Lab 3: B+ Tree 2

Instructor: Beom Heyn Kim

beomheynkim@hanyang.ac.kr

Department of Computer Science

Outline

- B+Tree Parent Page
- B+Tree Internal Page
- Assignment

B+Tree Parent Page

b_plus_tree_page.h

```
class BPlusTreePage {
 public:
  bool IsLeafPage() const;
  bool IsRootPage() const;
  void SetPageType(IndexPageType page_type);
  IndexPageType GetPageType() const;

  int GetSize() const;
```

Declare GetPageType() in the class definition of BPlusTreePage, which will be used to return the type of the page instance

B+Tree Parent Page

Implement Helper Functions

b_plus_tree_page.cpp

```cpp
namespace bustub {

/*
 * Helper methods to get/set page type
 * Page type enum class is defined in b_plus_tree_page.h
 */
bool BPlusTreePage::IsLeafPage() const { return page_type_ == IndexPageType::LEAF_PAGE; }
bool BPlusTreePage::IsRootPage() const { return parent_page_id_ == INVALID_PAGE_ID; }
void BPlusTreePage::SetPageType(IndexPageType page_type) { page_type_ = page_type; }
IndexPageType BPlusTreePage::GetPageType() const { return page_type_; }

/*
 * Helper methods to get/set size (number of key/value pairs stored in that
 * page)
 */
int BPlusTreePage::GetSize() const { return size_; }
void BPlusTreePage::SetSize(int size) { size_ = size; }
void BPlusTreePage::IncreaseSize(int amount) { size_ += amount; }

/*
 * Helper methods to get/set max size (capacity) of the page
 */
int BPlusTreePage::GetMaxSize() const { return max_size_; }
void BPlusTreePage::SetMaxSize(int size) { max_size_ = size; }
```

B+Tree Parent Page

Implement Helper Functions (Cont.)

b_plus_tree_page.cpp

```cpp
/*
 * Helper methods to get/set max size (capacity) of the page
 */
int BPlusTreePage::GetMaxSize() const { return max_size_; }
void BPlusTreePage::SetMaxSize(int size) { max_size_ = size; }

/*
 * Helper method to get min page size
 * Generally, min page size == max page size / 2
 */
int BPlusTreePage::GetMinSize() const { return max_size_ / 2; }

/*
 * Helper methods to get/set parent page id
 */
page_id_t BPlusTreePage::GetParentPageId() const { return parent_page_id_; }
void BPlusTreePage::SetParentPageId(page_id_t parent_page_id) { parent_page_id_ = parent_page_id; }

/*
 * Helper methods to get/set self page id
 */
page_id_t BPlusTreePage::GetPageId() const { return page_id_; }
void BPlusTreePage::SetPageId(page_id_t page_id) { page_id_ = page_id; }

/*
 * Helper methods to set lsn
 */
void BPlusTreePage::SetLSN(lsn_t lsn) { lsn_ = lsn; }

}  // namespace bustub
```

Outline

- B+Tree Parent Page
- B+Tree Internal Page
- Assignment

B+Tree Internal Page

b_plus_tree_internal.cpp

Implement Helper Functions

```
namespace bustub {
/*****************************************************************************
 * HELPER METHODS AND UTILITIES
 *****************************************************************************/
/*
 * Init method after creating a new internal page
 * Including set page type, set current size, set page id, set parent id and set
 * max page size
 */
INDEX_TEMPLATE_ARGUMENTS
void B_PLUS_TREE_INTERNAL_PAGE_TYPE::Init(page_id_t page_id, page_id_t parent_id, int max_size) {
  SetPageType(IndexPageType::INTERNAL_PAGE);
  SetSize(0);
  SetPageId(page_id);
  SetParentPageId(parent_id);
  SetMaxSize(max_size);
}
/*
 * Helper method to get/set the key associated with input "index"(a.k.a
 * array offset)
 */
INDEX_TEMPLATE_ARGUMENTS
KeyType B_PLUS_TREE_INTERNAL_PAGE_TYPE::KeyAt(int index) const { return array[index].first; }

INDEX_TEMPLATE_ARGUMENTS
void B_PLUS_TREE_INTERNAL_PAGE_TYPE::SetKeyAt(int index, const KeyType &key) { array[index].first = key; }

/*
 * Helper method to find and return array index(or offset), so that its value
 * equals to input "value"
 */
INDEX_TEMPLATE_ARGUMENTS
int B_PLUS_TREE_INTERNAL_PAGE_TYPE::ValueIndex(const ValueType &value) const {
  auto it = std::find_if(array, array + GetSize(), [&value](const auto &pair) { return pair.second == value; });
  return std::distance(array, it);
}

/*
 * Helper method to get the value associated with input "index"(a.k.a array
 * offset)
 */
INDEX_TEMPLATE_ARGUMENTS
ValueType B_PLUS_TREE_INTERNAL_PAGE_TYPE::ValueAt(int index) const { return array[index].second; }
```

array contains key value pairs in the node. array[index] returns MappingType object at the index position. Mapping Type is defined as std::pair<KeyType, ValueType>

B+Tree Internal Page

b_plus_tree_internal.cpp

```
/*****************************************************************************
 * LOOKUP
 *****************************************************************************/
/*
 * Find and return the child pointer(page_id) which points to the child page
 * that contains input "key"
 * Start the search from the second key(the first key should always be invalid)
 */
INDEX_TEMPLATE_ARGUMENTS
ValueType B_PLUS_TREE_INTERNAL_PAGE_TYPE::Lookup(const KeyType &key, const KeyComparator &comparator) const {
  auto k_it = std::lower_bound(array + 1, array + GetSize(), key,
                               [&comparator](const auto &pair, auto k) { return comparator(pair.first, k) < 0; });

  if (k_it == array + GetSize()) {
    return ValueAt(GetSize() - 1);
  }

  if (comparator(k_it->first, key) == 0) {
    return k_it->second;
  }

  return std::prev(k_it)->second;
}
```

Finding the first element in the internal node's array whose search-key value is equal to or greater than the given key.

If not found, return the last pointer
If found the element whose search-key value is equal to the given key, return the pointer
If found the element whose search-key value is greater than the given key, return the previous element's pointer (refer to the comment in b_plus_tree_internal_page.h for the description of the internal page format.

B+Tree Internal Page

b_plus_tree_internal.cpp

```
/*****************************************************************************
 * INSERTION
 *****************************************************************************/
/*
 * Populate new root page with old_value + new_key & new_value
 * When the insertion cause overflow from leaf page all the way upto the root
 * page, you should create a new root page and populate its elements.
 * NOTE: This method is only called within InsertIntoParent()(b_plus_tree.cpp)
 */
INDEX_TEMPLATE_ARGUMENTS
void B_PLUS_TREE_INTERNAL_PAGE_TYPE::PopulateNewRoot(const ValueType &old_value, const KeyType &new_key,
                                                     const ValueType &new_value) {
  array[0].second = old_value;
  array[1].first = new_key;
  array[1].second = new_value;
  SetSize(2);
}
/*
 * Insert new_key & new_value pair right after the pair with its value ==
 * old_value
 * @return:  new size after insertion
 */
INDEX_TEMPLATE_ARGUMENTS
int B_PLUS_TREE_INTERNAL_PAGE_TYPE::InsertNodeAfter(const ValueType &old_value, const KeyType &new_key,
                                                    const ValueType &new_value) {
  auto new_value_idx = ValueIndex(old_value) + 1;
  std::move_backward(array + new_value_idx, array + GetSize(), array + GetSize() + 1);

  array[new_value_idx].first = new_key;
  array[new_value_idx].second = new_value;

  IncreaseSize(1);

  return GetSize();
}
```

inserting a new element (key and value pair) at a specific location which is right next to the element containing given old_value.
Note: the page is ensured not to be full, as internal page will be splitted afterwards as soon as it gets full (refer to InsertIntoParent method in b_plus_tree.cpp)

B+Tree Internal Page

b_plus_tree_internal.cpp

```
/*****************************************************************************
 * SPLIT
 *****************************************************************************/
/*
 * Remove half of key & value pairs from this page to "recipient" page
 * Since it is an internal page, for all entries (pages) moved, their parents page now changes to `recipient`.
 * So `recipient` need to 'adopt' them by changing their parent page id, which needs to be persisted with
 * BufferPoolManger
 */
INDEX_TEMPLATE_ARGUMENTS
void B_PLUS_TREE_INTERNAL_PAGE_TYPE::MoveHalfTo(BPlusTreeInternalPage *recipient,
                                                BufferPoolManager *buffer_pool_manager) {
  auto start_idx = GetMinSize();
  SetSize(start_idx);

  recipient->CopyNFrom(array + start_idx, GetMaxSize() - start_idx, buffer_pool_manager);
}

/* Copy entries into me, starting from {items} and copy {size} entries.
 * Since it is an internal page, for all entries (pages) moved, their parents page now changes to me.
 * So I need to 'adopt' them by changing their parent page id, which needs to be persisted with BufferPoolManger
 */
INDEX_TEMPLATE_ARGUMENTS
void B_PLUS_TREE_INTERNAL_PAGE_TYPE::CopyNFrom(MappingType *items, int size, BufferPoolManager *buffer_pool_manager) {
  std::copy(items, items + size, array + GetSize());

  for (int i = 0; i < size; i++) {
    auto page = buffer_pool_manager->FetchPage(ValueAt(i + GetSize()));
    BPlusTreePage *node = reinterpret_cast<BPlusTreePage *>(page->GetData());
    node->SetParentPageId(GetPageId());
    buffer_pool_manager->UnpinPage(page->GetPageId(), true);
  }

  IncreaseSize(size);
}
```

Copy the half of elements (key and value pairs). Then, update the header (i.e. parent_page_id_ in b_plus_tree_page.h) of moved child pages to point the new parent (i.e. 'recipient' page). Refer to Split method in b_plus_tree.cpp.

B+Tree Internal Page

b_plus_tree_internal.cpp

```
/*****************************************************************************
 * REMOVE
 *****************************************************************************/
/*
 * Remove the key & value pair in internal page according to input index(a.k.a
 * array offset)
 * NOTE: store key&value pair continuously after deletion
 */
INDEX_TEMPLATE_ARGUMENTS
void B_PLUS_TREE_INTERNAL_PAGE_TYPE::Remove(int index) {
  std::move(array + index + 1, array + GetSize(), array + index);
  IncreaseSize(-1);
}

/*
 * Remove the only key & value pair in internal page and return the value
 * NOTE: only call this method within AdjustRoot()(in b_plus_tree.cpp)
 */
INDEX_TEMPLATE_ARGUMENTS
ValueType B_PLUS_TREE_INTERNAL_PAGE_TYPE::RemoveAndReturnOnlyChild() {
  SetSize(0);
  return ValueAt(0);
}
```

move all elements next to the deleted element by one position left

B+Tree Internal Page

b_plus_tree_internal.cpp

```
/*****************************************************************************
 * MERGE
 *****************************************************************************/
/*
 * Remove all of key & value pairs from this page to "recipient" page.
 * The middle_key is the separation key you should get from the parent. You need
 * to make sure the middle key is added to the recipient to maintain the invariant.
 * You also need to use BufferPoolManager to persist changes to the parent page id for those
 * pages that are moved to the recipient
 */
INDEX_TEMPLATE_ARGUMENTS
void B_PLUS_TREE_INTERNAL_PAGE_TYPE::MoveAllTo(BPlusTreeInternalPage *recipient, const KeyType &middle_key,
                                               BufferPoolManager *buffer_pool_manager) {
  SetKeyAt(0, middle_key);
  recipient->CopyNFrom(array, GetSize(), buffer_pool_manager);
  SetSize(0);
}
```

First set the middle_key given at the very first element's key.
Then, copy all elements in this page to the 'recipient' page including the middle_key

B+Tree Internal Page

b_plus_tree_internal.cpp

```
/*****************************************************************************
 * REDISTRIBUTE
 *****************************************************************************/
/*
 * Remove the first key & value pair from this page to tail of "recipient" page.
 *
 * The middle_key is the separation key you should get from the parent. You need
 * to make sure the middle key is added to the recipient to maintain the invariant.
 * You also need to use BufferPoolManager to persist changes to the parent page id for those
 * pages that are moved to the recipient
 */
INDEX_TEMPLATE_ARGUMENTS
void B_PLUS_TREE_INTERNAL_PAGE_TYPE::MoveFirstToEndOf(BPlusTreeInternalPage *recipient, const KeyType &middle_key,
                                                      BufferPoolManager *buffer_pool_manager) {
  SetKeyAt(0, middle_key);
  auto first_item = array[0];
  recipient->CopyLastFrom(first_item, buffer_pool_manager);

  std::move(array + 1, array + GetSize(), array);
  IncreaseSize(-1);
}

/* Append an entry at the end.
 * Since it is an internal page, the moved entry(page)'s parent needs to be updated.
 * So I need to 'adopt' it by changing its parent page id, which needs to be persisted with BufferPoolManger
 */
INDEX_TEMPLATE_ARGUMENTS
void B_PLUS_TREE_INTERNAL_PAGE_TYPE::CopyLastFrom(const MappingType &pair, BufferPoolManager *buffer_pool_manager) {
  *(array + GetSize()) = pair;
  IncreaseSize(1);

  auto page = buffer_pool_manager->FetchPage(pair.second);
  BPlusTreePage *node = reinterpret_cast<BPlusTreePage *>(page->GetData());
  node->SetParentPageId(GetPageId());
  buffer_pool_manager->UnpinPage(page->GetPageId(), true);
}
```

MoveFirstToEndOf and MoveLastToFrontOf in the next slide are used to move around elements for redistribution to reduce the unnecessary nodes. Refer to b_plus_tree.cpp for how they are used.

B+Tree Internal Page

b_plus_tree_internal.cpp

```cpp
/*
 * Remove the last key & value pair from this page to head of "recipient" page.
 * You need to handle the original dummy key properly, e.g. updating recipient's array to position the middle_key at the
 * right place.
 * You also need to use BufferPoolManager to persist changes to the parent page id for those pages that are
 * moved to the recipient
 */
INDEX_TEMPLATE_ARGUMENTS
void B_PLUS_TREE_INTERNAL_PAGE_TYPE::MoveLastToFrontOf(BPlusTreeInternalPage *recipient, const KeyType &middle_key,
                                                       BufferPoolManager *buffer_pool_manager) {
  auto last_item = array[GetSize() - 1];
  recipient->SetKeyAt(0, middle_key);
  recipient->CopyFirstFrom(last_item, buffer_pool_manager);

  IncreaseSize(-1);
}

/* Append an entry at the beginning.
 * Since it is an internal page, the moved entry(page)'s parent needs to be updated.
 * So I need to 'adopt' it by changing its parent page id, which needs to be persisted with BufferPoolManger
 */
INDEX_TEMPLATE_ARGUMENTS
void B_PLUS_TREE_INTERNAL_PAGE_TYPE::CopyFirstFrom(const MappingType &pair, BufferPoolManager *buffer_pool_manager) {
  std::move_backward(array, array + GetSize(), array + GetSize() + 1);
  *array = pair;
  IncreaseSize(1);

  auto page = buffer_pool_manager->FetchPage(pair.second);
  BPlusTreePage *node = reinterpret_cast<BPlusTreePage *>(page->GetData());
  node->SetParentPageId(GetPageId());
  buffer_pool_manager->UnpinPage(page->GetPageId(), true);
}

// valuetype for internalNode should be page id_t
template class BPlusTreeInternalPage<GenericKey<4>, page_id_t, GenericComparator<4>>;
template class BPlusTreeInternalPage<GenericKey<8>, page_id_t, GenericComparator<8>>;
template class BPlusTreeInternalPage<GenericKey<16>, page_id_t, GenericComparator<16>>;
template class BPlusTreeInternalPage<GenericKey<32>, page_id_t, GenericComparator<32>>;
template class BPlusTreeInternalPage<GenericKey<64>, page_id_t, GenericComparator<64>>;
}  // namespace bustub
```

Outline

- B+Tree Parent Page
- B+Tree Internal Page
- Assignment

Assignment: B+ Tree Leaf Page

B+Tree Leaf Page

B+Tree Leaf Page is in src/include/storage/page/b_plus_tree_leaf_page.h and src/storage/page/b_plus_tree_leaf_page.cpp

The Leaf Page stores an ordered **m** key entries and **m** value entries. In your implementation, value should only be a 64-bit record_id that is used to locate where actual tuples are stored, see RID class defined under in src/include/common/rid.h. Leaf pages have the same restriction on the number of key/value pairs as Internal pages, and should follow the same operations of merge, redistribute and split.

You must implement your Leaf Page in the designated files. You are only allowed to modify the header file (src/include/storage/page/b_plus_tree_leaf_page.h) and its corresponding source file (src/storage/page/b_plus_tree_leaf_page.cpp).

Important: Even though the Leaf Pages and Internal Pages contain the same type of key, they may have distinct types of value, thus the max_size of leaf and internal pages could be different.

Each B+Tree leaf/internal page corresponds to the content (i.e., the data_ part) of a memory page fetched by buffer pool. So every time you try to read or write a leaf/internal page, you need to first **fetch** the page from the buffer pool using its unique page_id, then reinterpret cast to either a leaf or an internal page, and unpin the page after any writing or reading operations.

The End