Proceedings of the 2nd International Conference on Computing Innovation and Applied Physics
DOI: 10.54254/2753-8818/18/20230348

Concurrency control in tree data structures

Xinyang Du[1, 3], Yingzhe Liu[2, *]

[1]School of Electronic Engineering and Computer, Queen Mary University of London, London, E1 2FX, United Kingdom
[2]School of Letters and Science, University of California at Davis, Davis, California, 95616, USA

[3]x.du@se20.qmul.ac.uk

[*]yizliu@ucdavis.edu

Abstract. With the increasing amount of data, there is a higher demand for fast access to data. Therefore, tree data structures are popular because they can quickly access data. In addition, with the increase of the cores of computer microprocessors, the tree data structure that can be concurrently controlled may be one of the most effective data structures today. However, there is still a lack of a summary of the differences in the data structures of various trees. Therefore, this paper collects code structures of different types of trees and then comments on the speed of various tree data structure types concerning parallel processing situations. The study covers the structure and performance of different kinds of trees and summarizes the method of concurrent control of these data structures. This paper can help to clearly understand the relationships and differences between various tree data structures and help them quickly learn the application of concurrency control in trees.

Keywords: binary trees, B trees, 2-3 trees, B+ trees, concurrency control.

1. Introduction

Tree data structures are crucial for all types of computer software. As the demand for fast access to large amounts of data increases, so does the technology for data storage. Therefore, to achieve the purpose of fetching data in a short period, many data structures are developed. Among them, trees are the most efficient ones [1]. There are extensive real-life applications of tree data structures, mainly including modern search engines, databases, game development, machine learning, etc. For example, modern search engines provide potential questions that the users may input. Specifically, for instance, when a user searches for "why does coronavirus…," Google automatically completes the question potentially as "why does coronavirus cause headaches," "why does coronavirus affect taste and smell," and "why does coronavirus mutate so much" as in Figure 1. Behind the scenes, Google uses the suffix tree for auto-completion. It abstracts each word to a node in the tree, so the word "why" becomes the root. Following the path that contains "why," "does," and "coronavirus," the search engine returns all children of the node that denotes the word "coronavirus" to auto-complete the question.

DOI: 10.54254/2753-8818/18/20230348

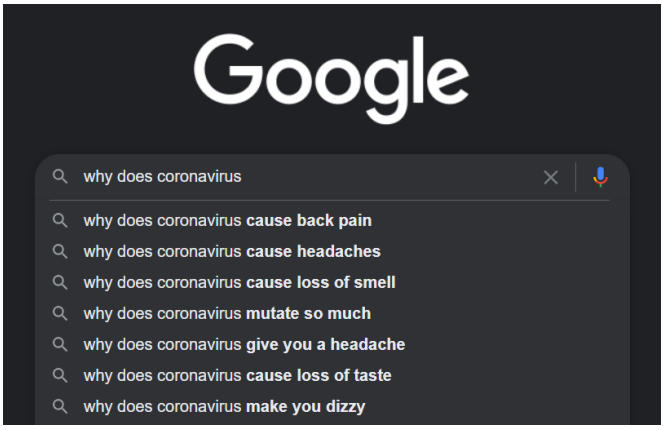

Figure 1. Google search auto-completion.

Besides search engines, tree structures are also employed by databases. The database MySQL uses a B+ tree to query for its data [2]. This tree inserts and searches for data in O(log N) time complexity, which satisfies the demand of its users to manipulate data efficiently. In addition, in game development, to simulate physics and to detect object collisions, game engines utilize binary trees to partition in-game space into small regions [3] such that in each frame, the engine only checks the region that objects are in and iterate through neighboring regions to test for collisions. In such a way, the game engine omits redundant collision checks to enable object detection in real time. Finally, tree data structures are also used in machine learning. To make an optimal decision, computers simulate the process of decision-making by evaluating a sequence of factors in the decision tree. Each internal node denotes the factor that is taken into consideration. Each leaf node is the decision. So, given the factors, the machine follows the path that has the factors to produce a decision. The above examples are sufficient to show that tree structures are important to study.

The above examples of implementations of tree structures in real life show how important they are and running them parallelly can further increase their efficiency. As modern CPUs and other processing units have more than one core, the additional cores can boost processing speed if they run threads that parallelly process the same task with other cores. In the case of auto-completion, multiple users providing their questions update the tree parallelly. The suffix tree thus needs to schedule the concurrent updates efficiently to maintain its structure and correctness. Also, in the game engine, multiple players are likely to be active at the same time, and the same tree that checks for collision may be accessed by both players concurrently. The binary search tree utilized by the game engine needs to ensure that the multiple accesses to the tree are correct even after the tree updates.

Therefore, to accommodate the need to modify and access tree structures concurrently, researchers have done extensive work to increase the efficiency of operations. Various concurrent control methods have been proposed that perform correct read operations: when a node is being written, it cannot be read, and otherwise, it must be successfully read. [4] The methods also need to maintain correct write operations: two write operations cannot modify the same node at the same time. In addition, there can be no deadlocks [4].

In this paper, we summarize several concurrent policies for the binary search tree, the multiway trees, and the B+ tree. The properties of the above-mentioned trees are thoroughly discussed, and their concurrent policies are analyzed. Finally, the search speed of the various concurrent tree structures is compared to test for efficiency. For the binary search tree, we introduce concurrency control in two types of trees: the unbalanced binary search tree and the Adelson-Velsky and Landis (AVL) tree. Later in the multiway trees section, we provide the definition and concepts of the multiway trees and then dive into the parallel methods of the two variants of the multiway tree: the 2-3 tree and the B tree. Finally, we illustrate the locking methods of the B+ tree and the B-link tree in the last section.

DOI: 10.54254/2753-8818/18/20230348

2. Concurrent binary search tree

Before analyzing the concurrency control methods of the binary search tree, we first provide the properties and algorithms of the binary search tree that is not concurrent (Sec. 2.1). Then, we explain how to build the concurrent unbalanced binary search tree (Sec. 2.2) and the concurrent AVL tree (Sec. 2.3).

2.1. Basic concepts of the binary search tree

A binary search tree is a type of binary tree, and a binary tree (without "search") is a tree in which each node has at most two children, a key, and a value [5]. A node in the binary tree that has no child is called a leaf. The binary tree itself has no significant use in computer software since it does not preserve some kind of ordering of the nodes, but the binary search tree is much more useful for its following properties. The left subtree of a node contains keys less than the node, and the right subtree contains keys greater than the node [6]. In addition, the two subtrees of a given node are also binary search trees. Figure 2 and 3 show an example that highlights the differences between the binary tree and the binary search tree (see the ordering of the nodes).

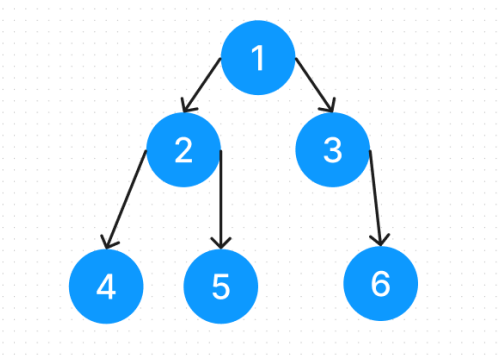

Figure 2. Binary tree.

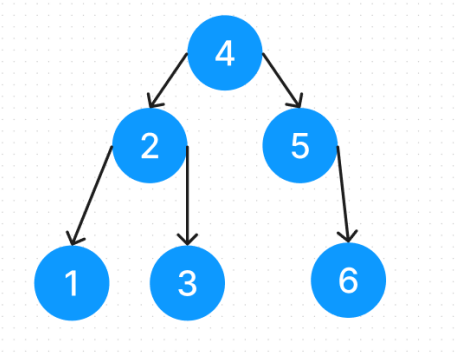

Figure 3. Binary search tree.

To read a value from the binary search tree is simple. Start from the root of the tree, if the target key is less than the key of the current node, go to the left child, and if the target key is greater than the key of the current node, go to the right child. This process is repeated until the target key is found or reaches a leaf node, and in that case, the target key does not exist in the tree. Algorithm 1 shows the details of the procedure of the binary search tree.

Table 1. Algorithm 1: Pseudocode of reading in the binary search tree.

Input: Target key and current node
Output: Value corresponding to the target key or null

```
function find_key(target, curr)
    if curr is null
        return null
    end if
    if target == curr.key
        return curr.value
    end if
    if target < curr.key
        return find_key(target, curr.left)
    end if
    return find_key(target, curr.right)
end function
```

DOI: 10.54254/2753-8818/18/20230348

To write a value in the binary search tree is much harder since many modern implementations of the binary search tree are self-balanced, and the balancing process is complicated. In section 2.2, the concurrent control of the writing process and the process itself are thoroughly discussed, but here is the general algorithm to write a value in the tree, shown in Algorithm 2. It is similar to the reading process, but after writing the target key and value into the tree, there is a self-balancing process for fast searching of elements in the tree. However, that process can be omitted if speed is not of concern.

Table 2. Algorithm 2: Pseudocode of writing in the binary search tree.

```
Input: Target key, target value, and the current node
Output: Boolean value indicating if the writing is successful
1: function write_pair(key, value, curr)
2:     if curr is null
3:         construct new node and set curr to the new node
4:         self_balance()  // various forms of self-balancing
5:         return true
6:     end if
7:     if key == curr.key
8:         return false // key exists in the tree
9:     end if
10:    if target < curr.key
11:        return write_pair(key, value, curr.left)
12:    end if
13:    return write_pair(key, value, curr.right)
14: end function
```

2.2. Basic unbalanced concurrent binary search tree

To introduce the concurrency operations into the binary search tree, the easiest way is to lock the whole tree while reading (Algorithm 1) and writing (Algorithm 2). For the reading operation, it is achieved by locking the read lock before executing the find_key function and unlocking it after the execution [7]. Similar to the writing operation, wrapping the write lock before and after executing the write_pair function can do the job [7]. This way of concurrency control satisfies the read rules and the write rules. When a node is being written, it cannot be read since the write operation write-locks the whole tree and blocks all read operations. Also, when a node is not being written, it can be successfully read, but it needs to wait for the read lock to unlock if there are writing processes happening. For the writing process, two threads cannot modify the same node at the same time because one of the threads must have acquired the write lock before the other. There also cannot be any deadlocks by design. However, such a locking mechanism is slow: this is a naïve approach for the concurrency control of the binary search tree, and it is not efficient enough when writings occur. Notice that the write lock is locked whenever the writing operation happens. So, in such cases, all reading processes and other writing processes are blocked, not utilizing the computing resources fully.

Table 3. Algorithm 3**:** Pseudocode of concurrent reading in the binary search tree.

```
Input: Target key and current node
Output: Value corresponding to the target key or null
1: function find_key_concurrent(target, curr)
2:     read_lock()
3:     let res = find_key(target, curr)
4:     read_unlock()
5:     return res
6: end function
```

DOI: 10.54254/2753-8818/18/20230348

Table 4. Algorithm 4: Pseudocode of concurrent writing in the binary search tree.

Input: Target key, target value, and the current node
Output: Boolean value indicating if the writing is successful
1: **function** write_pair_concurrent(key, value, curr)
2: write_lock()
3: **let** res = write_pair(key, value, curr)
4: write_unlock()
5: **return** res
6: **end function**

2.3. Concurrent AVL tree

Compared to the traditional binary search tree, the Adelson-Velsky and Landis (AVL) tree self-balances when the heights of the two child subtrees differ by 2 or more. In this way, the time complexity of insertion, deletion, and searching of a node is maintained at O(log N), where N is the total number of nodes in the tree. When inserting the keys into a binary search tree is ordered, the binary search tree becomes unbalanced. In such cases, the time complexity that it takes to reach a node deteriorates to O(N). The AVL tree solves this problem through self-balancing.

To stay balanced, the AVL tree rotates itself from the node which has its height difference of the left and right subtrees greater or equal to 2. In this process, there are four possibilities of rotations [8]. The detailed process of rotation can be found here [8]. The four possibilities are left rotation, right rotation, left-right rotation, and right-left rotation. In the latter two rotations, there are two steps performed while in the first two rotations, only one step is performed. These rotations add complexities to the locking control mechanism because it modifies many nodes in the tree. Here, we discuss one common method of locking the AVL tree.

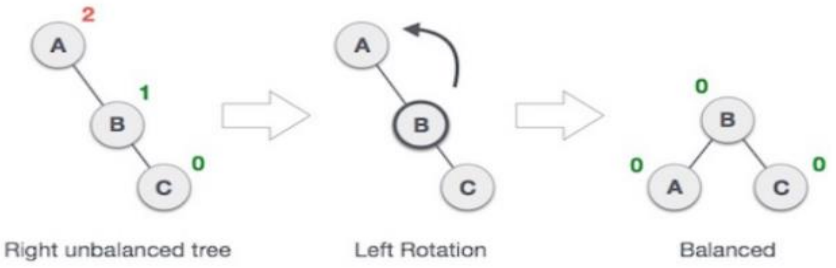

Figure 4. Left rotation.

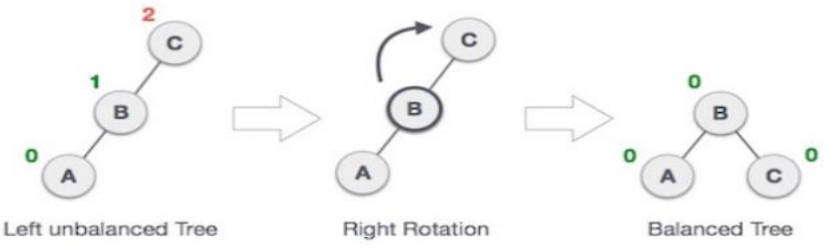

Figure 5. Right rotation.

First, during the searching period for the writing process, the tree is not modified. So, the reading process can access the same nodes as the writing process does, and a read lock is used to make sure that other writing processes do not modify the nodes being read. The algorithm of the reading process is not a recursive function like that in the naïve method because locking happens within the function. It

DOI: 10.54254/2753-8818/18/20230348

locks the path from the root to the target node while releasing previously locked nodes when they are no longer in use as in Algorithm 5.

Table 5. Algorithm 5: Pseudocode of reading in the AVL tree.

```
Input: Target key and current node
Output: Value corresponding to the target key or null
1: function find_key_AVL(target, curr)
2:     read_lock(curr)
3:     let son = curr
4:     while son != null and target != son.key
5:         read_lock(son)
6:         read_unlock(curr)
7:         curr = son
8:         /* determine appropriate son */
9:         if target < curr.key
10:            son = curr.left
11:        else son = curr.right
12:        end if
13:    end while
14:    read_unlock(current)
15:    if son is null
16:        return null
17:    return son.value
18: end function
```

The writing process is much more complicated because of the rotations. Two types of locks are used, write lock and exclusive lock. Write lock excludes other writing processes from modifying the nodes while searching for the place to insert. After the place is found, the writing process inserts the target node and adjusts the height field used for the re-balancing of the parent of the target node. Then, the writing process evaluates the necessity of re-balancing by looking into the height difference between the left and right subtrees. If it is necessary to rotate, an exclusive lock is used to exclude both other reading processes and other writing processes to maintain correctness.

This concurrent method of AVL tree is faster than the naïve approach because of three reasons. First, the AVL tree itself is faster than the unbalanced version of the binary search tree. In addition, the reading process is faster because it unlocks the nodes that are no longer in use while the naïve approach does not do that. Also, the writing process only uses the exclusive lock when there are rotations, minimizing the usage of the least parallel locking mechanism.

3. Concurrent B trees

In this chapter, we describe the structure of multiway trees (Sec.3.1), then provide the properties and algorithms of 2-3 trees (Sec.3.2) and B-tree (Sec.3.3).

3.1. Multiway trees

Although the operating efficiency of binary trees is relatively high, it also has problems. As we know, a full binary tree has n-1 nodes. To use a binary tree, data are needed to be loaded into memory to build a binary tree. When there are fewer nodes in the binary tree, this will not expose any problems. However, if there are many nodes in the binary tree, to read the data stored in the document or database into the memory, the system needs to perform a large number of I / O operations, which will affect the operation speed. In addition, since the binary tree has only left and right child nodes and one

DOI: 10.54254/2753-8818/18/20230348

node can only have one data when the number of nodes of the binary tree is large, the binary tree will have a high height and the operation speed will also be reduced.

To solve this problem, a data structure named multiway tree is created, which allows each node to have more data elements and child nodes than binary trees. B-trees, a data structure used to organize dynamic files [9], are one of the most famous multiway trees. B-trees can reduce the height of the tree data structure by reorganizing nodes so that the number of I/O operations can be decreased and the efficiency will increase. As shown in Figure 6, each circle represents a data item.

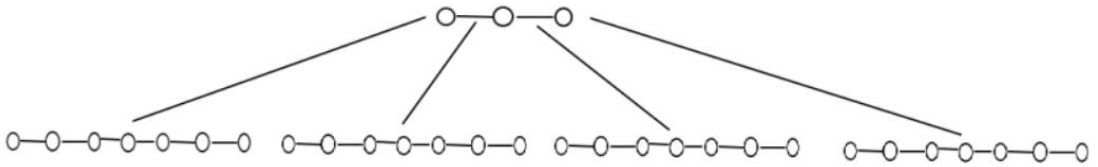

Figure 6. Multiway tree.

Besides, the designer of file systems and database systems generally uses the principle of disk read-ahead. They set the size of a node to the size of a page (The size of a page is usually 4K). So that each node needs only one I / O operation to load.

3.2. 2-3 trees

2-3 trees are the simplest B-trees data structure and each node in 2-3 trees has 2 or 3 child nodes. The node that has 2 sons has named 2 nodes and the node that has 3 sons is named 3 nodes. If a 3 node has two numbers k1 and k2, the number of its left child node should smaller than k1, the number of intermediate child node should be between k1 and k2, and the right child node's number should be larger than k2, which shown in Figure 7. Because the nodes of 2-3 trees are small, it is very suitable for internal data structure, but not for external storage [10].

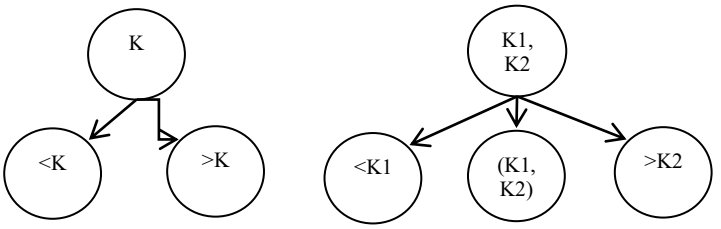

Figure 7. 2-3 trees.

When inserting a number to a node of 2-3 trees, it should satisfy the following rules. Firstly, just like all B-trees, all leaf nodes of 2-3 trees are on the same layer; for every 2 nodes, either there are no child nodes or there are two child nodes, and for every 3 nodes, either there are no child nodes or there are three child nodes.

If the insert operation makes 2-3 trees fail to meet the above three conditions, it should move the middle element to the parent and split the current node. If the parent node is full, it should split the parent node into one parent node with two sons.

In addition to 2-3 trees, there are 2-3-4 trees. The concept is similar to 2-3 trees, and it is also a B-tree. As shown in the figure:

DOI: 10.54254/2753-8818/18/20230348

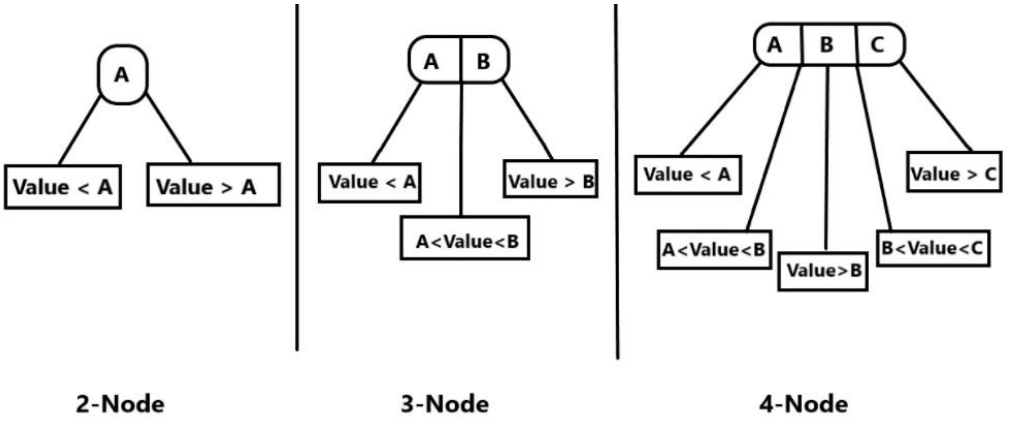

Figure 8. 2-3-4 trees.

For 2-3 trees concurrency control, Ellis [11] claimed it is useful that allows concurrent readers to share nodes with writers in the reorganization phase and the search phase based on the third protocol of Bayer and Schkolnick. It also applied the technology proposed by Lamport [12] to improve concurrency. That is, when the reader process and the writer process access the same node, the reader scans the node from top to bottom, reads the links and labels in the node from left to right, and the writer reconfigures from bottom to top, changing the links and labels from right to left.

3.3. B-trees

Although there is no unified explanation for B in B-tree, the most widespread view is that it is balanced. Since the number of disk accesses increases with the increase of the height of the tree, the B-tree, as a balanced search tree specially designed to be stored on the disk, can always maintain a low height [13]. The B-tree is defined by the minimum degree "t" depending on the disk block size. The root node can contain at least 1 key, while other nodes must contain at least T-1 keys. In addition, all nodes can contain up to 2 t –1 key. The number of child nodes of a node is equal to the number of keys plus 1.

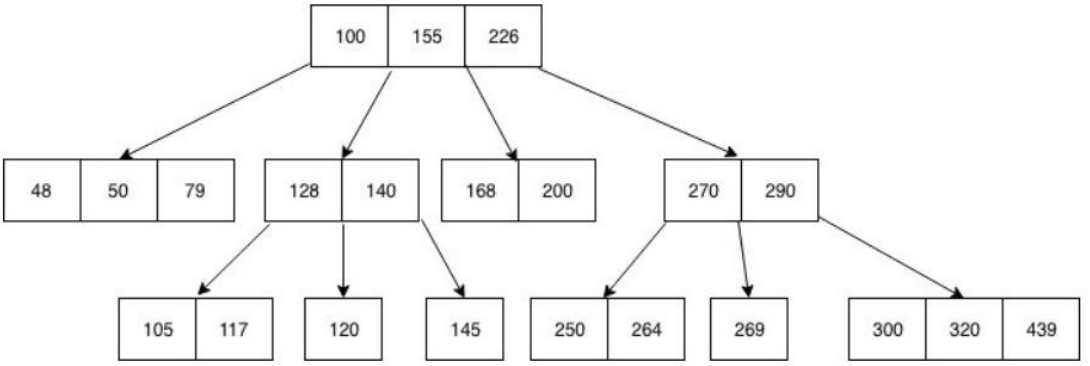

Figure 9. B-trees.

The search operation for a value from B-tree is similar to the search in the Binary trees. Set the key value to be searched as K. The algorithm will start from the root node and iterate down recursively. For each reached non-leaf node, if the node has the key that is searched, the node needs to be returned. Otherwise, continue recursion to the appropriate child node of the node. If the leaf node is reached, but K is not found in the leaf node, null is returned.

In the insert operation of B-trees, if the tree is empty, the algorithm will assign a root node is assigned and insert the key is inserted, then update the number of keys allowed in the node, and search

DOI: 10.54254/2753-8818/18/20230348

for the appropriate node for insertion. If the node is full: insert elements in ascending order. Since the number of inserted elements is greater than its limit, they are separated in the median. After that, move the median key upward, with the left key as the left sub key and the right key as the right sub key; If the node is not full: the nodes are inserted in ascending order. To concurrently control B-trees, Samadi [14] claimed that it is possible to lock the entire path exclusively using semaphores because any given modification to the tree can change. This effectively locks the entire subtree of the highest affected node.

4. Concurrent B+trees

In this chapter, this paper will discuss the structure and algorithms of B+ Trees and B-link Trees. And then talk about their concurrent control methods.

4.1. B+ trees

B+ tree evolved from B-tree and index sequential access method. The B+ tree is a balanced search tree designed for disks or other direct access auxiliary devices. In the B+ tree, all record nodes are stored in the leaf nodes of the same layer in the order of key values, and each leaf node pointer is connected. The structure of the B+ Tree is shown in Figure10.

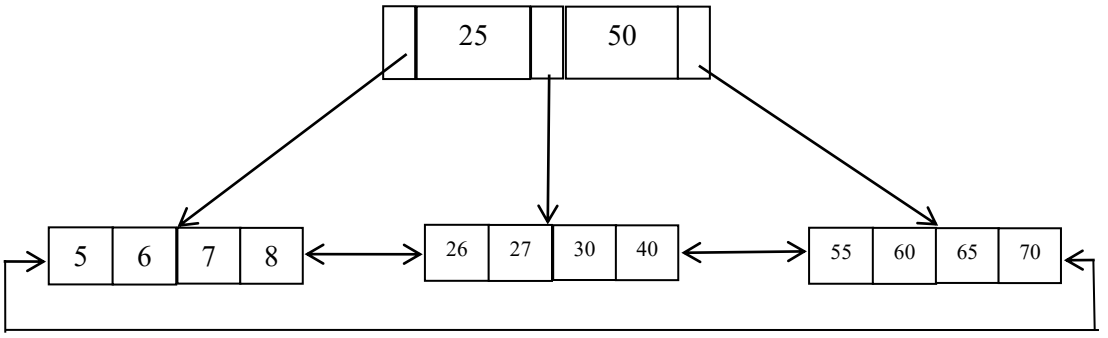

Figure 10. B+ trees.

The insertion of the B+ tree must ensure that the records in the leaf node are still sorted after the insertion. There are three cases of inserting a B+ tree that needs to be considered when the system executes the insert operation. Firstly, if both the leaf page and index page are not full, it can insert the value into the leaf node directly. Secondly, if the leaf page is full but the index page is not full, the leaf page should be split and put the node in the middle into the index page. Put the value that is smaller than the middle node on left. If the value is larger than the middle node, put it into right. Thirdly, if both the leaf page and index page are full, the index page should be split after splitting of leaf page. Then the value is put that is smaller than the middle node into the left or put the value that is larger than the middle node into the right, just like what to do after the leaf page split. Finally, the middle node is put into the previous index page.

4.2. Traditional concurrent B+ tree

Here, we discuss the traditional concurrent control of the B+ tree. It is fairly simple. The reading process resembles that of the binary search tree: it traverses the tree from the root to the target leaf, locking one node while unlocking the last node traversed until it reaches the target node. However, there are minor differences. Since one node in the B+ tree contains multiple children (greater than 2 most of the time), just looking into the left child or the right child is not enough. To find the correct child for the next level, the reading process needs to compare the target to the key of each child until it finds a child that is greater than the target [15]. If such a child does not exist, it goes to the rightmost

DOI: 10.54254/2753-8818/18/20230348

child. After reaching a leaf node, the reading process finds the target in the leaf or could not find it since it is not in the tree.

The writing process shares similar ideas to the reading process. It traverses from the root to the leaf, but the locking and unlocking methods it uses are different from those the reading process uses. The writing process utilizes a stack to track the previously locked nodes. For each level down the tree, if the current node is safe (the number of children it contains is less than the maximum capacity), the writing process pops all nodes in the stack and unlocks them. Else, they are left in the stack and stay locked because later insertion of the target may trigger a split of the leaf node, which may in turn trigger multiple splits of its parents. So, the parents stay locked. After finding the correct leaf node to insert the target, the writing process modifies the leaf, manages all splits of the leaf node and its parents, and unlocks all previously locked nodes. A detailed writing process algorithm is shown in Algorithm 6.

Table 6. Algorithm 6**:** Pseudocode of writing in the traditional concurrent B+ tree.

```
Input: Target key, value, and the current node
Output: Boolean value indicating if the writing is successful
1: function write_pair_bplus(key, value, curr)
2:     write_lock(curr)
3:     while curr is not leaf
4:         write_lock(curr.child)
5:         curr = current.child
6:         if curr is safe
7:             write_unlock(locked ancestors on stack)
8:         end if
9:     end while
10:    modify the leaf and manage the splits and return false if key is already in the leaf
11:    write_unlock(curr) and write_unlock(locked ancestors on stack)
12:    return true
13: end function
```

The traditional concurrent B+ tree utilizes read and write locks for concurrency control. It has the problem that when there is any number of writing operations on one write-locked node, no reading of that node can happen. It significantly slows down the reading speed of the B+ tree, and later improvements can solve this problem.

4.3. B-link tree

Instead of using the read locks in the reading process, the improvements introduced by the B-link tree turn the process lock-free [16]. The addition of the link pointer and the high key in each node achieves that purpose. In the addition, although the writing process may change the shape of the tree, the reading process does not need any locks. The reason is that when a node splits, the reading process compares the target with the high key (maximum key a node can contain). If the target is greater than or equal to the high key, the reading process just follows the next pointer (pointer to the next node on the same level) and searches for the target in the next node. Else, the current node contains the target, and the reading process works like before. This clever design solves the problem of parallel reading while there is writing. In Figure 11, a B-link tree node is shown where K2k+1 is the high key, and the rightmost arrow is the next pointer.

DOI: 10.54254/2753-8818/18/20230348

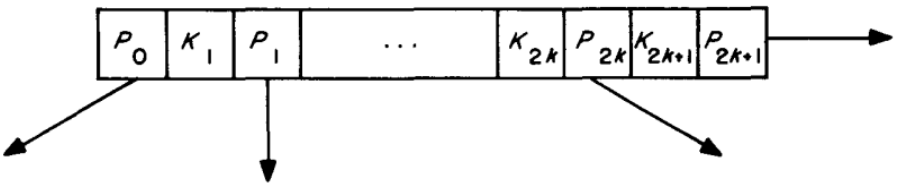

Figure 11. A B-link tree node.

The writing process is almost the same as that of the traditional concurrent B+ tree. The only difference is that instead of using the write lock, it uses the exclusive lock to prevent other writing processes from modifying the mutually exclusive zone.

5. Conclusion

In this paper, we summarize many concurrency control algorithms in different tree data structures. We first illustrate the concurrency control methods in the most well-known tree structure: the binary search tree. Both the unbalanced and the balanced binary search tree are explained in their properties and concepts. We provide concurrent algorithms for the two types of binary search trees and explain the reason for the increased efficiency in the AVL tree. Then, we cover the multiway trees. They sustain lower rates of I/O by containing more than two key-value pairs in one node. We also explain the concurrency control algorithms of the multiway trees in detail. Finally, we comment on the B+ trees, which many common databases use. The traditional concurrent B+ tree is covered and explained of its problem that slows it down in the situation where multiple reading happens while writing also happens. Later, we explain that the B-link tree solves this problem by turning the reading process lock-free through the addition of the high key and the next pointer in each node.

The summary of the concurrency control algorithms in tree data structures helps to learn the differences between the tree structures and quickly grasp the common algorithms in the concurrent versions of those structures. It also helps to choose the right tree structure for the right situations by viewing the properties and concepts of the structures. Researchers in the field can also get inspiration from this summary and further optimize the algorithms.

The past researches in concurrency control of tree data structures already provide many efficient ways to manipulate the structures in parallel situations. However, as the amount of data transmitted and accessed by people keeps increasing, algorithms with higher efficiency are needed. It is beneficial for future research to investigate ways to increase the efficiency of the concurrent tree structures even more.

References

[1] Augenstein, M., Tenenbaum, A. (1977) Program efficiency and data structures. In: ACM SIGCSE Bulletin. New York. pp. 21-27.
[2] MySQL. (2022) The physical structure of an InnoDB index. https://dev.mysql.com/doc/refman/8.0/en/innodb-physical-structure.html.
[3] Baeldung. (2021) Real world examples of tree structures. https://www.baeldung.com/cs/tree-examples.
[4] Hoare, C. A. R., Hayes, I. J., He, J., Morgan, C. C., Roscoe, A. W., Sanders, J. W., Sorensen, I. H., Spivey, J. M., Sufrin, B. A. (1987) Laws of programming. Commun. ACM, 30(8): 672-686.
[5] Parlante, N. (2001) Binary Trees. http://cslibrary.stanford.edu/110/BinaryTrees.html.
[6] Berman, A. M., Duvall, R. C. (1996) Thinking about binary trees in an object-oriented world. In: ACM SIGCSE '96. New York. pp. 185-189.
[7] Luedtke, D. (2018) A concurrency-safe, iterable binary search tree. https://danrl.com/btree/.
[8] Ellis, C. (1980) Concurrent search and insertion in AVL trees. In: IEEE Transactions on

DOI: 10.54254/2753-8818/18/20230348

Computers. pp. 811-817.
[9] Bayer, R., McCreight, R. (1970) Organization And Maintenance of Large Ordered Indices. In: Mathematical and Information Sciences Report No. 20. pp. 107-141.
[10] Comer, D. (1979) The Ubiquitous B-Tree. In: Computing Surveys, Vol 11, No. 2. pp. 121-137
[11] Ellis, C.S. (1978) Concurrent search and insertion in 2–3 trees. In: Acta Information 14. pp. 63-86.
[12] Lamport, L. (1977) Concurrent reading and writing. In: Communications of the Acm. pp. 806-811.
[13] Bača, M., Kuroga, P. (2010) Analysis of B-tree data structure and its usage in computer forensics. In: Proceedings of the 21st Central European Conference on Information and Intelligent Systems. pp. 423-428.
[14] SAMADI, B. (1976) B-trees in a system with multiple users. In: Information Processing Letters. pp. 107-112.
[15] Hellerstein, J., Brewer, E. (2001) Concurrency control and recovery for search trees. https://dsf.berkeley.edu/jmh/cs262b/treeCCR.html
[16] Lehman, P. L., Yao, B. (1981) Efficient locking for concurrent operations on B-trees. In: ACM Transactions on Database Systems. New York. pp. 650-670.