Yunfan Long

Muchan Li

Design Report:

1. Implementation Choices:
    - Pinned Pages: In our implementation, pages are kept pinned only when we read them in from the buffer manager for accessing their information. Once the required operations on a page are completed, we aim to unpin that page as soon as possible so that other pages can be loaded into the buffer pool, as well as freeing that page up for subsequent read access and operation. This approach minimizes the time that pages are kept pinned and maximizes the efficient utilization of the buffer pool. However, although we aim to unpin and free up pages as frequently as possible, we may still have lost track of the pinning once or twice, so as a safety net, we implemented one function for our buffer manager for unpinning all pages in the buffer pool as the index is de-structed. This way we ensure that all pages are freed up and changes are propagated back to disk.
    - Recursive Insertion and Bottom Up Update: we depended very heavily on recursive function calls to propagate key updates as a result of insertion starting from the bottom-most leaf level upward. For instance, if we try to insert a new key value of 10, our first recursive function call would just find the appropriate page number at the root level to descend to and halt; the next recursive function call, assuming it reaches the leaf level, would call the function that actually performs insertion, and should a node-split happen, record the key to be copied up and return. Coming back to the first recursive call, the keys at the leaf levels are already updated and the key that's copied up is retained, the function continues to handle the copied-up key and finishes. We believe our approach is very efficient and elegant given that all operations are carried out during recursive stacks and resources are created and let go as one level of stack frame finishes, and changes are propagated back to the parent frame–no memory is wasted and the recursive stack frames are kept short and concise since the depth of a B+ tree grows very slowly.
2. Additional Design Choices:
    - Support for Multiple Attribute Types: our design includes support for multiple attribute types such as integers, doubles, and strings. This allows flexibility in handling different types of keys in the B+ Tree structure. The implementation utilizes generified templates to handle each attribute type separately and performs appropriate type casting and comparisons based

on the key types. This design choice enables a versatile/generified B+ Tree implementation that can handle a wide range of attribute types with very concise code.
    - Level Marking: Our design also leverages the level attribute of any non-leaf data node so that when performing tree traversal, we conveniently know when to stop for insertion when certain levels are present. For example, we made it so that the non-leaf layer directly on top of the leaf nodes is marked with level 1 while all other non-leaf layers higher up in the tree have level 0. Then, whenever during the recursive function calls we see a non-leaf node with level 1, we know we are 1 step away from stop and performing insertions.
3. Handling Duplicate Keys: If the B+ Tree were allowed to have duplicate keys, some design and implementation changes would be necessary:
    - Our main focus was to preserve that duplicate keys would only be allowed at the leaf level but nowhere else. To achieve so, we strictly enforced that when a "push-up" happens from the leaf level, the pushed up key can be copied up; whereas when a "push-up" happens from the non-leaf level, we made sure that we remove the pushed up key and shift the resulting key and page number arrays.
4. Test Design:
    - We essentially designed our extra test case "additionalTest" around edge cases so that our model can be more thoroughly tested. For this test case, we first created a randomized relation since we believe this would be the most robust approach to testing whether our index tree can still carry out proper scans. Intuitively, if we can guarantee correctness on randomized inputs, we should easily handle inputs that are in either sorted ascending or descending order. We tested our implementation in particular against the edge cases where we scan the left or right end of all inputs, so for instance, from 0 to some value or from some value to 5000, given that 0 to 5000 is the range of input keys. We shuffled a few pairs of range operators on these edge cases and it all worked fine, making us confident that our design and implementation should generalize to even more robust inputs and hidden cases.