Performance Boosts of Using a B-Tree: Takeaways

Syntax

- Btree complete node implementation:

```
import bisect
class Node:
    def __init__(self, keys=None, values=None, children=None, parent=None):
        self.keys = keys or []
        self.values = values or []
        self.parent = parent
        self.set_children(children)
    def set_children(self, children):
        self.children = children or []
        for child in self.children:
            child.parent = self
    def is_leaf(self):
        return len(self.children) == 0
    def contains_key(self, key):
        return key in self.keys
    def get_value(self, key):
        for i, k in enumerate(self.keys):
            if k == key:
                return self.values[i]
        return None
    def get_insert_index(self, key):
        return bisect.bisect(self.keys, key)
    def insert_entry(self, key, value):
        insert_index = self.get_insert_index(key)
        self.keys.insert(insert_index, key)
        self.values.insert(insert_index, value)
        return insert_index
    def split(self):
        if self.parent is None:
            return self.split_no_parent()
        return self.split_with_parent()
    def split_no_parent(self):
        split_index = len(self) // 2
        key_to_move_up = self.keys[split_index]
        value_to_move_up = self.values[split_index]
        # Create right node
        right_node = Node(
            self.keys[split_index+1:],
```

```
            self.values[split_index+1:],
            self.children[split_index+1:]
        )
        # Update left node (self)
        self.keys = self.keys[:split_index]
        self.values = self.values[:split_index]
        self.children = self.children[:split_index+1]
        # Create parent
        parent = Node([key_to_move_up], [value_to_move_up], [self, right_node])
        return parent
    def insert_child(self, insert_index, child):
        self.children.insert(insert_index, child)
        child.parent = self
    def split_with_parent(self):
        split_index = len(self) // 2
        key_to_move_up = self.keys[split_index]
        value_to_move_up = self.values[split_index]
        # Create right node
        right_node = Node(
            self.keys[split_index+1:],
            self.values[split_index+1:],
            self.children[split_index+1:]
        )
        # Update left node (self)
        self.keys = self.keys[:split_index]
        self.values = self.values[:split_index]
        self.children = self.children[:split_index+1]
        # Add new child to parent
        insert_index = self.parent.insert_entry(key_to_move_up, value_to_move_up)
        self.parent.insert_child(insert_index + 1, right_node)
        return self.parent
    def __len__(self):
        return len(self.values)
```

Concepts

- A B-tree is a special type of search tree in which nodes can have more than two children.
- We can use the binary search algorithm to insert an entry into a node while preserving its sorted property in the keys. This is more efficient than inserting the entry and then sorting the list.
- The `bisect` Python module provides an implementation of the binary search algorithm. It can find in *O(log(n))* the correct index of the insertion of a key in a sorted list.
- Nodes in B-tree keep several entries. Special split operations allow us to split the nodes into two nodes with the objective of keeping the maximum number of entries in any given node below a given threshold. This, in turn, keeps the insertion of entries inside nodes a constant time operation, *O(1)*.

Resources

- B-Tree