Design Report

Design Choices

The following are some assumptions on which our B+ tree design is based on:

- All records in a file have the same length.
- It only supports single attribute indexing.
- It only supports integer as the indexed attribute type.
- Duplicate keys will not be inserted.

The following are some special values we use:

- The relation name should be less than 20 characters.
- We use the constant `Page::INVALID_NUMBER` which is 0 to denote that it is invalid.
- A record ID will be invalid if its page ID field is invalid.
- When there is no next valid record during a scan, the variable `nextEntry` is set to -1.
- The level of the non leaf nodes is set to 1 if their children are leaves. 0 otherwise.

We don't add additional information in the given structures, but we do define some private methods to help implement the public ones. Their functionalities are explained in the documentation.

The following are some of our implementation choices:

- We set the root to be always non leaf. Initially, the root has no key, and a leaf page is stored at the first index. The inserted data entries will go into this leaf page until it splits. This will help the recursive methods determine the base case more easily.
- When there is a need to split, and the number of data entries is odd, the newly split page contains one more entry than the original one. The detailed information is in comments of the auxiliary methods of insertion.
- To begin scanning, the leftmost leaf possibly containing the lower search bound is found. To be specific, such leaf is found recursively by going into pages with an upper bound greater than or at least (GT/GTE) the lower search bound. Note that it might not actually contain the keys we want. In this case, we simply go to the next leaves by following the right sibling pointers, unless the key does not lie within the search bound.
- Our implementation is efficient since the B+ tree is balanced.

Additional Test Cases

We add 6 more test cases in `main.cpp`:

Test4: In this test, we want to test if the B+ Tree index can correctly deal with negative keys.

Test5: In this test, we want to test the performance when the relation size is large. Here we take 100000 data, and it does take some time but still not long.

Test6: In this test, we want to test when the data is discrete where step size is 2, the number of data entries would decrease half compared to the continuous data.

Test7: In this test, we want to test the boundary, and check if the B+ Tree index can deal with edge cases.

Test8: In this test, we want to test if the size returns 0 when the tree is empty.

Test9: In this test, we want to test the range not from 0. Here we set the range from 100 to 10000, and if scan over the range, it returns 0.