Design Report

One of the design principles of the implementation is to keep pages pinned for the least amount of time. Except when during scanning, we usually read and unpin a page immediately. For example, when trying to locate a key value in the leaf node, we unpin the parent page before we read the child page. We enforce this principle by always do read and unpin in the same method.

A rule of thumb is that we never read a page and pass it to another method to unpin, we think that is extremely error-prone. (One exception of this is of course the scan function). In order to adhere to this rule, we had to use a function that initially seems to be doing too much, the nonLeafSplitInsertEntry method. This method first splits a non leaf node, and then inserts the new entry to the correct node. The original design decision was to break them down into two different methods, one to split and another one to insert (which we need to implement anyway). The problem with only implementing a split method is that we either need to pass a page up to the pass to the caller to be freed or we need to pass the page number to the caller to be re-read, neither of which seems to be an efficient choice. Therefore, we integrated the insertion inside the method.

Another design principle is to keep testing as straight forward as possible. As you probably can see in the source code, we have rigorous unit tests for almost every method. We also have many assertions before and after critical methods to ensure the data structures are in a desired state. This boosts the confidence level in the correctness of the implementation.

One guarantee that is provided is that keys will be unique. This helped simplify the code that deals with many instances of the same key. Specifically, suppose the leaf node can only store 100 value/rid pairs, and we have 100 occurrences of the key '1'. Now if we want to insert another '1'. We can have one of the two scenarios:

1. We use an overflow page to hold the '1', instead of splitting the page. This requires us to do additional bookkeeping in the data structure to manage overflow page.

2. We split the leaf page. We will have something like (pointer A) 1 (pointer B) 1 (pointer C). This breaks a assumption in the non-leaf node: the left pointer of a key points to value that is less than the key, and the right pointer points to value larger than the key. But this is fine in our current implementation (as far as this project is concerned. i.e. no deletion). To illustrate it suppose later the non-leaf node looks like this: (pointer A) 1 (pointer B) 1 (pointer C) 1 (pointer D) 2 (pointer E), and when we am going to insert 3 (pointer F), we need to split the node. In the process, we pop the key '1' up to the parent level. But now at the parent level if we going to search for '1', we should be conservative and fall back to the first key that is to the left of '1', since there could be many '1's between the two keys. This traversal may bring performance penalties and coding complexity. Therefore, the fact that we don't need to handle duplicate keys simplified our implementation.

To make insertion as simple as possible, we guarantee that the root node must be a non-leaf node, even if all data can fit in leaf node. During initialization of the index, we create an empty non-leaf node as root, and set the first pageNo to point to an empty leaf node. In this way, all subsequent inserts can be correctly performed.

One problem with the original interface provided in the assignment is that there is no built-in to detect whether a node is full. To solve this problem, we use a special value in the key array (EMPTY_SLOT) to indicate that the slot is not initialized. As long as we can find a slot with value EMPTY_SLOT, we assume the array is not full. The value EMPTY_SLOT is set to be the minimum negative number (1<<(sizeof(int)*8-1)), since we think this number will interfere with real data the least. But this is a bad design to begin with. The ideal solution is to add a 'count' field in both the leaf and non-leaf node to indicate if they are full.

Lastly, we created an additional class of PageIDPair, which holds a page pointer and a PageId. This class is returned from our internal helper methods to create new page.