IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 31 July 2025, accepted 10 August 2025, date of publication 13 August 2025, date of current version 22 August 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3598736

RESEARCH ARTICLE

SB-Tree: A B+-Tree for In-Memory Time Series Databases With Segmented Block

CHRISTINE EUNA JUNG[1,2], JAESANG HWANG[1,3], YEDAM NA[1], HAENA LEE[1], AND WOOK-HEE KIM[1], (Member, IEEE)

[1]Department of Computer Science and Engineering, Konkuk University, Seoul 05029, Republic of Korea
[2]Toss Payments, Seoul 06133, Republic of Korea
[3]Dnotitia, Seoul 06628, Republic of Korea

Corresponding author: Wook-Hee Kim (wookhee@konkuk.ac.kr)

This work was supported in part by Korea Government (KASA, Korea AeroSpace Administration) (grant number RS-2022-00155668, Development of data processing and analysis system for health monitoring of commercial aircraft), and in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) under the Metaverse Support Program to Nurture the Best Talents (IITP-2025-RS-2023-00256615) grant funded by the Korea Government (MSIT).

ABSTRACT Various devices such as smart cars, healthcare systems, and IoT platforms generate a massive amount of time series data. This type of data exhibits unique characteristics that pose new challenges for in-memory indexes, including heavy insert rates, monotonically increasing keys, and workloads dominated by range queries. In this paper, we propose a new in-memory index, Segmented Block-based Tree (SB-Tree), designed specifically for time series workloads. To support high insert throughput, SB-Tree decouples the index into a search layer and a data layer, and updates the search layer asynchronously to reduce write amplification. It introduces a shortcut mechanism to reduce tree traversal overhead and uses segmented blocks with per-thread data blocks to minimize contention during inserts. In addition, SB-Tree employs a lightweight block allocator to reduce system call overhead during memory management. We evaluate SB-Tree using synthetic time series workloads with varying levels of delayed data, inserting up to 1 billion keys across 80 threads. Our results show that SB-Tree outperforms state-of-the-art in-memory indexes, achieving up to 7× higher throughput on insert-only workloads and 2.4× lower 99.99th percentile tail latency on scan workloads compared to state-of-the-art.

INDEX TERMS In-memory index, time series database, insert-heavy workloads, segmented block.

I. INTRODUCTION

The explosive growth of time series data across diverse domains such as Internet of Things (IoT) [1], [2], [3], sensor measurements [4], [5], network traffic [6], [7], [8], and financial transactions [9], [10], [11] has heightened the demand for scalable and efficient indexing mechanisms. Time series data exhibits unique characteristics, including monotonically increasing timestamps, delayed data, and high insert intensity. These properties impose significant challenges on conventional indexing structures. For example, in IoT environments, numerous edge devices periodically collect data from sensors and transmit it to edge servers for processing, which demands both high-throughput ingestion and low-latency query capabilities.

The associate editor coordinating the review of this manuscript and approving it for publication was Hang Shen.

Traditional tree-based indexes such as B+-trees [12], [13], [14], [15] and tries [16], [17], [18] are optimized for Online Transaction Processing (OLTP) workloads and are not tailored for time series workloads.

In particular, time series data has monotonically increasing keys, causing heavy contention when multiple threads attempt concurrent inserts. This contention limits the scalability of indexes, so effectively distributing the concurrent inserts becomes a critical requirement for time series databases. A previous study [19] distributes concurrent inserts by employing a hash table-based node in the B^{link}-tree. The hash table-based node successfully reduces the contention from concurrent inserts, but it incurs higher tail latency when converting a hash-table based node to a B^{link}-tree style node.

In this paper, we propose *Segmented Block-based Tree* (SB-Tree), a novel in-memory index structure tailored for

time series workloads. SB-Tree is composed of two logical layers: a data layer and a search layer. The data layer consists of segmented blocks and data blocks, which are designed to support both high-throughput inserts and efficient range queries.

To achieve high insert performance, SB-Tree employs a *per-thread data block* to reduce contention from the concurrent inserts and a shortcut to minimize the tree traversal overhead. SB-Tree also adopts a customized memory allocation scheme. Conventional memory system calls are not scalable under high-concurrency workloads and pose a performance bottleneck. While allocators such as TCMalloc or jemalloc can be used, they also suffer from scalability issues in highly threaded environments. To address this, SB-Tree employs a *block allocator* that minimizes memory allocation overhead.

To maximize range query performance, SB-Tree also applies two key optimizations. First, SB-Tree separates the memory layouts for inserts and queries. We observe that access patterns differ between inserts and queries. Second, SB-Tree introduces an additional optimization called the *N-ary search table*. This table stores a subset of keys in a contiguous memory region to assist in narrowing the search range. By leveraging sequential access to this table, SB-Tree reduces random memory accesses and minimizes the number of cache lines required to search for a target key.

We conduct an extensive performance evaluation of SB-Tree under time series workloads, comparing it with existing tree-based indexes. In our experiments, SB-Tree demonstrates scalable and superior performance under monotonic insert scenarios, achieving up to 4.38× higher throughput than the state-of-the-art index. We also evaluate its performance on delayed data using a novel method and compare memory footprint to further validate the efficiency of SB-Tree.

The contributions of this paper are as follows:

- We propose SB-Tree, a new in-memory index tailored for time series workloads. Its design integrates segmented blocks, a shortcut mechanism, an N-ary search table, and a lightweight block allocator to reduce contention, minimize traversal overhead, improve cache efficiency, and reduce memory management overhead.
- We conduct an extensive evaluation with up to 1 billion keys from time series workloads using 80 threads, demonstrating that SB-Tree achieves up to 7× higher insert throughput and 2.4× lower 99.99th percentile scan latency compared to state-of-the-art in-memory indexes.

The remainder of this paper is organized as follows. section II explains the characteristics of time series workloads and discusses the limitations of existing index structures. section III presents the design of SB-Tree, and section IV describes how SB-Tree's operation works. section V presents comprehensive experimental results and analyzes the performance. section VI discusses the limitations and generality of SB-Tree and section VII concludes the paper.

II. BACKGROUND

A. TIME SERIES DATA AND TIME SERIES DATABASE

1) TIME SERIES DATA

Nowadays, the importance of efficiently storing and analyzing time series data is increasing. Time series data refers a sequence of data that is generated periodically. It is collected and utilized in various applications, including sensor monitoring, network traffic analysis, and stock market prediction [20].

The Internet of Things (IoT) is a well-known source of time series data. In IoT scenarios, edge devices collect sensor data from attached sensors and transmit it to edge servers. These edge servers analyze the data and make real-time decisions based on the results [21]. Time series data on edge servers is typically stored in a key-value format within data management systems. The timestamp, which is the time when the data was measured by the sensor is used as a key. Most data arrives at edge servers in the order in which it was measured. However, there can be some anomalies, known as delayed data or out-of-order data. Since data is transmitted over the network, its arrival can be delayed due to network conditions [22]. The delay can vary widely, ranging from a few seconds to several days [23]. Additionally, most analytical queries for the time series data are range queries, which retrieve all data within a specific time frame [24]. Their performance strongly depends on the index structure on which the system relies. To the best of our knowledge, there are only a few index structures that support both high performance concurrent monotonic inserts and range queries.

2) TIME SERIES DATABASE

Time series databases are used for processing the time series data. They typically adopt Log Structured Merge Tree (LSM-Tree) [25], as time series workloads are dominated by insert operations. InfluxDB [26], which is one of the most widely used in industry, employs the LSM-tree to store time series data. Apache IoTDB [23] maintains two LSM-trees to separate delayed data. In the LSM-tree, compaction overhead is critical to performance, so Apache IoTDB avoids additional compaction caused by the delayed data. ForestTI [27] employs two-level inverted index, combining a trie and a LSM-tree, to improve both insert and query performance. TigerData [28], previously known as TimescaleDB, is built on PostgreSQL. Since PostgreSQL uses B+-tree as its default index, TigerData also relies on B+-tree, which is not optimized for time series workloads. Note that the LSM-tree is designed for block devices, so it is not a direct competitor of SB-Tree.

B. LIMITATIONS OF TRADITIONAL IN-MEMORY INDEXES

In-memory indexes are typically optimized for OLTP workloads and often exhibit performance degradation under time series workloads. B^{link}-tree [14] is a variant of the B+-tree designed to support high concurrency. By adding

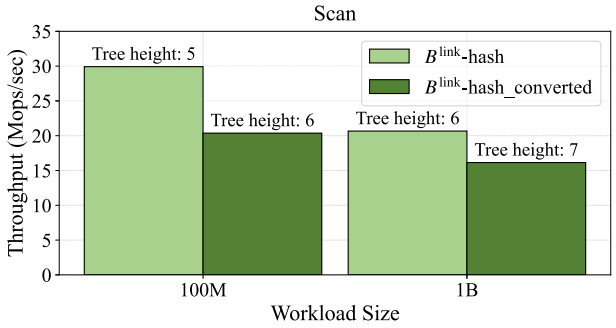

FIGURE 1. Scan throughput of B^{link}-Hash under time series workload (80 threads), showing performance drop after hash nodes are converted to tree nodes due to increased tree height.

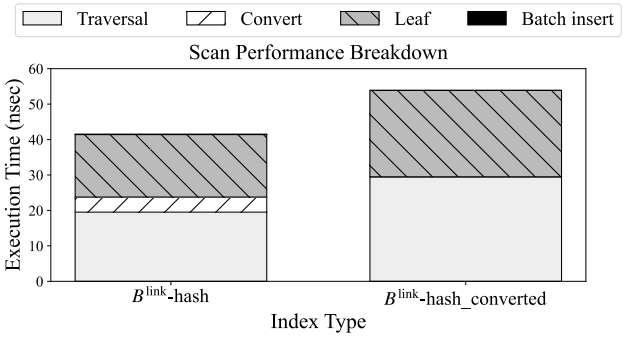

FIGURE 2. Scan performance breakdown of B^{link}-Hash in time series workload with 80 threads.

a link pointer to each node that points to the next node at the same level, B^{link}-tree can efficiently handle concurrent operations in the presence of multiple writers. Adaptive Radix Tree (ART) [16] is a trie-based structure and offers high insert and lookup performance by reducing the tree height. Height Optimized Trie (HOT) [18] is another trie-based structure optimized for memory efficiency and traversal performance. It employs a binary Patricia trie [29] with a maximum fanout of k, and dynamically adjusts each node's span based on the data distribution. Additionally, HOT maintains a minimal tree height during insert operations to improve cache efficiency. Masstree [17] is a hybrid structure that combines a B+-tree and a trie to efficiently store and retrieve variable-length keys. In Masstree, keys are divided into 8-byte chunks and managed at each level using one or more layers of B+-trees. These index structures are optimized for OLTP workloads and are therefore not well-suited for concurrent monotonic insert operations [19]. B+-tree variants suffer performance degradation due to concurrent insert operations on the same node. Moreover, structural modification operations (SMOs), such as node splits, occur frequently and incur significant overhead. To avoid frequent splits, B+-tree based indexes typically leave empty space in nodes which results in inefficient memory utilization. On the other hand, trie-based index structures do not support high-performance range queries, as finding consecutive keys requires traversing multiple levels.

An in-memory index structure for time series data should satisfy the following requirements. First, it must efficiently store simultaneously arriving data, as time series workloads are dominated by insert operations. Second, it should efficiently handle delayed data, which is a common characteristic of time series data. Third, it must provide high-performance range query, as time series data analysis heavily relies on them. Lastly, the index structure should be memory-efficient, as it consumes a significant portion of the system memory [30], and time series data is continuously generated.

C. B^{LINK}-HASH

B^{link}-Hash [19] is the only index structure specifically designed for time series workloads. It uses *hash nodes* as leaf nodes, leveraging the hash table structure to achieve high write performance and concurrency. When multiple threads attempt to write to the same hash node, they are distributed to different insert positions, reducing contention among concurrent threads. However, splitting a hash node can become a new performance bottleneck. When every bucket is full and no empty space is left, a split operation is triggered. Since a hash node contains multiple buckets, its size is relatively large, making key sorting within the hash node highly inefficient. The optimizations of B^{link}-Hash relieve this problem. By using median approximation, the split operation can begin immediately without the need of finding the exact median key. Additionally, to minimize split bottlenecks, split operations occur lazily at the bucket level. These optimizations result in high and scalable write performance.

The major challenge of using hash tables is supporting range queries. Hash tables do not support range queries by default. To overcome this limitation, B^{link}-Hash dynamically converts a hash node into a *tree node*, which has a similar structure to traditional B+-tree nodes. This conversion is triggered when a range query is executed on a hash node that is not the rightmost leaf node. As a result, B^{link}-Hash achieves range query performance comparable to that of B^{link}-tree, which offers the highest range query performance among traditional tree-based indexes.

Nevertheless, B^{link}-Hash does not fully satisfy every requirement for time series indexing. First, once all hash nodes are converted to tree nodes, high range query performance can no longer be maintained. Figure 1 presents the scan performance of B^{link}-Hash in the time series workload, described in subsection V-B. We evaluated two scenarios: (1) dynamically converting the hash nodes when range query is executed, and (2) an index composed solely of tree nodes. In the latter case, every hash node was converted into tree nodes before measuring the performance. When every hash node was converted, the performance degraded to 0.75×. Figure 2 illustrates the breakdown of B^{link}-Hash's

scan performance. *Traversal* shows the time spent on tree traversal, *Convert* shows the time spent converting hash nodes into tree nodes, *Leaf* shows the time spent fetching data from leaf nodes, and *Batch insert* shows the time spent inserting tree nodes into internal nodes in batches. Comparing the two cases, the most significant performance degradation occurred in *Traversal*, which increased by 1.51×. This was caused by the increased tree height. As the tree grows taller, the number of nodes visited during traversal also increases. This leads to more memory accesses and reduced cache efficiency [18], which degrades the overall lookup performance. Second, hash nodes are large and thus cannot efficiently utilize memory. According to B^{link}-Hash's default configuration [19], a hash node is 512KB and contains 474 buckets which is 512 bytes each. Compared to other indexing techniques, B^{link}-Hash leaves more empty space, leading low memory utilization. The measurement results are presented in subsection V-G.

To summarize, no existing indexing techniques satisfies every requirement for time series data indexing and a new indexing technique is required.

III. DESIGN OF SB-TREE

A. DESIGN OVERVIEW

SB-Tree is an in-memory index structure designed to efficiently manage time series data. To meet the requirements of indexes in time series workloads – such as high-throughput inserts and efficient range queries – SB-Tree consists of two layers as shown in Figure 3: a *data layer* and a *search layer*. The data layer contains both *data blocks* and a *segmented block*, while the search layer has *search blocks*, which provide access paths to the data blocks.

1) DATA LAYER

In the data layer, data blocks are connected as a linked list. The keys in the data layer are stored in order. Since the inserting keys are increasing in monotonic manner, most insert operations attempt to insert data into the rightmost data block in the data layer. This causes high contentions that may lead to performance degradation. To resolve the high contention, SB-Tree introduces a *segmented block* and uses it as a rightmost data block in the data layer. The segmented block contains pointers to the *per-thread data blocks*. Each thread inserts time series data using its own per-thread data block. When any of the per-thread data blocks within the segmented block becomes full, the segmented block is converted into multiple data blocks. The details of each block type will be explained in subsection III-C. Furthermore, the data layer has an additional pointer, called *shortcut* (subsection III-B), to the segmented block in the data layer. The shortcut is used to reduce tree traversal overhead during insert operations.

2) SEARCH LAYER

The search layer is a cache-optimized B+-tree designed to efficiently support data block searches. The search layer is built in a bottom-up manner by adding new data to the rightmost search block. A bottom-up build ensures a compact tree structure, efficient memory utilization, and simplified SMO operations. The search layer is updated when the segmented block is converted to data blocks. Since the SMO operations do not happen frequently, the search layer employs Read-Optimized Write EXclusion (ROWEX) [31]. The search layer has a dedicated thread that updates the search layer, while other threads read the search layer to find the proper data block.

B. SHORTCUT

Every operation in the index involves a search to find the proper position. In time series workloads, most data is inserted into the same node that holds the largest key, because the keys are monotonically increasing. Leveraging this characteristic, we reduce tree traversals using the *shortcut*. The shortcut directly points to the segmented block, allowing inserts without performing a tree traversal. In this way, SB-Tree enables fast insert and lookup operations for time series workloads.

C. SEGMENTED BLOCK AND DATA BLOCKS

Since the keys are monotonically increasing in time series workloads and SB-Tree maintains keys in sorted order, most threads attempt to insert data into the same rightmost data block in the data layer. To efficiently process the time series data, SB-Tree provides three types of blocks in the data layer as shown in Figure 4. SB-Tree assigns a *per-thread data block* to each thread to eliminate the contention during the insert operations. To manage the per-thread data blocks, SB-Tree introduces *segmented blocks*. The segmented block provides the addresses and information of the per-thread data blocks. Using this information, segmented block determines when to convert to *data blocks*, as described in subsection IV-B.

The segmented block design performs well in both high- and low-resolution workloads. In high-resolution settings, where the time series workload involves frequent write operations, SB-Tree benefits from its per-thread data blocks, which mitigate contention among threads. In low-resolution settings, where the intervals between the write operations are longer, SB-Tree can reduce the size of each per-thread data block to improve the memory utilization.

A data block is similar to a typical B+-tree leaf node designed for efficient range queries. To reduce memory accesses during key searches, data blocks store keys and values in separate arrays. They also leverage *N-ary search table* (subsection III-D) to enhance query performance.

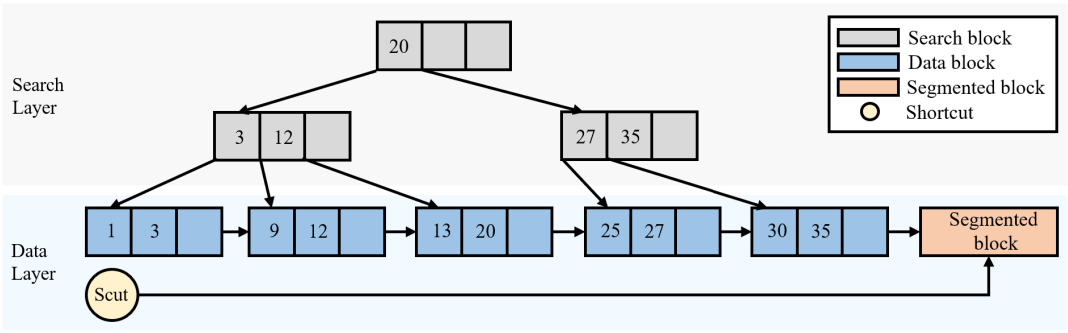

FIGURE 3. Design overview of SB-Tree.

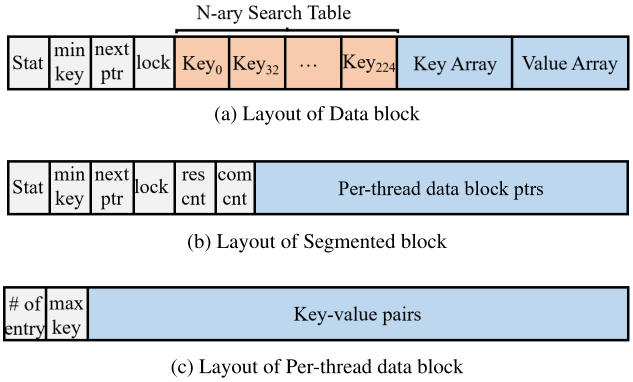

FIGURE 4. Block Layouts in SB-Tree.

D. N-ARY SEARCH TABLE

In SB-Tree, a lookup or scan operation is performed on a data block using a linear search. However, since the size of data block is 4KB, linear search shows low performance. Binary search can be an alternative solution, but it incurs random memory access, which does not efficiently leverage the memory prefetcher and caches. To take the advantages of the memory prefetcher and CPU caches, SB-Tree proposes the *N-ary search table*. SB-Tree divides each data block into multiple small buckets and selects the smallest key from each bucket to construct the N-ary search table. When SB-Tree finds a key from a data block, SB-Tree first searches the N-ary search table to determine the bucket where the key might be stored, and then performs a linear search for that bucket. This approach improves query performance by minimizing random memory accesses during search operations.

E. BLOCK ALLOCATOR

System call overhead for allocating and freeing memory spaces becomes a performance bottleneck in in-memory data structures. SB-Tree uses its own memory allocator for allocating its blocks. Since the size of the blocks is the same, the freed blocks can be easily reused across different types of allocations. The block allocator allocates a large memory space at once and slices it into multiple blocks as needed. Additionally, each thread manages its own memory blocks to minimize contention.

F. SYNCHRONIZATION

SB-Tree consists of a search layer and a data layer, each with its own concurrency control mechanism. The search layer leverages ROWEX [31] by allowing only a single dedicated thread to insert data into the tree. Thanks to the monotonically increasing characteristic of time series data, most data is appended to the segmented block, and explicit sorting is unnecessary. Meanwhile, reader threads can access the search layer without locking to locate the appropriate data block, ensuring that query performance is not degraded. Note that the intermediate process of adding new data is invisible to reader threads, as the dedicated thread atomically updates the number of keys in a node after inserting the new key-value pair.

In the data layer, data blocks use a variant of version-based locking protocol [32]. The version-based locking protocol is primarily used for inserting delayed data. In contrast, segmented block, which manages per-thread data blocks, do not require a concurrency mechanism since each thread inserts its data into its own per-thread data block. When a per-thread data block becomes full, the segmented block needs to be converted into data blocks. In this case, SB-Tree uses version-based locking to prevent additional insert operations into the segmented block and its associated per-thread data blocks.

IV. OPERATIONS

A. INSERT OPERATION FOR IN-ORDER DATA

In time series workloads, newly arriving data always generally has a greater key than the keys in the index structure. Hence, SB-Tree leverages a shortcut to directly access the segmented block, as shown in algorithm 1. Since the shortcut bypasses the search layer, it can efficiently reduce the tree traversal overhead. Note that accessing the segmented block is critical for insert operations, as the segmented block notifies that the per-thread data blocks

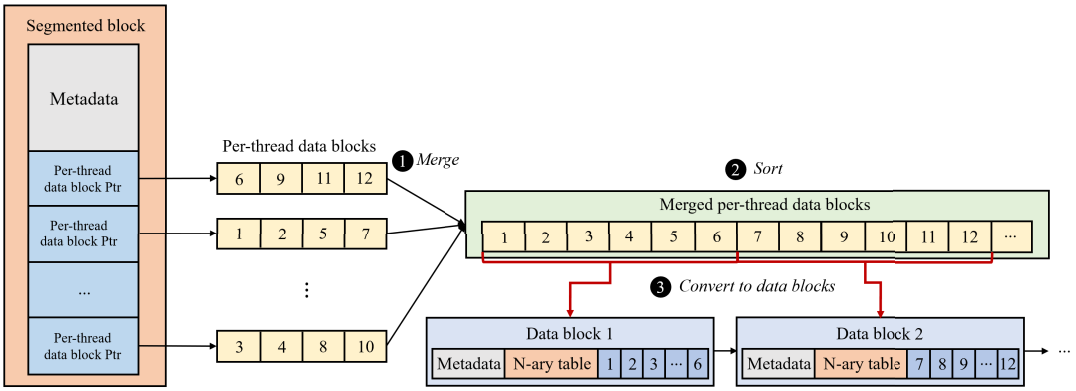

FIGURE 5. Conversion process from a Segmented block to multiple data blocks in SB-Tree.

Algorithm 1 SB-Tree Insert

```
1 Procedure Insert(key, value)
2     if ShortcutInsert(DataList, key, value) then
3         return
4     block ← FindBlock(SearchLayer, key)
5     while Covers(block.nextBlock, key) do
6         block ← block.nextBlock
7     InsertIntoBlock(block, key, value)
```

within the segmented block are being converted into data blocks.

1) INSERTING AT SEGMENTED BLOCK

In the segmented block, the insert operation checks whether the current thread already has an allocated per-thread data block. When there is no per-thread data block for the thread or the existing block is full, SB-Tree allocates a new one and assigns it to the segmented block. Otherwise, the data is stored in the current per-thread data block.

The per-thread data block stores data in the form of key-value (KV) entries, which are optimized for insert operations. Storing data as KV entries ensures that inserts occur in contiguous memory regions. Storing keys and values in separate contiguous regions would cause each insertion to access different cache lines, increasing cache usage. By storing data as KV entries, SB-Tree enables sequential memory accesses, which reduces memory operations and improves cache locality.

In SB-Tree, the per-thread data blocks are dynamically distributed to threads based on demand. The number of per-thread data blocks is atomically increased and used to get the location of the pointer variable. Each segmented block has a maximum threshold of the per-thread data block. When the number of per-thread data block exceeds this maximum, the segmented block does not allow further insert operations and is converted into the multiple data blocks.

B. CONVERTING SEGMENTED BLOCKS TO DATA BLOCKS

When a per-thread data block is full, the segmented block containing it must be converted. To maintain continuous insertion during the conversion, a new segmented block is created and pointed to by the shortcut prior to the conversion.

Figure 5 illustrates how a segmented block with per-thread data blocks is converted into multiple data blocks. The conversion process merges and sorts the KV entries from the per-thread data blocks managed by a segmented block, and converts them into multiple data blocks. During the conversion, a thread initially merges KV entries stored in per-thread data blocks and sorts them in ascending key order. Since the keys are monotonically increasing, KV entries in the per-thread data blocks tend to be partially sorted. Hence, the sorting overhead across the multiple per-thread data blocks is not significant. For sort operation, SB-Tree uses `std::sort`. The sorted KV entries are divided into multiple data blocks and appended to the data layer. In the data blocks, keys and values are stored in separate arrays. This design improves range query performance, as it allows retrieving a large number of values with fewer memory accesses.

The conversion process requires a number of memory allocation and deallocation operations. In in-memory indexes, the memory allocation and deallocation overheads are significant as these operations involve system calls. To reduce these overhead, SB-Tree uses its own memory allocator called the *block allocator*. The block allocator is optimized for blocks that are used in SB-Tree. SB-Tree uses fixed-size memory blocks and reuses freed blocks for future allocations. By designing blocks to have the same sizes, freed block memory can be easily recycled for new block allocations. To minimize memory deallocation overhead, the block allocator manages per-thread free lists instead of releasing memory back to the system.

C. INSERTING AT THE SEARCH LAYER

The conversion process creates multiple data blocks and adds them to the data layer. To efficiently look up the data, the data

Algorithm 2 SB-Tree Lookup

```
1  Procedure Lookup(key)
2      block ← FindBlock(SearchLayer, key)
3      while Covers(block.nextBlock, key) do
4          block ← block.nextBlock
5      return SearchInBlock(block, key)
```

blocks need to be indexed in the search layer. In SB-Tree, the search layer and the data layer are separated and they are asynchronously updated. To improve the concurrency, SB-Tree employs the ROWEX protocol [31] and uses a dedicated thread to update the search layer. The dedicated thread scans the data blocks created by the conversion process and inserts the minimum key and address of each data block into the search layer. Since the keys that are newly added to the search layer also have a monotonic increasing order, the search layer is built in a bottom-up manner. Note that since the newly inserted data is appended to the rightmost search block, the critical section is minimal. Data blocks that have not yet been inserted into the search layer can still be located by traversing the data layer similar to a linked list, even if the search layer points to a neighboring node.

D. INSERTING DELAYED DATA

Delayed data refers to data that does not arrive in order due to network congestion, scheduling, or other factors. SB-Tree classifies data that cannot be inserted through the shortcut as delayed data. The search layer traversal is first performed to find the appropriate data block for inserting the delayed data, as shown in algorithm 1. Since the search layer is updated asynchronously, the search operation may locate the neighboring node. To address this issue, SB-Tree performs the data layer traversal after search layer traversal to ensure that the target data block is found correctly. Once the target data block is found, the data is inserted in a way that preserves the sorted order within the data block. Maintaining this sorted structure is essential for supporting efficient range queries. When the target data block is full, the thread performs a split similar to that of a general B+-tree's split operation.

E. QUERY OPERATION

1) LOOKUP OPERATION

Algorithm 2 presents the pseudocode for the lookup operation in SB-Tree. The lookup operation performs a search layer traversal to find the appropriate data block. In the data block, an N-ary search table is used to determine whether the key is found, or if it should move to the neighboring data block to continue the search. When SB-Tree finds the appropriate data block, it scans the N-ary search table to find the bucket from which to begin a linear search. By using the N-ary search table, SB-Tree can access the minimum value of each bucket with fewer memory references, allowing for more efficient use of CPU caches and the memory prefetcher.

Algorithm 3 SB-Tree Scan

```
1  Procedure Scan(startKey, count)
2      block ← FindBlock(SearchLayer, startKey)
3      while Covers(block.nextBlock, startKey) do
4          block ← block.nextBlock
5      while count > 0 and block ≠ null do
6          scannedValues ← ScanInBlock(block,
           startKey, count)
7          results.append(scannedValues)
8          count ← count − |scannedValues|
9          block ← block.nextBlock
10     return results
```

After determining the target bucket, a linear search is performed to find the key, and if found, the corresponding value is returned.

2) SCAN OPERATION

Algorithm 3 shows the pseudocode for the scan operation in SB-Tree. The scan operation works similarly to the lookup operation. A search layer traversal is performed to find the data block where the scan operation starts. Within the data block, the N-ary search table is used to find the bucket from which to start a linear search. A linear search is then conducted to find the minimum value for the scan range. The values are retrieved sequentially as many as the query requested. Since values in the data block are stored contiguously, each memory reference can retrieve more values. This approach becomes more effective as the number of values to be retrieved increases. If a single data block does not contain enough data to satisfy the scan range, the scan operation continues by moving to the next data block until the query's requirements are met.

V. EVALUATION

A. EXPERIMENTAL SETUP

1) EXPERIMENTAL ENVIRONMENT

We conduct all experiments on a server equipped with two Intel Xeon Gold 5318Y CPUs, each containing 24 physical cores (48 Hyper-Threads per CPU). The server has 768GB of DDR4 DRAM. We pin each thread to a specific core, ensuring that each thread allocates memory on its local Non-Uniform Memory Access (NUMA) node, similar to previous work [19]. SB-Tree is implemented in C++. We use GCC 11.3.1 with the -O3 optimization flag. To accelerate searches, we utilize Intel Advanced Vector Extensions 512 (AVX-512) SIMD operations to search keys in the tree node and the N-ary search table in the data block.

2) WORKLOADS

We use a time series workload, as in previous studies [19], [33]. The time series workload simulates a scenario where data is concurrently received from 1,024 distributed

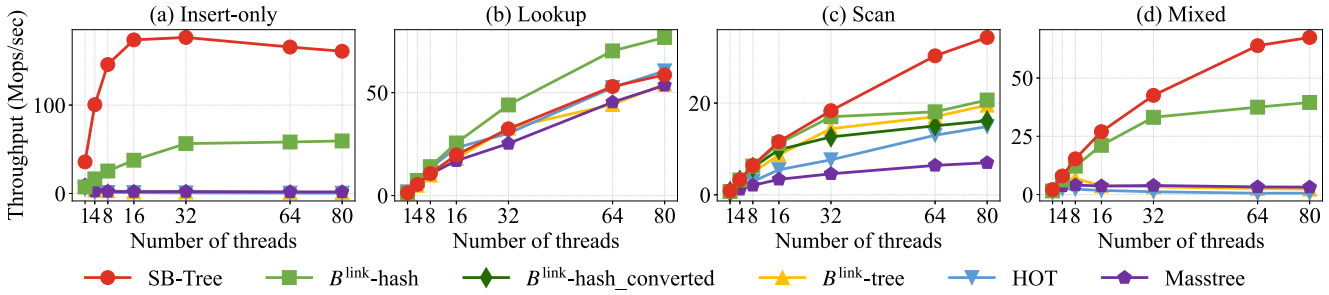

FIGURE 6. Throughput of in-memory indexes across (a) Insert-only, (b) Lookup, (c) Scan, and (d) Mixed workloads, showing SB-Tree's superior scalability in insert-heavy scenarios.

sensors. Both the key and value are composed of 8-byte integers. The key is composed of a 6-byte timestamp followed by a 2-byte sensor ID. Before inserting a key into the index, the timestamp is captured using the RDTSC (Read Time-Stamp Counter) instruction, ensuring that arriving keys are monotonically increasing. In our experiments, we measure the performance of 1 billion operations after building index structure using 1 billion key-value pairs except for the insert-only workload. The lookup and scan operation select the keys uniformly at random. The mixed workload follows the configuration used in previous work [19], consisting of 50% inserts, 30% long scans, 10% short scans, and 10% lookups. A long scan queries 10 to 100 values, while a short scan queries 5 to 10 values.

B. SCALABILITY EVALUATION

1) INSERT-ONLY

Figure 6 (a) shows the throughput of the insert-only workload with a varying number of threads. The keys in this workload are monotonically increasing. SB-Tree outperforms the other indexes by up to 2.7× compared to B^{link}-Hash with 80 threads. SB-Tree provides a segmented block with per-thread data blocks, which provides a dedicated per-thread data block for each thread, so there is no contention among threads. Furthermore, SB-Tree reduces system call overhead by using its block allocator. During the 1-billion insert workload with 80 threads, a total of 36,646 conversions occurred, averaging approximately 458 conversions per thread. We also observed that data layer traversals occurred in approximately 0.02% of inserts. On average, these traversals scanned 154 data blocks before locating the target key. Importantly, since our data blocks are grouped into segmented blocks, this range typically corresponds to scanning only 1–2 segmented blocks. B^{link}-Hash shows good scalability in throughput because its leaf node is implemented as a hash node, allowing concurrent insert operations to be distributed evenly across multiple hash table buckets within the hash node. In contrast, other indexes suffer from performance bottlenecks because the monotonically increasing keys are inserted into the rightmost leaf node of the indexes. Since most indexes do not allow more than one insert thread for each node, other insert threads must restart the tree traversal to find the target leaf node for insertion, leading to further performance degradation.

2) LOOKUP

In Figure 6 (b), SB-Tree shows relatively lower lookup performance than B^{link}-Hash, since the leaf nodes in B^{link}-Hash employ a hash table structure, referred to as hash nodes. Also, the hash node can store a large number of key-value pairs, as it has multiple buckets of large size. This reduces the height of the tree structure. Other indexes show similar lookup performance to SB-Tree. Since the timestamp information is stored as an integer value, the B+-tree variants show comparable performance to trie-based tree structure. HOT shows the best performance as it has a trie structure and low tree height. B^{link}-Hash also has similar tree height, but it has more key comparisons.

3) SCAN

Figure 6 (c) shows the scan performance of in-memory indexes. SB-Tree achieves superior throughput compared to other indexes. This is because SB-Tree's data blocks can store more entries than the leaf nodes of other in-memory indexes. The large-sized data block is beneficial not only for the prefetcher but also for reducing the overall tree size. In addition, SB-Tree separates keys and values into different arrays, enabling more efficient value retrieval. The N-ary search table in SB-Tree also accelerates the lookup of the minimum value in the scan range.

For the scan evaluation, we use two variants of B^{link}-Hash: one in which hash nodes are converted into tree nodes during scan operations (baseline), and another in which all hash nodes are converted before the scan operations (pre-converted hash nodes). In our experiments, the configuration with pre-converted hash nodes achieves only 0.86× the average performance of the baseline configuration. This performance degradation is due to the fact that converting all hash nodes in advance results in a taller tree, which increases the number of memory references during traversal. B^{link}-tree shows better performance than trie-based trees such as HOT and Masstree. Masstree shows the worst performance because it has higher

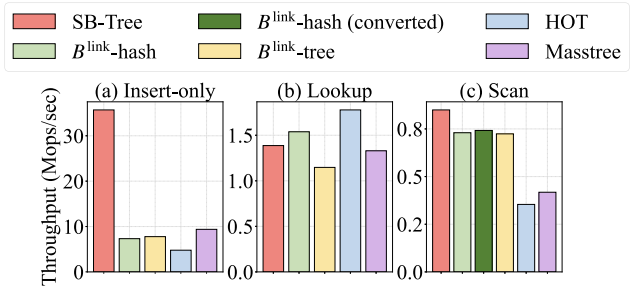

FIGURE 7. Single-threaded throughput in time series workload.

tree height than other indexes. Also, Masstree requires more node visits.

4) MIXED

Figure 6 (d) represents the performance for the mixed workload. SB-Tree and B^{link}-Hash exhibit scalable performance as the number of threads increases. Since 50% of the mixed workload consists of insert operations, other indexes suffer from performance bottleneck. The keys are sorted and monotonically increasing, causing insert operations to be skewed toward a specific node.

C. SINGLE-THREADED PERFORMANCE

1) INSERT-ONLY

In Figure 7 (a), SB-Tree achieves up to 7× better performance than other indexes because SB-Tree directly accesses the segmented block via a shortcut. SB-Tree also achieves a high cache hit rate, leading to fast insertion performance. Other indexes show lower performance because they need to traverse the tree to search for the appropriate node for insertion and perform structural modification operation. B^{link}-Hash shows similar performance to B^{link}-tree due to its split operation. In B^{link}-Hash, the split operation first sorts the key-value pairs and divides them into two nodes. To reduce the performance overhead, it proposes median approximation and a lazy split operation. HOT shows lower throughput than other indexes because it incurs more structural modification operations to maintain the minimum tree height. B^{link}-tree shows comparable performance to B^{link}-Hash although it has higher tree height, because B^{link}-tree's split operation is simpler than that of B^{link}-Hash. Masstree shows better performance than B^{link}-tree and B^{link}-Hash since it uses a combination of a trie and B+-tree structure.

2) LOOKUP

Figure 7 (b) illustrates the lookup performance of indexes in a single-threaded environment. SB-Tree shows comparable lookup performance to other indexes even though it has large-sized data blocks. Since SB-Tree builds the search layer in a compact manner, each search block is fully utilized. B^{link}-Hash shows 1.1× better performance than SB-Tree because it has hash nodes. Moreover, it has a lower tree height as the hash table-based leaf node can store more key-value pairs. HOT outperforms other indexes by 1.33× because it dynamically adjusts each node's span based on the data distribution, maintaining minimal tree height to maximize cache efficiency. B^{link}-tree shows lower performance than B^{link}-Hash and SB-Tree because it has higher tree height. Masstree also has the same tree height as B^{link}-tree but it shows better performance thanks to its trie structure.

3) SCAN

In Figure 7 (c), the variants of B+-tree, SB-Tree, B^{link}-Hash, and B^{link}-tree outperform trie-based indexes, HOT and Masstree. Since the variants of B+-tree store similar key-value pairs in a sequential manner, they can take advantage of CPU cache efficiency. SB-Tree shows superior performance because it has lower tree height and large, sorted data blocks. SB-Tree also separates keys and values to make data blocks more CPU cache and prefetch friendly.

For fair comparison, we also measure the performance of both versions of B^{link}-Hash: (1) when hash nodes are dynamically converted into tree nodes during scan operations, and (2) after all hash nodes have been converted into tree nodes. In the latter case, the tree height is increased. In our experiment, the dynamic conversion case shows 0.98× the performance of the other because all hash node conversions are handled by a single thread.

D. INSERT PERFORMANCE WITH DELAYED DATA

In this experiment, we measure the performance of indexes with delayed data. To simulate delayed data, we first build the indexes by inserting 1 billion timestamp keys and generate delayed data by uniformly sampling the keys used during the index-building phase. This approach ensures that the delayed data is not skewed and is evenly distributed across multiple nodes. The uniform sampling method used to generate delayed data represents a worst-case scenario in terms of locality, as it allows keys to be inserted far from the current insertion position. While this does not accurately reflect real-world delay distributions, it provides a more consistent and fair baseline for comparing different index structures.

Figure 8 presents the index performance with delayed data. SB-Tree shows decreasing performance as the proportion of delayed data increases. In our experiment, SB-Tree shows up to 0.79× decreases in the insert-only workload and a 0.4× decrease in the mixed workload. This is because SB-Tree is optimized for monotonically increasing keys.

In particular, the shortcut in SB-Tree becomes inefficient when inserting delayed data, as it follows conventional B+-tree traversal. Furthermore, insert operations to the same data block are serialized. In our experiment, SB-Tree begins to show lower performance than B^{link}-Hash when the proportion of delayed data exceeds 30%. Note that the data distribution will be similar to a random distribution when the proportion of the delayed data is increasing. B^{link}-Hash distributes insertions across multiple buckets within a hash node, enabling concurrent insertions. This maintains scalable performance

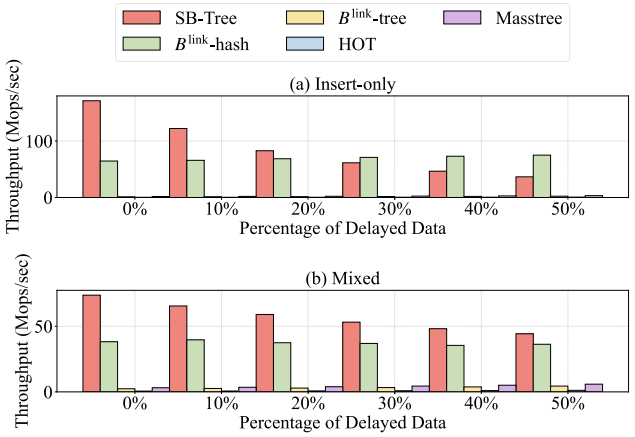

FIGURE 8. Throughput under delayed data (0–50%) with 80 threads, showing SB-Tree performance degradation beyond 30% due to reduced shortcut effectiveness.

even in the insert-only workload with delayed data. Other indexes show better performance as the proportion of delayed data increases in both the insert-only and mixed workloads. Specifically, in both workloads, B^{link}-Hash improves by an average of 1.03×, B^{link}-tree by 1.13×, HOT by 1.14×, and Masstree by 1.31×. However, B^{link}-tree, HOT, and Masstree still show low performance because they still suffer from skewed insertions.

E. PERFORMANCE BREAKDOWN

We analyze the performance of each operation in SB-Tree while varying the number of threads as shown in Figure 9. "Search Layer" and "Data Layer" in Figure 9 represent the traversal time for each respective layer. In insert operations, most of the time is consumed by the insert operation using the shortcut mechanism, as shown in Figure 9 (a). The traversal time for the data layer increases as the number of threads increases, because the search layer is asynchronously updated with a dedicated thread. As the number of threads is increasing, the insertions to the search layer can be delayed. The time for the conversion operation from per-thread data blocks to the data blocks is reduced as the number of threads increases.

In lookup and scan operations, the ratio of time spent on each component remains constant, as shown in Figure 9 (b) and (c). The lookup operation spends about 40% of the time for traversing the search layer and about 27% of the time searching the N-ary search table. In scan operations, the traversal time for data layer accounts for 30% of the total execution time.

F. LATENCY ANALYSIS

Figure 10 illustrates the cumulative distribution function (CDF) of latency for each operation under the time series workload.

In the insert-only workload, SB-Tree achieves remarkably low latency for the 99.9th percentile because SB-Tree utilizes

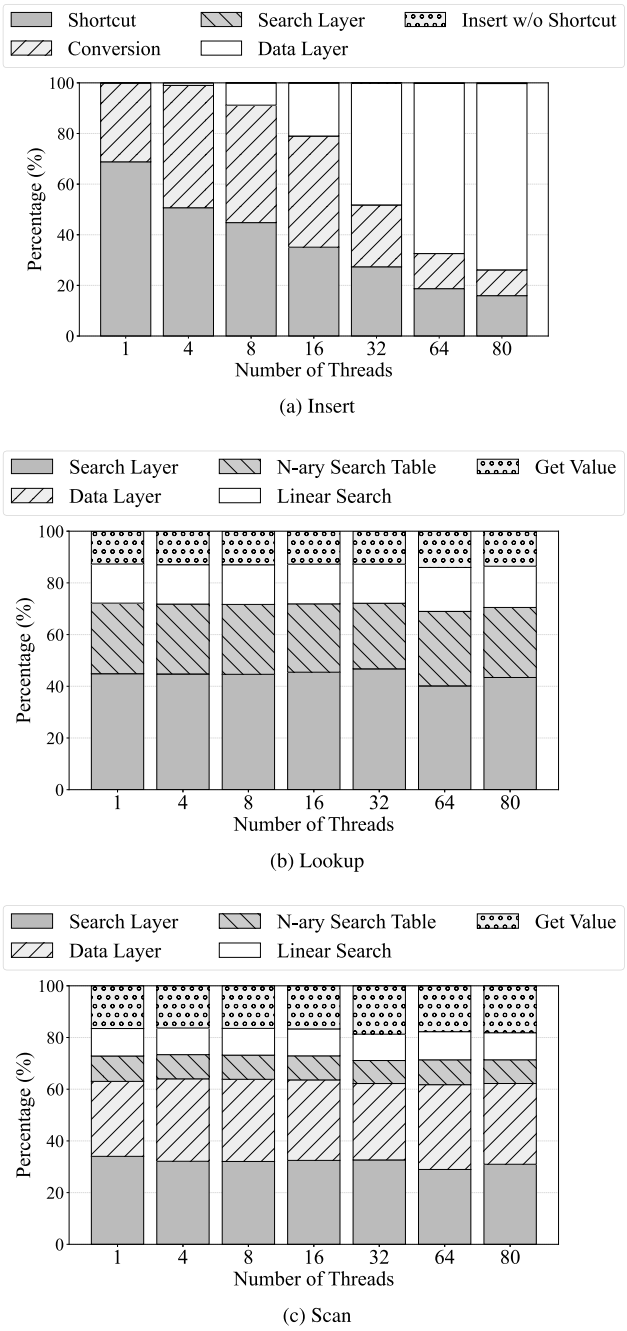

FIGURE 9. Execution time breakdown of SB-Tree: (a) Insert dominated by shortcut, (b) Lookup with N-ary search and data traversal, (c) Scan highlighting data layer efficiency.

shortcuts to directly access segmented blocks. When a per-thread data block has already been allocated within the corresponding segmented block, the insert is executed immediately. This mechanism leads to a high cache hit ratio and low latency. However, SB-Tree exhibits higher 99.99th percentile latency compared to other indexes. This is due to a single thread being responsible for converting a segmented block into data blocks, which increases tail latency. However, in the case of inserts, SB-Tree mitigates

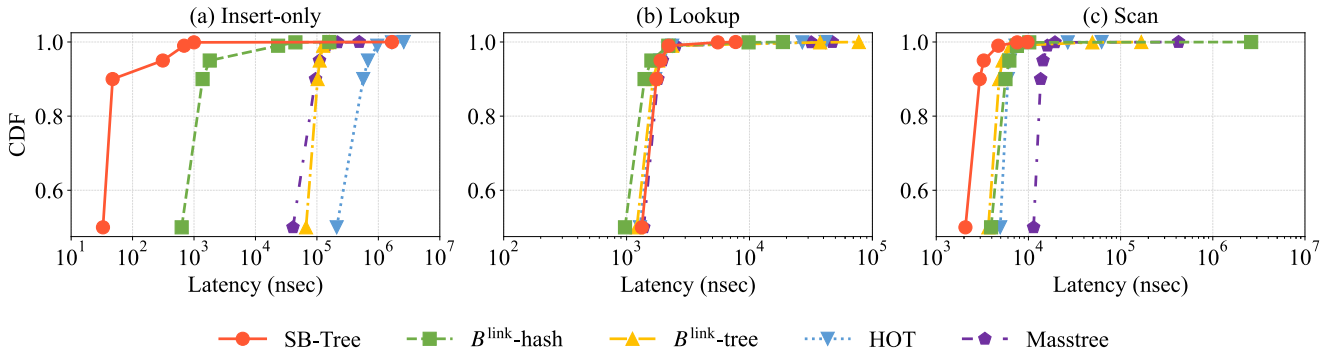

FIGURE 10. Latency CDFs at 80 threads for inserts, lookups, and scans.

blocking by allocating additional segmented blocks and by provisionally over-allocating per-thread data blocks. This allows other threads to continue their insert operations with minimal interference, even when one thread is performing a conversion.

B^{link}-tree and Masstree show high latency because they are not optimized for time series data. The monotonically increasing keys decrease the node utilization. HOT shows the highest latency because of its structural modification operation to maintain the minimum tree height.

In the lookup workload, B^{link}-Hash achieves the lowest latency for the 50th, 90th, and 95th percentiles thanks to its hash nodes. The hash nodes enable efficient key lookups using hash functions and fingerprints within the nodes, resulting in fast average-case performance. However, SB-Tree shows superior tail latency performance, achieving 1.78× and 2.42× lower latency than B^{link}-Hash at the 99.9th and 99.99th percentiles, respectively. This is because SB-Tree's structural modification operation is simpler than that of B^{link}-Hash and is done asynchronously. Masstree shows high latency because it has the highest tree height.

In the scan workload, SB-Tree achieves the lowest latency for the 99.9th percentile. B^{link}-Hash suffers from an extremely high tail latency, reaching 266.61× that of SB-Tree. When a thread encounters a hash node during the scan operation, the thread must convert the node into a tree node by itself. This conversion significantly increases the tail latency for scan operations in B^{link}-Hash. Trie-based indexes such as HOT and Masstree show high latencies in the scan workload because of their unbalanced and non-contiguous structure.

G. MEMORY USAGE

Figure 11 shows the memory usage of each index. The measurement was taken after inserting 1 billion key-value pairs from the time series workload. We also present the absolute index sizes in GB in Table 1. While Figure 11 illustrates relative memory usage patterns, Table 1 provides the exact sizes, enabling a more precise comparison.

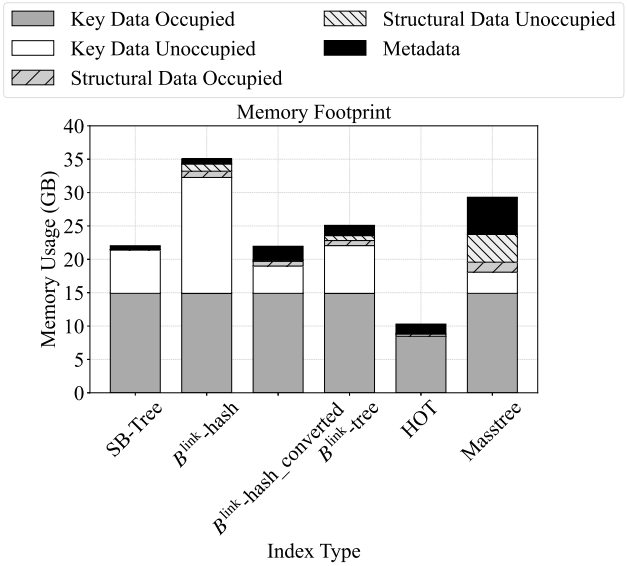

FIGURE 11. Memory usage breakdown after 1B inserts.

Following [19], we classify memory usage into five parts to analyze SB-Tree efficiency: key data occupied, key data unoccupied, structural data occupied, structural data unoccupied, metadata. (1) structural data refers to the keys and pointers stored in internal nodes. (2) key data represents the keys and values stored in leaf nodes. (3) metadata includes additional per-node information beyond keys and pointers. Unoccupied refers to the portion of memory that has been allocated but not used, while occupied indicates the portion that is actively in use. SB-Tree shows an average of 0.8× lower memory usage compared to variants of B+-tree including B^{link}-Hash. The largest reduction appears in the structural data, particularly due to the memory usage of SB-Tree's search layer. Compared to other B+-tree-based indexes, SB-Tree's structural data occupied is only about 0.11×, and its structural data unoccupied is 0.004×. This is because the search layer is constructed by inserting data in a bottom-up manner, fully filling each tree node, resulting in

TABLE 1. Memory usage breakdown (absolute size in GB and percentage of total) after inserting 1B key–value pairs.

Index	Key Occ.	Key Unocc.	Struct. Occ.	Struct. Unocc.	Metadata
SB-Tree	14.90 (67.62%)	6.48 (29.41%)	0.089 (0.40%)	0.0015 (0.01%)	0.563 (2.56%)
Blink-Hash	14.90 (42.50%)	17.37 (49.55%)	0.935 (2.67%)	1.088 (3.10%)	0.763 (2.18%)
Blink-Hash (conv.)	14.90 (67.86%)	4.09 (18.63%)	0.693 (3.16%)	0.179 (0.81%)	2.094 (9.54%)
Blink-Tree	14.90 (59.39%)	7.15 (28.50%)	0.760 (3.03%)	0.735 (2.93%)	1.544 (6.15%)
HOT	8.47 (82.19%)	0.00 (0.00%)	0.367 (3.56%)	0.000 (0.00%)	1.468 (14.25%)
Masstree	14.90 (50.86%)	3.17 (10.82%)	1.497 (5.11%)	4.160 (14.20%)	5.571 (19.01%)

a compact tree structure. Moreover, since data blocks in the data layer can store a large number of keys, the number of keys required in the search layer is also reduced. B^{link}-Hash shows the highest memory usage among all indexes due to its large hash node structure at the leaf level. Its key data unoccupied alone accounts for approximately 49.57% of the total memory usage and is 2.43× larger than that of B^{link}-tree, because each hash node contains multiple buckets, many of which remain partially empty. After converting all hash nodes into tree nodes, the memory usage becomes comparable to B^{link}-tree. In this case, B^{link}-Hash fills tree nodes up to 80% to enhance memory utilization, resulting in its key data unoccupied being 0.57× that of B^{link}-tree. HOT achieves the lowest memory usage among all indexes. As a trie-based structure, its key data occupied is only 0.57× that of other indexes. By dynamically resizing nodes based on the number of stored entries, it maintains a minimal tree height, stores data compactly, and uses memory efficiently. Masstree manages two types of nodes, interior and border nodes, due to its trie structure composed of multiple B+-trees. Both types are variants of B+-tree nodes. Interior nodes serve as typical internal nodes, while Border nodes store key-value entries and can also point to lower levels, effectively functioning as both leaf and internal nodes. As a result, Masstree's Structural data occupied is on average 2.55× higher than other indexes. Additionally, Border nodes manage lots of metadata such as key lengths, suffixes, and permutation data for sorted access, leading to metadata usage that is 4.91× higher than other indexes.

VI. DISCUSSION

A. LIMITATIONS OF SB-TREE

1) STALE SEARCH LAYER

Under high-contention workloads, SB-Tree can suffer from stale search layers, which degrade insert performance. As the number of threads increases, the conversion process may take longer, and the resulting increase in data blocks can introduce additional overhead during data layer traversal.

2) SORTING DURING CONVERSION

When the number of threads increases, the number of keys that need to be sorted during conversion also grows. Currently, SB-Tree employs a single thread for sorting, based on the assumption that each worker thread handles one index operation independently. This may lead to high tail-latency, and parallel merge-and-sort support could alleviate this issue. However, it would require allocating additional dedicated threads for conversion. While this is technically feasible, we chose to avoid such parallelism to maintain a simple per-thread execution model and minimize thread coordination overhead.

B. GENERALITY OF SB-TREE

1) RANDOM KEYS AND VARIABLE LENGTH KEYS

SB-Tree is optimized for time series workloads where keys are mostly increasing. Under non-monotonic workloads, SB-Tree falls back to the standard B+-tree algorithm. Similarly, for variable-length keys, SB-Tree inherits the original B+-tree's insertion and lookup algorithm, and thus may exhibit suboptimal performance in such cases.

2) SB-TREE WITH PERSISTENT MEMORY TECHNOLOGIES

SB-Tree is designed for in-memory time series databases and, as such, does not provide crash consistency; the index must be rebuilt after a system crash. However, when used with persistent memory technologies such as NVDIMM-N [34] and CMM-H [35], SB-Tree can be consistent across system crashes and used efficiently without requiring a full rebuild. Because time series workloads typically involve monotonically increasing keys, the keys are appended to data blocks rather than inserted randomly. This eliminates the need to re-sort existing data during writes, resulting in minimal persistence overhead.

3) APPLICABILITY OF SB-TREE

Index structures play an important role in modern systems. Thus, SB-Tree can be employed as an index structure in various environments. For example, SB-Tree can be applied to sensor data management in edge devices [36] and to index structures of embedded database systems [37]. In addition, it can be utilized in key-value stores such as RocksDB and LevelDB [38], [39].

VII. CONCLUSION

In this paper, we proposed SB-Tree, an efficient in-memory index structure tailored for time series workloads. SB-Tree

was designed to meet the specific requirements of time series workloads, supporting both high-throughput insert operations and efficient range queries. To achieve scalable and fast insertions, SB-Tree introduced shortcut mechanisms, segmented blocks, and per-thread data blocks that minimized contention and enable scalable insertions. Furthermore, SB-Tree adopted a block allocator to reduce system call overhead. For range query performance, SB-Tree used data blocks optimized for sequential scans. It also employed an N-ary search table and a cache-optimized B+-tree based search layer to improve key lookup efficiency. Through extensive evaluations, we demonstrated that SB-Tree achieved high performance in time series workloads, excelling in both insertion speed and range query throughput while maintaining memory efficiency.

ACKNOWLEDGMENT

(Christine Euna Jung and Jaesang Hwang are co-first authors.)

REFERENCES

[1] Y. Sun, T. Chen, Q. V. H. Nguyen, and H. Yin, "TinyAD: Memory-efficient anomaly detection for time-series data in industrial IoT," *IEEE Trans. Ind. Informat.*, vol. 20, no. 1, pp. 824–834, Jan. 2024.

[2] Y. Liu, Y. Zhou, K. Yang, and X. Wang, "Unsupervised deep learning for IoT time series," *IEEE Internet Things J.*, vol. 10, no. 16, pp. 14285–14306, Aug. 2023.

[3] E. C. P. Neto, S. Dadkhah, R. Ferreira, A. Zohourian, R. Lu, and A. A. Ghorbani, "CICIoT2023: A real-time dataset and benchmark for large-scale attacks in IoT environment," *Sensors*, vol. 23, no. 13, p. 5941, Jun. 2023.

[4] M. A. Farahani, M. R. McCormick, R. Gianinny, F. Hudacheck, R. Harik, Z. Liu, and T. Wuest, "Time-series pattern recognition in smart manufacturing systems: A literature review and ontology," *J. Manuf. Syst.*, vol. 69, pp. 208–241, Aug. 2023.

[5] E. Ghaderpour, S. D. Pagiatakis, G. S. Mugnozza, and P. Mazzanti, "On the stochastic significance of peaks in the least-squares wavelet spectrogram and an application in GNSS time series analysis," *Signal Process.*, vol. 223, Oct. 2024, Art. no. 109581.

[6] J. Koumar, K. Hynek, T. Čejka, and P. Šiška, "CESNET-TimeSeries24: Time series dataset for network traffic anomaly detection and forecasting," *Sci. Data*, vol. 12, no. 1, p. 338, Feb. 2025, doi: 10.1038/s41597-025-04603-x.

[7] J. Koumar, K. Hynek, J. Pešek, and T. Čejka, "NetTiSA: Extended IP flow with time-series features for universal bandwidth-constrained high-speed network traffic classification," *Comput. Netw.*, vol. 240, Feb. 2024, Art. no. 110147.

[8] N. Sivaroopan, D. Bandara, C. Madarasingha, G. Jourjon, A. P. Jayasumana, and K. Thilakarathna, "Netdiffus: Network traffic generation by diffusion models through time-series imaging," *Comput. Netw.*, vol. 251, Sep. 2024, Art. no. 110616.

[9] L. C. de Jesus, F. Fernández-Navarro, and M. Carbonero-Ruz, "Enhancing financial time series forecasting through topological data analysis," *Neural Comput. Appl.*, vol. 37, no. 9, pp. 6527–6545, Mar. 2025.

[10] Y. Hu, Y. Li, P. Liu, Y. Zhu, N. Li, T. Dai, S.-t. Xia, D. Cheng, and C. Jiang, "FinTSB: A comprehensive and practical benchmark for financial time series forecasting," 2025, *arXiv:2502.18834*.

[11] L. Guo, "BiLSTM based temporal modeling for financial risk forecasting using multivariate time series data," in *Proc. 3rd Int. Conf. Data Sci. Inf. Syst. (ICDSIS)*, May 2025, pp. 1–6.

[12] D. Comer, "Ubiquitous B-tree," *ACM Comput. Surveys*, vol. 11, no. 2, pp. 121–137, Jun. 1979.

[13] J. J. Levandoski, D. B. Lomet, and S. Sengupta, "The bw-tree: A B-tree for new hardware platforms," in *Proc. IEEE 29th Int. Conf. Data Eng. (ICDE)*, Apr. 2013, pp. 302–313.

[14] P. L. Lehman and S. B. Yao, "Efficient locking for concurrent operations on B-trees," *ACM Trans. Database Syst.*, vol. 6, no. 4, pp. 650–670, Dec. 1981, doi: 10.1145/319628.319663.

[15] M. A. Bender, M. Farach-Colton, W. Jannen, R. Johnson, B. C. Kuszmaul, D. E. Porter, J. Yuan, and Y. Zhan, "An introduction to be-trees and write-optimization," *Login, Mag.*, vol. 40, no. 5, pp. 1–7, 2015.

[16] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: ARTful indexing for main-memory databases," in *Proc. IEEE 29th Int. Conf. Data Eng. (ICDE)*, Apr. 2013, pp. 38–49.

[17] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, Bern, Bern, Switzerland, Apr. 2012, pp. 183–196.

[18] R. Binna, E. Zangerle, M. Pichl, G. Specht, and V. Leis, "HOT: A height optimized trie index for main-memory database systems," in *Proc. Int. Conf. Manage. Data*, May 2018, pp. 521–534, doi: 10.1145/3183713.3196896.

[19] H. Cha, X. Hao, T. Wang, H. Zhang, A. Akella, and X. Yu, "B link -hash: An adaptive hybrid index for in-memory time-series databases," *Proc. VLDB Endowment*, vol. 16, no. 6, pp. 1235–1248, Feb. 2023.

[20] S. K. Jensen, T. B. Pedersen, and C. Thomsen, "Time series management systems: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 11, pp. 2581–2600, Nov. 2017.

[21] Y. Yang, Q. Cao, and H. Jiang, "EdgeDB: An efficient time-series database for edge computing," *IEEE Access*, vol. 7, pp. 142295–142307, 2019.

[22] W. Weiss, V. J. E. Jiménez, and H. Zeiner, "Dynamic buffer sizing for out-of-order event compensation for time-sensitive applications," *ACM Trans. Sensor Netw.*, vol. 17, no. 1, pp. 1–23, Sep. 2020, doi: 10.1145/3410403.

[23] C. Wang, J. Qiao, X. Huang, S. Song, H. Hou, T. Jiang, L. Rui, J. Wang, and J. Sun, "Apache IoTDB: A time series database for IoT applications," *Proc. ACM Manage. Data*, vol. 1, no. 2, pp. 1–27, Jun. 2023.

[24] A. Khelifati, M. Khayati, A. Dignös, D. Difallah, and P. Cudré-Mauroux, "TSM-bench: Benchmarking time series database systems for monitoring applications," *Proc. VLDB Endowment*, vol. 16, no. 11, pp. 3363–3376, Jul. 2023, doi: 10.14778/3611479.3611532.

[25] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Inf.*, vol. 33, no. 4, pp. 351–385, Jun. 1996.

[26] influxdata. *InfluxDB: Real-time Visibility Into Stacks, Sensors and Systems*. [Online]. Available: https://www.influxdata.com/

[27] Z. Wang and Z. Shao, "ForestTI: A scalable inverted-index-oriented time-series management system with flexible memory efficiency," *Proc. ACM Manage. Data*, vol. 1, no. 2, pp. 1–25, Jun. 2023, doi: 10.1145/3589260.

[28] *TigerData*. Accessed: Jun. 25, 2025. [Online]. Available: https://www.tigerdata.com/

[29] D. R. Morrison, "PATRICIA—Practical algorithm to retrieve information coded in alphanumeric," *J. ACM*, vol. 15, no. 4, pp. 514–534, Oct. 1968, doi: 10.1145/321479.321481.

[30] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen, "Reducing the storage overhead of main-memory OLTP databases with hybrid indexes," in *Proc. Int. Conf. Manage. Data*, Jun. 2016, pp. 1567–1581, doi: 10.1145/2882903.2915222.

[31] V. Leis, F. Scheibner, A. Kemper, and T. Neumann, "The ART of practical synchronization," in *Proc. 12th Int. Workshop Data Manage. New Hardw.*, Jun. 2016, pp. 1–8, doi: 10.1145/2933349.2933352.

[32] A. Mathew and C. Min, "HydraList: A scalable in-memory index using asynchronous updates and partial replication," in *Proc. 46th Int. Conf. Very Large Data Bases (VLDB)*, vol. 13, Tokyo, Japan, Aug. 2020, pp. 1332–1345.

[33] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo, "SuRF: Practical range query filtering with fast succinct tries," in *Proc. Int. Conf. Manage. Data*, Houston, TX, USA, May 2018, pp. 323–336.

[34] *Ddr4 Sdram Non-volatile Dimm (NVDIMM-N)*, document JESD245A, JEDEC Solid State Technology Association, 2018. [Online]. Available: https://www.jedec.org/standards-documents/docs/jesd245a

[35] J. Zeng, S. Pei, D. Zhang, Y. Zhou, A. Beygi, X. Yao, R. Kachare, T. Zhang, Z. Li, M. Nguyen, R. Pitchumani, Y. Soek Ki, and C. Jung, "Performance characterizations and usage guidelines of Samsung CXL memory module hybrid prototype," 2025, *arXiv:2503.22017*.

[36] T. Park, H. Lee, C. E. Jung, W.-H. Kim, and H.-W. Jin, "LION: A learned index for on-device sensor data management," *IEEE Embedded Syst. Lett.*, early access, Jun. 19, 2025, doi: 10.1109/LES.2025.3580080.

[37] *SQLite*. Accessed: Jun. 25, 2025. [Online]. Available: http://www.sqlite.org/
[38] Facebook. *RocksDB*. Accessed: Jun. 25, 2025. [Online]. Available: http://rocksdb.org/
[39] Google. *LevelDB*. Accessed: Jun. 25, 2025. [Online]. Available: http://leveldb.org/

CHRISTINE EUNA JUNG received the B.E. degree from Konkuk University, in 2024. She is currently at Toss Payments as a Server Developer. Her research interests include time series database systems and distributed systems.

JAESANG HWANG is currently pursuing the B.E. degree with Konkuk University. He is at Dnotitia as a Storage System Engineer, developing agent clients for retrieval-augmented generation services. His research interests include computer systems and database systems.

YEDAM NA received the B.E. degree from Konkuk University, in 2025, where she is currently pursuing the M.S. degree in computer science and engineering. Her research interests include distributed systems and database systems.

HAENA LEE received the B.E. degree from Konkuk University, in 2025, where she is currently pursuing the M.S. degree in computer science and engineering. Her research interests include vector database systems and time series database systems.

WOOK-HEE KIM (Member, IEEE) received the B.S. and Ph.D. degrees from Ulsan National Institute of Science and Technology, in 2013 and 2019, respectively. He was a Postdoctoral Associate at Virginia Tech and Sungkyunkwan University. He is currently an Assistant Professor with the Department of Computer Science and Engineering, Konkuk University. His research interests include database systems and storage systems.