PROJECT #2 - B+TREE

Do not post your project on a public Github repository.

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 tasks and it has two checkpoints.

Checkpoint #1 — Due Date: Oct 11 @ 11:59pm

- Task #1 - B+Tree Pages
- Task #2.a - B+Tree Data Structure (Insertion & Point Search)

Checkpoint #2 — Due Date: Oct 25 @ 11:59pm

- Task #2.b - B+Tree Data Structure (Deletion)
- Task #3 - Index Iterator
- Task #4 - Concurrent Index

This is a single-person project that will be completed individually (i.e., no groups).

Release Date: Sep 28, 2020
Due Date: Nov 01, 2020 @ 11:59pm

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 **not** remove 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 first checkpoint is closely related to the second checkpoint 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 Checkpoint #1; you do not need to change or call any functions relate to the parameter until Task #4.

CHECKPOINT #1

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.

B+Tree Parent Page Content

Variable Name	Size	Description
page_type_	4	Page Type (internal or leaf)
lsn_	4	Log sequence number (Used in Project 4)
size_	4	Number of Key & Value pairs in page
max_size_	4	Max number of Key & Value pairs in page
parent_page_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/storage/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/storage/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/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 type 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 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.A - B+TREE DATA STRUCTURE (INSERTION & POINT SEARCH)

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.

of key/value pairs after insertion equals to `max_size`. 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/storage/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.

CHECKPOINT #2

TASK #2.B - B+TREE DATA STRUCTURE (DELETION)

Your B+Tree Index is required to support deletions. Your B+Tree Index should correctly perform merge or redistribute if deletion cause certain page to go below the occupancy threshold. Again, your B+Tree Index could only support unique key and you should follow the same guidelines in Task #2.a .

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++17, including the ability to iterate through a range of elements using a set of operators, and for-each loop (with at least the increment, dereference, equal and not-equal operators). Note that in order to support for-each loop function for your index, your `BPlusTree` should correctly implements `begin()` and `end()`.

You must implement your index iterator in the designated files. You are only allowed to modify the header file (`src/include/storage/index/index_iterator.h`) and its corresponding source file (`src/index/storage/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.
- `operator==()`: Return whether two iterators are equal
- `operator!=()`: Return whether two iterators are not equal.

TASK #4 - CONCURRENT INDEX

In this part, you need to update your original single-threaded B+Tree index so that it can support concurrent operations. We will use the latch crabbing technique described in class and in the textbook. The thread traversing the index will acquire then release latches on B+Tree pages. A thread can **only** release latch on a parent page if its child page considered "safe". Note that the definition of "safe" can vary based on what kind of operation the thread is executing:

- `Insert` : Starting with root page, grab write (**W**) latch on child. Once child is locked, check if it is safe, in this case, not full. If child is safe, release **all** locks on ancestors.
- `Delete` : Starting with root page, grab write (**W**) latch on child. Once child is locked, check if it is safe, in this case, at least half-full. (NOTE: for root page, we need to check with different standards) If child is safe, release **all** locks on ancestors.

Important: The write up only describe the basic concepts behind latch crabbing, before you start your implementation, please consult with lecture and textbook Chapter 15.10.

REQUIREMENTS AND HINTS

- You are not allowed to use a global scope latch to protect your data structure. In other words, you may not lock the whole index and only unlock the latch when operations are done. We will check grammatically and manually to make sure you are doing the latch crabbing in the right way.
- We have provided the implementation of read-write latch (`src/include/common/rwlatch.h`). And have already added helper functions under page header file to acquire and release latch (`src/include/storage/page/page.h`).
- We will not add any **mandatory** interfaces in the B+Tree index. You can write any private/helper functions in your implementation as long as you keep all the original public interfaces intact for test purpose.
- For this task, you have to use the passed in pointer parameter called `transaction` (`src/include/concurrency/transaction.h`). It provides methods to store the page on which you have acquired latch while traversing through B+ tree and also methods to store the page which you have deleted during `Remove` operation. Our suggestion is to look closely at the `FindLeafPage` method within B+ tree, you may wanna modify your previous implementation (note that you may need to change to **return value** for this method) and then add the logic of latch crabbing within this particular method.
- The return value for FetchPage() in buffer pool manager is a pointer that points to a Page instance (`src/include/storage/page/page.h`). You can grab a latch on `Page`, but you cannot grab a latch on B+Tree node (neither internal node nor leaf node).

COMMON PITFALLS

- You are NOT tested for thread-safe scans in this project (no concurrent iterator operations will be tested). However, a correct implementation would requires the Leaf Page to throw an `std::exception` when it cannot acquire a latch on its sibling to avoid potential dead-locks.
- Think carefully about the order and relationship between `UnpinPage(page_id, is_dirty)` method from buffer pool manager class and `UnLock()` methods from page class. You have to release the latch on that page **BEFORE** you unpin the same page from the buffer pool.
- If you are implementing concurrent B+tree index correctly, then every thread will **ALWAYS** acquire latch from root to bottom. When you release the latch, make sure you follow the same order (a.k.a from root to bottom) to avoid possible deadlock situation.
- One of the corner case is that when insert and delete, the member variable **root_page_id** (`src/include/storage/index/b_plus_tree.h`) will also be updated. It is your responsibility to protect from concurrent update of this shared variable(hint: add an abstract layer in B+ tree index, you can use `std::mutex` to protect this variable)

INSTRUCTIONS

SETTING UP YOUR DEVELOPMENT ENVIRONMENT

See the Project #0 instructions on how to create your private repository and setup your development environment.

⚠ You must pull the latest changes on our BusTub repo for test files and other supplementary files we have provided for you. Run `git pull`.

CHECKPOINT 1

Update your project source code like you did in Project 1

? Be sure to include your Project 1 code into the submission.

- `src/include/buffer/lru_replacer.h`
- `src/buffer/lru_replacer.cpp`
- `src/include/buffer/buffer_pool_manager.h`
- `src/buffer/buffer_pool_manager.cpp`

Make sure you are getting the correct splitting behaviour by checking your solution with `b_plus_tree_print_test`

You are not required to pass the below tests for Checkpoint #1:

- `test/index/b_plus_tree_insert_test.cpp`
- `test/index/b_plus_tree_delete_test.cpp`
- `test/index/b_plus_tree_concurrent_test.cpp`

CHECKPOINT 2

You will need to pass the below tests for Checkpoint #2:

- `test/index/b_plus_tree_insert_test.cpp`
- `test/index/b_plus_tree_delete_test.cpp`
- `test/index/b_plus_tree_concurrent_test.cpp`

TESTING

You can test the individual components of this assigment using our testing framework. We use GTest for unit test cases. In the `test/index/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
```

Remember to **ENABLE** the test when running the print test.

Remember to **DISABLE** the test when running other tests with `make check-tests`.

TREE VISUALIZATION

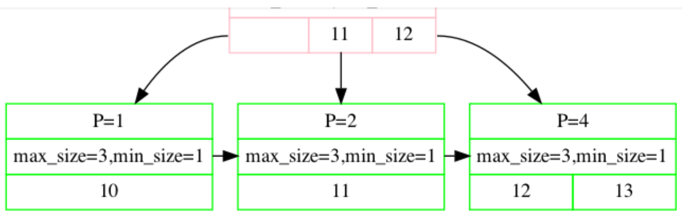
Use the `b_plus_tree_print_test` to generate a dot file after constructing a tree:

```
$ ./b_plus_tree_print_test
>> ... USAGE ...
>> 5 5 // set leaf node and internal node max size to be 5
>> f input.txt // Insert into the tree with some inserts
>> g my-tree.dot // output the tree to dot format
>> q // Quit the test (Or use another terminal)
```

You should now have a `my-tree.dot` file with the DOT file format in the same directory as your test binary, you can then visualize it through either a command line visiualizer or an online visualizer:

1. Dump the content to http://dreampuf.github.io/GraphvizOnline/.
2. Or download a command line tool at https://graphviz.org/download/ on your platform.
3. Draw the tree with `dot -Tpng -O my-tree.dot`, a PNG file will be generated.

Consider the following example generated with GraphvizOnline:



- The first row `P=?` tells you the page id of the tree page.
- Pink is an internal node, Green is leaf node.
- The second row prints out the properties.
- The third row prints out keys, and pointers from internal node to the leaf node is constructed from internal node's value.
- Note that the first box of the internal node is empty. This is not a bug.

MEMORY SAFETY

To ensure your implementation does not have memory leak, you can run the test with Valgrind.

```
cd build
make b_plus_tree_insert_test
valgrind --trace-children=yes \
   --leak-check=full \
   --track-origins=yes \
   --soname-synonyms=somalloc=*jemalloc* \
   --error-exitcode=1 \
   --suppressions=../build_support/valgrind.supp \
   ./test/b_plus_tree_insert_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.

DEVELOPMENT HINTS

Instead of using `printf` statements for debugging, use the `LOG_*` macros for logging information like this:

```
LOG_DEBUG("Fetching page %d", page_id);
```

To enable logging in your project, you will need to reconfigure it like this:

```
cd build
cmake -DCMAKE_BUILD_TYPE=DEBUG ..
make
```

The different logging levels are defined in `src/include/common/logger.h`. After enabling logging, the logging level defaults to `LOG_LEVEL_INFO`. Any logging method with a level that is equal to or higher than `LOG_LEVEL_INFO` (e.g., `LOG_INFO`, `LOG_WARN`, `LOG_ERROR`) will emit logging information.

Using assert to force check the correctness of your index implementation. For example, when you try to delete a page, the `page_count` must equals to 0.

Using a relatively small value of page size at the beginning test stage, it would be easier for you to check whether you have done the delete and insert in the correct way. You can change the page size in configuration file (`src/include/common/config.h`).

Post all of your questions about this project on Piazza. Do **not** email the TAs directly with questions.

GRADING RUBRIC

Each project submission will be graded based on the following criteria:

Does the submission successfully execute all of the test cases and produce the correct answer?
Does the submission execute without any memory leaks?

LATE POLICY

See the late policy in the syllabus.

SUBMISSION

After completing the assignment, you can submit your implementation of to Gradescope. Since we have two checkpoints for this project, you will need to submit them separately through the following link.

https://www.gradescope.com/courses/163907/

You only need to include the following files:

CHECKPOINT #1

- Every file for Project 1 (4 in total)
- `src/include/storage/page/b_plus_tree_page.h`
- `src/storage/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`

CHECKPOINT#2

- Every file for Project 1 (4 in total)
- `src/include/storage/page/b_plus_tree_page.h`
- `src/storage/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`

You can submit your answers as many times as you like and get immediate feedback. Your score will be sent via email to your Andrew account within a few minutes after your submission.

COLLABORATION POLICY

Every student has to work individually on this assignment.
Students are allowed to discuss high-level details about the project with others.
Students are **not** allowed to copy the contents of a white-board after a group meeting with other students.
Students are **not** allowed to copy the solutions from another colleague.

⚠ WARNING: All of the code for this project must be your own. You may not copy source code from other students or other sources that you find on the web. Plagiarism **will not** be tolerated. See CMU's **Policy on Academic Integrity** for additional information.

Last Updated: Nov 01, 2020