B+ Tree Implementation Explanation

Konstantinos Pisimisis

February 21, 2024

Introduction

In this document, I present a detailed analysis of a B+ tree implementation authored by Ashwin's. B+ trees are fundamental data structures known for their efficiency in supporting balanced search and retrieval operations. This exploration aims to provide a comprehensive understanding of the implementation, shedding light on its design principles and functionality.

The B+ tree code crafted by Ashwin'sserves as the focal point of this analysis. The report seeks to dissect various aspects of the code, ranging from the intricacies of node structures to the algorithms employed for insertion and search operations. Through this exploration, I aim to unravel the thought processes and decisions that shaped Ashwin'sB+ tree implementation.

As the author of this document, my goal is to offer readers an in-depth perspective on the code, providing valuable insights for those interested in comprehending B+ tree structures and algorithms. This analysis is not only a dissection of Ashwin'swork but also an opportunity for collaborative learning and exploration within the realm of B+ tree implementations.

Code Explanation

1. Node Structure

Attributes:

The node structure is fundamental to the B+ tree's organization. Each node possesses the following attributes:

1. `uidCounter`: A unique identifier assigned to each node.
2. `order`: The maximum number of keys that each node can accommodate.
3. `parent`: A reference to the parent node.
4. `keys`: A list containing the keys held by the node.
5. `values`: A list of child nodes.

Methods:

The node class includes the following methods:

1. `split`: This method is responsible for splitting a node into two new nodes. To elaborate, it creates left and right nodes and updates their pointers accordingly. Consider a scenario where a node, let's call it N, has keys [1, 2, 3, 4] and values [a, b, c, d, e]. Given an order of 5, with `max_keys = order - 1 = 4`, a split is necessary. The resulting left node will have keys [1, 2] and values [a, b, c], while the right node will have keys [4] and values [d, e]. Subsequently, node N retains the key [3] with updated pointers to the left and right nodes.
2. getSize : Helper method about the state of the node
3. isEmpty: Helper method about the state of the node
4. isFull: Helper method about the state of the node
5. isNearlyUnderflow: Helper method about the state of the node
6. isUnderflow: Helper method about the state of the node
7. isRoot: Helper method about the state of the node

1 Leaf Node Structure

The 'LeafNode' class, a subclass of 'Node', represents the leaf nodes within the B-Tree. Apart from inheriting attributes from the 'Node' class, each 'LeafNode' object incorporates two additional attributes: 'prevLeaf' and 'nextLeaf'. These attributes serve as references to the preceding and succeeding leaf nodes in the B-Tree, facilitating seamless traversal of keys in sorted order.

One notable method overridden by the 'LeafNode' class is 'add', inherited from the 'Node' class. This method enables the addition of key-value pairs to the leaf node. In cases where the key already exists, the new value is appended to the list of values corresponding to that key. If the key is absent, it is inserted in the appropriate position to ensure the preservation of the sorted key order.

Additionally, the 'LeafNode' class overrides the 'split' method inherited from the 'Node' class. This particular method plays a crucial role in dividing a full leaf node into two new nodes. Differing from the 'split' method in the 'Node' class, the 'LeafNode' variant also updates the 'prevLeaf' and 'nextLeaf' pointers of the affected nodes. This update is necessary for maintaining the linked list of leaf nodes, ensuring the integrity of the B-Tree structure.

2 BplusTree Structure

The 'BPlusTree' class serves as an implementation of the B+ Tree data structure, offering a balanced and efficient means of organizing and managing data. Below is an overview of the key methods within this class:

1. `__init_(self, order=5)`: The constructor method initializes a new B+ Tree with a specified order (default is 5). The order determines the maximum number of children a node can have.

2. `_find(self, node: Node, key)`: A helper method that identifies the appropriate child node for a given key, returning both the child node and its index in the parent's values list.

3. `_mergeUp(self, parent: Node, child: Node, index)`: This helper method facilitates the merging of a child node into its parent at a specified index. It becomes crucial when a node undergoes a split, necessitating the promotion of one of its children to the parent's level.

4. `insert(self, key, value)`: The `insert` method adds a key-value pair to the B+ Tree. It initiates at the root and navigates down the tree to locate the appropriate leaf node for the key. If the insertion causes a node to exceed its capacity, the node undergoes a split, and the middle key is promoted to the parent node.

5. `retrieve(self, key)`: This method retrieves the value associated with a given key. Beginning at the root, it traverses the tree to find the leaf node containing the key, subsequently returning the associated value.

6. `delete(self, key)`: The `delete` method removes a key and its associated value(s) from the B+ Tree. If the deletion results in a node becoming underfull, the node either borrows an entry from a sibling or merges with a sibling.

7. `_borrowLeft(self, node: Node, sibling: Node, parentIndex)`: A helper method that borrows an entry from a node's left sibling. This operation becomes necessary when a node becomes underfull following a deletion.

8. `_borrowRight(self, node: Node, sibling: Node, parentIndex)`: A helper method that borrows an entry from a node's right sibling, serving the same purpose as `_borrowLeft` in managing underfull nodes.

9. `_mergeOnDelete(self, l_node: Node, r_node: Node)`: Another helper method, `_mergeOnDelete`, combines two sibling nodes post-deletion. This occurs when both a node and its sibling are underfull after a deletion.

3. Insert Method in B+ Tree

The following code snippet represents the 'insert' method of a B+ Tree implementation:

```
def insert(self, key, value):
    node = self.root

    while not isinstance(node, LeafNode):

    # Node is now guaranteed a LeafNode!
    node.add(key, value)

    while len(node.keys) == node.order:  # 1 over full
        if not node.isRoot():
            parent = node.parent
            node = node.split()  # Split & Set node as the 'top' node.
            jnk, index = self._find(parent, node.keys[0])
            self._mergeUp(parent, node, index)
            node = parent
        else:
            node = node.split()  # Split & Set node as the 'top' node.
```

```
18                 self.root = node # Re-assign
```

Listing 1: Insert Method

Explanation:

1. `node = self.root`: The method begins at the root of the B+ Tree.
2. `while not isinstance(node, LeafNode)`: This loop continues as long as the current node is not a leaf node, ensuring the method descends to the appropriate leaf level.
3. `node, index = self._find(node, key)`: The `_find` method is invoked to locate the child node that should contain the key, returning both the child node and its index in the parent node's values list.
4. `node.add(key, value)`: Once the suitable leaf node is identified, the key-value pair is added to it using the `add` method of `LeafNode`.
5. `while len(node.keys) == node.order`: This loop continues as long as the node is full. If the node reaches its order, it needs to be split.
6. `if not node.isRoot()`: If the node is not the root, it has a parent.
7. `parent = node.parent`: The parent of the node is stored in `parent`.
8. `node = node.split()`: The `split` method is called to split the node into two. The method returns the parent of the two new nodes.
9. `jnk, index = self._find(parent, node.keys[0])`: The `_find` method is called again to find the index of the first key of the split node in the parent's keys list.
10. `self._mergeUp(parent, node, index)`: The `_mergeUp` method is invoked to update the parent node with information about the split node.
11. `node = parent`: The method continues with the parent node.
12. `else: node = node.split()`: If the node is the root, it is split without updating a parent node.
13. `self.root = node`: If the root was split, the new root becomes the parent of the two new nodes.

This method ensures the B+ Tree maintains its properties post-insertion, including consistent leaf node depth and adherence to the order constraints for internal nodes.

4. Retrieve Method in B+ Tree

The following code snippet represents the 'retrieve' method of a B+ Tree implementation:

```
node = self.root
while not isinstance(node, LeafNode):
    node, index = self._find(node, key)

for i, item in enumerate(node.keys):
    if key == item:
        return node.values[i]

return None
```

Listing 2: Retrieve Method

Explanation:

1. `node = self.root`: The method initiates at the root of the B+ Tree.
2. `while not isinstance(node, LeafNode)`: This loop persists as long as the current node is not a leaf node, ensuring traversal to the appropriate leaf level.
3. `node, index = self._find(node, key)`: The `_find` method is invoked to locate the child node that should contain the key, returning both the child node and its index in the parent node's values list.
4. `for i, item in enumerate(node.keys)`: This loop iterates over the keys in the leaf node. The `enumerate` function returns both the index `i` and the key `item`.
5. `if key == item: return node.values[i]`: If the key matches the item, the corresponding value is returned. Each key in a leaf node has a corresponding value in the `values` list at the same index.
6. `return None`: If the key is not found in the tree, the method returns `None`.

This method facilitates the retrieval of the value associated with a given key in the B+ Tree. In case the key is not present in the tree, it gracefully returns `None`.

5. Delete Method in B+ Tree

The following code snippet represents the 'delete' method of a B+ Tree implementation:

```
node = self.root
while not isinstance(node, LeafNode):
    node, parentIndex = self._find(node, key)

if key not in node.keys:
    return False

index = node.keys.index(key)
node.values[index].pop()

if len(node.values[index]) == 0:
    node.removeKeyAndValue(index)

while node.isUnderflow() and not node.isRoot():
    self._manageUnderflow(node, parentIndex)
    node, parentIndex = self._find(self.root, key)
```

```
17
18 if node.isRoot() and not isinstance(node, LeafNode) and len(node.values
       ) == 1:
19     self.root = node.values[0]
```

Listing 3: Delete Method

Explanation:

1. `node = self.root`: The method commences at the root of the B+ Tree.
2. `while not isinstance(node, LeafNode)`: This loop persists as long as the current node is not a leaf node, ensuring traversal to the appropriate leaf level.
3. `node, parentIndex = self._find(node, key)`: The `_find` method is invoked to locate the child node that should contain the key, returning both the child node and its index in the parent node's values list.
4. `if key not in node.keys: return False`: If the key is not in the leaf node's keys, the method returns `False`, indicating an unsuccessful deletion.
5. `index = node.keys.index(key)`: The index of the key in the node's keys list is found.
6. `node.values[index].pop()`: The last inserted data associated with the key is removed.
7. `if len(node.values[index]) == 0:`: If there are no more data associated with the key, the key and its associated value are removed from the node.
8. `while node.isUnderflow() and not node.isRoot():`: If the node is underflowing and it's not the root, the method attempts to rebalance the tree by borrowing a key from a sibling node or merging with a sibling node.
9. `self._manageUnderflow(node, parentIndex)`: The `_manageUnderflow` method is invoked to handle underflow conditions.
10. `node, parentIndex = self._find(self.root, key)`: The `_find` method is used again to update the node and parent index post-deletion.
11. `if node.isRoot() and not isinstance(node, LeafNode) and len(node.values) == 1:`: If the node is the root, not a leaf node, and has only one child, the root of the tree is updated to be that child.
12. `self.root = node.values[0]`: The root of the tree is updated to be the sole child of the former root.

This method is employed to delete a key and its associated data from the B+ Tree. If the key is not found in the tree, it returns `False`. If the deletion of the key causes a node to underflow, the method endeavors to rebalance the tree. If the root node becomes unnecessary (i.e., it has only one child), it is removed, and its child becomes the new root.