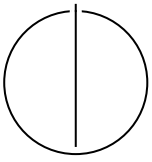
SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Reducing Write-Amplification in B-Trees

Marlene Bargou



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

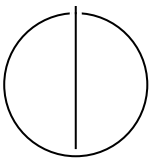
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Reducing Write-Amplification in B-Trees

Reduzierung der Schreib-Verstärkung in B-Bäumen

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



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

Munich, 03.11.2025 Marlene Bargou

Acknowledgments

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

Abstract

B-Trees are the most used data structure in modern database systems, due to their efficient access patterns and excellent lookup performance on large volumes of data. However, B-Trees perform suboptimal under random writes, a particularly common pattern for secondary indexes. Such workloads introduce write amplification in B-Trees, a phenomenon where the amount of data written to storage is significantly larger than the amount of data that logically changed. As a result, B-Trees suffer increased latency, reduced throughput, and premature device wear with write-intensive workloads.

As an alternative, Log-Structured-Merge-Trees (LSM-Trees) were proposed, which trade off low read performance for high write performance. However, this trade makes LSM-Trees unsuitable for generic database systems that require excellent read performance. Other attempts to reduce write amplification in B-Trees either reduce concurrency, impact read performance or rely on hardware-specific features, limiting their effectiveness and applicability.

This thesis introduces a lightweight buffering layer that minimizes the frequency and volume of write operations to external storage by reducing write amplification. We hereby enable high performance under random writes, while sustaining all the benefits of traditional B-Trees.

We implement the proposed structure, evaluate its performance under different workloads, and compare it against state-of-the-art methods. Compared to LSM-Trees, our approach offers [...]. Compared to traditional B-Trees, our method achieves [...] while maintaining excellent read performance.

These results suggest that write-aware B-Tree optimizations can extend the lifespan of storage devices and significantly improve the efficiency of write-intensive applications; contributing to the broader effort of designing storage-efficient data structures suited for modern hardware.

Contents

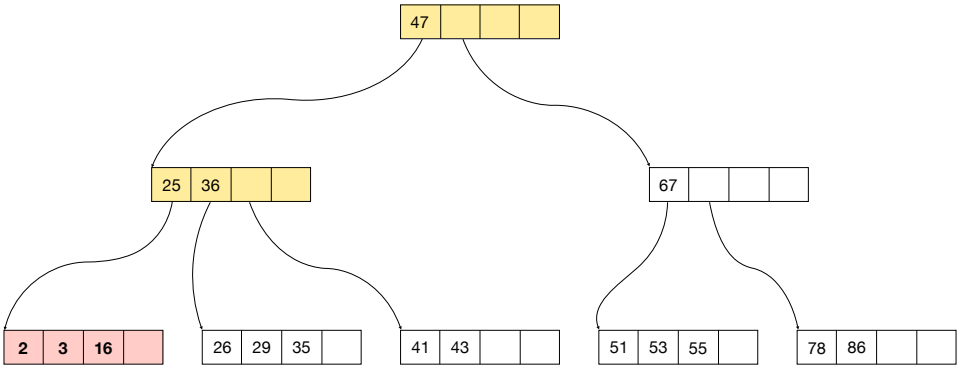
1. Introduction

1.1. Motivation

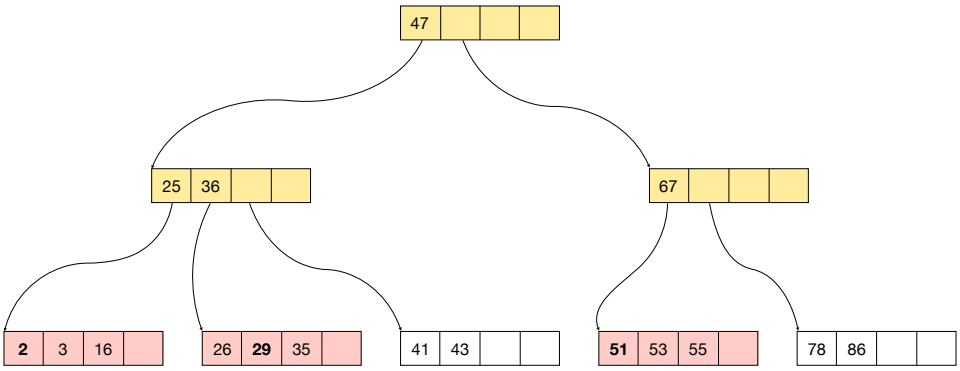
Efficiently managing large data sets is a core requirement for database systems. The largest overhead in beyond memory systems lies in Input/Output (I/O) operations on external storage. Therefore, minimizing I/O operations remains the fundamental premise for designing high performance modern Database Management Systems (DBMS). Primarily, this is done by caching frequently accessed pages in Dynamic Random Access Memory (DRAM) using a buffer manager [**leis2018leanstore**]. All data in the system are stored in pages, the buffer manager can uniformly serve and cache pages for all components in the system. This modular design allows a separation of concerns between the buffer manager and its users, such as indexes and data structures. However, this also means that the buffer manager is agnostic of its user's access patterns. While the buffer manager minimizes the number of I/O operations to the best of its knowledge, every component in the systems must design its access patterns to be as efficient as possible.

A prominent example of such a component is the B-Tree. B-Trees are the dominant data structure for indexing large datasets in disk-based DBMS due to their excellent lookup performance, support for range queries, and simplicity. However, random writes, a particularly common pattern for secondary indexes, lead to inefficient access patterns that a buffer manager cannot hide for out-of-memory workloads.

B-Trees organize their nodes as pages. Due to their sorted order, accessing random keys leads to random accesses of different pages that need to be loaded into the buffer. At eviction time, each modified page requires a full rewrite to storage, even if only a small portion of the page changed. Figure 1.1 illustrates this effect by comparing a sequential and a random update pattern in a B-Tree. We consider three updates to the tree. In the sequential update, only three pages are read and one of them is written. In the random update, six pages are read and three of them are written. Merely changing the access pattern from sequential to random leads to a threefold increase in the amount of data written to storage. If we assume that each page is 4 KB, the sequential update requires one storage write of 4 KB. The random update requires three storage writes of 12 KB in total. Random writes introduce *write amplification*, a phenomenon where the amount of data written to storage is significantly larger than the amount of data that



(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.

logically changed.

Write amplification inflates I/O operations, wastes bandwidth, and ultimately increases latency in bandwidth-bound scenarios. For example in cloud environments, where storage can be remote, an unnecessary network round-trip directly translates to increased latency and reduced throughput in the system.

Additionally, Solid State Drives (SSD) come with their own write amplification due to their flash translation layer performing garbage collection [**haas2023modern**]. This leads to a multiplication of unnecessary physical writes. Lastly, due to SSD's limited write endurance, this not only degrades performance but also wears out the device faster.

In summary, to design a truly efficient, high performance system, we must minimize I/O operations in all components of the storage stack. In this thesis, we focus on closing the efficiency gap in B-Trees by reducing write amplification.

1.2. Problem Statement

While B-Trees are the backbone of indexing in modern storage engines, their in-place updates introduce significant write amplification, leading to performance degradation and reduced device lifespan.

LSM-Trees address write costs by always writing sequentially, but they introduce high read amplification and complex tuning requirements, making them unsuitable for general-purpose database systems.

Bϵ-Trees buffer and batch updates starting from the root and propagating them down the tree to reduce write amplification. Firstly, this introduce two searches per node, one for the next pivot and one for a buffered update for the looked up key. Secondly, the reduced space for pivots in each node reduces the fanout of the tree, leading to taller trees and more I/O operations per lookup. Most importantly though, it significantly limits concurrency in the data structure, as the hottest nodes are locked for longer periods of time to write the update messages, reducing throughput in the tree.

We identify a research gap for a B-Tree variant that effectively reduces write amplification while preserving the excellent query efficiency and concurrency traits of traditional B-Trees. This thesis addresses the problem of write amplification in B-Trees while preserving their query efficiency and operational practicality.

1.3. Objectives

The primary objective of this thesis is to design, implement and evaluate a B-Tree variant that reduces write amplification while maintaining the high read performance

and concurrency of traditional B-Trees. We focus on the following research questions:

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

While we reflect on significant hardware trends in this thesis, such as the increasing prevalence of SSD, we do not target optimizations for specific hardware features. Instead, we aim to design a solution that is broadly applicable across different storage media and hardware configurations.

We also do not aim to outperform LSM-Trees in write-intensive workloads, as they are fundamentally optimized for such scenarios, trading off lookup performance.

While the page-oriented design is one reason for I/O amplification in B-Trees in general, we do not aim to redesign the data structure from the ground up. Instead, we focus on a lightweight extension to the traditional B-Tree that can be integrated into existing systems with minimal changes.

1.4. Contributions

This thesis introduces 3B-Tree, a B-Tree variant that incorporates a lightweight buffering layer to minimize write amplification. The buffering layer aggregates and batches write operations, reducing the frequency and volume of writes to external storage. This design preserves concurrency and read performance of traditional B-Trees while effectively mitigating write amplification. We minimize I/O operations to those strictly necessary.

We hereby contribute to the broader effort of minimizing overhead of beyond memory systems and designing efficient, high-performance datbase systems for modern hardware.

2. Background

2.1. Database System Architecture Overview

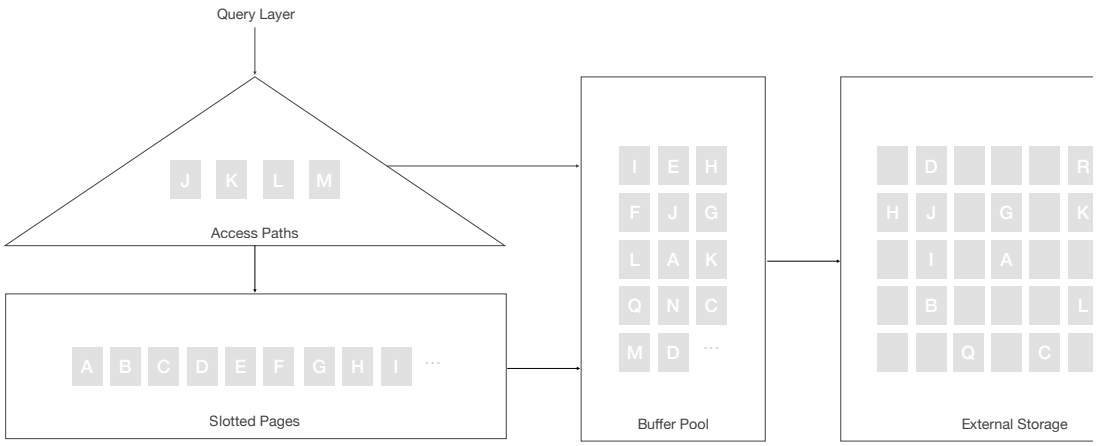


Figure 2.1.: The storage and access layer of a database system.

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

tuple can be retrieved from the slotted pages. The TID encodes the page identifier 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.

2.2. Index Structures

The task of an index structure is to efficiently lookup the position of a tuple in storage. Index structures map a key to a constant, unique TID. A TID never changes for a tuple. Keys can be arbitrary types and therefore of fixed or variable size, such as integers or strings. We will consider both within this thesis. When the key of a tuple changes, the index must be updated to reflect the new key.

Some key-value stores directly map keys to tuples within their index structure, omitting the indirection via TID and slotted pages. However, in a general purpose DBMS, we typically want to support multiple indexes on the same data. If we stored tuples directly in the index, we would need to update all indexes when a tuple changes. Therefore the access and storage layer are decoupled via TIDs.

Indexes can be classified into primary and secondary indexes. A primary index is built on the primary key of a table, which uniquely identifies each tuple. A secondary index is built on a non-primary key, which can be non-unique.

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

2.3. B-Trees

B-Trees [**bayer1970organization**] are a self-balancing tree data structure that maintains sorted data and allows for insertion, deletion, and search operations in logarithmic time, $\mathcal{O}(\log n)$. A B-Tree is organized in fixed-size pages, called nodes. These pages are transferred transparently by the buffer manager between external storage and DRAM. Each node can split off a sibling once is it 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

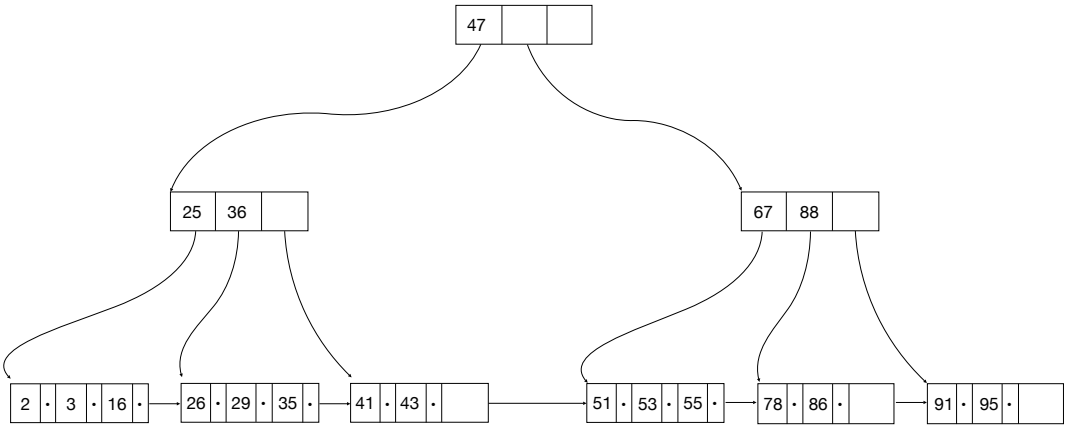


Figure 2.2.: A B+-Tree. Child pointers are repesented as arrows. Values (the TID) are represented as bullet points •.

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

When searching for a key in the tree, we start at the root node. On each node, we perform a binary search to find the appropriate pivot key and follow the corresponding child pointer. We stop when we reach a node with the desired key.

B+-Trees

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

Node Size & Fanout

The node size is a crucial parameter in the design of a B-Tree, as it affects the height of the tree. The height of a B-Tree is $\log_f(n)$, where f is the fanout (the number of children per node) of the tree and n is the number of keys in the tree. The node size determines how many entries fit into a node, which directly impacts the fanout f of the tree and therefore its height. Larger nodes lead to more entries per node, increasing the fanout. When we can address more children per node, we need fewer levels in the tree to address the same number of keys. Therefore, larger nodes lead to a more shallow tree. Since every lookup requires a traversal from the root to a leaf node, a more shallow tree leads to fewer pages involved in the lookup. Thus, larger nodes lead to fewer I/O operations per lookup. Additionally, since we need fewer distrinct pages, we induce less page management overhead in the buffer manager. This is particularly beneficial for analytical workloads, which often perform large scans and are interested in large parts of the data. However, workloads that perform many updates and point queries, we are often 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. Research on modern SSD shows that a good compromise is given for page sizes of 4 KB [**haas2023modern**]. While we will be investigating different node sizes in our evaluation, this parameter is not the primary focus of this thesis. Instead, we focus on reducing I/O amplification in B-Trees at any page size, but larger nodes profit more significantly from our approach.

2.4. Log-Structured-Merge-Trees

LSM-Trees **[oneil1996log]** are a write-optimized data structure that trade off read performance for high write performance. They consist of multiple levels of sorted runs, where each level is larger than the previous one. New data is first written to an in-memory component (often called C0) and then flushed to disk as a new sorted run in the first level (C1). When a level reaches its size limit, it is merged with the next level (C2), creating a new sorted run in that level. This process continues recursively, with each level being merged into the next one when it reaches its size limit. Since writes are sequential and batched, LSM-Treess can achieve high write throughput.

LSM-Trees are especially well-suited for remote object storage like S3. Objects are immutable and updates require rewriting the entire object, analogous to runs in LSM-Trees.

When performing a point lookup in LSM-Trees, we need to search through all levels, starting from the in-memory component down to the last level on disk. This is because the key we are looking for could be in any of the levels. To optimize lookups, LSM-Trees often use Bloom filters to check if a key is present in a level before performing a more expensive search. However, Bloom filters themselves can become large when indexing large data sets. Range queries always require a full search of all levels. Consequently, LSM-Trees exhibit high read amplification, as we need to read multiple pages from different levels to find a key. LSM-Trees are suitable in scenarios where index updates dominate lookups and are therefore unsuitable for a general purpose DBMS.

2.5. External Storage Characteristics

2.6. Write Amplification

3. Related Work

(Position each data structure in their attempt to solve a certain problem and how it does not solve ours yet/well)

3.1. B-Epsilon Trees

3.2. Bw-Trees

3.3. In-Memory B-Trees.

see https://www.cs.cit.tum.de/fileadmin/w00cfj/dis/papers/btrees-are-back.pdf

3.4. Write-Optimized B-Trees

3.5. Fractal Trees

3.6. Buffer Trees

3.7. LSM Trees

3.8. Summary

4. Method

4.1. Design Goals

4.2. High-Level Description of the Optimization

4.3. Data Structure Modifications

4.4. Algorithms for Insertion, Deletion, and Rebalancing

4.5. Theoretical Implications on Write Amplification

5. Implementation

5.1. Environment and Tools Used

5.2. System Architecture

5.3. Data Layout and Page Management

5.4. Caching and Buffering Strategies

5.5. Challenges and Trade-offs

6. Evaluation

6.1. Experimental Setup and Datasets

6.2. Baseline Systems for Comparison

6.3. Performance Metrics

6.4. Results and Analysis

6.5. Discussion

7. Discussion and Future Work

7.1. Summary of Findings

7.2. Limitations

7.3. Future Directions

7.4. Potential Applications

8. Conclusion

8.1. Recap of Contributions

8.2. Final Thoughts

A. Appendices

A.1. Pseudocode

A.2. Additional Graphs and Tables

A.3. Configuration Files / Benchmarking Scripts

Abbreviations

LSM-Trees Log-Structured-Merge-Trees

SSD Solid State Drives

DBMS Database Management Systems

DRAM Dynamic Random Access Memory

I/O Input/Output

TID Tuple ID

List of Figures

List of Tables