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, and Altan Birler for their inspiration, continuous support, and guidance throughout this thesis.

Abstract

B-trees are one of the most widely used data structures in modern database systems because of their efficient access patterns and excellent lookup performance on large volumes of data. However, B-trees perform suboptimally under random writes, a pervasive pattern for secondary indices. Such workloads introduce write amplification in B-trees, a condition in which the amount of data written to storage is significantly larger than that of logically changed data. As a result, B-trees suffer increased latency, reduced throughput, and premature device wear with write-intensive workloads.

An alternative is Log-Structured-Merge-Trees (LSM-trees), which trade lower read performance for higher write performance. However, this trade-off makes LSM-trees unsuitable for general-purpose database systems that require more balanced performance characteristics. Other attempts to reduce write amplification in B-trees either reduce concurrency, degrade read performance, or rely on hardware-specific features, thereby 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 doing so, we reduce write amplification and enable high performance under random writes, while preserving the benefits of traditional B-trees.

We implement the proposed structure, evaluate its performance, and analyze its trade-offs. Compared to traditional B-trees, our method achieves up to 70% fewer page writes while maintaining comparable read performance.

These results suggest that write-aware optimizations for B-trees can significantly improve the efficiency of write-intensive applications, contributing to the broader effort to design storage-efficient data structures suited for modern hardware.

Contents

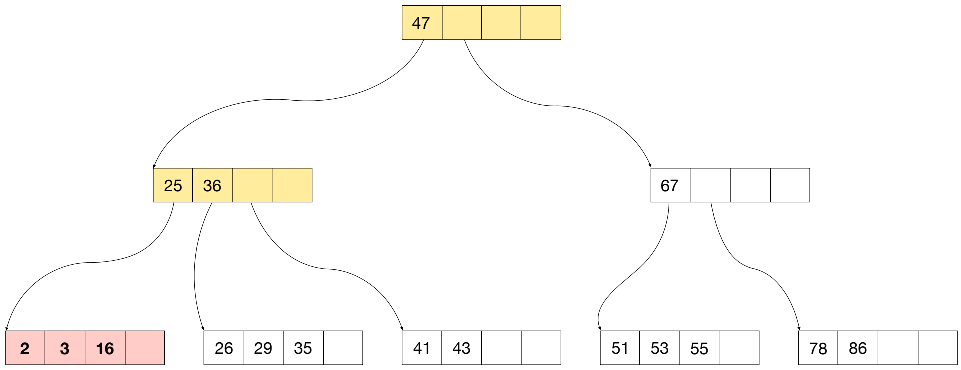
1 Introduction

Efficiently managing large datasets is a core requirement for database systems. Therefore, minimizing input/output (I/O) operations remains a fundamental principle in the design of modern, high-performance database management systems (DBMS). This is primarily achieved by caching frequently accessed pages in Dynamic Random Access Memory (DRAM) using a buffer manager [16]. The system stores data in pages, which the buffer manager can cache and uniformly serve to all components in the system. This modular design separates concerns between the caching layer and data layer, such as indices and data structures. However, this also means that the buffer manager is agnostic to user access patterns. While the buffer manager minimizes the number of I/O operations, to the best of its knowledge, every system component must design its access patterns to be as efficient as possible.

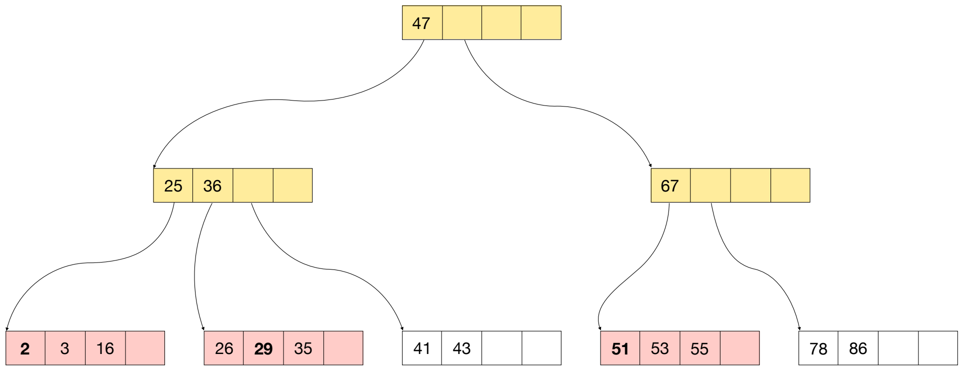
A prominent example of such a component is the B-tree [2]. B-trees are the dominant data structure for indexing large datasets in disk-based DBMS due to their excellent lookup performance, support for range queries, and simplicity. However, random writes, a prevalent pattern for secondary indices, lead to inefficient access patterns that a buffer manager cannot hide in 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, which the buffer manager must load into the buffer. At eviction time, each modified page requires a complete rewrite to storage, even if only a small portion of the page has changed. Figure 1.1 illustrates this effect by comparing a B-tree's sequential and random update patterns. We consider three updates to the tree. In the sequential update, the B-tree reads only three pages and writes to one of them. In the random update, the B-tree reads six pages and writes to three of them. Merely changing the access pattern from sequential to random leads to a threefold increase in the amount of data written to storage. Assuming that each page is 4 KB, the sequential update requires one storage write of 4 KB, while the random update requires three storage writes of 12 KB in total. Random writes introduce *write amplification*, a phenomenon in which 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 I/O-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) has its own internal write amplification due to its flash translation layer's garbage collection process [12]. This results in additional unnecessary physical writes, causing the device to wear out faster.
In summary, to design a truly efficient, high-performance system, we must minimize I/O operations in all components of the storage stack. This thesis focuses 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 introduces 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 more extended 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 two 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?

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. Instead, we aim to close the B-tree's efficiency gap under random writes while preserving its strengths: versatility, concurrency, and simpliticity.
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 incorporating 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 the buffer manager evicts a B-tree node from memory, we determine whether it has changed significantly enough to warrant a full write to storage. If not, we buffer the changes. We aim to minimize write operations to those strictly necessary.
The novelty of our approach lies in its non-intrusive design: We only perform additional operations when the buffer manager exchanges B-tree nodes between memory and external storage. In contrast to other approaches (see Chapter 3), we neither alter the B-tree's structure nor its fundamental operations, nor impact concurrency in the tree. We keep read amplification low by only introducing additional lookups during a B-tree's page reads. Our approach is easy to integrate into existing systems and preserves the desirable properties of traditional B-trees.
We hereby contribute to the broader effort of minimizing the overhead of beyond memory systems and designing efficient, high-performance database systems for modern hardware.

2 Background

This chapter provides the necessary background for understanding the problem of write amplification in B-trees and the proposed method by outlining the characteristics of external storage, the architecture of database systems, and the behavior of B-trees in out-of-memory environments. Furthermore, we introduce the concepts of write, read, and space amplification, which allows for a differentiated analysis of index structures for their efficiency and performance characteristics.

2.1 External Storage Characteristics

For some time, in-memory database systems like Hyper [13] have gained popularity due to the decreasing cost of DRAM. However, that trend has reversed recently, as DRAM prices have stagnated [12] and SSD price-performance-ratios have improved significantly [15]. 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 SSDs have a smaller difference between random and sequential I/O, they still exhibit asymmetric performance, especially in writes [12]. 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. This assumes that subsequent accesses exhibit some locality, which is often the case in practice. 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. Note that this architecture is not a constraint of our method, which

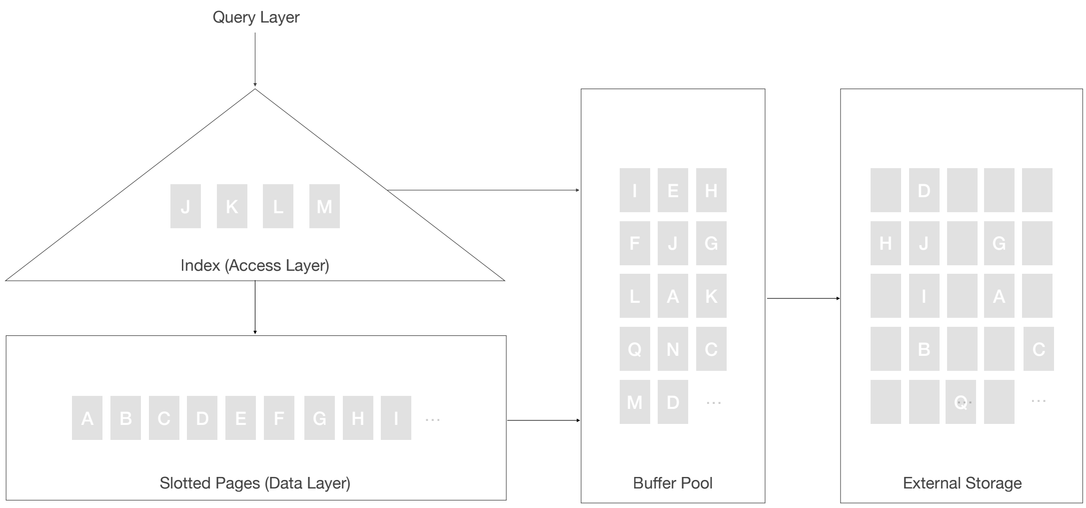


Figure 2.1: The storage and access layer of a database system. The index provides access to the tuples stored in the slotted pages. Each component requests data from the buffer manager, which handles caching and loading pages from external storage. Adapted from "Database Systems on Modern CPU Architectures" [19].

can be applied to any database system using B-trees as an index structure. However, for clarity, we describe our method in the context of this architecture.
The access and storage layer of a database system typically consists 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 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. Ideally, the TID never changes, as this would require all indices pointing to that tuple to be updated. 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 indices on the same data. If we stored tuples directly in the index, we would need to update all indices when a tuple changes. Therefore, the access and storage layers are decoupled via TIDs. For the context of this thesis, however, it does not matter whether the index maps keys to TIDs or directly to tuples.
Indices can be classified into primary and secondary indices. 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.
Consider the following example use-case: a user database with a primary key on the user ID and a secondary index on the email address. When inserting several new users, we update the indices. The primary index is updated with an auto-incrementing user ID, thus, the primary key follows a sequential access pattern. However, the email

addresses of new users are likely to be random and not follow any specific order. Therefore, if the secondary index is sorted on the keys, as they are in a B-tree, the index exhibits a random access pattern. Such access patterns have implications for the performance of index structures, as we will discuss in the following sections.

2.4 B-trees

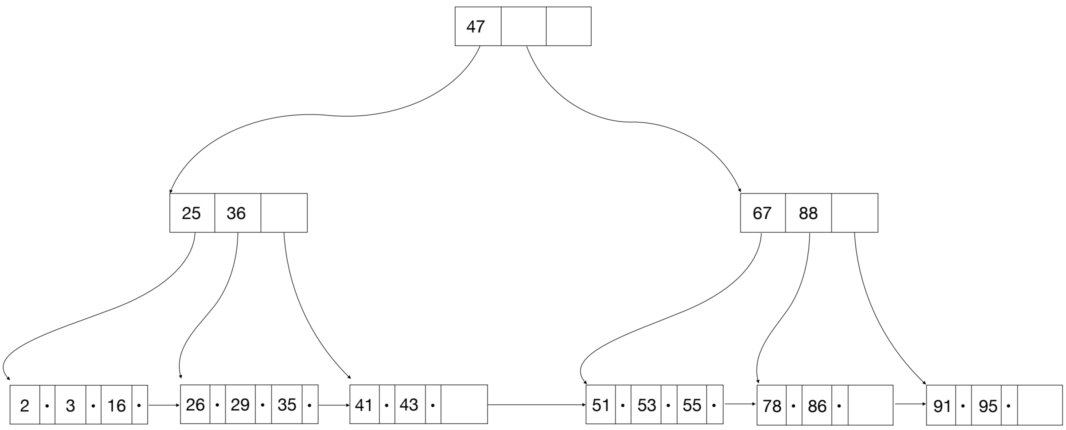


Figure 2.2: A B+-tree. Child pointers are represented as arrows. Values (the TID) are represented as bullet points •.

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, $\mathcal{O}(\log n)$, where n is the number of entries in the tree. 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 simplicity, we omit the 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 fewer pages are involved in reaching the leaf level. Also, it allows for efficient range queries by scanning the leaf nodes in order. Due to their excellent lookup performance, versatility, and simplicity, B+-trees are the dominant data structure for external storage [19].

Disk-Access-Model

To understand the disk access patterns of B-trees, we analyze the number of I/O operations using the Disk-Access-Model (DAM) [1] [14]. The model has two levels of memory: an internal memory of size M and an 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 do **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 as well. 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.
Node sizes are a tuning parameter that is not the primary focus of this thesis. Instead, we focus on reducing I/O amplification in B-trees at any node size.

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 B / 64 B = 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 indices 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 nodes, and update the parent nodes. Thus, a database system inherently comes with write amplification merely to perform its purpose: manage data. This is 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 has 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. We essentially reduce write amplification by reducing the number of pages written physically for a given number of bytes written logically by the B-tree (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. SSDs 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 [11]. 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 [14]. 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 a common tradeoff when reducing write amplification, as we will see in alternative data structures (see Chapter 3). We will analyze the introduced read amplification of our method compared to the traditional B-tree in our evaluation (see Subsection 6.3.2).

2.7 Space Amplification

Space amplification is the ratio of the amount of space used by the data structure versus the amount of logical data stored [14]. 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. Space amplification is not the focus of the thesis; however, we will analyze the space utilization of our method compared to the traditional B-tree in our evaluation (see Subsection 6.3.3).

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. In Section 3.1, we discuss LSM-trees, a popular write-optimized data structure alternative to B-trees. In Section 3.2, we review B^{ϵ}-trees, a write-optimized B-tree variant. In Section 3.3, we discuss the Bw-tree, another B-tree variant optimized for modern hardware. In Section 3.4, we review Graefe's Write-Optimized B-trees, an approach to improve write performance in B-trees by batching page writes. In Section 3.5, we discuss FineLine, a storage manager that decouples in-memory data structures from their disk representation. Finally, in Section 3.6, we review a hardware-based approach to reduce write amplification in B-trees using transparent compression.

3.1 Log-Structured-Merge-Trees

LSM-trees [20] 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 [24] 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 run in a file called SSTable. 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 [22].

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 LSM-trees makes this data structure suitable for write-heavy workloads, such as time-series data or logging systems.

Write Amplification. LSM-trees exhibit write amplification because data is written multiple times as it moves through the tree's storage hierarchy [14]. Each write is first stored in memory, then flushed to disk as an SSTable, and repeatedly rewritten during compactions that merge overlapping files across levels. These repeated rewrites cause the total bytes written to disk to exceed the amount of user data written, resulting in write amplification.

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 [8].
When performing point lookups, LSM-trees check 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 look up 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 [6]. 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 [26], they are not designed for efficient range queries, as range data is scattered across the tree [22].

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 the 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 is the key-value pair removed from the leaf. Each message encodes a timestamp to ensure that the updates are applied in the correct 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. In comparison, 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 a good eviction strategy, this can effectively reduce the number of write operations to external storage. Another effect of this design is that it 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 taken into account 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^{ϵ}-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 number 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 [18] is a variant of the B-tree 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. A garbage collector reclaims outdated base nodes. 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 for batch writes to external storage. Only when creating new pages during consolidation do they need to write the entire page. In that case, the node has experienced sufficient modifications that justify writing the 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 its implementation details to implement this logic. This makes changes to the caching layer difficult. For example, pointer swizzling [10] would be infeasible with this design. Every time 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 [16] 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" [25] that the Bw-tree's performance is actually not competitive with traditional B-trees using optimistic lock coupling [17]. 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

Graefe proposes "Write-Optimized B-trees" [9] to address 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 pages and in-place updates. We avoid writing individual pages as a whole by deferring updates and buffering them in batches instead.

3.5 Decoupling Storage from In-Memory Data Structures

Sauer et al. propose FineLine [23], a storage manager that decouples in-memory data structures from persistence concerns in a database system. In-memory data structures are never propagated to disk. Instead, FineLine merely uses the log for persistence. When an in-memory data structure (for example, a B-tree) causes a cache miss, FineLine reconstructs its state by replaying the log of operations from disk. When a transaction commits, its log of changes to the data structures is flushed to disk in a sequential manner, allowing for high write performance. To improve read performance for reconstruction, FineLine employs a compaction mechanism that merges partitions of the log, similar to LSM-trees.
Since all flushes to disk are sequential, write amplification is small. Some write amplification is introduced by the compaction process. Also, traditional systems need to write both the data structure pages and the WAL to disk. In FineLine, the log is the only component that requires persistence. This reduces random writes in the system further.
FineLine also allows for a graceful recovery mechanism, as the log can be replayed when the in-memory state is lost (e.g., after a crash) in a similar way that a cache miss

is handled.
One drawback of this approach is the high read amplification when reconstructing data structures from the log. Every time a node is reconstructed, the log needs to be scanned for all entries that affect the node. Due to compaction, this search can be efficient. However, assume a high update rate on a particular data structure node. Many partitions of the log may contain updates for this node, requiring multiple reads from disk to reconstruct the node.

3.6 Transparent Compression

Qiao et al. [21] propose a hardware-based approach to reduce write amplification in B-trees. They use transparent compression, a hardware feature of some modern storage devices that offers 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 that do not actually waste space on the storage device.
In their approach, Qiao et al. 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; 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 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.

Reducing 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 has changed. This page-oriented design is crucial to utilize the 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 minimize 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.

Comparable 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, LSM-trees incur high read amplification to achieve high write efficiency. Bw-trees introduce a delta chain to each node that needs to be traversed on every node read. Bϵ-trees introduce an extra binary search for every node on the search path. While every write-optimized data structure has to trade off read performance for write efficiency to some degree, we aim to keep the read performance of B-trees as close as possible to the original B-tree.

Preserving 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, without introducing further locking in the B-tree. Our method remains independent of the concurrency control mechanism used in the B-tree.

Maintaining 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 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.

Reducing Write Amplification. Write amplification in a B-tree happens at the point of eviction, when a page is written to storage. The buffer manager is only aware that a page is dirty. It does not know which parts of the page have 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 and accumulate them in the Delta Tree until we have enough changes to justify a full rewrite to storage. We aim for a batching effect in two ways:

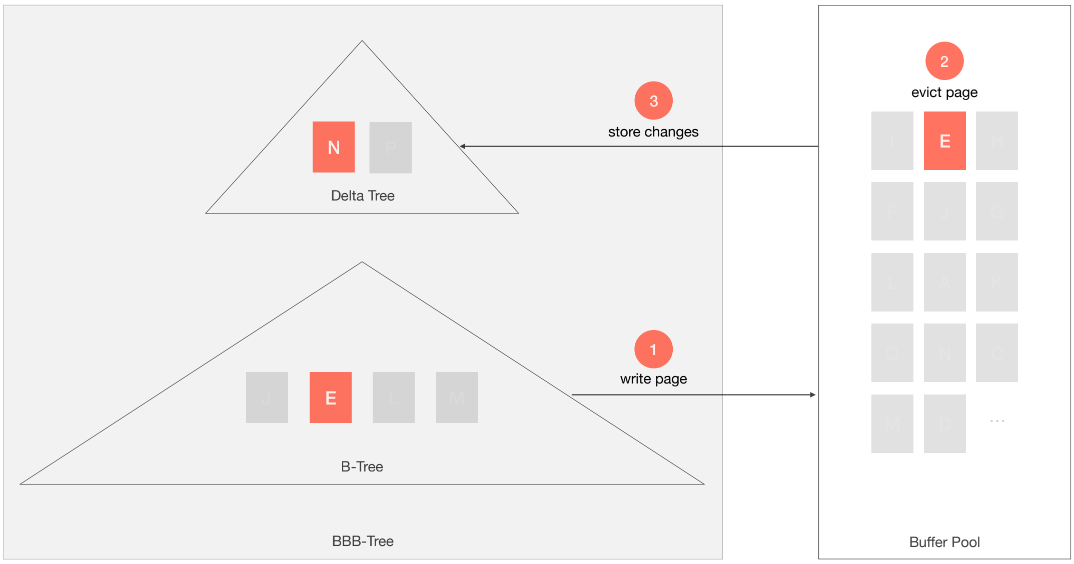


Figure 4.1: High-level architecture of the data structure. We add a Delta Tree to a B-tree. When evicting a dirty page from the B-tree, we can buffer the changes of the B-tree pages instead of writing them to storage. When loading a page from storage, we apply all buffered changes to it before returning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

1. **Batching Changes of Single Pages:** We buffer multiple small changes to the same page until we have accumulated enough changes to justify a full rewrite. We then write all changes to the page in one batch.

2. **Batching Changes Across Pages:** A single Delta Tree page can buffer changes for multiple B-tree pages. Therefore, we can batch changes across multiple pages into a single write operation. This way, we can turn many small random writes into fewwer large writes.

Both effects help to reduce write amplification in the B-tree.
We also perform writes to the Delta Tree itself. Firstly, when storing changes to B-tree pages. Secondly, when enough changes accumulate in the B-Tree pages, we write them back to storage. In that case, we need to remove the corresponding changes from the Delta Tree. At the point of applying changes from the Delta Tree to a B-tree page, we can keep the changes in the Delta Tree for future evictions. This way, we perform fewer 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 subsequent eviction can simply discard the page without any writes. These observations suggest that the Delta Tree should be small and fit into memory as much as possible (see Chapter 6). As a result, we can expect fewer evictions of Delta Tree pages compared to B-tree pages.

Comparable 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. 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 itself is a disk-based data structure, which might require I/O operations to look up buffered changes. We will be evaluating the read overhead in Chapter 6.

Preserving Concurrency. The Delta Tree is separate from the B-tree. Therefore, we do not compromise the 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.

Maintaining 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

We modify the traditional B-tree in three ways to support our approach.

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 number 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 was 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 reaccessed at a later point in time. We do not incur any space overhead for the state field as elaborated in Section 5.1.

3. **Injecting Callbacks:** As described above, we need to execute specific logic at the eviction time and the loading time of a B-tree page. However, the B-tree has no control over the point in time at which a page is evicted or loaded. Therefore, we inject callbacks into the buffer frames that the buffer manager later invokes. 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 immediately insert changes into the Delta Tree whenever a change occurred 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. Imagine a sequential workload that updates the same B-tree node multiple times while it is in memory. Every write would require an update to the Delta Tree as well. However, when evicting the page, we would write the full page to storage anyway, as many changes occurred. We would degrade performance unnecessarily. 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 do we insert them into the Delta Tree. This way, we keep the performance impact of B-tree operations minimal. We evaluate the tracking overhead of our approach in Chapter 6.

Delta Tree

The Delta Tree is the component that buffers changes to 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 every time 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 choose not to 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 be discarded.

   Should it choose 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.

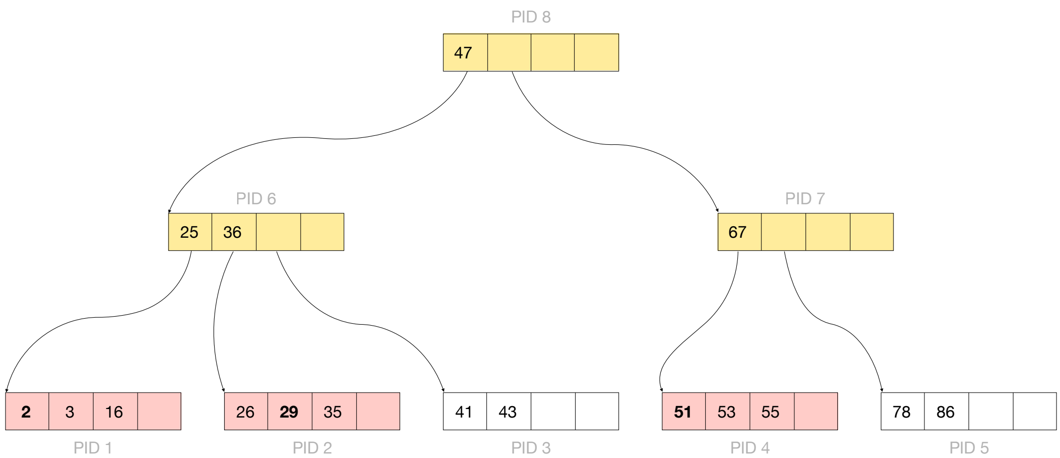
2. **Loading Time:** When loading a B-tree page from storage, the Delta Tree looks up if there are any buffered changes for that page. If so, it applies the changes to the page before returning it to the B-tree. Together with the state of the page on storage, we can reconstruct the state of the page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

   After changes were applied, we keep them in the Delta Tree, as they might be useful for future evictions. This way, we perform fewer 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 subsequent eviction can discard the page without any writes.

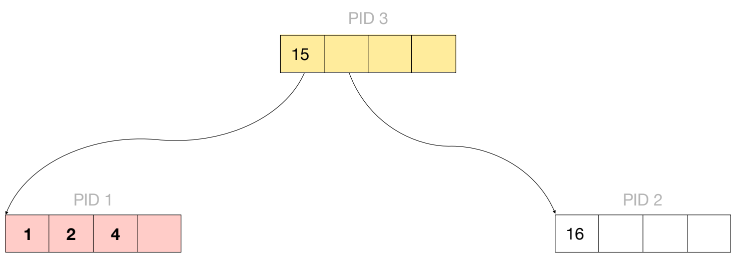
The Delta Tree itself contains pages that are managed by the buffer manager. Its pages can be evicted to storage as well. Therefore, we want to keep the Delta Tree small to batch changes more effectively.

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, and 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.
Another effect is that we can avoid writing B-Tree pages that are frequently evicted with small updates. With a single Delta Tree page, we can address the changes of multiple B-tree pages. Therefore, the Delta Tree can be much smaller than the B-tree itself and fit into memory more easily. While the Delta Tree might evict pages occasionally, the B-tree pages are less likely to be evicted frequently. As a result, we can accumulate changes of B-tree pages over a longer period of time, reducing the frequency of writes to storage.



(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 in 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 discard the page.

Slotted Pages

We store tuples within slotted 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 variable-length tuples [19]. 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 [10]. Since our approach is orthogonal to these optimizations, we do not implement them in our prototype.
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 similarly 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 for this B-tree instantiation as described in the next section. This allows us to use the same B-tree implementation for both the standard B-tree and the B-tree with tracking required for the 3B-tree.

3B-tree

We call our method the 3B-tree. It wraps a B-tree with tracking information and a corresponding Delta Tree to store the changes made to the B-tree nodes. Both components are described in the following.

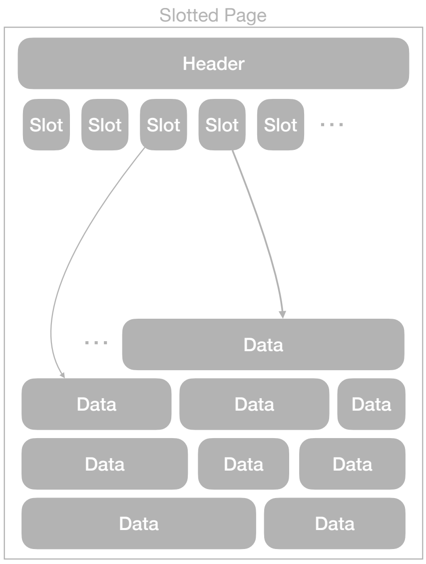


Figure 5.1: The data layout of a slotted page. The header contains metadata about the page, such as the number of slots and the pointer to the data segment. Each slot points to the actual tuple data stored in the data segment. Tuples can be of variable length and are accessed through their TID. Adapted from "Database Systems on Modern CPU Architectures" [19].

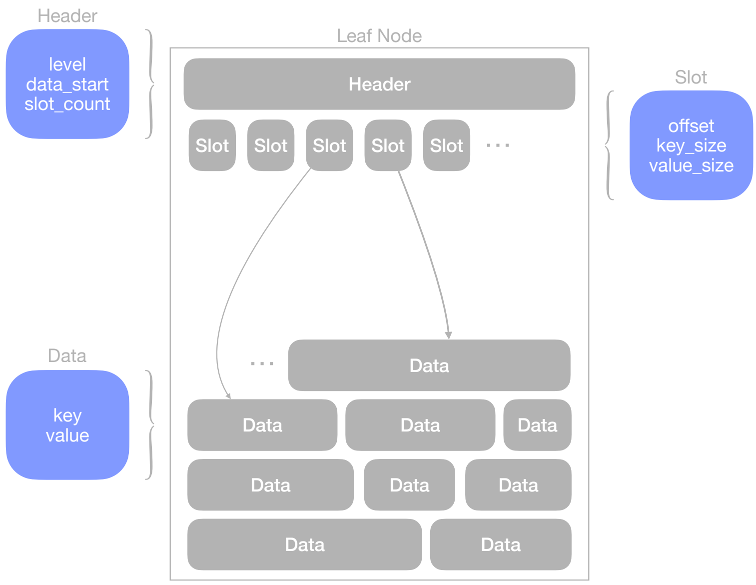


Figure 5.2: The data layout of a leaf node in the B-tree supports variable-sized keys and values. The header contains metadata about the node, such as the level in the tree, the offset to the data segment, and the number of slots. The slots point to the actual entries stored in the data segment. The data segment contains the keys and values. Inner nodes store keys and pointers to child nodes. Adapted from "Database Systems on Modern CPU Architectures" [19].

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 `uint16_t bytes_changed` field to track the degree of write amplification on the page. Every time 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 is 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`. To reconstruct the state of a node when loading it from storage again, we need to know whether an entry was newly inserted or whether it existed on disk already and was only updated in memory. Assume a deletion of an entry that was inserted since the last write to storage. In that case, we can simply discard the slot and do not need to store any delta for it. If the entry existed on disk already, we need to track the deletion as a delta to ensure that we do not let the entry reappear when loading the node from storage again. Therefore, we need to differentiate between newly inserted entries and existing entries that were updated. The state machine of the operation state field is shown in Figure 5.4 and elaborated in Section 5.2.
Since we only have four states, we can store the state in two bits only. Therefore, we can hide the space overhead of the state field by using the upper two bits of the 32-bit offset field in the slot, assuming that we do not need more than 30 bits to address all offsets on the page. This is crucial to keep the fanout of the tree as high as possible. Otherwise, we would introduce more pages than the baseline B-tree, negatively impacting performance.

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.

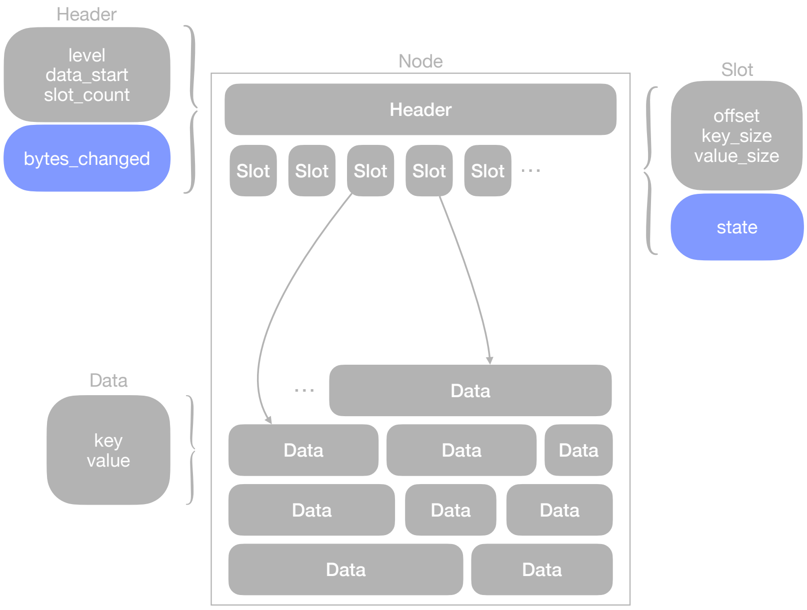


Figure 5.3: A B-tree with tracking enabled. The header is extended by a `bytes_changed` field to track the degree of write amplification on the page. Each slot is extended by a `state` field to track whether the corresponding entry is `Unchanged`, `Inserted`, `Updated`, or `Deleted`.

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` represent changes to inner nodes; therefore, they store keys and PID changes. `LeafDeltas` represent changes to leaf nodes; consequently, 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 extracting 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 on 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 them to insert, update, delete, and look up 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 3B-tree as the 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 3B-tree. While most operations are similar to a standard B-tree [19], we describe how we extend them to support tracking changes and ensure that we do not lose 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.

Lookup

A lookup operation is straightforward in a 3B-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

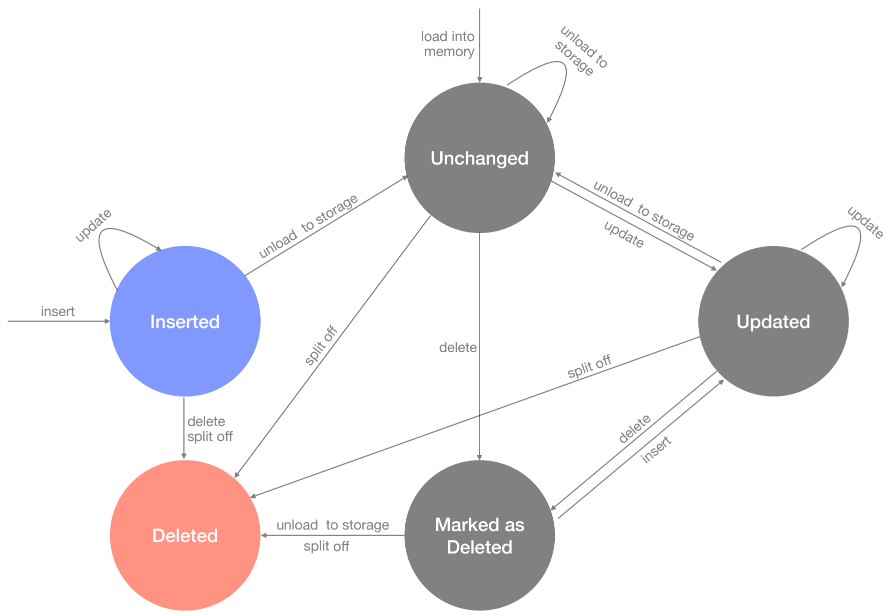


Figure 5.4: The state machine of a slot's state. The state always resembles the delta of a slot in comparison to its image on disk. Therefore, every slot starts in the `Unchanged` state when being loaded from memory. A slot's lifetime starts with an insertion (highlighted in blue). Its lifetime ends when the slot is deleted (highlighted in red). When a slot already exists on disk, we only mark it as `Deleted` to account for removing it when loading the node from storage again. When a page is written to storage, all `Inserted` and `Updated` slots are reset to `Unchanged` state.

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.
The leaf node may 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 essential 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 store the latest key and value of this new slot. If the new entry is deleted again before the page is evicted, we can 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

To perform an update on an entry, 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 a binary search to find the slot with the given key. If the key is found, we update the corresponding entry in the data segment. Our prototype only supports updates to

values of the same size as the old value. During node splits, for example, we must update the child pointers of the entries in the parent node (see Split 5.2).
Alternatively, all updates can be implemented as a delete followed by an insert.

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` or `Deleted`, 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 a 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. Should an insertion for the same key follow, we can reuse the slot marked as `Deleted`, changing its state to `Updated`. To the disk image, it is as if the entry was never deleted in between.

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. Our method is independent of the split strategy and can therefore be easily adapted to different strategies to optimize for concurrency, for example.
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, 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 [19]. 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. Arguably, when splitting a node, about half of the node's entries are modified. We carry enough change on a node to justify a write-out of the page. We could therefore simply set the `bytes_changed` field in the header to the size of the node and stop tracking changes for that page. However, we leave it to the evaluation (see Chapter 6) to determine a reasonable threshold for the degree of change required to justify a write-out. Therefore, to investigate this, we implemented tracking for node splits.

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 `Updated`, those `bytes_changed` were already accounted for.

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. Because 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`. This is not necessary, though, when we create a new node, as new nodes are always written out to storage on eviction. At that point, all slots of the new node return to an `Unchanged` state.

Due to node splits, the `bytes_changed` field can become larger than the node's page size. Assume a node that is full on disk. When loading it into memory, we split it, counting the bytes of half of the slots that we split off. This should set the `bytes_changed` field to about 50% of the node size with fixed-sized entries. However, when we insert new entries into the node after the split, we fill the freed space. This can increase the `bytes_changed` field to 100% of the node size. Assume that all these inserted entries are in the left half of the slots. This can happen because the slots are sorted by key, and we might insert keys that are smaller than the keys in the right half. When splitting the node again, we again count the bytes of half of the slots that we split off, exceeding the node size in the `bytes_changed` field. The key point here is the interpretation of the `bytes_changed` field. It is not a precise measurement of the actual bytes changed. It is an indicator of how much change we have carried on the page since the last write to storage. This helps us to decide whether a write is justified or not.

Tracking Inner Node Changes. 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 now 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. This shows that tracking changes in inner nodes is similar to tracking changes in leaf nodes.

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 essential to keep the fanout of the tree high. We want 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 Chapter 6, we will see that requiring a degree of change of 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 *write_threshold* between 0.0 and 1.0. It determines the minimum *degree_of_change* required to justify a write-out of the page. $write\_threshold = 0.0$ means that we always write out the page to storage and never store deltas. High thresholds (e.g., over 50%) are not expected to be useful in practice, as this will simply move write amplification to the Delta Tree. However, we want to evaluate all ranges of thresholds to understand the trade-offs in Chapter 6. Therefore, we allow the user to configure this threshold freely.

Resetting Deltas

If $degree\_of\_change > write\_threshold$, 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 $degree\_of\_change \leq write\_threshold$, 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. First, we apply all deletions. Then, we cut off any slots that were split off during a node split by using the stored `slot_count` in the delta array. Then, we trigger the compactification of the node to reclaim any space from deleted slots and to defragment the data segment. This ensures that we do not exceed the node size when applying inserts. Then, we apply all updates and inserts. This must be correct, since we only store 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.

Deleting Deltas from the Delta Tree

After applying deltas to a node, we could delete the corresponding deltas from the Delta Tree. However, we decided against this for the following reasons: Assume a node is written to in memory and then evicted. Its deltas are extracted and stored in the Delta Tree. Later, the node is loaded from storage again. Now, this node is needed again, loaded into memory, but only read from until the subsequent eviction. If we had deleted the deltas after applying them, we would need to extract and store them again, dirtying the page. This would introduce unnecessary overhead. Instead, by keeping the deltas in the Delta Tree after applying them, we can discard the page when evicting it again without any changes. We can keep the page in a clean state with the buffer manager.

Therefore, after applying deltas to a node, we keep the corresponding deltas in the Delta Tree. Only when a node is written to again do we update the deltas in the Delta Tree accordingly during the subsequent eviction. When a node is written to storage again, we delete the corresponding deltas from the Delta Tree.

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 3B-tree separately. We also wrote integration tests to test the interaction between the components.

6 Evaluation

This chapter evaluates the performance and effectiveness of the proposed 3B-tree in reducing write amplification compared to traditional B-trees. We describe the experimental setup (see Section 6.1) and workloads (see Section 6.2) used, followed by an analysis of the results (see Section 6.3). The evaluation aims to quantify the trade-offs between write amplification (see Subsection 6.3.1), read amplification (see Subsection 6.3.2), space and memory overhead (see Subsection 6.3.3 and Subsection 6.3.4), and to demonstrate that the 3B-tree achieves significant I/O reductions while maintaining the desirable properties of conventional B-trees.

6.1 Experimental Setup

Since we are interested in write amplification, we observe the number of page writes to disk for different workloads and different memory limitations. This way, our results are not biased by the specific implementation, optimizations, and hardware on which we run our experiments. All experiments were conducted locally on an Apple MacBook Pro with the specification listed in Table 6.1.

Table 6.1: Hardware Specifications

Component	**Specification**
Device	Apple MacBook Pro (2021)
Processor (CPU)	Apple M1 Pro (8-core, up to 3.2 GHz)
GPU	Integrated 14-core Apple GPU
Memory (RAM)	16 GB
Storage	512 GB NVMe SSD
Operating System	macOS Sonoma 14.6.1

6.2 Workloads and Datasets

We evaluate our approach on synthetic and real-world datasets. The real-world dataset allows us to evaluate our approach on realistic data distributions and access patterns. With the synthetic dataset, we can control the data distribution to evaluate the performance of our approach under different scenarios. This allows us to identify the strengths and weaknesses of our approach.
We will be evaluating our system as a whole, benchmarking the database with the different indices to gain a holistic view of the performance. To take a closer look at the indices themselves, we will also be benchmarking the indices in isolation, without the overhead of the database system. For example, when benchmarking the whole database, we have to maintain the table data as well as the index, which introduces additional overhead. When benchmarking the index in isolation, we can focus on the performance of the index itself.

Wikipedia Pageviews Workload

We use an augmented Wikipedia Pageviews dataset [7] as a real-world dataset for our evaluation. The primary goal of this dataset is to evaluate the performance of our approach on realistic data distributions and access patterns. The dataset contains pageview statistics for all Wikipedia articles within a certain time frame. It is publicly available and can be downloaded from the Wikimedia Dumps website[1]. Each pageview record is of the form

```
en Google_Chrome 10406 0
```

consisting of the domain code, the page title, the number of views, and the total response size in bytes.

Data Augmentation

We use the hourly Pageview Wikipedia dataset from **from 1 October 2025 at 00:00 UTC** as our base dataset. We augment the dataset in the following way:

1. We filter out all non-English articles, i.e., we only keep articles with the domain code `en`.
2. For benchmarks with integer keys, we turn the page title into an integer key. For benchmarks with variable-sized keys, we use the original page title as key.

[1] https://dumps.wikimedia.org/other/pageviews/

3. We create a lookup operation for each view of an article, i.e., if an article has 100 views, we create 100 lookups for that article.

4. To generate a mixed workload, we turn a certain percentage of the lookups into updates. This assumes that an article with more views is more likely to be updated.

5. We then shuffle the operations to create a mixed workload.

6. To create a smaller dataset, we take a random fraction of the articles as samples.

To populate the database, we insert all articles from the filtered dataset once. We then run the workload on the indices.

Workload Characteristics

The resulting workload has the following characteristics:

- The dataset contains 59,240 distinct articles, translating to 59,240 distinct keys in the database.
- The workload contains 146,068 views in total.
- The keys follow a Zipf-like distribution, i.e., a small number of articles are very popular and receive a large number of operations, while the majority of articles receive only a few.
- In fact, 40,670 articles (~69%) are viewed only once in the dataset. The most viewed article, *Jon_Stewart*, received 2,998 views (~10% of all views).
- **Update workload**: We transform 7,303 lookups into updates (~5%) by default. When mentioned, we vary the update ratio from 0% to 100%.
- With an update ratio of 5%, we update 5,724 distinct articles (~10% of all articles) in the generated workload, whereas the majority of articles (~86%) are only updated once. The most frequently updated article, *Jon_Stewart*, is updated 132 times (~10% of all updates).
- **Insert workload**: To evaluate the performance of our approach on insert-heavy workloads, we create a workload with 100% inserts as well. In this workload, we insert all articles of the sample into an empty database or index.
- The keys are variable-sized strings with an average length of 20.5 characters, a maximum length of 236 characters, and a minimum length of 1 character.

6.3 Results and Analysis

In this section, we present the results of our benchmark experiments and analyze the performance of the B-tree and 3B-tree indices. Our primary metric for this evaluation is the number of page writes to disk, which we aim to reduce. To analyze the introduced read amplification, we also measure the total I/O operations (page reads + page writes) for both indices (see Subsection 6.3.2).
The goal of this evaluation is to determine whether the 3B-tree can reduce I/O operations compared to a traditional B-tree under different workloads and constraints and to understand the trade-offs involved in using a 3B-tree.

6.3.1 Write Amplification

For a first analysis, we consider the write amplification of the B-tree and 3B-tree indices when running the mixed Wikipedia Pageviews workload with 5% updates on the sample dataset. We run the indices in isolation to focus on the performance of the indices themselves without the overhead of maintaining the table data. We benchmarked with 4 KB pages, a buffer pool of 400 pages (~80% of the B-Tree pages), and a write threshold of 5% (i.e., we only write pages to disk if >5% of the page has been modified in the case of a 3B-tree index). The base B-tree in both indices produces a tree with 502 nodes in total. Both inner and leaf nodes fit 170 entries each, while the average node is filled to ~70%. There is no difference in fanout between the two indices (see Space Overhead 6.3.3). We compare the number of page writes to disk for both indices.
The results are shown in Figure 6.1. We observe a reduction of ~69% in total page writes when using the 3B-tree compared to the B-tree. We can see that in the 3B-tree index, the majority of page writes occur in the Delta Tree rather than its base B-Tree. This shows that we successfully defer small writes across the B-tree and batch them more efficiently in fewer pages in the Delta Tree.
In the following sections, we analyze the performance of the 3B-tree under different scenarios and workloads to understand its strengths and weaknesses.

Different Memory Constraints

To understand the impact of memory constraints on the performance of the 3B-tree, we run the same workload with different buffer pool sizes. To repeat, the workload produces a B-tree of 502 pages in total. We vary the buffer pool size from 25 (~5% of the B-tree) to 500 pages (~99% of the B-tree), while keeping the page size at 4 KB and the write threshold at 5%. The results are shown in Figure 6.2.

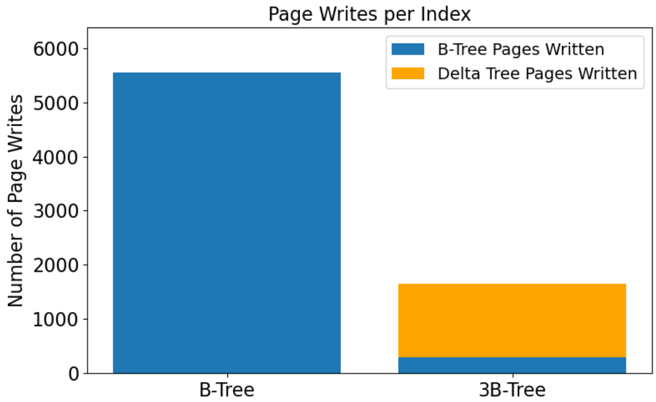


Figure 6.1: Running the mixed Wikipedia Pageviews workload with 5% updates, 4 KB pages, a buffer pool of 400 pages (~80% of the B-tree pages), and a write threshold of 5%. We see a reduction of ~69% in page writes when using the 3B-tree compared to the B-tree.

One key observation is that the 3B-tree degrades more smoothly with lowering memory capacities than the B-tree. When the buffer pool size is reduced from 500 (99%) to 400 pages (80%), the B-tree experiences a sharp increase in page writes. Further reductions in buffer pool size lead to smaller increases in page writes. In contrast, the 3B-tree experiences a steady increase in page writes as the buffer pool size is reduced.
In the following, we analyze the performance of the 3B-tree under different memory constraints in more detail. We differentiate between low (5%-10%), restricted (20%-80%) and high (>99%) memory capacities. Percentages are given with respect to the total number of B-tree pages. Both indices produce the same number of B-tree pages in all settings.

Low Memory Capacity. With a buffer pool of 25 to 50 pages, we can see an increase in page writes with the 3B-tree compared to the B-tree. For example, with 25 buffer pool pages, the 3B-tree writes its B-tree pages 4,993 times less, but introduces 6,506 page writes in the Delta Tree. Two questions arise:

1. *Why do we not reduce page writes?* When we do not have enough memory to cache pages, we cannot accumulate changes in the Delta Tree effectively. The Delta Tree improves page writes only when at least two changes can be accumulated on a Delta Tree page before writing it to disk again. The inserted deltas between page writes of the Delta Tree page must be > 1 on average to see an improvement. When constantly swapping pages in and out of memory, we move the page write

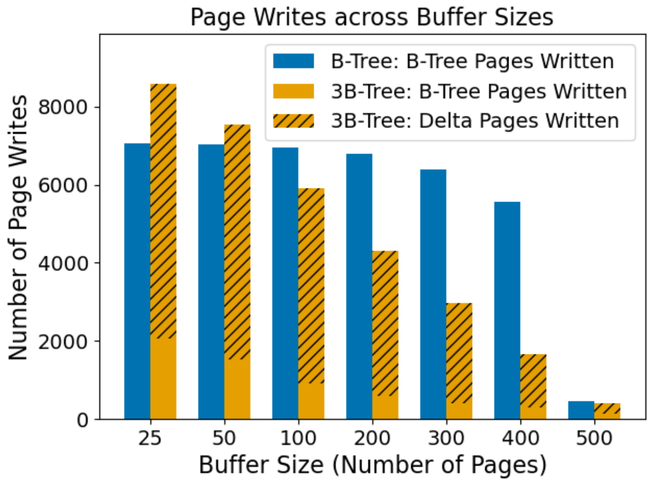


Figure 6.2: The impact of buffer pool sizes on the amount of page writes per index with a 4 KB page size and a 5% write threshold. The B-tree in both indices has 502 pages to address the dataset. The 3B-tree requires enough memory to effectively batch changes in the Delta Tree and degrades more smoothly with lowering memory capacities than the B-tree.

from one tree to another. Since the 3B-tree shares the buffer pool between two trees, memory is more constrained for each tree. This leads to more page traffic overall, further reducing the probability of accumulating changes in a Delta Tree leaf.

2. *Why do we have even more page writes?* When moving page writes from one tree to another, we expect to see the same amount of page writes overall. The average amount of deltas inserted (i.e., B-tree page writes deferred) in between Delta Tree page writes should be equal to 1. However, we see more page writes with the 3B-tree than with the standalone B-tree. The average amount of deltas inserted in between Delta Tree page writes is actually smaller than 1 ($4,993/6,506 \approx 0.77$) with a buffer pool of 25 pages. This means we can dirty more Delta Tree pages than we inserted deltas. Firstly, when inserting a delta, we can trigger a split in the Delta Tree, causing a new inner node to be created or updated in the B-tree. Secondly, we must delete entries from the Delta Tree when B-tree pages have accumulated enough changes to actually write them to disk.

The 3B-tree cannot utilize its batching capabilities without caching. In such a setting, we merely introduce further write amplification in the system since we have more pages to maintain. The 3B-tree requires enough memory to accumulate changes over time. Only with effective page caching can we reduce total number of page writes.

Restricted Memory Capacity. With limited memory capacities of 100-500 pages, we see a significant reduction in page writes with the 3B-tree compared to the B-tree. The 3B-tree can reduce page writes up to ~70% compared to the standalone B-tree. The larger the available memory, the more changes we can accumulate in our Delta Tree before writing them to disk. For example, with a buffer pool of 400 pages, we can see that a Delta Tree page write accumulates 4 deltas on average before being written to disk. We achieve the batching effect that we are aiming for, leading to fewer total page writes. This shows that our method can effectively utilize the available memory to reduce write amplification in memory-constrained settings.

High Memory Capacity. With a buffer pool of 500 pages, almost all pages can be kept in memory; therefore, the effect is less pronounced. With a buffer pool of 502 pages (100% of the B-tree), we could hold the entire B-tree in memory. The Delta Tree is obsolete in this setting, as we can keep all changes in memory without writing them to disk. Since the Delta Tree is empty, and no B-tree pages are loaded from disk once they are in memory, we do not have to perform any lookups into the Delta Tree. Therefore, the 3B-tree does not incur any overhead compared to the B-tree except for tracking changes in the B-tree nodes. However, this overhead is negligible, as shown in Subsection 6.3.4.

Summary. To summarize, the 3B-tree can reduce write amplification in memory-constrained settings where the entire dataset does not fit in memory. This optimization is achievable only when enough memory is available for effective caching. Enough memory must be available to accumulate changes in the Delta Tree over time. An ideal buffer pool size is ~20%-80% of the B-tree pages. Across all memory constraints, the 3B-tree degrades more smoothly than the B-tree, making it a robust choice for varying memory conditions.

Different Write Thresholds

To repeat, the write threshold defines the minimum percentage of a page that has to be modified before we write it to disk. When it is smaller than the threshold, we buffer the changes in the Delta Tree and discard the page. To understand the impact of the write threshold on the performance of the 3B-tree, we run the same workload with different write thresholds. We fixate the page size at 4 KB and the buffer pool size at 400 pages (~80% of the B-tree), as this setting showed the most significant improvements in the previous section. We vary the write threshold from 0% to 50%. Thresholds beyond

50% are not meaningful, as Delta Tree would not be able to accumulate more than one change on leaf nodes. The results are shown in Figure 6.3. For the baseline B-tree index, the write threshold has no effect, as we always write every change to disk immediately. Therefore, we show the results in percentage improvement of the 3B-tree page writes over the standalone B-tree.

Without buffering, at a write threshold of 0%, we write every changed page to disk immediately, just as a traditional B-tree would. As expected, we see no improvement in page writes in this setting. In fact, we can have a few more page writes since we still buffer the empty root of the Delta Tree (it is still read when loading B-tree pages from storage), leaving slightly less memory for the B-tree pages. Starting from a write threshold of 1%, we can accumulate changes in the Delta Tree and reduce page writes by ~70% compared to the B-tree.

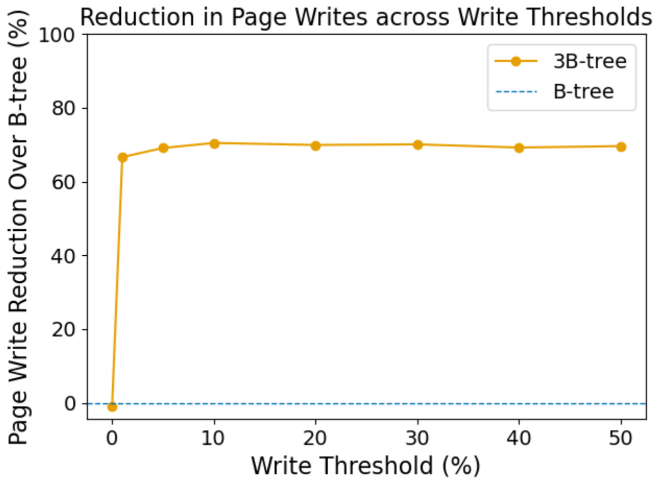


Figure 6.3: The impact of different write thresholds on the amount of page writes per index when running the mixed workload with 5% updates. The page writes of the B-tree index (marked in blue) are constant across all thresholds. The 3B-tree (marked in orange) page writes are shown in percentages of improvement over the B-Tree page writes. At 0% threshold, we write every B-tree node to disk after every change, leading to no improvement in write amplification. With higher thresholds, we can accumulate changes before writing them to disk, leading to significantly fewer page writes.

Expected Diminishing Returns. We do not see a diminishing return with higher write thresholds, as expected. We expected to see smaller improvements with larger write thresholds for three reasons:

1. **Fewer Changes Fit into a Page:** We achieve our goal of reducing page writes when we can accumulate changes on y pages into a single page of the Delta Tree, where $y > 1$. We can then save up to $y - 1$ page writes. However, with larger write thresholds, fewer pages' changes fit into a single page. For example, with a 1% threshold, we can fit 100 pages' changes into a single 4 KB page. At a 25% threshold, we can only fit 4 pages worth of changes in the worst case. The maximum accumulation factor y decreases with larger write thresholds, leading to fewer potential savings in page writes.

2. **Less Likelihood of Batching Changes:** With fewer changes fitting into a single page, the likelihood of batching changes in between Delta Tree page writes decreases. For example, assume a Delta Tree node d that holds x delta arrays for x B-tree nodes. This means that, at some point, x B-tree nodes have been modified. However, this does not directly translate to x saved page writes, because d itself might have been written to disk before all x B-tree nodes were modified. Therefore, we only batch changes that happen between two writes of d. When d holds the changes of 100 B-tree nodes, the likelihood of batching changes is much higher than when d only holds the changes of 4 B-tree nodes. On average, we can save $\frac{y-1}{s}$ page writes per Delta Node d, where s is the number of times d has been written to disk.

3. **More Leaf Nodes in Delta Tree:** Additionally, with larger deltas, we require more leaf nodes in the Delta Tree to address all changes. This introduces more page overhead in the system, leading to more page traffic. This increases s, the number of times a Delta Tree node is written to disk. (The fanout itself is not affected, since inner nodes only hold fixed-size PIDs as keys. The variable-sized deltas are only stored in the leaf nodes, so more pages need to be addressed by the Delta Tree.)

Stagnating Improvements. For all three reasons, we should see shrinking improvements, the larger the write threshold becomes. However, the improvements we see more or less stagnate after a write threshold of 20%. The reason is that we do not see many pages that are modified more than 20% between page writes. In Table 6.2, we can see the distribution of the modification degree for all modified B-tree nodes at the time of eviction. There are two reasons why we only see a small degree of change for most pages:

1. **Zipf-like Distribution:** Due to the Zipf-like distribution of the keys in the workload, a small number of very popular articles receive a large number of operations, while the majority of articles receive only a few. With an update ratio

of 5% of all operations in the workload, most articles are not updated at all (~90%). Naturally, with updates scattered across the index, most pages are only slightly modified between page writes. Therefore, we run the same experiment with a workload of 100% insert to see if the improvements change. The results are shown in Figure 6.4. The insertion workload distributes inserts more evenly across the index, as every article is inserted once. The order of articles is shuffled, so we do not have any locality in the inserts. Indeed, we can observe the diminishing returns more clearly in this workload. However, the improvements still stagnate after a write threshold of 20%. This workload introduces a lot of modification activity per page; therefore, we expect to see a more drastic reduction. The reason is explained in the next point.

2. **Pages are Written Before They Reach High Degrees of Change:** While investigating, we found that pages are often written to disk before they can accumulate many changes. In Table 6.2, we can see the distribution of the modification degree for all modified B-tree nodes at the time their changes are stored in the Delta Tree. We chose a write threshold of 100% for this experiment to see the full distribution of modification degrees. It only shows the pages that are inserted into the Delta Tree, not the pages that bypass the Delta Tree and are directly written to disk. We can see that the Delta Tree only receives small deltas from the B-tree for the vast majority of nodes across workloads. Even in a workload with 100% inserts, where some nodes experience a lot of changes, such as node splits, we see that they do not end up in the Delta Tree. One reason is that **we write new pages to disk immediately** at eviction. Another reason is that node splits combined with many inserts introduce **change degrees over 100%** for a page, which is higher than the write threshold of 100% and therefore is written to disk. Lastly, we sometimes have to write B-tree nodes to disk even though their degree of change is below the write threshold. Sometimes we have to lock the Delta Tree to protect against recursive evictions that want to modify the Delta Tree. For example, when a B-tree node is evicted and its delta is inserted into the Delta Tree, this might cause a split in the Delta Tree. The split causes a new node to be created, which has to be inserted into the B-tree. In this moment, the Delta Tree is in an incomplete state and cannot be accessed by other operations. However, the newly created node can trigger another eviction in the buffer manager, which might evict a modified B-tree node. This modified B-tree node cannot insert its delta into the Delta Tree, as it is currently experiencing structure modifications and therefore is locked. In such cases, **we force B-tree nodes to disk even though their degree of change is below the write threshold**.

   While investigating, we observed that every B-tree node is forced to disk at least once during the workloads for one of these reasons. Therefore, we never see high degrees of change for any page, as we write them to disk before they can accumulate more changes.

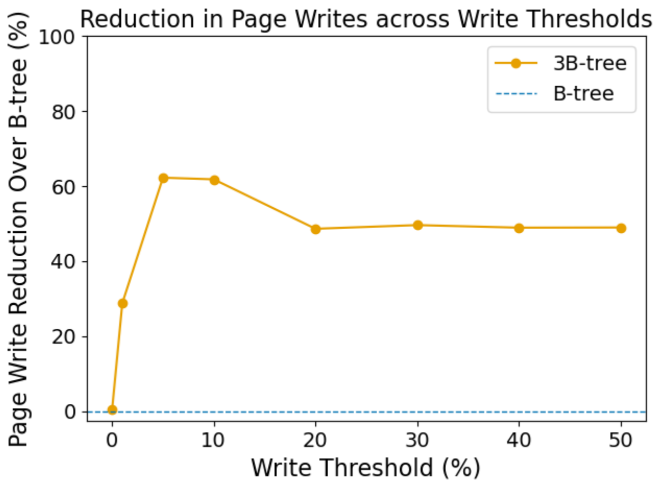


Figure 6.4: The impact of different write thresholds on the amount of page writes per index when running a workload with 100% inserts. Similar to the mixed workload, we see significant improvements with small write thresholds that stagnate with larger thresholds. However, we can see a diminishing return as expected.

A more sequential workload would increase the degree of change for pages that end up in the Delta Tree. However, in such a scenario, traditional B-trees already perform very well. Therefore, we cannot expect to see significant improvements with very high write thresholds in such a workload.

Summary. Our method performs best with small write thresholds. For higher thresholds, we expect diminishing returns due to fewer changes fitting into a page, less likelihood to batch changes, and more leaf nodes in the Delta Tree. However, due to the workload characteristics and the eviction behavior of our implementation, we do not see high degrees of change for most pages. The sweet spot for the write threshold is 5%, where we can achieve significant reductions in page writes for different random write workloads (random updates and random inserts). Therefore, we continue to use this setting in the following experiments.

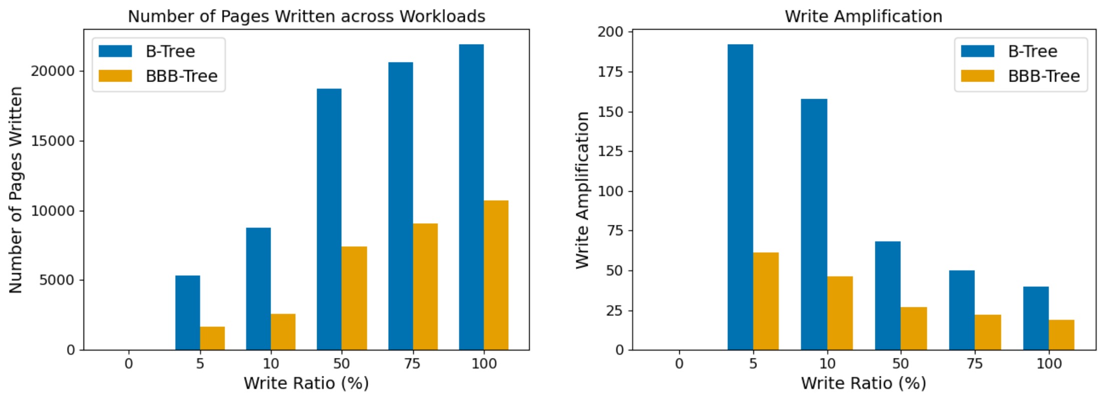
	Num. Pages		
Modified	**5% Updates**	**100% Updates**	**100% Inserts**
0–10%	5274	38530	6462
10–20%	670	1906	1892
20–30%	1	61	586
30–40%	0	1	129
40–50%	0	0	39
50–60%	0	0	156
60–70%	0	0	51
70–80%	0	0	3
80–90%	0	0	0
90–100%	0	0	0
>100%	0	0	0

Table 6.2: Distribution of B-tree pages by their modification degree (percentage intervals) when being unloaded to the Delta Tree with a Write Threshold of 100% for different workloads. The majority of pages are only slightly modified between writes, even with a workload of 100% inserts. Pages are unloaded to disk before they can accumulate more changes.

Different Read/Write Ratios

To understand the impact of different read/write ratios on the performance of the 3B-tree, we run the same workload with different update ratios. We fixate the buffer pool size at 500 pages, the page size at 4 KB, and the write threshold at 5%. We vary the update ratio from 0% to 100%, i.e., we run workloads with only lookups, only updates, and mixed workloads in between. The results are shown in Figure 6.5. We measure write amplification as the ratio of total bytes written to disk over the total bytes of the entries we modify in the index.

In a read-only workload, we perform no page writes at all, as expected. With writes present in the workload, we see a significant reduction in page writes with the 3B-tree compared to the B-tree. The write amplification itself is higher for fewer updates, as we scatter small writes across the index, leading to more pages written for a small set of updates. Here, the 3B-tree can reduce write amplification by up to ~70% by batching them to fewer pages in the Delta Tree. With more updates, more entries are modified in the same pages, leading to lower write amplification. The 3B-tree can still reduce write amplification by ~50% in a workload with 100% updates. Even though we update every entry in the index with this workload, the 3B-tree can defer the page writes until more changes have accumulated, while the B-tree writes its pages to disk immediately upon eviction.



(a) Page Writes per workload. (b) Write Amplification per workload.

Figure 6.5: Page Writes and Write Amplification of the B-tree and 3B-tree with different write/read ratios in the workload. Without updates, we have no page writes and therefore no write amplification. While page writes increase with more updates, write amplification decreases, as more changes happen in the same pages. The 3B-tree can reduce write amplification significantly across all workloads.

6.3.2 Read Amplification

Looking at page writes only does not give the full picture of the performance of the 3B-tree. We need to consider the page reads as well to understand the overall I/O performance. Therefore, we investigate the read amplification of the 3B-tree compared to a traditional B-tree. To repeat, read amplification describes the number of I/O operations we need to perform to answer a query (see Section 2.6). However, for the purpose of this analysis, we specifically refer to **the ratio of page reads we add compared to the traditional B-tree**. In addition to the page writes we save, we analyze the total amount of I/O operations, i.e., the read amplification.

Additional Page Reads. We introduce additional page reads in the following way:

1. Every time we load a B-tree page from disk, we have to look up the PID in the Delta Tree for any changes that need to be applied.

2. Every time we unload a B-tree's dirty page, we either delete its deltas from the

Delta Tree (if the page is written to disk) or insert/update its deltas in the Delta Tree (if the page is not written to disk).

All these operations require a traversal through the Delta Tree, loading nodes from disk if they are not in memory. Additionally, sharing the buffer pool between two trees leads to more page traffic in both trees. We show the sources of additional page reads in the following.

Buffer Pool Analysis. To analyze the introduced page reads, we collect the buffer hits and misses per index under varying buffer sizes on the mixed workload with 5% updates. We show the misses and hits caused by the underlying B-trees and the Delta Tree separately. Each buffer miss directly translates to a page read from disk. The results are illustrated in Figure 6.6. We can observe the two sources of additional page reads (buffer misses) for the 3B-tree:

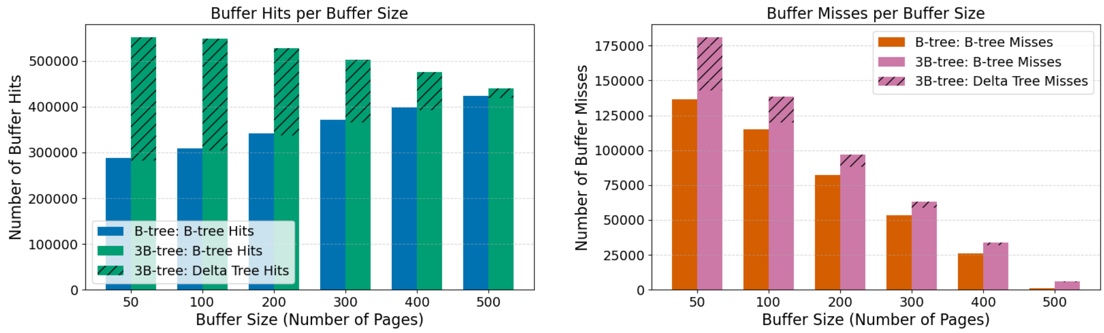
1. **More B-tree Misses:** The 3B-tree has more buffer misses for B-tree accesses compared to the standalone B-tree. Since the buffer pool is shared between the two trees, the 3B-tree has slightly less space to cache B-tree pages, leading to more buffer misses for B-tree accesses. However, the difference is similar across different buffer sizes.

2. **Delta Tree Misses:** The 3B-tree has additional buffer accesses for the Delta Tree. While most of these accesses result in buffer hits with enough memory available, we still have some buffer misses that lead to additional page reads. With small buffer sizes, the B-tree has more page swaps and therefore, the Delta Tree is accessed more frequently, leading to more total buffer accesses (hits and misses).

The key observation is that reads are mostly amplified due to the Delta Tree's buffer misses. Therefore, caching the Delta Tree effectively is critical to keep additional page reads low.

Buffer Size. The smallest amount of page reads is introduced with a buffer size of 200 pages (~40% of the B-tree), where we only introduce ~17% more reads compared to the baseline B-tree. We therefore use this buffer size in the following analysis of total I/O operations. While higher buffer sizes (300-400 pages) lead to more savings in write amplification (see Figure 6.2), we prioritize the reduction of reads, since we perform more reads than writes overall.

Delta Tree Size. If we can cache most of the Delta Tree pages, we only add a small number of additional page reads. By choosing small write thresholds, we can keep the Delta Tree small in relation to its B-tree. We can analyze the maximum size of



(a) Buffer Hits (b) Buffer Misses

Figure 6.6: Buffer hits and misses per index across different buffer sizes, separated into B-tree accesses and Delta Tree accesses. We used a page size of 4 KB, a write threshold of 5% and a mixed workload with 4% updates. The B-tree needs 502 pages to address the dataset in all cases.

the Delta Tree given the write threshold and the size of the B-tree: Every node in the B-tree can produce deltas of the maximum size of $write\_threshold \cdot page\_size$ that need to be stored in the Delta Tree. Therefore, the maximum size of all deltas is $write\_threshold \cdot page\_size \cdot num\_btree\_nodes$, which the Delta Tree has to store. Since the Delta Tree has the same page size as the B-tree, we can calculate the maximum number of leaf nodes in the Delta Tree (we neglect the space overhead of storing deltas in nodes for simplicity here):

$$num\_delta\_leaf\_nodes = write\_threshold \cdot num\_btree\_nodes$$

With a write threshold of 5% for example, the number of leaf nodes in the Delta Tree is around 5% of the total nodes of the B-tree. If 1% of the B-tree nodes are inner nodes, we can assume that inner nodes are always cached in memory and therefore never need to store deltas in the Delta Tree. We can reduce the number of Delta Tree nodes to store deltas from B-tree leaf nodes only:

$$num\_delta\_leaf\_nodes = write\_threshold \cdot num\_btree\_leaf\_nodes$$

We have argued in Section 6.3.1 that the write threshold should be kept small to maximize write amplification reduction. Also, we have seen that the amount of change per page usually remains small between page writes, leading to small deltas even with higher write thresholds. Therefore, we can keep the Delta Tree small compared to the B-tree, allowing us to cache it effectively in memory and keep the Delta Tree page reads low.

Total I/O Operations. Even when we can cache most of the Delta Tree in memory under reasonable memory constraints, we still introduce some additional page reads, as shown in Figure 6.6. To analyze the overall I/O performance, we have illustrated the total I/O operations, separated into page reads and writes, in Figure 6.7 for different update ratios in the workload. We can see that the baseline B-tree performs the same number of reads across all workloads, since the workload size and the index size remain the same. We only change the number of updates in the workload, which affects the number of writes. In contrast, the 3B-tree performs more reads the higher the update threshold is, since we introduce more page reads with the Delta Tree as argued before. However, the more writes we have in the workload, the more writes we can save with the 3B-tree. The read amplification is up to ~33%, while the write reduction is up to ~62%.

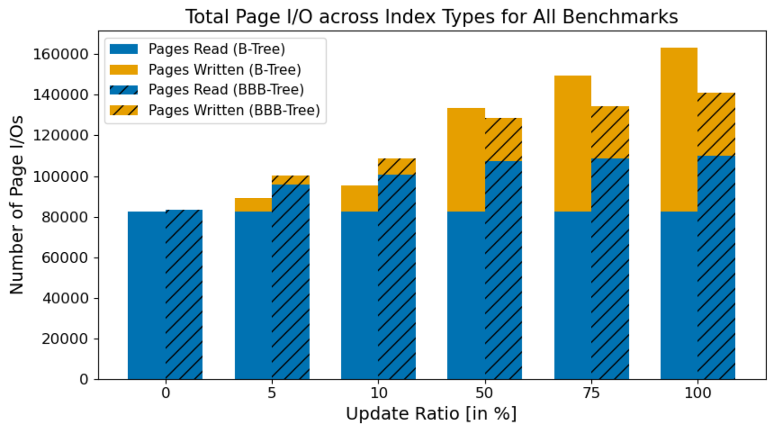


Figure 6.7: Total I/O operations per index, separated into reads and writes with different update ratios in the workload. When we introduce more page reads than we save page writes, we have more total I/O operations. This benchmark runs with a buffer pool of 200 pages, a page size of 4 KB, and a write threshold of 5%. The B-tree has 502 nodes.

These findings show that the 3B-tree is not always beneficial in every scenario. While we can reduce write amplification significantly, we introduce read amplification in the process. Only in workloads where we have enough writes to compensate for the additional reads can we achieve a net reduction in I/O operations.
We can express this trade-off in a payoff condition for our method. Consider a workload consisting of read operations and write operations. The proposed method amplifies reads by a fraction $read\_amplification$ (expressed as a percentage) and reduces writes by a fraction $write\_reduction$ (also in percent).

$$\frac{reads}{writes} < \frac{write\_reduction}{read\_amplification} \tag{6.1}$$

With the empirical values $read\_amplification = 33\%$ and $write\_reduction = 62\%$ from the insert workload, Equation 6.3 simplifies to:

$$\frac{reads}{writes} < \frac{62}{33} \approx 1.88 \tag{6.2}$$

Hence, the method pays off when the workload exhibits fewer than approximately 1.88 read operations per write. For example, in a workload where most pages are written to after being read from disk, we have one read per write most of the time < 1.88. This is the pattern we see in the insert workload, for example, building the index from scratch. Also, in update-heavy workloads, we can see such patterns. For example, in the 50% update workload shown in Figure 6.7, we have about 1.6 reads per write, so about 60% of all read pages are written to before being evicted. Since $1.6 < 1.88$, we reach a net reduction in I/O operations. A scenario where our method performs well.
Depending on the underlying hardware, read and write operations can have different costs. In a setting where random writes are significantly more expensive than random reads, as on the M1 Pro's SSD with 512 GB, we could see faster runs in the 3B-tree even in scenarios where we have more total I/O operations due to a high fraction of read operations but fewer write operations. However, some SSDs can be faster in writing than in reading. We can extend this condition to account for different costs of read and write operations. Let c_r and c_w denote the cost per read and per write, respectively.

$$\frac{reads}{writes} < \frac{c_w}{c_r} \cdot \frac{write\_reduction}{read\_amplification} \tag{6.3}$$

To summarize, the 3B-tree is beneficial in workloads where the majority of pages are written to after being read from disk, i.e., workloads with a low read-to-write ratio. To keep read amplification low, we need to be able to cache the Delta Tree effectively in memory.

6.3.3 Space Utilization

To track changes, we need to store additional metadata in the B-tree nodes. This introduces a fixed overhead per B-tree node.

Tracking Modified Entries. Firstly, we store the state of each entry in the B-tree node to track whether it has been modified. We inject this information into the slot of each entry. As mentioned in Chapter 5, we can use 2 bits to encode each state (Unchanged, Inserted, Updated, Deleted). Since the slot data layout is critical to maximize fanout, we hide this information in the existing slot structure. We use the two most significant bits of the offset to store the state.

Tracking Degree of Change of Nodes. Secondly, we track the degree of change for each node. To that end, we store a counter in each B-tree node that tracks the approximate number of bytes that have been modified since the last write to disk. This counter is used to determine whether the write threshold has been reached and the node needs to be written to disk. We use a 16-bit unsigned integer for this counter, which allows us to track changes up to 65535 bytes. This is sufficient for our use case, as we can track changes up to 100% of a 64 KB page. More realistically, though, we do not want to track changes beyond 50% of a page; therefore, we could even track larger page sizes with this counter.
This means that our implementation introduces a fixed overhead of 2 B per node. However, this does not affect the fanout of the index significantly. In some scenarios, it does not affect the fanout at all. For example, assume we store entries of 16 B each (8 B key, 8 B value) on 4 KB pages as we do in our previous benchmarks. On a leaf node, each entry requires 24 B, a slot of 8 B (4 B offset, 2 B key size, 2 B value size), and the data of 16 B (8 B key, 8 B value). Therefore, our B-tree leaf nodes can hold 170 entries on a 4 KB page. With a header of 12 B, a full leaf node occupies $12B + 170 \cdot 24B = 4092B$. With the additional overhead of 2 B per node, $4092B + 2B = 4094B < 4096B$, we can still fit the same amount of entries on a page. The same applies to inner nodes. In this scenario, we do not lose any fanout due to the additional overhead. However, in other scenarios, we might lose one entry per node due to the additional overhead. With a 4 KB page and 170 entries per node, losing one entry means a loss of ~0.6% in fanout.

6.3.4 In-Memory Overhead

To analyze the memory overhead of tracking changes on the pages in memory, we run the mixed Wikipedia Pageviews workload purely in memory, once on a B-tree with tracking enabled and once on a B-tree without tracking. The B-tree without tracking does not have a Delta Tree attached, so we can see the pure overhead of tracking changes in the B-tree nodes. We fixate the buffer pool size to 600 pages to ensure that the whole index fits in memory. This way, we can look at the pure memory overhead without the influence of page swaps. The results are shown in Table 6.3. We see that the overhead of tracking changes in memory is negligible.

Index Type	Time
B-tree without Tracking	13.54 ms
B-tree with Tracking	13.55 ms (+0.07%)

Table 6.3: Time to insert the Wikipedia Pageviews workload on an empty B-tree with and without tracking changes. The buffer manager fits the whole index in memory. The memory overhead of tracking changes is negligible.

6.3.5 Variable-Sized Entries

To analyze the effect of variable-sized entries, we run the same Wikipedia Pageviews workload, but this time with the variable-sized `page_title` as key. As expected, the index size increases due to the larger average key size of ~20.5 B compared to the fixed size keys of 8 B. We observed that the B-tree has 757 nodes with variable-sized keys compared to 502 nodes with integer keys. To remove the influence of different index sizes, and therefore more pages, we run the benchmarks with a buffer pool that fits ~40% of the B-tree in memory, respectively (300 pages for fixed-sized keys, 200 pages for variable-sized keys). We run the same workload with 100% insertions. The results are shown in Figure 6.8.
With the different buffer sizes, we achieve similar page reads and writes for the baseline B-trees. However, with variable-sized keys, the 3B-tree achieves a smaller reduction in page writes compared to fixed-sized keys. With variable-sized keys, we can reduce page writes by ~56% compared to ~64% with fixed-sized keys. This has two reasons:

1. **Larger Delta Tree:** With variable-sized keys, we generate more B-tree nodes (757 vs 502). Therefore, we have more pages that can produce deltas, leading to a larger Delta Tree. Our Delta Tree implementation stores the updated entries; therefore, variable-sized entries lead to larger deltas, increasing the size of the Delta Tree further. While both Delta Trees are ~3.8% of the size of their respective B-trees, the Delta Tree is larger in absolute terms. A larger Delta Tree leads to more page traffic, and therefore, more page writes.

2. **Larger Modification Degrees:** With variable-sized keys, we have more variance in the modification degree of pages, as shown in Table 6.4. Therefore, we have fewer pages with a modification degree below the write threshold of 5%. This leads to fewer pages being buffered in the Delta Tree, and therefore, a smaller overall write reduction.

While a conclusion could be that variable-sized keys require a larger write threshold to achieve similar write reductions, we could not confirm this hypothesis with our

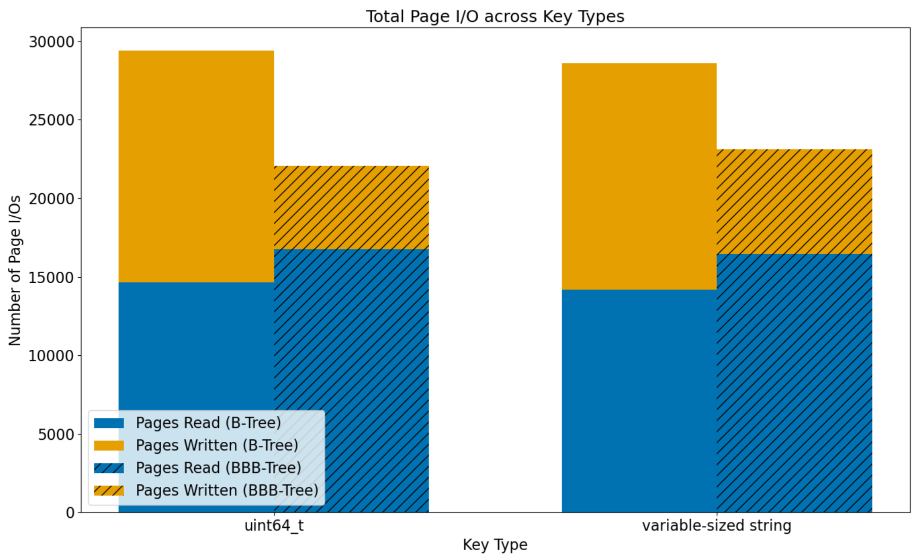


Figure 6.8: Total I/O operations per index, separated into reads and writes with 100% insertions of variable-sized string keys and fixed-sized integer keys. The buffer pool fits ~40% of the B-tree in memory, the page size is 4 KB and the write threshold is 5%. While we can reduce total I/O operations in both cases, the write reduction is smaller with variable-sized keys.

experiments. Larger write thresholds lead to larger Delta Trees and no further reduction in write amplification. To summarize, our method works with variable-sized keys as well, but the write reduction is smaller, mainly due to the larger modification degrees of pages.

	Num. Pages	
Modified	**Fixed-Size Keys**	**Variable-Size Keys**
0-5	12487	11013
5-10	2470	3303
10-15	64	180
15-20	5	26
20-25	1	4
25-30	0	2
30-35	0	0
35-40	0	1
40-45	0	10
45-50	104	239
50-55	446	477
55-60	59	139
60-65	15	43
65-70	13	23
70-75	11	19
75-80	5	16
80-85	2	13
85-90	7	9
90-95	6	11
95-100	9	13
>=100	109	163
Total Evictions	15813	15705

Table 6.4: Distribution of B-tree pages by their modification degree (percentage intervals) when being evicted with a write threshold of 5% for fixed-sized keys and variable-sized keys in a workload with 100% insertions. While both have similar number of page evictions, the variable-sized keys have more variance in the modification degree and fewer pages with a degree of change below the write threshold of 5%.

7 Discussion

This chapter discusses the results of the evaluation in the context of the research objectives. It summarizes the findings on how the proposed 3B-tree affects write amplification, read performance, and overall efficiency compared to traditional B-trees. Furthermore, it highlights the limitations of the current design and suggests areas for future research.

7.1 Summary of Findings

The proposed 3B-tree successfully reduces write amplification in B-trees by buffering and batching updates in a secondary structure called the Delta Tree. The 3B-tree consistently demonstrates lower write amplification than the baseline B-tree while maintaining comparable space overhead and moderate read amplification. When evaluated on the mixed Wikipedia Pageviews workload, the 3B-tree achieved up to **70% fewer page writes** compared to the baseline.

An analysis of **read/write trade-offs** revealed that the 3B-tree introduces read amplification due to additional page reads for the Delta Tree whenever the B-tree reads or writes pages. This cost is offset when workloads exhibit enough page write savings, i.e., in insert-heavy or update-intensive workloads. Under such conditions, total I/O operations are reduced, leading to net performance gains. Conversely, read-dominant workloads experience regressions since the small amount of page write savings cannot compensate for the introduced read overhead.

Memory availability was found to be a decisive factor. In low-memory capacities ($\leq$10% of the index fits in memory), the 3B-tree can produce more page writes since we cannot batch changes when constantly swapping pages in and out of memory. Batching is only possible when enough memory exists to cache the Delta Tree effectively. With **moderate memory capacities** (>10% of the index fits in memory), the 3B-tree can achieve substantial write reductions of up to 70%. With high memory capacities (>99% of the index fits in memory), the use of a 3B-tree is not meaningful, as the B-tree rarely performs page writes. The introduced overhead of the Delta Tree lookups outweighs the small amount of write savings in this scenario. Overall, batching changes in memory is beneficial once memory is limited, but sufficient space exists to cache updates. Across all memory scenarios, the 3B-tree shows a clear advantage over the baseline B-tree:

it degrades smoothly with decreasing memory, while the B-tree's write amplification increases sharply.
The 3B-tree performed best with **small write thresholds** between 1% and 10%, balancing the number of buffered deltas per node and the size of the Delta Tree. At these thresholds, the structure accumulates small, scattered changes efficiently, avoiding unnecessary full-page writes. Higher thresholds lead to larger Delta Trees, since each node fits fewer B-tree pages, diminishing the overall batching effect and introducing more page overhead.
The experiments with **variable-sized keys** confirmed that the 3B-tree remains effective, although the write reduction was smaller (approximately 8% less compared to fixed-sized keys). Variance in key length increased the number of nodes exceeding the write threshold, reducing opportunities for batching. However, the Delta Tree size remained compact (same proportion to the respective B-tree), demonstrating that the overhead of managing variable-length entries is modest. Larger write thresholds do not improve write reduction for the reasons discussed above.
In summary, the 3B-tree reduces write amplification substantially across workloads. However, due to the introduced read amplification, the overall I/O reduction depends on the workload mix. The design achieves its intended balance between reduced writes, read amplification, and limited space overhead.

7.2 Limitations

While the 3B-tree demonstrates strong potential, some limitations constrain its current applicability:

1. **Memory Dependence.** The method's effectiveness relies on the ability to cache portions of the Delta Tree in memory. In extremely memory-limited environments, the additional lookups into the Delta Tree can increase total I/O activity and negate the intended benefits.

2. **Read Amplification.** By introducing an additional layer for change buffering, each page load may trigger additional Delta Tree page loads. Although the observed read amplification is moderate, it can increase total I/O in read-intensive workloads.

3. **Workload Diversity.** The experiments were run on a single dataset (Wikipedia Pageviews) with Zipf-like key distribution. Additional datasets with different locality and access patterns (such as sequential or uniformly random) would be needed to generalize the results.

7.3 Future Directions

The findings open several directions for further research and system development:

1. **Smart Eviction Policies.** Implementing page eviction strategies that consider the access patterns and update frequencies could enhance the performance of the 3B-tree. In our implementation, pages were evicted randomly, which does not reflect real-world buffer management strategies. More sophisticated policies, such as LFU or 2Q [19], could be explored to improve cache hit rates. Prioritizing Delta Tree pages that contain frequently updated nodes could reduce read amplification significantly.

2. **Alternative Buffering Data Structures.** The 3B-tree could benefit from exploring different data structures for the Delta Tree. We decided on a B-tree for simplicity, but other structures could offer better batching or lower overhead. For instance, one could write deltas into a log-structured format. We would naturally achieve higher locality for batching deltas. In a B-tree structure, we risk scattering deltas across multiple pages. More ideally, if temporally close deltas are also spatially close, we could achieve better batching. In contrast to an LSM-trees, where data is organized in levels, the 3B-tree only requires searching the Delta Tree during B-tree page reads and writes. However, it could incur higher read amplification, so that a careful design would be necessary.

3. **Bloom Filters for Delta Tree Lookups.** Most read amplification arises from additional Delta Tree lookups during B-tree page reads. However, not all B-tree nodes have pending updates in the Delta Tree. In a read-dominant workload, many Delta Tree lookups may be unnecessary. A Bloom filter could be maintained in memory to quickly check whether a B-tree node has pending updates in the Delta Tree. A technique often applied to skip unnecessary level lookups in LSM-trees. This would avoid unnecessary Delta Tree lookups when reading B-tree nodes that have no pending updates. This poses an opportunity to reduce read amplification significantly.

4. **Tracking Updates in Table Data.** While this work focused on indexing structures, the same principles could be applied to the underlying table data. In write-intensive workloads, buffering updates to table pages in a similar Delta Tree structure could reduce write amplification at the data level.

5. **Levelled Delta Trees.** Another strategy to bring temporally close updates close together spatially is to introduce multiple levels of Delta Trees. Whenever a Delta Tree page itself is below a certain write threshold, it could be buffered into a

higher-level Delta Tree. This would allow for multi-level batching of updates, potentially reducing the number of page writes even further. The tree on each level would become significantly smaller, making it easier to cache in memory.

The primary objective of future work should be to refine the 3B-tree design to minimize read amplification, the primary source of overhead in read-heavy workloads. Reducing read overhead would maximize the overall I/O savings across a broader range of workloads, establishing the 3B-tree as a robust replacement for traditional B-trees.
In conclusion, the 3B-tree demonstrates that *write amplification in B-trees can be significantly reduced through buffered batching without significant structural changes*. Its design preserves the simplicity of B-trees while improving write efficiency, making it a practical foundation for future write-optimized indexing in modern storage environments. Keeping read costs low while minimizing write costs remains a key challenge.

8 Conclusion

This thesis set out to address the problem of excessive write amplification in traditional B-tree index structures, which impacts efficiency and storage longevity in modern database systems. While B-trees remain the most widely used indexing structure due to their efficient lookups and range queries, their page-oriented update model causes high write costs under random or mixed workloads. The objective of this work was therefore to design, implement, and evaluate a method that reduces write amplification in B-trees while preserving their core advantages: simplicity, concurrency, and read performance.

8.1 Recap of Contributions

To achieve this goal, this thesis introduced the 3B-tree, a lightweight, write-aware extension of the traditional B-tree. The key idea was to introduce an intermediate buffering structure, the Delta Tree, that intercepts page evictions and selectively batches modifications instead of performing immediate full-page writes to external storage. By deferring and consolidating small changes, the 3B-tree minimizes unnecessary I/O operations without altering the internal layout and fundamental operations of the B-tree.
The main contributions of this work can be summarized as follows:

- **A lightweight buffering layer for B-trees:** The Delta Tree acts as a transparent, non-intrusive extension between the B-tree and buffer manager, reducing write amplification through selective batching and deferred persistence.
- **A detailed implementation and evaluation:** The 3B-tree was implemented in C++ and benchmarked under realistic workloads derived from the Wikipedia Pageviews dataset. Experiments investigated write amplification, read overhead, and space utilization.
- **Empirical evidence of reduced write amplification:** Across a variety of configurations, the 3B-tree achieved up to 70% fewer page writes than a standard B-tree. This confirms that lightweight batching can significantly mitigate write amplification.

- **Identification of workload-dependent trade-offs:** Caching is the fundamental idea of this method; therefore, a certain memory availability is required. Further, the evaluation revealed that the balance between write reduction and read amplification affects overall performance. The 3B-tree performs best in moderately sized buffer pools and update-heavy workloads.

The reason B-trees are so widely used is their balanced performance characteristics across a range of workloads. However, their random write performance lags behind the otherwise strong efficiency. With the 3B-tree, we sacrifice some read performance but enable substantial write reductions in write-intensive scenarios. We argue that this evens out the performance landscape between read and write operations in traditional B-trees.
The goal of this thesis has thus been met by demonstrating that write amplification in B-trees can be substantially reduced through a simple buffering extension.

8.2 Significance of Results

The results of this work demonstrate that substantial write amplification reductions can be achieved without resorting to invasive structural changes, a limit on concurrency, or a prohibitive read overhead, such as LSM-trees or Bε-trees. Moreover, by lowering write amplification, the 3B-tree indirectly mitigates device-level wear on SSDs, contributing to longer hardware lifespans and reduced operational cost.

8.3 Outlook and Final Thoughts

The promising results of the 3B-tree open multiple paths for future research. Extending the design to a fully concurrent, crash-consistent implementation would enable integration into production database systems. Additionally, levelled Delta Trees or log-structured buffers instead of trees could further optimize the trade-off between write reduction and read overhead.

In conclusion, this thesis demonstrates that *reducing write amplification in B-trees is feasible*. The 3B-tree is a step towards bridging the gap between high-performance and write-efficient indexing.

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] W. Foundation. *Wikimedia Pageview Dataset*. 2025. URL: `https://dumps.wikimedia.org/other/pageviews/2025/` (visited on 10/09/2025).

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

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

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

[16] 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.

[17] 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.

[18] 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.

[19] 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).

[20] 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.

[21] 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.

[22] 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.

[23] C. Sauer, G. Graefe, and T. Härder. “FineLine: log-structured transactional storage and recovery.” In: *Proceedings of the VLDB Endowment* 11.13 (2018), pp. 2249–2262.

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

[25] 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.

[26] 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.