BP-tree: Overcoming the Point-Range Operation Tradeoff for In-Memory B-trees

Helen Xu
Lawrence Berkeley National Laboratory
hjxu@lbl.gov

Amanda Li
Massachusetts Institute of Technology
amandali@mit.edu

Brian Wheatman
Johns Hopkins University
wheatman@cs.jhu.edu

Manoj Marneni
University of Utah
u1320407@utah.edu

Prashant Pandey
University of Utah
pandey@cs.utah.edu

ABSTRACT

B-trees are the go-to data structure for in-memory indexes in databases and storage systems. B-trees support both point operations (i.e., inserts and finds) and range operations (i.e., iterators and maps). However, there is an inherent tradeoff between point and range operations since the optimal node size for point operations is much smaller than the optimal node size for range operations. Existing implementations use a relatively small node size to achieve fast point operations at the cost of range operation throughput.

We present the ***BP-tree***, a variant of the B-tree, that overcomes the decades-old point-range operation tradeoff in traditional B-trees. In the BP-tree, the leaf nodes are much larger in size than the internal nodes to support faster range scans. To avoid any slowdown in point operations due to large leaf nodes, we introduce a new insert-optimized array called the ***buffered partitioned array*** (BPA) to efficiently organize data in leaf nodes. The BPA supports fast insertions by delaying ordering the keys in the array. This results in much faster range operations and faster point operations at the same time in the BP-tree.

Our experiments show that on 48 hyperthreads, on workloads generated from the Yahoo! Cloud Serving Benchmark (YCSB), the BP-tree supports similar or faster point operation throughput (between .94×−1.2× faster) compared to Masstree and OpenBw-tree, two state-of-the-art in-memory key-value (KV) stores. On a YCSB workload with short scans, the BP-tree is about 7.4× faster than Masstree and 1.6× faster than OpenBw-tree. Furthermore, we extend the YCSB to add large range workloads, commonly found in database applications, and show that the BP-tree is 30× faster than Masstree and 2.5× faster than OpenBw-tree.

We also provide a reference implementation for a concurrent B$^+$-tree and find that the BP-tree supports faster (between 1.03×−1.2× faster) point operations when compared to the best-case configuration for B$^+$-trees for point operations while supporting similar performance (about .95× as fast) on short range operations and faster (about 1.3× faster) long range operations.

PVLDB Reference Format:
Helen Xu, Amanda Li, Brian Wheatman, Manoj Marneni, and Prashant Pandey. BP-tree: Overcoming the Point-Range Operation Tradeoff for In-Memory B-trees. PVLDB, 16(11): 2976 - 2989, 2023.
doi:10.14778/3611479.3611502

Proceedings of the VLDB Endowment, Vol. 16, No. 11 ISSN 2150-8097.
doi:10.14778/3611479.3611502

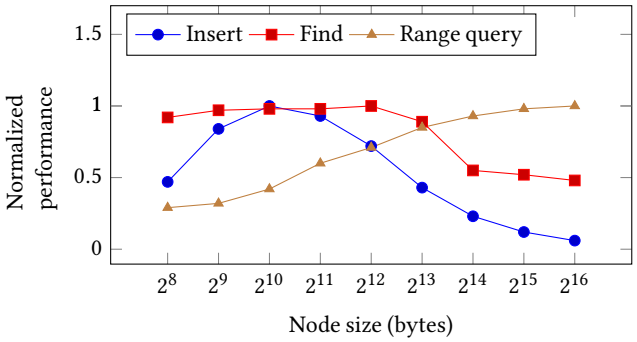


Figure 1: Normalized performance compared to the fastest configuration (for each operation) in a concurrent B$^+$-tree with 48 hyperthreads.

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

1 INTRODUCTION

The B-tree (or B$^+$-tree[1]) [7] has been the fundamental access path structure in databases and storage systems for over five decades [20, 33]. B-trees are a generalization of self-balancing binary search trees to arbitrary fanouts (with more than two children per node). They store elements in each node in a sorted array. Given a cache-line size Z [1], a B-tree with N elements and node size $B = \Theta(Z)$ supports the point operations `insert` and `find` in $O(\log_B(N))$ cache-line transfers in the I/O model [1].

B-trees are one of the top choices for in-memory indexing [81] due to their cache efficiency though they were initially introduced for indexing data stored on disk [7]. B-trees are especially popular in databases and file systems because they support logarithmic point operations (inserts and finds) and efficient range operations (range queries and scans) that read blocks of data [45, 69]. They are also extensively used as the in-memory index in many popular databases such as MongoDB [54], CouchDB [24], ScyllaDB [71], PostgreSQL [64], and SplinterDB [21].

There is an inherent tradeoff between point and range operations in practice for B-trees as the optimal node size is quite different for these two classes of workload. For range queries, the B-tree requires $O(\log_B(N) + k/B)$ cache-line transfers, where k is the size of the range. For long range queries, the k/B factor is the higher-order

[1] A B$^+$-tree is a scan-optimized variant of B-trees that stores all data records in the leaves and only pivot keys in the internal nodes. The B$^+$-tree is widely implemented in real-world applications as it supports faster range scans [21, 24, 44, 44, 54, 64, 71, 78, 79] compared to the traditional B-tree. In this paper, we refer to B$^+$-tree as the B-tree.

term, so increasing the node size improves range query performance. However, the performance of point insertions does not improve and may even suffer with larger nodes because the cost of keeping the items ordered in the nodes grows with the size of the node.

Figure 1 illustrates the empirical[2] point-range operation tradeoff with varying node size in a reference concurrent B-tree. Prior work showed that setting the node size much larger than the cache-line size can improve both point and range operations [19, 36]. Similarly, we found that the optimal node size for point operations was 2^{10} bytes in the tested B-tree. In contrast, the range query performance improves with the node size as the nodes grow past 2^{10} bytes in size. However, it starts to stagnate beyond 2^{16} bytes. Large nodes improve range queries because they convert random cache accesses to sequential ones by reading more contiguous data. However, as the node size grows, insertion performance suffers because more elements are shifted around to maintain order upon each insert.

Point and range operations in key-value stores. One of the core use cases for B-trees is to support key-value (KV) stores [53], a ubiquitous method of storing data as a collection of records, or key-value pairs. KV stores are used extensively in systems such as Dynamo [25], Redis [43], and Memcached [28, 61].

KV stores have traditionally been optimized and benchmarked for point operations (e.g., get and put) that underlie online transaction processing (OLTP) applications such as those in the Yahoo! Cloud Serving Benchmark (YCSB) [22]. The original YCSB workloads center around point operations such as point insertions and queries. They contain range queries, but only in a limited capacity because range operations are not as common as point operations in OLTP.

However, emerging real-time analytics applications increasingly require both fast point and range operations (e.g., range queries and maps) in the same in-memory KV store [6, 14]. Range queries from transactional workloads are often short (i.e. they involve only a few elements) — the default configuration in YCSB generates range queries with a maximum length of 100. In contrast, range queries from analytical workloads may be much longer and access a constant fraction of the data (e.g., 1% or 10%) [63]. To address this issue, prior work has focused on efficiently supporting both transactional and analytic processing applications on disk [18, 29, 35, 63, 67]. Recently, there has also been significant effort towards optimizing large range queries for in-memory KV stores [6, 14, 16, 26, 27, 74]. These works are motivated by in-memory workloads from databases [14, 51] and graph analytics [26, 58, 59] that require long scans. This paper focuses on the case of in-memory KV stores.

Systems optimized for either point operations or range operations may suffer on the other type of workload. For example, prior work showed that state-of-the-art KV stores such as Cassandra [17], RAMCloud [56], and RocksDB [68] perform poorly on long range queries because they were designed for point and short queries [62, 63]. Furthermore, HBase [37] integrates support for point and range operations but has been shown to underperform on point lookups compared to other KV stores.

Overcoming the point-range tradeoff in B-trees. The goal of this paper is to overcome the inherent point-range tradeoff in B-trees by making the nodes bigger to support fast range operations without compromising on point insert performance. As shown in Figure 1, simply making the nodes bigger while keeping the same sorted-array data structure in the nodes does not solve the problem because it improves range operations at the cost of point operation throughput.

To design a B-tree that achieves high performance for both point and range operations, we base our design on two insights about data layouts from Idreos *et al.* [40]: 1) that the internal nodes and leaf nodes in a B-tree do not have to be the same size, and 2) that the elements in a B-tree node are not necessarily sorted. This paper builds on these observations to achieve high throughput for both point and range operations in B-trees with optimized data structure design in B-tree nodes.

Overcoming the point-range tradeoff by making the leaf nodes large (for faster range operations) requires cleverly organizing data records inside leaf nodes to support efficient updates. Naively increasing the size of only the leaves does not solve the problem because it improves range operation throughput at the cost of point operation throughput[3]. Furthermore, simply relaxing the sortedness constraint in the nodes by using a hash-based organization (as in [40]) does not solve the issue because hash tables do not efficiently support ordered range queries, which are a key component of scan-based workloads such as the ones generated by YCSB [22].

To improve range operation performance while maintaining fast point operations, we introduce a new insert-optimized array-based data structure called the ***buffered partitioned array*** (BPA) that we incorporate into B-tree leaf nodes to allow them to grow in size without sacrificing point insert throughput.

The BPA is faster for inserts than a traditional array for two main reasons. First, it buffers insertions to avoid shifting elements on every insert, drawing inspiration from write-optimized data structures [32]. Next, it partitions the array into blocks and leaves empty spaces in the blocks to further avoid shifting elements upon buffer flushes, emulating the Packed Memory Array (PMA) data structure [10, 42].

We use the BPA to create the ***BP-tree***, a variant of the traditional B-tree that uses the BPA in the leaves and sets the leaf size to be much larger than the internal node size. This optimized B-tree design focuses on the leaves because the internal nodes are rarely updated in the steady state of the B-tree.

The BP-tree is optimized for long range queries that traverse multiple leaves in the B-tree but does not give up on point operation performance. It improves long range queries by avoiding pointer indirections that would have occurred with smaller leaf nodes, thereby converting random reads into sequential ones. Furthermore, the insert-optimized BPA ensures that there is no impact on the performance of point operations. In fact, it speeds up the point operations in some benchmarks.

Results summary. We implemented a concurrent BP-tree using the state-of-the-art TLX B-tree [12] and an optimistic concurrency control scheme [46] as the starting point. We include the baseline concurrent version of the TLX B-tree with 1024-byte nodes in the evaluation because this is the best node size for concurrent point operations on the machine. Additionally, we compare the BP-tree to Masstree [52] and OpenBw-tree [75], two state-of-the-art concurrent in-memory KV stores. To test all systems on both point and

[2] Section 6 contains all details about the experimental setup and method.

[3] We found that only changing the leaf size in B-trees results in similar trends to those in Figure 1.

range operations, we generated workloads from the YCSB [22] using both uniform-random and Zipfian distributions.

As we shall see in Section 6, the BP-tree achieves similar or better (between .95×-7.4× faster) performance on range operations from the original YCSB workloads while supporting similar point operation performance (between .94×−1.2× faster) when compared with the other systems. Specifically, the BP-tree is between $1\times-1.2\times$ faster than the baseline B-tree on point operations and achieves similar (.95× as fast) performance on short range queries from YCSB. Furthermore, the BP-tree is slightly slower (within .94× as fast) on point operations but about 7.4× faster on short range queries when compared to Masstree. Finally, the BP-tree is about 1.2× faster on point operations and 1.6× faster on short range operations when compared to OpenBw-tree.

Since the BP-tree is optimized for long range queries, we also tested all systems on long range query workloads by extending the YCSB workloads. The BPA enables the BP-tree to support faster range operations with large[4] leaves. We found that the BP-tree is about 1.3× faster than the concurrent B-tree, about 30× faster than Masstree, and about 2.5× faster than OpenBw-tree on large range queries.

Contributions

Specifically, the paper's contributions are as follows:

- An empirical evaluation of the impact of the node size on various B-tree operations in a concurrent setting.
- Overcoming the decades-old point-range operation tradeoff in B-trees with the BP-tree, a variant of the B-tree that incorporates the BPA in the leaves.
- The design and implementation (in C++) of the buffered partitioned array (BPA), an insert-optimized data structure that reduces element moves compared to a sorted array.
- An evaluation of the BP-tree compared to a traditional B-tree, Masstree, and OpenBw-tree on workloads from YCSB that demonstrates that BP-tree supports faster range operations with similar or better point operation performance.

2 PRELIMINARIES

This section provides necessary background to understand the data structure improvements in this paper. First, we introduce the classical "Disk-Access Machine (DAM) model" and a refinement to the DAM model called the "affine model." Next, we review details about B-trees, their operations, and optimistic concurrency control.

Memory models. The ***Disk-Access Machine (DAM) model*** [1] captures algorithm cost in hierarchical memory by taking into account non-uniform access times in different levels of memory. It models two levels of memory: a small fixed-size cache and an unbounded-size slow memory. Any data must first be in the cache before it is processed. Data is transferred between the two levels in cache lines of size Z. An algorithm's cost is measured in cache-line transfers.

The ***affine model*** [3, 8, 9, 70] refines the DAM model to explicitly account for the cost of random vs contiguous memory access. In the affine model, an I/O of x words has cost $1+\alpha x$, where $\alpha \ll 1$ is a hardware parameter. The 1 represents the normalized setup cost of doing an indirection (or seek, on disk) and α is the normalized bandwidth cost.

B-tree structure and operations. B-trees generalize self-balancing binary trees to arbitrary fanouts to take advantage of the speed of contiguous memory access [7]. Just like binary trees, B-trees store elements in sorted order. Traditionally, B-trees store $B=\Theta(Z)$ elements per node in a sorted contiguous array. The height of a B-tree with N elements and node size B is $O(\log_B(N))$.

A B-tree exposes four operations.

- `insert(k, v)`: inserts a key-value pair `(k, v)`.
- `find(k)`: returns a pointer to the element with the smallest key that is at least `x`.
- `iterate_range(start, length, f)` applies the function `f` to `length` elements in order (by key) starting with the element with the smallest key that is at least `start`.
- `map_range(start, end, f)`: applies the function `f` to all elements with keys in the range [`start`, `end`).

This paper considers both range iteration and map because the importance of ordered iteration in the scan depends on the use case. For example, the YCSB requires range iteration to simulate an application example of threaded conversations [22]. On the other hand, some applications may not necessarily need to access the keys in order. For example, graph-processing systems [5, 72], feature storage for machine learning [38, 50, 60], and file system metadata management [66, 69]. Cassandra [17] supports range iterations natively, while HBase's range query API [37] supports range maps natively.

B-tree theoretical bounds. We will now review the asymptotic bounds for the four main operations on a B-tree with N elements and node size $B=\Theta(Z)$.

The B-tree supports the point operations `insert` and `find` in $O(\log_B(N))$ cache-line transfers. To find an element in a B-tree, we traverse the internal nodes and follow the pivots (elements at internal nodes) to find the leaf that the target element might reside in. This procedure takes $O(1)$ cache-line transfers at each level of the tree for a total cost of $O(\log_B(N))$ cache-line transfers. Inserts begin with a find for the correct leaf to insert the element into. To maintain elements in sorted order, we shift elements in the array in the target leaf to make space and place the element in the correct position. If the leaf becomes full, it ***splits*** into two leaves and the midpoint is promoted to become a pivot in the internal nodes. This promotion procedure proceeds up the tree, if necessary.

The B-tree supports the range operations `iterate_range` and `map_range` in $O(\log_B(N) + k/B)$ cache-line transfers where k is the number of elements in the range. A range operation in a B-tree is comprised of a `find` for the smallest element with a key that is at least `start`, which takes $O(\log_B(N))$ cache-line transfers. Since the B-tree stores elements in sorted order, it implements both `iterate_range` and `map_range` with a forward scan from the starting element, which takes $\Theta(k/B)$ cache-line transfers.

2.1 Concurrency control

This section describes the optimistic concurrency control mechanism [46] used in the B-tree and BP-tree in this paper to support simultaneous operations (i.e., concurrent inserts, finds, and range queries).

[4] The default configuration for the BP-tree sets $B_{internal}$=1024 bytes and B_{leaf}=17408 bytes.

Each node is locked with a read/write lock, where multiple readers are allowed to access the node concurrently if no write lock is held.

The operations below use hand-over-hand locking [39] to traverse the tree. Hand-over-hand locking is a concurrency control mechanism where each thread holds at most two locks at a time. In hand-over-hand locking, a worker acquires the lock on a successor node (e.g., the lock on the child node) prior to releasing a lock on its predecessor (e.g., the lock on the parent node). To prevent deadlock, we maintain that locks are acquired in top-down[5], then left to right order.

Insert. Insertions first make an optimistic descent by taking hand-over-hand read locks from the root down to the leaf, then taking the write lock on the leaf. If we are able to insert into the leaf without causing a split, then we successfully insert into the leaf and release all locks. If inserting into the leaf would cause a split, we check if the parent can handle an additional element. If the parent can handle the additional insert, the change will not propagate farther up the tree and we just need to acquire the write lock on the parent of the leaf. We first try to upgrade the current read lock on the parent to a write lock, which can only be done if no other threads hold the read lock on the parent. If the lock can be upgraded, we can complete the insert. If the parent of the leaf cannot be upgraded, since some other thread is trying to read or write the parent, we release all locks and restart the insertion operation with a pessimistic second descent. In the second descent, we take write locks from the root down to the leaf. Then we lock the leaf and the new right leaf (from the split), insert into the appropriate leaf, and propagate the midpoints back up the tree, unlocking as we go.

Find. Finds take read locks in a hand-over-hand fashion from the root down to the leaf, then take the read lock on the leaf and search for the key within the leaf.

Range query. Range queries first perform a find on the start key to locate the starting leaf of the query, then take read locks from left-to-right as needed in a hand-over-hand fashion starting from the leaf that resulted from the find.

3 BUFFERED PARTITIONED ARRAY: OPTIMIZING LARGE B-TREE LEAVES FOR INSERTS

This section describes the buffered partitioned array (BPA) data structure that enables the BP-tree to maintain large leaf nodes without sacrificing updatability. It then describes how the BPA supports the operations described in Section 2.

The main idea behind the BPA is to create a data structure for B-tree nodes that uses a larger contiguous block of memory compared to traditional B-tree node sizes to enable fast scans while maintaining fast inserts. As demonstrated in Section 1, simply increasing the node size significantly degrades B-tree insert performance. Furthermore, the leaf node size determines overall performance because most of the writes affect only the leaves in the steady state. Therefore, we design a new data structure specifically for B-tree leaves.

At a high level, the BPA increases the size of the leaves by collapsing the last directory level of a B-tree and its children into a single contiguously allocated region of memory. However, it also employs additional optimizations to support fast insertions.

The BPA improves insertion throughput when compared to a sorted array by reducing data movement in two ways. First, the BPA buffers inserts to amortize data movement across operations. Next, it maintains empty spaces in the data structure in a "blocked structure" to avoid element shifting as much as possible even when the buffer becomes full. Finally, it does not maintain the elements in sorted order across the array. Specifically, the items inside blocks are not stored in order, which allows new inserts to be placed at the ends of blocks instead of requiring element shifts.

We omit the discussion of deletions for simplicity, but deletions are symmetric to insertions (by using tombstones).

Layout. The BPA uses a contiguous array to store its data, but partitions that array into three sections called the "log," the "header," and the "blocks," as illustrated in Figure 2a. It is parametrized by the following values:

- `log_size`: the maximum number of buffered inserts.
- `num_blocks`: the number of blocks in the data structure.
- `block_size`: the maximum number of elements per block.

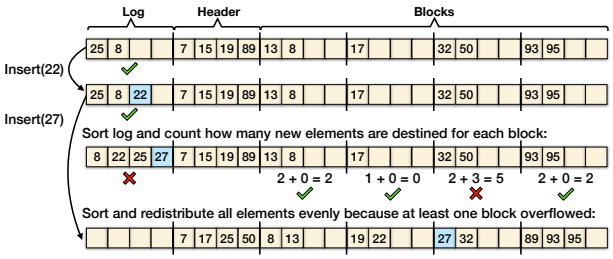
The ***log*** encompasses the first `log_size` slots (i.e., locations [0,`log_size`) and is used to buffer inserts that will later propagate to the rest of the data structure. The log coalesces multiple L1 fetches by ensuring that most inserts only read the first few cache lines of each BPA even if the overall size of the BPA is large (i.e., several Kb or more). The ***header*** uses the next `num_blocks` slots (i.e., locations [`log_size`, `log_size` + `num_blocks`)) to partition the rest of the data structure by range. Each slot in the header holds the minimum element (or a block marker) in the corresponding block. The elements in the header are sorted. Finally, there are `num_blocks` ***blocks*** of `block_size` slots each. Each block's elements fall in the range denoted by the corresponding header element. That is, the elements in block i fall in the range denoted by the elements at positions [`log_size`$+i$,`log_size`$+i+1$). The i-th block's elements start at position `block_start` = `log_size`+`num_blocks`$+i\times$`block_size`. The i-th block encompasses the cells in positions [`block_start`,`block_start+block_size`). In contrast to the elements in the header, elements in the log and each individual block may not be sorted.

Just as in other buffered data structures such as the B^ε-tree [32], there may be two copies of an element in a BPA: one in the log and one in the blocks. However, if an element is present in both the log and the blocks, the copy in the log must have been the one that arrived later and is therefore the one returned during queries.

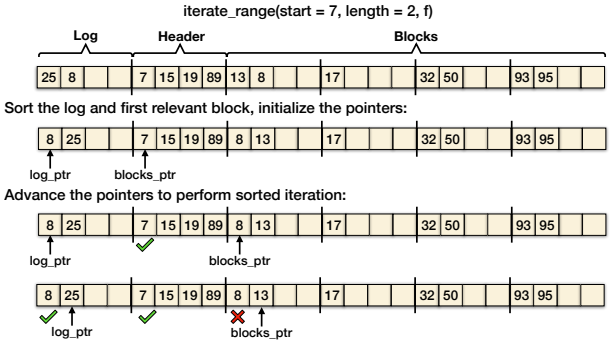
Insert. The BPA is an insert-optimized data structure that supports fast inserts by buffering inserts and storing elements in a contiguous array partitioned into fixed-size blocks with empty spaces. The BPA maintains the invariant that there is always at least one empty cell in the log and in each block after an insertion has been completed.

Suppose we want to insert a key-value pair (`k`, `v`) into the BPA. The BPA first scans all elements in the log. If any of the elements in the log have `k` as their key, the BPA replaces that element with (`k`, `v`) and returns. Otherwise, it appends the element to the end of the log. There are two cases after adding the new element to the log:
Case 1: There is at least one empty cell left in the log.

[5]Two-phase locking [11], a classical concurrency control mechanism in databases, also grabs locks in a top-down fashion. However, it differs from hand-over-hand locking in that it 1) phases the acquiring and releasing of locks and 2) allows workers to hold more than two locks at a time.



(a) An example of insertions in a BPA. The log has enough space to hold the first insertion, but the second one overflows the log and causes a flush to the blocks. Flushing to the block causes a redistribute because it overflows at least one of the blocks.



(b) An example of a call to `iterate_range` in a BPA. The BPA sorts parts of the data structure as needed. To perform the ranged iteration, the BPA executes a two-finger merge through the relevant (and now sorted) parts of the BPA.

Figure 2: Insertion and range operation in the BP-tree.

The insertion is complete. The first insert in Figure 2a illustrates appending an element at the end of the buffer.

Case 2: There are no empty cells left in the log.

To maintain the invariant that there are empty cells in the log before any insert, the BPA sorts and then ***flushes***, or moves, the elements to the rest of the data structure based on the partitioning given in the header. If this is the first time elements are being flushed from the log (e.g., near the beginning of the lifetime of the BPA) and there are no elements yet in the header, the log is ordered and simply moved to the header. Otherwise, if there are elements in the header, the BPA first counts up how many elements are in each block and stores the result in an array called `count_per_block` that stores how many elements are currently in each block. It then determines how many new elements (excluding duplicates) are destined for each block (based on the partitioning from the header) and stores the result in an array called `new_destined_per_block`. There are two possible cases:

Case 2a: For all $i = 0,1,\ldots,$`num_blocks`-1, `count_per_block[i] + new_destined_per_block[i]` $<$ `block_size`.

In this case, each block has enough space to accomodate elements from the log while still maintaining the invariant that there is at least one empty space in each block. The BPA flushes elements from the log to a block by 1) replacing any elements in the header/block with the same key as an element in the log with the newer version from the log and 2) moving all other elements in the log destined for the block into that block. If there are currently elements in the block, the BPA appends any relevant elements from the log to the end of those elements.

After the flush, the BPA completes the insertion by clearing the log.

Case 2b: For some $i = 0,1,\ldots,$`num_blocks`-1, `count_per_block[i] + new_destined_per_block[i]` $\geq$ `block_size`.

The second insert in Figure 2a illustrates this case of possibly filling one of the blocks upon a flush.

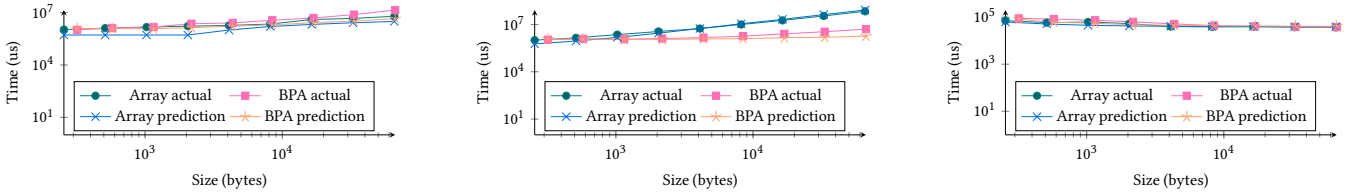
If there is not enough space in at least one of the blocks to flush elements from the log, the BPA sorts each block and merges all elements (from the log, header, and blocks) into a separate array, removing duplicate keys (i.e., if there are elements with the same key in the log and the header/blocks) along the way. At the end of the merge, all elements in the data structure are stored in sorted order in the separate array. The BPA then performs a ***redistribute***[6] that chooses a new header that split elements as evenly as possible amongst the blocks.

After the redistribute, the BPA completes the insertion by clearing the log.

Find. Point queries in a BPA first check the log, then the header, and then finally the blocks. If the key is found in the inserts in the log, the BPA returns that element. If the key is not found in the log, the BPA checks the header to see if the element is in the header. If the element is in the header, the BPA returns that element. If it is not in the header, the BPA uses the header to determine the block that the target element might possibly reside in. Finally, the BPA checks all elements in the relevant block and returns the element with the matching key, if there is one. Otherwise, it returns `null` because there was no element with a matching key.

Range iteration. Although the BPA is not globally sorted, it supports sorted iteration in a range (`iterate_range`) by sorting elements as necessary and then processing the elements in sorted order. Suppose the user calls the procedure `iterate_range(start, length, f)`. The BPA first sorts the inserts in the log and the block that the `start` key would reside in. It then initializes two pointers: 1) a `log_ptr` which starts at the smallest element in the log that is greater than or equal to `start`, and 2) a `blocks_ptr` which starts at the smallest element in the header/blocks that is greater than or equal to `start`. The BPA then performs a co-iteration with the two pointers to process `length` elements and applies the function `f` to those elements while advancing the pointers as necessary. If the query is not finished but the `blocks_ptr` reaches the end of its current block, the BPA sorts the next block (if there is one) and moves the `blocks_ptr` to the start of that block. If either of the pointers reaches the end of their respective sections (the log or the blocks) and fewer than `length` elements

[6]The redistribute procedure is inspired by the Packed Memory Array (PMA) data structure [10, 42], but the PMA is distinct in that it may perform local redistributes that do not include all of the blocks, while the BPA only performs global redistributes of all blocks.



(a) Find **(b) Insert** **(c) Scan**

Figure 3: Predicted and actual times for operations on parallel copies of a sorted array and a BPA.

have been processed, the BPA advances the remaining pointer until it reaches the correct number of elements or runs out of elements.

As an additional optimization to avoid unnecessary sorting, the BPA keeps a bit vector of length `num_blocks` that denotes whether the elements in each block are currently sorted. It sorts a block during a range query if and only if the corresponding bit in the bit vector is unset. If the BPA sorts a block as part of a redistribute or sorted range query, it sets the corresponding bit. If there are elements flushed to a block during an insert, the BPA unsets the corresponding bit because the elements in the block may have become unsorted.

Figure 2b presents a worked example of a sorted range query.

Range map. The procedure for range maps in the BPA is similar to the one for range iteration except that maps do not need to perform a two-finger co-iteration. The map first copies all elements in the log into a separate array and sorts it to avoid scanning through the entire log during the map for each block. It then scans through the inserts in the log and applies the function `f` to all elements that fall within the range. The BPA then scans through the relevant blocks, skipping all elements that are not in the range or that have a duplicate in the log.

4 PERFORMANCE PREDICTION USING THE AFFINE MODEL

In this section, we use the affine model [8] to theoretically analyze the performance of a sorted array and BPA data structure. To isolate the performance difference between the sorted array and the BPA, we only consider the array data structures to model the performance of the tree as both the B-tree and BP-tree have the same internal structure (including the size of internal nodes) and only differ in the representation and the size of the leaf nodes. We then perform experiments to show how closely the theoretical models fits the experimental performance. Finally, we consider other empirical factors that are not included in the original affine model.

The goal of this section is to demonstrate that the BPA achieves a better insert-scan tradeoff than the sorted array (that is, the insert performance degrades slower as a function of size in the BPA than the sorted array). This makes the BPA appropriate to represent leaf nodes in the B-tree to achieve an better overall insert-scan tradeoff.

We find that the affine model closely predicts the performance of inserts and scans in memory. For finds, the prediction follows the same curve as the empirical performance but with a higher magnitude. We attribute this to the fact that find incurs random cache line accesses and it is harder to determine the base case due to memory-level parallelism and prefetching in the system.

Methodology. The main components of the in-memory affine model are Z, the size of a cache line, α, the relative cost of a sequential cache-line access, r, the cost of reading a random location in DRAM, and w, the cost of writing to a random location in DRAM.

We empirically measure α and r using the following ***scan test***[7]. At a high level, this test allocates a large contiguous chunk of memory of N bytes and reads (and sums) the entire chunk in parallel in randomly-ordered fixed-size blocks of size ℓ (using the same systems setup and pthreads as described in Section 6). All times are the average of 10 trials. Although the total number of bytes read is constant regardless of the setting of ℓ, the total time of the experiment decreases with increasing ℓ because each block involves reading more contiguous memory. By setting $\ell = Z$, we can compute the latency of reading a random cache line in DRAM by dividing the total time by the number of lines read. Furthermore, we can compute α by increasing ℓ and calculating the relative cost of reading consecutive cache lines compared to reading random cache lines. We set $N = 2^{33}$ and set ℓ as powers of 2 from 2^3 to 2^{20}.

To measure the performance differences between a sorted array and the BPA, we perform a parallel ***copies test***. We now describe the experiment for sorted arrays for simplicity, but the test for the BPA has the same structure. We insert and process n total elements in c distinct sorted arrays with s slots each (where $n \gg s$). We generate s uniform random key-value pairs (of 8 byte keys and values, for a total of 16 bytes per element). We first insert half of the s key-value pairs into the c copies without timing them to simulate the steady state of a B-tree where the leaves are at least half full. To time the insertions, we iterate serially over the remaining $s/2$ elements. We insert each element into all c copies in a random order in parallel. Therefore, the total number of timed insertions is $n/2$. To test worst-case finds, we iterate serially over s elements not present in the data structures and in parallel try to find each of those elements in the c copies (in a random order of copies) to simulate reading leaves in a B-tree search that are unlikely to be in cache. Finally, in parallel, we perform a sum in all of the copies. We perform the same experiment for copies of the BPA where the number of slots is the number of slots in the blocks (not counting those in the log and header). We set $n = 2^{28}$ and vary the number of slots s from 2^2 to 2^{12}.

Empirical parameters. Using the scan test described above, we find $\alpha = 0.3$ and $r = 1.95$ ns. The machine has a cache-line size $Z = 64$ bytes and $w = 2r$.

[7] The code for the scan test can be found at https://github.com/wheatman/scan_test.

Let A_Z be the size of the array in cache lines. Let L_Z be the size of the log, H_Z be the size of the header, and B_Z be the size of blocks (all in terms of cache lines).

Find prediction. To analytically measure the cost of a find in a sorted array, we compute the number of random cache misses due to a binary search as $\lg(A_Z)$ using the cache-line size as the base case for the recursion. To find the total latency per find, we multiply the number of cache misses by r, the latency of a random read in DRAM.

To analytically measure the cost of a find in a BPA, we compute the number of cache-line reads in the affine model to search in log, the header, and blocks. We compute the cost to search a log of L bytes (in cache-line reads) as $r(1+\alpha(L_Z/2-1))$ because the log is half-full on average. We model the cost of reading the header without a random access because the log and header are contiguous (i.e. as $r(\alpha H_Z/2)$). Since we fill the BPA, we compute the cost of reading the relevant block as $r(1+\alpha(B_Z/2-1)$.

Figure 3a shows that the BPA is empirically slower than the sorted array for finds, but that the affine model predicts that the BPA should be faster for point finds once the arrays are sufficiently large. The affine model is more accurate for BPA finds because they involve sequential reads, while the sorted array finds perform random accesses in a binary search.

Insert prediction. To predict the insert cost in a sorted array, we build upon the analytical cache misses for find. An insertion consists of a find and item shifts to make space for the new item. On average, half of the elements are moved upon each insert since the inputs are uniform random. Therefore, the latency of moving elements in the affine model upon each insert is $w\alpha A_Z/2$.

Each insertion iterates through the elements in the log to check for duplicates. If there is not a duplicate and there is enough space in the log, it places the new element at the end of the log. Therefore, since the log is half full on average, the average latency to iterate through the log and append to the end is $r(1+\alpha(L_z/2-1))+\alpha w$. The BPA incurs this cost upon every insert.

The total cost of an insert in a BPA is amortized because the BPA flushes the elements to the blocks when the log is full. The first step in a flush is to read the header to determine which blocks elements are destined for, which has latency $r\alpha H_z$. To determine how many blocks will be flushed to, we use a balls-and-bins analysis to compute the expected number of empty bins when b balls are tossed into b bins uniformly at random [23]. The expected number of empty bins is $X = b(1-1/b)^b$. Given b blocks in the BPA, the number of bins touched during the flush is $T = b-X$. The next step is to search every block for duplicates of the elements in the log. Since each block is on average half full, this search incurs latency $Tr(1+\alpha(B_Z/2-1)$ per block. Appending to the end of all block has an additional expected latency $T\alpha w$. The cost of this flush is amortized over the number of elements in the log.

Finally, since the input is uniform random, there is 1 expected global redistribute. We omit this cost since it is several orders of magnitude smaller than the total time to perform all insertions.

Figure 3b demonstrates that the affine model captures the empirical insert results: the insertion latency in a BPA grows more slowly as a function of the array size when compared with a sorted array.

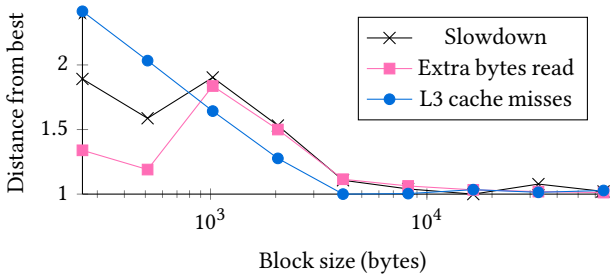


Figure 4: Relative slowdown (time), L3 cache misses, and bytes read during the scan test.

Scan prediction. We use the affine model to model the number of cache misses to scan over the entire array and BPA. Just as in the classical I/O model, the affine model only accounts for cache-line transfers and does not measure the cost of additional computational overhead once the data is brought into cache (e.g., due to sorting). Given an array with A_Z cache lines, the cost in the affine model to scan it (in cache lines) is $1+\alpha(A_Z-1)$. Similarly, the cost to scan a BPA in the affine model is $1+\alpha(L_Z+H_Z+L_ZB_Z-1)$. We omit the additional computational processing cost of a BPA scan because the affine model only counts cache misses.

Figure 3c shows that the scan cost of both the sorted array and BPA decreases with the size of the size of the arrays. Furthermore, the affine model provides an accurate prediction of BPA scan performance by accounting for cache misses.

Reasoning about scan performance. We use the cache line and the DRAM memory controller (MC) to reason about the trends that drive the BP-tree design and performance. Although the MC is not included in the original affine model, we perform empirical measurements[8] to explain the performance trends seen in Figure 1 and report the results in Figure 4.

As demonstrated by Figure 1, modern processors include adjacent cache-line prefetchers to speed up contiguous access of even a few cache lines [41]. Although the cache-line size is 64 bytes, the affine model captures the fact that prefetching makes contiguous access faster. Therefore, the goal of the BPA is to take advantage of the speed of contiguous access.

Although the cache-line prefetcher does not cross page boundaries [41], we continue to observe performance improvements in the scan test with larger block sizes than the page size (a scan with a block size of 4Kb is about 1.1× slower than the best scan with a block size of 64Kb). The number of L3 cache misses does not improve significantly when the block size is greater than 4Kb.

We hypothesize that the improvements in scan performance for block sizes larger than a page shown in Figure 1 stem from the DRAM row buffers. Data is loaded in an interleaved fashion into DRAM row buffers of 8Kb each (one per bank). When data is accessed in DRAM, the entire row is brought into the row buffer for the corresponding bank. Figure 4 shows that block sizes smaller than 64Kb results in extra bytes read by the DRAM memory controller. For block sizes larger than 4Kb, the scan performance correlates exactly with the extra bytes read.

[8]We used the Intel Performance Counter Monitor (https://github.com/intel/pcm) on the scan test run serially.

5 BP-TREE: INTEGRATING THE BPA INTO B-TREE LEAVES

This section introduces the ***BP-tree***, a modified version of the concurrent B-tree described in Section 2 that uses large leaf nodes to optimize range operations without slowing down point operations. It uses the BPA from Section 3 in the leaf nodes and standard sorted arrays in the internal nodes. Next, this section describes how to implement the B-tree operations outlined in Section 2 in the BP-tree.

Structure. The BP-tree replaces the sorted array in the leaves of a B-tree with the BPA. The leaf nodes in the BP-tree do not have to be the same size as the internal nodes, and often are much larger because of the specialized BPA data structure.

Traditional B-trees keep track of exactly how many elements are in each leaf to determine when a leaf becomes too full during insertions. Since the BPA may contain duplicates of elements due to buffering, each leaf in a BP-tree keeps track of its `num_elts`, or the number of elements in each leaf (including duplicates), to determine how full the leaf node is. Although the count `num_elts` in the BP-tree may be an overestimate of the number of elements with different keys, it is at most `log_size` away from the correct count.

Concurrency control. As described in Section 2, the optimistic concurrency control mechanism in the BP-tree uses reader-writer locks on each node. To ensure thread-safety, the BP-tree acquires either the reader or the writer lock on the nodes depending on whether the node needs to be updated. The locks taken in the BP-tree are almost identical to those in the B-tree as described in Section 2. As we shall detail later in this section, the only difference is that the B-tree takes only read locks during range iterations, while the BP-tree may occasionally write to the leaves (and therefore occasionally take the write lock on the leaves) during a range iteration.

Insert. Like inserts in the plain B-tree, inserts in the BP-tree first traverse down to the correct leaf by following the root-to-leaf path and then check if the target leaf is full (i.e., the count `num_elts` is equal to the number of slots in the BPA).

If the leaf is not full, we call `insert` on the BPA and increment the `num_elts` in the leaf. If the leaf is full, we first flush the log in the BPA in that leaf and then perform a split. A split creates a new "right" leaf and divides the elements as evenly as possible between the current leaf and the new leaf. Since `log_size` is much smaller than the size of the BPA, there are always enough elements in the BPA to perform a valid split even if all of the elements in the log are duplicates. The split moves the upper half of the full BPA's elements in sorted order into a new BPA structure. After a split, the current leaf contains the first half of elements in sorted order, and the new right leaf contains the new BPA with the remaining half of the elements. The BP-tree then checks which leaf the new element should be inserted into, and calls insert on that BPA.

Find. Finds in the BP-tree first traverse down to the leaf where the key would be located, then use the BPA's find functionality.

Range iteration. BP-tree range iterations use the `iterate_range` functionality in the BPA to process elements in sorted order at the leaves. Given a call to `iterate_range(start, length, f)` in the BP-tree, the first step is to traverse down to the leaf where the `start` key would be located. The BP-tree then calls the `iterate_range` on the BPA (as described in Section 3) with the same parameters. The BPA reports the number of elements found in the query. If the reported number of elements found equals the length of the query, the query is finished. Otherwise, the BP-tree keeps track of how many elements have been processed so far. It then continues onto the next leaf and calls `iterate_range` on the BPA in this new leaf with the remaining number of elements in the query and adjusts the count of elements processed so far accordingly. This is repeated until the total number of elements processed is equal to `length`, or no next leaf exists.

The function `iterate_range` in the BPA may require taking the write lock on the leaf, since the iteration may need to sort the log and at least one of the blocks.

We implemented an additional optimization to reduce the number of times that a leaf's write lock is taken, which is necessary to avoid contention on the write lock when the input distribution is skewed. When processing a leaf, the BP-tree first takes the read lock on that leaf and checks whether 1) the log is sorted, and 2) the relevant blocks are sorted (using the bit vector described in Section 3). If both conditions are met, the iteration can proceed with just the read lock because the relevant data is already sorted. If either is not true, however, the worker releases the read lock and acquires the write lock to sort the target data.

Range map. Range maps in a BP-tree are similar to range iterations, except that they use the `map_range` functionality in the BPA. Since range maps may require the write lock to sort the log, we employ a similar optimization to first take the read lock and only upgrade it to a write lock if necessary.

6 EVALUATION

This section demonstrates that the BP-tree improves long range operations without giving up point operation performance on a suite of microbenchmarks as well as on workloads generated from YCSB [22]. We try several node-size configurations of the concurrent B-tree and BP-tree and report the results in Section 6.1. We choose default configurations for the B-tree and BP-tree based on the results in Section 6.1 and compare them to Masstree [52] and OpenBw-tree [75], two state-of-the-art concurrent comparison-based in-memory KV stores, on a suite of workloads from the YCSB [22]. Section 7 reviews related work in KV stores and explains the choice of systems in this evaluation. We do not compare against hash-based approaches, since they do not support efficient range iteration.

The microbenchmarks demonstrate that the BP-tree supports long range operations up to 1.5× faster when compared with the best-case B-tree configuration for inserts. Furthermore, the BP-tree achieves slightly faster (about 1.05× faster) performance on point operations compared to the B-tree. The BP-tree achieves similar performance (about 1.04× faster) compared to the B-tree on short range operations.

Using the best configuration for inserts in the BP-tree and B-tree, we evaluate the BP-tree, B-tree, Masstree, and OpenBw-tree on point and range operations from the Yahoo! Cloud Serving Benchmark (YCSB) [22]. We found that the BP-tree supports similar point operation throughput (between .94×−1.2×) compared to other data structures. On the default YCSB workload with short scans, the BP-tree is about 7.4× faster than Masstree and 1.6× faster than OpenBw-tree. Furthermore, we extend the YCSB to add long-range workloads and

find that the BP-tree is 30× faster than Masstree and 2.5× faster than OpenBw-tree for large range queries.

Systems setup. All experiments were run on a 48-core 2-way hyperthreaded Intel® Xeon® Platinum 8275CL CPU @ 3.00GHz with 189 GB of memory from AWS [2]. The machine has 1.5MiB of L1 cache, 48 MiB of L2 cache, and 71.5 MiB of L3 cache across all of the cores. To avoid non-uniform memory access (NUMA) issues across sockets, we ran all experiments on a single socket with 24 physical cores and 48 hyperthreads.

All times are the median of 5 trials after one warm-up trial.

Data structures setup. We used the B-tree [20] from the TLX library [12] with 64-bit elements in map mode (i.e. with keys and values) as the starting point for our implementation. We then implemented the optimistic concurrency control scheme described in Section 2 on top of the main operations. We ran the operations concurrently using pthreads [55]. Finally, we implemented the BP-tree by replacing the arrays in the B-tree leaves with the BPA from Section 3.

We test various node sizes in two different types of blocked trees: 1) the standard B-tree which sets the internal and leaf node sizes to be the same, and 2) the ***BP-tree*** with BPAs in the leaf nodes. We also tried a variant of B-trees that only grows the leaf nodes and keeps the size of the internal nodes fixed, but found that the performance was similar to the traditional B-tree because the leaf nodes are the most affected during operations.

The B-tree takes a parameter `node_size` (in bytes) for both the internal and leaf nodes. We tested different (power of 2) node sizes ranging from 2^8 up to 2^{16}.

The BP-tree takes several parameters as described in Section 3: `internal_size`, `header_size`, and `block_size`. We tested various settings of all of the parameters and report results with a fixed[9] `internal_size` = 1024 bytes and various leaf sizes.

The default configuration for the B-tree sets `node_size` = 1024 bytes. The default BP-tree sets `internal_size` = 1024 bytes, and `header_size` = `block_size` = 32 slots (for a total of 17408 bytes in each leaf.) We chose these configurations because they achieve the best insertion performance for the B-tree and BP-tree.

6.1 Evaluation on microbenchmarks

Workloads setup. We concurrently insert 100M uniform random elements in the range $[1,2^{64}-1]$ into a tree that already has 100M uniform random elements. We then performed finds (point queries) for 1M of those elements. Finally, we tested range queries with varying maximum lengths. For each maximum length `max_len` tested, we performed 1M range iterations with lengths distributed uniformly randomly in the interval $[0, \texttt{max\_len}]$. We saved the start and end points of each of these queries and used them to perform 1M range maps on the same ranges.

Furthermore, to test the case of sequential insertions (i.e. insertions with monotonically increasing keys), we conducted an additional experiment of adding 100M monotonically increasing keys. Each thread sequentially inserts a set of monotonically increasing keys that is disjoint from any other thread's set. The threads proceed concurrently with respect to each other.

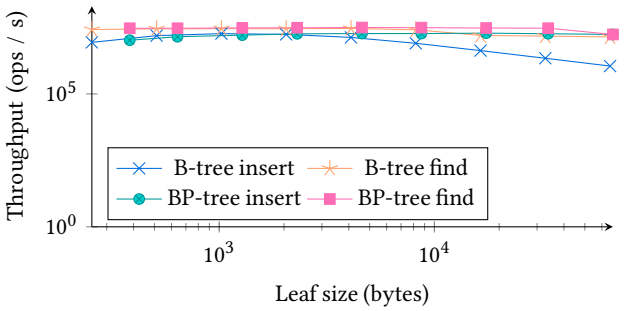


Figure 5: Point operation throughput as a function of leaf size.

Inserts. Figure 5 shows that the BP-tree achieves similar (1.04× faster) insertion throughput on random inputs when compared to the best-case insertion throughput of the baseline B-tree (at `node_size` = 1024). However, as shown in Table 1, the BP-tree is over 4.5× faster for inserts when compared to a B-tree with similar-sized leaf nodes. Figures 1 and 5 illustrates the decline in insertion throughput in the B-tree as the leaf size grows.

Although the default BP-tree has large leaf nodes (over 16k bytes), it achieves high insertion throughput because the optimized BPA data structure amortizes element moves across inserts. In the traditional B-tree, each insertion shifts existing elements within a leaf's sorted array to make space for the new element. Therefore, the insertion performance in B-trees with sorted arrays in the leaves degrades with increasing leaf size because the number of element moves grows proportionally with the leaf size. In contrast, the BPA relaxes the sortedness of the leaves and buffers insertions in the log. It flushes elements to the blocks only when the log is full, amortizing accesses to the blocks across inserts.

We also found that the concurrent BP-tree achieves similar[10] (.96×) performance on sequential (monotonically increasing) insertions when compared to the B-tree. Sequential insertions are the worst case for the BP-tree because they maximize element redistributions in the BPA. In contrast, they are the best case for the B-tree because they minimize element moves in an array. However, the performance in both trees is similar because the log in the BPA amortizes the work of redistributes across insertions.

Finds. Figure 5 shows that the BP-tree supports finds about 1.06× faster than the default B-tree. Finding an element in a BPA avoids looking at the entire data structure via the header. In contrast, finding an element in an array requires a scan when the node size is small (up to 256 bytes) or a binary search when the node size is large.

As shown in Figure 1, the find throughput does not change as much as the insert throughput as a function of node size in a B-tree. When the nodes are sufficiently large, finds can be implemented in the B-tree with a binary search, which only requires looking at a logarithmic number of elements in each node (in contrast to a linear number of element moves upon an insert).

[9] We found that varying the size of the internal nodes did not have much of an effect on the performance, so we report results for a fixed internal node size.

[10] The BP-tree achieved 2.11E7 inserts/s and the B-tree achieved 2.03E7 inserts/s on sequential inserts.

Table 1: Throughput (thr., in operations per second) and normalized performance of point operations in the B-tree and BP-tree. Point operation throughput is reported in operations/s. We use N.P. to denote the normalized performance in the B-tree (BP-tree) compared to the best B-tree (BP-tree) configuration for that operation (1.0 is the best possible value).

B-tree					*BP-tree*						
	Insert		*Find*					*Insert*		*Find*	
Node size (bytes)	*Thr.*	*N.P.*	*Thr.*	*N.P.*	*Header size (elts)*	*Block size (elts)*	*Total size (bytes)*	*Thr.*	*N.P.*	*Thr.*	*N.P.*
256	8.72E6	0.47	2.66E7	0.92	4	4	384	1.05E7	0.54	2.96E7	0.94
512	1.56E7	0.84	2.81E7	0.97	4	8	640	1.42E7	0.73	2.96E7	0.94
1024	1.86E7	1	2.86E7	0.98	8	8	1280	1.63E7	0.84	3.05E7	0.96
2048	1.74E7	0.93	2.84E7	0.98	8	16	2304	1.83E7	0.94	3.09E7	0.98
4096	1.34E7	0.72	2.91E7	1	16	16	4608	1.87E7	0.97	3.16E7	1.00
8192	8.04E6	0.43	2.60E7	0.89	16	32	8704	1.87E7	0.97	3.12E7	0.99
16384	4.27E6	0.23	1.59E7	0.55	32	32	17408	1.94E7	1.00	3.02E7	0.96
32768	2.20E6	0.12	1.50E7	0.52	32	64	33792	1.84E7	0.95	2.97E7	0.94
65536	1.12E6	0.06	1.40E7	0.48	64	64	67584	1.73E7	0.89	1.73E7	0.55

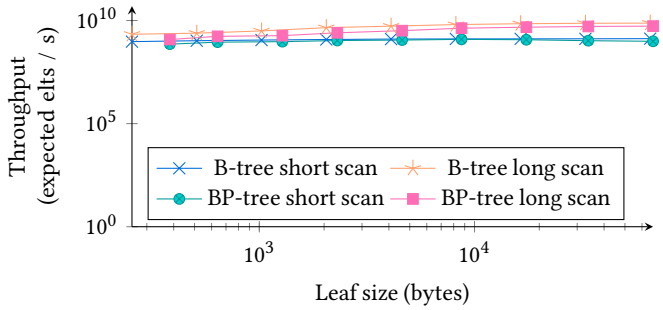


Figure 6: Range iteration throughput as a function of leaf size. The short scan has `max_len` = 100 and the long scan has `max_len` = 100,000.

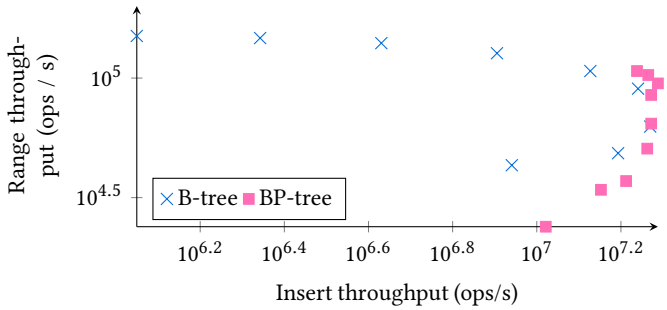


Figure 7: Point and long range iteration throughput in various configurations of the B-tree and BP-tree.

Range operations. Figure 6 reports the throughput of short and long range iterations with growing node sizes in the B-tree and the BP-tree. We find that given a similar leaf size in the B-tree and BP-tree, the B-tree is about 1.5× faster for range scans because it does not incur the computational overhead of the BPA. However, Figure 7 shows that the B-tree must give up on insertion performance to match the range performance of the default BP-tree configuration.

The BP-tree achieves about 1.5× speedup on large range iterations when compared to the default B-tree configuration because the large leaves in the BP-tree reduce data movement as shown in Section 4. Furthermore, we find that the default BP-tree is slightly (1.04×) faster for short range iterations when compared to the default B-tree because the range fits within the B-tree leaf. The large leaves in the BP-tree make the BP-tree better suited to machines with the HugePage optimization as it performs long sequential scans that can fit in huge pages compared to the B-tree.

As mentioned in Section 2, we also evaluate range maps, which processes elements in a range in any order. The BP-tree is about .8× as fast on short range maps and about 1.7× faster on long range maps compared to the B-tree. The BP-tree achieves similar throughput for long range maps and iterations, but slower (about .6×) performance on short range maps because of the relative cost of copying the log. Additionally, range maps are slightly slower (up to .9×) compared to range iterations in a B-tree (which stores elements sorted) because of the difference in the range API, which leads to different compiler optimizations around unrolling.

6.2 Evaluation on YCSB workloads

We also evaluate the B-tree, BP-tree, Masstree, and OpenBw-tree[11]. on workloads from YCSB [77] and report the results in Figures 8 and 9 and Table 3.

Experimental setup. Table 3 presents details of the core workloads from YCSB [77]. We tested workloads[12] A, B, C, and E from the core YCSB workloads by adapting the YCSB driver from RECIPE [47]. Running a workload in YCSB has two consecutive phases: 1) the *load* phase, which adds elements to the data structure, and 2) the *run* phase, which performs operations specified by the workload. For each workload, we generated 100M elements to insert in the load phase and 100M operations to perform in the run phase. We ran all 100M operations in each phase concurrently.

We tested both uniform random and zipfian distributions in the run phase using the generator from RECIPE [47]. 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

[11]The implementation of Masstree is from https://github.com/kohler/masstree-beta, and the implementation of OpenBw-tree is from https://github.com/wangziqi2013/BwTree.git.

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

Table 2: Throughput (thr., in expected elements per second) of range queries of varying maximum lengths (`max_len`) in the B-tree and BP-tree. We also report the normalized performance (N.P.) compared to the best-case performance for each operation (up to 1.0).

B-tree									*BP-tree*									
	Short (`max_len` = 100)				*Long* (`max_len` = 100,000)							*Short* (`max_len` = 100)				*Long* (`max_len` = 100,000)			
	Map		*Iterate*		*Map*		*Iterate*					*Map*		*Iterate*		*Map*		*Iterate*	
Node size (bytes)	*Thr.*	*N.P.*	*Thr.*	*N.P.*	*Thr.*	*N.P.*	*Thr.*	*N.P.*	*Header size (elts)*	*Block size (elts)*	*Total size (bytes)*	*Thr.*	*N.P.*	*Thr.*	*N.P.*	*Thr.*	*N.P.*	*Thr.*	*N.P.*
256	8.56E8	0.77	9.48E8	0.72	1.88E9	0.25	2.16E9	0.29	4	4	384	4.76E8	0.53	7.15E8	0.59	7.32E8	0.14	1.20E9	0.22
512	9.58E8	0.86	1.05E9	0.80	2.12E9	0.28	2.43E9	0.32	4	8	640	6.86E8	0.76	8.93E8	0.73	1.32E9	0.25	1.71E9	0.32
1024	1.01E9	0.91	1.13E9	0.85	2.69E9	0.36	3.13E9	0.42	8	8	1280	7.91E8	0.88	9.45E8	0.78	1.72E9	0.32	1.85E9	0.35
2048	1.08E9	0.97	1.20E9	0.91	4.23E9	0.56	4.51E9	0.60	8	16	2304	8.98E8	1.00	1.07E9	0.88	2.46E9	0.46	2.54E9	0.47
4096	1.11E9	1.00	1.26E9	0.95	5.18E9	0.69	5.33E9	0.71	16	16	4608	8.99E8	1.00	1.13E9	0.93	3.17E9	0.59	3.22E9	0.60
8192	1.10E9	0.99	1.28E9	0.97	5.97E9	0.80	6.36E9	0.85	16	32	8704	8.86E8	0.99	1.22E9	1.00	4.19E9	0.78	4.25E9	0.79
16384	1.08E9	0.98	1.29E9	0.98	6.60E9	0.88	7.00E9	0.93	32	32	17408	8.14E8	0.91	1.17E9	0.96	4.75E9	0.89	4.75E9	0.89
32768	1.08E9	0.97	1.30E9	0.98	7.18E9	0.96	7.36E9	0.98	32	64	33792	6.73E8	0.75	1.05E9	0.87	5.21E9	0.97	5.16E9	0.96
65536	1.09E9	0.98	1.32E9	1.00	7.50E9	1.00	7.49E9	1.00	64	64	67584	5.74E8	0.64	9.83E8	0.81	5.35E9	1.00	5.35E9	1.00

Table 3: Throughput (in operations/s) of the BP-tree (BPT), B-tree (B^+T), Masstree (MT), and OpenBw-tree (BWT) on uniform random and zipfian workloads from YCSB.

		Uniform							*Zipfian*						
Workload	Description	*BPT*	*B^+T*	*B^+T/ BPT*	*MT*	*MT/ BPT*	*BWT*	*BWT/ BPT*	*BPT*	*B^+T*	*B^+T/ BPT*	*MT*	*MT/ BPT*	*BWT*	*BWT/ BPT*
A	50% finds, 50% inserts	2.91E7	2.33E7	0.80	3.07E7	1.06	2.47E7	0.85	3.00E7	2.78E7	0.93	3.20E7	1.07	2.56E7	0.85
B	95% finds, 5% inserts	4.70E7	4.46E7	0.95	4.79E7	1.02	3.98E7	0.85	5.63E7	4.84E7	0.86	5.82E7	1.03	4.74E7	0.84
C	100% finds	4.99E7	4.81E7	0.96	5.18E7	1.04	4.21E7	0.84	6.01E7	5.99E7	1.00	6.40E7	1.06	5.10E7	0.85
E	95% short range iterations (`max_len` = 100), 5% inserts	2.58E7	2.71E7	1.05	3.49E6	0.14	1.54E7	0.60	3.25E7	3.35E7	1.03	3.96E6	0.12	1.70E7	0.52
X	100% long range iterations (`max_len` = 10,000)	8.89E5	6.90E5	0.78	2.74E4	0.03	3.60E5	0.40	1.05E6	7.96E5	0.76	2.76E4	0.03	3.65E5	0.35
Y	100% long range maps (`max_len` = 10,000)	9.18E5	6.45E5	0.70	2.74E4	0.03	3.63E5	0.40	1.08E6	7.44E5	0.69	2.76E4	0.03	3.71E5	0.34

phase elements are generated from a uniform distribution, while the elements in the run phase are generated from a zipfian distribution with the default YCSB zipfian constant (i.e., theta [34] of 0.99).

The YCSB experiments use the `insert` (put), `find` (get), and `iterate_range` (scan) operations defined in Section 2. To generate scan operations, the YCSB workload generator takes as input a `max_len` parameter and generates range iteration operations with lengths uniformly distributed in the range [0,`max_len`].

The core YCSB workloads focus mostly on point operations and short range operations. To illustrate the strenghts of the BP-tree, we added two new workloads: 1) workload X, which performs long range iterations, and 2) workload Y which performs long range maps. Both have `max_len` = 10,000. Although the original YCSB workloads do not include range maps[13], we include them in addition to the provided workloads to illustrate how the different systems perform on operations from other application areas.

Finds and inserts. The BP-tree achieves similar (within .94×−1.2×) performance compared to the other systems on the workloads containing point operations (workloads A, B, and C). The BP-tree is optimized for long range operations but is between 1×−1.2× faster than the best-case B-tree for point insertions despite the much larger leaves in the BP-tree because of the insert-optimized BPA in the leaves. Furthermore, the BP-tree is slightly slower (within .94×) on point operations compared to Masstree but about 1.2× faster on point operations when compared to OpenBw-tree.

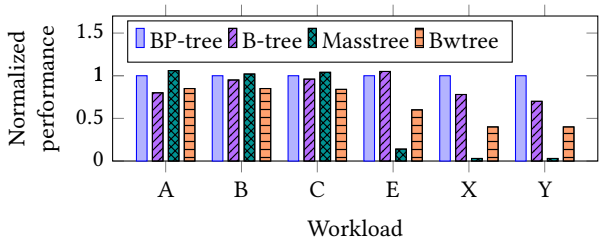


Figure 8: Relative performance compared to the BP-tree on uniform random workloads generated from YCSB.

Range operations. We test both short and long range queries to illustrate the benefits of the BP-tree. The BP-tree is optimized for

[13] Masstree does not provide an API for range maps, so we use the range iteration throughput for both workloads X and Y. OpenBw-tree provides a distinct API for both range iterations and maps.

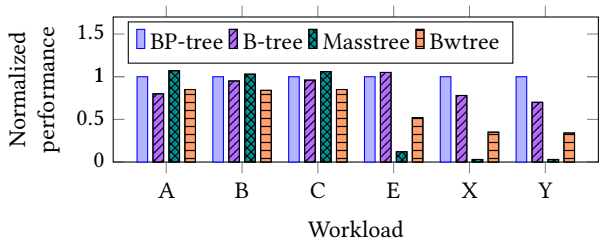


Figure 9: Relative performance compared to the BP-tree on zipfian workloads generated from YCSB.

long range operations : the BP-tree is about 1.3× faster than the concurrent B-tree, about 30× faster than Masstree, and about 2.5× faster than OpenBw-tree on workloads X and Y. For short range operations (workload E), the BP-tree is about 7.4× faster than Masstree and about 1.6× faster than OpenBw-tree. However, the BP-tree is about .95× as fast as the B-tree on short range iterations (workload E) because the benefits of improved locality do not outweigh the computational overhead in the BP-tree when the range size fits in one node.

Effect of input distribution. We found that all systems were faster on the zipfian dataset than the uniform dataset because there are fewer unique elements, so there is more temporal locality in the operations. The relative performance between systems is similar across both input distributions.

7 RELATED WORK

This section reviews related work on KV stores to explain the choice of systems in the evaluation. It then discusses the relationship between the BP-tree and buffer trees.

In-memory KV stores. This paper focuses on comparison-based concurrent in-memory KV stores for concreteness, so the evaluation includes Masstree [52] and OpenBw-tree [75]. The literature on KV stores includes many other works, but we omit them because they are not directly comparable. For example, the hybrid B+-tree only provides API support for point operations [80], and Google's B-tree is not thread-safe for concurrent writes [31]. Several range-optimized data structures [6, 15, 16] appear in the literature but are implemented in Java, so we omit them to avoid performance differences due to the underlying language. Finally, recent works have introduced non-comparison-based indexes such as ART [48] and HOT [13]. Future work includes exploring the potential for the BPA to speed up non-comparison-based indexes.

Disk-based KV stores. The log-structured merge (LSM) tree [57] is another hierarchical structure used frequently in disk-based key-value stores such as LevelDB [49] and RocksDB [68]. At a high level, the LSM tree contains multiple levels of tree-like structures of increasing size. The BPA has the potential to improve the smallest level of the LSM tree that resides in-memory (i.e., the Memtable) because it must support concurrent point and range operations.

The BP-tree design takes inspiration from buffer trees, an extension of blocked trees optimized for minimizing writes in external memory [4, 32], but differs in terms of the tree structure and node organization. At a high level, buffer trees (e.g., the B^ε-tree [32]) have the same pointer-based structure as B-trees but have a buffer at each node (both internal and leaf nodes). In contrast, the BP-tree only buffers elements in the leaf nodes to optimize for the in-memory case. Existing buffer trees are designed (and implemented) for external memory [4, 21] where each block fetch has a high fixed cost. However, in the in-memory setting, internal nodes high in the tree are likely to be maintained in close caches, so the in-memory BP-tree buffers elements only in the leaves, which are most likely not be in cache. Furthermore, traditional buffer trees store the non-buffered elements in sorted arrays, while the BP-tree stores elements in the block structure described in Section 3 for reduced data movement.

8 CONCLUSION

This paper overcomes the decades-old point-range operation tradeoff in B-trees by developing an insert-optimized leaf-specific data structure. Traditionally, a user could decide to make B-tree nodes smaller to achieve the best possible point operation performance, or make the nodes larger to improve range operations but sacrifice point operations. This paper introduces the BP-tree, which overcomes this tradeoff by replacing the sorted array in traditional B-tree leaves with a specialized data structure called the buffered partitioned array.

The BP-tree is an ideal candidate for emerging applications that serve both range and point operations with the same data store.

On a suite of workloads generated from YCSB, the BP-tree supports short range queries between 1.6×-7.4× faster and long range queries between 2.5×-30× when compared to Masstree and OpenBw-tree, two state-of-the-art KV stores. The BP-tree is optimized for range queries but achieves similar or better (between .94×−1.2×) point operation throughput compared to Masstree and OpenBw-tree.

Future work includes integrating the BPA into the leaves of an enterprise B-tree-based key-value store such as MongoDB [54] or CouchDB [24]. Furthermore, although this paper implements and evaluates the BPA in the leaves of a B-tree for concreteness, the higher-level technique of replacing sorted arrays in the nodes of a blocked structure to overcome the empirical point-range tradeoff can be used in other blocked data structures such as CSB trees [36, 65], LSM trees [57], and cache-optimized skiplists [30, 73, 76, 82].

ACKNOWLEDGMENTS

We would like to thank Charles E. Leiserson for helpful discussions. Research was sponsored in part by the United States Air Force Research Laboratory and the Department of the Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Department of the Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein. Also, research was sponsored in part by the Advanced Scientific Computing Research (ASCR) program within the Office of Science of the Department of Enery (DOE) under contract number DE-AC02-05CH11231, the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

REFERENCES

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

[2] Amazon. [n.d.]. Amazon Web Services. https://aws.amazon.com/. Last accessed 11/1/2022.

[3] Matthew Andrews, Michael A. Bender, and Lisa Zhang. 2002. New algorithms for disk scheduling. *Algorithmica* 32, 2 (2002), 277–301.

[4] Lars Arge. 2005. The buffer tree: A new technique for optimal I/O-algorithms. In *Algorithms and Data Structures: 4th International Workshop, WADS'95 Kingston, Canada, August 16–18, 1995 Proceedings*. Springer, 334–345.

[5] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1185–1196.

[6] Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. 2020. KiWi: A Key-Value Map for Scalable Real-Time Analytics. *ACM Trans. Parallel Comput.* 7, 3, Article 16 (jun 2020), 28 pages. https://doi.org/10.1145/3399718

[7] Rudolf Bayer and Edward M. McCreight. 1972. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica* 1, 3 (1972), 173–189.

[8] Michael A. Bender, Alex Conway, Martín Farach-Colton, William Jannen, Yizheng Jiao, Rob Johnson, Eric Knorr, Sara McAllister, Nirjhar Mukherjee, Prashant Pandey, et al. 2019. Small refinements to the DAM can have big consequences for data-structure design. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*. 265–274.

[9] Michael A. Bender, Alex Conway, Martin Farach-Colton, William Jannen, Yizheng Jiao, Rob Johnson, Eric Knorr, Sara McAllister, Nirjhar Mukherjee, Prashant Pandey, Donald E. Porter, Jun Yuan, and Yang Zhan. 2021. External-memory Dictionaries in the Affine and PDAM Models. *ACM Trans. Parallel Comput.* 8, 3 (2021), 15:1–15:20. https://doi.org/10.1145/3470635

[10] Michael A. Bender, Erik D Demaine, and Martin Farach-Colton. 2005. Cache-oblivious B-trees. *SIAM J. Comput.* 35, 2 (2005), 341–358.

[11] Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman, et al. 1987. *Concurrency control and recovery in database systems*. Vol. 370. Addison-Wesley Reading.

[12] Timo Bingmann. 2018. TLX: Collection of Sophisticated C++ Data Structures, Algorithms, and Miscellaneous Helpers. https://panthema.net/tlx. Last accessed 10/7/2020.

[13] 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*. 521–534.

[14] Lucas Braun, Thomas Etter, Georgios Gasparis, Martin Kaufmann, Donald Kossmann, Daniel Widmer, Aharon Avitzur, Anthony Iliopoulos, Eliezer Levy, and Ning Liang. 2015. Analytics in Motion: High Performance Event-Processing AND Real-Time Analytics in the Same Database. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 251–264. https://doi.org/10.1145/2723372.2742783

[15] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A practical concurrent binary search tree. *ACM Sigplan Notices* 45, 5 (2010), 257–268.

[16] Trevor Brown and Hillel Avni. 2012. Range queries in non-blocking k-ary search trees. In *Principles of Distributed Systems: 16th International Conference, OPODIS 2012, Rome, Italy, December 18-20, 2012. Proceedings 16*. Springer, 31–45.

[17] Cassandra. [n.d.]. https://cassandra.apache.org/_/index.html. Last accessed 10/20/2022.

[18] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. 2016. Realtime Data Processing at Facebook. 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, 1087–1098. https://doi.org/10.1145/2882903.2904441

[19] Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. 2001. Improving index performance through prefetching. *ACM SIGMOD Record* 30, 2 (2001), 235–246.

[20] Douglas Comer. 1979. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)* 11, 2 (1979), 121–137.

[21] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. 2020. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 49–63.

[22] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[23] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). MIT Press.

[24] CouchDB. [n.d.]. https://couchdb.apache.org/. Last accessed 10/20/2022.

[25] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.

[26] Laxman Dhulipala, Guy E. Blelloch, Yan Gu, and Yihan Sun. 2022. Pac-trees: Supporting parallel and compressed purely-functional collections. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 108–121.

[27] Panagiota Fatourou, Elias Papavasileiou, and Eric Ruppert. 2019. Persistent non-blocking binary search trees supporting wait-free range queries. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*. 275–286.

[28] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux journal* 2004, 124 (2004), 5.

[29] Anil K. Goel, Jeffrey Pound, Nathan Auch, Peter Bumbulis, Scott MacLean, Franz Färber, Francis Gropengiesser, Christian Mathis, Thomas Bodner, and Wolfgang Lehner. 2015. Towards Scalable Real-Time Analytics: An Architecture for Scale-out of OLxP Workloads. *Proc. VLDB Endow.* 8, 12 (aug 2015), 1716–1727. https://doi.org/10.14778/2824032.2824069

[30] Daniel Golovin. 2010. The B-Skip-List: A Simpler Uniquely Represented Alternative to B-Trees. arXiv:1005.0662 [cs.DS]

[31] Google. [n.d.]. BTree implementation for Go. https://pkg.go.dev/github.com/google/btree. Last accessed 3/1/2023.

[32] Goetz Graefe. 2004. Write-optimized B-trees. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. 672–683.

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

[34] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. 1994. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*. 243–252.

[35] Rui Han, Zhen Jia, Wanling Gao, Xinhui Tian, and Lei Wang. 2015. Benchmarking Big Data Systems: State-of-the-Art and Future Directions. arXiv:1506.01494 [cs.PF]

[36] Richard A Hankins and Jignesh M Patel. 2003. Effect of node size on the performance of cache-conscious B+-trees. In *Proceedings of the 2003 ACM SIGMETRICS international conference on measurement and modeling of computer systems*. 283–294.

[37] HBase. [n.d.]. https://hbase.apache.org/. Last accessed 10/20/2022.

[38] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, et al. 2014. Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the eighth international workshop on data mining for online advertising*. 1–9.

[39] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. 2020. *The art of multiprocessor programming*. Newnes.

[40] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S. Kester, and Demi Guo. 2018. The data calculator: Data structure design and cost synthesis from first principles and learned cost models. In *Proceedings of the 2018 International Conference on Management of Data*. 535–550.

[41] Intel. [n.d.]. Intel 64 and IA-32 Architectures Optimization Reference Manual. https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf. Last accessed 3/1/2023.

[42] Alon Itai, Alan G. Konheim, and Michael Rodeh. 1981. A sparse table implementation of priority queues. In *ICALP*. 417–431.

[43] Vijay Janapa Reddi, Benjamin C. Lee, Trishul Chilimbi, and Kushagra Vaid. 2010. Web search using mobile cores: quantifying and mitigating the price of efficiency. *ACM SIGARCH Computer Architecture News* 38, 3 (2010), 314–325.

[44] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael A. 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, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*, Jiri Schindler and Erez Zadok (Eds.). USENIX Association, 301–315. https://www.usenix.org/conference/fast15/technical-sessions/presentation/jannen

[45] Donald E. Knuth. 1998. *The Art of Computer Programming* (3rd ed.). Fundamental Algorithms, Vol. 1. Addison Wesley Longman Publishing Co., Inc. (book).

[46] 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.

[47] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. RECIPE: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Ontario, Canada, 462–477.

[48] 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)*. IEEE, 38–49.

[49] LevelDB. [n.d.]. http://ccrma.stanford.edu/jos/bayes/bayes.html. Last accessed 10/20/2022.

[50] Cheng Li, Yue Lu, Qiaozhu Mei, Dong Wang, and Sandeep Pandey. 2015. Click-through prediction for advertising in twitter timeline. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1959–1968.

[51] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. 2017. BatchDB: Efficient isolated execution of hybrid OLTP+ OLAP workloads for interactive applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 37–50.

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

[53] Andreas Meier and Michael Kaufmann. 2019. Nosql databases. In *SQL & NoSQL databases*. Springer, 201–218.

[54] MongoDB. [n.d.]. https://www.mongodb.com/. Last accessed 10/20/2022.

[55] Bradford Nichols, Dick Buttlar, Jacqueline Farrell, and Jackie Farrell. 1996. *Pthreads programming: A POSIX standard for better multiprocessing*. " O'Reilly Media, Inc.".

[56] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, et al. 2011. The case for RAMCloud. *Commun. ACM* 54, 7 (2011), 121–130.

[57] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[58] Prashant Pandey, Yinjie Gao, and Carl Kingsford. 2021. VariantStore: an index for large-scale genomic variant search. *Genome biology* 22, 1 (2021), 1–25.

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

[60] Apostolos N. Papadopoulos, Spyros Sioutas, Christos Zaroliagis, and Nikolaos Zacharatos. 2019. Efficient distributed range query processing in apache spark. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 569–575.

[61] Jure Petrovic. 2008. Using memcached for data distribution in industrial environment. In *Third International Conference on Systems (icons 2008)*. IEEE, 368–372.

[62] Markus Pilman, Kevin Bocksrocker, Lucas Braun, Renato Marroquin, and Donald Kossmann. 2017. Fast scans on key-value stores. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1526–1537.

[63] Pouria Pirzadeh, Junichi Tatemura, Oliver Po, and Hakan Hacıgümüş. 2012. Performance evaluation of range queries in key value stores. *Journal of Grid Computing* 10, 1 (2012), 109–132.

[64] PostgreSQL. [n.d.]. https://www.postgresql.org/. Last accessed 10/20/2022.

[65] Jun Rao and Kenneth A. Ross. 2000. Making B+- Trees Cache Conscious in Main Memory. *SIGMOD Rec.* 29, 2 (may 2000), 475–486. https://doi.org/10.1145/335191.335449

[66] Kai Ren and Garth Gibson. 2013. TABLEFS: Enhancing Metadata Efficiency in the Local File System. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 145–156.

[67] Vincent Reniers, Dimitri Van Landuyt, Ansar Rafique, and Wouter Joosen. 2017. On the State of NoSQL Benchmarks. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion* (L'Aquila, Italy) *(ICPE '17 Companion)*. Association for Computing Machinery, New York, NY, USA, 107–112. https://doi.org/10.1145/3053600.3053622

[68] RocksDB. [n.d.]. http://rocksdb.org/. Last accessed 10/20/2022.

[69] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)* 9, 3 (2013), 1–32.

[70] Chris Ruemmler and John Wilkes. 1994. An introduction to disk drive modeling. *Computer* 27, 3 (1994), 17–28.

[71] ScyllaDB. [n.d.]. https://www.scylladb.com/. Last accessed 10/20/2022.

[72] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 135–146.

[73] 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.

[74] Yihan Sun, Daniel Ferizovic, and Guy E. Blelloch. 2018. PAM: parallel augmented maps. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 290–304.

[75] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G Andersen. 2018. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*. 473–488.

[76] Zhongle Xie, Qingchao Cai, HV Jagadish, Beng Chin Ooi, and Weng-Fai Wong. 2016. PI: a Parallel in-memory skip list based Index. *arXiv preprint arXiv:1601.00159* (2016).

[77] YCSB. [n.d.]. Core Workloads. https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads. Last accessed 2/15/2023.

[78] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2016. Optimizing Every Operation in a Write-optimized File System. In *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*, Angela Demke Brown and Florentina I. Popovici (Eds.). USENIX Association, 1–14. https://www.usenix.org/conference/fast16/technical-sessions/presentation/yuan

[79] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2017. Writes Wrought Right, and Other Adventures in File System Optimization. *ACM Trans. Storage* 13, 1 (2017), 3:1–3:26. https://doi.org/10.1145/3032969

[80] 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*. 1567–1581.

[81] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. 2015. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering* 27, 7 (2015), 1920–1948.

[82] 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.