B^{link}-hash: An Adaptive Hybrid Index for In-Memory Time-Series Databases

Hokeun Cha
University of Wisconsin-Madison
hcha@cs.wisc.edu

Xiangpeng Hao
University of Wisconsin-Madison
xiangpeng.hao@wisc.edu

Tianzheng Wang
Simon Fraser University
tzwang@sfu.ca

Huanchen Zhang
Tsinghua University
huanchen@tsinghua.edu.cn

Aditya Akella
University of Texas at Austin
akella@cs.utexas.edu

Xiangyao Yu
University of Wisconsin-Madison
yxy@cs.wisc.edu

ABSTRACT

High-speed data ingestion is critical in time-series workloads that are driven by the growth of Internet of Things (IoT) applications. We observe that traditional tree-based indexes encounter severe scalability bottlenecks for time-series workloads that insert monotonically increasing timestamp keys into an index; all insertions go to a small memory region that sees extremely high contention.

In this work, we present a new index design, B^{link}-hash, that enhances a tree-based index with hash leaf nodes to mitigate the contention of monotonic insertions — insertions go to random locations within a hash node (which is much larger than a B+-tree node) to reduce conflicts. We develop further optimizations (median approximation and lazy split) to accelerate hash node splits. We also develop structure adaptation optimizations to dynamically convert a hash node to B+-tree nodes for good scan performance. Our evaluation shows that B^{link}-hash achieves up to 91.3× higher throughput than conventional indexes in a time-series workload that monotonically inserts timestamps into an index, while showing comparable scan performance to a well-optimized B+-tree.

PVLDB Reference Format:
Hokeun Cha, Xiangpeng Hao, Tianzheng Wang, Huanchen Zhang, Aditya Akella, and Xiangyao Yu. B^{link}-hash: An Adaptive Hybrid Index for In-Memory Time-Series Databases. PVLDB, 16(6): 1235 - 1248, 2023.
doi:10.14778/3583140.3583143

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/chahk0129/Blink-hash.

1 INTRODUCTION

High-speed data ingestion and query processing are gaining importance driven by the proliferation of Internet of Things (IoT) devices such as sensors, RFID readers, health care devices, etc. In these application scenarios, timestamped data are constantly and rapidly generated at a rate of millions of events per second [21]. This large volume of data requires not only an efficient data storage solution but also linearly scaling performance so that the speed of data processing can keep up with that of data generation [6, 27].

While many aspects of traditional database systems have been redesigned to meet the requirements of high-speed data ingestion and efficient query processing [1, 2, 5, 21, 44, 49], indexes have not received sufficient attention. Although various techniques have been proposed to improve index performance regarding cache efficiency [7, 45, 46], space efficiency [9, 11, 12, 32, 55], and concurrency control [15, 23, 31, 33, 34, 39], existing solutions do not target the new challenges that appear in time-series workloads.

The characteristics of time-series workloads are different from those of generic OLTP workloads. In particular, keys are typically monotonically increasing timestamps. This property makes existing tree-based indexes sub-optimal in time-series workloads. When using the timestamps as the index keys, insertions will be highly skewed towards the rightmost leaf nodes in a tree, creating extremely high contention and severe scalability bottleneck.

While monotonic insertions do not scale in a tree-based index, they can scale perfectly in a hash index, which distributes insertions randomly to different memory regions. However, a hash index does not support range scans, and thus cannot be used in many practical workloads. Our key insight is to combine a tree-based index with a hash index to achieve the best of both worlds while addressing their limitations. In particular, we replace leaf nodes of a B+-tree index with hash nodes if those leaf nodes experience high monotonic insertion traffic to mitigate the insertion hotspot.

We present a new in-memory index design, B^{link}-hash, that incorporates the above insights. B^{link}-hash enhances a B^{link}-tree [31] with hash leaf nodes to achieve scalability for monotonic insertions while maintaining good performance on conventional workloads. A hash node is considerably larger than a B+-tree node to distribute the insertions to different memory regions. Data is randomly distributed within a hash node, but is sorted between consecutive hash nodes like in a B+-tree.

Replacing B+-tree nodes with hash nodes introduces several key challenges. Since a hash node is larger than a B+-tree node, splitting a hash node leads to a longer critical section which can become a new performance bottleneck. We develop two optimizations to address this issue. The first optimization is *median approximation* which calculates an approximate, instead of a precise, median key for a split operation based on a sampled subset of keys; it significantly reduces the number of accessed keys for median selection with minimal imbalance in the trees. The second optimization is *lazy split* which reduces the critical section of splitting a hash node

Proceedings of the VLDB Endowment, Vol. 16, No. 6 ISSN 2150-8097.
doi:10.14778/3583140.3583143

by deferring the actual data movement to subsequent threads accessing the corresponding buckets; this breaks down the expensive per-node critical section into multiple cheaper per-bucket critical sections, thus mitigating the scalability bottleneck.

Another challenge in B^{link}-hash is to support efficient scans that are inherently difficult with a hash index. We develop a third optimization called *structure adaptation* to dynamically convert a hash node to B+-tree nodes when a hash node is no longer insertion-intensive and becomes scan-intensive. Eventually after a workload runs for sufficiently long, a B^{link}-hash tree will converge into a traditional B^{link}-tree to fully leverage its features.

We evaluate the performance of different tree-based indexes in a time-series workload with monotonic insertions and observed that B^{link}-hash is the only design that scales linearly, achieving 91.3× speedup over other indexes at 32 threads. We also evaluate indexes using a YCSB workload (without monotonic insertions) with string keys, and observed that B^{link}-hash either outperforms the other indexes or has very close performance to the best one.

The contributions of this paper are as follows:

- We identify a fundamental limitation of existing tree-based indexes under time-series workloads.
- We introduce B^{link}-hash, a new index design that overcomes the limitation and achieves high performance for both monotonically increasing and scan workloads.
- We comprehensively evaluate B^{link}-hash against representative index designs, and show that B^{link}-hash outperforms those structures in both time-series and generic OLTP workloads.

The rest of this paper is organized as follows. Section 2 presents the background and motivation of the research. We discuss the design of B^{link}-hash and its optimizations in Sections 3 and 4, respectively. Section 5 presents our empirical evaluation results of B^{link}-hash against representative index designs. Section 6 reviews related work, and Section 7 concludes this paper.

2 BACKGROUND AND MOTIVATION

Time-series workloads have become one of the core workloads in database systems. These workloads typically generate a massive volume of real-time data from geographically-dispersed devices [53]. This requires high-speed data ingestion and efficient query processing where traditional indexes deliver sub-optimal performance.

2.1 Limitation of Conventional Indexes

Today, most time-series systems still use conventional B+-trees or skip-lists as their indexes [1, 3, 5, 22, 49]. In this section, we investigate the scalability of representative indexes on a multicore processor using a workload that inserts new data with monotonically increasing keys (i.e., timestamps).

Figure 1 compares the throughput of tree indexes and a hash index in two different workloads (random and monotonic insertion) with a varying number of threads. Random insertion inserts uniformly random keys while monotonic insertion inserts monotonically increasing keys. Each workload consists of 100 million insert operations. We choose four representative tree indexes; Adaptive Radix Tree (ART) [32] for tries, Masstree [38] for hybrid structures, OpenBw-tree [52] for latch-free structures, and an in-memory version of B^{link}-tree [31] for B+-trees. We use an efficient implementation of concurrent cuckoo hashing (libcuckoo) [35] for a hash index evaluation. While both hash index and tree indexes scale with random insertions (Figure 1(a)), tree indexes do not scale at all when the insertion keys are monotonically increasing (Figure 1(b)). This is because insertions are highly skewed to the rightmost nodes — every thread writes into the same memory region, creating extremely high contention. As a result, concurrent operations are serialized, and the system fails to exploit parallelism in hardware [17].

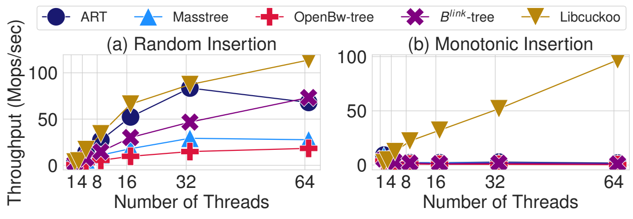


Figure 1: Throughput of tree-based indexes and Libcuckoo under random (a) and monotonic (b) insert operations.

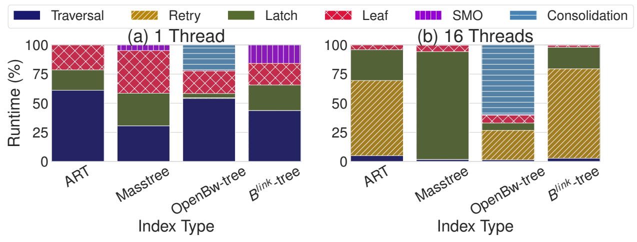


Figure 2: Breakdown of index operations for monotonic insertion with (a) 1 thread and (b) 16 threads.

In contrast, such behavior is not observed in libcuckoo, as shown in Figure 1(b). It is because accesses are distributed to different hash buckets, allowing high scalability regardless of key patterns. A hash index, however, cannot support range scans and is, thus, insufficient in many scenarios including time-series applications.

To better understand the behavior of tree indexes in monotonic insertion, we break down their runtime into six categories.

- *Traversal* denotes tree traversal time.
- *Retry* is the time spent on retried tree traversal due to conflicts.
- *Latch* includes the overhead to acquire an exclusive latch. For OpenBw-tree, we include the time spent on executing compare-and-swap instructions.
- *Leaf* covers the time spent in leaf nodes (inserting new records).
- *SMO* includes time spent on splitting nodes for B+-trees, and adding intermediate nodes and node adaptation for tries.
- *Consolidation* measures the time spent on consolidating delta chains in OpenBw-tree.

Figures 2(a) and (b) show the runtime breakdown with 1 thread and 16 threads, respectively. Most of the time is spent in tree traversal and leaf node accesses with 1 thread. As the number of threads increases to 16, all of them suffer from high contention.

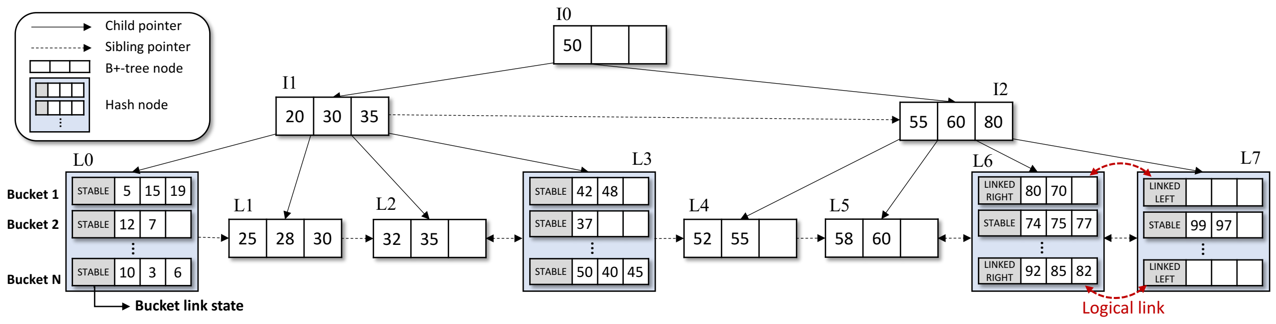


Figure 3: Architecture of B^{link}-hash.

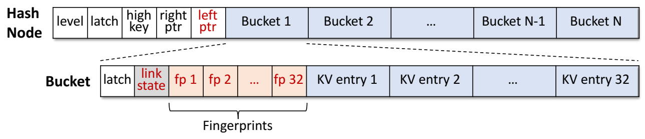


Figure 4: Layout of a hash node — New structures are highlighted in red color.

ART and B^{link}-tree show huge amount of retries, and Masstree spends the most time in latching due to local retries. OpenBw-tree spends the most time in consolidation since all the threads try to consolidate delta chains in rightmost nodes at the same time.

There are two main reasons for this behavior that correlate to each other. First, the high contention itself with skewed accesses results in excessive aborts caused by consecutive splits. Threads try to insert in newly allocated nodes which are filled up shortly, causing another splits. These splits invalidate traversals by forcing them to abort, which causes multiple retries even for a single operation. Second, stressing a single memory location with atomic operations introduces performance bottleneck [17]. To insert a key, all the threads atomically load and try to update the shared memory whether it is a latch-based or latch-free structure. This incurs high cache coherence latency since each update needs to be propagated to the cache in all the accessed cores.

Tree-based indexes do not scale in time-series workloads, but are crucial as they support range scans. We make a key observation that the dilemma can be resolved by combining B+-tree and hashing into a single index structure to get the best of both worlds. To avoid certain B+-tree leaf nodes becoming a hotspot, we can replace them with hash indexes that are sufficiently large to support concurrent insertions from multiple threads. Those hash indexes can be converted back to B+-tree nodes later, when they are no longer a hotspot. In Section 3, we present the insights, challenges, and design of B^{link}-hash that achieves the goal based on these intuitions.

2.2 B^{link}-tree

B^{link}-tree [31] is a classic variant of B+-tree that achieves high concurrency in disk-based systems. It adds a link pointer and a high key to each node, and resolves the limitation of latch crabbing. Latch crabbing releases latch on its parent node if its child node latch is acquired, and thus, frequently touches multiple latches to traverse a tree. The bottom-up latching in B^{link}-tree may affect tree traversal as concurrent split can cause a thread to access an incorrect node. Such an issue is resolved by checking the high key, and the correct node can be found by chasing the link pointers.

We build B^{link}-hash on top of B^{link}-tree due to its simplicity and efficiency in synchronization. B^{link}-tree has been thoroughly studied and noted for its superiority in concurrency in previous work [50, 51], and it has been demonstrated that B^{link}-tree has lightweight concurrency support; for example, an operation requires at most three latches at a time [31]. However, we note that the general idea of B^{link}-hash also applies to other B+-tree variants.

3 BASIC DESIGN OF B^{link}-HASH

B^{link}-hash combines hashing and B+-tree into a single index. A key challenge is to combine them smoothly such that neither correctness nor performance is sacrificed while keeping the design simple. For B^{link}-hash, we decide to keep the B^{link}-tree structure as much as possible and only replace the leaf nodes that experience high insertion traffic with hash nodes to reduce conflicts caused by monotonic insertions. When a hash node is no longer under high insertion traffic and starts to be queried with scans, B^{link}-hash converts the hash nodes back to B+-tree nodes. This makes B^{link}-hash completely compatible with traditional B^{link}-tree — B^{link}-hash will eventually be converted to B^{link}-tree after the monotonic insertion pattern disappears for certain parts of the tree.

Next, we will present the data structure of a B^{link}-hash (Section 3.1) and the operations in a basic (i.e., unoptimized) B^{link}-hash (Section 3.2) and their limitations (Section 3.3). We will discuss the performance optimizations of the basic B^{link}-hash in Section 4.

3.1 Data Structure

Figure 3 describes the overall structure of B^{link}-hash. A leaf node in B^{link}-hash is either a B+-tree node or a hash node. A B+-tree node contains a pointer to its right sibling node (as in B^{link}-tree), while a hash node has pointers to both its left and right sibling nodes. Initially, a leaf node starts as a hash node and will later be converted to B+-tree nodes when it encounters scan queries, following a policy that we will discuss in Section 4.3. A split creates a new leaf node that has the same layout as the original node, e.g., a new hash node is created when splitting a hash node.

Figure 4 illustrates the layout of a hash node. It shares the common metadata with a B+-tree node such as *level*, *latch*, *high key*, and *right sibling pointer*. The level specifies the level of a node in the tree. The latch occupies 8 bytes, consisting of a latch bit and a version number. Note that the latch bit protects only structural changes in a node (e.g., split) but does not implicitly latch buckets in the node. The high key is a fence key that represents the largest key in a node. A hash node additionally contains a *left sibling pointer* to facilitate node adaptation and to reduce synchronization overhead (its detailed purpose is discussed in Sections 4.2 and 4.3).

A hash node has multiple buckets to store data, each of which consists of 32 key-value entries. Each bucket also contains metadata including a latch and *link state*. The bucket latch controls the concurrency at a bucket granularity. The link state is used to parallelize node splits, which will be explained in Section 4.2.

Fingerprints in each bucket are hashed keys [41]. Each fingerprint is 1B and all fingerprints in a bucket combined are 32B. The least-significant bit of a fingerprint indicates the occupancy of the corresponding key-value entry, and the rest of the bits determine if the hashed search key matches the hashed key stored in the bucket. The fingerprint allows search to avoid unnecessary accesses to actual key space by filtering out many accesses that do not match any hash keys. Since false positives may still exist, after a fingerprint hit, a search reads the full key to determine whether the keys indeed match. For high performance, we use SIMD instructions to compare multiple fingerprints in a bucket with a single instruction call.

Various optimization techniques have been studied to maintain good utilization of a hash index such as stashing, distance probing [43], cuckoo displacement [42], and chaining [28]. Among the above, we apply distance probing [43] and double hashing to hash nodes to improve memory efficiency. That is, upon a hash collision, a writer first probes up to a specific distance of adjacent buckets to find an empty entry. If an empty entry is still not found, it picks another bucket using a different hash function, and repeats the probing. Although these optimizations degrade search performance by potentially increasing the amount of key comparisons and accessed buckets, this overhead is mitigated using fingerprints by filtering out unnecessary accesses to actual key space.

3.2 Basic Operations in B^{link}-hash

In this section, we discuss the basics of concurrent operations in B^{link}-hash. B^{link}-hash uses optimistic latch coupling [30] for its concurrency, validating a node version change after reading contents to verify any updates. We focus on leaf operations and internal nodes have similar behavior [31]. We first describe the operations in a hash node followed by the operations in a B+-tree node.

3.2.1 Hash Node Operations.

Each data operation in a hash node is processed at a bucket granularity. In the following, we discuss how read, insert, update, delete, and scan operations work. We omit fingerprints, distance probing, and double hashing for clarity.

Read. A reader first finds a bucket index using a hash function, and reads the version number of the bucket. The read aborts and retries if the bucket latch has already been acquired by another thread. Then, it reads the value of the matching key and validates the version number of the bucket. If the version numbers do not match, another thread must have updated the bucket, so the current reader aborts and retries from the root. Finally, it validates the version number of the node by comparing its current version with the version number obtained during tree traversal. Note that no shared data is modified in a read operation as it simply reads the content.

Insert. A writer does not need to latch the entire hash node for an insert. Instead, only the bucket that is inserted into is latched. The node cannot be updated while the thread is holding a bucket latch since each structural modification operation (SMO) requires the node latch and all the bucket latches. Then, the writer validates the version number of the current node obtained during tree traversal to check if the node has been updated by a concurrent thread. This prevents writers from updating new records in an incorrect location. For example, suppose two threads access the same leaf node. One of them tries to insert a record in a bucket, while the other tries to split the node. If the second thread completes the split before the first thread latches the bucket, the first thread does not know if the node has been updated, and it may insert in an incorrect node. To prevent this, we validate the version number of a node after latching a bucket. Then, new data is written in the bucket. If the insertion fails due to a hash collision, it splits the node.

Update. An update works in a similar way as an insert. The only difference is that a thread changes an old value to a new value for the matching key in the target bucket.

Delete. A delete works in a similar way as an update. Instead of changing a value, it marks the value for a matching key as deleted. Note that any under-utilized hash nodes are not merged, but reclaimed and converted back to B+-tree nodes later by subsequent scanners following a simple heuristic that we discuss in Section 4.3.

Scan. A range scan follows the steps below to ensure reading a consistent snapshot. First, the scanner reads the version number of each bucket, sequentially scans its contents to collect key-value pairs that satisfy the requested key range, and validates the version. Then, it returns the values after sorting the collected pairs. Finally, the node's version number is validated to guarantee the node has not been updated by a concurrent thread.

Scans may span more than one leaf node when the requested range is large. The qualified nodes can be a mixture of B+-tree and hash nodes. In such cases, a scanner traverses to its right sibling node via the link pointer after the version validation of the current node. Similar to B^{link}-tree, in B^{link}-hash, keys in the right sibling node are always larger than keys in the current node. The result records from each leaf node are combined and returned.

Split. When an insert fails due to the lack of empty space, a split is triggered. A new node is pre-allocated before any synchronization to minimize the length of the critical section. Then, the thread tries to acquire a node latch. It succeeds only if the current version of the node matches the version that is obtained during tree traversal. Furthermore, it acquires all the bucket latches. After finding a median key by scanning all the buckets and sorting the collected keys, key-values are migrated to the new node by scanning the buckets again. Then, it checks the existence of the right sibling node. If the sibling is a hash node, it updates its left sibling pointer to point to the new node. Finally, the right sibling pointer in the current node is updated, and the pointer for the new node is returned.

3.2.2 B+-tree Node Operations. Data operations in B+-tree nodes work the same way in the legacy OLC-based B+-tree [15]. Synchronization is done at a node granularity, and the only difference lies in how SMOs are done. The placement of hash nodes adds one extra update for the left sibling pointer in its right sibling node.

3.3 Limitations of the Basic B^{link}-hash Design

While hash nodes improve scalability for monotonic insertions, they also introduce the following new challenges.

First, splitting a hash node can potentially become a new scalability bottleneck. A split requires to obtain a median key to provide a key range to parent nodes, which further requires to scan all the keys and sort. Then, half of the key-value pairs need to be migrated to a new node, and this involves comparing every key in the current node to the median. Both median calculation and data movement happen on the critical path of a split in the basic design of B^{link}-hash. They are expensive computations that can block subsequent threads from accessing the splitting node for a long time.

Second, range scan may become a scalability bottleneck. The unsorted data in hash nodes requires a scanner to read every bucket to collect key-value pairs and sort to find values in the correct order. Such a large scan may even access irrelevant keys that do not satisfy the requested key range. This expensive process may be repeated many times due to conflicts if the accessed node is a hotspot.

4 OPTIMIZATIONS

In this section, we present three optimization techniques to overcome the challenges of basic B^{link}-hash described in Section 3.3.

4.1 Median Approximation

In a hash node split, a median key is found by scanning all the buckets and sorting the collected keys. This incurs significant memory accesses and sorting overhead on the critical path, which excludes other thread accesses for a long time.

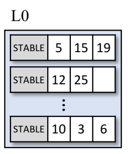
We make the key insight that a precise median is not required for the correctness of a split. Instead, a key that is close to the median is enough to achieve acceptable performance. We identify such an approximate key through sampling a subset of keys, instead of reading all the keys.

Specifically, a split thread sequentially scans buckets until it collects a sufficient number of keys, and calculates a median with the sampled collection. Sampling guarantees the upper bound error rate for an approximate median with a sufficient sample size [37, 47].

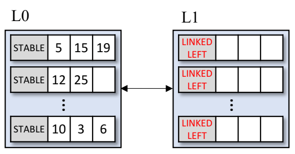
Random sampling ensures that the results obtained from a sample approximate what would have been obtained if the entire population had been measured [48]. The randomness of selecting a sample in B^{link}-hash comes from hashing. Every key is randomly distributed across buckets based on a hash function, and collecting the keys with sequential scan guarantees random selection.

4.2 Lazy Split

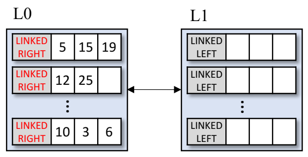
Migrating key-value entries to a new node is another scalability bottleneck in a hash node split. Since each bucket is similar to a B+-tree node in size, a large amount of data needs to be moved. In the meantime, every entry needs to compare to the median to determine whether it should be migrated.



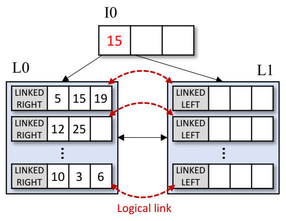
(a) Initial state.



(b) New node allocation.



(c) Bucket state update.



(d) Create a parent.

Figure 5: Steps for lazy split.

We present *lazy split* to minimize the migration cost by breaking down the node-level critical section into multiple smaller bucket-level critical sections. It avoids expensive data movement on the critical path, and the data are migrated in a lazy manner at a bucket granularity. Specifically, we place a *link state* variable per bucket to capture the state of lazy migration. A bucket holds one of the three states; `STABLE`, `LINKED_LEFT`, or `LINKED_RIGHT`, and each state informs the following:

- `STABLE` indicates that all the key-value entries inside the bucket are in the correct position.
- `LINKED_LEFT` denotes that some key-value entries may have not been migrated from the bucket in its left sibling node.
- `LINKED_RIGHT` implies that some key-value entries in the current bucket may belong to the bucket in its right sibling node but have not been migrated yet.

Figure 5 shows an example of splitting a hash node in a lazy manner. First, all the buckets in L0 are `STABLE` (Figure 5(a)). An approximate median, 15, is found in this phase. Then, a new node L1 is allocated with the states initialized to `LINKED_LEFT` in all its buckets (Figure 5(b)). This indicates that those buckets in L1 are not stable, and some key-value entries in L0's buckets may belong to L1's buckets. Note that L0 and L1 are connected via doubly-linked pointers. Next, all the buckets in L0 are updated to `LINKED_RIGHT` (Figure 5(c)). This implies that those buckets in L0 are not stable, and some key-value entries in L0's buckets may belong to L1's buckets. Note that none of the key-value entries has physically moved to L1. Finally, the split completes by creating its parent node with the median 15 found in the first phase (Figure 5(d)).

Lazy split modifies the basic operations in a hash node. That is, subsequent threads help along stabilizing unstable buckets by physically migrating key-value entries at a bucket granularity. The unstable buckets always come in pairs, and they are stabilized before the chain can possibly grow in length. For example, a bucket that is `LIKNED_RIGHT` is connected to the same-indexed bucket that is `LINKED_LEFT` in its right sibling node, and they are stabilized before an SMO at latest. We discuss how it changes each hash node operation by revisiting the operations in Section 3.2.

Bucket stabilization. Buckets become unstable after lazy split, and those buckets are stabilized outside the critical section, i.e., split operation. A writer starts with holding a latch for a bucket in basic B^{link}-hash operations (e.g., insert, update and delete). The bucket link state is always first examined before accessing contents in a bucket. If the bucket is not `STABLE`, it stabilizes the logically connected two buckets by acquiring a latch in the same-indexed bucket in its left or right sibling node according to the link direction.

First, a writer reads the link direction of an unstable bucket in the current node. If it is `LINKED_LEFT`, the same-indexed bucket in the left sibling node becomes the left bucket, and the current bucket becomes the right bucket. The writer then tries to latch the left bucket. In case of any failures of acquiring a latch, it aborts and retries. If the bucket in the current node is `LINKED_RIGHT`, it becomes the left bucket, and the same-indexed bucket in the right sibling node becomes the right bucket. Then, the right bucket is latched. Next, all the keys in the left bucket are examined if any of them are larger than the high key in the left node. Those keys are migrated to the right bucket. Then, it updates the states of left and right buckets to `STABLE`, and releases the acquired bucket latch.

Read. After obtaining the version number of a bucket, a reader first examines the state of the bucket to determine if it needs to be stabilized. If stable, it works the same way as the basic operation, returning the value after reading the content of the bucket for a matching key. Otherwise, the reader is promoted to a writer to help along stabilizing the unstable bucket. The promotion is always done by upgrading the bucket latch with the obtained version number. After the latch acquisition, it stabilizes the bucket in the way discussed above. In case of any failures of latch acquisition or bucket stabilization, it aborts and retries from the root. After the work, it releases the bucket latch and reads the newest version number of the bucket. Then, it proceeds to the next steps in the basic operation, finding a value for the matching key.

Insert, Update, and Delete. The insert, update, and delete operations for lazy split work in a similar way to the read. After the version number validation of the node, a writer reads the state of a bucket to stabilize it on demand as done in read operation. If the bucket is stable, it follows the same logic as the basic operations, trying to insert a key-value pair into the bucket or update/delete a value for the matching key in the bucket. Otherwise, it also proceeds to the same step after the stabilization.

Scan. A range scan also helps along stabilizing unstable buckets. Each bucket state is investigated before reading the contents to make sure it is stable. In case of finding an unstable bucket, a scanner is promoted to a writer by upgrading the bucket latch with the version number obtained at the very first access of the bucket. Then, the thread follows the same logic in Section 3.2, proceeding to scanning the rest of the buckets, after releasing the latch.

Split. Unlike the work in the basic operation, a split thread first scans all the bucket states in the current node to make sure they are stable. It tries to stabilize unstable buckets if any of them are seen during the scan. This reduces the length of the critical section, and guarantees a logical link between the buckets in adjacent nodes to be always pairwise. To minimize the works done inside the critical section, the new node allocation initializes all of its bucket states to `LINKED_LEFT`. After obtaining an approximate median, key-value pairs are logically migrated. This only sets all the bucket states in the current node to `LINKED_RIGHT`, while the basic split physically migrates half of the records via key comparison with the median. The bucket state updates connect pairwise logical links between the buckets in the current and the new node. Note that no data are moved to the new node, but the bucket state update delegates the work for data movements to subsequent threads. Proceeding to the steps of updating the sibling pointers completes the lazy split.

While lazy split modifies the data operations in a hash node, it does not affect any operations in a B+-tree node as all the buckets are required to be stable prior to a structure modification. That is, every structure modification in hash nodes is executed with the guarantee of data stability in buckets, which creates logical links only between two hash nodes.

As each data operation helps along stabilizing unstable buckets, it may affect the latency performance. We mitigate this overhead by stabilizing them before the chain can possibly grow in length so that each operation stabilizes a pair of buckets at most.

4.3 Structure Adaptation

While hash nodes provide scalable performance in monotonic insertion, it hurts range scan performance. Unsorted keys force threads to scan every bucket to collect all the key-value pairs and to sort them to find values in the correct order. This is both inefficient and vulnerable to conflicts. If another thread updates a bucket while the scan is happening, the whole process may need to abort and retry for data consistency.

We develop the *structure adaptation* technique to efficiently support range scans in B^{link}-hash. That is, a hash node is dynamically converted to B+-tree nodes upon a scan access. A scanning thread first collects and sorts all the key-value pairs in a hash node, and replaces the hash node with B+-tree nodes via copy-on-write (CoW). Introducing a set of new nodes propagates to the upper levels of the tree, and such updates may cause multiple updates in a parent node. To minimize the cost of adding new records, every parent is updated in a batch fashion.

Structure adaptation benefits range scans as it enables sorted sequential reads in B+-tree nodes. In B^{link}-hash, we use a simple heuristic to trigger conversion for high performance for both monotonic insertion and scanning at the same time. Specifically, the rightmost node is always a hash node; this is because time-series workloads typically have monotonically increasing keys that all go to the rightmost leaf node. Then, a hash node is converted to B+-tree nodes upon the first scan request to it. We observe such a simple heuristic works well for the workloads we tested.

Node Type Conversion. Hash node conversion works in a similar way to lazy split. Once a scanner finds a hash node that is not the rightmost node, it converts the node into B+-tree nodes in the following steps: First, it visits every bucket as done in lazy split, stabilizing unstable buckets to make sure no data is lost. Then, the thread additionally latches the left sibling node to safely introduce new B+-tree nodes in a CoW manner, after acquiring the node and all the bucket latches. New B+-tree nodes are allocated after collecting and sorting all the key-value pairs. We maintain an 80% utilization for the new B+-tree nodes for space efficiency. Then, the thread updates the right sibling pointer in the left sibling node to point to the first new B+-tree node, and releases its latch. Likewise, if the right sibling node is a hash node, it updates the left sibling pointer in the right sibling node to point to the last B+-tree node. The updates are propagated to the upper levels of the tree in a batch fashion, and the replaced hash node is reclaimed with an epoch-based reclamation protocol [24].

B^{link}-hash follows the lock guarantee of B^{link}-tree [31]. A hash node conversion holds at most three node latches at a time, i.e., a hash node that is to be converted, its parent node, and its left sibling node if it exists, as done in B^{link}-tree.

5 EVALUATION

In the following, we evaluate B^{link}-hash against representative in-memory indexes and provide performance analyses.

5.1 Experimental Setup

Hardware. We conduct all experiments on a CloudLab [19] machine with c6420 instance type, which contains two Intel Xeon Gold 6142 CPUs. Each CPU has 16 cores (32 hyper-threads) with 22 MB L3 cache, and each core has 32 KB of L1 instruction cache, 32 KB L1 data cache, and 1 MB L2 cache. The server is equipped with 512 GB of DDR4 DRAM.

Software. We pin each thread to a specific core to avoid unnecessary CPU migration and remote memory access. Threads are assigned to local regions first before using a remote NUMA socket. We use `tcmalloc` [4], a parallel memory allocator, for scalable memory allocation. All the indexes are compiled with gcc 7.5 with the `-O3` optimization flag. We measure multiple runs of experiments and report the average of three stable runs.

We implemented B^{link}-hash[1] on top of B^{link}-tree [31]. It is written in C++ with 3800 lines of code (LoC), which includes 1640 LoC of B^{link}-tree implementation. We use Intel Advanced Vector Extensions (AVX2) [25] to accelerate fingerprint operations, e.g., when comparing a hash key with fingerprints in a hash node.

5.2 Workloads

For each experiment, we first populate an index with 100 million records, before executing 100 million benchmark operations. We use two different types of workloads to evaluate B^{link}-hash against other representative indexes: (1) *Time-series* workload that performs insertions with monotonically increasing keys (i.e., timestamps), and (2) *YCSB-Email* that runs YCSB with email keys. We use YCSB workload to evaluate B^{link}-hash in OLTP workload scenarios since its design goal is not limited to efficiently supporting time-series workloads, but is to extend the capabilities of B^{link}-tree on the new pattern of monotonic keys while keeping its benefits.

Time-Series. We use a synthetic dataset for time-series workload from previous work [56], which models data arrival from distributed sensors with timestamp keys. It simulates 1K sensors to record events. The key for each event consists of 6-byte timestamp followed by 2-byte sensor ID, and the value for each event is an 8-byte integer. We implement the workload by having each thread read the synchronized local clock through `RDTSC` (read timestamp counter) instruction. We first evaluate individual index operations (insert, read, scan), and then evaluate a mixture of the operations (50% insert, 30% long scan, 10% short scan, 10% of read). Inserts are monotonic, while reads and scans are uniformly random. Scan workload queries 50 records at average, while mixed workload queries 5–10 and 10–100 records for short and long scans, respectively.

YCSB-Email. The Yahoo! Cloud Serving Benchmark (YCSB) [16] is a widely-used key-value store benchmark. In this workload, we use publicly available email addresses to evaluate indexes in string keys [20], instead of using the dataset generated by the YCSB workload generator. The average length of email addresses is 19 bytes, and the length of each record varies from 12 bytes to 32 bytes. The value for each record is an 8-byte integer. The details of evaluated workload configurations are summarized in Table 1. Note that we do not include workloads `D` and `F` in our experiments. Workload `D` consists of 5% of inserts and 95% of reads, which read recently inserted items. We believe workload `B` serves the same purpose in terms of index behavior as it has similar characteristics to workload `D`. Workload `F` consists of 50% of reads and 50% of read-modify-writes (i.e., read an item, modify it, and write it back). As each modification is reflected after a write-back which updates an old value to a new value, it can be implemented with read and update operations in indexes, and we believe workload `A` covers the scenario.

Table 1: YCSB workload configurations.

Workload	Operations
Load	Insert (100%)
A	Read (50%), Update (50%)
B	Read (95%), Update (5%)
C	Read (100%)
E	Scan (95%), Insert (5%)

5.3 Indexes under Comparison

Modern indexes generally fall into three categories: B+-tree-based, trie-based, and hybrid structures. We carefully choose five representative in-memory indexes in these categories to compare with B^{link}-hash experimentally. For B+-tree-based indexes we choose B^{link}-tree [31] and Bw-tree [34]. For trie-based indexes we choose Adaptive Radix Tree (ART) [32] and Height Optimized Trie (HOT) [11]. We choose Masstree [38] as a representative hybrid index.

B^{link}-tree. B^{link}-tree [31] is a disk-based B+-tree designed to overcome the limitation of latch crabbing as discussed in Section 2. We implement the in-memory version of B^{link}-tree from scratch based on previous studies [15, 31]. For fair comparison, we apply performance optimization to deliver good performance in a main-memory system. We tune the node size in the main-memory setting, instead of using the default page access granularity. We also employ linear search instead of binary search inside each index node as sequential access can take advantage of hardware prefetching.

OpenBw-Tree. Bw-tree [34] is a latch-free B+-tree designed to overcome the scalability limitation of latch-based B+-trees with mapping table and delta records. In our experiments, we use OpenBw-tree, an open-source in-memory Bw-tree implementation [52].

ART. ART [32] is a trie-based in-memory index that effectively addresses the shortcomings of large spans in tries. Instead of using a fixed node size, it employs adaptive nodes to dynamically increase

[1]Implementation available at https://github.com/chahk0129/Blink-hash.

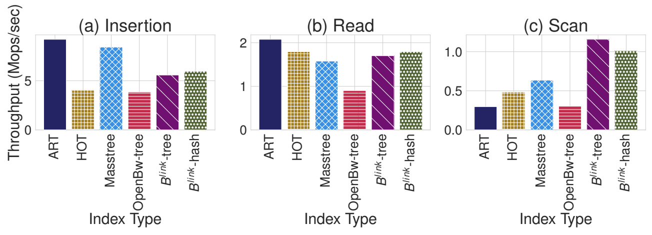


Figure 6: Single-thread throughput of indexes in time-series workload.

and decrease the fanout of a node on demand. We use the open-source implementation for ART by the original authors.

HOT. HOT [11] is another trie-based index that improves search efficiency by minimizing the overall tree height. It dynamically adapts its structure to maintain a balanced tree by comparing the heights between subtrees. We use unmodified open-source implementation for HOT by the original authors.

Masstree. Masstree [38] is a hybrid of B+-tree and trie designed to support efficient queries for variable-length keys. Each 8-byte key slice is treated as a key for each subtree which is a B+-tree, and the subtrees form a trie. We also use unmodified open-source implementation for Masstree by the original authors.

5.4 Single-threaded Performance in Time-Series Workload

We first evaluate single-threaded performance of indexes in time-series workload. Figure 6 shows the throughput of indexes with different operations.

Insertion. Figure 6(a) shows the throughput for monotonic insertion. ART achieves the highest throughput because of its node adaptation technique that gradually increases the fanout based on the utilization of each node. HOT, on the other hand, shows one of the lowest throughput. This is because it frequently modifies the tree structure to balance the height of subtrees, and each record is written via CoW. Among the B+-tree variants, OpenBw-tree shows the lowest throughput, because it prepends a delta record for each write to avoid in-place updates, which leads to occasional delta chain consolidation by subsequent operations. B^{link}-hash shows slightly better performance than B^{link}-tree, since the large hash node size setting reduces the height of the tree.

Read. Figure 6(b) reports the throughput for read operations. While most of the indexes show similar performance, OpenBw-tree shows low throughput. It suffers from completing partial SMOs which leads to delta chain consolidation. If a reader encounters a split delta record on top of a delta chain, it helps along to complete the partially applied split, which consolidates the delta chain. Different from insertion performance in Figure 6(a), HOT especially shows increased throughput, because it benefits from a balanced tree height which reduces the distance from a root node to leaf nodes.

Scan. Figure 6(c) shows the throughput for scan operations. B+-tree variants generally perform better than trie variants for range accesses with linked leaf node traversal. B^{link}-hash shows comparable performance to B^{link}-tree because of the structure adaptation, which dynamically converts hash nodes to B+-tree nodes and thus, slightly slows down the scan performance. Masstree shows lower throughput than B^{link}-tree and B^{link}-hash, although it acts as a B+-tree for 8-byte keys. Keys in its border nodes are not physically sorted, and it causes random accesses to collect keys in the correct order which is not cache-friendly. Tries, in contrast, need to traverse up and down the nodes to access the next keys. HOT shows higher throughput than ART, because it takes advantage of its balanced tree height which reduces the distance to the next keys, while ART suffers from a substantial amount of pointer dereferences.

5.5 Scalability in Time-Series Workload

We now go beyond a single thread and evaluate the scalability of indexes in time-series workload. Figure 7 shows the throughput of the indexes with varying number of threads.

Figure 7(a) shows the performance of insert workload. B^{link}-hash is the only index that scales in monotonic insertion, outperforming the others by up to 91.3× at 32 threads. The hash node in B^{link}-hash evenly distributes insertions to different buckets, while all the other trees suffer from extremely high contention on the rightmost leaf node as discussed in Section 2.1.

Figure 7(b) shows the performance of read workload. While most indexes scale linearly, B^{link}-hash outperforms the others as it provides constant lookup time in leaf nodes. Moreover, the larger node size in the leaf layer allows B^{link}-hash to shorten the tree height which reduces the number of memory accesses. The height of B^{link}-hash is 4, while that of the others is 6 (for B^{link}-tree), 7 (for ART, HOT, and Masstree), 5 (for OpenBw-tree without delta records), and 8 (for OpenBw-tree with delta records).

Figure 7(c) reports the performance of scan workload. B^{link}-tree shows the highest scalability as it sequentially scans leaf nodes with linked node traversal. B^{link}-hash shows comparable performance to B^{link}-tree because it takes advantage of the structure adaptation. Trie-variants generally show poor performance, because they need to traverse up and down the nodes to access the next keys. Masstree causes random accesses in border nodes as discussed in Section 5.4.

Figure 7(d) shows the performance of mixed workload, which consists of a mixture of insert, read, and scan operations. B^{link}-hash outperforms all the other indexes up to 74.2× at 64 threads. It achieves scalable performance with the optimizations in hash nodes and dynamic structure adaptation for in-flight range scans. The other indexes fail to scale due to the monotonic insertions.

5.6 Insertions with Delays

In practice, timestamps may not arrive in a strict monotonic order. Instead, many insertions may be delayed and the timestamps may slightly lag behind due to sensor malfunction, network delay, or scheduling overhead. This delay creates out-of-order keys in data arrival time. We model the effect by injecting random delays to timestamp keys that follow a Poisson distribution.

In this experiment, we additionally evaluate a simple straw-man solution to the high contention problem in existing indexes. Monotonic keys can be inserted into an intermediate hash buffer in parallel, and those buffered data can later be flushed down to an index in batch to mitigate the contention. The approach can

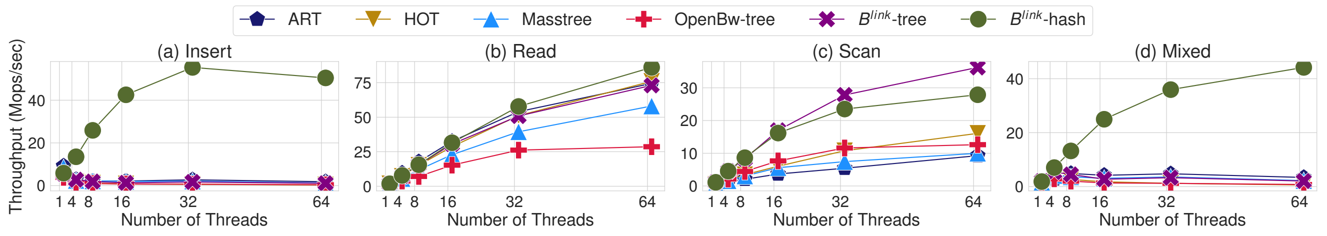


Figure 7: Throughput of the time-series workload with varying number of threads.

efficiently support monotonic insertion by distributing thread accesses to different hash buckets in the buffer, while maintaining the benefits of tree indexes in range scans.

We design two buffer management approaches that differ in handling out-of-order keys (i.e., keys that are smaller than the highest key of a tree index). The first approach flushes a buffer when an out-of-order key arrives (*Flush on out-of-order*). The other approach flushes a buffer when it becomes full, and inserts an out-of-order key directly into an index (*Insert on out-of-order*). A buffer flush creates a subtree by collecting and sorting the data in the buffer, and the subtree is inserted into a B+-tree. Each approach is implemented on top of B^{link}-tree, and uses two buffers which are switched upon a flush operation. We use the same layout of the hash node in B^{link}-hash for the buffer implementation for fair comparison, and we tune the buffer size to its best performance setting, 512 KB. We vary the expected random delays to evaluate their performance under a varying amount of out-of-order keys.

Figure 8(a) shows the performance of insert workload with delays. While B^{link}-hash dominates, all the trees maintain constant performance up to 1 msec delay, and their performance starts to increase at 10 msec delay. It is because skewed accesses to rightmost nodes are distributed to different leaf nodes as the amount of out-of-order keys increases. However, HOT and OpenBw-tree still suffer from high contention at 10 msec delay. HOT causes higher conflicts as it writes a new record via CoW due to height balancing. OpenBw-tree also results in higher contention consolidating delta chains. This performance trend implies that conventional indexes cannot provide high-speed data ingestion for time-series workload. The buffer approaches show higher performance than the conventional indexes when the expected delay is small, but their performance is lower than B^{link}-hash. It is because data collection and sorting in a buffer flush are executed sequentially, while B^{link}-hash leverages high parallelism by breaking down a node-level critical section into bucket-level critical sections with lazy split.

Figure 8(b) shows the performance of mixed workload. B^{link}-hash shows constantly increasing performance across the random delays, while the performance of the other trees stays the same until the delay of 10 msec. It is because the amount of stress in rightmost nodes has decreased as insertion only accounts for 50% in this workload, and such mitigated stress allows B^{link}-hash to achieve better parallelism as more keys become out-of-order. However, the contention is still high enough to destroy the performance of the other indexes. When the expected delay gets large enough, B^{link}-tree dominates due to its optimized link pointer-based split and superiority in scans. Buffer approaches follow the similar

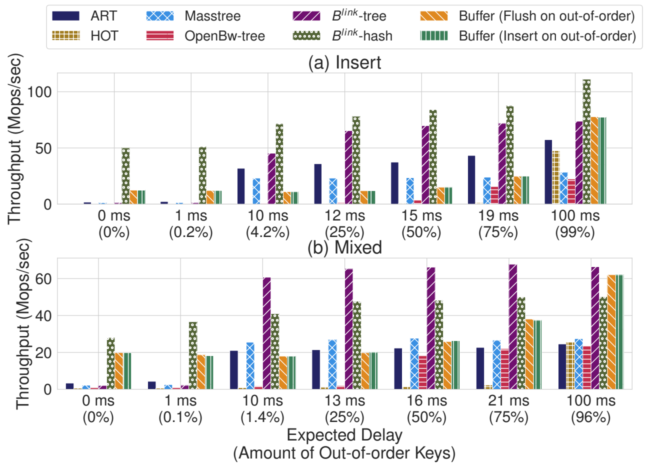


Figure 8: Throughput of time-series workload under varying random delays with 64 threads.

performance trend in insert workload but at a higher rate. When keys become highly out-of-order, they show bigger performance gaps to B^{link}-tree. It is because long range scan accesses additionally trigger buffer flushes which are not fully utilized.

5.7 YCSB-Email Workload

In this section, we evaluate the indexes on YCSB-Email workload to investigate their performance under the generic OLTP workload scenarios where keys may have variable lengths. This experiment shows the competitiveness of B^{link}-hash when there are no monotonic insertions.

Column (a) of Figures 9 and 10 show the throughput and CDF of latency (at 64 threads), respectively, of the indexes for different workload configurations under YCSB-Email workloads. Trie variants generally show better throughput and lower latency than B+-tree variants. It is because they only store partial keys, which reduces the cost of data reads and writes, while B+-trees store the entire keys. ART shows linearly scaling performance within a local socket, but the improvement rate decreases at the maximum number of threads due to the increased amount of remote memory access in CoW node adaptation. B^{link}-hash, however, shows linear scalability even across a remote socket at 64 threads. In addition to its hash optimization, it takes advantage of short critical sections benefiting from median approximation and lazy split.

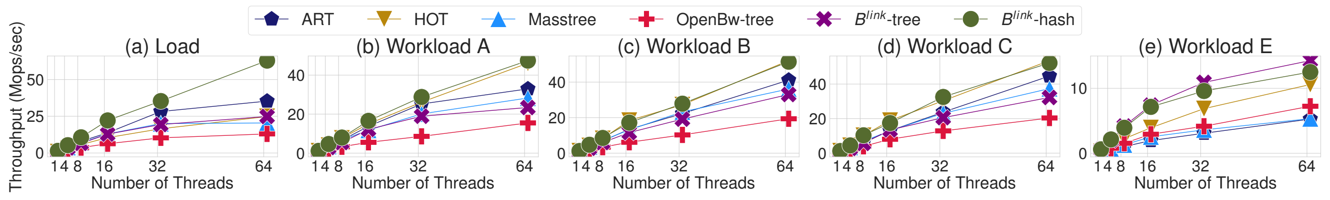


Figure 9: Throughput of YCSB-Email with varying number of threads.

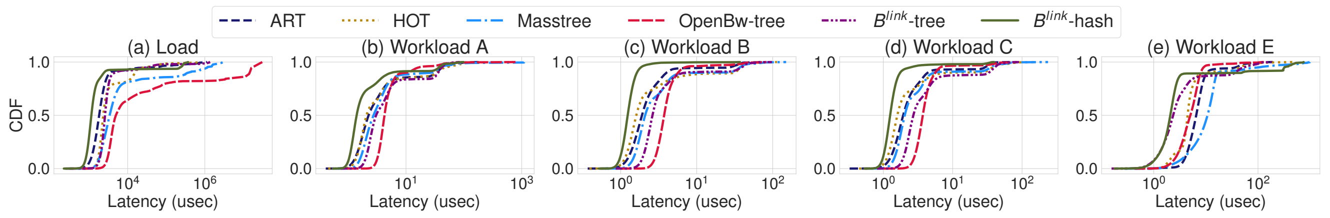


Figure 10: CDF of latency in YCSB-Email with 64 threads.

Columns (b), (c), and (d) of Figures 9 and 10 report the throughput and latency of workload A, B, and C, respectively. While all the indexes show scaling performance, B^{link}-hash and HOT outperform the other trees. B^{link}-hash benefits from fingerprints that filter out unnecessary string key accesses. HOT takes advantage of sparse partial key layout which enables further key compaction, allowing more descendants to share a common prefix. B+-tree variants, however, suffer from expensive string key comparisons.

As Column(e) of Figures 9 and 10 show, under workload E, B^{link}-tree outperforms the others as it leverages linked-leaf traversals. B^{link}-hash follows the same trend and shows comparable throughput as it dynamically adapts its structures. However, it shows high tail latency as thread accesses to hash nodes are blocked until their conversions complete. Among trie variants, HOT shows high throughput and low latency by leveraging its balanced tree structure which reduces the distance to next keys.

5.8 Memory Footprint

We now compare the memory footprint of indexes with different key types. For a deeper analysis, we break down the total amount of memory consumption into five categories.

- *Metadata* includes the information inside a node other than keys, child pointers, and values (e.g., latch, counter, and link pointer).
- *Structure data* consists of child pointers and separator keys (e.g., keys in non-leaf nodes) that are needed to form a tree structure.
- *Key data* represents unique keys and values.
- *Occupied* and *unoccupied* represents populated and empty entries in a node, respectively.

Unlike conventional tries, Masstree stores suffix keys in a separate data structure called *string bag*. For a fair comparison, we show the footprint of Masstree with and without suffixes. As OpenBw-tree supports explicit delta chain consolidation, we also report the memory consumption with and without it.

Figure 11(a) describes the memory footprint of indexes for 8B integer keys. HOT shows the lowest memory consumption due to

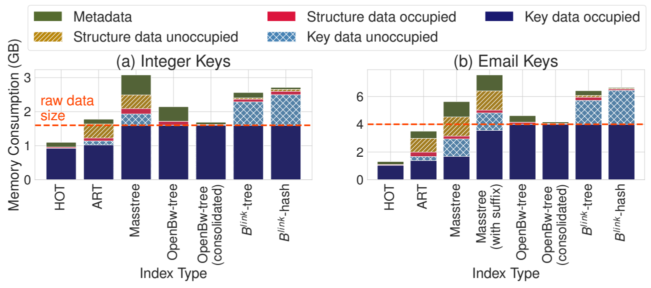


Figure 11: Memory footprint of indexes on different key types.

its dynamic adaptive nodes. ART, however, wastes a large amount of space on the structure data, because there is a large space gap between static adaptive nodes. Masstree spends much memory on metadata since its node structure contains more information than conventional B+-trees. In addition to a parent pointer, a border node includes permutation for key indirection, doubly-linked pointers, and additional metadata for suffixes. The difference of OpenBw-tree and consolidated OpenBw-tree shows the memory consumption on delta chains. A larger amount of key data is unoccupied in B^{link}-tree and B^{link}-hash because of their static node structures. B^{link}-hash shows slightly higher memory consumption due to large hash nodes. Note that HOT does not have unoccupied data as it adapts its node size via CoW. OpenBw-tree also uses elastic nodes, but it has unoccupied structure data in its mapping table.

As the key size increases to 32 bytes, trie variants gain more benefits as shown in Figure 11(b), because of the characteristics of key types. Bit representations in each byte slice of integer keys are more uniformly distributed, while those in email keys are skewed. Therefore, more keys can share a common prefix in email keys. Masstree, however, shows increased memory consumption although it inherits the characteristics of tries. It creates new layers to store next slices of keys, each of which is equivalent to a B+-tree, introducing

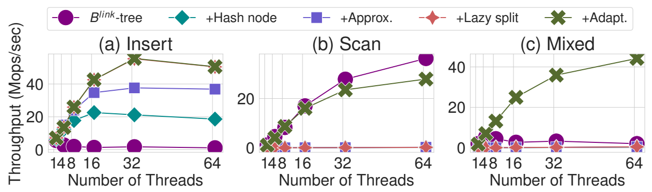


Figure 12: Factor analysis under time-series workload.

more interior nodes and border nodes that are not fully utilized. B^{link}-hash consumes similar amount of memory space as B^{link}-tree.

5.9 Analysis of Design Chocies

In this section, we analyze the performance impact of design choices and performance optimizations in B^{link}-hash. We first break down the performance gap between B^{link}-tree and B^{link}-hash, and incrementally add features to B^{link}-tree. Figure 12 shows the scalability of each increment using time-series workload. Each feature is added in horizontal order in the legend. Note that hash node optimizations (i.e., +Hash node, +Approximate median, and +Lazy split) do not have any performance improvements in the workloads that include scan operations, until the structure adaptation is applied.

+Hash node. Replacing a B+-tree node with a hash node mitigates extremely high contention by distributing highly skewed thread accesses to different buckets. It improves the performance in insert workload by up to 17.2× compared to the baseline, B^{link}-tree. However, the throughput does not further scale as it introduces a new scalability bottleneck, splitting a hash node.

+Approximate median. Median approximation eliminates the need to collect all the keys in a node to calculate the median during a split. As the split operation becomes the scalability bottleneck in insert workload, sampling keys improves the performance by up to 1.8×. This substantially reduces the amount of expensive bucket scans and sorting overhead inside the critical section.

+Lazy split. Lazy split breaks down a node-level critical section into much smaller bucket-level critical sections. It improves insert performance by up to 1.5× compared to the previous increment, +Approximate median. Lazy split leverages high parallelism by delegating expensive key comparisons and data migrations to subsequent threads accessing the corresponding buckets.

+Adaptation. Structure adaptation resolves another scalability bottleneck, scan operations. It improves the performance in scan and mixed workloads by up to 140.1× and 87.4×, respectively, compared to the previous increment, +Lazy split. Dynamic conversion of a hash node into B+-tree nodes allows B^{link}-hash to achieve comparable performance to B^{link}-tree in scan workload. It also allows B^{link}-hash to achieve linear scalability in mixed workload, while B^{link}-tree and all the other previous increments fail to scale.

5.10 Analysis of Splitting Points

In this section, we analyze the performance impact of different splitting points in B^{link}-hash. Table 2 shows the throughput with 50th, 60th, 70th, 80th, and 90th percentile of splitting points. That is, 90th percentile uses 90th percentile key as a median in the sampled subset of keys.

Table 2: Throughput under monotonic and random insertion with different splitting points at 64 threads.

Workload Type	50th (Percentile)	60th	70th	80th	90th
Monotonic Insertion	48.9 (Mops/sec)	55.5	59.6	63.2	67.8
Random Insertion	110.4 (Mops/sec)	109.9	107.8	104.2	93.5

Table 3: Throughput with varying hash node sizes for time-series workload at 64 threads.

Workload Type	64 KB	128 KB	256 KB	512 KB
Insert	21.9 (Mops/sec)	34.8	48.9	16.5
Mixed	24.7 (Mops/sec)	36.7	44.4	17.2

The second row of Table 2 shows the performance under monotonic insertion. The performance increases as the splitting point gets larger because of reduced amount of node splits due to larger free space in new nodes.

The third row of Table 2 shows the performance under random insertion. There is no huge difference in performance among the criteria, but the performance slightly degrades as it increases. The unbalanced hash nodes may easily lead to consecutive splits, especially in keys that follow an uniformly random distribution.

Although larger splitting points provide better performance in monotonic insertion, we use 50th percentile as our default splitting criterion. Keys in practice are not strictly monotonic as discussed in Section 5.6, and such out-of-orderness may lead to unnecessary splits, causing memory inefficiency. Therefore, we provide the splitting point as a tuning parameter based on application requirements.

5.11 Analysis of Different Hash Node Sizes

In this section, we analyze the performance effects of different hash node size settings. The node size plays a significant role in its performance as it should be not too small to reduce the amount of hash collisions and not too large to maintain the length of its critical section short, i.e., updating all bucket states in a split operation.

Table 3 shows the performance of B^{link}-hash with 64, 128, 256 and 512 KB hash node sizes for insert and mixed workloads in time-series workload at 64 threads. Among the different node sizes, 256 KB shows the highest throughput. 64 and 128 KB settings suffer from a large amount of hash collisions, and 512 KB setting spends increased amount of time scanning all buckets in a node which blocks other thread accesses. By default, B^{link}-hash uses 256 KB for its hash node size.

5.12 Data Skew for Approximate Median

Although approximating a median brings a huge performance gain, it may create data skew between the original node and the new node in split operations. For example, if the approximate median is highly left-skewed, it migrates the majority of data to the new node, resulting in resource inefficiency in the original node. To identify the potential data skew problem, we vary the sampling ratio, and compare the index of each approximate median with that of the true median by sorting all keys in a node.

Figure 13 shows the CDF of data skew rates of different sampling ratios for 1K split operations. The value of each skew rate denotes

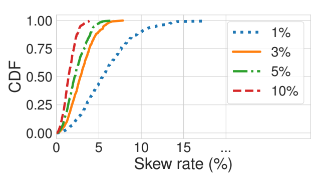


Figure 13: CDF of data skew rate with varying sample ratios.

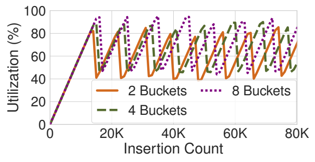


Figure 14: Utilization of the rightmost hash node with different bucket probing distances.

the difference in node utilization between the original and new node. For example, if the original node is 90% utilized, and the skew rate is 10%, one of the nodes will be utilized 35% and the other will be utilized 55% right after the split. The result reports that approximate medians do not create a data imbalance problem in most cases even with a small sampling ratio. The average skew rates are 3.73%, 1.95%, 1.47%, and 0.97% for 1%, 3%, 5%, and 10% sampling ratios, respectively, and their 95th percentile rates are 8.81%, 4.55%, 3.81%, and 2.32%. By default, B^{link}-hash employs 3% for its sampling ratio to reduce both occasional popping skew and bucket scanning overhead in median approximation.

5.13 Utlization of Hash Node

Maintaining good node utilization is critical to memory efficiency. In this section, we investigate the utilization of hash nodes with a varying number of bucket probing distances.

Figure 14 shows the utilization of the rightmost hash node with 2, 4, and 8 bucket probing distances under monotonic insertion. The load factor increases as the probing distance increases. Each setting shows around 5-6% performance difference in throughput (i.e., the smaller probe setting, the higher performance). For efficient memory usage, we use 4 bucket probe setting by default.

6 RELATED WORK

Synchronization. The scalability of an index highly depends on its underlying synchronization mechanism [13, 17, 18]. Latch coupling [8] has been widely adopted in traditional B+-trees to safely execute transactions. Optimistic latch coupling [30] has been proposed to overcome the disadvantage of frequent updates to shared data by avoiding unnecessary writes incurred by acquiring and releasing latches in latch coupling. Various index designs have adopted optimistic latch coupling, and presented further techniques to reduce synchronization cost. Masstree [38] divides a version-based latch into different operation latches. It atomically updates a value via read-copy-update [40] to allow readers access a node while it is being updated. Hydralist [39] decouples its structure into data layer and search layer, and new updates in data layer are propagated to search layer asynchronously. B^{link}-hash uses optimistic synchronization and further reduces the cost by breaking down a long node-level critical section into multiple bucket-level critical sections.

Hybrid data structures. Hybrid data structures gain performance improvement by combining two indexes into a single structure. Wormhole [54] combines a hash index on top of tries for better search performance in prefix match. Bounded-disorder access method [36] uses multiple hash buckets for its data node in a tree to reduce disk accesses. B^{link}-hash also integrates a hash index into a B+-tree, but focuses on resolving the limitation of existing indexes in monotonic insertion while maintaining efficient sequential scans by converting hash nodes back to B+-tree nodes.

Indexes in time-series database systems Time-series database systems have gained dramatic popularity with the advent of IoT devices and smart sensors. While they are designed to rapidly handle data ingestion and efficiently store time-series data, conventional indexes are still used which hurt the scalability in timestamp insertions. InfluxDB [1] and TimescaleDB [5] use B+-tree for their index. Log-structured merge (LSM) tree-based systems are also preferred in such workloads, e.g., OpenTSDB [49], RocksDB [3], and LevelDB [22]. Yet they use skip-list for the in-memory component which also suffers from highly skewed thread accesses to rightmost entries. B^{ϵ}-tree [14] is often compared with LSM-trees as a write-optimized index [10, 26]. However, it also faces the same issue in timestamp insertions, as the data in internal nodes are gradually flushed down to rightmost leaf nodes. B^{link}-hash overcomes the scalability bottleneck by replacing its leaf nodes with hash nodes, and further achieves fast write performance by minimizing the critical section with median approximation and lazy split.

Index studies under monotonic insertion. Monotonic insertion workload has often been used to evaluate indexes for the worst-case insertion scenario in multi-core environments. Kowalski et al. used static monotonic workloads to evaluate modern indexes [29]. However, a pre-generated chunk of keys is statically assigned to each thread before the execution, which no longer becomes monotonic over time. Wang et al. used dynamic monotonic keys, but focused on the evaluation with a fixed number of threads [52]. We use dynamic monotonic keys to exercise the application scenario of time-series workloads that timestamped data are generated in real-time, and identify the limitation of existing indexes.

7 CONCLUSION

In this paper, we identify the fundamental limitation of today's indexes under monotonic insertion, and introduce a new scalable index, B^{link}-hash, that overcomes the limitation. It efficiently distributes highly skewed thread accesses to rightmost nodes into different memory regions by integrating a hash index into a B+-tree. We further present optimization techniques, median approximation and lazy split, to resolve the scalability bottleneck in B^{link}-hash, splitting a hash node, by breaking down a per-node critical section into per-bucket critical sections. B^{link}-hash also dynamically adapts to workload variations by converting its structural layout on demand. It achieves the best of both worlds by efficiently supporting the new pattern of monotonic keys, while maintaining the benefits of B+-trees in range scans. Our evaluation study shows that B^{link}-hash outperforms today's representative indexes not only in time-series workloads, but also in generic OLTP workloads.

ACKNOWLEDGMENTS

This work was partially supported by the National Science Foundation CNS-2106199.

REFERENCES

[1] [n.d.]. InfluxDB. https://www.influxdata.com Accessed: 2023-02-10.

[2] [n.d.]. Kdb+. https://kx.com Accessed: 2023-02-10.

[3] [n.d.]. RocksDB. http://rocksdb.org Accessed: 2023-02-10.

[4] [n.d.]. TCMalloc Allocator. https://google.github.io/tcmalloc/design.html Accessed: 2023-02-10.

[5] [n.d.]. TimeScaleDB. https://www.timescale.com Accessed: 2023-02-10.

[6] Mervat Abu-Elkheir, M. Hayajneh, and Najah Abu Ali. 2013. Data Management for the Internet of Things: Design Primitives and Solution. *Sensors (Basel, Switzerland)* 13 (11 2013), 15582–612.

[7] Nikolas Askitis and Ranjan Sinha. 2007. HAT-Trie: A Cache-Conscious Trie-Based Data Structure for Strings. In *Proceedings of the Thirtieth Australasian Conference on Computer Science - Volume 62* (Ballarat, Victoria, Australia) *(ACSC '07)*. Australian Computer Society, Inc., AUS, 97–105.

[8] R. Bayer and M. Schkolnick. 1977. Concurrency of Operations on B-Trees, Vol. 9. Acta Informatica, 1–21.

[9] Rudolf Bayer and Karl Unterauer. 1997. Prefix B-Trees. *ACM Transactions on Database Systems* 2, 1 (1997), 11–26.

[10] Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan. 2015. An Introduction to Bϵ-trees and Write-Optimization. *login Usenix Mag.* 40 (2015).

[11] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 521––534.

[12] Matthias Boehm, B. Schlegel, Peter Benjamin Volk, Ulrike Fischer, D. Habich, and Wolfgang Lehner. 2011. Efficient In-Memory Indexing with Generalized Prefix Trees. In *BTW*.

[13] Silas Boyd-wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2012. Non-scalable locks are dangerous.

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

[15] Sang K. Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. 2001. Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 181––190.

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

[17] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2013. Everything You Always Wanted to Know about Synchronization but Were Afraid to Ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 33–48.

[18] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 631–644.

[19] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[20] fonxat. 2018. 300 Million Email Database. https://archive.org/details/300MillionEmailDatabase Accessed: 2023-02-10.

[21] Christian Garcia-Arellano, Hamdi Roumani, Richard Sidle, Josh Tiefenbach, Kostas Rakopoulos, Imran Sayyid, Adam Storm, Ronald Barber, Fatma Ozcan, Daniel Zilio, Alexander Cheung, Gidon Gershinsky, Hamid Pirahesh, David Kalmuk, Yuanyuan Tian, Matthew Spilchen, Lan Pham, Darren Pepper, and Gal Lushi. 2020. Db2 Event Store: A Purpose-Built IoT Database Engine. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3299–3312.

[22] Sanjay Ghemawat and Jeff Dean. [n.d.]. LevelDB. https://github.com/google/leveldb Accessed: 2023-02-10.

[23] Leo J. Guibas and Robert Sedgewick. 1978. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (SFCS 1978)*. 8–21.

[24] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. 2007. Performance of Memory Reclamation for Lockless Synchronization. *J. Parallel Distrib. Comput.* 67, 12 (dec 2007), 1270—-1285.

[25] Intel. 2022. Intel AVX-2 Instructions. https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-avx2.html Accessed 2023-02-10.

[26] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2015. BetrFS: A Right-Optimized Write-Optimized File System. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (Santa Clara, CA) *(FAST'15)*. USENIX Association, USA, 301–315.

[27] Lihong Jiang, Li Da Xu, Hongming Cai, Zuhai Jiang, Fenglin Bu, and Boyi Xu. 2014. An IoT-Oriented Data Storage Framework in Cloud Computing Platform. *IEEE Transactions on Industrial Informatics* 10, 2 (2014), 1443–1451.

[28] L. R. Johnson. 1961. An Indirect Chaining Method for Addressing on Secondary Keys. *ACM Commun.* 4, 5 (1961), 218—-222.

[29] Thomas Kowalski, Fotios Kounelis, and Holger Pirk. 2020. High-Performance Tree Indices: Locality matters more than one would think. In *ADMS@VLDB*.

[30] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.* 6, 2 (jun 1981), 213–226.

[31] Philip L. Lehman and s. Bing Yao. 1981. Efficient Locking for Concurrent Operations on B-Trees. *ACM Trans. Database Syst.* 6, 4 (1981), 650––670.

[32] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 38–49.

[33] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of Practical Synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware* (San Francisco, California) *(DaMoN '16)*. Association for Computing Machinery, New York, NY, USA.

[34] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 302–313.

[35] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. 2014. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands) *(EuroSys '14)*. Association for Computing Machinery, New York, NY, USA.

[36] Witold Litwin and David B. Lomet. 1986. The bounded disorder access method. In *1986 IEEE Second International Conference on Data Engineering*. 38–48.

[37] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. 1998. Approximate Medians and Other Quantiles in One Pass and with Limited Memory. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data (SIGMOD '98)*. Association for Computing Machinery, New York, NY, USA, 426–435.

[38] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-Value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 183––196.

[39] Ajit Mathew and Changwoo Min. 2020. HydraList: A Scalable in-Memory Index Using Asynchronous Updates and Partial Replication. *Proc. VLDB Endow.* 13, 9 (may 2020), 1332––1345.

[40] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, and Orran Krieger. 2002. Read Copy Update. In *Proceedings of Ottawa Linux Symposium 2002*. 338–367.

[41] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 371–386.

[42] R. Pagh and G. A. Gibson. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004), 122–144.

[43] W. W. Peterson. 1957. Addressing for Random-Access Storage. *IBM Journal of Research and Development* 1, 2 (1957), 130–146.

[44] Björn Rabenstein and Julius Volz. 2015. Prometheus: A Next-Generation Monitoring System (Talk). USENIX Association, Dublin.

[45] Jun Rao and Kenneth A. Ross. 1999. Cache Conscious Indexing for Decision-Support in Main Memory. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 78–89.

[46] Jun Rao and Kenneth A. Ross. 2000. Making B+-Trees Cache Conscious in Main Memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data* (Dallas, Texas, USA) *(SIGMOD '00)*. Association for Computing Machinery, New York, NY, USA, 475–486.

[47] Sankardas Roy, Mauro Conti, Sanjeev Setia, and Sushil Jajodia. 2008. Securely Computing an Approximate Median in Wireless Sensor Networks. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks (SecureComm '08)*. Association for Computing Machinery, New York, NY, USA.

[48] William R. Shadish, Thomas D Cook, and Donald T. Campbell. 2001. Experimental and Quasi-Experimental Designs for Generalized Causal Inference.

[49] Benoît Sigoure. 2011. OpenTSDB: A Scalable, Distributed Time Series Database.

[50] V. Srinivasan and Michael J. Carey. 1991. Performance of B-Tree Concurrency Control Algorithms. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data* (Denver, Colorado, USA) *(SIGMOD '91)*. Association for Computing Machinery, New York, NY, USA, 416–425.

[51] V. Srinivasan and Micahel J. Carey. 1993. Performance of B+tree concurrency control algorithms. 2 (oct 1993), 361–406.

[52] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 473—-488.

[53] Geng Wu, Shilpa Talwar, Kerstin Johnsson, Nageen Himayat, and Kevin D. Johnson. 2011. M2M: From mobile to embedded internet. *IEEE Communications Magazine* 49, 4 (2011), 36–43.

[54] Xingbo Wu, Fan Ni, and Song Jiang. 2019. Wormhole: A Fast Ordered Index for In-Memory Data Management. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA.

[55] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1567–1581.

[56] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 323–336.