PROJECT #2 - B+TREE

OVERVIEW

The second programming project is to implement an index in your database system. The index is responsible for fast data retrieval without having to search through every row in a database table, providing the basis for both rapid random lookups and efficient access of ordered records.
You will need to implement B+Tree dynamic index structure. It is a balanced tree in which the internal pages direct the search and leaf pages contains actual data entries. Since the tree structure grows and shrink dynamically, you are required to handle the logic
of split and merge. The project is comprised of the following three tasks:
Task #1 - B+Tree Pages
Task #2 - B+Tree Data Structure
Task #3 - Index Iterator
This is a single-person project that will be completed individually (i.e., no groups).

PROJECT SPECIFICATION

Like the first project, we are providing you with stub classes that contain the API that you need to implement. You should not modify the signatures for the pre-defined functions in these classes. If you do this, then it will break the test code that we will use to grade your assignment you end up getting no credit for the project. If a class already contains certain member variables, you should notremove them. But you may add private helper functions/member variables to these classes in order to correctly realize the functionality.

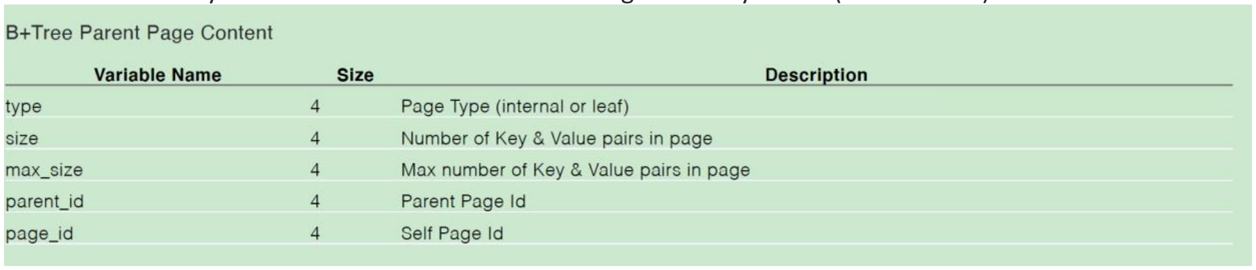
The correctness of B+Tree index depends on the correctness of your implementation of buffer pool, we will not provide solutions for the previous programming projects. Since the second project is closely related to the third project in which you will implement index crabbing within existing B+ index, we have passed in a pointer parameter called transaction with default value nullptr . You can safely ignore the parameter for now; you don't need to change or call any functions relate to the parameter.

TASK #1 - B+TREE PAGES

You need to implement three Page classes to store the data of your B+Tree tree.
B+Tree Parent Page
B+Tree Internal Page
B+Tree Leaf Page

B+TREE PARENT PAGE

This is the parent class that both the Internal Page and Leaf Page inherited from and it only contains information that both child classes share. The Parent Page is divided into several fields as shown by the table below. All multi-byte fields are stored with the most significant byte first (little-endian).


B+Tree Parent Page Content

Variable Name	Size	Description
type	4	Page Type (internal or leaf)
size	4	Number of Key & Value pairs in page
max_size	4	Max number of Key & Value pairs in page
parent_id	4	Parent Page Id
page_id	4	Self Page Id

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

B+TREE INTERNAL PAGE

Internal Page does not store any real data, but instead it stores an ordered m key entries and m+1 child pointers (a.k.a page_id). Since the number of pointer does not equal to the number of key, the first key is set to be invalid, and lookup methods should always start with the second key. At any time, each internal page is at least half full. During deletion, two half-full pages can be joined to make a legal one or can be redistributed to avoid merging, while during insertion one full page can be split into two.

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

B+TREE LEAF PAGE

The Leaf Page stores an ordered m key entries and m value entries. In your implementation, value should only be 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 Internal Page in the designated files. You are only allowed to modify the header file (src/include/page/b_plus_tree_leaf_page.h) and its corresponding source file (src/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 type of value, thus the max_size of leaf and internal pages could be different. You should calculate the max_size within both Internal Page's and Leaf Page's Init() method.

Each B+Tree leaf/internal page corresponds to the content (i.e., the byte array data_) 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 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.

TASK #2 - B+TREE DATA STRUCTURE

Your B+Tree Index could only support unique key. That is to say, when you try to insert a key-value pair with duplicate key into the index, it should not perform the insertion and return false.

Your B+Tree Index is required to correctly perform split if insertion triggers current number of key/value pairs exceeds max_size . It also needs to correctly perform merge or redistribute if deletion cause certain page to go below the occupancy threshold. Since any write operation could lead to the change of root_page_id in B+Tree index, it is your responsibility to update root_page_id in the header page (src/include/page/header_page.h) to ensure that the index is durable on disk. Within the BPlusTree class, we have already implemented a function called UpdateRootPageId() for you; all you need to do is invoke this function whenever the root_page_id of B+Tree index changes.

Your B+Tree implementation must hide the details of the key/value type and associated comparator, like this:

```
template <typename KeyType,
typename ValueType,
typename KeyComparator>
class BPlusTree{
// ---
};
```

These classes are already implemented for you:

- KeyType : The type of each key in the index. This will only be GenericKey , the actual size of GenericKey is specified and instantiated with a template argument and depends on the data type of indexed attribute.
- ValueType : The type of each value in the index. This will only be 64-bit RID.
- KeyComparator : The class used to compare whether two KeyType instances are less/greater-than each other. These will be included in the KeyType implementation files.

TASK #3 - INDEX ITERATOR

You will build a general purpose index iterator to retrieve all the leaf pages efficiently. The basic idea is to organize them into a single linked list, and then traverse every key/value pairs in specific direction stored within the B+Tree leaf pages. Your index iterator should follow the functionality of Iterator defined in c++11, including the ability to iterate through a range of elements using a set of operators (with at least the increment and dereference operators).

You must implement your index iterator in the designated files. You are only allowed to modify the header file (src/include/index/index_iterator.h) and its corresponding source file (src/index/index_iterator.cpp). You do not need to modify any other files. You must implement the following functions in the IndexIterator class found in these files. In the implementation of index iterator, you are allowed to add any helper methods as long as you have those three methods below.
isEnd() : Return whether this iterator is pointing at the last key/value pair.
operator++() : Move to the next key/value pair.
operator*() : Return the key/value pair this iterator is currently pointing at.

TESTING

You can test the individual components of this assigment using our testing framework. We use GTest for unit test cases. You can
compile and run each test individually from the command-line:

```
cd build
make b_plus_tree_test
./test/b_plus_tree_test
```

In the b_plus_tree_print_test, you can print out the internal data structure of your b+ tree index, it's an intuitive way to track and find early mistakes.

```
cd build
make b_plus_tree_print_test
./test/b_plus_tree_print_test
```

Important: These tests are only a subset of the all the tests that we will use to evaluate and grade your project. You should write
additional test cases on your own to check the complete functionality of your implementation.