A1: B+Tree

Andrew D. Pham, B.Sc. CS
CSCI S-165 (34555), Data Science and Machine Learning
Harvard University
31 July 19

***Abstract*— Documentation for implementation of B+Tree, theory, and findings. B+Trees are efficient at accessing information, updating information, and most of the time, are efficient at inserting unless the tree requires rebalancing.**

I. B+Trees

THE B+tree is a data structure like a binary search tree except with a larger fan out and an in order traversal link at the bottom. Data is only held at the bottom leaves. The large fan out of a B+Tree makes them highly efficient since the number of nodes to traverse is dramatically reduced with fan out, making them ideal for large datasets. The trade off is highly increased complexity and occasional insertion slows when the tree requires rebalancing

II. My Implementation

A. Structure

The B+Tree is structured with a fan out of 3 meaning nodes have two keys (a and b), two values (vala and valb), and three pointers (p1, p2, and p3). We use the same node for internal nodes and leaves. In the internal nodes, the pointers point to nodes one depth down “less than a”, “between a and b”, and “greater than b”; the keys are used as pivots. In leaf nodes, the pointers point to the prior and next leaves. Lastly we have a boolean value, leaf, that tells us if we are at a data node. Since we use the same node structure for leaves and internal nodes, the maximum size of both is two data points, three fan out.

B. Find

Find simply traverses the tree until it encounters a leaf. If it encounters a leaf it will check its keys and return the value if found. If the value is not found it looks to its data neighbors to check them for the key. It halts the in order check when the key it’s checking for is larger than the keys stored in the current node traversing left, and smaller than the keys stored traversing right.

C. Insert

Insertion first uses the find function to check if they key is already in the tree. If the key is found, we update the value. If the key is not found, the new key and value is inserted at the proper location in the leaf nodes. The function then uses the in order traversal pointers to shift all greater data down one spot, creating a new node at the end if overflowing. Then a tree rebuild is called.

D. Tree Build

A new tree is built at the beginning of the program, and during every insert that causes a shift. The tree builder takes the root node of an existing tree, follows p1 until it finds the first data node, then loads all of the data into a dynamic array. The tree builder then builds a tree with one depth and checks if it is large enough to fit the data. It adds depth to the tree until all the data can be loaded into the bottom leaves. Once the tree fits and the data is loaded into the bottom leaves, the internal node values are calculated bottom up. Therefore the minimum content threshold required in this implementation before a rebalance is three, an overflow. If there is an overflow and we need to insert one more piece of data than the leaf can hold, we shift and rebalance the tree.

E. Range

Range searches for keys falling between a high key and a low key. Range uses the same implementation as the find function to search for the low key. Then range uses the in order pointers to save all of the keys in an array until it finds the high key and returns the array. The other method for finding and returning the qualifying keys is to find both the high and low keys and collect all of the keys between them. This may be better because we do not need to compare and check every key to the high key when traversing each node

III. Theoretical Efficiency

As stated before, the efficiency of a B+Tree comes from its fan out. We are going to be focusing on our fan out of two and not a general case. From elementary data structures, the worse case access for a binary search tree would be a leaf at the bottom of the tree. A binary search trees have max nodes equal to 2^d-1. Say we have 10,000 data points. A balanced BST we would have a worse case depth of 13. While the worse B+Tree we can make, with a fan out only one larger than a BST would have a worse case depth of 8. The maximum number of data points our B+Tree can hold at a certain depth is described by the formula 2*3^d.

IV. Insertion vs. Find

A. Intuition

From our implementation, the find algorithm should take significantly less time than the insertion algorithm. First and most obviously, insert calls find to check if it needs to update a value or add a new data point. Secondly, find never calls for a rebuild or rebalancing of the tree. It should be noted however that both insert and find take the same amount of node traverses because all data is located at the bottom of a B+Tree.

B. Big O

The Big O efficiency of both lies in efficiency class, logarithmic time O(log n) since the dominating factor is how long it takes to traverse a tree.

C. Method

To empirically time insert and find, the C library <time> was used. To ensure consistency, all print statements were purged from the code and the tests were run on the same machine.

D. Empirical Analysis

Using a tree with depth 3 or 54 data points, a find operation takes on average 2 microseconds while insert takes on average 5 microseconds, over double the time. While both are in the same efficiency class, since insertion uses find its algorithm, we see that it is much faster. However, in very large data sets, we can expect the percentage gap to close as they are part of the same efficiency class.

V. Find vs Range

Find and range use the same algorithm to find a value, or target key. Their only difference being, range then in order traverses to find the high key and add keys to the list. In a worst case scenario, range would travel from the first leaf, all the way to the end leaf giving it a Big O efficiency of O(n) with n being the number of data points. Range does scale with depth of the tree, but its dominating factor is the size of the data. The maximum amount of nodes needed to access this would be all of the nodes it traverses through to reach the data, same as find, plus all of the nodes it takes to reach the high key. Specifically range traverses through O(log n+n) nodes. We already know that find has a Big O efficiency of O(log n). Meaning the cost of find is an entire efficiency class faster than range's.[2]

VI. Conclusion

Thus we've learned the basic form and function of a B+Tree, the tradeoffs between quick accesses to inserts, and the challenging complexities of these database models.

Notes

[1] Tests are described in readme of project, and in console print out when run.