Patrick Loughran, Zahaan Motiwala, Sion Wilks-Boguszewicz

CS564 – Lec002

3/8/2022

B-Tree Design Report

To understand our B+ Tree implementation, we will lay out the major design choices we choose and how they affect the final time complexity of our index. Most important to note is that our implementation maintains the most powerful efficiency a Btree has to offer: low I/O cost searches. There are inefficiencies in our code that could be improved upon in future use, but the main goal of keeping page I/Os' down is maintained throughout our implementation.

Search:

When searching through our Btree, we correctly utilize the reduced search space provided by Btrees and only search through one page per level. This brings our I/O cost to be the height of the tree along +/- a couple I/Os depending on how much of the tree can be kept in RAM and if the index is a type c index or not. Our implementation could be further optimized by how we perform our internal page searches. Currently we iterate through the page to determine where we need to go next. While this still provides us low I/O cost, we lose time due to extra internal searches that could be implemented using binary search or interpolation search if we had insights into our relation.

Insert: Our insert algorithm functions very similarly to our search function. The number of pages we need to get is limited to one per level as in search. We then insert the record if it fits, and if it doesn't, we split the node. Once again, this could be optimized as all internal searches are linear as opposed to log(n). Furthermore, moving around Keys so they stay in order is done linearly. This could also be optimized.

Notes:

- Our root node was elected to always be a NonLeafNode to keep insertion logic easier to implement.
- Our index may fail when performing NonLeafNode splits. We tried to ensure this but ran out of time to fully make sure this always works. Obviously, this would be an issue in a real B+ tree. Were we to ensure this, I think keeping a stack of parent nodes as referenced

in piazza would be a wise way to accomplish this without needing to add parent node pointers to every leaf.

- Any page we pin in unpinned as soon as it is of no more use. The only page that stays unpinned for a while is the root page in the insert method. This is because we are constantly referencing it to perform operations and update fields. The root is unpinned when no longer needed.
- We decided to use many helper functions as our code grew quite large.

Tests:

We were asked to add two additional tests. We added three addition tests and commented out one due to long run time. All tests are in main.cpp.

- **testNonLeafSplit() ->** Tested the splitting of non leaf node to verify our index implementation is capable of indexing a large amount of relations.
- **testNegative() - >** The second test we created was designed to check for negative numbers. Checking for negative numbers was tested since are implementation was supposed to not behave differently when processing negative number, i.e, our logic should also work for negative numbers.
- **testEmptyTree() ->**The third test we created was for testing an emptyTree. We wrote this test to check that our scan methods could handle a tree index made from a file with no records. We later learned from Piazza that this test case will not be tested.