Concurrent Path-Copying Update to Tree Structures

GUANHAO HOU*, The Chinese University of Hong Kong, Hong Kong SAR
DECHUANG CHEN*, The Chinese University of Hong Kong, Hong Kong SAR
QINTIAN GUO, The Hong Kong University of Science and Technology, Hong Kong SAR
FANGYUAN ZHANG, The Chinese University of Hong Kong, Hong Kong SAR
SIBO WANG†, The Chinese University of Hong Kong, Hong Kong SAR

Historical data are widely used in science, business, and web applications. In this context, tree structures play a fundamental role in databases, particularly in query processing. Path copying offers a cost-effective approach to providing immutable snapshots of a tree. If snapshots preserve the order of update requests, each snapshot version is mapped to a specific point in history, thereby facilitating historical data queries and analysis.

Recently, Contreap enables concurrent path-copying updates on BSTs, primarily targeting historical queries on predefined statistics by maintaining the corresponding augments in tree nodes. However, real-world applications often require general-purpose operations, such as element retrieval or range scanning, while BSTs are not well-suited for such operations. Nevertheless, existing path-copying implementations for popular database tree structures generally lack efficient support for concurrent updates, limiting their applicability.

In this paper, we present ConTree, a lightweight programming library that transparently supports concurrent path-copying updates for tree structures. We show how to integrate a recursive update process — subject to certain constraints to ensure correctness — into ConTree. We further discuss techniques for optimising existing tree structures to improve the efficiency of concurrent path copying.

Building on this foundation, we instantiate ART and B+Tree, addressing key obstacles to concurrent path copying with two novel solutions: AERT and BeTree, using our ConTree library. These optimised variants incorporate structural and algorithmic refinements to support scalable updates. Extensive experiments show that our concurrent update strategy is effective, delivering significant speedups for AERT and BeTree over their serial counterparts.

CCS Concepts: • **Information systems → Data structures**.

Additional Key Words and Phrases: Concurrent, Path Copy, Tree Structures

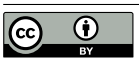
ACM Reference Format:
Guanhao Hou, Dechuang Chen, Qintian Guo, Fangyuan Zhang, and Sibo Wang. 2026. Concurrent Path-Copying Update to Tree Structures. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 153 (June 2026), 27 pages. https://doi.org/10.1145/3802030

1 Introduction

Tree indexes play an important role for query processing in database systems. Historical data have a wide range of applications in science [65], business [25, 43], web applications [62, 72], and more.

*Guanhao Hou and Dechuang Chen are joint first authors.
†Sibo Wang is the corresponding author.

Authors' Contact Information: Guanhao Hou, ghhou@se.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong SAR; Dechuang Chen, dcchen@se.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong SAR; Qintian Guo, qtguo@ust.hk, The Hong Kong University of Science and Technology, Hong Kong SAR; Fangyuan Zhang, fzhang@se.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong SAR; Sibo Wang, swang@se.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong SAR.



ACM 2836-6573/2026/6-ART153
https://doi.org/10.1145/3802030

In this paper, we focus on the retrieval of historical states in key-value stores associated with keys. For instance, determining the price of a product at the moment an order is placed, or the view count of a video at the conclusion of an event.

Path copying [31] provides a cost-effective way to create immutable snapshots of a tree structure, thereby enabling queries about the state at any moment in the past. In particular, path copying produces a new version by duplicating only the nodes whose keys, data, or child pointers are modified, and relinking these copies to form the new version. For example, in a vanilla binary search tree (BST), only the nodes along the search path to the updated key are duplicated during an update, while untouched subtrees are shared with previous versions, thereby maintaining space efficiency. The path copying technique is not only essential in functional programming languages for supporting set operations [19], but are also employed in file systems [63], distributed platforms [4, 30], as well as a variety of in-memory databases [9, 10, 12].

During the path copy process, once a node has been duplicated, subsequent modifications to the original node are not reflected in its copy. Therefore, to ensure correctness—namely, that the new version faithfully retains the state of the replicated version—any modifications to a node must be finalised before the node becomes visible to other threads. Efficiently guaranteeing this constraint is non-trivial: correct synchronisation is complex and error-prone, risking data corruption or inconsistent versions, while overly conservative synchronisation can severely degrade performance.

Contreap [42] recently implemented concurrent path copying for balanced BSTs to accelerate historical queries on predefined statistics. It resolves concurrency via a hybrid scheduler: the upper layers run in a pipeline, while lower layers use independent worker threads to process updates concurrently. Although Contreap delivers strong update speedups, BSTs remain a poor fit for many general-purpose indexing tasks such as element retrieval or range scanning. Meanwhile, widely used database tree structures generally lack efficient support for concurrent path copying, creating bottlenecks under write-heavy workloads.

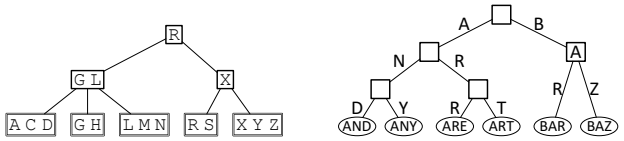
To close this gap, this paper focuses on a general approach to enable concurrent path copying across various tree structures. We summarise our contributions as follows:

- In Sec. 3, we introduce ConTree, a lightweight and reusable library that transparently supports non-blocking, starvation-free, and order-preserving concurrent path-copying updates in common tree indexes, enabling efficient per-update versioning and freeing developers from low-level concurrency concerns.
- In Sec. 4, we present the implementation of ConTree. We design a low-overhead dependency resolution mechanism using version and stage information to improve efficiency. We further describe a hybrid policy that applies different strategies for different tree levels to boost performance.
- In Sec. 5, we instantiate ART and B+Tree with ConTree, addressing key challenges in concurrent path copying via two novel solutions: AERT and BeTree. These optimised variants combine structural and algorithmic refinements to boost update efficiency.
- In Sec. 6, we conduct extensive experiments. The results show that our ConTree is effective, with AERT and BeTree achieving speed-ups of up to 15× and 7.5× over their serial counterparts, respectively. We further evaluate end-to-end performance on the hybrid TPC-CH workload (transactional + analytical), where our approach outperforms prior hybrid-workload solutions, demonstrating the effectiveness of our path-copying framework.

2 Preliminaries

2.1 Tree Structures

Tree structures are commonly used to implement key-value stores in database systems. A tree is defined recursively, where each node is either a leaf node or contains several child nodes. The



(a) B+Tree (b) Radix Tree

Fig. 1. Example of Tree Structures

height of a node T, denoted as $\mathsf{h}(T)$ is defined as the length of the path from T to the farthest leaf node in its subtree, with the height of a leaf node being 0. The **depth** of a node T, denoted as $\mathsf{d}(T)$, is the length of the path from the root node to that node, with the depth of the root node being 0. In this paper, we use the term the 'd-th' **layer** to refer to the set of all nodes at depth d.

Binary Search Tree (BST). Search trees partition keys based on key comparisons. The simplest form of search tree is the BST, in which each node represents a single element and has two child nodes, referred to as the left and right subtrees. Elements in the left subtree have keys less than the key of the node, while elements in the right subtree have keys greater than the key of the node.

B+Tree. A B+Tree is a height-balanced search tree, where all elements are stored in leaf nodes, while internal nodes only store keys. Given a B+Tree of **order** b, the size (i.e., the number of records) of a leaf node falls within the range $[b, 2b)$. Each non-leaf node has a size, also known as fanout (i.e., the number of child nodes), in the range $[b, 2b)$. The root node has a size in the range $[2, 2b)$. A non-leaf node with k child nodes contains $(k-1)$ keys, where the i-th key separates the i-th and $(i+1)$-th subtrees, and all elements in the i-th (resp. the $(i+1)$-th) subtree are less (resp. no less) than the i-th key. Fig. 1(a) shows an example of a B+Tree with $b = 2$.

After appending a record to a leaf node, its size may reach $2b$, resulting in overflow. Conversely, after removing a record, the size may fall below b, resulting in underflow. The B+Tree resolves overflow by splitting the node, and resolves underflow by merging nodes. Splitting or merging may trigger cascades that propagate overflow or underflow to parent nodes. The B+Tree addresses such conditions recursively, thereby maintaining structural balance.

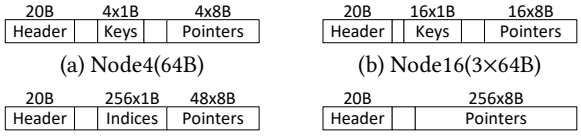
Adaptive Radix Tree (ART). Unlike search trees which perform comparison-based search, tries match keys by segments rather than comparing full keys. A root-to-leaf path represents a key, with each edge corresponding to a key segment. In a search tree of height h, searching a key of length l takes $O(lh)$ time, as each comparison may examine the entire key. In contrast, a trie performs a lookup in $O(l)$ time by matching each segment of the key along the path.

Radix trees are an efficient variant of tries that compress common prefixes in internal nodes. As shown in Fig. 1(b), the shared prefix 'BA' of the keys 'BAR' and 'BAZ' are stored in a compressed form. This representation eliminates degenerate nodes with only a single child, reducing both space and lookup depth.

ART is a 256-way radix tree where each level matches one byte of the key. Instead of using a fixed 256-entry child array, ART employs four adaptive node types (Fig. 2) to match actual fanouts, reducing memory usage when a node has a small actual fanout. ART further improves performance by leveraging modern CPU features such as cache locality, branch prediction, and SIMD instructions.

2.2 Path Copy

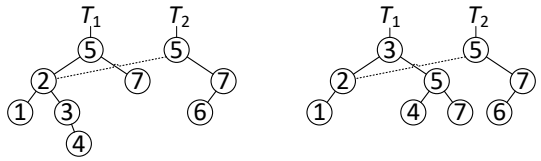
Update by path copying offers a cost-effective way to provide immutable snapshots of tree structures. Instead of mutating nodes in place, it copies the nodes affected by the update, applies changes to those copies, and relinks them to form a new root; with all untouched subtrees shared with prior versions. If the tree height is h, each update creates $O(h)$ new nodes. Each update yields a new



(a) Node4(64B) (b) Node16(3×64B)

(c) Node48(11×64B) (d) Node256(33×64B)

Fig. 2. Node Layouts of ART



(a) Insert Element 6 (b) Correctness Issue

Fig. 3. Example of Path Copying

version of the tree, and prior versions are never modified. Such a data structure is called a *purely functional* or *persistent* data structure [31] (not to be confused with persistence in external storage).

The path-copy technique naturally implements read-write separation. Query routines can be performed on a stable, immutable snapshot, completely isolated from ongoing updates. The practical importance of these snapshot-enabled indexes stems from their adoption in infrastructure systems like ETCD [4] — built on BoltDB [1] — which serves as the metadata store for Kubernetes [8].

Moreover, snapshots support accessing data at a specific version. Traversing from the new root node retrieves the updated data, while the previous data can still be accessed by traversing from the root node of the old version. If updates are processed in an **order-preserving** manner—meaning that earlier update requests correspond to earlier versions and each version incorporates the results of all prior updates—each version represents a specific historical moment, enabling queries about the state at that time.

Fig. 3(a) illustrates the creation of BST T_2 by inserting element 6 into T_1 via path copying. In T_2, nodes 5 and 7 along the search path are copied and node 6 is inserted, while the intact subtree rooted at node 2 is shared with T_1. Concurrent path copying demands meticulous care for correctness, as any subsequent changes to the original node will not be propagated to its copy. For example, a balanced BST may perform rebalancing after node 5 is copied in T_2. As shown in Fig. 3(b), node 5 in T_1 is modified, while its copy in T_2 remains unchanged, leading to an error.

Snap-Tree [22] employs an epoch-based mechanism to leverage multi-threading to provide a snapshot-enabled BST. Snap-Tree creates snapshots via lazy copying: it copies immutably-marked nodes (pushing the mark down to their child nodes) and updates other nodes in place. Within an epoch, updates are performed concurrently. When the clone operation is invoked, it waits for all ongoing updates to complete, and then marks the root node as immutable to create a snapshot and starts a new epoch. Since epochs are processed serially, Snap-Tree requires a sufficient epoch size to maintain efficiency, yielding coarse-grained versioning.

P-Tree [69] adopts a batch-based mechanism to create BST snapshots. P-tree batches multiple updates and, using a join-based BST [19], recursively partitions the batch into disjoint subsets that are processed in parallel by forked threads. Similar to Snap-Tree, P-Tree processes batches serially and requires a sufficient batch size to ensure efficiency, yielding coarse-grained versioning. At the same versioning granularity, P-Tree achieves higher efficiency than Snap-Tree because the threads operate on disjoint subtrees, eliminating synchronisation overhead.

Contreap [42] enables fine-grained concurrent path-copying updates on balanced BSTs in an order-preserving manner. Contreap utilises pipelining across tree layers, with each pipe handling nodes at a single depth level. Unlike deterministic balanced BSTs, treaps can pinpoint the rebalancing region during the search phase rather than upon backtracking. This allows Contreap to block subsequent pipes entering a rebalancing region, avoiding the issue in Fig. 3. As proven in [42], the expected height of the rebalancing subtree per update is $O(1)$. For a sequence of m updates, if $m = \omega(\log^2 n)$, where $n > m$ is the number of elements, its average height is $O(1)$ with high probability. Since a treap has an expected height of $O(\log n)$, the pipeline processes m updates in $O(\log n + m)$ time, in expectation and with high probability, with space cost $O(m \log n)$. Contreap further develops a hybrid execution strategy to improve practical performance by processing lower level nodes with different threads. Contreap provides efficient support for historical queries over predefined statistics. In our proposed ConTree, we also adopt a hybrid execution model. However, unlike Contreap's hard-coded treap-specific scheduler, our hybrid policy is tree-agnostic and applies to diverse tree structures and update rules.

In summary, existing multi-threaded path-copying solutions are largely tailored to BSTs, which are not ideal for general-purpose queries such as point lookups and range scans. Meanwhile, popular tree indexes lack efficient support for concurrent path copying. To fill this gap, we propose a general method for concurrent path copying in tree structures, instantiate it on ART and B+Tree, and address key challenges via two novel designs: AERT and BeTree.

2.3 Related Works

Temporal Database. Temporal databases have been studied extensively since the 1980s [13, 20, 27, 32, 68], supporting queries for various temporal patterns, such as time point/range queries, multi-temporal queries, and temporal joins. A series of database systems, such as Microsoft Immortal DB [53], Ganymed [59], Skippy [66], have been developed to support various applications. While these systems aim to support structured queries, we focus on per-update snapshots for key-value stores. Since our work enables the retrieval of the state associated with any key at any historical moment, it also supports queries over a time interval for accumulated metrics (e.g., counts), as the result can be computed from the difference between the states at the interval boundaries.

There also exists a line of research, such as Timeline Index [44], Period Index [16], and HINT [26], that focuses on duration queries of global information (e.g., orders within a period). Another line of research has proposed persistent sketches [67, 74] to support approximate aggregate queries (e.g., entropy) over time.

B-trees. Since its inception in the 1970s, the B-tree [28] accumulates an extensive body of research [35, 36] and has become the most dominant index structure in the field of databases. The B+Tree is the most common form of the B-tree. There are numerous variants of the B+Tree, such as B*Tree [17], Blink-tree [49], Bw-tree [52, 73], and Bp-tree [75] to accommodate different platforms and applications.

Although B-trees were originally designed for on-disk database systems, it is also widely used in in-memory databases [5, 9, 11]. Therefore, numerous studies [37, 38, 40, 48, 56, 57, 60, 61, 73, 75] seek to improve the performance of in-memory B-trees. These optimisations are orthogonal to our research. In addition, there is a broad body of work on enhancing B-tree performance in persistent memory [24, 58, 76] and flash memory [39, 41, 52, 64], as well as studies leveraging GPUs to accelerate B-tree performance [15, 80].

Tries. Tries are widely studied for in-memory databases, as well-engineered tries often outperform comparison-based indexes (e.g., B-trees) for lookups [14, 18, 46, 51, 77, 78]. Unlike hash tables, tries preserve key order and thus support range queries.

ART [51] is an efficient trie variant widely used as the default index in several in-memory databases [3, 12, 45]. A series of subsequent studies extend ART to non-volatile memory [55], disaggregated memory [54], and distributed systems [30, 79], and accelerates it with GPUs [47].

MassTree [56] is the default index structure in Silo [71]. It is a trie with a span of 8 bytes, where each node is a B+Tree. This structure could potentially be combined with our BeTree.

Non-Blocking Tree Structures. A line of work designs latch-free (non-blocking) trees to improve multi-threaded performance. NB-BST [33] is a latch-free BST, and PNB-BST [34] extends it with persistent updates, but neither maintains balance, limiting practicality. A general technique for non-blocking trees [23] targets efficient concurrent in-place updates. In contrast, ConTree targets concurrent path copying to efficiently support snapshots.

3 The ConTree Library

On modern platforms, exploiting multiprocessors via concurrency is a principal approach to improving performance. Nevertheless, constructing a concurrent tree index remains a complex task. In the context of concurrent path copying, ensuring consistency among versions while maintaining performance remains a major challenge.

In this paper, we present **ConTree**, which is a lightweight programming library that models the recursive update process as a state machine and provides encapsulated interfaces for state changes. We will introduce the details of these interfaces in Sec. 3.1.

Through ConTree, we offer a general framework that facilitates concurrent path copying to tree structures, compatible with heterogeneous node layouts and various update process. Notably, ensuring the correctness of concurrent path copying imposes inherent constraints on the update process. Once an update process adheres to these constraints, it can be systematically transformed into a state machine using general techniques, as we will discuss in Sec. 3.2.

As noted in Sec. 2.2, path copying provides immutable snapshots. Thus, query operations, such as lookups and scans, are *wait-free* and remain unchanged from their original serial implementations.

ConTree transparently enables *non-blocking* and *starvation-free* concurrent updates while guaranteeing *order-preservation*. The term **update** refers to the modify-if-exists, insert-otherwise operation, which is also known as an *upsert*. ConTree abstracts away low-level concurrency concerns—such as thread scheduling, memory ordering, and fine-grained synchronisation—allowing developers to focus on designing the tree index itself. Formally, we define the function `Update(key,F)`, where $F : K \times V_\perp \mapsto V_\perp$ is a function that returns a new value corresponding to the key and the current value. The behaviour of the update function is:

- if the element exists, update the value to $F(\texttt{key}, \texttt{val})$;
- otherwise, create a new node with value $F(\texttt{key}, \perp)$.

For simplicity, we use value to denote the return value of F in pseudocode and descriptions when no ambiguity arises. Deletion employs the soft-deletion technique of setting the value to $\perp$, as widely supported in systems such as HBase [6] and Kafka [7].

ConTree requires a `version` field in each node to indicate its creation version. In Sec. 4, we will introduce how to utilise this field, instead of locks or atomic variables, to ensure correctness under concurrency. In the rest of this section, we focus on implementing update functions with ConTree.

3.1 The State Machine Model

ConTree uses a state machine model to capture an update process with three components: states $\mathcal{S}$, contexts $\mathcal{C}$ and handlers $\mathcal{H}$.

The state set $\mathcal{S}$ includes two reserved states, `INIT` and `DONE`, which represent the initial and terminal states, respectively. Other states can be defined as needed by the data structure developer.

Algorithm 1: An Instance of Context

```
1 Class context :
2     uint   ver
3     K,V    key, val
4     node   *T_root, *T_past, *T_curr
5     args   ···    // customized arguments
```

During the update process, ConTree interacts with the developer through the context. Alg. 1 presents an example of such a context. Essentially, the context includes the version number, the key and value of the update, as well as several pointers to tree nodes. Specifically, the following three pointers are used: (1) $T_{\texttt{root}}$, which points to the root node of the current version; (2) $T_{\texttt{past}}$, which points to the parent node of the node in the old version that requires copying and updating; and (3) $T_{\texttt{curr}}$, which points to the parent node of the node that is copied and updated in the new version. In addition, the context may include additional parameters that are utilised during the update process of certain tree structures.

All states $\mathsf{q} \in \mathcal{S}$, except for DONE, have a corresponding handler $H \in \mathcal{H}$ which is a procedure that, given a context, returns the next state. As a side effect, H updates the current version of the tree by copying or creating new nodes as necessary, according to the state and context, and correspondingly updates $T_{\texttt{past}}$, $T_{\texttt{curr}}$ and the customised parameters in the context. ConTree enforces the following initialisation: before executing the handler for INIT state, $T_{\texttt{past}}$ in the context is set to the root node of the previous version.

3.2 Convert Recursive Process to State Machine

Tree updates are naturally *recursive*. To make concurrency explicit and implementation-agnostic, we express a standard recursive update by first modeling it as a state machine and then mapping each state to a handler. We use a BST as a purely didactic example; similar conversion can apply to other trees (e.g., ART, B+Tree).

Alg. 2 shows the recursive process for serially performing path copying on a BST. If the current node is nil, indicating that the requested key does not exist, a new node is created and returned (Line 5). Otherwise, the current node is copied (Line 6). The value of the copied node is updated if its key is equal to the requested key (Line 7). If not, the recursive process continues on either the left or right subtree, depending on the key comparison (Lines 8–11).

Alg. 3 shows the corresponding concurrent implementation using ConTree, where the object name for members of the context is omitted for simplicity. For example, $T_{\texttt{root}}$, $T_{\texttt{past}}$ and $T_{\texttt{curr}}$ are shorthand for ctx.$T_{\texttt{root}}$, ctx.$T_{\texttt{past}}$ and ctx.$T_{\texttt{curr}}$, respectively. For convenience, we refer to the CopyUpdate function as the **search function**. The first parameter, $T_{\&}$, of this function is *a reference to the node pointer* to be initialised. Developers can convert the recursive update process into a state machine as follows:

- Classify node data into two parts: (1) pointers to child nodes, and (2) all other data, which is referred to as the node content.
- In the search function, process the case directly and return DONE if it does not involve node copying (Lines 3–5).
- Each remaining case corresponds to an intermediate state. In the search function, copy only the content of the node, leaving the child pointers uninitialised (Line 6).
- If a case corresponds to a final step (e.g. REPLACE), copy the child pointers and return DONE in the corresponding handler.

Algorithm 2: Path-Copying Update to BST

```
1  Class node :
2    key, val    // node content
3    lch, rch    // child pointers
4  Function CopyUpdate(T_past, key, val) :
5    if T_past = nil then return NewNode(key, val)
6    T_curr := Copy(T_past)
7    if key = T_past.key then T_curr.val := val
8    else if key < T_past.key then
9      T_curr.lch := CopyUpdate(T_past.lch, key, val)
10   else
11     T_curr.rch := CopyUpdate(T_past.rch, key, val)
12   return T_curr
```

- Otherwise, determine the recursive direction in the search function based on the node content (Line 10). In the corresponding handler (e.g. SEARCH), copy the irrelevant child pointers (e.g., the opposite subtree in a BST), update $T_{\texttt{past}}$ and $T_{\texttt{curr}}$, and recursively invoke the search function.

To maintain consistency across versions in concurrent path copying, other threads may safely access the fields of a node only after the node becomes immutable. If some fields of a node depend on updates in descendant nodes, those fields must not be accessed until all relevant lower-layer updates complete. In this context, if each node is treated as an update unit, its child pointers inevitably depend on the child nodes. This dependency recurses upwards from the leaf, making the root the last node to become immutable during the update process. This issue makes updates degenerate into serial execution, since the root node for the next update must use information from the current root. Hence, in Alg. 3, we separate each node into content and child pointers, deferring initialising child pointers until child node update finishes (Lines 16, 20, 22).

Accordingly, if a node is no longer subject to modification during backtracking, it becomes safely accessible to other threads once the update operation recurses into its child nodes. Such a recursive process, where recursive calls are directly followed by returns, is termed **tail-recursion**. In a tail-recursive process, each call corresponds to a state transition. Hence, a tail-recursive process can be converted into a state machine by the approach outlined above.

Still, for some tree structure, the updates are inherently non–tail-recursive. If backtracking is needed only within a subtree, the subtree can be updated as a whole and exposed to other threads only after completion. This ensures the preceding search remains tail-recursive, and if the height of the subtree is bounded, as in the Contreap split, the preceding part can still proceed concurrently.

So far, we have considered only the case of creating a copy based on the content of the original node. However, in certain cases [42], the content of a subtree root may depend on information from the whole subtree. A node content may still depend on the content of its child nodes even when the update process is tail-recursive for certain tree structures [63]. In the next section, we present how ConTree extends the state machine to support such cases.

4 The ConTree Implementation

In this section, we show how ConTree enables efficient concurrent path copying without locks or atomic operations such as CAS. Both context and node types are aligned to cache lines (typically 64/128 bytes), following established practice to prevent false sharing [21].

Algorithm 3: Path-Copying Update to BST with ConTree

```
1  Class context : ···, dir
2  Function CopyUpdate(T_&, ctx) :
3    if T_past = nil then
4      T_& := NewNode(key, val)
5      return DONE
6    T_& := CopyContent(T_past)
7    if key = T_past.key then
8      T_&.val := val
9      return REPLACE
10   dir := if key < T_past.key then 0 else 1
11   return SEARCH
12 Handler INIT(ctx) :
13   q := CopyUpdate(T_root, ctx), T_curr := T_root
14   return q
15 Handler REPLACE(ctx) :
16   T_curr.ch[0] := T_past.ch[0], T_curr.ch[1] := T_past.ch[1]
17   return DONE
18 Handler SEARCH(ctx) :
19   T_curr.ch[1−dir] := T_past.ch[1−dir]
20   T_past := T_past.ch[dir]
21   q := CopyUpdate(T_curr.ch[dir], ctx)
22   T_curr := T_curr.ch[dir]
23   return q
```

4.1 Dependency Resolution

The **worker pattern** is a straightforward concurrent scheme: each thread, referred to as a **worker**, repeatedly invokes handlers on a received update request until its state becomes `DONE`. To ensure correctness when multiple workers independently process different updates, it is essential to ensure that the node content and child pointers used by the handler are initialised prior to each invocation.

Alg. 4 shows how a worker executes a state machine in ConTree. We use **stage** to denote the depth of the node being processed in the current version. ConTree maintains a global buffer to track the stage of each ongoing update. A worker uses the `SyncStage` primitive to advance the stage of the update it is executing. The primitive `WaitStage` blocks until the stage of the referenced version is at least the specified stage. Recall that, after a tail-recursive update process begins to access a node at a certain depth, no further modifications are made to the shallower nodes on the search path. Therefore, to ensure correctness, it suffices to place memory barriers within `SyncStage` and `WaitStage` to prevent instruction reordering.

There are several concepts in Alg. 4 that need to be clarified:

- The **comitted** version (`cv` in Lines 2, 6, 8). Updates are *order-preserving*, meaning each version fully incorporates all prior updates, yet updates on different keys may complete *out-of-order*. For example, in Fig. 3(a), while node 5 in T_2 must be created after that in T_1, node 6 in T_2 can be inserted before node 4 in T_1 because the remaining search paths are disjoint. Accordingly, we define an update as committed only when it and all prior updates have completed. The stage of a committed version is regarded as ∞.

Algorithm 4: Execute a State Machine with a Worker

```
1  Procedure Execute(ver, key, val) :
2    WaitStage(ver−1, 1), cv := CommittedVersion()
3    T_past := Root(ver−1), q := INIT, stage := 1
4    while q ≠ DONE do
5      repeat
6        if cv < T_past.ver then
7          WaitStage(T_past.ver, stage+D[q])
8          cv := CommittedVersion()
9        q := H[q](ctx)
10     until FA[q]
11     SyncStage(ver, stage), stage := stage+1
12   Finish(ctx)
```

- The **downward dependency** (`D[q]` in Line 7). `D[q]` takes values 0, 1, or ∞. The default value is 0, indicating that the handler only utilises contents on the current search path. Developers can set it to 1 or ∞ when a handler needs to access a child node content or the entire subtree to update current node, respectively.
- The **forge ahead flag** (`FA[q]` in Line 10). By default, `FA[q]` is `TRUE`, meaning the handler copies and updates a node and advance the version stage. Developers can set it to `FALSE` to declare a handler as a sub-step of the current stage, facilitating the decomposition of complex handlers—particularly for subcases with different downward dependencies.

At the beginning, the worker waits for the previous version to reach stage 1 to ensure its root node is initialised (Line 2). Then, the worker sets T_{past} to the root node of the previous version (Line 3). After that, the worker repeatedly checks dependencies (Lines 6–8), invokes the handler (Line 9), and advances the stage (Line 11) until the status becomes `DONE`. This process shows that:

Lemma 4.1. *Alg. 4 is non-blocking and starvation-free.*

As discribed in Sec. 2.2, a data item is written only once during its initialisation. Therefore, threads will not compete to write data, preventing starvation. Instead, writers exhibit data dependencies. Since data from newer versions unidirectionally dependent on older ones, the earliest incomplete update is always eligible for execution, which guarantees the non-blocking progress.

When a worker encounters a node created in a committed version, it will skip dependency checks for the remaining steps. The correctness of this optimisation relies on the following property:

Property 4.2. *In a tree updated by path copying, a parent node is always created in a version that is no earlier than its child nodes.*

Specifically, the child node on the search path shares the same version as it parent node, while the versions of other child nodes are earlier than that of the parent node. Consequently, if a node is created in a committed version, all nodes in its subtree also belong to completed versions. Therefore, dependency checks can be safely omitted in subsequent steps.

4.2 Reduce Synchronisation Overhead

The worker pattern supports flexible concurrency and, in principle, supports arbitrarily many workers. In practice, however, concurrency is constrained by the tree's structure.Specifically, the workers actually execute one by one when setting the root node (Alg. 4 Line 2). Moreover, when the number of possible search paths is limited, the search paths of different workers may overlap.

In this scenario, the overhead incurred by frequent synchronisation among workers may outweigh the benefits of concurrent execution, thereby becoming a performance bottleneck.

To reduce synchronisation overhead, we *adopt* the hybrid execution idea of Contreap [42]. In contrast to the hard-coded, treap-specific hybrid scheduler of Contreap, we propose a tree-agnostic scheduling policy. Our hybrid execution model is configurable and applicable to diverse trees and update rules.

Our hybrid executor uses a parameter $d > 0$ to partition the tree: nodes with depth $< d$ form the **upper part** and the rest the **lower part**. The upper part runs in a **pipeline pattern**—each layer is handled serially by a dedicated **pipe**—to reduce contention on hot nodes. Below depth d, it switches to a *worker* pattern so independent workers can proceed on mostly disjoint paths. This hybrid is a *pluggable policy* in ConTree, not tree-specific, so only d needs tuning per index.

Alg. 5 presents the components of the hybrid execution framework, which consists of the following four roles:

- The first pipe (Lines 8–13), denoted as the entry, receives submissions and assigns each request a monotonically increasing version. It then initialises the context, processes the root node via the handler, and forwards the result to the next pipe.
- An inner pipe (Lines 14–17) repeatedly receives the state and context from the preceding pipe, processes an internal node by invoking the handler, and forwards the result to the next pipe.
- The last pipe (Lines 18–20), denoted as the exit, maintains a pool of idle workers and assigns the state and context to an available worker upon receipt from the preceding pipe.
- Workers continuously receive updates that have been processed by the pipeline and execute remaining steps until completion.

In a pipeline, each pipe processes update steps sequentially within its designated layer. Furthermore, pipes eliminate stage synchronisation. Instead, downward dependencies are addressed via the direct use of the committed version only if inevitable (Line 4). Building on these principles, an inner pipe, adhering to a single-producer-single-consumer pattern, exclusively receives data from its preceding pipe and transmits data to its next pipe. On modern architectures, this feature introduces substantially lower overhead in comparison to the synchronisation primitives utilised by concurrent workers. Nevertheless, the low synchronisation cost achieved by the pipeline pattern comes at the expense of certain trade-offs.

On the one hand, the pipeline pattern limits the degree of concurrency, as each pipe executes update steps sequentially within a given layer. As discussed in Sec. 4.1, when the subtrees assigned to different workers are mutually disjoint, dependency checks may be safely omitted by inspecting the committed version, thereby reducing synchronisations. Consequently, as the probability of update overlap diminishes with increasing depth, the benefit of reduced synchronisation overhead provided by the pipeline becomes outweighed by the cost of its limited concurrency.

On the other hand, the efficiency of the pipeline is determined by the performance of the slowest pipe. If a particular pipe frequently executes handlers with high overhead, or encounters downward dependencies that result in prolonged waiting, it becomes a performance bottleneck and diminishes the efficiency of the entire pipeline. In contrast, when workers operate without overlap, they can execute independently and be free from mutual interference.

We summarise the following strategies about the pipeline depth and the handlers in pipeline for achieving high update efficiency:

Principle 4.3 (Pipe Cardinality and Handler Discipline). *Use the minimum number of pipes required to keep worker overlap probability acceptably low. Avoid high-overhead handlers and any downward dependencies within pipeline.*

Algorithm 5: Components of Hybrid Framework

```
1  Function ForgeAhead(q, ctx) :
2    if q = DONE then return q
3    repeat
4      if D[q] ≠ 0 then WaitCommit(T_past.ver)
5      q := H[q](ctx)
6    until FA[q]
7    return q
8  Pipe Entry Loop :
9    ver := ver+1, Receive(key, val)
10   ctx := {ver, key, val, T_past = T_last}
11   q := ForgeAhead(INIT, ctx)
12   T_last := ctx.T_root
13   {q, ctx} → P_1
14 Pipe Inner(i) Loop :
15   {q, ctx} ← P_i
16   q := ForgeAhead(q, ctx)
17   {q, ctx} → P_{i+1}
18 Pipe Exit Loop :
19   {q, ctx} ← P_d
20   SendToFreeWorker(q, ctx)
21 Thread Worker Loop :
22   ReceiveFromPipeline(q, ctx)
23   ··· // execute Lines 4–12 in Alg.4
```

4.3 Enable Global Operations via Checkpoints

In Sec. 3, we described how to utilise ConTree to implement concurrent path copy, while in the earlier part of Sec. 4, we discussed how ConTree transparently supports concurrent path copy. However, there are cases where certain maintenance cannot be performed during path copy. For example, developers may prefer reconstructing the entire tree or certain subtrees when the number of soft deletions, either globally or locally, exceeds a given threshold.

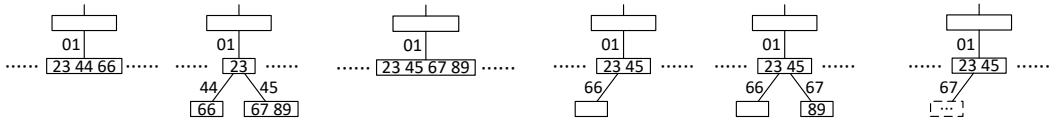
ConTree allows developers to configure a checkpoint interval and include a global-operation field gop into the context. Upon update completion, ConTree retains this field if its value is not NONE. At the end of each cycle, if the record is not empty, the entry pipe waits for all prior updates to complete, then passes these records along with the latest tree version to a customised method.

It is important to note that if a checkpoint is triggered, we must wait for all preceding updates to complete. Besides, a checkpoint will block subsequent updates until the customised global function has finished execution. We should follow the following design principle to keep update efficiency:

Principle 4.4 (Checkpoint and Global Function Discipline). *Avoid frequent checkpoint and high-overhead global function.*

5 Tree Structures based on ConTree

In Sec. 3, we demonstrate how developers can leverage ConTree to implement concurrent path copying, while avoiding the intricacies of concurrent programming. However, in contrast to its serial counterpart, the concurrent implementation requires the update process to be tail-recursive.



(a) Prefix Mismatch (b) Element Found (c) Child Not Found (d) Child Found

Fig. 4. Updating Key ··· **01 23 45 67 89** in ART

Algorithm 6: Update Process for ART

```
1  Class context : ···, ofs
2  Function CopyUpdate(T_&, ctx) :
3    l := PrefixMatch(T_past, key, ofs)
4    if l < T_past.len then
5      T_trunc := Truncate(T_past, l)
6      T_new := NewLeaf(key,val,ofs+l+1)
7      T_& := Fork(key, ofs, l, T_trunc, T_new)
8      return FORK
9    ofs := ofs+l
10   if ofs=strlen(key) then
11     T_& := CopyUpdateValue(T_past, val)
12     return REPLACE
13   k := PartialKey(key, ofs), ofs := ofs+1
14   if ChildExist(T_&, k) then
15     T_& := CopyContent(T_past)
16     return SEARCH
17   T_new := NewLeaf(key,val,ofs)
18   if Full(T_past) then
19     T_& := Grow(T_past), SetChild(T_&, k ,T_new)
20     return INSERT_GROW
21   else
22     T_& := CopyContent(T_past), SetChild(T_&, k ,T_new)
23     return INSERT
```

Furthermore, as discussed in Sec. 4, it is desirable for the update process of the tree structure to satisfy certain design principles in order to enhance practical efficiency. Therefore, designing a tree structure that supports efficient path copying remains a significant challenge.

In this section, we analyse the bottlenecks associated with concurrent path copying in ART and B+Tree, and propose **AERT** and **BeTree** as respective solutions. Notably, as the handlers primarily set pointers and invoke the search function recursively, our discussion will focus on the search function. Due to limited space, the details on the handlers are provided in our technical report [2].

5.1 Adaptive Equalised Radix Tree

Fig. 4 shows the four cases that may arise when updating an ART:

(a) Prefix mismatch: Fork to create a new internal node with the common prefix truncated from the current node. Attach the current node and a new leaf to the new internal node as its child nodes, with the leaf storing the mismatched key suffix.

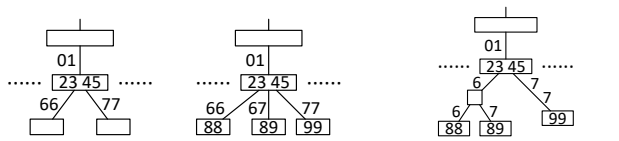
(b) Prefix match the entire key suffix: The current node is the target. Update the value for this key in the ART index.

Algorithm 7: Additional Case for AERT

```
1 Function CopyUpdate(T_&, ctx) :
2   ...
3   if Full(T_past) and Size(T_past) = 16 then
4     T_& := SplitInsert(T_past, k)
5     return INSERT_SPLIT
```

(a) Half Node4 (64 B) (b) Half Node16 (3×64 B)

Fig. 5. Half Node Layouts



(a) Child Nodes Exceed 16 (b) Split into Half Nodes

Fig. 6. Updating Key ··· **01 23 45 67 89** in AERT

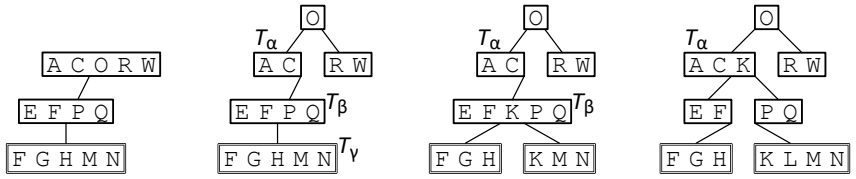
(c) Prefix match but no matching child: Create a new leaf node containing the element. Expand the current node if it is full. Insert the new leaf node as a child node of the current node.
(d) Prefix match with matching child node: Recursively proceed the update to that child node.

Cases (a) and (c) indicate a missing element, completing the update with a leaf insertion. The update recurses only in Case (d) and does not modify the the node after recursively visiting its child node. Hence, the update process of ART is tail-recursive.

As outlined in Sec. 3.2, each case arising in the update process is mapped into a distinct intermediate state. In the case of ART, node insertion can be further classified into two sub-cases, depending on whether the current node is full. Hence, mapping the ART update process as a state machine yields five intermediate states: `FORK`, `REPLACE`, `INSERT`, `INSERT_GROW`, and `SEARCH`.

Alg. 6 shows the search function for ART. An offset field, denoted as `ofs`, is introduced into the context to specify the starting position of the key substring to be matched. Correspondingly, the `INIT` handler sets the offset to 0 and invokes the search function to update the root node. If Case (d) occurs (Lines 14–16), it returns `SEARCH`, thereby prompting the next stage to continue the search. The search process terminates if other cases arise. For Cases (a) (Lines 4–8) and (b) (Lines 10–12), the search function returns `FORK` and `REPLACE`, respectively. Case (c) is further divided into two subcases. If the current node is full—for example, when a Node4 already has four child nodes—the search function responds by creating an expanded node, inserting the new leaf node, and returning `INSERT_GROW` (Lines 18–20). Otherwise, the search function copies the current node, inserts the new leaf node, and returns `INSERT` (Lines 21–23).

Performance Bottleneck of Path Copying in ART. Unfortunately, this straightforward implementation performs poorly in practice. Regardless of the number of pipes and workers, the throughput remains relatively low. Recall that, ART nodes have the same logical fanout of 256, corresponding to one byte, and it employs adaptive node types to optimise for nodes with varying actual fanouts. The largest node type, Node256, is much larger than the smallest not type, Node4. Therefore, copying a large node is much more costly than copying a small node. Empirically, copying a Node256 requires approximately 1.5 microseconds, which is about 5, 20, and 45 times slower than copying a Node48, Node16, and Node4, respectively. This is problematic because, for ARTs



(a) (b) (c) (d)

Fig. 7. Inserting Element 'K' and 'L' into B+Tree

managing large-scale data, the upper part are typically comprised of large nodes. Consequently, Alg. 6 incurs high overhead in the upper part, which in turn forms the main bottleneck.

Adaptive Equalised Radix Tree (AERT). To address this issue, we employ **half nodes**, with a logical fanout of 16 and thus represent half a byte. Accordingly, nodes with a logical fanout of 256 are called regular nodes. By introducing half nodes, we can adjust the logical fanout of nodes to prevent the creation of overly large nodes, thus helping to equalise node sizes. We refer to the ART with half nodes as the AERT. In AERT, a Node16 is split into two layers of half nodes, rather than expanding to a Node48. We refer to the half node containing the leading half-byte as the **top-half**, and the half nodes containing the following half-byte as the **bottom-halves**.

Analogous to regular nodes, we introduce Half4 and Half16 to accommodate varying fanouts. Their layouts are shown in Fig. 5: Half4 resembles Node4/Node16, storing 4 partial keys and 4 child pointers, while Half16 follows the Node256 layout in original ART, using a 16-entry lookup table indexed by half-byte keys.

When a child node with a partial key of 67 is inserted into a node, assuming that the node already has 16 child nodes, it will grow into a Node48 in ART as shown in Fig. 6(a). By contrast, in AERT, the split process is invoked instead of growing the node, as shown in Fig. 6(b). The current node then becomes the top-half, and its child nodes are partitioned into different bottom-halves according to the leading half-byte of their keys. Since a node is split if it has 17 child nodes after inserting a new child node, by the pigeonhole principle, there must be at least two different leading half-bytes. As a result, the top half must contain at least two child nodes. However, for some leading half-bytes, there may be only one corresponding following half-byte, such as the child node with a partial key of 77 in Fig. 6. AERT employs tagged pointers to prevent the creation of a degenerate bottom-half. Specifically, a half-byte is embedded in the lowest 4 bits of the pointer. As all nodes are 64-byte aligned, the lowest 6 bits of the pointer are guaranteed to be 0, allowing the use of these bits for tagging without affecting pointer correctness.

Alg. 7 shows the update function for AERT, omitting cases already covered in Alg. 6. We add a new subcase under Case (c): If a Node16 is full, the search splits it, inserts the leaf node into the appropriate bottom-half, and returns `INSERT_SPLIT`. Accordingly, in the next stage, the child pointers from the original node are copied to the new bottom-halves, and the update process is completed by inserting the new node into the corresponding bottom-half.

5.2 B-Elastic Tree

A typical B+Tree implementation first recursively locates the relevant leaf node to insert the new element, and resolves any overflow during the backtracking phase. Clearly, such an update process is not tail-recursive. A top-down update process [63] for the B+Tree can be achieved by temporarily allowing non-leaf nodes to overflow, delaying the split until the node is revisited. Fig. 7 shows an example of insertion process of a B+Tree with an order of 3:

Algorithm 8: Update Process for B+Tree

```
Class context : ···, idx
Function CopyUpdateRoot(T_&, ctx) :
  if Overflow(T_past) then
    T_& := SplitRoot(T_past), idx := 0
    return SEARCH_SPLIT
  T_& := CopyContent(T_past)
  return SEARCH_ROOT
Function CopyUpdate(T_&, ctx) :
  idx := Find(T_past, key)
  T_&.ch[idx] := CopyContent(T_past.ch[idx])
  if IsLeaf(T_&.ch[idx]) then
    UpdateLeaf(T_&.ch[idx], key, val)
    if Overflow(T_&.ch[idx]) then SplitLeaf(T_&, idx)
    return DONE
  if Overflow(T_&.ch[idx]) then
    SplitChild(T_&, idx)
    return SEARCH_SPLIT
  return SEARCH
```

Algorithm 9: Update Process for BeTree

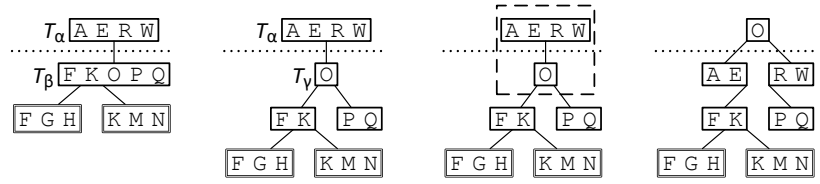
```
Function CopyUpdateUpper(T_&, ctx) :
  T_& := CopyContent(T_past)
  idx := Find(T_past, key)
  if IsBoundary(T_&) then return SEARCH_ELASTIC
  return SEARCH_UPPER
Function CopyUpdateElastic(T_&, ctx) :
  ··· // Similar to CopyUpdateRoot in Alg.8
Function CopyUpdateLower(T_&, ctx) :
  ··· // Similar to CopyUpdate in Alg.8
```

(a) Before accessing the root, we detect its overflow (Fig. 7(a)) and therefore split it (Fig. 7(b)). We then search the new root to locate its child T_α. Since a node cannot overflow immediately after a split, we directly search T_α and proceed to its child T_β.

(b) Before accessing T_β, we check for overflow. We then search it and reach leaf T_γ since T_β does not overflow. Insertion causes T_γ to overflow, triggering a split and inserting the pivot into T_β, which consequently overflows. We tolerate this temporary overflow and complete the update (Fig. 7(c) after inserting 'K').

(c) When inserting 'L', we find no overflow at the root or T_α. However, T_β is found overflowed before access and is split, with its middle key promoted to T_α. We then search the newly split node to find the target leaf. With no overflow at the leaf after insertion, the update completes (Fig. 7(d) after inserting 'L').

The variant of B+Tree [63] is slightly different from the typical structure, in that it omits leaf-to-leaf links. This is because the path-copying process cannot efficiently maintain the linked list of leaf nodes. Scanning therefore requires backtracking to parent nodes in order to find successive leaves. For in-memory scans, this variant experiences a slight ($\sim$ 6%) performance degradation.



(a) Overflow (b) Split (c) Collect (d) Rebuild

Fig. 8. Update Process of BeTree

Alg. 8 shows the search function for B+Tree, including three intermediate states: `SEARCH_ROOT`, `SEARCH`, and `SEARCH_SPLIT`. An index field, denoted as `idx`, is introduced into the context to indicate the child node to be accessed in the next step. Specifically, if the child node is split, the index field indicates the position of the key that is promoted during the split. In the process of updating a B+Tree, if the child node to be accessed overflows, a key is promoted into the current internal node. This indicates that the content of a non-leaf node in a B+Tree is determined by the content of its child nodes, resulting in a downward dependency `D[q]` of 1 for these intermediate steps.

The `INIT` handler invokes the search function `CopyUpdateRoot` to initialise the root node. For simplicity, corner cases where the root node is either empty or a leaf node are omitted. If the root node overflows (Lines 3–5), this corresponds to the case when the element 'K' is inserted. In this case, the search function splits the current root node and creates a new root node, and, since the content of the new root node has been determined, advances the search process to the next stage by returning `SEARCH_SPLIT`. If the root node does not overflow (Lines 6–7), the search function copies the content of the root node and returns `SEARCH_ROOT`. In this case, the content remains undetermined, as it will be modified in the event that the next child node to be accessed overflows. Therefore, the forge ahead flag `FA[SEARCH_ROOT]` should be set to `FALSE`.

Intermediate states are processed using the search function `CopyUpdate`. The search function first locates the child node corresponding to the key and then copies its content to create a new child node (Lines 9–10). If this child node is a leaf node (Lines 11–14), the search function directly updates it, resolves leaf node overflow as necessary, and returns `DONE`. Otherwise (Lines 15–18), the search function determines whether to adjust the content of the current node based on whether the child node overflows, and returns either `SEARCH_SPLIT` or `SEARCH` accordingly.

Performance Bottleneck of Path Copying in B+Tree. As introduced in Sec. 4, ConTree resolves downward dependencies on the pipeline side by waiting for the commitment of the given version. Since all intermediate states of the B+Tree exhibit downward dependencies, this causes the update process to degrade into serial execution. Given that prior work [42] has shown the pure worker pattern to be inefficient in practice, we further attempt to specify a pipeline that satisfies the dependencies of child nodes without blocking and waiting for the previous version to commit.

However, addressing downward dependencies within the pipeline still incurs substantial overhead. Consequently, as we will report in the experiments, the optimal pipeline configuration for B+Tree employs only two stages. In this case, after the entry pipe processes the root node, the remaining steps are directly forwarded to the worker through the exit pipe. This shallow pipeline is insufficient to effectively reduce the likelihood of data overlap among workers. The absence of an appropriate trade-off between the extra synchronisation overhead and the data conflicts constitutes the main bottleneck, thereby constraining the performance of concurrent path copying in the B+Tree.

B-Elastic Tree (BeTree). Motivated by this limitation, we introduce **elastic nodes** to eliminate downward dependencies in the upper part, thus mitigating the performance bottleneck. Fig. 8 shows an example of a BeTree, where the area above the dotted line is the upper part, and the area

Algorithm 10: Build B+Tree from Sorted Elememts

```
1  Function Build(b, n, elems) :
2    if n < 2·b then return NewLeaf(elems, 0, n)
3    m := n/b, NewArray(keys, m), NewArray(nodes, m)
4    for i ∈ range(m) do
5      keys[i] := elems[i·m].key
6      nodes[i] := NewLeaf(elems, i·m, (i+1)·m)
7    return BuildInner(b, m, keys, nodes)
8  Function BuildInner(b, n, keys, nodes) :
9    if n < 2·b then return NewInner(keys, nodes, 0, n)
10   m := n/b, NewArray(keys′, m), NewArray(nodes′, m)
11   for i ∈ range(m) do
12     keys′[i] := keys[i·m]
13     nodes′[i] := NewInner(keys, nodes, i·m, (i+1)·m)
14   return BuildInner(b, m, keys′, nodes′)
```

below is the lower part. Elastic nodes are defined as the uppermost nodes in the lower part, while their parent nodes—that is, the lowest nodes in the upper part—are termed **boundary nodes**. As depicted in Fig. 8(a), when the elastic node T_β overflows, it is split directly, in a manner similar to the root split in a B+Tree. This split creates two new child nodes and a parent node T_γ. As shown in Fig. 8(b), T_γ becomes the new elastic node, and is attached as a child to the boundary node T_α, thereby avoiding any impact on the contents of T_α. The child nodes are converted into regular internal nodes. We term the layers containing such child nodes the **cushion layers**. Note that, cushion layers are defined per lower tree and their presence and count may vary, depending on how many elastic-node splits have occurred in that particular lower tree. For example, in Fig. 8(b), in the lower tree rooted at T_γ, the nodes with keys 'FK' and 'PQ' reside in a cushion layer.

With the help of elastic nodes, we employ distinct update procedures for the upper part and the elastic layer. This leads to two additional intermediate states: `SEARCH_UPPER` and `SEARCH_ELASTIC`.

Alg. 9 shows the search functions for BeTree. During the update process, the entry pipe handles the `INIT` state, invoking the search function `CopyUpdateUpper` to initialise the root node. Subsequently, each inner pipe utilises this search function to process the `SEARCH_UPPER` state, continuing until the final inner pipe encounters a boundary node and returns the `SEARCH_ELASTIC` state. The remaining steps of the update will be delegated to a worker for further execution. It is evident from Alg. 9 that, in a BeTree, the search process in the upper part does not exhibit downward dependencies, and thus, ConTree can be effectively utilised to implement concurrent path copy.

The BeTree improves the efficiency of processing within the upper part. Elastic nodes intercept the effects of all updates. We refer to a subtree rooted at an elastic node as a **lower tree**. Each lower tree resembles a B+Tree and evolves independently during updates. Accordingly, the upper part becomes static. As updates accumulate, the heights of the lower trees may diverge, resulting in leaf nodes of the BeTree appearing at varying depths. Excessive updates to a particular lower tree may cause its height to far exceed that of others, rendering the BeTree unbalanced and adversely impacting performance. To restore balance and thus ensure efficiency, we leverage the checkpoint mechanism of ConTree as introduced in Sec. 4.3 to perform on-demand reconstruction. Then, we will present our partial reconstruction algorithm.

Algorithm 11: Partial Rebuild for BeTree

```
1  Function Collect(T, keys, nodes) :
2    if IsUpper(T) or strain(T)> 0 then
3      for i ∈ range(size(T)) do
4        if i > 0 then Append(keys, T.keys[i−1])
5        Collect(T.ch[i], keys, nodes)
6    else Append(nodes, T)
7  Function Rebuild(T, b, d) :
8    keys := EmptyArray(), nodes := EmptyArray()
9    Append(keys, ⊥), Collect(T, keys, nodes)
10   T′ := BuildInner(b, size(keys), keys, nodes)
11   return SetBoundary(T′, d)
```

Partial Reconstruction. Alg. 10 shows the process of building a B+Tree from elements sorted by key. For simplicity, we assume that n is a multiple of b at each level; otherwise, the remaining elements or child nodes are evenly assigned among nodes.

Building a tree with n records yields a $O(n)$ computation cost, which is prohibitively high for large-scale data. To address this issue, we propose a partial rebuild approach which involves *only a limited subset of nodes*. This method restores the BeTree to a fully balanced state, ensuring that all leaf nodes reside at the same depth.

In a BeTree built by Alg. 10, all lower trees have uniform height. After several updates, different lower trees may undergo different numbers of elastic-node splits, resulting in varying numbers of cushion layers and, consequently, different heights. However, the property holds that the height of each subtree rooted at a bottommost cushion node equals that of any lower tree that has never split. Thus, the reconstruction scope can be limited to nodes across *the upper part*, *the elastic layer* and *the cushion layers*.

For clarity, we introduce the term **elastic strain** to describe a property of nodes in the elastic and cushion layers. In a newly built BeTree, all elastic nodes are initialised with elastic strain 0. A node split results in two peer nodes that retain the strain of the original node. Splitting an elastic node produces a new elastic node, whose strain is increased by 1. For example, the elastic node T_β in Fig. 8(a), has a strain of 0. Split T_β creates two peer nodes at strain 0 and the new elastic node T_γ at strain 1. If T_γ undergoes a subsequent split, the resulting elastic node will then have a strain of 2.

Alg. 11 shows how BeTree rebalances via partial rebuild. Define the partial set $\mathcal{T}$ as the set of nodes with zero elastic strain. The first step is to collect nodes in $\mathcal{T}$, along with their corresponding keys. The collection process is performed recursively, starting from the root node and proceeding downwards to nodes in the lower part with non-zero elastic strain, ultimately collecting those nodes whose elastic strain is zero (Line 6). The set of keys relevant to the rebuild process comprises all keys contained in the nodes traversed during the collection of subtrees (Line 4). Note that, the smallest key cannot be obtained during collection and a void key $\perp$ is inserted before the collection (Line 9). This is valid since the first key is not needed during the rebuild. Fig. 8(c) shows an example of the collection process, where the dashed box frames the collected keys, and the outgoing edges from the box point to the collected subtrees.

Since all subtrees rooted at nodes in $\mathcal{T}$ have equal height, restructuring the nodes in $\mathcal{T}$ restores balance to the entire tree structure. When the collection is completed, we invoke the `BuildInner` function, as shown in Alg. 10, to partially restructure the upper part of the tree. Then, we reinitialise boundary nodes and elastic nodes (Line 11). Fig. 8(d) illustrates the tree after rebalancing. Since the higher levels of the tree contain far fewer nodes than the lower levels, a partial rebuild is

substantially less costly than a complete rebuild. This significantly reduces the execution time of a checkpoint upon triggering. Next, we introduce the trigger conditions.

The Frequency of Reconstruction. We adopt a simple and practical strategy: when an elastic node of strain 2 is created, the SEARCH_ELASTIC handler sets gop field in the context. As noted in Sec. 4.3, this triggers the global function to rebuild the tree at the end of the current cycle. Since each lower tree in BeTree is a B+Tree, we estimate the rebuild frequency by lower-bounding the number of insertions needed for a B+Tree to grow two levels.

Theorem 5.1 (Rebuilds are infrequent). *For a lower B+Tree of order b and height h, triggering the second root split requires amortised $\Omega(b^{h+2})$ insertions. Hence, BeTree rebuilds are rare when lower trees have moderate height.*

Proof. For a B+Tree of order b, we define the potential of each non-root node T as $\phi(T) = b^{\mathsf{h}(T)}(\mathsf{sz}(T)-b)$, where $\mathrm{sz}(T)$ denotes the number of child nodes if T is a non-leaf node, or the number of records if T is a leaf node. Clearly, $\phi(T) \geq 0$. The total potential of a tree is defined as the sum of the potentials of all non-root nodes.

We use T_0 to denote the root node and ignore the trivial case where T_0 is a leaf node. When a new element is inserted, the number of elements in a certain leaf node increases by 1, resulting in the total potential increases by 1. A non-root node T splits when $\mathsf{sz}(T) = 2b$, at which point its potential is $\phi_\Delta = b^{\mathsf{h}(T)+1}$. The split produces two nodes, T_l and T_r, each with $\mathsf{sz}(T_l) = \mathsf{sz}(T_r) = b$, and thus each has potential 0. Meanwhile, for the parent node of T, referred to as T_p, $\mathsf{sz}(T_p)$ increases by 1 after the split. When $T_p \neq T_0$, $\phi(T_p)$ increases by ϕ_Δ, thus the total potential remains unchanged. Otherwise, the total potential decreases by ϕ_Δ. Consequently, since the total potential must be non-negative, at least *amortized* $b^{\mathsf{h}(T_0)}$ insertions are required to add a new key to the root.

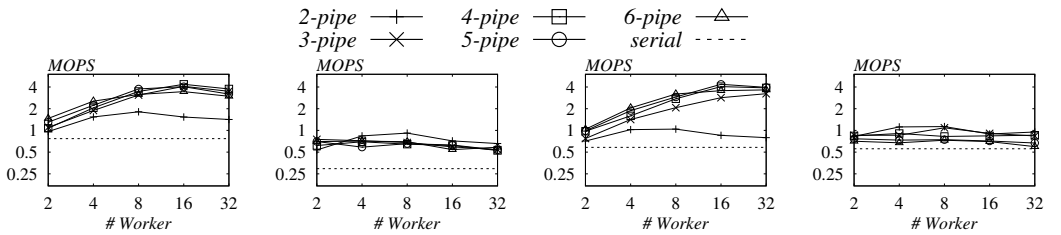
We omit the insertions required to trigger the first split of the root node. After the split, the new root has a height of $\mathsf{h}(T_0) + 1$ and contains only two child nodes, which requires at least $2(b-1) \cdot b^{\mathsf{h}(T_0)+1}$ insertions to trigger the second split of the root node. □

In practice, the upper part is typically very small, thus the lower trees tend to have a sufficient height. Therefore, under our strategy, the frequency of BeTree rebuilds is extremely low. Furthermore, under this strategy, the height difference between lower trees is typically no more than 2, which ensures the efficiency of updates and queries in the BeTree.

6 Experiment

In this section, we conduct several sets of experiments to show the advantages of our solution. All experiments are performed on a server with 128 GB memory, two Intel Xeon Gold 5320 processors, each offering 26 cores with hyper-threading. All code [2] is implemented in C++20 and compiled with full optimisation using GCC. We use mimalloc [50] for multithreaded memory management.

In our experiments, we include several methods. The first two methods are the ART that directly applies ConTree library (Ref. to Sec. 5.1) and our proposed AERT to gain better concurrency using the half-node optimisation. The next two methods are the B+Tree using ConTree with a customised pipeline (Ref. to Sec. 5.2) and our proposed BeTree to gain better concurrency by introducing elastic nodes. We also adopt Contreap [42] as a baseline. The test data are generated from YCSB [29], a widely adopted benchmarking framework for evaluating the performance of general-purpose key-value stores. All experiments are conducted over 10 independent runs, and the average results are reported. In each run, we first generate 100 million records as the initial dataset, followed by the execution of 1 million operations using 32 clients.



(a) ConTree AERT (b) ConTree ART (c) ConTree BeTree (d) ConTree B+Tree

Fig. 9. Update Performance and Scalability

Update Performance and Scalability. We first evaluate the update performance of all four methods with path-copying update implemented with our ConTree library by performing 1 million insertions across various pipeline and worker configurations. For comparison, the serial counterpart of each data structure, where all path-copying updates are processed by a single process, is included as a baseline represented by the dashed line. This allows us to illustrate the performance benefits of concurrent updates.

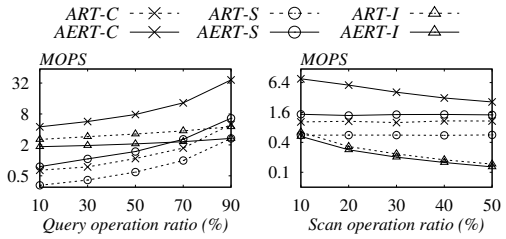
Fig. 9(a) reports the throughput of AERT in million operations per second (MOPS). AERT, with 4 pipes and 16 workers, achieves a throughput approximately 5.2× that of its serial counterpart. In comparison, as shown in Fig. 9(b), concurrent ART achieves a peak throughput of roughly 3.1× that of the serial one, with 2 pipes and 8 workers. A comparison between Fig. 9(a) and Fig. 9(b) further demonstrates that AERT achieves significantly higher update throughput than ART in serial execution, primarily due to the considerable overhead incurred by copying large nodes in ART. By introducing the half nodes, AERT effectively reduces the overhead associated with copying large nodes, thereby achieving an update throughput roughly 2.6× that of ART under serial path-copying update. Under concurrent execution with the best configuration, AERT achieves a throughput approximately 4.8× that of ART, and about 15× that of the serial path-copying updates on ART.

Fig. 9(c) reports the throughput of BeTree. With 4 pipes and 16 workers, the concurrent BeTree attains approximately 7.5× the throughput of its serial counterpart. In contrast, concurrent B+Tree achieves at most 2× the throughput of the serial one under the 2-pipe, 8-worker configuration, as illustrated in Fig. 9(d). Given that BeTree and B+Tree exhibit nearly identical throughput in serial execution, this result highlights the significant efficiency gain that BeTree achieves in concurrent environments. Notably, under test configuration for B+Tree, the upper part consists only of the root node processed by the entry pipe, as the exit pipe does not perform updates. This observation suggests that resolving downward dependencies within the pipeline incurs significant overhead. BeTree addresses this performance bottleneck by introducing an elastic layer, enabling more efficient concurrent path-copying updates.

We also evaluate the update throughput of AERT and BeTree under a Zipfian workload. The results show similar trends and are deferred to the technical report [2] due to limited space.

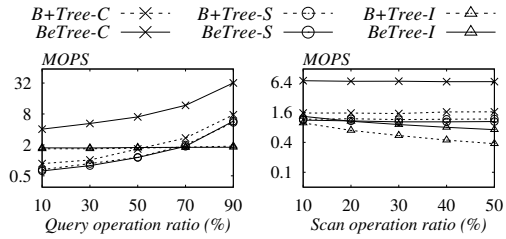
Performance under Mixed Workload. Next, we evaluate our solution under workloads with varying proportions of mixed operations. The '-C' suffix denotes the concurrent implementation using ConTree, while the corresponding serial counterparts are indicated by the '-S' suffix. According to the scalability experiment results above, we employ 4 pipes and 16 workers for concurrent path copying in BeTree and AERT, and 2 pipes and 8 workers in B+Tree and ART. For each data structure, we also include a serial in-place update variant denoted by the suffix '-I' to show the overhead of maintaining immutable snapshots via path copying.

Fig. 10 shows the throughput of AERT and ART under mixed workloads, while Fig. 11 shows the corresponding results for BeTree and B+Tree. The first setting varies the ratio of lookups (point



(a) Varying query ratio (b) Varying scan ratio

Fig. 10. Mixed Throughput: AERT & ART



(a) Varying query ratio (b) Varying scan ratio

Fig. 11. Mixed Throughput: BeTree & B+Tree

queries) mixed with updates. The second fixes the update ratio at 50% and varies the scan ratio in queries. A 50% scan ratio means that scans and lookups each constitute 25% of the workload.

We begin with the mixed update–lookup workload. Fig. 10(a) shows the results for AERT and ART. Clearly, AERT-C outperforms the other methods by a substantial margin, achieving roughly 7.2× the performance of ART-C when the query ratio is 50%. As described in Sec. 5.1, AERT significantly reduces the overhead of copying large nodes by employing half nodes, giving AERT-S higher update efficiency than ART-S. As a trade-off, in bytes with dense partial key distribution, splitting a single Node48/Node256 into a two-level half-node structure increases tree height, resulting in roughly a 30% reduction in lookup performance. Since in-place updates do not involve copying, half nodes do not offer benefits in this case, making AERT-I slower than ART-I. As shown later in Fig. 12, update operations are considerably more expensive than queries. Hence, supporting concurrent updates delivers substantial performance gains, as shown in Fig. 10(a), underscoring the effectiveness of our concurrent path-copying framework.

Fig. 11(a) shows the results for BeTree and B+Tree. Concurrent updates again provide clear throughput gains. BeTree-C outperforms other methods by a considerable margin, achieving roughly 4.6× the throughput of B+Tree-C when the query ratio is 50%. Since splits rarely occur at upper levels, BeTree-S and B+Tree-S maintain similar structures, resulting in nearly identical performance. This is also observed for BeTree-I and B+Tree-I with in-place updates.

Next, we report the performance under the scan-intensive workloads. As shown in Fig.10(b) and Fig. 11(b), both AERT-C and BeTree-C outperform their respective competitors. For in-place update methods, the throughput declines for all data structures as scans block updates for extended periods. For AERT-C, throughput declines further as the scan ratio increases, because the relatively poor scan performance of ART makes the scan operations the bottleneck. In contrast, BeTree-C, leverages the high scan performance of B+Tree, maintaining consistently high throughput.

Clearly, serial update by path-copying is slower than in-place update due to the copy overhead. As a result, in Fig. 10(a) and Fig. 11(a), serial path-copy implementations perform worse than their in-place counterparts under mixed workloads when the read ratio is low. Besides, when the read ratio increases, the benefit of read-write separation — avoiding interference between queries and updates — outweighs the copy overhead. Therefore, as the query ratio increases, serial path-copy implementations outperform their in-place counterparts. Furthermore, for long-running queries like scans, as shown in Fig. 10(b) and Fig. 11(b), the advantage of read-write separation becomes even more pronounced.

Comparison with Existing Solutions. Fig. 12 shows the update, lookup, and scan performance of AERT, BeTree, and Contreap [42]. Here, lookup and scan represent historical queries that randomly access one version. AERT and BeTree have similar update performance, while Contreap achieves around half their update throughput at the same scale, mainly because a BST has significantly greater depth than the structures in AERT or BeTree at the same data scale.

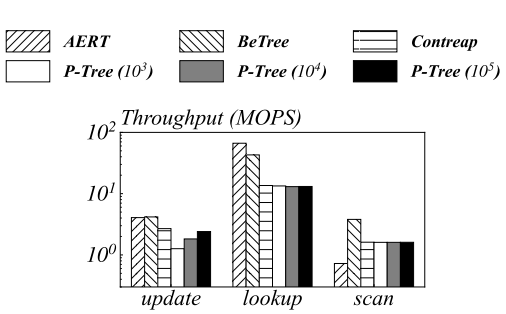


Fig. 12. Throughput for Update, Lookup and Scan.

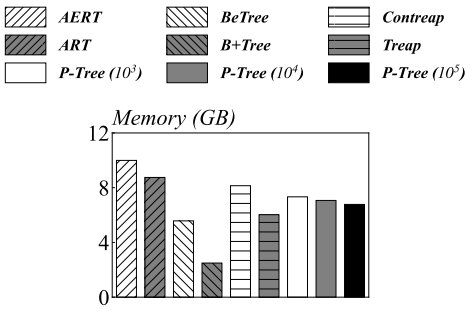


Fig. 13. Memory Usage.

For lookup operations, AERT achieves around 1.5× the throughput of BeTree. This result shows that AERT inherits ART's advantages by utilising adaptive layouts, exploiting modern CPU features such as cache locality, branch prediction, and SIMD instructions. Due to the greater node depth, the lookup throughput of Contreap is less than 20% and 30% of that of AERT and BeTree, respectively.

BeTree demonstrates significantly superior scan performance compared to other tree structures, with throughput roughly 3× that of Contreap and 6× that of AERT. In contrast to lookups, node depth is not a bottleneck for scanning. Instead, node with a sparse format in adaptive layouts hinders scan efficiency. Consequently, the scan throughput of AERT is about half that of Contreap.

Fig. 12 further includes P-Tree [69]. Note that P-Tree includes a tunable batch size to trade off throughput vs. latency: larger batches amortise overhead but increase waiting time before updates become visible, which is often impractical in real deployments with latency budgets. We therefore report the results when the batch sizes are 10^3, 10^4, and 10^5. P-Tree achieves similar throughput as Contreap at a batch size of 10^5. But 10^5 is already quite large in practice, making it impractical in real deployments. To evaluate P-Tree's native performance, we use only the coarse-grained snapshots it provides, locating the last one before the query (lookup/scan) version, rather than maintaining additional versioning information. In this scenario, the lookup and scan performance of P-Tree is similar to that of Contreap, as both utilise balanced BSTs for data storage.

In summary, although BSTs are well-suited for range aggregation [42], it is not ideal for general-purpose queries, such as lookups or range scanning. Moreover, we can observe that different tree indexes offer unique strengths tailored to specific data types and workloads, and thus each excels in distinct application scenarios. ConTree bridges this diversity by providing a unified and flexible scheme for concurrent path copying. This library empowers developers to efficiently design and implement tree indexes—not limited to BST, AERT, or BeTree—ensuring that index structures can match the requirements of variant applications.

Memory Costs. We examine the memory overhead introduced by path-copying updates. We compare each structure with its in-place counterpart: AERT vs. ART, BeTree vs. B+Tree, and Contreap vs. Treap. As shown in Fig. 13, after updating 1% of the data, AERT uses about 1.15×, BeTree about 2.2×, and Contreap about 1.35× the memory of their in-place versions. This ratio captures the memory overhead of a practical strategy, which keeps fine-grained snapshots for recent (e.g., the latest 1%) updates in memory to boost query performance. Older updates are maintained with coarse-grained versions, while the rest are compressed and offloaded to disk. In this case, memory use remains manageable while still supporting the efficient versioning and historical queries essential for many analytical, business, and scientific use cases.

P-Tree use less memory than Contreap since is only maintains coarse-grained versions, and the index size converges toward that of an in-place BST as the batch size grows (thus coarser granularity): about 1.22× at 10^3, 1.17× at 10^4 and 1.12× at 10^5.

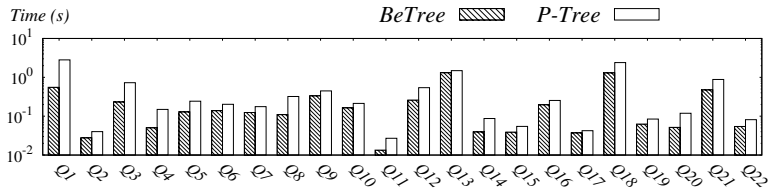


Fig. 14. TPC-H (SF=10)

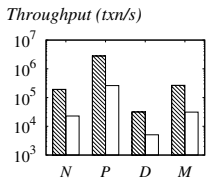


Fig. 15. TPC-C (SF=10)

End-to-End Performance. Finally, we use P-Tree's benchmark framework [69] to evaluate the end-to-end performance under the TPC-CH workload, which mixes the 22 analytical queries from TPC-H with the 3 update types from TPC-C to test concurrent performance. Since the analytical tasks in TPC-H are processed via scans, the experimental results in Fig. 12 indicate that BeTree is better suited for this application scenario. Therefore, our experimental setup replaces the underlying P-Tree with BeTree, and directly compare our solution against P-Tree in its default configuration.

Fig. 14 shows the execution time to run the 22 TPC-H queries on P-Tree and BeTree. Existing BST-based indexes [42, 70] support efficient queries for simple, predefined range aggregates by maintaining aggregated information in tree nodes. However, TPC-H includes complex and diverse analytical tasks that often requires scanning. With its superior scan efficiency, BeTree (B+Tree-based) processes queries faster than P-Tree (BST-based), outperforming P-Tree on all 22 queries.

Fig. 15 shows the transaction throughput of P-Tree and BeTree. The labels **N**, **P**, and **D** represent the *New-Order*, *Payment*, and *Delivery* transactions, respectively, and **M** denotes the TPC-C workload mixed using its predefined ratio. Due to inter-transaction dependencies, P-Tree only enables intra-transaction parallelism. In contrast, our solution supports concurrent updates since it is order-preserving, which significantly increases concurrency, yielding an 8.5× speedup for mixed updates. For individual transaction types, speedups are 8.6×, 10.7×, and 6.2× for N, P, and D, respectively.

Since both our solution and P-Tree use path-copying for read-write separation, their end-to-end performance is directly comparable. Beyond the experiments reported above, we also evaluated additional mixed workloads and observed the same performance trends. We omit these additional results since both BeTree and P-Tree implement read-write separation via path copying, and a direct comparison reveals BeTree's superior read and write performance. In summary, BeTree outperforms P-Tree under the TPC-CH workload, showing the advantage of our approach on hybrid workloads that combine transactions with analytical tasks.

7 Conclusion

In this paper, we present ConTree, a lightweight library, to facilitate the implementation of concurrent path-copying updates for tree structures. We further explore tree designs to boost update performance, introducing AERT and BeTree—variants of ART and B+Tree, respectively—optimised for concurrent updates using ConTree. Experimental results demonstrate that our solutions achieve significantly higher update efficiency compared to existing approaches.

Acknowledgments

This work is supported by the RGC GRF grant (No. 14217322) and the "1+1+1" CUHK-CUHK(SZ)-GDST Joint Collaboration Fund (No. 2025A0505000045).

References

[1] 2026. BoltDB. https://github.com/etcd-io/bbolt.
[2] 2026. code&tr. https://github.com/CUHK-DBGroup/Contrees.
[3] 2026. DuckDB. https://duckdb.org.
[4] 2026. ETCD. https://etcd.io/.

[5] 2026. H-Store. https://hstore.cs.brown.edu.
[6] 2026. HBase. https://hbase.apache.org.
[7] 2026. Kafka. https://kafka.apache.org.
[8] 2026. kubernetes. https://kubernetes.io/.
[9] 2026. LMDB. http://www.lmdb.tech/doc.
[10] 2026. RaimaDB. https://raima.com.
[11] 2026. SAP HAHA. https://www.sap.com/products/data-cloud/hana.html.
[12] 2026. SurrealDB. https://surrealdb.com.
[13] James F. Allen. 1983. Maintaining knowledge about temporal intervals. *Commun. ACM* 26, 11 (1983), 832–843.
[14] Nikolas Askitis and Ranjan Sinha. 2007. HAT-trie: a cache-conscious trie-based data structure for strings. In *ACSC*. 97–105.
[15] Muhammad A. Awad, Serban D. Porumbescu, and John D. Owens. 2023. A GPU Multiversion B-Tree. In *PACT*. 481–493.
[16] Andreas Behrend, Anton Dignös, Johann Gamper, Philip Schmiegelt, Hannes Voigt, Matthias Rottmann, and Karsten Kahl. 2019. Period Index: A Learned 2D Hash Index for Range and Duration Queries. In *SSTD*. 100–109.
[17] Hans Berliner. 1981. The B*Tree Search Algorithm: A Best-First Proof Procedure. In *Readings in Artificial Intelligence*. 79–87.
[18] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *SIGMOD*. 521–534.
[19] Guy E Blelloch, Daniel Ferizovic, and Yihan Sun. 2016. Just join for parallel ordered sets. In *SPAA*. 253–264.
[20] Michael H. Böhlen, Anton Dignös, Johann Gamper, and Christian S. Jensen. 2018. Temporal Data Management – An Overview. In *Business Intelligence and Big Data*. 51–83.
[21] William J. Bolosky and Michael L. Scott. 1993. False sharing and its effect on shared memory performance. In *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4 (Sedms'93)*. 3.
[22] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A practical concurrent binary search tree *(PPoPP)*. 257–268.
[23] Trevor Brown, Faith Ellen, and Eric Ruppert. 2014. A general technique for non-blocking trees *(PPoPP)*. 329–342.
[24] Hokeun Cha, Moohyeon Nam, Kibeom Jin, Jiwon Seo, and Beomseok Nam. 2020. B3-Tree: Byte-Addressable Binary B-Tree for Persistent Memory. *ACM Trans. Storage* 16, 3 (2020).
[25] Hsinchun Chen, Roger HL Chiang, and Veda C Storey. 2012. Business intelligence and analytics: From big data to big impact. *MIS quarterly* (2012), 1165–1188.
[26] George Christodoulou, Panagiotis Bouros, and Nikos Mamoulis. 2022. HINT: A Hierarchical Index for Intervals in Main Memory. In *SIGMOD*. 1257–1270.
[27] James Clifford and Abdullah Uz Tansel. 1985. On an algebra for historical relational databases: two views. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 247–265.
[28] Douglas Comer. 1979. Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (1979), 121–137.
[29] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC*. 143–154.
[30] Ankur Dave, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. 2016. *Persistent Adaptive Radix Trees: Efficient Fine-Grained Updates to Immutable Data.* Technical Report. UC Berkeley.
[31] James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. 1986. Making Data Structures Persistent. In *STOC*. 109–121.
[32] Curtis Dyreson, Fabio Grandi, Wolfgang Käfer, Nick Kline, Nikos Lorentzos, Yannis Mitsopoulos, Angelo Montanari, Daniel Nonen, Elisa Peressi, Barbara Pernici, John F. Roddick, Nandlal L. Sarda, Maria Rita Scalas, Arie Segev, Richard Thomas Snodgrass, Mike D. Soo, Abdullah Tansel, Paolo Tiberio, and Gio Wiederhold. 1994. A consensus glossary of temporal database concepts. *SIGMOD Rec.* 23, 1 (1994), 52–64.
[33] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-blocking binary search trees. In *PODC*. 131–140.
[34] Panagiota Fatourou, Elias Papavasileiou, and Eric Ruppert. 2019. Persistent Non-Blocking Binary Search Trees Supporting Wait-Free Range Queries. In *SPAA*. 275–286.
[35] Goetz Graefe. 2011. Modern B-Tree Techniques. *Found. Trends Databases* 3, 4 (2011), 203–402.
[36] Goetz Graefe. 2024. More Modern B-Tree Techniques. *Found. Trends Databases* 13, 3 (2024), 169–249.
[37] Goetz Graefe and Per-Åke Larson. 2001. B-Tree Indexes and CPU Caches. In *ICDE*. 349–358.
[38] Goetz Graefe, Haris Volos, Hideaki Kimura, Harumi Kuno, Joseph Tucek, Mark Lillibridge, and Alistair Veitch. 2014. In-memory performance for big data. *Proc. VLDB Endow.* 8, 1 (2014), 37–48.
[39] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *CIDR*.

[40] Richard A. Hankins and Jignesh M. Patel. 2003. Effect of node size on the performance of cache-conscious B+Trees. In *SIGMETRICS*. 283–294.
[41] Ferenc Havasi. 2011. An Improved B+ Tree for Flash File Systems. In *Theory and Practice of Computer Science (Lecture Notes in Computer Science, Vol. 6543)*. 297–307.
[42] Guanhao Hou, Jinchao Huang, Fangyuan Zhang, and Sibo Wang. 2025. Efficient Concurrent Updates to Persistent Randomized Binary Search Trees. *Proc. VLDB Endow.* 18, 5 (2025), 1481–1494.
[43] Shirish Jeble, Sneha Kumari, and Yogesh Patil. 2017. Role of big data in decision making. *Operations and Supply Chain Management: An International Journal* 11, 1 (2017), 36–44.
[44] Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter Michael Fischer, Donald Kossmann, Franz Färber, and Norman May. 2013. Timeline index: a unified data structure for processing queries on temporal data in SAP HANA. In *SIGMOD*. 1173–1184.
[45] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*. 195–206.
[46] Thomas Kissinger, Benjamin Schlegel, Dirk Habich, and Wolfgang Lehner. 2012. KISS-Tree: smart latch-free in-memory indexing on modern architectures. In *DaMoN*. 16–23.
[47] Martin Koppehel, Tobias Groth, Sven Groppe, and Thilo Pionteck. 2021. CuART - a CUDA-based, scalable Radix-Tree lookup and update engine. In *ICPP*.
[48] Yongsik Kwon, Seonho Lee, Yehyun Nam, Joong Na, Kunsoo Park, Sang Cha, and Bongki Moon. 2023. DB+Tree: A new variant of B+Tree for main-memory database systems. *Information Systems* 119 (2023).
[49] Philip L. Lehman and S. Bing Yao. 1981. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Syst.* 6, 4 (1981), 650–670.
[50] Daan Leijen, Benjamin Zorn, and Leonardo Moura. 2019. *Mimalloc: Free List Sharding in Action.* 244–265.
[51] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*. 38–49.
[52] Justin Levandoski, David Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for New Hardware Platforms. In *ICDE*. 302–313.
[53] David B. Lomet, Roger S. Barga, Mohamed F. Mokbel, German Shegalov, Rui Wang, and Yunyue Zhu. 2005. Immortal DB: transaction time support for SQL server. In *SIGMOD*. 939–941.
[54] Xuchuan Luo, Pengfei Zuo, Jiacheng Shen, Jiazhen Gu, Xin Wang, Michael R. Lyu, and Yangfan Zhou. 2023. SMART: A High-Performance Adaptive Radix Tree for Disaggregated Memory. In *OSDI*.
[55] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. 2021. ROART: Range-query Optimized Persistent ART. In *FAST*.
[56] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *EuroSys*. 183–196.
[57] Marcus Müller, Lawrence Benson, and Viktor Leis. 2025. B-Trees Are Back: Engineering Fast and Pageable Node Layouts. *Proc. ACM Manag. Data* 3, 1 (2025).
[58] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *SIGMOD*. 371–386.
[59] Christian Plattner, Andreas Wapf, and Gustavo Alonso. 2006. Searching in time. In *SIGMOD*. 754–756.
[60] Jun Rao and Kenneth A. Ross. 1999. Cache Conscious Indexing for Decision-Support in Main Memory. In *Proc. VLDB Endow.* 78–89.
[61] Jun Rao and Kenneth A. Ross. 2000. Making B+Tree cache conscious in main memory. *SIGMOD Rec.* 29, 2 (2000), 475–486.
[62] Bruno Ribeiro, Minh X. Hoang, and Ambuj K. Singh. 2015. Beyond Models: Forecasting Complex Network Processes Directly from Data. In *WWW*. 885–895.
[63] Ohad Rodeh. 2008. B-trees, shadowing, and clones. *ACM Trans. Storage* 3, 4, Article 2 (2008).
[64] Hongchan Roh, Sanghyun Park, Sungho Kim, Mincheol Shin, and Sang-Won Lee. 2011. B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives. *Proc. VLDB Endow.* 5, 4 (2011), 286–297.
[65] Adam Seering, Philippe Cudré-Mauroux, Samuel Madden, and Michael Stonebraker. 2012. Efficient Versioning for Scientific Array Databases. In *ICDE*. 1013–1024.
[66] Ross Shaull, Liuba Shrira, and Hao Xu. 2008. Skippy: a new snapshot indexing method for time travel in the storage manager. In *SIGMOD*. 637–648.
[67] Benwei Shi, Zhuoyue Zhao, Yanqing Peng, Feifei Li, and Jeff M. Phillips. 2021. At-the-time and Back-in-time Persistent Sketches. In *SIGMOD*. 1623–1636.
[68] Richard Snodgrass. 1987. The temporal query language TQuel. *ACM Trans. Database Syst.* 12, 2 (1987), 247–298.
[69] Yihan Sun, Guy E. Blelloch, Wan Shen Lim, and Andrew Pavlo. 2019. On supporting efficient snapshot isolation for hybrid workloads with multi-versioned indexes. *Proc. VLDB Endow.* 13, 2 (2019), 211–225.

[70] Yihan Sun, Daniel Ferizovic, and Guy E. Belloch. 2018. PAM: parallel augmented maps. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 290–304.
[71] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *SOSP*. 18–32.
[72] Hao Wang, Yilun Cai, Yin David Yang, Shiming Zhang, and Nikos Mamoulis. 2014. Durable Queries over Historical Time Series. *IEEE Trans. Knowl. Data Eng.* 26, 3 (2014), 595–607.
[73] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *SIGMOD*. 473–488.
[74] Zhewei Wei, Ge Luo, Ke Yi, Xiaoyong Du, and Ji-Rong Wen. 2015. Persistent Data Sketching. In *SIGMOD*. 795–810.
[75] 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 (2023), 2976–2989.
[76] Bowen Zhang, Shengan Zheng, Liangxu Nie, Zhenlin Qi, Hongyi Chen, Linpeng Huang, and Hong Mei. 2024. Revisiting PM-based B^+-Tree with Persistent CPU Cache. *IEEE Trans. Parallel Distributed Syst.* PP (2024).
[77] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *SIGMOD*. 323–336.
[78] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2020. Succinct Range Filters. *ACM Trans. Database Syst.* 45, 2 (2020).
[79] Wei Zhang, Houjun Tang, Suren Byna, and Yong Chen. 2018. DART: distributed adaptive radix tree for efficient affix-based keyword search on HPC systems. In *PACT*.
[80] Weihua Zhang, Chuanlei Zhao, Lu Peng, Yuzhe Lin, Fengzhe Zhang, and Jinhu Jiang. 2022. High performance GPU concurrent B+tree. In *PPoPP*. 443–444.

Received October 2025; revised January 2026; accepted February 2026