#3 INDEX MANAGER

#3.1 实验概述

Index Manager 负责数据表索引的实现和管理，包括：索引的创建和删除，索引键的等值查找，索引键的范围查找（返回对应的迭代器），以及插入和删除键值等操作，并对外提供相应的接口。

在上一个实验中，同学们应该能够发现，通过遍历堆表的方式来查找一条记录是十分低效的。为了能够快速定位到某条记录而无需搜索数据表中的每一条记录，我们需要在上一个实验的基础上实现一个索引，这能够为快速随机查找和高效访问有序记录提供基础。索引有很多种实现方式，如B+树索引，Hash索引等等。在本实验中，需要同学们实现一个基于磁盘的B+树动态索引结构。

Bonus: 实现除B+树索引外的另一种索引（如Hash索引），并在最终的验收中能够让你实现的MiniSQL支持选择这种索引。

#3.2 B+树数据页

B+树中的每个结点（Node）都对应一个数据页，用于存储B+树结点中的数据。因此在本节中，你需要实现以下三种类型的B+树结点数据页：

#3.2.1 BPlusTreePage

`BPlusTreePage` 是 `BPlusTreeInternalPage` 和 `BPlusTreeLeafPage` 类的公共父类，它包含了中间结点和叶子结点共同需要的数据：

- `page_type_`：标记数据页是中间结点还是叶子结点；
- `lsn_`：数据页的日志序列号，目前不会用到，如果之后感兴趣做Crash Recovery相关的内容需要用到；
- `size_`：当前结点中存储Key-Value键值对的数量；
- `max_size_`：当前结点最多能够容纳Key-Value键值对的数量；
- `parent_page_id_`：父结点对应数据页的 `page_id`；
- `page_id_`：当前结点对应数据页的 `page_id`。

你需要在 `src/include/storage/page/b_plus_tree_page.h` 和 `src/page/b_plus_tree_page.cpp` 中实现 `BPlusTreePage` 类。

#3.2.2 BPlusTreeInternalPage

中间结点 `BPlusTreeInternalPage` 不存储实际的数据，它只按照顺序存储 m 个键和 $m+1$ 个指针（这些指针记录的是子结点的 `page_id`）。由于键和指针的数量不相等，因此我们需要将第一个键设置为INVALID，也就是说，顺序查找时需要从第二个键开始查找。在任何时候，每个中间结点至少是半满的（Half Full）。当删除操作导致某个结点不满足半满的条件，需要通过合并（Merge）相邻两个结点或是从另一个结点中借用（移动）一个元素到该结点中（Redistribute）来使该结点满足半满的条件。当插入操作导致某个结点溢出时，需要将这个结点分裂成为两个结点。

你需要在 `src/include/storage/page/b_plus_tree_internal_page.h` 和 `src/page/b_plus_tree_internal_page.cpp` 中实现 `BPlusTreeInternalPage` 类。

Note: 为了便于理解和设计，我们将键和指针以 `pair` 的形式顺序存储，但由于键和指针的数量不一致，我们不得已牺牲一个键的空间，将其标记为INVALID。也就是说对于B+树的每一个中间结点，我们都付出了一个键的空间代价。实际上有一种更为精细的设计选择：定义一个大小为 m 的数组连续存放键，然后定义一个大小为 $m+1$ 的数组连续存放指针，这样设计的好处在于，一是没有空间上的浪费，二是在键值查找时CPU缓存的命中率较高（局部性原理）。学有余力的同学可以尝试着使用这种方式去实现。

#3.2.3 BPlusTreeLeafPage

叶结点 `BPlusTreeLeafPage` 存储实际的数据，它按照顺序存储 m 个键和 m 个值，其中键由一个或多个 `Field` 序列化得到（参考#3.2.4），在 `BPlusTreeLeafPage` 类中用模板参数 `KeyType` 表示；值实际上存储的是 `RowId` 的值，它在 `BPlusTreeLeafPage` 类中用模板参数 `ValueType` 表示。叶结点和中间结点一样遵循着键值对数量的约束，同样也需要完成对应的合并、借用和分裂操作。

你需要在 `src/include/storage/page/b_plus_tree_leaf_page.h` 和 `src/page/b_plus_tree_leaf_page.cpp` 中实现 `BPlusTreeLeafPage` 类。

#3.2.4 KeyType、ValueType & KeyComparator

在B+树的数据页以及索引中，考虑到索引键类型可能会不同（对不同长度的索引键使用不同的索引键类型，如为最大长度不超过32字节的索引键使用 `GenericKey<32>`（在 `src/include/index/generic_key.h` 中定义），为最大长度不超过64字节的索引键使用 `GenericKey<64>` 等等）、值类型也可能不同（叶结点存储 `RowId`，而非叶结点存储 `page_id`）、对应的比较方式也有可能不同（如对 `GenericKey<32>` 使用 `GenericComparator<32>` 进行比较），因此我们使用模板对 `BPlusTreeLeafPage`、`BPlusTreeLeafPage` 等类进行定义。

C++ 复制代码

```cpp
template <typename KeyType, typename ValueType, typename KeyComparator>
class BPlusTreeLeafPage : public BPlusTreePage {
    /* b plus tree leaf page implement */
};

template <typename KeyType, typename ValueType, typename KeyComparator>
class BPlusTreeInternalPage : public BPlusTreePage {
    /* b plus tree internal page implement */
};
```

对于B+树中涉及到的索引键的比较，由于模板参数 `KeyType` 实际传入的对象并不是基本数据类型，因此不能够直接使用比较运算符 `>`、`<` 等进行比较（除非对传入的对象的比较运算符进行重载，但这种设计方式难以应对需要不同比较方式的场景）。为此，我们需要借助传入的比较器 `KeyComparator` 对两个索引键进行比较。由于B+树在构建时，会传入一个 `KeyComparator` 类型的比较器 `comparator`，在编码时只需要调用 `comparator()` 即可对两个对象进行比较，以下是一个例子：

```cpp
void Example(KeyType &k1, KeyType &k2) {
    if (comparator(k1, k2) > 0) {
        // k1 > k2
    } else if (comparator(k1, k2) < 0) {
        // k1 < k2
    } else {
        // k1 == k2
    }
}
```

比较器的实现在框架中已经给出（在 `src/include/index/generic_key.h` 中定义），其基本原理是，对于两个待比较的索引键 `GenericKey` （为了将索引键存储到B+树数据页中，需要将索引键进行序列化，也就是说 `GenericKey` 内部实际上存储的是索引键序列化后得到的字符串，参考下面代码中 `GenericKey` 类的定义），首先将其按照索引键定义的模式 `key_schema_` 进行反序列化，然后对反序列化得到的每一个域 `Field` ，调用 `Field` 的比较函数进行比较。 `Field` 类型的比较函数已经在代码框架中给出，具体细节请同学们自行学习了解。

```cpp
template<size_t KeySize>
class GenericKey {
public:
    inline void SerializeFromKey(const Row &key, Schema *schema);
    inline void DeserializeToKey(Row &key, Schema *schema) const;
    // actual location of data, extends past the end.
    char data[KeySize];
}

template<size_t KeySize>
inline int GenericComparator::operator()(
    const GenericKey<KeySize> &lhs,
    const GenericKey<KeySize> &rhs) const {
    int column_count = key_schema_->GetColumnCount();
    Row lhs_key(INVALID_ROWID);
    Row rhs_key(INVALID_ROWID);
    lhs.DeserializeToKey(lhs_key, key_schema_);
    rhs.DeserializeToKey(rhs_key, key_schema_);
    for (int i = 0; i < column_count; i++) {
      Field *lhs_value = lhs_key.GetField(i);
      Field *rhs_value = rhs_key.GetField(i);
      if (lhs_value->CompareLessThan(*rhs_value) == CmpBool::kTrue)
        return -1;

      if (lhs_value->CompareGreaterThan(*rhs_value) == CmpBool::kTrue)
        return 1;
    }
    return 0;
}
```

#3.2.5 Some Tips

- `BPlusTreePage::GetMinSize()` 所返回的值通常情况下为 `max_size_/2` ，但它实际上对于叶子结点/非叶结点/根结点/非根结点可能会有所不同。且 `size` 的概念通常情况下表示的是指针的数量（即结点中键值对的数量），换而言之，在中间结点中，包含 $k-1$ 个键和 k 个指针的 `size` 为 k 。
- `BPlusTreePage` 中的内容实际上存储于 `Page` 中的 `data_` ，每当需要对B+树的数据页进行读写时，首先需要从 `BufferPoolManager` 中获取（ `Fetch` ）这个页，此时拿到的数据页为 `Page` 类型，但我们需要用到的数据页 `BPlusTreeInternalPage` 和 `BPlusTreeLeafPage` 是 `BPlusTreePage` 类的子类， `BPlusTreePage` 类和 `Page` 类的 `data_` 域在内存分布上是相同的（通俗来说， `data_` 域中 `PAGE_SIZE` 个字节存放的就是 `BPlusTreePage` 对象），因此需要通过 `reinterpret_cast` 将 `Page` 中的 `data_` 重新解释成为我们需要使用的类。最后，在使用完毕后需要将该页释放（ `Unpin` ），以下是一个使用 `reinterpret_cast` 将 `Page` 类的 `data_` 域重新解释成 `BPlusTreeInternalPage` 对象例子：

```cpp
auto *page = buffer_pool_manager->FetchPage(page_id);
if (page != nullptr) {
    auto *node = reinterpret_cast<BPlusTreeInternalPage *>(page->GetData());
    /* do something */
    buffer_pool_manager->UnpinPage(page_id, true);
}
```

- 在不需要使用数据页时，请务必将其释放，我们将会在测试代码中加入 `CheckAllUnpinned()` 机制检查所有的数据页最终是否被释放。

#3.3 B+树索引

在完成B+树结点的数据结构设计后，接下来需要完成B+树的创建、插入、删除、查找和释放等操作。注意，所设计的B+树只能支持 `Unique Key` ，这也意味着，当尝试向B+树插入一个重复的Key-Value键值对时，将不能执行插入操作并返回 `false` 状态。当一些写操作导致B+树索引的根结点发生变化时，需要调用 `BPLUSTREE_TYPE::UpdateRootPageId` 完成 `root_page_id` 的变更和持久化。

你需要在 `src/include/index/b_plus_tree.h` 和 `src/index/b_plus_tree.cpp` 中实现整个 `BPlusTree` 类。

在实现 `BPlusTree` 时，你无需考虑 `KeyType` 、 `ValueType` 、 `KeyComparator` 的实现：

C++ | 复制代码

```cpp
template <typename KeyType, typename ValueType, typename KeyComparator>
class BPlusTree{
   /* b plus tree implement */
};
```

与 `KeyType` 、 `ValueType` 、 `KeyComparator` 相关的类已经实现，其中 `KeyType` 和 `KeyComparator` 位于 `src/include/index/generic_key.h` 中的 `GenericKey` 和 `GenericComparator` ，而 `ValueType` 则是一个 `32` 或 `64` 位的整数，它在中间结点中表示子结点的 `page_id` ，在叶子结点中表示对应记录的 `RowId` 。这些类的实例将会在构造 `BPlusTreeIndex` 时传入。

#3.4 B+树索引迭代器

与堆表 `TableHeap` 对应的迭代器类似，在本节中，你需要为B+树索引也实现一个迭代器。该迭代器能够将所有的叶结点组织成为一个单向链表，然后沿着特定方向有序遍历叶结点数据页中的每个键值对（这在范围查询时将会被用到）。

你需要在 `src/include/index/index_iterator.h` 和 `src/index/index_iterator.cpp` 中实现B+树索引的迭代器 `IndexIterator` 。同样地，你需要在 `BPlusTree` 类中实现 `Begin()` 和 `End()` 函数以获取B+树索引的首迭代器和尾迭代器。

#3.5 模块相关代码

- `src/include/storage/page/b_plus_tree_page.h`
- `src/page/b_plus_tree_page.cpp`
- `src/include/storage/page/b_plus_tree_internal_page.h`
- `src/storage/page/b_plus_tree_internal_page.cpp`
- `src/include/storage/page/b_plus_tree_leaf_page.h`
- `src/storage/page/b_plus_tree_leaf_page.cpp`
- `src/include/storage/index/b_plus_tree.h`
- `src/storage/index/b_plus_tree.cpp`
- `src/include/storage/index/index_iterator.h`
- `src/storage/index/index_iterator.cpp`
- `test/index/b_plus_tree_index_test.cpp`
- `test/index/b_plus_tree_test.cpp`
- `test/index/index_iterator_test.cpp`

#3.6 开发提示

1. 推荐在**夏学期第4周前**完成本模块的设计。
2. 这是一个展现B+树插入和删除操作的可视化网站，可以帮助熟悉B+树的相关操作：链接 <https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html>
3. 在调试时，可以通过 `BPlusTree::PrintTree(std::ofstream &out)` 将B+树的结构以DOT格式输出到输出流中，然后可以通过一个可视化网站：链接 <http://dreampuf.github.io/GraphvizOnline/> ，查看当前B+树的状态。具体的使用方法可以参考测试模块中给出的代码。

#3.7 诚信守则

1. 请勿从其它组或在网络上找到的其它来源中复制源代码，一经发现抄袭，成绩为 `0`；
2. 请勿将代码发布到公共Github存储库上。