
# Efficient Concurrent Updates to Persistent Randomized Binary Search Trees

Guanhao Hou
ghhou@se.cuhk.edu.hk
The Chinese University of Hong Kong
Hong Kong SAR

Jinchao Huang
jchuang@se.cuhk.edu.hk
The Chinese University of Hong Kong
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

## ABSTRACT

In the era of big data, the demand for historical data analytics is growing across various applications. Simultaneously, range queries have been extensively explored within the domain of databases. Binary search trees are a classic type of in-memory index for facilitating range queries. Persistent binary search trees provide read-only snapshots of these trees, allowing range queries to be processed during updates while ensuring consistency. Additionally, multiple versions of snapshots support queries related to historical moments to meet the demands of numerous applications.

However, existing implementations do not support both high-speed updates and efficient, accurate historical queries on multi-core platforms. Motivated by this gap, we propose a novel concurrent update strategy to balance update and query performance. For a binary search tree containing $n$ elements, our approach completes $m$ updates in $O(\log n + m)$ time using $O(\log n)$ threads. We further implement a hybrid concurrent strategy to improve the scalability and practical performance of our solution.

The experimental results demonstrate that our proposal strikes a good balance between update and query performance. In particular, our proposal outperforms existing solutions under workloads with different data distributions and varying update-query ratios.

**PVLDB Reference Format:**
Guanhao Hou, Jinchao Huang, Fangyuan Zhang, and Sibo Wang. Efficient Concurrent Updates to Persistent Randomized Binary Search Trees. PVLDB, 18(5): 1481 - 1494, 2025.
doi:10.14778/3718057.3718074

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/CUHK-DBGroup/Contreap.

## 1 INTRODUCTION

In the era of big data, characterized by exponential growth in data volume and information richness, the effective processing of historical data meets a wide array of real-world needs, facilitating sophisticated analyses in practical scenarios. Historical data have a wide range of applications in science [60], web applications [59, 70], complex business analytics [23, 42], and more. This broad applicability highlights the need for storage solutions capable of managing such data. In reality, numerous applications, such as Microsoft Immortal DB [47], Ganymed [55], Skippy [62], and other multi-version systems, have been developed to meet various demands.

In the meantime, range aggregation is a classic topic in the database area. Various range queries have been explored within the realm of relational database [22, 34, 37, 41], spatial databases [33, 53, 54, 73], large scale key-value stores [6, 67], and more. In the real world, most database systems utilize column stores to support efficient data statistics, while transaction processing is typically based on row stores [3]. However, in many applications, it is not necessary to maintain a HTAP [32] database for complex analytical tasks that involve numerous predicates or costly operations like joins. Instead, user may simply need a basic statistical result within a range. In these situations, selecting an appropriate in-memory index to manage aggregate information can be advantageous.

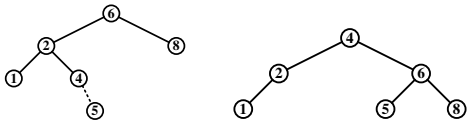
Binary search trees stand out as a traditional and effective choice among the various types of in-memory indices used for set operations and range queries [10, 11, 69]. They provide efficient search and retrieval capabilities, making them well-suited for handling large datasets. Compared to other tree structures such as B-Trees, it incurs lower overhead for range aggregation due to its small node fanout. Additionally, the small node size of the binary search tree results in lower overhead when implementing the read-copy update strategy [50]. This enables both efficient updates and concurrent range queries. Furthermore, by retaining copies of every updates, persistent[1] binary search tree [29] can serve as an index to support efficient historical range statistics for key-value stores.

However, existing implementation of persistent binary search trees struggle to simultaneously accommodate updates performance and efficient accurate historical queries. Algorithms on modern computer platforms enhance performance by adopting multithreading, while existing concurrent persistent binary search trees [31] cannot ensure that the order of committed versions matches the order of updates submissions, making it difficult to associate a version with a timestamp to support accurate historical queries. Another approach is based on batch processing [13], where a sufficient number of updates are processed as a batch to leverage the power of multi-core platforms. As a trade-off, batch-based multithreaded updates generate coarse-grained versions, where each version contains numerous updates. This results in additional costs to pinpoint the specific timestamp when handling historical queries.

*Sibo Wang is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 18, No. 5 ISSN 2150-8097.
doi:10.14778/3718057.3718074

[1] A persistent data structure indicates that it is immutable, meaning any changes will create an updated copy instead of modifying it in place. This differs from a data structure on persistent media, which is designed for non-volatile storage.



(a) Before Rotation (b) After Rotation

**Figure 1: BST Rebalancing by Rotation.**

In this paper, we introduce a novel concurrent update strategy for persistent binary search trees. Our solution, namely *Contreap* (Concurrent treap), leverages randomized binary search trees, aka treaps [61], to implement efficient concurrent updates. For a set consisting of $n$ elements, Contreap completes $m$ updates in $O(\log n+m)$ time using a pipeline with $O(\log n)$ stages. Based on the pipeline, Contreap provides one immutable version for each update and guarantees that the versions are generated in the same order as the updates. These features enable Contreap to support efficient and accurate historical queries. We further propose some optimizations to improve the scalability and practical performance of Contreap.

We experimentally evaluate the performance of our solution. The results indicate that Contreap outperforms batch-based updates for moderate batch sizes, such as $10^3$. Even when batch sizes increase to $10^7$, Contreap remains competitive with batch-based methods. By maintaining immutable versions for each update, Contreap significantly improves query efficiency compared to batch-based methods. The experimental results further demonstrate that Contreap outperforms the batching strategy under workloads that incorporate interleaved reads and writes, providing evidence that it achieves an excellent balance between update and query performance.

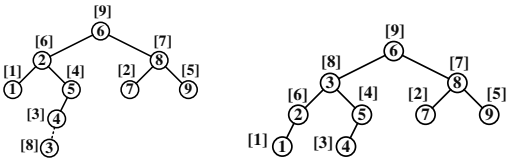
## 2 PRELIMINARIES

### 2.1 BSTs and Range Queries

A *binary search tree* (BST) is either an *empty tree*, denoted as nil, or a tuple $T = \langle T_L, E, T_R\rangle$, where $E = \langle k, v\rangle$ is a key-value pair stored at the root of $T$, while $T_L$ denotes the *left subtree* and $T_R$ denotes the *right subtree* of $T$. We also use the notation $data(T)$ to denote the element of $T$, i.e., $data(T) = E$, while the notations $lc(T) = T_L$ and $rc(T) = T_R$ represent the subtrees of $T$. For an element $E$, we use $key(E) = k$ and $val(E) = v$ to denote the key and the value of $E$, respectively. For simplicity, we use $key(T)$ and $val(T)$ as synonyms for $key(data(T))$ and $val(data(T))$, respectively. We say a non-empty node $T$ is a *leaf node* if both $lc(T)$ and $rc(T)$ are empty. Otherwise, $T$ is called an *internal node*. For any internal node $T$, all keys in its left subtree must be less than $key(T)$, and all keys in its right subtree must be greater than $key(T)$. In other words, the in-order traversal of a BST is a sorted sequence, which is referred to as its *sorted property*.

The *height* of a binary tree, denoted as $h(T)$, is defined as 0 for nil and $1+max(h(lc(T)), h(rc(T)))$ otherwise. The *size* of a binary tree, denoted as $s(T)$, is defined as the number of nodes in the tree, which is 0 for nil and $s(lc(T))+s(rc(T))+1$ otherwise. Besides, the *depth* of a node $T$ with respect to a tree $\hat{T}$, where $\hat{T}$ is an ancestor of $T$, is the distance between $T$ and $\hat{T}$. This is denoted as $d_{\hat{T}}(T)$, with $d_{\hat{T}}(\hat{T}) = 0$. For simplicity, we also use $d(T)$ to denote the distance from $T$ to the actual root, which is the node without any ancestors.

A BST takes $O(h(T))$ time to support inserting, deleting, or searching for an element. Additionally, a range scan, which returns all elements with keys within the given range, can be executed in $O(h(T)+z)$ time, where $z$ is the number of returned elements [9].



(a) Before Rotation (b) After Rotation

**Figure 2: Treap Rebalancing by Rotation.**

Sometimes, it is not necessary to retrieve all elements within a range. Instead, we may want to obtain some statistical properties within that range. An *augmented binary search tree* [67] speeds up statistics within any given range. In augmented BSTs, each node $T$ maintains some extra statistical information about its subtree, referred to as *augment* and denoted as $aug(T)$. The augment for a leaf node is statistical information about the element $E$, denoted as $aug(E)$. For an internal node $T$, $aug(T)$ is obtained by sequentially aggregating $aug(lc(T))$, $aug(data(T))$, and $aug(rc(T))$. After a single element in the augmented BST is updated, it takes $O(h(T))$ time to correct the affected augments. With this augment scheme, range statistics can be processed by aggregating augments of the subtrees that cover the query range in $O(h(T))$ time.

### 2.2 Balanced BSTs

If updates are generated randomly, a vanilla BST $T$ is expected to have a height $h(T) = O(\log s(T))$. However, in the worst case, a BST can degenerate to a linked list, where its height $h(T)$ is identical to the size $s(T)$, leading to a terrible efficiency for updates and search. Hence, a series of *balanced binary search trees*, such as *AVL trees* [5], *BBα trees* [52], and *Red-Black trees* [36], maintain the balance that $h(T) = O(\log s(T))$. These balanced BSTs have different balancing invariants to guarantee update and query complexity. If the balancing condition is violated after an update, it performs rotations, a class of operations that change the positions of nodes while holding the sorted property, to rebalance itself. Consider the following example for better illustration:

Example 1. *Assuming that we have a set of elements* $\{1, 2, 4, 6, 8\}$*, insert elements* 5*,* 7*,* 9*, and* 3 *into this set in order.*

Fig. 1(a) shows an instance corresponding to the example, where the balance is violated after inserting node 5 into the AVL tree. Fig. 1(b) shows the rebalanced AVL tree achieved by performing a left rotation followed by a right rotation on node 4. Other balanced trees can also use rotations to ensure that the rebalanced BST adheres to their balancing conditions.

Apart from the deterministic rebalancing schemes, *treaps* [61], also known as *randomized binary search trees*, assign a uniformly random positive *priority*, denoted as $pry(T)$, to each node $T$. A treap is thus organized by key-priority pairs, where the keys follow the sorted property as BSTs and the priorities adhere to the **heap property**. Formally, it is ensured that for any node $T$, the conditions $pry(T)>pry(lc(T))$ and $pry(T)>pry(rc(T))$ hold, with the priority of nil defined as 0. With uniformly random priorities, for any set, a treap holds that $h(T) = O(\log s(T))$ in expectation and with high probability. Fig. 2(b) shows an example of treap, where the number in the brackets indicates the priority.

Updates to a treap may spoil the heap property, and a treap can restore this through rotations. Fig. 2(a) shows a treap after inserting node 3 with priority 8. Then, the heap property is violated. To fix this, it then repeatedly do a left/right rotate with its parent until the heap property is no longer violated. For the example in Fig. 2(a), by performing two right rotations followed by one left rotation on node 3, we can restore the heap property, as shown in Fig. 2(b).

Treaps are a type of *uniquely represented data structure* [66]. Assuming that the priorities of different elements are distinct and that the priority of an element is fixed, the pattern of a treap is unique for a particular set of elements, regardless of the order of updates [14]. In practice, we can adopt a perfect hash function with the hash value of the key serving as the priority. We use the notation $pry(k)$ to represent the priority corresponding to the key $k$, and the notation $pry(E)$ as a synonym for $pry(key(E))$.

## 2.3 Consistency in Multithreaded Environments

In real-world applications, data is continuously updated while a large number of queries are processes simultaneously. Modern computer platforms leverage multithreading to enhance performance, which poses challenges for maintaining the evolving data.

One issue is supporting concurrent queries during updates while ensuring consistency. Consider the case that we need to handle range scans during processing the updates as Example 1 shows. In a single-threaded environment, one possible case is that a scan for the range $[3, 8]$ is performed after the insertion of 5 finishes but before the insertion of 7 begins. In this case, the scan operation will return $\{4, 5, 6, 8\}$. In a multi-threaded environment, suppose we use one thread, denoted as **P**, to handle the updates and another thread, denoted as **S**, to perform a range scan from 3 to 8. Without loss of generality, assume that **S** iterates to list elements within the range from front to back. A possible case is that **P** inserts 5 when **S** reaches 6, followed immediately by **P** inserting 7, after which **S** retrieves 7 and 8. In this case, **S** fetches 7 but misses 5. However, the result $\{4, 6, 7, 8\}$ has never existed at any moment, indicating an illegal state. Therefore, in a single-writer environment, we desire the consistency that if the result of a query incorporates the impact of an update, it also reflects the effects of all preceding updates, which is crucial for avoiding illegal results.

If updates are processed by multiple threads concurrently, differences in update complexity, thread performance, or other influencing factors may cause updates submitted earlier to be completed later. Assume that we have two threads, $\mathbf{P}_1$ and $\mathbf{P}_2$, to handle the updates. One possible scenario is that $\mathbf{P}_1$ completes the insertion of 5 later than $\mathbf{P}_2$ completes the insertion of 7, although 5 appears before 7. In this case, the order of updates is disrupted, which is crucial in many applications. To avoid such unexpected results, we should support *order preservation* in a multi-writer environment, meaning that updates submitted first must be completed first.

## 2.4 Persistent BSTs and Join-Based Methods

*Persistent data structures* are a class of data structures that do not visibly update in place. Instead, they perform a copy-on-write (CoW) process that yields a new *version* to represent the updated structure. Based on this feature, persistent data structures naturally implement the read-copy update (RCU) mechanism [50]. This ensures the consistency when querying during updates by separating reads and writes. In detail, queries are always performed on a read-only snapshot, ensuring that updates do not affect existing queries. When a new stable version is ready, subsequent queries will operate on this new version, while ongoing queries remain unaffected.

**Algorithm 1:** Join-Based Parallel BST

```
1  Function Join(T_L, E, T_R)
       // return a new balanced tree containing E
          and all elements in T_L and T_R
2  Function Expose(T)
3      return ⟨lc(T), data(T), rc(T)⟩
4  Function Split(T, k)
5      if T = nil then return ⟨nil, nil, nil⟩
6      else if k == key(T) then return ⟨T_L, E, T_R⟩
7      else
8          ⟨T_L, E, T_R⟩ ← Expose(T)
9          if k < key(E) then
10             ⟨T_LL, E_L, T_LR⟩ ← Split(T_L, k)
11             return ⟨T_LL, E_L, Join(T_LR, E_L, T_R)⟩
12         else
13             ⟨T_RL, E_R, T_RR⟩ ← Split(T_R, k)
14             return ⟨Join(T_L, E, T_RL), E_R, T_RR⟩
15 Function Union(T_A, T_B)
16     if T_A = nil then return T_B
17     else if T_B = nil then return T_A
18     else
19         ⟨T_BL, E_B, T_BR⟩ ← Expose(T_B)
20         ⟨T_AL, E_A, T_AR⟩ ← Split(T_A, key(E_B))
21         || T_L ← Union(T_AL, T_BL)
           || T_R ← Union(T_AR, T_BR)
22         if E_A ≠ nil then E_B ← Update(E_A, E_B)
23         return Join(T_L, E_B, T_R)
```

Additionally, since a persistent data structure preserves previous versions rather than modifying in place, it supports accessing data at a specific version. This gives persistent data structures the potential to support temporal data management. If updates are order-preserved, i.e., first submitted first committed, persistent data structures can generate a version for each item in data stream, thereby supporting queries for the state at any point in history.

*Persistent binary search trees* (PBSTs) is usually built with a *join-based* process [13], which leverages the power of multithreading by applying *nested fork-join parallelism* [18]. The efficiency of algorithms based on fork-join parallelism can be analyzed using the *work-span model*, which states that a procedure with a work of $w$ (the total number of operations executed by the procedure) and a span of $d$ (the longest chain of sequential dependencies in the computation), can be processed in $O(w/p + d)$ time using $p$ processors.

Alg. 1 shows the pseudo-code of the join-based method to union two balanced BSTs. The *expose* function takes a balanced tree $T$ as a parameter and separates it into $lc(T)$, $data(T)$, and $rc(T)$. The *join* function takes three parameters: two balanced trees $T_L$ and $T_R$, and an element $E$, where all keys in $T_L$ are less than $key(E)$, and all keys in $T_R$ are greater than $key(E)$; returns a balanced tree that includes $E$ and all elements from the two subtrees. The implementation of

join varies depending on the balancing strategy [13], and it does not necessarily have its root store $E$ after rebalancing. Based on join and expose, the *split* function divides a balanced tree $T$ into two subtrees with keys smaller than and larger than the given key $k$, and also returns an element exactly matching the key if it exists. Derived from expose, join, and split, the union operation can be performed in parallel using a divide-and-conquer strategy. First, expose one of the two trees (Line 19) and then use the exposed key $k$ to split the other (Line 20). At this point, the problem is divided into two disjoint parts: merging the two trees that contain elements less than $k$, and merging the two trees that contain elements greater than $k$. These processes can be executed in parallel (Line 21). Finally, return the join of the exposed element and the merged subtrees as the union of the inputs (Lines 22-23).

Using the parallel union operation, we can handle the parallel insertion to a PBST with *batch processing*. To insert $m$ elements into a tree containing $n \geq m$ elements using join-based method, it first constructs a balanced tree consisting of new elements in $O(m \log m)$ work and $O(\log m)$ span [13, 15, 56]. Then, it unions the old tree with the newly generated tree in $O(m \log (n/m))$ work and $O(\log n \log m)$ span [4, 13, 16]. Similarly, the batch deletion can be handled using set difference with the same theoretical complexity. Finally, we have that the insertion and deletion can finish with a work of $O(m \log n)$ and a span of $O(\log n \log m)$. Besides, the space overhead for storing this new immutable version is $O(m \log (n/m))$ since that number of nodes has been modified during update.

Compared to updating each element serially with a single thread, parallel processing with a batching strategy is efficient in both time and space. As a trade off, batch-based method generates an immutable version for each batch containing $m$ updates rather than for each update. As a result, if we want to handle a historical query for a specific moment, we need to perform additional computations in the batched version to retrieve the data corresponding to that time point. Therefore, the performance of historical query processing depends on the batch size. If the batch size is large, the version of the PBST will be *coarse-grained*, making it challenging to handle historical queries with both high efficiency and accuracy. Conversely, maintaining a *fine-grained* version for better query performance requires a small batch size, which limits the advantages of the batching strategy, as it necessitates packing a significant number of elements into each batch for efficient updates.

## 2.5 Concurrent BSTs

There also exist some methods to update a BST concurrently. The *Relaxed balance AVL trees* [19] implement concurrent updates using locks, with a relaxed balancing condition. A lock-free implementation [51] of internal BST is proposed using the *compare-and-swap* (CAS) operation, while it has not guarantee to be balanced. Besides, NB-BSTs [30] implement lock-free concurrent updates for *leaf-oriented* BSTs without balance guarantee. These solutions perform updates in-place and do not provide consistency between range queries and updates without blocking.

Subsequently, *PNB-BST* [31] implements a persistent variety of NB-BST that supports *unit-grained* updates, i.e., it maintains a persistent version for each update. Nevertheless, PNB-BST is not order-preserving for updates. In a PNB-BST, threads compete for updates via CAS operations, which means an earlier update may appear in a later version. Then, a version of PNB-BST cannot correspond to all updates up to a specific timestamp, making it ineffective for historical queries. Besides, PNB-BST is also proposed without balancing strategy, which greatly limits its applicability.

**Algorithm 2:** Insertion for Vanilla PBSTs

```
1  Function CopyInsert(T, E)
2      if T = nil then return NewNode(E)
3      T' ← Copy(T)
4      if key(E) = key(T) then
5          val(T') ← Update(val(T), val(E))
6      else if key(E) < key(T) then
7          lc(T') ← CopyInsert(lc(T), E)
8      else
9          rc(T') ← CopyInsert(rc(T), E)
10     return T'
```

# 3 CONCURRENT UPDATES WITH PIPELINE

Motivated by the issues discussed in Sec. 2, in this section, we propose a practical, concurrent, balanced, persistent binary search tree. Literally, our solution, Contreap, is a PBST with a balancing strategy that supports efficient concurrent updates. In contrast to batch-based parallel solutions, our solution is unit-grained, meaning that it maintains an immutable version for each update. In contrast to existing concurrent PBSTs, our solution is order-preserving. These properties ensure efficiency and accuracy for historical queries.

We start by implementing the concurrent insertions for a PBST without balancing strategy in Sec. 3.1. Then, in Sec. 3.2, we discuss the challenges of keeping balance when performing concurrent updates in a PBST, along with our solution for concurrent insertions with balancing guarantee. Next, in Sec. 3.3, we provide our solution for concurrent deletions with balancing guarantee.

## 3.1 Update without Balancing Scheme

Consider serially inserting an element into a PBST without rebalancing schemes. To create a new version of the tree, we first fetch the previous version as a reference. Then, as shown in Alg. 2, we perform a top-down search starting from the root, copying all nodes along the search path until we reach an empty node or find that the given key exists. Fig. 3 shows an instance of persistent tree for Example 1, where $T^n$ represent the $n$-th version of the tree.

If we want to perform insert operations concurrently, a straightforward approach is to assign each thread to handle a single update. To preserve the order of updates, it is clear that the $(n{+}1)$-th update must refer to the $n$-th version. That is, the thread handling the $(n{+}1)$-th update can only start after the thread handling the $n$-th update sets the reference to the root node. Besides, any visited node along the search path, including its data and references to its left and right subtrees, must be fully initialized before it is copied.

In concurrent environments, the naive recursive process shown in Alg. 2 is not applicable. In the non-terminal case, i.e., if the current node is not empty and its key differs from that of the element to be inserted, one of the subtrees of this node will be set when backtracking. Therefore, the root is the last to be set in the thread

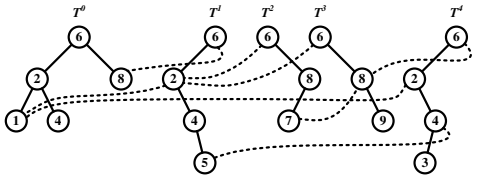



**Figure 3: Insertions to a Vanilla PBST.**

handling the $n$-th update. However, the thread handling the ($n$+1)-th update needs to visit the root of the $n$-th version first, resulting in the concurrent updates degrading to a serial process.

To avoid the degeneration of concurrency, we adopt a two-phase construction as shown in Alg. 3, a thread starts processing the $n$-th update by invoking **Insert**($\mathcal{T}$, $n$, $E$), where $\mathcal{T}$ is a collection storing all versions of the tree. It first creates an uninitialized node as a *placeholder* (Lines 9, 12, 17-18) and then initializes its contents during the search process (Lines 2, 5, 7, 10, 13). We use the primitive **Wait** (Lines 4, 15) which spins until the given boolean expression is satisfied, to ensure the referenced node is fully initialized.

For the instance as shown in Fig. 3, we can draw a *dependency graph* as Fig. 4(a) shows, where INS($n$, $T$, $k$) represents a processing step that visits node $T$ to insert $k$ for the $n$-th update. The notation $T_{k'}$ represents the node storing an element whose key is $k'$. In the dependency graph, a directed edge $\langle v, u\rangle$ represents that $v$ depends on $u$, meaning $v$ can only start after $u$ is completed. Moreover, an edge represented with a solid arrow, e.g., INS(1,$T_4$,5) to INS(1,$T_2$,5), indicates a *control dependency*, meaning $v$ is invoked by $u$, while a dashed arrow, e.g., INS(3,$T_8$,9) to INS(2,$T_8$,7), indicates a *data dependency*, meaning $v$ uses the data provided by $u$. Clearly, Alg. 3 is non-blocking, as there is no cycle in the dependency graph. In any case, the process handling the earliest unfinished update is always guaranteed to make progress. However, this method adopts spin-waiting with checking of shared variables, which usually becomes a bottleneck in practice if triggered frequently.

Revisiting the concurrent insertion process, copying and updating a node, w.l.o.g., $T_k^i$, has only one data dependence, which is a node $T_k^j$ where $j < i$ is the last version visiting the node containing key $k$. Therefore, if the initialization of $T_k^i$ always occurs after that of $T_k^j$ is complete, the updates will be order-preserved. Note that, for an insert operation in a BST without rebalancing, the depth $d(T)$ of an existing node $T$ remains invariant. Additionally, for any depth $l$, a search path contains only one node $T$ such that $d(T) = l$. Therefore, if we constrain that a node $T_k^i$ can only be initialized after the initialization of the node $T_{k'}^{i-1}$ is completed, where $d(T_k^i) = d(T_{k'}^{i-1})$, we still ensure that $T_k^j$ is completely initialized before constructing $T_k^i$. For example, we stipulate that INS(4,$T_2$,3) must be executed after INS(3, $T_8$, 9) completes, even though there is no dependency between them. Additionally, if the previous update is finished, it is considered that all nodes of the version have been initialized. This addresses the issue that different updates may involve varying depths. For example, inserting 9 into $T^2$ is finished at depth 2, while inserting 3 into $T^3$ needs to reach depth 3.

With the simplified constraint, we implement concurrent insertion based on *pipeline parallelism*. A pipeline is a series of data processing units, called *pipes*, and each pipe corresponds to a step

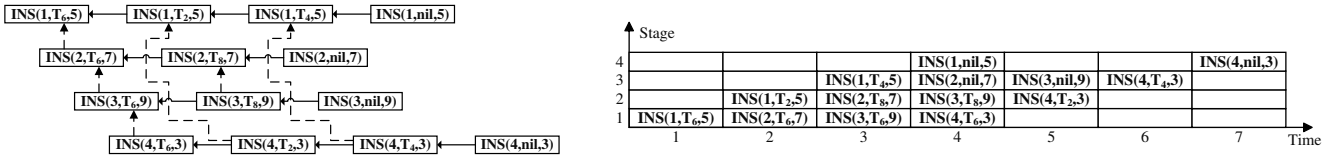
**Algorithm 3:** Concurrent Insertion for Vanilla PBSTs

```
1  Procedure CopyInsert(T', T, E)
2      if T = nil then InitLeaf(T', E)
3      else
4          Wait(IsInit(T))
5          InitCopy(T', T)
6          if key(E) = key(T) then
7              val(T') ← Update(val(T), val(E))
8          else if key(E) < key(T) then
9              lc(T') ← NewNode()
10             CopyInsert(lc(T'), lc(T), E)
11         else
12             rc(T') ← NewNode()
13             CopyInsert(rc(T'), rc(T), E)
14 Procedure Insert(T, n, E)
15     Wait(Ver(T) = n − 1)
16     T ← GetRoot(T, n − 1)
17     T' ← NewNode()
18     SetRoot(T, n, T')
19     CopyInsert(T', T, E)
```

in data processing, referred to as a *stage*. The output of one pipe is the input for the pipe in the next stage. The pipes are executed in a time-sliced fashion to parallelize computation handled by different pipes. In our case, each thread corresponds to a pipe, and the $i$-th stage corresponds to updates to any node $T$ where $d(T) = i$. Alg. 4 shows the concurrent insertion of persistent vanilla BSTs using a pipeline, the first pipe starts processing the $n$-th update by invoking **Insert**($\mathcal{T}$, $n$, $E$). After completing the first stage of the $n$-th update, it uses the primitive **Next** to output a reference to a procedure along with the current status of the $n$-th update as input to the second pipe (Line 17), and the second pipe will execute the given procedure with the current status. Similarly, when the $l$-th pipe completes one stage of the $n$-th update, unless the update is finished (Line 2), it uses the primitive **Next** to pass the arguments to the ($l$+1)-th pipe (Lines 9, 12). Note that in a pipeline, we do not need to synchronize by spin-waiting. Since each pipe operates serially, the $i$-th pipe starts processing node $T_k^n$ after completely processing the previous node $T_{k'}^{n-1}$ where $d(T_{k'}^{n-1}) = i$. Hence, concurrent insertions with a pipeline are naturally guaranteed to be order-preserved.

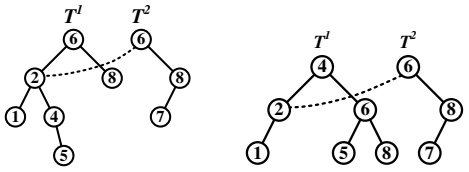
Assuming that each stage of an update uses the same amount of time, called a *time slice*, we have the *space-time diagram* using a pipeline as shown in Fig. 4(b). In the space-time diagram, the horizontal axis represents time and the vertical axis represents stages. Processes in the same column are executed concurrently by different threads at the same time. Processes in the same row are executed serially by the same thread, and an empty cell indicates that the thread is idle during this time slice.

Considering a time period, where the scale of updates does not exceed that of existing data. If the tree is not highly skewed, the height of the tree changes insignificantly over extended periods. Besides, most elements reside in the deeper parts of the tree. Therefore, if the number of threads is equal to the height of the tree, i.e., $h(T)$, for a sufficient number of updates, after time $h(T)$, almost all threads are continuously making progress. In addition, for each




(a) Dependency Graph (b) Space-Time Diagram

**Figure 4: The Dependency Graph and Space-Time Diagram w.r.t the Instance shown in Fig. 3**




(a) Before Rotation (b) After Rotation

**Figure 5: Error Caused by Rotation.**

update, we need to copy all the nodes along the search path, which totals $O(h(T))$ nodes. Formally, according to the properties of a pipeline, we have the following conclusion:

Corollary 3.1. *For a binary search tree $T$ that is not highly skewed, a pipeline with $h(T)$ stages handles $m = O(s(T))$ insertions into $T$ and constructs $m$ immutable versions using $O(h(T) + m)$ time and $O(n + m \cdot h(T))$ space.*

Note that as the concurrent constraint is tightened, the concurrency using a pipeline is theoretically not as high as having each thread handle one update. For example, INS(1,$T_2$,5) and INS(2,$T_8$,7) are executed serially in a pipeline, even though there is no data dependency between them. We will discuss this trade-off in Sec. 4. In the remainder of this section, we focus on introducing a balancing strategy to the concurrent persistent BST based on a pipeline.

## 3.2 Rotate-Free Concurrent Insertion

As we described in Sec. 2.2, balanced BSTs often perform bottom-up rotations after update for rebalancing. This rebalancing process may involve all nodes along the search path and their siblings. During rebalancing, some of these nodes may shift from their original positions. For a single update, the nodes on the search path are all created in the current version, so these nodes can be modified before the update is completed rather than being copied again. However, this can result in errors for concurrent PBSTs. Fig. 5 shows one of the error cases. Considering the instance depicted in Fig. 3, after we insert the element with key 5 into $T^0$, we obtain an unbalanced $T^1$ as shown in Fig. 5(a). We then perform a rebalancing operation to obtain the balanced $T^1$, as demonstrated in Fig. 1. Note that the constructions of $T^1$ and $T^2$ are executed concurrently, so it is possible for $T_6^2$ to set its left subtree to $T_2^1$ before $T^1$ performs rebalancing. As a result, we end up with an incorrect $T^2$ after $T^1$ is rebalanced, as Fig. 5(b) shows. This case states that any modifications to these parts can cause errors because PBSTs directly reference previous versions for parts not on the search path.

Meanwhile, rebalancing strategies for BSTs assume that the tree is balanced before being updated. If $T^n$ is unbalanced, there may not necessarily be a valid rotation plan to rebalance $T^{n+1}$ since the update is applied to an unbalanced BST. This means that $T^{n+1}$ may

**Algorithm 4:** Pipeline Insertion for Vanilla PBSTs

```
1  Procedure CopyInsert(T', T, E)
2      if T = nil then InitLeaf(T', E)
3      else
4          InitCopy(T', T)
5          if key(E) = key(T) then
6              val(T') ← Update(val(T), val(E))
7          else if key(E) < key(T) then
8              lc(T') ← NewNode()
9              Next(CopyInsert(lc(T'), lc(T), E))
10         else
11             rc(T') ← NewNode()
12             Next(CopyInsert(rc(T'), rc(T), E))
13 Procedure Insert(T, n, E)
14     T ← GetRoot(T, n − 1)
15     T' ← NewNode()
16     SetRoot(T, n, T')
17     Next(CopyInsert(T', T, E))
```

be ill-formed, even if we copy the rotated nodes along the search path to avoid modifying existing nodes. Similarly, errors can occur if a rotation on $T^n$ happens along the search path of $T^{n+1}$. In this case, $T^{n+1}$ will not return to a balanced state when performing rebalancing on $T^n$, as it has already copied the corresponding parts.

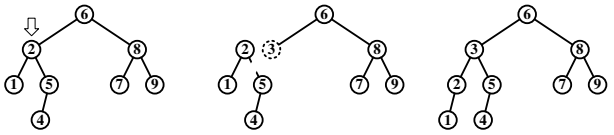
To address these issues, we propose a novel concurrent update strategy using treaps, more accurately, join-based treaps. As treaps are uniquely represented, the final pattern of the treap being updated is deterministic. Alg. 5 shows the concurrent insertion method that adopts spin to satisfy data dependencies, similar to Alg. 3.

When inserting an element $E$ into a treap, upon encountering the first node $T$ satisfying $pry(T)<pry(E)$ (Line 12), we can confirm that the element needs to be placed in the current position to hold the heap property. Therefore, we can directly initialize the placeholder $T'$ as a leaf containing the new element at this position (Line 13). Next, we invoke the split process to construct the left and right subtrees of the newly inserted node by splitting $T$ with $key(E)$ (Line 14). Note that the split process may involve any node in the subtree rooted at $T$, so we should use the primitive **Wait** to spin until the update of this subtree is complete (Line 13). Note that the case where the key to be inserted already exists will necessarily be processed during the search phase since we use the perfect hash of the key as the priority. Therefore, there is no need to address this situation during the split phase.

From Alg. 3, we can see that a join-based treap keeps the heap property during the search, split, and join processes [16] without invoking any rotations. Thus, unlike other balancing strategies,

**Algorithm 5:** Concurrent Insertion for Persistent Treaps

```
1  Function Split(T, k)
2      if T = nil then return ⟨nil, nil⟩
3      if k < key(T) then
4          T_R ← Copy(T)
5          ⟨T_L, lc(T_R)⟩ ← Split(lc(T), k)
6      else
7          T_L ← Copy(T)
8          ⟨rc(T_L), T_R⟩ ← Split(rc(T), k)
9      return ⟨T_L, T_R⟩
10 Procedure CopyInsert(T′, T, E)
11     if T = nil then InitLeaf(T′, E)
12     if pry(T) < pry(E) then
13         InitLeaf(T′, E)
14         Wait(IsComplete(T))
15         ⟨lc(T′), rc(T′)⟩ ← Split(T, key(E))
16     else
17         ··· // recursive search as in Alg. 3
```



(a) Find $pry(2)<pry(3)$ (b) Init $T_3$ and Split $T_2$ (c) Join to complete

**Figure 6: Join-Based Treap Insertion.**

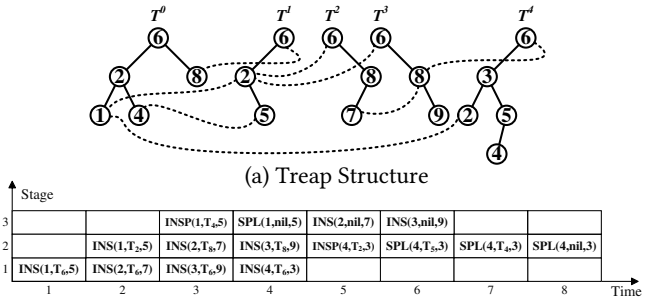
insertions in join-based treaps are *rotation-free*. Fig. 6 shows a running case of inserting a new element into a treap using a join-based method, where the priorities of each element are the same as those depicted in Fig. 2. When searching the treap from top to bottom, we first find the node $T_2$ such that $pry(2)<pry(3)$. Then, we construct the new node $T_3$ at the original position of $T_2$. Next, we split the subtree rooted at $T_2$. Finally, we complete the update by joining $T_3$ with the two split subtrees.

The search process of Alg. 5 is similar to that of Alg. 3. They differ only if the position to insert the new element is found, where Alg. 3 completes the update using the split-join scheme. In the same way, we can implement join-based treaps using a pipeline. As Alg. 6 shows, the search process is similar to that of Alg. 4 using a pipeline. When the position to insert the new element is found, the current pipe blocks to wait for the referenced subtree to be completed and then processes the split-join method to finish the update. This blocking strategy guarantees the correctness of the update process. On one hand, as we described above, the split process may involve any node in the subtree. On the other hand, each iteration of the split process only updates either the left or the right subtree of the current node. This does not guarantee that all nodes at the current depth are initialized. Hence, we must prevent subsequent updates from being processed based on the incomplete subtree by blocking the current pipe until the update is finished.

Fig. 7(a) presents the join-based implementation of Contreap related to Example 1, where the priorities of each element are the same as those depicted in Fig. 2. Assuming that each step of the split process uses one time slice like a search step, we can illustrate the space-time diagram of the update process, as shown

**Algorithm 6:** Pipeline Insertion in Contreap

```
1  Procedure CopyInsert(T′, T, E)
2      if T = nil then InitLeaf(T′, E)
3      if pry(T) < pry(E) then
4          InitLeaf(T′, E)
5          Wait(IsComplete(T))
6          ⟨lc(T′), rc(T′)⟩ ← Split(T, key(E))
7      else
8          ··· // recursive search as in Alg. 4
```




(a) Treap Structure

(b) Space-Time Diagram

**Figure 7: Insertions to Contreap**

in Fig. 7(b), where SPL($n$, $T$, $k$) represents a processing step that splits the subtree rooted at $T$ with respect to the key $k$ for the $n$-th update, and INSP($n$, $T$, $k$) is the combination of INS($n$, $T$, $k$) and SPL($n$, $T$, $k$). From the space-time diagram, it can be seen that the split operation causes subsequent updates to be delayed at the current pipe. We refer to such a delay caused by each split step as a *stall*.

In a pipeline, the stall will delay all the subsequent processing steps. Formally, The pipeline has the following property:

Property 3.2. *In a pipeline with $p$ pipes, $m$ operations will be completed in $O(p + md)$ time if the stalls caused by each operation is $d$ on average.*

From Alg. 5 and Alg. 6, we can observe that the time to split a subtree rooted at $T$ is $O(h(T))$, and the time to complete the update with join is $O(1)$. Furthermore, we have the following theorem:

Theorem 3.3. *In a treap, the height of the subtree rooted at the newly inserted node is $O(1)$ in expectation.*

Due to limited space, all omitted proofs can be found in our technical report [1]. Since treaps are balanced with high probably, and a treap with $n$ nodes has an expected height of $O(\log n)$, according to Corollary 3.1 and Property 3.2, we have the following result:

Corollary 3.4. *For a treap containing $n$ nodes, Alg. 5 concurrently processes $m = O(n)$ insertions and constructs $m$ immutable versions using $O(\log n)$ pipes with an expected time of $O(\log n + m)$ and a space consumption of $O(n + m \log n)$.*

Besides, for a sufficient number of updates, the running time is bounded with high probability. Formally, we have:

Theorem 3.5. *For a treap containing $n > 2e^{\alpha}$ nodes, where $\alpha > e^2$ is an arbitrary parameter, Alg. 5 concurrently processes $m$ insertions, where $m = \omega(\alpha^2 \log^2 n)$ and $m = O(n)$, using $O(\log n)$ pipes in $O(m)$ time with a probability of at least $1 - \frac{2m}{n^{\alpha-1}}$.*

**Algorithm 7:** Pipeline Deletion in Contreap

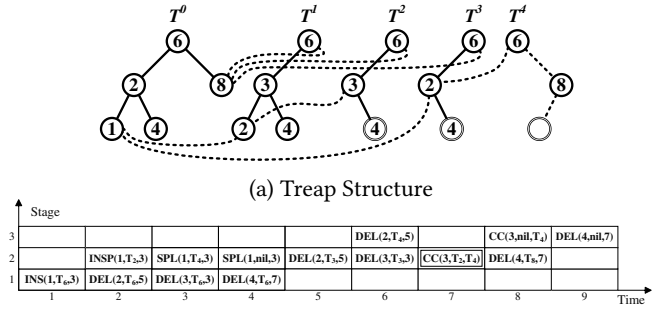
```
1  Procedure Concat(T′, T_L, T_R)
2      if T_L = nil then InitCopy(T′, T_R)
3      else if T_R = nil then InitCopy(T′, T_L)
4      else if pry(T_L) > pry(T_R) then
5          InitCopy(T′, T_L)
6          rc(T′) ← NewNode()
7          WaitNext()
8          Next(Concat(rc(T′), rc(T_L), T_R))
9      else
10         InitCopy(T′, T_R)
11         lc(T′) ← NewNode()
12         WaitNext()
13         Next(Concat(lc(T′), T_L, lc(T_R)))
14 Procedure CopyDelete(T′, T, k)
15     if T = nil then MarkAsNil(T′)
16     else if pry(T) < pry(k) then InitCopy(T′, T)
17     else if k = key(T) then
18         WaitNext()
19         Concat(T′, lc(T), rc(T))
20     else
21         ··· // recursive search as in Alg. 4
```

## 3.3 Rotate-Free Concurrent Deletion

Alg. 7 shows the rotate-free concurrent deletion algorithm using pipeline parallelism. Note that, the placeholder is created in advance, we cannot modify the parent node that references it, which prevents us from releasing the placeholder. Therefore, we mark the placeholder as equivalent to the empty tree when we find that the current tree is nil (Line 15). During the searching phase, once we encounter a node $T$ such that $pry(T) < pry(k)$, we can confirm that $k$ does not exist in $T$. In this case, we set the placeholder to be a copy of $T$ without making any modifications (Line 16).

When we find the node $T$ where $key(T) = k$, we connect its left and right subtrees using the *concat* method to create a new version with $T$ removed (Lines 17-19). Here, we use a new primitive **WaitNext** to deal with data dependency (Line 18). When processing the $n$-th update at the $l$-th pipe, this primitive produces stalls in the current pipe until the next pipe finishes processing the $(l{+}1)$-th step of the $(n{-}1)$-th update. The behavior of **WaitNext** guarantee correctness in concurrent deletion. On one hand, it confirms that the left and right children of the deleted node are initialized before being used, as they are managed by the next pipe. On the other hand, it hangs the current pipe and prevents subsequent updates from referencing the placeholder that has not yet been initialized.

The concat method takes three arguments: the placeholder $T'$ and two treaps $T_L$ and $T_R$, where the keys in $T_L$ are all smaller than the keys in $T_R$. If one of $T_L$ or $T_R$ is empty, we set the placeholder as a copy of the other one (Lines 2-3). We stipulate that copying nil will mark the placeholder as nil to handle the case where both $T_L$ and $T_R$ are empty. If both $T_L$ and $T_R$ are not empty, we select the one with the higher priority between them as the root to initialize the placeholder (Line 4). If the root of $T_L$ becomes the new root, we connect the right subtree of $T_L$ with $T_R$ as the new right subtree since all keys in $T_R$ are larger (Lines 5-8). Symmetrically, if the root




(a) Treap Structure

(b) Space-Time Diagram

**Figure 8: Deletions to Contreap**

of $T_R$ becomes the new root, we connect $T_L$ with the left subtree of $T_R$ as the new left subtree (Lines 10-13). Note that, since the children of $T_L$ and $T_R$ may be managed by the next pipe, we should use **WaitNext** to produce stalls until they are initialized.

To demonstrate the deletion process, Fig. 8(a) presents the result related to the following example:

Example 2. *Assuming that we have a set of elements* $\{1, 2, 4, 6, 8\}$, *insert element* 3, *then delete elements* 5, 3, *and* 7. *The priorities of keys are the same as those in Fig. 2.*

Note the nodes represented by concentric circles: $T_4^2$ is the replica of $T_4^1$ and $T_4^3$ is the replica of $T_4^2$. They are all the same due to initialization by simply copying the referenced nodes. These happend because the placeholders were created in advance, mapped to the cases in Line 16 and Lines 2-3, respectively. The blank node in $T^4$ is the placeholder marked as nil when reaching an empty subtree during the search process, corresponding to the case in Line 16.

Then, we can draw the space-time diagram as shown in Fig. 8(b), where CC($n$, $T_L$, $T_R$) represents a processing step that concats $T_L$ and $T_R$ to finish the $n$-th update. When we attempt to delete the element with key 5 from $T^1$, we find that $pry(T_4^1) < pry(5)$ in the seventh time slice at the third pipe and exit by determining that the element with key 5 does not exist. When deleting the element with key 3 from $T^2$, we cannot immediately execute CC(3, $T_2$, $T_4$) upon discovering $T_3^2$ in the sixth time slice at the second pipe. Instead, we must wait until the third pipe completes DEL(2, $T_4$, 5) and then perform concat in the seventh time slice. When we try to delete the element with key 7 from $T^3$ but find the empty tree in the ninth time slice at the third pipe, we finish the deletion by marking the placeholder as nil. Generally, unlike the split process for insertion, which must wait until the entire subtree is completely updated, a deletion step at any pipe requires only the state processed by the next pipe. As a result, each deletion operation incurs at most one stall. According to Corollary 3.1 and Property 3.2, we have:

Theorem 3.6. *For a treap containing n nodes, the join-based method concurrently processes* $m = O(n)$ *deletions and constructs m immutable versions, using* $O(\log n)$ *pipes in* $O(\log n + m)$ *time, with a space consumption of* $O(n + m \log n)$.

# 4 HYBRID SCHEDULING

**Observations.** In Sec. 3, we provide an implementation of persistent treaps, Contreap, that supports concurrent updates using a pipeline. However, as mentioned in Sec. 3.1, a pipeline tightens the

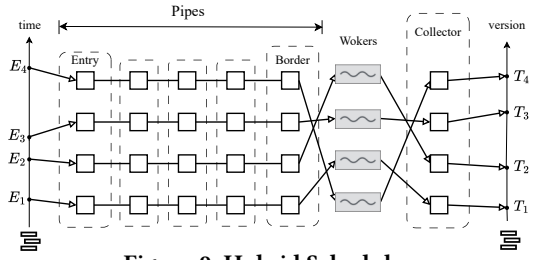



**Figure 9: Hybrid Scheduler**

constraints on concurrent execution, thereby reducing the potential for concurrency. This constraint is actually required only when the search paths of the elements to be updated are always the same.

Additionally, the number of stages in a pipeline is not always suitable for the updates. To leverage pipeline parallelism, we divide a treap $T$ into $h(T)$ layers. In each step, we allocate and initialize a specific node on one layer. Ideally, the number of pipes should match $h(T)$, allowing each pipeline stage to correspond exactly to one processing step. However, the height of the treap and the depth of the modified elements are variable, while the number of threads is limited by the hardware, making it less flexible in practice. The mismatch between the number of pipeline stages and the number of stages required for updates significantly affects the practical performance of pipeline parallelism. If the number of stages is insufficient, the last pipe must continuously processes the remaining work until the update is finished. Hence, the last pipe becomes the bottleneck, as it typically handles much heavier tasks than other pipes. In contrast, if a pipeline has too many stages, the update will be completed at an intermediate stage. However, the processing flow cannot stop prematurely in a pipeline. Hence, the surplus stages do not perform any effective operations but increase the overhead of updates and amplify the stalls caused by splitting.

Compared to pipelines, the blocking-based concurrent solution, as shown in Alg. 5, offers good scalability. Since each thread handles an entire update, it launches as many threads as possible and assign updates to them. Also, this solution has strong adaptability, as a new update can be assigned to a thread as soon as the thread finishes its current task. Yet, the high overhead of synchronization between threads still compromises the performance of blocking-based solutions in practice.

In summary, concurrent updates based on a pipeline benefit from having the number of stages closely aligned with the depth of the tree. Additionally, if the search paths of the updates rarely intersect, resulting in minimal synchronization overhead, then a blocking-based concurrent solution—where each thread handles an entire update—will perform effectively. These observations motivated us to adopt a hybrid scheduling strategy to enhance efficiency.

**The hybrid scheduler.** The design of our scheduler is shown in Fig. 9. The hybrid scheduler consists of three parts: a pipeline containing ($p_u$+2) pipes, an executor pool containing $p_d$ workers, and a collector thread. To handle an update to a treap with hybrid scheduling, we first divide the treap into two parts: the upper part includes all nodes with a depth smaller than $p_u$, and the lower part contains the remaining nodes.

In the pipeline side, the first pipe, known as the *entry*, receives submitted updates and creates the placeholder for the root. Each of the following $p_u$ pipes performs one step as outlined in Alg. 6, initializing the nodes at their corresponding depths. The final pipe in this pipeline is called the *border*, which does not perform updates but instead maintains a concurrent message queue.

The concurrent workers repeatedly fetch tasks from the message queue maintained by the border pipe and finish updates for the lower part, where a worker handles the entire lower part of updates it fetches as outlined in Alg. 5. When a task is finished, it passes the new version to the collector and then fetches the next task.

Finally, the collector commits the new versions in the order the updates are submitted. Even though the pipeline parallelism and the waiting mechanism adopted by concurrent workers have already preserved the order of updates, this rule is necessary because the $n$-th version may reference parts of previous versions that are still in progress. Thus, we must wait until all previous $n-1$ updates are fully completed to ensure the completeness of the $n$-th version.

The hybrid method combines the advantages of both concurrent strategies. For the pipeline side, the upper part of a sufficiently large balanced BST can almost be considered a full binary tree, while most nodes are located in the lower part. It is rare that an update will be completed in the upper part, preventing performance degradation of the pipeline. For concurrent workers, if the elements to be updated are not overly concentrated, the probability of the search paths for different updates intersecting decreases exponentially with increasing depth. The synchronization overhead is significantly reduced for concurrent workers handling the lower part, allowing us to efficiently utilize more threads to increase concurrency.

Besides, with the enrolling of the collector, we can further optimize concurrent workers to slightly reduce the synchronization overhead. During the update process of a PBST, it is evident that the version of any node $T$ will never be older than the version of its children. Therefore, once we find that the version of the currently visited node has already been committed, we no longer need to check whether the nodes in its subtrees have been initialized.

The remaining problem is how to divide the upper and lower parts to achieve optimal performance with a total of $p$ threads. A crucial factor is the splitting operation. When it occurs in the pipeline part, it introduces stalls that delay all subsequent updates. If it is processed by concurrent workers, it only impacts updates whose search paths fall into that subtree. Therefore, the number of pipeline stages should be limited to reduce the probability of splits occurring in the pipeline part. In the meantime, we aim for the updates being processed by concurrent workers to be as independent as possible. Intuitively, when the depth of the upper part is $\log p_d$, there are $p_d$ nodes at the boundary between the upper and lower parts. In expectation, each boundary node having one search path starting at it. However, the generic birthday paradox suggests that for $n$ possible values, there is a high probability of collisions after just $O(\sqrt{n})$ attempts [68]. Therefore, we finally set $p_u = 2 + 2\log p_d$ in hybrid scheduler for better performance.

## 5 RELATED WORKS

Managing historical data has been a longstanding research area. Early methods, such as storing all updates in a log for replay or saving a version after each update, incurred significant time or storage costs. Driscoll et al. [29] introduced persistent data structures, providing techniques that reduce these costs. These structures are

commonly used in database versioning and transaction-time applications [44, 48], as well as in software version control [21].

PBSTs support historical queries and implement read-write separation by providing read-only snapshots, allowing queries during updates [12, 20, 57, 71]. Furthermore, they serve as the foundation for multiversion graph processing [28] and enhance data with additional timestamps for graph mining [65].

Temporal indices [8, 24, 25, 43] treat time intervals as ranges to support history queries, differing from our focus on efficient historical range statistics. Some approaches provide approximate historical query results using persistent sketches [63, 72], whereas we aim to deliver accurate results for arbitrary versions. Research on compression schemes for persistent balanced trees [27, 28] is orthogonal to our objectives.

Concurrent data structures enable multiple processes to access and manipulate data simultaneously without conflicts, enhancing performance in multi-threaded environments. They ensure thread safety using mechanisms like locks [64], transactional memory [40], or non-blocking algorithms [39], which ensure operations are performed atomically and maintain data integrity during concurrent access. Common examples include concurrent priority queues [58], stacks [38], hash maps [49], and trees [7, 45], each suited to specific scenarios and performance needs. The choice and implementation of a concurrent data structure significantly influence system scalability and throughput, especially under high load [17].

## 6 EXPERIMENT

In this section. We perform several groups of experiments to demonstrate the advantages of our proposal. All experiments are conducted on a Linux machine with 2 pieces of Intel® Xeon® Gold 5320 Processor and 1TB of memory.

Our main competitor is PAM [67], which is the augmented join-based PBST. Additionally, we implement an augmented PBST with serial updates. These solutions, along with our proposal (named **Contreap**), are implemented in C++20 and compiled with full optimization using GCC. We utilize the built-in AES-NI [35] to provide a perfect hash function and mimalloc [46] for multithreaded memory management. Furthermore, we adopt another baseline: Google's open source B-Tree [2], which implements a Clone method to create a persistent snapshot of the B-Tree (PB-Tree). For our proposal, the number of pipes is set to $2 \log P$. For the other methods, we adhere to the default parameter settings as specified in their original papers or source codes. By default, in the following experiments, we initialize the set with $10^8$ records and then perform $10^7$ operations.

### 6.1 Effectiveness of Hybrid Scheduling

In this section, we examine the effectiveness of our hybrid scheduling. We compare our full-fledged Contreap with hybrid scheduling against (i) Contreap with the naive blocking-based solution (dubbed as Contreap-wait) as Alg. 5 describes and (ii) Contreap with the pure pipeline method (dubbed as Contreap-pipe) as Alg. 6 describes.

Fig. 10 shows the total time spent on completing the insertions. The pipeline solution *Contreap-pipe* does not significantly outperform the sequential execution. This is primarily because the last pipe bears the bulk of the workload and thus becomes the bottleneck. Yet, the efficiency markedly improves when the number of

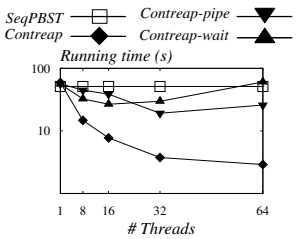



**Figure 10: Scheduling**

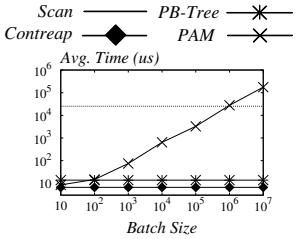



**Figure 11: Query Efficiency**

pipeline stages is 32, as the average depth of nodes being slightly less than 30 in a treap with $10^8$ nodes. When the number of threads increases beyond this point, most of the subsequent pipes stay idle, and the performance of the pipeline starts to decline due to the overhead as we described in Sec. 4. In the meantime, the naive blocking-based solution *Contreap-wait* also results in inferior performance. In the upper part of the treap, there are only a small number of possible search paths, so the concurrency is actually limited if naive blocking is applied. Therefore, when too many threads are involved, the data races lead to a decline in performance.

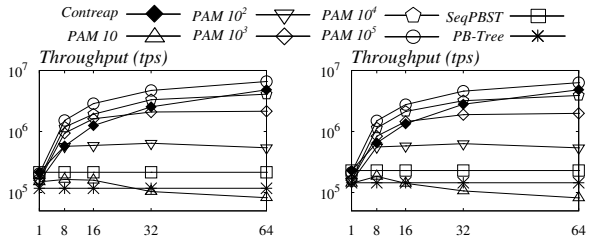
Combining the strengths of both methods, the full-fledged *Contreap* with the hybrid strategy shows a clear advantage and can gain better efficiency with more threads. Notably, our Contreap gains approximately 10x speed-up over Contreap-wait and 20x speed-up over Contreap-pipe when the number of threads reaches 64.

### 6.2 Range Query Performance

In this set of experiment, we examine the query performance of Contreap against competitors on range statistic queries. We compare our Contreap against PB-Tree, which maintains a version for each update using a B-Tree structure, allowing direct range statistics queries on any specific version. Additionally, as discussed in Sec. 2.4, existing parallel BST algorithm PAM [67] maintains coarse-grained versions by applying batch updates, which compromises query efficiency—a fact we will verify in our experiments. For PAM, we report its query performance with different batch sizes (ranging from 10 to $10^7$). Each query covers $0 \sim 10\%$ of the elements. We also include a naive *Scan* method that performs a scan over a chunked linked list, where each chunk contains 64 elements. This simulates the process of computing range statistics by scanning memory storage based on a B+-Tree, without using any indexing. Fig. 11 shows the average time of performing $10^5$ range statistics.

First, comparing PB-Tree and our Contreap, Contreap is over 2x faster than PB-Tree. This difference can be explained by the time complexity: for BST, the range statistics query has a complexity of $O(\log n)$, while for PB-Tree, it is $O(B \log_B n)$. Furthermore, our Contreap also outperforms PB-Tree in terms of update efficiency and indexing costs (as to be shown shortly), making it a more suitable choice for historical range statistic queries.

Next, both Contreap and PB-Tree are significantly faster than PAM when the batch size for updates exceeds $10^3$. When the batch size is 10, PAM's query performance is slightly slower than our Contreap. However, as the batch size increases, coarse-grained updates introduce significant overhead in range statistic queries. For example, consider the number of elements in the BST at timestamp $t$. If we maintain this aggregation information at each node, we can directly access the root of the BST at timestamp $t$ to obtain the



(a) Insertion (b) Deletion

**Figure 12: Insertion and Deletion Performance.**

count. However, with a coarse-grained version, if the BST contains some elements that have been modified at a moment later than timestamp $t$, we need to traverse downward to find the correct version, potentially repeating this process multiple times, which leads to high query costs. As the batch size continues to grow, the proportion of updates modified later than the queried timestamp will also increase. The overhead from the coarse-grained index becomes the dominant factor in query time, resulting in a nearly linear relationship between query time and batch size. When the batch size reaches $10^6$, the query performance is even slower than the naive Scan method. Therefore, we do not include results for batch sizes of $10^6$ and $10^7$ in subsequent efficiency tests.

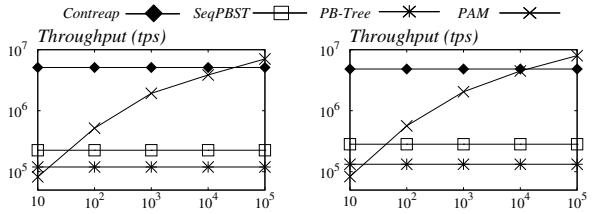
## 6.3 Update Performance

In this section, we compare the update performance of Contreap with existing solutions. Figs. 12(a) and 12(b) show the results for insertions and deletions, respectively. The $y$-axis represents throughput in transactions per second (tps). Across all solutions, the performance of insertions and deletions is comparable. Using 64 threads, our method achieves over 20 times higher throughput compared to sequential execution, while preserving order for updates.

PAM avoids synchronization overhead since divide-and-conquer scheme ensures that tasks among different threads are completely independent. On the one hand, when the batch size is sufficiently large, PAM demonstrates extremely high efficiency. On the other hand, since batches are executed one-by-one, serially, when the batch size is small, PAM cannot utilize all the computing resources inside each batch. When the batch size is 10, its throughput is smaller than that of sequentially applying all the updates with a single thread due to the scheduling overhead.

With 64 threads, the throughput of Contreap is close to that of PAM with a batch size of $10^4$, approximately 2.5x that of PAM with a batch size of $10^3$. Moreover, even with a batch size of $10^5$, the update throughput of Contreap is still 70% of that of PAM. This demonstrates that our method still maintains competitive efficiency compared to the batch-based method PAM, when its batch size is set to be sufficiently large, causing degraded query performance.

In addition, the update throughput of the PB-Tree is around half of that of the SeqPBST. Compared to BSTs, the larger size of B-Tree nodes requires more time during copy-on-write, and the larger fanout of B-Tree nodes leads to longer time to compute the node augmentation. In addition, there is currently no concurrent update scheme for PB-Tree while our Contreap can support efficient concurrent update and scales linearly with the number of threads.



(a) Uniform (b) Zipfian

**Figure 13: Performance under Different Distributions.**

## 6.4 Performance under Skewed Workloads

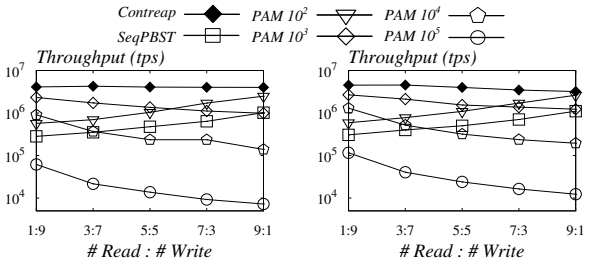
In this section, we use YCSB [26] to generate workloads with different distributions of updates. Fig. 13 shows the update throughput under different distributions using 64 threads.As shown in Fig. 13(a) and (b), the throughput of Contreap is similar under both uniform and Zipfian distributions. Under the Zipfian distribution, 80% of the updates are concentrated on 20% of the elements, leading to a significant increase in the access frequency of the most frequently accessed elements. Although a skewed distribution can result in increased conflicts among parallel workers in hybrid scheduling, this impact on performance is not significant. Under the Zipfian distribution, the throughput of PAM slightly 5% increases. This is because a key is more likely to be updated multiple times within a batch under under a skewed distribution. In this case, PAM can merge these updates in advance to reduce the actual update cost.

From the above experiments, we found that the update throughput of PAM with a batch size of 10 is lower than that of serial updates, and its query efficiency is also inferior to that of SeqPBST. Hence, we will exclude this setting from subsequent efficiency tests.

## 6.5 Space Consumption

In this section, we compare the space consumption of Contreap against existing solutions. Fig. 15 shows the memory peak when inserting $10^7$ elements into the data structure that initially contains $10^8$ elements. As we describe in Sec. 2.4, if all updates are executed as a huge batch, the space consumption of PAM is $O(n+m\log(n/m))$ to store the original version and the updated version, which corresponds to the case where the batch size is $10^7$. Accordingly, if the batch size of PAM is set to $b$, then a total of $m/b$ batch are executed to complete the updates, resulting in a total space consumption of $O(n+(m/b)\cdot b\log(n/b)) = O(n+m\log(n/b))$ to store those versions. As $b$ decreases, the space overhead gradually increases, ultimately becoming $O(n+m\log n)$, which is the same as that of Contreap and SeqPBST. Note that, when the batch size is 10 or 100, the actual space consumption of PAM exceeds that of SeqPBST. This is because PAM keeps additional information at each node to ensure that information is not lost when a key is updated multiple times within a batch, which introduces additional space overhead.

The space consumption to maintain SeqPBST is approximately 2.3x that of PAM with a batch size of $10^7$, and 1.5x that of PAM with a batch size of $10^5$. Additionally, although Contreap introduces additional space to enhance concurrency efficiency compared to the SeqPBST, it results in only about 1% additional space overhead in our experiments. Additionally, in this paper, we focus solely on updating the index while retaining all versions. In practical applications, different garbage collection strategies, such as discarding old




(a) Uniform (b) Zipfian

**Figure 14: Performance Under Interleaved Workloads.**

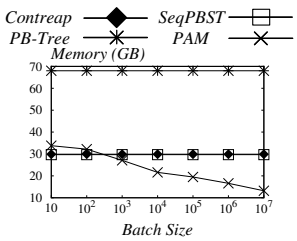



**Figure 15: Memory Usage.**

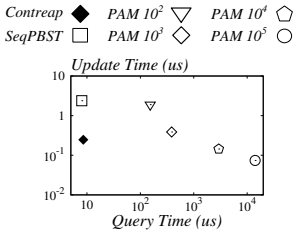



**Figure 16: Case Study.**

versions, writing them to external storage, or compressing them, can be employed in various scenarios.

To maintain a persistent data structure, the space overhead of the PB-Tree is much higher than that of the SeqPBST and Contreap. Theoretically, the space for creating a new version of a B-Tree is $O(B \log_B n)$. In practice, with the default settings of the Google B-Tree, where $B$=8, the space overhead of the PB-Tree is over 2.1x that of the PBST. So far, we find that when maintaining a persistent index to support historical range statistics, the update efficiency, query efficiency, and space consumption of PB-Tree are all inferior to those of SeqPBST, and hence our Contreap, which supports more efficient updates with our concurrent update strategies. This indicates that PB-Tree is not suitable for this scenario. Hence, we will not include PB-Tree in subsequent experiments.

## 6.6 Performance under Interleaved Workloads

Next, we compare Contreap and PAM using YCSB under workloads that contain interleaved updates and queries in varying proportions. Fig. 14(a) (resp. Fig. 14(b)) shows the throughput with 16 clients and 64 background threads with keys generated with uniform distribution (resp. Zipfian distribution). Generally, under different update-query ratios and varying data distributions, Contreap consistently achieves higher throughput than competitors. Under uniform setting, we have the following observations. Firstly, compared to PAM with a update batch size of $10^5$ (resp. $10^4$), our Contreap achieves up to 550x (resp. 28x) higher throughput. The key reason is that with a large batch size, the query efficiency is significantly degraded. Besides, compared to PAM with a update batch size of $10^3$, our Contreap achieves around 4x higher throughput when the query ratio is no smaller than 50%. In addition, compared to PAM with update batch size of $10^2$, the update performance becomes a key bottleneck and our Contreap gains up to 7x higher throughput. Finally, for SeqPBST, its key bottleneck is update. Hence, when the query ratio increases, it gains better throughput as expected. Our Contreap is 20x faster than SeqPBST when the query ratio is 10% and still 4x faster than SeqPBST when the query ratio reaches 90%. We have similar observations under Zipfian distribution.

In summary, our solution, Contreap, efficiently maintains a version for each update to support high speed range statistics queries. This allows Contreap to adapt to various interleaved workloads.

## 6.7 Case Study

In this section, we perform a case study to illustrate potential applications of Contreap. We used our proposed Contreap as an index for accelerating a critical query in E-commerce. In such a scenario, the database stores all the order information. We consider statistical queries in the following form: given a historical moment $t$ and a price range $[v_{min}, v_{max}]$, return the gross merchandise volume (GMV) of the products whose prices fall within the range $[v_{min}, v_{max}]$ up to time $t$. To speed up the query efficiency of the GMV queries, we construct the query index using the prices of the products as keys. To simulate this application, we first create a product list containing 10 million products. Then, we generated prices for each product using a normal distribution to mimic the pricing patterns in reality. Next, we generated transactions from a Zipfian distribution to simulate the sales volume of popular and less popular products in real life.

Fig. 16 shows the update time and the average query time using different index. It is worth noting that, under the influence of both the normal distribution and the Zipfian distribution, the keys, i.e., the price of the sold product, become highly concentrated. Even under such more skewed workload, it has almost no impact to our Contreap in terms of update performance.

With the increase of the update batch size, the query efficiency of PAM significantly declines as the range statistics need to examine more nodes. When the update batch size is 100, the query time of PAM is about 20x that of Contreap while the update time is 10x that of Contreap. When the batch size reaches $10^5$, the query time becomes 1700x slower even though it gains slight improvement for update efficiency. Hence, our solution gains a tremendous balance among update and query efficiency. In contrast, PAM with different batch sizes will either degrade the update efficiency or the query efficiency. Compared to SeqPBST, our Contreap gains much better update efficiency due to our proposed concurrent update strategies. Overall, the experimental results demonstrate that our Contreap is the preferred indices for such applications.

## 7 CONCLUSION

In this paper, we propose Contreap, a novel concurrent update method for persistent BSTs. Our solution applies $m$ updates to a tree containing $n$ elements in $O(\log n + m)$ time using a pipeline, utilizing $O(\log n)$ stages. We further propose a hybrid scheduling scheme to improve the scalability and practical performance of our solution. Experimental results show that our proposal achieves a good balance between update and query performance.

## ACKNOWLEDGMENTS

This work is supported by the RGC GRF grant (No. 14217322), Hong Kong ITC ITF grant (No. MRP/071/20X), and Tencent Rhino-Bird Focused Research Grant.

# REFERENCES


[1] 2025. code and technical report. https://github.com/CUHK-DBGroup/Contreap.

[2] 2025. Google BTree. https://github.com/google/btree.

[3] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. 2008. Column-stores vs. row-stores: how different are they really?. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*. 967–980.

[4] Stephen Adams. 1993. Functional pearls efficient sets—a balancing act. *Journal of functional programming* 3, 4 (1993), 553–561.

[5] Georgii Maksimovich Adelson-Velskii and Evgenii Mikhailovich Landis. 1962. An algorithm for organization of information. In *Doklady Akademii Nauk*, Vol. 146. 263–266.

[6] Ahmed M. Aly, Hazem Elmeleegy, Yan Qi, and Walid Aref. 2016. Kangaroo: Workload-Aware Processing of Range Data and Range Queries in Hadoop. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*. 397–406.

[7] Maya Arbel-Raviv, Trevor Brown, and Adam Morrison. 2018. Getting to the Root of Concurrent Binary Search Tree Performance. In *ATC*. 295–306.

[8] 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.

[9] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

[10] Jon Louis Bentley and Jerome H. Friedman. 1979. Data Structures for Range Searching. *ACM Comput. Surv.* 11, 4 (1979), 397–409.

[11] Jon Louis Bentley and James B Saxe. 1978. Decomposable searching problems. (1978).

[12] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *SIGMOD*. 1–10.

[13] Guy E Blelloch, Daniel Ferizovic, and Yihan Sun. 2016. Just join for parallel ordered sets. In *SPAA*. 253–264.

[14] Guy E Blelloch, Daniel Golovin, and Virginia Vassilevska. 2008. Uniquely represented data structures for computational geometry. In *SWAT*. 17–28.

[15] Guy E Blelloch and Margaret Reid-Miller. 1997. Pipelining with futures. In *SPAA*. 249–259.

[16] Guy E Blelloch and Margaret Reid-Miller. 1998. Fast set operations using treaps. In *SPAA*. 16–26.

[17] Andre B. Bondi. 2000. Characteristics of scalability and their impact on performance. In *WOSP*. 195–203.

[18] Richard P. Brent. 1974. The Parallel Evaluation of General Arithmetic Expressions. *J. ACM* 21, 2 (1974), 201–206.

[19] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A practical concurrent binary search tree. In *PPoPP*. 257–268.

[20] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2008. Serializable isolation for snapshot databases. In *SIGMOD*. 729–738.

[21] Scott Chacon and Ben Straub. 2014. *Pro git*. Apress.

[22] Chee Yong Chan and Yannis E. Ioannidis. 1999. Hierarchical Prefix Cubes for Range-Sum Queries. In *Proceedings of the 25th International Conference on Very Large Data Bases*. 675–686.

[23] 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.

[24] George Christodoulou, Panagiotis Bouros, and Nikos Mamoulis. 2022. HINT: A Hierarchical Index for Intervals in Main Memory. In *SIGMOD*. 1257—-1270.

[25] George Christodoulou, Panagiotis Bouros, and Nikos Mamoulis. 2024. LIT: Lightning-fast In-memory Temporal Indexing. *Proc. ACM Manag. Data* 2, 1, Article 20 (2024).

[26] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. 143–154.

[27] Laxman Dhulipala, Guy E. Blelloch, Yan Gu, and Yihan Sun. 2022. PaC-trees: supporting parallel and compressed purely-functional collections. In *PLDI*. 108–121.

[28] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2019. Low-latency graph streaming using compressed purely-functional trees. In *PLDI*. 918–934.

[29] James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. 1986. Making Data Structures Persistent. In *STOC*. 109–121.

[30] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-blocking binary search trees. In *PODC*. 131–140.

[31] Panagiota Fatourou, Elias Papavasileiou, and Eric Ruppert. 2019. Persistent Non-Blocking Binary Search Trees Supporting Wait-Free Range Queries. In *SPAA*. 275–286.

[32] Florian Funke, Alfons Kemper, and Thomas Neumann. 2012. Compacting transactional data in hybrid OLTP&OLAP databases. *Proc. VLDB Endow.* 5, 11 (2012), 1424–1435.

[33] Sathish Govindarajan, Pankaj K. Agarwal, and Lars Arge. 2003. CRB-Tree: An Efficient Indexing Scheme for Range-Aggregate Queries. In *International Conference on Database Theory*.

[34] J. Gray, A. Bosworth, A. Lyaman, and H. Pirahesh. 1996. Data cube: a relational aggregation operator generalizing GROUP-BY, CROSS-TAB, and SUB-TOTALS. In *Proceedings of the Twelfth International Conference on Data Engineering*. 152–159.

[35] Shay Gueron. 2010. Intel® Advanced Encryption Standard (AES) New Instructions Set. https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf

[36] Leo J Guibas and Robert Sedgewick. 1978. A dichromatic framework for balanced trees. In *FOCS*. 8–21.

[37] Antonin Guttman. 1984. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*. 47–57.

[38] Danny Hendler, Nir Shavit, and Lena Yerushalmi. 2004. A scalable lock-free stack algorithm. In *SPAA*. 206–215.

[39] Maurice Herlihy, Victor Luchangco, and Mark Moir. 2003. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *ICDCS*. 522–529.

[40] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA*. 289–300.

[41] Ching-Tien Ho, Rakesh Agrawal, Nimrod Megiddo, and Ramakrishnan Srikant. 1997. Range queries in OLAP data cubes. *SIGMOD Rec.* 26, 2 (1997), 73–88.

[42] 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.

[43] Hans-Peter Kriegel, Marco Pötke, and Thomas Seidl. 2000. Managing Intervals Efficiently in Object-Relational Databases. In *VLDB*. 407–418.

[44] Krishna G. Kulkarni and Jan-Eike Michels. 2012. Temporal features in SQL: 2011. *SIGMOD Rec.* 41, 3 (2012), 34–43.

[45] H. T. Kung and Philip L. Lehman. 1980. Concurrent Maipulation of Binary Search Trees. *ACM Trans.Database Syst.* 5, 3 (1980), 354–382.

[46] Daan Leijen, Benjamin Zorn, and Leonardo Moura. 2019. *Mimalloc: Free List Sharding in Action*. 244–265.

[47] 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.

[48] David B. Lomet, Mingsheng Hong, Rimma V. Nehme, and Rui Zhang. 2008. Transaction time indexing with version compression. *Proc. VLDB Endow.* 1, 1 (2008), 870–881.

[49] Tobias Maier, Peter Sanders, and Roman Dementiev. 2019. Concurrent Hash Tables: Fast and General(?)! *ACM Trans. Parallel Comput.* 5, 4 (2019), 16:1–16:32.

[50] Paul E. McKenney and Jack Slingwine. 2002. Read-Copy Update: Using Execution History to Solve Concurrency Problems.

[51] Aravind Natarajan and Neeraj Mittal. 2014. Fast concurrent lock-free binary search trees. In *PPoPP*. 317–328.

[52] Jürg Nievergelt and Edward M Reingold. 1972. Binary search trees of bounded balance. In *STOC*. 137–142.

[53] Dimitris Papadias, Panos Kalnis, Jun Zhang, and Yufei Tao. 2001. Efficient OLAP Operations in Spatial Data Warehouses. In *Advances in Spatial and Temporal Databases*. 443–459.

[54] Dimitris Papadias and Yufei Tao. 2004. Range Aggregate Processing in Spatial Databases . *IEEE Transactions on Knowledge & Data Engineering* 16, 12 (2004), 1555–1570.

[55] Christian Plattner, Andreas Wapf, and Gustavo Alonso. 2006. Searching in time. In *SIGMOD*. 754–756.

[56] Chung Keung Poon and Hao Yuan. 2013. A Faster CREW PRAM Algorithm for Computing Cartesian Trees. In *Algorithms and Complexity*. 336–344.

[57] Dan R. K. Ports and Kevin Grittner. 2012. Serializable Snapshot Isolation in PostgreSQL. *Proc. VLDB Endow.* 5, 12 (2012), 1850–1861.

[58] V. Nageshwara Rao and Vipin Kumar. 1988. Concurrent Access of Priority Queues. *IEEE Trans. Computers* 37, 12 (1988), 1657–1665.

[59] Bruno Ribeiro, Minh X. Hoang, and Ambuj K. Singh. 2015. Beyond Models: Forecasting Complex Network Processes Directly from Data. In *WWW*. 885–895.

[60] Adam Seering, Philippe Cudré-Mauroux, Samuel Madden, and Michael Stonebraker. 2012. Efficient Versioning for Scientific Array Databases. In *ICDE*. 1013–1024.

[61] Raimund Seidel and Cecilia R Aragon. 1996. Randomized search trees. *Algorithmica* 16, 4-5 (1996), 464–497.

[62] 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.

[63] 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.

[64] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. 2018. *Operating System Concepts*.

[65] Uriel Singer, Ido Guy, and Kira Radinsky. 2019. Node Embedding over Temporal Graphs. In *IJCAI*. 4605–4612.

[66] Lawrence Snyder. 1977. On uniquely represented data strauctures. In *FOCS*. 142–146.

[67] Yihan Sun, Daniel Ferizovic, and Guy E Belloch. 2018. PAM: parallel augmented maps. In *PPoPP*. 290–304.
[68] Kazuhiro Suzuki, Dongvu Tonien, Kaoru Kurosawa, and Koji Toyota. 2006. Birthday Paradox for Multi-collisions. In *ICISC*. 29–40.
[69] Yufei Tao, Cheng Sheng, Chin-Wan Chung, and Jong-Ryul Lee. 2014. Range Aggregation With Set Selection. *IEEE Transactions on Knowledge and Data Engineering* 26, 5 (2014), 1240–1252.
[70] 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.
[71] Yuanhao Wei, Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. 2021. Constant-time snapshots with applications to concurrent data structures. In *PPoPP*. 31–46.
[72] Zhewei Wei, Ge Luo, Ke Yi, Xiaoyong Du, and Ji-Rong Wen. 2015. Persistent Data Sketching. In *SIGMOD*. 795–810.
[73] Dongxiang Zhang, Yeow Meng Chee, Anirban Mondal, Anthony K. H. Tung, and Masaru Kitsuregawa. 2009. Keyword Search in Spatial Databases: Towards Searching by Document. In *2009 IEEE 25th International Conference on Data Engineering*. 688–699.