Justin Baraboo, class ID 2
group 1 Homework 1

September 11, 2016

1 Cheat Sheet

Indexing techniques.

B-Trees: Data is throughout the tree.

Advantage: You don't always have to reach a leaf though. Will also have few nodes than a B+ tree as keys aren't redundant.

Searching $O(log_2(n))$.

Disadvantage: leaves can't be linked.

B+Trees: All of the data (generally pointers) are in the bottom leaves. Leaves may be linked to the next leaf. You must traverse to the bottom.

Advantages: High fan-out makes few block accesses. Grows from root, making splitting easier. block-oriented. It's balanced. Searching $O(log_{fanout}n)$.

Disadvantages: Multidimensional trees (indexing by two or more values) are complicated.

Tree Traversals for Binary Tree

All are $O(n)$.

Pre: Display, left, right

In: left, Display, right

Post: left, right, Display

KD Trees: split the entire region/subregion. generally based off median if point based. $O(nlogn)$ to build. $O(logn)$ to insert.

Advantages: Tree based searching/insertion/deletion. Good for multidimensional data. Great for point based data. Balanced tree and densties.

Disadvantage: Tree can grow large for many points. Slices entire regions on decison. Rebalancing issues.

R trees:based on minimum bounded boxes to enclose regions or points.

Advantages: Tree based searching/insertion/deletion. Only needs to segment parts of the space. Regional data is easier to index and you have flexibilty of what shapes you use to make the tree (doesn't always have to be a rectangle). Different ways to split and form the tree.

Disadvantages:Overlapping shapes is complicated. Splitting rectangles is time consuming.

Z-Order Curve Index: way of indexing space from xd to 1d.

Advantage: Allows B+ trees and other 1D methods to affect multiple dimensions

Disadvantage: The hopping it does makes points that aren't really near it each, become near each other Area must also be tileable.

Hilbert Curve: Uses a rotating grammar of forward/turn to fill the space

Advantage: fast and points tend to stay near each other. Disadvantage: area must be tileable.

2 Bloom Filter

I implemented a Bloom Filter in Spark.

Solution A Bloom filter is a probablistic way of indexing, where it chiefly answers whether an element might be stored in a structure or is not stored. That is, it may say something is in, and it isn't, but will never say something isn't in, and it is. While it isn't always true, it is very fast and easily parallelized. This creates a great primary filter to see if you need to search for the element in a block or if you'd just be wasting your time doing so.

To create a Bloom filter, you use several hashing functions and an array to map them to, corresponding to true or false. For example, if you have an array of 100 fields, you would hash "cats", to some value within that for each hashing functions. Then, to see if "cats" is in it, you'd hash it again under the same functions, and check the positions and see if they are true.

False positives come into into play as follows: Let's say you have 2 hashers and an array of 100 values. You also hash "cats" to 5 and 10 and "dogs" to 1 and 20. Let's say you query for "mice" which hashes to 10, 1, then the filter would report true even though it hasnt' been added.

The values we choose for our array length and how many hashers is important to reduce the false positive rate. If you only had 1 hasher, you'd be doing basic hashing and needs a huge array to give everyone their own seperate value. Multiple hashers require multiple points to be set to true for the value to be considered to be contained. However, if you have a small arrary length compared to how many hashers you employ, you'll fill up the array with very few elements. Thus, to reduce the false positive rate, you want to keep fairly large array length in relation to the hash number, employing many hashers as possible.

In fact, assuming independence of each bit being set, the false positive rate for k hashers, n elements inserted, and an m sized array is approximately:

$(1 - e^{-kn/m})^k$

giving the optimal number of hash functions as : $k = \frac{m}{n} ln2$.

cite: `[https://en.wikipedia.org/wiki/Bloom_filter]`

In my code, I insert 104 elements. I then test on 113 queries where only 3 of them are the same. With $m = 4000, k = 24$ we achieve 5 matches, 3 false positive, and 105 true negatives. With $m = 100, k = 2$ (small array), we achieve 5 matches, 87 false positives, and 21 true negatives. With $m = 700, k = 1$ (single haser) we achieve 5 matches, 21 false positives, and 87 true negatives. With $m = 700, k = 4$ (optimal hasher) we achieve 5 mathces, 12 false positives, and 87 true negatives.

For $m = 700, k = 4$: Running it on 100 words query and insertion gave 4902 milliseconds.
1000 words: 4870
10000 words: 5129
100000 words: 5250
1000000 words: 6120
2700000 words: 6885
2849748 words: 6819
(I gave it the same input for insertion and query, copy pasted the file data over and over until I got around those values of words). So it is time taken grows very slowly. The complexity is really just the complexity of the hash function for each element so it's probably $O(n/k)$ (n input, k workers).

I'll have the github code and input/output linked in the github pages.

3 B+ Tree and Binary tree

I also tried to construct a B+ tree and Binary using parallel techniques.

Note that these are only partially done. Also, each of these would implement a sorting map/reduce that is easy to do in Scala but really hard to implement in Java (where I'm using Java as I'm not used to and was having difficulty with structures in Scala). Java unfortuantely doesn't have a sortByKey() method.

BST The issue with creating a binary tree is that we can no longer use recursion to solve the problem. What I do is find the element, find it's index in the sorted array, and then based on that know where its children are. To make life simplier, I only looked at full binary trees (you could add false elements to the array until you had the proper number).

If we start our indexing at 1 for our sorted array, every odd node indexed element will be a leaf node. For the inner nodes, we look at what fraction of their index and the total. The basic idea is that if you are at $n/2$ position, your children will be $n/4$ away from from you. So you just have to find the node at that location, and set your children to be that.

This runs in awful time compared to the basic recursive way of converting a sorted array to a bst (even if we assume that the values are just the same as the indexes or that they are mapped together as a (value, index) pair).

For comparison a tree with $2^20 - 1$ nodes takes 2830 milliseconds for the spark version and only 145 for the regular recursive version. (BST Spark vs BST Regular in git)

Querying for items is really fast though, especially if the tree it tall enough and you have a lot of items to query.

B+ Tree For the B+ tree, you have m elements per node and $m + 1$ pointers to other nodes.

I partition the sorted array into m paritions and add them to the nodes, these become the leaves. I then try and merge the leaves together, starting from the first and working my way onward. There are some tricky places where you have to manage splits of nodes when they reach capacity.

I also added a function to find if a value is in the table or not. I didn't add a insert or delete function (mostly as merging the guys together was basically an insert function and I didn't want to mess around with that anymore and that deleting isn't that usefull when working with large data sets as you can just reallocate the value with something new instead).