# Bridging Cache-Friendliness and Concurrency: A Locality-Optimized In-Memory B-Skiplist


## Abstract

Skiplists are widely used for in-memory indexing in many key-value stores, such as RocksDB and LevelDB, due to their ease of implementation and simple concurrency control mechanisms. However, traditional skiplists suffer from poor cache locality, as they store only a single element per node, leaving performance on the table. Minimizing last-level cache misses is key to maximizing in-memory index performance, making high cache locality essential.

In this paper, we present a practical concurrent B-skiplist that enhances cache locality and performance while preserving the simplicity of traditional skiplist structures and concurrency control schemes. Our key contributions include a top-down, single-pass insertion algorithm for B-skiplists and a corresponding simple and efficient top-down concurrency control scheme.

On 128 threads, the proposed concurrent B-skiplist achieves between 2×–9× higher throughput compared to state-of-the-art concurrent skiplist implementations, including Facebook's concurrent skiplist from Folly and the Java ConcurrentSkipListMap. Furthermore, we find that the B-skiplist achieves competitive (0.9×–1.7×) throughput on point workloads compared to state-of-the-art cache-optimized tree-based indices (e.g., Masstree). For a more complete picture of the performance, we also measure the latency of skiplist- and tree-based indices and find that the B-skiplist achieves between 3.5×–103× lower 99% latency compared to other concurrent skiplists and between 0.85×–64× lower 99% latency compared to tree-based indices on point workloads with inserts.



## Keywords

B-skiplist, blocked skiplist, concurrency

**ACM Reference Format:**

. 2018. Bridging Cache-Friendliness and Concurrency: A Locality-Optimized In-Memory B-Skiplist. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX


## 1 Introduction

The skiplist [28] has become a widely used in-memory index (i.e., the memtable) in many popular databases, including HBase [14], RocksDB [29], and LevelDB [21]. Additionally, Java features a skiplist as its primary concurrent set and map implementation [18].

| *YCSB Workload* | *Skiplist* | *B-tree* | ***B-skiplist*** (This paper) | *SL/BSL* | *BT/BSL* |
|---|---|---|---|---|---|
| Load + C | 4.9E9 | 2.1E9 | 1.5E9 | 3.2 | 1.4 |
| Load + E | 1.1E10 | 2.3E9 | 2.0E9 | 5.6 | 1.2 |

**Table 1: LLC load misses[1] of Facebook's folly skiplist [1], a concurrent B-tree [32] and the concurrent B-skiplist (this paper) during the YCSB [7] load and run phase. The load phase has 100% inserts, workload C has 100% finds, and E has 95% range queries/5% inserts.**

The main reason for the choice of skiplists [28] over trees (e.g., the B-tree [4]) is because skiplists enable simple structural modification operations. In contrast to tree-based indices, skiplists do not require complex rebalancing operations because elements are randomly (using coin tosses) assigned a height upfront. As a result, skiplists support simple and effective (both lock-based and lock-free) concurrency control (CC) schemes [10, 11, 15, 16, 27].

**Locality issues in skiplists.** Unfortunately, traditional skiplists exhibit poor spatial locality because they store a single element per node. In contrast, cache-friendly indexes such as B-trees [4] store multiple elements per node, reducing the height of the structure and therefore the number of memory fetches during top-to-bottom traversals. Furthermore, storing multiple elements per node further reduces cache misses during horizontal traversals.

Table 1 shows that skiplists incur significantly more cache misses than B-trees, leaving performance on the table. Concretely, on the tested workloads, a state-of-the-art skiplist [1] incurs 2.4 − 4.8× more cache misses than a comparable B-tree [32] on both point and range workloads. This discrepancy in cache misses translates into actual performance: the B-tree achieves between 2×-8× higher throughput than the skiplist on these workloads.

*In this paper, we introduce a **concurrent B-skiplist** that improves the cache locality (and therefore the performance) of the traditional skiplist without giving up on the skiplist's simple and effective CC schemes. Furthermore, due to its simplified CC scheme, the B-skiplist achieves lower worst-case latency than B-trees.*

**Skiplist structure.** At a high level, skiplists [28] are randomized self-balancing data structures that support fast search, insertion, and deletion in $O(\log n)$ time *with high probability*[2] (w.h.p.) by layering multiple linked lists. The bottommost level is a standard sorted linked list, while higher levels serve as "express lanes," allowing searches to skip over multiple elements at a time.

The skiplist is parameterized by a ***promotion probability*** $p$, which determines the likelihood of elements appearing in higher levels of linked lists. Upon insertion, an element is randomly (using coin tosses with probability of heads $p$) assigned a height equal to the number of successive coin tosses until heads for that particular

[1]Section 5 contains all details about the experimental setup.

[2]An event $E_n$ on a problem of size $n$ occurs ***with high probabilty*** if $Pr[E_n] \geq 1 - 1/n^c$ for some constant $c$.




*Conference acronym 'XX, Woodstock, NY*





ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

element. The height of an element is the number of linked-list levels that element appears in, starting from the lowest level.

**Theoretical steps towards a cache-optimized skiplist.** We will use the I/O model (or external-memory model) [3], which measures how well an algorithm or data structure takes advantage of spatial locality by measuring cache-line transfers. The model is parameterized by a cache-line size $Z$. Transferring $Z$ contiguous elements in one cache line has unit cost in the model.

Theoretically, given some node size $B = \Theta(Z)$, the straightforward way to improve the cache-friendliness of skiplists is to promote elements with probability $1/B$ rather than $1/2$ as in traditional skiplists. Indeed, Golovin [12] proposed this method with the ***B-skiplist***, a cache-optimized skiplist that matches the B-tree's bounds *in expectation*. For each level in this theoretical B-skiplist, consecutive unpromoted elements are stored in the same node. However, this initial paper on B-skiplists stops short of providing theoretical guarantees w.h.p., parallelization, and implementation, leaving a massive gap in making B-skiplists practical.

**Challenges to blocking skiplists.** In practice, addressing locality issues in skiplists via blocking (i.e., storing multiple elements per node) raises challenges due to the randomized *variable size of nodes*. Theoretically, each node in a B-skiplist has $\Theta(B)$ elements in expectation, but there exist nodes with as many as $\Theta(B \log n)$ elements w.h.p. [5, 6]. Since nodes can be large, finds and inserts, which require scanning and potentially shifting a linear number of elements in a node, can cost as much as $O(\log n)$ cache-line transfers in the I/O model, matching a regular skiplist's randomized bounds.

**Bounding the element moves in a B-skiplist.** To mitigate this issue in practice, we enforce ***fixed-size*** physical nodes in the B-skiplist to bound the maximum number of element moves during insertions. B-skiplist nodes are allowed to grow to arbitrary sizes according to the results of randomized promotions. However, if a logical B-skiplist node contains $k > B$ elements, we physically store it as $\lceil k/B \rceil$ nodes at the same level connected by pointers.

The design choice of fixed-size nodes is subtle but is key to fast inserts in practice because it limits the worst-case number of cache-line writes to $\Theta(1)$ per level. In contrast, if physical nodes are allowed to grow arbitrarily, the number of cache-line writes in any node is $O(\log n)$ w.h.p. To be clear, this choice does not affect the overall insertion bound in the I/O model as each insert still requires $O(\log n)$ cache-line reads w.h.p. to determine the correct position. However, as we shall see in the empirical evaluation, fixed-size nodes enable the B-skiplist to support fast insertions and low variance in the latency of operations.

**Prior steps towards concurrent blocked skiplists.** Furthermore, any candidate for the in-memory index in databases, including the proposed B-skiplist, must be *concurrent* to take full advantage of parallel resources in today's multicore machines.

On the practical side, there have been several steps towards improving locality in concurrent skiplists [24, 31], but these often involve periodic rebuilding of the upper levels, increasing the worst-case latency of individual operations. For example, CSSL [30] and PI [31] periodically rebuild the upper levels of the index, blocking reads and writes during the restructuring process.

Other blocked skiplists vary the component data structures and CC schemes at different levels, giving up on the simplicity of the original concurrent skiplists. For example, ESL [24] first inserts elements only into the bottom level and later updates the upper index levels asynchronously with background threads. The ESL is composed of two levels with distinct index structures and CC mechanisms. Similarly, S3 [34], another cache-sensitive skiplist, employs a similar strategy of a two-level index with different CC schemes. Furthermore, it adaptively chooses "guard entries" with a neural model, giving up on the randomization of skiplists and therefore the skiplist's probabilistic theoretical guarantees.

**Designing a concurrent B-skiplist.** In this work, we focus on designing a *simple* and *effective* CC mechanism for B-skiplists without modifying its high-level structure. Specifically, we propose a CC mechanism based on fine-grained locking to achieve both high throughput and low worst-case latency. The ideal CC scheme for B-skiplists inherits the simplicity and performance of skiplist-based CC schemes [10, 15, 16, 27], so we use them as a starting point. However, they do not handle node splits and merges.

At a high level, skiplist insertion algorithms (and their corresponding CC mechanisms) make two traversals through the index if an element is promoted: one "top-down" read-only phase to determine the location(s) that an element should be inserted, and then a corresponding "bottom-up" write-only phase that links in the new skiplist nodes [10, 15, 16, 26, 27]. In both the top-down and bottom-up phase, a CC scheme for skiplists may either 1) hold a constant number of locks via hand-over-hand locking [27] or 2) hold locks on all levels that an element is promoted to [10, 15, 16, 26].

To reduce the number of traversals in both sequential and concurrent insertions, we introduce a ***top-down*** insertion[3] algorithm and that exploits inherent skiplist properties to traverse the B-skiplist *only once* (top-to-bottom and left-to-right). In contrast, existing skiplist insertion algorithms make two traversals if an element is promoted. As we shall see, the design of the insertion algorithm has major implications for the corresponding CC mechanism.

We build upon this top-down insertion algorithm to develop a *simple* yet *efficient* top-down CC scheme based on reader-writer locks [8], a common synchronization primitive for in-memory indexes. The top-down CC scheme minimizes overheads by 1) avoiding multiple retires from root-to-leaf, and 2) minimizing both the number of exclusive locks held at one time and the duration that the locks are held. Using the top-down CC scheme, a thread only needs to lock nodes on at most two layers at once, and locks only a constant number of nodes at once.

**Contributions.** The contributions of the paper are as follows:

- The design of the B-skiplist with fixed-size nodes to improve cache locality and mitigate worst-case behavior.
- A novel top-down concurrency control mechanism for B-skiplists built on a corresponding insertion algorithm for B-skiplists that completes insertions in one pass from top-to-bottom.
- A C++ implementation of the concurrent B-skiplist.
- An empirical evaluation of the concurrent B-skiplist compared to several in-memory skiplist-based and B-tree-based indexes that demonstrates that the B-skiplist achieves between 2×–9× higher throughput on YCSB workloads [7] compared to state-of-the-art concurrent skiplists.

[3]We focus on the case of insertions due to space constraints, but deletions are symmetric.

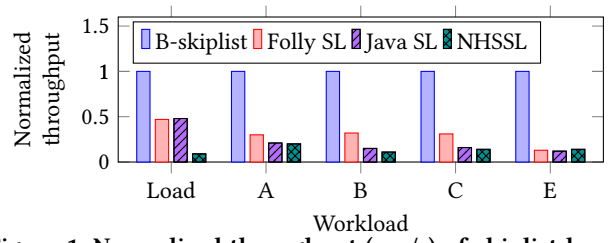


**Figure 1: Normalized throughput (ops/s) of skiplist-based indices relative to the B-skiplist with uniform random keys.**

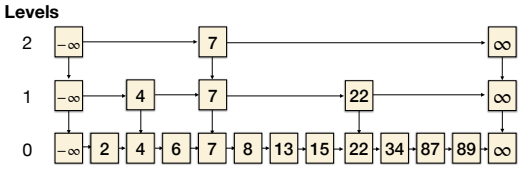


**Figure 2: Example of a skiplist.**

**Results summary.** Figure 1 demonstrates that the B-skiplist achieves between 2×–9× higher throughput on workloads from the popular YCSB [7] compared to state-of-the-art concurrent skiplists including Facebook's concurrent skiplist from the Folly library [1], the Java ConcurrentSkipListMap [18], and the No Hot Spot Skiplist (NHSSL) [9]. The YCSB workloads include a mix of point operations (finds/inserts) and range operations.

Furthermore, we also evaluate two tree-based indices: a state-of-the-art concurrent B-tree [32] and Masstree [22], a popular cache-optimized in-memory index. We find that the B-skiplist achieves competitive throughput (between 0.9×–1.7×) on point workloads, and between 0.7×–7.5× throughput on range workloads.

In addition to throughput, we evaluate all data structures on latency as well for a more complete picture of data-structure performance. As we shall detail in Section 5, the B-skiplist achieves between 3.5×–103× lower 99% latency compared to existing concurrent skiplists.

## 2 Preliminaries

This section will give background on the structure and operations of skiplists and B-skiplists necessary to understand later sections. For the operations, it will focus on inserts for simplicity, but deletes are symmetric. Furthermore, it will review reader-writer synchronization primitive, which is the core functionality that the proposed top-down single-pass concurrency control scheme for B-skiplists is based on.

**Operations.** A key-value dictionary data type stores pairs of keys and values `(k, v)`. Their main operations are as follows:

- `find(k)`: return the associated value `v`.
- `insert(k, v)`: add `(k, v)` to the data structure.
- `range(k, f, length)`: apply the function `f` to the `length` key-value pairs with the smallest keys that are at least `k`.

We consider these operations since they comprise the popular YCSB [7] workloads we use to perform the evaluation in Section 5.

### 2.1 Skiplist structure and operations

**Structure.** The skiplist [28] is a data structure that stores a *hierarchy* of levels of linked lists. Each linked list is sorted by the keys of the elements in that linked list, with sentinels at the beginning and end for $-\infty$ and $\infty$. The bottommost level (level 0) contains all of the elements in the data structure. The list size decreases by a constant factor in expectation at each successive level as we move up the hierarchy. Each linked list at some level $\ell > 0$ contains a subset of the elements in the linked list below it (level $\ell - 1$). This property is called *inclusion invariant*, where every element present at level $\ell$ must also be present at levels $0, \ldots, \ell - 1$.

Figure 2 illustrates the skiplist's pointer structure. Just like in trees, we will refer to the nodes at the lowest level of the skiplist ($\ell = 0$) as *leaf* nodes and those at higher levels ($\ell > 0$) as *internal* nodes. All nodes have a `next` pointer to a successor node in the same level, and all internal nodes also have a `down` pointer to the node with the same key in the level below. The `root` node is the leftmost node at the highest level.

An element that appears at some maximum level $\ell > 0$ is said to be ***promoted*** to that level. Promotions are determined upfront at the start of an insertion via randomization with a series of coin flips. Notably, the level that an element is promoted to in a skiplist is unrelated to the current structure of elements in the skiplist. That is, the highest level that an element appears at in a skiplist is equal to the number of successive "heads" seen when flipping a coin with some constant probabilty $p$ (usually $p = 1/2$, but any probabilty $1/c$ where $c$ is a constant sufficies to achieve the asymptotic bounds). Given a skiplist with $n$ elements, the maximum height of any element is $O(\log n)$ in expectation and with high probability.

**Operations.** All operations in a skiplist start at the upper left sentinel ($-\infty$) at the highest level and traverse through the pointer structure in a left-to-right and top-to-bottom fashion. The skiplist is a self-balancing data structure and does not require pointer rotation to maintain its bounds. A skiplist with $n$ elements supports all point operations (insert, delete, find) in $O(\log n)$ time in expectation and with high probability.

For ease of understanding, given a node with key `x` at a given level $\ell$, let its `pred` (predecessor) element be the largest key strictly less than `x`, and let its `succ` (successor) element be the smallest key strictly greater than `x` at level $\ell$. Except for the beginning and end sentinels, all nodes in a skiplist have logical `pred` and `succ` elements which correspond to nodes in the skiplist.

To `find` an element `k`, the traversal searches left-to-right starting from the left sentinel on the highest level. Upon finding its `succ` node at that level, the search follows the down pointer of the `pred` node. The search continues in this way until the lowest level, where it scans left to right until it either finds `k` or does not find it and encounters some element greater than `k`.

The `range` operation is a direct extension of a `find`. Rather than terminating at the search key `k` or its `succ` element in the leaf level, the `range` search continues a left-to-right search through the leaf layer until `length` elements have been read, or the search reaches the right sentinel.

Inserts are similar to finds in terms of traversal order, but must link in a new node containing some key `k` at each level that it is promoted to. Let $h \geq 0$ be the height that the newly inserted element

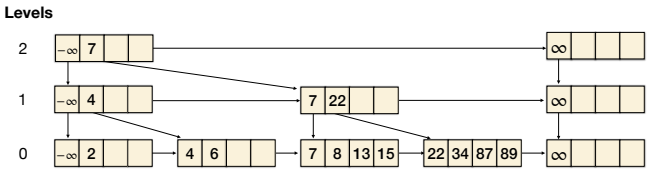


**Figure 3: Example of a B-skiplist with node size $B = 4$.**

is promoted to. One way to perform this structural modification is to keep track of the `pred` nodes of `k` at all levels during the downward traversal. After the search reaches the bottom level, the insert operation creates $h + 1$ new nodes containing `k` and adjusts the `next` pointers at each level so that `k->next = pred->next` and then `pred->next = k`. Furthermore, the insert must update the down pointers in each internal node with the key `k` to the node containing `k` in the next lowest level.

## 2.2 B-skiplist structure and operations

**Structure.** The B-skiplist is a "blocked" version of a skiplist that contains multiple elements per node [12]. The elements are totally sorted at each level of the B-skiplist (by pointer structure and within nodes). Given a cache-line size $Z$ and desired node size $B = \Theta(Z)$, the B-skiplist's promotion probability is generally set to $p = \Theta(1/B)$. In theory, the expected number of elements per node is $\Theta(B)$, but in a B-skiplist with $n$ elements, the maximum number of elements in a node is $\Theta(B \log n)$ w.h.p. [5, 6]. Golovin's original theoretical paper on B-skiplists [12] does not address how to handle this case in practice, but one straightforward solution is table doubling when the node becomes full.

The block structure depends on promoted height of each element in the B-skiplist. The *header* key of a node in a B-skiplist is the first (and smallest) element in that node. By construction, every header in a node at some level $\ell$ has been promoted to level $\ell + 1$.

Figure 3 illustrates the node and pointer structure of a B-skiplist. Just as in a skiplist, each node has a `next` pointer pointing to the next node. Given a node `x`, all keys in `x` are smaller than the header key of `x->next`. Furthermore, each internal node (at level $\ell > 0$) contains an array of $B$ `down` pointers (one per key in the node), each pointing to the corresponding node at the level below.

**Bounds.** In the external-memory model described in Section 1, given a cache-line size $Z$, a B-skiplist with $n$ elements and node size $B = \Theta(Z)$ supports finds and inserts in $\Theta(\log_Z(n))$ cache-line transfers in expectation (matching B-tree bounds). Furthermore, range queries with $r$ elements in the range take $\Theta(\log_Z(n) + r/Z)$ cache-line transfers in expectation. However, finds and inserts take $\Theta(\log n)$ cache-line transfers in the worst case w.h.p., matching the bounds of a non-blocked skiplist.

**Operations.** The left-to-right and top-to-bottom pointer traversal in a B-skiplist during finds and inserts are similar to operations in a skiplist. The main difference is that traversals must look at multiple elements within a single node and determine which `down` pointer to follow at an internal nodes.

To `find` an element `k` in a B-skiplist, the traversal begins at a `curr` node initialized to the upper left sentinel (with header $-\infty$), just as in the regular skiplist. The search then examines the header of `curr->next`. If it is less than `k`, the `curr` node is updated to `curr->next`. We repeat this left-to-right traversal until the header of `curr->next` is greater than `k`. At that point, we search within `curr` for the `pred` element and follow it's `down` pointer. The search continues in this way until we reach the leaf level, at which point we determine if `k` is present. The main difference from a traditional skiplist is that the `pred` element may be in the same node as `k`.

To `insert` an element in a B-skiplist, we first perform a `find`-like traversal to find all the `pred` elements at each level. However, the node-modification operations during a B-skiplist insert are similar to those in a B-tree. Let $h$ denote the height at which the key to be inserted `k` is promoted to. If it not promoted (i.e., $h = 0$), we can simply add it to the same node its `pred` element resides in and shift all subsequent elements in the node one slot down. However, if it is promoted (i.e., $h > 0$), we must perform a `split` at levels $\ell = 0, 1, \ldots, h - 1$. Let `old_node` be the node with `pred`. A node `split` in a B-skiplist creates a `new_node` with `k` as the header and copies all elements (and their down pointers, if the split occurs at an internal node) in `old_node` greater than `pred` (and also greater than `k`) after `k`. To link in the new node, we set `new_node->next = old_node->next` and `old_node->next = new_node`.

**Open problem.** Golovin's paper on B-skiplists does not address the *order* of levels in which an element is inserted (i.e., starting from the top or the bottom) [12]. However, most traditional skiplist insertion algorithms insert elements in a "bottom-up" fashion - they link in new nodes starting from the leaf level up until the maximum level an element is to be promoted to [15, 16, 27, 28]. A similar algorithm for B-skiplists would perform a find to the bottom level, keeping track of the affected nodes along the way, and insert the element starting from the leaf level, performing splits and adjusting pointers on the way up as necessary to always maintain the inclusion invariant.

## 2.3 Concurrency control primitives

**Hand-over-hand locking.** Next, we will review the classical *hand-over-hand* (HOH) fine-grained locking scheme (also known as *latch crabbing* in databases) for sorted singly-linked lists [17], which we will be building upon in later sections. HOH traverses a list while holding at most two locks at a time, starting at the `head` node of the linked list, then acquiring the lock on the successor before releasing the lock on the predecessor. After locking a node `curr`, it is safe to access `curr->next` to either perform a comparison with a target key (e.g., for a search), or to see whether we have reached the end of the list. During inserts, a traversal must hold two locks at a time to link in a new node in the correct place between a node `curr` and `curr->next`. HOH locking can also be extended to skiplists because skiplists are simply towers of linked lists.

**Reader-writer locks.** Finally, we will review reader-writer locks (RW locks) [8], as we will use them as the core synchronization primitive in the proposed concurrent B-skiplist. RW locks enable concurrent access for read-only operations but exclusive access for write operations. Given a RW lock, a thread can either call `read_lock()` to access it in shared read mode, or `write_lock()` to access it in exclusive write mode. Multiple threads can read the data in parallel if they all hold the lock in read mode, but all other threads must wait if a thread holds the lock in write mode.

RW locks are the foundation of many concurrency control protocols in B-trees, including the classical optimistic concurrency control (OCC) [19].

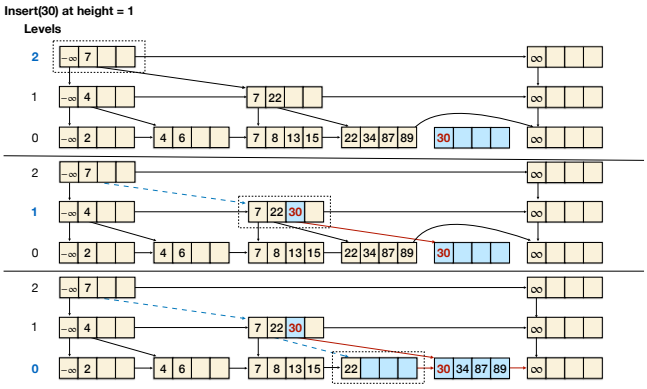


**Figure 4: Example of a top-down insertion of key k = 30 and height $h = 1$ into a B-skiplist with node size $B = 4$. The dotted box represents the current node visited during the traversal, the blue dashed lines denote the pointers followed during the top-down traversal, and the blue cells represent the cells written during the insertion.**

## 3 Top-Down Insertion Algorithm

This section presents the ***top-down*** insertion algorithm that will form the basis of the concurrent B-skiplist in the next section. For ease of understanding, we will first describe the algorithm serially and show that it results in the same B-skiplist structure as the original "bottom-up" insertion algorithm. In the next section, we will introduce a corresponding top-down concurrency control scheme. Furthermore, we will show that the proposed insertion algorithm achieves the same asymptotic runtime bounds of the theoretical B-skiplist. For simplicity, we will describe it for keys only, but storing associated values involves only updating the leaf level. We will initially describe the insertion algorithm using logical nodes for simplicity and then explain how to adapt it with fixed-size physical nodes.

**Description.** At a high level, the main change in the proposed "top-down" insertion algorithm compared to existing "bottom-up" algorithms is the order in which new elements are added to the skiplist. Although the distinction about the order of insertion into levels is subtle, it has important implications for the traversal pattern, and as we shall see in Section 4, for CC mechanisms.

The goal of the top-down algorithm is to complete insertions in *one pass* of the skiplist without revisiting nodes by taking advantage of the skiplist property that the height at which an element is promoted to is determined *upfront* and is *independent of the current structure*. In contrast to promotions in a B-tree, which depend on the current structure and fullness of the nodes, element promotions in a B-skiplist depend only on a sequence of random coin flips.

Rather than traversing down to the leaf level and then linking in elements from bottom to top, as in most skiplist insertion algorithms, we propose to insert elements starting from the highest level at which they are promoted and ending with the leaf level. Therefore, an insertion is finished once the traversal reaches the leaf level and performs a write at the relevant node.

In contrast, as described in Section 2, a "bottom-up" insertion algorithm for B-skiplists would first insert the element at the leaf level and work its way up the levels, performing splits as necessary until the level that the element should be promoted to. Such an algorithm requires a "top-down" read-only pass to the leaf level, and then a "bottom-up" write pass from the leaves upwards to insert the element into any relevant level(s). With a bottom-up strategy, an insertion is not necessarily complete once the traversal reaches the leaf level, since the element may be promoted to higher levels.

Concretely, suppose we are inserting an element k that will be promoted to level $h \geq 0$ as determined by random coin flips. We traverse the B-skiplist from top to bottom starting at the leftmost sentinel at the top level as described in Section 2. If the traversal is currently on some level $\ell > h$, the element k will not appear on $\ell$, so the search order is exactly the same as an original B-skiplist `find`. By following `next` and `down` pointers, we will eventually reach level $h$, where we will write k into the same node as its predecessor `pred` at level $h$. If $h = 0$, the insertion is complete.

The main challenge comes when the element k is promoted to level $h > 0$ because the corresponding `down` pointer should point to a new node (due to a split) at level $h - 1$ that has not yet been created in a naive top-down traversal. Recall from Section 2 that in a B-skiplist, every header element in a node at level $i$ must have been promoted to level $i + 1$. It is clear which node to set the `down` pointer to with a "bottom-up" algorithm, because the corresponding node has been created at level $h - 1$ before the `down` pointer at level $h$ needs to be set. However, with the proposed "top-down" algorithm, the node with the new element k has not yet been created at level $h - 1$ since we are going from $h$ down to 0.

We can resolve the issue by again exploiting the property of B-skiplist insertions that the height of every key is determined upfront to *allocate all the nodes in advance* that will be created during an insertion. That is, since by construction, an element promoted to height $h > 0$ will generate new nodes at levels $0, 1, \ldots, h - 1$, we can allocate $h$ new nodes with k as the header at the start of any insertion before any interaction with the skiplist at all. If $h > 1$, we can link these preallocated nodes together in a stack via `down` pointers in non-leaf nodes. Therefore, we can fill in the appropriate slot in the `down` pointer array at level $h$ with a pointer to the top of the preallocated stack of new nodes.

To determine where to splice in the other new nodes in levels $0, 1, \ldots, h - 1$, we continue the traversal level-by-level. Let us consider the levels in turn, starting with level $h - 1$. Just as it has in the higher levels, the traversal will follow the `down` pointer corresponding to the `prev` element in level $h$ which will point to some node at level $h - 1$. At level $h - 1$, the traversal will proceed left-to-right until we find some node x such that `x->header < k` but `x->next->header > k`. Let n be the preallocated node destined to be at level $h - 1$. The insert should move all elements greater than k in node x to n. The new node n is then linked into the B-skiplist by setting `n->next = x->next` and `x->next = n`. The insertion then proceeds in this way until it reaches the leaf level by following the `down` pointer of the last element in x, which is the largest element at that level less than k. The element k will appear in the correct nodes with the correct structure at the end of this top-down method with splits on the way down.

Figure 4 contains a worked out example of a top-down insertion. Notice how in the example, since the element is promoted to height $h = 1$, the corresponding node at level 0 is allocated upfront and linked in with `down` and `next` pointers at levels 1 and 0, respectively.

**Correctness and bounds.** The remainder of the section will describe the differences resulting from the top-down insertion algorithm and how they do not affect the B-skiplist's correctness or asymptotic time bounds.

The main change with the proposed top-down insertion algorithm is a slight relaxation in the inclusion invariant, or the property that every element present in level $h$ must also be present at all lower levels, only *during* insertions. It still preserves the property upon every insertion's completion by linking in the preallocated new nodes via `down` and `next` pointers.

The inclusion invariant is necessary for the B-skiplist to achieve its asymptotic bounds, because the bounds come from the expected number of levels and the expected number of nodes traversed in each level. These properties are derived from the probabilistic coin flips that determine each element's height in the B-skiplist. However, after each insertion, the pointer structure in the B-skiplist with the top-down insert algorithm is identical to what it would have been with a bottom-up algorithm that links in nodes starting from the leaf level. Therefore, both the number of levels and the number of nodes traversed per level are unaffected.

**Fixed-size nodes.** As mentioned in Section 1, due to the B-skiplist's randomized structure, any practical B-skiplist implementation must handle the case where the number of elements in a node exceeds some fixed-size array allocation. The number of keys that are supposed to be in a node may exceed a fixed bound, depending on the sequence of coin flips. Given a B-skiplist with $n$ elements and promotion probability $p = 1/B$, the worst-case number of elements in a given node is $\Theta(B \log n)$ w.h.p. Therefore, it is highly likely that there will be nodes that exceed any fixed size $\Theta(B)$.

The naive solution of table doubling in nodes enables all keys meant for a node to fit in the corresponding array, but leads to suboptimal performance of insertions. As mentioned in Sections 1 and 2, w.h.p., there is a node in the B-skiplist with $k = \Theta(B \log n)$ elements, so the worst-case insertion cost in a B-skiplist is $\Theta(\log n)$ cache-line accesses w.h.p.

To alleviate this issue, we modify the B-skiplist design to require fixed-size node allocations, which may potentially result in node ***overflow splits*** (i.e., node splits due to overflow) in addition to ***promotion splits*** due to randomization. If nodes were allowed to grow arbitrarily large, the worst-case number of elements moved (i.e., shifted to maintain sorted order) during an insert is $\Theta(B \log n)$, which would take $\Theta(\log n)$ cache misses. However, by requiring that the nodes have at most $B$ elements, the number of element moves in any node is at most $B$.

The nodes created from overflow splits do not affect the correctness of inserts or searches, as operations in the B-skiplist still follow a left-to-right and top-to-bottom traversal order. As mentioned in Section 1, the choice of fixed-size nodes does not affect the theoretical bounds, because a query would still have to perform $\Theta(\log n)$ cache-line transfers in the worst case w.h.p. However, minimizing the cost of inserts is important for practical efficiency, as element moves must be linear in the node size, while queries can skip over parts of the node e.g., via binary search.

# 4 Top-down concurrency control

This section presents a single-pass top-down concurrency control scheme for B-skiplists based on reader-writer locks and the top-down insert scheme described in Section 3. The goal of the proposed scheme is *simplicity* in the locking protocol in both the *number of top-down traversals* and the *number of locks held at a time*. Specifically, we will show that each operation only needs to make a single root-to-leaf traversal. Furthermore, this traversal only holds a constant number of locks in at most two levels of the B-skiplist at a time. Due to space reasons, we will include the proof of deadlock-freedom in the full version.

**Concurrent finds and range queries.** Let us start with how to implement concurrent finds and range queries with RW concurrency as an intermediate step to understanding concurrent inserts in B-skiplists. Since finds and range queries are read-only operations, they only need to acquire locks in reader mode[4]. Queries begin on the highest level at the left sentinel and proceed in a hand-over-hand fashion left-to-right, as described in the HOH scheme for linked lists in Section 2. When the search reaches the node with the appropriate `prev` element, it acquires the child node at the next lowest level via the `down` pointer using HOH locking in a top-down fashion. Searches proceed left-to-right within a level and top-to-down to move between one level at a time until the query reaches the appropriate node in the leaf level that should contain the target key. For point finds, the search is then complete and can release all locks. In contrast, range queries acquire locks left-to-right at the leaf level in a HOH fashion until the range is exhausted or the search reaches the end of the skiplist.

To recap, both concurrent finds and range queries acquire RW locks in read mode in a left-to-right and top-to-bottom order in HOH fashion. That is, a thread holds at most two locks at a time.

**Concurrent inserts.** Next, we will introduce the proposed top-down concurrency control scheme for inserts. At a high level, insertions follow a similar left-to-right and top-to-bottom traversal order to queries. However, inserts raise additional challenges, since they potentially require structural modification operations (i.e., splits) if elements are promoted to higher levels. Figure 5 illustrates a worked example of the sequence of reader-writer lock acquisitions and node updates with the top-down concurrency control protocol.

Just as in the insertion algorithm from Section 3, the proposed top-down concurrency control scheme leverages the randomized property of skiplists to complete insertions in *one pass* through the data structure and to acquire writer locks *only at the levels where writes will occur*. That is, it relies on the observation that the level to which an element is promoted to in a B-skiplist depends only on a sequence of random coin flips and importantly, is can be determined upfront independently of the current structure of the skiplist.

For concreteness, suppose that we are inserting some key `k` that will be promoted to level $h$. Furthermore, suppose that we have preallocated any new nodes that will be spliced into the skiplist as described in Section 3. Since they are not currently linked in the skiplist, taking their write locks will not delay any other threads.

Inserts begin at the highest level and proceed just like queries for all levels greater than $h$. Since there will be no writes until level

[4]Reader locks are necessary in mixed insert-query workloads.

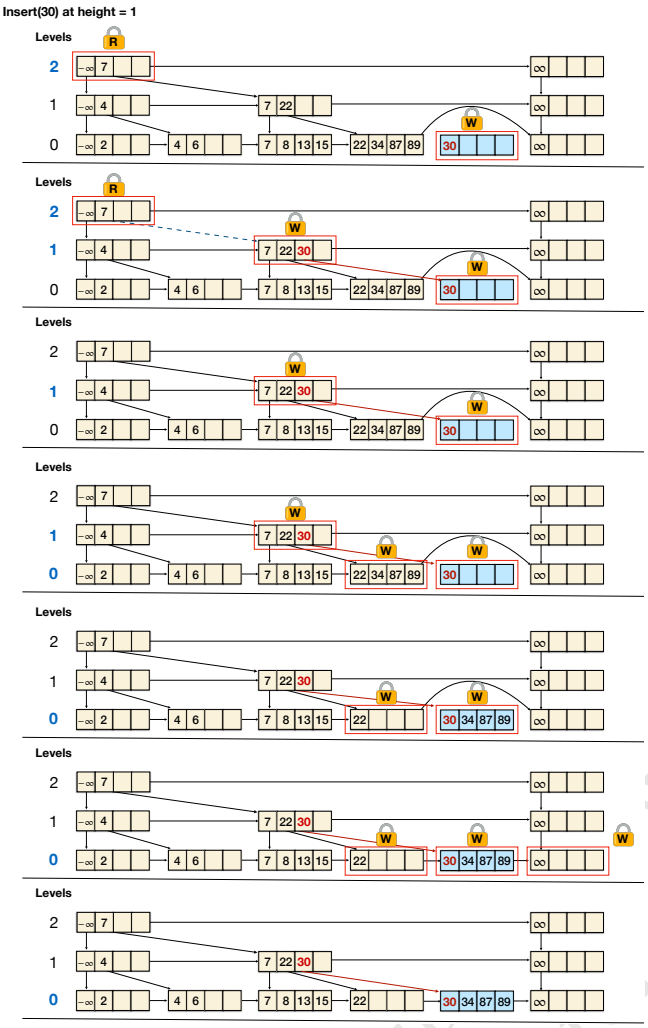


**Figure 5: Example of the insertion of element with key k = 30 and height $h = 1$ into a B-skiplist with node size $B = 4$ using the top-down CC scheme. The locks with R and W denote acquiring the lock in read and write mode, respectively.**

$h$, the traversal only needs to acquire locks in read mode in the left-to-right and top-to-bottom traversal until it reaches level $h$.

Once it has reached the appropriate node with the correct down pointer in level $h+1$, the insert then acquires the lock on the child node in write mode on level $h$. At level $h$, the traversal will continue left-to-right in a HOH fashion taking locks in write mode until we reach the node where k should be inserted (i.e., some node x such that `x->header < k` and `x->next->header > k`). Since we have the write lock on x, we can directly insert k in the appropriate slot. If $h = 0$, the insert is completed, and we can release all of the locks.

If there need to be splits at lower levels (i.e., $h > 0$), we use HOH locking from top-to-bottom and left-to-right. Consider the case of a split at level $h-1$. To find the starting point traversal in level $h-1$, we first take the write lock on the node pointed to the by down pointer associated with the `prev` element in level $h$. Just as in the single-threaded case, the traversal will then proceed left-to-right until we find the two nodes that the new preallocated node will need to be inserted between. However, in the concurrent case, we take writer locks in HOH fashion left-to-right for thread safety. The actual split mechanism is unchanged from the serial case, and this process is repeated until the leaf level.

**Integration with fixed-size nodes.** So far, we have described the CC mechanism on logical nodes, but overflow splits do not affect the guarantees of HOH locking. An insert still only needs to acquire a constant number of locks at any time. The CC protocol in the case of overflow splits is even simpler than the case of promotion splits. Recall that if an element is promoted, the top-down scheme acquires at most three locks at once to perform splits: one to update the down pointer at some level $\ell$ and two to splice in the new node between two existing nodes at level $\ell - 1$). In overflow splits, there is no need for the lock to be held on level $\ell$ because there is no down pointer to the new node, so we only need to hold the two locks on the nodes that we are splicing the new node between.

**Correctness and deadlock-freedom.** The correctness of the proposed HOH-based locking scheme follows directly from existing theory about HOH locking in linked lists and skiplists [17]. To insert an element into a node without a split, just acquiring the write lock on that node is sufficient. To split a node and insert a new node between two existing ones, acquiring the write lock on both the predecessor node and its `next` node is sufficient. There is a slight relaxation of the maximum number of locks held at once from two to three during splits. However, locks are held on at most two levels at once.

Finally, the proposed top-down concurrency scheme is deadlock-free because there is a *total ordering* on locks from left-to-right within levels and then top-to-bottom between the levels, thereby avoiding circular wait.

## 5 Evaluation

This section evaluates the proposed concurrent B-skiplist on the YCSB application benchmark [7] compared to several state-of-the-art concurrent skiplist- and tree-based indices. As mentioned in Section 1, we evaluate all indices in terms of both *throughput* and *latency*. Due to space limitations, we omit the raw data tables, but will include them in the full version of the paper.

**Result summary.** At a high level, the B-skiplist achieves 2×–9× higher throughput and 3.5×–103× lower latency than non-cache-optimized concurrent skiplists. Furthermore, the B-skiplist achieves 0.7×–7.5× throughput on point workloads and 0.7× throughput on range workloads compared to tree-based indices.

**Systems setup.** All experiments were run on a server with 64-core 2-way hyperthreaded Intel Xeon Gold 6338 CPU @ 2.00GHz with 1008 GB of memory. The server has a 3 MiB L1 data cache, a 2 MiB L1 instruction cache, a 80 MiB L2 cache and a 96 MiB L3 cache. We ran all experiments with 64 physical cores and 128 hyperthreads.

All times are the median of 5 trials after one warm-up trial. To measure latency, each thread measures the average time taken for a batch of ten operations[5] and stores it in a thread-safe vector. This allows us to sort and calculate the latency at each percentile after running each benchmark.

[5]We measure the average of 10 operations instead of individual operation to preserve the contention between threads.

| Workload | Description |
|---|---|
| Load | 100% inserts from empty |
| A | 50% finds, 50% inserts |
| B | 95% finds, 5% inserts |
| C | 100% finds |
| E | 95% short range iterations (`max_len` = 100), 5% inserts |

**Table 2: YCSB workload descriptions.**

We measured the cache misses in Table 1 with `perf`.

**Workloads.** Table 2 presents details of the core workloads from YCSB [33]. We tested workloads[6] A, B, C, and E from the core YCSB workloads generated with RECIPE [20].

We follow the standard YCSB procedure which consists of two phases: 1) the *load* phase-where elements are inserted into the data structure-and 2) the *run* phase-where operations are executed according to the workload's find/insert ratio. Each workload consists of 100 million (100M) elements inserted during the load phase, followed by 100M operations executed during the run phase, both phases are run concurrently.

We evaluate each workload under both uniform random and zipfian distribution in the run phase. In the uniform workload, the elements in both the load and run phases are generated from a uniform distribution. In the zipfian workload, the load phase elements are generated from a uniform distribution, while the elements in the run phase are generated from a zipfian distribution.

We omit the results on the zipfian distirbution due to space limitations, but the results were similar (on average within 20%) to the uniform distribution. We will include results from both distributions in the full paper.

**B-skiplist setup.** We implemented the concurrent B-skiplist in C++ with an open-source reader-writer lock library [2]. The test driver executes concurrent operations using pthreads [25]. We ran the B-skiplist with 8-byte keys and 8-byte values (for 16-byte key-value pairs). We set the max height of the B-skiplist to 5 in our tests.

We compiled the B-skiplist using g++ 11.4.0 with -O3.

**B-skiplist sensitivity analysis.** We performed a parameter sweep over promotion probability and node size in the B-skiplist to empirically determine which settings yield the best performance on the YCSB workloads. Due to space limitations, we will include the complete data in the full version of the paper. Golovin's theoretical paper on B-skiplists proposes a scaling factor, some constant $c$, on the promotion probability for $p = 1/cB$. Therefore, for each tested node size we also experiment with $c \in \{0.5, 1.0, 2.0\}$. size In theory, any constant $c$ should suffice to achieve the theoretical randomized B-skiplist bounds [12].

Based on the results of this sensitivity analysis, we set the node size in the B-skiplist to 2048 bytes (i.e., 128 key-value pairs) and $c = 0.5$, for promotion probability $p = 1/(0.5 \times 128) = 1/64$.

## 5.1 Comparison to skiplist-based indices

In this section, we evaluate the B-skiplist against No Hot Spot Skip List [9], Java ConcurrentSkipListMap [18], and Facebook Folly's ConcurrentSkipList [1] on the YCSB workloads [33] and report the results in Figures 1 and 6.

[6]We omit workload D from YCSB because it benchmarks the read-latest operation, which is not the focus of this work.

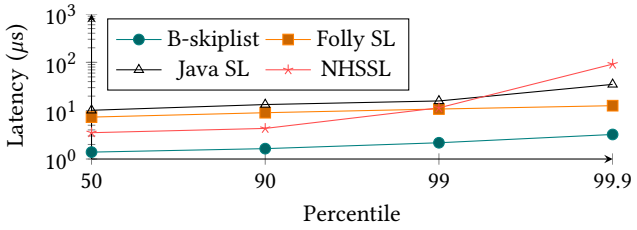


**Figure 6: Latencies of skiplist-based indices at different percentiles in YCSB workload A with uniform random keys.**

**Systems setup.** Java ConcurrentSkipListMap (JSL) implements a concurrent skiplist using a tree-like 2D linked skiplist [23].All operations except range queries are done natively. We implement range queries with the `subMap` interface.

Facebook's Folly (FLY) library provides a C++ concurrent skiplist[7]. We used the native interface for all point operations and the iterator interface for range queries (since it does not have native support). Folly's skiplist does not support values so we store only the keys.

No Hot Spot Skip List[8] (NHS) is a concurrent, lock free skiplist in C++ that relies on a background *adaptation* thread to maintains the structure of the skiplist and manage garbage collection. This includes rebalancing the upper index level to ensure traversals can be done in $O(\log n)$ time. NHS takes in a *sleep time* parameter that determines how frequent the background thread checks the index and modifies it. In the load phase, we set this parameter to a relatively small value (100 microseconds) to ensure the index is frequently balanced. After the load phase, we must wait for the background thread to balance the height of the tree to $\lg n$ to ensure that operations in the run phase achieve the desired performance. In the run phase, we set the sleep time higher to 1 second to ensure the background thread does not stall operations. We do not count the rebalance time between the load and run phases.

We compiled FLY and NHS using g++ 11.4.0 with -O3 and Java ConcurrentSkipListMap using javac 11.0.25.

**Discussion.** Figures 1 and 6 illustrate the throughputs and various percentile latencies of the skiplists.

As shown in Figure 1, the B-skiplist significantly improves upon the throughput of the other skiplists on all operations. For the load phase (all inserts), B-skiplist achieves about 2×, 11×, and 2.1× higher throughput, respectively, compared to the Java skiplist (JSL), the no hot spot skiplist (NHS), and the Folly skiplist (FLY). On point workloads (Workloads A, B, and C) with finds/inserts, the B-skiplist is about 4.6× – 6.6× faster than JSL, 5× – 9× faster than NHS, and 3.1× – 3.3× faster than FLY. For range queries (Workload E), the B-skiplist is about 9× faster than JSL and about 6× faster than NHS and FLY. All systems achieves better throughputs on the zipfian workloads than on uniform workloads, but the their relative performance to each other remains almost the same.

Additionally, as shown in Figure 6, the B-skiplist achieves lower latency across all workloads in all tested percentiles (50%, 90%, 99%, and 99.9%). The most competitive non-blocked skiplist is the one

[7]https://github.com/facebook/folly/blob/main/folly
[8]https://github.com/wangziqi2016/index-microbench/tree/master/nohotspot-skiplist

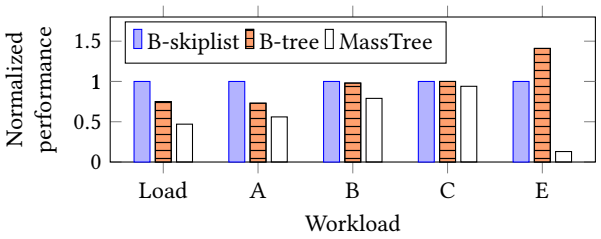


**Figure 7: Normalized throughput (ops/s) of tree-based indices relative to the B-skiplist with uniform random keys.**

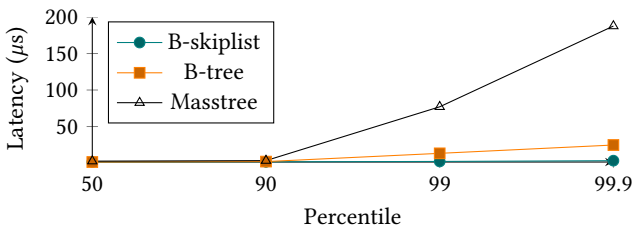


**Figure 8: Percentile latencies of the B-skiplist and tree-based indices on YCSB workload A with uniform random keys.**

from Folly, which has at least 3.5× and often at least 3.7× higher latency compared to the B-skiplist in the different percentiles.

The B-skiplist achieves better throughput and latency compared to unblocked skiplists because it reduces cache misses with better spatial locality in the nodes, as shown in Table 1. The folly skiplist, the fastest of the state-of-the-art skiplists, incurs between 3.2×-5.6× more cache misses compared to the B-skiplist. Furthermore, the B-skiplist maintains the simple structure and CC schemes that make the skiplist a popular choice for in-memory indexing.

## 5.2 Comparison to tree-based indices

**Systems setup.** We compare the B-skiplist with a high-performance concurrent B+-tree[9] [32] (a common B-tree variant) based on optimistic concurrency control [19]. The default configuration for the B+-tree sets `node_size = 1024` bytes. Both the concurrent B-skiplist and concurrent B+-tree use the same RW lock.

We also compare the B-skiplist with Masstree[10] [22], a cache friendly B+-tree variant. It utilizes an optimistic concurrency scheme as well. There is native support for all YCSB operations.

We compiled the B+-tree and Masstree with g++ 11.4.0 and -O3.

**Discussion.** Figures 7 and 8 compare the B-skiplist to tree-based indices on throughput and latency, respectively. We tested both uniform and zipfian datasets but only illustrate the performance on uniform as the results are similar.

Overall, we find that the B-skiplist achieves competitive (between 1×–1.4× higher) throughput compared to the B-tree and between 1×–2.1× higher throughput than Masstree on point workloads (load and A-C). We expect the B-tree and B-skiplist to be similar on read-heavy workloads because the B-skiplist has a similar height and node size compared to the B-tree. Furthermore, the B-tree only has to read one node per level, while the B-skiplist may have to take horizontal steps (following `next` pointers) along each level. Concretely, we found that on average, the B-skiplist takes about 1.7 horizontal steps per level in workloads A-C. As a result, the B-skiplist has a slightly lower throughput (within 0.9×) compared to the B-tree on workload C.

The B-skiplist achieves the most consistent speedups (1.3×–2.1× higher throughput) over tree-based indices on insert-heavy workloads (load and A). To understand why, we will first briefly review optimistic concurrency control (OCC) [19], the CC scheme in both the B-tree and Masstree. OCC is a classical CC scheme for B-trees based on RW locks that leverages the observation that most insertions only affect the leaf level. Almost all insertions under OCC make one root-to-leaf pass with reader locks on the internal nodes. However, if an element must be promoted, the insert *retires* back to the root, taking write locks on all nodes on the way down. In contrast, the top-down CC scheme in the B-skiplist is guaranteed to always make one pass from root-to-leaf and to take write locks only on the levels that the element is promoted to. To measure this difference, we counted the number of times the root lock was taken in write mode (blocking all other operations) during the load phase and workload A in the B+-tree and B-skiplist. In the load phase, the B+-tree root write lock was taken 26K times, compared to 7 times in the B-skiplist. In workload A, the B-tree took the write lock on the root about 8.3K times, compared to 3 times in the B-skiplist. The B+-tree also exhibits higher latency in the 99th percentile compared to the B-skiplist due to the B+-tree's retires back to the root.

On the other hand, the B-tree achieves about 1.4× higher throughput than the B-skiplist on range queries (Workload E). As mentioned earlier, although the B-skiplist has $\Theta(B)$ elements per node in expectation, the number of elements actually in a logical node can vary by up to a factor of $O(\log n)$. Therefore, the average density (number of elements in a node) is lower in the B-skiplist compared to the B+-tree, which deterministically splits nodes when they become full. To measure this difference, we counted the number of nodes at the bottom level traversed in both the B-skiplist and B+-tree during workload E. On average, the B-skiplist accesses about 2 nodes per range query while the B+-tree accesses only about 1.5 nodes per range query. We note that in-memory indexes are traditionally optimized for point (OLTP) workloads and range queries are often an optional function. Future work involves improving range queries in B-skiplists by improving the average node density.

## 5.3 Strong Scaling

Figures 9 and 10 shows the scaling performance of B-skiplist, Folly, B-tree, JSL, and Masstree on YCSB workload A and C with uniform random keys. We omit NHS because it times out on lower thread counts.

On the write-heavy workload A, all systems except Masstree scale number of threads increases. Masstree's performance peaks at 32 threads and declines afterward. On 128 threads, the B-skiplist, B-tree, and Folly skiplist all achieve about 35×–38× speedup, while JSL achieves about 45× speedup. Although JSL achieves higher parallel scalability, its overall throughput is lower than the C++-based data structures because it does not employ blocking. In contrast, on

[9] https://github.com/wheatman/BP-Tree/tree/main/tlx-plain/container
[10] https://github.com/kohler/masstree-beta

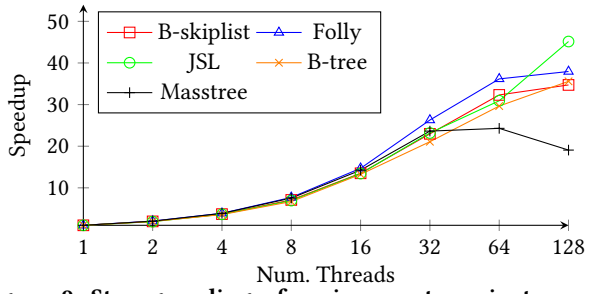


**Figure 9: Strong scaling of various systems in terms of throughput on YCSB workload A.**

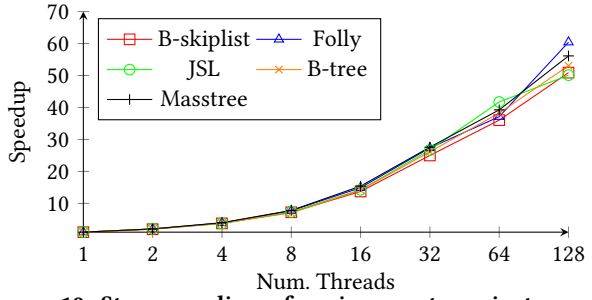


**Figure 10: Strong scaling of various systems in terms of throughput on YCSB workload C.**

the read-heavy workload C, all systems achieve higher speedup compared to workload A because there are no writes and therefore less lock contention. On workload C, all systems achieve between $50\times - 60\times$ speedup on 128 threads.

## 6 Conclusion

We present the B-skiplist, a high-performance concurrent in-memory index based on the skiplist data structure. The proposed concurrent B-skiplist adapts the theoretical description of a B-skiplist [12] for practical considerations to minimize data movement and mitigate the probabilistic worst case of element moves. To take advantage of parallel resources, we propose a top-down insertion algorithm that completes insertion in one pass and a corresponding simple yet effective CC scheme. The B-skiplist inherits the simple structure that make the skiplist a popular choice for in-memory indexing.

The empirical evaluation demonstrates that the B-skiplist achieves between 2×–9× higher throughput and between 3.5×–103× lower 99th percentile latency compared to popular state-of-the-art concurrent skiplist implementations such as those from Facebook's folly library and the Java concurrent skiplist library. These results suggest that the B-skiplist is a good candidate for high-performance in-memory indexing because it resolves locality issues in skiplists while minimizing CC overhead.

For future work, we plan to integrate the B-skiplist into key-value stores like RocksDB and LevelDB to evaluate its impact on application performance. Additionally, its high cache locality makes it well-suited for disk-based indexes. With its strong theoretical guarantees and practical efficiency, we anticipate that B-skiplist could match or even surpass traditional skiplists in modern databases.

## References

[1] [n. d.]. folly. https://github.com/facebook/folly. Last accessed 2/26/25.
[2] [n. d.]. ParallelTools. https://github.com/wheatman/ParallelTools. Last accessed 1/16/25.
[3] Alok Aggarwal and Jeffrey S. Vitter. 1988. The input/output complexity of sorting and related problems. *Commun. ACM* 31, 9 (Sept. 1988), 1116–1127.
[4] Rudolf Bayer and Edward M. McCreight. 1972. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica* 1, 3 (1972), 173–189.
[5] Michael A Bender, Jonathan W Berry, Rob Johnson, Thomas M Kroeger, Samuel McCauley, Cynthia A Phillips, Bertrand Simon, Shikha Singh, and David Zage. 2016. Anti-persistence on persistent storage: History-independent sparse tables and dictionaries. In *PODS*. 289–302.
[6] Michael A Bender, Martin Farach-Colton, Rob Johnson, Simon Mauras, Tyler Mayer, Cynthia A Phillips, and Helen Xu. 2017. Write-optimized skip lists. In *PODS*. 69–78.
[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC*. 143–154.
[8] Pierre-Jacques Courtois, Frans Heymans, and David Lorge Parnas. 1971. Concurrent control with "readers" and "writers". *Commun. ACM* 14, 10 (1971), 667–668.
[9] Tyler Crain, Vincent Gramoli, and Michel Raynal. 2013. No hot spot non-blocking skip list. In *ICDCS*. IEEE, 196–205.
[10] Mikhail Fomitchev and Eric Ruppert. 2004. Lock-free linked lists and skip lists. In *PODC*. 50–59.
[11] Keir Fraser. 2004. *Practical lock-freedom.* Technical Report. University of Cambridge, Computer Laboratory.
[12] Daniel Golovin. 2010. The B-skip-list: A simpler uniquely represented alternative to B-trees. *arXiv preprint arXiv:1005.0662* (2010).
[13] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. 1994. Quickly generating billion-record synthetic databases. In *SIGMOD*. 243–252.
[14] HBase. [n. d.]. https://hbase.apache.org/. Last accessed 10/20/2022.
[15] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. 2006. A provably correct scalable concurrent skip list. In *OPODIS*, Vol. 103.
[16] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. 2007. A simple optimistic skiplist algorithm. In *SIROCCO*. Springer, 124–138.
[17] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. 2020. *The art of multiprocessor programming*. Newnes.
[18] Java. [n. d.]. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentSkipListSet.html. Last accessed 1/3/2025.
[19] H.T. Kung and John T. Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)* 6, 2 (1981), 213–226.
[20] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. RECIPE: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *SOSP*. Ontario, Canada, 462–477.
[21] LevelDB. [n. d.]. https://github.com/google/leveldb. Last accessed 10/20/2022.
[22] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *EuroSys*. 183–196.
[23] Maged M. Michael. 2002. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*. 73–82. doi:10.1145/564870.564881
[24] Yedam Na, Bonmoo Koo, Taeyoon Park, Jonghyeok Park, and Wook-Hee Kim. 2023. ESL: A High-Performance Skiplist with Express Lane. *Applied Sciences* 13, 17 (2023), 9925.
[25] Bradford Nichols, Dick Buttlar, Jacqueline Farrell, and Jackie Farrell. 1996. *Pthreads programming: A POSIX standard for better multiprocessing.* " O'Reilly Media, Inc.".
[26] Kenneth Platz, Neeraj Mittal, and S Venkatesan. 2019. Concurrent unrolled skiplist. In *ICDCS*. IEEE, 1579–1589.
[27] William Pugh. 1990. *Concurrent maintenance of skip lists.* Citeseer.
[28] William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM* 33, 6 (1990), 668–676.
[29] RocksDB. [n. d.]. http://rocksdb.org/. Last accessed 10/20/2022.
[30] Stefan Sprenger, Steffen Zeuch, and Ulf Leser. 2016. Cache-sensitive skip list: Efficient range queries on modern CPUs. In *Data Management on New Hardware*. Springer, 1–17.
[31] Zhongle Xie, Qingchao Cai, HV Jagadish, Beng Chin Ooi, and Weng-Fai Wong. 2017. Parallelizing skip lists for in-memory multi-core database systems. In *ICDE*. IEEE, 119–122.
[32] Helen Xu, Amanda Li, Brian Wheatman, Manoj Marneni, and Prashant Pandey. 2023. BP-Tree: Overcoming the Point-Range Operation Tradeoff for In-Memory B-Trees. *Proc. VLDB Endow.* 16, 11 (July 2023), 2976–2989. doi:10.14778/3611479.3611502
[33] YCSB. [n. d.]. Core Workloads. https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads. Last accessed 2/15/2023.
[34] Jingtian Zhang, Sai Wu, Zeyuan Tan, Gang Chen, Zhushi Cheng, Wei Cao, Yusong Gao, and Xiaojie Feng. 2019. S3: a scalable in-memory skip-list index for key-value store. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2183–2194.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009