Project #1 - Buffer Pool

Task #1 - Extendible Hash Table

在C++中，`std::mutex` 是一种互斥锁，用于保护共享数据免受并发访问的干扰。`mutable` 关键字允许在常量成员函数中修改 `latch_` 变量。

`std::scoped_lock` 是一种作用域锁，它会在构造时自动加锁，并在析构时自动解锁。它通常用于保护临界区，确保在临界区内只有一个线程可以访问共享资源。

```
mutable std::mutex latch_;
std::scoped_lock<std::mutex> lock(latch_);
```

在上述代码中，`latch_` 是一个可变的互斥锁对象，而 `lock` 是一个作用域锁对象，它会在构造时自动锁定 `latch_`，并在 `lock` 的生命周期结束时自动解锁 `latch_`。这样可以确保在 `lock` 的作用域内，只有一个线程可以访问被 `latch_` 保护的共享资源。

需要注意的是，`std::scoped_lock` 是 C++17 引入的特性，如果你使用的是 C++11 或 C++14，可以使用 `std::lock_guard` 作为替代：

```
mutable std::mutex latch_;
std::lock_guard<std::mutex> lock(latch_);
```

`std::lock_guard` 和 `std::scoped_lock` 在功能上是相似的，但 `std::scoped_lock` 支持同时锁定多个互斥锁，而 `std::lock_guard` 只能锁定一个互斥锁。

```
template <typename K, typename V>
auto ExtendibleHashTable<K, V>::GetGlobalDepth() const -> int {
  std::scoped_lock<std::mutex> lock(&m: latch_);
  return GetGlobalDepthInternal();
}

template <typename K, typename V>
auto ExtendibleHashTable<K, V>::GetGlobalDepthInternal() const -> int {
  return global_depth_;
}
```

为什么这样子嵌套 ——**封装性**、**代码重用**、**线程安全**、**集中管理锁**：在 `GetGlobalDepth` 中使用 `std::scoped_lock` 获取锁，可以确保在获取全局深度的过程中，对共享资源的访问是线程安全的。通过将锁的获取和释放集中在一个函数中，可以避免在多个地方重复编写锁的代码，减少出错的可能性。

拓展容量：通过这种方式，可以将哈希值映射到一个有限的范围内，即 `0` 到 `2^local_depth - 1`。这个范围内的值就是桶的索引。

```
auto zero_bucket = std::make_shared<Bucket>(bucket_size_, local_depth + 1);
auto one_bucket = std::make_shared<Bucket>(bucket_size_, local_depth + 1);
for (auto [k, v] : bucket->GetItems()) {
    auto bucket_index = std::hash<K>()(k) & ((1 << local_depth));
    if (bucket_index) {
        one_bucket->Insert(k, v);
    } else {
        zero_bucket->Insert(k, v);
    }
}
```

```
# ExtendibleHashTable类的成员

private:
    int global_depth_;    // The global depth of the directory
    size_t bucket_size_;  // The size of a bucket
    int num_buckets_;     // The number of buckets in the hash table
    mutable std::mutex latch_;
    std::vector<std::shared_ptr<Bucket>> dir_;  // The directory of the hash
table
```

global_depth与local_depth 的理解：它的类成员命名为 std::vectorstd::shared_ptr> **dir**; 这可见其思想就是借鉴的目录，global_depth表示目录树（Bucker）的最大深度，local_depth表示当前这个目录（Bucker）的深度。目录数组（`dir_`）中的每个元素指向一个桶（bucket）。目录数组的大小是 2 的幂次方，而全局深度（`global_depth_`）表示目录数组的大小为 2global_depth 2global_depth 。每个桶有一个局部深度（`local_depth`），表示它在目录数组中被引用的次数。如果全局深度是 4，而桶的局部深度是 2，那么目录数组中会有 24−2=424−2=4 个项指向同一个桶。

Task #2 - LRU-K Replacement Policy

这段代码是LRU-K页面置换算法中的一部分，LRU-K是一种改进的最近最少使用（LRU）算法，它考虑了页面在过去k次访问中的使用情况，以更好地预测将来可能不会被使用的页面。

在LRU-K算法中，页面被分为两个列表：一个先进先出（FIFO）队列和一个最近最少使用（LRU）队列。FIFO队列用于存储最近访问次数少于k次的页面，而LRU队列用于存储访问次数至少为k次的页面。当页面被访问时，根据其访问次数和当前状态进行相应的更新。

问题

在LRU-K算法中，当 (K = 1) 时，算法的行为确实会有一些特殊之处，但仍然遵循LRU-K的基本原则。让我们详细分析一下你给出的访问序列和算法的行为。

- 时刻 1：访问页面 a
- 时刻 2：访问页面 a
- 时刻 3：访问页面 b
- 时刻 4：访问页面 b
- 时刻 5：访问页面 a

时刻 5 时的状态

- 页面 a 的访问历史：时刻 1, 时刻 2, 时刻 5
- 页面 b 的访问历史：时刻 3, 时刻 4

LRU-K 算法的行为

1. **Backward 1-distance**：

- 对于页面 a，时刻 5 的访问，其前一次访问是在时刻 2，因此其Backward 1-distance为 (5 - 2 = 3)。
- 对于页面 b，时刻 4 的访问，其前一次访问是在时刻 3，因此其Backward 1-distance为 (4 - 3 = 1)。

2. **驱逐规则**：
   - LRU-K算法会驱逐Backward 1-distance最大的页面。在时刻 5，页面 a 的Backward 1-distance为 3，页面 b 的Backward 1-distance为 1。因此，页面 a 的Backward 1-distance更大。
   - 但是，根据你的代码实现，页面 b 会在时刻 4 被添加到 LRU 队列中，而页面 a 在时刻 5 被再次访问时，会从 FIFO 队列移动到 LRU 队列的末尾。因此，LRU 队列的顺序会是 b, a。

Task #3 - Buffer Pool Manager Instance

```
//===----------------------------------------------------------------------===//
//
//                         BusTub
//
// buffer_pool_manager_instance.h
//
// Identification: src/include/buffer/buffer_pool_manager.h
//
// Copyright (c) 2015-2021, Carnegie Mellon University Database Group
//
//===----------------------------------------------------------------------===//

#pragma once

#include <list>
#include <mutex>  // NOLINT
#include <unordered_map>

#include "buffer/buffer_pool_manager.h"
#include "buffer/lru_k_replacer.h"
#include "common/config.h"
#include "container/hash/extendible_hash_table.h"
#include "recovery/log_manager.h"
#include "storage/disk/disk_manager.h"
#include "storage/page/page.h"

namespace bustub {

/**
 * BufferPoolManager reads disk pages to and from its internal buffer pool.
 */
class BufferPoolManagerInstance : public BufferPoolManager {
 public:
  /**
   * @brief Creates a new BufferPoolManagerInstance.
   * @param pool_size the size of the buffer pool
   * @param disk_manager the disk manager
   * @param replacer_k the lookback constant k for the LRU-K replacer
   * @param log_manager the log manager (for testing only: nullptr = disable
logging). Please ignore this for P1.
   */
  BufferPoolManagerInstance(size_t pool_size, DiskManager *disk_manager, size_t
replacer_k = LRUK_REPLACER_K,
```

```
                            LogManager *log_manager = nullptr);

  /**
   * @brief Destroy an existing BufferPoolManagerInstance.
   */
  ~BufferPoolManagerInstance() override;

  /** @brief Return the size (number of frames) of the buffer pool. */
  auto GetPoolSize() -> size_t override { return pool_size_; }

  /** @brief Return the pointer to all the pages in the buffer pool. */
  auto GetPages() -> Page * { return pages_; }

 protected:
  /**
   * TODO(P1): Add implementation
   *
   * @brief Create a new page in the buffer pool. Set page_id to the new page's
id, or nullptr if all frames
   * are currently in use and not evictable (in another word, pinned).
   *
   * You should pick the replacement frame from either the free list or the
replacer (always find from the free list
   * first), and then call the AllocatePage() method to get a new page id. If
the replacement frame has a dirty page,
   * you should write it back to the disk first. You also need to reset the
memory and metadata for the new page.
   *
   * Remember to "Pin" the frame by calling replacer.SetEvictable(frame_id,
false)
   * so that the replacer wouldn't evict the frame before the buffer pool
manager "Unpin"s it.
   * Also, remember to record the access history of the frame in the replacer
for the lru-k algorithm to work.
   *
   * @param[out] page_id id of created page
   * @return nullptr if no new pages could be created, otherwise pointer to new
page
   */
  auto NewPgImp(page_id_t *page_id) -> Page * override;

  /**
   * TODO(P1): Add implementation
   *
   * @brief Fetch the requested page from the buffer pool. Return nullptr if
page_id needs to be fetched from the disk
   * but all frames are currently in use and not evictable (in another word,
pinned).
   *
   * First search for page_id in the buffer pool. If not found, pick a
replacement frame from either the free list or
   * the replacer (always find from the free list first), read the page from
disk by calling disk_manager_->ReadPage(),
   * and replace the old page in the frame. Similar to NewPgImp(), if the old
page is dirty, you need to write it back
   * to disk and update the metadata of the new page
   *
```

```
   * In addition, remember to disable eviction and record the access history of
the frame like you did for NewPgImp().
   *
   * @param page_id id of page to be fetched
   * @return nullptr if page_id cannot be fetched, otherwise pointer to the
requested page
   */
  auto FetchPgImp(page_id_t page_id) -> Page * override;

  /**
   * TODO(P1): Add implementation
   *
   * @brief Unpin the target page from the buffer pool. If page_id is not in the
buffer pool or its pin count is already
   * 0, return false.
   *
   * Decrement the pin count of a page. If the pin count reaches 0, the frame
should be evictable by the replacer.
   * Also, set the dirty flag on the page to indicate if the page was modified.
   *
   * @param page_id id of page to be unpinned
   * @param is_dirty true if the page should be marked as dirty, false otherwise
   * @return false if the page is not in the page table or its pin count is <= 0
before this call, true otherwise
   */
  auto UnpinPgImp(page_id_t page_id, bool is_dirty) -> bool override;

  /**
   * TODO(P1): Add implementation
   *
   * @brief Flush the target page to disk.
   *
   * Use the DiskManager::WritePage() method to flush a page to disk, REGARDLESS
of the dirty flag.
   * Unset the dirty flag of the page after flushing.
   *
   * @param page_id id of page to be flushed, cannot be INVALID_PAGE_ID
   * @return false if the page could not be found in the page table, true
otherwise
   */
  auto FlushPgImp(page_id_t page_id) -> bool override;

  /**
   * TODO(P1): Add implementation
   *
   * @brief Flush all the pages in the buffer pool to disk.
   */
  void FlushAllPgsImp() override;

  /**
   * TODO(P1): Add implementation
   *
   * @brief Delete a page from the buffer pool. If page_id is not in the buffer
pool, do nothing and return true. If the
   * page is pinned and cannot be deleted, return false immediately.
   *
   * After deleting the page from the page table, stop tracking the frame in the
replacer and add the frame
```

```
   * back to the free list. Also, reset the page's memory and metadata. Finally,
you should call DeallocatePage() to
   * imitate freeing the page on the disk.
   *
   * @param page_id id of page to be deleted
   * @return false if the page exists but could not be deleted, true if the page
didn't exist or deletion succeeded
   */
  auto DeletePgImp(page_id_t page_id) -> bool override;

  /** Number of pages in the buffer pool. */
  const size_t pool_size_;
  /** The next page id to be allocated  */
  std::atomic<page_id_t> next_page_id_ = 0;
  /** Bucket size for the extendible hash table */
  const size_t bucket_size_ = 4;

  /** Array of buffer pool pages. */
  Page *pages_;
  /** Pointer to the disk manager. */
  DiskManager *disk_manager_ __attribute__((__unused__));
  /** Pointer to the log manager. Please ignore this for P1. */
  LogManager *log_manager_ __attribute__((__unused__));
  /** Page table for keeping track of buffer pool pages. */
  ExtendibleHashTable<page_id_t, frame_id_t> *page_table_;
  /** Replacer to find unpinned pages for replacement. */
  LRUKReplacer *replacer_;
  /** List of free frames that don't have any pages on them. */
  std::list<frame_id_t> free_list_;
  /** This latch protects shared data structures. We recommend updating this
comment to describe what it protects. */
  std::mutex latch_;

  /**
   * @brief Allocate a page on disk. Caller should acquire the latch before
calling this function.
   * @return the id of the allocated page
   */
  auto AllocatePage() -> page_id_t;

  /**
   * @brief Deallocate a page on disk. Caller should acquire the latch before
calling this function.
   * @param page_id id of the page to deallocate
   */
  void DeallocatePage(__attribute__((unused)) page_id_t page_id) {
    // This is a no-nop right now without a more complex data structure to track
deallocated pages
  }

  // TODO(student): You may add additional private members and helper functions
};
}  // namespace bustub
```

Project #2 - B+Tree

Task #1 - B+Tree Pages

B+树父页面内容

Variable Name 变量名	Size 大小	Description 描述
page_type_ 页面类型	4	Page Type (internal or leaf) 页面类型（内部或叶）
lsn_	4	Log sequence number (Used in Project 4) 日志序列号（用于项目4）
size_ 尺寸_	4	Number of Key & Value pairs in page 页面中的键值对数量
max_size_ 最大尺寸	4	Max number of Key & Value pairs in page 页面中的键和值对的最大数量
parent_page_id_ 父页面id	4	Parent Page Id 父页面ID
page_id_ 页面ID	4	Self Page Id 自页ID

```
auto BPlusTreePage::GetMinSize() const -> int {
  if (IsLeafPage()) {
    return max_size_ / 2;       // 7/2=3
  }
  return (max_size_ - 1) / 2 + 1;  // (7-1)/2 +1 = 4 至少有一半是满的
}
```

内部页面和叶页面都继承自父页面，在任何时候，每个内部页面至少有一半是满的。

内部页面不存储任何真实的数据，而是存储**有序**的**m个**键条目和**m+1个**子指针（即page_id）。由于指针的数量不等于键的数量，因此第一个键被设置为无效，查找方法应始终从第二个键开始。叶页存储有序的**m个**键条目和**m个**值条目。内部页面的key记录的是叶子节点的最小key，value是该中间节点的子节点的page_id（子节点不一定是叶子节点，可能还是中间节点）。但是叶子节点的key和value值也是不一样的，key的含义是该叶子节点存的数据的key值,而value比较复杂是rid类型（相当于一个指向真正数据的指针，因为rid里有page_id和slot_number可以去磁盘页里取到真正的数据）。

例如m = 3, 那么内部结点的个数就可以是3。而叶子结点则最多是2，但是内部结点的array[0]实际上就是个存地址的。

```
 * Internal page format (keys are stored in increasing order):
 *  ---------------------------------------------------------------------------
 * | HEADER | KEY(1)+PAGE_ID(1) | KEY(2)+PAGE_ID(2) | ... | KEY(n)+PAGE_ID(n) |
 *  ---------------------------------------------------------------------------

  * Leaf page format (keys are stored in order):
 *  ---------------------------------------------------------------------
 * | HEADER | KEY(1) + RID(1) | KEY(2) + RID(2) | ... | KEY(n) + RID(n)
 *  ---------------------------------------------------------------------
```

- 查找Key的时候，用的是二分查找。
- 插入K V的时候先吧index之后的往后挪，然后插入在index+1的位置。

```
INDEX_TEMPLATE_ARGUMENTS
void B_PLUS_TREE_INTERNAL_PAGE_TYPE::MoveTo(BPlusTreeInternalPage *other_node,
BufferPoolManager *bpm) {
  for (int i = GetMinSize(); i < GetMaxSize() + 1; i++) {
    other_node->SetKeyAt(i - GetMinSize(), KeyAt(i));
    other_node->SetValueAt(i - GetMinSize(), ValueAt(i));
    auto child_node = reinterpret_cast<BPlusTreePage *>(bpm-
>FetchPage(ValueAt(i))->GetData());
    child_node->SetParentPageId(other_node->GetPageId());
    other_node->IncreaseSize(1);
    this->IncreaseSize(-1);
    bpm->UnpinPage(child_node->GetPageId(), true);
  }
}
```

```
INDEX_TEMPLATE_ARGUMENTS
void B_PLUS_TREE_LEAF_PAGE_TYPE::MoveTo(BPlusTreeLeafPage *other_node) {
  for (int i = GetMaxSize() / 2; i < GetMaxSize(); i++) {
    other_node->SetKeyAt(i - GetMaxSize() / 2, KeyAt(i));
    other_node->SetValueAt(i - GetMaxSize() / 2, ValueAt(i));
    other_node->IncreaseSize(1);
    this->IncreaseSize(-1);
  }
}
```

Task 2.A - B+TREE DATA STRUCTURE (INSERTION & POINT SEARCH)

```
INDEX_TEMPLATE_ARGUMENTS
class BPlusTree {
  using InternalPage = BPlusTreeInternalPage<KeyType, page_id_t, KeyComparator>;
  using LeafPage = BPlusTreeLeafPage<KeyType, ValueType, KeyComparator>;

 public:
  explicit BPlusTree(std::string name, BufferPoolManager *buffer_pool_manager,
const KeyComparator &comparator,
                     int leaf_max_size = LEAF_PAGE_SIZE, int internal_max_size =
INTERNAL_PAGE_SIZE);

  // Returns true if this B+ tree has no keys and values.
  auto IsEmpty() const -> bool;

  // Insert a key-value pair into this B+ tree.
  auto Insert(const KeyType &key, const ValueType &value, Transaction
*transaction = nullptr) -> bool;

  auto FindLeafPage(const KeyType &key) -> LeafPage *;

  template <typename ClassType>
  auto Split(ClassType *origin_node) -> ClassType *;

  template <typename ClassType>
```

```
  void InsertIntoParent(ClassType *origin_node, const KeyType &key, ClassType
*new_node);
  // Remove a key and its value from this B+ tree.
  void Remove(const KeyType &key, Transaction *transaction = nullptr);

  template <typename ClassType>
  void DeleteEntry(const KeyType &key, ClassType *delete_node);

  template <typename ClassType>
  void Borrow(ClassType *left_node, ClassType *right_node);

  template <typename ClassType>
  void Merge(ClassType *left_node, ClassType *right_node);

  // return the value associated with a given key
  auto GetValue(const KeyType &key, std::vector<ValueType> *result, Transaction
*transaction = nullptr) -> bool;

  // return the page id of the root node
  auto GetRootPageId() -> page_id_t;

  // index iterator
  auto FindLeafPage(bool left) -> LeafPage *;
  auto Begin() -> INDEXITERATOR_TYPE;
  auto Begin(const KeyType &key) -> INDEXITERATOR_TYPE;
  auto End() -> INDEXITERATOR_TYPE;

  // print the B+ tree
  void Print(BufferPoolManager *bpm);

  // draw the B+ tree
  void Draw(BufferPoolManager *bpm, const std::string &outf);

  // read data from file and insert one by one
  void InsertFromFile(const std::string &file_name, Transaction *transaction =
nullptr);

  // read data from file and remove one by one
  void RemoveFromFile(const std::string &file_name, Transaction *transaction =
nullptr);

 private:
  void UpdateRootPageId(int insert_record = 0);

  /* Debug Routines for FREE!! */
  void ToGraph(BPlusTreePage *page, BufferPoolManager *bpm, std::ofstream &out)
const;

  void ToString(BPlusTreePage *page, BufferPoolManager *bpm) const;

  // member variable
  std::string index_name_;
  page_id_t root_page_id_;
  BufferPoolManager *buffer_pool_manager_;
  KeyComparator comparator_;
  int leaf_max_size_;
  int internal_max_size_;
};
```

- **插入一个KV**

先判断B+树是否为空，为空则创建一个根结点（叶子结点），创建新结点是通过`buffer_pool_manager_`来获取的，创建后要`unpinPage`。如果不为空，则这个Key对应所在的叶节点，这个查找的过程是，`buffer_pool_manager`通过`root_page_id`去获取一个page，调用内部页面的Findkey找到key所在的pageid，循环找，直到找到的page为叶结点。然后调用叶结点的插入函数（如下，是二分查找，都往后挪一位插入）。插入后的叶页面达到`leaf_max_size_`则调用`Split`分裂页面

```
INDEX_TEMPLATE_ARGUMENTS
auto B_PLUS_TREE_LEAF_PAGE_TYPE::Insert(const KeyType &key, const ValueType
&value, KeyComparator &comparator) -> bool {
  int left = 0;
  int right = GetSize() - 1;
  if (GetSize() == 0) {
    SetKeyAt(0, key);
    SetValueAt(0, value);
    IncreaseSize(1);
    return true;
  }
  if (GetSize() == 1) {
    if (comparator(key, KeyAt(left)) < 0) {
      SetKeyAt(1, KeyAt(0));
      SetValueAt(1, ValueAt(0));
      SetKeyAt(left, key);
      SetValueAt(left, value);
      IncreaseSize(1);
    } else if (comparator(key, KeyAt(left)) > 0) {
      SetKeyAt(1, key);
      SetValueAt(1, value);
      IncreaseSize(1);
    } else if (comparator(key, KeyAt(left)) == 0) {
      return false;
    }
    return true;
  }
  while (left <= right) {
    int mid = left + (right - left) / 2;
    if (comparator(key, KeyAt(mid)) < 0) {
      right = mid - 1;
    } else if (comparator(key, KeyAt(mid)) > 0) {
      left = mid + 1;
    } else if (comparator(key, KeyAt(mid)) == 0) {
      return false;
    }
  }
  for (int i = GetSize(); i >= left + 1; i--) {
    SetKeyAt(i, KeyAt(i - 1));
    SetValueAt(i, ValueAt(i - 1));
  }
  SetKeyAt(left, key);
  SetValueAt(left, value);
  IncreaseSize(1);
  return true;
}
```

- 分裂叶页面（Split）

创建一个新page和源page有相同的size和parent。调用`MoveTo`函数把后半是数据移动到新page里。把新page插入到父结点里面。建立好源page和新page之间的顺序关系。返回插入完成的新page。新page插入到父结点分为两部分：如果是根结点，新建page然后取代原来的root，否则查找在父结点找源page第一个key所在的位置，在之后插入新page的第一个key，如果此时插入后的父结点大小超过了`internal_max_size`，则需要递归调用`Split`。

递归调用中待分裂结点为内部结点时，分裂过程和叶页面类似，但在moveTo过程中，除了移动后半内容到新page，还需要建立新page和子页面的联系。在叶子结点split的时候需要进行双向链表的维护，而在内部结点则不需要，共有操作都是获得一个新页--> 类型转换 ---> MoveTo

- **删除一个KV**

调用`Remove`删除Key，通过`FindLeafPage`找到叶结点，然后调用`DeleteEntry`删除该结点里的Key，真正删除的操作由结点的成员函数`Delete`函数执行，删除后需要判断大小，是否要合并还是借，这个过程是在`DeleteEntry`里执行。内部页面至少有一半是满的，而对叶页面没有这个要求，所以叶页面都是直接删了Key就好。

需要调整的判断分支：1. 是根节点但不是只有根节点——把左边的结点提上来作为根结点（`delete_internal_node->ValueAt(0)`）这句也说明了为什么内部结点第一个键被设置为无效（因为设置为地址了），查找方法应始终从第二个键开始。2. 如果是内部结点（非根）：先通过父结点取出左右兄弟结点（如果有的话，比如index为0则没有左兄弟），然后再判断是要合并还是要借，加起来的size小于最大值则合并，反之借，这种判断是先判断左边再判断右边，即优先找左边进行合并或借。在`merge`里面有递归调用`DeleteEntry`.

这b+树和408里面教的不一样！！！！结构不一样！！！气死了，我还以为我记错了，怎么合并操作不一样。

如何考虑并发搜索的问题？搜索怎么加快？新建、移动页面的过程耗时太多怎么办？

Task #3 - Index Iterator

实现了一个针对叶结点的迭代器。重载了* == != ++ 运算符。++的逻辑就是依次往后++，找到头了就跳到下一个leaf_node。

```
INDEX_TEMPLATE_ARGUMENTS
auto INDEXITERATOR_TYPE::operator++() -> INDEXITERATOR_TYPE & {
  if ((pos_ == leaf_node_->GetSize() - 1 && leaf_node_->GetNextPageId() ==
INVALID_PAGE_ID) ||
      pos_ + 1 <= leaf_node_->GetSize() - 1) {
    pos_++;
  } else {
    auto leaf_next_node = reinterpret_cast<BPlusTreeLeafPage<KeyType, ValueType,
KeyComparator> *>(
        buffer_pool_manager_->FetchPage(leaf_node_->GetNextPageId())-
>GetData());
    leaf_node_ = leaf_next_node;
    pos_ = 0;
  }
  return *this;
}
```

Task #4 - Concurrent Index

更新原来的单线程B+Tree索引，使其能够支持并发操作——latch crabbing

- **乐观锁、悲观锁**

乐观的假设大部分操作是不需要进行合并和分裂的。因此在我们向下的时候都是读Latch而不是写Latch。只有在叶子结点才是write Latch，当我们到最后一步发现不安全的时候。则需要像没有引入乐观🔓的时候一样。重新执行一遍，即对不同的操作，加读写不同的锁。

	lock	latch
对象	事务	线程
保护	数据库内容	内存数据结构
持续时间	整个事务过程	临界资源
模式	行锁、表锁、意向锁	读写锁、互斥量
死锁	通过 waits-for graph、time out 等机制进行死锁检测与处理	无死锁检测与处理机制。仅通过应用程序加锁的顺序（lock leveling）保证无死锁的情况发生
存在于	Lock Manager 的哈希表中	每个数据结构的对象中

每个Page自带的读写锁实现

```
/**
 * Reader-Writer latch backed by std::mutex.
 */
class ReaderWriterLatch {
 public:
  /**
   * Acquire a write latch.
   */
  void WLock() { mutex_.lock(); }

  /**
   * Release a write latch.
   */
  void WUnlock() { mutex_.unlock(); }

  /**
   * Acquire a read latch.
   */
  void RLock() { mutex_.lock_shared(); }

  /**
   * Release a read latch.
   */
  void RUnlock() { mutex_.unlock_shared(); }

 private:
  std::shared_mutex mutex_;
};
```

这部分的实现貌似并没有。于是看的网上的别人写的。

并发的原理如Reference 1。在`FindLeafPage`中使用的每次存查找过程的结点是用`Transaction`类，里面用队列存查找时的结点并进行管理。transaction->GetPageSet(); 就是之前访问过的page集合。

Project #3 - Query Execution

Task #1 - Access Method Executors

在`ExecutorFactory`工厂里根据需要生产执行器（比如`SeqScan`）,每个执行器继承自AbstractExecutor，重写了`Init`、`Next`、`GetOutputSchema`三个方法。Next方法是一个递归调用的过程。关于BusTub表结构参考CSND的总结

SeqScan

实现逻辑

1. 使用`exec_ctx`属性获取对应的TableInfo
2. 调用`MakeIterator`方法，获取表的迭代器
3. 在Next方法中，每次利用迭代器获得一个满足条件的元组(检查元组是否被删除、元组是否满足filter)

Insert

实现逻辑

1. 在Next方法中调用`child_executor_->Next`获取要插入的元组
2. 使用TableHeap类的InsertTuple方法插入元组
3. 插入索引

Delete

实现逻辑

1. 在Next方法中调用`child_executor_->Next`获取要删除的元组
2. 利用TableHeap类的UpdateTupleMeta方法，将对应元组标记为删除
3. 删除对应的索引

SeqScan优化为IndexScan

1. 当计划类型为PlanType::SeqScan，尝试进行优化
2. 遍历filter_predicate_中的表达式，当满足有且只有一个索引时，构造`IndexScanPlanNode`并且返回

IndexScan

1. 获取哈希表

```
htable_ = dynamic_cast<HashTableIndexForTwoIntegerColumn *>(index_info_-
>index_.get())
```

1. 利用`ScanKey`方法获取对应的元素即可

Task #2 - Aggregation & Join Executors

Aggregation

和之前的算子不同，Aggregation算子的核心流程在`Init`方法中完成，而`Next`方法仅用于返回结果。

1. 利用`child_--->Next`方法遍历所有元组，通过`InsertCombine()`将相应的聚合值聚合到到相应的聚合组中（哈希表——聚合键-聚合值）。具体的聚合规则由`CombineAggregateValues()`函数中的实现来决定。
2. `Next()`函数会通过哈希迭代器依次获取每个聚合组的键与值，返回给父执行器。

NestedLoopJoin

NestedLoopJoin算子也应当在`Init`方法中就处理完逻辑，`Next`方法仅做结果的返回

1. 将左侧执行器`left_executor_`和右侧执行器`right_executor_`的结果保存到两个vector中
2. 使用一个两层嵌套循环，将满足`plan_->predicate_->EvaluateJoin`的元组保存到结果集合中即可
3. 对于left join，当遍历完右侧所有元组仍然没有满足条件的元组时，需要构造空值添加进结果集

NestedLoopJoin优化为NestIndexJoinExecutor

1. 当计划类型为`PlanType::NestedLoopJoin`时，尝试进行优化
2. 只有当连接条件是一个或者多个等值连接时，并且连接的右侧具有该条件的索引，才应当进行优化

NestIndexJoinExecutor

1. 构造一个用于Hash Join的`SimpleHashJoinHashTable`
2. 获取左右子节点的结果集，并利用右子节点的结果集构造哈希表
3. 遍历左子节点结果集，对于每一个元组查找哈希表并尝试进行连接即可

Task #3 - Sort + Limit Executors and Top-N Optimization

Sort

1. 实现Comparator比较类，重载（）方法。Value类自带`CompareEquals`，`CompareLessThan`，`CompareGreaterThan`方法

```
auto operator()(const Tuple &t1, const Tuple &t2) -> bool {
    for (auto const &order_by : this->order_bys_) {
      const auto order_type = order_by.first;
      // 使用Evaluate获取值
      AbstractExpressionRef expr = order_by.second;
      Value v1 = expr->Evaluate(&t1, *schema_);
      Value v2 = expr->Evaluate(&t2, *schema_);
      if (v1.CompareEquals(v2) == CmpBool::CmpTrue) {
        continue;
      }
      // 如果是升序（ASC 或 DEFAULT），比较 v1 是否小于 v2（CompareLessThan）
      if (order_type == OrderByType::ASC || order_type == OrderByType::DEFAULT)
{
        return v1.CompareLessThan(v2) == CmpBool::CmpTrue;
      }
      // 如果是降序（DESC），比较 v1 是否大于 v2（CompareGreaterThan）
      return v1.CompareGreaterThan(v2) == CmpBool::CmpTrue;
    }
    // 两个元组所有键都相等
    return false;
}
```

2. 在Init里使用`std::sort`实现。

```
// 获取排序字段
  auto order_by = plan_->GetOrderBy();
  // 排序
  std::sort(tuples_.begin(), tuples_.end(), Comparator(&this->GetOutputSchema(),
order_by));
```

3. Next仅返回结果。

Limit Executors

获取符号数量的结果。

```
// 获取符合条件数量的元组
  while (count < limit && child_executor_->Next(&tuple, &rid)) {
    count++;
    tuples_.emplace_back(tuple);
  }
  if (!tuples_.empty()) {
    iter_ = tuples_.begin();
  }
```

Top-N Optimization

1. 定义一个可以排序的（HeapComparator实现）、存储top-n元组的堆（底层是vector）。
2. Init里调用子执行器`child_executor_->Next()`获取元组加入优先队列。

```
void TopNExecutor::Init() {
  child_executor_->Init();
  //使用优先队列存储topN，升序用大顶堆，降序用小顶堆
  std::priority_queue<Tuple, std::vector<Tuple>, HeapComparator> heap(
      HeapComparator(&this->GetOutputSchema(), plan_->GetOrderBy()));
  Tuple tuple{};
  RID rid{};
  //遍历子执行器，将子执行器返回的元组加入优先队列
  while (child_executor_->Next(&tuple, &rid)) {
    heap.push(tuple);
    heap_size_++;
    //因为只需要topN个元组，所以当优先队列大小大于topN时，弹出堆顶元组（如果是升序，堆顶是最大
的元组，如果是降序，堆顶是最小的元组）
    if (heap.size() > plan_->GetN()) {
      heap.pop();
      heap_size_--;
    }
  }
  while (!heap.empty()) {
    this->top_entries_.push(heap.top());
    heap.pop();
  }
}
```

3. Next里输出结果。

优化：将带有 ORDER BY + LIMIT 子句的查询转换为使用 TopNExecutor。

Project #4 - Concurrency Control

并发控制采用 2PL 阶段锁设计，实现 RU，RC，RR3 种隔离级别，实现全局 Lock Manager 管理 R/W 锁，死锁处理基于 wound-wait 算法。

事务（transaction）是数据库（DBMS）状态变更的基本单元，包含一个或多个操作（例如多条SQL语句）。一个事务只能有执行和未执行两个状态，不存在部分执行的状态。——ACID（原子性、一致性、隔离性、持久性）。

在**并发环境**中，多个事务可能同时访问和修改相同的数据，如果没有严格的并发控制，可能会导致以下**问题**：

- **脏读（Dirty Read）**：一个事务读取了另一个未提交事务的数据。
- **不可重复读（Non-Repeatable Read）**：一个事务两次读取同一数据，结果不一致。
- **幻读（Phantom Read）**：一个事务两次查询同一范围的数据，结果集不一致。

事务隔离级别定义了事务在并发执行时如何与其他事务的修改交互，选择隔离级别的考虑因素**数据一致性要求**、**并发性能需求**、**应用场景**

隔离级别对比

隔离级别	脏读（Dirty Read）	不可重复读（Non-Repeatable Read）	幻读（Phantom Read）	性能	适用场景
Read Uncommitted	可能	可能	可能	最高	对数据一致性要求极低，且需要高并发的场景（如统计分析）
Read Committed	避免	可能	可能	较高	是许多数据库系统（如 PostgreSQL、Oracle）的默认隔离级别
Repeatable Read	避免	避免	可能	中等	是 MySQL InnoDB 存储引擎的默认隔离级别
Serializable	避免	避免	避免	最低	金融系统。可能导致死锁

两段锁协议(2PL):（Two-phase locking）是数据库最常见的基于锁的并发控制协议，简单来说就将事务划分为两个阶段,生长阶段(growing/expanding)和衰退阶段(shrinking),生长阶段加锁,衰退阶段解锁。2PL 确保事务之间的冲突操作（如读写、写写）会按照锁的顺序执行，从而保证并发执行的结果与某种串行执行顺序一致。缺点：2PL 可能导致级联中止——事务2读了事务1修改的数据，而事务1终止后，事务2就会变成脏堵也要回滚。——避免办法：**S2PL（严格两阶段锁）：每个事务在结束之前，其写过的数据不能被其它事务读取或者重写**

死锁

互斥条件、请求和保持、循环等待、不可剥夺。

死锁预防（Deadlock Prevention）

死锁预防是一种事前行为，采用这种方案的 DBMS 无需维护 waits-for 图，也不需要实现 detection 算法，而是在事务尝试获取其它事务持有的锁时直接决定是否需要将其中一个事务中止。

通常死锁预防会按照事务的年龄来赋予优先级，事务的时间戳越老，优先级越高。有两种预防策略：

1. **Old Waits for Young**：如果 requesting txn 优先级比 holding txn 更高则等待后者释放锁；更低则自行中止
2. **Young Waits for Old**：如果 requesting txn 优先级比 holding txn 更高则后者自行中止释放锁，让前者获取锁，否则 requesting txn 等待 holding txn 释放锁——**Would-Wait**算法

死锁检测（Deadlock Detection）

数据库系统根据 waits-for 图来跟踪每个事务正在等待 (释放锁) 的其它事务。系统会定期地检查 waits-for图，如果发现环，则需要选择一个"受害者" 事务 Abort。

"受害者" 的指标可能有很多：事务持续时间、事务的进度、事务锁住的数据数量、级联事务的数量、事务曾经重启的次数等等。

分层锁定的锁模式

在分层锁定中，常见的锁模式包括：

1. **意向锁（Intention Locks）**
   意向锁是一种轻量级锁，用于表示事务可能在某个子节点上请求更细粒度的锁。常见的意向锁包括：
   - **意向共享锁（IS）**：表示事务可能在子节点上请求共享锁（S）。
   - **意向排他锁（IX）**：表示事务可能在子节点上请求排他锁（X）。
   - **共享意向排他锁（SIX）**：表示事务已经持有共享锁，并可能在子节点上请求排他锁。
2. **共享锁（S）和排他锁（X）**
   - **共享锁（S）**：允许多个事务同时读取资源。
   - **排他锁（X）**：只允许一个事务读写资源。

Task #1 - Lock Manager

```
/**
 * Transaction states for 2PL:
 *
 *     _________________________
 *    |                         v
 * GROWING -> SHRINKING -> COMMITTED   ABORTED
 *    |__________|________________________^
 *
 * Transaction states for Non-2PL:
 *     __________
 *    |          v
 * GROWING  -> COMMITTED     ABORTED
 *    |_____________________________^
 *
 **/
```

txn有四个状态：GROWING -> SHRINKING -> COMMITTED ABORTED，对于三种隔离级别（RU、RC、RR）在不同事务状态时操作不同。

LOCK	required	GROWING	SHRINKING
RU	IX、X	IX、X通过。若S/IS/SIX，抛`LOCK_SHARED_ON_READ_UNCOMMITTED`	若为 IX/X 锁，抛`LOCK_ON_SHRINKING`，否则抛`LOCK_SHARED_ON_READ_UNCOMMITTED`
RC	ALL	ALL通过	IS、S，否则抛`LOCK_ON_SHRINKING`
RR	ALL	ALL通过	造成事务终止，并抛出`LOCK_ON_SHRINKING`异常

UNLOCK	S	X
RU	S 锁不会出现	X 锁释放使事务 Shrinking
RC		X 锁释放使事务进入 Shrinking 状态
RR	S锁释放会使事务进入 Shrinking 状态	X 锁释放会使事务进入 Shrinking 状态

首先理一理 Lock Manager 的结构：

- `table_lock_map_`：记录 table 和与其相关锁请求的映射关系。
- `row_lock_map_`：记录 row 和与其相关锁请求的映射关系。

这两个 map 的值均为锁请求队列 `LockRequestQueue`：

- `request_queue_`：实际存放锁请求的队列。
- `cv_` & `latch_`：条件变量和锁，配合使用可以实现经典的等待资源的模型。
- `upgrading_`：正在此资源上尝试锁升级的事务 id。

锁请求以 `LockRequest` 类表示：

- `txn_id_`：发起此请求的事务 id。
- `lock_mode_`：请求锁的类型。
- `oid_`：在 table 粒度锁请求中，代表 table id。在 row 粒度锁请求中，表示 row 属于的 table 的 id。
- `rid_`：仅在 row 粒度锁请求中有效。指 row 对应的 rid。
- `granted_`：是否已经对此请求授予锁?

Lock Manager 的作用是处理事务发送的锁请求，例如有一个 SeqScan 算子需要扫描某张表，其所在事务就需要对这张表加 S 锁。而加读锁这个动作需要由 Lock Manager 来完成。事务先对向 Lock Manager 发起加 S 锁请求，Lock Manager 对请求进行处理。如果发现此时没有其他的锁与这个请求冲突，则授予其 S 锁并返回。如果存在冲突，例如其他事务持有这张表的 X 锁，则 Lock Manager 会阻塞此请求（即阻塞此事务），直到能够授予 S 锁，再授予并返回。

锁升级： While upgrading, only the following transitions should be allowed: IS -> [S, X, IX, SIX] S -> [X, SIX] IX -> [X, SIX] SIX -> [X]

以下代码可能不正确。是dk写的。

```
auto LockManager::LockShared(Transaction *txn, const RID &rid) -> bool {

    // 第一步，检查 txn 的状态
  if (txn->GetState() == TransactionState::ABORTED ||
TransactionState::COMMITTED ) {
    return false;
```

```
  }

  if (txn->GetState() == TransactionState::SHRINKING) {
    if(txn->GetIsolationLevel() == IsolationLevel::REPEATABLE_READ){
        // 造成事务终止，并抛出 `LOCK_ON_SHRINKING` 异常
        return false;
    }else if(txn->GetIsolationLevel() == IsolationLevel::READ_COMMITTED){
        // 若为 IS/S 锁，则正常通过，否则抛 LOCK_ON_SHRINKING。
        continue; // LockShared是S锁
    }else if(txn->GetIsolationLevel() == IsolationLevel::READ_UNCOMMITTED){
        // 若为 IX/X 锁，抛 LOCK_ON_SHRINKING，否则抛
LOCK_SHARED_ON_READ_UNCOMMITTED。
        return AbortReason::LOCK_SHARED_ON_READ_UNCOMMITTED;
    }
  }

    // 第二步，获取 table 对应的 lock request queue
  std::unique_lock<std::mutex> ul(latch_);
  auto &queue = lock_table_[rid];
  // 第三步，检查是否已持有该锁,检查此锁请求是否为一次锁升级
  for (const auto &req : queue.request_queue_) {
    if (req.txn_id_ == txn->GetTransactionId() && req.granted_) { // 已持有该锁
        if(req.lock_mode_ == LockMode::SHARED) {return true;}
        else{ // 准备尝试升级为X锁
            if(req->upgrading_ == INVALID_TXN_ID){// 没有事务在升级,判断兼容性
                // 删除原来的锁
            }else return  UPGRADE_CONFLICT;
        }
    }
  }

  // 添加共享锁请求
  queue.request_queue_.emplace_back(txn->GetTransactionId(), LockMode::SHARED);
  auto &request = queue.request_queue_.back();

  // 等待直到锁被授予或事务中止
  while (!request.granted_ && txn->GetState() != TransactionState::ABORTED) {
    if (ShouldAbort(txn)) { // 死锁预防策略
      txn->SetState(TransactionState::ABORTED);
      queue.request_queue_.remove(request);
      return false;
    }
    queue.cv_.wait(ul);
    ProcessLockRequests(&queue, rid); // 重新检查是否可以授予锁
  }
  return txn->GetState() != TransactionState::ABORTED;
}
```

```
auto LockManager::Unlock(Transaction *txn, const RID &rid) -> bool {
  std::unique_lock<std::mutex> ul(latch_);
  auto &queue = lock_table_[rid];

  // 查找事务的锁请求
  auto it = std::find_if(queue.request_queue_.begin(),
queue.request_queue_.end(),
                         [txn](const LockRequest &req) {
```

```
                            return req.txn_id_ == txn->GetTransactionId() &&
req.granted_;
                          });
  if (it == queue.request_queue_.end()) {
    txn->SetState(TransactionState::ABORTED);
    return false;
  }

  // 根据隔离级别处理阶段转换（例如，2PL在解锁后进入收缩阶段）
  if (txn->GetState() == TransactionState::GROWING &&
      txn->GetIsolationLevel() == IsolationLevel::REPEATABLE_READ) {
    txn->SetState(TransactionState::SHRINKING);
  }

  // 移除锁请求并唤醒其他事务
  queue.request_queue_.erase(it);
  ProcessLockRequests(&queue, rid);

  return true;
}
```

Task #2 Deadlock Detection

所谓的**上锁冲突**, 我们自然需要考虑S锁和X锁: （wound-wait算法）

1. 如果老事务的S锁请求前面有年轻事务的S锁请求, 那这个年轻事务不需要回滚; 但如果年轻事务有X锁请求则需要回滚
2. 老事务的X锁请求直接回滚任何前面的年轻事务。

我们用 wait for 图来表示事务之间的等待关系。wait for 是一个有向图，`t1->t2` 即代表 t1 事务正在等待 t2 事务释放资源。当 wait for 图中存在环时，即代表出现死锁，需要挑选事务终止以打破死锁。构建 wait for 图的过程是，遍历 `table_lock_map` 和 `row_lock_map` 中所有的请求队列，对于每一个请求队列，用一个二重循环将所有满足等待关系的一对 tid 加入 wait for 图的边集。满足等待关系是指，对于两个事务 a 和 b，a 是 waiting 请求，b 是 granted 请求，则生成 `a->b` 一条边。成功构建 wait for 图后，对 wait for 图实施环检测算法。常见的环检测算法包括深度优先搜索（DFS）、拓扑排序（Topological Sort）和并查集（Union-Find）

```
创建一个访问标记数组 visited，用于记录每个节点的访问状态。初始时，所有节点都未被访问。
创建一个递归栈 recStack，用于记录当前递归调用栈中的节点。这有助于检测回边，即指向正在访问的祖先节点的边。
对图中的每一个节点执行DFS。如果该节点尚未被访问，则从该节点开始进行DFS。
    在DFS过程中，对于当前访问的节点u：
    将节点u标记为已访问，并将其添加到递归栈中。
    遍历节点u的所有邻接节点v：
    如果节点v未被访问，则递归地对v进行DFS。
    如果节点v已在递归栈中，说明存在一个从v到u的回边，因此图中存在环。
    DFS完成后，将节点u从递归栈中移除。
如果在任何DFS调用中检测到环，则整个图包含环
```

挑选出 youngest 事务（ tid 最大）后，将此事务的状态设为 Aborted。并且在请求队列中移除此事务，释放其持有的锁，终止其正在阻塞的请求，并调用 `cv_.notify_all()` 通知正在阻塞的相关事务。此外，还需移除 wait for 图中与此事务有关的边。不是不用维护 wait for 图，每次使用重新构建吗？这是因为图中可能存在多个环，不是打破一个环就可以直接返回了。需要在死锁检测线程醒来的时候打破当前存在的所有环。

```
std::unique_lock<std::mutex> lock(queue->latch_);
while (!GrantLock(...)) {
    queue->cv_.wait(lock);
    if (txn->GetState() == Aborted) {
        // release resources
        return false;
    }
}
```

Task #3 Concurrent Query Execution

需修改 `SeqScan`、`Insert` 和 `Delete` 三个算子。其中 `Insert` 和 `Delete` 几乎完全一样，与 `SeqScan` 分别代表着写和读。

SeqScan

如果隔离级别是 `READ_UNCOMMITTED` 则无需加锁。加锁失败则抛出 `ExecutionException` 异常。

在 `READ_COMMITTED` 下，在 `Next()` 函数中，若表中已经没有数据，则提前释放之前持有的锁。在 `REPEATABLE_READ` 下，在 Commit/Abort 时统一释放，无需手动释放。

该加什么锁？直观上来说应该直接给表加 S 锁，但实际上会导致 MixedTest 用例失败。具体原因请看评论区。实际上需要给表加 IS 锁，再给行加 S 锁。另外，在 Leaderboard Test 里，我们需要实现一个 Predicate pushdown to SeqScan 的优化，即将 Filter 算子结合进 SeqScan 里，这样我们仅需给符合 predicate 的行加上 S 锁，减小加锁数量。

那么在实现了 Predicate pushdown to SeqScan 之后，有没有可以给表直接加 S 锁的情况？有，当 SeqScan 算子中不存在 Predicate 时，即需要全表扫描时，或许可以直接给表加 S 锁，避免给所有行全部加上 S 锁。

Insert & Delete

在 `Init()` 函数中，为表加上 IX 锁，再为行加 X 锁。同样，若获取失败则抛 `ExecutionException` 异常。另外，这里的获取失败不仅是结果返回 false，还有可能是抛出了 `TransactionAbort()` 异常，例如 `UPGRADE_CONFLICT`，需要用 try catch 捕获。

锁在 Commit/Abort 时统一释放，无需手动释放。

另外，在 Notes 里提到的

> you will need to maintain the write sets in transactions

似乎 `InsertTuple()` 函数里已经帮我们做好了，不需要维护 write set。而 index 的 write set 需要我们自己维护。

Reference

[已完结]CMU数据库(15-445)实验2-B+树索引实现(下) - 周小伦 - 博客园

CMU 15-445 Project #3 - Query Execution（Task #1、Task #2） bustub文件架构-CSDN博客

CMU 15-445(Fall 2023) Project3 Query Execution个人笔记 - 焚风 - 博客园

cmu15445 2023fall project3 详细过程（下）QUERY EXECUTION-CSDN博客

CMU15445 踩坑指南-PROJECT #4 - CONCURRENCY CONTROL_cmu15445 project4-CSDN博客

[做个数据库：2022 CMU15-445 Project4 Concurrency Control - 知乎]

项目问题

- shared_ptr在项目中的使用场景，比如每个表或者队列需要并发访问lock_request队列，这时候就可以用shared_ptr，unique_ptr经常作为工厂函数的返回值，比如生成的查询计划或者执行器。