DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Optimizing Write-Heavy Database Operations Using B^{ε}-Trees

Christoph Rotte

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Optimizing Write-Heavy Database Operations Using B$^{\varepsilon}$-Trees

Optimierung von schreibintensiven Datenbankoperationen mit B$^{\varepsilon}$-Bäumen

Author:	Christoph Rotte
Supervisor:	Prof. Dr. Thomas Neumann
Advisor:	Philipp Fent, M.Sc.
Submission Date:	15.08.2022

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.08.2022 Christoph Rotte

Abstract

Today, log-structured merge trees (LSM-Trees) are de facto the default index structure for database storage engines that aim to improve performance when processing write-heavy workloads. While they are widely used in various key-value stores, their structure is not asymptotically optimal regarding minimizing costly IO operations.

This thesis instead focuses on the B^{ε}-Tree, an adaption of the typical B^{+}-Tree, which asymptotically offers the write performance of an LSM-Tree combined with the read performance of a B/B^{+}-Tree. We introduce a page-buffered B^{ε}-Tree design with the primary objective to increase scalability with multiple threads. We then evaluate our B^{ε}-Tree against a reference B^{+}-Tree implementation as well as LevelDB and RocksDB, two key-value stores built on top of an LSM-Tree. Our evaluation shows that our design can outperform all three comparative systems in some instances. Therefore, we believe that, with additional modifications and optimizations, B^{ε}-Trees can be used to optimize write operations while still performing reads with similar performance as typical database systems.

Contents

1 Introduction

In the past, data management systems focused on storing data on hard disk drives (HDD) and caching frequently accessed index structures in main memory. HDDs were around 50 times less expensive than main memory per byte. Thus, generally, only a small part of the data was cached. However, with the falling costs of main memories in the last decades, the amount of cached data gradually increased. This resulted in main memory systems which waive IO operations at runtime and thus enable orders of magnitude higher throughput. [28]

In recent years, however, solid-state drives (SSD) have become a viable solution for data storage, although modern SSDs are still at least ten times slower than DRAM. Increasing the main memory might not always be economically feasible with growing data sizes as SSDs are generally around ten times less expensive per byte [26]. Such situations again require data management systems that efficiently keep data in memory and on disk while minimizing costly IO operations.

The change from main memory systems to SSD-oriented database engines resulted in new implementations like Google's LevelDB [12], Facebook's RocksDB [11] or Apache's HBase[1]. They do not use traditional index structures, like B-Trees built on top of buffer managers. Instead, these key-value stores often include different indexes optimized for rather write-heavy workloads [41].

In the scope of this thesis, we examine an example of such an index structure, the B^{ε}-Tree. Although it has been used relatively rarely for storage engines until now, its structure promises write performance comparable to the commonly used write-optimized structures. At the same time, it appears to offer read performance close to that of a typical B-Tree. [2]

After describing the structures in Chapter 2 and Chapter 3 first, Chapter 4 analyzes their behavior by comparing the respective asymptotics. In Chapter 5, we then introduce a design for a B^{ε}-Tree which we will use to evaluate the performance practically. The main objective of our design is to enable the throughput to scale with multiple threads. Afterward, Chapter 6 contains a detailed three-part benchmark comparison between all discussed structures. While Chapter 7 and 8 include related work and ideas for future continuation of the presented design, Chapter 9 concludes the outcome of this thesis.

[1]https://hbase.apache.org/

2 Background

Typical database engines are optimized for both read and write operations, i.e., point and range lookups as well as point and bulk inserts and updates. This is because general-purpose databases often handle mixed workloads and balance their performance trade-offs between reads and writes.

Apart from databases, however, storage engines are also used for read-heavy services like search indexes or write-heavy services such as stream processing, logging, and SSD caching [9]. These areas have to deal with unevenly distributed workloads, making the use of engines tuned for mixed workloads suboptimal. Especially write-heavy workloads usually suffer from this asymmetry in standard database systems [21].

2.1 Traditional Database Structures

Usually, their design forces general-purpose database engines to use in-place writes that often result in multiple random I/O operations [15]. Sequential disk reads and writes are generally orders of magnitude faster than random operations [23]. Nonetheless, depending on the workload, a B^+-Tree, for example, has to successively access nodes that often do not reside next to each other on disk. This leads to many random disk reads and writes.

Additionally, every update operation potentially includes reading the current value from disk and storing the updated value again, which results in two I/O operations. Although we can circumvent this by storing the index entirely in memory, cases with large data sets and limited memory consequently still scale poorly in general [26]. Furthermore, if we have a lot of small values, the index may outgrow the actual working set in size, increasing the needed memory even more. Alternative approaches, like hash-based partitioning, have the same problems [17].

While write-focused structures like append-only logs can profit from sequential writes instead of random accesses, key-based reads have to scan the entire log in the worst case. This makes them viable for linear read patterns but impracticable for mixed workloads with random point lookups [25, 38].

2.2 LSM-Trees

A popular data structure choice for write-focused storage engines is the log-structured merge tree (LSM-Tree) [32]. Compared to general-purpose structures like B/B$^+$-Trees, LSM-Trees are designed to handle write operations more efficiently by taking advantage of fast sequential disk accesses.

In memory, LSM-Trees store write operations as encoded messages in a sorted buffer called a "memtable" [32]. Consequentially, the average write operation does not require any I/O operation. When the memtable is full, we store it on disk in an immutable index file. While the original design [32] references only two levels, C_0 (the memtable in memory) and C_1 (the index file), modern implementations typically store the messages in multiple "runs" on disk [19]. Index files with similar sizes are periodically merged into larger sorted files, resulting in multiple tree-like levels with exponentially increasing sizes (see figure 2.1).
Because writes are buffered in memory and saving a memtable is done sequentially on disk, LSM-Trees generally increase write performance, especially compared to other general-purpose structures when dealing with limited memory [32, 19].

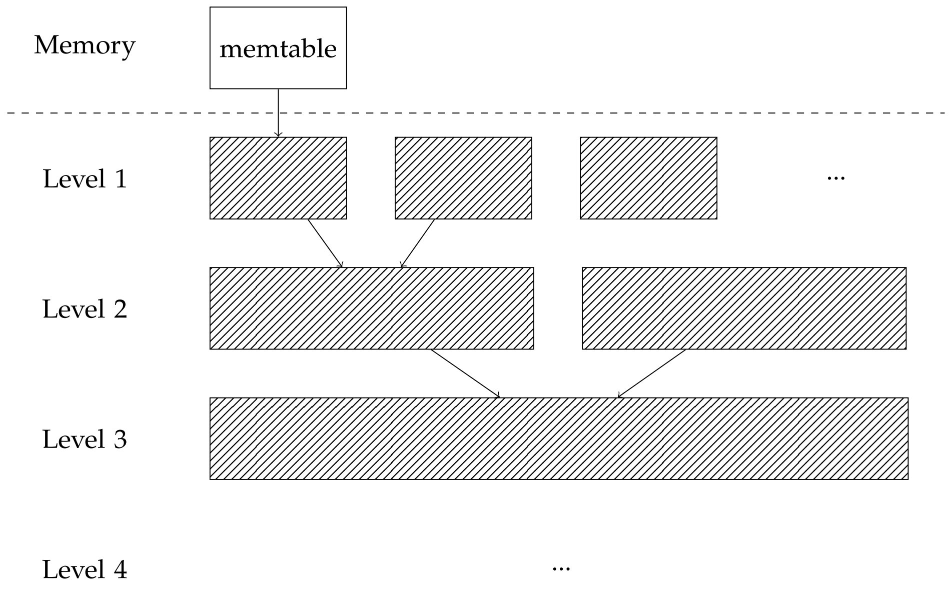


Figure 2.1: LSM-Tree Merges

Read operations, however, potentially have to search through multiple files, which again requires randomly accessing the disk multiple times. Thus, reads in LSM-Trees

are generally slower than in B$^+$-Trees and similar structures [19]. In practice, most data stores based on LSM-Trees like Facebook's RocksDB [11] or Google's LevelDB [12] try to work against this by introducing fence pointers, block caches, and bloom filters, among other techniques. [7, 9]. Fence pointers contain references to each run's lower and upper bounds, thereby decreasing the number of required I/O operations. A block cache stores frequently accessed file parts that do not have to be read from disk every time.
Bloom filters, on the other hand, use additional memory to save overlapping small hashes of the written keys [3]. If a key is not found in the filter, a reader can abort its operation, knowing that the key has not been used before. False positives due to hash collisions may occur. Nonetheless, by using only a fraction of the key size for each respective filter, bloom filters can often prevent unnecessary I/O reads for non-existing keys [3].

Although techniques like the ones mentioned above can improve the general read performance of LSM-Trees, they come with a need for additional memory, which in turn again limits the size available for the memtable. Additionally, integrating LSM-Trees into databases with preexisting general purpose structures like B$^+$-Trees requires an additional backend since their base design is fundamentally incompatible with typical I/O management systems like page buffers.

3 B$^\varepsilon$-Trees

3.1 General Structure

B$^+$-Trees usually consist of two types of nodes: inner nodes and leaves. The inner nodes contain pivot keys and pointers to the corresponding child nodes, while the leaves store the key-value pairs (or corresponding pointers). Subsequent updates and deletes are then directly performed on these pairs. Consequently, each thread must reach its respective target leaf node once for every insert, update and delete operation. A B$^\varepsilon$-Tree instead encodes every write operation as an "upsert" message. Each upsert message U consists of the operation type O_U, a key K_U and its associated value V_U (for inserts and updates). Furthermore, it contains a timestamp T_U that marks the time the operation was initially requested. Therefore, operations are treated as data and do not have to be performed immediately. [2]

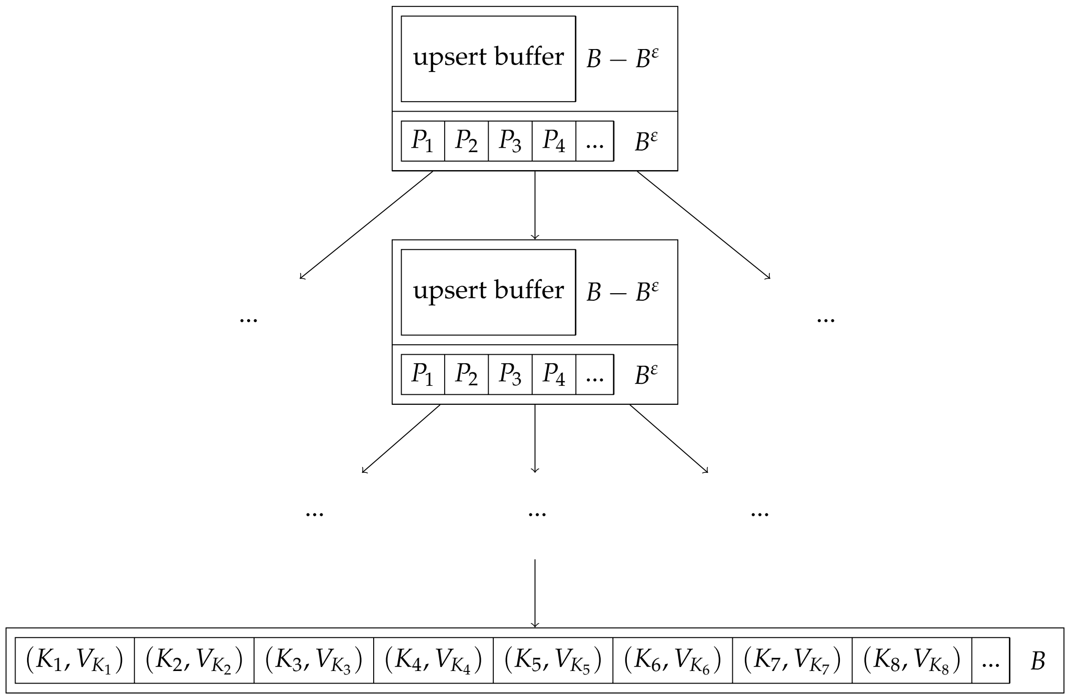
In contrast to a B$^+$-Tree, B$^\varepsilon$-Trees split the space of the inner nodes and additionally store a buffer for the upsert messages next to the pivot keys and their child pointers (see figure 3.1). Write operations are temporarily stored in these buffers and then flushed to the buffers of inner nodes in lower levels or the leaf nodes. The leaf nodes are structured similarly to the leaves of a B$^+$-Tree and merely contain tuples representing each key's last flushed state and its associated value.

Each inner node of B bytes uses B^ε bytes for the pivot elements and pointers and $B^\varepsilon - B$ bytes for the allocation of its upsert buffer ($\varepsilon \in [0;1]$). The value of ε thus describes the trade-off between a buffered binary tree (also called a "buffered repository tree" [4]) at $\varepsilon \approx 0$ and buffering no operations at all, similarly to a B$^+$-Tree, at $\varepsilon \approx 1$.

3.2 Write Operations

To insert, update or delete a value, we first encode the operation as a new upsert and pass it to the root node. If the tree consists of one level, the root node is a leaf node, and the operation represented by the upsert will be executed. Otherwise, we do not differentiate between the different message types and append the upsert to the buffer and return.

When the root buffer reaches its total capacity, we scan the buffer and select the child



upsert buffer
$B - B^\varepsilon$
P_1 P_2 P_3 P_4 ...
B^ε
(K_1, V_{K_1}) (K_2, V_{K_2}) (K_3, V_{K_3}) (K_4, V_{K_4}) (K_5, V_{K_5}) (K_6, V_{K_6}) (K_7, V_{K_7}) (K_8, V_{K_8}) ...
B

Figure 3.1: B$^\varepsilon$-Tree Structure

node with the most upserts addressed to it. We then flush these upserts collectively to the respective node. This happens recursively if the buffers in the lower levels also do not have enough space, sometimes flushing to more than one child node. Again, after an upsert reaches its target leaf, the encoded operation will be executed. Delete operations are encoded as upsert messages called "tombstone messages" [2], marking a key-value pair as removed. Nevertheless, because the upsert does not reach the leaf level immediately, the tombstone message and the underlying pair may temporarily co-exist.
Consequently, B$^\varepsilon$-Trees accumulate multiple write operations and split the cost of traveling down amongst them. This also means that write operations do not return an immediate response, regardless of the success of their execution. For example, without a subsequent lookup or a separate callback, a worker thread can not identify whether its update request will find an existing value or not.

Note that, because of the buffered operations, a B$^\varepsilon$-Tree does not contain the current state of every value on the leaf level. Thus, when a B$^\varepsilon$-Tree is used as a secondary index structure, we have to either save its inner nodes and the underlying pairs or flush down all buffered upserts.

3.3 Lookups

In contrast to a B$^+$-Tree, the B$^\varepsilon$-Tree contains information about the change of values in its inner nodes in the form of upserts. Therefore, to get the current state of a value V_K associated with a key K, a worker thread has to scan each buffer on its way down for upserts addressed to K. It then has to accumulate these upserts until it finds a base state, either an insert or a tombstone message, or the value on the leaf level. After that, it can reconstruct the current state of V_K by applying the upserts according to the order of their timestamps on the base state. Consequently, lookup operations do not modify the buffer content in any way - instead of flushing down the upserts addressed to K, they create their own copy in place. Figure 3.2 illustrates the procedure of a point lookup in a B$^\varepsilon$-Tree.

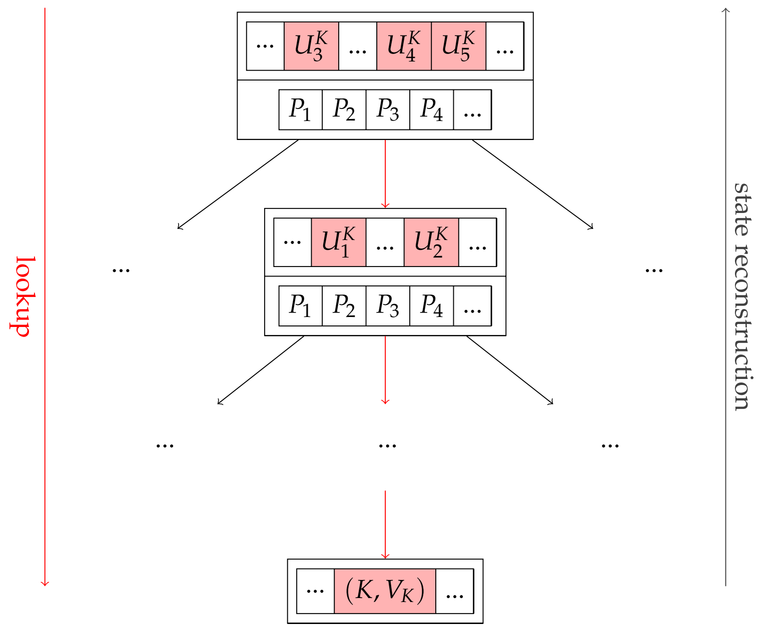

lookup
state reconstruction
... U_3^K ... U_4^K U_5^K ...
P_1 P_2 P_3 P_4 ...
... U_1^K ... U_2^K ...
P_1 P_2 P_3 P_4 ...
... (K, V_K) ...

Figure 3.2: A B$^\varepsilon$-Tree point lookup. The current value V_K associated with key K is reconstructed by applying all corresponding upserts in reverse order to the value stored in the respective leaf node:
$V_K^{current} = V_K \leftarrow U_1^K \leftarrow U_2^K \leftarrow U_3^K \leftarrow U_4^K \leftarrow U_5^K$

Range queries for a key range $K_1, K_2, ..., K_N$ behave similarly as each value depends on potential buffered upserts. Thus, to get every final value $V_{K_1}, V_{K_2}, ..., V_{K_N}$, we first have to scan every upsert buffer in the subtree of $K_1, K_2, ..., K_N$.

4 Theoretical Performance

4.1 The Parameter ε

As described in section 3.1, the value of ε determines the trade-off related to the node size B between the number of pivots/child pointers and the buffer size of each node. Since we use B^ε for the first and $B - B^\varepsilon$ for the second, the fanout grows exponentially with the value of ε. Smaller values increase the buffer size, thus increasing write throughput, while larger values increase the fanout, therefore decreasing the tree height and increasing read throughput. The original paper proposes $\varepsilon = \frac{1}{2}$ as a good value [2]. This doubles the tree height and thus asymptotically halves the read performance compared to a B-Tree since we only have $\sqrt{B}$ space left for the fanout.

Writes, however, can theoretically be performed orders of magnitude faster than the writes in a B-Tree, as shown in the following section. As a result, regarding update operations, B$^\varepsilon$-Trees should prefer "blind" updates over read-modify-write operations as these are bound by the performance of the read operation [2].

4.2 Asymptotic Comparison

Table 4.1 compares the I/O operations needed for inserts and lookups of a B/B$^+$-Tree, a B$^\varepsilon$-Tree as well as an LSM-Tree. B denotes the node size, N the number of stored entries, and K the size of the key range. We focus on I/O operations since these are usually the primary source of performance bottlenecks, as described in section 2.1. The comparison assumes that all key-value pairs are of the same size and that a maximum of one I/O operation (random seek) is needed to access a node.
Since we focus on working sets that exceed the memory size, the comparison makes use of the "disk access model" (DAM) [1, 8], also called the "I/O model", "external memory model" or "cache-aware model". The DAM considers the fact that parts of the data structures are cached in memory as well as the performance differences between random and sequential I/O operations.

The value of ε determines the fanout of the B$^\varepsilon$-Tree and thus impacts its height. However, when set to a fixed value, it disappears in the asymptotic analysis, leaving the B$^\varepsilon$-Tree with the same lookup complexity as the B-Tree of $O(\log_B N)$ for point and

Table 4.1: Asymptotic Comparison of the Amortized Number of I/O Operations

	B-Tree / B$^+$-Tree	B$^\varepsilon$-Tree	LSM-Tree
Upsert	$O(\log_B N)$	$O(\frac{\log_B N}{\varepsilon B^{1-\varepsilon}})$	$O(\frac{\log_B N}{\varepsilon B^{1-\varepsilon}})$
Point Lookup	$O(\log_B N)$	$O(\frac{\log_B N}{\varepsilon})$	$O(\frac{\log_B^2 N}{\varepsilon})$
Range Lookup	$O(\log_B N + \frac{K}{B})$	$O(\frac{\log_B N}{\varepsilon} + \frac{K}{B})$	$O(\frac{\log_B^2 N}{\varepsilon} + \frac{K}{B})$

$O(\log_B N + \frac{K}{B})$ for range lookups.
In the worst case, if all respective buffers are full, we must perform a flush on every level, thus traveling down to the respective leaf node. This creates an upper bound for write operations of $O(\frac{\log_B N}{\varepsilon})$.
Since write operations in a B$^\varepsilon$-Tree are bundled and flushed together, a B$^\varepsilon$-Tree divides the cost of one write by the size of the upsert buffer. This heavily reduces the net costs for the upsert operation. A value of $\varepsilon = \frac{1}{2}$, for example, divides the cost by $\sqrt{B}$, as described above. Consequently, the node size is directly proportional to the performance gain of writes when compared to a B-Tree.

For the LSM-Tree, we use B^ε for the factor by which the runs on disk exponentially increase in size to simplify the comparison. This factor behaves similarly to the branch-buffer trade-off of the B$^\varepsilon$-Tree, thus its use as syntactic sugar. Write operations are asymptotically equal when compared to a B$^\varepsilon$-Tree, dividing the costs by $O(\log_B N + \frac{K}{B})$. Lookups, on the other hand, are asymptotically slower than those of a B$^\varepsilon$-Tree with $O(\frac{\log_B^2 N}{\varepsilon})$ for point lookups and $O(\frac{\log_B^2 N}{\varepsilon} + \frac{K}{B})$ for range lookups. While bloom filters can increase the point lookup performance to $O(\log_B N)$, this does not work if there are messages addressed to the search key in different runs. This is due to the fact that scanning multiple levels defeats the purpose of a bloom filter whose goal is to return early upon queries with non-existent keys. Additionally, bloom filters do not improve the performance of range queries since those have to be performed on all runs in the LSM-Tree as well [31].

From an asymptotic point of view, a B$^\varepsilon$-Tree thus has the read performance of a B-Tree and the write performance of an LSM-Tree, making it the theoretically optimal choice. [2]

5 Implementation

Usually, to operate on working sets that do not fit into main memory, B^+-Trees and other index structures are implemented on top of a page buffer. The tree structure is then logically mapped onto these pages. This lets the page buffer evict currently unused parts of the tree for others that are requested by worker threads operating on the tree. To minimize these page evictions and, therefore, expensive I/O writes, the buffer generally employs a replacement strategy to detect "hot" pages that are often requested and keep them in memory while swapping out unneeded "cold" pages for new ones.

Our B^ε-Tree reference implementation, which we describe in the following, is built on such a page buffer. Additionally, in order to support multiple worker threads at the same time, we modify the design of the textbook B^ε-Tree to enable parallel access to keep node contention on simultaneous write operations at a minimum.

5.1 Base System

Figure 5.1 shows the layers of our implementation design. The B^ε-Tree offers key-value inserts via `insert`, updates via `update`, deletes via `erase` and point lookups via `find`. Every node of the B^ε-Tree is stored on one page with a fixed size. Pages are stored as-is on disk. The underlying page buffer employs a variant of the 2Q algorithm [22] for its replacement operations. Finally, we use a segment manager to store the pages in a single file on disk.

5.1.1 Disk Storage

We logically group the pages saved on disk into different segments of blocks of the same size as the pages. The segments are arranged next to each other and operate on independent ranges of a shared file. Every segment is individually responsible for managing the blocks in its assigned range. Blocks can be created, written to, read from, and marked as deleted. We can delete blocks by saving them in a free list, a singly linked list connecting all deleted blocks.

The segments are collectively referenced by a segment manager. In order to circumvent calling `ftruncate` for each new block, the segments grow exponentially in size (we

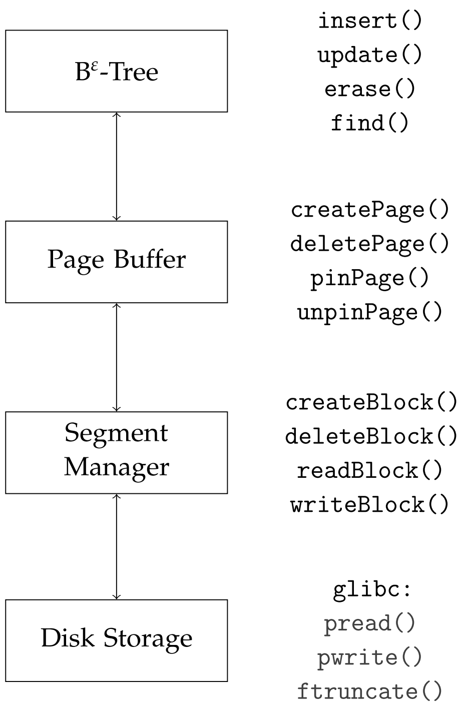


Figure 5.1: Implementation Layers and Their Respective Interfaces

use a growth factor of 1.25). Additionally, to keep the creation and deletion of blocks thread-safe, the segment manager protects itself and each segment respectively using a `std::shared_mutex`. However, concurrent reads and writes to the same block within one segment are not protected by the segment manager or the segment itself. This is handled by the page buffer.

5.1.2 Page Buffer

The 2Q strategy combines a Least Recently Used (LRU) queue with a First In, First Out (FIFO) queue. Implementation-wise, both the FIFO and the LRU queues are realized using a linked list to store the elements and a hash map that maps the keys to the list nodes to grant constant modification and lookup times. LRU strategies usually perform well when recognizing hot pages. However, pages that are accessed once take time to be evicted again since an LRU queue moves every accessed page to its front. The 2Q algorithm tries to prevent that by moving only pages to the LRU queue that were accessed at least twice while being in memory. The others are kept in the FIFO queue.

Listings 5.1 and 5.2 show both the insert/update and lookup/removal procedures of

Listing 5.1: 2Q Update Procedure

```
function UPDATE_QUEUE (page_id)
    if page_id $\in$ FIFO then
        FIFO.remove(page_id)
        LRU.push(page_id)
    else if page_id $\in$ LRU then
        LRU.remove(page_id)
        LRU.push(page_id)
    else
        FIFO.push(page_id)
    end if
end function
```

Listing 5.2: 2Q Lookup Procedure

```
function FIND_IN_QUEUE ( )
    if FIFO.size() $\geq$ 1 then
        page_id $\leftarrow$ FIFO.pop(page_id)
        return page_id
    else if LRU.size() $\geq$ 1 then
        page_id $\leftarrow$ LRU.pop(page_id)
        return page_id
    end if
    error("both queues are empty")
end function
```

a 2Q. By preferring the FIFO queue when looking for eviction candidate pages, we can keep the hot pages in memory while swapping out pages that were referenced only once. [22]

Worker threads can request ("pin") and release ("unpin") pages by referencing their ID. These operations are again protected by a `std::shared_mutex`. The buffer internally keeps a hashtable to map each ID to the actual location of the page in memory. Apart from its data, each page additionally contains metadata in the form of a flag indicating whether its data is different from the one of the disk ("dirty"), a pin counter, and a `std::shared_mutex` to, again, protect its data from concurrent operations.

In order to pin a page, each thread first assumes that the page is already in memory and acquires the shared lock of the page buffer. If the page is indeed in memory, the thread can look up its location, increase its pin counter, release the buffer's mutex, and lock the page. Else, it has to restart, lock the mutex of the buffer exclusively and check whether the page can be loaded into memory without evicting another page. Whenever it cannot, it has to select a page with zero pins from either the FIFO or the LRU queue, evict it, and load the requested page into memory.

When a page is evicted, the buffer first checks whether its data differs from the state currently saved on disk. If it does, we first have to write it to disk before we can remove it from the buffer. Since I/O operations are usually one of the main bottlenecks for these kinds of operations, we always release the main mutex of the buffer before them so that other threads may pin their pages in the meantime. However, we can neither guarantee that our page has already been loaded into the buffer nor that our eviction candidate was pinned in the meantime. Thus, after the I/O operation has finished, we have to re-lock the buffer mutex, check whether the two conditions described above are still fulfilled, and restart if they are not.

Since the page buffer design naturally involves overhead in the form of I/O operations and mutex-protected data structures, we also implement an "optimal" page buffer. This optimal page buffer stores the entire structure in memory and uses an atomic counter (`std::atomic_size_t`) for the page IDs. Each ID describes the memory offset of the respective page at the same time. Consequently, there are no I/O operations, and mapping an ID to its page can happen concurrently while the interface of pinning and unpinning pages stays the same. This lets us additionally evaluate the performance of our B^{ε}-Tree without the overhead of the 2Q page buffer later on.

5.2 B^{ε}-Tree

We define our B^{ε}-Tree to handle fixed key types (`class K`) and value types (`class V`) as template parameters. Additionally, as described above, each B^{ε}-Tree object depends on a predefined page (or block) size (`std::size_t B`) and an $\varepsilon \in [0;1]$ (`short EPSILON`). Since C++ does not allow floating point values as template parameters, we pass the value of ε in percent as an integer and process it with a `constexpr`.

5.2.1 Nodes

Layout

Listing 5.3 shows the definition for the upserts each B^{ε} inner node stores apart from its pivot elements, and their child pointers in the form of page ids. We use one byte for the upsert type and a `std::uint64_t` for the timestamp, which prevents the timestamps from overflowing in practice. A global atomic counter generates a strictly monotonically increasing timestamp for each new operation. Even though a delete operation does not require any value, we do not wrap the value into a `std::optional` which would need additional memory. Instead, we require `V` to be default-constructible.

Textbook B^{ε}-Tree upsert buffers are handled as an unordered write-append buffer, although implementations typically use a balanced tree structure [2]. In order to enable later optimizations, we instead treat each upsert buffer as a sorted array. Upserts are sorted first according to their keys and then by their timestamps. Additionally, we overload the three-way comparison function to be able to compare keys against an upsert key.

Leaf nodes store each key-value pair directly instead of using disk pointers. This prevents unnecessary I/O-operations when reading or writing to a leaf since the pairs can be cached in memory among frequent accesses.

Listing 5.3: Upsert Definition

```
template<class K, class V>
struct Upsert {
    K key;
    V value;
    std::uint64_t timeStamp = -1;
    unsigned char type = UpsertType::INSERT;

    auto operator<=>(const Upsert&) const;
    auto operator<=>(const K&) const;
};
```

Since the maximum number of pivot, pointer, and buffer slots in the inner nodes as well as the maximum number of key-value pairs in the leaf nodes solely depends on the template parameters `K`, `V`, `B` and `EPSILON`, we can calculate them at compile time and use a `std::array` for storage. Note that this approach would not work for dynamic key-value types and sizes.

To allocate a node on the preexisting memory of a page, we reserve one byte of that memory to differentiate between the different node types and allocate the node using a `placement new`. We can then cast the pointer to the memory location to gain access to the node. Additionally, we align the node memory with `std::max_align_t` to avoid undefined behavior when interpreting the memory as a node.

Operations

Like any B-Tree, our B$^{\varepsilon}$-Tree has to support node splits. While leaf splits are identical to B^{+}-Tree leaf splits, splitting inner nodes additionally requires partitioning the upsert buffers. Note that because we typically only split an inner node with respect to its pivots, the two resulting upsert buffers are rarely evenly occupied.

Since B$^{\varepsilon}$-Trees are usually used with insert-heavy workloads, we accept under-full nodes and forego node merges. The idea is that subsequent insert operations fill empty key-value pair slots in the leaf nodes left behind by delete operations.

Inserting an upsert into a buffer behaves similarly to inserting a pivot-pointer pair. As described above, we keep each upsert buffer sorted in the order defined by the key and the timestamp of the included upserts. Thus, in the worst case, inserting an upsert requires shifting all preexisting upserts to the right. Treating the upsert buffers as unsorted write-append buffers would let us insert upserts in constant time.

However, this approach enables logarithmic lookups regarding the buffer size $B - B^\varepsilon$ to continuous upserts addressed to the same key. Storing these compact "message blocks" benefits both lookup operations, which would otherwise need to scan the entire buffer, and node flushes. In order to keep a node buffer as dense as possible with respect to the message blocks after a flush, we need to merge two upsert buffers, which can be done in linear time if both buffers are sorted. Section 5.2.3 further describes how we perform such a flush.

When a node buffer contains multiple upsert messages $\{U_1^K, U_2^K, ..., U_n^K\}$ that are addressed to the same key K, these messages together describe a section of the ongoing change of K and its associated value V_K. However, since we are only interested in the latest state of the tuple (K, V_K), we can merge $\{U_1^K, U_2^K, ..., U_n^K\}$ into a new upsert U_{new}^K without losing any required information, assuming associativity of the update operation. Consequently, merging U_{new}^K into another upsert at a lower level or applying it to the leaf node will result in the same state for (K, V_K).
Instead of waiting for a buffer to be fully occupied and then accumulating the upserts with the same key, the operation can happen in place when an upsert is added to a node buffer. As a result, this will avoid shifting other upserts to the right if there is a preexisting upsert with the same key, as the message will get replaced with the resulting upsert of the operation.
Table 5.1 defines the binary merge operation for every input combination (we use $V_A + V_B$ as syntactic sugar for an update of value V_A with value V_B). We can then use this operation to prevent the upsert buffers from containing more than one upsert addressed to the same key. Note that we assume updates and deletes to have no effect if no value exists yet, and inserts to overwrite preexisting values.

5.2.2 Multithreading

Usually, for a B-Tree to safely handle concurrent write operations with one lock per node, each thread has to exclusively lock every node along its path to the respective leaf node since splitting the leaf node may propagate up to the root node. Because the root node is thus always part of the path, every write operation requires locking the whole structure. This makes it impossible for the B-Tree to scale with multiple threads. Since splits in general, however, happen rarely, a common approach is to acquire the respective shared locks of the path first, except for the leaf node [13]. If the thread detects that a node would overflow, it aborts the operation and restarts with an exclusively locked path. This enables multiple threads to perform writes simultaneously most of the time.

Another approach is to preemptively split full nodes on the way down [13]. Although this often results in early splits that are not immediately required, it enables the thread to

Table 5.1: Binary Upsert Merge Operation

First Upsert (Timestamp t)	Second Upsert (Timestamp $t+n$)	Resulting Upsert
$Insert_t$	$Insert_{t+n}$	$Insert_{t+n}$ (overwrites $Insert_t$)
$Insert_t$	$Update_{t+n}$	new $Insert_{t+n}$ with value $V_t + V_{t+n}$
$Insert_t$	$Delete_{t+n}$	$Delete_{t+n}$ (must propagate to delete any old value)
$Update_t$	$Insert_{t+n}$	$Insert_{t+n}$ (overwrites any old value)
$Update_t$	$Update_{t+n}$	new $Update_{t+n}$ with value $V_t + V_{t+n}$ (assumes associativity)
$Update_t$	$Delete_{t+n}$	$Delete_{t+n}$
$Delete_t$	$Insert_{t+n}$	$Insert_{t+n}$ (already overwrites the old value)
$Delete_t$	$Update_{t+n}$	$Delete_t$ (updating a non-existing value has no effect)
$Delete_t$	$Delete_{t+n}$	$Delete_t$ (deleting a non-existing value has no effect)

use lock coupling where only a maximum of two nodes are locked at once. Additionally, with a growing write-heavy workload, these splits would occur later anyway. While the nodes still have to be locked exclusively, the upper tree levels, especially the root node, do not have to be locked during the whole operation.

In contrast to a B-Tree, where write operations mainly occur on the leaf level, writes to a B$^\varepsilon$-Tree heavily involve the upper levels due to the buffered upserts. Since every write first inserts an upsert into the root node by design, the first approach cannot be applied as-is.

Therefore, we make use of preemptive splitting: Every time we flush down a range of upserts, we check whether the current node can manage a split of the respective child nodes. If this is not possible, we split it (see fig. 5.2), which lets us release parent nodes early during a flush operation for other threads to access the buffers of these parents simultaneously.

Note that the current node may need more than one free pivot slot since a flush can target multiple child nodes, as further described in section 5.2.5.

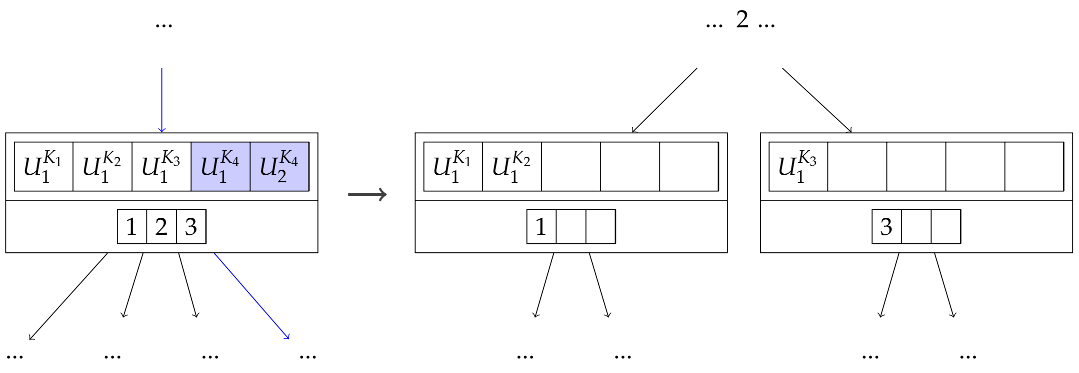


Figure 5.2: A preemptive node split. As we flush $U_1^{K_4}$ and $U_2^{K_4}$ to the fourth child, we preemptively split the node since its pivot slots are fully occupied. The remaining upserts $U_1^{K_1}, U_1^{K_2}, U_1^{K_3}$ are divided between the two resulting nodes.

Flushes, however, only happen when a buffer is full. This means that, even with preemptive splitting, large buffers cause scaling issues since most operations happen solely on the not fully occupied root node buffer. To counter this, we modify the B$^{\varepsilon}$-Tree textbook design and introduce a separate root node. The root node does not have a buffer but only pivots and child pointers - similar to the inner nodes of a B^{+}-Tree.
We then combine this idea with the aforementioned second approach. Since our new root node does not contain upsert messages, a writer thread does not have to lock it exclusively most of the time (see figure 5.3). Instead, it can optimistically acquire its shared lock and access the respective buffered node on the second level exclusively (see listing 5.4). When it detects an upcoming split on that level, it aborts and restarts the operation by locking the root exclusively. However, when the root node eventually overflows, we need to split it into multiple child nodes. Since every node as of the second level is a buffered node, and we use the same page size for all nodes, they thus contain fewer pivot slots than the root node.

This idea does not fundamentally change the structure of the B$^{\varepsilon}$-Tree but rather organizes multiple B$^{\varepsilon}$-Subtrees in a node that can be accessed concurrently. It also means that our B$^{\varepsilon}$-Tree design will eventually stop scaling with both the tree size and the number of worker threads if we do not adjust the root node size accordingly.

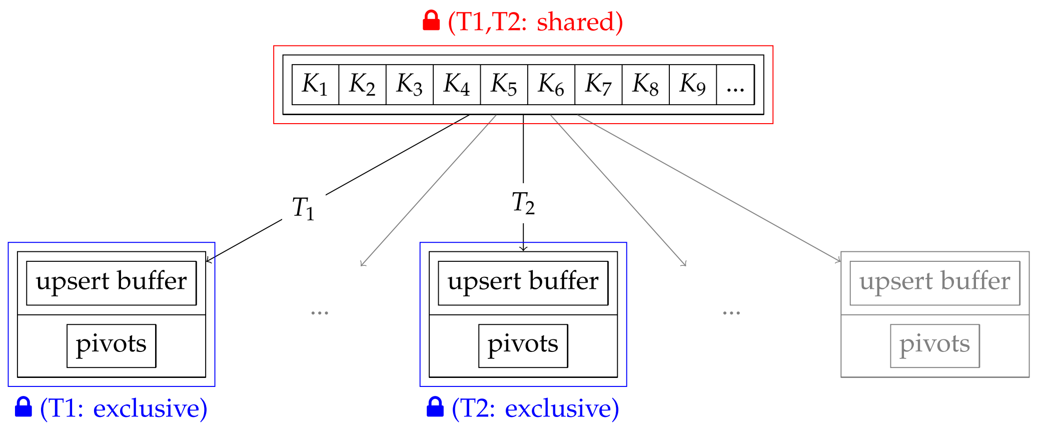


Figure 5.3: Lock Coupling with a separate root. T_1 and T_2 could acquire the shared lock of the root. Therefore, both threads can access their respective subtree simultaneously.

5.2.3 Upsert Flushes

When a buffer is fully occupied, we need to remove a subset of its upserts and divide it over the buffers of its respective children. Typically, a B$^\varepsilon$-Tree selects the child with the most pending messages as its target, which is sufficient in most cases [2]. Except for the root node, depending on the current node's key distribution, it can be necessary to flush upserts to multiple children.

Each flush makes use of the `remove_message` procedure (see listing 5.5). This procedure operates on the target node buffer $\{U_1^T, U_2^T, ..., U_n^T\}$ and takes in the incoming upserts $\{U_1^P, U_2^P, ..., U_m^P\}$ from the parent node. It then merges both $\{U_1^T, U_2^T, ..., U_n^T\}$ and $\{U_1^P, U_2^P, ..., U_m^P\}$, and again flushes down enough message blocks, beginning with the largest, to prevent the target node from overflowing. The remaining message blocks are kept in the current target node. Note that, because we never store two different upserts addressed to the same key in one buffer, as described above, we cannot use `std::merge` which already merges two sorted ranges. Instead, we use an adjusted version of `std::merge` which combines these upserts accordingly to table 5.1.

The combination of flushing multiple message blocks and preemptive splitting prevents us from using the typical depth-first approach like the reference implementation of the original paper does [33]. Otherwise, we would basically lock the whole tree during flushes. Instead, we use a level order traversal from the second level downwards using a FIFO `std::deque`. This queue holds tuples containing the current node and its former upserts removed using `removed_messages`. Each iteration removes the current tuple and calls `traverse_tree` (see listing 5.6). If the current node flushes its upserts to

Listing 5.4: Upsert Method

```
template<class K, class V, std::size_t B, short EPSILON>
void BeTree<K, V, B, EPSILON>::upsert(Upsert<K, V> upsert) {
    bool rootLeaf = (header.treeHeight == 1);
    PageT* rootPage = &pageBuffer.pinPage(header.rootID, rootLeaf);
    // first case: leaf node (direct operation)
    if (accessNode(*rootPage).nodeType() == NodeType::LEAF) {
        handleLeafUpsert(std::move(upsert), rootPage);
        return;
    }
    // second case: root node -> first, try shared locking
    bool success = handleRootUpsert(upsert, rootPage, false);
    if (!success) {
        // overflow detected -> retry using exclusive locking
        // (handleRootRootUpsert already unpinned the root page)
        rootPage = &pageBuffer.pinPage(header.rootID, true);
        handleRootUpsert(std::move(upsert), rootPage, true);
    }
}
```

a leaf node, traverse_tree treats the upserts as operations again and applies them to the respective node. Else, after potentially executing a preemptive split, each message block is flushed to the corresponding child node.
We then call remove_messages on each child node and push it into the queue if its buffer is also fully occupied. By applying this level order traversal, we can quickly unlock the upper levels again, which are exponentially denser regarding the number of nodes due to the nature of the tree structure.

Note that listing 5.6 simplifies the actual traversal algorithm. For example, we apply preemptive splitting on the leaf level to be able to unpin the parent node quickly. Only then can we execute the collected upserts on the respective leaf node.

As described above, this design uses temporarily storing upserts during each flush. Therefore, the tree is not resistant against crashes, even if the page buffer would be.

5.2.4 Lookups

When our B^{ε}-Tree has a height of at least one, each node on the way down, except for the root node, potentially contains information about the current state of the value

Listing 5.5: Messages Removal Procedure

```
function REMOVE_MESSAGES (node, upserts)
    needed_slots ← |node.upserts| + |upserts| - node.upserts.maxSize()
    all_upserts ← merge(node.upserts, upserts)  # merges upserts with equal keys
    message_block_references ← scan_blocks(all_upserts)
    sort(message_block_references)  # sorts the message blocks by decreasing size
    spare_upserts ← []
    for block ∈ message_block_references do
        if |spare_upserts| ≥ needed_slots then
            break
        end if
        message_block_references.remove(block)
        spare_upserts.push(block)
    end for
    sort_by_key(message_block_references)  # sorts the remaining messages by key
    node.upserts.clear()
    for block ∈ message_block_references do
        node.upserts.push(block)
    end for
    return spare_upserts
end function
```

V_K associated with the given lookup key K. In order to reconstruct this state, our `point_lookup` procedure (see listing 5.7) therefore has to keep track of all updates addressed to K. Since older upserts are flushed first and thus are located on lower node levels, we have to temporarily save every update message and afterward apply them to the leaf value in reverse order. Inserts and deletes can be used instead of the leaf values to terminate the traversal earlier.

Note that listing 5.7 describes the procedure for a general update operation. If the update operation is associative, we can use a single placeholder value instead of both the `accumulator_queue` and the `local_update_vector`, similar to the merge operation described in section 5.2.1.

5.2.5 Considerations and Limitations

Preemptive Splitting

Let n be the maximum number of pivot elements an inner node can hold. After splitting an arbitrary inner node I with n pivot elements into I_{left} and I_{right}, each I_{left} and I_{right} will have at least $\lceil \frac{n}{2} \rceil$ free pivot slots.
Likewise, every node can hold a maximum of $n+1$ child pointers. Additionally, a flush of upsert messages can temporarily overflow the buffer of a node up to twice its size, as figure 5.4 illustrates. Thus, the maximum number of addressed children during one flush is equal to $\lceil \frac{n+1}{2} \rceil$, which means that all children currently receive approximately the same number of messages. The messages addressed to the other $\lfloor \frac{n+1}{2} \rfloor$ children can be stored in the current buffer.

In order to be able to perform a flush with preemptive splitting from a parent node to its addressed children $\{C_1, C_2, ...\}$, we need to have enough space for a potential split of every child $\{C_1, C_2, ...\}$. Each path taken to the child nodes represents a potential split that has to fit into the current node. With one preemptive split, this can only be achieved if the resulting number of free pivot slots of both I_{left} and I_{right} is equal to the number of addressed children, or $\lceil \frac{n}{2} \rceil = \lceil \frac{n+1}{2} \rceil$, which is only fulfilled for odd n.

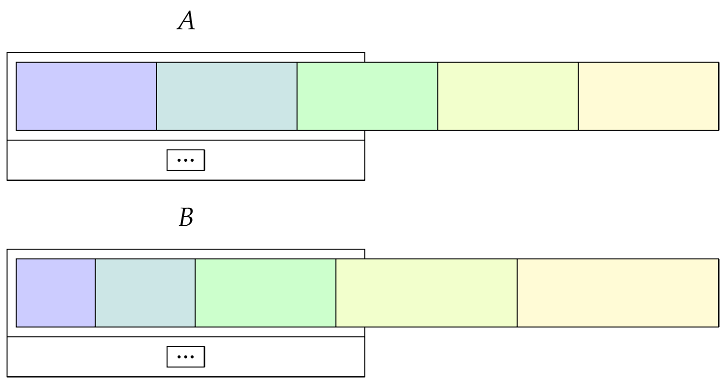


Figure 5.4: The state of two overflowed upsert buffers. Both A and B received every upsert of their parent buffer, overflowing their buffers by a factor of 2. A greedy approach that first moves the largest message blocks to the respective children is optimal for minimizing flushes. The more evenly distributed the message blocks are in size (A), the more message blocks need to be flushed.

Upsert Flushes

When a flush targets a leaf node L, it is possible for the messages to only consist of new inserts, which creates an upper bound for the maximum number of flushed messages. If we always split leaf nodes in half, we only make space for $\lfloor \frac{s}{2} \rfloor$ tuples in both the old and the new leaf node, where s denotes the maximum number of tuples any leaf node can hold. As a result, any upsert flush may only remove $\lfloor \frac{s}{2} \rfloor$ upserts from any given node buffer.

In order to circumvent this, we can split L with respect to the median key K_M of the union of the flushed upsert keys $\{K_{U_1}, K_{U_2}, ..., K_{U_n}\}$ and the existing keys $\{K_L^1, K_L^2, ..., K_L^m\}$ of the leaf node. By choosing K_M as the pivot key for the split, we can divide $\{K_{U_1}, K_{U_2}, ..., K_{U_n}\} \cup \{K_L^1, K_L^2, ..., K_L^m\}$ and their values evenly on both the resulting nodes L_{left} and L_{right} after the split. Because both key ranges are sorted, this can be done in logarithmic time with respect to $|\{K_{U_1}, K_{U_2}, ..., K_{U_n}\}| + |\{K_L^1, K_L^2, ..., K_L^m\}|$ without having to merge them beforehand. Consequently, the upper bound for the maximum number of flushed messages increases from $\lfloor \frac{s}{2} \rfloor$ to s.

Since any upsert contains a key-value pair as well as an additional timestamp, its size will always be greater than that of a leaf tuple. Thus, the maximum amount of upserts a node buffer can hold will never exceed s (regardless of the chosen value for ε) if both the inner and leaf nodes share the same page size. In summary, preemptive splitting does not prevent flushing all messages of one full node buffer.

The Range of ε

As described above, for any given $\varepsilon \in [0; 1]$ and a node size of B bytes, we use B^ε bytes for the pivots (and their pointers) and $B - B^\varepsilon$ bytes for the upsert buffer. However, in reality, an inner node of a B$^\varepsilon$-Tree always needs at least a buffer with one upsert slot to handle write operations and two pivot elements to split itself (or three if we use preemptive splitting).

Thus, instead of defining B as the total node size B_{total}, we use it to describe $B_{total} - B_{base}$ where B_{base} marks the space needed for one buffer slot as well as three pivot slots and their pointers. This lets us define $\varepsilon \in [0; 1]$ while leaving the B$^\varepsilon$-Tree operational.

Listing 5.6: Flush Traversal Procedure

```
function TRAVERSE_TREE (node, upsert_map)
    # <upsert_map> contains the removed upserts from <node>
    for (child_index, upserts) $\in$ upsert_map do
        child $\leftarrow$ node.children[child_index]
        # leaf node level
        if child.isLeaf() then
            future_size $\leftarrow$ child.size()
            for upsert $\in$ upserts do
                apply_upsert(child, upsert)
                if need_to_split(child) then
                    split(child)
                end if
            end for
        else
            # inner node level
            if child.upserts.max_size() - child.upserts.size() $\geq$ upserts.size() then
                # the child has enough space for the upserts
                child.upserts $\leftarrow$ merge(child.upserts, upserts)
            else
                map $\leftarrow$ remove_messages(child, upserts)
                if child.pivots.max_size() - child.pivots.size() $\leq$ map.size() then
                    # preemptive split
                    (left_node, right_node) $\leftarrow$ split(child)
                    # divide the removed upserts with respect to the split
                    (left_map, right_map) $\leftarrow$ split_map(map, left_node, right_node)
                    queue.push(left_node, left_map)
                    queue.push(right_node, right_map)
                else
                    queue.push(child, upserts)
                end if
            end if
        end if
    end for
end function
```

Listing 5.7: Lookup Procedure

```
function POINT_LOOKUP (key)
    child $\leftarrow$ binary_search(root.children(), key)
    accumulator_queue $\leftarrow$ []  # contains the accumulated updates
    current_value $\leftarrow$ NULL  # the final base value
    while child.type = INNER do
        deleted $\leftarrow$ false
        local_update_vector $\leftarrow$ []  # contains the updates of the current node
        message_block $\leftarrow$ binary_search(child.upserts(), key)
        for upsert $\in$ message_block do
            if upsert.type = DELETE then
                local_update_vector $\leftarrow$ []
                current_value $\leftarrow$ NULL
                deleted $\leftarrow$ true
            else if upsert.type = UPDATE then
                local_update_vector.push(upsert.value)
            else if upsert.type = INSERT then
                local_update_vector $\leftarrow$ []
                current_value $\leftarrow$ upsert.value
                deleted $\leftarrow$ false
            end if
        end for
        if deleted then
            accumulator_queue $\leftarrow$ []
        end if
        accumulator_queue.push_front(local_update_vector)
        if deleted = true $\vee$ current_value $\neq$ NULL then
            break
        end if
        child $\leftarrow$ binary_search(child.children(), key)
    end while
    if current_value = NULL then
        current_value $\leftarrow$ binary_search(child.upserts(), key)
    end if
    for update_value $\in$ accumulator_queue do
        current_value $\leftarrow$ current_value + update_value
    end for
    return current_value
end function
```

6 Evaluation

6.1 Benchmark Setup

A commonly used benchmark tool to analyze key-value stores is the Yahoo Cloud Serving Benchmark (YCSB) [6]. YCSB uses workload generators with pre-defined templates and an interface for inserting, updating, deleting, and reading single key-value tuples, as well as performing range scans.
Because YCSB is written in Java, one has to create a separate interface to analyze structures purely written in C++ with YCSB. Alternatively, there are ports like YCSB-C[1], its successor YCSB-cpp[2] or, what we will use in the scope of this thesis, the Unum Cloud Serving Benchmark (UCSB) [39], a rewrite of YCSB based on the Google Benchmark[3] library [40].

In addition to the B^{ε}-Tree, we additionally built a B^{+}-Tree for comparison. It uses the same techniques as the B^{ε}-Tree by splitting nodes preemptively to enable exclusive lock coupling and trying to travel down to the leaf level with shared lock coupling first. The key-value pairs are stored directly in the leaf nodes as well.

If not specified otherwise, we prepare the following test runs using workloads with the Zipfian distribution to simulate "hot" keys with a higher probability of being targeted. For the Zipfian generator, YCSB and UCSB both use a ϑ ("Zipfian constant") of 0.99. We set the key size to 8B and the value size to 100B, similar to the default configuration of YCSB. Additionally, both the B^{ε}-Tree and the B^{+}-Tree will work with a typical page size of 16KiB.

We run all benchmarks on a system with an Intel i9-7900X (10 cores, 20 hardware threads) with 128GiB of RAM and a Samsung SSD 970 EVO running Ubuntu 22.04.

6.2 Analyzing the Impact of ε

First, the performance of our B^{ε}-Tree for different values of ε is evaluated. In order to independently analyze the behavior, we make use of our "optimal" buffer (see section

[1]https://github.com/basicthinker/YCSB-C
[2]https://github.com/ls4154/YCSB-cpp
[3]https://github.com/google/benchmark

5.1.2), which omits the overhead of locking and I/O operations. This lets us analyze the performance of the B^{ε}-Tree with multiple threads more precisely.

Figure 6.1 shows the results for two workloads of 200M (million) inserts and 200M reads. We fixed the number of threads on 1 and 16, respectively, and evaluated both the Zipfian and the uniform distribution for our workloads. The workloads were run on a preloaded B^{ε}-Tree (200M inserts with the same respective distribution). Note that we cut off the diagrams at $\varepsilon = 0.3$. Since the fanout grows exponentially with ε, the B^{ε}-Tree effectively uses the same layout for $\varepsilon \in [0; 0.36]$.
Metadata like page allocations, the final tree height as well as the number of pivot and buffer slots can be seen in figure 6.2.

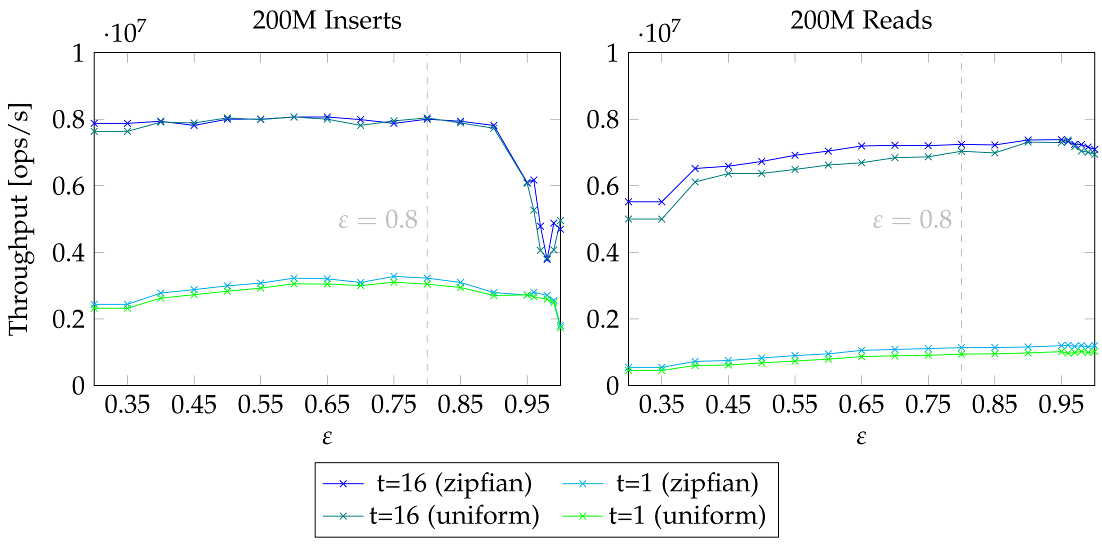


Figure 6.1: 200M Operations on 200M Preloaded Values

When comparing the results for the first workload, we can see that the trends of the singlethreaded and the multithreaded runs are headed in the same direction. While the number of buffer slots stays roughly the same until $\varepsilon \approx 0.75$, the fanout increases noticeably already at $\varepsilon \approx 0.45$. Consequently, until $\varepsilon \approx 0.75$, the throughput of both runs slightly increases, although especially the multithreaded runs overall perform similarly. After $\varepsilon \approx 0.8$, the buffer slots shrink heavily in number, and both throughputs start to decline. The performance of the multithreaded runs drops by $\approx 53\%$ after $\varepsilon \approx 0.9$ but slightly recovers at $\varepsilon \approx 0.99$ due to the fact that the multiple worker threads benefit from a higher fanout. The singlethreaded variant declines until $\varepsilon \approx 0.98$ and then drops by $\approx 36\%$. The negative theoretical impact of larger values for ε on throughput becomes apparent for $\varepsilon \in [0.8; 1]$ with the exception of $\varepsilon \geq 0.99$ for the multithreaded

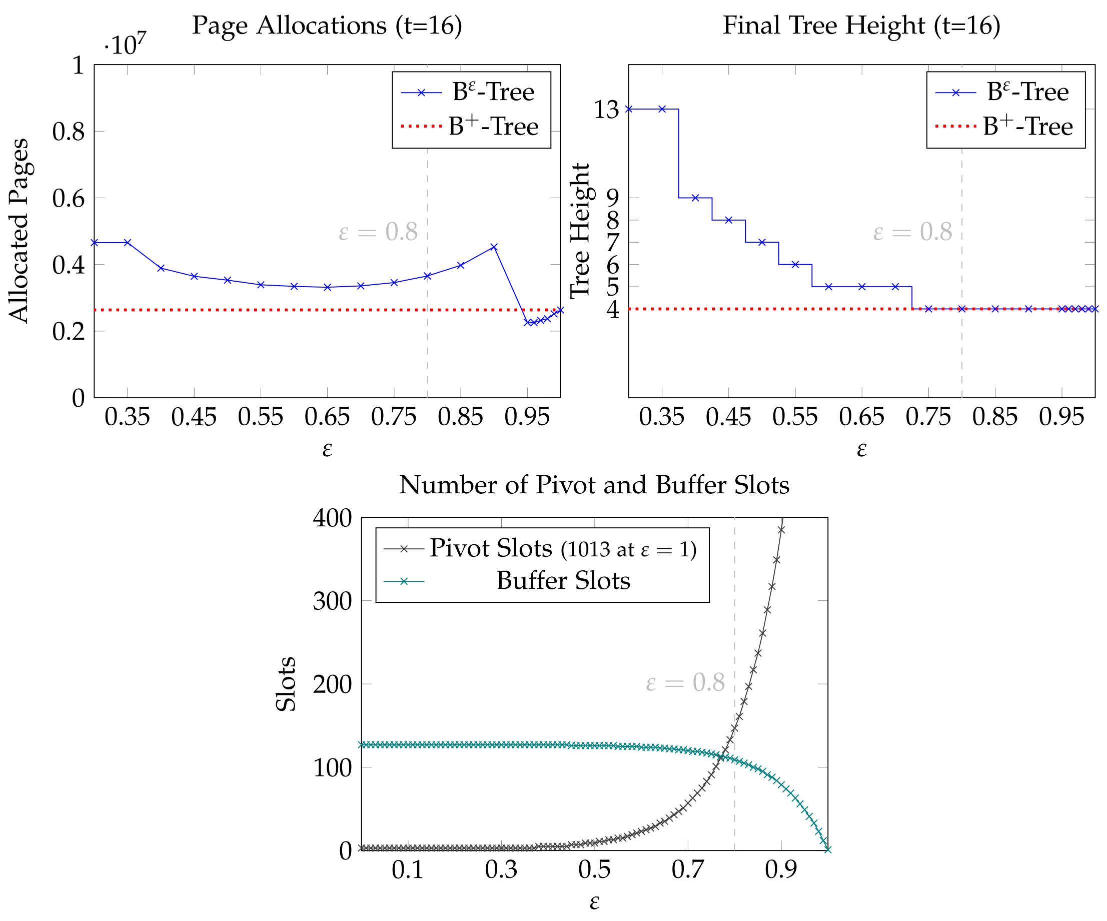


Figure 6.2: 200M Operations on 200M Preloaded Values: Metadata

runs. Because the number of buffer slots effectively does not change for small values of ε, the throughput is consistent and even slightly increases in that range as the fanout grows and the tree height declines.

With the shrinking height and the growing fanout of the B$^\varepsilon$-Tree, the throughput of the read-only workload run with one thread steadily increases from ≈ 500000 ops/s to ≈ 1100000 ops/s throughout all values for ε. The multi-threaded runs especially benefit from this after $\varepsilon \approx 0.35$ when the final tree height drops from 13 to 9, which means that each worker thread now has to access $\approx \frac{2}{3}$ of the previous number of pages for its operations. After increasing from ≈ 5250000 ops/s to ≈ 7300000 ops/s at $\varepsilon = 0.95$, the throughput slightly declines. With the heavily declining number of buffer slots and the fact that the final tree height of 4 effectively does not change after $\varepsilon = 0.7$, fewer worker threads encounter cached inserts on higher levels and thus are unable to finish the query early.

Accessing a subset of values multiple times allows the B$^\varepsilon$-Tree to cache them in the buffers of the inner nodes. As described in sections 5.2.1 and 5.2.4, insert and read operations consequently can be performed faster on this subset. Thus, in both cases, the workloads with the Zipfian distribution can always be processed as fast or even slightly faster than the uniform distribution. Especially the second workload benefits from the Zipfian distribution as the multithreaded runs show. Up until $\varepsilon \approx 0.85$, the Zipfian distribution increases the throughput by ≈ 400000 to 500000 ops/s. The decline after $\varepsilon = 0.95$ confirms that the Zipfian distribution causes a flatter performance decrease than with the uniform distribution since it encounters buffered inserts more often.

Caching inserts also impacts the density of our B$^\varepsilon$-Tree. Figure 6.2 shows the total number of page allocations for the first workload with the Zipfian distribution, the difference to the uniform distribution was negligible in this case. When we compare the two ranges $\varepsilon < 0.9$ and $\varepsilon \geq 0.9$, we can see that the second produces $\approx 40\%$ denser trees. At $\varepsilon = 0.95$, it even has $\approx 15\%$ fewer page allocations than a comparable B$^+$-Tree due to operations that encounter cached inserts and thus can be merged early. At $\varepsilon = 1$, the B$^\varepsilon$-Tree allocates the same number of pages as the B$^+$-Tree when the fanout is effectively identical, and no inserts are buffered. While the first range allocates ≈ 4600000 pages at both $\varepsilon = 0$ and $\varepsilon = 0.9$, this number drops to ≈ 3300000 in-between at $\varepsilon = 0.65$.

Because of the resulting smaller fanout, smaller values for ε generally cause more page allocations, which is amplified by the use of preemptive splitting. The increase in allocations for $\varepsilon \in [0.65; 0.9]$ is again due to the asymmetry in the number of pivot slots and buffer slots. After $\varepsilon = 0.7$, the tree height does not change anymore, making the decline in the number of buffered inserts cause more flushes and thus preemptive splits. Once the number of pivots per node surpasses the number of buffer slots after $\varepsilon = 0.9$, the B$^\varepsilon$-Tree quickly approaches the fanout of that of the B$^+$-Tree. Amplified by the size difference between a child-pointer pair and an upsert, the fanout decreases the number of preemptive splits, resulting in fewer page allocations.

For the following tests, we set ε to 0.8 as this value yielded the highest throughput result for the insert-only workload. The value $\varepsilon = 0.8$ consequently seems to mark the boundary after which the insert-only performance starts to decline. Although the results were similar for $\varepsilon \in [0.4; 0.8]$ in both the singlethreaded and the multithreaded runs, the second workload showed that a higher fanout and thus smaller tree heights benefit read operations.

It should be noted that the behavior of our B$^\varepsilon$-Tree heavily depends on the size of the value type. Therefore, workloads with different parameters may produce substantially deviating results for both the performance and the throughputs.

6.3 B^+-Tree Comparison

For comparing our B^ε-Tree and the reference B^+-Tree implementation, we again use the "optimal" page buffer, i.e., we operate fully in memory. In addition, we evaluate a variant of our B^ε-Tree without a separate root node, which resembles the textbook design, to test whether our variant increases the performance with multiple threads. Figure 6.3 shows the achieved throughputs for the following five workloads:

1. 200M inserts (write-only)
2. 200M inserts (after a prerun of 200M inserts) (write-only)
3. 200M reads (after a prerun of 200M inserts) (read-only)
4. 190M reads + 10M inserts (after a prerun of 200M inserts) (mixed)
5. 100M reads + 100M updates (after a prerun of 200M inserts) (mixed)

In contrast to the B^+-Tree, the average writer thread in a B^ε-Tree only has to access two nodes (or one if no separate root is used) and perform one sorted insert. Therefore, with one thread, both variants of the B^ε-Tree process the first workload $\approx$ 25% faster than the B^+-Tree. However, if we process the same workload on pre-populated trees, the B^ε-Tree without a separate root node slightly decreases in performance compared to our variant. The separate root produces multiple B^ε-Subtrees that are organized in a node with the same fanout of that of our B^+-Tree. Populating this variant results in fewer expensive flushes at the cost of only one additional node traversal.

With multiple threads, inserting can always be performed orders of magnitude faster when using a separate root node. Splitting the B^ε-Tree into multiple subtrees enables the worker threads to access their respective buffers concurrently. However, using a buffer in the root node causes expensive thread contention. Consequently, with 20 threads, we can perform inserts up to 10 times faster on empty trees and up to 11.5 times faster on pre-populated trees. The B^+-Tree struggles to scale after six threads using the first workload due to the initial small tree height as well as the asymmetry in the number of slots in the leaf and inner nodes.
With 20 threads, building up the tree can still be performed $\approx$ 3.6 times faster compared to the B^ε-Tree without a separate root. The throughput achieved on the second workload with 20 threads ($\approx$ 9.4 times faster) further shows that inserts do not scale on lock-based textbook B^ε-Trees. However, our variant increases throughput with multiple threads, processing the first two workloads 2.8 and 1.2 times faster than the B^+-Tree.

Reads, on the other hand, cannot be performed as fast compared to the B^+-Tree because of their height difference. While all three trees scale until $\approx$ 15 threads and then stagnate, the B^+-Tree performs the third workload with up to $\approx$ 1300000 more operations per second than our variant of the B^ε-Tree. Up until 4 threads, both B^ε-Tree variants behave similarly. However, after that, our variant scales better, with up to $\approx$ 1070000 more operations per second.

The mixed workloads again show the importance of adapting the textbook design of a B^ε-Tree for multithreading. While our variant scales when processing the fourth and fifth workload, the B^ε-Tree without a separate root struggles after a few threads and steadily holds a throughput of $\approx$ 25% and $\approx$ 8% of that of our variant, respectively. The B^+-Tree, however, beats both as it achieves up to $\approx$ 1760000 more operations per second in the fourth workload. Regarding the fifth workload, the B^+-Tree even doubles the throughput of our B^ε-Tree variant throughout nearly the entire thread range.
While the average B^+-Tree operation can use shared lock coupling to travel to the respective leaf node, every writer thread operating on a B^ε-Tree has to lock its visited nodes exclusively (apart from the root node). Due to the nature of the `std::shared_mutex` that protects each node, this results in conflicts between reader and writer threads and thus causes thread contentions. Because workload #5 has a more balanced number of reads and writes, our B^ε-Tree consequently scales slower compared to workload #4.

Furthermore, workload #5 eventually causes the upsert buffers in the higher levels to only contain updates. This prevents more reader threads from returning early upon encountering a cached insert, which the Zipfian distribution benefits from. Thus, even with one thread, the performance of our B^ε-Tree variant drops by $\approx$ 45% compared to workload #4.

6.4 Key-Value Store Comparison

For the final evaluation, we compare our B^ε-Tree and B^+-Tree to Googles's LevelDB [12] and Facebook's RocksDB [11], a fork of LevelDB, which both make use of a LSM-Tree. We make our 2Q buffer and the two LSM-Trees process the first four workloads (see section 6.3) once with 40GB of memory for the data to fully fit into the RAM (see figure 6.4), and once with 10GB of memory.

Since the default UCSB configurations limit the available memory to 5% or 10% of the workloads, we adapt the default database configurations of the 1TB workloads for LevelDB and RocksDB. The changes can be seen in table 6.1. While LevelDB only keeps up to two memtables in memory [12], RocksDB can be adjusted to use more [11], which increases performance, especially during the multithreaded runs. We found `max_write_buffer_number=32` to be a good value. For the insert-only workloads #1 and #2, we let LevelDB and RocksDB use the given memory exclusively for the memtables. Since workload #3 and #4 additionally make use of read operations, we used 50% of

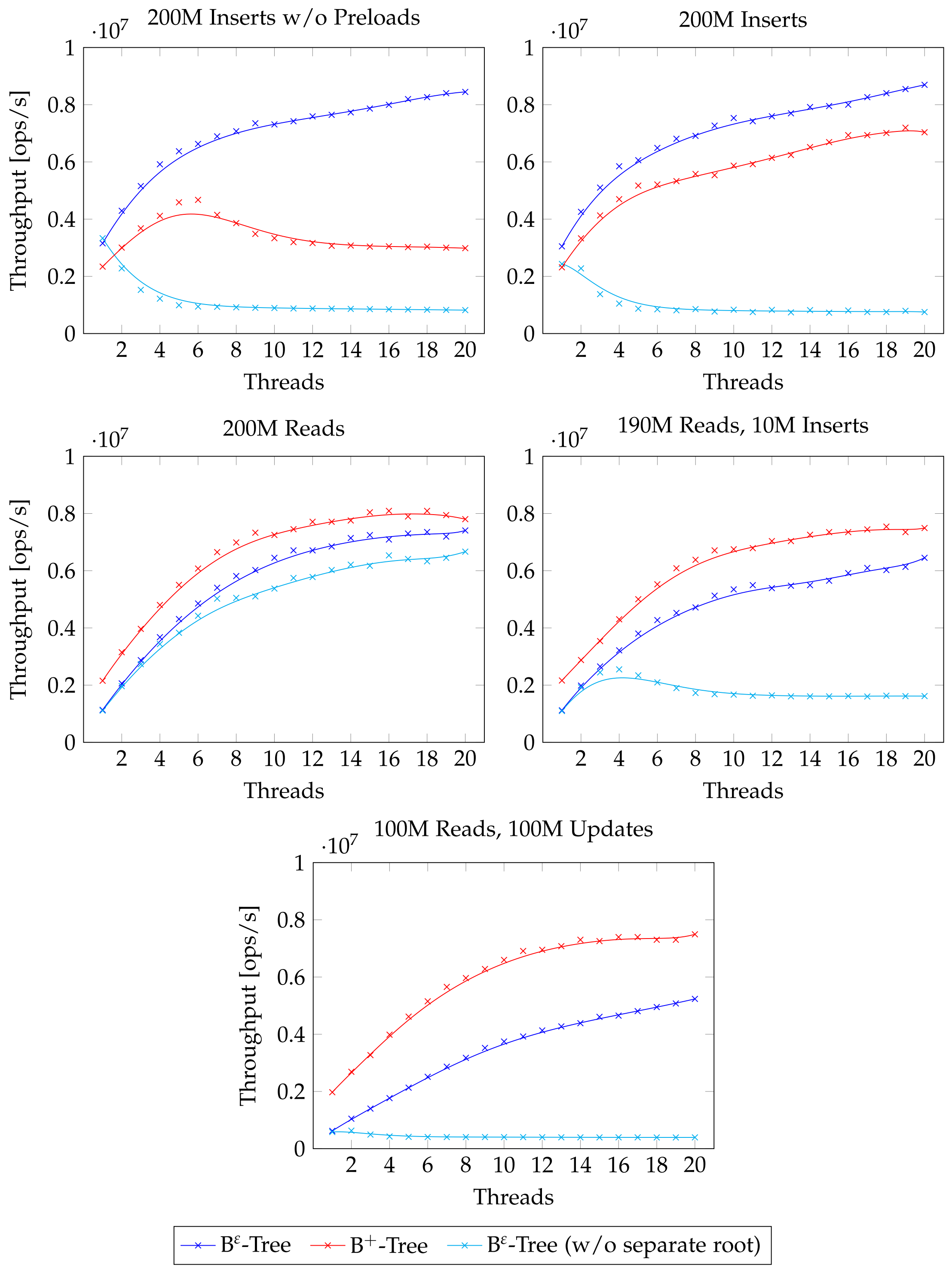


Figure 6.3: 200M Operations: B$^\varepsilon$-Tree ($\varepsilon = 0.8$) / B$^+$-Tree

the memory for the block cache there.
Additionally, we used a bloom filter of 10 bits for both LSM-Trees. These settings made RocksDB use approximately as much physical memory as our 2Q buffer did. It should be noted, that LevelDB indeed did the same, but only regarding virtual memory. LevelDB generally seemed to prefer early disk writes to free its memtables quickly. Thus, we observed an average physical memory usage of only $\approx$ 30% of the given boundaries.

Table 6.1: Adjustments of the Default UCSB Configs (1TB) for LevelDB and RocksDB ($\texttt{M} \in \{\texttt{40GB}, \texttt{10GB}\}$)

	LevelDB	RocksDB

Workloads #1, #2	`write_buffer_size=M`	`write_buffer_size=M/32`
		`max_write_buffer_number=32`

Workloads #3, #4	`write_buffer_size=M/2`
	`block_cache=M/2`
	`filter_bits=10`
		`max_write_buffer_number=32`
		`write_buffer_size=M/64`
		`block_cache=M/2`
		`(+ filter of 10 bits)`

Figures 6.4 and 6.5 show the corresponding results, which follow a similar direction regardless of the respective workload. LevelDB is generally on the slow end, independent of the number of threads. Regarding RocksDB, because the data in memory is divided on 16 times the number of memtables compared to LevelDB, the average worker thread encounters fewer write stalls, which occur when all memtables are full [29]. Therefore, RocksDB always achieves the highest throughput for ≥ 3 threads and generally scales with multiple threads.
However, this cannot be said about the performance of the B^ε-Tree and the B^+-Tree, which drops the more threads are in use. As described in section 5.1.2, our 2Q buffer uses a lock-protected hashtable that maps the page IDs to the respective location in memory. Although we handle the I/O operations of page evictions and loads outside of the scope of that lock, the logic of the 2Q strategy still relies on being protected by said `std::shared_mutex`. Consequently, most of the time, the average B^ε-Tree and B^+-Tree worker thread operates on the page buffer rather than on its respective nodes during its operations. The resulting thread contentions cause the throughput to drop.

Nonetheless, both the B^ε-Tree and the B^+-Tree manage to beat both LevelDB and RocksDB when staying in memory with one thread. With a memory limit of 10GB, the B^ε-Tree is the only one, achieving $\approx$ 180% and $\approx$ 140% of RockDB's throughput for workload #1 and #2, respectively. Especially the result of the second workload, which is processed on a prefilled tree, again shows the efficiency of the B$^{\varepsilon}$-Tree, regarding I/O operations, compared to the B^{+}-Tree. Each upsert flush divides the cost of accessing the target node on a lower level by the number of flushed upserts and, therefore, operations. Thus, the average writer thread has to access fewer nodes than the B^{+}-Tree worker threads. This explains the performance gain of $\approx 2.5\times$ when staying in memory and $\approx 7.2\times$ with a memory limit of 10GB.

Reads, on the other hand, require the B$^{\varepsilon}$-Tree to access more nodes compared to the B^{+}-Tree due to the larger tree height, which results in more I/O operations. Therefore, the B$^{\varepsilon}$-Tree performs workloads #3 and #4 consistently slower, although the difference is orders of magnitude smaller compared to the performance gain in workloads #1 and #2.

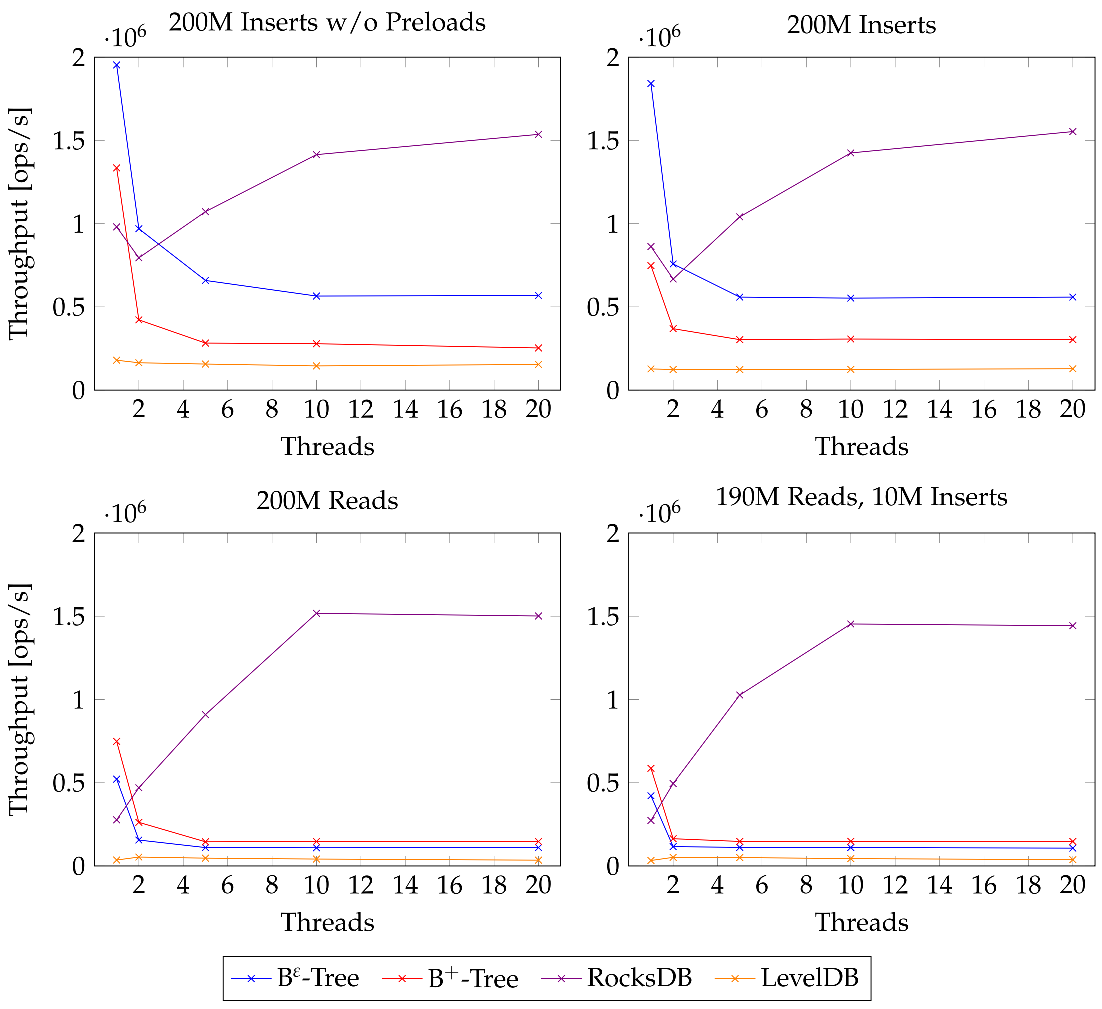


Figure 6.4: 200M Operations In-Memory: B$^\varepsilon$-Tree ($\varepsilon = 0.8$, 2Q) / B$^+$-Tree (2Q) / RocksDB / LevelDB

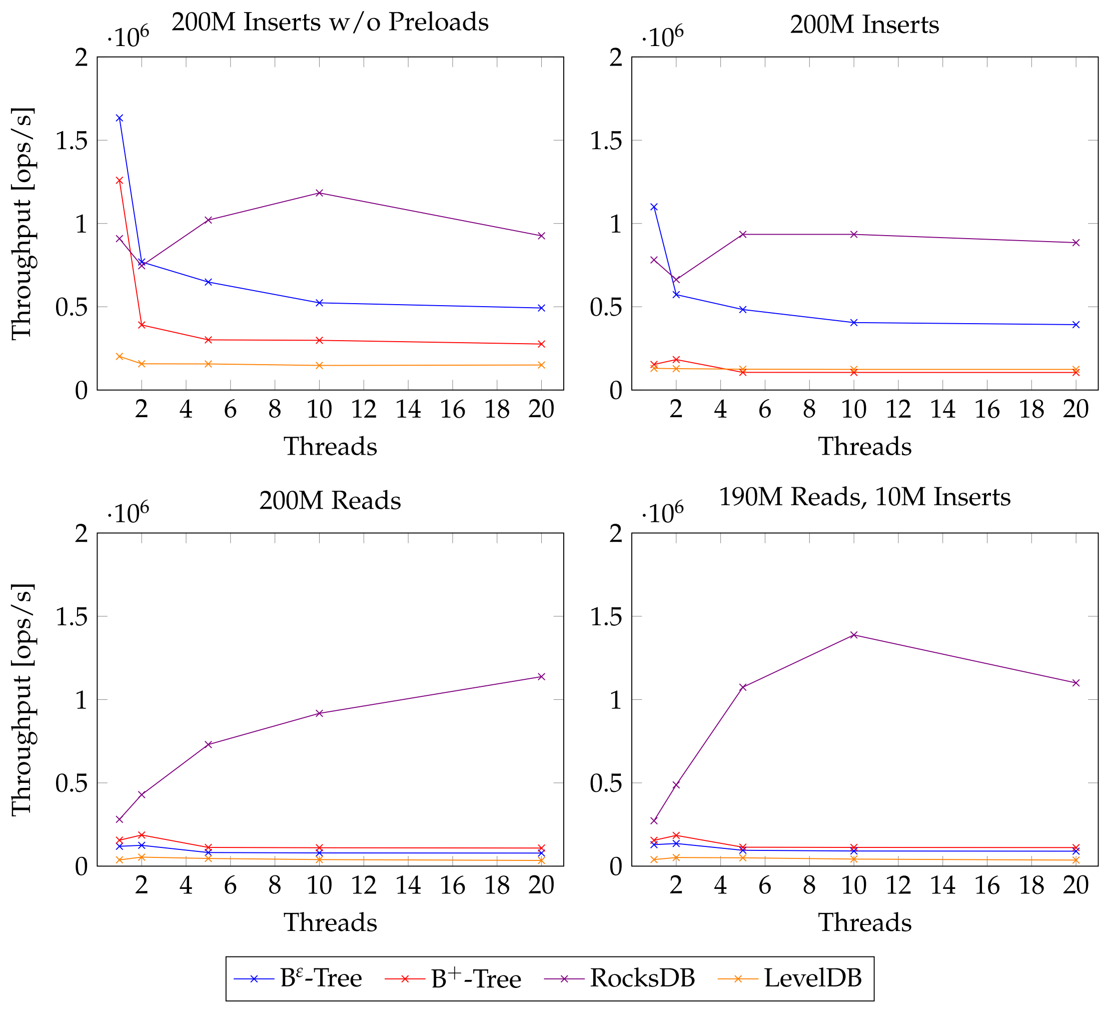


Figure 6.5: 200M Operations with 10GB of Memory: B^ε-Tree ($\varepsilon = 0.8$, 2Q) / B^+-Tree (2Q) / RocksDB / LevelDB

7 Related Work

Additional work that makes use of B$^\varepsilon$-Trees includes Percona's[1] PerconaFT [35], a transactional key-value store used in the TokuDB storage engine [36]. PerconaFT, and thus TokuDB, are built using a fractal tree index. In the past, fractal trees indexes used cache-oblivious lookahead arrays (COLA) [24] which keep their data in either fully occupied or completely empty arrays on multiple levels both in memory and on disk, similar to a LSM-Tree. Nowadays, however, fractal tree indexes extend B$^\varepsilon$-Trees by, for example, additionally guaranteeing properties like ACID [34].
Examples for systems using TokuDB are TokuMX [37], a distribution of MongoDB[2] or BetrFS [20], an in-kernel file system which uses B$^\varepsilon$-Trees as the main data structure to organize data stored on disk. While BetrFS cannot match e.g. ext4[3] when performing large sequential I/O operations, creating small files can be done orders of magnitude faster due to the nature of the used B$^\varepsilon$-Tree [20].

Although PerconaFT, as a key-value store, operates on the same layer as LevelDB and RocksDB, we did not include it in our benchmarks as the public repository seemed rather unmaintained and none of the tested commits were buildable on our machine[4]. Nonetheless, related comparisions between TokuDB and RocksDB [5, 18] suggest that, while TokuDB's is theoretically more performant due to its fractal tree index, LSM-Tree based systems like RocksDB can be tuned to have a higher throughput for different workload classes in practice. Percona even deprecated TokuDB in newer versions for Percona Server [14] and replaced it with MyRocks [30], an adaptation of RocksDB for MySQL.

[1]https://www.percona.com/
[2]https://www.mongodb.com/
[3]https://ext4.wiki.kernel.org/
[4]Tested commits:
`master:930d71f`,
`tokudb-7.5.8:00891ca`,
`releases/tokudb-7.5:eed6ab3`,
`releases/Percona-Server-5.6.27-76.0:cc7b178`

8 Future Work

While chapter 6 provided insight into the performance behavior of our B^{ε}-Tree, additional evaluations could clarify its behavior even more. Apart from switching workloads as well as altering page sizes, key sizes, and value sizes, especially testing with different node sizes would let us investigate the impact of the trade-off between random and sequential I/O operations.
Generally, B^{ε}-Trees should benefit from larger node sizes which amortize the costs of the random seek away [2]. However, the question remains whether this would benefit the combination of multiple threads and the current lock-based node protection approach. With increased node sizes, each worker thread has to lock a larger part of the index for its operations, potentially impacting scaling performance.

Apart from further benchmarks, the following modification ideas may improve performance, both in-memory and with memory limitations, and broaden applications.

8.1 Optimistic Lock Coupling

Our B^{ε}-Tree design performs write-only workloads faster than our B^{+}-Tree and is only slightly slower when processing read-only workloads (see section 6.3). Mixed workloads, however, cause the performance to drop due to the lock-based protection of the pages.

A general-purpose method that increases scalability for tree structures with similar lock granularity is Optimistic Lock Coupling (OLC) [27]. Instead of a `std::shared_mutex`, Optimistic Locks protect node writes with a `std::mutex`. Reader threads concurrently access their respective nodes optimistically by verifying an atomic version counter after reading from a node. When the version changes during the read, the thread restarts its operations. OLC benefits from the fact that, while writer threads have to lock their nodes exclusively to be able to handle the worst case of splitting, these splits happen rarely. Consequently, readers do not have to wait for the writers to finish as with traditional lock coupling with a `std::shared_mutex` but can access the nodes simultaneously most of the time. Additionally, the overhead of thread contentions on single mutexes occurs less often.

Since our B^{ε}-Tree suffers from these conflicts, which OLC solves in traditional tree structures, applying OLC could result in increased performance when processing mixed workloads with multiple threads. However, in contrast to a B^+-Tree, each exclusive node access results in a modification of the upsert buffer. Therefore, a reader thread would have to restart more often since the modifications generally happen in the higher levels of the B^ε-Tree. While this could be encountered by creating a node-specific index for the upsert buffers, like the original paper suggests [2], and a more granular lock structure, the question remains whether the B^ε-Tree design would benefit from OLC as much as, for example, B^+-Trees do.

8.2 Dynamic Values for ε

The benchmarks also showed that different values of ε resulted in different performances for read-only and insert-only workloads. Thus, if the general distribution of reads and writes ε is known at runtime, ε could be chosen dynamically to improve the respective throughput. Alternatively, the trade-off between pivot slots and buffer slots could be changed at runtime to minimize splits and flushes, similar to the reference implementation of the original paper [33].

8.3 Parallel Flushes

Flushing upserts sometimes requires a single thread to lock and operate on multiple nodes of the same level, as outlined in section 5.2.3. Because of the level order traversal, the calls to `traverse_tree` for these nodes can happen independently of the others. We could perform these flushes in parallel by spawning a new thread or leaving exclusive access of a node to an existing waiting thread.

8.4 Durability

The current B^ε-Tree design is not durable, i.e., crashes may cause data loss. Even if the page data would be protected from this by, for example, the page buffer, our level-order traversal requires the worker threads to temporarily store upserts in memory, which are not protected during the flush. To encounter this, the threads could use a separate write-ahead log (WAL) which also stores temporary copies of the upserts. After a crash, the B^ε-Tree could use the WAL to restore the state of each thread and rebuild the upsert buffers.

8.5 Advanced Buffer Manager

As described in section 6.4, the lock-protected page table represents the main bottleneck of our buffered B$^\varepsilon$-Tree. In fact, traditional buffer managers often are the main reason for scaling problems since pages are typically organized similarly [10, 16].

In order to increase multithreaded throughput, we could make use of buffer managers like LeanStore [26]. LeanStore uses pointer swizzling and thus enables the worker threads to directly map the identifiers of loaded pages to the respective location in memory without additional lookups. This could improve the performance of our trees in the third benchmark section, and we could compare the behavior of RocksDB and our B$^\varepsilon$-Tree more accurately.

9 Conclusion

In this thesis, we proposed a design for a B$^\varepsilon$-Tree on top of a 2Q page buffer. The B$^\varepsilon$-Tree offers concurrent access for both read and write operations. Since all write operations require exclusive access to the nodes, the tree makes use of a separate root node that is similar to the inner nodes of a B$^+$-Tree, splitting it into multiple subtrees. This enables the average worker threads to simultaneously access its respective node.

Flushes are performed with the main objective to release exclusive locks on higher node levels as early as possible. We therefore combined traversing the tree in level order with preemptive splitting of the nodes to enable exclusive lock coupling for the write operations.

For the evaluation, we analyzed the influence of ε on the performance. Additionally, we compared the B$^\varepsilon$-Tree design to a B$^+$-Tree that makes use of similar techniques as well as two LSM-Tree implementations, Google's LevelDB and Facebook's RocksDB.

Our evaluation showed that the B$^\varepsilon$-Tree performed write-only operations faster than the B$^+$-Tree both with and without the overhead of the 2Q buffer, regardless of the number of worker threads. Read-only workloads achieved a slightly lower throughput. While the performance of the B$^\varepsilon$-Tree scaled with multiple threads during all workloads, mixed workloads caused the B$^\varepsilon$-Tree scalability to drop significantly due to the conflicts of the lock-based node accesses.
Generally, the overhead caused by the 2Q buffer outweighed the performance of the B$^\varepsilon$-Tree when multiple threads were in use. At the same time, its throughput was still higher than that of LevelDB during all test runs. Regarding writes, while RocksDB beat our design with ≥ 3 threads, the B$^\varepsilon$-Tree still always outperformed RocksDB with one thread and sometimes even with two threads.

Although our design does not cover properties such as crash-resistance, we still believe that B$^\varepsilon$-Trees could be used to increase write throughput in key-value stores. Additionally, due to their similarities with traditional B$^+$-Trees, they can be easily built on top of preexisting buffer structures.

List of Figures

List of Listings

List of Tables

Bibliography

[1] A. Aggarwal and S. Vitter Jeffrey. "The Input/Output Complexity of Sorting and Related Problems." In: *Commun. ACM* 31.9 (Sept. 1988), pp. 1116–1127. ISSN: 0001-0782. DOI: 10.1145/48529.48535.

[2] M. A. Bender, M. Farach-Colton, W. Jannen, R. Johnson, B. C. Kuszmaul, D. E. Porter, J. Yuan, and Y. Zhan. "An Introduction to Bϵ-trees and Write-Optimization." In: *login Usenix Mag.* 40.5 (2015).

[3] B. H. Bloom. "Space/Time Trade-Offs in Hash Coding with Allowable Errors." In: *Commun. ACM* 13.7 (July 1970), pp. 422–426. ISSN: 0001-0782. DOI: 10.1145/362686.362692.

[4] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. "On External Memory Graph Traversal." In: *IN PROC. ACM-SIAM SYMP. ON DISCRETE ALGORITHMS*. ACM-SIAM, 2000, pp. 859–860.

[5] C. Chen, W. Zhong, and X. Wu. "Building an Efficient Key-Value Store in a Flexible Address Space." In: *Proceedings of the Seventeenth European Conference on Computer Systems*. EuroSys '22. Rennes, France: Association for Computing Machinery, 2022, pp. 51–68. ISBN: 9781450391627. DOI: 10.1145/3492321.3519555.

[6] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. "Benchmarking Cloud Serving Systems with YCSB." In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, pp. 143–154. ISBN: 9781450300360. DOI: 10.1145/1807128.1807152.

[7] N. Dayan, M. Athanassoulis, and S. Idreos. "Monkey: Optimal Navigable Key-Value Store." In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: Association for Computing Machinery, 2017, pp. 79–94. ISBN: 9781450341974. DOI: 10.1145/3035918.3064054.

[8] E. D. Demaine. "Cache-oblivious algorithms and data structures." In: *Lecture Notes from the EEF Summer School on Massive Data Sets* 8.4 (2002), pp. 1–249.

[9] S. Dong, A. Kryczka, Y. Jin, and M. Stumm. "RocksDB: Evolution of Development Priorities in a Key-Value Store Serving Large-Scale Applications." In: *ACM Trans. Storage* 17.4 (Oct. 2021). ISSN: 1553-3077. DOI: 10.1145/3483840.

[10] W. Effelsberg and T. Haerder. "Principles of Database Buffer Management." In: *ACM Trans. Database Syst.* 9.4 (Dec. 1984), pp. 560–595. ISSN: 0362-5915. DOI: `10.1145/1994.2022`.

[11] Facebook. *RocksDB: A Persistent Key-Value Store for Flash and RAM Storage*. 2022. URL: `https://github.com/facebook/rocksdb` (visited on 07/29/2022).

[12] Google. *LevelDB is a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values.* 2022. URL: `https://github.com/google/leveldb` (visited on 07/29/2022).

[13] G. Graefe. "A Survey of B-Tree Locking Techniques." In: *ACM Trans. Database Syst.* 35.3 (July 2010). ISSN: 0362-5915. DOI: `10.1145/1806907.1806908`.

[14] L. Grimmer. *Heads-Up: TokuDB Support Changes and Future Removal from Percona Server for MySQL 8.0*. May 2021. URL: `https://www.percona.com/blog/2021/05/21/tokudb-support-changes-and-future-removal-from-percona-server-for-mysql-8-0` (visited on 07/29/2022).

[15] S. Hardock, I. Petrov, R. Gottstein, and A. Buchmann. "From In-Place Updates to In-Place Appends: Revisiting Out-of-Place Updates on Flash." In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: Association for Computing Machinery, 2017, pp. 1571–1586. ISBN: 9781450341974. DOI: `10.1145/3035918.3035958`.

[16] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. "OLTP through the Looking Glass, and What We Found There." In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD '08. Vancouver, Canada: Association for Computing Machinery, 2008, pp. 981–992. ISBN: 9781605581026. DOI: `10.1145/1376616.1376713`.

[17] D. Hu, Z. Chen, J. Wu, J. Sun, and H. Chen. "Persistent Memory Hash Indexes: An Experimental Evaluation." In: *Proc. VLDB Endow.* 14.5 (Jan. 2021), pp. 785–798. ISSN: 2150-8097. DOI: `10.14778/3446095.3446101`.

[18] S. Iyer. *Comparing TokuDB, RocksDB and InnoDB Performance on Intel(R) Xeon(R) Gold 6140 CPU*. Aug. 2018. URL: `https://minervadb.com/index.php/2018/08/06/comparing-tokudb-rocksdb-and-innodb-performance-on-intelr-xeonr-gold-6140-cpu/` (visited on 07/29/2022).

[19] V. Jain, J. Lennon, and H. Gupta. "LSM-Trees and B-Trees: The Best of Both Worlds." In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD '19. Amsterdam, Netherlands: Association for Computing Machinery, 2019, pp. 1829–1831. ISBN: 9781450356435. DOI: `10.1145/3299869.3300097`.

[20] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter. “BetrFS: A Right-Optimized Write-Optimized File System.” In: *13th USENIX Conference on File and Storage Technologies (FAST 15)*. Santa Clara, CA: USENIX Association, Feb. 2015, pp. 301–315. ISBN: 978-1-931971-201.

[21] V. D. Jogi and A. Sinha. “Performance evaluation of MySQL, Cassandra and HBase for heavy write operation.” In: *2016 3rd International Conference on Recent Advances in Information Technology (RAIT)*. 2016, pp. 586–590. DOI: `10.1109/RAIT.2016.7507964`.

[22] T. Johnson and D. Shasha. “2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm.” In: *Proceedings of the 20th International Conference on Very Large Data Bases*. VLDB ’94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 439–450. ISBN: 1558601538.

[23] T. Kaldewey, A. Blas, J. Hagen, E. Sedlar, and S. A. Brandt. “Memory matters.” In: *RTSS’06* (2008).

[24] O.-Y. Khavrona. *B-epsilon-tree and cache-oblivious lookahead array: a comparative study of two write-optimised data structures*. July 2021. URL: `http://essay.utwente.nl/87368/` (visited on 07/29/2022).

[25] J. Kreps. *I Heart Logs: Event Data, Stream Processing, and Data Integration*. O’Reilly Media, 2014. ISBN: 9781491909331.

[26] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann. “LeanStore: In-Memory Data Management beyond Main Memory.” In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 2018, pp. 185–196. DOI: `10.1109/ICDE.2018.00026`.

[27] V. Leis, M. Haubenschild, and T. Neumann. “Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method.” In: *IEEE Data Eng. Bull.* 42.1 (2019), pp. 73–84.

[28] D. Lomet. “Cost/Performance in Modern Data Stores: How Data Caching Systems Succeed.” In: DAMON ’18. Houston, Texas: Association for Computing Machinery, 2018. ISBN: 9781450358538. DOI: `10.1145/3211922.3211927`.

[29] A. Mahajan. *Write Stalls*. 2021. URL: `https://github.com/facebook/rocksdb/wiki/Write-Stalls` (visited on 07/29/2022).

[30] Y. Matsunobu, S. Dong, and H. Lee. “MyRocks: LSM-Tree Database Storage Engine Serving Facebook’s Social Graph.” In: *Proc. VLDB Endow.* 13.12 (Aug. 2020), pp. 3217–3230. ISSN: 2150-8097. DOI: `10.14778/3415478.3415546`.

[31] D. Medjedovic and E. Tahirovic. “Algorithms and Data Structures for Massive Datasets.” In: Simon and Schuster, 2022, pp. 254–255.

[32] P. O’Neil, E. Cheng, D. Gawlick, and E. O’Neil. “The log-structured merge-tree (LSM-tree).” In: *Acta Informatica* 33.4 (June 1996), pp. 351–385. DOI: `10.1007/s002360050048`.

[33] OSCAR Lab. *Be-Tree*. 2017. URL: `https://github.com/oscarlab/Be-Tree` (visited on 07/29/2022).

[34] Percona. *Percona Server for MySQL*. 2022. URL: `https://github.com/percona/percona-server` (visited on 07/29/2022).

[35] Percona. *PerconaFT*. 2022. URL: `https://github.com/percona/PerconaFT` (visited on 07/29/2022).

[36] *Percona TokuDB - Documentation*. July 2015. URL: `https://www.percona.com/doc/percona-tokudb/index.html` (visited on 07/29/2022).

[37] *Percona TokuMX - Documentation*. July 2015. URL: `https://www.percona.com/doc/percona-tokumx/index.html` (visited on 07/29/2022).

[38] B. Stopford. “Power of the Log: LSM and Append Only Data Structures.” QCon. 2017.

[39] Unum. *UCSB*. 2022. URL: `https://github.com/unum-cloud/UCSB` (visited on 07/29/2022).

[40] A. Vardanian. *UCSB: Extending the Ultimate Yahoo NoSQL benchmark*. Mar. 2022. URL: `https://unum.cloud/post/2022-03-22-ucsb/` (visited on 07/29/2022).

[41] W. Zhang, Y. Xu, Y. Li, and D. Li. “Improving Write Performance of LSMT-Based Key-Value Store.” In: *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*. 2016, pp. 553–560. DOI: `10.1109/ICPADS.2016.0079`.