**Winter 2022 - CS540 – Assignment 4 (Written) (Group 15)**


| Full Name | OSU Email ID | ONID |
|---|---|---|
| Bharghav Srikhakollu | srikhakb@oregonstate.edu | 934289891 |
| Akhil Sai Chintala | chintala@oregonstate.edu | 934409906 |

1. **Tree Indexing:**

**(a) Explain how we should change the insertion and deletion algorithms in a B+-tree to implement and maintain prefix key compression.**

Insertion Algorithm in a B+ tree: For insertion algorithm, through search algorithm, first we will identify the leaf on which insertion must be done. If this leaf is to be split, we should make sure that we choose a split point with the shortest separator (prefix of a key). Insert the shortest separator to the index part and continue splits towards the root based on the data. After doing this, if we want to split a branch node (inner node) select the shortest separator again within the split and move it to the parent node.

In this way, we should adjust the tree to maintain the prefix key compression.

Deletion Algorithm in a B+ tree: Deletion is always done from a leaf. If the deletion results in a merge of two leaves, then we should delete the respective separator from the index part. Just in case we are deleting the smallest or largest key/record on a leaf, then the separator in the index part needs to be replaced with a shorter separator as we deleted the smallest/largest key.

In this manner, we should adjust the tree to maintain the prefix key compression.

Here as mentioned, shortest separator can be understood as below:

We can select a string (part of string) such that x1(node1) < shortest separator <= x2(node2) and we store it in the index part to separate the two nodes. Such a string can be called as the shortest separator (prefix key compressed).

2. **Tree Indexing:**

**Consider relation R (X, Y) that stores points in a in a two-dimensional space. Assume that users would like to search the points stored in R based on their relative positions on x-axis or y-axis. More precisely, users would like to find all points that are on the left (right) of an input point (x-axis), find all points that are up (down) relative to an input point (y-axis), or both. Our goal is to build a single index over R that can efficiently answer these queries.**

**(a) Explain why a B+-tree may not be a good data structure for these types of queries.**

Usually, B+-tree is best suited for making splits in one dimension. Here as per the information provided, we must deal with two dimensions, hence we cannot use a regular B+-tree. B+ trees can handle only one-dimensional data and are not suitable for multidimensional data. Hence these trees may not be a good data structure for these types of queries.

**(b) Describe a tree-index structure that will efficiently perform search over R. Explain its search and insertion algorithms.**

To handle two-dimensional scenarios, we can check for R trees and their types which can handle point data and region data. These tree-index structures will efficiently perform search over R.

R-trees are usually meant for spatial access methods such as indexing multi-dimensional information such as rectangles, polygons, and geographical coordinates. A type of R-tree – R+-tree is suitable for checking data using coordinates points (x, y).

Reference citation:

download;jsessionid=AB3A17A117E8E340FCEB51B907ED061F (psu.edu)

R+-tree Search Algorithm:

For a given search space, first, we must decompose the space into sub-regions. Then for each sub-region, we must descend the tree until we find the data objects in the leaves.

1. Let RECT be the bounds of data objects from the input, W is the search window, R be the root node.
2. We need to decompose the given search space and search the tree recursively.

--> Search Intermediate notes

If R is not a leaf node, then consider an entry (x, RECT) of R. Check if R overlaps with the search window W. If yes, then search the Child, RECT, where Child is the node pointed to by x.

--> Search Leaf nodes

If R is the leaf node, we must check all the objects in R and return the ones which overlap within the search window W.

R+-tree Insertion Algorithm:

We have to insert a new rectangle of data in the R+-tree. For this, we must search the tree and add the rectangle in leaf nodes.

1. Let RECT be the bounds of data objects from the input, Y is the input rectangle, R be the root node.
2. We need to find where Y should get inserted and we should add in the corresponding leaf nodes.

--> Search Intermediate notes

If R is not a leaf node, then for every entry of R we should check if RECT overlaps Y. If yes, then we should insert Child, Y, where Child is the node pointed to by x.

--> Insert into Leaf nodes

If R is the leaf node, we should add Y in R. If the new rectangle which is inserted into R has more entries than required, then we must split the root node R and reorganize the tree.