PaB-tree: Parallel B^+-tree with Batch Updates

Ziyang Men
ziyang.men@email.ucr.edu
University of California, Riverside

Yan Gu
ygu@cs.ucr.edu
University of California, Riverside

Yihan Sun
yihans@cs.ucr.edu
University of California, Riverside

Helen Xu
hxu615@gatech.edu
Georgia Institute of Technology

arXiv:2611.09172v1 [cs.DS] 6 Nov 2026

Abstract

The B^+-tree is a ubiquitous search index due to its provable I/O (cache) efficiency. Yet unlike batch-parallel binary search trees, which are both fast and widely available, no batch-parallel B^+-tree offers strong empirical performance together with provable, near-optimal I/O efficiency. Prior B^+-tree-based designs are either parallel but purely theoretical, with no high-performance implementation, or concurrent indexes such as the Bw-tree and Mass-tree, which run an order of magnitude slower on batch updates.

We present PaB-tree, the first batch-parallel B^+-tree with guarantees on work, span, and cache complexity alongside strong practical performance. Its core challenge—repairing node overflows and underflows without eroding the tree's I/O advantage—is resolved by one simple invariant: a repair never touches a sibling and is always pushed to the parent. A Promote primitive reduces batch construction and insertion to repeated node promotion, and batch deletion to reinsertion. Batch updates take $O(B \log_B n)$ work and $O(\log_B n)$ I/O per element, matching the sequential B^+-tree bounds, with polylogarithmic span. Against highly optimized parallel and concurrent baselines, PaB-tree delivers the best overall performance trade-off: up to 1.4× faster on construction and up to 2.0× faster on batch insertion than the strongest baseline, while preserving B^+-tree query efficiency.

1 Introduction

B/B$^+$-trees [7, 23] are among the most fundamental data structures and search indexes, widely adopted in both in-memory and external-memory systems. In this paper, we focus on the in-memory setting. Unlike binary search trees (BSTs), which use a single key as a separator, B/B$^+$-trees keep a $\Theta(B)$-way fan-out so that each node aligns with the hardware access unit (typically a few cache lines in the in-memory setting). Every node access therefore fetches $\Theta(B)$ keys for comparison, and a single access is logically equivalent to descending $\Theta(\log B)$ levels of a BST. This design positions the B/B$^+$-tree as one of the most canonical data structures for achieving ***provable I/O (cache) efficiency***. Backed by this theoretical guarantee, in-memory B/B$^+$-trees typically outperform binary trees by 2–8× in the sequential setting, and therefore have been natively offered across a wide spectrum of software libraries [10, 11, 36, 37, 66].

Batch-Parallel Paradigm. Driven by growing data scales and the multicore era, batch-parallel data structures have attracted significant research interest [1, 5, 14, 17, 26, 27, 29, 44–46, 52, 53, 57, 62, 64, 65, 68, 70, 71, 74, 75, 77–79, 85]. These structures apply updates (insertions or deletions) in batches in parallel, and have been adopted in graph processing [27, 29, 31, 57, 69, 75], databases [61, 64], parallel algorithms for dynamic programming [28, 39], computational geometry [13, 52], vector similarity search [53, 84, 85], etc.

Although both B/B$^+$-trees [7, 23] and BSTs [2, 41, 55] were introduced over 50 years ago, and while B/B$^+$-trees are the go-to index design due to their cache-friendliness, there is a striking disparity in the maturity of their batch-parallel counterparts. Parallel BST implementations have been studied extensively and are widely accessible in practice, with examples including the PAM [14, 65], Aspen [27], CPAM [26], and MCSTL [32]. Many of them are also theoretically efficient, achieving high parallelism while preserving optimal or near-optimal sequential work bounds. In contrast, there is a notable lack of accessible batch-parallel B/B$^+$-tree implementations with both strong empirical performance and provable, near-optimal I/O efficiency. Existing efforts are either purely theoretical [35, 43, 59], with no high-performance implementation, or concurrent, such as the well-known Bw-tree [72] and the Mass-tree [50]. However, concurrent structures do not exploit the inherent advantages of batching; consistent with prior findings [65], Sec. 4 shows that they run an order of magnitude slower than parallel trees in batch updates. Other studies have explored binary trees with expanded node sizes [26, 27], but they remain a suboptimal compromise for I/O efficiency (also see Fig. 1), due to looser tree-height bounds compared to a true B/B$^+$-tree, and none offer non-trivial cache complexity.

Challenges in Designing Efficient Parallel B/B$^+$-trees. The primary challenge in parallelizing B/B$^+$-trees lies in achieving low-cost structural repairs without eroding their I/O advantage. While the high fan-out in a B/B$^+$-tree decreases memory transfers (i.e., the cache complexity), it does not reduce the total work (in terms of the number of comparisons), and is inherently more complicated to deal with. Whereas BST rotations remain strictly local, the high fan-out nature of B/B$^+$-trees may cause structural imbalances (overflows or underflows) to propagate across neighboring parallel subtasks, demanding dedicated efforts to repair. Since the reduction in memory access, $\Theta(\log_2 B)$, is relatively modest in practice, any overhead added to reconcile structural modifications is subtracted from the sole term that constitutes the benefit. Consequently, a merely correct design is easily slower than a well-tuned parallel BST.

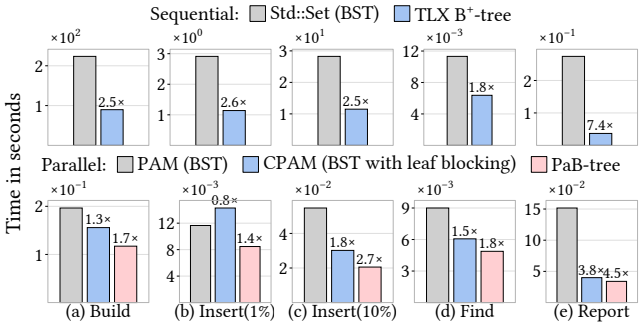


Figure 1. Running time (in seconds) of the sequential and parallel BSTs and B^+-trees. Lower is better. We measure the time under a uniform distribution, including the time to build a tree containing 10^8 entries and the time to insert 1% and 10% of entries into the tree. The query time is measured by searching 10^6 entries, and reporting $[10^3, 10^5]$ entries in a range.

This raises a critical question: *Can the high fan-out design of B/B$^+$-trees still offer provable cache efficiency in the parallel setting, and can this advantage translate into better performance in practice?* In this paper, we answer the question affirmatively by presenting the PaB-tree, the first batch-parallel B$^+$-tree implementation with both theoretical guarantees (work, span, cache complexity) and strong practical performance across a wide range of workloads.

Approach Overview. In light of the challenges discussed above, our design prioritizes conceptual simplicity to minimize parallel overhead during structural repairs. The design follows a single rule: *a repair never touches a sibling; it is always pushed to the parent.*

The difficulty is that a repair must never reach sideways to a sibling, which another parallel task may be modifying at the same time; instead, every repair travels up the ancestor path the recursion already owns. We realize this by having each recursive call return a *heavy node*—a possibly oversized node that the parent absorbs and re-partitions—rather than a fixed-shape subtree, and by keeping a node-capacity margin that lets the more involved deletion cost amortize fully.

A single overflow-like operation (our Promote primitive detailed in Sec. 3) resolves an oversized heavy node, and both construction and insertion reduce to repeated applications of it. Underflow in batch deletion, however, is more challenging to handle. Our solution reduces underflow repair to insertion—an operation we have already made parallel and cache-efficient—so no node ever consults a sibling. Specifically, given a set of underflowed subtrees invalidated by a batch deletion, we carefully identify a few entry points for them to be reinserted, letting them share the work, such that the total overhead can be bounded in an amortized fashion (Thm. 3.3). Fig. 5 shows that the reinserted volume also stays modest in practice, so the reduction is inexpensive as well as simple.

Results Summary. For both batch insertion and deletion, PaB-tree achieves provable bounds on work, span, and cache complexity. A batch insertion or deletion incurs $O(B \log_B n)$ work and $O(\log_B n)$ I/O cost per element, matching the bounds of a sequential B^+-tree, and both operations achieve polylogarithmic span (Thm. 3.1 and Thm. 3.3). The theoretical efficiency also translates into strong performance in practice. Fig. 1 confirms this observation: PaB-tree matches the cache-friendliness of sequential B^+-tree without sacrificing parallelism, and excels when processing large batches.

Fig. 4 confirms this trend on a more comprehensive benchmark across different operations, data distributions, and more baselines. Overall, PaB-tree achieves the best performance trade-off among various existing data structures. It has an advantage in parallel construction and batch insertions, scaling especially well to processing large batches, as well as outperforming all other tree-based baselines on queries. As a result, PaB-tree demonstrates consistent gains on real-world geometric and graph workloads (Sec. 4.2 and Sec. 4.3). We release our anonymous code [51].

2 Preliminaries

This section reviews the analysis models, background on B^+-trees, and key subroutines that we will use to prove theoretical guarantees on the PaB-tree operations.

In this paper, we use n to denote the input size or the tree size. We write $\log n$ for $\log_2(n+1)$, and $\log_B n$ for $\log_B(n+1)$.

Computational Model. We assume a shared-memory multiprocessor that supports fork-join parallelism with binary forking [6, 15, 18]. A thread is a sequential RAM with a fork instruction that creates two parallel child threads and suspends the parent until both finish; a parallel for-loop compiles to a logarithmic number of forks. We analyze our algorithms using the work-span model, where the work is the total number of operations in the algorithm, and the span is the longest dependence chain in the parallel computation graph. On p processors, a randomized work-stealing scheduler executes a computation with work W and span S in $W/p + O(S)$ time with high probability in W [6, 18, 40].

We use ***I/O cost*** to refer to cache complexity, and analyze it using the ideal-cache model [3, 33]: a cache of size M above an unbounded memory, exchanging size-B blocks at unit cost under an optimal offline policy. The cache complexity is the total number of block transfers. We also use B for the block size of our PaB-tree, which can be any value in $\Theta(B)$.

***B$^+$*-trees.** The B$^+$-tree is a canonical data structure for storing ordered data. Every node keeps a $\Theta(B)$ fan-out so that it aligns with a hardware access unit (e.g., a cache line or a disk block). All entries are stored in the leaves, and all leaves are at the same depth. Each leaf contains a sequence of entries in sorted order. An interior node contains a sequence of child pointers and a sequence of splits that separate its children. Every interior node except the root has between B and $4B$ child nodes. A leaf can hold more entries since it stores no child pointers. However, for simplicity we assume that a leaf

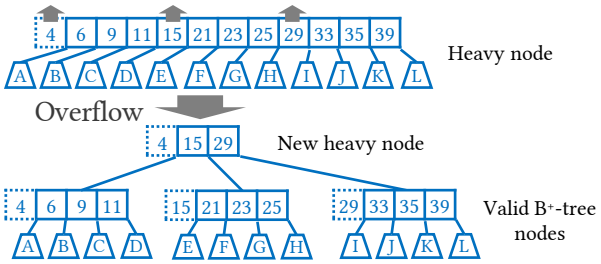


Figure 2. Illustration of the PROMOTE primitive. Assume $B = 2$. Then *PROMOTE* adjusts each node to a fan-out of $2B = 4$. Trapezoids with uppercase letters represent subtrees. The virtual leading keys are marked in dotted boxes. The resulting subtrees are all valid and at the same level.

also holds between B and $4B$ entries.

A B$^+$-tree with n entries has height $O(\log_B n)$. It can be constructed sequentially with $O(n \log_B n)$ work and $O(n/B \cdot \log_B n)$ I/O cost. During insertion, a node overflows if its size exceeds $4B$. This overflow is resolved by splitting the node into two and promoting the midpoint key to its parent. Conversely, deleting an entry may decrease a node's size below B. This triggers an underflow that is handled by merging the node with an adjacent sibling and removing a key from the parent. Both overflows and underflows can cascade up the tree, requiring recursive rebalancing.

3 Parallel Algorithms

3.1 PaB-tree Structure and Primitives

In this section, we present the design of PaB-tree. A PaB-tree is conceptually a B$^+$-tree, where all data entries reside in the leaf nodes and internal nodes store *separators* as routing keys. Unlike a standard B$^+$-tree, where a node with b subtrees contains $b-1$ separators, an internal node in the PaB-tree always maintains b separators and the same number of subtrees, where each subtree follows a key. A key and the subtree following it are ***paired***. Such a node can be viewed as a regular B$^+$-tree node augmented with a *leading key*, which is the smallest key in its subtree. We always augment the entire dataset with a leading key $-\infty$ to ensure correctness. Fig. 2 and Fig. 3 both illustrate this scheme. The leading key simplifies the algorithm design, since handling internal and leaf nodes can then follow the same logic. It also simplifies our proofs. In practice, there is no need to store the leading key explicitly, since it can always be retrieved from the recursive calls.

A valid node has size in $[B, 4B)$. Any cB with $c > 2$ works as the upper bound and guarantees theoretical and practical efficiency. We fix $c = 4$ for simplicity, and later show in our proofs how it bounds the costs.

Heavy Node. The concept of a ***heavy node*** simplifies the description of the PaB-tree algorithms. A heavy node encapsulates the transient state of an invalidated B$^+$-tree node. We allow a heavy node to have any size of at least B, with no upper bound, but all its subtrees must be valid B$^+$-trees of equal height. This property is crucial, as structural invalidities are isolated and resolved locally within the node, streamlining the overall correctness justification (see Fig. 2 for an illustration). Specifically, a heavy node H maintains its *size*, also denoted by $|H|$, and two arrays *subtrees*[] and *separators*[], each of size $|H|$. Here, $H.separators[i]$ is the smallest key in $H.subtrees[i]$. A valid tree node is therefore a heavy node with size in $[B, 4B)$. We use the heavy node as the core operational unit in our algorithms.

Algorithm 1: PaB-tree Primitive: PROMOTE

A heavy node maintains
- ***size***: size of the heavy node, also denoted by $|\cdot|$
- ***separators***: an array of b separator keys
- ***subtrees***: an array of b PaB-tree subtrees

// Resolves overflow in heavy node H and return a parent heavy node.

1 **Function** $H' \leftarrow$ PROMOTE(H)
2 **if** H is a valid node **then** return H
3 $b' \leftarrow \max(\lfloor |H|/2B \rfloor, 1)$ *// to be split into b' valid PaB-tree nodes*
4 $H'.size \leftarrow b'$
5 **parallel-for** $i \leftarrow 0$ to $b'-1$ **do** *// partition H into b' groups*
// Group i contains the s-th to the e-th separator and subtree in H, each with similar sizes (differing by at most 1)
6 $s \leftarrow \lfloor i \cdot b/b' \rfloor$, $e \leftarrow \lfloor (i+1) \cdot b/b' \rfloor$
7 Create a PaB-tree node v using subtrees $H.subtrees[s..e)$ and separators $H.separators[s..e)$
// Set the i-th node and separator in return heavy node H'
8 $H'.subtrees[i] \leftarrow v$
9 $H'.separators[i] \leftarrow$ the first key in v
10 **return** H' *// Return the parent heavy node*

The PROMOTE Primitive. Our algorithm follows the standard overflow and underflow rebalancing scheme of B$^+$-trees. We first introduce the PROMOTE primitive (Alg. 1), which simulates the handling of an overflow. It restructures a heavy node into a sequence of valid nodes at the same level, each of size about $2B$, and generates a higher-level heavy node. In detail, the PROMOTE first check whether H is a valid node, and if so, returns it directly. This is consistency with the definition of a heavy node since a valid node is also a heavy node. Otherwise, it partitions the b subtrees and separators of a heavy node H into $b' = \lfloor b/2B \rfloor$ groups, each of size $\lceil b/b' \rceil$ or $\lfloor b/b' \rfloor$. It then packs each group into a standard node and extracts its leading key to form b' separators. Crucially, even when H does not overflow, at least one separator is still promoted: the PROMOTE packs H into a single node and promotes the key paired with it, so the operation applies uniformly. The resulting b' subtrees and separators form a smaller heavy node, which is then passed to the parent (Fig. 2).

The PROMOTE primitive extends the sequential overflow process naturally, and also admits parallelism: the grouping depends only on local indices, so all subtrees and separators are processed in parallel, resolving an overflow of any size in a single step. With this simple primitive, our insertion algorithm largely reduces to applying this primitive repeatedly, and our deletion algorithm also implicitly leverages it.

Algorithm 2: Batch Insertion

Input: A sequence of entries E and a PaB-tree T.
Output: A PaB-tree with E inserted.

```
// Insert the entries E into T and maintain the B+-tree properties
1  Function BatchInsert(T, E)
2      Sort the entries in E by their keys
3      H ← InsertSorted(T, E)        // Insert new entries into leaves
4      while H.size ≥ 4B do
5          H ← Promote(H)
6      return H
// Insert sorted entries E into T; return updated node, possibly heavy
7  Function InsertSorted(T, E)
8      if E is empty then return T                    // Unchanged
9      if T is a leaf then
10         Insert E into T in place
           // Use Promote to split the tree when necessary, which also
              identifies the upper-level separators to return to the parent
11         return Promote(T)
       // Otherwise, T is an interior node; continue recursive insertion
12     b ← T.size
13     P[0..b) ← Slices of E partitioned by T.separators
14     Initialize H as a sequence of b heavy nodes, which will hold
         the results returned by the b recursive calls
15     parallel-for i ← 0 to b − 1 do                 // Recursively insert
16         H[i] ← InsertSorted(T.subtrees[i], P[i])
17     Initialize H' as a heavy node, which will represent the state of
         T after absorbing all insertions
18     H'.subtrees ← concatenate all H[·].subtrees[] in order
19     H'.separators ← concatenate all H[·].separators[] in order
       // Resolving possible overflow and pass new separators to parent
20     return Promote(H')
```

Below, we detail the complete algorithms based on the heavy node and the Promote primitive.

3.2 Batch Insertion and Construction

We now present our parallel batch update algorithms for PaB-trees. We consider the batch-parallel setting, where a batch of entries is inserted into or deleted from the current PaB-tree T. We start with the insertion algorithm.

With the Promote primitive, the insertion process is relatively straightforward. We present the algorithm in Alg. 2 and illustrate it in Fig. 3. The algorithm inserts a batch of entries E into a PaB-tree T. We first sort all keys in E, then follow a divide-and-conquer scheme and pass all entries to the root by calling the helper function InsertSorted. This function takes a subtree T and inserts the sorted entries E into it. It returns a *heavy node* H holding the subtrees replacing T, each paired with its leading key. By definition, the returned node is a sequence of b' separators, each followed by a subtree. Here b' can be any integer no smaller than B.

InsertSorted has two base cases: 1) an empty batch leaves the subtree unchanged, and 2) a leaf merges the entries E in place and calls Promote. Even though a leaf stores no subtrees, the Promote logic applies identically, with the entries playing the role of subtrees; i.e., it splits the leaf into new leaves of about $2B$ entries each and promotes the leading keys. At an internal node, we partition E by the separators in T and recurse in parallel on every pair of subtree and slice. Each recursive call returns a heavy node. Because every heavy node retains its leading key, these results concatenate directly into one transient heavy node H'. A final call to Promote(H') further partitions H' into valid subtrees and returns them to the parent as a heavy node (line 20).

Parallel Construction. We construct a PaB-tree in parallel by inserting a batch of entries E into an empty tree. This is equivalent to viewing the entire batch as one large heavy node and applying Promote to it repeatedly, until a valid root is generated.

Theoretical Analysis. We can now show the bounds for the PaB-tree insertion algorithm.

Theorem 3.1. *Given a PaB-tree of size n, a batch insertion of m entries takes $O(Bm\log_B(m+n))$ work, $O(m\log_B(m+n))$ I/O cost, and $O(\log n \log(m+B))$ span.*

The I/O cost per entry is $O(\log_B(m+n))$, which aligns with the sequential I/O cost of B/B$^+$-trees. The work bound is larger by a multiplicative factor of $O(B)$. Given the high compute throughput and limited memory bandwidth of modern processors, our PaB-tree is rarely bottlenecked by computation, unless the tree fits in cache. The span is polylogarithmic.

The analysis of I/O cost is based on the following fact.

Fact 3.2. *Given a PaB-tree with n leaves and height no more than $\lceil\log_B n\rceil$, the total number of nodes among any m leaves and their ancestors is $O\big(m\log_B \frac{n}{m}\big)$.*

Because the fan-out of a PaB-tree is at least B, this height is an overestimation. We pessimistically assume that all nodes in the top $\lceil\log_B m\rceil$ levels are ancestors of the m leaves, which gives $O(m)$ nodes in total. Each of the remaining $\lceil\log_B n\rceil - \lceil\log_B m\rceil = O(\log_B(n/m))$ levels contains at most one ancestor per leaf, hence at most m nodes, which leads to the stated bound.

We can now prove the batch insertion bounds in Thm. 3.1.

Proof of Thm. 3.1. We can directly obtain the I/O bound from Fact 3.2. Since each node in a PaB-tree aligns with $O(1)$ memory blocks (cache lines), inserting m entries into a tree of size n can touch at most the number of nodes bounded by Fact 3.2. This corresponds to $O(m\log_B((m+n)/m))$ cache line reads and writes. Note that this bounds the nodes touched in the output tree. The input tree (before modification) must also be accounted for, but in an insertion operation, the number of tree nodes in the input tree is less than or equal to that of the output tree. Adding both together yields the stated $O(m\log_B((m+n)/m))$ I/O bound. We also need to sort the input entries (line 2) with $O(m\log_M m)$ I/Os [16], and adding all pieces gives the overall $O(m\log_B(m+n))$ I/O bound.

The work bound is obtained by multiplying the I/O cost by B. As each node is accessed and processed, operations such as scanning, merging, and concatenating all take work linear in the input size, which is $O(B)$. Therefore, the total work excluding the sorting part is $O(Bm\log_B(m+n))$. Sorting

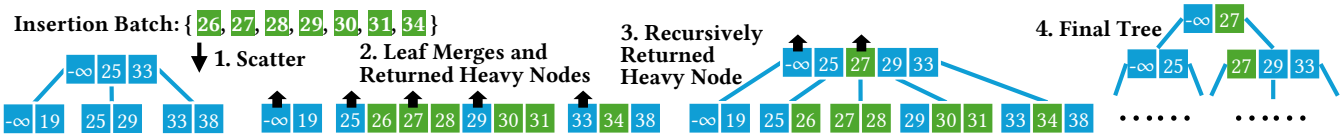


Figure 3. An illustration of the batch insertion algorithm with $B = 1$. The algorithm runs in an asynchronous divide-and-conquer approach, and the output of each subtree is always a heavy node. The two heavy nodes each with a size over 4 apply the PROMOTE function.

Algorithm 3: Batch Deletion

Input: A sequence of unsorted entries E and a PaB-tree T.
Output: A PaB-tree with E deleted.

```
1  Function BatchDelete(T, E)
2      Sort the entries in E by their keys
3      DeleteSorted(T, E)                          // Remove entries from T
4      Group the dropped subtrees in line 14 by height, and reinsert
         each group into T using Alg. 2

5  Function DeleteSorted(T, E)
6      if E is empty then return T
7      if T is a leaf then Remove E from T, and return T
       // T is an interior node; continue the recursive deletion.
8      P ← Slices of E partitioned by T.separators
9      b ← T.size
10     Initialize N as a sequence of b PaB-tree nodes, which will
         hold the results returned by the b recursive calls
11     parallel-for i ← 0 to b − 1 do              // Recursively delete
12         N[i] ← DeleteSorted(T.subtrees[i], P[i])
13     if the number of non-empty N[i] is less than B then
14         Drop non-empty N[i] to be reinserted later, and return ∅
15     Create a PaB-tree node N by concatenating N, and return N
```

takes $O(m \log m) = O(m \log(m+n)) = O(Bm \log_B(m+n))$; thus, the total work remains $O(Bm \log_B(m+n))$.

Finally, we analyze the span bound. The insertion algorithm recurses for $\log_B n$ levels. At each level, the parallel-for loop of size B produces a span of $\log B$. Multiplying these two factors yields a span of $\log n$. At each level, the work for operations including merge and concatenate can be trivially parallelized with a span logarithmic in the input size. Since the worst-case input size is $m + B$, the span per level is bounded by $O(\log(m+B))$. Together, the total span is bounded by $O(\log n \log(m+B))$, leading to the stated bound. □

3.3 Batch Deletion

Classic underflow logic is inherently sequential. An underflowing node is repaired by borrowing from or merging with a sibling, so the outcome depends on that sibling rather than on the node alone. A batch of deletions may trigger many underflows at once, and their repairs would conflict: the sibling borrowed from may itself underflow, or may be deleted entirely. Resolving these interactions requires serializing the dependencies among the repairs, which blocks parallelism.

Our deletion algorithm (Alg. 3) avoids these interactions entirely, and the design is straightforward. It follows the same divide-and-conquer approach as the insertion algorithm to maximize efficiency. The algorithm first checks whether it has reached a base case: an empty entry list or a leaf node. Otherwise, it recursively deletes the entries in the subtrees and collects the updated subtrees. If at least B subtrees survive (line 13), we build a node wraps them and return it (line 15). If fewer survive, node T underflows. The algorithm removes it and returns $\emptyset$ to mark its absence (line 14). The surviving subtrees of the underflow node are set aside and later reinserted into the global PaB-tree. This involves collecting all dropped subtrees into an array in parallel, using a prefix sum over per-node counts to compute offsets, and then group subtrees by height for reinsertion.

Reinserting these subtrees uses the same insertion algorithm (Alg. 2). The only difference is that the inserted items are subtrees instead of leaf entries. The logic of Alg. 2 still applies to both cases, because inserting a height-h subtree into a node at height $h+1$ is the same operation as inserting an entry into a leaf.

Theoretical Analysis. We now analyze the complexity of batch deletion, which is summarized in Thm. 3.3. The high-level idea is to amortize the cost of reinsertion over the deletions that caused it. A node is created with at least $2B$ children and is dropped only when fewer than B elements remain, so at least B deletions must happen below it before triggering a reinsertion. Those B deletions together have paid $O(B \log_B n)$, which covers reinserting the at most B surviving subtrees.

Theorem 3.3. *Given a PaB-tree of size n, batch deleting m entries takes $O(Bm \log_B n)$ amortized work, $O(m \log_B n)$ amortized I/O cost, and $O(\log^2 n \log(m+B))$ span.*

Proof. We analyze the complexity by dividing the deletion process into two phases: the DELETESORTED execution and the subsequent subtree reinsertions.

We first bound the I/O cost. The DELETESORTED phase recursively routes the deleted entries down to the tree leaves and removes them. Since this phase requires no structural repairs and follows a simpler access pattern than INSERTSORTED, its I/O cost is $O(m \log_B n)$.

The reinsertion phase requires an amortized analysis. By the definition of the PaB-tree structure, any valid tree node is initially constructed with a fan-out of at least $2B$. During deletion, a node is dropped only if the number of its remaining subtrees falls below the degree bound B. This implies that at least B subtrees must have been previously dropped to trigger the drop of the current node. Generalizing this property, we know that for a subtree at height h (where leaves are at $h = 0$) to underflow and be dropped, at least B^h entries must have been deleted from it.

When such a subtree drops, we collect its remaining valid

subtrees and reinsert them into the PaB-tree. Since there are at most B subtrees left, reinserting them incurs an I/O cost of $O(B \log_B n)$. We can charge this cost to the previously deleted entries in this subtree following a standard amortized analysis: each previously deleted entry pays $O((B \log_B n)/B^h) = O((\log_B n)/B^{h-1})$ for the restructuring of its height-h ancestor. Summing this geometric series across all heights from $h = 1$ to $O(\log_B n)$ yields an amortized cost of $O(\log_B n)$ per entry. Combining this with the $O(\log_B n)$ per-entry I/O cost from the DeleteSorted phase, we obtain an overall I/O cost of $O(m \log_B n)$ for batch deletion.

The work bound analysis is similarly structured. In the DeleteSorted phase, the algorithm performs simple data scanning and key routing without structural repairs, and its work is bounded by a factor of B times its I/O cost. For the reinsertion phase, because the algorithm directly reuses the batch insertion subroutine, its work likewise remains a factor of B times its I/O cost. Scaling the overall amortized I/O cost by B yields an amortized work complexity of $O(Bm \log_B n)$.

Finally, we analyze the span bound. The span of the DeleteSorted phase is bounded by that of the InsertSorted phase. However, rather than inserting all dropped subtrees simultaneously, we group the dropped subtrees by height and iteratively reinsert each group level by level. This iterative approach adds an extra $O(\log_B n)$ factor to the span depth. Multiplying the span of the insertion algorithm by this factor yields an overall span bound of $O(\log^2 n \log(m + B))$. This extra log factor could be removed by reinserting all heights together, at the cost of redesigning insertion. Since the span is already polylogarithmic and parallelism is strong in practice, we keep the simpler design. □

3.4 Optimizations

To better accommodate dynamic workloads, PaB-tree incorporates a probabilistic leaf-sorting strategy controlled by the *leaf randomness ratio* α: an update leaves a leaf unsorted with probability α, and re-sorts it otherwise. Accordingly, queries scan an unsorted leaf linearly instead of using a binary search. For write-heavy workloads, this reduces the work of batch updates (e.g., insertions can directly append), at the cost of slightly slower queries. Note that the I/O cost is not affected because once a leaf node is loaded into cache, scanning it incurs no additional cache misses. We use $\alpha = 0.5$ by default, but a user can adjust it according to the update/query ratio of their target workload. We experimentally evaluate this strategy in Sec. 4.5.

4 Experiments

We conduct comprehensive experiments to evaluate the performance of PaB-tree against several baselines on both synthetic and real-world datasets. Our results show that PaB-tree offers a clear advantage in batch updates while delivering competitive or better query performance compared to existing tree-based baselines.

Experimental Setup. We perform experiments on a machine with 112 cores (224 hyper-threads), four Intel Xeon Platinum 8176 CPUs, and 1.47 TB RAM. We implement PaB-tree in `C++` and compile it with GCC 15.2.0 with the `-O3` optimization flag. We use ParlayLib [12] for fork-join parallelism. The reported numbers are the average of 3 runs after a warm-up run. We use block size $B = 64$[1] and set the leaf randomness ratio $\alpha = 0.5$. The entries are sorted when the tree is initialized. We release our code at [51].

Baselines. We compare with several state-of-the-art batch-parallel and concurrent data structures. For the parallel case, we evaluate (1) ***PAM*** [65], a parallel binary search tree, (2) ***CPAM*** [26], which improves the cache-friendliness of PAM with leaf wrapping, and (3) ***PMA*** [73, 77], a Packed Memory Array that stores all entries in a single contiguous sorted array, optimized for queries. For the concurrent case, we evaluate (1) ***Mass-tree*** [50] and (2) ***Bw-tree*** [72], two well-known in-memory concurrent B^+-tree implementations.

4.1 Evaluation on Synthetic Benchmark

Setup. We test different batch update patterns and key distributions. We test both the static and dynamic cases. In the static case, we measure the time to construct a tree with 10^9 entries. In the dynamic case, we measure both the incremental insertion and deletion. For insertion, we divide $n = 10^9$ entries into batches of size m and insert them into the tree n/m times, starting from an empty tree. For deletion, we first construct a tree with all entries, then similarly remove them in batches until the tree is empty. We report the total time of all batches. Under this workload, we also measure the query performance when half of the batches are applied. We evaluate membership (Find) and range queries (Range-Report): 10^8 random Find queries and 5×10^4 Range-Report queries, with range sizes spanning 10^2 to 10^4. The query implementations all directly follow the standard B$^+$-tree algorithms. Each query runs serially, with different queries in parallel. We also test the update time for a single batch, compare it with that of sequential point updates, and show the results in Secs. A and D.

We use three key distributions for batches: uniform (`Uni`), skewed (`Skew`), and adversarial (`Adv`). `Uni` generates keys uniformly at random within the range. `Skew` generates keys from a normal distribution with mean 0.5 and standard deviation 0.1, which simulates the real-world scenario where the data in a batch has certain locality. `Adv` sorts the keys before dividing them into batches, so that each batch contains a consecutive range of keys. This simulates the worst-case scenario for PaB-tree, since the batch updates always trigger the largest possible subtrees to overflow or underflow, maximizing the cost of node splitting and reinsertion. However, this is the best case for PAM and CPAM since their

[1] Evaluation for different block sizes is shown in Sec. B.

Dist.	Trees	Build	Incremental Insert				Query after Inc. Ins.		Incremental Delete				Query after Inc. Del.	
			Batch Size				Find	Range Report	Batch Size				Find	Range Report
			0.01%	0.1%	1%	10%			0.01%	0.1%	1%	10%		
Uniform	PaB-tree	**1.25**	**29.4**	**15.3**	**5.06**	**1.68**	.099	.078	**19.6**	**11.5**	**5.40**	1.77	.078	**.085**
	PAM	2.11	29.8	19.2	11.8	5.70	.128	.496	28.8	18.7	11.3	5.50	.130	.497
	CPAM	1.70	55.2	30.9	7.29	1.81	.098	.088	54.4	30.9	7.03	**1.58**	.099	.087
	PMA	3.45	30.1	19.0	13.3	9.88	**.031**	**.073**	22.0	16.6	8.92	3.01	**.034**	.099
	Mass-tree	96.6	88.4	77.8	77.0	81.4	.082	.594	64.4	55.2	53.7	53.9	.103	.813
	Bw-tree	39.6	55.8	39.6	33.3	32.9	.222	.455	63.2	43.8	39.6	39.1	.235	.632
Skewed	PaB-tree	**1.25**	**25.7**	**11.7**	**3.65**	**1.41**	.073	**.079**	**19.2**	**9.96**	**3.70**	1.82	.074	.087
	PAM	2.18	27.0	16.6	9.26	3.80	.108	.500	25.8	16.2	8.80	3.76	.124	.501
	CPAM	1.79	48.4	21.0	5.10	1.56	.081	.087	48.0	21.1	4.43	**1.36**	.093	**.087**
	PMA	3.40	39.3	27.8	20.3	9.68	**.025**	.095	27.1	22.1	11.4	4.30	**.030**	.107
	Mass-tree	87.8	86.4	75.7	75.8	79.3	.067	.554	64.0	54.6	53.3	53.4	.088	.817
	Bw-tree	39.5	52.9	38.9	33.2	33.0	.161	.459	59.6	42.0	36.1	35.6	.174	.667
Adversarial	PaB-tree	**1.34**	7.75	4.50	2.55	1.99	.044	**.081**	7.14	2.60	1.68	1.36	.043	**.086**
	PAM	2.33	8.40	5.52	4.57	4.07	.069	.503	8.96	4.90	3.66	3.35	.076	.501
	CPAM	1.81	**7.59**	**3.42**	**2.23**	**1.76**	.056	.087	**4.96**	**2.13**	**1.35**	**1.13**	.063	.088
	PMA	3.59	192.0	82.2	36.2	14.2	**.018**	.096	143.4	74.7	20.0	3.83	**.021**	.106
	Mass-tree	87.5	84.7	78.1	76.5	79.9	.064	.580	64.6	55.6	54.5	54.6	.065	.738
	Bw-tree	39.2	55.1	36.8	32.2	33.5	.109	.450	61.7	38.6	35.2	36.2	.113	.534

The **fastest time** is in bold and underlined : < 1.1x the fastest : < 1.5x the fastest : < 2x the fastest : < 5x the fastest : > 5x the fastest

Figure 4. Running time (in seconds) on synthetic data. Lower is better. The fastest running time for each dataset is bold and underlined. We use different colors to mark results within 1.1×, 1.5×, 2×, 5× and > 5× of the fastest time. Detailed settings for build, queries, and incremental insertion/deletion are introduced at the beginning of Sec. 4.1.

algorithms reduce to simply merging in or splitting out a subtree on the right spine. We use 32-bit integer keys and values, and summarize the results in Fig. 4.

Construction. For tree construction, our PaB-tree is consistently faster than all baselines across all workloads. This advantage mainly comes from the shallower tree height and the blocked node layout of the B^+-tree, which enable fewer memory allocations and yield better cache behavior. Both PAM and CPAM are binary trees with higher tree heights, and are therefore up to 1.7× and 1.4× slower than PaB-tree, respectively. The PMA is up to 2.8× slower than PaB-tree, possibly due to the additional work required to scatter entries across the underlying array. More detailed hardware profiling is provided in Sec. 4.4. Both concurrent indexes, Mass-tree and Bw-tree, are orders of magnitude slower than PaB-tree, as concurrent insertion on a large batch incurs high node contention.

Batch Updates. For batch updates, PaB-tree retains its advantage in most cases. It is always faster than the state-of-the-art BST PAM (by up to 3.4×). In contrast, while CPAM also leverages larger node sizes to improve I/O performance, it can be up to 1.9× slower than the standard BST in PAM, particularly on small batch sizes. Altogether, on the `Uni` and `Skew` cases, PaB-tree is also almost always faster than CPAM, by up to 2.8×.

On large deletion batches (10%) of `Uni` and `Skew` data, and more markedly on `Adv` data, CPAM outperforms PaB-tree. As noted earlier, the `Adv` workload consistently inserts m keys strictly larger than the tree's maximum value or deletes the m smallest keys. This represents the worst-case scenario for PaB-tree because it is more prone to triggering node overflows and underflows; hence, we include this workload as a stress test for PaB-tree. Interestingly, this is the best-case scenario for PAM and CPAM, as their Join-based algorithms simply merge or split trees along the rightmost path, reducing the cost to $O(\log n + m)$, rather than $O(m \log n)$ in the general case. In this adversarial case, PaB-tree is expectedly slower than CPAM, but only by 1.02–1.32× for insertions and 1.20–1.44× for deletions. It also remains faster than PAM at every batch size, by up to 2.0× for insertions and 2.5× for deletions.

Compared to the other baselines, including PMA, Mass-tree, and Bw-tree, PaB-tree is faster in every batch-update case. As an array-based structure, PMA optimizes queries at the cost of update efficiency. As a result, PaB-tree is on average 4.0× faster (up to 28.7×) than PMA on updates. Furthermore, both concurrent data structures perform worse than batch-parallel ones on batch updates, since larger batches increase contention rather than parallelism; PaB-tree achieves an average speedup of 13.1× (up to 56×) over Mass-tree and 7.9× (up to 27×) over Bw-tree.

Queries. As expected, B-tree-like structures, including PaB-tree, Mass-tree, and even CPAM, perform better than the strict BST PAM on all queries. In Find queries, PMA has a significant advantage over all other implementations, due to its array-based layout and query-focused optimizations. However, this comes at the cost of updates that are up to 28.7× slower than those of PaB-tree. Apart from PMA, PaB-tree is almost always the fastest, or within 10% of the fastest, among the tree-based structures in Find queries. For RangeReport queries, PaB-tree is the fastest in four of the six cases and within 7% of the fastest in the other two. It achieves speedups of up to 6.4× over PAM, up to 1.1× over CPAM, up

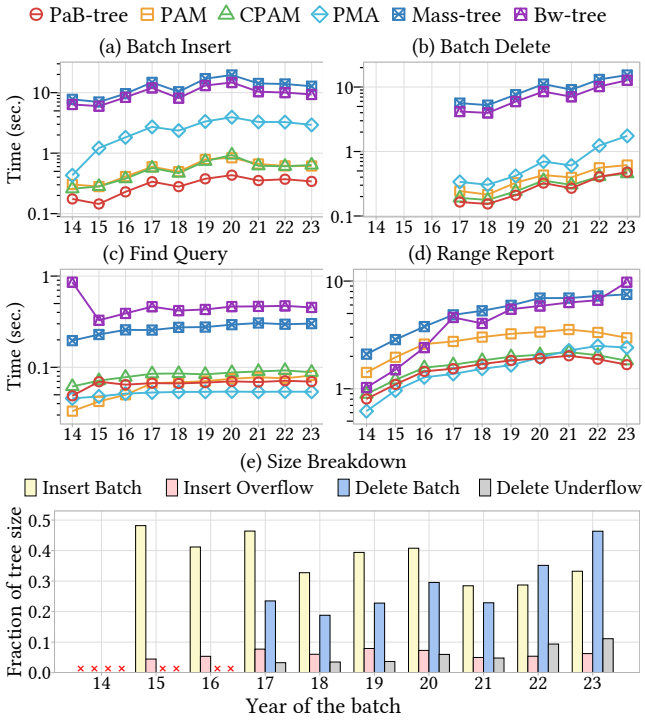


Figure 5. Running time (in seconds) of batch updates and queries on the OSM dataset. Lower is better. The inputs are batched by years from 2014 to 2023. In each year, we insert the batch in that year and then delete the batch from three years ago if applicable. After batch updates, we perform the queries on the current index. The FIND query contains 10^8 candidates, and the RANGE-REPORT query contains 5×10^5 queries with output size between $10^2 - 10^4$. Both queries are generated randomly. The yellow and blue bars depict the fraction of input batch sizes w.r.t the tree size in each year. The pink and grey bars show the fraction of overflow and underflow size w.r.t the tree size during the batch insertion and deletion, respectively.

to 1.2× over PMA, and 5.6–9.6× over Mass-tree and Bw-tree.

4.2 Processing Real-world Geometric Data

We evaluate PaB-tree as a general key-value store on a real-world geometric dataset.

Setup. We use `OpenStreetMap` (`OSM`) [42] as the benchmark. For each record in `OSM`, we use a 64-bit Morton code that encodes its geographical information as the key, and a 32-bit integer for its timestamp as the value. We divide the 1.35B inputs into 10 batches by their timestamps, from year 2014 to 2023. We simulate a sliding-window setting following the methodology of [53] with window length 3. In each year, we insert the corresponding batch and, where applicable, delete the batch from three years prior. After the update for each year is applied, we run membership queries for 10^8 randomly picked entries and 5×10^5 range-report queries with output sizes in $[10^2, 10^4]$.

Results. We summarize the results in Fig. 5. PaB-tree is consistently faster than all baselines for batch updates. For

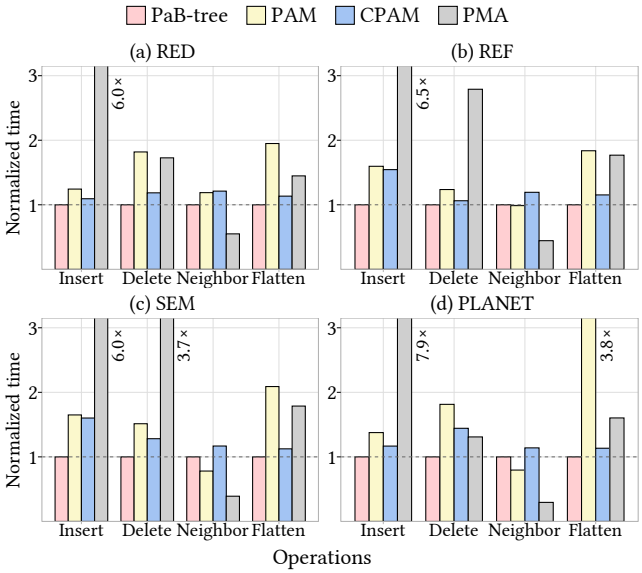


Figure 6. Normalized running time of batch updates and queries on temporal graph datasets. Lower is better. Time exceeds 3 times slower than PaB-tree is chopped. Detailed settings can be found in Sec. 4.3. For all operations, we measure the sum of running time across all batches and normalized to the PaB-tree. The NEIGHBOR query lists all neighbors for 10^7 vertices with the largest degree. The FLATTEN query outputs all entries in a sorted order.

insertions, PAM suffers from its large tree height, and CPAM must maintain a total order within its leaves; these make them slower than PaB-tree by 2.09× and 1.95×, respectively. For deletions, the gap between PMA and the other parallel baselines is smaller, since deletion involves fewer memory-allocation issues. Even though reinserted entries are up to 11.2% of the total tree size, PaB-tree stays up to 1.16× faster than CPAM, showing the practical value of its I/O efficiency.

For FIND queries, PMA benefits from the absence of pointer chasing and is the fastest among all baselines, 1.07–1.45× faster than PaB-tree. Its advantage on RANGE-REPORT queries is smaller, since the output size dominates the query cost. PAM provides better FIND performance than PaB-tree and CPAM at small tree sizes, since the latter two must scan all entries in a leaf to find the target key. Meanwhile, without leaf blocking, PAM has poor cache locality when reporting many leaf entries, which also makes it much slower than the others for RANGE-REPORT queries. Both concurrent baselines, Mass-tree and Bw-tree, are orders of magnitude slower than PaB-tree for both queries, as locks block the queries.

4.3 Processing Real-world Graph Data

We evaluate PaB-tree as a graph container on temporal graphs. Every edge in a temporal graph has a timestamp, and we group edges into batches by their timestamps (in years) to simulate the evolution of the graph over time.

Setup. We compare the baselines on four benchmarks from GraphTide [76]: `RED` (746M edges, in 17 batches), `REF` (1.14B

edges, in 27 batches), SEM (2.49B edges, in 229 batches), and PLANET (1.01B edges, in 20 batches). The batch sizes generally grow over the years. For graphs without explicit insert or delete tags on their edges, such as RED, REF, and SEM, we simulate the sliding-window setting as in Section 4.2 with window length 3; otherwise, we form the insertion and deletion batches according to those tags. For each year, we first apply the batched insertions, then the deletions. After the batch updates in each year, we run two fundamental primitives in graph algorithms: (1) `Neighbor`(V), which retrieves the neighbors of every vertex in V, and (2) `Flatten`, which returns all edges in sorted order, forming a compressed-sparse-row-like (CSR-like) array. `Flatten` is widely used in dynamic graph systems to extract the graph prior to query execution (e.g., the *flat snapshot* operation in Aspen [27]). We encode each edge as a 64-bit integer and use it as the key, with another 32-bit integer as the value for its timestamp. We sum each operation's running time over all years and normalize the total to PaB-tree.

Results. We summarize the results in Fig. 6. For both batch insertion and batch deletion, PaB-tree is the fastest among all baselines, achieving 1.09–1.60× speedup over the fastest baseline CPAM on insertion and 1.06–1.44× speedup on deletion. This is consistent with the findings in Sec. 4.2: we believe that keeping leaf entries unsorted is the main reason for PaB-tree's strong batch-update performance, while the overhead of reinsertion-based deletion remains modest in practice. As a static index, PMA is much slower than the others under highly dynamic updates, due to frequent memory allocation and reconstruction; this aligns with the findings on the synthetic datasets (Section 4.1).

For graphs with a small average vertex degree (≤ 10), such as PLANET and SEM, Neighbor degenerates to Find, which favors PAM. For large-degree graphs (RED), this advantage diminishes. For Flatten, PAM is the slowest due to excessive pointer chasing, while PMA must skip empty slots in its underlying array; PaB-tree and CPAM are comparable.

4.4 Hardware Profiling

We profile the CPU cycles, instruction count, and cache misses of tree construction, incremental batch updates, and Find queries on the Skew distribution, with 10^9 entries and batches with size of 10^7. The instruction count reflects the total work of the algorithm, and the cache misses reflect its I/O efficiency. We summarize the results in Fig. 7 and provide details in Sec. C.

PaB-tree benefits the most from its I/O efficiency, which gives it the best performance on tree construction and batch insertion. The low instruction count from the algorithm's simplicity also contributes to the performance gains of PaB-tree. Compared with tree indexes, the PMA incurs the fewest cache misses for Find queries due to the absence of pointer chasing in inner tree structures, which makes it the fastest

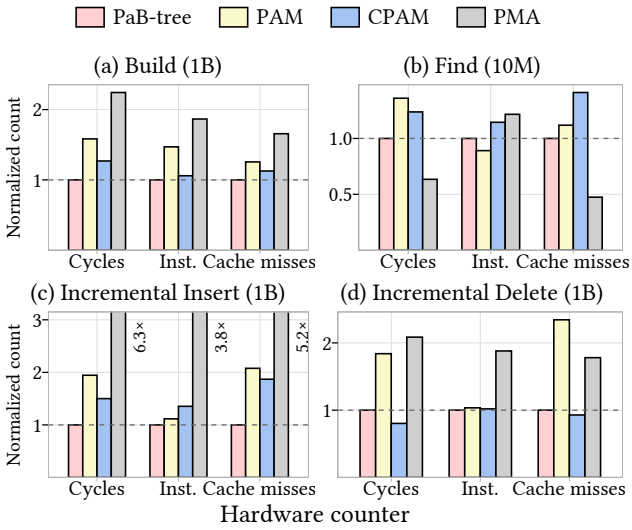


Figure 7. Profiling the cycles, instructions and cache-misses of tree construction, incremental batch updates, and membership queries on Skew distribution. Lower is better. The numbers are normalized to PaB-tree. We use the Skew dataset with batch size of 10^7, in total 10^9 entries are processed.

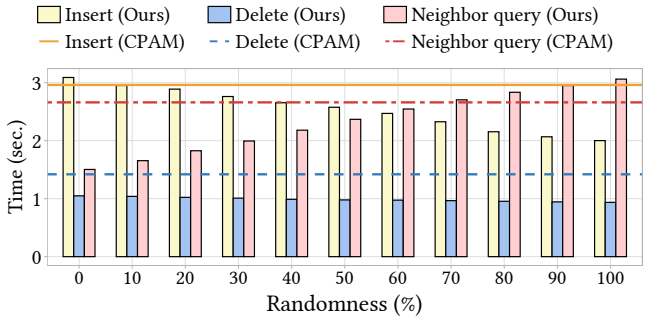


Figure 8. Running time (in seconds) of batch updates and Neighbor queries w.r.t different randomness ratio in PaB-tree. Lower is better. We use the temporal graph datasets PLANET. The update edges are divided into 20 batches by their timestamps. For each timestamp, after the batch insertion and deletion, we run 10^7 Neighbor queries. The bar shows the sum of the running time of the batch updates and the Neighbor queries. The corresponding times for CPAM are marked in lines.

for Find queries. However, incremental batch insertion frequently triggers memory allocation and reconstruction of overflowed blocks; as a result, the large number of cache misses and instruction counts make the PMA the slowest for batch insertion. The leaf blocking in CPAM enables it to have a lower tree height than the PAM and therefore fewer cache misses needed to traverse the tree, making it consistently faster than PAM for all operations.

4.5 Evaluation of Randomness in Leaves

We evaluate the impact of sortedness in the leaves on the performance of PaB-tree on the temporal graph dataset PLANET and summarize the results in Fig. 8. The randomness ratio α

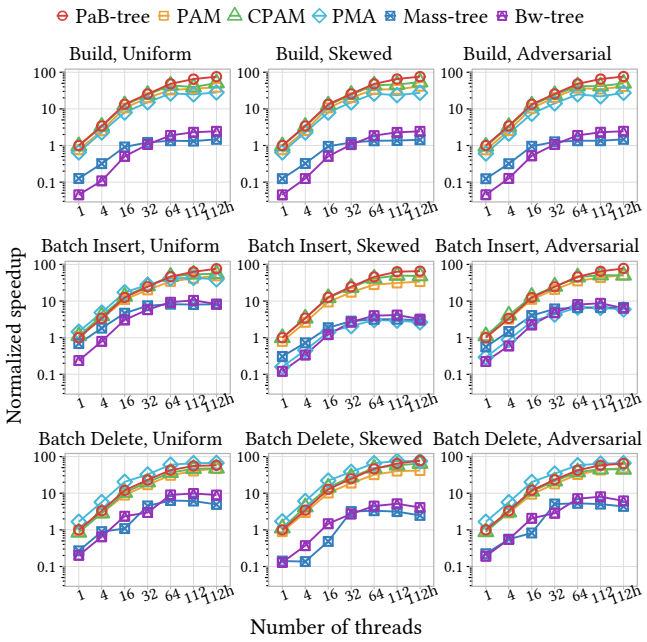


Figure 9. Normalized parallel speedup of tree construction, batch insertion and batch deletion on different number of processors. Higher is better. The speedup is normalized to the running time on the PaB-tree on 1 thread respectively. The 10^9 entries, and batch updates apply a single batch of 10^7 entries. "112h" means using all 112 cores (224 hyper-threads).

controls the degree of randomness in the leaf entries, with $\alpha = 0$ corresponding to a fully sorted leaf and $\alpha = 1$ corresponding to a fully random leaf. Sorted leaves improve query performance at the cost of more expensive batch updates.

Both batch insertion and deletion become faster if we tolerate more randomness in the leaf. Insertion is more sensitive to the ratio than deletion. This comes from the asymmetry between the two operations. With a larger randomness α, an insertion will likely just append new entries to an unsorted leaf, whereas a sorted leaf requires merging, resulting in a more significant difference between the two cases. On the other hand, a deletion has to filter the leaf in one pass regardless of whether it is sorted, so α minimally affects its leaf-level cost. The remaining effect of α on deletion comes from the reinsertions it triggers, which follow the insertion path. Since the reinserted subtrees are usually much smaller than the batch itself, α affects the total deletion time much less than the insertion time.

A Neighbor query needs to locate the start and end of the range in the leaf. A sorted leaf enables binary search, which is much faster than linear search in a random leaf. As a result, the query time increases as the randomness ratio grows. Overall, the randomness ratio $\alpha = 0.5$ provides a good balance between batch update and query performance.

4.6 Scalability

We also evaluate the scalability of the tested indexes on write operations and report the results in Fig. 9. Overall, except for the two concurrent indexes, which do not scale due to lock contention, all indexes scale well on 224 hyper-threads. This indicates that their performance is mostly bounded by the work and I/O efficiency of the underlying algorithms. PaB-tree generally achieves the best scalability for tree construction on 224 hyper-threads (75.6×), much better than the fastest baseline CPAM (50.7×). Similar trends appear for batch updates as well. PMA also scales well for single batch updates, since applying one batch of 1% of the tree size rarely triggers a reconstruction.

5 Related Work

Parallel B^+-trees. Since the B^+-tree [7, 23] was introduced in the 1970s, many parallel variants have been proposed in the literature. Early parallel B^+-trees in the PRAM model focused on improving the bounds on construction [25] and updates [19, 43, 48, 58]. These approaches are based on pipelining and rely on cycle-based synchronization, which makes them unsuited for modern hardware. Goodrich et al. [35] attempted to adapt the Join-based framework [14] to B^+-trees; however, the resulting algorithm exhibits high design complexity, posing significant challenges for practical implementation. These works are purely theoretical, however.

Many practical implementations—for DRAM [47, 67, 72, 81], disk [9, 60, 63], persistent memory [21, 30, 49], and GPUs [24, 82, 83]—rely on *locks* to handle node splits/merges. When processing batch updates, such designs inevitably incur high synchronization overhead due to intense lock contention, which degrades the performance.

Various batch-parallel BSTs have been developed. Our main baseline, PAM [65], uses weight-balanced trees. Later works, Aspen [27] and CPAM [26], extend PAM with enlarged tree nodes to improve I/O performance, yet preserve the binary layout, partly to support functional updates via path copying. Unlike these systems, we focus on in-place updates. Ultimately, their binary structure prevents them from achieving non-trivial I/O bounds.

Several lock-free parallel B^+-trees exist. The PALM tree [61] performs batch updates in the Bulk Synchronous Parallel (BSP) model [34]. Like PaB-tree, it splits overflow nodes during insertion and reinserts underflow nodes. However, its BSP design targets communication cost and load balancing, exposing limited parallelism: work within each node is pinned to a single thread, so underflow reinsertion is sequential. It also provides no I/O optimization or theoretical guarantees. Akhremtsev et al. [4] present a parallel (a, b)-tree with provable work and span bounds for batch updates. However, their analysis assumes constant a and b, and therefore also does not yield I/O bounds. Since their code is not publicly available, we could not include it in our evaluation.

Concurrent Search Indexes. Concurrent B/B$^+$-trees, such as Mass-tree [50] and Bw-tree [72], are widely studied [8, 20, 22, 38, 47, 54, 56, 80] due to their prevalence in storage systems and databases. Concurrent data structures target a different setting of simultaneous point updates, and thus rarely deliver peak performance on batch-parallel workloads. However, applying updates individually makes single-update latency independent of batch size. On the other hand, parallel structures (including our PaB-tree) suit batch-based applications such as those in graph processing [27, 57, 75].

6 Conclusion

We introduce PaB-tree, the first batch-parallel B^+-tree to pair provable guarantees on work, span, and cache complexity with state-of-the-art practical performance. Its design rests on a single invariant—a repair never touches a sibling and is always pushed to the parent—one that reduces every overflow and underflow to one uniform, PROMOTE-based mechanism. This simplicity is what makes PaB-tree scale: repairs stay on the ancestor path, so the divide-and-conquer structure exposes ample parallelism with little coordination overhead. As a result, across synthetic, key–value, and temporal-graph workloads, PaB-tree achieves the fastest batch updates among the baselines while remaining competitive on queries. Natural next steps include extending it to the external- and persistent-memory settings and interleaving concurrent point queries with batched updates.

7 Acknowledgments

This work is supported by NSF grants IIS-2227669, TI-2346223, NSF CAREER Awards CCF-2238358, CCF-2339310, and OAC-2543534, and the Google Research Scholar Program.

References

[1] Umut A Acar, Daniel Anderson, Guy E Blelloch, and Laxman Dhulipala. 2019. Parallel batch-dynamic graph connectivity. In *ACM Symposium on Parallelism in Algorithms and Architectures*. 381–392.

[2] M AdelsonVelskii and Evgenii Mikhailovich Landis. 1963. *An algorithm for the organization of information.* Technical Report. DTIC Document.

[3] Alok Aggarwal and Jeffrey S. Vitter. 1988. The input/output complexity of sorting and related problems. *Commun. ACM* 31, 9 (1988), 1116–1127. doi:10.1145/48529.48535

[4] Yaroslav Akhremtsev and Peter Sanders. 2016. Fast Parallel Operations on Search Trees. In *IEEE International Conference on High Performance Computing (HiPC)*.

[5] Daniel Anderson, Guy E. Blelloch, and Kanat Tangwongsan. 2020. Work-Efficient Batch-Incremental Minimum Spanning Trees with Applications to the Sliding-Window Model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[6] Nimar S Arora, Robert D Blumofe, and C Greg Plaxton. 2001. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems (TOCS)* 34, 2 (2001), 115–144. doi:10.1007/s00224-001-0004-z

[7] Rudolf Bayer and Edward McCreight. 1970. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. 107–141.

[8] Rudolf Bayer and Mario Schkolnick. 1977. Concurrency of operations on B-trees. *Acta informatica* 9, 1 (1977), 1–21.

[9] Michael A Bender, Jeremy T Fineman, Seth Gilbert, and Bradley C Kuszmaul. 2005. Concurrent cache-oblivious B-trees. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*. 228–237.

[10] Timo Bingmann. 2013. STX B+ Tree C++ Template Classes. https://panthema.net/2007/stx-btree/.

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

[12] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib—A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*. 507–509. doi:10.1145/3350755.3400254

[13] Guy E Blelloch and Magdalen Dobson. 2022. Parallel Nearest Neighbors in Low Dimensions with Batch Updates. In *Algorithm Engineering and Experiments (ALENEX)*. SIAM, 195–208. doi:10.1137/1.9781611977042.16

[14] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. 2016. Just Join for Parallel Ordered Sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[15] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. 2020. Optimal Parallel Algorithms in the Binary-Forking Model. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*. 89–102. doi:10.1145/3350755.3400227

[16] Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. 2010. Low depth cache-oblivious algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 189–199. doi:10.1145/1810479.1810519

[17] Guy E. Blelloch, Yan Gu, Yihan Sun, and Kanat Tangwongsan. 2016. Parallel Shortest Paths Using Radius Stepping. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 443–454.

[18] Robert D. Blumofe and Charles E. Leiserson. 1998. Space-Efficient Scheduling of Multithreaded Computations. *SIAM J. on Computing* 27, 1 (1998), 202–229. doi:10.1137/s0097539793259471

[19] Anastasia Braginsky and Erez Petrank. 2012. A lock-free B+ tree. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*. 58–67.

[20] Sang K. Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. 2001. Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems. In *Proc. 27th International Conference on Very Large Data Bases (VLDB)*. 181–190.

[21] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. 2020. uTree: a persistent B+-tree with low tail latency. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2634–2648.

[22] Jiwon Choe, Andrew Crotty, Tali Moreshet, Maurice Herlihy, and R Iris Bahar. 2022. Hybrids: Cache-conscious concurrent data structures for near-memory processing architectures. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*. 321–332.

[23] Douglas Comer. 1979. Ubiquitous B-tree. *ACM computing surveys (CSUR)* 11, 2 (1979), 121–137.

[24] Mayank Daga and Mark Nutter. 2012. Exploiting coarse-grained parallelism in B+ tree searches on an APU. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, 240–247.

[25] Narsingh Deo, Amit Jain, and Muralidhar Medidi. 1994. Parallel construction of (a, b)-trees. *J. Parallel and Distrib. Comput.* 23, 3 (1994), 442–448.

[26] Laxman Dhulipala, Guy E. Blelloch, Yan Gu, and Yihan Sun. 2022. PaC-trees: Supporting Parallel and Compressed Purely-Functional Collections. In *ACM Conference on Programming Language Design and Implementation (PLDI)*.

[27] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. 2019. Low-

latency graph streaming using compressed purely-functional trees. In *ACM Conference on Programming Language Design and Implementation (PLDI)*. 918–934.

[28] Xiangyun Ding, Yan Gu, and Yihan Sun. 2024. Parallel and (Nearly) Work-Efficient Dynamic Programming. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*. 219–232. doi:10.1145/3626183.3659958

[29] Xiaojun Dong, Yan Gu, Yihan Sun, and Yunming Zhang. 2021. Efficient Stepping Algorithms and Implementations for Parallel Shortest Paths. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 184–197.

[30] Mingzhe Du and Michael L Scott. 2025. Buffered persistence in b+ trees. In *Proceedings of the 3rd Highlights of Parallel Computing Workshop*. 34–36.

[31] Jeremy T. Fineman. 2018. Nearly Work-Efficient Parallel Algorithm for Digraph Reachability. In *ACM Symposium on Theory of Computing (STOC)*. 457–470.

[32] Leonor Frias and Johannes Singler. 2007. Parallelization of bulk operations for STL dictionaries. In *European Conference on Parallel Processing*. Springer, 49–58.

[33] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. 1999. Cache-Oblivious Algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*. 285–297. doi:10.1109/sffcs.1999.814600

[34] Alexandros V Gerbessiotis and Leslie G Valiant. 1994. Direct bulk-synchronous parallel algorithms. *Journal of parallel and distributed computing* 22, 2 (1994), 251–267.

[35] Michael T Goodrich, Yan Gu, Ryuto Kitagawa, and Yihan Sun. 2025. Parallel Joinable B-Trees in the Fork-Join I/O Model. (2025).

[36] Google. 2018. Abseil C++ Common Libraries: B-tree Containers. https://github.com/abseil/abseil-cpp. `absl::btree_map`, `absl::btree_set`, `absl::btree_multimap`, `absl::btree_multiset`. Design rationale: https://abseil.io/about/design/btree. Accessed 2026-08-02.

[37] Google. 2024. `btree`: An Ordered, In-Memory B-Tree for Go. https://github.com/google/btree.

[38] Goetz Graefe. 2010. A survey of B-tree locking techniques. *ACM Transactions on Database Systems (TODS)* 35, 3 (2010), 1–26.

[39] Yan Gu, Ziyang Men, Zheqi Shen, Yihan Sun, and Zijin Wan. 2023. Parallel Longest Increasing Subsequence and van Emde Boas Trees. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[40] Yan Gu, Zachary Napier, and Yihan Sun. 2022. Analysis of Work-Stealing and Parallel Cache Complexity. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*. SIAM, 46–60. doi:10.1137/1.9781611977059.4

[41] Leo J Guibas and Robert Sedgewick. 1978. A Dichromatic Framework for Balanced Trees. (1978).

[42] Mordechai Haklay and Patrick Weber. 2008. Openstreetmap: User-generated street maps. *IEEE Pervasive computing* 7, 4 (2008), 12–18.

[43] Lisa Higham and Eric Schenk. 1994. Maintaining B-trees on an EREW PRAM. *Journal of parallel and distributed computing* 22, 2 (1994), 329–335.

[44] Hongbo Kang, Phillip B Gibbons, Guy E Blelloch, Laxman Dhulipala, Yan Gu, and Charles McGuffey. 2021. The Processing-in-Memory Model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 295–306.

[45] Hongbo Kang, Phillip B Gibbons, Guy E Blelloch, Laxman Dhulipala, Yan Gu, and Charles McGuffey. 2023. A Skew-Resistant Trie for Processing-in-Memory. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[46] Hongbo Kang, Yiwei Zhao, Guy Blelloch, Laxman Dhulipala, Yan Gu, Charles McGuffey, and Phillip Gibbons. 2023. PIM-tree: A Skew-resistant Index for Processing-in-Memory. In *Proceedings of the VLDB Endowment (PVLDB)*.

[47] Philip L Lehman and S Bing Yao. 1981. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems (TODS)* 6, 4 (1981), 650–670.

[48] Wei Quan Lim. 2019. Optimal multithreaded batch-parallel 2-3 trees. *arXiv preprint arXiv:1905.05254* (2019).

[49] Yibo Liu, Bing Wu, Shiyi Song, Xueliang Wei, Huan Cheng, Heng Zhou, Wei Tong, and Dan Feng. 2026. pTree: Building Efficient B+-Tree on Non-Volatile Memory With Processing-in-Memory. *IEEE Trans. Comput.* (2026).

[50] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*. 183–196.

[51] Ziyang Men, Yan Gu, Yihan Sun, and Helen Xu. 2026. Implementation for PaB-tree. https://github.com/ucrparlay/pab-tree.

[52] Ziyang Men, Bo Huang, Yan Gu, and Yihan Sun. 2026. Parallel Dynamic Spatial Indexes. In *Proceedings of the 31st ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 150–163. doi:10.1145/3774934.3786412

[53] Ziyang Men, Zheqi Shen, Yan Gu, and Yihan Sun. 2025. Parallel kd-tree with Batch Updates. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–26. doi:10.1145/3709712

[54] Jacob Nelson-Slivon, Ahmed Hassan, and Roberto Palmieri. 2022. Bundling linked data structures for linearizable range queries. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 368–384.

[55] Jürg Nievergelt and Edward M Reingold. 1973. Binary search trees of bounded balance. *SIAM J. on Computing* 2, 1 (1973), 33–43.

[56] 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 *Proceedings of the 2016 International Conference on Management of Data*. 371–386.

[57] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluc. 2021. Terrace: A hierarchical graph container for skewed dynamic graphs. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 1372–1385.

[58] Heejin Park, Kunsoo Park, and Yookun Cho. 2004. Deleting keys of B-trees in parallel. *J. Parallel and Distrib. Comput.* 64, 9 (2004), 1041–1050.

[59] Wolfgang J. Paul, Uzi Vishkin, and Hubert Wagener. 1983. Parallel Dictionaries in 2-3 Trees. In *Intl. Colloq. on Automata, Languages and Programming (ICALP)*. 597–609.

[60] Hongchan Roh, Sungho Kim, Daewook Lee, and Sanghyun Park. 2014. AS B-tree: A Study of an Efficient B+-tree for SSDs. *J. Inf. Sci. Eng.* 30, 1 (2014), 85–106.

[61] Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Satish, and Pradeep Dubey. 2011. PALM: Parallel architecture-friendly latch-free modifications to B+ trees on many-core processors. *Proceedings of the VLDB Endowment* 4, 11 (2011), 795–806.

[62] Julian Shun and Guy E Blelloch. 2014. Phase-concurrent hash tables for determinism. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 96–107.

[63] Venkathachary Srinivasan and Michael J Carey. 1991. Performance of B-tree concurrency control algorithms. In *Proceedings of the 1991 ACM SIGMOD international conference on Management of data*. 416–425.

[64] Yihan Sun, Guy E Blelloch, Wan Shen Lim, and Andrew Pavlo. 2019. On supporting efficient snapshot isolation for hybrid workloads with multi-versioned indexes. *Proceedings of the VLDB Endowment (PVLDB)* 13, 2 (2019), 211–225.

[65] Yihan Sun, Daniel Ferizovic, and Guy E Blelloch. 2018. PAM: Parallel Augmented Maps. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*.

[66] The Rust Project Developers. 2024. `std::collections::BTreeMap` — The Rust Standard Library. https://doc.rust-lang.org/std/collections/btree_map/struct.BTreeMap.html.

[67] Ruiqin Tian, Junqiao Qiu, Zhijia Zhao, Xu Liu, and Bin Ren. 2019. Transforming query sequences for high-throughput b+ tree processing on many-core processors. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 96–108.

[68] Thomas Tseng, Laxman Dhulipala, and Guy Blelloch. 2019. Batch-parallel euler tour trees. In *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 92–106.

[69] Letong Wang, Xiangyun Ding, Yan Gu, and Yihan Sun. 2023. Fast and Space-Efficient Parallel Algorithms for Influence Maximization. *Proceedings of the VLDB Endowment (PVLDB)* 17, 3 (2023), 400–413.

[70] Yiqiu Wang, Yan Gu, and Julian Shun. 2020. Theoretically-Efficient and Practical Parallel DBSCAN. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2555–2571.

[71] Yiqiu Wang, Shangdi Yu, Laxman Dhulipala, Yan Gu, and Julian Shun. 2022. ParGeo: a library for parallel computational geometry. In *European Symposium on Algorithms (ESA)*.

[72] 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 *Proceedings of the 2018 International Conference on Management of Data*. 473–488.

[73] Brian Wheatman, Randal Burns, Aydın Buluç, and Helen Xu. 2023. Optimizing Search Layouts in Packed Memory Arrays. In *2023 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics, 148–161.

[74] Brian Wheatman, Randal Burns, and Helen Xu. 2024. Batch-Parallel Compressed Sparse Row: A Locality-Optimized Dynamic-Graph Representation. In *2024 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE.

[75] Brian Wheatman, Xiaojun Dong, Zheqi Shen, Laxman Dhulipala, Jakub Łącki, Prashant Pandey, and Helen Xu. 2024. BYO: A Unified Framework for Benchmarking Large-Scale Graph Containers. In *Proceedings of the VLDB Endowment (PVLDB)*.

[76] Brian Wheatman, Quinten De Man, Xiangyun Ding, Zhongqi Wang, Randal Burns, Laxman Dhulipala, Xiaojun Dong, Yan Gu, Yihan Sun, and Letong Wang. 2026. Use Your Time(stamp) Wisely: The GraphTide Real-World Dynamic Graph Datasets. (2026). Manuscript under review.

[77] Brian Wheatman and Helen Xu. 2018. Packed compressed sparse row: A dynamic graph representation. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE.

[78] Brian Wheatman and Helen Xu. 2021. A parallel packed memory array to store dynamic graphs. In *Algorithm Engineering and Experiments (ALENEX)*. SIAM, 31–45.

[79] Brian Wheatman and Helen Xu. 2021. A parallel packed memory array to store dynamic graphs. In *2021 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 31–45.

[80] Evan Wrench, Ajay Singh, Younghun Roh, Panagiota Fatourou, Siddhartha Jayanti, Eric Ruppert, and Yuanhao Wei. 2026. Concurrent Balanced Augmented Trees. In *Proceedings of the 31st ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 136–149.

[81] 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. *Proceedings of the VLDB Endowment* 16, 11 (2023), 2976–2989.

[82] Zhaofeng Yan, Yuzhe Lin, Lu Peng, and Weihua Zhang. 2019. Harmonia: a high throughput B+ tree for GPUs. In *Proceedings of the 24th symposium on principles and practice of parallel programming*. 133–144.

[83] Weihua Zhang, Chuanlei Zhao, Lu Peng, Yuzhe Lin, Fengzhe Zhang, and Jinhu Jiang. 2022. High performance GPU concurrent B+ tree. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 443–444.

[84] Yiwei Zhao, Hongbo Kang, Yan Gu, Guy E. Blelloch, Laxman Dhulipala, Charles McGuffey, and Phillip B. Gibbons. 2025. Optimal Batch-Dynamic kd-trees for Processing-in-Memory with Applications. In *Proceedings of the 37th ACM Symposium on Parallelism in Algorithms and Architectures* (Portland, OR, USA) *(SPAA '25)*. Association for Computing Machinery, New York, NY, USA, 350–366. doi:10.1145/3694906.3743318

[85] Yiwei Zhao, Hongbo Kang, Ziyang Men, Yan Gu, Guy E. Blelloch, Laxman Dhulipala, Charles McGuffey, and Phillip B. Gibbons. 2026. PIM-zd-tree: A Fast Space-Partitioning Index Leveraging Processing-In-Memory. In *Proceedings of the 31st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Sydney, Australia) *(PPoPP '26)*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3774934.3786411

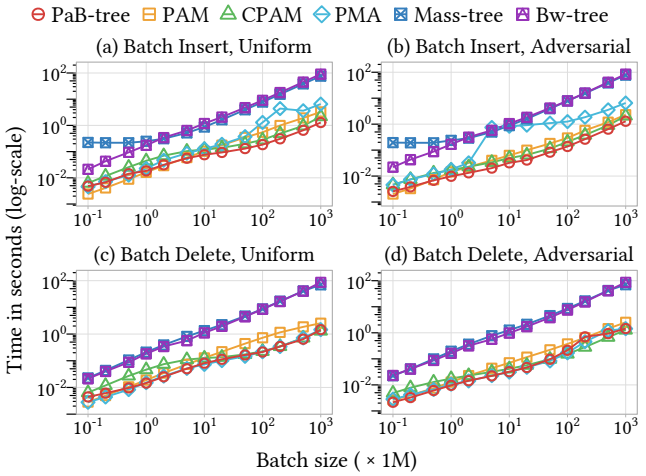


Figure 10. Running time (in seconds) of single batch update on synthetic datasets. Lower is better. The tree contains 10^9 entries, and the batch size ranges from 10^5 to 10^9. The batch insertion is to insert another batch from same distribution, and the batch deletion is to delete a batch of entries from the tree. Plots are in log-log scale.

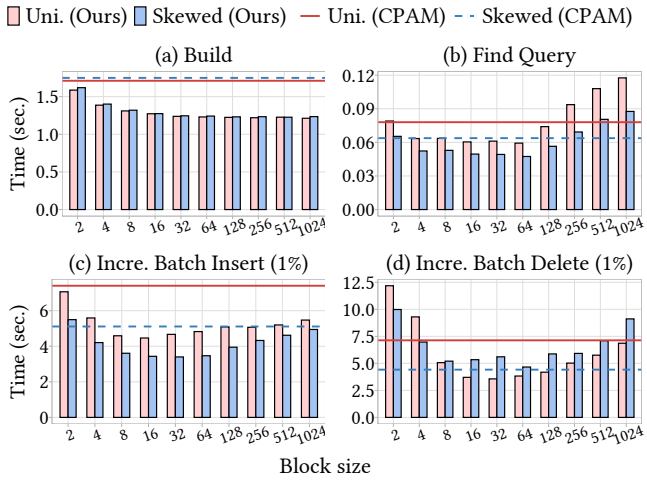


Figure 11. Running time (in seconds) of tree construction, incremental batch updates and Find queries w.r.t different block sizes in PaB-tree on synthetic datasets. Lower is better. The time for CPAM on the same datasets is marked in lines. We measure both the Uni and Skew distribution.

A Single Batch Updates Evaluation

We measure the time for PaB-tree and the baselines to apply a single batch of insertions or deletions on the Uni and Adv datasets. All baselines behave on Skew much as they do on Uni, so we omit it. We construct a tree with 10^9 entries, apply one batch to it, and report the running time. The batch size ranges from 10^5 to 10^9; we exclude smaller batches because their low computation cost diminishes the practical benefits of parallelism. We summarize the results in Fig. 10.

In the incremental setting of Sec. 4.1, the running time is dominated by handling invalid subtrees, while a single batch update behaves more like the static case: for small batches, routing the entries to their leaves accounts for most of the cost; and for large batches, handling invalid subtrees poses less overhead than it does in the incremental setting, where the updates trigger invalid subtrees far more often. Overall, PaB-tree is the fastest or among the fastest under both distributions. On small batches, CPAM and PAM are slightly slower than PaB-tree because of their larger tree height. The gap narrows as the batch size grows, since applying the batch invalidates larger subtrees, where both PAM and CPAM can utilize tree rotations to fix them easily, while PaB-tree must reinsert every entry in the invalid subtree. The insertion curve of PMA shows several spikes, most likely because the entries exceed the capacity of the underlying array and force PMA to reallocate the memory and copy every entry. For batch deletion, PMA performs similarly to the others, since a freshly constructed tree leaves every leaf block nearly full and a single batch rarely removes enough entries to trigger an underflow.

B Evaluation of the Block Size

We compare the performance of PaB-tree under different block sizes on synthetic datasets. The block size offers a trade-off between tree height and the cost of searching within leaves. A smaller block size yields a taller tree, which incurs additional rounds of partitioning during construction and batch updates, but it reduces the search cost within each leaf and thereby accelerates queries, and vice versa. We measure the performance of PaB-tree on tree construction, incremental batch updates, and membership queries for block sizes B ranging from 2 to 1024. We evaluate both the Uni and Skew distributions, compare against CPAM with a block size of 64, and summarize the results in Fig. 11.

For tree construction, the time decreases as the block size grows and stabilizes beyond $B = 64$, which is about 1.31× faster than with $B = 2$ and 1.39× faster than the CPAM. For membership queries, a smaller block size produces a taller tree and thus more pointer chasing, increasing query latency; a larger block size, in turn, demands more work to search within each leaf, also raising query time and causing PaB-tree to fall behind CPAM once $B \geq 512$.

For batch insertion, the running time rebounds under both distributions as the block grows. The reason is twofold: a small block raises the tree height and therefore the number of rounds needed to reach the leaves, while a large block demands more work to restore the order of the entries within each leaf. However, the PaB-tree is still faster than the CPAM due to a better cache access pattern for large block sizes. Similar rebounds are observed for batch deletion as well. However, for batch deletion, the cost of reinsertion dominates the overall cost, especially for the Skew distribution. A larger block size can absorb more entries during deletion, while it also leads to more entries being involved if a reinsertion is triggered, which makes it much slower than the CPAM for

Bench.	Trees	Build (1B)				Incre. Batch Insert (10M)				Incre. Batch Delete (10M)				Find (10M)			
		CC	Inst	CRs	CMs	CC	Inst	CRs	CMs	CC	Inst	CRs	CMs	CC	Inst	CRs	CMs
Uni	PaB-tree	689102	427666	2757	1896	2606644	1916753	7499	6732	2527648	1131125	9065	8541	41462	12126	229	135
	PAM	1309474	641143	3676	2450	5010064	1769777	14481	13654	7713895	1682153	28172	26765	54587	9944	257	154
	CPAM	1141550	460966	2913	2172	3970271	2557469	14894	13246	3833292	2306154	13942	12550	51803	12059	322	200
	PMA	1659997	782120	3931	3137	8056747	4005441	19041	16428	5071312	2026665	14687	13641	22512	12389	162	63
Skew	PaB-tree	723544	434176	2759	1884	1898140	1342657	5414	4731	2962518	1425902	8599	7727	33791	11387	196	106
	PAM	1145419	638249	3645	2366	3692366	1497891	10573	9827	5449122	1475377	19262	18105	45955	10145	219	118
	CPAM	918078	459656	2854	2123	2847642	1817654	9932	8844	2379974	1453233	8032	7177	41816	13034	269	149
	PMA	1623022	810203	3888	3122	12012571	5086143	28184	24814	6178408	2680818	15210	13757	21433	13849	134	50
Adv	PaB-tree	755753	427042	2700	1847	1492662	1028816	2869	2303	1501640	880613	2992	2297	24772	11319	118	71
	PAM	1069354	638485	3599	2340	1836388	944630	5682	4908	1684620	920743	4583	3908	31828	10358	128	76
	CPAM	932701	470555	2852	2105	1486898	919756	4024	3407	769352	451312	1668	1270	30904	13207	163	101
	PMA	1757184	767818	3711	3094	18835645	7638258	43640	39373	10353164	4490375	22971	20477	16384	12977	86	34

Table 1. Hardware profiling of the baselines. We report the number of CPU cycles (CC), instructions (Inst), cache references (CRs) and cache misses (CMs). The detailed setting is shown in Sec. 4.1. The FIND query contains 10M queries on a tree with 1B entries. All units are in millions. The best result in each column is underlined.

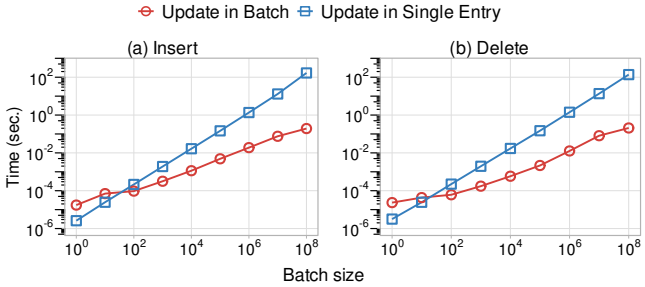


Figure 12. Running time (in seconds) of batch updates vs. sequential point updates. Lower is better. We measure the time to apply insertion or deletion batches of different sizes to a tree of 10^9 entries. Entries in both the tree and the batches are uniformly distributed. Because of the inherent overhead of batching and parallelism, batch updates may be slower than point updates on small batches. However, at a batch size of about 100, the benefits of batching and parallelism start to outweigh this overhead. Accordingly, PaB-tree can fall back to sequential point updates for batches below this size.

$B > 64$.

Overall, $B = 64$ balances update and query performance best, so we adopt it as the default block size in all other experiments.

C Hardware Profiling Details

Table 1 reports the raw counters behind Fig. 7 and extends the profiling from the Skew distribution to all three synthetic distributions. For tree construction, PaB-tree attains both the lowest instruction count and the fewest cache misses under every distribution, and the two counters barely change across different distributions, because the construction process is purely comparison-based and thus resistant to different distributions. On incremental batch insertion under Uni, PAM executes 8% fewer instructions than PaB-tree, yet incurs 2.03× the cache misses, which in turn makes it 1.92× slower. FIND repeats the pattern: PAM has the lowest instruction count of all indexes in all three distributions, but its large tree height requires fetching more cache lines and leaves it slower.

For batch deletion, CPAM is faster than PaB-tree on Skew and Adv, by up to 1.95×. On Skew, PaB-tree still executes the fewest instructions, and the gap comes entirely from I/O, whereas on Adv, CPAM wins on both counters, since its join-based algorithm can easily split out the single tree that is triggered by applying a batch of consecutive keys, while PaB-tree must reinsert every underflow subtree. Once this shortcut disappears under Uni, PaB-tree leads on all four counters again. For FIND queries, the PMA incurs about 2.1× fewer cache misses than the PaB-tree on every distribution. This gain is confined to queries on different distributions. Rebuilding overflowed blocks during incremental insertion costs PMA up to 12.6× the cycles of PaB-tree, with the gap widening as the keys in a batch become more local, as in the Adv distribution, which makes PMA the slowest index for updates. Overall, the hardware profiling confirms the importance of cache efficiency in achieving high-performance parallel B^+-trees, which is the main reason that PaB-tree outperforms the baselines in most of the experiments.

D Handling of Small Batches

As shown in Alg. 3, batch deletion repairs node underflows by setting aside dropped subtrees and reinserting them level by level. For very small deletion batches, the reinsertion overhead may dominate the total cost. Although Fig. 5 shows that reinsertions occur infrequently in practice, we discuss how to handle small batches here for completeness.

More generally, small batches may diminish the benefits of batch-parallel updates because each thread receives too little work. Existing work on parallel data structures such as CPAM has shown that below a certain batch size, applying a batch as a sequence of point updates is faster. In any case, the cost of such small batches is low. We test PaB-tree with

very small batches on a tree of 10^9 entries under the Uni distribution and show the results in Fig. 12. Batch updates start to outperform point updates at a batch size of about 100, where a batch takes less than 1 millisecond. Below this cutoff, PaB-tree can simply fall back to sequential point updates, which are even faster.