《数据库系统实验》课程设计报告

说明：本模板为“Rucbase 实验配置与存储管理”课程设计专用。

题目	Rucbase 实验配置与存储管理		
小组成员信息			
姓名	学号	班级	分工
杨子昂	21307181	计算机科学与技术（人工智能）	任务 1.3 缓冲池管理器

提交时间：　　年　　月　　日

一.　开发环境与开发工具

具体实验目的

梁铭恩

二.　具体模块设计（40 分）

各子任务的核心代码（一定要加适当注释），包括：

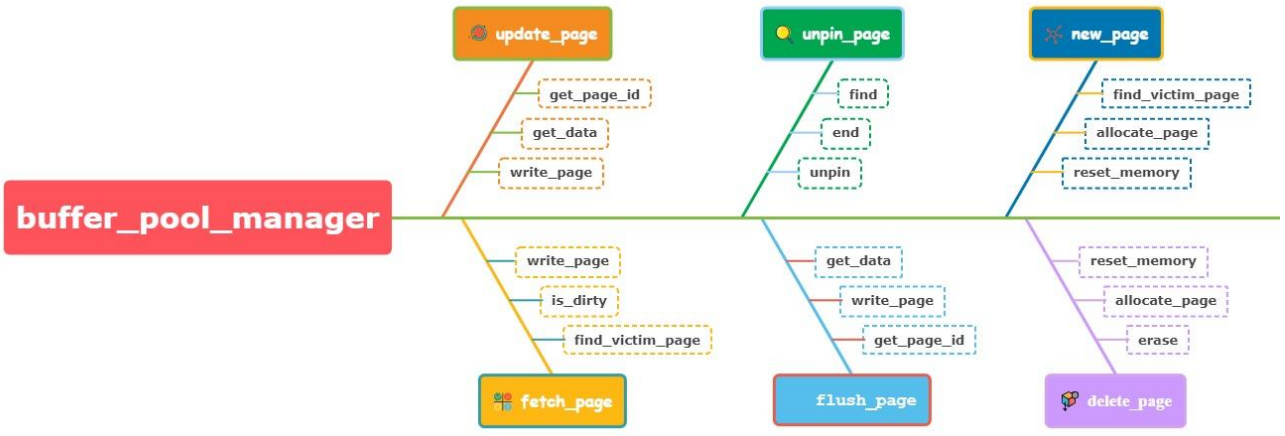
任务一 缓冲池管理器

任务 1.1 磁盘存储管理器（10 分）张序

任务 1.2 缓冲池替换策略（10 分）梁铭恩

任务 1.3 缓冲池管理器（10 分） 杨子昂

Overview：



1. bool BufferPoolManager::find_victim_page(frame_id_t* frame_id): 用于在缓冲池管理器中查找一个可被替换的页面。参数是指向 frame_id 的指针，返回值为布尔类型，表示是否找到了合适的页面。

2. void BufferPoolManager::update_page(Page *page, PageId new_page_id, frame_id_t new_frame_id): 用于更新缓冲池中的某个页面。接受页面指针、新的页面 ID 和帧 ID 作为参数，无返回值。

3. Page* BufferPoolManager::fetch_page(PageId page_id): 用于获取缓冲池中的指定页面。参数为页面 ID，返回值为指向该页面的指针。

4. bool BufferPoolManager::unpin_page(PageId page_id, bool is_dirty): 用于“取消固定”缓冲池中的一个页面。接受页面 ID 和布尔值（表示页面是否已被修改）作为参数，返回布尔值表示操作是否成功。

5. bool BufferPoolManager::flush_page(PageId page_id): 用于将缓冲池中的特定页面刷新到存储中。参数为页面 ID，返回布尔值表示操作是否成功。

6. Page* BufferPoolManager::new_page(PageId* page_id): 用于在缓冲池中创建一个新页面。参数为指向页面 ID 的指针，返回值为指向新创建页面的指针。

7. bool BufferPoolManager::delete_page(PageId page_id): 用于删除缓冲池中的一个页面。参数为页面 ID，返回布尔值表示操作是否成功。

8. void BufferPoolManager::flush_all_pages(int fd): 用于将缓冲池中的所有页面刷新到存储中。参数为文件描述符，无返回值。

在该部分实验中，我们主要需要实现上面这 8 个函数，并通过最终的测试程序。

关键代码展示：

由于一共有八个函数且基本上每一个函数都是表层调用的级别，因此都十分重要，并且也会需要频繁调用前面两位同学的函数，比较繁琐而且代码篇幅很长，因此我会从中挑选几个比较有代表性的难度较大的函数进行分析：

（1）fetch_page

```cpp
Page* BufferPoolManager::fetch_page(PageId page_id) {
    //Todo:
    // 1.     从page_table_中搜寻目标页
    // 1.1    若目标页有被page_table_记录，则将其所在frame固定(pin)，并返回目标页。
    // 1.2    否则，尝试调用find_victim_page获得一个可用的frame，若失败则返回nullptr
    // 2.     若获得的可用frame存储的为dirty page，则须调用updata_page将page写回到磁盘
    // 3.     调用disk_manager_的read_page读取目标页到frame
    // 4.     固定目标页，更新pin_count_
    // 5.     返回目标页


    std::scoped_lock lock{latch_};
    assert(page_id.page_no != INVALID_PAGE_ID);

    auto it = page_table_.find(page_id);
    if (it != page_table_.end()) {  // Page found in the page table
        frame_id_t frame_id = it->second;
        replacer_->pin(frame_id);
        pages_[frame_id].pin_count_++;
        return &pages_[frame_id];
    } else {  // Page not in page table
        frame_id_t frame_id;
        if (find_victim_page(&frame_id)) {  // Found a victim page to replace
            Page& victim_page = pages_[frame_id];
            if (victim_page.is_dirty()) {
                disk_manager_->write_page(victim_page.get_page_id().fd, victim_page.get_page_id().page_no, victim_page.data_, PAGE_SIZE);
            }
            page_table_.erase(victim_page.get_page_id());  // Remove victim page from page table

            Page& new_page = pages_[frame_id];
            new_page.id_ = page_id;
            new_page.reset_memory();  // Clear any old data
            disk_manager_->read_page(page_id.fd, page_id.page_no, new_page.data_, PAGE_SIZE);
            new_page.pin_count_ = 1;  // Reset pin count for new page
            page_table_[page_id] = frame_id;  // Add new page to page table
            return &new_page;
        }
    }
    return nullptr;
}
```

上面截图中的函数是 fetch_page 函数

1. 搜索页表：首先在页表中搜索目标页面。
2. 页面存在：如果页面已在页表中，则固定该页面并返回它。
3. 页面不存在：如果页面不在页表中，则尝试找到一个可替换的页面（受害者页面）。
4. 脏页处理：如果找到的页面是脏页（已被修改），则先将其写回磁盘。
5. 读取新页面：从磁盘读取目标页面到缓冲池。
6. 更新页表：更新页表，记录新页面的位置。
7. 返回页面：返回指向新读取页面的指针。

总结来说，这个函数的主要目的是获取缓冲池中的一个指定页面。如果页面已存在于缓冲池中，则直接返回它；如果不存在，则尝试找到一个可用的帧（可能需要替换一个已存在的页面），将所需的页面从磁盘读入该帧，并返回它。这个过程涉及对页面的固定（pinning）、引用计数管理、脏页的写回以及页表的更新。

（2）flush_page

```cpp
bool BufferPoolManager::flush_page(PageId page_id) {
    // Todo:
    // 0. lock latch
    // 1. 查找页表,尝试获取目标页P
    // 1.1 目标页P没有被page_table_记录 ，返回false
    // 2. 无论P是否为脏都将其写回磁盘。
    // 3. 更新P的is_dirty_

    std::scoped_lock lock{latch_};

    auto it = page_table_.find(page_id);
    if (it == page_table_.end()) {
        // Page not found in the page table
        return false;
    }

    Page& page = pages_[it->second];
    if (page.get_page_id().page_no != INVALID_PAGE_ID) {
        disk_manager_->write_page(page.get_page_id().fd, page.get_page_id().page_no, page.get_data(), PAGE_SIZE);
        page.is_dirty_ = false;
        return true;
    }

    return false;
}
```

1. 加锁: 使用 std::scoped_lock 对象 lock 锁定，以确保线程安全。
2. 搜索页表: 在页表 page_table_ 中查找目标页面 page_id。
   如果页面不存在：如果目标页面没有被记录在页表中（it == page_table_.end()），函数返回 false。
3. 写回磁盘:
   获取目标页面的引用（Page& page = pages_[it->second]）。
   无论页面是否脏（即是否被修改过），都将其内容写回磁盘（disk_manager_->write_page）。
   更新页面的脏标志（page.is_dirty_ = false），表示页面已同步到磁盘。
4. 返回结果:
   如果成功执行了写回操作，函数返回 true。
   如果页面的页面号无效（page.get_page_id().page_no != INVALID_PAGE_ID 不成立），函数返回 false。

（3）new_page

```cpp
Page* BufferPoolManager::new_page(PageId* page_id) {
    // 1.   获得一个可用的frame，若无法获得则返回nullptr
    // 2.   在fd对应的文件分配一个新的page_id
    // 3.   将frame的数据写回磁盘
    // 4.   固定frame，更新pin_count_
    // 5.   返回获得的page

    std::scoped_lock lock{latch_};

    frame_id_t frame_id;
    if (!find_victim_page(&frame_id)) {
        return nullptr; // 无法获得可用frame
    }

    // 分配新的page_id
    page_id->page_no = disk_manager_->allocate_page(page_id->fd);

    // 更新页表和页面状态
    page_table_.erase(pages_[frame_id].id_);
    pages_[frame_id].reset_memory();
    pages_[frame_id].id_ = *page_id;
    pages_[frame_id].pin_count_ = 1;
    page_table_[*page_id] = frame_id;

    // 固定新页面
    replacer_->pin(frame_id);

    return &pages_[frame_id];
}
```

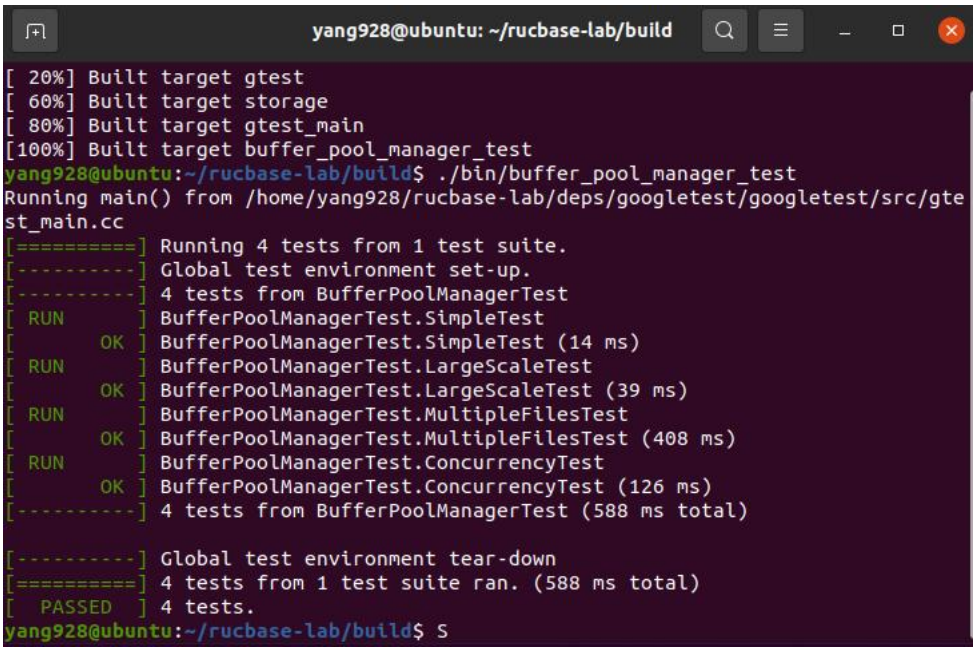
1. 获取可用的帧:
   使用 find_victim_page(&frame_id) 尝试获得一个可用的帧（frame）。如果无法获得（函数返回 false），则函数返回 nullptr。
2. 分配新的页面 ID:
   调用 disk_manager_->allocate_page(page_id->fd) 为新页面分配一个新的 page_id。
3. 更新页表和页面状态:
   从页表 page_table_ 中移除当前帧存储的旧页面的记录。
   重置帧的内存（pages_[frame_id].reset_memory()）。
   将新的页面 ID 赋值给帧（pages_[frame_id].id_ = *page_id）。
   设置新页面的引用计数为 1（pages_[frame_id].pin_count_ = 1）。
   在页表中添加新页面的记录。
4. 固定新页面:
   使用 replacer_->pin(frame_id) 固定新的页面，防止它被其他操作替换。
5. 返回新页面:
   返回指向新分配页面的指针（&pages_[frame_id]）。

（4）delete_page

```cpp
bool BufferPoolManager::delete_page(PageId page_id) {
    // 1.   在page_table_中查找目标页，若不存在返回true
    // 2.   若目标页的pin_count不为0，则返回false
    // 3.   将目标页数据写回磁盘，从页表中删除目标页，重置其元数据，将其加入free_list_，返回true

    std::scoped_lock lock{latch_};

    auto it = page_table_.find(page_id);
    if (it == page_table_.end()) {
        // 页面不在页表中，可以安全地删除
        return true;
    }

    Page& page = pages_[it->second];
    if (page.pin_count_ > 0) {
        // 页面仍然被固定，不能删除
        return false;
    }

    // 删除操作
    page_table_.erase(page_id);
    disk_manager_->deallocate_page(page_id.page_no);
    page.reset_memory();
    free_list_.push_back(it->second);

    return true;
}
```

1. 搜索页表:
   在页表 page_table_ 中查找目标页面 page_id。
   如果页面不存在：如果页表中没有找到目标页面（it == page_table_.end()），函数返回 true，表示页面可以安全删除（因为它本来就不存在于缓冲池中）。
2. 检查页面固定状态:
   获取目标页面的引用（Page& page = pages_[it->second]）。
   如果页面被固定：如果目标页面的 pin_count_ 大于 0（即页面被固定，不能被删除），函数返回 false。
3. 执行删除操作:
   从页表中移除目标页面的记录（page_table_.erase(page_id)）。
   调用 disk_manager_->deallocate_page(page_id.page_no) 在磁盘上释放页面。
   重置页面的内存（page.reset_memory()）。
   将页面的帧 ID 加入到空闲列表 free_list_ 中。
4. 返回成功:
   删除操作完成后，函数返回 true。

三， 功能测试



可以看到全部都是绿色的 pass 测试点。

五. 总结

本课程设计中用到的《数据库系统原理》理论课概念与知识。

缓冲池管理（Buffer Pool Management）：

- 这是数据库管理系统中的一个核心组件，负责管理内存中的数据页，以减少磁盘 I/O 操作。
- 缓冲池提供了一种机制，使得频繁访问的数据可以保留在内存中，而不是每次都从磁盘读取。

页面置换算法（Page Replacement Algorithm）：

- 当缓冲池满时，需要选择一个页面进行替换。这涉及到复杂的算法，比如 LRU（最近最少使用）或其他策略，以确定哪个页面应该被移除。
- 例如，find_victim_page 函数可能就是用来实现这种页面置换策略的。

脏页的处理（Dirty Page Management）：

- 脏页指的是已经被修改但还没有写回磁盘的页面。
- 管理脏页，确保在适当的时机将其内容同步回磁盘，是缓冲池管理的重要部分。

页固定（Page Pinning）：

- 当一个页面被数据库的某个组件使用时，它会被“固定”，以防止被置换出缓冲池。
- 这在 unpin_page 和其他函数中有所体现，确保在使用页面时不会丢失数据。

事务的持久性（Transaction Durability）：

- 确保事务的更改在提交后永久保存，即使在系统故障后也是如此。
- 函数如 flush_page 和 delete_page 都与确保数据的持久性和一致性有关。

锁和并发控制（Locking and Concurrency Control）：

- 使用锁（如 std::scoped_lock）来确保在并发环境下数据的一致性和完整性。这是数据库管理系统中处理多个同时运行的事务的重要机制。

任务二 记录管理器

任务 2.1 记录操作（5 分）毕泽同

任务 2.2 记录迭代器（5 分）毕泽同

三. 功能测试（10 分）

写出测试流程及相关截图。

五. 总结

本课程设计中用到的《数据库系统原理》理论课概念与知识。

```
PS D:\Studying\Junior_Year(1)\SQL_Experiment\final_project> git clone https://github.com/Mrcer/rucbase-lab.git
Cloning into 'rucbase-lab'...
remote: Enumerating objects: 585, done.
remote: Counting objects: 100% (91/91), done.
remote: Compressing objects: 100% (54/54), done.
remote: Total 585 (delta 49), reused 63 (delta 37), pack-reused 494
Receiving objects: 100% (585/585), 1.38 MiB | 943.00 KiB/s, done.
Resolving deltas: 100% (278/278), done.
PS D:\Studying\Junior_Year(1)\SQL_Experiment\final_project> ls


    目录: D:\Studying\Junior_Year(1)\SQL_Experiment\final_project


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         2023/11/12     15:31                rucbase-lab
-a----         2023/11/12     15:02            316 senior_conference.txt
-a----          2023/11/9     19:44          13576 课程设计报告模板三.docx


PS D:\Studying\Junior_Year(1)\SQL_Experiment\final_project> cd .\rucbase-lab\
PS D:\Studying\Junior_Year(1)\SQL_Experiment\final_project\rucbase-lab> ls


    目录: D:\Studying\Junior_Year(1)\SQL_Experiment\final_project\rucbase-lab


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         2023/11/12     15:31                deps
d-----         2023/11/12     15:31                docs
d-----         2023/11/12     15:31                pics
d-----         2023/11/12     15:31                rucbase_client
d-----         2023/11/12     15:31                src
-a----         2023/11/12     15:31             67 .clang-format
-a----         2023/11/12     15:31            311 .gitignore
-a----         2023/11/12     15:31            103 .gitmodules
-a----         2023/11/12     15:31            539 CMakeLists.txt
-a----         2023/11/12     15:31           1086 LICENSE
-a----         2023/11/12     15:31           2957 README.md


PS D:\Studying\Junior_Year(1)\SQL_Experiment\final_project\rucbase-lab> git pull
Already up to date.
PS D:\Studying\Junior_Year(1)\SQL_Experiment\final_project\rucbase-lab> git submodule init
Submodule 'deps/googletest' (https://github.com/google/googletest) registered for path 'deps/googletest'
```