MAPLe: A B$^+$-tree with Multi-Access Parallel Leaves to Improve Access Concurrency and Locality

Luna Wang*
Cupertino High School
Cupertino, CA, USA
lunawang257@gmail.com

Shuaihua Zhao
University of Texas at Arlington
Arlington, TX, USA
sxz6329@mavs.uta.edu

Song Jiang
University of Texas at Arlington
Arlington, TX, USA
song.jiang@uta.edu

Abstract

As one of the most commonly used ordered indexes, the B$^+$-tree offers high access performance with well-bounded tree heights.

One key to the B$^+$-tree's competitiveness in practice lies in its ability to scale its throughput with an increasing number of threads running on different CPU cores. This is especially true for the in-memory B$^+$-tree, where accessing tree nodes is very fast. The bottleneck to this scalability is the use of locks that serialize concurrent access to tree nodes. In this paper, we show that for in-memory B$^+$-trees with write-heavy skewed accesses, the primary source of scalability loss is not the inner nodes but rather the contention at the leaves.

In this paper, we propose an optimized B$^+$-tree named MAPLe (Multi-Access Parallel Leaves) tree to minimize contention by allowing multiple threads to simultaneously insert or delete keys in the same leaf and improving access locality within the leaf. Furthermore, it does not move key-value pairs within the leaf during insertion or deletion, which is advantageous when keys and values are large. The key technique in our design is to introduce a new leaf layout scheme. In a MAPLe leaf, a level of indirection called the Slice is introduced, which makes the leaf resemble a two-level tree, where the "root" points to multiple child slices. In other words, the leaf is partitioned into logically separate slices, each of which can handle lookups, insertions, and deletions concurrently.

We implemented the MAPLe tree in the codebase of the state-of-the-art TLX B$^+$-tree with an optimistic concurrency control scheme. Experimental results show that our MAPLe design can significantly improve scalability for workloads with large values and skewed access patterns. For example, for a balanced 16-thread workload where there are an equal number of lookups, inserts, deletes, and short scans, the MAPLe tree outperforms the B$^+$-tree by up to 158%.

CCS Concepts

• **Information systems → Data management systems**.

*Corresponding author.

SAC '25, March 31-April 4, 2025, Catania, Italy

ACM ISBN 979-8-4007-0629-5/25/03
https://doi.org/10.1145/3672608.3707803

Keywords

Data structure, index, concurrency, B+-tree

ACM Reference Format:
Luna Wang, Shuaihua Zhao, and Song Jiang. 2025. MAPLe: A B$^+$-tree with Multi-Access Parallel Leaves to Improve Access Concurrency and Locality. In *The 40th ACM/SIGAPP Symposium on Applied Computing (SAC '25), March 31-April 4, 2025, Catania, Italy*. ACM, New York, NY, USA, Article 4, 9 pages. https://doi.org/10.1145/3672608.3707803

1 Introduction

Ordered key-value index is a core and performance-critical data structure in various database systems and other data management systems. Among the indexes, the B$^+$-tree is one of the most widely used structures for organizing data on disks and in memory. For on-disk B$^+$-trees, the optimization efforts are focused on reducing disk access and improving the buffer cache management. To meet the demands of many applications and services for very high performance, the buffer cache becomes increasingly large, and more accesses of index and data are carried out in memory. In contrast, in-memory databases keep all their indexes and data in the memory [5, 11, 21]. Accordingly, it is increasingly important to study the in-memory B$^+$-tree, which has been used as an in-memory index in many popular database systems such as PostgreSQL [7], IBM DB2 [10], Microsoft SQL Server [15], and MongoDB [20].

The B$^+$-tree is a self-balancing tree structure whose height is $O(log_B N)$, where B is the tree fanout (or number of children per inner node) and N is the number of keys in the tree. While the height of a B$^+$-tree has a direct impact on the latency to service individual requests, the effectiveness of its concurrency support is critical to its throughput, as well as the scalability of that throughput. In today's database systems, which often process intensive transactions on multi-core or even many-core servers, a scalable throughput under multiple cores is becoming an increasingly relevant metric to assess a system's capability to process its workloads. When there are multiple threads attempting to simultaneously read or write a B$^+$-tree, it must be locked properly to preserve its integrity. However, the use of this concurrency control inevitably serializes some concurrent operations and compromises the throughput.

This paper aims to identify the performance bottleneck of the B$^+$-tree under highly concurrent workloads, and ameliorate it with novel and non-disruptive optimizations. A key finding in this study is that, under skewed access patterns, the bottleneck does not lie in the B$^+$-tree's inner nodes, even though the inner nodes are more frequently accessed. Instead, the bottleneck is at the leaves. To address this issue, we propose a B$^+$-tree design named MAPLe (Multi-Access Parallel Leaves) tree. The MAPLe tree represents a non-disruptive optimization of the B$^+$-tree, meaning it does not

change the structure and self-balancing strategy of the conventional B$^+$-tree. Instead, it restructures the leaf node by organizing data into multiple logically separated arrays called "slices" to enable concurrent accesses for higher scalability and to reduce data movement for insertion and deletion.

In summary, we make a number of contributions in this paper:

- We experimentally identify the performance bottleneck due to lock contention in the B$^+$-tree and analyze the finding that the leaf nodes are more likely to become the bottleneck under skewed workloads.
- We design an optimized B$^+$-tree (MAPLe), in which the leaf nodes are restructured to reduce the contention. MAPLe's design can also handle large key-value (KV) items more efficiently than the B$^+$-tree.
- We evaluate the MAPLe tree with workloads of representative access patterns and compare it with the B$^+$-tree. The experimental results demonstrate that for highly skewed write workloads, the MAPLe tree achieves higher throughput than the B$^+$-tree's in all workloads tested in §5. The MAPLe tree outperforms the B$^+$-tree by up to 158% under skewed workloads with large values.

The rest of the paper is organized as follows. §2 discusses the motivation of this paper by showing evidence that the contention on the B$^+$-tree's leaf pages is the bottleneck under skewed concurrent workloads. §3 gives a detailed description of the design of the MAPLe tree. §4 presents a theoretical analysis of the time complexity and space complexity of the MAPLe tree design. §5 compares the MAPLe tree's performance with the B$^+$-tree's under different workloads to show the advantages of the MAPLe tree. §6 lists previous related work and how the MAPLe tree design differentiates from them. §7 concludes the paper.

2 Pinpointing the Contention Bottleneck

When multiple threads access a B$^+$-tree, a series of nodes from the root (inner nodes) to the leaves are locked and accessed. Since there are much fewer inner nodes than leaf nodes, the inner nodes are locked much more frequently than the leaves and may incur more lock contentions. However, as the inner nodes are read-locked most of the time, whereas the leaves must be write-locked during insertions and deletions, the leaf nodes may bear more lock contentions. In a workload with many insertions and/or deletions, which nodes are more likely to become the performance bottleneck due to lock contention?

To answer this question, we create a "Lock-free-at-Leaves" (LfaL) tree by artificially removing any locks, thus removing contentions, on the leaf nodes, and compare its throughput with that of the B$^+$-tree's with varying numbers of threads. Each experiment inserts 100 million KV items (16 bytes each) into an empty tree. The keys are generated with uniform, Zipfian (α=0.95), and Zipfian (α=0.99) distributions, respectively. Since LfaL trees do not use locks to protect leaves, the KV items in each leaf may no longer be sorted and some keys may be lost during insertions. However, these losses are negligible (less than 0.1% of keys are lost during our experiments) and do not affect the outcome of our comparisons. Furthermore, even when the keys are not perfectly ordered, insertions still take a similar amount of data movements, and so should not have a noticeable impact on the measured throughput.

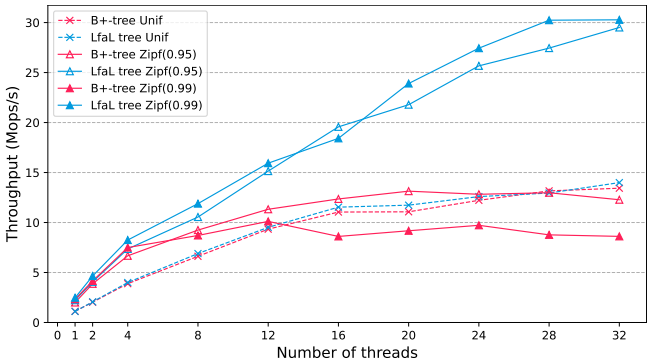

Figure 1: Throughput of B$^+$-tree vs LfaL Tree with Insertions
(One NUMA (Non-Uniform Memory Access) node with 16 physical cores is used. HyperThreading is enabled to allow 32 threads.)

The comparison results are presented in Figure 1. As expected, the throughput of both the B$^+$-tree and LfaL tree under a uniform key distribution are close. Because there are only dozens of threads (limited by CPU core count) accessing tens of thousands of leaves, the chance of two threads accessing the same leaf and causing lock contention is low if the accesses are uniformly distributed.

In contrast, under a workload with strong skewness, there is a much higher chance that the same leaf is accessed by multiple threads at the same time. This would cause high lock contention on the B$^+$-tree but none on the LfaL tree due to its no-lock design. Therefore, the B$^+$-tree's throughput is much lower than that of the LfaL tree and doesn't scale nearly as well. The LfaL tree's performance advantage becomes larger with more threads. When the LfaL tree (artificially) removes the contention, the negative impact of this skewness disappears. In the meantime, the positive impact, which is the higher CPU cache hit ratio, is unlocked to deliver a higher and scalable throughput.

We conclude that under write-heavy workloads with skewed accesses, the B$^+$-tree has poor CPU scalability due to the heavy lock contention at the leaves, because the only difference between the B$^+$-tree and the LfaL tree is that the LfaL tree doesn't have locks on its leaf nodes. Since practical workloads are often skewed and write-heavy workloads are important to handle, we focus on optimizing the B$^+$-tree's performance and scalability under such conditions in this paper.

3 The MAPLe Design

The previous section has found that under write-heavy and skewed workloads, a B$^+$-tree is most likely bottlenecked at the leaves, due to lock contention and slow movements of KV items. This section proposes MAPLe, a B$^+$-tree with Multi-Access Parallel Leaves, to ameliorate this performance bottleneck.

3.1 The Two-level Tree Design of a MAPLe Leaf

A B$^+$-tree leaf typically contains a "slot array" of size *Slots* storing all KV items in order. The MAPLe tree reconstructs the leaf by

building a simplified two-level B$^+$-tree-like structure on top of the slot array. The root node ("slice locator") points to a fixed number of child nodes ("slices"), as shown in Figure 2. Note that for simplicity, only keys are presented in the slot array in the figure.

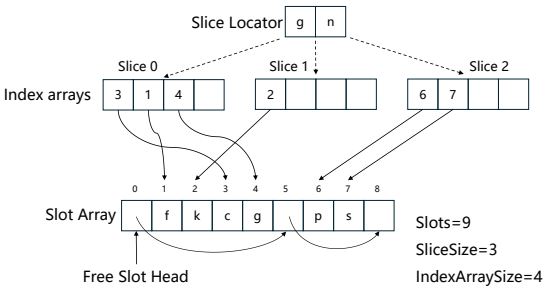

Figure 2: Illustration of the MAPLe Leaf Structure

The slice locator stores "slice boundary keys" in sorted order and identifies which slice contains the required key range. As an example, in Figure 2, the two keys in the slice locator are `'g'` and `'n'`, indicating that any keys `<='g'` are in Slice 0, and any keys `>'g'` and `<='n'` are in Slice 1.

The slices logically partition the KV items stored in the slot array into segments with non-overlapping key ranges. In a full leaf (containing *Slots* KV items), each slice holds *SliceSize* items, where *SliceSize* is a parameter of the MAPLe tree. The number of slices N can be calculated as

$$N = \left\lceil \frac{Slots}{SliceSize} \right\rceil \tag{1}$$

Each slice contains an index array of size *IndexArraySize*. Each item of this array is a 2-byte index pointing to the KV item at that index in the slot array. For example, in Slice 0's index array, the first item "3" refers to the KV item 'c' at index 3 of the slot array. The index array is always filled consecutively from start to end.

Let L_s be the number of indexes in the index array of slice s, where $0 \le L_s \le IndexArraySize, \forall s \in [0, N)$. Let $key_{s,i}$ be the key[1] pointed by the i^{th} item of the slice s's index array . Let R_i denote the range of the keys in slice i, and K_i be the i^{th} slice boundary key in the slice locator, a MAPLe leaf always has the following invariants:

$$key_{s,i} < key_{s,i+1}, \forall s \in [0, N), i \in [0, L_s) \tag{2}$$

$$key_{s,L_s-1} < key_{s',0}, \forall s, s' \in [0, N-1), s < s', L_s > 0, L_{s'} > 0 \tag{3}$$

$$R_i = \begin{cases} (-\infty, K_0] & i = 0 \\ (K_{i-1}, K_i] & i \in [1 \ldots N-2] \\ (K_{N-2}, \infty) & i = N-1 \end{cases} \tag{4}$$

Expression (2) shows that the keys are laid out in ascending order[2] within each slice. Expression (3) says that the keys are also laid out in ascending order across slices[2]. Expression (4) specifies that the slice boundary keys define the key range for each slice and thus can be used to find the slice containing the target key.

The boundary keys are updated only during MAPLe leaf redistribution operations, which will be discussed in §3.3.2. Since the slice locator contains only keys, which are typically much smaller than KV items, it is more likely to fit into the CPU cache and can be accessed quickly.

Free slots in the slot array (e.g., slots 0, 5, and 8 in Figure 2) are linked into a "free slot list". The index of the head of the list, called "free slot head", is stored in the MAPLe leaf.

3.2 The Concurrency Model of the MAPLe Tree

The MAPLe tree's concurrency model is built on top of the concurrent version of the state-of-the-art in-memory TLX B$^+$-tree [1, 24].

3.2.1 Locking Model of the TLX B$^+$-Tree. The concurrent TLX B$^+$-tree is under the directory `tlx-plain` of the project [24]. It has two locking modes: optimistic and pessimistic.

In the optimistic mode, the tree uses top-down hand-over-hand locking to take read locks from the root down to the last level of inner node, and a read lock for lookups or a write lock for insertions/deletions on the leaf node. This mode allows for high concurrency in the tree.

In the pessimistic mode, the entire path from the root to the leaf is write-locked. This mode allows tree Structure Modification Operations (SMO), such as splits, merges, or rebalances, to be performed on the tree without any race conditions. However, this mode also significantly limits the concurrency of the tree.

The TLX B$^+$-tree maintains an invariant where the key in any inner node must equal the largest key in the inner node's left subtree. Consequently, when the largest key in a leaf node is deleted, unless it was the largest key in the whole tree, there will always be at least one key in an ancestor inner node that needs to be updated to the new largest key in the leaf node. This is called key propagation. If either key propagation or SMO is needed for a thread's operations, that thread will free all of its locks and retry from the root node, this time in pessimistic mode. Because pessimistic mode is expensive, there is an optimization for the insertion operation: if the parent of the leaf node to be split is not full, the thread will try to upgrade its read lock to a write lock. If the try-to-upgrade operation succeeds, which is true in most cases, the thread can perform the node split without switching to pessimistic mode.

The TLX B$^+$-tree has a scalable read-write lock for inner nodes; this takes more memory (linear to the CPU cores) but provides higher performance for read locks. This design has a per-CPU read counter where each reader thread can update efficiently without flushing the CPU cache. This design is suitable for inner nodes because most locks are read locks, except during rare SMO or key propagation operations. The high memory usage of this lock is not an issue because the number of inner nodes is low compared to the number of leaf nodes.

3.2.2 Locking Model of the MAPLe Tree. The MAPLe tree uses the same concurrency model as the concurrent TLX B$^+$-tree except for the leaf level. In addition to the read-write lock on the leaf, each slice also bears a separate read-write lock. For non-SMO operations (lookups, insertions, and deletions), the leaf is read-locked, and the target slice is read-locked for lookups and write-locked for insertions/deletions. The read lock at the leaf is a key design advantage of the MAPLe tree because it allows multiple threads to operate on different slices of the same leaf concurrently.

[1]Without ambiguity, keys mean the keys of the KV items.
[2]Because the MAPLe tree studied in this paper contains only unique keys, $<$ instead of $\le$ is used.

The free slot list uses a mutex lock to protect slot allocation and deallocation during insertion and deletion. Although this mutex is at the leaf level, it is unlikely to cause significant lock contention due to the short duration it is held.

The MAPLe leaf is write-locked only by some rare operations where multiple slices are involved. For example, the redistribution operation, discussed in §3.3.2, touches data structures in all slices and therefore must write-lock the entire leaf.

3.3 Restructuring MAPLe Leaves

The MAPLe tree has three operations to restructure the layout of a leaf node by enabling, disabling, or optimizing its MAPLe structure. They are called maplization, unmaplization, and redistribution respectively.

3.3.1 Maplization and Unmaplization. When a MAPLe leaf needs to undergo an SMO, it is unmaplized to become a regular B$^+$-tree leaf before the SMO and then maplized back afterward.

To unmaplize a leaf, a temporary slot array is first allocated, and all KV items are copied from the MAPLe leaf to the temporary array in sorted order. Finally, the KV items are copied back into the slot array of the original leaf.

To maplize a leaf, first the slice boundary keys are identified and copied into the slice locator, then each slice's index array is created and filled with the right indexes, and finally the free slot list is built with unused slots. No KV items are moved during the maplization.

The maplization and unmaplization are purely a simplification of the MAPLe tree implementation to reuse the existing B$^+$-tree SMO handling code. It is possible to handle SMOs on MAPLe leaves directly. However, since SMOs are rare, as discussed in §5.2.5, this simplification is unlikely to cause any noticeable performance degradation.

3.3.2 Redistribution. When a slice's index array becomes full, the entire MAPLe leaf undergoes a "redistribution" process to evenly spread the indexes of the KV items across all slices. During redistribution, a temporary array is first allocated. The index values in all index arrays from all slices are copied into this temporary array in the same order as they are in the slices, so that the KV items these indexes point to are in a sorted order. Then the indexes are copied back to each slice's index array in a balanced way. After the redistribution, every slice's index array will have at least one free entry to allow an insertion to proceed, i.e. $IndexArraySize \geq SliceSize + 1$. Since no actual KV items are moved during redistribution, it is a relatively fast operation.

If a B$^+$-tree leaf is full or about to underflow, it needs to perform an SMO, which involves expensive KV item movements. The MAPLe leaf achieves a similar outcome with redistribution without moving KV items. This efficient redistribution operation is one of the key performance advantages of the MAPLe tree.

3.4 MAPLe Leaf Operations

Lookup, insertion, and deletion are the three basic MAPLe leaf operations.

3.4.1 Lookup. A MAPLe lookup operation first searches for the key from the root to the target leaf like in a typical B^+-tree. Once reaching the leaf, it locates the target slice by searching through the slice locator using binary search. It then binary-searches the index array of that slice to find the index of the matching key. It returns the found KV item to the caller or `NOT_FOUND` otherwise.

3.4.2 Insertion. During insertion, a lookup is first performed to find the matching key. If it is found, the operation simply returns `FALSE` to the caller without any actions.

If the key does not exist, the operation checks if the target leaf is full. If it is, a B$^+$-tree split is performed. As discussed in §3.3, the leaf is unmaplized before the split and maplized after. Then the target leaf is looked up again.

If the leaf is not full, the operation checks if the target slice's index array is full. If it is, a redistribution operation, as discussed in §3.3.2, is conducted. When it is done, the slice locator is searched again to find a potentially different target slice, as the key may be in a different slice due to redistribution.

After the leaf and the slice's fullness check and handling are completed, the first free slot is removed from the free slot list, and the free slot head is updated. The KV item is copied to this free slot, and the index of this slot is inserted into the target slice's index array at the right position.

As an example, suppose the caller wants to insert the key `'e'` to the MAPLe tree constructed in Figure 2. Firstly, the slice locator is searched. Because `'e'`≤`'g'`, which is the first slice boundary key, slice 0 is identified as the target slice. Since slice 0's index array is not full, the new key can be inserted directly. The first slot of the free slot list, which is at the index 0, is removed, and the free slot head is updated to 5, the index of the next free slot. Then slice 0's index array is binary-searched to find the position where `'e'` should be inserted. Since `'e'` is between `'c'` and `'f'` (at index array positions 0 and 1, and slot array positions 3 and 1, respectively), its index, 0, should be inserted between the indexes 3 and 1 in the index array. The indexes 1 and 4 are shifted to the right, and the new index 0 is placed at position 1. Slice 0's index array now contains the following indexes: 3, 0, 1, and 4. Figure 3 illustrates the MAPLe leaf's structure after `'e'` is inserted.

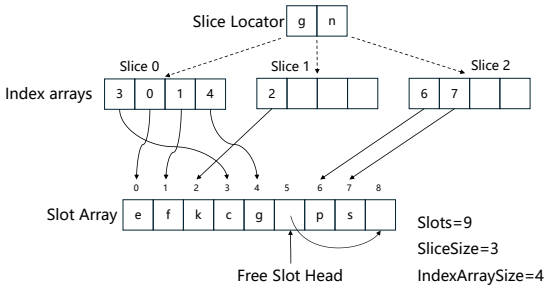

Figure 3: The MAPLe Leaf After Inserting 'e'

3.4.3 Deletion. Deletion is similar to insertion. First a lookup is performed. If the target key is not found, the operation returns with error `NOT_FOUND`. If the key exists, the operation checks whether the leaf is about to underflow, in which case there needs to be an SMO operation, and the leaf is unmaplized before the SMO and maplized back after the SMO. Then the deletion restarts with the lookup again.

After the target slice and target position in the slice are found, all of the indexes after the target key index are shifted to the left to delete the target index. The slot where the target KV item resides is added to the head of the free slot list and the free slot head is updated to point to the just deleted slot.

Note that unlike insertion, even if a slice becomes completely empty after the deletion, redistribution is not triggered to reduce unnecessary redistribution.

As an example, suppose the caller wants to delete `'p'` from the MAPLe tree in Figure 3. First, the slice locator is binary-searched to find the first slice boundary key that is ≥ `'p'`. Since `'p'` is larger than the last boundary key `'n'`, slice 2 is the target slice. Then slice 2's index array is binary-searched and `'p'` is found at position 0, which points to slot 6 of the slot array. The index array is shifted left and the index 6 is removed. The index array now contains only one index, 7. Slot 6 of the slot array is added to the free slot list. This is done by storing the free slot head's current value 5 into slot 6 where `'p'` was and then setting the free slot head to 6 (the new head).

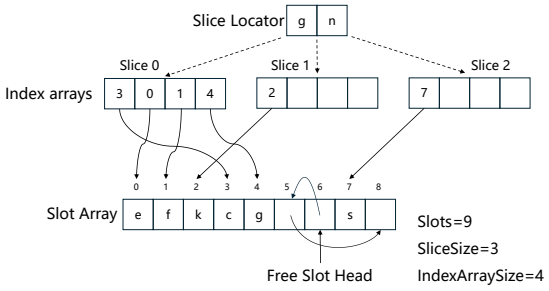

Figure 4: The MAPLe's Leaf After Deleting 'p'

3.4.4 Design Discussion. In the MAPLe tree, insertions and deletions boil down to only moving 2-byte indexes, whereas in the B$^+$-tree, KV items are moved. This is one of the key performance advantages of the MAPLe tree, since normally, KV items are much larger than 2 bytes.

One argument is that large KV items can be replaced by pointers pointing to dynamically allocated memory storing the KVs. However, this is not straightforward to implement because of the complexities in memory management. In C++, `std::unique_ptr` is a common way used to manage the life cycle of such dynamically allocated KV items. However, this cannot be used in the TLX B$^+$-tree directly. `unique_ptr` doesn't allow copy, so `std::move` must be used when shifting KV items in the slot array. This requires extensive changes to the B$^+$-tree's code, because many places need to copy KVs around. If raw pointers are used instead of `unique_ptr`, the life cycle management of the KV pointers becomes even harder.

Alternatively, an index array can be added to a B$^+$-tree leaf as a level of indirection, so that indexes in the index array can point to different slots in the leaf. This is similar to what is used in the MAPLe leaf's slice structure and has been used in the COW B$^+$-tree [18]. This design can be simulated using a single-slice MAPLe leaf. Our performance evaluation has compared our MAPLe tree with a single-slice MAPLe tree and has found that multiple slices perform better than the single-slice variation because multiple slices allow for much better concurrency.

4 Time and Space Complexity Analysis of the MAPLe Design

Before conducting a thorough experiment-based evaluation on the MAPLe tree, we analyze its time and space complexity and justify some of its design choices.

4.1 Time Complexity Analysis

Since the MAPLe tree differs from the B$^+$-tree only in its leaf design, the time complexity of common operations on their leaves is compared.

Assuming a MAPLe tree leaf has S slices and each slice has N indexes, there are a total of $S \cdot N$ KV items in the MAPLe leaf. A B$^+$-tree leaf containing the same number of KV items is compared with the MAPLe leaf.

For lookup operations, the binary search on the B$^+$-tree leaf takes $O(\log(SN))$. The MAPLe leaf first takes $O(\log S)$ to find the target slice, and then $O(\log N)$ to find the target index. The total time is $O(\log S + \log N) = O(\log(SN))$. Therefore, both trees should have similar lookup performance.

For insertions/deletions, a B$^+$-tree leaf first uses lookup to find the right position ($O(\log(SN))$, and then moves $O(SN)$ slots. A MAPLe leaf takes $O(\log(SN))$ to find the right index and moves $O(N)$ index entries. Therefore, insertions/deletions in a MAPLe leaf should be faster than those in a B$^+$-tree leaf.

Scanning a B$^+$-tree leaf takes $O(SN)$ as it scans the slot array from the beginning to the end, which has good cache locality. Scanning a MAPLe leaf also takes $O(SN)$ by iterating all indexes in the index array of each slice and then the KV item pointed by each index. Because of this, the MAPLe leaf scan may be slower due to the extra level of indirection and worse CPU cache locality.

4.2 Memory Overhead

The MAPLe leaf incurs memory overhead with its extra data structures, mainly the slice locator and the index arrays. This overhead can be calculated as:

$$Overhead = \frac{TotalIndexArraySize + SliceLocatorSize}{LeafSize} \quad (5)$$

where:

$$LeafSize = Slots \cdot (KeySize + ValueSize) \quad (6)$$

$$TotalIndexArraySize = IndexSize \cdot NumSlices \cdot (SliceSize + 1) \quad (7)$$

$$SliceLocatorSize = KeySize \cdot (NumSlices - 1) \quad (8)$$

Eqn (6) is the leaf size of a B$^+$-tree. $NumSlices$ in Eqn (7) and (8) is calculated in Eqn (1). In Eqn (7) , $IndexSize$ is 2 bytes, and $SliceSize + 1$ is the value of $IndexArraySize$ as explained in §3.3.2. In Eqn (8), $NumSlices - 1$ is the number of boundary keys in the slice locator.

Table 1 presents the memory overhead of a MAPLe leaf with various parameters. As shown in the table, the memory overhead is insignificant in all cases, especially for trees with larger value sizes. This is because the slice index arrays, containing 2-byte indexes,

take up a smaller proportion of the memory as the value size becomes bigger. In addition, as *SliceSize* becomes bigger, *NumSlices* is smaller, so the slice locator is smaller, which reduces its relative overhead to the leaf size.

Slots	KeySize	ValueSize	SliceSize	Overhead
64	8	128	16	1.84%
64	8	256	16	0.95%
1024	8	256	32	0.87%
8192	8	256	32	0.88%

Table 1: MAPLe's Memory Overhead

(All sizes are in bytes)

5 Performance Evaluation

In this section, we conduct a thorough experiment-based evaluation on the MAPLe tree.

5.1 System Setup and Common Parameters

The experiments are performed on a Dell PowerEdge T630 server with an Intel Xeon processor (E5-2683 v4 at 2.10GHz), containing 2 sockets and 16 cores per socket. The memory size is 128 GB. The hyperthreading is disabled. One NUMA (Non-Uniform Memory Access) node is used in all experiments in this section to avoid extra factors influencing performance. The OS is Ubuntu Linux 24.04.

A key size of 8 bytes and a value size of 256 bytes are used in all experiments.

5.2 Benchmarking MAPLe Leaves

The MAPLe tree's optimization efforts are targeted at the leaf nodes. In order to clearly understand the performance characteristics of MAPLe leaves, microbenchmark experiments were performed without considering the rest of the tree structures.

Instead of creating a large B$^+$-tree or MAPLe tree holding many leaves, an array of tree leaves (10,000 in the experiments) is allocated. Each leaf is initialized to be 75% full, since the space utilization of a leaf oscillates between 50%∼100% and 75% is the expected average.

The keys are generated from a uniform distribution, and the values are composed of random characters. Note that the KV items in each leaf are sorted only in the lookup tests, but are not sorted in all the other tests (insertion, deletion, redistribution, scan, maplization, and unmaplization). This is because all operations except lookup do not need binary search, and therefore, unsorted KV items do not affect their experimental results.

Every benchmark runs 1,000,000 operations on random leaves, to avoid the leaf staying in the CPU cache all the time.

The microbenchmark results confirm the time complexity analysis in §4.1: the MAPLe tree's insertion and deletion are much faster than the B$^+$-tree's. Its lookup performance is similar, but its scan is slower.

5.2.1 The Insertion and Deletion Microbenchmark. The insertion and deletion microbenchmark is composed of 50% insertions and 50% deletions, except for two cases. If the slice or the leaf picked is full, only deletion is performed; likewise, if the slice or the leaf is empty, only insertion is performed. Since only the actual insertion and deletion actions are of interest, this microbenchmark skips the lookup process. Instead, it picks a random slot on a B$^+$-tree leaf, or a random slice and a random entry in the slice's index array on a MAPLe leaf, to perform insertion or deletion.

Table 2 presents the insertion and deletion performance with varying *Slots* (the size of the leaf) and *SliceSize* (the size of the slice). As shown in the table, the insertion time of the MAPLe leaves barely changes as the size of the leaves increases. The insertion time of the B$^+$-tree's leaves, on the other hand, grows linearly with the leaf size. This is because the MAPLe leaf only moves 2-byte indexes, making its performance insensitive to leaf size change, while the B$^+$-tree leaf moves KV items, causing its throughput to shrink linearly with the leaf size. For very large trees (i.e., Slots=8192), the MAPLe leaves can outperform the B$^+$-tree's leaves by more than 500 times. Because deletion has similar properties to insertion, no further experiments on deletion are performed in the rest of this section.

5.2.2 The Lookup Microbenchmark. As shown in Table 2, the MAPLe leaves' lookup is on-par or slightly slower than the B$^+$-tree's for very large trees. Note that lookup is already a fast operation compared to insertion and deletion since it doesn't require any data movements.

5.2.3 The Scan Microbenchmark. A scan operation traverses over all of the KV items in the randomly chosen leaf. Since the MAPLe leaf has one more level of indirection when accessing each KV item, it is slower than the B$^+$-tree leaves, as shown in Table 2.

5.2.4 The MAPLe Redistribution Microbenchmark. As shown in table 2, the redistribution operation is relatively fast even for very large trees, which is not a surprise since redistribution does not involve any KV item movements, as discussed in §3.3.2.

5.2.5 The Maplize and Unmaplize Microbenchmark. As discussed in §3.3.1, maplization and unmaplization only happen before and after an SMO operation. Maplization is fast since there are few data movements, whereas unmaplization is relatively expensive as it needs to copy all KV items out and copy them back in order.

In order to understand how maplizations/unmaplizations impact the performance of other operations, a balanced workload (defined in §5.3) is used to measure the count and the run time of maplization and unmaplization in proportion to the total run time. This balanced workload initializes the leaf array with 25 million KV items and then performs another 25 million balanced operations with 16 threads. 0.2% of the operations generate an SMO, and the total time spent on maplization and unmaplization is 6.4% of the total runtime. This overhead could be optimized by performing SMOs directly on the MAPLe leaves. We leave this as a future work because the extent of this improvement is limited.

5.3 Performance of the MAPLe Tree

This section evaluates the performance of the MAPLe tree and compares it with that of the B$^+$-tree. As a previous study has found that the performance of leaves is the most critical and that the size of the inner nodes does not matter much [23], we set the size of all inner nodes to be the same as the size of the leaf nodes (denoted in §3.1 as *Slots*) in all experiments.

Slots	Slice Size	B$^+$-tree Insert	MAPLe Insert	B$^+$-tree Delete	MAPLe Delete	B$^+$-tree Lookup	MAPLe Lookup	B$^+$-tree Scan	MAPLe Scan	MAPLe Redis-tribute	MAPLe Map-lize	MAPLe UnMap-lize
64	32	0.42	0.14	0.38	0.16	0.15	0.18	0.63	0.83	0.13	0.25	1.58
256	32	1.88	0.15	1.81	0.19	0.18	0.19	2.90	5.21	0.28	1.36	11.64
2048	64	29.23	0.17	28.56	0.26	0.29	0.52	31.13	82.54	1.34	9.56	121.34
8192	32	108.60	0.20	107.10	0.35	0.39	0.65	129.20	359.2	6.85	49.41	509.11
8192	64	108.60	0.20	107.10	0.34	0.39	0.71	129.20	336.90	4.90	39.22	482.30

Table 2: Leaf Operation Microbenchmark Performance. All numbers are in microseconds (μs). Lower is better

In addition to the *Slots* parameter, the MAPLe tree has two more parameters, *SliceSize*, and *IndexArraySize*, as explained in §3.1. Choosing a larger *IndexArraySize* uses more memory but reduces the need for redistribution. Since redistribution only moves the 2-byte indexes in all index arrays, it is quite efficient. As the results in §5.2.4 show, a redistribution operation is relatively fast considering its low frequency compared to lookup, insertion, or deletion. This allows *IndexArraySize* to use its minimum value to reduce memory usage. In all experiments, *IndexArraySize* is set to *SliceSize* + 1, and its value will no longer be presented below.

When comparing the B$^+$-tree and the MAPLe tree, the best parameter values (*Slots* and *SliceSize*) for the B$^+$-tree and the MAPLe tree when running a "balanced workload" are chosen for all the performance tests. This workload is a mix of 4 tree operations with equal weights (25% insert, 25% delete, 25% lookup, and 25% scan of 100 items). In general, the B$^+$-tree's insertion and deletion are faster on small leaf sizes (i.e. small *Slots*), while the MAPLe tree's insertion and deletion are faster on a small *SliceSize* with a large *Slots*. This balanced workload attempts to find the "best" parameters that are fair to both trees and compares their performance in various scenarios.

The MAPLe tree has two factors that improve its performance: its concurrent slice design, which reduces lock contention; and its level of indirection, which reduces data movement. As discussed in §3.4.4, the COW B$^+$-tree has a level of indirection in its leaf node [18] as well. The COW B$^+$-tree is an on-disk tree which relies on a buffer cache to provide its locking, so it cannot be easily compared with the TLX B$^+$-tree or the MAPLe tree. However, by setting the number of slices to be 1 and the *SliceSize* to be identical to *Slots*, the MAPLe tree can mimic the design of the COW B$^+$-tree. This is called "1-slice" in the experiments below. This helps us understand how much performance gain is attributed to the level of indirection as opposed to the improved concurrency of the tree's multi-slicing. The best parameter values (*Slots*) are also chosen for the 1-slice tree. It turns out that the 1-slice tree has the same best *Slots* value as the MAPLe tree for balanced workloads.

In these experiments, the key size is 8 bytes and the value size is 256 bytes. The keys are generated by a uniform distribution and a Zipfian distribution with $\alpha = 0.99$. The measurements are taken by preloading the trees with 25 million KV items, and then conducting 25 million "balanced" operations on them. Figure 5 (a) presents the results for B$^+$-trees. The "best" performance is achieved with small *Slots* = 64, regardless of whether the keys follow a uniform or skewed distribution. Figure 5 (b) shows the results for MAPLe trees. With no surprise, the "best" performance is achieved with large *Slots* = 8192 and small *SliceSize* = 32. The Zipfian key distribution performs better than uniform key distribution, as skewed access of Zipfian has much better CPU cache locality. Figure 5 (c) compares the best MAPLe tree with the best B$^+$-tree and the best 1-slice tree. The MAPLe tree scales much better than the B$^+$-tree, as the MAPLe tree reduces lock contention by allowing multiple threads to access different slices within a MAPLe leaf. The MAPLe tree also outperforms the B$^+$-tree by up to 158%, since the MAPLe tree only moves 2-byte indexes while the B$^+$-tree moves large KV items during insertion and deletion. The MAPLe tree performs better than the 1-slice tree because the smaller slices in the MAPLe tree provide better concurrency. These "best" parameters, found in Figure 5, are summarized in Table 3.

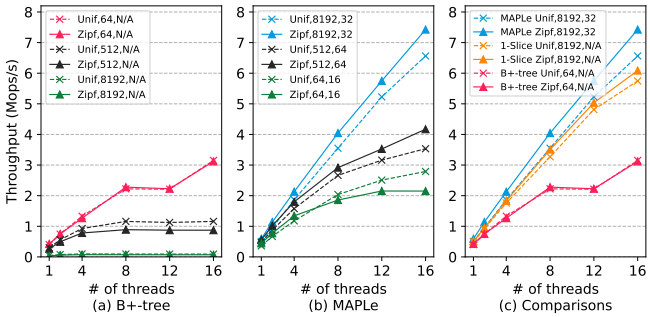

Figure 5: Throughput of Balanced Workloads

(Balanced workload has 25% insert, 25% delete, 25% lookup, and 25% scan of length 100. The first number is *Slots*, and the second number is *SliceSize*. A B$^+$-tree does not have *SliceSize*)

Type	Slots	SliceSize
B$^+$-tree	64	N/A
MAPLe Tree	8192	32

Table 3: The Best Parameters for Balanced Performance

(KeySize=8B, ValueSize=256B, IndexArraySize=SliceSize+1)

Figure 6 compares the lookup, insertion, and scan performances of the B$^+$-tree and the MAPLe tree with the parameters presented in Table 3. Both uniform and skewed (Zipfian) key distributions are tested. The trees are initialized with 25 million KV items. For the lookup and insertion experiments, 2.5 million operations are conducted. For the scan experiment, 0.25 million operations are executed, since scan is a slow operation, as shown in Table 2.

The experimental results in Figure 6 indicate that the MAPLe tree consistently outperforms the B$^+$-tree. For lookup, the MAPLe tree is up to 34% faster; for insertion, the MAPLe tree is up to 85% faster; and for scan, the MAPLe tree is up to 41% faster. The benchmarking experiments in Table 2 show that the MAPLe tree's lookup and scan are slower than those of the B$^+$-tree when they have the same leaf size. The MAPLe tree outperforms the B$^+$-tree in end-to-end lookup and scan tests because the MAPLe tree has a much bigger *Slots* and therefore has less levels. This reduces the time spent on inner nodes.

Both trees scale linearly for lookup and scan. This is expected because only read locks are taken. The MAPLe tree scales better than the B$^+$-tree for insertion in Zipfian key distributions, since each MAPLe leaf can have multiple threads writing in it and therefore has less contention than a B$^+$-tree leaf.

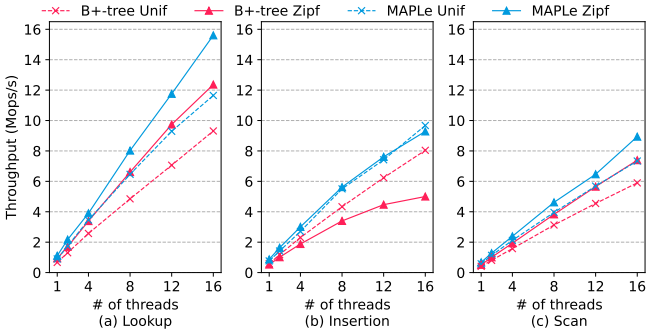

Figure 6: Throughput of Lookup, Insertion, and Scan

Figure 7 compares the performance of the MAPLe tree and the B$^+$-tree under the YCSB[3] A, B, and E benchmarks [3]. The YCSB C benchmark contains 100% lookups, and those performance results are already presented in Figure 6(a). YCSB A has 50% lookups and 50% inserts, YCSB B has 95% lookups and 5% inserts, and YCSB E has 95% scan of length 100 and 5% inserts. Both uniform and Zipfian distributions are used. The MAPLe tree outperforms the B$^+$-tree in all cases under the same distribution: it is up to 55% faster in YCSB A; up to 30% faster in YCSB B; and up to 37% faster in YCSB E.

5.4 Discussion of Workloads

For in-memory ordered KV indexes, there are many workload types. The MAPLe tree design focuses on workloads with the following properties:

(1) *Skewed access.* Some KV items are accessed much more frequently than others.
(2) *Concurrent access.* Multiple threads on multiple CPU cores access the tree concurrently.
(3) *Large value sizes.* The value size is a few hundred bytes.
(4) *Update heavy.* A high proportion of insertion and deletion operations is needed.

We believe that the above workload is important. If the workload is less skewed or if there are less updates, the lock contention on the leaf nodes would be reduced, which may move the bottleneck elsewhere.

[3]Yahoo! Cloud Serving Benchmark

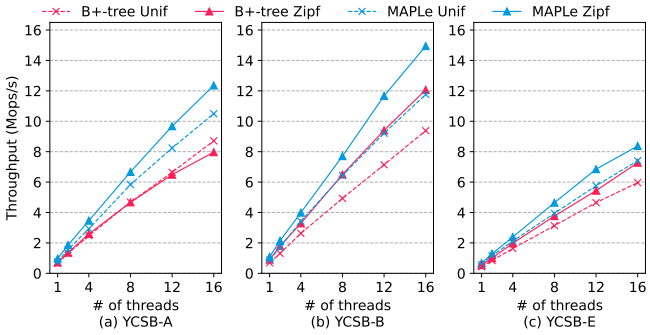

Figure 7: Throughput of YCSB Workloads A, B, and E
(YCSB A has 50% lookups and 50% inserts; YCSB B has 95% lookups and 5% inserts; YCSB E has 95% scan of length 100 and 5% inserts.)

The MAPLe tree works well both in trees with many levels and in trees with few levels because in both types, most of the operations will happen on the leaf nodes. A shallow tree has more contention on the inner nodes, but less binary searches are needed to reach the leaf node. In this section's performance tests, the MAPLe tree is shallow. Its height is only 2 because the MAPLe tree has large leaves (*Slots* = 8192). Because the read lock at the only inner node (which is also the root) is highly scalable, no performance slowdowns were observed on the shallow tree. When the inner node fanout was reduced from 8192 to 64, our experiments found that the performance of the MAPLe tree on the balanced workload changed by less than 10%.

6 Related Work

Improving comparison-based ordered indexes has always been an active research topic that may have a profound impact on a variety of data management systems. Among the indexes, the B-tree (or B$^+$-tree) is one of the most commonly used in various database systems, such as LMDB [9] and MongoDB [16]. Because these indexes' lookup cost is bounded by $O(\log N)$, the efforts on optimizing their performance are mainly focused on the improvement of concurrency and caching efficiency. For example, the Bw-tree employs latch-free operations on the B$^+$-tree to improve its lookup efficiency on multi-cores [13]. FAST uses architecture-specific knowledge to optimize the B$^+$-tree's layout in the memory to reduce cache and TLB misses [12]. Many studies have proposed using hardware accelerators, such as GPU, to improve index lookups without changing the underlying data structure [8, 19, 25]. Our MAPLe tree takes a new approach, which is to optimize data organization and operations in the leaf nodes of a B$^+$-tree. The leaf nodes are where all data are stored and a majority of the memory space is used. The MAPLe tree design does not depend on specific hardware and avoids the complexity that comes with using lock-free operations.

Many synchronization approaches have been proposed to improve concurrency in a tree structure. For example, MemC3 [4] and Masstree [14] adopt version numbers to enable lock-free access for readers. Atomic operations, such as CAS and LL/SC, have been extensively used to implement lock-free skip lists [6] and binary

search trees [17]. The MAPLe tree enables fine-grained locking within the leaf node by partitioning the KV items in the node into multiple slices. Being a B$^+$-tree, the MAPLe tree also avoids the memory fragmentation of skip lists and the space waste of binary search trees.

Caching techniques may effectively improve index searches for workloads with strong locality. For example, SLB uses a small cache to reduce the lookup cost for frequently accessed data [22]. The B$^\epsilon$-Tree is a B$^+$-tree-like index that allocates a buffer at each inner node to reduce the high write amplification of the B$^+$-tree [2]. However, the use of buffers incurs additional overhead for lookups.

A recent work that also optimizes the B$^+$-tree at the leaf nodes is the BP-tree. To better support scans, it uses a Buffered Partitioned Array to allow leaves to be large while still allowing fast insertion and deletion [23, 24]. However, it still needs to move KV items with large values around when flushing the data in the log area to the main array. The MAPLe tree allows large leaves because it internally divides a large leaf into multiple slices that can be concurrently searched and changed. By doing this, it solves the lock contention problem of large leaves. Also, the MAPLe tree can handle large KV items efficiently by not moving them during insertion and deletion.

7 Conclusion

In this work, we identify a performance optimization opportunity in the leaves of a B$^+$-tree: their insufficient concurrency under highly skewed write workloads. To address this, we propose the MAPLe (Multi-Access Parallel Leaves) tree design, where the KV items are logically partitioned into slices that can be concurrently modified, and KV item movements during insertion and deletion are virtually eliminated. We have experimentally evaluated its performance with workloads of different access patterns and varying numbers of threads. The results show that the MAPLe tree improves on the B$^+$-tree's throughput by up to 158% for large value sizes. The source code of MAPLe's implementation is available at https://github.com/lunawang257/tlx/tree/mapl.

References

[1] Timo Bingmann. 2018. TLX: Collection of Sophisticated C++ Data Structures, Algorithms, and Miscellaneous Helpers. https://panthema.net/tlx, retrieved July 13, 2024.

[2] Gerth Stolting Brodal and Rolf Fagerberg. 2003. Lower Bounds for External Memory Dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (Baltimore, Maryland) *(SODA '03)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 546–554.

[3] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. ACM, 143–154. https://doi.org/10.1145/1807128.1807152

[4] Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Lombard, IL) *(NSDI'13)*. USENIX Association, Berkeley, CA, USA, 371–384.

[5] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28–33. http://sites.computer.org/debull/A12mar/hana.pdf

[6] Mikhail Fomitchev and Eric Ruppert. 2004. Lock-free Linked Lists and Skip Lists. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing* (St. John's, Newfoundland, Canada) *(PODC '04)*. ACM, New York, NY, USA, 50–59. https://doi.org/10.1145/1011767.1011776

[7] PostgreSQL Global Development Group. 2024. *PostgreSQL: The World's Most Advanced Open Source Relational Database*. Available: https://www.postgresql.org.

[8] Bingsheng He and Jeffrey Xu Yu. 2011. High-throughput Transaction Executions on Graphics Processors. *Proc. VLDB Endow.* 4, 5 (Feb. 2011), 314–325. https://doi.org/10.14778/1952376.1952381

[9] Howard Chu. 2024. *LMDB: Lightning Memory-Mapped Database*. Symas Corporation. Available: https://www.symas.com/lmdb.

[10] IBM Corporation. 2024. *IBM DB2 Database Management System*. Available: https://www.ibm.com/products/db2.

[11] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*. 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[12] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. 2010. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. ACM, New York, NY, USA, 339–350. https://doi.org/10.1145/1807167.1807206

[13] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for New Hardware Platforms. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13)*. IEEE Computer Society, Washington, DC, USA, 302–313. https://doi.org/10.1109/ICDE.2013.6544834

[14] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. ACM, New York, NY, USA, 183–196. https://doi.org/10.1145/2168836.2168855

[15] Microsoft Corporation. 2024. *Microsoft SQL Server*. Available: https://www.microsoft.com/sql-server.

[16] MongoDB, Inc. 2024. *MongoDB: The Developer Data Platform*. Available: https://www.mongodb.com.

[17] Aravind Natarajan and Neeraj Mittal. 2014. Fast Concurrent Lock-free Binary Search Trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Orlando, Florida, USA) *(PPoPP '14)*. ACM, New York, NY, USA, 317–328. https://doi.org/10.1145/2555243.2555256

[18] Ohad Shacham, Marc de Kruijf, Monia Ghobadi, Sudipta Sengupta, and Yandong Mao. 2019. The COW B-Tree: A Data Structure for Flash Storage and Other Demanding Applications. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 198–213. https://doi.org/10.1145/3341301.3359652

[19] Amirhesam Shahvarani and Hans-Arno Jacobsen. 2016. A Hybrid B+-tree As Solution for In-Memory Indexing on CPU-GPU Heterogeneous Computing Platforms. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. ACM, New York, NY, USA, 1523–1538. https://doi.org/10.1145/2882903.2882918

[20] MonetDB Team. 2024. *MonetDB: A High Performance Database System*. MonetDB B.V. Available: https://www.monetdb.org.

[21] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 18–32. https://doi.org/10.1145/2517349.2522713

[22] Xingbo Wu, Fan Ni, and Song Jiang. 2017. Search Lookaside Buffer: Efficient Caching for Index Data Structures. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. ACM, New York, NY, USA, 27–39. https://doi.org/10.1145/3127479.3127483

[23] Helen Xu, Amanda Li, Brian Wheatman, Manoj Marneni, and Prashant Pandey. 2023. BP-Tree: Overcoming the Point-Range Operation Tradeoff for In-Memory B-Trees. *Proc. VLDB Endow.* 16, 11 (jul 2023), 2976–2989. https://doi.org/10.14778/3611479.3611502

[24] Helen Xu, Amanda Li, Brian Wheatman, Manoj Marneni, and Prashant Pandey. 2023. *tlx-plain Concurrent B-tree*. https://github.com/wheatman/BP-Tree/tree/artifact_evaluation/tlx-plain.

[25] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. 2015. Mega-KV: A Case for GPUs to Maximize the Throughput of In-memory Key-value Stores. *Proc. VLDB Endow.* 8, 11 (July 2015), 1226–1237. https://doi.org/10.14778/2809974.2809984