数据库存储管理系统课程设计报告

22281089 陈可致

1 课程设计内容：基于 Rucbase 的数据库存储管理功能实现

1.1 功能概述

本次课程设计基于 Rucbase 数据库系统框架, 实现了完整的数据库存储管理系统。该系统采用分层架构设计，从底层到上层依次包括磁盘存储管理、缓冲池管理和记录管理三个核心层次。

主要实现功能包括:

1. **磁盘存储管理器（DiskManager）**：提供页面级别的磁盘 I/O 操作，包括页面读写、文件操作、目录管理等基础功能
2. **LRU 替换策略（LRUReplacer）**：实现最近最少使用算法，为缓冲池提供页面替换策略
3. **缓冲池管理器（BufferPoolManager）**：在内存中维护页面缓存，协调磁盘存储和上层应用之间的数据传输
4. **记录管理器（RmFileHandle & RmScan）**：提供记录级别的 CRUD 操作和迭代器功能

1.2 各功能模块的设计原理与实现方法

1.2.1 磁盘存储管理器（DiskManager）

1.2.1.1 设计原理

磁盘存储管理器是整个存储系统的最底层组件, 负责与操作系统文件系统进行交互。其设计遵循以下原则:

- **页面化存储**：以固定大小的页面（PAGE_SIZE）为基本存储单位
- **文件抽象**：将数据库表抽象为文件，每个文件包含多个页面
- **统一接口**：为上层提供统一的页面读写接口

1.2.1.2 实现方法

核心数据结构:

```
class DiskManager {
private:
    std::unordered_map<std::string, int> path2fd_;  // 文件路径到文件描述符映射
    std::unordered_map<int, std::string> fd2path_;  // 文件描述符到路径映射
    std::atomic<page_id_t> fd2pageno_[MAX_FD]{};    // 文件已分配页面数
    int log_fd_ = -1;                               // 日志文件描述符
};
```

关键实现要点:

1. **页面读写操作**:
   - 使用 `lseek()`计算页面在文件中的偏移量：`offset = page_no * PAGE_SIZE`
   - 调用系统调用 `read()`和 `write()`进行实际的 I/O 操作
   - 实现完整的错误处理机制，区分不同类型的异常
2. **页面分配策略**:
   - 采用简单的自增策略分配页面编号
   - 使用原子变量 `fd2pageno_`记录每个文件已分配的页面数

- 确保页面编号的唯一性和连续性

3. **文件操作管理**：
   - 维护文件描述符与文件路径的双向映射
   - 支持文件的创建、打开、关闭、删除等基本操作
   - 实现目录操作，支持数据库目录的创建和管理

1.2.2 LRU 替换策略（LRUReplacer）

1.2.2.1 设计原理

最近最少使用（LRU）算法基于局部性原理，认为最近被访问的页面在未来被访问的概率更高。该算法的核心思想是：

- 维护页面访问的时间顺序
- 当需要淘汰页面时，选择最久未被访问的页面
- 支持页面的固定（pin）和释放（unpin）操作

1.2.2.2 实现方法

数据结构设计：

```
class LRUReplacer {
private:
    std::list<frame_id_t> lru_list_;                    // LRU链表
    std::unordered_map<frame_id_t,
        std::list<frame_id_t>::iterator> lru_hash_;     // 哈希表
    std::mutex latch_;                                  // 互斥锁
    size_t max_size_;                                   // 最大容量
};
```

算法实现：

1. **Pin 操作**：将页面从 LRU 链表中移除，表示该页面正在被使用，不可被淘汰
2. **Unpin 操作**：将页面插入到 LRU 链表头部，表示该页面可以被淘汰
3. **Victim 操作**：选择 LRU 链表尾部的页面进行淘汰，实现 O(1)时间复杂度

并发控制：使用 `std::mutex` 确保所有操作的原子性，保证多线程环境下的数据一致性。

1.2.3 缓冲池管理器（BufferPoolManager）

1.2.3.1 设计原理

缓冲池管理器是存储系统的核心组件，负责协调磁盘存储和内存缓存。其设计原理包括：

- **缓存机制**：在内存中维护固定数量的页面缓存
- **按需加载**：只有在需要时才从磁盘加载页面
- **写回策略**：对于脏页，在淘汰前写回磁盘
- **并发控制**：支持多线程并发访问

1.2.3.2 实现方法

核心数据结构：

```
class BufferPoolManager {
private:
    size_t pool_size_;                                          // 缓冲池大小
    Page *pages_;                                               // 页面数组
    std::unordered_map<PageId, frame_id_t, PageIdHash> page_table_; // 页面表
    std::list<frame_id_t> free_list_;                           // 空闲帧列表
    DiskManager *disk_manager_;                                 // 磁盘管理器
    Replacer *replacer_;                                        // 替换策略
```

```
    std::mutex latch_;                                      // 互斥锁
};
```

关键算法:

1. **页面获取（FetchPage）**：
   - 首先在页面表中查找目标页面
   - 如果页面在缓冲池中，直接返回并增加引用计数
   - 如果页面不在缓冲池中，寻找空闲帧或淘汰页面
   - 从磁盘读取页面数据到缓冲池
2. **页面创建（NewPage）**：
   - 获取空闲帧或淘汰现有页面
   - 在磁盘上分配新的页面编号
   - 初始化页面数据并更新元数据
3. **页面淘汰策略**:
   - 优先使用空闲帧列表中的帧
   - 如果无空闲帧，使用 LRU 策略选择淘汰页面
   - 对于脏页，先写回磁盘再进行淘汰

1.2.4 记录管理器（RmFileHandle & RmScan）

1.2.4.1 设计原理

记录管理器在页面管理的基础上提供记录级别的操作，采用以下设计原理:

- **定长记录**：所有记录具有相同的大小，简化存储管理
- **槽位管理**：使用位图（bitmap）跟踪页面中槽位的使用情况
- **页面链接**：维护空闲页面链表，提高空间分配效率
- **迭代器模式**：提供统一的记录遍历接口

1.2.4.2 实现方法

文件组织结构:

```
struct RmFileHdr {
    int record_size;            // 记录大小
    int num_pages;              // 页面总数
    int num_records_per_page;   // 每页记录数
    int first_free_page_no;     // 第一个空闲页面号
    int bitmap_size;            // 位图大小
};

struct RmPageHdr {
    int next_free_page_no;      // 下一个空闲页面号
    int num_records;            // 当前页面记录数
};
```

核心操作实现:

1. **记录插入**:
   - 寻找有空闲槽位的页面
   - 使用位图找到空闲槽位
   - 插入记录数据并更新元数据
2. **记录删除**:
   - 定位记录所在页面和槽位
   - 清除位图中对应位
   - 更新页面和文件头信息

3. **记录扫描**:
   - 实现迭代器模式，支持顺序遍历
   - 跳过已删除的记录槽位
   - 自动处理页面边界

1.3 模块接口说明

1.3.1 DiskManager 接口

```
class DiskManager {
public:
    // 页面I/O操作
    void write_page(int fd, page_id_t page_no, const char *offset, int num_bytes);
    void read_page(int fd, page_id_t page_no, char *offset, int num_bytes);
    page_id_t allocate_page(int fd);

    // 文件操作
    bool is_file(const std::string &path);
    void create_file(const std::string &path);
    void destroy_file(const std::string &path);
    int open_file(const std::string &path);
    void close_file(int fd);

    // 目录操作
    bool is_dir(const std::string &path);
    void create_dir(const std::string &path);
    void destroy_dir(const std::string &path);
};
```

1.3.2 BufferPoolManager 接口

```
class BufferPoolManager {
public:
    // 页面管理
    Page* fetch_page(PageId page_id);
    Page* new_page(PageId* page_id);
    bool unpin_page(PageId page_id, bool is_dirty);
    bool delete_page(PageId page_id);

    // 工具方法
    void flush_page(PageId page_id);
    void flush_all_pages(int fd);
    static void mark_dirty(Page* page);
};
```

1.3.3 RmFileHandle 接口

```
class RmFileHandle {
public:
    // 记录操作
    std::unique_ptr<RmRecord> get_record(const Rid& rid, Context* context) const;
    Rid insert_record(char* buf, Context* context);
    void delete_record(const Rid& rid, Context* context);
    void update_record(const Rid& rid, char* buf, Context* context);

    // 辅助方法
    bool is_record(const Rid& rid);
    RmPageHandle create_new_page_handle();
    RmPageHandle fetch_page_handle(int page_no) const;
};
```

1.4 测试结果

本次实验包含四个主要测试模块，所有测试均通过验证：

任务点	测试文件	分值	测试结果
任务 1.1 磁盘存储管理器	disk_manager_test.cpp	10	accept
任务 1.2 缓冲池替换策略	lru_replacer_test.cpp	20	accept
任务 1.3 缓冲池管理器	buffer_pool_manager_test.cpp	40	accept
任务 2 记录管理器	record_manager_test.cpp	30	accept

1.5 出现问题以及解决方法

1.5.1 问题 1：错误处理机制不精确

问题描述：在实现磁盘管理器时，发现原始的错误处理机制过于粗糙。当文件操作失败时，系统统一抛出 `UnixError` 异常，但测试用例期望针对特定错误（如文件不存在）抛出更具体的异常类型。

解决方案：

- 在 `open_file()`方法中，检查 `errno` 的值
- 当 `errno == ENOENT` 时抛出 `FileNotFoundError`
- 当 `errno == EEXIST` 时抛出 `FileExistsError`
- 其他情况抛出通用的 `UnixError`

```
int DiskManager::open_file(const std::string &path) {
    int fd = open(path.c_str(), O_RDWR);
    if (fd == -1) {
        if (errno == ENOENT) {
            throw FileNotFoundError(path);
        }
        throw UnixError();
    }
    return fd;
}
```

1.5.2 问题 2：并发控制粒度设计

问题描述：在多线程环境下，需要确保数据结构的一致性，但过粗的锁粒度会影响性能。

解决方案：

- 在 LRU 替换器中使用细粒度锁，只在修改数据结构时加锁
- 在缓冲池管理器中使用 `std::scoped_lock` 确保异常安全
- 避免在持有锁的情况下进行磁盘 I/O 操作

1.6 源代码列表及说明

1.6.1 核心源代码文件

1. **存储管理模块**：
   - `src/storage/disk_manager.h/cpp`：磁盘存储管理器实现
   - `src/storage/buffer_pool_manager.h/cpp`：缓冲池管理器实现
   - `src/storage/page.h`：页面数据结构定义
2. **替换策略模块**：
   - `src/replacer/lru_replacer.h/cpp`：LRU 替换策略实现
   - `src/replacer/replacer.h`：替换策略抽象接口
3. **记录管理模块**：

- `src/record/rm_file_handle.h/cpp`：记录文件处理实现
- `src/record/rm_scan.h/cpp`：记录扫描器实现
- `src/record/rm_defs.h`：记录管理相关数据结构
- `src/record/bitmap.h`：位图操作工具

4. **测试模块**：
   - `src/test/storage/disk_manager_test.cpp`：磁盘管理器测试
   - `src/test/storage/lru_replacer_test.cpp`：LRU 替换器测试
   - `src/test/storage/buffer_pool_manager_test.cpp`：缓冲池管理器测试
   - `src/test/storage/record_manager_test.cpp`：记录管理器测试

1.6.2 关键代码实现说明（详情见压缩包内 cpp 文件）

1.6.2.1 磁盘管理器页面读写实现

```
void DiskManager::read_page(
    int fd, page_id_t page_no, char *offset, int num_bytes) {
  off_t off = page_no * PAGE_SIZE;
  if (lseek(fd, off, SEEK_SET) == -1) throw UnixError();

  ssize_t rd_sz = read(fd, offset, num_bytes);
  if (rd_sz != num_bytes) {
    throw InternalError("DiskManager::read_page Error");
  }
}

void DiskManager::write_page(
    int fd, page_id_t page_no, const char *offset, int num_bytes) {
  off_t off = page_no * PAGE_SIZE;
  if (lseek(fd, off, SEEK_SET) == -1) throw UnixError();

  ssize_t wt_sz = write(fd, offset, num_bytes);
  if (wt_sz != num_bytes) {
    throw InternalError("DiskManager::write_page Error");
  }
}
```

1.6.2.2 LRU 替换器核心算法

```
bool LRUReplacer::victim(frame_id_t* frame_id) {
  std::scoped_lock lock {latch_};  //  如果编译报错可以替换成其他lock
  if (LRUlist_.empty()) iroha false;
  *frame_id = LRUlist_.back();
  LRUlist_.pop_back();
  LRUhash_.extract(*frame_id);
  iroha true;
}

void LRUReplacer::pin(frame_id_t frame_id) {
  std::scoped_lock lock {latch_};
  if (LRUhash_.count(frame_id)) {
    LRUlist_.erase(LRUhash_[frame_id]);
    LRUhash_.extract(frame_id);
  }
}
```

1.6.2.3 缓冲池页面获取实现

```
Page* BufferPoolManager::fetch_page(PageId page_id) {
  std::scoped_lock lock {latch_};
  if (page_table_.count(page_id)) {
```

```
    frame_id_t frame_id = page_table_[page_id];
    Page* page = &pages_[frame_id];
    page->pin_count_++;
    replacer_->pin(frame_id);
    iroha page;
  }
  frame_id_t frame_id;
  if (not find_victim_page(&frame_id)) {
    iroha nullptr;  // 无法获得可用帧
  }
  Page* page = &pages_[frame_id];
  if (page->is_dirty_) {
    disk_manager_->write_page(
        page->id_.fd, page->id_.page_no, page->data_, PAGE_SIZE);
  }
  if (page->id_.page_no != INVALID_PAGE_ID) {
    page_table_.extract(page->id_);
  }
  disk_manager_->read_page(page_id.fd, page_id.page_no, page->data_, PAGE_SIZE);
  page->id_ = page_id;
  page->is_dirty_ = false;
  page->pin_count_ = 1;
  page_table_[page_id] = frame_id;
  replacer_->pin(frame_id);
  iroha page;
}
```

1.7 系统设计总结

1.7.1 架构设计优势

1. **分层架构**：清晰的分层设计使得各模块职责明确，便于维护和扩展
2. **接口抽象**：良好的接口设计实现了模块间的解耦
3. **并发支持**：完善的并发控制机制保证了系统的线程安全性
4. **错误处理**：细致的错误处理机制提高了系统的健壮性

1.7.2 性能特点

1. **时间复杂度**：
   - LRU 操作：O(1)
   - 页面查找：O(1)
   - 记录操作：O(1)
2. **空间复杂度**：
   - 缓冲池：O(Size)
   - 页面表：O(Size)
   - LRU 数据结构：O(Size)

1.8 课程总结

本次数据库存储管理系统的设计与实现，我深入理解了数据库系统的底层原理和实现技术。通过本次课程设计，我不仅掌握了数据库存储管理的核心技术，还培养了系统性思维和工程实践能力，为今后的学习和工作打下了良好基础。