SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Reducing Write-Amplification in B-Trees

Marlene Bargou

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Reducing Write-Amplification in B-Trees

Reduzierung der Schreib-Verstärkung in B-Bäumen

Author:	Marlene Bargou
Examiner:	Prof. Thomas Neumann
Supervisor:	Prof. Thomas Neumann
Submission Date:	03.11.2025

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

Munich, 03.11.2025 Marlene Bargou

Acknowledgments

I thank my supervisor Prof. Thomas Neumann as well as Altan Birler for their inspiration, continuous support and guidance throughout this thesis.

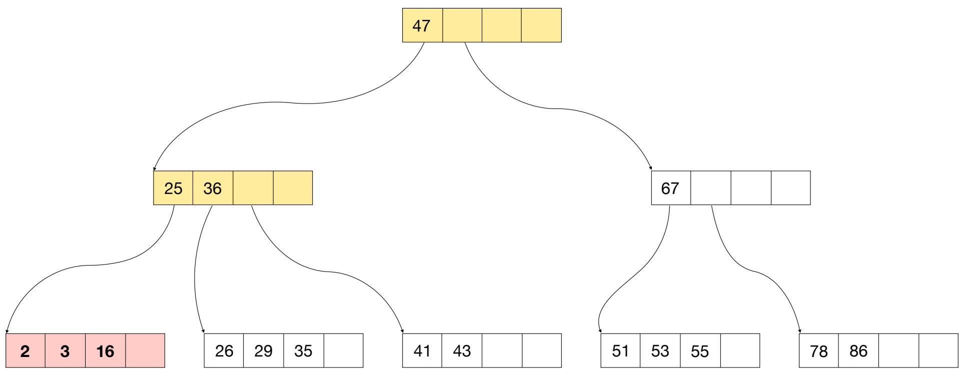
Abstract

B-Trees are the most used data structure in modern database systems, due to their efficient access patterns and excellent lookup performance on large volumes of data. However, B-Trees perform suboptimally under random writes, a particularly common pattern for secondary indexes. Such workloads introduce write amplification in B-Trees, a phenomenon where the amount of data written to storage is significantly larger than the amount of data that logically changed. As a result, B-Trees suffer increased latency, reduced throughput, and premature device wear with write-intensive workloads.
As an alternative, Log-Structured-Merge-Trees (LSM-Trees) were proposed, which trade off low read performance for high write performance. However, this trade makes LSM-Trees unsuitable for generic database systems that require excellent read performance. Other attempts to reduce write amplification in B-Trees either reduce concurrency, impact read performance or rely on hardware-specific features, limiting their effectiveness and applicability.
This thesis introduces a lightweight buffering layer that minimizes the frequency and volume of write operations to external storage by reducing write amplification. We hereby enable high performance under random writes, while sustaining all the benefits of traditional B-Trees.
We implement the proposed structure, evaluate its performance under different workloads, and compare it against state-of-the-art methods. Compared to LSM-Trees, our approach offers [...]. Compared to traditional B-Trees, our method achieves [...] while maintaining excellent read performance.
These results suggest that write-aware B-Tree optimizations can extend the lifespan of storage devices and significantly improve the efficiency of write-intensive applications; contributing to the broader effort of designing storage-efficient data structures suited for modern hardware.

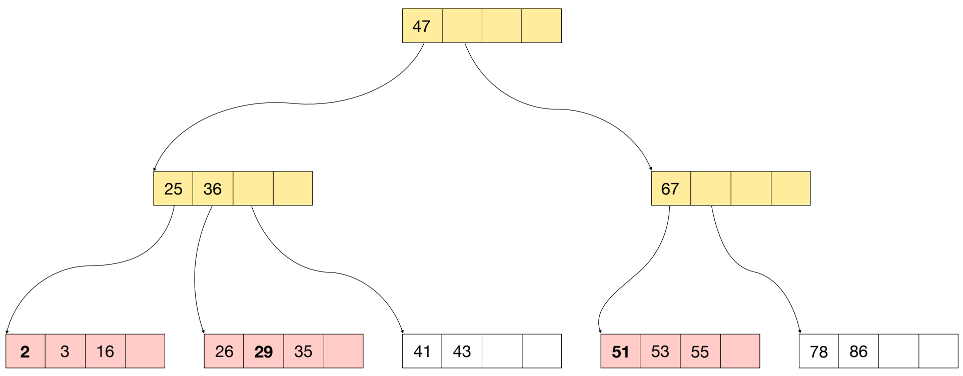
Contents

1. Introduction

Efficiently managing large data sets is a core requirement for database systems. Therefore, minimizing Input/Output (I/O) operations remains the fundamental premise for designing modern, high-performance Database Management Systems (DBMS). Primarily, this is done by caching frequently accessed pages in Dynamic Random Access Memory (DRAM) using a buffer manager [15]. All data in the system is stored in pages, which the buffer manager can cache and uniformly serve to all components in the system. This modular design allows a separation of concerns between the buffer manager and its users, such as indexes and data structures. However, this also means that the buffer manager is agnostic of its user's access patterns. While the buffer manager minimizes the number of I/O operations to the best of its knowledge, every component in the systems must design its access patterns to be as efficient as possible. A prominent example of such a component is the B-Tree [2]. B-Trees are the dominant data structure for indexing large datasets in disk-based DBMS due to their excellent lookup performance, support for range queries, and simplicity. However, random writes, a particularly common pattern for secondary indexes, lead to inefficient access patterns that a buffer manager cannot hide for out-of-memory workloads.
B-Trees organize their nodes as pages. Due to their sorted order, accessing random keys leads to random accesses of different pages that need to be loaded into the buffer. At eviction time, each modified page requires a full rewrite to storage, even if only a small portion of the page changed. Figure 1.1 illustrates this effect by comparing a sequential and a random update pattern in a B-Tree. We consider three updates to the tree. In the sequential update, only three pages are read and one of them is written to. In the random update, six pages are read and three of them are written to. Merely changing the access pattern from sequential to random leads to a threefold increase in the amount of data written to storage. If we assume that each page is 4 KB, the sequential update requires one storage write of 4 KB. The random update requires three storage writes of 12 KB in total. Random writes introduce *write amplification*, a phenomenon where the amount of data written to storage is significantly larger than the amount of data that logically changed.
Write amplification inflates I/O operations, wastes bandwidth, and ultimately increases latency in bandwidth-bound scenarios. For example in cloud environments, where storage can be remote, an unnecessary network round-trip directly translates to increased



(a) Sequential Access Pattern: Updating keys {2, 3, 16}.



(b) Random Access Pattern: Updating keys {29, 51, 2}.

Figure 1.1.: Comparison of access patterns in a B-Tree. Written pages are highlighted in red. Read pages are highlighted in yellow.

latency and reduced throughput in the system.
Additionally, the Solid State Drive (SSD) comes with its own, internal write amplification due to its flash translation layer performing garbage collection [11]. This leads to a multiplication of unnecessary physical writes, wearing out the device faster.
In summary, to design a truly efficient, high-performance system, we must minimize I/O operations in all components of the storage stack. In this thesis, we focus on closing the efficiency gap in B-Trees by reducing write amplification.

1.1. Problem Statement

While B-Trees are the backbone of indexing in modern storage engines, their in-place updates introduce significant write amplification, leading to performance degradation and reduced device lifespan.
LSM-Trees address write costs by always writing sequentially, but they introduce high read amplification and complex tuning requirements, making them unsuitable for general-purpose database systems.
Bϵ-Trees buffer and batch updates starting from the root and propagating them down the tree to reduce write amplification. Firstly, this introduce two searches per node, one for the next pivot and one for a buffered update for the looked up key. Secondly, the reduced space for pivots in each node reduces the fanout of the tree, leading to taller trees and more I/O operations per lookup. Most importantly though, it significantly limits concurrency in the data structure, as the hottest nodes are locked for longer periods of time to write the update messages, reducing throughput in the tree.
We identify a research gap for a B-Tree variant that effectively reduces write amplification while preserving the excellent query efficiency and concurrency traits of traditional B-Trees.

1.2. Objectives

The primary objective of this thesis is to design, implement and evaluate a B-Tree variant that reduces write amplification while maintaining the high read performance and concurrency of traditional B-Trees. We focus on the following research questions:

1. How can we effectively reduce write amplification in B-Trees?
2. How can we preserve read performance and concurrency in the presence of write optimizations?
3. How does the proposed approach compare to existing methods in terms of write amplification, query performance and throughput?

While we reflect on significant hardware trends in this thesis, such as the increasing prevalence of SSD, we do not target optimizations for specific hardware features. Instead, we aim to design a solution that is broadly applicable across different storage media and hardware configurations.
We also do not aim to outperform LSM-Trees in write-intensive workloads, as they are fundamentally optimized for such scenarios, trading off lookup performance.
While the page-oriented design is one reason for I/O amplification in B-Trees in general, we do not aim to redesign the data structure from the ground up. Instead, we focus on a lightweight extension to the traditional B-Tree that can be integrated into existing systems with minimal changes.

1.3. Contributions

This thesis introduces 3B-Tree, a B-Tree variant that incorporates a lightweight buffering layer to minimize write amplification. The buffering layer batches small write operations, reducing the frequency and volume of writes to external storage. When a B-Tree node is evicted from memory, we determine whether it has changed significantly enough to warrant a full write to storage. If not, we buffer the changes. Essentially, we minimize I/O operations to those strictly necessary.
The novelty of our approach lies in its non-intrusive design: We only perform additional operations when B-Tree nodes are exchanged between memory and external storage. In contrast to other approaches (see Chapter 3), we neither alter the B-Tree structure or its fundamental operations, nor do we impact concurrency or read performance in the tree. This makes our approach easy to integrate into existing systems and preserves the desirable properties of traditional B-Trees.
We hereby contribute to the broader effort of minimizing overhead of beyond memory systems and designing efficient, high-performance datbase systems for modern hardware.

2. Background

2.1. External Storage Characteristics

For some time, in-memory database systems like Hyper [12] have gained popularity due to the decreasing cost of DRAM. However, that trend has reversed recently, as DRAM prices have stagnated [11] and SSD price-performance-ratios have improved significantly [14]. Therefore, modern database systems are designed to operate efficiently on external storage and since index structures are the performance-critical component, out-of-memory indexing has become a key consideration again. B-Trees have been the dominant index structure for out-of-memory indexing, since their high fanout minimizes the number of I/O operations.
Historically, hard disks were the dominant storage medium. Hard disks have a significant imbalance in latency between random and sequential I/O due to their mechanical nature. While SSD have a smaller difference between random and sequential I/O, they still exhibit asymmetric performance, especially in writes [11]. Therefore, to amortize the cost of random I/O, database systems and their index structures are designed to access data in pages of multiple kilobytes instead of individual tuples. While we will be referencing disk-based systems throughout this thesis, we speak of systems operating on external storage, which can be either disk-based or flash-based.

2.2. Database System Architecture Overview

In the scope of this thesis we focus on a classic architecture of a single-node, disk-based database system. The access and storage layer of a database system typically consist of a buffer manager, one or more index structures and the slotted pages that store tuples identified by Tuple ID (TID)s, as illustrated in Figure 2.1. Since we operate in a beyond memory setting, the buffer manager is responsible for caching pages in DRAM and loading them from external storage when needed. Therefore, all components accessing physical data interact with the buffer manager to load and store their pages. When a query is executed, the index is accessed by a given key (e.g. the primary key) to find the TID of the relevant tuple. The index is typically stored in pages, which are loaded into the buffer pool by the buffer manager. Using the TID, the corresponding tuple can

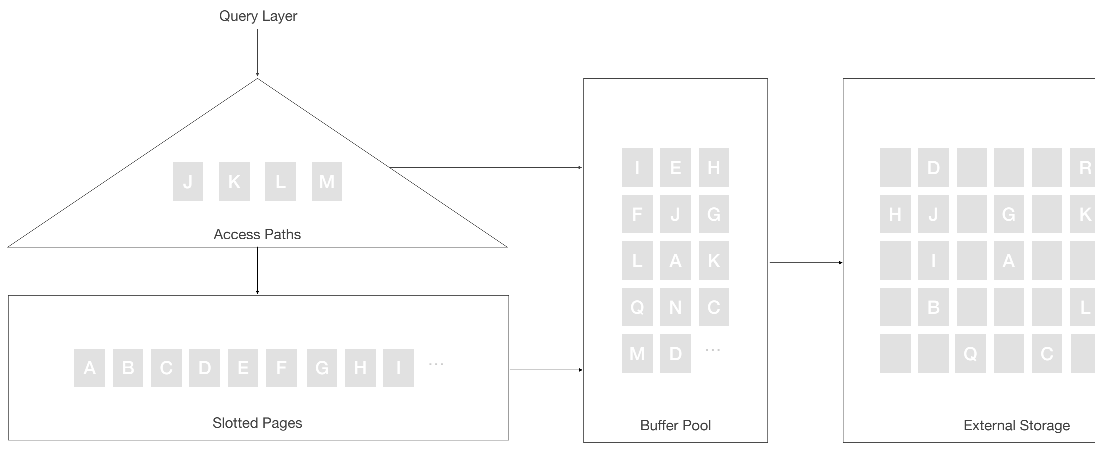


Figure 2.1.: The storage and access layer of a database system.

be retrieved from the slotted pages. The TID encodes the Page ID (PID) and the slot number within the page. When a tuple is updated, the corresponding page is loaded into the buffer pool, modified, and marked as dirty. Should the buffer pool be full, the buffer manager evicts pages based on its replacement policy. Clean, unchanged pages can be discarded, while dirty, modified pages must be written back to external storage.

2.3. Index Structures

Index structures are data structures that enable efficient access to data stored in a database. Typically, they map a key to a constant, unique TID. A TID never changes for a tuple. Keys can be arbitrary types and therefore of fixed or variable size, such as integers or strings. We will consider both within this thesis. When the key of a tuple changes, the index must be updated to reflect the new key.
Some key-value stores directly map keys to tuples within their index structure, omitting the indirection via TID and slotted pages. However, in a general purpose DBMS, we typically want to support multiple indexes on the same data. If we stored tuples directly in the index, we would need to update all indexes when a tuple changes. Therefore the access and storage layer are decoupled via TIDs.
Indexes can be classified into primary and secondary indexes. A primary index is built

on the primary key of a table, which uniquely identifies each tuple. A secondary index is built on a non-primary key, which can be non-unique.
Having sequential access to a primary key is common, for example when inserting new tuples with an auto-incrementing primary key. However, secondary keys are often accessed randomly. For example, consider a user's email address as a secondary key. When inserting a new user, the email address is likely to be random and not follow any specific order. Therefore, secondary indexes often exhibit random access patterns, which can lead to inefficient access patterns in traditional index structures like a B-Tree.

2.4. B-Trees

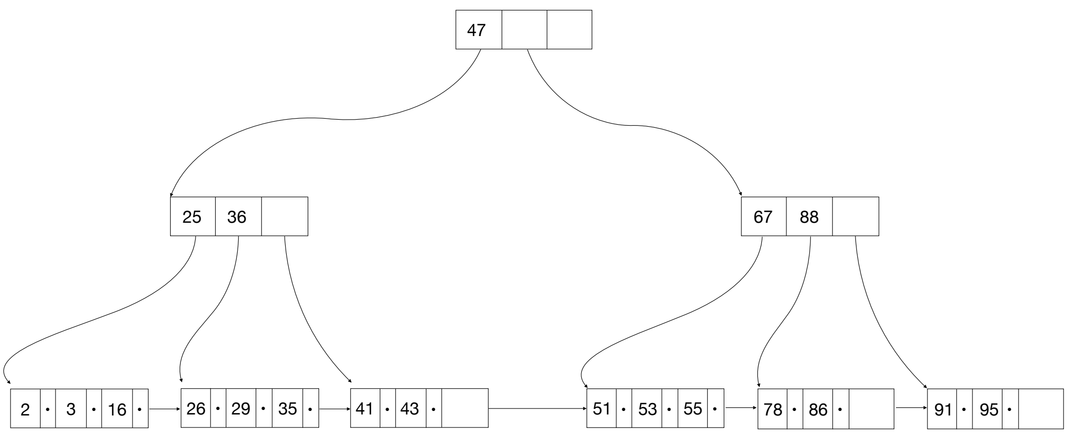


Figure 2.2.: A B+-Tree. Child pointers are repesented as arrows. Values (the TID) are represented as bullet points •.

B-Trees [2] are a self-balancing tree data structure that maintains sorted data and allows for insertion, deletion, and search operations in logarithmic time, $\mathcal{O}(\log n)$. A B-Tree is organized in fixed-size pages, called nodes. These pages are transferred and cached transparently by the buffer manager between external storage and DRAM. Each node can split off a sibling once it is full. If a node is full and a new key needs to be inserted, the node splits into two nodes, and the middle key is promoted to the parent node. Additionally, nodes can merge with a sibling if they become less than half full. For

simpliticity, we omit merging of nodes in this thesis.
The tree only increases in height when the root node splits. Each node contains between 2 and 2k entries, except for the root node, which can contain between 1 and 2k entries. Each entry is a triple of a key, a pointer to a child node, and optionally a value (the TID). The entries in each node are sorted by key. On leaf nodes (nodes without children), the pointer to a child node is undefined. An inner node (a node that is not a leaf) with k keys has k+1 children. Each entry in an innder node separates the key space of its children. The additional child pointer is necessary to separate the key space above the largest key in the node. For example, consider an inner node with keys {10, 20, 30}. The first child contains all keys less than 10, the second child contains all keys between 10 and 20, and the third child contains all keys between 20 and 30. The fourth child contains all keys greater than 30.
When searching for a key in the tree, we start at the root node. On each node, we perform a binary search to find the appropriate pivot key and follow the corresponding child pointer. We stop when we reach a node with the desired key.

B+-Trees

When addressing B-Trees in this thesis, we actually refer to B+-Trees, a variant of B-Trees where all values are stored in the leaf nodes and internal nodes only store keys and child pointers to guide the search. The separator keys in internal nodes may or may not occur in the data. An example B+-Tree is illustrated in Figure 2.2. The lookup procedure is the same as in a B-Tree, however we always traverse the full tree from root to leaf to find a key. Not only does this simplify the B-Tree logic, it also increases the fanout of inner nodes, leading to a lower tree height and therefore fewer I/O operations for lookups since less pages are involved in reaching the leaf level. Also, it allows for efficient range queries by scanning the leaf nodes in order. Due to its excellent lookup performance, support for range queries, and simplicity, B+-Trees are the dominant data structure for external storage [18].

Disk-Access-Model

To analyze the performance of B-Trees in a beyond memory setting, we use the Disk-Access-Model (DAM) [1] [13]. The model has two levels of memory: an internal memory of size M and external storage of infinite size. The storage device is organized in fixed-size pages, which are the units of data transfer between memory and storage and determine the size of nodes in a B-Tree. For simplicity of this analysis of the DAM for B-Trees, we assume that records are constant sized (An assumption that simplifies this explanation but does not hold in practice. Therefore we will **not** assume this in our

method and implementation.) and that nodes are always completely filled. When a database has N records, and the storage device has pages of size P, the B-Tree has a height of $\log_P(N/P)$, where inner nodes contain $O(P)$ children and leaf nodes contain $O(P)$ records.
Each lookup/insertion/deletion requires a traversal from the root to a leaf node, leading to $\mathcal{O}(\log_P(N/P))$ I/O operations. Since the majority of nodes in the tree are leaf nodes, we can assume that most inner nodes can be cached by the buffer manager. In that case, we would require only a single I/O per B-Tree operation.
In practice, B-Trees are often used to index variable-sized keys and values. Therefore, we will consider variable-sized records in our method and implementation. However, the DAM provides an approximation.

Node Size & Fanout

The node size (i.e. the page size) is a crucial parameter in the design of a B-Tree, as it affects the height of the tree. Larger nodes lead to more entries per node, increasing the fanout for inner nodes and decreasing the height of the tree. When we can address more children per node, we need fewer levels in the tree to address the same number of keys. Since every lookup/insert/delete operation requires a traversal from the root to a leaf node, fewer levels lead to fewer pages involved in the lookup. Thus, larger nodes lead to fewer I/O operations per lookup. Additionally, since we need fewer distinct pages, we induce less page management overhead in the buffer manager.
Large nodes are particularly beneficial for analytical, read-heavy workloads, which often perform large scans and are interested in large parts of the data. However, workloads that perform many updates and point queries, are sometimes only interested in a small portion of the page. As a result, larger nodes lead to more I/O amplification, as we read and write significantly more data than necessary to perform the operation. If subsequent operations follow a sequential access pattern, the I/O amplification is not a problem, as we will be operating on pages already cached by the buffer manager. Ideally, no or very few additional I/O operations are necessary. With random access patterns, however, this I/O amplification due to large page sizes becomes a problem.
While we will be evaluating performance under different node sizes, this tuning parameter is not the primary focus of this thesis. Instead, we focus on reducing I/O amplification in B-Trees at any page size. However, larger nodes are expected to profit more significantly from our approach, as they induce more I/O amplification in B-Trees.

2.5. Write Amplification

Write amplification is the ratio of the amount of data written to storage versus the amount of logical data written by the user. For example, if the database updates an entry of 64 B, but needs to write a full page of 4 KB to storage, the write amplification is 4096 / 64 = 64. Write amplification *WA* is formally defined as:

$$WA = \frac{BytesWrittenPhysically}{BytesWrittenLogically}$$

There are multiple layers of write amplification in a database system, which we need to differentiate from each other.

Application Layer. At the application layer, we consider an external end user interacting with the database system, e.g. through SQL. When an end user inserts a tuple through a query, the database system must insert the tuple in the table itself, in the Write-Ahead Log (WAL) and all indexes that serve to access the tuple later. Consequently, we write significantly more bytes than the user requested logically. Additionally, updating a B-Tree index might cause structural changes such as node splits or merges that create new nodes, delete new nodes and update the parent nodes. Thus, a database system inherently comes with write amplification merely to perform its purpose: manage data. This is however not the focus of this thesis; we consider all updates to tables, data structures and the WAL as necessary and focus on reducing write amplification within the index layer specifically.

Index Layer. At the index layer, we consider the B-Tree as the user of pages. When a B-Tree entry is updated, the bytes written logically include not only the updated key but also any additional metadata required to maintain the tree structure. This can include information about node splits, merges, and the promotion of keys to parent nodes. The writes are amplified by the B-Tree's page structure that requires rewriting the entire page even if only a small portion changed. As mentioned in Section 2.4, the node size directly impacts the write amplification at this level. This is the write amplification we focus on in this thesis, by minimizing the number of pages written to storage for a given set of updates (see Chapter 4).

Physical Layer. At the physical layer, we consider the database system as the user (the host) of the physical storage device. When the database system writes a page to storage, the bytes written logically are the size of the page. However, due to the characteristics of the storage device, the actual bytes written physically can be larger. SSD typically operate in larger units called blocks, which consist of multiple pages. When a page is updated, the entire block containing that page must be rewritten, leading to write amplification. Additionally, when garbage collection is performed, valid pages within a block must be copied to a new block before the old block can be erased and reused. Recent research observed write amplification factors up to 10x on modern SSD [10]. Consequently, a page write of 4 KB can lead to physical writes of up to 40 KB on the device, using up valuable bandwidth and wearing out the device faster. While we do not focus on hardware-level write amplification in this thesis, it shows the importance of reducing write amplification at the host level. Unnecessary writes by the database system are multiplied by SSD.

2.6. Read Amplification

Read amplification is the number of I/O operations required to answer a query [13]. As described above, a B-Tree lookup requires a traversal from the root to a leaf node, which is $\mathcal{O}(\log_P(N/P))$ I/O operations in the DAM, assuming that our cache is cold. In practice, the buffer manager caches pages in DRAM, which can significantly reduce the number of I/O operations. To answer range queries, we can scan the leaf nodes in order, which is efficient in B+-Trees. Therefore, we only traverse the tree once to find the start of the range and then scan the leaf nodes sequentially. Read amplification is not the focus of this thesis, but it will help understanding the tradeoffs of alternative data structures in Chapter 3.

2.7. Space Amplificaiton

Space amplification is the ratio of the amount of space used by the data structure versus the amount of logical data stored [13]. Most nodes in a B-Tree are leaf nodes, which store the actual data. However, inner nodes only store keys and child pointers to guide the search, inflating the space usage. Additionally, B-Tree nodes are not always completely filled. Therefore, B-Trees exhibit some space amplification.

3. Related Work

In this chapter, we review related work on write-optimized data structures, focusing on B-Tree variants and alternatives and compression techniques to reduce write amplification. For each data structure, we discuss its design, how it addresses write amplification, and its trade-offs in terms of read performance, concurrency and applicability. We outline the gap in existing work that we aim to address in this thesis.

3.1. Log-Structured-Merge-Trees

LSM-Trees [19] are a popular alternative index data structure to B-Trees for write-heavy workloads. They are increasingly used in key-value stores such as RocksDB at Meta [22] or BigTable at Google [6].

Basic Structure. LSM-Trees consist of two main components: an in-memory component and a disk-based component. The in-memory component is typically implemented as a balanced tree such as a red-black tree, called a MemTable. The MemTable accepts and applies updates in memory. Once it is full, it is flushed to disk as a sorted, immutable runs in files called SSTables. Over time, multiple runs accumulate on disk. Since those runs may have overlapping key ranges, lookups need to check both memory and multiple disk files to find a key.
To limit the number of runs on disk and improve lookup performance, LSM-Trees organize runs into multiple levels, where each level is larger and more data is sorted than in the previous one. When a level reaches its size limit, it triggers a compaction process to sort-merge runs into the next level, retaining only the latest version of each key. As a result, higher levels contain more recent data with several, smaller files with overlapping key ranges while lower levels contain few, large files with non-overlapping key ranges. Since runs are immutable, each compaction generates new files for the merged runs. Outdated files are deleted by a garbage collector [21].

High Write Performance. B-Trees maintain a fully sorted view of the data and update this view in-place. In contrast, LSM-Trees update out-of-place in a sequential, log-structured manner by buffering updates in memory and flushing them to external storage in large, sorted batches, enabling high write throughput. The excellent write performance of an LSM-Trees makes this data structure suitable for write-heavy workloads, such as time-series data or logging systems.

Low Read Performance. Essentially, LSM-Trees trade high write performance at the cost of low read performance. This is useful for specific scenarios where writes dominate reads. However, it makes LSM-Trees unsuitable for general-purpose DBMS as they incur significantly higher lookup costs compared to a B-Tree as shown in [7].
When performing point lookups, the LSM-Trees checks the MemTable first and then each level on disk from top to bottom until it is found or not. In use-cases, where we only lookup hot keys that are likely to be in memory, LSM-Trees can perform well. However, such temporal locality is an assumption that we cannot make in a general-purpose system that needs to balance performance for all use-cases. When looking up cold keys that are not in memory, LSM-Trees need to check multiple files on disk, leading to high read amplification.
To improve lookup performance, each SSTable has an in-memory Bloom filter to check if a key is present in the file before performing a search. However, Bloom filters come with other problems. For one, it can yield false positives. Secondly, the larger the data set they are addressing, the larger the Bloom filter needs to be, inflating the memory footprint of LSM-Trees.
Most importantly though, Bloom filters cannot handle range queries. For range queries, all SSTables across levels must be checked. While there is an effort to improve range query performance in LSM-Trees [24], they are not designed for efficient range queries, as range data is scattered across the tree [21].

Summary. Overall, both LSM-Trees and B-Trees are efficient data structures, but built for different scenarios. This update/query trade-off has been well studied in literature [4]. In this thesis we focus on general-purpose database systems, which require balanced performance characteristics across-the-board. For such a system, B-Trees are the superior data structure. We therefore investigate how to improve B-Trees to close the gap in write performance to LSM-Trees while retaining their superior read performance.

3.2. B$^\epsilon$-Trees

Basic Structure. B$^\epsilon$-Trees [3] are a write-optimized variant of B-Trees. Each internal node has a buffer to temporarily encode incoming updates as messages. When a buffer is full, messages are flushed to the appropriate child node. When messages reach a leaf node, they are applied to the respective leaf. Deletes are handled as tombstone messages that mark a key as deleted. Only when the message reaches the leaf, the

key-value pair is removed from the leaf. Each message encodes a timestamp to ensure that the updates are applied in the right order.
The ϵ, which is a value between 0 and 1, refers to the tunable parameter that controls the size of the buffers in relation to the node size. Given a page size B, it determines how much of its space is used for storing pivots (B^ϵ) versus buffering updates ($B - B^\epsilon$). Choosing a larger ϵ increases the space for keys and pointers, improving read performance similar to a B-Tree, while a smaller ϵ increases the buffer size, enhancing write performance similar to a buffered repository tree [5].

Mitigation of Write Amplification. This design allows B$^\epsilon$-Trees to batch updates, reducing the number of I/O operations and improving write performance while maintaining comparable read performance to B-Trees. A benefit of using a top-down approach to propagate updates is that it primarily writes to higher levels of the tree which are more frequently accessed and thus more likely to be cached in memory. Alongside with a good eviction strategy, this can effectively reduce number of write operations to external storage. Another effect of this design it that is allows for large node sizes. For one, we need larger node sizes to accommodate the buffers and maintain a high fanout. But more importantly, batching updates mitigates write amplification. At the time of reaching a leaf node to apply updates, many updates have accumulated and can be applied at once. A leaf node will not be rewritten for individual updates. Therefore, the larger node sizes are less problematic in B$^\epsilon$-Trees, since they do not incur as much write amplification as in B-Trees.

Read Overhead. Messages are usually binary search trees like a red-black tree to allow efficient searching within the buffer. When searching for a key, the tree is traversed from the root to the leaf, checking each buffer along the path for messages that belong to the key. This ensures that the most recent updates are considered during the search. However, this also means that two searches are required per node: one for the pointer to the child node and one for messages in the buffer. This introduces some overhead for read operations compared to B-Trees.
On the other hand, B$^\epsilon$-Trees can achieve faster scans, because larger node sizes are more attractive in this design, better utilizing the bandwidth of external storage.

Concurrency Limitation. Since updates are propagated top-down, we introduce contention on higher levels of the tree. However, higher levels of the tree are more frequently accessed to locate entries. When they are written to, this blocks a large amount of nodes below. This is especially problematic for the root node, which needs to be accessed by every operation in the tree, limiting concurrency in the system significantly.

Summary. While B^ϵ-Trees have been shown to effectively mitigate write amplification in a single-threaded scenario, they significantly limit concurrency in the data structure. A characteristic that makes B^ϵ-Trees unsuitable for high-performance database systems. In this thesis, we aim to reduce write amplification in B-Trees while retaining high concurrency.

3.3. Bw-Trees

Basic Structure. The Bw-Tree [17] is a B-Tree variant optimized for modern hardware. It introduces a latch-free design, leveraging atomic compare-and-swap operations to ensure consistency without traditional locking mechanisms. They employ out-of-place updates, where deltas are prepended to nodes as linked lists instead of modifying them in place. This avoids cache invalidation, enabling higher concurrency in the tree. To update the delta chain, they use atomic Compare-And-Swap (CAS) operations to allow latch-free updates. The delta chain of a node is eventually consolidated, by creating a new node that applies the deltas to the base node. Outdated base nodes are reclaimed by a garbage collector. Additionally, they employ a log-structured store that migrates nodes to contiguous storage locations. While they specifically target flash-based storage, the design principles apply to other storage media as well.

Mitigation of Write Amplification. When a page with a delta-chain is flushed to external storage, only the new deltas need to be written, not the entire page. This effectively reduces write amplification, as they only write the changes instead of the whole page. The deltas of several pages can be consolidated in memory, allowing to batch writes to external storage. Only when creating new pages during consolidation, they need to write the entire page. In that case, the node has experienced sufficient modifications that justify writing the entire page.

Read Overhead. The delta chain needs to be traversed for every single node on the read path to a leaf, introducing overhead for every lookup. While the goal is to keep cache lines valid, applying deltas out-of-place and traversing a linked list of deltas per node, pollutes the caches of every core. When loading a page from external storage, the entire delta chain needs to be read and applied to reconstruct the current state of the node. This requires multiple random read operations on storage.

High Coupling. This design introduces invasive changes to the B-Tree's implementation, requiring a change in lookup and update logic as well as a consolidation mechanism and garbage collection. Most importantly, it heavily couples the cache management layer with the indexing layer. For example, the indirection via the mapping table from PIDs to physical addresses becomes a requirement to implement the CAS logic to update the delta chain. The data structure needs to be aware of the storage layer and their implementation details to implement this logic. This makes changes to the caching layer difficult. For example, pointer swizzling [9] would be infeasible with this design. Everytime a delta is prepended to a node, the swizzled pointers would become invalid. Updating each pointer to the new root of the delta chain would require updating all outdated pointers in the tree. However, pointer swizzling is a common technique for disk-based database systems [15] to compete with in-memory database systems. Therefore, we want to keep each layer transparent to the other, allowing independent optimizations.

Summary. Overall, the Bw-Tree presents a novel approach to reduce write amplification in B-Trees and we take notes for our own design. However, Wang et al. showed in their paper "Building a Bw-Tree Takes More Than Just Buzz Words" [23] that the Bw-Tree's performance is actually not competitive with traditional B-Trees using optimistic lock coupling [16]. In our approach, we aim to introduce a small overhead when loading a page from external storage, not for every read operation in memory. Additionally, we want to keep the changes to the B-Tree minimal, introducing a lightweight layer between the data structure and the storage manager that buffers and batches updates.

3.4. Write-Optimized B-Trees

In his paper "Write-Optimized B-Trees" [8], Goetz Graefe addresses the write efficiency gap between log-structured file systems and the B-Tree. Log-structured data structures write large, sequential chunks of data to disk, making optimal use of the available bandwidth. B-Trees, on the other hand, perform many small, random writes by writing individual pages. To improve write efficiency in B-Trees, Graefe proposes to batch multiple dirty nodes and write them to disk in a single, large write operation.
The buffer manager can invoke such page migrations for the chosen dirty pages. By introducing logical fence keys, the pages can be written to arbitrary locations, without requiring an update to the sibling pointers. Since page migrations are optional, the B-Tree can still decide to update pages in place if that is more efficient.
This work provides an approach to get the log-structured style of writing in a B-Tree, without changing the data structure itself. It supports the effort of improving write performance in B-Trees, while retaining their high read performance and concurrency. While this approach improves write performance by batching multiple pages, in this thesis we address the write amplification caused by individual page and in-place

updates. We avoid writing individual pages as a whole, by deferring updates and buffering them in batches instead.

3.5. Transparent Compression

Some modern storage devices offer hardware-based lossless data compression transparent to the host. The storage device compresses data before writing it to the physical medium. When a page is empty, no data is written at all. When a page is partially filled, only the actual data is written, not the empty space. This can be used to reduce write amplification in B-Trees, as it allows for sparse data structures which do not actually waste space on the storage device.
In the paper [20], apply out-of-place updates like previous approaches. Each node's page is followed by a modification log that records changes to the node. In contrast to previous approaches of out-of-place updates, they do not need to collect updates to a node across storage but instead, they can perform a single read operation to load the node and its modification log. When loading the node from storage, they apply the modifications in the log to reconstruct the current state of the node. When a node is modified, they append the modification to the log instead of rewriting the entire page. When a node is flushed to storage, they obtain the delta, and decide whether they invoke the page modification logging which is appended to the page or whether they write the entire page in-place.
This approach reduces write amplification in B-Trees, as they avoid rewriting entire pages for small updates. However, for small deltas they still need to perform an I/O operation to write the modification log to storage. The purpose of reducing write amplification is not only to reduce the amount of data written, but primarily to reduce the frequency of writing data by batching updates. We want to avoid writing at all for small updates.
A benefit of this approach is that it does not require invasive changes to the B-Tree structure itself. In memory, the B-Tree remains the same and only the storage manager needs to be aware of the modification log. This only requires overhead at the point of loading and unloading a page to and from external storage. A similar approach is taken in this thesis.
However, this approach relies on hardware-based compression which is not widely available. We aim to provide a software-based solution that can be used on any storage device.

4. Method

4.1. Design Goals

Chapter 3 reviewed existing approaches to reduce write amplification and their trade-offs. We disclose the gaps in existing work that we aim to address in this thesis and identify the following design goals for our approach to reduce write amplification in B-Trees.

Reduce Write Amplification

Write Amplification in B-Trees is primarily caused by the page-oriented design that requires rewriting entire pages to storage even if only a small portion changed. This page-oriented design is crucial to utilize bandwidth of modern storage devices and to minimize buffer management overhead. Merely reducing write amplification by reducing the amount of data written to storage is not the goal. Instead, high write amplification suggests that we perform unnecessary writes to storage, which we would like to avoid. We aim to mimimize the number of I/O operations induced by the B-Tree to perform a set of updates. The overall goal is to create a B-Tree variant that is more write-efficient regardless of the access pattern.

Maintain Read Performance

B-Trees are widely used in general-purpose database systems due to their excellent read performance for point lookups and range scans. As we showed in Chapter 3, many write-optimized data structures sacrifice read performance. For example the LSM-Trees incurs high read amplification to achieve high write-efficiency. The Bw-Tree introduces a delta chain to each node that needs to be traversed on every node read. The B$B\epsilon$-Tree introduces an extra binary search for every node on the search path. In contrast, we aim to keep the overhead of our optimizations minimal and preserve read performance of B-Trees.

Maintain Concurrency

B-Trees are designed for high concurrency, allowing multiple threads to perform operations simultaneously. Many write-optimized data structures compromise concurrency to achieve high write-efficiency. For example, the Bϵ-Tree introduces longer locking periods of the most frequently accessed nodes, reducing throughput in the tree. In contrast, we aim to keep our optimizations outside of the data structure itself to not compromise concurrency of B-Trees.

Maintain Simplicity

B-Trees are widely used in practice due to their simplicity. Some write-optimized data structures, like the B-ϵ-Tree, introduce significant complexity to the data structure itself. In contrast, we aim to keep our optimizations lightweight, allowing for easy integration into existing systems. The changes we introduce to the B-Tree itself should be minimal. We aim to keep a low coupling between the data structure and the storage manager, allowing for optimizations in both layers independently. Neither do we require special hardware features, making our approach broadly applicable across different storage media and hardware configurations.

4.2. High-Level Description of the Data Structure

We avoid writing a page to storage if changes are small. To achieve this, we introduce a Delta Tree that acts as a hesitation layer, as illustrated in Figure 4.1. When evicting a dirty page, we can buffer the changes of the B-Tree pages instead of writing them to storage immediately. We can just discard the page, saving us the write to storage. When loading a page from storage, we apply all buffered changes to it before returning it to the B-Tree. Only when enough changes accumulate, we write the full page to storage.

Reduce Write Amplification. Write amplification in a B-Tree happens at the point of evicting a page to storage. The buffer manager is only aware that a page is dirty and needs to be written to storage. It does not know which parts of the page changed. Therefore, we keep track of the modifications made to each page in the B-Tree. Our component interacts with the buffer manager to intercept the eviction of dirty pages. When evicting a dirty page, we can buffer the changes in the Delta Tree instead of writing them to storage immediately. This way, we can defer small random writes until we have enough changes to justify a full rewrite to storage. In the Delta Tree we can batch changes to the same page.

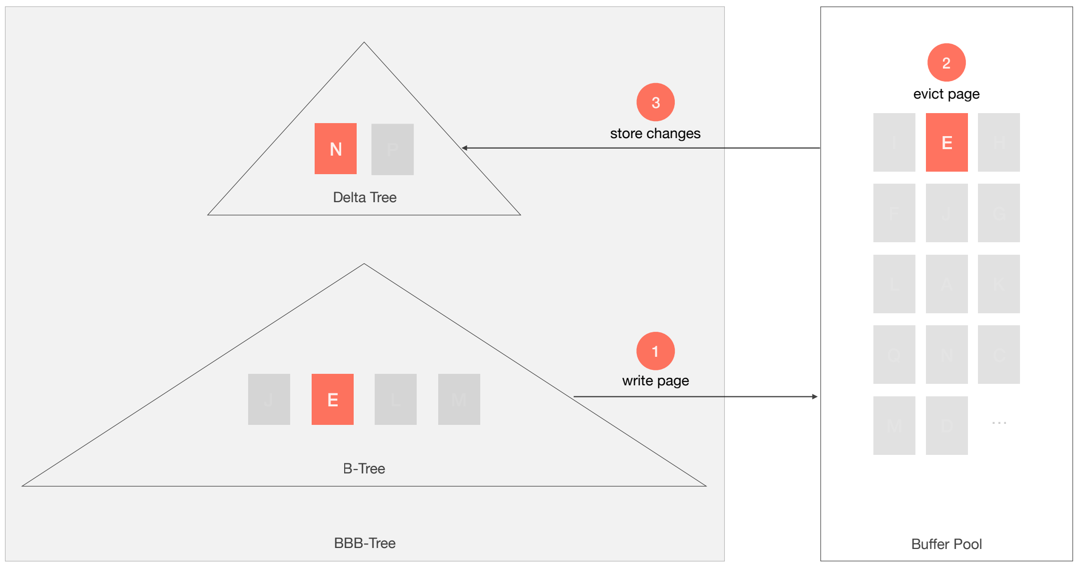


Figure 4.1.: High-level architecture of the data structure. We add a Delta-Tree to a B-Tree. When evicting a dirty page from the B-Tree, we can buffer the changes of the B-Tree pages instead of writing them to storage. When loading a page from storage, we apply all buffered changes to it before returning it to the B-Tree.

Maintain Read Performance. To nodes in memory, this method is transparent. When searching pages in memory, we do not incur overhead.
The only time we incur overhead is when loading a page from storage or unloading it to storage. We argue that this overhead is acceptable, as we are already loading a page from storage. The overhead of applying the buffered changes is small compared to the I/O operation.
However, the buffering layer itselt is a disk-based data structure, which migh require I/O operations to look up buffered changes. We will be analyzing the overhead in Chapter 6.

Maintain Concurrency. The Delta Tree is separate from the B-Tree. Therefore, we do not compromise concurrency of the B-Tree itself, since we do not introduce further locking in the B-Tree itself. The Delta Tree can be implemented as a B-Tree variant itself, allowing for high concurrency.

Maintain Simplicity. The modifications we introduce to the B-Tree itself are minimal. Within the B-Tree, we only need to track the modifications made to the nodes. Essentially, we mark entries as dirty.
The Delta Tree is a separate component that interacts with the buffer manager. We do not dictate how the buffer manager is implemented or how it manages pages. As a result, we maintain a low coupling between the data structure and the storage manager, allowing for optimizations in both layers independently. For example, pointer swizzling is an optimization that could be applied with our method.
The Delta Tree itself is a B-Tree variant, which we already have present in our system. Therefore, we can reuse existing code and concepts, reducing implementation complexity. Neither do we assume any special hardware features, making our approach broadly applicable across different storage media.

4.3. Data Structure Modifications

Buffer Manager

The buffer manager is the component that decides when a page is evicted from memory and when a page is loaded into memory. At the same time, the buffer manager should not be aware of any semantics of its pages. More specifically, it does not know if it is evicting or loading pages of a B-Tree or any other data structure. However, we require specific logic to be executed when evicting or loading pages of a B-Tree. We need a way to inject this logic into the buffer manager without leaking B-Tree specific logic into the buffer manager itself. Therefore, users can register function pointers that are invoked at eviction time and loading time. That way, the buffer manager remains agnostic of the semantics of its pages.

B-Tree

1. **Tracking Write Amplification:** We need to be aware of the degree of write amplification per node. Whenever we modify a node, for example through an insertion or a node split, we keep track of the amount of bytes that were changed. Then, in relation to the page size, we can determine the write amplification of the node. Based on that parameter, we can decide if a write operation to external storage is justified, or if we want to defer it.

2. **Tracking Deltas:** We need to determine the changes that occurred on a node since the last time it has been loaded from external storage. To that end, each entry on a node has an additional "state" field that indicates if the entry was inserted, updated, or deleted since the last time the node was loaded from external storage. That way, we can buffer the "delta" image of a node at eviction-time and apply it again at loading time to ensure that we can reconstruct the logical state of a node when it is accessed again at a later point in time.

3. **Injecting Callbacks:** As described above, we need to execute specific logic at eviction time and loading time of a B-Tree page. However, the B-tree has no control over the point in time a page is evicted or loaded. Therefore, we inject callbacks into the buffer frames that are later invoked by the buffer manager. Whenever we request a B-Tree page from the buffer manager, we register function pointers for the Delta Tree. At eviction- and loading-time, these function pointers are called by the buffer manager to execute the necessary logic.

Alternatively, we could have immediately inserted changes into the Delta Tree whenever a change occured on a B-Tree node. This way, we would not need to track changes on the B-Tree nodes themselves. However, this would introduce significant overhead on every write operation to the B-Tree. Should a node be changed multiple times while it is in memory, we would need to update the Delta Tree multiple times as well. In that case, we have many updates to a B-Tree node and therefore want to perform the write to storage anyway. We would have introduced the most overhead for situations without any benefit. Therefore, we only interact with the Delta Tree at eviction- and loading-time of a B-Tree page instead. Only if we decide to buffer changes, we insert them into the Delta Tree. This way, we keep the overhead for B-Tree operations minimal.

Delta Tree

The Delta Tree is the component that buffers changes of B-Tree nodes. The Delta Tree itself is a B-Tree with PIDs as keys and lists of changes as values. The buffer manager calls back the Delta Tree everytime a dirty B-Tree page is evicted from memory or loaded into memory.

1. **Eviction Time:** The Delta Tree can decide if a B-Tree page should be written to storage or not based on the write amplification of the page. Should it decide to not write the page to storage, it buffers its changes. It does so, by scanning the node for entries that were marked as dirty and inserting them into its own B-Tree. The buffer manager is informed that the page does not need to be written to storage anymore and can simply be discarded.

   Should it decide to continue writing the page to storage, it simply returns and the buffer manager writes the page to storage as usual. In this case, we clean the state of the B-Tree page and remove any buffered changes from the Delta Tree, as it is now in sync with storage.

2. **Loading Time:** When loading a B-Tree page from storage, the Delta Tree looks up if there are any buffered changes for that page. If so, it applies the changes to the page before returning it to the B-Tree. Together with the state of the page on storage, we can reconstruct the state of the page in memory.

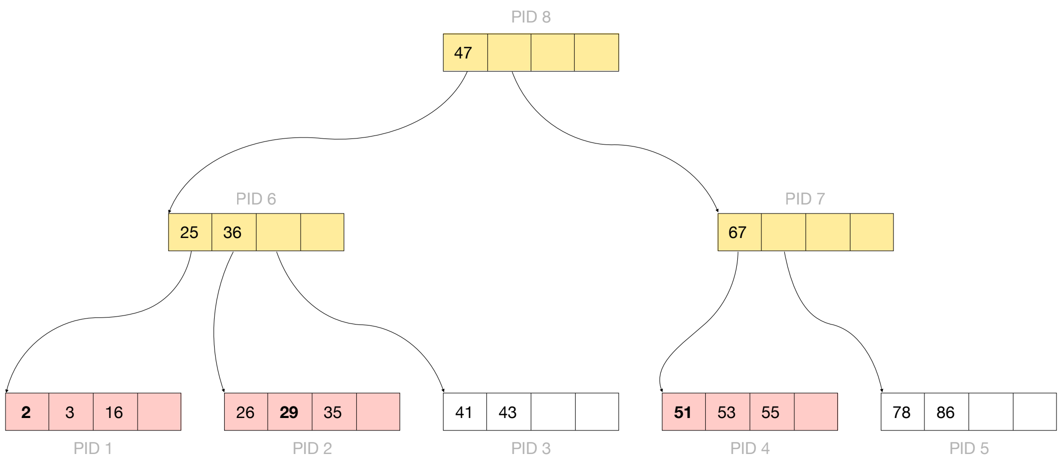
   When changes were applied, we keep them in the Delta Tree, as they might be useful for future evictions. This way, we perform less updates to the Delta Tree and can keep the state of the B-Tree page clean. Should there be no further changes to that page, the next eviction can simply discard the page without any writes.

The Delta Tree itself contains pages that are managed by the buffer manager. Its pages can be evicted to storage as well. Therefore, we want to keep the Delta Tree small to batch changes more effectively.

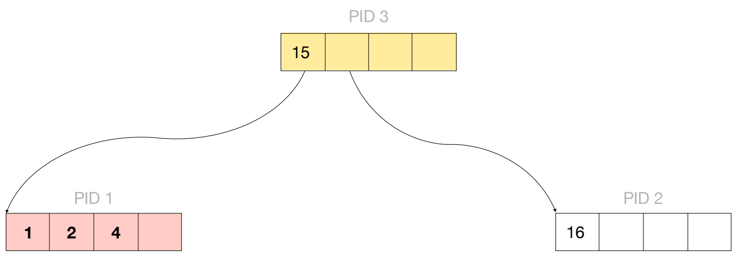
4.4. Implications on Write Amplification

We can illustrate the effect of our approach on write amplification with a simple example. Figure 4.2 shows a B-Tree with updates across three nodes with PIDs 1, 2, 4. In a traditional B-Tree, we would need to write all three pages to storage. Assuming a page size of 4 KB and each update being 64 B, the write amplification is $(4096B/64B) * 3 = 192$.

With our approach, we can buffer the changes of the three pages in the Delta Tree. In this example, the Delta Tree only needs to write one page to storage, containing the three updates. Assuming a page size of 4 KB again and approximating the delta entries to be 64 B each, the write amplification is now $4096B/(3 * 64B) = 21.33$. In this simple example, we have reduced the write amplification by a factor of >9.
This demonstrates the goal of our approach: batching small writes on many distinct pages into larger writes on fewer pages. This way, we can reduce the number of pages written to storage for a given set of updates, thereby reducing write amplification.



(a) A B-Tree with updates to nodes with PID 1, 2, 4.



(b) The corresponding Delta Tree. Buffered the changes of B-Tree nodes by their PIDs 1, 2, 4.

Figure 4.2.: Example of a B-Tree and its corresponding Delta Tree. The B-Tree has three updated nodes marked in red. In this example, we require only one page write for the Delta Tree instead of three writes for the B-Tree.

5. Implementation

5.1. System Architecture

We have implemented our system according to Figure 2.1 in C++. In the following, we inspect the particular implementation of each component.

Buffer Manager

Our buffer manager serves our system with pages, transparently swapping them between memory and external storage. Upon construction, it receives a `page_size` that determines the fixed-sized number of bytes of every page in the system, as well as a `page_count` that determines the maximum number of pages that can be buffered in memory. Each component requests a page with `fix_page` returning a buffer frame. After operating on the page, the page is released again by calling `unfix_page` on the given frame. The user can pass a boolean flag to indicate whether the page was modified or not.
When fixing a page we can pass a pointer to a `PageLogic` object. `PageLogic` is an abstract class that can be defined by the user to inject user-specific logic into the buffer manager. This object will be called by the buffer manager when a page is loaded from external storage to memory or evicted from memory. This allows us to insert user-specific logic, such as invoking the Delta Tree upon eviction, without coupling the two components.
If a page is not already present in the buffer pool, it is loaded from a file from storage. If a `PageLogic` object is injected into the page's corresponding frame, we call it to perform user-specific logic on the loaded page. When the user unfixes the page again, we keep the page in the buffer pool until it is chosen for eviction.
When the buffer pool is full, our buffer manager selects a page for eviction. The eviction strategy is not under inspection in this thesis, therefore we choose a page at random. Should the page be marked dirty or new, we call the `PageLogic` object. Should it return true, we continue writing the page to storage. Should it return false, we do not continue with the write and simply discard the page.

Slotted Pages

We store tuples within pages, accessed through the buffer manager. As shown in Figure 5.1, a slotted page consists of a header, a slot array and a data segment. The header contains metadata about the page, such as the number of slots and the pointer to the data segment. Each slot points to the corresponding tuple data stored in the data segment. Through this indirection we can accommodate for variable-length tuples [18]. Introducing this indirection impacts the cache locality as we need to follow an additional pointer for every comparison. There are some optimizations, such as storing parts of the key in the slot itself to speed up comparisons [9].
Each tuple in the system is identified through its unique TID. Each TID consists of 8 B, whereas the upper 6 B contain the PID and the lower 2 B contain the slot's ID.
When looking up a tuple in the system, we retrieve the corresponding TID from the index given the key. Through the TID we can request the corresponding page through the PID from the buffer manager. When loaded into memory, we can access the slot through the slot ID and retrieve the actual tuple data.
When inserting a new tuple, we first need to find an appropriate page to store it in. If the buffer pool is full, we need to evict a page before we can load a new one. Once we have a page, we can allocate a slot for the new tuple in the page's slot array and store the tuple data in the data segment. We then create a new TID for the tuple and insert it into the index.

B-Tree

As shown in Figure 5.2, each node in our B-Tree is implemented similar to a slotted page to accommodate variable-sized keys and values. While we depict a leaf node in the figure, inner nodes are implemented similarly. Leaf nodes store keys and values, whereas inner nodes store keys and PIDs to child nodes.
We template our B-Tree implementation on the key and value type. A third boolean template parameter indicates whether we require tracking information to operate a corresponding Delta Tree for this B-Tree instantiation.

BBB-Tree

A BBB-Tree consists of a B-Tree with tracking information and a corresponding Delta Tree.

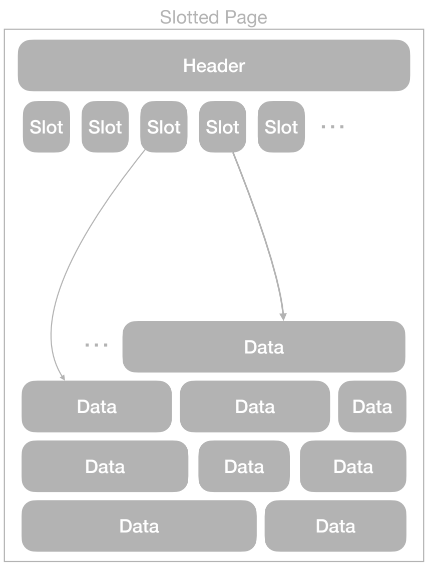


Figure 5.1.: The data layout of a slotted page. The header contains metadata about the page, such as the number of slots and the pointer to the data segment. Each slot points to the actual tuple data stored in the data segment. Tuples can be of variable length and are accessed through their TID. Adapted from "Database Systems on Modern CPU Architectures" [18].

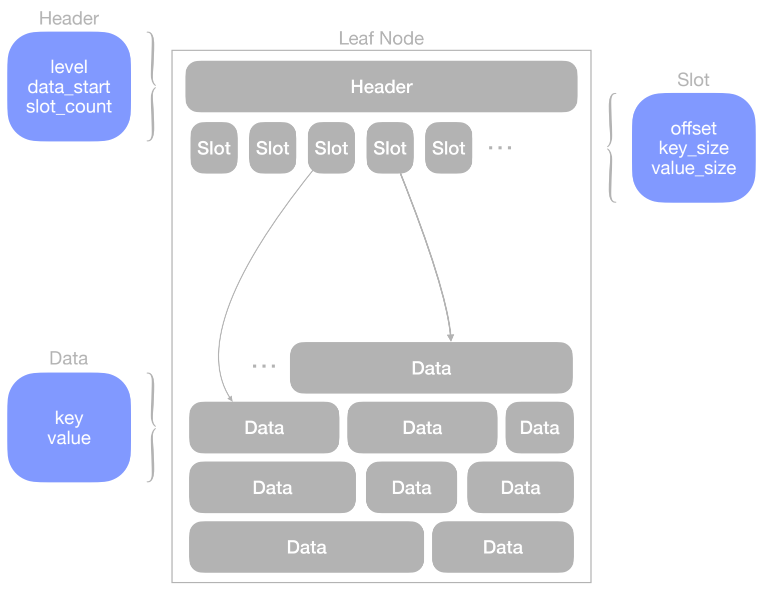


Figure 5.2.: The data layout of a leaf node in the B-Tree supporting variable-sized keys and values. The header contains metadata about the node, such as the level in the tree, the offset to the data segment and the number of slots. The slots point to the actual entries stored in the data segment. The data segment contains the keys and values. Inner nodes store keys and pointers to child nodes. Adapted from "Database Systems on Modern CPU Architectures" [18].

B-Tree with Tracking

The B-Tree with tracking is a standard B-Tree as described above, but with the addition of tracking changes made to its nodes. It is templated on the key type `KeyT` and the value type `ValueT`, which is always a TID in our case. As described above, we template the B-Tree on a third boolean template parameter. When set to true, we extend the page header and slots with additional fields to track changes made to the node, as shown in Figure 5.3.

Header. The header is extended by a `bytes_changed` field to track the degree of write amplification on the page. Everytime a change is made to the page, we increase this counter by the number of bytes changed. When the page is evicted, we can use this information to decide whether to write out the page or not. However, this is only an approximation of the actual write amplification, as some changes might be overwritten by subsequent changes. For example, a node split, where we remove half the entries, followed by several insertions can lead to more bytes changed than the actual node size.

Slots. Each slot is extended by a `state` field to track whether the corresponding entry was `Unchanged`, `Inserted`, `Updated` or `Deleted` since the last time the page was written to storage. More specifically, it does not track the change of the entry itself, but rather the change of the entry from the perspective of the node. For example, if an entry it split off during a node split, the slot is deleted from the perspective of the node. The entry still exists in the tree, but it is now part of a different node. To the new sibling node, where we move over the split off entry, the slot is marked as `Inserted`. The state machine of the operation state field is shown in Figure 5.4 and elaborated in Section 5.2.

Buffer Manager Integration. Our Delta Tree uses this information to determine which changes to store. When a B-Tree with tracking enabled, fixes a page through the buffer manager, it injects a `PageLogic` object into the page's frame. The buffer manager calls this object later when evicting the node to interact with the Delta Tree to extract deltas and to apply deltas when loading the node from storage again.

Delta Tree

The Delta Tree is responsible for storing and applying the changes made to the B-Tree nodes. It is also a B-Tree but templated on a PID as `KeyT` and a variable-sized `Delta` array as `ValueT`.
The `Delta` array stores the changes made to the corresponding page in the B-Tree. A `Delta` array can contain either `InnerNodeDeltas` or `LeafDeltas`. `InnerNodeDeltas` rep-

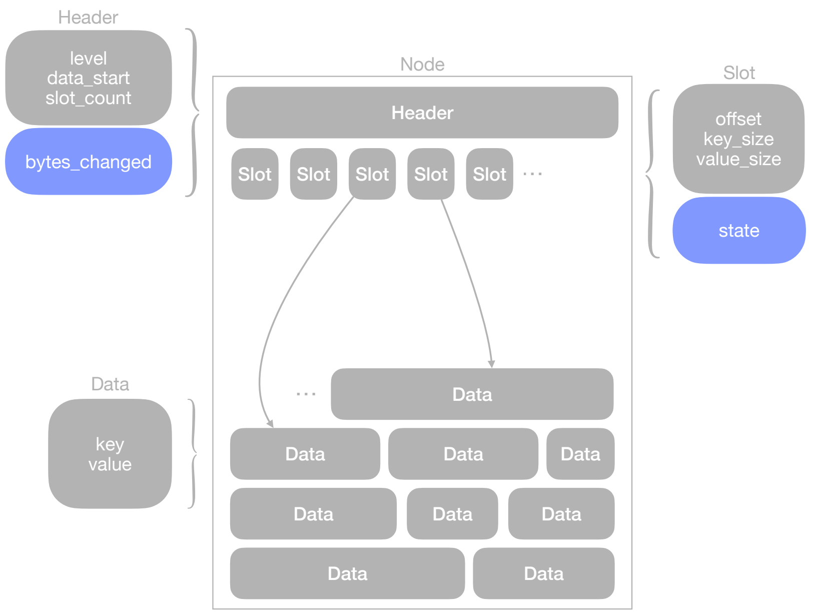


Figure 5.3.: A B-Tree with tracking enabled. The header is extended by a `bytes_changed` field to track the degree of write amplification on the page. Each slot is extended by a `state` field to track whether the corresponding entry is `Unchanged`, `Inserted`, `Updated` or `Deleted`.

resent changes to inner nodes, therefore they store keys and PID changes. `LeafDeltas` represent changes to leaf nodes, therefore they store keys and TID changes.
Each `Delta` array stores the `slot_count` of the corresponding page in the B-Tree at the time of eviction. For `InnerNodeDeltas` we additionally store the `upper` child PID. We do not need to store the level of the node, as a page never changes its level. Therefore, we can retrieve this information from the B-Tree when extracting or applying deltas.
After extracing and storing the `Deltas` of a B-Tree node in the Delta Tree, we can discard the node's page from the buffer manager without writing it to storage. When the node is loaded from storage again, we can apply the stored `Deltas` to reconstruct the state of the node at the time of eviction. In Section 5.2 we elaborate how we can reconstruct the state of a node from the information stored in the Delta Tree together with the disk-state of the node.

Database

Our database class ties all components together. It owns the buffer manager, the index and the slotted pages. It exposes a simple key-value interface to the user, allowing to insert, update, delete and lookup tuples by keys. The class is templated on the key type `KeyT` and the index type `IndexT`. For simplicity, we only support a single table and a single `uint64_t` value in our implementation. Through the `IndexT` template parameter, the user can choose between a standard B-Tree or a BBB-Tree as index structure.
Whenever a user requests an operation, the database class translates it into the corresponding operations on the index and the slotted pages.

5.2. Algorithms

In this section we describe the algorithms for the main operations of our BBB-Tree. While most operations are similar to a standard B-Tree [18], we describe how we extend them to support tracking changes and ensure that we do not loose any changes made to a node. We then describe how we use the tracking information in the B-Tree nodes to extract deltas when evicting a node from memory and how we can reconstruct the state of a node when loading it from storage again.
We do not implement concurrency or node merges in our implementation and therefore do not describe them here. However, all algorithms were implemented with concurrency in mind and therefore can be extended to support it by introducing locks when fixing nodes.

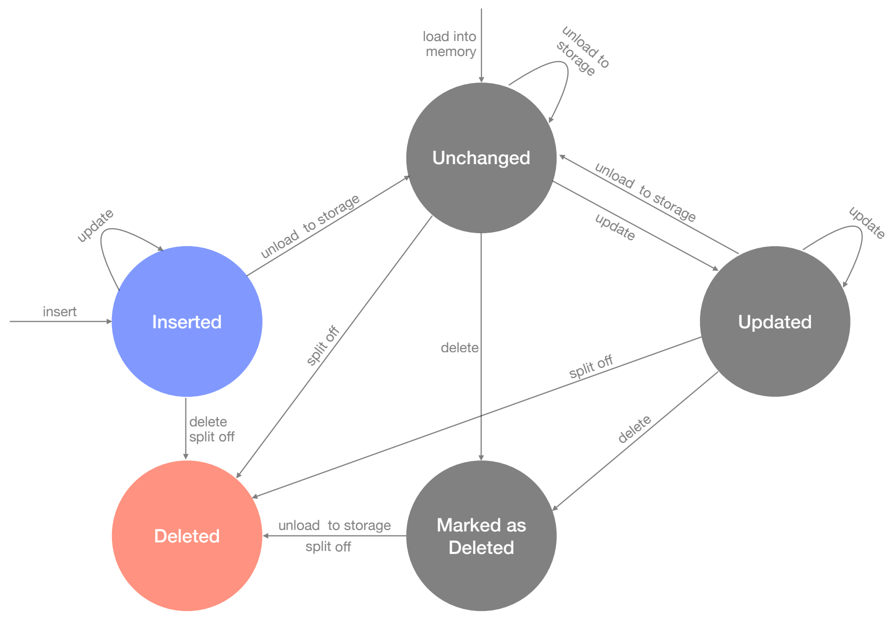


Figure 5.4.: The state machine of a slot's state. The state always resembles the delta of a slot in comparison to its image on disk. Therefore, every slot starts in the `Unchanged` state when being loaded from memory. A slot's lifetime starts with an insertion (highlighted in blue). Its lifetime ends when the slot is deleted (highlighted in red). When a page is unloaded back to storage, all `Inserted` and `Updated` slots are reset to `Unchanged` state. A slot marked as `Deleted` is actually deleted from the node once the page is unloaded.

Lookup

A lookup operation is straightforward in a BBB-Tree, we merely perform a standard B-Tree lookup. We first access the root node and traverse the tree down to the leaf level, following the appropriate child pointers based on the key being looked up. We always perform binary search within a node to find the appropriate slot. Once we reach the leaf level, we perform a final binary search to find the key. If the key is found, we return the corresponding value, the TID. Since we return an `std::optional<ValueT>`, we can also indicate that the key was not found by returning an empty optional `nullopt`.

Tracking. Since we do not modify any nodes during a lookup, we do not need to update any tracking information.

Insert

To perform an insertion, we first traverse the tree down to the appropriate leaf node as described in the lookup operation.
Possibly, the leaf node does not have enough space to accommodate the new entry. In this case, we need to split the node first (see Split 5.2). Once we have an appropriate leaf node with enough space, we insert the new entry into the node.
Since we want to perform binary search within the node, we keep the slots sorted by key. Therefore, we search for the `lower_bound` of the new key to find the appropriate position in the slot array to insert the new slot. We then shift all slots after the insertion position by one to make space for the new slot. We then insert the new slot and the corresponding key and value into the data segment.
The property that slots are always sorted by key is important for the following algorithms.

Tracking. When inserting a new entry into a node, we need to update the tracking information accordingly. Firstly, we increase the `bytes_changed` field in the header by the size of the new entry and the size of the new slot. Secondly, we set the `state` field of the new slot to `Inserted`.
A new slot always starts in the `Inserted` state, as it does not exist on disk (see Figure 5.4). Any subsequent updates to this slot do not change this state, as to the disk image it is still a new slot. During delta extraction, we simply store the latest key and value of this new slot. If the new entry is deleted again before the page is evicted, we can simply discard the slot and do not need to store any delta for it. To the disk image, it is as if the entry was never inserted.

Update

In this operation, we only allow updating the value of an existing entry, not the key. A key update is equivalent to a delete followed by an insert.
To perform an update on a value, we first traverse the tree down to the appropriate leaf node as described in the lookup operation. Once we reach the leaf level, we perform binary search to find the slot with the given key. If the key is found, we update the corresponding value in the data segment. For simplicity, we assume that the new value has the same size as the old value.
In theory, we could translate all updates into a delete followed by an insert. Since the delete operation loads all pages into memory already, the insert operation would not require any additional page loads (unless we require a node split). However, if we delete an entry, we need to keep the entry in the node as a tombstone until the page is written back to storage (see Delete 5.2). This would reduce the space available in the node for new entries, impacting the fanout of the tree. However, one of our design goals is to minimize the performance impact on the B-Tree. During node splits, for example, we must update the child pointers of the entries in the parent node (see Split 5.2). Therefore, we track updates separately to conserve space in the nodes.

Tracking. When updating an existing entry in a node, we increase the `bytes_changed` field in the header by the size of the entry only if the entry was previously unchanged. If the entry was previously marked as `Inserted`, we do not increase the `bytes_changed` field, as we have already accounted for it.
We then set the `state` field of the slot to `Updated` if it was previously `Unchanged`. If the slot was previously marked as `Inserted`, we do not change its state, as to the disk image it is still a new slot.

Delete

To perform a delete operation, we would first traverse the tree down to the appropriate leaf node as described in the lookup operation. We then perform binary search to find the slot with the given key. If the key is found, we remove the slot from the array and the corresponding entry from the data segment. We do not reclaim the space in the data segment eagerly on every deletion. Instead, we compactify lazily. For example, during a node split we remove half of the slots and their data, justifying a compactification.

Tracking. While we do not implement deletion tracking in our current implementation, we describe how we would implement it here for completeness. When deleting an existing entry in a node, we increase the `bytes_changed` field in the header by the size of the entry and the size of the slot only if the entry was previously unchanged. When it was previously inserted, we decrease the `bytes_changed` field, since the insert never happened from the perspective of the disk image of the node. For the same reason, we only actually delete the slot and entry if it was also inserted since its last write to storage. In any other case, we cannot immediately delete the slot. This is, because we must indicate the deletion to the Delta Tree. The entry exists on disk and without an entry to track the deletion, we would not be able to create a delta for it. When loading the node again from storage, the deleted entry would reappear. Therefore, we set the `state` field of the slot to `Deleted` instead. Only when the page is written to disk, we actually delete all slots marked as `Deleted` from the node.

Split

When a leaf node does not have enough space to accommodate a new entry, we need to split the node first. To address the possibility of concurrent splits in the future, we use a restart mechanism.
We first traverse the tree down to the appropriate leaf node as described in the lookup operation. However this time, we keep track of the path taken down the tree and keep the pages fixed in memory. Once we reach the leaf level, we check whether the node has enough space to accommodate the new entry in the current iteration. Firstly, this is necessary in a concurrent setting, as another thread might have split the node in the meantime. Secondly, this is necessary, because splitting a leaf node once might not be enough to accommodate the new entry when supporting variable-sized entries. For example, we could have a new entry that is as large as the entire node. In that case, we would need to split the node multiple times until we reach an empty node.
If the leaf has enough space, we return. Otherwise, we split the node. We allocate a new sibling node and move half of the entries from the current node to the sibling. Then, the new PID of the sibling and the new fence key are propagated up the tree. The new fence key is the largest key in the left node after the split. We can move upwards now, since we locked the path exclusively when traversing down the tree. If the parent node does not have enough space to accommodate the new fence key and child pointer, we need to split the parent node as well. We repeat this process until we reach a node that has enough space in an upwards motion. Should we reach the root node and it does not have enough space, we create a new root node, increasing the tree's height by one. Once a node has enough space, we can insert the new entry and move back down the tree to insert the respective entries on each level. Again, we might need to split nodes on the way down again, since we support variable-sized keys. Therefore, we can repeatedly move down and up the tree until we reach the leaf level again. When reaching the leaf level, we repeat the process until the leaf has enough space to accommodate the new entry.

Finally, we insert the new entry into the leaf node.
Some implementations use "safe" inner pages that always have enough space to accommodate a new entry [18]. This simplifies the split operation, as we never have cascading splits that need to propagate up the tree. However, this is limited to fixed-sized keys. With variable-sized keys, we cannot know how much space we need to reserve in the inner nodes to accommodate a new fence key.
When creating a new node, we always write it out to storage on eviction. This is because new nodes carry enough information to justify the write. Also, it greatly simplifies tracking, since we can express all deltas as slot changes.
After splitting a node, we compactify the node to reclaim space from deleted slots and to defragment the data segment.

Tracking. When splitting a node, we delete half of the slots from the perspective of the current node. For each deleted slot, we increase the `bytes_changed` field in the header by the size of the entry and the size of the slot only if the entry was previously unchanged. If the entry was previously inserted or updates, those bytes were already accounted for. The `bytes_changed` field is only an approximation of the actual degree of change on the page. This is, because we can change the same bytes multiple times by deleting and inserting slots. For example, we can split a node, deleting half of the slots, and then insert new slots again. We do not aim to account for every single byte change, but rather to get a rough estimate of the degree of change on the page. This allows us to make informed decisions whether a write-out is justified at the time of eviction.
As we discussed in the delete operation (see Subsection 5.2), we usually cannot actually delete the slots from the node, as we need to track the deletion as a delta to ensure that we do not make entries reappear when loading the node from storage again. However, we cannot mark the slots as `Deleted`, since that would keep the data in the node and therefore not free up any space, negating the purpose of a node split. For node splits we use a different approach. Due to the fact that slots are always sorted by key and we always split the node in half, we can use the `slot_count` field in the header to determine which slots are still part of the node and which slots were split off. Therefore, we do not need to keep any tracking information for the split off slots. Instead, we store the `slot_count` of the node at the time of eviction in the corresponding `Delta` array. When applying deltas to a node, we can use this information to determine which slots are still part of the node and which slots were split off. This way, we can discard split off nodes, freeing up space for new entries. To the new sibling node, the moved over slots are marked as `Inserted`.
As described above, when a child node splits, we need to insert the new fence key and new child pointer into the parent node. The key that is already present in the parent node remains unchanged, as it still forms the upper bound of that range. However,

it is not the upper bound of the new sibling node. Therefore we perform an update on the existing slot, changing the child pointer to the new sibling node. The tracking information for that node changes according to the update operation described above. The new fence key is now the upper bound for the split node. Therefore, we insert a new slot into the parent node with the new fence key and the old child pointer. The tracking information for that node changes according to the insert operation described above.

Compactification

To reclaim free space, we compactify a node by defragmenting the data segment. Fragmentation in a node occurs through deletions, since we do not reclaim the space in the data segment eagerly. This would require moving possibly all entries in the data segment and updating the corresponding slots. This would be an expensive operation to perform on every deletion. Instead, we perform compactification lazily, for example after a node split. After a node split, half of the slots are removed from the node, freeing up a significant amount of space. More importantly, we split a node in particular to free up space for new entries.
First, we collect pointers to all slots that still point to valid entries. We also collect slots marked as `Deleted`, as we need to keep them in the node until the page is written back to storage to track the deletion as a delta. We then sort them by their offset in the data segment, starting with the highest offset. Then, we move the entries to the end of the data segment, updating the corresponding offsets in the slot accordingly. Finally, we update the header to point to the new start of the data segment.

Tracking. Since compactification only moves entries around in the data segment, we do not change the node logically. Therefore, we do not change any tracking information. However, compactification is an expensive operation and reclaiming space is important to keep the fanout of the tree high. We would like to maintain the space gains from compactification also after discarding the page and loading it again from memory. Therefore it makes sense to always write out a page after a node split, as a node split usually carries enough information through its structural changes to justify the write. If we require more than 50% change on the page to perform a write-out, this would be a given, since we change half of the node during node splits. In 6 we will see that requiring a degree of change of even less than 50% is a reasonable threshold to perform a write-out.

Eviction

When the buffer manager evicts a B-Tree's node from memory, it evokes the injected Delta Tree. It needs to decide whether to write out the page to storage or not. To that end, we calculate the degree of change, by comparing the `bytes_changed` field in the header to the size of the node.

$$degree\_of\_change = bytes\_changed / page\_size$$

The Delta Tree is passed a threshold parameter *wa_threshold* between 0 and 100. It determines the minimum *degree_of_change* required to justify a write-out of the page.

Resetting Deltas

If $degree\_of\_change > wa\_threshold$, we write the page to storage. In that case, we scan the slot array of the node and reset all tracking information: We set all slots to `Unchanged` state. We remove all `Deleted` slots from the node, as they are now actually deleted from the disk image of the node. We set the `bytes_changed` field in the header to 0. Finally, we erase any corresponding deltas for this page from the Delta Tree, as they are now obsolete. We can perform a standard B-Tree deletion. We indicate to the buffer manager to continue writing the page to storage.

Extracting Deltas

If $degree\_of\_change \leq wa\_threshold$, we extract all deltas from the node and store them in the Delta Tree. We scan the slot array of the node and for each slot that is not `Unchanged`, we create a corresponding delta. The resulting delta array is then stored in the Delta Tree with the node's PID as key. This is done with a standard B-Tree insertion. Finally, we indicate to the buffer manager to discard the page without writing it to storage.

Applying Deltas

When the buffer manager loads a B-Tree's node from storage, it invokes the injected Delta Tree to apply any stored deltas to the node. We first check whether there are any deltas stored for the node's PID in the Delta Tree. If not, we are done. Otherwise, we apply the deltas to the node. In the end we cut off any slots that were split off during a node split by using the stored `slot_count` in the delta array. Since we apply inserts and updates before cutting off split slots, we can ensure that we do not let any entries reappear that were actually split off. This must be correct, since we only stored deltas

for slots that are still part of the node at eviction time. Everything beyond the stored `slot_count` was split off and therefore cannot be part of the node.
However, when applying deltas to a node, we might exceed the node size temporarily. For example, we could have a node that is full on disk. In memory it had been split and then some insertions followed. When applying the deltas to the full node, we would exceed the node size, since we perform the insertions before cutting off the split slots. To address this, we allow exceeding the node size temporarily when a node becomes full while applying deltas After cutting off split slots, we compactify the node and shrink it back to its original size.
We keep the deltas in the Delta Tree after applying them. This allows us to keep the page in a clean state should it be evicted again without any changes. We can simply discard the page, without needing to extract and store the deltas again.

5.3. Testing

All components of our system are covered by unit tests. We used the Google Test framework to write and run our tests. We tested the buffer manager, the slotted pages, the B-Tree, and the BBB-Tree separately. We also wrote integration tests to test the interaction between the components.

6. Evaluation

6.1. Experimental Setup

6.2. Results and Analysis

Write Amplification

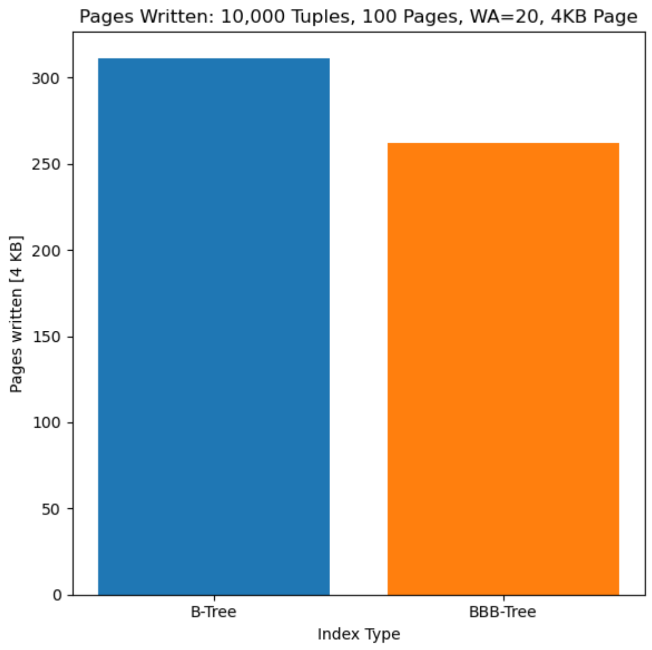


Figure 6.1.: Comparison of B-Tree and BBBTree performance for 10,000 insertions, building the index from scratch. We observe 15% reduction in write amplification with our approach.

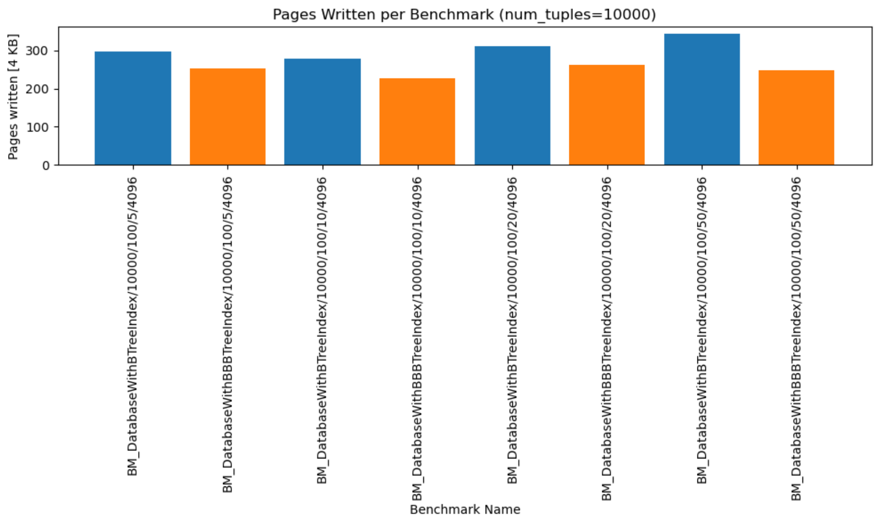


Figure 6.2.: Comparison of B-Tree and BBBTree performance for 10,000 insertions, building the index from scratch. We observe reduction in write amplification across different write amplification thresholds.

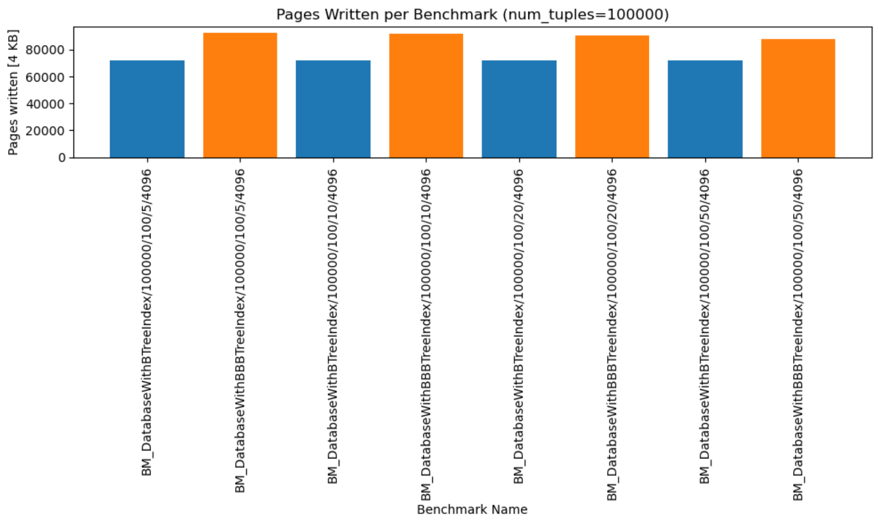


Figure 6.3.: Comparison of B-Tree and BBBTree performance for 100,000 insertions, building the index from scratch. We observe increase in write amplification across different write amplification thresholds.

Latency and Throughput

Space Utilization and Memory Overhead

Sensitivity Analysis

6.3. Discussion

7. Discussion and Future Work

7.1. Summary of Findings

7.2. Limitations

7.3. Future Directions

7.4. Potential Applications

8. Conclusion

8.1. Recap of Contributions

8.2. Final Thoughts

A. Appendices

A.1. Pseudocode

A.2. Additional Graphs and Tables

A.3. Configuration Files / Benchmarking Scripts

Abbreviations

LSM-Trees	Log-Structured-Merge-Trees
SSD	Solid State Drive
DBMS	Database Management Systems
DRAM	Dynamic Random Access Memory
I/O	Input/Output
TID	Tuple ID
PID	Page ID
CAS	Compare-And-Swap
WAL	Write-Ahead Log
DAM	Disk-Access-Model

List of Figures

List of Tables

Bibliography

[1] A. Aggarwal and S. Vitter Jeffrey. “The Input/Output Complexity of Sorting and Related Problems.” In: *Communications of the ACM* 31.9 (1988), pp. 1116–1127.

[2] R. Bayer and E. McCreight. “Organization and Maintenance of Large Ordered Indices.” In: *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. 1970, pp. 107–141.

[3] M. A. Bender, M. Farach-Colton, W. Jannen, R. Johnson, B. C. Kuszmaul, D. E. Porter, J. Yuan, and Y. Zhan. “An Introduction to B^ϵ-trees and Write-Optimization.” In: *login; magazine* 40.5 (2015).

[4] G. S. Brodal and R. Fagerberg. “Lower bounds for external memory dictionaries.” In: *SODA*. Vol. 3. 2003, pp. 546–554.

[5] A. L. Buchsbaum, M. H. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. “On external memory graph traversal.” In: *SODA*. 2000, pp. 859–860.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. “Bigtable: A Distributed Storage System for Structured Data.” In: *ACM Transactions on Computer Systems (TOCS)* 26.2 (2008), pp. 1–26.

[7] A. Gorrod. *Btree vs LSM*. 2017. URL: `https://github.com/wiredtiger/wiredtiger/wiki/Btree-vs-LSM` (visited on 09/15/2025).

[8] G. Graefe. “Write-optimized B-trees.” In: *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. 2004, pp. 672–683.

[9] G. Graefe, H. Volos, H. Kimura, H. Kuno, J. Tucek, M. Lillibridge, and A. Veitch. “In-Memory Performance for Big Data.” In: *Proceedings of the VLDB Endowment* 8.1 (2014), pp. 37–48.

[10] G. Haas, B. Lee, P. Bonnet, and V. Leis. “SSD-iq: Uncovering the Hidden Side of SSD Performance.” In: *Proceedings of the VLDB Endowment* 18.11 (2025), pp. 4295–4308.

[11] G. Haas and V. Leis. “What Modern NVMe Storage Can Do, And How To Exploit It: High-Performance I/O for High-Performance Storage Engines.” In: *Proceedings of the VLDB Endowment* 16.9 (2023), pp. 2090–2102.

[12] A. Kemper and T. Neumann. "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots." In: *2011 IEEE 27th International Conference on Data Engineering*. IEEE. 2011, pp. 195–206.

[13] B. C. Kuszmaul. "A Comparison of Fractal Trees to Log-Structured Merge (LSM) Trees." In: *Tokutek White Paper* (2014).

[14] V. Leis. "LeanStore: A High-Performance Storage Engine for NVMe SSDs." In: *Proceedings of the VLDB Endowment* 17.12 (2024), pp. 4536–4545.

[15] 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)*. IEEE. 2018, pp. 185–196.

[16] 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.

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

[18] T. Neumann. *Database Systems on Modern CPU Architectures - Chapter 3: Access Paths*. 2024. URL: `https://db.in.tum.de/teaching/ss24/moderndbs/chapter3.pdf?lang=en` (visited on 09/14/2025).

[19] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. "The Log-Structured Merge-Tree (LSM-Tree)." In: *Acta informatica* 33.4 (1996), pp. 351–385.

[20] Y. Qiao, X. Chen, N. Zheng, J. Li, Y. Liu, and T. Zhang. "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)*. 2022, pp. 69–82.

[21] S. Sarkar and M. Athanassoulis. "Dissecting, designing, and optimizing LSM-based data stores." In: *Proceedings of the 2022 International Conference on Management of Data*. 2022, pp. 2489–2497.

[22] F. R. team. *A Persistent Key-value Store for Fast Storage Environments*. 2021. URL: `https://rocksdb.org/` (visited on 09/15/2025).

[23] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, and D. G. Andersen. "Building a Bw-tree Takes More Than Just Buzz Words." In: *Proceedings of the 2018 International Conference on Management of Data*. 2018, pp. 473–488.

[24] W. Zhong, C. Chen, X. Wu, and S. Jiang. "REMIX: Efficient Range Query for LSM-trees." In: *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 2021, pp. 51–64.