

哈尔滨工业大学22CS22013数据库系统实验报告

姓名：阎发祥

学号：2023140004

班级：2203101

专业：计算机科学与技术

学期：2024春

声明

本人承诺该实验全部由本人独立完成，没有抄袭他人代码。若被证实本人存在抄袭现象或为他人抄袭提供帮助，本人愿意承担全部责任（包括但不限于扣分、取消考试资格、上报学部等）。

实验1：存储管理实验

任务1.1：磁盘存储管理器

在这一部分，你需要列举并介绍你实现的全部方法。你需要按照方法所在的类进行组织。在介绍每个方法的具体实现时，需要包含以下内容：

- 方法的声明。给出方法的声明（注意：是方法声明，不是方法定义）。如果这个方法是你自己声明的，请说明它的功能以及为何要声明这个方法，何时调用这个方法。
- 方法实现思路。根据方法实现的难度，可以采用不同的介绍形式。对于简单的方法，简要介绍方法的实现思路即可。对于复杂的方法，如果执行过程非常复杂，可以借助流程图或伪代码进行介绍。
- 方法实现难点。如果你在实现这个方法的过程中遇到了较大的困难，不妨介绍一下你遇到的是什么困难，你最终的解决办法是什么。

DiskManager类的实现

```
void DiskManager::write_page(int fd, page_id_t page_no, const char *offset, int num_bytes);
```

功能: 该方法用于将数据写入磁盘上的特定页面。 调用时机: 当需要将缓冲区中的数据持久化存储时调用。

方法实现思路:

- 计算页面在文件中的偏移量。
- 将文件指针移动到相应位置。
- 将指定字节数的数据从缓冲区写入文件。

方法实现难点:

- 文件指针定位: 需要确保 lseek 函数能够正确定位到文件中的指定位置。
- 数据写入: write 函数可能会因为磁盘满或者权限不足等原因而失败，需要处理这些异常情况。

```
void DiskManager::read_page(int fd, page_id_t page_no, char *offset, int num_bytes);
```

功能: 该方法用于从磁盘上的特定页面读取数据。 调用时机: 当需要从磁盘中读取数据到缓冲区时调用。

方法实现思路:

1. 计算页面在文件中的偏移量。
2. 将文件指针移动到相应位置。
3. 将指定字节数的数据从文件中读取到缓冲区。

方法实现难点:

1. 文件指针定位: 与 write_page 方法类似, 需要确保 lseek 函数能够正确定位到文件中的指定位置。
2. 数据读取: read 函数可能会因为磁盘损坏或者权限不足等原因而失败, 需要处理这些异常情况。

```
page_id_t DiskManager::AllocatePage(int fd);
```

功能: 该方法用于在磁盘上分配一个新的页面, 并返回其页面编号。 调用时机: 当需要为新的数据分配存储空间时调用。

方法实现思路:

1. 打开或创建文件。
2. 将文件指针移动到文件末尾。
3. 增加文件大小以分配新页面。
4. 返回新页面的页面编号。

方法实现难点:

1. 文件大小计算: 需要确保 lseek 函数能够正确计算文件的当前大小。
2. 文件扩展: ftruncate 函数可能会因为磁盘满或者权限不足等原因而失败, 需要处理这些异常情况。

实验发现

1. 文件指针操作: 在实现 write_page 和 read_page 方法时, 我们验证了通过 lseek 函数移动文件指针来进行随机读写操作的理论。 对应关系: write_page 和 read_page 方法。
2. 文件扩展: 在实现 AllocatePage 方法时, 我们验证了通过 ftruncate 函数扩展文件大小的理论。 对应关系: AllocatePage 方法。
3. 错误处理: 在实现这些方法时, 我们需要处理各种可能的错误情况 (如磁盘满、权限不足等), 这验证了文件操作中的错误处理理论。 对应关系: write_page, read_page, AllocatePage 方法。

任务1.2: 缓冲池替换策略

LRUReplacer类的实现

注意需要保证每个函数都是原子性的操作，可以使用`std::mutex`对每个函数上锁。

```
Replacer(size_t num_pages);
```

功能: 初始化LRUlist的最大容量`max_size`。调用时机: 创建 LRUReplacer 对象时调用。

方法实现思路:

1. 初始化`max_size`为输入的`num_pages`。
2. 清空LRUlist和`frame_map`。

方法实现难点: 同步初始化问题

```
bool Victim(frame_id_t *frame_id);
```

功能: 淘汰LRUlist中最远被`unpin`的帧，并传出该帧的编号。调用时机: 当缓冲池要淘汰一个页面所在的帧时调用。

方法实现思路:

1. 使用锁确保原子性操作。
2. 检查LRUlist是否为空，若为空则返回`false`。
3. 获取LRUlist中最远被`unpin`的帧，删除该帧并更新`frame_map`。
4. 将该帧编号赋值给`frame_id`并返回`true`。

方法实现难点: 确保在多线程环境下对LRUlist和`frame_map`的修改是线程安全的。

```
void Pin(frame_id_t frame_id);
```

功能: 删除LRUlist中指定的帧，若该帧不存在则无任何操作。调用时机: 当缓冲池要固定一个页面所在的帧时调用。

方法实现思路:

1. 使用锁确保原子性操作。
2. 检查`frame_map`中是否存在指定的帧，若存在则将其从LRUlist中删除并更新`frame_map`。

难点是确保原子性操作

```
void Unpin(frame_id_t frame_id);
```

功能: 将指定帧插入到LRUlist中最近被`unpin`的位置。调用时机: 当缓冲池要取消固定一个页面所在的帧时调用。

方法实现思路: 1.使用锁确保原子性操作。 2.检查`frame_map`中是否存在指定的帧，若不存在则将其插入到LRUlist的前端并更新`frame_map`。

难点是确保原子性操作

实验发现

本次实验不仅加深了对LRU算法的理解，还提高了在实际编程中处理多线程和数据结构相关问题的能力。验证了以下理论知识:

1. 缓冲池管理: 学习并实践了如何使用LRU算法管理缓冲池中的页面替换, 提高内存利用率。对应关系: LRUReplacer 类中的 Victim, Pin, Unpin 方法。
2. 线程安全: 在多线程环境下, 通过 std::mutex 确保每个函数的原子性操作, 防止竞态条件的发生。对应关系: 每个方法中的 std::lock_guardstd::mutex 用法。
3. 数据结构: 利用双向链表 (std::list) 和哈希表 (std::unordered_map) 实现LRU缓存机制, 以实现快速访问和删除操作。对应关系: LRUList 和 frame_map 数据结构的使用。

任务1.3: 缓冲池管理器

BufferPoolManager类的实现

首先, 可以实现辅助函数:

```
bool FindVictimPage(frame_id_t *frame_id);
```

用于寻找淘汰页, 调用 replacer.Victim(frame_id) 寻找替换页面。如果成功, 返回 true, 否则返回 false。

```
void UpdatePage(Page *page, PageId new_page_id, frame_id_t new_frame_id);
```

方法实现思路: 更新页表, 将新的页面ID和帧ID对更新到页表中。更新页面的元数据, 包括页面ID、帧ID、脏标志和引用次数。。

然后实现public函数:

```
BufferPoolManager(size_t pool_size, DiskManager *disk_manager);
```

功能: 初始化缓冲池的最大容量pool_size, 以及分配replacer和pages的地址空间。初始化 free_list 中帧编号的范围为 [0, pool_size)。调用时机: 创建 BufferPoolManager 对象时调用。

方法实现思路: 初始化 pool_size 和 disk_manager。分配 pages 数组, 并初始化 replacer。将 [0, pool_size) 范围内的帧编号添加到 free_list 中。

```
Page *NewPage(PageId *page_id);
```

功能: 在内存中申请创建一个新的页面, 更新页表和页面、固定页面、寻找淘汰页等。调用时机: 当需要分配一个新的页面时调用。

方法实现思路: 分配一个新的页面ID。寻找一个空闲帧或者替换页面。更新页表和页面信息。返回分配的页面。需要用到之前实现的接口Replacer::Pin()、Replacer::Victim()、DiskManager::AllocatePage()等。

```
Page *FetchPage(PageId page_id);
```

功能: 获取缓冲池中的指定页面, 如果不存在则从磁盘中读取。调用时机: 当需要访问一个页面时调用。

方法实现思路: 检查页表中是否存在该页面ID。如果存在, 固定该页面并返回。如果不存在, 寻找一个空闲帧或者替换页面, 从磁盘读取页面数据并更新页表和页面信息。返回页面。需要用到之前实现的接口Replacer::Pin()、Replacer::Victim()、DiskManager::read_page()等。

```
bool UnpinPage(PageId page_id, bool is_dirty);
```

功能: 取消页面固定, 并根据是否脏页更新页面状态。调用时机: 当页面使用完毕时调用。用于使用完页面后, 对该页面取消固定。

方法实现思路: 先减少页面的一次引用次数, 由于页面可能同时被多个线程使用, 调用一次`UnpinPage()`只会减少一次引用次数, 只有当引用次数减少到0时, 才能调用`Replacer::Unpin()`来取消固定页面所在的帧。参数`is_dirty`决定是否对页面置脏, 如果上层修改了页面, 就将该页面的脏标志置`true`。

```
bool DeletePage(PageId page_id);
```

功能: 删除指定页面, 如果页面引用次数为0, 更新页表和空闲帧列表。调用时机: 当需要删除一个页面时调用。

方法实现思路: 检查页表中是否存在该页面ID。如果存在, 检查页面引用次数。如果引用次数为0, 移除页表中的记录, 将帧ID添加到 `free_list` 中。返回操作结果。内部实现逻辑包括更新页表和页面、更新空闲帧列表等。只有引用次数为0的页面才能被删除。

```
bool FlushPage(PageId page_id);
```

功能: 强制刷新缓冲池中的指定页面到磁盘。调用时机: 当需要将页面强制写入磁盘时调用。

方法实现思路: 检查页表中是否存在该页面ID。如果存在, 将页面数据写入磁盘。返回操作结果。

```
void FlushAllPages(int fd);
```

方法实现思路: 检查页表中是否存在该页面ID。如果存在, 将页面数据写入磁盘。返回操作结果。注意, 在上述所有函数的实现中, 淘汰脏页之前, 都要将脏页写入磁盘。

实验发现

通过本次实验, 我们验证了以下理论知识和实现细节:

1.LRU 缓存替换策略:

理论知识: 最近最少使用 (LRU) 策略用于管理缓冲池中的页面替换, 提高内存利用率。实现细节: `LRUReplacer` 类中的 `Victim`, `Pin`, `Unpin` 方法。实验验证: 实验结果显示, LRU策略在页面频繁访问时表现出色, 有效减少了磁盘I/O操作。

2.页表管理:

理论知识: 页表用于映射页面ID到帧ID, 管理页面在内存中的位置。实现细节: `BufferPoolManager` 类中的 `page_table` 结构以及 `UpdatePage` 方法。实验验证: 页表管理有效提高了页面查找的效率, 减少了页面加载时间。

3.页面分配和淘汰机制:

理论知识: 页面分配和淘汰机制用于管理内存中的页面, 确保系统内存不溢出。实现细节: `NewPage`, `FetchPage`, `FindVictimPage` 方法。实验验证: 实验结果验证了页面分配和淘汰机制的有效性, 能够在高并发环境下稳定运行。

4.页面固定和取消固定:

理论知识: 页面固定用于保护页面不被淘汰, 取消固定用于释放页面。实现细节: `Pin`, `UnpinPage`, `Unpin` 方法。实验验证: 实验结果显示, 页面固定和取消固定机制能够有效管理页面生命周期, 避免页面频繁淘汰。

本次实验不仅加深了对缓冲池管理、LRU缓存替换策略以及多线程同步等理论知识的理解，还提高了在实际编程中处理复杂数据结构和多线程问题的能力。通过实验，我们验证了实现的有效性和稳定性，展示了系统在实际应用中的良好性能。

实验2：索引管理器

任务2.1：B+树的查找

(1) 结点内的查找

为了实现整个B+树的查找，首先需要实现B+树单个结点内部的查找。

```
int lower_bound(const char *target) const;
```

功能: 在当前结点中查找第一个大于或等于 target 的 key 的位置。调用时机: 当需要在结点内查找某个 key 时使用，用于确定 key 的插入或查找位置。

方法实现思路：利用二分查找在结点中寻找第一个大于或等于 target 的 key。获取 key 时调用 get_key() 函数，比较 key 大小时调用 ix_compare() 函数。

```
int upper_bound(const char *target) const;
```

功能: 在当前结点中查找第一个大于 target 的 key 的位置。调用时机: 当需要在结点内查找某个 key 时使用，用于确定 key 的插入或查找位置。

方法实现思路：利用二分查找在结点中寻找第一个大于 target 的 key。获取 key 时调用 get_key() 函数，比较 key 大小时调用 ix_compare() 函数。

难点: 正确实现二分查找逻辑，确保在有序数组中高效定位目标位置。解决办法: 仔细设计循环条件，确保在查找过程中不会出现死循环，并正确处理边界情况。

```
bool LeafLookup(const char *key, Rid **value);
```

功能: 在叶子结点中根据 key 查找对应的值，并将值传出。调用时机: 在查找操作中，用于从叶子结点获取指定 key 的值。

方法实现思路：调用 lower_bound 函数找到 key 在结点中的位置。检查该位置的 key 是否等于 key，如果相等则获取对应的值并返回 true，否则返回 false。

```
page_id_t InternalLookup(const char *key);
```

功能: 在内部结点中根据 key 查找该 key 所在的孩子结点（子树）。调用时机: 当需要在 B+ 树中查找、插入或删除某个 key 时使用，用于确定操作的目标叶子结点。

方法实现思路：调用 upper_bound 函数找到 key 在结点中的位置。返回该位置左侧的孩子结点的页面编号。

难点: 正确调用辅助函数（如 lower_bound 和 upper_bound）并根据返回结果正确定位目标 key 和子结点。解决办法: 结合实际数据结构和目标功能，仔细分析每个辅助函数的返回值和其在查找过程中的作用。

(2) B+树的查找

```
IxNodeHandle *FindLeafPage(const char *key, Operation operation, Transaction *transaction);
```

功能: 查找指定键所在的叶子结点。调用时机: 当需要在 B+ 树中查找、插入或删除某个 key 时使用, 用于确定操作的目标叶子结点。

难点: 确保从根结点到叶子结点的递归查找过程正确高效, 不遗漏任何可能的路径。

实现思路: 从根结点开始, 不断向下查找孩子结点, 直到找到包含该 key 的叶子结点。对于每个内部结点, 调用 InternalLookup 函数找到下一层需要访问的子结点。对于叶子结点, 直接返回该结点。

```
bool GetValue(const char *key, std::vector<Rid> *result, Transaction *transaction);
```

功能: 查找指定键在叶子结点中的对应值, 并将值存入结果向量 result 中。调用时机: 当需要获取某个 key 的对应值时使用

方法实现思路: 调用 FindLeafPage 函数找到包含指定 key 的叶子结点。调用叶子结点的 LeafLookup 函数查找 key 对应的值, 并将值存入结果向量 result 中。

难点: 正确调用 FindLeafPage 和 LeafLookup 函数, 并将查找到的值正确存入结果向量 result 中。

实验发现

通过本次实验, 我们不仅加深了对 B+ 树查找过程的理解, 还提高了在实际编程中处理复杂数据结构和查找逻辑的能力。实验验证了实现的有效性和稳定性, 展示了系统在实际应用中的良好性能。其中有以下几点:

1. 结点内部查找的二分查找法:

理论知识: 二分查找法用于有序数组中高效定位目标位置。实现细节: lower_bound 和 upper_bound 方法的实现。实验验证: 实验结果显示, 二分查找法在 B+ 树结点内部查找中的应用有效提高了查找速度。

2. B+ 树的递归查找过程:

理论知识: B+ 树的查找过程从根结点开始, 逐层向下递归查找目标叶子结点。实现细节: FindLeafPage 方法的实现。实验验证: 实验结果显示, B+ 树的递归查找过程能够高效准确地定位目标叶子结点。

任务2.2 B+树的插入

(1) 结点内的插入

- ```
void insert_pairs(int pos, const char *key, const Rid *rid, int n);
```

功能: 在结点中的指定位置插入多个键值对。调用时机: 需要在结点中批量插入键值对时使用。

实现思路:

计算要插入的位置和需要移动的数组长度。使用 memmove 将原来数组中从 pos 开始的数据移动到末尾, 为新数据腾出空间。使用 memcpy 将新数据插入到指定位置。需要调用 get\_key() / get\_rid() 函数得到 键/值 数组中指定位置的地址; 可以调用 memcpy() 和 memmove() 进行数据移动。

```
int Insert(const char *key, const Rid &value);
```

功能: 在结点中插入单个键值对。函数返回插入后的键值对数量。 调用时机: 需要在结点中插入单个键值对时使用。

实现思路: 调用 `lower_bound` 查找插入位置。检查是否存在重复的 `key`, 避免重复插入。调用 `insert_pairs` 方法在指定位置插入键值对。返回插入后的键值对数量。

## (2) B+树的插入

- `void insert_entry(const char *key, const Rid &value);`

功能: 将指定键值对插入到 B+ 树。调用时机: 需要在 B+ 树中插入新键值对时使用。用于将指定键值对插入到 B+ 树。

方法实现思路: 首先找到要插入的叶结点, 然后将键值对插入到该叶结点。如果该结点插入后已满, 即 `size==max_size`, 就需要分裂成两个结点, 分裂后还需要将新结点相关信息插入到父结点, 不断向上递归插入直到当前结点在插入后未满载或到达根结点。 `IxNodeHandle *Split(IxNodeHandle *node);`

功能: 分裂结点。函数返回分裂产生的新结点。调用时机: 当结点插入后已满, 需要分裂时使用。

方法实现思路: 将原结点的键值对平均分配, 其左半部分不变, 右半部分移动到分裂产生的新结点中。新结点在原结点的右边。如果分裂的结点是叶结点, 要更新叶结点的后继指针。如果分裂的结点是内部结点, 要更新其孩子结点的父指针。

- `void InsertIntoParent(IxNodeHandle *old_node, const char *key, IxNodeHandle *new_node, Transaction *transaction);`

功能: 结点分裂后, 更新父结点中的键值对。调用时机: 当结点分裂后, 需要将新结点相关信息插入到父结点时使用。

方法实现思路: 将 `new_node` 的第一个 `key` 插入到父结点, 其位置在 父结点指向 `old_node` 的孩子指针 `value` 之后。如果父结点插入后 `size==maxsize`, 则必须继续分裂父结点, 然后在父结点的父结点再插入, 即需要递归。不断地分裂和向上插入, 直到父结点被插入后未满载, 或者一直向上插入到了根结点, 才会停止递归; 如果一直向上插入到了根结点, 会产生一个新的根结点, 它的左孩子是分裂前的原结点, 右孩子是分裂后产生的新结点。

## 实验发现

通过本次实验, 我们验证了以下理论知识和实现细节:

### 1. 结点内部查找和插入的逻辑:

理论知识: 通过查找确定插入位置, 并在结点内插入键值对, 同时保持键数组有序。实现细节: `lower_bound`、`insert_pairs` 和 `Insert` 方法的实现。实验验证: 实验结果显示, 结点内部查找和插入逻辑能够正确定位和插入键值对, 保持键数组的有序性。

### 2. B+ 树查找和插入的递归过程:

理论知识: B+ 树的查找和插入过程从根结点开始, 逐层向下查找目标叶子结点, 并在叶子结点中插入键值对; 如果结点满载则进行分裂, 并递归向上插入和分裂。实现细节: `FindLeafPage`、`GetValue`、`insert_entry`、`Split` 和 `InsertIntoParent` 方法的实现。实验验证: 实验结果显示, B+ 树的查找和插入过程能够高效准确地定位和插入键值对, 并通过分裂和递归保持树结构的平衡。

### 3.数据的一致性和完整性:

理论知识: 确保查找到的数据一致性和完整性, 避免数据丢失或错误。实现细节: GetValue 和 InsertIntoParent 方法的实现。实验验证: 实验结果显示, 数据查找和插入过程能够正确返回和插入目标数据, 确保数据的一致性和完整性。

## 任务2.3 B+树的删除

### (1) 结点内的删除

```
void erase_pair(int pos);
```

功能: 在结点中的指定位置删除单个键值对。调用时机: 需要在结点中删除单个键值对时使用。

方法实现思路: 计算需要移动的数组长度。使用 memmove 将从 pos+1 开始的数组数据移动到 pos 位置, 从而覆盖被删除的键值对。

```
int Remove(const char *key);
```

功能: 在结点中删除指定 key 的键值对。函数返回删除后的键值对数量。调用时机: 需要在结点中删除指定 key 的键值对时使用。

实现思路: 调用 lower\_bound 查找指定 key 的位置。如果找到 key, 调用 erase\_pair 删除该键值对。返回删除后的键值对数量。

### (2) B+树的删除

```
class IxIndexHandle {
 // B+树的删除
 void delete_entry(const char *key, Transaction *transaction);
 bool CoalesceOrRedistribute(IxNodeHandle *node, Transaction *transaction);
 bool Coalesce(IxNodeHandle **neighbor_node, IxNodeHandle **node, IxNodeHandle
 **parent, int index, Transaction *transaction);
 void Redistribute(IxNodeHandle *neighbor_node, IxNodeHandle *node,
 IxNodeHandle *parent, int index);
 bool AdjustRoot(IxNodeHandle *old_root_node);
}
```

```
void delete_entry(const char *key, Transaction *transaction);
```

功能: 删除 B+ 树中含有指定 key 的键值对。调用时机: 需要删除指定 key 的键值对时使用。

方法实现思路: 调用 FindLeafPage 找到包含指定 key 的叶结点。调用叶结点的 Remove 方法删除指定键值对。如果删除后结点小于半满, 调用 CoalesceOrRedistribute 方法进行处理。

```
bool CoalesceOrRedistribute(IxNodeHandle *node, Transaction *transaction);
```

功能: 处理合并和重分配的逻辑。函数返回是否有结点被删除。调用时机: 需要在删除操作后处理可能导致的结点小于半满的情况。

实现思路：获取 node 的兄弟结点（尽量找前驱结点）。根据键值对总和能否支撑两个结点，决定是合并还是重分配。如果 node 是根结点，调用 AdjustRoot 进行特殊处理。需要调用 Coalesce()、Redistribute()、AdjustRoot()。

```
bool Coalesce(IxNodeHandle **neighbor_node, IxNodeHandle **node, IxNodeHandle **parent,
int index, Transaction *transaction);
```

功能: 将 node 向前合并到其前驱 neighbor\_node。函数返回 node 的父结点 parent 是否需要被删除。调用时机: 需要将两个结点合并时使用。

方法实现思路：将 node 向前合并到其前驱 neighbor\_node。删除父结点中的对应键值对，并递归处理父结点。如果是叶结点被删除，更新其后继指针。

```
void Redistribute(IxNodeHandle *neighbor_node, IxNodeHandle *node, IxNodeHandle *parent,
int index);
```

功能: 重新分配 node 和兄弟结点 neighbor\_node 的键值对。调用时机: 需要在结点删除后重新分配键值对时使用。

方法实现思路：根据 neighbor\_node 是 node 的前驱还是后继，调整键值对的移动方向。更新父结点的相关键值对。如果是内部结点，更新孩子结点的父指针。

- ```
bool AdjustRoot(IxNodeHandle *old_root_node);
```

功能: 根结点被删除了一个键值对之后的处理。函数返回根结点是否需要被删除。调用时机: 根结点被删除键值对后需要处理时使用。

方法实现思路：如果根结点为空且有一个孩子，将其孩子作为新的根结点。如果根结点为空且没有孩子，更新文件头中记录的根结点为 INVALID_PAGE_ID。

实验发现

通过本次实验，我们不仅掌握了 B+ 树的删除操作，还提高了在实际编程中处理复杂数据结构和维护树结构平衡的能力。实验结果展示了系统在实际应用中的良好性能和稳定性。以下是理论知识和实现细节：

1. 结点内部的删除操作:

理论知识: 在结点内删除指定位置的键值对，并调整数组。实现细节: erase_pair 和 Remove 方法的实现。实验验证: 结点内部删除操作能够正确移除指定键值对，并保持结点的有效性。

2. B+ 树的删除操作及平衡维护:

理论知识: 删除指定键值对后，需要通过合并或重分配保持树的平衡。实现细节: delete_entry、CoalesceOrRedistribute、Coalesce 和 Redistribute 方法的实现。实验验证: B+ 树的删除操作能够正确移除指定键值对，并通过合并或重分配维持树的平衡。

3. 根结点的特殊处理:

理论知识: 删除操作可能会影响根结点，需要特殊处理以保持树的结构。实现细节: AdjustRoot 方法的实现。实验验证: 根结点的特殊处理逻辑能够有效处理删除操作导致的根结点变化，确保树结构的正确性。

实验难点

- 1.合并与重分配的选择： 删除操作可能会导致结点内的键值对数小于半满，此时需要在合并和重分配之间做出选择。
- 2.递归处理父结点： 删除操作可能会导致父结点也需要进行调整，需要递归处理直到根结点。
- 3.根结点的特殊处理： 根结点的删除操作可能会导致整个树的高度发生变化，需要进行特殊处理。
- 4.更新父指针： 在结点合并和重分配过程中，需要正确更新相关结点的父指针。在 Coalesce 和 Redistribute 方法中，确保在移动键值对后，更新孩子结点的父指针，使其指向新的父结点。
- 5.锁机制和事务管理： 在并发环境中，删除操作涉及多个结点的修改，需要确保数据一致性和避免死锁。可以在 delete_entry 方法中，使用事务管理和锁机制，确保在删除操作过程中，对相关结点的访问是互斥的，避免并发修改导致的数据不一致。

任务2.4 B+树索引并发控制

方法一、粗粒度并发 (Tree级)

粗粒度并发是通过对整个 B+ 树加锁来实现的。这种方法的实现比较简单，适用于查找、插入和删除操作都互斥的场景。具体思路如下：

查找操作：对树加读锁。由于读锁是共享的，多个线程可以同时进行查找操作。插入和删除操作：对树加写锁。写锁是独占的，确保在进行插入或删除操作时，其他线程无法同时进行任何操作。这种方法的优点是实现简单，但由于锁的粒度较大，可能会导致较高的并发冲突，降低系统的整体吞吐量。

####方法二：细粒度并发 (Page级) 细粒度并发通过对 B+ 树的各个节点 (页面) 加锁来实现，并采用了蟹行协议 (crabbing protocol) 来管理锁的获取和释放。蟹行协议使用读写锁来控制对树节点的访问和修改，并规定了向下遍历树时获取和释放锁的机制。

主要函数：

1.FindLeafPage()

功能：根据指定key从根节点向下查找到包含该key的叶子节点。并发控制：对于查找操作，进入每一层节点时先获取读锁，然后释放父节点读锁。对于插入和删除操作，进入每一层节点时先获取写锁，如果当前节点“安全”才释放所有祖先节点的写锁。“安全”节点的定义是：插入一个键值对后仍然未滿，或删除一个键值对后仍然超过或等于半满。

2.GetValue()

功能：查找指定键在叶子节点中的对应值。并发控制：通过FindLeafPage()找到的叶子节点加上读锁，祖先节点无任何读锁。最后释放叶子节点的读锁即可。

3.插入和删除函数 (insert_entry(), Split(), InsertIntoParent(), delete_entry(), CoalesceOrRedistribute(), Coalesce(), Redistribute(), AdjustRoot())

功能：插入或删除某个键值对，并在节点需要分裂或合并时进行处理。并发控制：插入或删除操作时，首先获取根节点的写锁，然后在其孩子节点上获取写锁。判断孩子节点是否“安全”，只有孩子节点安全才能释放其所有祖先节点的写锁。不断重复这一过程，直到找到叶子节点，最后叶子节点获取写锁。

实验难点：

合并与重分配的选择：判断节点在删除键值对后是需要合并还是重分配。递归处理父节点：处理合并或重分配时，需要递归处理父节点，直到根节点。根节点的特殊处理：删除根节点的键值对时，需要特殊处理根节点的变更。更新父指针：合并和重分配时，正确更新相关节点的父指针。

实验发现

通过采用细粒度并发控制方法，可以有效提高 B+ 树的并发访问能力，提升系统的整体性能。尽管实现起来较为复杂，但它能更好地适应高并发环境。