浙江大學
ZHEJIANG UNIVERSITY

MiniSQL 设计报告

2024-2025 春夏学期 数据库系统

课程实验报告

姓名 王浩雄

学号 3230106032

年级 2023 级

专业 混合班（计算机科学与技术）

班级 混合 2303 班

2025 年 5 月 29 日

目录

1 系统设计说明 **4**
1.1 实验目的与实验需求 4
1.2 系统架构与模块概述 4
1.3 实验分工 5

2 模块设计说明 **6**
2.1 DISK AND BUFFER POOL MANAGER 6
2.1.1 模块功能概述 6
2.1.2 位图页 6
2.1.3 磁盘数据页管理 8
2.1.4 LRU 替换策略 10
2.1.5 缓冲池管理 12
2.2 RECORD MANAGER 16
2.2.1 模块功能概述 16
2.2.2 记录的序列化和反序列化 16
2.2.3 通过堆表管理记录 19
2.2.4 堆表迭代器 21
2.3 INDEX MANAGER 24
2.3.1 模块功能概述 24
2.3.2 B+ 树的创建与销毁 24
2.3.3 B+ 树的查找操作 26
2.3.4 B+ 树的插入操作 27
2.3.5 B+ 树的删除操作 31
2.3.6 索引迭代器 36
2.4 CATALOG MANAGER 37
2.4.1 模块功能概述 37
2.4.2 元数据持久化与加载 38
2.4.3 核心接口实现 39
2.5 PLANNER AND EXECUTOR 49
2.5.1 模块功能概述 49
2.5.2 数据库的创建、查看、切换与删除操作执行器 49
2.5.3 表的创建、查看与删除操作执行器 51
2.5.4 索引的创建、查看与删除操作执行器 54
2.5.5 SQL 批量执行操作执行器 57

3 系统测试与验收 **59**

3.1 正确性测试 59

3.2 功能性测试 60

3.2.1 测试点：数据库的创建、查看、切换与删除操作 60

3.2.2 测试点：表的创建、查看与删除操作 61

3.2.3 测试点：索引的创建、查看与删除操作 61

3.2.4 测试点：SQL 批量执行与数据插入操作 63

3.2.5 测试点：点查询、条件查询与投影操作 64

3.2.6 测试点：表数据的更新与删除操作 65

3.2.7 测试点：主键和唯一性约束测评 66

3.2.8 测试点：索引效果的定量测评 67

MiniSQL 设计报告

1 系统设计说明

1.1 实验目的与实验需求

本项目旨在设计并实现一个简化版的单用户 SQL 引擎——MiniSQL。

MiniSQL 支持通过命令行界面输入标准 SQL 语句，实现对数据库表的基本增、删、改、查操作。系统同时内建 B+ 树索引机制以提升数据访问效率，并支持多个数据库实例之间的切换与管理，模拟现实数据库系统中的常见使用场景。

项目在功能层面支持三种基本数据类型（integer、char(n)、float），最多 32 个属性的表结构定义，支持主键与唯一约束，并为相关字段自动建立 B+ 树索引。在数据操作方面，系统允许通过复合逻辑条件进行等值及区间查询，支持单条插入与批量删除等操作。

1.2 系统架构与模块概述

本项目采用模块化架构设计，主要包括六个核心组件：

Disk Manager：负责数据库文件的页级管理，实现数据页的分配、回收及读写操作。系统采用共享表空间的设计策略，将记录、索引和元数据统一存储于单一文件中，便于集中管理。

Buffer Pool Manager：实现对数据页的缓存机制，支持 LRU 页面替换算法与脏页管理，以提升磁盘 I/O 效率。缓存页的最小操作单元为固定大小的数据页（默认 4KB）。

Record Manager：负责记录的插入、删除与查找等基本操作，采用堆表（Table Heap）结构存储数据，提供迭代器作为访问接口，支持非定长记录的管理。

Index Manager：提供基于 B+ 树的索引支持，包括索引的创建、查找、插入与删除操作，使得数据库支持基于索引的等值与范围查询，显著优化数据访问性能。

Catalog Manager：管理数据库中的所有模式信息，包括表结构、字段属性、主键定义与索引关系，向执行器提供必要的元数据接口。

Planner 与 Executor：承担 SQL 执行的核心流程。计划生成器（Planner）负责根据语法树生成执行计划，并对语义进行校验；执行器（Executor）执行计划树中的各个节点，完成实际的数据操作。

在系统架构中，SQL 语句首先由解释器（Parser）进行语法解析，生成对应的语法树。该语法树随后由执行计划生成器（Planner）转换为逻辑执行计划，并交由执行器（Executor）执行。执行器在运行过程中，根据执行计划的具体需求调用 Record Manager、Index Manager 和 Catalog Manager，完成数据操作与元信息访问。

当上述模块涉及数据读写操作时，将通过 Buffer Pool Manager 实现对数据页的缓存管理，并进一步由其调用 Disk Manager 完成底层磁盘的实际读写。整个系统执行流程遵循分层解耦、模块协同的设计理念，各组件通过标准化接口紧密配合，显著提升了系统的结构清晰度、可维护性与可扩展性。

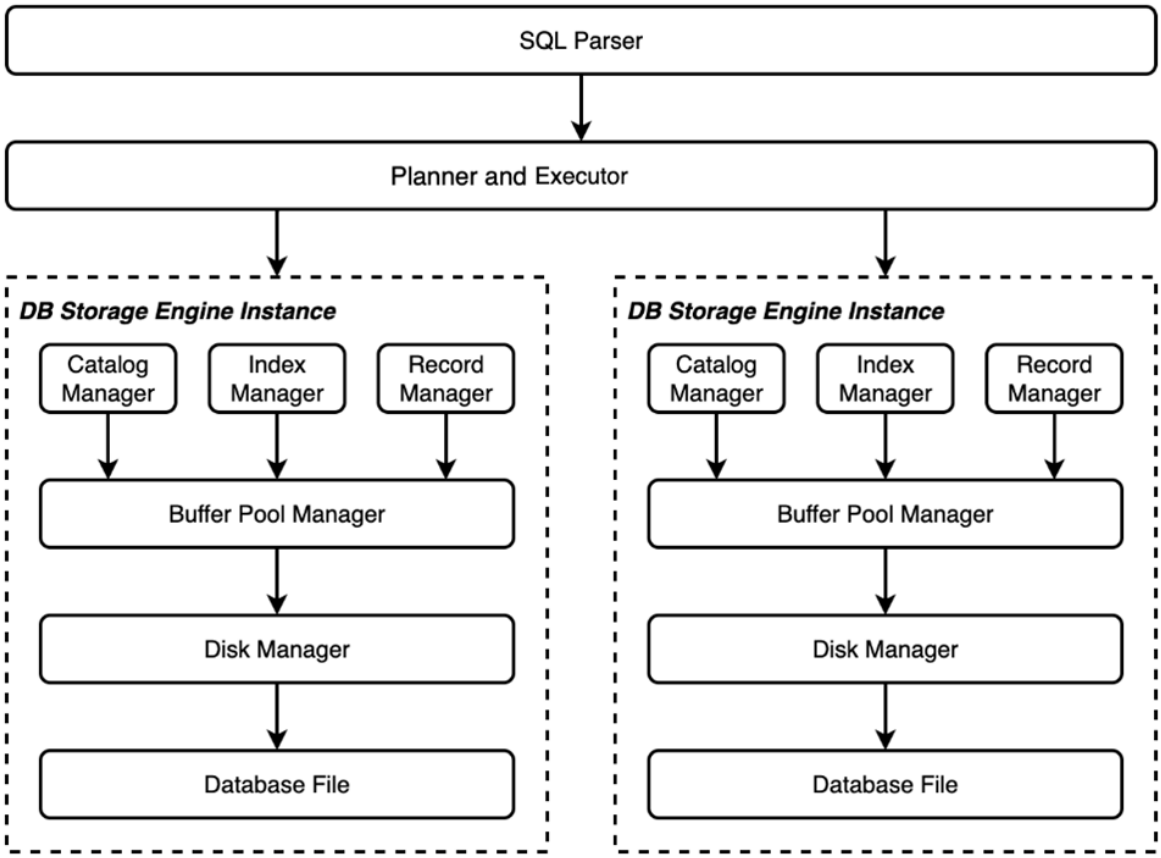

图 1: MiniSQL 系统架构与模块概述

1.3 实验分工

本实验由王浩雄单人完成。

2 模块设计说明

2.1 DISK AND BUFFER POOL MANAGER

2.1.1 模块功能概述

Disk Manager 与 Buffer Pool Manager 位于架构的最底层。其中，Disk Manager 负责数据库文件的页级管理，实现数据页的分配、回收及读写操作。Buffer Pool Manager 实现对数据页的缓存机制，支持 LRU 页面替换算法与脏页管理，以提升磁盘 I/O 效率。

Buffer Pool Manager 对系统中的其他模块是透明的。其他模块只需通过逻辑页号向 Buffer Pool Manager 请求所需的数据页，便可获取对应的页面内容。若该页面已缓存在内存中，Buffer Pool Manager 直接返回；否则，它将向 Disk Manager 发起请求。

Disk Manager 通过逻辑页号与物理页号的映射关系定位目标页面在磁盘文件中的物理位置，并将其读取到内存中，再由 Buffer Pool Manager 返还给上层模块。通过这一机制，系统屏蔽了底层存储细节，实现了模块化的页面访问与管理。

2.1.2 位图页

位图页是 Disk Manager 模块中的一部分，它使用位图标记一段连续页的分配情况，是实现磁盘页分配与回收工作的必要功能组件。位图页与数据页一样，占用 `PAGE_SIZE`（4KB）的空间。

成员变量：

每个位图页（Bitmap Page）包含三个成员变量：

1. `bytes`：一个字节数组，用于表示位图；在该位图中，每一位对应一个数据页。若某页已分配，则对应位为 1，未分配则为 0。
2. `page_allocated_`：记录当前已分配的数据页数量。
3. `next_free_page_`：标识下一可用空闲页的位置，用于加速空闲页的查找。

常量 `MAX_CHARS` 表示 `bytes` 数组的最大长度，即该位图页中可用于存储位图的字节数。其值等于一个页面的大小（`PAGE_SIZE`）减去用于存储位图页元数据的空间大小。

```
template <size_t PageSize>
class BitmapPage {
        uint32_t page_allocated_;
        uint32_t next_free_page_;
        unsigned char bytes[MAX_CHARS];
        static constexpr size_t MAX_CHARS = PageSize - 2 * sizeof(uint32_t);

};
```

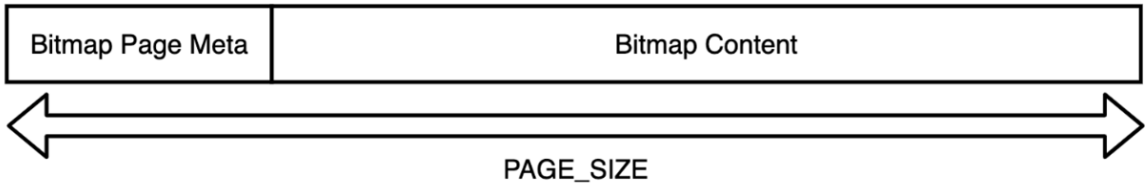

图 2: 位图页的结构

接口实现：

1. **页面分配**

   该函数需在位图中找到空闲的数据页位置，将新页面的位置以偏移量的形式通过参数 `page_offset` 进行返回。函数返回 true 代表分配成功。

```
template <size_t PageSize>
bool BitmapPage<PageSize>::AllocatePage(uint32_t &page_offset) {
        // 检查位图页是否已满
        if (page_allocated_ == MAX_CHARS * 8)
        return false;
        // 分配下一个页，更新位图页元数据和位图
        page_offset = next_free_page_;
        page_allocated_++;
        bytes[page_offset / 8 ] |= (1 << (page_offset % 8));
        // 寻找下一个页的位置，用于更新 next_free_page
        for (size_t i = 0; i < MAX_CHARS; i++) {
                if (bytes[i]==0xff) continue; // 若该字节对应的所有页都被分配，则跳过对
                    该字节的检查
                for (size_t j = 0; j < 8; j++) {
                        if (IsPageFreeLow(i,j)) {
                                next_free_page_ = i * 8 + j;
                                return true;
                        }
                }
        }
        return true;
}
```

2. **页面回收**

   该函数需释放参数 `page_offset` 指定的页面。函数返回 true 代表释放成功。

```
template <size_t PageSize>
bool BitmapPage<PageSize>::DeAllocatePage(uint32_t page_offset) {
        // 若该页未被分配则返回 false
        if (IsPageFree(page_offset)) {return false;}
        // 若位图页中无页被分配则返回 false
        if (page_allocated_ == 0) {return false;}
        // 更新 next_free_page_ 为当前页，是为了在位图页已满时，释放某个页后
            next_free_page 能正确设置
        next_free_page_ = page_offset;
        page_allocated_--;
        // 将当前页的分配状态设为 0
```

```
        bytes[page_offset / 8 ] ^= (1 << (page_offset % 8));
        return true;
}
```

3. **判断页面是否空闲**

   该函数负责判断参数 `page_offset` 指定的页面是否空闲，返回 true 代表空闲。

```
template <size_t PageSize>
bool BitmapPage<PageSize>::IsPageFree(uint32_t page_offset) const {
        return IsPageFreeLow(page_offset / 8, page_offset % 8);
}

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

2.1.3 磁盘数据页管理

由于单个位图页能够管理的数据页数量有限，为支持更大容量的数据管理，数据库文件被划分为若干个分区（Extent）。每个分区由一页位图页和若干连续的数据页组成，用于标记和管理该分区内数据页的分配状态。整个数据库文件的第 0 页为元信息页（Disk Meta Page），用于维护分区的全局信息，如分区数量、各分区已分配页数等。

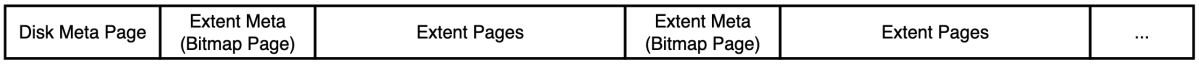

图 3: 元信息页与数据库文件分区的结构

需要注意的是，元信息页和各分区的位图页本身不存储用户数据，因此实际用于存储数据的页在物理上并不连续。为了对上层（如 Buffer Pool Manager）隐藏这一实现细节，系统引入了逻辑页号与物理页号的映射机制。对外统一使用逻辑页号，而物理页号则由 Disk Manager 在内部完成转换与定位，从而实现了逻辑上的页号连续性与物理上的灵活管理。

接口实现：

1. **页面分配**

   该函数从磁盘中分配一个空闲页，并返回空闲页的逻辑页号。在实现上，需要先找到可分配的 Extent，获取该 Extent 的位图页，再调用位图页的相关接口进行页面分配。

```
page_id_t DiskManager::AllocatePage() {
        // 读取元数据，寻找未放满的 Extent
        auto meta_data = reinterpret_cast<DiskFileMetaPage*>(meta_data_);
        size_t extent_num = 0;
        for (; extent_num < meta_data->num_extents_; extent_num++) {
```

```
                if (meta_data->extent_used_page_[extent_num] < BITMAP_SIZE) {break;}
        }
        // 若所有 Extent 都满，则新分配一个新的 Extent
        if (extent_num == meta_data->num_extents_) {
                meta_data->num_extents_++;
        }
        // 计算该 Extent 对应的位图页号，并获取该位图页
        size_t bitmap_physical_id = extent_num * (BITMAP_SIZE + 1) + 1;
        char bitmap_content[PAGE_SIZE];
        ReadPhysicalPage(bitmap_physical_id, bitmap_content);
        BitmapPage<PAGE_SIZE>* bitmap = reinterpret_cast<BitmapPage<PAGE_SIZE>*>(
            bitmap_content);
        // 尝试分配页
        uint32_t page_offset = 0;
        if (bitmap->AllocatePage(page_offset)) {
                // 分配成功
                auto meta_data = reinterpret_cast<DiskFileMetaPage*>(meta_data_);
                (meta_data->extent_used_page_)[extent_num]++;
                (meta_data->num_allocated_pages_)++;
                // 将位图页写回磁盘
                WritePhysicalPage(bitmap_physical_id, reinterpret_cast<const char *>(
                    bitmap));
        }
        else {
                // 分配失败
                LOG(ERROR) << "Failed to Allocate Page" << std::endl;
        }
        return extent_num * BITMAP_SIZE + page_offset;
}
```

2. **页面回收**

该函数释放磁盘中逻辑页号对应的物理页。在实现上，需要先找到待释放页面所在 Extent 的位图页，再调用位图页的相关接口进行页面回收。

```
void DiskManager::DeAllocatePage(page_id_t logical_page_id) {
        // 获取该页对应的位图页
        char bitmap_content[PAGE_SIZE];
        ReadPhysicalPage(MapBitmapPageId(logical_page_id), bitmap_content);
        BitmapPage<PAGE_SIZE>* bitmap = reinterpret_cast<BitmapPage<PAGE_SIZE>*>(
            bitmap_content);
        if (bitmap->DeAllocatePage(logical_page_id % BITMAP_SIZE)) {
                // 回收成功
                auto meta_data = reinterpret_cast<DiskFileMetaPage*>(meta_data_);
                meta_data->extent_used_page_[logical_page_id / BITMAP_SIZE]--;
                (meta_data->num_allocated_pages_)--;
                // 将位图页写回磁盘
                WritePhysicalPage(MapBitmapPageId(logical_page_id), reinterpret_cast<
                    const char *>(bitmap));
        }
        else {
                // 回收失败
                LOG(ERROR) << "Failed to DeAllocate Page" << std::endl;
        }
}
```

3. **判断页面是否空闲**

   该函数判断逻辑页号对应的数据页是否空闲。在实现上，需要先找到对应页面所在 Extent 的位图页，再调用位图页的相关接口进行判断。

```
bool DiskManager::IsPageFree(page_id_t logical_page_id) {
        // 获取该页对应的位图页
        char bitmap_content[PAGE_SIZE];
        ReadPhysicalPage(MapBitmapPageId(logical_page_id), bitmap_content);
        BitmapPage<PAGE_SIZE>* bitmap = reinterpret_cast<BitmapPage<PAGE_SIZE>*>(
            bitmap_content);
        // 调用位图页的方法进行检验
        bool ans = bitmap->IsPageFree(logical_page_id % BITMAP_SIZE);
        // 不需要delete bitmap, 因为它不是通过new分配的
        return ans;
}
```

4. **逻辑页号到物理页号的映射**

   实现两个函数，分别将页面的逻辑页号映射到页面的物理页号和页面所在 Extent 的位图页的物理页号。

```
page_id_t DiskManager::MapPageId(page_id_t logical_page_id) {
        return 2 + logical_page_id / BITMAP_SIZE * (BITMAP_SIZE + 1) + logical_page_id %
             BITMAP_SIZE;
}

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

2.1.4 LRU 替换策略

在 Buffer Pool Manager 的设计中，Replacer 类负责管理页面的固定（Pin）与释放（Unpin）状态。当缓冲池已满时，Replacer 根据页面的固定情况决定哪些页面可以被替换出缓冲池。

其中，LRU_Replacer 是 Replacer 的一个具体实现类，采用最近最少使用（LRU）策略，优先替换最长时间未被访问且未被固定的页面。

确切地说，缓冲池中的每一个页面被称为一个帧（frame），使用帧号（frame id）标识。在设计中，磁盘上的任意页面都可以被加载到缓冲池中的任一帧。因此，不同的磁盘页面在不同时间可能会被映射到同一个帧中。在 Replacer 的接口设计中，也统一使用帧号而非页号来标识和操作缓冲池中的页面。

成员变量：

LRU_Replacer 包含一个 `vector<frame_id_t>` 类型的成员变量 `lru_list_`，其含义如下：

1. `lru_list_` 用于按访问时间顺序维护当前在缓冲池中未被固定（Unpinned）的帧；
2. 列表头部是最久未使用的帧，应该优先被替换；

3. 列表尾部是最近刚变为未固定（Unpinned）的帧；

4. 只有未固定（Unpinned）的帧才会出现在列表中，才能被替换；当帧被固定（Pin）时，其对应的帧号会从列表中移除。

```
class LRUReplacer : public Replacer {
        vector<frame_id_t> lru_list_;
};
```

接口实现：

1. **页面固定**

   将数据页固定，使之不能被 Replacer 替换，即从 `lru_list_` 中移除该数据页对应的页帧。该函数应当在一个数据页被 Buffer Pool Manager 固定时被调用。

```
void LRUReplacer::Pin(frame_id_t frame_id) {
        // 若 frame_id 在列表中，则将其从列表中移除
        auto it = find(lru_list_.begin(),lru_list_.end(),frame_id);
        if (it != lru_list_.end())
        lru_list_.erase(it);
}
```

2. **页面释放**

   将数据页解除固定，放入 `lru_list_` 中，使之可以在必要时被 Replacer 替换掉。该函数应当在一个数据页的引用计数变为 0 时被 Buffer Pool Manager 调用，使页帧对应的数据页能够在必要时被替换。

```
void LRUReplacer::Unpin(frame_id_t frame_id) {
        // 若 frame_id 不在列表中，则将其插入列尾
        auto it = find(lru_list_.begin(),lru_list_.end(),frame_id);
        if (it == lru_list_.end())
        lru_list_.push_back(frame_id);
}
```

3. **页面替换**

   替换最近最少被访问的页，将其页帧号存储在参数 `frame_id` 中输出并返回 true，如果当前没有可以替换的元素则返回 false。

```
bool LRUReplacer::Victim(frame_id_t *frame_id) {
        // 检查列表是否为空
        if (lru_list_.empty()) {return false;}
        // 需要替换的是列表的第一个元素，将其返回并从列表中移除
        *frame_id = lru_list_[0];
        lru_list_.erase(lru_list_.begin());
        return true;
}
```

2.1.5 缓冲池管理

成员变量：

BufferPoolManager 类包含六个成员变量:

```
class BufferPoolManager {
        private:
        size_t pool_size_; // 缓冲池大小
        Page *pages_; // 数据页指针，指向一块在Buffer Pool Manager构造时分配的内存，用于作为缓冲池存储数据页
        DiskManager *disk_manager_; // DiskManager指针
        unordered_map<page_id_t, frame_id_t> page_table_; // 页表，记录了从逻辑页号到帧号的映射关系
        Replacer *replacer_; // Replacer指针
        list<frame_id_t> free_list_; // 空闲页列表
};
```

接口实现：

1. **页面获取**页面获取的步骤如下:

   (a) 在页表中搜索待获取的页

      i. 若当前页已经在页表中，即已经被存储在缓冲池中，则直接返回该页的指针

      ii. 若当前页不在缓冲池中，则需要选择一个替换帧

         A. 若存在空闲页，则直接选择空闲页作为替换帧

         B. 若没有空闲则，则需要使用 Replacer 选择一个替换帧

   (b) 如果替换帧被标记为脏页，则需要将该页先写回磁盘

   (c) 将待获取的页读取到替换帧中，更新相关元数据，并返回当前页指针

```
Page *BufferPoolManager::FetchPage(page_id_t page_id) {
        frame_id_t frame_id ;
        // 查找该页是否在 Buffer 中
        if (page_table_.find(page_id) != page_table_.end()) {
                // 该页在 Buffer 中
                frame_id = page_table_[page_id];
        }
        else {
                // 该页不在 Buffer 中，查找是否存在空闲空间
                if (free_list_.empty()) {
                        // 不存在空闲空间，进行 LRU 替换
                        if (!replacer_->Victim(&frame_id)) {
                                // 没有可以替换的空间
                                LOG(ERROR) << "No Space Available" << std::endl;
                                return nullptr;
                        }
                        // 有可以替换的空间
                        page_id_t page_to_be_victimed = pages_[frame_id].GetPageId();
                        if (pages_[frame_id].IsDirty()) {
                                // 脏块，需写回
```

```
                                disk_manager_->WritePage(page_to_be_victimed, pages_[
                                    frame_id].GetData());
                        }
                        page_table_.erase(page_to_be_victimed);
                }
                else {
                        // 存在空闲空间
                        frame_id = free_list_.back();
                        free_list_.pop_back();
                }
                // 更新 page_id 到 frame_id 的映射关系
                page_table_[page_id] = frame_id;
                // 更新页面的元数据
                pages_[frame_id].page_id_ = page_id;
                pages_[frame_id].pin_count_ = 0;

                // 从磁盘读入该页的存储数据
                disk_manager_->ReadPage(page_id, pages_[frame_id].GetData());
        }
        // Pin 该页，并返回页的地址
        replacer_->Pin(frame_id);
        pages_[frame_id].pin_count_++;
        return &pages_[frame_id];
}
```

2. **页面释放**

   当上层模块完成对页面的使用时，调用页面释放接口，并标记当前页在使用过程中是否被修改（页面释放变脏）。该函数将在页面完全被释放时将页面加入 Replacer 中，并相应地更新页面的 `is_dirty_` 属性。

```
bool BufferPoolManager::UnpinPage(page_id_t page_id, bool is_dirty) {
        // 若当前页不在页表中，则返回 false
        if (page_table_.find(page_id) == page_table_.end()) {
                return false;
        }
        frame_id_t frame_id = page_table_[page_id];
        if (pages_[frame_id].GetPinCount() != 0) pages_[frame_id].pin_count_--;
        // 若当前页已完全释放，则将其加入replacer中
        if (pages_[frame_id].GetPinCount() == 0) replacer_->Unpin(frame_id);
        // 若当前页已经被修改，则将其标记为脏页
        pages_[frame_id].is_dirty_ = is_dirty;
        return true;
}
```

3. **页面存储**

   强制将指定页面存储到磁盘。

```
bool BufferPoolManager::FlushPage(page_id_t page_id) {
        // 若当前页不在页表中，则返回 false
        if (page_table_.find(page_id) == page_table_.end()) {
                return false;
        }
        frame_id_t frame_id = page_table_[page_id];
```

```
        // 将当前页写回磁盘
        disk_manager_->WritePage(page_id, pages_[frame_id].GetData());
        return true;
}
```

4. **页面新建**

页面新建的步骤如下:

(a) 检查当前缓冲池是否已满，若已满返回空指针

(b) 调用 DiskManager 的 AllocatePage 接口分配一个数据页

(c) 需要选择一个替换帧，逻辑同获取页面接口

(d) 设置当前帧的元数据，并在页表中将当前帧指向之前分配的数据页

(e) 返回当前帧对应的数据页指针和逻辑页号

```
Page *BufferPoolManager::NewPage(page_id_t &page_id) {
        // 缓冲池已满时不能进行分配
        if (free_list_.empty() && replacer_->Size() == 0) {
                LOG(ERROR) << "No Space Available" << std::endl;
                return nullptr;
        }
        // 调用 AllocatePage 函数
        page_id = AllocatePage();
        frame_id_t frame_id;
        // 查找是否存在空闲空间以容纳该新页面
        if (free_list_.empty()) {
                // 不存在空闲空间，进行 LRU 替换
                if (!replacer_->Victim(&frame_id)) {
                        // 没有可以替换的空间
                        LOG(ERROR) << "No Space Available" << std::endl;
                        return nullptr;
                }
                // 有可以替换的空间
                page_id_t page_to_be_victimed = pages_[frame_id].GetPageId();
                if (pages_[frame_id].IsDirty()) {
                        // 脏块，需写回
                        disk_manager_->WritePage(page_to_be_victimed, pages_[frame_id].
                            GetData());
                }
                page_table_.erase(page_to_be_victimed);
        }
        else {
                // 存在空闲空间
                frame_id = free_list_.back();
                free_list_.pop_back();
        }
        // 更新 page_id 到 frame_id 的映射关系
        page_table_[page_id] = frame_id;
        // 更新页面的元数据，并重置页的内容
        pages_[frame_id].page_id_ = page_id;
        pages_[frame_id].pin_count_ = 0;
        pages_[frame_id].ResetMemory();
```

```
37          return &pages_[frame_id];
38  }
```

5. **页面删除**

   页面删除步骤如下:

   (a) 在硬盘上释放待删除页

   (b) 在页表中寻找待删除的页，若不在缓冲池中则直接返回

   (c) 若当前页仍在释放中，则停止释放

   (d) 将当前页从页表中删除，清空数据后将其加入到空闲页列表中

```
1   bool BufferPoolManager::DeletePage(page_id_t page_id) {
2           DeallocatePage(page_id);
3           // 查找该页是否在 Buffer 中
4           if (page_table_.find(page_id) == page_table_.end()) {
5                   // 该页不在 Buffer 中
6                   return true;
7           }
8           frame_id_t frame_id = page_table_[page_id];
9           if (pages_[frame_id].GetPinCount()) {
10                  // 其它进程正在使用该页
11                  return false;
12          }
13          // 更新 page_id 到 frame_id 的映射关系
14          page_table_.erase(page_id);
15          // 更新页面的元数据
16          pages_[frame_id].page_id_ = INVALID_PAGE_ID;
17          // 更新 free_list
18          free_list_.push_back(frame_id);
19          return true;
20  }
```

2.2 RECORD MANAGER

2.2.1 模块功能概述

Record Manager 负责管理数据库的所有记录，向上提供插入、删除、更新等接口。

在实现上，一张表由元数据（Table Metadata）与堆表（Table Heap）构成，分别保存表的模式与数据内容。其中，元数据中存储表的模式（Schema）信息，即各列（Column）的定义，以及表的名称、id 等信息，已由框架实现；而表中的实际数据行（Row）存储在堆表中。

堆表的实现与管理是本模块的核心。一个堆表由多个数据页（Table Page）组成，数据页之间通过双向指针相连，构成一个双向链表。如图所示，每个数据页中包含若干条序列化后的数据行，存储在不同的槽（Slot）中，每条记录通过唯一的行号（RowId，由所在页号与槽号拼接而成）进行标识。

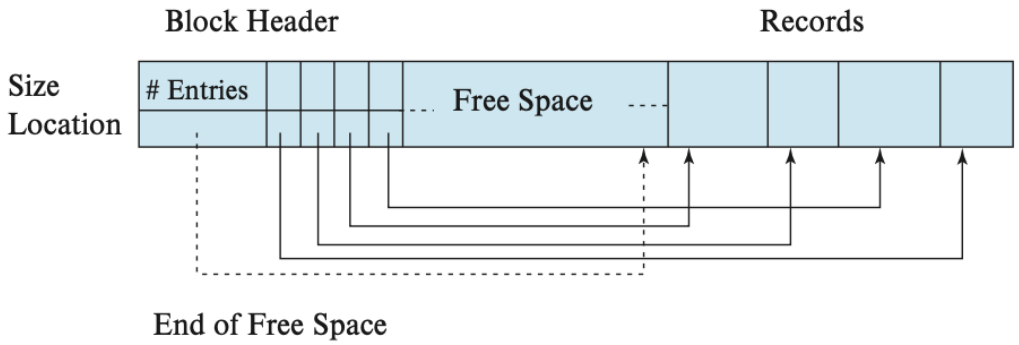

图 4: 堆表数据页的结构

堆表完成后，在其上封装一个迭代器（Table Iterator），支持对整张表的逐行遍历，便于上层模块处理数据记录。

2.2.2 记录的序列化和反序列化

与记录（Record）相关的概念有以下几个：

1. **列（Column）**：用于定义和表示数据表中的某一个字段，包含字段的字段名、字段类型、是否唯一等信息；
2. **模式（Schema）**：用于表示一个数据表或索引的结构，一个 Schema 由一个或多个 Column 构成；
3. **域（Field）**：对应于一条记录中某一个字段的数据信息，如存储数据的数据类型、是否为空、存储数据的值等信息；
4. **行（Row）**：用于存储记录或索引键，一个 Row 由一个或多个 Field 构成。

为满足数据库的持久化要求，所有的记录在存储时都需要通过序列化（Serialization），将具体的记录对象转换为二进制数据后，再存储到数据页中。在需要对记录进行操作时，再将其从数据页中反序列化（Deserialization）成对应的记录对象以供操作。

为了确保数据的正确存储与恢复，我们在对象的序列化中引入魔数 `MAGIC_NUM`，它在序列化时被写入到字节流的头部并在反序列化中被读出，以对反序列化时生成的对象进行校验。

接口实现（以 `Column` 对象为例）：

1. **序列化**

将当前记录序列化到缓冲区 `buf`。

```
uint32_t Column::SerializeTo(char *buf) const {
        // 记录偏移量
        uint32_t offset = 0;
        // 序列化魔数
```

```
        MACH_WRITE_UINT32(buf + offset, COLUMN_MAGIC_NUM);
        offset += sizeof(COLUMN_MAGIC_NUM);
        // 序列化列名
        MACH_WRITE_UINT32(buf + offset, name_.length());
        MACH_WRITE_STRING(buf + offset + sizeof(uint32_t), name_);
        offset += MACH_STR_SERIALIZED_SIZE(name_);
        // 序列化类型
        MACH_WRITE_TO(TypeId, buf + offset, type_);
        offset += sizeof(TypeId);
        // 序列化数据长度
        MACH_WRITE_UINT32(buf + offset, len_);
        offset += sizeof(uint32_t);
        // 序列化列号
        MACH_WRITE_UINT32(buf + offset, table_ind_);
        offset += sizeof(uint32_t);
        // 序列化 null 标记
        MACH_WRITE_TO(bool, buf + offset, nullable_);
        offset += sizeof(bool);
        // 序列化 unique 标记
        MACH_WRITE_TO(bool, buf + offset, unique_);
        offset += sizeof(bool);
        return offset;
}
```

2. **获取序列化字节数**

   获取当前记录序列化后的字节数。

```
uint32_t Column::GetSerializedSize() const {
        return sizeof(COLUMN_MAGIC_NUM) + MACH_STR_SERIALIZED_SIZE(name_) + sizeof(
            TypeId) + 2 * sizeof(uint32_t) + 2 * sizeof(bool);
}
```

3. **反序列化**

   将记录从缓冲区 buf 中反序列化到参数中提供的对象指针。

```
uint32_t Column::DeserializeFrom(char *buf, Column *&column) {
        // column 指针非空警告
        if (column != nullptr) {
                LOG(WARNING) << "Pointer to column is not null in column deserialize."
                    << std::endl;
        }
        // 记录偏移量
        uint32_t offset = 0;
        // 反序列化魔数，并对魔数进行检验
        uint32_t magic_num = MACH_READ_UINT32(buf);
        offset += sizeof(COLUMN_MAGIC_NUM);
        if (magic_num != COLUMN_MAGIC_NUM) {
                LOG(ERROR) << "Magic num incorrect." << std::endl;
        }
        // 反序列化列名
        uint32_t name_len = MACH_READ_FROM(uint32_t, buf + offset);
        offset += sizeof(uint32_t);
        std::string column_name(buf + offset, name_len);
        offset += name_len;
```

```
        // 反序列化类型
        TypeId type = MACH_READ_FROM(TypeId, buf + offset);
        offset += sizeof(TypeId);
        // 反序列化数据长度
        uint32_t len = MACH_READ_FROM(uint32_t, buf + offset);
        offset += sizeof(uint32_t);
        // 反序列化列号
        uint32_t table_ind = MACH_READ_UINT32(buf + offset);
        offset += sizeof(uint32_t);
        // 反序列化 null 标记
        bool nullable = MACH_READ_FROM(bool, buf + offset);
        offset += sizeof(bool);
        // 反序列化 unique 标记
        bool unique = MACH_READ_FROM(bool, buf + offset);
        offset += sizeof(bool);
        // 构造返回对象
        if (type == TypeId::kTypeChar) {
                column = new Column(column_name, type, len, table_ind, nullable, unique)
                    ;
        }
        else {
                column = new Column(column_name, type, table_ind, nullable, unique);
        }
        return offset;
}
```

2.2.3 通过堆表管理记录

成员变量：

TableHeap 类包含三个成员变量:

```
class TableHeap {
        private:
        BufferPoolManager *buffer_pool_manager_; // 缓冲池管理
        page_id_t first_page_id_; // 堆表首页的页号
        Schema *schema_; // 堆表元素模式
};
```

接口实现：

1. 插入行

   将行对象 row 插入堆表。在实现上，需从堆表的第一个页面开始，沿链表遍历所有页面，依次尝试插入。若遍历所有页面依然无法插入，则新建一个页面添加到链表末尾，对新页面进行初始化并维护链表，将待插入的行对象插入新页面中。

```
bool TableHeap::InsertTuple(Row &row, Txn *txn) {
        page_id_t page_id = first_page_id_;
        page_id_t last_page_id = INVALID_PAGE_ID;  // 记录最后一个页面的ID

        while (page_id != INVALID_PAGE_ID) {
                // 从第一个页面开始，沿链表遍历所有页面，依次尝试插入
                TablePage* page = reinterpret_cast<TablePage *>(buffer_pool_manager_->
                    FetchPage(page_id));
```

```
                // 如果该页面为空指针，则跳出循环
                if (page == nullptr) {break;}
                // 尝试插入元组到该页面
                if (page->InsertTuple(row, schema_, txn, lock_manager_, log_manager_)) {
                        // 插入成功，退出函数
                        buffer_pool_manager_->UnpinPage(page->GetTablePageId(), true);
                        return true;
                }
                // 插入失败，记录当前页面为可能的最后页面，尝试下一页面
                last_page_id = page_id;  // 记录当前页面ID
                page_id_t next_page_id = page->GetNextPageId();
                buffer_pool_manager_->UnpinPage(page->GetTablePageId(), false);
                page_id = next_page_id;
        }

        // 新建一个页面添加到链表末尾
        page_id_t new_page_id;
        auto new_page = reinterpret_cast<TablePage *>(buffer_pool_manager_->NewPage(
            new_page_id));
        if (new_page == nullptr) {return false;}

        // 对新页进行初始化
        new_page->Init(new_page_id, INVALID_PAGE_ID, log_manager_, txn);

        // 维护双向链表：将新页添加到末尾
        new_page->SetPrevPageId(last_page_id);
        new_page->SetNextPageId(INVALID_PAGE_ID);

        // 立即持久化新页面的链表信息
        buffer_pool_manager_->UnpinPage(new_page_id, true);

        // 更新原来最后页面的 next_page_id
        auto last_page = reinterpret_cast<TablePage *>(buffer_pool_manager_->FetchPage(
            last_page_id));
        if (last_page != nullptr) {
                last_page->SetNextPageId(new_page_id);     // 原最后页面指向新页面
                buffer_pool_manager_->UnpinPage(last_page_id, true);  // 立即持久化
        }

        // 重新获取新页面进行插入
        new_page = reinterpret_cast<TablePage *>(buffer_pool_manager_->FetchPage(
            new_page_id));
        if (new_page->InsertTuple(row, schema_, txn, lock_manager_, log_manager_)) {
                // 插入成功，退出函数
                buffer_pool_manager_->UnpinPage(new_page_id, true);
                return true;
        }
        buffer_pool_manager_->UnpinPage(new_page_id, false);
        return false;
}
```

2. **更新行**

   使用新的行对象 `row` 更新 RowId 为 `rid` 处的行数据。在实现上，若因原页面无法容纳更新后的新行，则将原行删除后重新插入新行的方式实现间接更新。

```cpp
bool TableHeap::UpdateTuple(Row &row, const RowId &rid, Txn *txn) {
        TablePage* page = reinterpret_cast<TablePage *>(buffer_pool_manager_->FetchPage(
            rid.GetPageId()));
        auto old_row = new Row(rid);
        // 尝试更新元组
        if (page->UpdateTuple(row, old_row, schema_, txn, lock_manager_, log_manager_))
            {
                // 更新成功
                buffer_pool_manager_->UnpinPage(page->GetTablePageId(), true);
                delete old_row;
                return true;
        }
        // 更新失败，从原来页先删除后再重新插入
        delete old_row;
        page->ApplyDelete(rid, txn, log_manager_);
        buffer_pool_manager_->UnpinPage(page->GetTablePageId(), true);
        return InsertTuple(row, txn);
}
```

3. **获取行**

   获取 RowId 为 `rid` 处的行数据，使用参数中提供的指针 `row` 返回获取的行数据。函数返回 true 代表获取成功。在实现上，先通过 RowId 获取该行所在的页，再调用数据页的相关方法获取指定的行数据。

```cpp
bool TableHeap::GetTuple(Row *row, Txn *txn) {
        page_id_t page_id = row->GetRowId().GetPageId();
        // 获取该元组所在页
        TablePage* page = reinterpret_cast<TablePage *>(buffer_pool_manager_->FetchPage(
            page_id));
        // 从数据页中获取当前行
        bool res = page->GetTuple(row, schema_, txn, lock_manager_);
        buffer_pool_manager_->UnpinPage(page_id, false);
        return res;
}
```

4. **执行删除行**

   对 RowIdrid 处的行数据执行删除。在实现上，先通过 RowId 获取该行所在的页，再调用数据页的相关方法执行删除，最后标注对该行所在的页取消固定并标注脏页。

```cpp
void TableHeap::ApplyDelete(const RowId &rid, Txn *txn) {
        TablePage* page = reinterpret_cast<TablePage *>(buffer_pool_manager_->FetchPage(
            rid.GetPageId()));
        page->ApplyDelete(rid, txn, log_manager_);
        buffer_pool_manager_->UnpinPage(page->GetTablePageId(), true);
}
```

2.2.4 堆表迭代器

成员变量：

TableIterator 类包含两个成员变量:

```
class TableIterator {
        TableHeap *table_heap_; // 堆表指针
        RowId cur_rid_; // 当前行RowId
};
```

接口实现：

1. 迭代器的运算符重载

```
bool TableIterator::operator==(const TableIterator &itr) const {
        return table_heap_ == itr.table_heap_ && rid_.Get() == itr.rid_.Get() && txn_ ==
             itr.txn_;
}

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

const Row &TableIterator::operator*() {
        Row* row = new Row(rid_);
        table_heap_->GetTuple(row, txn_);
        row->SetRowId(rid_);
        return *row;
}

Row *TableIterator::operator->() {
        Row* row = new Row(rid_);
        table_heap_->GetTuple(row, txn_);
        row->SetRowId(rid_);
        return row;
}

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

// ++iter
TableIterator &TableIterator::operator++() {
        rid_ = GetNextRId();
        return *this;
}

// iter++
TableIterator TableIterator::operator++(int) {
        RowId prev_rid = rid_;
        Txn *prev_txn = txn_;
        ++(*this);
        return TableIterator(table_heap_, prev_rid, prev_txn);
}
```

2. 获取堆表的首尾迭代器

首迭代器即指向第一个页面中的第一个元组的迭代器。在实现上，从第一个页面

开始，沿链表遍历所有页面，依次检查页面中是否含有元组。尾迭代器即指向无效元组的迭代器。

```
TableIterator TableHeap::Begin(Txn *txn) {
        RowId rid;
        page_id_t page_id = first_page_id_;
        while (1) {
                // 从第一个页面开始，沿链表遍历所有页面，依次检查页面中是否含有元组
                if (page_id == INVALID_PAGE_ID) {break;}
                TablePage* page = reinterpret_cast<TablePage *>(buffer_pool_manager_->
                    FetchPage(page_id));
                // 如果该页面为空指针，则跳出循环
                if (page == nullptr) {break;}
                // 尝试在该页面寻找元组
                if (page->GetFirstTupleRid(&rid)) {
                        // 找到元组，退出函数
                        buffer_pool_manager_->UnpinPage(page->GetTablePageId(), false);
                        return TableIterator(this, rid, txn);
                }
                // 未找到元组，尝试下一页面
                page_id = page->GetNextPageId();
                buffer_pool_manager_->UnpinPage(page->GetTablePageId(), false);
        }
        return End();
}

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

3. **迭代器自增**

在实现 `iter++` 与 `++iter` 的逻辑中，使用如下逻辑实现迭代器的自增。在实现上，先尝试从当前页面查找下一个元组；若找不到，则沿链表遍历所有页面，依次检查页面中是否含有元组。若找不到下一个元组，则直接返回尾迭代器。

```
RowId TableIterator::GetNextRId() {
        // 获取当前页
        auto cur_page = reinterpret_cast<TablePage *>(table_heap_->buffer_pool_manager_
            ->FetchPage(rid_.GetPageId()));
        // 如果能从当前页找到下一个元组，则直接返回
        RowId next_rid;
        if (cur_page->GetNextTupleRid(rid_, &next_rid)) {
                table_heap_->buffer_pool_manager_->UnpinPage(cur_page->GetTablePageId(),
                     false);
                return next_rid;
        }
        // 当前页没有下一个元组，尝试跨页查找
        page_id_t next_page_id = cur_page->GetNextPageId();
        table_heap_->buffer_pool_manager_->UnpinPage(rid_.GetPageId(), false);

        while (next_page_id != INVALID_PAGE_ID) {
                auto next_page = reinterpret_cast<TablePage *>(table_heap_->
                    buffer_pool_manager_->FetchPage(next_page_id));
                if (next_page == nullptr) {
                        break;
                }
                if (next_page->GetFirstTupleRid(&next_rid)) {
```

```
                        table_heap_->buffer_pool_manager_->UnpinPage(next_page_id, false
                            );
                        return next_rid;
                }
                // 否则继续往下一页找
                next_page_id = next_page->GetNextPageId();
                table_heap_->buffer_pool_manager_->UnpinPage(next_page->GetTablePageId()
                    , false);
        }
        // 未找到有效元组
        return INVALID_ROWID;
}
```

2.3 INDEX MANAGER

2.3.1 模块功能概述

Index Manager 模块负责管理表上的索引。其核心任务是实现底层的 B+ 树容器，而上层的索引封装和管理由框架完成。考虑到数据库系统的持久化需求，本模块实现的 B+ 树采用磁盘存储模型：每个节点占据一个数据页，其内容持久化在磁盘中。访问节点时，需将其从磁盘加载到内存进行处理。

成员变量：

BPlusTree 类包含六个成员变量:

```
class BPlusTree {
        index_id_t index_id_; // B+树对应的索引
        page_id_t root_page_id_{INVALID_PAGE_ID}; // B+树根页面
        BufferPoolManager *buffer_pool_manager_; // 缓冲池管理
        KeyManager processor_; // 键值比较器
        int leaf_max_size_; // 叶子节点最大size
        int internal_max_size_; // 中间节点最大size
};
```

2.3.2 B+ 树的创建与销毁

1. **B+ 树的对象初始化**

   在 B+ 树的构造函数中，除需对 `leaf_max_size_` 与 `internal_max_size_` 等成员变量进行初始化外，还需对 B+ 树的根页面页号（`root_page_id`）进行初始化。在实现上，构造函数尝试根据索引号 `index_id` 从索引根页获取根页面的页号。若获取失败，则将根页面页号置为 `INVALID_PAGE_ID`。

```
BPlusTree::BPlusTree(index_id_t index_id, BufferPoolManager *buffer_pool_manager, const
    KeyManager &KM,
int leaf_max_size, int internal_max_size)
: index_id_(index_id),
buffer_pool_manager_(buffer_pool_manager),
processor_(KM),
leaf_max_size_(leaf_max_size),
internal_max_size_(internal_max_size) {
```

```
        root_page_id_ = INVALID_PAGE_ID;
        // 尝试从索引根页获取已有的root_page_id
        Page *page = buffer_pool_manager->FetchPage(INDEX_ROOTS_PAGE_ID);
        auto roots_page = reinterpret_cast<IndexRootsPage *>(page->GetData());
        page_id_t root_page_id;
        // 如果找到了当前索引的root page id，则更新root_page_id_
        if (roots_page->GetRootId(index_id, &root_page_id)) {
                root_page_id_ = root_page_id;
        }
        buffer_pool_manager->UnpinPage(INDEX_ROOTS_PAGE_ID, false);
}
```

2. **B+ 树的创建**

   若当前的 B+ 树不存在根页面，则在首次向 B+ 树插入条目时将调用下述函数。该函数为 B+ 树创建一个新页面作为根节点，并相应地对索引根页进行更新。

```
void BPlusTree::StartNewTree(GenericKey *key, const RowId &value) {
        // 从缓冲池获取一个新页面作为根叶子节点
        page_id_t new_page_id;
        Page *root_page = buffer_pool_manager_->NewPage(new_page_id);
        if (root_page == nullptr) {
                throw std::runtime_error("out of memory");
        }
        // 初始化根叶子节点
        auto *root = reinterpret_cast<LeafPage *>(root_page->GetData());
        root->Init(new_page_id, INVALID_PAGE_ID, processor_.GetKeySize(), leaf_max_size_
            );
        // 插入键值对
        root->Insert(key, value, processor_);
        // 更新根页面ID
        root_page_id_ = new_page_id;
        // 更新索引根页
        UpdateRootPageId(1);
        // 解除pin
        buffer_pool_manager_->UnpinPage(new_page_id, true);
}
```

3. **B+ 树的销毁**

   该函数仅由 BPlusTreeIndex 类的 Destroy 方法调用，通过从根节点递归向下释放每一个数据页，实现整个 B+ 树的销毁。

```
void BPlusTree::Destroy(page_id_t current_page_id) {
        // 如果当前页面ID无效或空树，直接返回
        if (current_page_id == INVALID_PAGE_ID || IsEmpty()) {
                return;
        }
        // 获取当前页面
        Page *page = buffer_pool_manager_->FetchPage(current_page_id);
        if (page == nullptr) {
                return;
        }
        auto *node = reinterpret_cast<BPlusTreePage *>(page->GetData());
        // 如果是内部节点，先递归销毁所有子节点
```

```
        if (!node->IsLeafPage()) {
                auto *internal_page = reinterpret_cast<InternalPage *>(node);
                // 递归删除所有子节点
                for (int i = 0; i < internal_page->GetSize(); i++) {
                        page_id_t child_page_id = internal_page->ValueAt(i);
                        Destroy(child_page_id);
                }
        }
        // 解除pin
        buffer_pool_manager_->UnpinPage(current_page_id, true);
        // 删除当前页面
        buffer_pool_manager_->DeletePage(current_page_id);
        // 如果是根页面，重置根页面ID
        if (current_page_id == root_page_id_) {
                root_page_id_ = INVALID_PAGE_ID;
                UpdateRootPageId(0);
        }
}
```

2.3.3 B+ 树的查找操作

在实现上，需要访问 B+ 树，将目标 key 和 B+ 树中各个节点的 key 进行比较，逐层向下直至最终的目标节点，并返回对应的行号 RowId。

```
bool BPlusTree::GetValue(const GenericKey *key, std::vector<RowId> &result, Txn *transaction)
    {
        // 如果树为空，直接返回false
        if (IsEmpty()) {
                return false;
        }
        // 查找包含key的叶子节点
        Page *page = FindLeafPage(key, root_page_id_, false);
        if (page == nullptr) {
                return false;
        }
        // 转换为叶子节点
        auto *leaf = reinterpret_cast<LeafPage *>(page->GetData());
        RowId row_id;
        // 在叶子节点中查找key
        bool found = leaf->Lookup(key, row_id, processor_);
        // 如果找到key，添加到结果中
        if (found) {
                result.push_back(row_id);
        }
        // 解除pin
        buffer_pool_manager_->UnpinPage(leaf->GetPageId(), false);
        return found;
}
```

2.3.4 B+ 树的插入操作

B+ 树的插入操作流程如下:

1. 找到目标叶子节点
2. 将数据插入叶子节点（InsertIntoLeaf）
3. 若插入后节点大小超过限制，则分裂当前叶子节点（Split），并将对应的 key 插入上层节点（InsertIntoParent）
4. 若上层节点也发生分裂，则递归向上分裂直到根节点
5. 在各层节点进行插入或分裂后,需对应地更新上层节点的记录(UpdateParentKeyAfterChange)

函数实现：

1. **Insert**

```
bool BPlusTree::Insert(GenericKey *key, const RowId &value, Txn *transaction) {
        // 若当前树为空，建立新树
        if (IsEmpty()) {
                StartNewTree(key, value);
                return true;
        }
        // 否则，插入现有节点
        return InsertIntoLeaf(key, value, transaction);
}
```

2. **InsertIntoLeaf**

```
bool BPlusTree::InsertIntoLeaf(GenericKey *key, const RowId &value, Txn *transaction) {
        // 查找合适的叶子节点
        Page *page = FindLeafPage(key, root_page_id_, false);
        if (page == nullptr) {
                return false;
        }
        auto *leaf = reinterpret_cast<LeafPage *>(page->GetData());
        // 检查key是否已存在
        RowId tmp_value;
        if (leaf->Lookup(key, tmp_value, processor_)) {
                // key已存在，解除pin并返回false
                buffer_pool_manager_->UnpinPage(leaf->GetPageId(), false);
                return false;
        }
        // 插入key-value对
        int size_after_insertion = leaf->Insert(key, value, processor_);
        // 检查叶子节点是否需要分裂
        if (size_after_insertion > leaf->GetMaxSize()) {
                // 分裂叶子节点
                auto *new_leaf = Split(leaf, transaction);
                // 获取中间键
                GenericKey *middle_key = new_leaf->KeyAt(0);
                // 将中间键和新叶子节点插入到父节点
                InsertIntoParent(leaf, middle_key, new_leaf, transaction);
                // 不需要在此解除pin新叶子节点，因为InsertIntoParent方法会处理这个节点
        }
        // 解除pin叶子节点
```

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

3. Split

```
BPlusTreeInternalPage *BPlusTree::Split(InternalPage *node, Txn *transaction) {
        // 创建新页面
        page_id_t new_page_id;
        Page *new_page = buffer_pool_manager_->NewPage(new_page_id);
        if (new_page == nullptr) {
                throw std::runtime_error("out of memory");
        }
        // 初始化新内部节点
        auto *new_node = reinterpret_cast<InternalPage *>(new_page->GetData());
        new_node->Init(new_page_id, node->GetParentPageId(), processor_.GetKeySize(),
            internal_max_size_);
        // 将一半键值对移动到新节点
        node->MoveHalfTo(new_node, buffer_pool_manager_);
        return new_node;
}

BPlusTreeLeafPage *BPlusTree::Split(LeafPage *node, Txn *transaction) {
        // 创建新页面
        page_id_t new_page_id;
        Page *new_page = buffer_pool_manager_->NewPage(new_page_id);
        if (new_page == nullptr) {
                throw std::runtime_error("out of memory");
        }
        // 初始化新叶子节点
        auto *new_node = reinterpret_cast<LeafPage *>(new_page->GetData());
        new_node->Init(new_page_id, node->GetParentPageId(), processor_.GetKeySize(),
            leaf_max_size_);
        // 将一半键值对移动到新节点
        node->MoveHalfTo(new_node);
        // 设置新叶子节点的下一个页面指针
        new_node->SetNextPageId(node->GetNextPageId());
        node->SetNextPageId(new_page_id);

        return new_node;
}
```

4. InsertIntoParent

```
void BPlusTree::InsertIntoParent(BPlusTreePage *old_node, GenericKey *key, BPlusTreePage
     *new_node, Txn *transaction) {
        if (old_node->IsRootPage()) {
                // 创建新的根节点
                page_id_t new_page_id;
                Page *page = buffer_pool_manager_->NewPage(new_page_id);
                if (page == nullptr) {
                        throw std::runtime_error("out of memory");
                }
                // 初始化新根节点
                auto *new_root = reinterpret_cast<InternalPage *>(page->GetData());
```

```
                new_root->Init(new_page_id, INVALID_PAGE_ID, processor_.GetKeySize(),
                    internal_max_size_);
                // 填充新根节点
                new_root->PopulateNewRoot(old_node->GetPageId(), key, new_node->
                    GetPageId());
                // 更新父节点指针
                old_node->SetParentPageId(new_page_id);
                new_node->SetParentPageId(new_page_id);
                // 更新根页面ID
                root_page_id_ = new_page_id;
                UpdateRootPageId(0);
                // 解除pin新根节点
                buffer_pool_manager_->UnpinPage(new_page_id, true);
                return;
        }
        page_id_t parent_page_id = old_node->GetParentPageId();
        Page *parent_page = buffer_pool_manager_->FetchPage(parent_page_id);
        if (parent_page == nullptr) {
                throw std::runtime_error("out of memory");
        }
        auto *parent = reinterpret_cast<InternalPage *>(parent_page->GetData());
        new_node->SetParentPageId(parent_page_id);
        int size_after_insertion = parent->InsertNodeAfter(old_node->GetPageId(), key,
            new_node->GetPageId());
        // 用新子节点的最小key更新分隔键
        int new_index = parent->ValueIndex(new_node->GetPageId());
        UpdateParentKeyAfterChange(parent, new_index);
        if (size_after_insertion > parent->GetMaxSize()) {
                auto *new_parent = Split(parent, transaction);
                GenericKey *middle_key = parent->KeyAt(parent->GetSize());
                InsertIntoParent(parent, middle_key, new_parent, transaction);
        }
        buffer_pool_manager_->UnpinPage(parent_page_id, true);
}
```

5. UpdateParentKeyAfterChange

```
void BPlusTree::UpdateParentKeyAfterChange(InternalPage *parent, int index) {
        if (index < parent->GetSize()) {
                page_id_t right_child = parent->ValueAt(index);
                Page *right_page = buffer_pool_manager_->FetchPage(right_child);
                if (right_page) {
                        BPlusTreePage *right_node = reinterpret_cast<BPlusTreePage *>(
                            right_page->GetData());
                        GenericKey *min_key = nullptr;
                        if (right_node->IsLeafPage()) {
                                auto *leaf = reinterpret_cast<LeafPage *>(right_node);
                                if (leaf->GetSize() > 0) min_key = leaf->KeyAt(0);
                        } else {
                                auto *internal = reinterpret_cast<InternalPage *>(
                                    right_node);
                                page_id_t leftmost = internal->ValueAt(0);
                                Page *leftmost_leaf_page = FindLeafPage(nullptr,
                                    leftmost, true);
                                if (leftmost_leaf_page) {
                                        auto *leaf = reinterpret_cast<LeafPage *>(
```

```
                                        leftmost_leaf_page->GetData());
                                    if (leaf->GetSize() > 0) min_key = leaf->KeyAt
                                        (0);
                                    buffer_pool_manager_->UnpinPage(leaf->GetPageId
                                        (), false);
                                }
                            }
                            if (min_key) parent->SetKeyAt(index, min_key);
                            buffer_pool_manager_->UnpinPage(right_child, false);
                        }
                }
        }
```

2.3.5 B+ 树的删除操作

B+ 树的删除操作流程如下:

1. 找到目标叶子节点（Remove）
2. 将数据从叶子节点中删除
3. 若删除后叶子节点小于限制，则根据情况选择（CoalesceOrRedistribute）重排或合并
4. 执行重排（Redistribute）或合并（Coalesce）
5. 对应更新父节点、根节点（AdjustRoot）

函数实现：

1. **Remove**

```
void BPlusTree::Remove(const GenericKey *key, Txn *transaction) {
        // 如果树为空，直接返回
        if (IsEmpty()) {
                return;
        }
        // 查找包含key的叶子节点
        Page *page = FindLeafPage(key, root_page_id_, false);
        if (page == nullptr) {
                return;
        }
        auto *leaf = reinterpret_cast<LeafPage *>(page->GetData());
        // 获取删除前的大小
        int size_before_deletion = leaf->GetSize();
        // 尝试删除键值对，返回删除后的大小
        int size_after_deletion = leaf->RemoveAndDeleteRecord(key, processor_);
        // 如果大小没有变化，说明key不存在，直接返回
        if (size_before_deletion == size_after_deletion) {
                buffer_pool_manager_->UnpinPage(leaf->GetPageId(), false);
                return;
        }
```

```
        // 检查是否需要合并或重新分配
        bool should_delete = CoalesceOrRedistribute(leaf, transaction);
        // 解除pin叶子节点
        buffer_pool_manager_->UnpinPage(leaf->GetPageId(), true);
        // 如果需要删除节点
        if (should_delete) {
                buffer_pool_manager_->DeletePage(leaf->GetPageId());
        }
}
```

2. CoalesceOrRedistribute

```
template <typename N>
bool BPlusTree::CoalesceOrRedistribute(N *&node, Txn *transaction) {
        // 如果是根节点，特殊处理
        if (node->IsRootPage()) {
                return AdjustRoot(node);
        }
        // 如果节点大小大于等于最小大小，不需要合并或重新分配
        if (node->GetSize() >= node->GetMinSize()) {
                return false;
        }
        // 获取父节点
        page_id_t parent_page_id = node->GetParentPageId();
        Page *parent_page = buffer_pool_manager_->FetchPage(parent_page_id);
        if (parent_page == nullptr) {
                return false;
        }
        auto *parent = reinterpret_cast<InternalPage *>(parent_page->GetData());
        // 获取node在父节点中的索引
        int node_index = parent->ValueIndex(node->GetPageId());
        // 确定兄弟节点的索引
        int sibling_index;
        if (node_index == 0) {
                // 如果当前节点是第一个子节点，选择右兄弟
                sibling_index = 1;
        } else {
                // 否则选择左兄弟
                sibling_index = node_index - 1;
        }
        // 获取兄弟节点
        page_id_t sibling_page_id = parent->ValueAt(sibling_index);
        Page *sibling_page = buffer_pool_manager_->FetchPage(sibling_page_id);
        if (sibling_page == nullptr) {
                buffer_pool_manager_->UnpinPage(parent_page_id, false);
                return false;
        }
        N *sibling = reinterpret_cast<N *>(sibling_page->GetData());
        // 判断是重新分配还是合并
        bool should_delete = false;
        if (sibling->GetSize() + node->GetSize() > node->GetMaxSize()) {
                // 重新分配
                if (node_index == 0) {
                        // 如果当前节点是第一个子节点，则从右兄弟重新分配
                        // index=0表示将兄弟节点的第一个键值对移到node的末尾
                        Redistribute(sibling, node, 0);
```

```
                } else {
                        // 否则从左兄弟重新分配
                        // index=1表示将兄弟节点的最后一个键值对移到node的开头
                        Redistribute(sibling, node, 1);
                }
        }
        else {
                // 合并
                if (node_index == 0) {
                        // 如果当前节点是第一个子节点，则与右兄弟合并
                        // 注意：我们始终保留左侧节点，所以交换参数顺序
                        // 节点位置顺序是：[node] [key at index=1] [sibling]
                        // 删除的是index=1的键，将右兄弟合并到左侧节点
                        should_delete = Coalesce(node, sibling, parent, 1, transaction);
                } else {
                        // 否则与左兄弟合并
                        // 节点位置顺序是：[sibling] [key at index=node_index] [node]
                        // 删除的是index=node_index的键，将当前节点合并到左兄弟
                        should_delete = Coalesce(sibling, node, parent, node_index,
                            transaction);
                }
        }
        // 解除pin父节点和兄弟节点
        buffer_pool_manager_->UnpinPage(parent_page_id, true);
        buffer_pool_manager_->UnpinPage(sibling_page_id, true);

        return should_delete;
}
```

3. Coalesce

```
bool BPlusTree::Coalesce(LeafPage *&neighbor_node, LeafPage *&node, InternalPage *&
    parent, int index,
Txn *transaction) {
        // 记录合并前的关键信息
        // int original_neighbor_size = neighbor_node->GetSize();
        // 移动所有键值对到neighbor_node
        node->MoveAllTo(neighbor_node);
        // 更新nextPageId指针
        neighbor_node->SetNextPageId(node->GetNextPageId());
        // 从父节点中删除对应的键值对
        parent->Remove(index);
        // 检查父节点是否需要合并或重新分配
        bool parent_should_delete = false;
        if (parent->GetSize() < parent->GetMinSize()) {
                parent_should_delete = CoalesceOrRedistribute(parent, transaction);
        }
        // 用新子节点的最小key更新分隔键
        UpdateParentKeyAfterChange(parent, index - 1);
        return parent_should_delete;
}

bool BPlusTree::Coalesce(InternalPage *&neighbor_node, InternalPage *&node, InternalPage
     *&parent, int index,
Txn *transaction) {
        // 获取中间键
```

```
        GenericKey *middle_key = parent->KeyAt(index);
        // 移动所有键值对到neighbor_node
        node->MoveAllTo(neighbor_node, middle_key, buffer_pool_manager_);
        // 从父节点中删除对应的键值对
        parent->Remove(index);
        // 检查父节点是否需要合并或重新分配
        bool parent_should_delete = false;
        if (parent->GetSize() < parent->GetMinSize()) {
                parent_should_delete = CoalesceOrRedistribute(parent, transaction);
        }
        // 用新子节点的最小key更新分隔键
        UpdateParentKeyAfterChange(parent, index - 1);
        return parent_should_delete;
}
```

4. Redistribute

```
void BPlusTree::Redistribute(InternalPage *neighbor_node, InternalPage *node, int index)
    {
        // 获取父节点
        page_id_t parent_page_id = node->GetParentPageId();
        Page *parent_page = buffer_pool_manager_->FetchPage(parent_page_id);
        if (parent_page == nullptr) {
                throw std::runtime_error("out of memory");
        }
        auto *parent = reinterpret_cast<InternalPage *>(parent_page->GetData());
        // 获取node在父节点中的索引
        int node_index = parent->ValueIndex(node->GetPageId());
        int neighbor_index = parent->ValueIndex(neighbor_node->GetPageId());
        if (index == 0) {
                // 从兄弟节点的第一个位置移动到node的最后一个位置
                GenericKey *middle_key = parent->KeyAt(node_index + 1);
                neighbor_node->MoveFirstToEndOf(node, middle_key, buffer_pool_manager_);
                UpdateParentKeyAfterChange(parent, neighbor_index);
        }
        else {
                // 从兄弟节点的最后一个位置移动到node的第一个位置
                GenericKey *middle_key = parent->KeyAt(node_index);
                neighbor_node->MoveLastToFrontOf(node, middle_key, buffer_pool_manager_)
                    ;
                UpdateParentKeyAfterChange(parent, node_index);
        }
        // 解除pin父节点
        buffer_pool_manager_->UnpinPage(parent_page_id, true);
}

void BPlusTree::Redistribute(LeafPage *neighbor_node, LeafPage *node, int index) {
        // 获取父节点
        page_id_t parent_page_id = node->GetParentPageId();
        Page *parent_page = buffer_pool_manager_->FetchPage(parent_page_id);
        if (parent_page == nullptr) {
                throw std::runtime_error("out of memory");
        }
        auto *parent = reinterpret_cast<InternalPage *>(parent_page->GetData());
        // 获取node在父节点中的索引
        int node_index = parent->ValueIndex(node->GetPageId());
```

```
        int neighbor_index = parent->ValueIndex(neighbor_node->GetPageId());
        if (index == 0) {
                // 从兄弟节点的第一个位置移动到node的最后一个位置
                neighbor_node->MoveFirstToEndOf(node);
                UpdateParentKeyAfterChange(parent, neighbor_index);
        }
        else {
                // 从兄弟节点的最后一个位置移动到node的第一个位置
                neighbor_node->MoveLastToFrontOf(node);
                UpdateParentKeyAfterChange(parent, node_index);
        }
        // 解除pin父节点
        buffer_pool_manager_->UnpinPage(parent_page_id, true);
}
```

5. **AdjustRoot**

```
bool BPlusTree::AdjustRoot(BPlusTreePage *old_root_node) {
        // 情况1：当根节点是内部节点且只有一个子节点时
        if (!old_root_node->IsLeafPage() && old_root_node->GetSize() == 1) {
                auto *root = reinterpret_cast<InternalPage *>(old_root_node);
                // 获取唯一的子节点
                page_id_t child_page_id = root->RemoveAndReturnOnlyChild();
                Page *child_page = buffer_pool_manager_->FetchPage(child_page_id);
                if (child_page == nullptr) {
                        return false;
                }
                auto *child = reinterpret_cast<BPlusTreePage *>(child_page->GetData());
                // 更新子节点的父节点为INVALID_PAGE_ID（因为子节点将成为新的根节点）
                child->SetParentPageId(INVALID_PAGE_ID);
                // 更新根页面ID
                root_page_id_ = child_page_id;
                UpdateRootPageId(0);
                // 解除pin子节点
                buffer_pool_manager_->UnpinPage(child_page_id, true);
                return true;
        }
        // 情况2：当根节点是叶子节点且为空时
        if (old_root_node->IsLeafPage() && old_root_node->GetSize() == 0) {
                // 树现在为空
                root_page_id_ = INVALID_PAGE_ID;
                UpdateRootPageId(0);
                return true;
        }
        return false;
}
```

2.3.6 索引迭代器

接口实现：

1. 运算符 * 的重载

```
std::pair<GenericKey *, RowId> IndexIterator::operator*() {
```

```
        // 确保页面和索引有效
        if (current_page_id == INVALID_PAGE_ID || item_index < 0 ||
        page == nullptr || item_index >= page->GetSize()) {
                throw std::runtime_error("Invalid Iterator");
        }
        // 返回当前键值对
        return page->GetItem(item_index);
}
```

2. 运算符 ++ 的重载

```
IndexIterator &IndexIterator::operator++() {
        // 确保页面和索引有效
        if (current_page_id == INVALID_PAGE_ID ||
        page == nullptr || item_index < 0 || item_index >= page->GetSize()) {
                return *this;
        }
        // 移动到下一个元素
        item_index++;
        // 如果到达当前页面的末尾，尝试移动到下一个页面
        if (item_index >= page->GetSize()) {
                page_id_t next_page_id = page->GetNextPageId();
                // 如果没有下一个页面，设置为无效
                if (next_page_id == INVALID_PAGE_ID) {
                        // 解除当前页面的pin
                        buffer_pool_manager->UnpinPage(current_page_id, false);
                        current_page_id = INVALID_PAGE_ID;
                        page = nullptr;
                        item_index = 0;
                        return *this;
                }
                // 获取下一个页面前先解除当前页面的pin
                buffer_pool_manager->UnpinPage(current_page_id, false);
                // 获取下一个页面
                current_page_id = next_page_id;
                Page *next_page = buffer_pool_manager->FetchPage(next_page_id);
                if (next_page == nullptr) {
                        // 如果获取失败，设置为无效
                        current_page_id = INVALID_PAGE_ID;
                        page = nullptr;
                        item_index = 0;
                        return *this;
                }
                // 更新页面和索引
                page = reinterpret_cast<LeafPage *>(next_page->GetData());
                item_index = 0;
        }
        return *this;
}
```

2.4 CATALOG MANAGER

2.4.1 模块功能概述

Catalog Manager 负责管理数据库中的所有元数据信息，充当着数据库的“数据字典”。它维护着关于表、索引、列等对象的定义和结构信息。当执行器需要了解表的结构、索引的列或者数据类型等信息时，都会向 Catalog Manager 发出请求。

具体而言，Catalog Manager 的核心功能包括：

- **表的管理：**创建新表、删除已有表、获取表信息（包括表名、表 ID、表的 Schema、表数据在磁盘上的起始页号等）。
- **索引的管理：**在指定表上创建新索引、删除已有索引、获取索引信息（包括索引名、索引 ID、所属表名、索引键、索引类型等）。
- **元数据持久化：**将所有元数据信息持久化存储到磁盘中，并在数据库启动时加载这些信息，保证数据库重启后元数据不丢失。

为了高效管理元数据，Catalog Manager 内部维护了多个映射关系，这些映射使得 Catalog Manager 能够快速定位和访问所需的元数据。例如：

- 表名到表 ID 的映射:`table_names_ (std::unordered_map<std::string, table_id_t>)`
- 表 ID 到表信息对象的映射:`tables_ (std::unordered_map<table_id_t, TableInfo *>)`
- 索引名到索引 ID 的映射（按表组织）:`index_names_ (std::unordered_map<std::string, std::unordered_map<std::string, index_id_t>>)`
- 索引 ID 到索引信息对象的映射：`indexes_ (std::unordered_map<index_id_t, IndexInfo *>)`

此外，Catalog Manager 还依赖于一个特殊的 `CatalogMeta` 对象来管理元数据页的分配。`CatalogMeta` 记录了所有表元数据页和索引元数据页的物理页号，并负责自身的序列化与反序列化，确保元数据的持久性。

2.4.2 元数据持久化与加载

Catalog Manager 的一个关键职责是确保元数据的持久性。这是通过 `CatalogMeta` 对象和 Buffer Pool Manager 实现的。

CatalogMeta 对象：

`CatalogMeta` 类专门用于管理和持久化目录的元信息。

- `table_meta_pages_`：一个 `std::map<table_id_t, page_id_t>`，存储每个表 ID 对应的元数据页的页号。

- `index_meta_pages_`：一个 `std::map<index_id_t, page_id_t>`，存储每个索引 ID 对应的元数据页的页号。

`CatalogMeta` 对象本身会被序列化并存储在一个固定的元数据页（由 `CATALOG_META_PAGE_ID` 定义，默认为逻辑页号 0）上。

```
class CatalogMeta {
        friend class CatalogManager;

        public:
        void SerializeTo(char *buf) const;
        static CatalogMeta *DeserializeFrom(char *buf);
        uint32_t GetSerializedSize() const;
        inline table_id_t GetNextTableId() const;
        inline index_id_t GetNextIndexId() const;
        static CatalogMeta *NewInstance();
        inline std::map<table_id_t, page_id_t> *GetTableMetaPages();
        inline std::map<index_id_t, page_id_t> *GetIndexMetaPages();
        bool DeleteIndexMetaPage(BufferPoolManager *bpm, index_id_t index_id);

        private:
        CatalogMeta();

        private:
        static constexpr uint32_t CATALOG_METADATA_MAGIC_NUM = 89849;
        std::map<table_id_t, page_id_t> table_meta_pages_;
        std::map<index_id_t, page_id_t> index_meta_pages_;
};
```

初始化与加载流程：

1. **初始化 (init=true)：**当数据库首次创建或需要重新初始化目录时，Catalog Manager 会创建一个新的 `CatalogMeta` 对象，并将其序列化到 `CATALOG_META_PAGE_ID` 指定的页面。此时，`table_meta_pages_` 和 `index_meta_pages_` 均为空。

2. **加载 (init=false)：**当数据库启动时，Catalog Manager 会从 `CATALOG_META_PAGE_ID` 读取并反序列化 `CatalogMeta` 对象。随后，它会遍历 `catalog_meta_->table_meta_pages_` 和 `catalog_meta_->index_meta_pages_`，调用 `LoadTable` 和 `LoadIndex` 方法，从磁盘加载每个表和索引的详细元数据，并重建内存中的 `tables_`、`table_names_`、`indexes_` 和 `index_names_` 等映射关系。

刷新元数据：每当目录信息发生更改（如创建表、删除索引等），Catalog Manager 会调用 `FlushCatalogMetaPage()` 方法。此方法将当前的 `catalog_meta_` 对象序列化并写回磁盘上的 `CATALOG_META_PAGE_ID`，确保更改的持久性。

2.4.3 核心接口实现

成员变量：

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

接口实现：

1. 创建表

```
dberr_t CatalogManager::CreateTable(const string &table_name, TableSchema *schema, Txn *
    txn, TableInfo *&table_info) {
        // 检查表是否已存在
        if (table_names_.find(table_name) != table_names_.end()) {
                return DB_TABLE_ALREADY_EXIST;
        }
        // 分配表元数据页
        page_id_t table_meta_page_id;
        Page *table_meta_page = buffer_pool_manager_->NewPage(table_meta_page_id);
        if (table_meta_page == nullptr) {
                return DB_FAILED;
        }
        // 创建表元数据（暂时使用临时页面ID）
        table_id_t table_id = next_table_id_++;
        // 使用 Schema 的深拷贝函数创建拷贝
        TableSchema *schema_copy = Schema::DeepCopySchema(schema);
        if (schema_copy == nullptr) {
                buffer_pool_manager_->DeletePage(table_meta_page_id);
                return DB_FAILED;
        }
        // 先创建表堆，获取真实的第一页ID
        TableHeap *table_heap = TableHeap::Create(buffer_pool_manager_, schema_copy, txn
            , log_manager_, lock_manager_);
        if (table_heap == nullptr) {
                delete schema_copy;
                buffer_pool_manager_->DeletePage(table_meta_page_id);
                return DB_FAILED;
        }
        // 使用表堆的真实第一页ID创建表元数据
        TableMetadata *table_meta = TableMetadata::Create(table_id, table_name,
            table_heap->GetFirstPageId(), schema_copy);
        if (table_meta == nullptr) {
                delete schema_copy;
                delete table_heap;
                buffer_pool_manager_->DeletePage(table_meta_page_id);
```

```
                return DB_FAILED;
        }
        // 序列化表元数据
        table_meta->SerializeTo(table_meta_page->GetData());
        buffer_pool_manager_->UnpinPage(table_meta_page_id, true);
        // 创建表信息
        table_info = TableInfo::Create();
        if (table_info == nullptr) {
                delete table_meta;
                delete table_heap;
                buffer_pool_manager_->DeletePage(table_meta_page_id);
                return DB_FAILED;
        }
        table_info->Init(table_meta, table_heap);
        // 更新目录元数据
        catalog_meta_->table_meta_pages_[table_id] = table_meta_page_id;
        table_names_[table_name] = table_id;
        tables_[table_id] = table_info;
        // 刷新目录元数据页
        FlushCatalogMetaPage();
        return DB_SUCCESS;
}
```

2. 获取表

```
dberr_t CatalogManager::GetTable(const string &table_name, TableInfo *&table_info) {
        auto iter = table_names_.find(table_name);
        if (iter == table_names_.end()) {
                return DB_TABLE_NOT_EXIST;
        }
        table_id_t table_id = iter->second;
        auto table_iter = tables_.find(table_id);
        if (table_iter == tables_.end()) {
                // This case should ideally not happen if table_names_ is consistent
                // It implies an internal inconsistency, possibly after loading.
                // For robustness, try loading the table if not found in memory.
                page_id_t table_meta_page_id;
                if (catalog_meta_->table_meta_pages_.count(table_id)) {
                        table_meta_page_id = catalog_meta_->table_meta_pages_[table_id];
                        if (LoadTable(table_id, table_meta_page_id) == DB_SUCCESS) {
                                table_iter = tables_.find(table_id);
                                if (table_iter != tables_.end()) {
                                        table_info = table_iter->second;
                                        return DB_SUCCESS;
                                }
                        }
                }
                return DB_FAILED; // Or a more specific error like DB_TABLE_NOT_LOADED
        }
        table_info = table_iter->second;
        return DB_SUCCESS;
}
```

3. 创建索引

```cpp
dberr_t CatalogManager::CreateIndex(const std::string &table_name, const string &index_name,
const std::vector<std::string> &index_keys, Txn *txn, IndexInfo *&index_info,
const string &index_type) {
        // 检查表是否存在
        TableInfo *table_info = nullptr;
        if (GetTable(table_name, table_info) != DB_SUCCESS) {
                return DB_TABLE_NOT_EXIST;
        }
        // 检查索引是否已存在
        if (index_names_.count(table_name) && index_names_[table_name].count(index_name)) {
                return DB_INDEX_ALREADY_EXIST;
        }
        // 创建索引元数据页
        page_id_t index_meta_page_id;
        Page *index_meta_page = buffer_pool_manager_->NewPage(index_meta_page_id);
        if (index_meta_page == nullptr) {
                return DB_FAILED;
        }
        // 获取索引键的列索引
        std::vector<uint32_t> key_map;
        const Schema *schema = table_info->GetSchema();
        for (const auto &key : index_keys) {
                uint32_t col_idx;
                if (schema->GetColumnIndex(key, col_idx) == DB_COLUMN_NAME_NOT_EXIST) {
                        buffer_pool_manager_->DeletePage(index_meta_page_id);
                        // buffer_pool_manager_->DeletePage(index_root_page_id); // No longer needed
                        return DB_COLUMN_NAME_NOT_EXIST;
                }
                key_map.push_back(col_idx);
        }
        // 创建索引元数据
        index_id_t index_id = next_index_id_++;
        IndexMetadata *index_meta = IndexMetadata::Create(index_id, index_name, table_info->GetTableId(), key_map);
        if (index_meta == nullptr) {
                buffer_pool_manager_->DeletePage(index_meta_page_id);
                // buffer_pool_manager_->DeletePage(index_root_page_id); // No longer needed
                return DB_FAILED;
        }
        // 序列化索引元数据
        index_meta->SerializeTo(index_meta_page->GetData());
        buffer_pool_manager_->UnpinPage(index_meta_page_id, true);
        // buffer_pool_manager_->UnpinPage(index_root_page_id, true); // No longer needed
        // 创建索引信息
        index_info = IndexInfo::Create();
        if (index_info == nullptr) {
                delete index_meta;
                buffer_pool_manager_->DeletePage(index_meta_page_id);
                // buffer_pool_manager_->DeletePage(index_root_page_id); // No longer needed
                return DB_FAILED;
```

```
        }
        // The Init function for IndexInfo will internally create the BPlusTreeIndex,
        // which handles its own root page creation.
        index_info->Init(index_meta, table_info, buffer_pool_manager_);
        // 更新目录元数据
        catalog_meta_->index_meta_pages_[index_id] = index_meta_page_id;
        index_names_[table_name][index_name] = index_id;
        indexes_[index_id] = index_info;
        // 为表中已有的数据建立索引条目
        TableHeap *table_heap = table_info->GetTableHeap();
        for (auto iter = table_heap->Begin(txn); iter != table_heap->End(); ++iter) {
                Row index_key_row; // Renamed from 'Row index_key' to avoid conflict
                iter->GetKeyFromRow(table_info->GetSchema(), index_info->
                    GetIndexKeySchema(), index_key_row);
                index_info->GetIndex()->InsertEntry(index_key_row, iter->GetRowId(), txn
                    );
        }
        // 刷新目录元数据页
        FlushCatalogMetaPage();
        return DB_SUCCESS;
}
```

4. 获取索引

```
dberr_t CatalogManager::GetIndex(const std::string &table_name, const std::string &
    index_name,
IndexInfo *&index_info) const {
        // 检查表是否存在于索引名称映射中
        auto table_iter = index_names_.find(table_name);
        if (table_iter == index_names_.end()) {
                return DB_INDEX_NOT_FOUND;
        }
        // 检查索引是否存在于该表的索引名称映射中
        auto index_iter = table_iter->second.find(index_name);
        if (index_iter == table_iter->second.end()) {
                return DB_INDEX_NOT_FOUND;
        }
        index_id_t index_id = index_iter->second;
        auto index_info_iter = indexes_.find(index_id);
        if (index_info_iter == indexes_.end()) {
                page_id_t index_meta_page_id;
                CatalogManager* self = const_cast<CatalogManager*>(this);
                if (self->catalog_meta_->index_meta_pages_.count(index_id)) {
                        index_meta_page_id = self->catalog_meta_->index_meta_pages_[
                            index_id];
                        if (self->LoadIndex(index_id, index_meta_page_id) == DB_SUCCESS)
                             {
                                index_info_iter = indexes_.find(index_id);
                                if (index_info_iter != indexes_.end()) {
                                        index_info = index_info_iter->second;
                                        return DB_SUCCESS;
                                }
                        }
                }
                return DB_FAILED;
        }
```

```
        index_info = index_info_iter->second;
        return DB_SUCCESS;
}
```

5. 删除表

```
dberr_t CatalogManager::DropTable(const string &table_name) {
        // 检查表是否存在
        TableInfo *table_info = nullptr;
        if (GetTable(table_name, table_info) != DB_SUCCESS) {
                return DB_TABLE_NOT_EXIST;
        }
        // 删除表的所有索引
        std::vector<IndexInfo *> table_indexes;
        GetTableIndexes(table_name, table_indexes);
        for (auto index_info_ptr : table_indexes) {
                DropIndex(table_name, index_info_ptr->GetIndexName());
        }
        // 获取表元数据
        table_id_t table_id = table_info->GetTableId();
        page_id_t table_meta_page_id = catalog_meta_->table_meta_pages_[table_id];
        // 删除表堆（所有数据页）
        if (table_info->GetTableHeap()) {
                table_info->GetTableHeap()->Destroy(buffer_pool_manager_); // This
                    method needs to be implemented in TableHeap
        }
        // 删除表元数据页
        buffer_pool_manager_->DeletePage(table_meta_page_id);
        // 从目录中移除表
        catalog_meta_->table_meta_pages_.erase(table_id);
        table_names_.erase(table_name);
        // 从内存中的 tables_ 映射中移除并删除 TableInfo 对象
        auto it = tables_.find(table_id);
        if (it != tables_.end()) {
                delete it->second; // This should also delete TableMetadata and Schema
                    via TableInfo's destructor
                tables_.erase(it);
        }
        // 刷新目录元数据页
        FlushCatalogMetaPage();
        return DB_SUCCESS;
}
```

6. 删除索引

```
dberr_t CatalogManager::DropIndex(const string &table_name, const string &index_name) {
        // 检查索引是否存在
        IndexInfo *index_info = nullptr;
        if (GetIndex(table_name, index_name, index_info) != DB_SUCCESS) {
                return DB_INDEX_NOT_FOUND;
        }
        // 获取索引 ID
        auto table_iter = index_names_.find(table_name);
        // table_iter is guaranteed to be valid here due to successful GetIndex call
        auto index_iter = table_iter->second.find(index_name);
        // index_iter is guaranteed to be valid here
```

```
        index_id_t index_id = index_iter->second;
        // 删除构成索引的所有数据页
        if (index_info->GetIndex()) {
                index_info->GetIndex()->Destroy();
        }
        // 删除索引元数据页
        page_id_t index_meta_page_id = catalog_meta_->index_meta_pages_[index_id];
        buffer_pool_manager_->DeletePage(index_meta_page_id);
        // 从目录元数据中移除索引
        catalog_meta_->index_meta_pages_.erase(index_id);
        // 从内存中的 index_names_ 映射中移除索引
        table_iter->second.erase(index_name);
        if (table_iter->second.empty()) {
                index_names_.erase(table_name);
        }
        // 从内存中的 indexes_ 映射中移除并删除 IndexInfo 对象
        auto it = indexes_.find(index_id);
        if (it != indexes_.end()) {
                delete it->second;
                indexes_.erase(it);
        }
        // 刷新目录元数据页
        FlushCatalogMetaPage();
        return DB_SUCCESS;
}
```

2.5 PLANNER AND EXECUTOR

2.5.1 模块功能概述

本模块承担 SQL 执行的核心流程。计划生成器（Planner）负责根据语法树生成执行计划，并对语义进行校验；执行器（Executor）执行计划树中的各个节点，完成实际的数据操作。

在系统架构中，SQL 语句首先由解释器（Parser）进行语法解析，生成对应的语法树。该语法树随后由执行计划生成器（Planner）转换为逻辑执行计划，并交由执行器（Executor）执行。执行器在运行过程中，根据执行计划的具体需求调用 Record Manager、Index Manager 和 Catalog Manager，完成数据操作与元信息访问。

其中，Parser、Planner 与 Executer 模块的 SeqScan、Insert、Update、Delete、IndexScan 五个算子由框架实现。Executer 模块的其余函数在本实验中实现。

2.5.2 数据库的创建、查看、切换与删除操作执行器

1. **数据库的创建**

   在实现上，首先由语法树解析数据库名，然后将新数据库插入 `Executor` 对象的 `dbs_` 成员；若已有同名数据库，则提示创建失败。

```
dberr_t ExecuteEngine::ExecuteCreateDatabase(pSyntaxNode ast, ExecuteContext *context) {
        string db_name = ast->child_->val_;
```

```
        if (dbs_.find(db_name) != dbs_.end()) {
                return DB_ALREADY_EXIST;
        }
        dbs_.insert(make_pair(db_name, new DBStorageEngine(db_name, true)));
        cout << "Database "<< db_name <<" created successfully." << endl;
        return DB_SUCCESS;
}
```

2. 数据库的查看

在实现上，访问 Executor 对象的 dbs_ 成员以获取数据库列表，将数据库信息以格式化的方式进行打印。

```
dberr_t ExecuteEngine::ExecuteShowDatabases(pSyntaxNode ast, ExecuteContext *context) {
        if (dbs_.empty()) {
                return DB_NOT_EXIST;
        }
        int max_width = 8;
        for (const auto &itr : dbs_) {
                if (itr.first.length() > max_width) max_width = itr.first.length();
        }
        cout << "+" << setfill('-') << setw(max_width + 2) << "" << "+" << endl;
        cout << "| " << std::left << setfill(' ') << setw(max_width) << "Database" << "
            |" << endl;
        cout << "+" << setfill('-') << setw(max_width + 2) << ""  << "+" << endl;
        for (const auto &itr : dbs_) {
                cout << "| " << std::left << setfill(' ') << setw(max_width) << itr.
                    first << " |" << endl;
        }
        cout << "+" << setfill('-') << setw(max_width + 2) << ""  << "+" << endl;
        return DB_SUCCESS;
}
```

3. 数据库的删除

在实现上，首先由语法树解析数据库名，然后将待删除数据库从 Executor 对象的 dbs_ 成员中移除，并在文件系统中删除对应的数据库文件。

```
dberr_t ExecuteEngine::ExecuteDropDatabase(pSyntaxNode ast, ExecuteContext *context) {
        string db_name = ast->child_->val_;
        if (dbs_.find(db_name) == dbs_.end()) {
                return DB_NOT_EXIST;
        }
        remove(("./databases/" + db_name).c_str());
        delete dbs_[db_name];
        dbs_.erase(db_name);
        if (db_name == current_db_)
        current_db_ = "";
        cout << "Database "<< db_name <<" droped successfully." << endl;
        return DB_SUCCESS;
}
```

4. 数据库的切换

在实现上，首先由语法树解析数据库名，检测数据库的存在性后，直接修改 Executor

对象的 current_db_ 成员。

```
dberr_t ExecuteEngine::ExecuteUseDatabase(pSyntaxNode ast, ExecuteContext *context) {
        string db_name = ast->child_->val_;
        if (dbs_.find(db_name) != dbs_.end()) {
                current_db_ = db_name;
                cout << "Database changed to "<< db_name <<"." << endl;
                return DB_SUCCESS;
        }
        return DB_NOT_EXIST;
}
```

2.5.3 表的创建、查看与删除操作执行器

1. **表的创建**

   在实现上，首先由语法树解析列定义、主键与唯一键标记，然后根据列的数据类型调用正确的构造函数创建 Column 对象，再由 vector<Column*> 对象构建 Schema 对象，并调用 Catalog Manager 提供的创建表接口完成表的创建。此外，若表创建成功，自动为主键和唯一键约束创建索引。

```
dberr_t ExecuteEngine::ExecuteCreateTable(pSyntaxNode ast, ExecuteContext *context) {
        if (current_db_.empty()) {
                cout << "No database selected." << endl;
                return DB_FAILED;
        }
        string table_name = ast->child_->val_;
        // 解析主键
        vector<string> primary_keys;
        pSyntaxNode column_list = ast->child_->next_;
        pSyntaxNode column = column_list->child_;
        for (; column != nullptr && column->type_ != kNodeColumnList; column = column->
            next_) {}
        if (column != nullptr && column->type_ == kNodeColumnList) {
                for (column = column->child_; column != nullptr; column = column->next_)
                     {
                        primary_keys.push_back(column->val_);
                }
        }
        // 解析列定义
        vector<Column *> columns;
        column = column_list->child_;
        int index = 0;
        for (; column != nullptr && column->type_ != kNodeColumnList; column = column->
            next_) {
                bool is_unique = false;
                if (column->val_!=nullptr) is_unique = string(column->val_) == "unique";
                string column_name = column->child_->val_;
                string type_name = column->child_->next_->val_;
                TypeId type;
                uint32_t length = 0;
                // 解析数据类型
                if (type_name == "int") {
                        type = TypeId::kTypeInt;
                } else if (type_name == "float") {
```

```
32                      type = TypeId::kTypeFloat;
33              } else if (type_name == "char") {
34                      type = TypeId::kTypeChar;
35                      length = atoi(column->child_->next_->child_->val_);
36                      if (length <= 0) {
37                              cout << column_name <<"'s length is illegal, failed to
                                    create table."<<endl;
38                              return DB_FAILED;
39                      }
40              } else {
41                      return DB_FAILED;
42              }
43              // 判断主键
44              bool is_primary = false;
45              for (auto& primary_key : primary_keys) {
46                      if (column_name == primary_key) {
47                              is_primary = true;
48                              break;
49                      }
50              }
51              if (type == TypeId::kTypeChar)
52              columns.push_back(new Column(column_name, type, length, index++, !
                    is_primary, is_unique));
53              else
54              columns.push_back(new Column(column_name, type, index++, !is_primary,
                    is_unique));
55      }
56      // 创建表
57      TableInfo *table_info = nullptr;
58      dberr_t res = dbs_[current_db_]->catalog_mgr_->CreateTable(table_name,
59      new Schema(columns),
60      nullptr,
61      table_info);
62      // 如果表创建成功，自动为主键和unique约束创建索引
63      if (res == DB_SUCCESS) {
64              // 创建主键索引
65              if (!primary_keys.empty()) {
66                      IndexInfo *pk_index_info = nullptr;
67                      string pk_index_name = table_name + "_pk_index";
68                      dberr_t pk_res = dbs_[current_db_]->catalog_mgr_->CreateIndex(
69                      table_name, pk_index_name, primary_keys, nullptr, pk_index_info,
                            "bptree");
70                      if (pk_res != DB_SUCCESS) {
71                              cout << "Warning: Failed to create primary key index for
                                    table " << table_name <<"."<< endl;
72                      }
73              }
74              // 为unique列创建索引
75              for (size_t i = 0; i < columns.size(); i++) {
76                      if (columns[i]->IsUnique()) {
77                              IndexInfo *unique_index_info = nullptr;
78                              string unique_index_name = table_name + "_" + columns[i
                                    ]->GetName() + "_unique_index";
79                              vector<string> unique_keys = {columns[i]->GetName()};
80                              dberr_t unique_res = dbs_[current_db_]->catalog_mgr_->
                                    CreateIndex(
```

```
                                table_name, unique_index_name, unique_keys, nullptr,
                                    unique_index_info, "bptree");
                                if (unique_res != DB_SUCCESS) {
                                        cout << "Warning: Failed to create unique index
                                            for column " << columns[i]->GetName()
                                        << " in table " << table_name << "." <<endl;
                                }
                        }
                }
                cout << "Table "<< table_name <<" created successfully." << endl;
        }
        // 清理内存
        for (auto column : columns) {
                delete column;
        }
        return res;
}
```

2. **表的查看**

在实现上，调用 Catalog Manager 提供的获取表接口完成表的获取，然后将表信息以格式化的方式进行打印。

```
dberr_t ExecuteEngine::ExecuteShowTables(pSyntaxNode ast, ExecuteContext *context) {
        if (current_db_.empty()) {
                cout << "No database selected." << endl;
                return DB_FAILED;
        }
        vector<TableInfo *> tables;
        if (dbs_[current_db_]->catalog_mgr_->GetTables(tables) == DB_FAILED || tables.
            size() == 0) {
                return DB_TABLE_NOT_EXIST;
        }
        string table_in_db("Tables in " + current_db_);
        uint max_width = table_in_db.length();
        for (const auto &itr : tables) {
                if (itr->GetTableName().length() > max_width) max_width = itr->
                    GetTableName().length();
        }
        cout << "+" << setfill('-') << setw(max_width + 2) << ""
        << "+" << endl;
        cout << "| " << std::left << setfill(' ') << setw(max_width) << table_in_db << "
             |" << endl;
        cout << "+" << setfill('-') << setw(max_width + 2) << ""
        << "+" << endl;
        for (const auto &itr : tables) {
                cout << "| " << std::left << setfill(' ') << setw(max_width) << itr->
                    GetTableName() << " |" << endl;
        }
        cout << "+" << setfill('-') << setw(max_width + 2) << ""
        << "+" << endl;
        return DB_SUCCESS;
}
```

3. **表的删除**

在实现上，调用 Catalog Manager 提供的删除表接口完成表的删除。

```
dberr_t ExecuteEngine::ExecuteDropTable(pSyntaxNode ast, ExecuteContext *context) {
        if (current_db_.empty()) {
                cout << "No database selected." << endl;
                return DB_FAILED;
        }
        string table_name = ast->child_->val_;
        dberr_t res = dbs_[current_db_]->catalog_mgr_->DropTable(table_name);
        if (res == DB_SUCCESS) {
                cout << "Table "<< table_name <<" droped successfully." << endl;
        }
        return res;
}
```

2.5.4 索引的创建、查看与删除操作执行器

1. **索引的创建**

在实现上，首先由语法树解析索引列名，然后调用 Catalog Manager 提供的创建索引接口完成索引的创建。

```
dberr_t ExecuteEngine::ExecuteCreateIndex(pSyntaxNode ast, ExecuteContext *context) {
        if (current_db_.empty()) {
                cout << "No database selected." << endl;
                return DB_FAILED;
        }
        string index_name = ast->child_->val_;
        string table_name = ast->child_->next_->val_;
        vector<string> column_names;
        pSyntaxNode column_syntax_node = ast->child_->next_->next_->child_;
        while (column_syntax_node != nullptr) {
                column_names.push_back(column_syntax_node->val_);
                column_syntax_node = column_syntax_node->next_;
        }
        // 创建索引
        IndexInfo *index_info = nullptr;
        dberr_t res = dbs_[current_db_]->catalog_mgr_->CreateIndex(table_name,
            index_name,
        column_names, nullptr, index_info,"");
        if (res == DB_SUCCESS) {
                cout << "Index "<< index_name <<" created successfully." << endl;
        }
        return res;
}
```

2. **索引的查看**

在实现上，调用 Catalog Manager 提供的获取表接口完成所有表的获取，然后对每个表逐个调用 Catalog Manager 提供的获取索引接口完成索引的获取，将索引信息以格式化的方式进行打印。

```
dberr_t ExecuteEngine::ExecuteShowIndexes(pSyntaxNode ast, ExecuteContext *context) {
        if (current_db_.empty()) {
                cout << "No database selected." << endl;
```

```
                return DB_FAILED;
        }

        vector<TableInfo *> tables;
        if (dbs_[current_db_]->catalog_mgr_->GetTables(tables) == DB_FAILED) {
                return DB_INDEX_NOT_FOUND;
        }

        bool has_index = false;
        for (const auto &table : tables) {
                vector<IndexInfo *> indexes;
                if (dbs_[current_db_]->catalog_mgr_->GetTableIndexes(table->GetTableName
                    (), indexes) == DB_SUCCESS) {
                    if (!indexes.empty()) {
                            has_index = true;

                            string reminder = "Indexes in table " + table->
                                GetTableName();
                            uint max_width = reminder.length();
                            for (const auto &index : indexes) {
                                if (index->GetIndexName().length() > max_width)
                                    max_width = index->GetIndexName().length();
                            }
                            cout << "+" << setfill('-') << setw(max_width + 2) << ""
                                  << "+" << endl;
                            cout << "| " << std::left << setfill(' ') << setw(
                                max_width) << reminder << " |" << endl;
                            cout << "+" << setfill('-') << setw(max_width + 2) << ""
                                  << "+" << endl;

                            for (const auto &index : indexes) {
                                cout << "| " << std::left << setfill(' ') <<
                                    setw(max_width) << index->GetIndexName() <<
                                    " |"<< endl;
                            }
                            cout << "+" << setfill('-') << setw(max_width + 2) << ""
                                  << "+" << endl;
                    }
                }
        }

        if (!has_index) {
                return DB_INDEX_NOT_FOUND;
        }
        return DB_SUCCESS;
}
```

3. 索引的删除

在实现上，调用 Catalog Manager 提供的删除索引接口完成索引的删除。

```
dberr_t ExecuteEngine::ExecuteDropIndex(pSyntaxNode ast, ExecuteContext *context) {
        if (current_db_.empty()) {
                cout << "No database selected." << endl;
                return DB_FAILED;
        }

```

```
        string index_name = ast->child_->val_;
        string table_name = ast->child_->next_->val_;

        dberr_t res = dbs_[current_db_]->catalog_mgr_->DropIndex(table_name, index_name)
            ;
        if (res == DB_SUCCESS) {
                cout << "Index "<< index_name <<" droped successfully." << endl;
        }
        return res;
}
```

2.5.5 SQL 批量执行操作执行器

在实现上，首先打开指定的 SQL 文件，然后按分号对语句进行分割，支持跨行语句，随后对每条语句分别调用 `yyparse()` 生成语法树并进行执行。在文件中所有语句完成执行后，输出总和的执行时间。

```
dberr_t ExecuteEngine::ExecuteExecfile(pSyntaxNode ast, ExecuteContext *context) {
        string filename = ast->child_->val_;
        ifstream file(filename);
        if (!file.is_open()) {
                cout << "Cannot open file " << filename << "." << endl;
                return DB_FAILED;
        }
        string line;
        string sql;
        auto start_time = std::chrono::system_clock::now();
        while (getline(file, line)) {
                while (line[line.length() - 1] != ';') {
                        string nextline;
                        getline(file, nextline);
                        line += nextline;
                }
                // 解析并执行SQL语句
                YY_BUFFER_STATE bp = yy_scan_string(line.c_str());
                if (bp == nullptr) {
                        cout << "Failed to scan SQL string." << endl;
                        return DB_FAILED;
                }
                yy_switch_to_buffer(bp);
                yyparse();
                if (MinisqlParserGetError()) {
                        // error
                        printf("%s\n", MinisqlParserGetErrorMessage());
                }
                auto result = Execute(MinisqlGetParserRootNode());
                ExecuteInformation(result);
                // clean memory after parse
                MinisqlParserFinish();
                yy_delete_buffer(bp);
                yylex_destroy();
        }
        file.close();
        auto stop_time = std::chrono::system_clock::now();
```

```
38          double duration_time =
39          double((std::chrono::duration_cast<std::chrono::milliseconds>(stop_time - start_time))
                .count());
40          cout << "File executes finished(" << fixed << setprecision(4) << duration_time / 1000
                << " sec)." << std::endl;
41          return DB_SUCCESS;
42  }
```

3 系统测试与验收

3.1 正确性测试

正确性测试通过测试用例进行评判。本地测试中，程序已通过实现模块的全部测试用例，验证了其功能实现的正确性。由于本实验为单人完成，时间匆忙，心力交瘁，未能自行设计测试代码对正确性进行测试，恳请助教老师理解。

程序已通过的测试用例列表：

DISK AND BUFFER POOL MANAGER:

1. TEST(BufferPoolManagerTest, BinaryDataTest)
2. TEST(LRUReplacerTest, SampleTest)
3. TEST(DiskManagerTest, BitMapPageTest)
4. TEST(DiskManagerTest, FreePageAllocationTest)

RECORD MANAGER:

1. TEST(TableHeapTest, TableHeapSampleTest)
2. TEST(TupleTest, FieldSerializeDeserializeTest)
3. TEST(TupleTest, RowTest)

INDEX MANAGER:

1. TEST(BPlusTreeTests, BPlusTreeIndexGenericKeyTest)
2. TEST(BPlusTreeTests, BPlusTreeIndexSimpleTest)
3. TEST(BPlusTreeTests, SampleTest)
4. TEST(BPlusTreeTests, IndexIteratorTest)

CATALOG MANAGER:

1. TEST(CatalogTest, CatalogMetaTest)
2. TEST(CatalogTest, CatalogTableTest)
3. TEST(CatalogTest, CatalogIndexTest)

PLANNER AND EXECUTOR:

1. TEST_F(ExecutorTest, SimpleSeqScanTest)

2. TEST_F(ExecutorTest, SimpleDeleteTest)

3. TEST_F(ExecutorTest, SimpleRawInsertTest)

4. TEST_F(ExecutorTest, SimpleUpdateTest)

3.2 功能性测试

3.2.1 测试点：数据库的创建、查看、切换与删除操作

1. 执行语句 `show databases;`，显示当前存在的如下数据库。

```
minisql > show databases;
+---------------+
| Database      |
+---------------+
| test          |
| db_70000_copy |
| db            |
+---------------+
minisql >
```

2. 执行语句 `create database test_2;`，创建一个新的数据库，然后执行语句 `show databases;`，可以查看到新的数据库 `test_2`。

```
minisql > create database test_2;
Database test_2 created successfully.
minisql > show databases;
+---------------+
| Database      |
+---------------+
| test_2        |
| test          |
| db_70000_copy |
| db            |
+---------------+
minisql >
```

3. 执行语句 `drop database test_2;`，删除刚刚创建的数据库，然后执行语句 `show databases;`，可以查看到数据库 `test_2` 不再显示在列表中。

```
minisql > show databases;
+---------------+
| Database      |
+---------------+
| test_2        |
| test          |
| db_70000_copy |
| db            |
+---------------+
minisql > drop database test_2;
Database test_2 droped successfully.
minisql > show databases;
+---------------+
| Database      |
+---------------+
| test          |
| db_70000_copy |
| db            |
+---------------+
```

4. 执行语句 `use test;`，将使用的数据库切换至 `test`。

```
minisql > use test;
Database changed to test.
minisql [test] >
```

3.2.2 测试点：表的创建、查看与删除操作

1. 执行下述语句，创建一个新的表，然后执行语句 `show tables;`，可以查看到新的表 `account`。

```
create table account(
id int,
name char(16) unique,
balance float,
primary key(id)
);
```

```
minisql [test] > create table account(
        id int,
        name char(16) unique,
        balance float,
        primary key(id)
);
Table account created successfully.
minisql [test] > show tables;
+----------------+
| Tables in test |
+----------------+
| account        |
+----------------+
minisql [test] >
```

2. 执行语句 `drop table account;`,删除刚刚创建的表,然后执行语句 `show tables;`，可以查看到表 `account` 不再显示在列表中。

```
minisql [test] > show tables;
+----------------+
| Tables in test |
+----------------+
| account        |
+----------------+
minisql [test] > drop table account;
Table account droped successfully.
minisql [test] > show tables;
Table not exists.
minisql [test] >
```

3.2.3 测试点：索引的创建、查看与删除操作

1. 执行下述语句，创建一个新的表，然后执行语句 `show tables;`，可以查看到新的表 `account`。

```
create table account(
id int,
name char(16) unique,
balance float,
primary key(id)
);
```

```
minisql [test] > create table account(
        id int,
        name char(16) unique,
        balance float,
        primary key(id)
);
Table account created successfully.
minisql [test] > show tables;
+----------------+
| Tables in test |
+----------------+
| account        |
+----------------+
minisql [test] >
```

2. 执行语句 `show indexes;`，查看到表 `account` 具有两个自动创建的索引，分别对应主键约束和属性 `name` 的唯一性约束。

```
minisql [test] > show indexes;
+---------------------------+
| Indexes in table account  |
+---------------------------+
| account_name_unique_index |
| account_pk_index          |
+---------------------------+
minisql [test] >
```

3. 执行语句 `drop index account_name_unique_index on account;`，删除为属性 `name` 自动创建的索引，然后执行语句 `show indexes;`，可以查看到表 `account` 的索引 `account_name_unique_index` 不再显示在列表中。

```
minisql [test] > drop index account_name_unique_index on account;
Index account_name_unique_index droped successfully.
minisql [test] > show indexes;
+---------------------------+
| Indexes in table account |
+---------------------------+
| account_pk_index         |
+---------------------------+
```

4. 执行语句 `create index idx01 on account(name);`，为属性 `name` 重新创建索引，然后执行语句 `show indexes;`，查看到表 `account` 上新创建的索引 `idx01`。

```
minisql [test] > show indexes;
+--------------------------+
| Indexes in table account |
+--------------------------+
| account_pk_index         |
+--------------------------+
minisql [test] > create index idx01 on account(name);
Index idx01 created successfully.
minisql [test] > show indexes;
+--------------------------+
| Indexes in table account |
+--------------------------+
| idx01                    |
| account_pk_index         |
+--------------------------+
minisql [test] >
```

3.2.4 测试点：SQL 批量执行与数据插入操作

1. 编写如下 Python 脚本，为表 `account` 生成 100000 条测试数据，并保证属性 `id` 和 `name` 的唯一性。

```python
# generate_test_data.py
import random
import string

def generate_unique_strings(n, length=8):
        seen = set()
        while len(seen) < n:
                s = ''.join(random.choices(string.ascii_letters + string.digits, k=
                    length))
                if s not in seen:
                        seen.add(s)
                        yield s

total = 100_000
with open("test_data.sql", "w", encoding="utf-8") as f:
        unique_strings = generate_unique_strings(total)
        for i, rand_str in enumerate(unique_strings, start=1):
                rand_float = round(random.uniform(0, 10000), 2)
                f.write(f"insert into account values({i},\"{rand_str}\",{rand_float});\n
                    ")
```

2. 将生成的 100000 条数据分割为 10 个文本文件，命名格式为`"test_data_i.sql"`，其中 `i` 为 0-9 的整数。其中，`"test_data_0.sql"` 的内容如下所示：

```
insert into account values(1,"s4rw7AKJ",4010.54);
insert into account values(2,"vnwjLIEQ",9055.97);
insert into account values(3,"8olxBxsQ",62.5);
...
insert into account values(9998,"Cbo1VVu7",1650.74);
insert into account values(9999,"pTLsyeny",3552.99);
insert into account values(10000,"1c8a1GPZ",6837.89);
```

3. 分多次执行语句 `execfile "test_data_i.sql";`,在测试中,共分 7 次向表 `account` 插入了 70000 条测试数据。下图显示批量插入前 10000 条数据的用时为 18.6370s。

```
Query OK, 1 row affected(0.0030 sec).
Query OK, 1 row affected(0.0030 sec).
Query OK, 1 row affected(0.0030 sec).
Query OK, 1 row affected(0.0070 sec).
Query OK, 1 row affected(0.0030 sec).
File executes finished(18.6370 sec).
minisql [test] >
```

4. 执行全表查询语句 `select * from account where id > 0;`，显示查询到 70000 条数据，证明数据插入的成功性。

```
| 69990 | ck5FYLQR | 2573.659912 |
| 69991 | 52vyCe68 | 7026.140137 |
| 69992 | cLJDIR2a | 5311.870117 |
| 69993 | BpWYNBkj | 1260.680054 |
| 69994 | CXNbHas7 | 4573.810059 |
| 69995 | QjlMvJV5 | 1642.109985 |
| 69996 | mFUCrwG1 | 1081.680054 |
| 69997 | TeQRJYro | 3376.770020 |
| 69998 | YmmOsobF | 3264.189941 |
| 69999 | lExATKjz | 1292.729980 |
| 70000 | WDWdLAes | 632.520020  |
+-------+----------+-------------+
70000 row in set(13.5710 sec).
minisql [db] >
```

3.2.5 测试点：点查询、条件查询与投影操作

1. 执行点查询语句 `select * from account where id = 32768;`,查询到对应数据。

```
minisql [db] > select * from account where id = 32768;
+-------+----------+-------------+
| id    | name     | balance     |
+-------+----------+-------------+
| 32768 | V1qdAsoA | 7346.540039 |
+-------+----------+-------------+
1 row in set(0.0210 sec).
minisql [db] >
```

2. 执行点查询语句 `select * from account where balance = 1260.680054;`，查询到对应数据。

```
minisql [db] > select * from account where balance = 1260.680054;
+-------+----------+-------------+
| id    | name     | balance     |
+-------+----------+-------------+
| 69993 | BpWYNBkj | 1260.680054 |
+-------+----------+-------------+
1 row in set(0.8370 sec).
minisql [db] >
```

3. 执行点查询语句 `select * from account where name = "ck5FYLQR";`，查询到对应数据。

```
minisql [db] > select * from account where name = "ck5FYLQR";
+-------+----------+-------------+
| id    | name     | balance     |
+-------+----------+-------------+
| 69990 | ck5FYLQR | 2573.659912 |
+-------+----------+-------------+
1 row in set(0.0060 sec).
minisql [db] >
```

4. 执行多条件投影查询语句 `select id, name from account where id > 10000 and id < 20000 and name < "01234";`，查询到对应数据。

```
minisql [db] > select id, name from account where id > 10000 and id < 20000 and name < "01234";
+-------+----------+
| id    | name     |
+-------+----------+
| 18950 | 00Vnp3C6 |
+-------+----------+
1 row in set(2.0240 sec).
minisql [db] >
```

3.2.6 测试点：表数据的更新与删除操作

1. 执行表数据更新语句 `update account set id = 70001, balance = 99999 where name = "00Vnp3C6";`,然后执行表查询语句 `select * from account where name = "00Vnp3C6";`，观察到对应数据被成功更新。

```
minisql [db] > update account set id = 70001, balance = 99999 where name = "00Vnp3C6";
Query OK, 1 row affected(15.3010 sec).
minisql [db] > select * from account where name = "00Vnp3C6";
+-------+----------+--------------+
| id    | name     | balance      |
+-------+----------+--------------+
| 70001 | 00Vnp3C6 | 99999.000000 |
+-------+----------+--------------+
1 row in set(15.3160 sec).
minisql [db] >
```

2. 执行表数据删除语句 `delete from account where balance < 100;`，然后执行表查询语句 `select * from account where balance < 100;`，观察到对应数据被成功删除。

```
minisql [db] > delete from account where balance < 100;
Query OK, 697 row affected(18.0250 sec).
minisql [db] > select * from account where balance < 100;
Empty set(16.2670 sec).
minisql [db] >
```

3.2.7 测试点：主键和唯一性约束测评

1. 执行表数据插入语句 `insert into account values(65535, "lhyn", 525.525);`，由于属性 `id` 为主键，且 `id = 65535` 的数据已经存在，因此提示执行失败。

```
minisql [db] > insert into account values(65535, "lhyn", 525.525);
Violating primary key or unique key constraints, failed to insert tuples.
Query OK, 0 row affected(0.0010 sec).
minisql [db] >
```

2. 执行表数据插入语句 `insert into account values(80000, "s4rw7AKJ", 525.525);`，由于属性 `name` 具有唯一性约束，且 `name = "s4rw7AKJ"` 的数据已经存在，因此提示执行失败。

```
minisql [db_70000] > insert into account values(80000, "s4rw7AKJ", 525.525);
Violating primary key or unique key constraints, failed to insert tuples.
Query OK, 0 row affected(0.0000 sec).
minisql [db_70000] >
```

3.2.8 测试点：索引效果的定量测评

1. 在建立和未建立索引的不同数据库上分别执行表数据查询语句 `select * from account where name = "s4rw7AKJ";`。观察得：在建立索引的表上查询用时为 0.0090s，在未建立索引的表上查询用时为 0.7300s，证明索引在查询中发挥作用。

```
minisql [db_70000] > select * from account where name = "s4rw7AKJ";
+----+----------+-------------+
| id | name     | balance     |
+----+----------+-------------+
| 1  | s4rw7AKJ | 4010.540039 |
+----+----------+-------------+
1 row in set(0.7300 sec).
minisql [db_70000] > use db_70000_copy;
Database changed to db_70000_copy.
minisql [db_70000_copy] > select * from account where name = "s4rw7AKJ";
+----+----------+-------------+
| id | name     | balance     |
+----+----------+-------------+
| 1  | s4rw7AKJ | 4010.540039 |
+----+----------+-------------+
1 row in set(0.0090 sec).
minisql [db_70000_copy] >
```

2. 在建立和未建立索引的不同数据库上分别执行表数据查询语句 `select id, name from account where id > 10000 and id < 20000 and name < "01234";`。观察得：在建立索引的表上查询用时为 0.7980s，在未建立索引的表上查询用时为 0.8950s，证明索引在查询中发挥作用。

```
minisql [db_70000] > select id, name from account where id > 10000 and id < 20000 and name < "01234";
+-------+----------+
| id    | name     |
+-------+----------+
| 18950 | 00Vnp3C6 |
+-------+----------+
1 row in set(0.8950 sec).
minisql [db_70000] > use db_70000_copy;
Database changed to db_70000_copy.
minisql [db_70000_copy] > select id, name from account where id > 10000 and id < 20000 and name < "01234";
+-------+----------+
| id    | name     |
+-------+----------+
| 18950 | 00Vnp3C6 |
+-------+----------+
1 row in set(0.7980 sec).
minisql [db_70000_copy] >
```