实验 2：记录管理器实现

一、实验目的

1. 掌握 Rucbase 记录管理器的实现方法。

二、相关知识

1. 分槽页面布局
2. 堆文件组织

三、实验内容

本实验包括 2 项任务。

任务 1：记录操作实现

补全 RMFileHandle 类，实现文件记录的获取、插入、删除和修改操作。

每个 RMFileHandle 对象对应一个文件，当 RMManager 执行打开文件操作时，会创建一个指向 RMFileHandle 对象的指针。

一、RmFileHandle 类方法实现

RmFileHandle 类负责文件级别的记录操作，包括记录的增删改查以及页面管理。每个 RmFileHandle 对象对应一个表数据文件。

1.1 get_record()

1. 方法声明

```
std::unique_ptr<RmRecord> get_record(const Rid& rid, Context* context) const;
```

方法名: get_record

返回类型: std::unique_ptr<RmRecord>

功能: 获取表中指定记录号(Rid)对应的记录

参数列表:

	rid	**context**
类型	const Rid&	Context*
含义	记录号，包含页面号(page_no)和槽位号(slot_no)	上下文信息（暂未使用）

返回值: 返回指向 RmRecord 的智能指针，若指定位置无记录则返回 nullptr

2. 方法实现思路

1. 调用 `fetch_page_handle(rid.page_no)`获取记录所在页面的页面句柄

2. 使用 `Bitmap::is_set(page_handle.bitmap, rid.slot_no)`检查该槽位的位图标记
3. 如果位图标记为 0（无记录），直接返回 nullptr
4. 如果位图标记为 1（有记录），通过 `page_handle.get_slot(rid.slot_no)`获取槽位数据的内存地址
5. 创建 RmRecord 对象，将槽位数据和记录大小作为参数
6. 使用 `std::make_unique` 返回智能指针

代码位置: `src/record/rm_file_handle.cpp:19-36`

3. 方法实现难点

难点: 理解 RmPageHandle 的内存布局和槽位地址计算

解决方法: RmPageHandle 结构体已经封装了 `get_slot()`方法，该方法通过 `slots + slot_no * file_hdr->record_size` 计算出指定槽位的起始地址。只需直接调用即可，无需手动计算偏移量。

1.2 insert_record()

1. 方法声明

```
Rid insert_record(char* buf, Context* context);
```

方法名: insert_record

返回类型: Rid

功能: 在表中插入一条新记录，系统自动选择插入位置

参数列表:

	buf	context
类型	char*	Context*
含义	指向要插入的记录数据的指针	上下文信息（暂未使用）

返回值: 返回插入记录的 Rid（记录号）

2. 方法实现思路

7. 调用 `create_page_handle()`获取一个有空闲空间的页面（该方法会自动判断是使用现有空闲页还是创建新页）
8. 使用 `Bitmap::first_bit(false, page_handle.bitmap, file_hdr_.num_records_per_page)`找到第一个位图值为 0 的槽位（即空闲槽位）
9. 通过 `page_handle.get_slot(slot_no)`获取槽位地址，使用 `memcpy()`将 buf 的数据复制到该槽位
10. 调用 `Bitmap::set(page_handle.bitmap, slot_no)`将位图对应位置置 1，标记槽位已使用
11. 更新页面头：`page_handle.page_hdr->num_records++`

12. 检查页面是否已满：如果 `page_hdr->num_records == file_hdr_.num_records_per_page`，说明页面已满，需要更新文件头的 `first_free_page_no` 为当前页的 `next_free_page_no`，将当前页从空闲链表中移除
13. 调用 `buffer_pool_manager_->unpin_page()`并传入 true 标记页面为脏页
14. 返回 Rid{page_no, slot_no}

代码位置: `src/record/rm_file_handle.cpp:44-80`

3. 方法实现难点

难点 1：正确维护空闲页面链表

当插入记录后页面变满时，需要将该页面从空闲链表中移除。具体操作是将文件头的 `first_free_page_no` 更新为当前页面的 `next_free_page_no`。

解决方法：在更新页面记录数后，增加判断条件：

```
if (page_handle.page_hdr->num_records == file_hdr_.num_records_per_page) {
    file_hdr_.first_free_page_no = page_handle.page_hdr->next_free_page_no;
}
```

难点 2：理解堆文件组织方式

堆文件不关心记录插入的具体位置，只需找到任意有空闲空间的页面即可。因此使用 `create_page_handle()`统一获取可用页面，简化了实现。

1.3 delete_record()

1. 方法声明

```
void delete_record(const Rid& rid, Context* context);
```

方法名: delete_record

返回类型: void

功能：删除表中指定记录号对应的记录

参数列表:

	rid	context
类型	const Rid&	Context*
含义	要删除的记录号	上下文信息（暂未使用）

2. 方法实现思路

15. 调用 `fetch_page_handle(rid.page_no)`获取记录所在页面
16. 使用 `Bitmap::is_set()`检查该位置是否有记录，若无则直接返回
17. 在删除前，记录页面当前是否已满：`bool was_full = (page_hdr->num_records == file_hdr_.num_records_per_page)`

18. 调用 `Bitmap::reset(page_handle.bitmap, rid.slot_no)`将位图对应位置置 0，标记槽位为空闲
19. 更新页面头：`page_handle.page_hdr->num_records--`
20. 判断页面是否从"已满"变为"未满"：如果 `was_full` 为 true，说明删除前页面已满，删除后变为未满，需要调用 `release_page_handle(page_handle)`将页面重新加入空闲链表
21. 调用 `buffer_pool_manager_->unpin_page()`并传入 true 标记页面为脏页

代码位置: `src/record/rm_file_handle.cpp:96-127`

3. 方法实现难点

难点：判断何时需要将页面加入空闲链表

只有当页面从"已满"状态变为"未满"状态时，才需要调用 `release_page_handle()`。如果页面本来就未满，删除记录后仍然未满，不需要额外操作（因为它已经在空闲链表中）。

解决方法：在删除操作前先记录页面是否已满的状态，删除后根据这个状态判断：

```
bool was_full = (page_handle.page_hdr->num_records == file_hdr_.num_records_per_page);
// ... 执行删除操作 ...
if (was_full) {
    release_page_handle(page_handle);
}
```

1.4 update_record()

1. 方法声明

```
void update_record(const Rid& rid, char* buf, Context* context);
```

方法名: update_record

返回类型: void

功能：更新表中指定记录号对应的记录数据

参数列表:

	rid	buf	context
类型	const Rid&	char*	Context*
含义	要更新的记录号	新记录的数据指针	上下文信息（暂未使用）

2. 方法实现思路

22. 调用 `fetch_page_handle(rid.page_no)`获取记录所在页面
23. 使用 `Bitmap::is_set()`检查该位置是否有记录，若无则直接返回
24. 通过 `page_handle.get_slot(rid.slot_no)`获取槽位地址

25. 使用 `memcpy(slot_data, buf, file_hdr_.record_size)`将新数据覆盖旧数据

26. 调用 `buffer_pool_manager_->unpin_page()`并传入 true 标记页面为脏页

代码位置: `src/record/rm_file_handle.cpp:136-156`

3. 方法实现难点

本方法实现较为简单，无特殊难点。只需注意：

- 更新操作不改变记录数量，因此不需要更新 `num_records`
- 不需要修改位图
- 不会影响页面的满/未满状态，不需要维护空闲链表

1.5 fetch_page_handle()

1. 方法声明

```
RmPageHandle fetch_page_handle(int page_no) const;
```

方法名: fetch_page_handle

返回类型: RmPageHandle

功能: 获取指定页面号对应的页面句柄

参数列表:

	page_no
类型	int
含义	页面号

返回值: 返回对应的 RmPageHandle 对象

2. 方法实现思路

27. 检查页面号的有效性：如果 `page_no >= file_hdr_.num_pages`，说明页面号超出范围，抛出 `PageNotExistError` 异常

28. 构造 PageId 对象：`PageId{fd_, page_no}`，其中 fd_是文件描述符

29. 调用 `buffer_pool_manager_->fetch_page(PageId{fd_, page_no})`从缓冲池获取页面

30. 创建 RmPageHandle 对象：`RmPageHandle(&file_hdr_, page)`

31. 返回 RmPageHandle 对象（RmPageHandle 的构造函数会自动解析页面内部结构，设置 page_hdr、bitmap、slots 等指针）

代码位置: `src/record/rm_file_handle.cpp:166-182`

3. 方法实现难点

难点：理解 RmPageHandle 的构造过程

RmPageHandle 构造函数会根据文件头信息，自动计算页面内部各部分的地址：

- `page_hdr` 指向页面头
- `bitmap` 指向位图区域
- `slots` 指向记录槽位区域

这些计算已经封装在 RmPageHandle 构造函数中，只需传入正确的参数即可。

1.6 create_new_page_handle()

1. 方法声明

```
RmPageHandle create_new_page_handle();
```

方法名: create_new_page_handle

返回类型: RmPageHandle

功能：创建一个全新的页面句柄，会在磁盘上分配新页面

返回值：返回新创建的 RmPageHandle 对象

2. 方法实现思路

32. 构造新页面的 PageId：`PageId new_page_id = {fd_, file_hdr_.num_pages}`，页面号为当前页面总数（即下一个可用页面号）
33. 调用 `buffer_pool_manager_->new_page(&new_page_id)`在缓冲池中创建新页面
34. 创建 RmPageHandle 对象：`RmPageHandle page_handle(&file_hdr_, page)`
35. 初始化页面头：
    - `page_handle.page_hdr->next_free_page_no = RM_NO_PAGE`（-1，表示没有下一个空闲页）
    - `page_handle.page_hdr->num_records = 0`（记录数初始化为 0）
36. 初始化位图：调用 `Bitmap::init(page_handle.bitmap, file_hdr_.bitmap_size)`将位图全部置 0
37. 更新文件头：
    - `file_hdr_.num_pages++`（页面总数加 1）
    - `file_hdr_.first_free_page_no = page->get_page_id().page_no`（新页面成为第一个空闲页）
38. 返回 page_handle

代码位置: `src/record/rm_file_handle.cpp:188-214`

3. 方法实现难点

难点: 正确初始化新页面的元数据

新页面分配后，必须初始化以下内容:

- 页面头的两个字段（next_free_page_no 和 num_records）
- 位图（全部置 0 表示所有槽位空闲）
- 文件头的页面总数和第一个空闲页号

解决方法: 按照顺序依次初始化，确保不遗漏任何字段。特别注意新页面自动成为第一个空闲页，因为它完全空闲。

1.7 create_page_handle()

1. 方法声明

```
RmPageHandle create_page_handle();
```

方法名: create_page_handle

返回类型: RmPageHandle

功能: 创建或获取一个有空闲空间的页面句柄。如果有空闲页则返回空闲页，否则创建新页

返回值: 返回有空闲空间的 RmPageHandle 对象

2. 方法实现思路

39. 检查文件头的 `first_free_page_no` 字段
40. 如果 `first_free_page_no == RM_NO_PAGE`（值为-1），说明当前没有空闲页，调用 `create_new_page_handle()`创建新页面并返回
41. 如果 `first_free_page_no` 不为-1，说明存在空闲页，调用 `fetch_page_handle(file_hdr_.first_free_page_no)`获取第一个空闲页并返回

代码位置: `src/record/rm_file_handle.cpp:222-238`

3. 方法实现难点

本方法实现简单，无特殊难点。这是一个统一的接口方法，封装了"获取空闲页"的逻辑，调用者无需关心是使用现有空闲页还是创建新页，由该方法自动判断。

1.8 release_page_handle()

1. 方法声明

```
void release_page_handle(RmPageHandle& page_handle);
```

方法名: release_page_handle

返回类型: void

功能: 当页面从已满状态变为未满状态时调用，将页面重新加入空闲页面链表

参数列表:

	page_handle
类型	RmPageHandle&
含义	要释放的页面句柄引用

2. 方法实现思路

采用头插法将页面插入到空闲链表头部:

42. 将当前页面的 next_free_page_no 设置为文件头的 first_free_page_no:
    `page_handle.page_hdr->next_free_page_no = file_hdr_.first_free_page_no`
43. 将文件头的 first_free_page_no 更新为当前页面的页面号:
    `file_hdr_.first_free_page_no = page_handle.page->get_page_id().page_no`

这样，当前页面就成为了空闲链表的第一个节点。

代码位置: src/record/rm_file_handle.cpp:243-256

3. 方法实现难点

难点: 理解空闲页面链表的维护机制

空闲页面链表是通过文件头和页面头中的页面号字段构成的单向链表:

- 文件头的 first_free_page_no 指向链表头
- 每个页面头的 next_free_page_no 指向链表中的下一个节点
- RM_NO_PAGE（-1）表示链表结束

解决方法: 采用头插法是最简单的方式，只需要两步操作即可完成插入。这种方法的时间复杂度为 O(1)。

任务 2：记录迭代器实现

补全 RmScan 类，实现对文件记录的遍历。

二、RmScan 类方法实现

RmScan 类负责遍历文件中的所有记录，提供迭代器功能。它内部维护一个 Rid 指向当前扫描位置。

2.1 RmScan() 构造函数

1. 方法声明

```
RmScan(const RmFileHandle *file_handle);
```

方法名: RmScan（构造函数）

功能: 初始化记录扫描器，设置扫描的起始位置为第一个存放了记录的位置

参数列表:

	file_handle
类型	const RmFileHandle*
含义	要扫描的文件句柄指针

2. 方法实现思路

1. 保存文件句柄指针到成员变量 `file_handle_`
2. 初始化内部 Rid 为第一个数据页的第一个槽位之前的位置:
    - `rid_.page_no = RM_FIRST_RECORD_PAGE`（值为 1，因为 0 号页是文件头页）
    - `rid_.slot_no = -1`（设为-1 是为了让 next()从第 0 个槽位开始查找）
3. 调用 `next()`方法找到第一个实际存放了记录的位置

代码位置: `src/record/rm_scan.cpp:18-29`

3. 方法实现难点

难点: 理解为什么 slot_no 要初始化为-1

因为 `next()`方法会从 `rid_.slot_no + 1` 开始查找，所以如果要从第 0 个槽位开始查找，就需要将 `slot_no` 初始化为-1。

2.2 next()

1. 方法声明

```
void next() override;
```

方法名: next

返回类型: void

功能: 将扫描器移动到下一个存放了记录的位置

2. 方法实现思路

使用循环遍历页面和槽位:

1. 外层循环：遍历所有数据页，条件为 `rid_.page_no < file_handle_->file_hdr_.num_pages`

2. 对于当前页面：
    - 调用 `file_handle_->fetch_page_handle(rid_.page_no)`获取页面句柄
    - 调用 `Bitmap::next_bit(true, page_handle.bitmap, num_records_per_page, rid_.slot_no)`查找下一个位图值为 1 的槽位（true 表示查找值为 1 的位）
    - 释放页面：`buffer_pool_manager_->unpin_page(..., false)`（false 表示未修改页面）
    - 如果找到的 slot_no 小于每页记录数，说明找到了记录，更新 `rid_.slot_no` 并返回
    - 如果没找到，说明当前页面没有更多记录
3. 移动到下一个页面：
    - `rid_.page_no++`
    - `rid_.slot_no = -1`（重置为-1，从下一页的第 0 个槽位开始）
4. 如果所有页面都遍历完，设置末尾标记：
    - `rid_.page_no = RM_NO_PAGE`（-1）
    - `rid_.slot_no = -1`

代码位置: `src/record/rm_scan.cpp:34-65`

3. 方法实现难点

难点 1: 跨页面扫描的逻辑

需要正确处理页面边界：当当前页面没有更多记录时，移动到下一个页面并重置 slot_no 为-1。

解决方法: 使用外层 while 循环遍历页面，内层使用 `Bitmap::next_bit()`查找记录。每次移动到新页面时重置 slot_no。

难点 2: 正确释放页面

每次获取页面后必须释放（unpin），否则会导致缓冲池资源耗尽。

解决方法: 在每次循环中，使用完页面后立即调用 `unpin_page()`，传入 false 表示未修改页面。

2.3 is_end()

1. 方法声明

```
bool is_end() const override;
```

方法名: is_end

返回类型: bool

功能: 判断扫描器是否已经到达文件末尾

返回值：如果到达末尾返回 true，否则返回 false

2. 方法实现思路

1. 检查当前 rid 的 page_no 是否等于 RM_NO_PAGE（值为-1）
2. 如果相等，说明已经扫描完所有记录，返回 true
3. 否则返回 false

代码位置：`src/record/rm_scan.cpp:70-76`

3. 方法实现难点

本方法实现简单，无特殊难点。只需要判断一个条件即可。RM_NO_PAGE 是在 next()方法中设置的末尾标记。

四、实验总结

任务 1 核心要点：

1. **get_record**：检查位图 → 获取槽位数据 → 返回记录
2. **insert_record**：找空闲页 → 找空闲槽位 → 复制数据 → 更新元数据 → 处理页面满的情况
3. **delete_record**：重置位图 → 更新记录数 → 处理页面从满到未满的情况
4. **update_record**：直接覆盖数据即可
5. **fetch_page_handle**：通过缓冲池获取页面
6. **create_new_page_handle**：分配新页 → 初始化元数据 → 更新文件头
7. **create_page_handle**：优先使用空闲页，无空闲页则创建新页
8. **release_page_handle**：头插法加入空闲链表

任务 2 核心要点：

1. **RmScan 构造**：从第一个数据页开始扫描
2. **next**：在页内查找 → 跨页查找 → 使用 RM_NO_PAGE 标记结束
3. **is_end**：判断 page_no 是否为 RM_NO_PAGE

注意事项：

1. 所有修改页面的操作都要 unpin 页面并标记为脏页
2. 页面从满到未满时必须调用 release_page_handle
3. 插入操作导致页面满时要更新 first_free_page_no
4. 扫描时要正确释放（unpin）获取的页面
5. 使用 RM_NO_PAGE (-1)作为无效页面标记

技术收获

通过 Lab2 的实现，不仅掌握了记录管理器的核心技术，还深入理解了数据库系统中数据组织和管理的基本原理。位图、空闲链表、缓冲池管理等技术在实际开发中都有广泛应用，这次实践经验将为后续学习更复杂的数据库组件打下坚实基础。