Bf-Tree: A Modern Read-Write-Optimized Concurrent Larger-Than-Memory Range Index

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

Badrish Chandramouli
Microsoft Research
badrishc@microsoft.com

ABSTRACT

A B-Tree is the most widely used range index for larger-than-memory data systems. It organizes data in pages (usually 4 KB) that efficiently align with disk IO operations, fully utilizing each IO operation to narrow down the search space. On the other hand, a B-Tree's page-based organization leads to inefficient caching and high write amplification, as it needs to cache the entire page as a whole while often only a small subset of records are hot, and it needs to write the entire page for a single record update.

The key insight of this paper is to *separate cache pages from disk pages*, i.e., a cache page is no longer a pure mirror of its disk content, but instead, it forms a judiciously chosen subset of the disk page that is worth caching, and can absorb both read and write operations in a consistent manner. Based on this insight, we propose Bf-Tree, a modern B-Tree that is *read-write-optimized* by building a new variable-length buffer pool to manage such cache pages, called *mini-pages*. Bf-Tree uses this in-memory buffer pool to support efficient record-level caching, buffering recent updates, caching range gaps, as well as mirrors of disk pages when needed. We implement a fully featured and modern Bf-Tree in Rust with 13k lines of code, and show that Bf-Tree is 2.5× faster than RocksDB (LSM-Tree) for scan operations, 6× faster than a B-Tree for write operations, and 2× faster than both B-Trees and LSM-Trees for point lookups. We believe these results firmly establish a new standard for database storage engines of the future.

PVLDB Reference Format:
Xiangpeng Hao and Badrish Chandramouli. Bf-Tree: A Modern Read-Write-Optimized Concurrent Larger-Than-Memory Range Index. PVLDB, 17(11): 3442 - 3455, 2024.
doi:10.14778/3681954.3682012

1 INTRODUCTION

A B-Tree is the most important range index for larger-than-memory data processing systems. It organizes data in pages that align well with block devices such as SSDs. This allows each disk IO (usually 4KB) to load exactly one page of the B-Tree from the disk, and the entire page is used to narrow down the search space.

However, a B-Tree is not a silver bullet for all workloads, as it faces two key challenges: (1) it incurs high write amplification: a small modification to a page requires us to write the entire page back to disk and (2) its native caching strategy is inefficient, as it caches data in memory at a page granularity, even if only a small subset of records on the page are hot. These problems are exacerbated when a B-Tree is used as a secondary index with small keys and values.

Many solutions have been proposed to mitigate these problems. LSM-Trees [10, 13, 16, 31, 37, 48] are the most notable alternative; they use log-structured writes (appends) to mitigate write amplification problem but incur a higher cost for read and compaction. Many studies have proposed to enhance the B-Tree itself [24], e.g., Bw-Tree [43, 61] and Bϵ-Tree [3, 11] employ *delta records* to reduce write amplification by chaining updates and batch writing them. Two-Tree [27, 69], Anti-Caching [15], Siberia [18], and Treeline [65] employ a separate *record cache* to improve caching efficiency.

These enhancements, however, are complex and introduce new challenges. For example, delta records improve write amplification but slow down read and scan operations: as a page's chain gets longer, it incurs excessive random memory access. A record-cache allows efficient point lookup but does not apply to range scans, leaving a smaller memory budget for caching scan records. They are also hard to maintain consistently and tune for memory usage relative to the page cache. Overall, modern B-Trees require a holistic re-design to address the aforementioned challenges and better optimize for both read and write operations.

Interestingly, the root cause of both problems comes from the core B-Tree design principle of page-based data organization: records of a page are *coupled* together and are transferred between memory

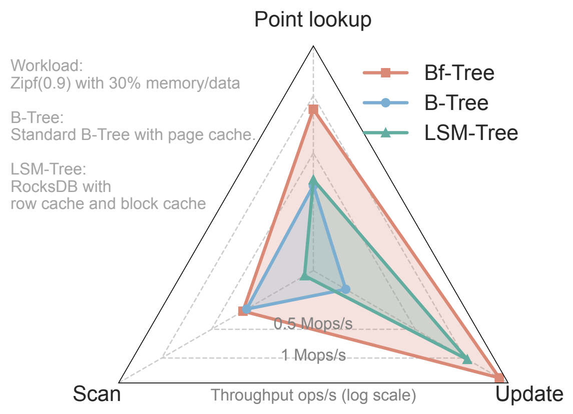

Figure 1: Throughput (log-scale) along three dimensions – point lookup, scan, and update. Each triangle is a different system; larger the better. Bf-Tree covers B-Tree and LSM-Tree along all three dimensions. Section 6.3 covers this experiment in detail.

*Research done during internship at Microsoft Research.

Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:10.14778/3681954.3682012

and disk as a whole. The fundamental tension is that the disk page size is much larger than the record size, and such coarse-grained data management limits performance. As a thought experiment, for example, note that if the disk page size equaled the record size, both problems would disappear.

The key insight of this paper is that we can *separate cache pages from disk pages*, i.e., the cache pages are no longer a mirror of their disk content. Instead, they contain a judiciously chosen subset of the disk page that is worth caching. Called *mini-pages*, these cached pages are a native part of the tree's memory component and can be both read from and written to consistently. Mini-pages have the freedom to cache individual hot records, a key range, the original page, and/or serve as a succinct buffer for recent updates. This design leads to more efficient point lookup, range scan, and write operations.

We crystallize this insight into a new data structure called the Bf-Tree[1]. Bf-Tree is a modern B-Tree built from the ground up using Rust, characterized by a new variable-length in-memory buffer pool to store mini-pages. Mini-pages serve three purposes: (1) cache frequently accessed records, (2) buffer recent updates, and (3) cache a *range gap* [24] (a range between two keys). This is made possible by changing the mini-page size dynamically: it can grow larger to accommodate more updates and cache more records. It may even grow to the full page size (4 KB) to allow efficient cross-node range scans. It can also shrink to allow other mini-pages to grow.

Variable-length mini-pages are ideal for addressing the above problems, but memory management of mini-pages faces challenges of memory alignment, fragmentation, and resource utilization. Specifically, the variable-length buffer pool that serves mini-pages must satisfy the following four requirements: (1) constrain the memory consumption of mini-pages to a configured value; (2) manage and track used/unused memory, i.e., allocate and de-allocate mini-pages; (3) identify hot and cold mini-pages: grow hot mini-pages while shrink or evict cold mini-pages. (4) interact with the on-disk leaf pages to ensure consistency and data integrity. Further, all these requirements must be handled efficiently in a multi-threaded (concurrent) setting.

To this end, we propose a novel variable-length buffer pool built upon a circular buffer. The circular buffer has a fixed total size, and all mini-pages are stored in the circular buffer. Allocation of mini-pages is done by advancing the tail pointer of the circular buffer, while deallocation is done by adding the memory region to a free list, which will be reused for future allocation. Growing and shrinking mini-pages is done by allocating a new mini-page and copying the content of the old mini-page to the new one as a *read-copy-update*. When the circular buffer is full, the mini-pages close to the head pointer will be evicted to disk, making room for new mini-page allocation.

Evaluations (Figure 1, Section 6) show that Bf-Tree with mini-pages has high cache-efficiency and low write-amplification in a YCSB-like benchmark for all point lookup, scan, and write operations. **Specifically, Bf-Tree has 2.5× higher throughput than RocksDB (LSM-Tree) for scan operations, 6× faster than a B-Tree for write operations, and 2× faster than both B-Trees and LSM-Trees for point lookup.** We believe these results firmly establish a new standard for database storage engines of the future.

The contributions of this paper are summarized below:

- We propose Bf-Tree, a new concurrent larger-than-memory range index that outperforms B-Trees and LSM-Trees on all measured aspects for larger-than-memory workloads and is on par with in-memory B-Trees for main-memory workloads.
- We design a new mini-page abstraction along with the first-of-its-kind practical variable-length buffer pool for mini-pages.
- We implement Bf-Tree as a fully featured index from scratch using modern Rust with strong correctness guarantees.
- We experimentally analyze the performance of Bf-Tree using a comprehensive set of workloads and show that Bf-Tree outperforms state-of-the-art B-Trees and LSM-Trees.

2 BACKGROUND

We start by discussing recent advancements in the design of B-Trees and LSM-Trees, the two most important data structures for larger-than-memory data systems. Although B-Trees and LSM-Trees are traditionally optimized for read and write-intensive workloads respectively, practical use cases must handle both workloads simultaneously. Therefore, they have each been extended to manage the opposing workload type efficiently.

2.1 B-Tree

A larger-than-memory B-Tree organizes data in pages and places hot pages in memory while cold pages reside on disk. It assigns each page with a unique ID and maintains a mapping from the page ID to the physical location of the page (on disk or in memory). This allows a B-Tree to grow larger than memory (by evicting cold pages to disk) but still allows efficient access to data on disk.

Point lookup. A B-Tree excels in efficient point lookups by traversing the tree from the root to the leaf node and then binary searching the leaf node for the target key-value pair. By caching inner nodes in memory [15], a B-Tree can complete a point query with a single IO operation involving a leaf node retrieval from disk. This design significantly enhances lookup speed for larger-than-memory data.

Despite their efficiency in disk lookups, B-Trees face challenges in caching hot records, especially when they are interspersed with cold ones on the same page. B-Tree's page-based caching brings the entire page in memory – caching hot records along with cold ones – leading to low cache efficiency. To mitigate this problem, multiple systems – such as Two-Tree [69], Siberia [18], Anti-caching [15], and Tree-line [65] – have incorporated record-level caching (i.e., cache individual records rather than entire pages) in their system to improve caching efficiency. While effective for point lookups, record-level caching cannot operate alone, as it does not benefit range scans, necessitating page-level caching just for range queries. Further, as a separate component, record caches are not helpful for writes. With two caching components, the system needs a delicate balance in memory allocation between the two types of caches.

Write. A write operation in a B-Tree often faces significant write amplification: a single update to a key-value pair requires a write to the entire page. This is because the record size (often less than

[1]The 'f' in Bf-Tree stands for "faster".

100 bytes [6]) is much smaller than the page size (ranging from 4KB to 64KB). A typical write operation on a B-Tree leaf page often involves reading the entire page from disk, modifying the page in memory, and writing the entire page back to disk. Such inefficiency is problematic in write-intensive workloads, prompting a preference for LSM-Trees, which are better suited to these scenarios.

To mitigate write amplification, researchers have proposed various modifications [3, 11, 43, 61] to a conventional B-Tree. Among these, the Bw-Tree's [43] delta chain approach represents a notable advancement. A write to a leaf page is "blindly" appended to an in-memory delta chain without touching the leaf page. As the delta chains grow, they eventually merge to the leaf page in a batched manner, thereby reducing write amplification. However, long delta chains slow down search during both read and write operations, as they need to pointer-chase the delta chain to find the target key-value pair.

Range scan. B-Trees support efficient range scans by chaining the leaf nodes for efficient forward and backward scans. Moreover, a B-Tree's page-based caching is ideal for range scans, as the entire page is cached in memory, preserving spatial localities of neighboring records.

2.2 LSM-Tree

The Log-Structured Merge-Tree (LSM-Tree [16, 23, 53]) complements the B-Tree, especially for write-intensive workloads. LSM-Trees allow efficient data ingestion but incur higher costs for the read workloads. Despite this trade-off, LSM-Trees are widely used in industry, e.g., RocksDB [16], TiDB [31], and Bigtable [10].

Write. LSM-Trees append incoming writes to an in-memory buffer and periodically flush the buffer to disk. When writing to disk, records are sorted and merged with existing data, a process known as compaction. LSM-Trees employ a multi-level file organization to manage compaction efficiently. Despite the optimizations, compaction remains resource-intensive, often leading to IO amplification and significant read latencies at the higher percentiles. Recent works have proposed to mitigate compaction overheads by offloading it from the critical path [68] and reducing data writes [48].

Point lookup. During a point lookup, an LSM-Tree needs to binary-search multiple levels of potentially large files. LSM-Trees employ many techniques to accelerate point lookups. Bloom filters [14, 20] serve as file-level filters that can terminate unnecessary file searches. A block cache [21, 49] caches data blocks from the SST file, reducing the number of disk IOs. A row cache[55] is a record-level cache that caches individual records in memory; it is more fine-grained than a block cache but does not apply to range scans.

Range scan. LSM-Trees are inefficient at range scans, as they need to search all levels of the tree to find all records in the given range and merge them to get the final result. Moreover, an LSM-Tree's row cache does not apply to range scans, leading to low cache efficiency for scan-intensive workloads.

2.3 Modern NVMe SSD

Disks have improved significantly in the past few decades, especially with the advent of modern NVMe SSDs. Although conventional systems continue to work with new hardware, many of their assumptions have changed, particularly those designed around the limitations of conventional HDDs. As a result, the operational trade-offs of B-Trees and LSM-Trees warrant a reevaluation in the context of new hardware capabilities.

Random vs. sequential writes. Conventional HDDs use spinning disks to locate data, so sequential writes are much faster than random writes. Log-structured write is a good fit for such hardware as it only requires sequential writes. Modern NVMe SSDs, on the other hand, have no mechanical moving parts, use flash memory for efficient parallel access, and incorporate fast hardware garbage collection schemes [62, 63]. This allows modern NVMe SSDs to have almost as fast random write as sequential write; recent studies [26, 65] suggest that 4KB random writes can almost saturate an SSD's bandwidth, making log-structured writes less attractive.

Kernel bypass IO. Conventional IO operations require the kernel to act as an intermediary, transferring data between the device and user space. This incurs overheads of context switches and dual copies of data. With modern NVMe SSDs, the IO latency can be a few microseconds, comparable to a context switch; the bandwidth can be >10GB/s [26], a significant fraction of memory bandwidth. SPDK[64] and io_uring[2] are two mechanisms that allow applications to bypass the kernel and access storage devices directly, eliminating unnecessary data copies and context switches.

B-Tree page size. Smaller page sizes are preferred to reduce amplification, especially on modern NVMe SSDs that support finer-grained access. Bf-Tree therefore by default uses 4KB page size instead of more common 8-32 KB page sizes [22, 32, 39, 52]. Recent evaluation [26] on B-Tree page size shows that modern SSD favors 4KB page size, smaller page size results in worse IOPS and latency due to excessive overhead in the flash translation layer and not optimized 512-byte page access. This is consistent with our preliminary experiment that 4KB page size is the sweet point on a PCIe 4.0 modern NVMe SSD [56].

3 BF-TREE ARCHITECTURE

Two major problems of conventional B-Trees – write amplification and inefficient caching – stem from the fact that disk pages are much larger than individual records. This granularity mismatch fundamentally limits the performance of B-Trees, forcing them to either cache cold records or write the entire page for a single record update.

A B-Tree's page organization is suitable for block devices and naturally implements page-based caching. However, the cache does not have to be page-based, as memory is byte-addressable. Bf-Tree starts with this thought experiment: *What if cached pages can have variable lengths?* By having variable-length pages, memory caches no longer need to align to disk pages. Instead, they have the freedom only to cache the data that is worth caching. In Bf-Tree, this means it can cache (1) only the "read hot" part of a page to serve reads efficiently and (2) the "write hot" part of a page to absorb updates (often in-place) as much as possible before batch writes to disk. It can also grow to cache the entire range gap if needed.

This section presents the high-level design of Bf-Tree, a modern B-Tree re-imagined with a native variable-length buffer pool for

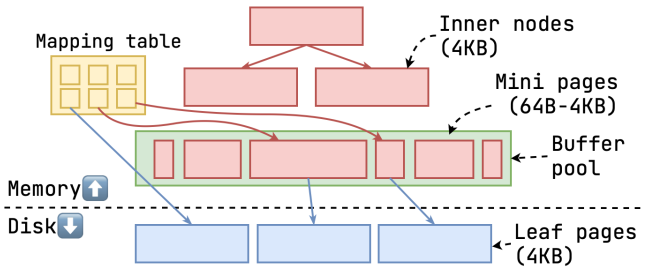

Figure 2: High level architecture of Bf-Tree. Like conventional B-Tree, but pages in the buffer pool are variable lengths.

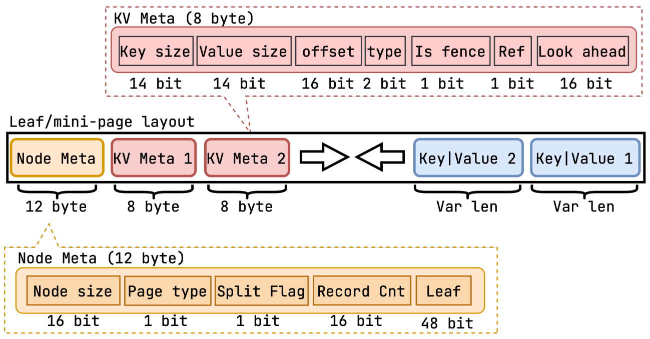

Figure 3: Mini-page (var len) / leaf node (4096 bytes) layout

memory. Bf-Tree can leverage variable-length cache entries, called *mini-pages*, to support efficient point lookup, range scan, and write operations. While this design can help address the aforementioned problems, it comes with new concurrency, memory management, and resource utilization challenges. Section 4 will discuss how such a buffer pool can be efficiently implemented for a Bf-Tree.

Figure 2 shows the high-level architecture of the Bf-Tree. It consists of four parts: (1) the inner nodes; (2) the buffer pool that caches mini-pages; (3) the on-disk leaf pages; and (4) the mapping table for leaf and mini pages. At a high level, the Bf-Tree's architecture is not much different from the conventional B-Tree, except that its buffer pool supports variable length pages.

The rest of this section discusses the design of each component in detail, specifically, where we store them in memory or on disk and how they interact with each other. The last sub-section discusses the optimizations we implemented to improve Bf-Tree's performance.

3.1 Mini-page

A mini-page is an in-memory slim version of the corresponding leaf page. It serves two purposes: (1) to buffer recent updates and (2) to cache frequently accessed records. Mini-pages are for leaf pages only (i.e., not for inner nodes), and each leaf page may have *at most one* corresponding mini-page.

Records in a mini-page are maintained as sorted, preserving spatial locality. This allows records to be efficiently searched using binary search, unlike the delta chain approach that requires a sequence of pointer chases to find the target record. We next discuss the high-level functionality of a mini-page. Section 5 will discuss in detail how mini-pages are used in Bf-Tree's core operations.

Absorbing write operations. A write operation tries to insert to the mini-page of the destination leaf page. If the leaf page does not already have a mini-page, it creates a minimal-sized (e.g., 64 bytes to align with a cache line) mini-page that can contain the new record. If the mini-page is full, it grows to accommodate the new records. Each time, the mini-page doubles its size until it can accommodate the new record. Ultimately, the mini-page can grow too large (up to 4KB), which can cause the insertion/search performance to degrade. Then, or when it needs to be evicted from memory, we batch-write and merge the mini-page into the base leaf page. We will discuss new mechanisms for a fast concurrent buffer pool implementation for mini-pages in Section 4.

Caching hot records. Before reading the leaf page from disk, a read operation first (binary) searches the mini-page for the desired record and terminates early if the record is found. Searching the mini-page is efficient as the records are sorted and in-memory. If the record is not found in the mini-page, we load the corresponding leaf page from disk.

After reading the record from disk, we can cache it by inserting it into the mini-page. This allows future read operations to terminate the search early. Note that the mini-page will cache the individual records, not the entire page, avoiding the inefficient page-level caching of conventional B-Trees. To avoid flooding the mini-page with cold records, we only cache the records from the leaf page at a low probability, e.g., 1%. Caching hot records is implemented as an in-memory insert operation to the mini-page, which may trigger the mini-page to grow as needed.

Caching range gaps. Record caching does not help with range scans because the range query has to look at the leaf page for the full set of records; in other words, record caching breaks the spatial locality needed for range scans. On the other hand, a page-level cache is ideal for range scans, as it preserves the spatial locality of records. Mini-pages support caching range gaps: when a mini-page grows to the full size, we merge (if necessary) and convert it into a full leaf page mirroring disk. Unlike systems such as RocksDB, where static memory partitioning is required for row-cache and block-cache, mini-pages in Bf-Tree automatically adapt to workload changes (e.g., point or range query intensive). This allows the same memory budget to be used for both range and point queries.

3.2 Mini and leaf page layout

Mini-pages and leaf pages share the same layout, storing key-value pairs in sorted order and allowing efficient lookups. In Bf-Tree, they share the same implementation, except that mini-pages can have varying lengths. This significantly reduces the complexity of the system and allows us to implement optimizations once and apply them to both mini-pages and leaf pages.

As shown in Figure 3, the page layout starts with a 12-byte Node Meta, which encodes the node size, page type (mini or leaf page), split flag (whether the page is full), and value count (the number of records on the page).

The Node Meta is followed by an array of KV Meta, which stores the metadata of the key-value pair. The metadata and key-value data are stored separately to support variable-length keys and values. The KV Meta is stored from the beginning of the page, and the actual key-value data is stored from the end of the page. The node is full when the KV Meta and the key-value data meet in the middle. Separating the metadata and the actual data also allows efficient insertion, as we only need to shift the metadata when inserting a new record instead of shifting the entire node.

Each KV Meta is 8 bytes, which stores the length of the key and value and the offset of the key and value in the page. It also stores the type of the key and value (more detail on Table 1), whether the key is a fence key (Section 3.5), the reference flag of the key-value pair (Section 5.5), and finally, the look-ahead bytes of the key (Section 3.5). The key-value data is stored sequentially on the page, starting at the specified offset in the KV Meta.

3.3 Leaf pages and mapping table

Leaf pages are always on disk, and may have at most one associated mini-page in memory. To locate the leaf page and its mini-page, the last level inner node stores a page ID, which references a mini-page or a leaf page. Bf-Tree maintains a mapping table that maps the page ID to the actual location. If the page ID points to a mini-page, its leaf page address can be found in the leaf field of the page header.

As shown in Figure 2, the mapping table is an in-memory container that maps the logical page ID to the physical location of the page (i.e., memory address or disk offset). The mapping table can be implemented using a hash table/indirection array [22, 36, 52], relying on the OS's page table [12, 29], or using pointer swizzling [39]. Bf-Tree uses an indirection array for its simplicity and performance. However, our design can work with any of the above approaches, as the mapping table is decoupled from the rest of the system.

In addition to the address translation, the mapping table stores the reader-writer lock for each page. The reader-writer lock (16 bits) is co-located with the page address (48 bits); they are together packed into a 64-bit word for efficiency. As the mini-page and its leaf page share the same page ID, locking the mini-page will also lock the leaf page, simplifying the locking mechanism and reducing overhead.

3.4 Inner nodes

Conventional B-Tree systems treat inner nodes and leaf nodes indistinguishably and use the same mapping table to translate an inner node page ID to a physical address. This creates a high overhead of inner node access (due to translation) and a contention hotspot on the mapping table. Inner nodes of practical B-Tree systems usually take less than 1% of the total B-Tree size [15, 24], and they are much more frequently accessed than leaf pages (e.g., every leaf page access will involve multiple inner node accesses).

Bf-Tree by default pins the inner nodes in memory and uses direct pointer addresses to reference them. This allows a simpler inner node implementation, efficient node access, and reduced contention on the mapping table. As inner nodes are pinned to memory, the buffer pool of Bf-Tree only needs to cache leaf pages. Pinning inner nodes to memory is not a design requirement, when deployed with constrained memory budget, users can choose to disable inner node pinning, or only pin the first few levels of inner nodes to memory. When a inner node is not pinned to memory, it will be accessed through the mapping table using a page ID instead of a direct memory address (similar to leaf pages).

Bf-Tree implements optimistic latch-coupling [41] on inner nodes to reduce their contention – based on the observation that although highly contended, inner nodes are rarely modified (only modified on nodes split/merge). Specifically, we use an 8-byte version lock to track the state of the inner node. A read operation compares the version number before and after the operation and only proceeds when the version number matches. A write operation will acquire an exclusive lock on the inner node and bump the version lock after the modification. Optimistic latch coupling for inner nodes allows Bf-Tree to scale to high concurrency, as the read operation does not pollute cache lines, thus avoiding cache coherence traffic.

3.5 Performance optimizations

Now we discuss the optimizations made to Bf-Tree's node layout, which apply to inner nodes, leaf nodes, and mini-pages.

Fence keys. Fence keys guard the key range of a page and determine the neighbor nodes for range queries. The first key of a mini/leaf page is the low fence key, and the second is the high fence, followed by the actual records. The low fence points to the node's left neighbor, and the high fence points to its right neighbor. Alternative approaches are possible, e.g., chained pointers. Bf-Tree uses fence keys for their simplicity.

Prefix compression. Keys of Bf-Tree can have long prefixes, e.g., URL keys and names. To reduce memory usage and accelerate the key search, Bf-Tree implements prefix compression based on the fence keys. The node's prefix is implicitly stored in the fence keys as they tell the node's key range; i.e., the common prefix of the low and high fence keys is the node's prefix. When inserting records into a node, the common prefix is skipped, and only the suffix is stored in the node; this reduces space consumption and allows higher fan out. To read the full key, we assemble the key's prefix and suffix stored in the node.

Look-ahead bytes. Figure 2 shows that Bf-Tree stores the KV pair's metadata and actual data apart. While this design has the benefits mentioned earlier, it can incur higher random memory access for key comparison: we first load the record's metadata, then use it to load the actual key. To accelerate this two-step pointer chasing, we store the first 2 bytes of the actual key (called look-ahead bytes) in the metadata and compare the look-ahead bytes first. Thanks to prefix compression, the first 2 bytes of the keys are usually different, and we can terminate the search early without loading the full key. We only need to load and compare the full key if the look-ahead bytes are the same.

4 BUFFER POOL FOR MINI-PAGES

So far, we have described how mini-pages can significantly reshape the design of a B-Tree, and how Bf-Tree leverages mini-pages to

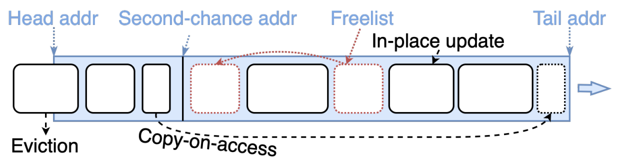

Figure 4: Circular buffer for variable length mini-pages.

achieve its efficiency goals. We next discuss how to manage the memory of mini-pages, as mini-pages can be of different sizes and grow and shrink dynamically.

The variable-length buffer pool has three challenges: (1) manages the memory of all mini-pages, i.e., the exact memory location of each mini-page, (2) tracks the hotness of each mini-page and evicts the cold mini-pages when needed. (3) concurrency challenge of evicting and allocating by many threads while maintaining memory safety and parallelism. The first challenge has to deal with memory fragmentation (like most allocators [19, 38]): When a large chunk of continuous memory is broken into smaller chunks of variable sizes, it is difficult to assemble them back to form a large chunk. The second challenge is to systematically decide which mini-page should be evicted and which should be kept in memory. The third challenge is safely issuing enough parallel IO requests to saturate SSD bandwidth.

Bf-Tree solves the problems above by designing a novel circular buffer to manage mini-pages: we store the mini-pages in the circular buffer, adding to the tail until the buffer is full. When it is full, the mini-pages close to the head address are evicted to disk, allowing the tail address to advance. This design is inspired by FASTER's hybrid log design [8, 9], where a similar goal is achieved. We next discuss how the circular buffer is designed and implemented to work in a concurrent setting.

4.1 Memory regions of circular buffer

The nature of the circular buffer evicts all mini-pages that reach the head address. This is undesirable behavior for hot mini-pages. As shown in Figure 4, the circular buffer has three addresses: the head, tail, and second-chance addresses. These three addresses divide the memory into 90% in-place-update region (between tail address and second-chance address) and 10% copy-on-access region (the rest). Mini-pages on the in-place-update region can be modified in place, while mini-pages on the copy-on-access region will be copied to the tail address on access (Section 5.5). This prevents hot mini-pages from reaching the head address and getting evicted.

The circular buffer also maintains multiple *free lists*, each with a different size class, to track recently de-allocated memory. A mini-page de-allocation happens when it tries to grow or shrink. This involves first allocating a new mini-page, copying the content of the old mini-page to the new one, and finally de-allocating the old mini-page. The de-allocated memory is added to a free list and reused for future allocation.

4.2 Circular buffer API

The circular buffer provides a succinct yet efficient API to support various operations on mini-pages.

Alloc. Memory for a mini-page can be allocated from two places: the free list of the requested size category and the tail address. When the free list has no memory, we allocate from the tail address by advancing the tail address by the requested size. When the circular buffer is full, i.e., the physical locations of tail and head addresses are close to each other; the circular buffer returns an error. The caller will then call eviction to make room for new allocations.
Eviction. Eviction is the process of making room for new allocations. Eviction starts from the mini-page closest to the head address. A callback function is invoked to merge dirty records in the mini-page to the leaf page on disk. Then, the mapping table is updated to point to the leaf page, and the head address is advanced. Eviction may happen simultaneously from multiple threads (more below).
Dealloc. De-allocation simply adds the memory region to the corresponding free list, for reuse during a future allocation.

4.3 Performance optimizations

The circular buffer is central to Bf-Tree's design and can easily become a performance bottleneck if not optimized sufficiently.
Memory fragmentation. There's no paging concept in the circular buffer, i.e., mini-pages are allocated with no spaces between them. Each allocated mini-page has an 8-byte metadata that stores its size and state (e.g., ready or free-listed). The meta-data is stored right before the mini-page. This design minimizes fragmentation as mini-pages do not need to align to specific memory boundaries.
Memory alignments. Without paging, each mini-page is only aligned to an 8-byte boundary; this means that some mini-pages may cross the physical 4KB paging boundary, incurring additional page table lookup while accessing the mini-page. This is mitigated using the Huge Table provided by the Linux kernel, where the entire physical memory of the circular buffer is backed by huge pages of 2MB or 1GB, heavily reducing the likelihood of mini-pages crossing the page boundary.
Concurrent evictions. Eviction ensures that cold mini-pages are evicted to disk, making room for hotter mini-pages. Eviction bumps the head address so that the tail address can be advanced. This is a sequential operation because the head address can only increase linearly. To parallelize the eviction process, we allow each thread to start evicting a mini-page concurrently but require all threads to finish the eviction in order, i.e., the head address can only be advanced when all threads have finished their eviction.

5 BF-TREE CORE OPERATIONS

So far, we have discussed the core mechanisms of Bf-Tree and the mini-pages. This section discusses how Bf-Tree connects the components to support efficient read, write, and range scans.

5.1 Get

The get operation starts with traversing the tree to the mini-page that may contain the target key-value pair and searching the mini-page for the key. If the record is found in the mini-page (cached), we will return the record and terminate the operation early. If the record is not found (or no mini-page exists), we search the corresponding leaf page on disk. We load the leaf page using the offset stored in the mini-page header (or the mapping table) and search

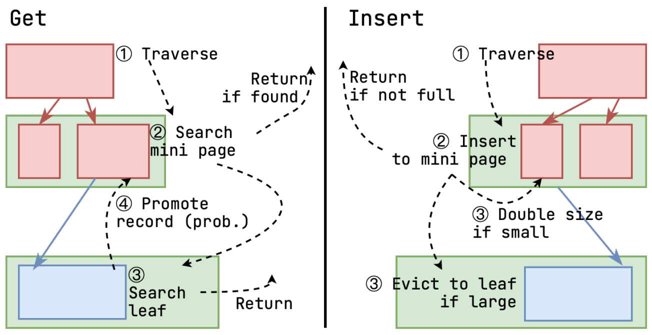

Figure 5: Bf-Tree's Get and Insert operations.

the page for the target key-value pair. With a small probability (defaulting to 1%), we cache the record by inserting it into the mini-page, creating a new mini-page if one does not already exist. This allows subsequent searches for that record to complete in memory. Algorithm 1 summarizes the get operation in pseudo-code.

```
def get(key):
    (mini_page, leaf_page) = traverse(key)
    if mini_page:
      result = mini_page.binary_search(key)
      if result:
        # early terminate if found
        return result

  # otherwise, read from the leaf page
  result = leaf_page.binary_search(key)
  if rand() < 0.01:
    # with a small probability, cache it in the mini-page
    mini_page.insert_or_create(result)
  return result
```

Listing 1: Get operation

5.2 Insert

```
def insert(key, value):
  (mini_page, _leaf_page) = traverse(key)
  ok = mini_page.insert(key, value)
  if ok:
    # early terminate if insert succeeds
    return

  new_size = mini_page.next_size()
  if new_size == 0:
    # need to merge to the base page
    mini_page.merge()
    mini_page.insert(key, value)
    return
  else:
    # need to grow the mini-page
    mini_page.resize(new_size)
    mini_page.insert(key, value)
    return
```

Listing 2: Insert operation

Like the get operation, the insert operation starts with traversing the tree to the mini-page, creating a new mini-page if it does not already exist. The size of the new mini-page is just enough to fit the new record so that we have minimal write amplification. Then, we try to insert the record into the mini-page. If success, i.e., the mini-page has enough space to absorb the record, we will terminate the operation early. No IO (i.e., write to the leaf page) is needed in this case. If the mini-page is full, we will try to resize it to fit its current size. The resize process will allocate a new memory chunk and copy-initialize the mini-page to the new location. Once the mini-page is resized, we insert the record into the new mini-page. If the current mini-page is already large (e.g., 4KB) – the mini-page has already absorbed enough records – we merge it to the leaf page. The merging process will evict all dirty (modified) and cold records from the mini-page.

5.3 Range scan

Range scan involves scanning a range of records in the mini-page and leaf page and then merging the records to get the final result. This means the mini-page will not help reduce the IO for the range scan because it needs to load the leaf page anyway.

To mitigate this, Bf-Tree allows a frequently scanned mini-page to grow to the full page size and cache the entire page in the circular buffer. Thus, the buffer pool can also behave as a page-level cache. Page-level caching fundamentally differs from record-level caching because it caches both individual records and entire range gaps. Caching the entire leaf page provides us a simple way to handle not only efficient range scan but also gap locking [24], negative lookup, etc.

5.4 Delete/Update

Deleting a record is essentially inserting a tombstone record into the mini-page. When a read operation reads the tombstone in the mini-page, it returns a not-found result without touching the leaf page. When the mini-page is merged to the leaf page, the tombstone record is removed from the leaf page, if it exists.

Like Delete, an update operation inserts the record into the mini-page. Future reads can directly read from the mini-page. The record is updated on the leaf page when the mini-page is merged.

5.5 Mini-page operations

Mini-page merge. There are two cases where a mini-page might be merged: (1) the mini-page is too large, and (2) the mini-page is cold. A mini-page grows in size to cache/buffer as many records as possible. But it can not grow arbitrarily large, as records in the mini-page are sorted; a large mini-page incurs high insertion overhead. When a mini-page grows beyond 2KB, it is merged with the leaf page on disk to become a 4KB mini-page that mirrors the leaf page. A mini-page will be evicted (merged to the leaf page) if it is not accessed while in the second-chance region, as discussed in Section 4.1.

To merge a mini-page with its leaf page, we first locate its leaf page using the leaf field in the mini-page header. Then, we calculate the space needed for the leaf page to accommodate the mini-page; if the leaf page does not have enough space, we first split the leaf page and then insert the records of the mini-page into the corresponding leaf pages. Once all records are merged into the leaf, we can discard the mini-page and reuse the memory.

Copy mini-page to tail. When a mini-page is accessed while it is in the second-chance region, it is copied to the tail address. This prevents a frequently accessed mini-page from reaching the head address and, thus, being evicted to disk. We simply allocate a new memory chunk from the circular buffer and copy the mini-page to the new location. The old address is marked as a tombstone, and eviction is not triggered. While copying a mini-page to the tail, we also remove cold records from the mini-page (discussed below).

This means that a record in the mini-page is evicted to disk if it is not accessed while it is in the in-place update region, and the entire mini-page is evicted to disk if it is not accessed while in the second-chance region.

Evicting cold records Mini-pages cache hot records, which can become cold over time. Cold records are evicted to disk when a mini page is copy-on-accessed in the second-chance region. Bf-Tree takes this opportunity to examine all records and only keep hot ones on the new mini page. Hot/cold records are differentiated by their *reference bit* in the meta data; if set, the record is kept, otherwise evicted. The reference bit is set when a record is accessed. When eviction starts, we evict all records whose reference bit is 0 and clear all other reference bits. If a cold record is a cache record (read cache or phantom record), we can directly discard it without writing back to the leaf page. If the record is dirty (insert or tombstone), we merge the entire mini-page to the leaf page (maximizing the utility of the disk write) while retaining the hot records in the mini-page.

Leaf page split. Like a conventional B-Tree, a leaf node splits into two nodes when it is full. For Bf-Tree, an insert always tries to insert to the mini-page first. Growing a mini-page beyond 4KB triggers an eviction, followed by a split operation of the leaf page. When the split happens, each of the remaining records in the mini-page is compared with the split key to determine which leaf page should be inserted.

5.6 Handling negative search

A common problem with existing record-caching systems is that they do not handle negative searches well. A negative search is an operation that tries to find a key-value pair that does not exist in the system. Existing record-caching systems only assert the existence of a key-value pair; if a record is not found in the cache, it either means the record is not cached or means the record does not exist. Those systems then need to look up the leaf page to confirm the record's existence, which is inefficient.

Bf-Tree solves this problem as follows: we cache the negative search result by inserting a phantom record into the mini-page. The observation is that the negative record is like any frequently searched record; we cache them in the mini-page. We note that advanced techniques such as Bloom filters [20] can improve the negativity test, but we leave this as future work as they may add significant design complexity.

Record type	Dirty?	Existence?
Insert	True	True
Cache	False	True
Tombstone	True	False
Phantom	False	False

Table 1: Four types of records in a mini-page.

So far we have discussed four types of records (Table 1) in a mini-page: insert, cache, tombstone, and phantom. Each of the types indicates the dirty and existence property of a record – a dirty record must be written back to the leaf page when the mini-page is merged, and the existence tells whether a record exists in the system.

5.7 Snapshotting, logging, and recovery

Bf-Tree is compatible with standard snapshotting, logging, and recovery mechanisms. Bf-Tree currently implements a simple ARIES-style [51] physiological logging mechanism; more advanced mechanisms are left as future work.

Logging. Before any write operation is committed, it must append a log entry to the WAL (write-ahead-log) and wait (log full or default 1ms interval) until the log is flushed. The log entry points to a page on disk and describes an operation (e.g., insert, delete).

Snapshotting/Checkpointing. Like Aurora [59], Bf-Tree checkpointing is done asynchronously and continuously offline by replaying the WAL entries in parallel. Online snapshotting is also implemented by pausing the write operation and writing back the dirty mini-pages to disk. The mapping table is appended to the last pages of the snapshot file. Inner pages are also written back to disk for faster/simpler recovery. A special mapping table is generated and appended to the snapshot file to map the inner pages' virtual memory addresses to the physical disk offset.

Recovery. Bf-Tree recovery consists of two steps: (1) build the in-memory representation from the snapshot file, (2) replay the WAL to recover to the latest state. We first load the special mapping table mentioned above to rebuild inner pages. Then for every inner page, we use the mapping table to find the physical disk offset and load it into memory. We recursively resolve and load its children's pages. Replaying the WAL consists of simply finding the corresponding page and re-applying the operation.

6 EVALUATION

Bf-Tree re-designs the traditional B-Tree using variable-length buffer pools and mini-pages. In this section, we compare the performance of Bf-Tree with state-of-the-art key-value stores. Specifically, we aim to answer the following questions:

- How does Bf-Tree compare against RocksDB [16], conventional B-Tree, its modern variants [3, 11, 43], and Leanstore [39] on various workloads?
- How does Bf-Tree perform on different workloads, contentions, and cache sizes?
- Where does Bf-Tree spend its time on its different components?

6.1 Experimental setup

We run a YCSB-like benchmark with 200 million initial records, each of which is 32 bytes (16-byte key and 16-byte value). We used a Zip-f distribution with a skew factor of 0.9 (80% of requests access 33% of records); more distributions will be explored in Section 6.6 The default workload consists of 50% of reads and 50% of writes with 2GB memory cache. We warm up the system, repeat the benchmark five times, and report the best throughput.

We implemented Bf-Tree in 13k lines of Rust. We use the latest io_uring feature of the Linux kernel for efficient IO. Specifically, we use the kernel polling mode [2, 26], which creates dedicated kernel polling threads to perform direct IO with zero system calls and bypass the OS page cache. The benchmark is performed on a CloudLab machine 'sm110p', with 32 hyper-threads clocked at 2.4GHz, 128 GB of memory, and 1TB of NVMe PCIe 4.0 SSD with over 600k IOP/s. It runs Ubuntu 22.04 with kernel 5.15 and ext4 as the filesystem. Logging/snapshotting/checkpointing are orthogonal to the core design of Bf-Tree and are disabled for all baselines.

We use lightweight formal methods [5] to validate the correctness of Bf-Tree implementation. Specifically, we employ differential fuzzing to check that Bf-Tree acts semantically the same as our reference model (the B-Tree in Rust's standard library), run Bf-Tree on CloudLab [17] using with libfuzzer [57] with address sanitizer [58] to continuously check for memory issues (e.g., memory leak, use-after-free). We use shuttle [5] to deterministically and systematically explore different thread interleaving to uncover the bugs caused by subtle multithread interactions. We leverage Rust's reference model [33] to statically check that all accesses to mini-pages are safe.

6.2 Baselines

We compare Bf-Tree with (1) RocksDB (commit 54d6286); (2) Bδ-Tree (explained below); (3) conventional B-Tree; (4) Leanstore (commit 677126c); and (5) TwoTree. This section discusses how we set up and configure the baselines. Due to the nature of the complex high-dimensional configuration space, we only highlight the key configurations of each of these systems.

RocksDB. RocksDB is the state-of-the-art LSM-tree-based key-value store. It is widely used in production and is highly optimized for both read and write workloads. RocksDB has three caching components: memtable, block cache, and row cache. The memtable absorbs write operations and is flushed to disk when it is full; the block cache caches the blocks of sstables, while the row cache caches individual records of sstables. It is up to the users to configure how to use the memory in these three caches. We spend half of the memory on writing and half on reading, among which half is on the block cache and half on the row cache. For better performance, we enabled direct IO and disabled fsync and WAL.

Conventional B-Tree. We implemented a conventional B-Tree from scratch in Rust. For fair comparisons, the conventional B-Tree enjoys all optimizations from Bf-Tree except the mini-page design, including hybrid latching, fence keys, prefix compression, look-ahead bytes, and io_uring based IO. In other words, the conventional B-Tree is Bf-Tree but with page-level caching.

Bδ-Tree. To study the effect of delta records – like Bw-Tree [43] – we implemented a B-Tree with delta records, which we call the *Bδ-Tree*. For fair comparisons, this implementation is also in Rust and has all the optimizations in Bf-Tree, except for the mini-page design. The Bδ-Tree is implemented to be as close in design as possible to the Bw-Tree, a state-of-the-art B-Tree used in production. To limit delta chains' memory consumption, we store all delta records and page cache into the same circular buffer similar to that from Bf-Tree, thereby preventing the system from using excessive memory.

Leanstore. Leanstore [39] is a B-Tree optimized for both in-memory and larger-than-memory workloads by reducing the overhead of the mapping table; Bf-Tree instead focuses on improving caching efficiency and reducing write amplification. We include Leanstore mainly to verify that Bf-Tree's in-memory performance is not compromised by its novel buffer pool design.

TwoTree. TwoTree [69] is a new index structure that reuses existing components to build record caching systems. We compare with TwoTree to verify that our record caching (achieved by mini-pages) is as performant as TwoTree.

6.3 Overall performance

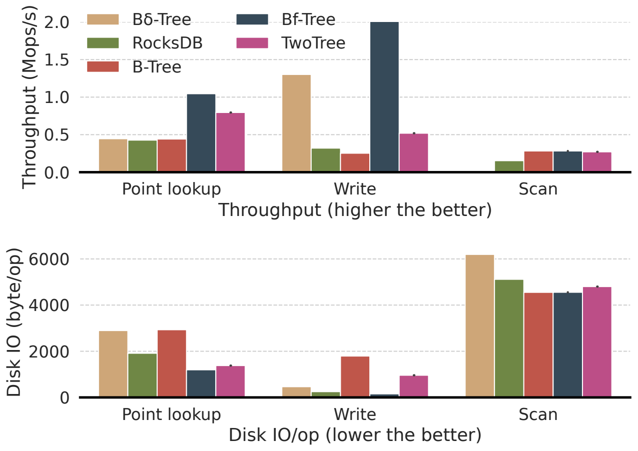

Figure 6: Bf-Tree vs. baselines on various workloads – Bf-Tree is the only system performing well on all three representative workloads.

This experiment examines how the five systems perform on the three most important workloads in practice: point lookup, write, and scan. We use the default benchmark setup mentioned in Section 6.1 and run each workload separately. Figure 6 (top) shows the throughput (the higher the better) of the systems in these workloads with 31 threads, and Figure 6 (bottom) shows the corresponding disk IO per operation (the lower the better).

For write, conventional B-Tree performs the worst, as a single record update would incur a full page write, as evidenced by the highest disk IO per operation. Bδ-Tree, RocksDB, and Bf-Tree all implement write buffers that batch write operations, leading to lower disk IO per operation. Bf-Tree is faster than Bδ-Tree as it allows larger mini-pages, while Bδ-Tree can only have up to 10 delta records, allowing it to absorb more writes before writing to disk. While having higher disk IO/operation, Bδ-Tree performs similarly to RocksDB due to its more efficient IO handling (e.g., zero-copy IO) and more efficient in-memory data structure.

For point lookup, Bf-Tree and TwoTree stand out, as they employ record-level caching in the mini-page, which is more efficient at identifying individual hot records. Bδ-Tree and B-Tree use page-level caching, which leads to similar throughput and disk IO. RocksDB's row cache also helps to improve caching efficiency, as indicated by the fact that its disk IO is lower than Bδ-Tree and B-Tree. However, RocksDB's in-memory skip list is less efficient than B-Tree based implementations, leading to similar throughput.

For range scan, all B-Tree based systems perform similarly, as they all cache the entire leaf page in the memory, preserving record locality and allowing efficient scan. RocksDB performs the worst, as it needs to scan all levels of the sstables and merge the results, leading to higher disk IO per operation.

This experiment provides a high-level view of the system's performance on different workloads. It shows that Bf-Tree is the only system that can perform well in all three representative workloads, making it a good choice for general purpose key-value store.

6.4 Scalability

This section examines the scalability of Bf-Tree and the baselines for larger-than-memory and in-memory workloads.

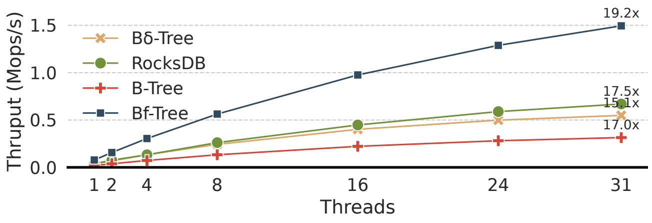

Figure 7: The scalability of Bf-Tree vs baselines. – Bf-Tree scales well before saturating the SSD bandwidth. RocksDB does not saturate the bandwidth at thread 31.

Figure 7 uses the default workload (50% read and 50% update) and varies the number of threads. All systems scale well before reaching the maximum disk bandwidth. Among them, Bf-Tree performs the best, with the lowest write amplification and the most efficient caching. Its efficient io-uring-based IO handling also allows it to scale well without being bottlenecked by the kernel overhead. Bf-Tree, Bδ-Tree, and B-Tree saturated the disk bandwidth at 31 threads, with a scale factor of 19.2×, 15.1×, and 17.0×, respectively. Bδ-Tree is worse than RocksDB (17.5×) because it does not have recording-level caching like the row cache in RocksDB. B-Tree is the worst among the four systems, as its page-granularity caching is inefficient and incurs high write amplification.

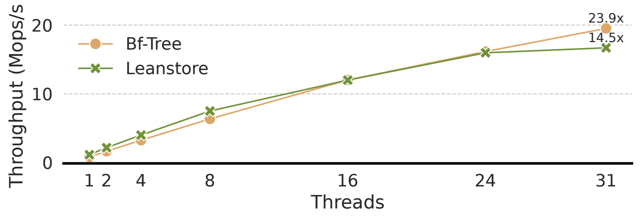

Figure 8: In-memory performance compared with Leanstore. – When data fits in memory, Bf-Tree, B-Tree, Bδ-Tree fold in a similar design. This is a sanity check that Bf-Tree's in-memory performance is as good as the state-of-the-art Leanstore.

Figure 8 uses the same workload as the previous experiment but ensures that the data fits in memory. When data fit in memory, Bf-Tree, B-Tree, and Bδ-Tree fold in the same design. This experiment compares them with Leanstore, a state-of-the-art B-Tree that is highly optimized for in-memory workloads. Leanstore uses pointer swizzling to reduce the overhead of the mapping table; its techniques are complementary to Bf-Tree's, we include Leanstore here to sanity check that Bf-Tree's in-memory performance is not compromised by its mini-page design. At low thread count, Leanstore performs slightly better than Bf-Tree, as it avoids the mapping table overhead in Bf-Tree, specifically, for each operation, Leanstore may have one less cache miss than Bf-Tree. At high thread count, Bf-Tree performs slightly better than Leanstore, reaching a scale factor of 23.9× as opposed to 14.5× of Leanstore, indicating that Bf-Tree is more efficient at handling contention. Note that Bf-Tree's techniques can be applied to Leanstore, and vice versa, and we leave it to future work to apply the Leanstore's techniques to Bf-Tree. Leanstore performs similarly to baseline B-Tree for larger-than-memory workloads, as they both use page-level caching.

6.5 Latency

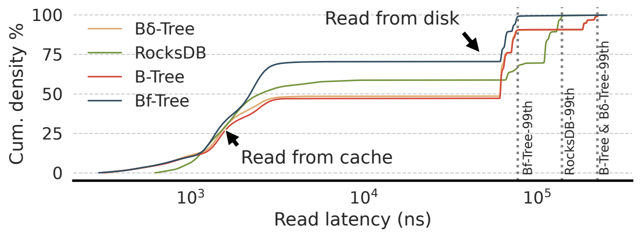

Figure 9: Latency distribution of Bf-Tree vs baselines. – Bf-Tree has the best 50th and 99th percentile latency due to its efficient caching and IO handling.

This experiment examines the latency distribution of Bf-Tree and baselines with read-only workload and single thread execution. The x-axis shows the latency in nanoseconds, and the y-axis shows the cumulative latency distribution, starting from 0% to 100%. We measured the end-to-end latency of the systems, i.e., from issuing the request to the system returning the result. We used dashed vertical lines to show each system's 99th-percentile tail latency.

Figure 9 shows that all systems present a two-stage latency distribution: when data is cached in memory, the latency is around 1us, and when accessing data from disk, the latency increases 100× to around 100μs. Bf-Tree's variable length mini-pages allow it to cache more records in memory, leading to an almost 75% cache ratio. In contrast, all other systems can only cache around 50% of the records, leading to the lowest 50th percentile latency of 1.18μs, while Bδ-Tree, RocksDB, B-Tree has the 50th percentile latency of 1.61μs, 1.86μs, and 58.7μs, respectively. When data is larger than memory, Bf-Tree has the lowest 99th percentile latency (70μs), almost 2× lower than RocksDB (131μs), thanks to its efficient zero-copy zero-syscall IO handling.

The multiple latency spikes in RocksDB are due to the LSM tree's multi-level structure. The last few latency spikes of B-Tree and Bδ-Tree are due to cache replacement, where new pages are being promoted to cache and old pages are being evicted to disk. Bf-Tree also presents a similar stair pattern but a much smaller spike due to its high cache ratio.

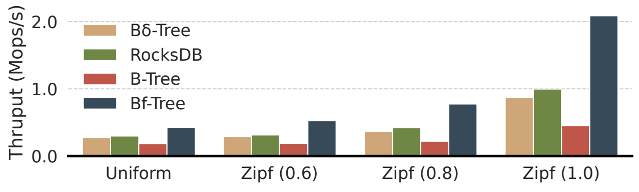

Figure 10: Impact of workload skewness. –Bf-Tree performs consistently the best across all skew levels.

6.6 Skewness

Figure 10 shows the impact of workload skewness on the systems' performance, from Uniform distribution to Zipf distribution with a skew factor of 1.0. This experiment evaluates how well the systems can handle workload skewness and whether they can handle high contention. The performance of all systems increases as the workload becomes more skewed, as the hot records are more likely to be cached in memory. Bf-Tree consistently performs the best, and its performance is the most pronounced when the workload presents a high skew. This is because Bf-Tree's record-level caching is more efficient at identifying hot records than the page-level caching used in B-Tree. RocksDB also has a row cache, but its row cache memory budget is divided among memtable, block cache, and row cache, leading to less efficient caching and Bf-Tree which can adapt the entire memory for record caching. Similarly, Bδ-Tree's delta chains and the page cache compete for the same memory budget, leading to less efficient caching than Bf-Tree.

All systems perform poorly on uniform distribution, as caching is less effective when every record has equal hotness. In this case, Bf-Tree still has the best caching ratio because its mini-page design can better handle the internal fragmentation of the leaf pages. Practical B-Tree leaf pages only have about 70% of the space used due to the nature of page splitting. When caching the entire page in memory, this leads to a 30% waste of memory. Bf-Tree's mini-pages will dynamically grow and shrink to fit the actual number of records in the page, leading to a better caching ratio.

6.7 Read write ratio.

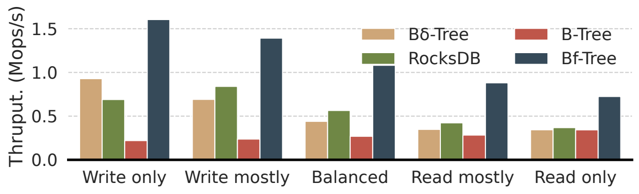

Figure 11: Impact of read and write workloads.

Figure 11 shows the impact of read and write workloads on Bf-Tree and baseline systems. For systems that implement buffered write, e.g., Bf-Tree, Bδ-Tree, and RocksDB, the throughput is higher with more write operations. This is because most write operations are buffered in memory and only flushed to disk when the buffer is full. But for read operations, each cache miss incurs a random disk IO to read the data from the disk. For conventional B-Tree, a higher read ratio leads to better performance, as read operations incur one read IO, while write operations incur both read and write IO. Overall, Bf-Tree performs significantly better than the baselines in all read and write ratios. This is because Bf-Tree's mini-page can cache individual hot records and absorb write operations, leading to the best performance across all read and write ratios.

6.8 Cache sensitivity

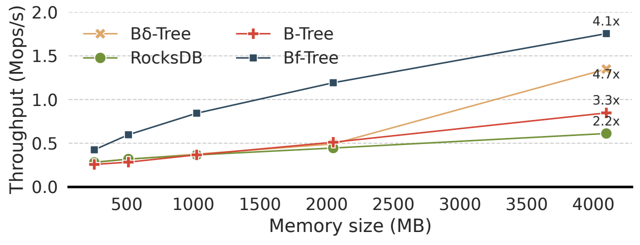

Figure 12: Impact of cache size.

This experiment examines the cache sensitivity of Bf-Tree and baseline systems. We used a read-only workload and varied the memory size from 256MB to 4GB, and reported how their throughput changed in response to the cache size. As memory size increases by 16×, RocksDB's performance steadily increases but only to 2.2×, indicating a low caching efficiency. Bδ-Tree and B-Tree perform similarly as they both use page-level caching for the leaf pages, with an increase of 4.7× and 3.3×, respectively. In particular, both Bδ-Tree and B-Tree have a slow performance increase when most data are on disk and a fast performance increase when most data are in memory, consistent with previous study [27]. Bf-Tree has a similar performance increase of 4.1× and is consistently the best across all memory sizes. The gap between Bf-Tree and the baseline systems is wider when most data are on disk, showcasing Bf-Tree's efficient caching mechanism. The gap becomes narrower when most data are in memory, as all B-Tree based systems fold to the same design when data fit in memory.

6.9 Time-component analysis

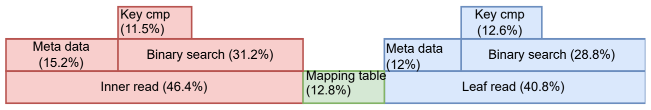

Figure 13: Time spent on different components of the system – Recreated from the flamegraph [25] for better clarity.

Figure 13 shows the flamegraph [25] of the Bf-Tree for a read-only in-memory workload. Larger-than-memory workloads present similar results, except that >95% of the time is spent on disk IO.

The inner node search takes roughly half the time, involving multiple inner node searches, one mapping table read, and one leaf node search. Searching an inner node first loads the metadata for key-value pairs, then binary searches the keys to find the next level pointer, and finally loads the next node. Each mapping table access first locates the slot from page ID, then loads and acquires a reader lock on the mapping table and uses the physical address to load the leaf page. Leaf page search is similar to inner-node search.

Multiple inner node searches spend similar time to one leaf node search. This is because inner nodes are more frequently accessed, most of which can fit in the CPU cache, while leaf node access is less frequent and almost always needs to load from memory. The mapping table uses only 12.8% of time, requiring only one memory access for address translation and lock acquisition.

Within inner and leaf node searches, they spend similar time on metadata load and binary search, as they have the same page layout and optimizations. Among binary searches, roughly one-third of the time is spent comparing the key bytes, and the rest is spent loading data from memory.

6.10 Copy-on-access region size

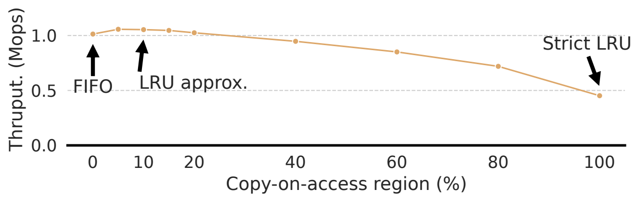

Figure 14: Impact of copy-on-access region size (% of total circular buffer size).

Figure 14 varies the copy-on-access region size from 0% to 100% of total circular buffer size, the y-axis shows the corresponding throughput. At 0%, the buffer acts like a FIFO queue, demonstrating low runtime overhead but low caching quality. At 100%, the buffer acts like a strict LRU cache, demonstrating high runtime overhead but high caching quality. Bf-Tree by default chooses a 10% copy-on-access region size, which balances the trade-off between runtime overhead and caching quality.

6.11 Impact on promotion rate

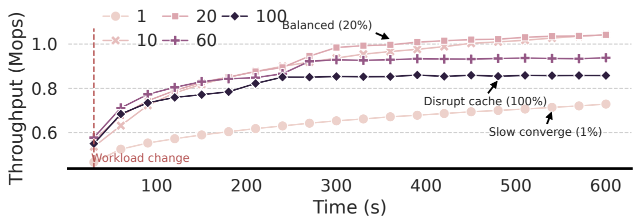

Figure 15: Impact of promotion rate: the probability of promoting a record from a disk page to its mini page.

Figure 15 varies the promotion rate (the probability of promoting a cold record to cache) from 1% to 100%. The x-axis shows the time since a workload change, and the y-axis shows the throughput. When the promotion rate is low (e.g., 1%), the system favors *caching frequency* and responds slowly to workload change but might converge to a higher throughput. When it is high (e.g., 100%), the system favors *caching recency* and converges quickly but ends up with a lower throughput as it disrupts the cache by promoting cold records. Bf-Tree by default chooses a 20% promotion rate, as it balances the trade-off between recency and frequency.

7 RELATED WORK

The design of Bf-Tree is inspired by many systems, including in-memory indexes, larger-than-memory key-value stores, B-Tree for new hardware, and LSM-Trees.

In-memory indexes ART (adaptive radix tree) [40] is an in-memory range index optimized for point lookup and memory efficiency. Bf-Tree's prefix compression is inspired by ART's prefix compression. Blink-hash [7] is an in-memory B-Tree optimized for append heavy workloads, Bf-Tree's mini-page is inspired by its write buffer design. BP-tree [44] is an in-memory B-Tree optimized for both point lookup and range scan using large leaf pages. HydraList [50] is a scalable in-memory index that separates data search and modification, Bf-Tree's optimistic latch coupling is inspired by its design. Pea hash [46] is a performant hash table that balances the trade-off between memory utilization and access latency. Bf-Tree is inspired by the above systems to improve its in-memory performance.

Larger-than-memory KV stores FASTER [8, 9, 45] is a state-of-the-art larger-than-memory hash key-value store optimized for point lookups and write-heavy workloads. Bf-Tree's circular buffer design is inspired by FASTER's hybrid log, which supports variable-length record allocations. Unlike FASTER, Bf-Tree is a range index that allows efficient range scans. TreeLine [65] is a B-Tree implementation incorporating record caching and insert forecasting. Unlike Bf-Tree, it currently only supports fixed-size records. Leanstore [39] is a B-Tree optimized for both in-memory and larger-than-memory workloads; it is the first system to use pointer swizzling to reduce the overhead of the mapping table. Two-tree [69] is a new index structure to support efficient record caching using existing B-Trees. SplinterDB [11] is a Bϵ-tree [4] implementation to focus on low read/write amplification. These systems help Bf-Tree to design efficient caching and IO handling while balancing trade-offs among point lookup, write, and range scan workloads.

B-Tree for new hardware B-Trees for new hardware, e.g., NVMe SSDs, PM, etc., shed light on how to design a Bf-Tree that works well for future hardware. PIM-tree [34] is a range index designed for PIM (process-in-memory) that balances inter-node communication and load balance. [54] leverages transparent compression in modern SSDs to design efficient B-Tree that achieves low write amplification. Bztree [1], Plin [67], NBTree [66], APEX [47], Halo [30], Hamming-tree [35] are range indexes designed for byte-addressable persistent memory. [28, 42] comprehensively evaluates persistent memory range indexes. Sherman [60] and [27] propose systems for disaggregated/tiered memory systems.

LSM-trees LSM-Trees provide insights on how to reduce write amplification of Bf-Tree. WALTZ [37] is an LSM-tree that leverages ZNS (zoned namespace SSDs) to reduce tail latency. Chucky [13] propose a new filter for LSM-tree that replaces the bloom filter for better access cost. WiscKey [48] is an LSM-tree structure that separates the key and value to reduce write amplification.

8 CONCLUSION

This paper presents Bf-Tree, a novel B-Tree variant optimized for point lookup, write, and range scan workloads. It uses a novel minipage design to achieve low write amplification and efficient caching. To manage the mini-pages, Bf-Tree designed a novel variable length buffer pool using a circular buffer. We implemented Bf-Tree in Rust with strong correctness guarantees. Our evaluations show that Bf-Tree is 2.5× faster than RocksDB (LSM-Tree) for scan operations, 6× faster than a B-Tree for write operations, and 2× faster than both B-Trees and LSM-Trees for point lookups.

ACKNOWLEDGMENTS

We would like to thank Phil Bernstein, Ted Hart, Vasileios Zois, Ao Li, Xinjing Zhou, Chenhao Ye, Shawn Zhong, and the anonymous reviewers for their comments and suggestions that helped improve this paper. We also thank the UW-Madison TA-ship, friends, mentors, and advisors for their support during the timeframe of this research work.

REFERENCES

[1] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. Bztree: A high-performance latch-free range index for non-volatile memory. *Proceedings of the VLDB Endowment* 11, 5 (2018), 553–565.

[2] Jens Axboe. 2019. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf.

[3] Michael A Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C Kuszmaul, Donald E Porter, Jun Yuan, and Yang Zhan. 2015. An introduction to B-trees and write-optimization. *login; magazine* 40, 5 (2015).

[4] Michael A Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C Kuszmaul, Donald E Porter, Jun Yuan, and Yang Zhan. 2015. An introduction to B-trees and write-optimization. *login; magazine* 40, 5 (2015).

[5] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, et al. 2021. Using lightweight formal methods to validate a key-value storage node in Amazon S3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 836–850.

[6] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, modeling, and benchmarking {RocksDB}{Key-Value} workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 209–223.

[7] Hokeun Cha, Xiangpeng Hao, Tianzheng Wang, Huanchen Zhang, Aditya Akella, and Xiangyao Yu. 2023. Blink-hash: An Adaptive Hybrid Index for In-Memory Time-Series Databases. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1235–1248.

[8] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*. 275–290.

[9] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: an embedded concurrent key-value store for state management. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1930–1933.

[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.

[11] 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.

[12] Andrew Crotty, Viktor Leis, and Andrew Pavlo. 2022. Are You Sure You Want to Use MMAP in Your Database Management System?. In *CIDR 2022, Conference on Innovative Data Systems Research*.

[13] Niv Dayan and Moshe Twitto. 2021. Chucky: A succinct cuckoo filter for lsm-tree. In *Proceedings of the 2021 International Conference on Management of Data*. 365–378.

[14] Biplob Debnath, Sudipta Sengupta, Jin Li, David J Lilja, and David HC Du. 2011. BloomFlash: Bloom filter on flash-based storage. In *2011 31st International Conference on Distributed Computing Systems*. IEEE, 635–644.

[15] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. 2013. Anti-caching: A new approach to database management system architecture. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1942–1953.

[16] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Transactions on Storage (TOS)* 17, 4 (2021), 1–32.

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

[18] Ahmed Eldawy, Justin Levandoski, and Per-Åke Larson. 2014. Trekking through siberia: Managing cold data in a memory-optimized database. *Proceedings of the VLDB Endowment* 7, 11 (2014), 931–942.

[19] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. of the bsdcan conference, ottawa, canada*.

[20] Facebook. 2021. RocksDB Bloom Filter. https://github.com/facebook/rocksdb/wiki/RocksDB-Bloom-Filter. Accessed: 2024-02-01.

[21] Facebook. 2023. Block Cache. https://github.com/facebook/rocksdb/wiki/Block-Cache. Accessed: 2024-02-01.

[22] Kevin P Gaffney, Martin Prammer, Larry Brasfield, D Richard Hipp, Dan Kennedy, and Jignesh M Patel. 2022. Sqlite: past, present, and future. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3535–3547.

[23] Google. 2021. LevelDB - A fast key-value storage library written at Google. https://github.com/google/leveldb. GitHub repository.

[24] Goetz Graefe et al. 2011. Modern B-tree techniques. *Foundations and Trends® in Databases* 3, 4 (2011), 203–402.

[25] Brendan Gregg. 2016. The flame graph. *Commun. ACM* 59, 6 (2016), 48–57.

[26] Gabriel Haas and Viktor Leis. 2023. What Modern NVMe Storage Can Do, and How to Exploit It: High-Performance I/O for High-Performance Storage Engines. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2090–2102.

[27] Xiangpeng Hao, Xinjing Zhou, Xiangyao Yu, and Michael Stonebraker. 2024. Towards Buffer Management with Tiered Main Memory. *Proc. ACM Manag. Data* 2, 1 (Feb. 2024), Article 31. https://doi.org/10.1145/3639286 SIGMOD.

[28] Yuliang He, Duo Lu, Kaisong Huang, and Tianzheng Wang. 2022. Evaluating Persistent Memory Range Indexes: Part Two [Extended Version]. *arXiv preprint arXiv:2201.13047* (2022).

[29] Gavin Henry. 2019. Howard chu on lightning memory-mapped database. *Ieee Software* 36, 06 (2019), 83–87.

[30] Daokun Hu, Zhiwen Chen, Wenkui Che, Jianhua Sun, and Hao Chen. 2022. Halo: A hybrid PMem-DRAM persistent hash index with fast recovery. In *Proceedings of the 2022 International Conference on Management of Data*. 1049–1063.

[31] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[32] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. 2009. Shore-MT: a scalable storage manager for the multicore era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. 24–35.

[33] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–34.

[34] Hongbo Kang, Yiwei Zhao, Guy E Blelloch, Laxman Dhulipala, Yan Gu, Charles McGuffey, and Phillip B Gibbons. 2023. PIM-tree: A Skew-resistant Index for Processing-in-Memory. In *Proceedings of the 2023 ACM Workshop on Highlights of Parallel Computing*. 13–14.

[35] Saeed Kargar and Faisal Nawab. 2023. Hamming Tree: The Case for Energy-Aware Indexing for NVMs. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

[36] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data*. 1675–1687.

[37] Jongsung Lee, Donguk Kim, and Jae W Lee. 2023. WALTZ: Leveraging Zone Append to Tighten the Tail Latency of LSM Tree on ZNS SSD. *Proceedings of the VLDB Endowment* 16, 11 (2023), 2884–2896.

[38] Daan Leijen, Benjamin Zorn, and Leonardo de Moura. 2019. Mimalloc: Free list sharding in action. In *Programming Languages and Systems: 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1–4, 2019, Proceedings 17*. Springer, 244–265.

[39] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-memory data management beyond main memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 185–196.

[40] 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.

[41] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*. 1–8.

[42] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating persistent memory range indexes. *Proceedings of the VLDB Endowment* 13, 4 (2019), 574–587.

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

[44] Amanda Li. 2023. *BP-Tree: Overcoming the Point-Range Operation Tradeoff for In-Memory B-trees*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[45] Tianyu Li, Badrish Chandramouli, and Samuel Madden. 2022. Performant almost-latch-free data structures using epoch protection. In *Proceedings of the 18th International Workshop on Data Management on New Hardware*. 1–10.

[46] Zhuoxuan Liu and Shimin Chen. 2023. Pea Hash: A Performant Extendible Adaptive Hashing Index. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–25.

[47] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. 2021. APEX: A high-performance learned index on persistent memory. *arXiv preprint arXiv:2105.00683* (2021).

[48] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)* 13, 1 (2017), 1–28.

[49] Chen Luo and Michael J Carey. 2020. Breaking down memory walls: Adaptive memory management in LSM-based storage systems. *Proceedings of the VLDB Endowment* 14, 3 (2020), 241–254.

[50] Ajit Mathew and Changwoo Min. 2020. HydraList: A scalable in-memory index using asynchronous updates and partial replication. *Proceedings of the VLDB Endowment* 13, 9 (2020), 1332–1345.

[51] Chandrasekaran Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)* 17, 1 (1992), 94–162.

[52] Bruce Momjian. 2001. *PostgreSQL: introduction and concepts*. Vol. 192. Addison-Wesley New York.

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

[54] Yifan Qiao, Xubin Chen, Ning Zheng, Jiangpeng Li, Yang Liu, and Tong Zhang. 2022. Closing the B+-tree vs. LSM-tree Write Amplification Gap on Modern Storage Hardware with Built-in Transparent Compression. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. USENIX Association, Santa Clara, CA, 69–82. https://www.usenix.org/conference/fast22/presentation/qiao

[55] RocksDB Team. 2021. *RocksDB Secondary Cache*. https://rocksdb.org/blog/2021/05/27/rocksdb-secondary-cache.html

[56] Samsung Semiconductor. 2024. PM9A3 (MZQL2960HCJR-00A07) - Datacenter SSD. https://semiconductor.samsung.com/us/ssd/datacenter-ssd/pm9a3/mzql2960hcjr-00a07/ Accessed: 2024-05-27.

[57] Kosta Serebryany. 2016. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 157–157.

[58] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*. 309–318.

[59] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1041–1052.

[60] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A write-optimized distributed b+ tree index on disaggregated memory. In *Proceedings of the 2022 International Conference on Management of Data*. 1033–1048.

[61] 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.

[62] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A Chien, and Haryadi S Gunawi. 2017. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs. *ACM Transactions on Storage (TOS)* 13, 3 (2017), 1–26.

[63] Pan Yang, Ni Xue, Yuqi Zhang, Yangxu Zhou, Li Sun, Wenwen Chen, Zhonggang Chen, Wei Xia, Junke Li, and Kihyoun Kwon. 2019. Reducing garbage collection overhead in {SSD} based on workload prediction. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*.

[64] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. 2017. SPDK: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 154–161.

[65] Geoffrey X Yu, Markos Markakis, Andreas Kipf, Per-Åke Larson, Umar Farooq Minhas, and Tim Kraska. 2022. Treeline: an update-in-place key-value store for modern storage. *Proceedings of the VLDB Endowment* 16, 1 (2022), 99–112.

[66] Bowen Zhang, Shengan Zheng, Zhenlin Qi, and Linpeng Huang. 2022. NBTree: a Lock-free PM-friendly Persistent B+-Tree for eADR-enabled PM Systems. *Proceedings of the VLDB Endowment* 15, 6 (2022), 1187–1200.

[67] Zhou Zhang, Zhaole Chu, Peiquan Jin, Yongping Luo, Xike Xie, Shouhong Wan, Yun Luo, Xufei Wu, Peng Zou, Chunyang Zheng, et al. 2022. Plin: a persistent learned index for non-volatile memory with high performance and instant recovery. *Proceedings of the VLDB Endowment* 16, 2 (2022), 243–255.

[68] Zigang Zhang, Yinliang Yue, Bingsheng He, Jin Xiong, Mingyu Chen, Lixin Zhang, and Ninghui Sun. 2014. Pipelined compaction for the LSM-tree. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 777–786.

[69] Xinjing Zhou, Xiangyao Yu, Goetz Graefe, and Michael Stonebraker. 2023. Two is Better Than One: The Case for 2-Tree for Skewed Data Sets. *memory* 11 (2023), 13.