Exploring the use of B+ trees in Database Management Systems

Chris Aring, Athmika Senthilkumar, Prava Dhulipalla

April 2019

1 Introduction

B+ tree is a data structure that can be used in database management systems to make the process of performing CURD (create, update, read, delete) operations more efficient, especially when large blocks of data need to be accessed from discs. B+ trees serve as a means of implementing multilevel indexing. Indexing is more efficient in B+ trees compared to using other data structures like hashmaps or arrays because of two main reasons. B+ trees is a data structure that has sorted nodes so it provides better time efficiency for searching for keys in comparison to unsorted data structures. Secondly, each node has multiple keys so when one memory call is made, multiple keys can be cached, improving time efficiency by reducing the number of calls to disk.

Since B+ trees are simply an extension of B trees to understand B+ trees we need to first explore the functioning of B trees.

1.1 What is a B-tree?

A B-tree is a self balancing, binary search tree structure in which each node can have multiple children. The degree, d, of a B-tree varies depending on block size, pointer size, and attribute size. For a B-tree of degree d, each node can have at most $2d - 1$ keys, and every node (expect for the root node) must have at least $d - 1$ keys.

In addition, to the ones listed above, there are certain properties, that constraint and define a B-tree:

- The keys in a B-tree are always sorted.
- Every leaf of a B tree lies in the same level.
- If a node has d keys, then it has $d + 1$ children.
- A child between key-1 and key-2 will have keys greater than key-1 and less than key-2

Because of the way B-trees are structured in a sorted order operations like searching, inserting etc. can be performed in O(log n) time, making them efficient for CURD operations.

1.2 What is a B+ tree?

A B+ tree is an extension of a B tree, constrained by the same proprieties outlined before. Unlike in B-trees, in which data can be stored in any node, in B+ trees data can only be stored in the leaf nodes. The leaf nodes are sometimes linked together, usually in a linked list, in order to make traversing data easier. However, because data is stored only in the leaf nodes, B+ trees have redundant keys. The figure below is an example of a B+ tree [1].

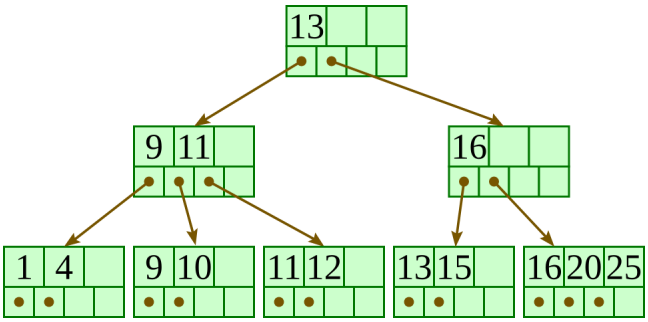


Figure 1: B+ tree

Notice, that all the internal nodes have the key values repeated in the leaf nodes. The dots shown in the leaf nodes represent data pointers that point to records in disks. Since the data pointers are not present in internal nodes, more key values can be fit into a single page in memory. This is one of the principal advantages of B+ trees over B trees. Another important advantage, is the time efficiency of scanning through all keys is better in B+ trees. If all the leaf nodes are linked together, then the scanning through all the key values takes only linear time in a B+ tree.

2 B+ Tree Degree

The degree of a B+ tree can be found using the following formula:
$Kd + P(d + 1) =< N$. Where K represents the key size, d is the degree, P is the pointer size, and N is the node (page) size.

[1] Source: http://www.cburch.com/cs/340/reading/btree/index.html

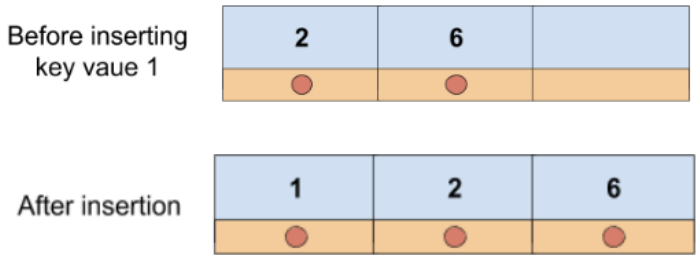
3 Searching algorithm

Principles of binary search can be applied to B+ trees in order to find a key value in O(log n) time. If performing a binary search locates a key that lies in a leaf node then the data with the key can be directly retrieved using the data pointer. However, if the binary search locates a key that lies in any of the internal nodes, that branch has to be traversed further to locate the same key in the leaf nodes.

4 Insertion algorithm

The first step before carrying out an insertion operation is locating the leaf node to which the key that is to be inserted in belongs to by performing a search. After that, depending on the state of that located node the procedure to insert varies.

- The simplest procedure to insert a key into a node occurs when the node still has space for another key. In which case, the key along with its associated data pointer should be inserted in a spot such that the order of the keys in the node is still maintained. For example, suppose the key to be inserted in the simple tree shown below is 1, then the existing keys in the node needs to be pushed one step to the right, and the new key will occupy the first index in the node. Note, that the maximum size of the node depends on the degree of the B+ tree. In the example shown above the tree has a degree of 4 so the maximum keys a single node can hold is 3.



- When a key needs to be inserted in a node that is already full, the node has to be split into two. The key values should be evenly distributed between the two nodes. At this point, the minimum value of the second node, that is the median key value of the node that was split, must be copied and inserted into the parent node. If the parent node is already

full, then this process has to be recursively repeated. When the node that is being split is not a leaf node, then the median value should be promoted to be inserted in the parent node without making a copy of the key value in the two nodes that resulted from the split. The figures below shown an example of such an insertion.

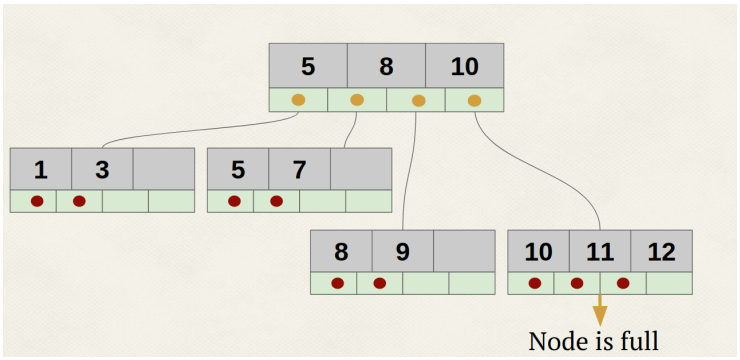


Figure 3: Before inserting 13

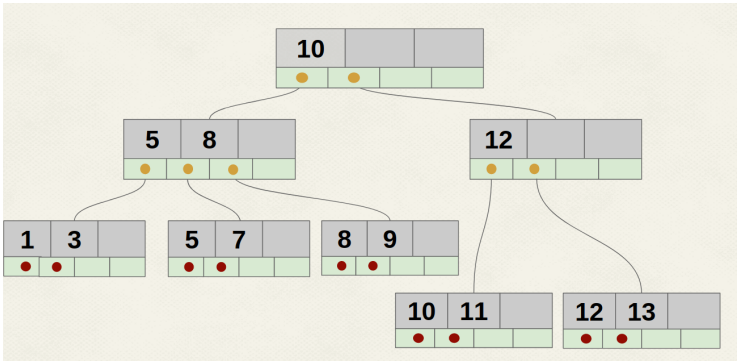


Figure 4: After inserting 13

5 Deletion algorithm

When deleting a key from a node does not violate any of the properties of the B+ trees, then the deletion process is simple. The key and its associated data pointer should be removed from the node, and the values in the node should be shifted to avoid intermediate null spaces in the node. If the deleted key appears in both a leaf node and an internal node, after deleting the key in the leaf node, the key in the internal node must also be deleted. When deleting a key leads to a scenario in which the number of keys in the node falls below $d/2$, keys need to be redistributed or nodes need to be merged such that no node (possibly

except for the root) has less than $d/2$ keys. Keys should be redistributed if it is possible to borrow from a sibling node, else nodes should be merged. These cases can be explained in more detail by looking at them individually:

- **Redistribution**: Redistribution is only possible if the right or left sibling of the node from which the key is being deleted has $d/2 + 1$ keys or more. A key from such a sibling node can be redistributed. The parent of the node has to updated in order to reflect this redistribution.
- **Merging**: If neither of the siblings have sufficient keys to allow for a redistribution, sibling nodes must be merged. After the merge, we must inspect the parent node to remove the extra key value that was previously separating the two nodes. If two internal nodes are merged, the key value in the parent around which the split happened needs to be included in the appropriate location in the newly merged node.

6 Our implementation

We implemented a B+ tree model in python that abstracts the functioning of a B+ tree. We implemented two classes, the B+ node class and the B+ tree class. Since we are implementing the data structure in Python, instead of using pointers from one node to the other, we used arrays to keep track of the keys and the pointers that they link to. Additionally, instead of storing and retrieving data from disks, the abstracted model we implemented stores data in an array. This method of modelling the data structure enabled us to focus on the functioning of the algorithms instead of being hindered by python specific details of storing and retrieving data from disks. We also focused on one specific degree of B+ tree.

7 Conclusion

Learning about B+ trees furthered our knowledge of how indexing can be efficiently implemented in the context of databases. Our implementation focuses on some of the key functions of B+ trees. A good way of extending this project would be to study all the edge cases and perform rigorous testing. Another way of extending this project is implementing a model in which the user can modify the degree of B+ tree, instead of implementing a specific degree of B+ tree.

8 Resources

- "B Tree - Javatpoint." Www.javatpoint.com, www.javatpoint.com/b-plus-tree.
- "Database File Indexing - B Tree (Introduction)." GeeksforGeeks, 4 May 2019, www.geeksforgeeks.org/database-file-indexing-b-tree-introduction/.

- "B -Trees." CSci 340: B -Trees, www.cburch.com/cs/340/reading/btree/index.html.
- "Python Implementation of a B Tree." Gist, gist.github.com/savarin/69acd246302567395f65ad6b97ee503d.
- "Part 7 - Introduction to the B-Tree." Let's Build a Simple Database, 23 Sept. 2017, cstack.github.io/db$_t$*utorial/parts/part*7.*html*.