current = clock_queue.front();  // 获取队列前面的元素
clock_queue.pop();

    if (!clock_status[current]) {
      // 如果参考位为 0，则选择该页面进行替换
      *frame_id = current;
      clock_status.erase(current);
      return true;
    } else {
      // 如果参考位为 1，则清除参考位并将页面移动到队列后面
      clock_status[current] = false;
```

```
      clock_queue.push(current);
    }
}
return false;
}
```

1.4.2 Pin

该方法用于固定一个页面，表示该页面不能被替换。当固定页面时，方法会从时钟队列中移除指定页面，并将其对应的参考位信息从时钟状态中移除。使用互斥锁 std::mutex 进行加锁，以确保方法的线程安全性。获取时钟队列的大小，以备后续遍历时钟队列使用。在一个循环中遍历时钟队列：

- 获取队列的第一个页面（队列头部）。
  - 如果当前页面的框架ID等于要固定的页面的框架ID，则将其对应的参考位信息从时钟状态中移除，并继续处理下一个页面。
  - 否则，将当前页面重新放回队列（即将其插入队列尾部），保持其在时钟队列中的位置。完成遍历后，时钟队列中不再包含要固定的页面，并且其对应的参考位信息已被移除。

```
// 固定一个页面，表示该页面不能被替换
void CLOCKReplacer::Pin(frame_id_t frame_id) {
  std::lock_guard<std::mutex> lock(mutx_);  // 加锁以保证线程安全
  int size = clock_queue.size();
  for (int i = 0; i < size; i++) {
    frame_id_t current = clock_queue.front();
    clock_queue.pop();
    if (current == frame_id) {
      clock_status.erase(frame_id);
      continue;
    }
    clock_queue.push(current);
  }
}
```

1.4.3 Unpin

该方法用于取消固定一个页面，表示该页面可以被替换。取消固定后，页面的参考位会被设置为1，表示页面可以参与替换。如果时钟队列的大小已经达到了最大容量，说明时钟队列已满，需要选择一个页面进行替换。
调用 Victim 方法选择一个页面进行替换，并将其框架ID存储在 to_delete_frame 中。检查时钟状态映射中是否存在要取消固定的页面：如果页面不在状态映射中，表示该页面之前未固定，需要将其添加到时钟队列和状态映射中，并将其参考位设置为1。如果页面在状态映射中，表示该页面之前已经固定过，只需将其参考位设置为1即可。

```
// 取消固定一个页面，表示该页面可以被替换
void CLOCKReplacer::Unpin(frame_id_t frame_id) {
  //std::lock_guard<std::mutex> lock(mutx_);  // 加锁以保证线程安全
  frame_id_t to_delete_frame ;
  if (clock_queue.size() >= capacity) {
    std::cout << "fuck you !" << std::endl;
    this->Victim(&to_delete_frame);
    std::cout << to_delete_frame << std::endl;
    // std::cout << "fuck you !" << std::endl;
  }
  if (clock_status.find(frame_id) == clock_status.end()) {
    // 如果页面不在状态映射中，将其添加到队列和状态映射中
    clock_queue.push(frame_id);
```

```
    clock_status[frame_id] = true;
  } else {
    // 如果页面在状态映射中，将参考位设置为 1
    clock_status[frame_id] = true;
  }
}
```

1.4.4 TEST

```
TEST(CLOCKReplacerTest, SampleTest) {
    CLOCKReplacer clock_replacer(7);

    // Scenario: unpin six elements, i.e. add them to the replacer.
    clock_replacer.Unpin(1);
    clock_replacer.Unpin(2);
    clock_replacer.Unpin(3);
    clock_replacer.Unpin(4);
    clock_replacer.Unpin(5);
    clock_replacer.Unpin(6);
    clock_replacer.Unpin(1);
    EXPECT_EQ(6, clock_replacer.Size());

    // Scenario: get three victims from the lru.
    int value;
    clock_replacer.Victim(&value);
    EXPECT_EQ(1, value);
    clock_replacer.Victim(&value);
    EXPECT_EQ(2, value);
    clock_replacer.Victim(&value);
    EXPECT_EQ(3, value);

    // Scenario: pin elements in the replacer.
    // Note that 3 has already been victimized, so pinning 3 should have no effect.
    clock_replacer.Pin(3);
    clock_replacer.Pin(4);
    EXPECT_EQ(2, clock_replacer.Size());

    // Scenario: unpin 4. We expect that the reference bit of 4 will be set to 1.
    clock_replacer.Unpin(4);

    // Scenario: continue looking for victims. We expect these victims.
    clock_replacer.Victim(&value);
    EXPECT_EQ(5, value);
    clock_replacer.Victim(&value);
    EXPECT_EQ(6, value);
    clock_replacer.Victim(&value);
    EXPECT_EQ(4, value);

    CLOCKReplacer clock_replacer_new(5);
    clock_replacer_new.Unpin(1);
    clock_replacer_new.Unpin(3);
    clock_replacer_new.Unpin(4);
    clock_replacer_new.Unpin(2);
    clock_replacer_new.Unpin(5);
```

```
    clock_replacer_new.Unpin(6);
    clock_replacer_new.Unpin(3);
    clock_replacer_new.Unpin(4);
    clock_replacer_new.Unpin(7);

    clock_replacer_new.Victim(&value);
    EXPECT_EQ(5, value);
    clock_replacer_new.Victim(&value);
    EXPECT_EQ(6, value);
    clock_replacer_new.Victim(&value);
    EXPECT_EQ(3, value);
}
```

1.5 缓冲池管理

- `BufferPoolManager` 类实现了一个缓冲池管理器，用于管理数据库页的加载、缓存和替换。该类提供了从磁盘加载页到内存缓冲池、将脏页写回磁盘、分配新页以及删除页的功能。
- `size_t pool_size_`: 缓冲池中页的数量。
- `Page *pages_`: 页的数组，表示缓冲池中的所有页。
- `DiskManager *disk_manager_`: 指向磁盘管理器的指针。
- `unordered_map<page_id_t, frame_id_t> page_table_`: 映射页ID到缓冲池中的帧ID。
- `Replacer *replacer_`: 用于找到未固定页进行替换的替换器。
- `list<frame_id_t> free_list_`: 用于找到空闲页的列表。
- `recursive_mutex latch_`: 用于保护共享数据结构的递归互斥锁。

1.5.1 FetchPage

FetchPage 方法用于在缓冲池中获取指定的页。如果页已存在于缓冲池中，则将其固定并返回。如果页不存在，则从空闲列表或替换器中找到一个替换页，将其写回磁盘（如果是脏页），然后读取请求页的数据并返回。 查找页表：在页表中查找请求的页。如果找到，则固定该页并返回。 找到替换页：如果页不在页表中，则从空闲列表中获取一个框架ID。如果空闲列表为空，则从替换器中获取一个框架ID。 处理脏页：如果找到的替换页是脏页，则将其写回磁盘，并重置其脏标志。 更新页表：从页表中删除替换页，并将请求页插入页表。更新元数据：重置替换页的内存，将其页ID设置为请求页ID，并从磁盘读取请求页的数据。固定请求页：固定请求页并返回其指针。

```
Page* BufferPoolManager::FetchPage(page_id_t page_id) {
  std::scoped_lock lock{latch_};
  // 1. 查找页表中的请求页
  auto search_page = page_table_.find(page_id);
  if (search_page != page_table_.end()) {
    // 1.1 如果找到，固定该页并返回
    frame_id_t frame_id = search_page->second;
    Page* page = &(pages_[frame_id]);
    replacer_->Pin(frame_id);
    page->pin_count_++;
    return page;
  } else {
    // 1.2 如果未找到，找到一个替换页
    frame_id_t frame_id = -1;
```

```
    // 优先从空闲列表中获取
    if (!free_list_.empty()) {
      frame_id = free_list_.front();
      free_list_.pop_front();
    } else if (!replacer_->Victim(&frame_id)) {
      return nullptr; // 如果替换器也没有可用页，返回nullptr
    }

    Page* page = &(pages_[frame_id]);

    // 2. 如果替换页是脏页，将其写回磁盘
    if (page->IsDirty()) {
      disk_manager_->WritePage(page->page_id_, page->data_);
      page->is_dirty_ = false;
    }

    // 3. 更新页表
    page_table_.erase(page->page_id_);
    page_table_.emplace(page_id, frame_id);

    // 4. 更新请求页的元数据
    page->ResetMemory();
    page->page_id_ = page_id;
    disk_manager_->ReadPage(page_id, page->data_);

    // 5. 固定请求页并返回
    replacer_->Pin(frame_id);
    page->pin_count_ = 1;
    return page;
  }
}
```

1.5.2 NewPage

NewPage 方法用于在缓冲池中创建一个新页。方法通过首先检查缓冲池中的可用页，找到一个可以被替换的页（如果需要），分配一个新的页ID，并返回新页的指针。调用 AllocatePage：确保调用 AllocatePage 方法分配一个新的页ID。检查固定状态：如果缓冲池中的所有页都被固定，返回 nullptr。找到替换页：从空闲列表或替换器中找到一个可以替换的页，优先从空闲列表中获取。更新元数据：更新替换页的元数据，将其页ID设置为新分配的页ID，并将其内容清零。更新页表：在页表中更新替换页的记录。固定新页：固定新页并返回其指针。

```
Page* BufferPoolManager::NewPage(page_id_t &page_id) {
  std::scoped_lock lock{latch_};
  frame_id_t frame_id = -1;

  // 找到替换页
  if (!free_list_.empty()) {
    frame_id = free_list_.front();
    free_list_.pop_front();
  } else if (!replacer_->Victim(&frame_id)) {
    return nullptr;
  }

  // 分配新页ID
  page_id = AllocatePage();
  Page* page = &(pages_[frame_id]);
```

```
  page->pin_count_ = 1;

  // 更新替换页的元数据
  if (page->IsDirty()) {
    disk_manager_->WritePage(page->page_id_, page->data_);
    page->is_dirty_ = false;
  }
  page_table_.erase(page->page_id_);
  page_table_.emplace(page_id, frame_id);
  page->ResetMemory();
  page->page_id_ = page_id;

  // 固定新页
  replacer_->Pin(frame_id);
  return page;
}
```

1.5.3 DeletePage

DeletePage方法的功能是删除指定的页。如果该页存在且未被固定（pin），则将其从页表中删除，重置其元数据并将其返回到空闲列表中。 使用 std::scoped_lock 锁定互斥锁 latch_ 以确保线程安全。检查页表中是否存在指定页。如果不存在，返回 true。如果存在，获取页的框架ID。 如果页的固定计数大于0，返回 false，表示页正在使用中，无法删除。调用 DeallocatePage 方法解除分配该页。如果页是脏页，则将其写回磁盘，并重置脏标志。从页表中删除该页，并将其框架ID重置为无效页ID。将页的元数据重置，并将其返回到空闲列表中。返回 true 表示删除成功。

```
bool BufferPoolManager::DeletePage(page_id_t page_id) {
  // 0.   Make sure you call DeallocatePage!
  // 1.   Search the page table for the requested page (P).
  // 1.   If P does not exist, return true.
  // 2.   If P exists, but has a non-zero pin-count, return false. Someone is using the page.
  // 3.   Otherwise, P can be deleted. Remove P from the page table, reset its metadata and return it to the free list.
  std::scoped_lock lock{latch_};
  if (page_table_.count(page_id) == 0) return true;
  frame_id_t frame_id = page_table_.find(page_id)->second;

  Page *page = &(pages_[frame_id]);
  if (page->pin_count_ > 0) return false;

  DeallocatePage(page_id);

  // Update page
  if (page->IsDirty()) {
    disk_manager_->WritePage(page->page_id_, page->data_);
    page->is_dirty_ = false;
  }
  page_table_.erase(page->page_id_);
  page_table_.emplace(INVALID_PAGE_ID, frame_id);
  page->ResetMemory();
  page->page_id_ = INVALID_PAGE_ID;

  ASSERT(page->page_id_ == INVALID_PAGE_ID, "FAILED DELETE!");
  free_list_.push_back(frame_id);
  return true;
```

```
}
```

1.5.4 UnpinPage

`UnpinPage`方法用于解固定（unpin）缓冲池中的指定页。如果页的固定计数减为0，则将其放入替换器中。如果页被标记为脏页，则设置其脏标志。使用std::scoped_lock 锁定互斥锁 latch_ 以确保线程安全。在页表中查找指定的页ID。如果页不在页表中，返回 false。 获取页的框架ID，并通过该框架ID获取页的指针。检查页的固定计数。如果固定计数已经是0，返回 false。将页的固定计数减1。如果固定计数减为0，调用替换器的 Unpin 方法。 如果标记为脏页，将页的脏标志设置为 true。返回 true 表示解固定成功。

```cpp
bool BufferPoolManager::UnpinPage(page_id_t page_id, bool is_dirty) {
  std::scoped_lock lock{latch_};
  auto search = page_table_.find(page_id);
  if (search == page_table_.end()) {
    return false;
  }
  frame_id_t frame_id = search->second;
  Page *page = &(pages_[frame_id]);
  if (page->pin_count_ == 0) {
    return false;
  }
  page->pin_count_--;
  if (page->pin_count_ == 0) {
    replacer_->Unpin(frame_id);
  }
  if (is_dirty) {
    page->is_dirty_ = true;
  }
  return true;
}
```

1.5.5 FlushPage

FlushPage 方法用于将指定页的内容刷新到磁盘上。如果指定的页在缓冲池中，并且是脏页（即已被修改），则将其内容写回到磁盘，并清除脏标志。如果指定的页不在缓冲池中，则返回 false。检查无效页ID：如果指定的页ID是无效页ID，直接返回 false。查找页表：在页表中查找指定页的记录。如果找到，继续执行步骤3。如果未找到，表示指定页不在缓冲池中，直接返回 false。刷新页到磁盘：将找到的页的内容写回磁盘。获取页对应的框架ID和指针。调用磁盘管理器的 WritePage 方法将页内容写回磁盘。将页的脏标志设置为 false。返回结果：返回 true 表示成功刷新页到磁盘。

```cpp
bool BufferPoolManager::FlushPage(page_id_t page_id) {
  std::scoped_lock lock{latch_};
  // 检查无效页ID
  if (page_id == INVALID_PAGE_ID) {
    return false;
  }
  // 在页表中查找指定页
  auto search = page_table_.find(page_id);
  if (search != page_table_.end()) {
    // 如果找到指定页
    frame_id_t frame_id = search->second;
    Page *page = &(pages_[frame_id]);
    // 将页内容写回磁盘
    disk_manager_->WritePage(page->page_id_, page->data_);
    // 清除脏标志
    page->is_dirty_ = false;
```

```
  } else {
    // 如果未找到指定页，返回 false
    return false;
  }
  // 返回成功刷新页到磁盘的结果
  return true;
}
```

2 CATALOG MANAGER

Catalog Manager 负责管理和维护数据库的所有模式信息，包括：

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

这些模式信息在被创建、修改和删除后还应被持久化到数据库文件中。此外，Catalog Manager还需要为上层的执行器Executor提供公共接口以供执行器获取目录信息并生成执行计划。

2.1 目录元信息

2.1.1 CatalogMeta

CatalogMeta 类负责管理数据库中表和索引的元数据。以下是其结构和关键方法的概述：

- table_meta_pages*：存储表 ID 和对应元数据页 ID 之间的映射关系的 map*。
- *index_meta_pages*：存储索引 ID 和对应元数据页 ID 之间的映射关系的 map。
  方法。
- `SerializeTo(char *buf) const`：将 `CatalogMeta` 对象序列化为字符缓冲区
- `DeserializeFrom(char *buf)`：将字符缓冲区反序列化为 `CatalogMeta` 对象。
- `GetSerializedSize() const`：计算序列化后的 `CatalogMeta` 对象的大小。
- `GetNextTableId() const`：返回下一个可用的表 ID。
- `GetNextIndexId() const`：返回下一个可用的索引 ID。
- `NewInstance()`：创建 `CatalogMeta` 的新实例。
- `GetTableMetaPages()`：返回指向存储表元数据页面的映射的指针（用于测试）。
- `GetIndexMetaPages()`：返回指向存储索引元数据页面的映射的指针（用于测试）。
- `DeleteIndexMetaPage(BufferPoolManager *bpm, index_id_t index_id)`：删除与给定索引 ID 相关联的元数据页。

DeleteIndexMetaPage(BufferPoolManager *bpm, index_id_t index_id)：删除与给定索引 ID 相关联的元数据页。该类与 CatalogManager 密切相关，CatalogManager 使用 CatalogMeta 来管理数据库系统中表和索引的元数据。它提供了维护数据库系统中表和索引目录的必要功能。

GetSerializedSize() 方法用于计算 CatalogMeta 对象序列化后的大小。根据给出的计算方法，该方法返回的值为：

- 一个固定大小的魔术数字（uint32_t，4 个字节）。两个 std::map 对象的大小：每个 std::map 包括键值对，每个键值对需要 8 个字节（4 个字节的键和 4 个字节的值）。因此，加上魔术数字，总共为 12 个字节。然后将每个 std::map 中的键值对数量相加，乘以 8，得到的值表示这些键值对的总大小。这样就得到了 CatalogMeta 对象序列化后的总大小。

```
uint32_t CatalogMeta::GetSerializedSize() const {
  //magic_num + size * 2 + map(4, 4)*2
  return 12 +(table_meta_pages_.size() + index_meta_pages_.size()) * 8;
}
```

2.1.2 IndexMetadata

IndexMetadata 类用于表示索引的元数据，包括索引的 ID、名称、所属表的 ID 以及索引键的映射。以下是该类的关键结构和方法：

- index_id：*索引的唯一标识符。index_name*：索引的名称。
- table_i：*该索引所属表的唯一标识符。*
- *key_map*：索引键与元组键的映射。
  方法：。
- `Create()`：静态方法，用于创建新的 `IndexMetadata` 实例。

  `SerializeTo(char *buf) const`：将 `IndexMetadata` 对象序列化为字符缓冲区。

  `GetSerializedSize() const`：计算序列化后的 `IndexMetadata` 对象的大小。

  `DeserializeFrom(char *buf, IndexMetadata *&index_meta)`：从字符缓冲区反序列化出一个 `IndexMetadata` 对象。

  `GetIndexName() const`：返回索引的名称。

  `GetTableId() const`：返回索引所属表的唯一标识符。

  `GetIndexColumnCount() const`：返回索引的列数。

  `GetKeyMapping() const`：返回索引键与元组键的映射。

  `GetIndexId() const`：返回索引的唯一标识符。

该类主要用于管理数据库中索引的元数据信息，为索引的创建、序列化和反序列化提供了必要的功能。

GetSerializedSize() 方法用于计算 IndexMetadata 对象序列化后的大小。根据给出的计算方法，该方法返回的值为索引名称长度加上键映射大小以及其他固定大小的总和。具体计算如下：

每个键映射使用 4 个字节（假设 uint32_t 类型）。索引名称的长度由 index_name_.length() 给出。加上其他固定大小的部分，其中包括键映射大小（4 个字节）、索引 ID（2 个字节）、表 ID（2 个字节）和魔术数字（2 个字节）。因此，将以上各项大小相加即可得到 IndexMetadata 对象序列化后的总大小。

```
uint32_t IndexMetadata::GetSerializedSize() const {
if (!index_name_.size()) return 0;
//key_map_ 4 and 2 id 2 size magic_num
return 4 * (key_map_.size() + 5) + index_name_.length();
}
```

2.1.3 IndexInfo

`Init` 方法用于初始化 `IndexInfo` 对象。具体步骤如下：

1. **初始化元数据和表信息**：将传入的元数据和表信息指针赋值给类成员变量。
2. **映射索引键到键模式**：使用 `Schema::ShallowCopySchema` 方法根据表的模式和索引键的映射关系创建索引键模式。
3. **创建索引**：调用 `CreateIndex` 方法创建索引。`CreateIndex` 方法需要根据索引类型（例如 "bptree"）和缓冲池管理器来创建索引对象。

```
  void Init(IndexMetadata *meta_data, TableInfo *table_info, BufferPoolManager
*buffer_pool_manager) {
    // Step1: init index metadata and table info
    meta_data_ = meta_data;
    table_info_ = table_info;
    // Step2: mapping index key to key schema
    key_schema_ = Schema::ShallowCopySchema(table_info_->GetSchema(), meta_data_-
>GetKeyMapping());
    // Step3: call CreateIndex to create the index
    index_ = CreateIndex(buffer_pool_manager,"bptree");
    //ASSERT(false, "Not Implemented yet.");
  }
```

`IndexInfo` 类的主要职责是：初始化索引元数据。映射索引键到键模式。创建索引。提供一些索引相关的信息访问方法。

2.1.4 TableMetadata

TableMetadata 类用于存储表的元数据，包括表的ID、名称、根页面ID以及表的模式（通过指向 Schema 对象的指针来表示）。以下是对该类的分析：

- table_id：*表的唯一标识符，类型为 table_id_t。*
- *table_name*：表的名称，类型为 std::string。
- root_page_id：*表的根页面ID，用于指示存储表数据的位置，类型为 page_id_t。*
- *schema*：指向 Schema 对象的指针，表示表的模式。
- `Create`：用于创建新的 `TableMetadata` 对象，需要提供表的ID、名称、根页面ID以及表的模式。
- `DeserializeFrom`：用于从缓冲区中反序列化数据，创建 `TableMetadata` 对象。该方法通常用于从磁盘中读取表的元数据并恢复对象。
- `SerializeTo`：将对象序列化为字符数组。该方法通常用于将表的元数据写入磁盘。
- `GetSerializedSize`：计算对象序列化后的大小，包括固定大小的头部信息和动态大小的成员变量（如表名的长度和模式的大小）。
- `GetTableId`：获取表的ID。
- `GetTableName`：获取表的名称。
- `GetFirstPageId`：获取表的根页面ID。
- `GetSchema`：获取表的模式对象。

使用了友元类 TableInfo，允许 TableInfo 类访问 TableMetadata 类的私有成员。类中包含了一个固定的魔术数字，用于标识序列化的对象类型。由于可能需要在对象销毁时释放 Schema 对象的内存，因此在实际使用中可能需要实现析构函数，但在提供的代码中被注释掉了。总体而言，TableMetadata 类提供了对表元数据的封装和管理，为数据库系统的元数据管理提供了基本的功能支持。

TableMetadata 类中的 GetSerializedSize() 方法用于计算序列化后对象的大小。根据该方法的实现和类的成员变量，计算得到的大小如下：

- 一个固定大小的魔术数字（uint32_t，4 个字节）。一个 table_id_t 类型的表ID（4 个字节）。一个字符串类型的表名，其大小等于字符串长度加上一个结尾的 null 字符（字符串长度 + 1）。一个 page_id_t 类型的根页面ID（4 个字节）。一个指向 Schema 对象的指针（通常是指针的大小，例如 4 或 8 个字节，取决于操作系统和编译器）。根据上述计算，将这些部分的大小相加即可得到 TableMetadata 对象序列化后的总大小。

```
uint32_t TableMetadata::GetSerializedSize() const {
    // 4 uint32
    return  16 + schema_->GetSerializedSize() + table_name_.length();
}
```

2.2 表和索引的管理

2.2.1 CreateTable

功能描述：创建新表，分配所需资源，并将新表的元数据信息序列化到磁盘。检查表是否已存在。创建新的表信息实例、表模式的深拷贝、表堆、表元数据。将表元数据序列化到新页并刷新到磁盘。更新表名映射、表信息、表元数据页映射。返回成功或失败。

```
dberr_t CatalogManager::CreateTable(const string &table_name, TableSchema *schema,
                                    Txn *txn, TableInfo *&table_info) {
  // 已经存在同名表
  if (table_names_.count(table_name) > 0) {
    return DB_TABLE_ALREADY_EXIST;  // 返回表已存在错误
  }
  // 创建新的表
  table_info = TableInfo::Create();  // 创建表信息实例
  Schema *newschema = Schema::DeepCopySchema(schema);  // 深拷贝表模式
  TableHeap *table_heap = TableHeap::Create(buffer_pool_manager_, newschema, nullptr,
                                            log_manager_, lock_manager_);  // 创建表堆

  TableMetadata *table_meta = TableMetadata::Create(++next_table_id_, table_name,
                                                    table_heap->GetFirstPageId(),
newschema);  // 创建表元数据
  table_info->Init(table_meta, table_heap);  // 初始化表信息

  // 序列化
  page_id_t page_id;
  Page *tablePage = buffer_pool_manager_->NewPage(page_id);  // 分配新页面
  ASSERT(page_id != INVALID_PAGE_ID && tablePage != nullptr, "unable to allocate
page");  // 分配页面失败检查
  table_meta->SerializeTo(tablePage->GetData());  // 序列化表元数据到页面

  table_names_.insert(pair<string, table_id_t>(table_name, table_info->GetTableId()));
 // 更新表名映射
  tables_.insert(pair<table_id_t, TableInfo *>(next_table_id_, table_info));  // 更新
表信息
  catalog_meta_->table_meta_pages_.insert(pair<table_id_t, page_id_t>(next_table_id_,
page_id));  // 更新表元数据页面映射
```

```
26
27    buffer_pool_manager_->UnpinPage(page_id, true);  // 解锁页面
28    buffer_pool_manager_->FlushPage(page_id);  // 刷新页面
29    FlushCatalogMetaPage();  // 刷新目录元数据页面
30    return DB_SUCCESS;  // 返回成功
31  }
```

2.2.2 GetTable

获取指定名称的表信息。根据表名查找表名映射，获取表的 ID。使用表的 ID 查找表信息，并存储到输出参数中。返回成功或失败。

```
1  dberr_t CatalogManager::GetTable(const string &table_name, TableInfo *&table_info) {
2    auto itr = table_names_.find(table_name);
3    if(itr == table_names_.end()) {
4      return DB_TABLE_NOT_EXIST;
5    }
6    ASSERT(tables_.count(itr->second) > 0, "Name is found while data can't be found");
7    table_info = tables_[itr->second];// 对应 id_t 存在 table
8    return DB_SUCCESS;
9  }
```

2.2.3 GetTables

获取所有表的信息。遍历表名映射，根据表的 ID 获取表信息，并存储到输出参数中。返回成功或失败。

```
1  dberr_t CatalogManager::GetTables(vector<TableInfo *> &tables) const {
2    if (tables_.empty()) return DB_FAILED;
3    for(auto itr = table_names_.begin(); itr != table_names_.end(); itr++) {
4      tables.push_back(tables_.find(itr->second)->second);
5    }
6    return DB_SUCCESS;
7  }
```

2.2.4 CreateIndex

为指定表创建索引。检查表是否存在，检查索引名称是否已被占用。获取表的 ID 和信息，初始化键映射。创建索引元数据、序列化到新页并刷新到磁盘。初始化索引信息，并更新索引名称映射、索引信息、索引元数据页映射。返回成功或失败。

```
1  dberr_t CatalogManager::CreateIndex(const std::string &table_name, const string
   &index_name,
2                                      const std::vector<std::string> &index_keys, Txn
   *txn,
3                                      IndexInfo *&index_info, const string &index_type)
   {
4    // 检查表是否存在
5    if (table_names_.count(table_name) == 0) {
```

```
    return DB_TABLE_NOT_EXIST; // 表不存在，返回错误码
  }

  // 获取表的 ID 和信息
  table_id_t table_id = table_names_.find(table_name)->second;
  TableInfo *table_info = nullptr;
  GetTable(table_id, table_info);

  ASSERT(table_info != nullptr, "Get Table FAILED");

  // 初始化键映射
  vector<uint32_t> key_map;
  uint32_t key_index;
  table_info->GetSchema();

  // 根据索引键名获取键索引
  for (auto it = index_keys.begin(); it != index_keys.end(); it++) {
    if (table_info->GetSchema()->GetColumnIndex(*it, key_index) ==
DB_COLUMN_NAME_NOT_EXIST) { // 未找到在键中的列
      return DB_COLUMN_NAME_NOT_EXIST; // 返回错误码
    }
    key_map.push_back(key_index);
  }

  // 获取新的索引 ID
  next_index_id_++;

  // 检查索引是否已存在
  if (index_names_.count(table_name) > 0) {
    unordered_map<string, index_id_t> map_index_id = index_names_[table_name];
    if (map_index_id.count(index_name) > 0) {
      return DB_INDEX_ALREADY_EXIST; // 索引已存在，返回错误码
    }
    index_names_[table_name].insert(pair<string, index_id_t>(index_name,
next_index_id_));
  } else {
    // 如果该表尚未存在索引，则创建一个新的索引映射
    unordered_map<string, index_id_t> map_index_Id;
    map_index_Id.insert(pair<string, index_id_t>(index_name, next_index_id_));
    index_names_.insert(pair<string, unordered_map<string, index_id_t>>(table_name,
map_index_Id));
  }

  // 处理索引元数据和索引信息
  IndexMetadata *index_meta = IndexMetadata::Create(next_index_id_, index_name,
table_id, key_map);
  page_id_t page_id;
  Page *page = buffer_pool_manager_->NewPage(page_id);
  ASSERT(page != nullptr, "Not able to allocate new page");
  index_meta->SerializeTo(page->GetData());
  catalog_meta_->index_meta_pages_.insert(pair<index_id_t, page_id_t>(next_index_id_,
page_id));

  // 初始化索引信息并插入索引集合
  index_info = IndexInfo::Create();
```

```
  index_info->Init(index_meta, table_info, buffer_pool_manager_);
  indexes_.insert(pair<index_id_t, IndexInfo *>(next_index_id_, index_info));
  buffer_pool_manager_->UnpinPage(page_id, true);
  buffer_pool_manager_->FlushPage(page_id);
  FlushCatalogMetaPage();
  return DB_SUCCESS; // 创建索引成功，返回成功码
}
```

2.2.5 GetIndex

获取指定表的指定索引的信息。检查表是否存在，检查索引是否存在。根据表名和索引名获取索引 ID。使用索引 ID 查找索引信息，并存储到输出参数中。返回成功或失败。

```
dberr_t CatalogManager::GetIndex(const std::string &table_name, const std::string &index_name,
                                 IndexInfo *&index_info) const {
  // 检查表是否存在
  if (index_names_.count(table_name) == 0) return DB_TABLE_NOT_EXIST; // 表不存在，返回错误码
  // 检查索引是否存在
  if (index_names_.find(table_name)->second.count(index_name) == 0) return DB_INDEX_NOT_FOUND; // 索引不存在，返回错误码
  // 获取索引的 ID
  index_id_t index_id = index_names_.find(table_name)->second.find(index_name)->second;
  // 检查索引是否存在
  if (indexes_.count(index_id) == 0) return DB_FAILED; // 索引不存在，返回失败码
  // 获取索引信息并返回成功
  index_info = indexes_.find(index_id)->second;
  return DB_SUCCESS; // 获取索引成功，返回成功码
}
```

2.2.6 GetTableIndexes

获取指定表的所有索引信息。检查表是否存在。根据表名获取该表的所有索引 ID。遍历索引 ID，根据索引 ID 获取索引信息，并存储到输出参数中。返回成功或失败。

```
dberr_t CatalogManager::GetTableIndexes(const std::string &table_name, std::vector<IndexInfo *> &indexes) const {

  // 检查表是否存在
  if (index_names_.count(table_name) == 0) return DB_TABLE_NOT_EXIST; // 表不存在，返回错误码
  //获取该表的所有索引  遍历索引映射并添加到索引向量中
  for (auto iter: index_names_.find(table_name)->second) {
    if (indexes_.count(iter.second) == 0) {
      return DB_FAILED; // 如果索引不存在，返回失败码
    }
    indexes.push_back(indexes_.find(iter.second)->second); // 添加索引到索引向量中
  }

  return DB_SUCCESS; // 获取索引成功，返回成功码
}
```

2.2.7 DropTable

删除指定表及其关联的所有索引。检查表是否存在。获取表的 ID 和信息。获取表中的所有索引，逐个删除。删除表的元数据页。更新表名映射、索引名称映射，刷新目录元数据页。返回成功或失败。

```
dberr_t CatalogManager::DropTable(const string &table_name) {

  // 检查表是否存在
  if (table_names_.count(table_name) == 0) return DB_TABLE_NOT_EXIST; // 表不存在，返回错误码

  // 获取表的ID和信息
  table_id_t table_id = table_names_.find(table_name)->second;
  TableInfo *table_info = tables_.find(table_id)->second;
  ASSERT(table_info != nullptr, "Table info not found"); // 确保表信息存在
  tables_.erase(table_id); // 从表信息映射中移除该表

  // 删除这个表中的所有索引
  std::vector<IndexInfo *> indexes_to_delete;
  GetTableIndexes(table_name, indexes_to_delete); // 获取表中的所有索引

  // 遍历并删除每个索引
  for (size_t i = 0; i < indexes_to_delete.size(); i++) {
    DropIndex(table_name, indexes_to_delete[i]->GetIndexName());
  }

  // 删除表的元数据页
  buffer_pool_manager_->DeletePage(catalog_meta_->table_meta_pages_.find(table_id)->second);
  catalog_meta_->table_meta_pages_.erase(table_id);

  // 注意：由于 DropIndex 使用 table_names_，因此在删除索引之后再移除这些操作
  index_names_.erase(table_name); // 从索引名称映射中移除该表
  table_names_.erase(table_name); // 从表名称映射中移除该表

  // 刷新目录元数据页
  FlushCatalogMetaPage();
  return DB_SUCCESS; // 表删除成功，返回成功码
}
```

2.2.8 DropIndex

删除指定表的指定索引。检查表和索引是否存在。获取索引 ID 和信息。销毁索引，删除索引的元数据页。更新索引名称映射，刷新目录元数据页。返回成功或失败。

```
dberr_t CatalogManager::DropIndex(const string &table_name, const string &index_name)
{
```

```
  // 检查指定的表是否存在
  if (index_names_.count(table_name) == 0) return DB_TABLE_NOT_EXIST;

  // 检查指定的索引是否存在于该表中
  if ((index_names_.find(table_name)->second).count(index_name) == 0) {
    return DB_INDEX_NOT_FOUND;
  }

  // 获取索引 ID
  index_id_t index_id = (index_names_.find(table_name)->second).find(index_name)->second;
  if (indexes_.count(index_id) == 0) return DB_FAILED;
  // 获取索引信息
  IndexInfo *index_info = indexes_.find(index_id)->second;
  // 从索引集合中移除该索引
  indexes_.erase(index_id);
  // 销毁索引（此处被注释掉）
  if(index_info->GetIndex()->Destroy() == DB_FAILED) return DB_FAILED;
  // 删除缓冲池中的页面，并从 catalog_meta_ 中移除该索引的元数据
  if(catalog_meta_->index_meta_pages_.count(index_id) == 0) return DB_FAILED;
  if(!(buffer_pool_manager_->DeletePage(catalog_meta_->index_meta_pages_.find(index_id)->second))) return DB_FAILED;
  catalog_meta_->index_meta_pages_.erase(index_id);

  // 从表的索引映射中移除该索引
  index_names_.find(table_name)->second.erase(index_name);

  // 刷新 catalog 元数据页
  FlushCatalogMetaPage();

  return DB_SUCCESS;
}
```

2.2.9 FlushCatalogMetaPage

将目录元数据页中的 catalog_meta_ 对象序列化到磁盘。序列化 catalog_meta_ 到缓冲池管理器获取的页数据中。取消页的固定，并将其标记为脏页。将页刷新到磁盘。返回成功或失败。

```
dberr_t CatalogManager::FlushCatalogMetaPage() const {
  // 将 catalog_meta_ 序列化到缓冲池管理器获取的页数据中
  catalog_meta_->SerializeTo(buffer_pool_manager_->FetchPage(CATALOG_META_PAGE_ID)->GetData());
  // 取消页的固定，并将其标记为脏页
  buffer_pool_manager_->UnpinPage(CATALOG_META_PAGE_ID, true);
  // 将页刷新到磁盘
  buffer_pool_manager_->FlushPage(CATALOG_META_PAGE_ID);
  return DB_SUCCESS;
}
```

2.2.10 LoadTable

从元数据页加载表信息。创建 TableInfo 对象。反序列化表元数据，并创建 TableHeap 对象。初始化 TableInfo 对象，并更新表名映射、表信息。返回成功或失败。

```
dberr_t CatalogManager::LoadTable(const table_id_t table_id, const page_id_t page_id)
{
  TableInfo *table_info = TableInfo::Create(); // 创建TableInfo对象
  // 反序列化
  TableMetadata *table_meta = nullptr;
  char *buf = buffer_pool_manager_->FetchPage(page_id)->GetData(); // 从缓冲区中获取页数据
  ASSERT(buf != nullptr, "Buffer not get"); // 确保缓冲区不为空
  TableMetadata::DeserializeFrom(buf, table_meta); // 反序列化得到TableMetadata对象
  ASSERT(table_meta != nullptr, "Unable to deserialize table_meta_data"); // 确保TableMetadata对象不为空
  buffer_pool_manager_->UnpinPage(CATALOG_META_PAGE_ID, false); // 释放页

  TableHeap *table_heap = TableHeap::Create(buffer_pool_manager_, table_meta->GetFirstPageId(), table_meta->GetSchema(), log_manager_, lock_manager_); // 创建TableHeap对象

  // 初始化table_info
  table_info->Init(table_meta, table_heap); // 初始化TableInfo对象
  table_names_.insert(pair<string, table_id_t>(table_info->GetTableName(), table_info->GetTableId())); // 将表名和表ID插入到table_names_中
  tables_.insert(pair<table_id_t, TableInfo *>(table_id, table_info)); // 将table_id和table_info插入到tables_中
  return DB_SUCCESS;
}
```

2.2.11 LoadIndex

从元数据页加载索引信息。参数：index_id 索引 ID，page_id 索引元数据页 ID。创建 IndexInfo 对象。反序列化索引元数据，并初始化 IndexInfo 对象。更新索引名称映射、索引信息。返回成功或失败。

```
dberr_t CatalogManager::LoadIndex(const index_id_t index_id, const page_id_t page_id)
{
  IndexInfo *index_info = IndexInfo::Create(); // 创建IndexInfo对象
  // Deserialize
  IndexMetadata *index_meta = nullptr;
  char *buf = buffer_pool_manager_->FetchPage(page_id)->GetData(); // 从缓冲区中获取页数据
  ASSERT(buf != nullptr, "Buffer not get"); // 确保缓冲区不为空
  IndexMetadata::DeserializeFrom(buf, index_meta); // 反序列化得到IndexMetadata对象
  ASSERT(index_meta != nullptr, "Unable to deserialize index_meta_data"); // 确保IndexMetadata对象不为空
  buffer_pool_manager_->UnpinPage(CATALOG_META_PAGE_ID, false); // 释放页

  // Initialize index_info
  index_info->Init(index_meta, tables_[index_meta->GetTableId()], buffer_pool_manager_); // 初始化IndexInfo对象
```

```
  indexes_.insert(pair<index_id_t, IndexInfo *>(index_id, index_info)); // 将index_info插入到indexes_中
  unordered_map<string, index_id_t> map_index_id;
  map_index_id.insert(pair<string, index_id_t>(index_info->GetIndexName(), index_id)); // 将索引名和索引ID插入到map_index_id中
  index_names_.insert(pair<string, unordered_map<string, index_id_t> >(index_info->GetTableInfo()->GetTableName(),
                                                                       map_index_id)); // 将表名和map_index_id插入到index_names_中
  return DB_SUCCESS;
}
```

2.2.12 GetTable

根据表 ID 获取表信息。根据表 ID 获取表信息，并存储到输出参数中。返回成功或失败。

```
dberr_t CatalogManager::GetTable(const table_id_t table_id, TableInfo *&table_info) {
  auto iter = tables_.find(table_id);
  if (iter == tables_.end()) return DB_TABLE_NOT_EXIST;
  table_info = iter->second;
  return DB_SUCCESS;
}
```

```
CatalogManager::CatalogManager(BufferPoolManager *buffer_pool_manager, LockManager *lock_manager,
                               LogManager *log_manager, bool init)
    : buffer_pool_manager_(buffer_pool_manager), lock_manager_(lock_manager), log_manager_(log_manager) {
  // 第一次生成数据库
  if (init) catalog_meta_ = CatalogMeta::NewInstance();
  else {
    // 读取 meta_page 的信息
    catalog_meta_ = CatalogMeta::DeserializeFrom(buffer_pool_manager_->FetchPage(CATALOG_META_PAGE_ID)->GetData());
    buffer_pool_manager_->UnpinPage(CATALOG_META_PAGE_ID, false);
    // 获取 table_meta
    for (auto iter: catalog_meta_->table_meta_pages_) {
      LoadTable(iter.first, iter.second);
    }
    // 获取 index_meta
    for (auto iter: catalog_meta_->index_meta_pages_) {
      LoadIndex(iter.first, iter.second);
    }
  }
  next_index_id_ = catalog_meta_->GetNextIndexId();
  next_table_id_ = catalog_meta_->GetNextTableId();
  FlushCatalogMetaPage();
}
```