Otto-von-Guericke-University Magdeburg
Faculty of Computer Science

Master Thesis

Implementation and Evaluation of a Unified B$^{\epsilon}$-Tree for Heterogeneous Storage Systems with Non-Volatile Memory

Author:
Adem Zarrouki
Matriculation No.: 231613

Advisors:
Prof. Dr. rer. nat. habil. Gunter Saake
Database and Software Engineering Group
Otto-von-Guericke University Magdeburg, Germany

Dr.-Ing. Balasubramanian Gurumurthy
IBM Research and Development
Böblingen, Baden-Württemberg, Germany

M.Sc. Sajad Karim
Database and Software Engineering Group
Otto-von-Guericke University Magdeburg, Germany

Magdeburg, October 1, 2025

Zarrouki, Adem:
Implementation and Evaluation of a Unified B^{ϵ}-Tree for Heterogeneous Storage Systems with Non-Volatile Memory
Master Thesis, Otto-von-Guericke University Magdeburg, 2025.

Abstract

The amount of data generated and processed by modern applications continues to grow each year. The volatile memory (DRAM), despite various improvements made to it over the years, will reach a point where it will not be able to handle this large amount of data. Non-volatile memory (NVM), also called persistent memory (PM), non-volatile random access memory (NVRAM), or storage class memory (SCM), is a new byte-addressable memory technology that combines persistence (like SSDs) with access speeds close to DRAM.

Modern computing systems rely on heterogeneous storage hierarchies that integrate DRAM, NVM, SSD, and/or HDD, despite the advantages offered by NVM. Most previous research has focused on optimizing B-trees or log-structured merge (LSM) trees to take advantage of NVM, while only a few studies have explored similar enhancements for the B^ϵ-tree, despite its advantages for write-intensive workloads. Furthermore, existing storage engines are not designed to fully exploit the unique properties of heterogeneous storage hierarchies. This creates the need for unified indexing structures capable of efficiently bridging these diverse storage layers.

This thesis presents the design, implementation, and evaluation of a unified B^ϵ-tree that integrates DRAM, NVM, and SSD into a single crash-consistent index structure. The proposed approach features a centralized shared persistent buffer in NVM, supports direct in-place updates, and dynamically migrates hot nodes to DRAM to optimize both performance and consistency. A comparison of the unified and the classic B^ϵ-tree shows that the proposed design significantly improves insertion throughput and reduces structural modification overhead. Moreover, the results of comparisons with other NVM-optimized data structures demonstrate improved insertion performance and reduced write overhead. The evaluation of the hybrid crash recovery mechanism further shows that the tree can restore a consistent state efficiently, even for large datasets.

Acknowledgments

I would like to express my sincere gratitude to **Prof. Dr. rer. nat. habil. Gunter Saake** for giving me the opportunity to work on my master thesis in the research group databases and software engineering and for allowing me to explore and dive deep into the field of modern database systems and heterogeneous persistent storage.

I would also like to thank **Dr.-Ing. Balasubramanian Gurumurthy** for his valuable time. He kindly accepted to supervise the topic of this thesis.

I am especially grateful to **M.Sc. Sajad Karim** for his continuous support throughout the entire journey of this thesis. He guided me during the initial steps and provided valuable feedback and suggestions for improvement during all stages of the work.

Last but not least, I would like to thank my family for their support, encouragement, and motivation throughout my studies.

Contents

List of Figures

List of Tables

List of Code Listings

List of Algorithms

List of Acronyms

BF	Bloom Filter
BRT	Buffered Repository Tree
CDDS	Consistent and Durable Data Structures
CoW	Copy-On-Write
CPU	Central Processing Unit
DCPMM	Intel® Optane™ Persistent Memory
DMA	Direct Memory Access
DML	Data Management Layer
DRAM	Dynamic Random Access Memory
EOF	End of File
FAIR	Failure-Atomic In-place Rebalance
FAST	Failure-Atomic Shift
FeRAM	Ferroelectric RAM
FIFO	First-In First-Out
FPTree	Fingerprinting Persistent Tree
HBtree	Hybrid B^+-Tree
HDD	Hard Disk Drive
HTM	Hardware Transactional Memory
I/O	Input/Output
IMC	Integrated Memory Controller
LRU	Least Recently Used
LSM	Log-Structured Merge
MLC-PCM	Multi Level Cell Phase Change Memory
MRAM	Magnetoresistive RAM
mw-B^ϵ tree	Multi-Write-Mode B^ϵ-Tree
NAW	NVM Atomic Write
NTFS	New Technology File System
NUMA	Non-Uniform Memory Access
NVDIMM	Non-Volatile Dual In-Line Memory Module
NVM	Non-Volatile Memory
NVML	Non-Volatile Memory Library
NVMM	Non-Volatile Main Memory
NVRAM	Non-Volatile Random Access Memory
PCM	Phase-Change Memory
PMDK	Persistent Memory Development Kit
PMem	Persistent Memory

RAM Random Access Memory
RDMA Remote Direct Memory Access
RMW Read Modify Write
RRAM Resistive RAM
RS Resistance Switching
SCM Storage Class Memory
SMT Simultaneous Multithreading
SPRAM Spin-Transfer Torque RAM
SRAM Static Random Access Memory
SSD Solid-State Drive
SStable Sorted String Table
WAL Write-Ahead Logging
wB$^+$-tree Write Atomic B$^+$-Tree
YCSB Yahoo! Cloud Serving Benchmarking

1. Introduction

1.1 Motivation

In computer science, data management and analysis have become more challenging because of the rapid growth of data. In [Kaz18], Kazemi showed that the volume of data stored globally doubles every two years. This phenomenon is called the "data-doubling effect". This exponential growth requires storage engines that can process, update, and query massive datasets efficiently.

Dynamic random access memory (DRAM), which is known for its simple design, cannot handle this growing amount of data, despite the improvements made to it. Furthermore, this type of memory is volatile, which means that data is lost when the power is turned off or a crash occurs. DRAM also consumes significant power, especially when accessing the memory frequently [ECKY13]. Therefore, DRAM alone cannot meet the needs of modern data-rich applications. To address the limitations of DRAM, NVM, also called persistent memory (PMem), is a new memory type that combines persistence with access speeds close to DRAM. It keeps data after power loss, which makes it an ideal memory for crash-consistent systems.

Although NVM provides significant advantages, modern systems rely on heterogeneous storage hierarchies that combine multiple layers, such as DRAM, NVM, solid-state drive (SSD), and/or hard disk drive (HDD). Each layer offers different trade-offs between speed, persistence, capacity, and cost. This implies the need to integrate these storage layers within a unified system. Existing storage engines, which mostly rely on B-trees [CJ15; HKWN18; LCW20; OLN+16; VTRC11; YWW+16; ZSW21] or log-structured merge (LSM)-trees [KBG+18; ZD21; ZMR+23], do not take into account these heterogeneous storage hierarchies. Furthermore, only a few storage systems [Hö21; LC25; Wie18] use the B^ϵ-tree, that handles write-intensive workloads and large datasets. This presents a critical research opportunity to revisit and adapt the B^ϵ-tree design to meet the demands of modern storage architectures.

This thesis builds on the research proposed by Karim et al. [KWB+24]. They presented different extensions and features that can be implemented in the classic B^ϵ-tree to exploit the unique characteristics of NVM and to support efficient operation across heterogeneous storage devices. This thesis extends their conceptual work by fully implementing the proposed unified B^ϵ-tree, analyzing it, evaluating its performance, and comparing it with prior research.

1.2 Goals

The key goals of this thesis are to implement, analyze, and evaluate the unified B^ϵ-tree proposed by Karim et al. [KWB+24], which supports heterogeneous storage environments. The primary objectives include:

- Implementing a single tree structure that spans DRAM, NVM, SSD, and/or HDD.
- Enabling direct reads and in-place updates in NVM without the need to copy nodes to DRAM.
- Using a centralized persistent shared buffer instead of internal node buffers to manage buffered operations.
- Migrating hot nodes from NVM to DRAM based on access frequency.
- Implementing a read buffer for frequently accessed keys in DRAM.
- Ensuring crash-safe recovery for the tree.
- Evaluating the performance of the tree and comparing it with the classic B^ϵ-tree and prior research.

1.3 Research Questions

This thesis addresses the following research questions:

- **RQ1**: How can the B^ϵ-tree be optimized for NVM?
- **RQ2**: How can the unified B^ϵ-tree design be fully implemented in a heterogeneous storage environment?
- **RQ3**: How does the unified B^ϵ-tree perform compared to the classic B^ϵ-tree and other NVM-optimized data structures?

1.4 Thesis Structure

The remainder of the thesis is structured as follows:

- **Chapter 2**: Provides the theoretical and technical foundation for this thesis. It introduces the different data structures used in storage engines such as B-tree, B^+-tree, and their variants. It also presents another category, which is the log-structured merge-tree. Furthermore, DRAM and NVM are presented and discussed in detail, including their advantages, disadvantages, and technologies.
- **Chapter 3**: Reviews existing research on persistent and heterogeneous storage trees. It presents the design and objectives of each data structure and identifies their limitations.

- **Chapter 4**: Explains the design goals and system architecture of the unified B$^\epsilon$-tree proposed by Karim et al. [KWB+24], including internal node structures, the shared buffer in NVM, and the read buffer in DRAM. Furthermore, other proposed features are discussed in detail.
- **Chapter 5**: Presents the full implementation of the proposed design. First, it describes the baseline implementation, which was necessary due to the lack of open-source B$^\epsilon$-tree implementations that are sufficiently extensible. Then, the features discussed in the previous chapter are implemented and presented in detail.
- **Chapter 6**: Presents the performance analysis of the unified B$^\epsilon$-tree, compares it with the classic B$^\epsilon$-tree and other NVM optimized data structure discussed in the related work. Moreover, the recovery overhead of the tree is evaluated.
- **Chapter 7**: Summarizes the key findings of this thesis.
- **Chapter 8**: Discusses potential directions for future work. This includes reducing lookup latency in the centralized shared buffer and auxiliary hash tables, enabling multithreading support, introducing partitioned buffers, and implementing dynamic buffer management.

2. Background

Storage engines are an important component of database systems and are responsible for managing and retrieving stored data. Different data structures are used by these storage engines, which makes it important to choose the correct data structure, as it impacts both the design and performance of the engine.

In this chapter, various data structures that are commonly used in different storage engines are presented. These include B-trees, B^{+}-trees, B$^{\epsilon}$-trees, and log-structured merge trees. For each data structure, its definition, properties, and different operations are explored. Moreover, a brief comparison of runtime of different read and write operations is made.

Furthermore, both volatile and non-volatile memories are introduced and discussed. For each storage technology, characteristics, categories, and applications are presented.

2.1 B-Tree

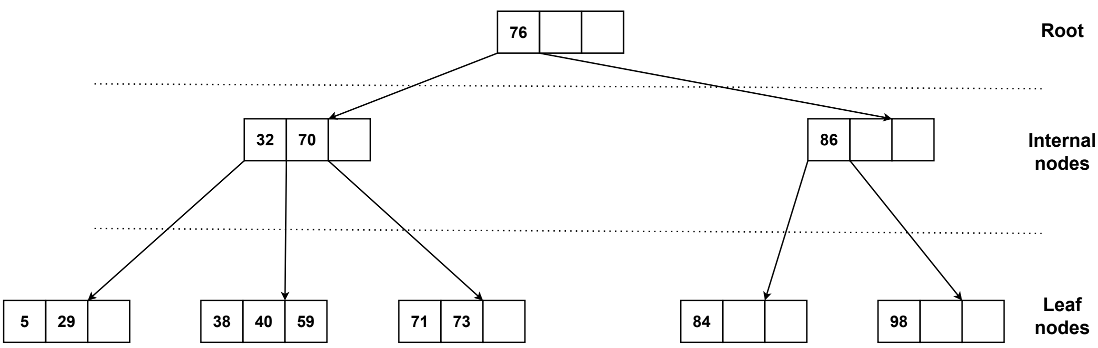


Figure 2.1: B-tree of order 4

2.1.1 Definition and Properties

A B-tree is a common data structure used in various applications, primarily in storage systems. It was introduced in 1972 by Bayer et al. [BM70], to efficiently manage large indices for random access files. The original B-tree algorithms were designed to minimize access latency [KJ92; TC09]. A B-tree is a self-balanced tree that keeps data sorted and supports many different operations, such as inserts, deletes, queries, and updates in logarithmic time [KB10]. As shown in Figure 2.1, the B-tree consists of two types of nodes:

- Internal nodes: an ordered set of keys and child pointers. A child can be an internal or a leaf node. Each internal node has a maximum and minimum number of children, which are determined by the degree of the tree itself. This criteria is defined to keep the tree balanced.
- Leaf nodes: an ordered set of key-value pairs. They do not have children and are always at the bottom of the tree.

Let a B-tree be of degree m. The following properties need to be met [CLRS09; Knu98; KR98]:

- Every internal node, except the root, has a minimum of $\left\lceil \frac{m}{2} \right\rceil$ children and $\left\lceil \frac{m-1}{2} \right\rceil$ keys, and a maximum of m children and $m-1$ keys.
- The number of keys in an internal node is always less than the number of children in the node by 1. Let k be the number of children of an internal node. It has $k-1$ keys.
- If the root is an internal node, then it has at least two children.
- Leaf nodes have the same depth (level).

Figure 2.1 shows an example of a B-tree of order 4. Each node can have at most 3 keys and 4 children (except the leaf).

As mentioned before, B-trees are self-balancing, as operations such as inserting or deleting a key-value pair trigger node splits or merges to ensure that the tree maintains its defined properties. The B-tree does not support redundant keys. When a node becomes overfull (holds more keys than the maximum), a split is performed by promoting the median key upward to the parent or to a new node. On the other hand, merges are triggered when a node becomes underfull (holds fewer keys than the minimum) by taking keys from siblings or by merging two nodes into one.

The B-tree has been widely used as a data structure for storing large amounts of information, especially on secondary storage devices [Knu98; LY81]. It is used in various applications, such as database indexing (e.g., PostgreSQL [Pos]) and embedded databases (e.g., SQLite [SQL]).

2.1.2 Operations

Let T be a B-tree of order m (m is the maximum number of children per internal node).

Insert a key-value pair (k, v)

1. Start at the root of T and go down until the leaf to which k belongs is found.
2. Insert (k, v) into the leaf in the correct position.
3. If the leaf overflows (i.e., the number of keys is greater than $m - 1$) $\rightarrow$ split.
   (a) Split the leaf into two leaf nodes:
      i. Remove the median key from the leaf and promote it to the parent.
      ii. Left leaf gets the first $\left\lceil \frac{m}{2} \right\rceil - 1$ keys.
      iii. The other leaf gets the remaining keys.
   (b) Recursively check if the parent overflows (may require splitting).
   (c) If the root overflows, create a new root and increase the height of the tree by one.

Update the value v **of a key** k

1. Search for the node that contains k. The node can be either internal or a leaf.
2. If k is found, update the value of k
3. Otherwise, return that the key is not present in T.

Delete a key k

1. If k is present in a leaf node:
   (a) Start at the root of T and traverse to the leaf where k is present.
   (b) Delete k.
   (c) If a leaf underflows (i.e., the number of keys is less than $(\left\lceil \frac{m}{2} \right\rceil - 1) \rightarrow$ merge.
      i. merge the leaf with siblings:
         A. Try to borrow a key from siblings.
         B. If borrowing is not possible, the leaf node is merged with a sibling, and the parent is adjusted.
         C. Recursively check if the parent underflows.
2. If k is present in an internal node:
   (a) Start at the root of T and traverse to the internal node where k is present.
   (b) Delete k by replacing it with its predecessor or successor key.
   (c) Recursively delete the replacement key from the corresponding leaf node.
   (d) If a node underflows or overflows $\rightarrow$ merge or split.
3. Otherwise, return that the key is not present in T.

Search for key k

1. Start at the root node x.
2. While x is not a leaf node:
   (a) find the smallest index so that the key in that index $\geq k$.
   (b) if the key is found, return it with its value.
   (c) Else, descend to the child.
3. if x is a leaf node: check if x contains k and return the result.
4. if k is not found, return that the key is not present in T.

2.2 B$^+$-Tree

2.2.1 Definition and Properties

The B$^+$-tree is a further data structure, which is an extension of B-tree. It was introduced by Rudolf Bayer and Edward M. McCreight [BM70]. Douglas Comer also showed the importance of this data structure [Com79]. B$^+$-trees have become the "de facto standard indices for file organization and databases" [TC09]. It is an ordered, n-ary branching, self-balancing, search tree, and is widely used in databases, file systems, and key-value store systems [HYZ$^+$25; ST08].

Like the B-tree, this data structure has two types of nodes, internal and leaf nodes. The key difference is that internal nodes store only copies of keys for routing, while leaf nodes store the actual data [EN10] (see Figure 2.2). Additionally, the data nodes are linked to each other as a linked list. These optimizations enable efficient range queries and allow the internal nodes to store more keys per node, which reduces the tree height and results in a higher fanout. In contrast to B-trees, B$^+$-trees support redundant keys. The properties defined in Section 2.1.1 are also valid for B$^+$-tree.

As the B$^+$-tree is an old data structure, many changes and improvements have been made to this structure, and several efficient trees were implemented over the years. B-link is one of these improvements. It is a B$^+$-tree with every node containing a pointer to its right sibling [JK93; LY81]. It was implemented to "build a dB tree with replicated index nodes across a message-passing multiprocessor architecture to improve parallelism and alleviate bottlenecks" [JK93; TC09].

B$^+$-trees are used in many applications, such as file systems (e.g., New Technology File System (NTFS), which is the primary file system for windows servers [Mic]) and database indexing (e.g., InnoDB in MySQL [Ora]).

2.2.2 Operations

The B$^+$-tree supports many operations, such as inserting or deleting a key-value pair, updating the value of a key, and searching for a key, with a time complexity of $O(log\ n)$. Let T be a B$^+$-tree of order m.

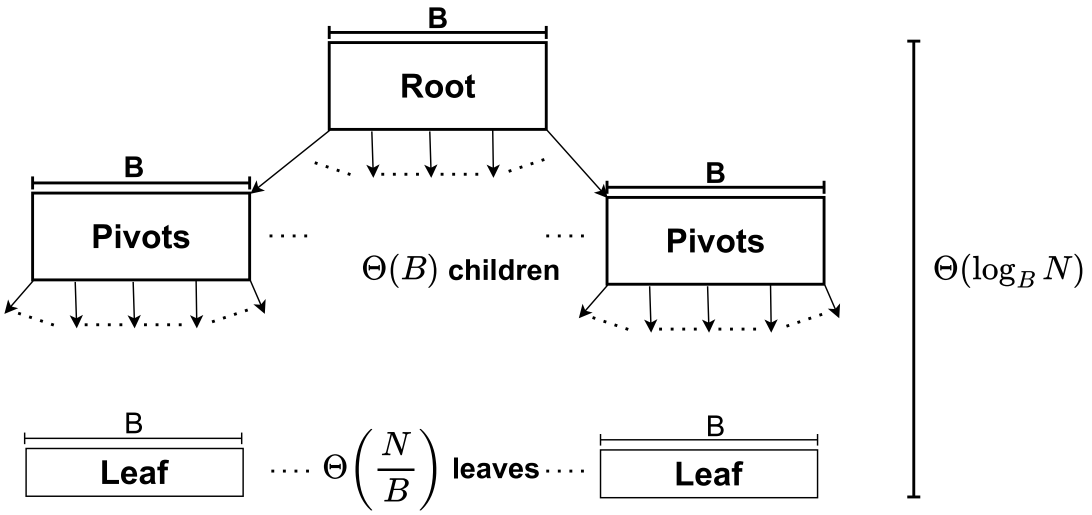


Figure 2.2: Overview of B+-tree [JYZ+15]

Insert a key-value pair (k, v)

The main difference between insertion in a B-tree and a B+-tree is where the data are stored and how the tree is structured. In B-tree, both internal and leaf nodes can store data. On the other hand, internal nodes in B+-trees contain only keys used for routing. During insertion in a B+-tree, the key and data are always inserted into a leaf node. If the leaf overflows (i.e., the number of keys is greater than $m - 1$), a split operation is applied recursively to ensure the correctness of the tree.

Update the value v **of a key** k

1. Search for the node that contains k. The node is a leaf node.
2. If k is found, update the value of k
3. Otherwise, return that the key is not present in T.

Delete a key k

First, starting from the root and with the help of routing nodes, the correct leaf node that contains the key is reached. The key k is deleted and, like in B-tree, it is necessary to check if the node underflows (i.e., the number of keys is less than $(\lceil \frac{m}{2} \rceil - 1)$. If this is the case, a merge or borrowing with a sibling is performed to restore balance. Since internal nodes contain routing keys that guide the search, deletion can affect these internal keys.

Search for key k

1. Start at the root node x.
2. For each internal node x:

(a) find the smallest index i where $k \leq \text{keys}[i]$.
(b) If no such i exists, descend to the rightmost child.
(c) Else, descend to child$[i]$.

3. if x is leaf node: check if x contains k and return the result.
4. if k is not found, return that the key is not present in T.

2.3 B$^\epsilon$-Tree

2.3.1 Definition and Properties

B-Epsilon-tree (B$^\epsilon$-tree), the focus of this thesis, was first introduced by Brodal et al. [BF03]. Bender et al. [BFCJ$^+$15] has also described it. This data structure is an extension of the B$^+$-tree, which offers significant improvements that will be discussed in this section.

The property that makes this tree unique is the buffers in the internal nodes. It serves as a cache to store different operations before propagating them down to the appropriate child nodes in batches. The buffer is unique to each internal node. Like in B$^+$-trees, the internal nodes serve as routing nodes, and the actual data is stored in leaf nodes, which are at the bottom of the tree. In contrast to internal nodes, leaf nodes in a (B$^\epsilon$-tree) do not have buffers. When operations are flushed (i.e., the buffer is full) from the buffers of internal nodes, they are applied directly to the corresponding leaf nodes.

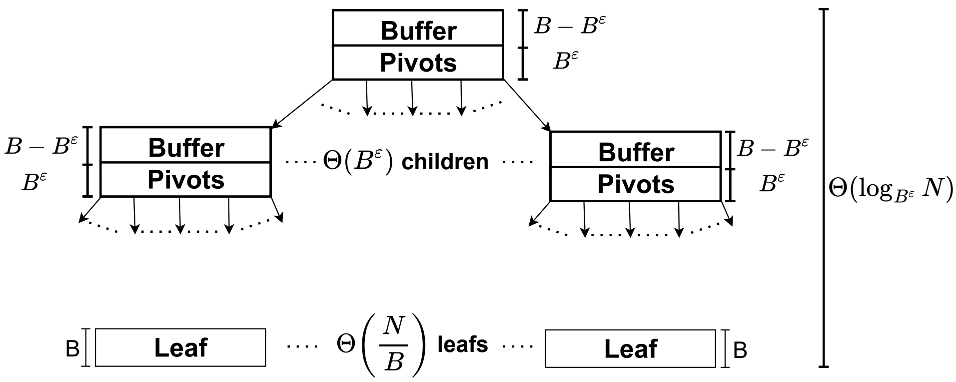


Figure 2.3: B$^\epsilon$-tree [BFCJ$^+$15]

2.3.2 Role of Tuning Parameter

The tuning parameter $\epsilon \in [0, 1]$ has a significant impact on the structure and performance of the tree (see Figure 2.3). It determines the size of the buffer(i.e., how many messages the buffer can hold before flushing), the depth of the tree ($O(\log_{B^\epsilon} N)$), and space used by internal nodes for both pivots (B^ϵ) and buffer ($B - B^\epsilon$).

When ϵ is equal to one, internal nodes allocate all available space to pivots, which results in the buffer does not exist anymore. Therefore, every operation is propagated

Type of Tree	Insert	Search	Range Query
B-Tree	$\log_B N$	$\log_B N$	$\log_B N + \frac{k}{B}$
B^ϵ-Tree	$\frac{\log_B N}{\epsilon B^{1-\epsilon}}$	$\frac{\log_B N}{\epsilon}$	$\frac{\log_B N}{\epsilon} + \frac{k}{B}$
B^ϵ-Tree with $\epsilon = \frac{1}{2}$	$\frac{\log_B N}{\sqrt{B}}$	$\log_B N$	$\log_B N + \frac{k}{B}$
LSM-Tree	$\frac{\log_B N}{\epsilon B^{1-\epsilon}}$	$\frac{\log_B^2 N}{\epsilon}$	$\frac{\log_B^2 N}{\epsilon} + \frac{k}{B}$
LSM-Tree with bloom filter	$\frac{\log_B N}{\epsilon B^{1-\epsilon}}$	$\log_B N$	$\frac{\log_B^2 N}{\epsilon} + \frac{k}{B}$

Table 2.1: Runtime of insert, search, and range query for different trees [BFCJ+15]

immediately to the correct child, and no buffering is needed. The tree acts as a B-tree. Contrarily, when $\epsilon = 0$, internal nodes allocate all available space to buffers, which results in a buffered repository tree (BRT) [BFCJ+15; BGVW00]. In most configurations, ϵ is set to $\frac{1}{2}$, as this allows the B^ϵ-tree to have asymptotic point search performance and asymptotically better insert performance compared to the B-tree [BFCJ+15] (see Table 2.1).

As presented in Table 2.1, the B^ϵ-tree is a suitable data structure for applications with high write loads. It is used in high-performance applications, such as the storage engine TokuDB [BFCJ+15; Per22], which is used in MySQL [MyS] and MariaDB [Mar], as well as in the streaming file system TokuFS [EBFCK12], and BetrFS [JYZ+15].

2.3.3 Operations

B$^\epsilon$-tree supports many operation, such as inserting, deleting a key-value pair, updating the value of a key, and searching for a key or a range of keys.

Let T be a B$^\epsilon$-tree of degree m and $\epsilon = \frac{1}{2}$.

Insert a key-value pair (k, v)

1. If T is empty:
   (a) Create a new leaf node as the root.
   (b) Insert (k, v) directly into the root node.
   (c) Return success.
2. Let $x \leftarrow$ root.
3. If x is an internal node:
   (a) Buffer the insert operation with (k, v) pair in the current node's buffer.
   (b) If the buffer size $\geq \frac{m}{2} \rightarrow$ Flush the buffer.
   (c) Return Success.

4. Else, current is a leaf node:
   (a) Insert (k, v) into the leaf in the correct position.
   (b) If the leaf node is overfull (i.e., the number of keys $\geq m$) $\rightarrow$ Split the node.
   (c) Return Success.

The full pseudocode is provided in Algorithm A.1.

Delete a key k

1. If T is empty:
   (a) return an error.
2. Let $x \leftarrow$ root.
3. If x is an internal node:
   (a) Buffer the delete operation with k in the current buffer of the node.
   (b) If the buffer size $\geq \frac{m}{2} \rightarrow$ Flush the buffer.
   (c) Return Success.
4. Else, current is a leaf node:
   (a) Search for the key in the leaf node.
   (b) If found remove k and its value from the leaf.
   (c) Else return an error.
   (d) If the leaf node is underfull (i.e., the number of keys $< \frac{m}{2}$) $\rightarrow$ call `handleUnderflow()`.
   (e) Return Success.

The full pseudocode is provided in Algorithm A.2.

Update the value v of a key k

1. Let $x \leftarrow$ root.
2. If x is an internal node:
   (a) Buffer the update operation with (k, v) in the current node's buffer.
   (b) If the buffer size $\geq \frac{m}{2} \rightarrow$ Flush the buffer.
   (c) Return Success.
3. Else current is a leaf node:
   (a) Search for the key in the leaf node.
   (b) If found update v.
   (c) Else, return an error.
   (d) Return Success.

The full pseudocode is provided in Algorithm A.3.

Search for key k

1. If T is empty:
   (a) return an error.
2. Let $x \leftarrow$ root.
3. Initialize an empty list collectedMessages.
4. While x is an internal node:
   (a) For each buffered message in node x:
      i. if the message contains $k \rightarrow$ add it to collectedMessages.
   (b) Determine the appropriate child index i using the upper bound.
   (c) Set $x \leftarrow x$.children$[i]$
5. if x becomes leaf node:
   (a) Search for the key in the leaf node.
      i. if k is found in the node $\rightarrow$ retrieve it.
      ii. Else, Check collectedMessages for a buffered `Insert` message:
         A. If found, use the (k, v) and continue.
         B. If a Delete message for k is found, return error.
         C. If no message for k is found, return an error.
6. Apply remaining messages in collectedMessages in reverse order:
   (a) If a Delete message for k is found, return error.
   (b) If an Update message for k is found, overwrite the value.
7. Return Success.

The full pseudocode is provided in Algorithm A.4

Another important operation that needs to be discussed is buffer flushing. The buffer has messages of this form $(opType, k, v)$:

- opType: the operation type (can be an insert, delete, or update).
- k: the key that the operation will be applied on.
- v: the associated value for the key.

Buffer Flushing

1. If the node is a leaf or its buffer is empty, return Success.
2. Create a copy of the buffer, then clear the buffer of the current node.
3. For each $(opType, k, v)$ in the copied buffer:

(a) Find the correct child index i for key k using upper-bound.
(b) Set child $\leftarrow$ current-node.children[i].
(c) Based on opType, do the following:

- **Insert:**
  - If `child` is internal:
    - Buffer the insert in the child.
  - Else:
    - Insert the pair (k, v) into leaf.
    - Check if split is required.
- **Delete:**
  - If `child` is internal:
    - Buffer the delete in the child.
  - Else:
    - Remove the pair (k, v) from the leaf.
    - handle underflow if required.
- **Update:**
  - If `child` is internal:
    - Buffer the update in the child.
  - Else:
    - If k exists, update the corresponding v.

The full pseudocode is provided in Algorithm A.6

2.4 Log-Structured Merge-Tree

2.4.1 Definition

LSM-tree is a further data structure that will be presented in this thesis. It is a disk-based, write-optimized index structure, and was first introduced by O'Neil et al. [OCGO96] in 1996. The main goal was to support efficient indexing of write-heavy logs.

LSM-tree is used in many different applications, especially in NoSQL systems [LC19; Mis24], such as BigTable [CDG+08], Dynamo [DHJ+07], HBase [Apab], Cassandra [Apaa], LevelDB [Goo21], RocksDB [Met22], and AsterixDB [ABB+14]. Furthermore, it is used in embedded storage engines such as PebblesDB [RKCA17] and WiscKey [LPG+17].

2.4.2 Components

LSM-trees do not have internal or leaf nodes like the B-tree family. Instead, as shown in Figure 2.4, an LSM-tree consists of two key components:

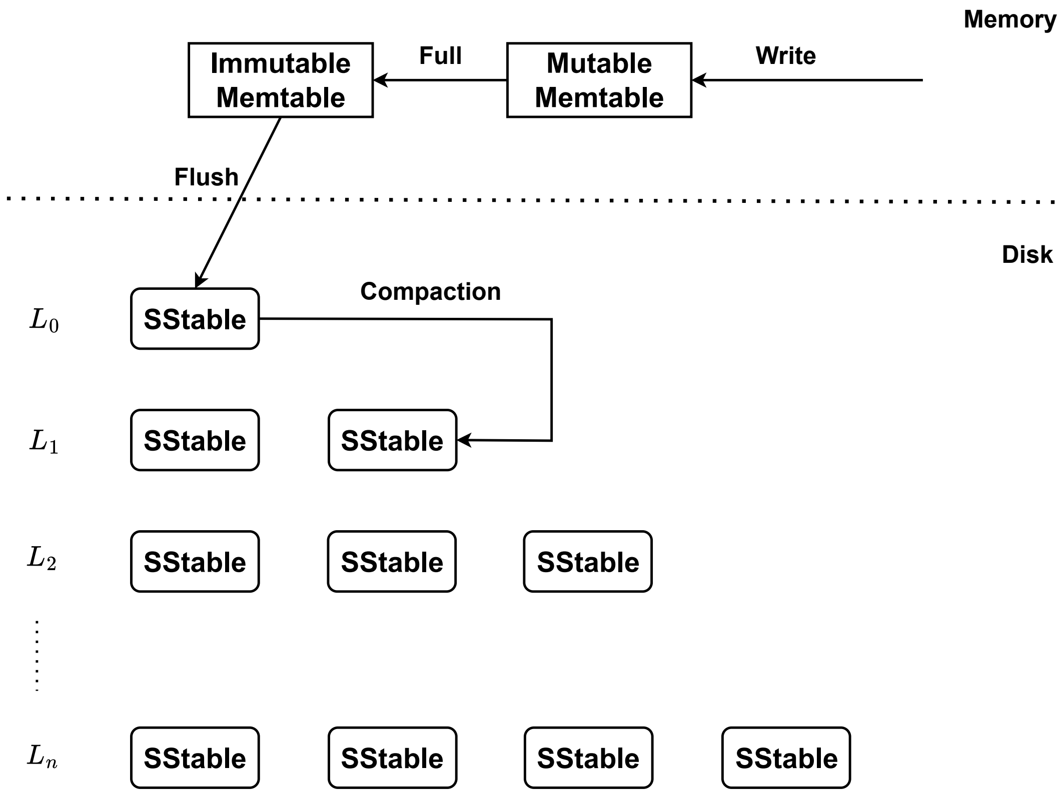


Figure 2.4: Architecture of LSM-tree [Met22; VDB+22; ZWC+20]

- In-memory structure: Also called memtable. In most implementations, a skip-list [Pug90] or B+-tree is used [LC19]. The role of the memtable, which is in DRAM, is to store the data, which comes from different operations such as inserts and deletes. If it becomes full, memtable is converted into an immutable memtable, and the data will be flushed into a new immutable, sorted file called a sorted string table (SStable), which is in the disk.
- In-disk structure: Also called SStables. These receive flushed data from the memtable and store it in sorted order (i.e., by key). An SStable has a list of data blocks (i.e., sorted list of key-value pairs) and an index block (used for queries). SStables are immutable, meaning once the data is stored, it cannot be modified. Each level contains multiple SStable. This results that some data can have obsolete copies in some SStables, as the current copy will be always saved in the most recent SStable. This is the reason to introduce the compaction operation. In each level, it reads and merges all SStables so that deleted and redundant keys are removed. This improves lookup performance by reducing the number of SStables.

2.4.3 Role of Bloom Filter

As mentioned in Section 2.4.2, each level in storage has multiple SStables. When searching for a key, the active memtable is searched. If the key is not found, the search continues in the active immutable memtable, and finally in all SStables across all

Property	DRAM	SRAM
Structure	one transistor and one capacitor per bit	6 transistor per bit
Endurance	10^{15}	10^{16}
Read Latency	10-60 ns	0.2-2 ns
Write Latency	10-60 ns	0.2-2 ns
Density	Up to 16 Gb	1-4 Gb

Table 2.2: DRAM vs SRAM [BO17; PZDR+19]

levels. This operation can become costly, especially for keys that do not exist, which impacts read latency. For this reason, bloom filters (BFs) is used with LSM-trees. "These probabilistic data structures allow to skip an SStable if it does not contain the searched key" [ZMRA21].

Table 2.1 presents the impact of using BFs in the runtime of different operations. BF does not affect the insertion cost. However, it decreases the cost of search operations, which makes the LSM-tree read optimized.

2.5 Volatile Memory

2.5.1 Definition

Volatile memory is a storage technology that requires power to retain data. If the power is turned off, the data is lost. "This makes it difficult for digital forensic investigators to preserve important evidence stored in volatile memory" [LPF19; SKP+22]. It is used as a temporary storage medium for data and code that the processor uses to perform fast read operations [HR24]. Volatile memory allows for quick data storage and processing, which makes it a useful memory technology. Random access memory (RAM) is one of most famous volatile memory which is primarily used in computers. It allows different applications in a computer to read and write data on a short-term basis.

2.5.2 Categories

Volatile memories can be classified into two different categories (see Figure 2.5):

- DRAM: Data are stored as charges on a capacitor (electrical charges).
- Static random access memory (SRAM): Flip-flops are used here instead of the capacitor. Data are stored as bits.

These two different types of RAM have various characteristics which are presented in Table 2.2.

As mentioned in Table 2.2, SRAM has both faster read and write access compared to DRAM. This makes SRAM ideal for high performance applications such as central processing unit (CPU) caches. However, it has a lower density, as it occupies more physical space per bit. This makes it not ideal for large memory capacities. DRAM, on the other hand, has a higher density, which makes it suitable to be used for storing large amounts of data.

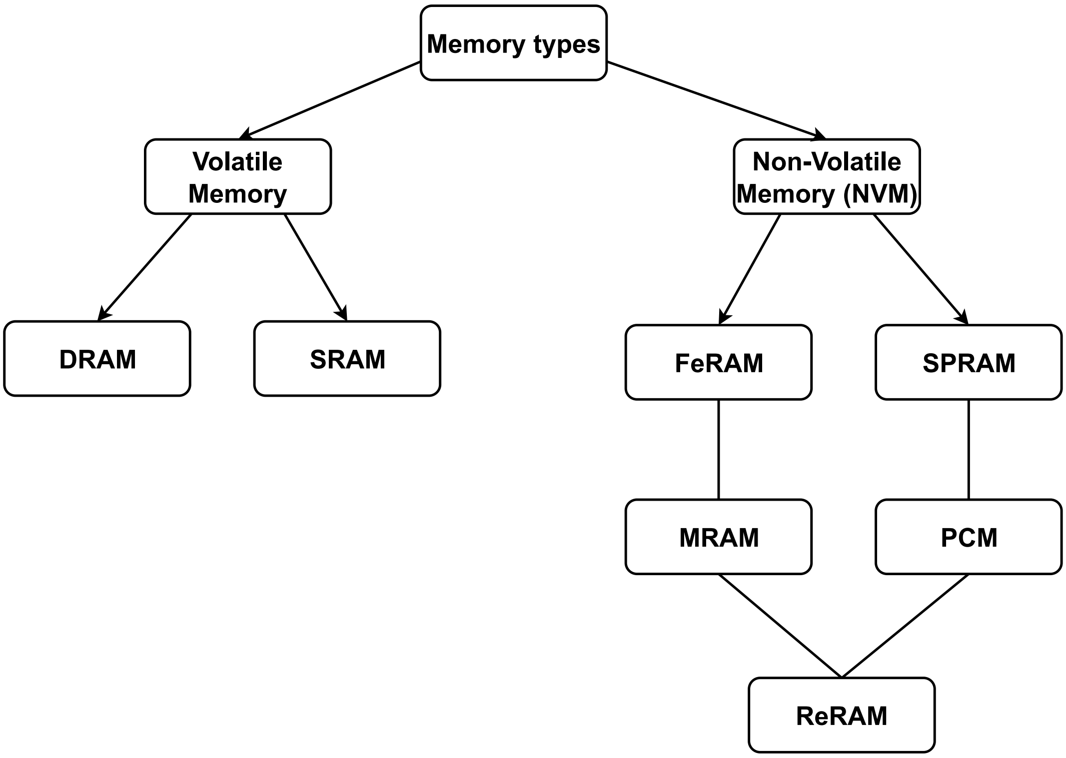


Figure 2.5: Memory types [DPV21; JTK+12]

2.6 Non-Volatile Memory

2.6.1 Definition

NVM, also called persistent memory PMem, is a storage technology that retains data even when power is removed [BCMV03; Bre04; CCA+11; MSCT14]. It is also known by other names such as non-volatile random access memory (NVRAM) [CHRG23] and storage class memory (SCM) [Lam10]. NVM is a new storage class that effectively fills the gap between main memory and secondary storage [KWB+24] and combines the performance and byte-addressability of DRAM with the persistence of traditional storage devices [VRLK+18]. “It is important for storing large volumes of data that must be saved over long periods, as it retains information even without a power supply” [Ram24].

2.6.2 Types of Non-Volatile Memory Technologies

Several important NVM technologies have been developed, each with different physical characteristics and design trade-offs (see Figure 2.5):

- Ferroelectric RAM (FeRAM): is an NVM that uses a ferroelectric film to store the polarization states [Ish12; MDH+01]. It has fast read and write operations with “no intrinsic limitation”, high write endurance, and low power operation [RRB10].
- Magnetoresistive RAM (MRAM): stores data as stable magnetic states of magnetoresistive devices [ADS16; IMJA20; SDD+02].

Property	FeRAM	MRAM	SPRAM	PCM	ReRAM
Density	1-128 MB	10-100 MB	2-16 MB	1-32 GB	64 MB-1 GB
Endurance	10^{14}-10^{15}	10^{15}-10^{16}	10^{15}	10^{6}-10^{9}	10^{4}-10^{10}
Read Latency	45-90 ns	5-50 ns	5-10 ns	30-60 ns	9-25 ns
Write Latency	40-100 ns	10-20 ns	12 ns	10-300 ns	1-100 ns
Program Energy	1-25 pJ	<1 pJ	0.02 pJ	90 pJ-1 nJ	5-25 pJ

Table 2.3: Characteristics of various NVM technologies [BKS+08; KWK+25; PZDR+19; RRB10]

- Spin-transfer torque RAM (SPRAM): is a non-volatile memory technology with a unique combination of speed, endurance, density, and ease of fabrication [KITO12; WH24]. It offers advantages including lower programming current and better scaling perspectives [RRB10].
- Phase-change memory (PCM): is a key enabling technology for non-volatile electrical data storage at the nanometer scale [GS20]. “It toggles chalcogenide materials between amorphous and crystalline phases” [FNW17]. PCM offers several advantages, including fast write speed, medium to low voltage requirements for writing, and good scalability [RRB10].
- Resistive RAM (RRAM) (or ReRAM): is a non-volatile memory technology that records data information based on resistance switching (RS) [WKK+19]. “It modulates the metal-insulator state via ionic migration” [Che20]. “RRAM has a good read signal window and excellent scalability” [RRB10].

These different technologies have various characteristics which are presented in Table 2.3.

2.6.3 Characteristics and Applications

Characteristics

NVM benefits from various characteristics that make it a unique memory technology used in a wide range of devices and systems:

- Persistence: As mentioned previously, NVM has the advantage that it can retain stored data even when the power is turned off.
- Byte-addressability: Non-volatile memories like Intel® Optane™ [1] allow CPU-style load/store operations at the byte level, unlike traditional block-based SSDs [Akr21].
- High endurance: FeRAM, for example, offers high endurance (up to 10^{15} read-/write cycles) (see Table 2.3).

[1] https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-memory.html

- Lower Latency: Intel® Optane™ DCPMM [2], which is “the first commercially available, byte-addressable NVM modules released in April 2019” [HT20], has a lower write latency than DRAM [HT20; YKH+20].
- High density: Emerging non-volatile main memory (NVMM) technologies provide a much higher memory density than DRAM at a lower cost [LCJ+20].
- Scalability: Intel Optane Persistent Memory (DCPMM), which is an NVM technology and “the first commercially available non-volatile dual in-line memory module (NVDIMM) that creates a new level between volatile DRAM and block-based storage” [IYZ+19], offers large capacity sizes (128 GB, 256 GB, 512 GB), significantly exceeding typical DRAM modules [Int22].
- “Data is CPU cache coherent” [Sca20a, p.12–13].
- Persistent memory provides direct memory access (DMA) and remote direct memory access (RDMA) operations [Sca20a, p.12–13].

Applications

NVM is used in many applications such as consumer electronics (e.g., mobile systems) [CWH+14; IS14; WWC+15], scientific applications [CCM+10; IRLP16; JWC+13], embedded systems [KGT+14; LKE18; LZL+13], databases [CCM+10; DAT+14; DZC+13], and medical equipment [AGEM24; JYY+18; LCT+22].

[2] https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html

3. Related Work

In this chapter, various related works are presented. The goal is to provide an overview of the storage engines that have been adapted or optimized for NVM. This includes solutions based on B-trees, B$^+$-trees, and LSM-trees, which are widely studied and commonly used in PMem storage engines. Furthermore, two storage engines are presented that extend B$^\epsilon$-trees to take advantage of NVM. For each work, the system architecture, key features, and identified limitations are discussed.

3.1 NoveLSM

NoveLSM is a NVM-optimized LSM-tree that was introduced in 2018 by Kannan et al. [KBG+18]. The main goal of this data structure was to optimize the traditional LSM-tree to fully exploit the byte-addressability, low latency, and persistence of NVM, which allows avoiding the high compaction overhead and logging costs of the traditional LSM-trees. It has a hybrid architecture optimized for heterogeneous storage systems.

Figure 3.1 shows the architecture of NoveLSM. The tree uses, in addition to the memtable in DRAM, a persistent skip-list memtable that is in NVM, which avoids serialization and deserialization to SStables costs and enables faster crash recovery. Furthermore, it has a further persistent memtable, which is mutable and allows direct in-place updates. This feature reduces the frequency of compaction, eliminates the WAL for insert operations in NVM memtable, and minimizes logging overhead. Moreover, this data structure enables read parallelism. When searching for a key, it performs parallel read operations on the DRAM memtable, the NVM memtable, and SStables.

However, NoveLSM suffers from some limitations in terms of performance. The parallel read mechanism introduces thread management overhead, which can be costly, especially when searching for a small amount of data. Furthermore, NoveLSM has performance fluctuations and delay overhead. As discussed in [ZLCW20], the mutable NVM memtable only postpones write stalls. When datasets exceed NVM capacity, large $L_0 - L_1$ compactions occur, resulting in higher write amplification.

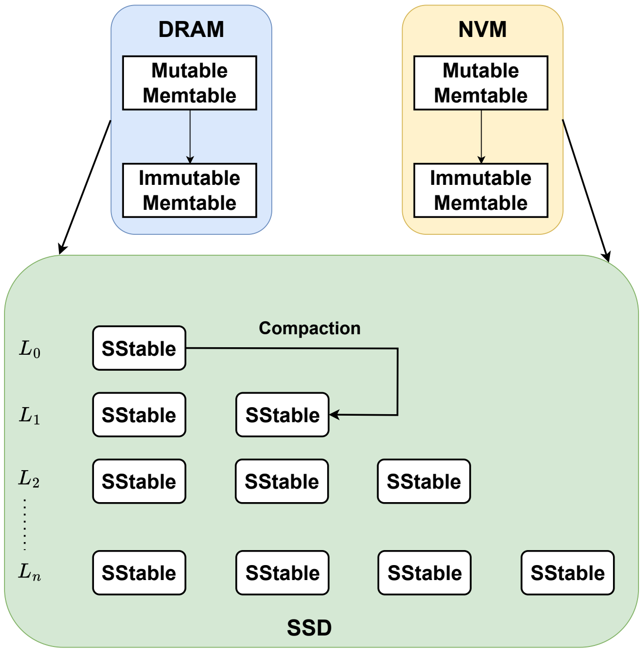


Figure 3.1: Architecture of NoveLSM [KBG+18]

3.2 Consistent and Durable Data Structures B-Tree

Consistent and durable data structures (CDDS) B-tree was introduced in 2011 by Venkataraman et al. [VTRC11]. It was one of the first works to implement the use of NVM for B-trees. This data structure is durable and consistent, which is achieved through versioning. It is used to allow atomic updates without requiring logging (e.g. WAL) or shadow paging. This reduces the need for extra writes. Additionally, this versioning supports failure recovery by restoring the data structure to the most recent consistent version.

A further property of CDDS B-tree, which makes it different from normal B-tree, is unsorted leaf nodes (see Table 3.1). The reason behind this is to reduce write amplification when updating the tree structure. Furthermore, unlike the B-tree, this data structure handles node updates through copy-on-write (CoW). This ensures the consistency of the update operations.

CDDS B-tree also has several limitations. CoW requires significant memory overhead, as for every update operation, a copy of the updated node with its version is created, and no overwriting is done. Additionally, it does not support batched updates. Furthermore, versioning causes high write overhead, which affects the performance of the tree. It also requires the use of garbage collection to remove old versions. This reduces the scalability of the tree.

3.3 Write-Atomic B^+-Tree

Write atomic B^+-tree (wB^+-tree) was introduced in 2015 by Chen et al. [CJ15]. The idea of this data structure originated from an earlier proposition [CGN11] by the same authors. It belongs, as the name suggests, to the B^+-tree family and was designed for byte-addressable NVM.

It introduced the use of a slot array and/or a bitmap as an indirect layer. Operations such as insert and delete will not cause the movement of index entries, which keeps the node sorted (see Table 3.1), without any sorting or shifting overhead. This ensures the consistency of the tree, as only atomic writes and redo-only logging are needed. This also helps to avoid linear search and instead use binary search.

The use of a slot array reduces write amplification and minimizes persistent write operations compared to WAL. Furthermore, both internal and leaf nodes are contrary to the B^+-tree unsorted. The reason behind this design is to optimize write operations.

However, wB^+-tree has some limitations. The use of a slot array/bitmap makes the node management logic more complex and increases the lookup latency. Additionally, the mechanism used for cache lines (e.g., `clflush`) and memory fence (e.g., `mfence`) have some overhead, which may impact overall performance. Furthermore, unsorted nodes can reduce the performance of traversing the tree to reach a leaf node.

3.4 NV-Tree

NV-Tree is a further persistent optimized B^+-tree that was introduced in 2015 by Yang et al. [YWW+16]. The main goal is to have a consistent and cache-optimized B^+-tree variant. Similarly to wB^+-tree, it belongs to the B^+-tree family. Internal nodes are sorted in NVM, but the leaves are unsorted. Each key-value pair in the leaf node is associated with a flag that indicates the operation applied to this pair (insert or delete).

Figure 3.2 shows the architecture of the NV-Tree. When searching for a key, the tree uses linear scans (see Table 3.1) on the leaf nodes, which negatively affects search performance. Furthermore, as leaf nodes are unsorted and updated in an append-only mode, write amplification is reduced. Only leaf nodes are required to be consistent, as from them internal nodes are rebuilt when a crash occurs, which reduces the number of NVM write operations.

The shutdown strategy used in NV-Tree consists of three steps. First, all internal nodes are flushed to NVM. Then the root pointer is also saved in NVM. This place is reserved only for the root. Finally, a special flag is marked alongside the root to indicate that the tree shutdown was successfully. This ensures a correct recovery.

However, NV-Tree suffers from some limitations. The complexity of search and range queries is high as the leaf nodes are not sorted.

3.5 Fingerprinting Persistent Tree

Fingerprinting persistent tree (FPTree) is a hybrid SCM-DRAM persistent B^+-Tree that was introduced in 2016 by Oukid et al. [OLN+16]. Like NV-Tree, leaf nodes are unsorted and stored in NVM, while internal nodes are sorted and stored in DRAM (see Table 3.1). It introduces the use of fingerprinting, which is a one-byte hash

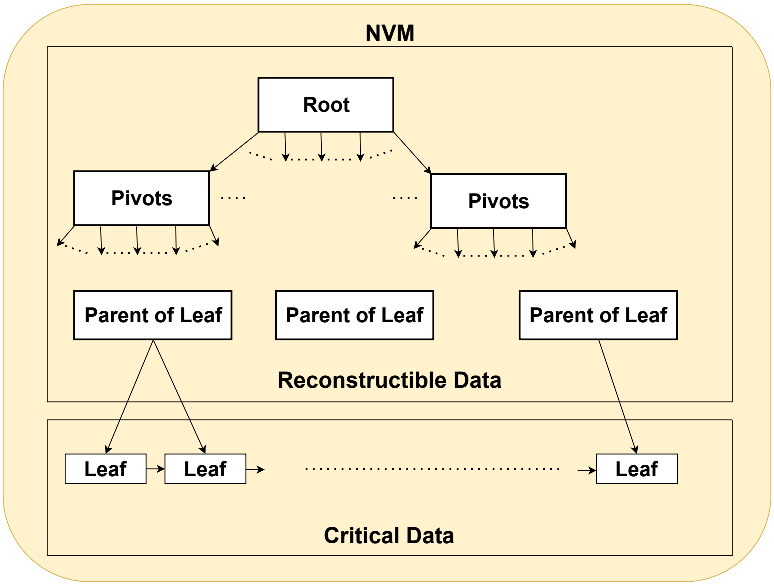


Figure 3.2: Architecture of NV-Tree [YWW+16]

of leaf keys. This helps to avoid the high complexity of search and range queries resulting from the higher access latency of PMem.

Each leaf node is divided into three parts (see Figure 3.3). The first part is a bitmap that tracks the validity of the entries. The second part is an array of fingerprints, where each fingerprint corresponds to a key. The last one are the key-value pairs.

Furthermore, for handling the concurrency of internal and leaf nodes, FP-tree uses hardware transactional memory (HTM) [CBB21] and fine-grained locks respectively.

FPTree has also some drawbacks in terms of efficiency. As shown in [HCLH20; ZWF+23], the search improvements made compared to NV-Tree are limited when the node size is small.

3.6 FAST and FAIR

`FAST-FAIR` is a byte-addressable persistent B^+-tree optimized for NVM. It was introduced in 2018 by Hwang et al. [HKWN18] to address the granularity mismatch between NVM hardware constraints and B^+-tree operations. In this work, the authors introduced two algorithms to enable high-performance, crash-consistent, and to solve the presented limitations:

- Failure-atomic shift (FAST): This algorithm decomposes node updates into sequences of 8-byte atomic writes. This ensures that the B^+-tree remains fully consistent or in a transiently inconsistent but detectable state [HKWN18]. Search operations can skip incomplete updates, which makes them lock-free.
- Failure-atomic in-place rebalance (FAIR): Same as FAST, this algorithm enables node splits and merges to be performed in-place, using ordered 8-byte writes.

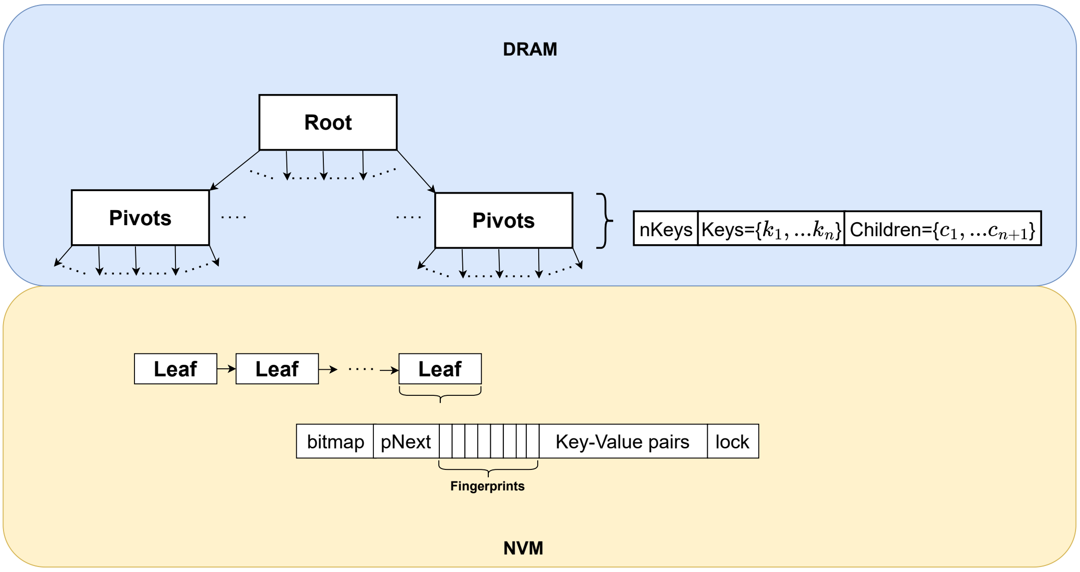


Figure 3.3: Architecture of FP-Tree [OLN+16]

These operations are implemented without logging or CoW mechanisms. Furthermore, sibling pointers are saved for both leaf and internal nodes to support lock-free search.

In `FAST-FAIR`, all nodes reside directly in byte-addressable NVM, which makes it a non-hybrid model. Leaf nodes are sorted, and when searching for a key, the tree performs linear scans (see Table 3.1) over leaf nodes instead of binary search. According to [HKWN18], although binary search is generally considered faster, it cannot be used with the lock-free search algorithm. Moreover, binary search performs worse than linear search for small node sizes.

3.7 Hybrid B+-Tree

Hybrid B+-tree (HBtree) is a further variant of B+-Tree that was introduced in 2021 by Zhou et al. [ZSW21]. This tree is hybrid, as it combines both NVM and DRAM. As shown in Figure 3.4, the tree consists of three layers:

- **Index layer**: This layer is a B+-tree in DRAM, which is used to index the middle layer metadata nodes.
- **Middle layer**: This layer is a double-linked list in NVM. Each node points to a `LogTree` in the data layer and stores the access frequency of the corresponding `LogTree`. This helps identify hot and cold data.
- **Data layer**: This layer consists of multiple `LogTrees`. Each `LogTree` consists of a `CacheTree`, which stores a hot `NVMTree` DRAM, an `NVMTree` in NVM, and a log tree that is responsible for recovering `CacheTree` data.

This structure helps to maintain the consistency of the tree after a crash. On restart, the middle layer (persisted in NVM) is traversed to reconstruct the index layer. `CacheTree` and `NVMTree` are rebuilt through logs and metadata.

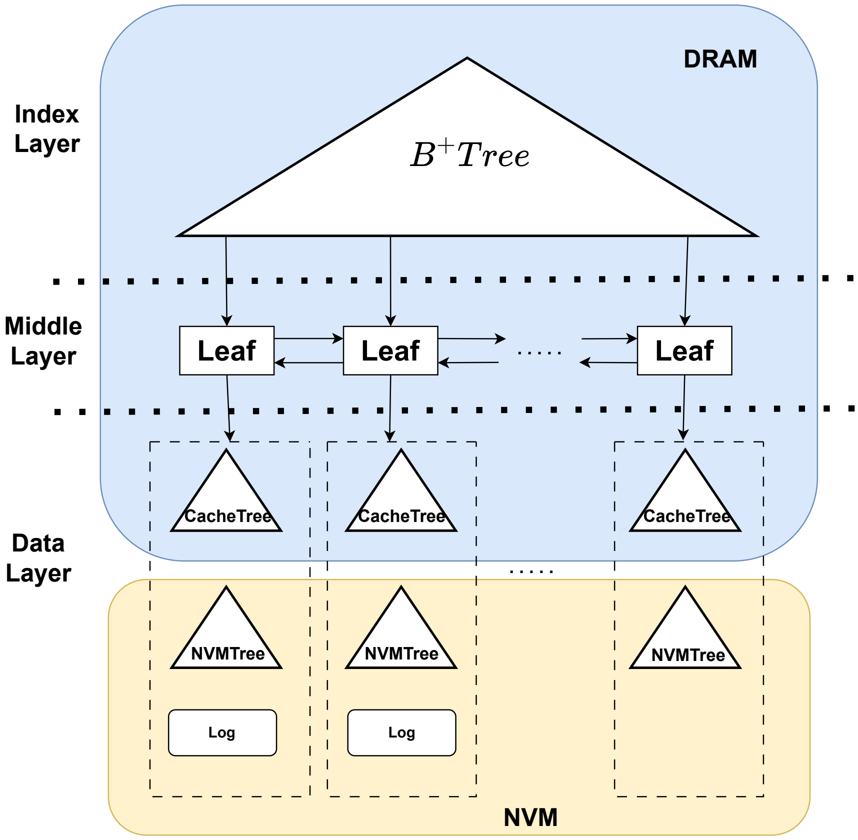


Figure 3.4: Architecture of HBtree [ZSW21]

3.8 LB$^+$-Tree

LB$^+$-Tree is a persistent B$^+$-tree proposed in 2020 by Liu et al. [LCW20]. It was specifically optimized for 3D XPoint (Intel® Optane™ DC persistent) memory, which implies that the nodes have a size that is a multiple of 256 bytes. The work proposes three techniques to improve the insert operation performance:

- **Entry moving**: Empty slots in the header of a leaf node are created. The goal is to reduce the number of NVM line writes.
- **Logless node splitting**: To avoid the extra overhead of logging when splitting a node, NVM atomic write (NAW) is used. It allows updating sibling pointers during splits.
- **Distributed headers**: Each leaf node is divided into m blocks of size 256 bytes. This makes the tree effective for multi-256B nodes and ensures logless splits across the entire node, which reduces write and update overhead.

Figure 3.5 presents an overview of the architecture of this data structure. The LB$^+$-Tree is a hybrid tree, since the leaf nodes are stored in NVM and the internal nodes in DRAM (see Table 3.1).

This data structure relies on 256 bytes access granularity and NAW support in 3D XPoint, which makes it not suitable for other NVM memories.

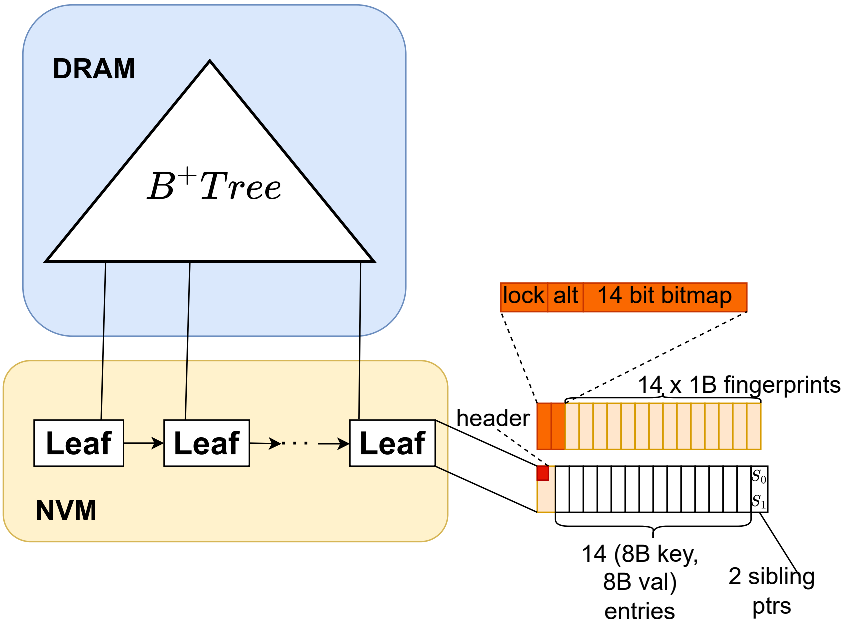


Figure 3.5: Architecture of LB$^+$-Tree [LCW20]

3.9 Haura

Haura [Hö21; Wie18] is a hybrid storage engine that integrates B$^\epsilon$-tree index on NVM. It follows the architecture proposed in ZFS [FHJ18], and is a B$^\epsilon$-tree write-optimized storage stack. It was first introduced and implemented in [Wie18] as a storage stack with key-value store, and then extended in [Hö21] to support object and tiered storage. Figure 3.6 presents the key components of Haura:

Object store

It is the first component of the architecture that was added in [Hö21]. It manages basic object operations such as create, read, write, and delete. Furthermore, it manages arbitrarily sized objects and introduces a chunking mechanism for large objects, which assigns a unique identifier to each chunk. Each chunk has a fixed size of 128 KiB. This mechanism allows for efficient partial reads and writes.

Database

This layer manages datasets and snapshots, which provide a key-value store interface. Snapshots are implemented as read-only copies using CoW [DSST89]. The used data structure is B$^\epsilon$-tree. It also provides basic database operations such as open, close, get, and range, and consists of multiple datasets. Beside B$^\epsilon$-trees that are used for both datasets and their corresponding snapshots, there is a separate, central B$^\epsilon$-tree called root tree. It allows storing information about the datasets and snapshots. Each dataset has a unique id (`DatasetId`).

Tree

The tree layer contains the implementation of the B$^\epsilon$-tree. It provides many operations, such as insert, delete, and upsert, which can be used by the upper layer (database). Each B$^\epsilon$-tree has a unique id (`TreeId`), which is determined by the upper layer. The implementation follows the architecture provided in [BFCJ+15]. Leaf nodes store key-value pairs, and internal node buffers store key-message pairs. According to [Wie18], keys are ordered lexicographically. However, the implemented B$^\epsilon$-tree has several improvements, such as support for variable-sized keys, values, and messages, as well as relaxed node size bounds.

Data management layer

Data management layer (DML) handles B$^\epsilon$-tree structural operations such as split, merge, and buffer flush. It interacts directly with the cache system to minimize disk access and supports compressing and decompressing on-disk objects.

Storage pool

The storage pool combines multiple storage devices and manages data allocation across them. This layer manages queues for asynchronous input/output (I/O) and also supports direct synchronous I/O [KWB+23]. It includes an extra offset (`DiskOffset`), which specifies the ID of the vdev where the data is stored.

Vdev

A vdev consists of multiple virtual devices and operates on a 4 KiB block size [Wie18]. Vdev is responsible for managing and interfacing with the underlying storage devices. In Haura, there are three different vdev implementations: `single`, `mirror`, and `parity1`.

`Single vdev`, as the name says, operates on single storage devices without redundancy. This implementation is simple and has minimal overhead. However, as there is no redundancy, data is lost if the device fails.

On the other hand, `mirror vdev` replicates data across multiple devices. It consists of at least two storage devices. This implementation provides high fault tolerance through duplication but requires more storage.

The last implementation is called `parity1`. It consists of at least three storage devices. This implementation uses one block device for parity and distributes data across other devices. It supports recovery from a single device failure and provides fault tolerance with lower storage overhead compared to mirroring. However, write operations are slower due to parity calculations.

Each layer provides interfaces to the layers above it, making the architecture modular and flexible. Despite this, Haura does not fully take advantage of the capabilities of NVM, presented in Section 2.6, because of two main limitations [KWB+24]:

- No direct reads: Read operations cannot access nodes that are in NVM. They must be copied into DRAM, which introduces additional overhead.
- No in-place updates: Haura handles node updates through CoW, which ensures the consistency of update operations. This prevents the use of in-place updates, which reduces write amplification.

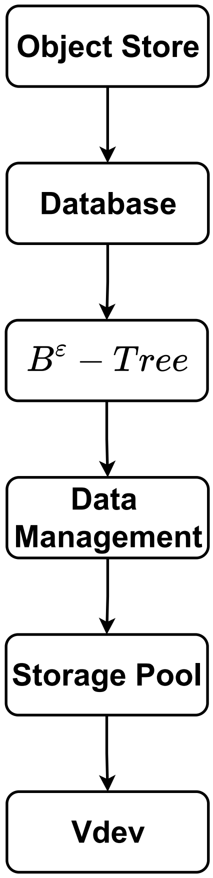


Figure 3.6: Components of Haura [Hö21; KWB+23; Wie18]

3.10 Multi-Write-Mode B$^\epsilon$-Tree

One of the most recent studies in the field of PMem introduces a new data structure called multi-write-mode B$^\epsilon$-tree (mw-B$^\epsilon$ tree), proposed by Luo et al. [LC25] earlier this year. The main idea of this research is to integrate multi-write mode support in NVM to reduce B$^\epsilon$-tree construction costs and improve energy efficiency. The authors proposed two core principles that the tree follows:

- **Retention-time awareness**: The flush operation and retention time information are linked together. Each node has its own retention time. Once this time expires, the keys of the node are flushed to the appropriate child node. The retention-time-aware flushing mechanism includes four different steps:
  1. Periodic retention time check: nodes that have an expired retention time need to be identified for rewriting. As presented in Figure 3.7, each node has a retention information (R_{Tn}), which is stored in metadata `n_reten` in DRAM.
  2. Bit vector loading: After identifying nodes that have an expired retention time, the bit vector (`Bit_vec`), which is in the metadata `n_bitmap` in DRAM (see Figure 3.7), of the corresponding nodes is loaded. Keys that have a zero in the `Bit_vec` are treated as flushed to the appropriate child and removed from the buffer.
  3. Parent node update: This step ensures that, after flushing keys from a node to its child, the parent indices are updated.
  4. Bit vector update: The bit vector is updated to mark flushed keys.
- **Dynamic mode selection**: The tree dynamically selects the appropriate write mode based on the update frequency of a node (hot or cold) and the type of write operation. This optimizes overall performance and reduces power consumption.

As shown in Figure 3.7, the mw-B$^\epsilon$ tree stores frequently updated metadata (i.e., bitmaps and retention timers) in DRAM, while the tree is in multi level cell phase change memory (MLC-PCM), which is an NVM technology (see Section 2.6). This makes the architecture of this data structure hybrid.

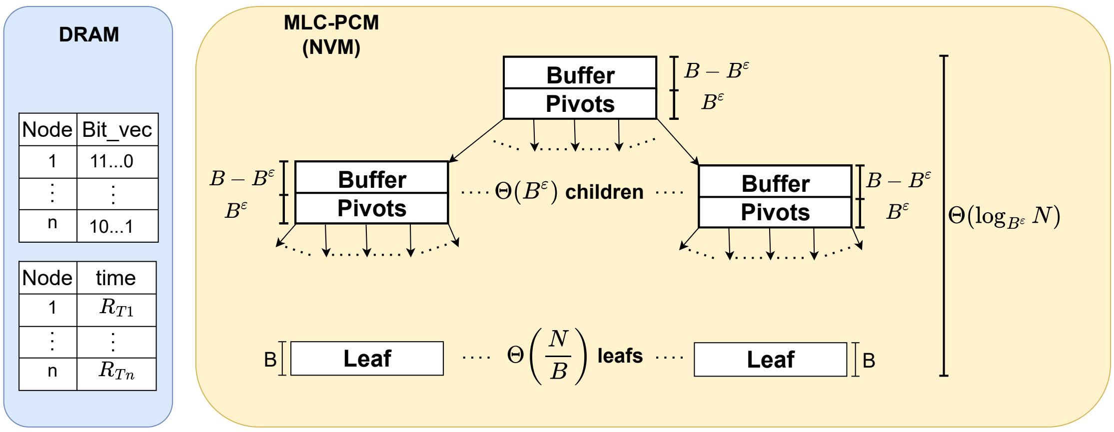


Figure 3.7: Architecture of mw-B$^\epsilon$-tree [LC25]

Tree	Sorted Leaf	Design	Leaf Search	Crash Recovery Strategy
NoveLSM [KBG+18]	Yes	Hybrid (DRAM + NVM + SSD)	Parallel lookup across levels	Persistent skip-list
CDDS B-Tree [VTRC11]	Yes	NVM	Binary search	Versioning + Scan entire data structure
wB^{+}-tree [CJ15]	No (slot arrays are sorted)	NVM	Slot array or bitmaps	Undo-Redo Logging + atomic writes
NV-Tree [YWW+16]	No	NVM	Linear search	Rebuild tree from leaf nodes
FPTree [OLN+16]	No	Hybrid (NVM + DRAM)	Fingerprints	Rebuild internal nodes from leaf nodes
`FAST-FAIR` [HKWN18]	Yes	NVM	Linear search	Detectable transient inconsistency + instant recovery
HBtree [ZSW21]	Yes	Hybrid (NVM + DRAM)	Query the `CacheTree` in DRAM	Multi-phase recovery
LB^{+}-Tree [LCW20]	Semi-sorted leaf	NVM	Fingerprints	Rebuild internal nodes from linked list of leaf nodes

Table 3.1: Overview of different NVM data structures [GvRL+18; ZWF+23]

4. Design of the Unified B$^{\epsilon}$-Tree

In this chapter, the design proposed in [KWB+24], which is implemented in this thesis, is presented and discussed. Furthermore, the goals of this proposal are presented. The unified B$^{\epsilon}$-tree design addresses the challenges of integrating heterogeneous storage technologies, including DRAM, NVM, SSD, and/or HDD, within a single tree structure. First, the design goals are presented, followed by an overview of the system architecture and its components. Afterward, the key differences that the design proposes and how it differs from the B$^{\epsilon}$-tree proposed in [BFCJ+15], as well as the improvements and specific adaptations made to better support heterogeneous storage environments, are discussed in detail.

4.1 Design Goals

The primary goal of the design of the unified B$^{\epsilon}$-tree proposed by Karim et al. [KWB+24] is to efficiently support heterogeneous storage systems. As presented in Table 3.1, most of the related works cited have not taken into account the heterogeneity of modern storage, except for [KWB+23; LC25]. Furthermore, they have primarily focused on optimizing B^{+}-trees for NVM, except for [Hö21; KWB+23; LC25; Wie18]. They did not explore optimizations for B$^{\epsilon}$-trees, despite their inherent efficiency and superior write performance in high-throughput environments.

The design introduces a NVM-optimized B$^{\epsilon}$-tree with the following goals:

- **Direct reads on NVM**: The tree should support direct read operations on NVM, unlike [Hö21; Wie18]. This prevents copying nodes to DRAM, which reduces read latency and memory overhead.
- **In-place updates**: Same as above, direct updates in NVM minimize write amplification.
- **Unified tree structure**: The design should have a unique B$^{\epsilon}$-tree that spans heterogeneous storage layers, including DRAM, NVM, SSD, and/or HDD, while allowing flexible node placement across these layers.

- **Heterogeneity awareness**: As the tree spans across different layers, the proposed design can manage data across storage layers with different performance characteristics while minimizing storage-specific handling complexity.
- **Efficient value buffering in DRAM**: As mentioned in [LC25], the concept of hot and cold nodes can significantly improve read performance. The design caches frequently accessed nodes in DRAM to minimize access to the storage device.
- **Optimized node layout**: The design allows for the movement of nodes between DRAM and NVM. This implies that the layout should allow for quick and easy transformation.

These goals address both performance and architectural flexibility to ensure that the storage engine can operate efficiently across DRAM, NVM, SSD, and/or HDD. Furthermore, they optimize the B$^\epsilon$-tree for efficient use in PMem technologies.

4.2 Architecture Overview

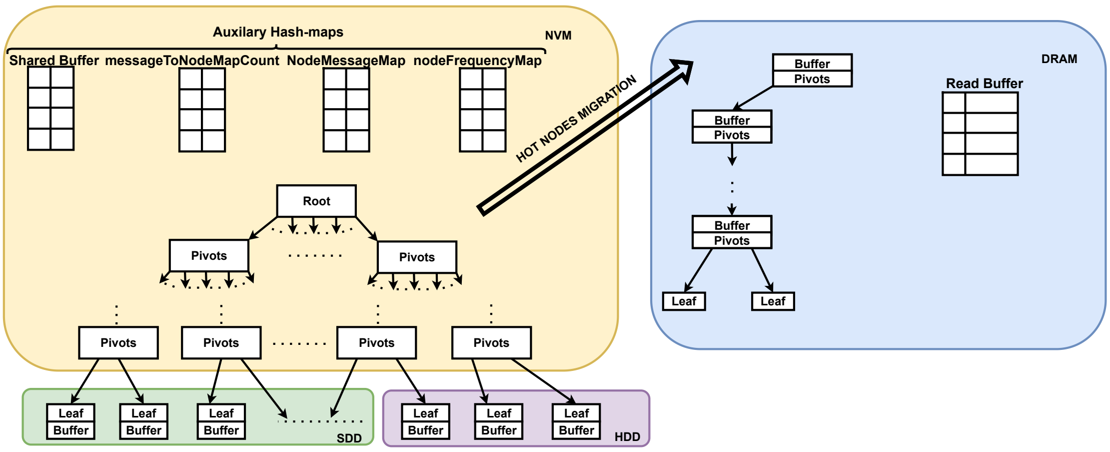


Figure 4.1: Architecture of the unified B$^\epsilon$-tree [KWB+24]

The unified B$^\epsilon$-tree design introduces a storage engine capable of operating efficiently across a heterogeneous memory hierarchy. Within a single tree, the system integrates multiple storage layers, including DRAM, NVM, SSD, and/or HDD.

The system architecture enables direct access to nodes stored in NVM, eliminating the need to copy data into DRAM for processing. In contrast to previous designs such as Haura [Hö21; Wie18], where nodes in NVM are treated identically to those in SSD or HDD, this system fully uses the unique capabilities of PMem by supporting in-place updates and direct reads.

Figure 4.1 presents the architecture of the proposed design. The tree architecture differs from the one in [BFCJ+15]. As described in Section 2.3, each internal node usually contains a buffer where all messages belonging to its children are stored. The design eliminates these per-node buffers and introduces a centralized shared buffer in NVM, which is used by all internal nodes in NVM. This buffer is associated with an auxiliary hash table that stores pointers to messages (i.e., to which node the message belongs). The shared buffer efficiently batches write operations, such as inserts and deletes, and reduces flushing overhead (see Figure 4.1).

Internal nodes are stored across NVM, DRAM, and SSD, but their layouts differ depending on the storage layer. In NVM, internal nodes do not maintain individual buffers; instead, they utilize a centralized shared buffer. In contrast, internal nodes stored in DRAM and SSD retain traditional per-node buffers.

On the other hand, leaf nodes are stored in SSD and/or HDD. It is also important to note that the leaf nodes can have a small buffer in which messages are saved (see Figure 4.1). If nodes are frequently accessed, they will become hot nodes.

The design implies that hot nodes should have working copies that exist in DRAM. Moreover, it introduces a read buffer in DRAM, where recent read values are stored. The goal of this is to reduce the read overhead and to benefit from read-recent workloads [KWB+24].

4.3 Internal Nodes

In the unified B^{ϵ}-tree design, internal nodes can be directly stored and accessed in NVM without being copied into DRAM. By enabling direct reads and in-place updates on NVM, the design fully leverages the low-latency and byte-addressable characteristics of PMem. As shown in Figure 4.1, the internal nodes in NVM contain keys and pivots, and no buffers. It serves, as in B^{+}-tree, for routing purposes. Instead, the centralized shared buffer is used to save messages. This separation simplifies the node layout in NVM, reduces memory overhead, supports efficient direct access, and allows direct access in NVM for search operations.

The design suggests that when a node becomes hot, a working copy is created in DRAM. However, the layout of the working node differs from the original in NVM. The inner node in DRAM has keys, pivots, and a buffer where messages are stored. This implies that the migration process should transform the node into the appropriate structure to ensure the consistency of the tree. When migrating a node from NVM to DRAM, key-value pairs are directly copied from the node, but the messages should be looked up in the shared buffer and retrieved to the buffer of the corresponding node. This reduces migration overhead.

Furthermore, the design improves search operations by allowing direct access to only the required portions of the data on NVM, rather than loading full data blocks into DRAM as in [KWB+23]. Moreover, the read buffer in DRAM reduces both latency and memory bandwidth consumption.

This flexibility allows the unified B^{ϵ}-tree to dynamically balance performance and durability across heterogeneous storage layers while maximizing the direct benefits of NVM.

4.4 Shared Buffer in NVM

In the original B^{ϵ}-tree design [BFCJ+15], each internal node maintains its own local buffer to temporarily store messages destined for its child nodes. When the buffer is full, messages are flushed to the correct child buffer or applied to leaf nodes. The unified B$^\epsilon$-tree eliminates per-node buffers for internal nodes stored in NVM. Instead, it introduces a centralized shared buffer that resides directly in NVM and is accessible by all internal nodes located in NVM. All messages are saved in the buffer. Furthermore, to efficiently track buffered messages, this buffer is associated with an auxiliary hash table in which pointers to the messages are stored. It maintains mappings between buffered messages and their corresponding target nodes.

The centralized shared buffer supports message coalescing. If a new message arrives with the same key as an existing one, the buffer merges or coalesces both messages. This minimizes buffer usage, eliminates redundant writes, and improves flush efficiency.

Only the auxiliary hash table needs to be updated when flushing the shared buffer, since only the pointers in the auxiliary hash table require modification [KWB+24]. By using this approach, the internal nodes are not required to be changed directly during the flush process. This increases flush efficiency and reduces write overhead.

4.5 Read Buffer in DRAM

The proposed design suggests including a centralized read buffer in DRAM. As the volatile memory offers lower read access latency compared to NVM, which can have up to four times higher latency than conventional DRAM [POL+19], the read buffer improves the performance of frequently accessed values.

The read buffer stores recently accessed nodes to prevent redundant reads from NVM, SSD, or HDD. This helps prevent traversing the entire tree or accessing the shared buffer in NVM when searching for a key.

As the read buffer has a fixed size, a replacement strategy needs to be defined. This ensures that only frequently accessed values are retained, and less frequently nodes are evicted when the buffer is full. The design employs an least recently used (LRU) policy.

The DRAM-based read buffer improves read performance, reduces storage device access, and lowers the load on the shared buffer and the NVM layer.

4.6 In-place Writes

Contrary to Haura, the unified B$^\epsilon$-tree supports in-place write operations on nodes stored in NVM. This implies that nodes in NVM must not be moved to DRAM for write operations, which reduces write amplification by modifying only the necessary parts of a node or buffer.

This design ensures that buffered messages are written sequentially within each block, enabling efficient block-aligned writes that are compatible with both NVM and SSD. This reduces the need for complex partial block updates and lowers I/O overhead.

The in-place update uses the shared buffer and auxiliary hash table in NVM presented in Section 4.4, to ensure that message coalescing and pointer management remain consistent. With this feature, the unified B$^\epsilon$-tree supports efficient writes and allows the system to fully exploit the low-latency and byte-addressable nature of NVM.

4.7 Summary of Key Advantages

As described in the previous sections, the unified B^ϵ-tree design introduces several key improvements that specifically target the challenges of heterogeneous storage hierarchies and fully leverage the capabilities of PMem.

The design supports **direct reads from NVM**, which eliminates the need to copy nodes into DRAM before processing. This feature reduces, as described in Section 4.3, the memory transfer overhead, minimizes latency, and makes search operations more effective.

The design also supports **in-place updates**, which allow the modification of nodes that are in NVM without relying on traditional read-modify-write cycles or CoW mechanism. This makes the tree more write-optimized and reduces I/O and computation costs.

Furthermore, the tree eliminates the node buffers from the internal node in NVM and instead uses a **centralized shared buffer** that is associated with an **auxiliary hash table**. This simplifies the node structure, supports efficient message coalescing, and reduces the complexity of flush operations.

Moreover, a **DRAM-based read buffer** is used to improve the overall performance of the tree by caching frequently accessed values, which reduces read latency and minimizes unnecessary storage accesses.

In addition, the design manages **optimized node layouts** across DRAM, NVM, and SSD, since different node structures are present in each layer. This reduces fragmentation and enables efficient node transitions between storage layers.

Overall, the unified B^ϵ-tree hybrid design offers a scalable, storage-aware solution that fully exploits the performance and persistence benefits of NVM.

5. Implementation

This chapter discusses the steps taken to implement the unified B$^\epsilon$-tree design. First, a baseline B$^\epsilon$-tree, as described by Bender et al. [BFCJ+15] was implemented and tested for correctness. This baseline was then extended with the various features discussed in the previous chapter (Chapter 4). The implementation focuses on integrating heterogeneous storage management, supporting direct reads and in-place updates in NVM, and introducing shared and read buffers to optimize performance across different storage layers.

5.1 Baseline B$^\epsilon$-Tree

Since there are no publicly available and extensible B$^\epsilon$-tree implementations in C++, which is the programming language chosen for the implementation, the development process began with creating a baseline B$^\epsilon$-tree based on the design proposed by Bender et al. [BFCJ+15].

5.1.1 Node Structure and Tree Initialization

The node structure was implemented using a template class that stores metadata, keys, child pointers, and buffer entries. The structure of the node is shown below (see Listing 5.1).

Listing 5.1: Simplified Internal Node Structure

```cpp
template <typename KeyType, typename ValueType>
struct Node {
    bool isLeaf;
    std::vector<KeyType> keys;
    std::vector<std::shared_ptr<Node>> children;
    std::vector<Message<KeyType, ValueType>> buffer;
    size_t bufferSize;
    size_t maxBufferSize;

    Node(bool leaf) : isLeaf(leaf), bufferSize(0) {}
};
```

Each internal node contains a sorted array of keys, pointers to child nodes, and a buffer to store messages. In contrast, the leaf nodes contain sorted key-value pairs without a buffer (`bufferSize` = 0).

The B$^\epsilon$-tree is initialized by specifying the node degree ($m\_nDegree$), which defines the maximum number of keys in each node, and the buffer size (`bufferSize`), which specifies the maximum number of buffered messages per node. Listing 5.2 shows the constructor responsible for creating the initial tree structure. At initialization, the tree creates a new root node, which is always a leaf at the beginning.

Listing 5.2: Tree initialization

```
BEpsilonTree(int m_nDegree, int bufferSize)
{
    root = std::make_shared<Node<KeyType, ValueType>>(true);
    this->m_nDegree = m_nDegree;
    this->bufferSize = bufferSize;
}
```

5.1.2 Buffering Mechanism

Each internal node has a corresponding buffer, where messages that belong to the children of the node are saved (see Figure 2.3).

The buffer is implemented as a vector of message objects, each associated with a key, a corresponding value (except for delete operations), and an operation type (i.e., insert, delete, update). Listing 5.3 shows how a message is inserted into a node buffer. The `insertBuffered` is associated with a coalescing strategy that updates, replaces, or discards buffered messages based on the operation type. This ensures the consistency of the buffer and the entire tree. If no existing message for the key is found, the new operation is appended to the buffer.

When a new message for a key is inserted into the buffer, the system checks whether a message for the same key already exists. If so, the existing message is replaced with the most recent one, effectively coalescing messages and minimizing redundancy.

5.1.3 Buffer Flushing

In the B$^\epsilon$-tree, each internal node maintains its own buffer with a fixed maximum capacity. When this capacity is reached, the buffer becomes full and cannot accept new messages. This implies that the content of the corresponding buffer must be flushed to the appropriate child node. If the child is an internal node, the messages are passed on to its buffer, otherwise, they are applied to the leaf node itself. After all messages are propagated to the child nodes, the buffer of the parent node is cleared. Algorithm A.6 presents detailed pseudocode explaining how the flushing mechanism works. It allows write batching, which reduces the number of frequent writes to lower levels of the tree.

Listing 5.3: Inserting a message into the buffer

```
template <typename KeyType, typename ValueType>
  ErrorCode insertBuffered(std::shared_ptr<Node<KeyType, ValueType>> node,
    Operations operation, KeyType key, ValueType value)
    {
        auto it = std::find_if(node->buffer.begin(), node->buffer.end(),
            [key](const tuple<Operations, KeyType, ValueType>& op) {
                return std::get<1>(op) == key;
            });

        if (it != node->buffer.end())
        {
            Operations opType = std::get<0>(*it);

            switch (operation)
            {
            case Operations::Insert:
                if (opType == Operations::Insert)
                {
                    *it = { Operations::Insert, key, value };
                }
                else if (opType == Operations::Delete)
                {
                    return ErrorCode::Success;
                }
                else if (opType == Operations::Update)
                {
                    *it = { Operations::Insert, key, value };
                }
                break;
            case Operations::Delete:
                if (opType == Operations::Insert || opType == Operations::Update)
                {
                    *it = { Operations::Delete, key, ValueType{} };
                }
                else if (opType == Operations::Delete) {
                    return ErrorCode::Success;
                }
                break;
            case Operations::Update:
                if (opType == Operations::Insert || opType == Operations::Update)
                {
                    *it = { Operations::Update, key, value };
                }
                else if (opType == Operations::Delete)
                {
                    return ErrorCode::Error;
                }
                break;
            default:
                return ErrorCode::Error;
            }
        }
        else
        {
            node->buffer.emplace_back(operation, key, value);
        }
        return ErrorCode::Success;
    }
}
```

5.2 Tree Operations

The insert, search, update, and delete operations are supported by the B$^\epsilon$-tree, as previously described in Chapter 2. In the baseline implementation, these operations were adapted to support the internal buffering mechanism and the message coalescing strategy described in this chapter. This ensures that buffered messages are consistently processed during flushes or read operations, preserving the correctness of the tree. Additional pseudocode and specific implementation details for these operations are provided and explained in Chapter 2.

5.3 Extensions to the Baseline B$^\epsilon$-Tree

After implementing the baseline B$^\epsilon$-tree, the features presented in [KWB+24], which address the limitations of the baseline structure, such as the lack of support for heterogeneous storage, direct NVM access, shared buffering, and efficient read optimization, are implemented.

5.3.1 Programming Language and NVM Library Selection

The implementation of the unified B$^\epsilon$-tree is developed in the programming language C++. This choice was made because of its efficient memory management and support for low-level system operations. In addition, the majority of related works [CJ15; HKWN18; LC25; LCW20; OLN+16; VTRC11; YWW+16; ZSW21] presented in Chapter 3 were implemented in C/C++ (Haura [Hö21; KWB+23; Wie18] was initially implemented in Rust). That was another reason to choose C++ over other programming languages, as this ensures that performance comparisons and evaluations remain consistent.

The persistent memory development kit (PMDK), originally introduced by Intel in 2014 under the name non-volatile memory library (NVML) [Int17], is used in this thesis as the core library for managing NVM. PMDK is implemented in C, and its libraries are designed to interface closely with hardware features such as cache line flushing, memory fencing, and non-volatile memory addressability. It supports other programming languages such as C++, java, and python but was only for C and C++ fully tested. This was another reason for choosing C++ as the programming language for the proposed design.

PMDK[3] includes several libraries [Int17; Sca20b]. Some of these are:

- **`libpmem`** [Inta]: Provides low-level PMem support.
- **`libpmemobj`** [Intd]: Provides memory management, transactions, and persistent pointers.
- **`libpmemblk`** [Intb]: Is used for use cases, where large arrays are needed.
- **`libpmemlog`** [Intc]: Provides append-only log functionality in PMem.
- **`libpmempool`** [Inte]: Supports PMem pool management and consistency checking.

In this implementation, the library `libpmemobj` is mainly used (see Listing 5.7). This library provides the functionality required for PMem management. It also allows to create and manage the persistent pools. Furthermore, it supports memory allocation and persistent pointers through its different headers that were used in the work (e.g., `libpmemobj.h`, `base.h`, `pool_base.h`, and `atomic_base.h`). Furthermore, the following features were used:

[3] https://github.com/pmem/pmdk

- Persistent memory pools for storing all data structures (e.g., `pmemobj_create` and `pmemobj_open`).
- Type-safe persistent pointers (`TOID` macros).
- Persistent object layouts (`POBJ_LAYOUT_*` macros), including a root object. It helps in object recovery after a crash.
- Direct allocation (e.g., `pmemobj_alloc`) and deallocation (e.g., `pmemobj_free`) of persistent objects.
- Atomic and consistent updates to persistent memory (e.g., `pmemobj_persist`).

5.3.2 Node Structures

As explained in the previous chapter (see Chapter 4), the node structures in the unified B$^\epsilon$-tree differ depending on the storage layer in which they are saved.

Persistent Nodes

Persistent nodes are stored in the NVM layer. They represent internal nodes of the tree and have a different structure from that of standard B$^\epsilon$-trees. The node eliminates the buffer. Instead, a centralized shared buffer is maintained. The internal node stores only keys and pivot pointers. The structure of the persistent nodes is defined as follows:

Listing 5.4: Structure of Persistent Nodes

```
struct PersistentNode
{
    uint8_t isLeaf;
    size_t keyCount;
    uint64_t keys[MAX_KEYS_PER_NODE];
    uint64_t children[MAX_KEYS_PER_NODE + 1];

    // Buffer for update messages
    char buffer[NODE_BUFFER_SIZE];
    size_t buffer_offset;
};
```

Volatile Nodes

As mentioned previously, the tree supports a migration mechanism: when a node is frequently accessed, it is marked as a hot node and migrated to DRAM. The node becomes a volatile node and has a different structure from the previous one. The working copy of the node has now a buffer, similar to the default internal nodes in standard B$^\epsilon$-trees. The structure of the volatile node is defined as follows:

Listing 5.5: Structure of Volatile Nodes

```
template <typename KeyType>
struct VolatileNode
{
    bool isLeaf;
    size_t keyCount;
    KeyType keys[MAX_KEYS_PER_NODE];
    uint64_t children[MAX_KEYS_PER_NODE + 1];
    message dramBuffer[MAX_DRAM_MESSAGES];
    int dramMessageCount;

    std::unordered_map<KeyType, int> keyIndexMap;

    VolatileNode()
        : isLeaf(false), keyCount(0), dramMessageCount(0)
    {
        std::memset(keys, 0, sizeof(keys));
        std::memset(children, 0, sizeof(children));
        std::memset(dramBuffer, 0, sizeof(dramBuffer));
    }
};
```

Each DRAM node maintains its own buffer (`dramBuffer`), which can store up to `MAX_DRAM_MESSAGES`. A fast in-memory hash map (`keyIndexMap`) is used to quickly locate the position of keys within the node. A counter is also defined to track the current number of buffered messages stored in the `dramBuffer` array. This helps control the buffer flushing when the node becomes full.

Disk-Based Leaf Nodes

Figure 4.1 shows that leaf nodes are stored in SSD or HDD. These leaf nodes differ slightly from those of the classic B^{ϵ}-tree. In the unified B^{ϵ}-tree leaf nodes include a small buffer that is used to store minor updates. This feature is useful as flushing small, random updates from internal nodes in NVM to a large number of leaf nodes causes huge write amplification. SSD leaf nodes are serialized to disk and are structured as follows:

Listing 5.6: Structure of Leaf Nodes

```
struct SerializedLeafNode {
    uint8_t isLeaf = 1;
    size_t keyCount = 0;
    uint64_t keys[MAX_KEYS_PER_NODE] = { 0 };
    uint64_t values[MAX_KEYS_PER_NODE] = { 0 };
    size_t buffer_offset = 0;
    char buffer[NODE_BUFFER_SIZE] = { 0 };
};
```

Each leaf node stores keys and their corresponding values in a fixed size array (`MAX_KEYS_PER_NODE`). It also maintains a local buffer with a fixed size as well (`NODE_BUFFER_SIZE`) for minor updates, which reduces write amplification and delay expensive disk splits.

5.3.3 Write Operations

Since internal nodes do not have local buffers, operations are buffered directly to the centralized shared buffer. Operations such as insert, update, and delete differ from those described in Section 2.3.

Insert Operation

The insert operation is adapted to fully leverage the heterogeneous storage hierarchy and the shared buffer mechanism introduced for internal nodes in NVM. Algorithm 5.1 shows how the insert works.

When an insert operation with a new key-value pair is triggered, the tree first checks whether the key already exists in the centralized shared buffer in NVM. If this is the case, the tree coalesces the new insert message with the existing message by retaining only the most recent insert operation. This minimizes redundancy and ensures that only one message per key is kept in the shared buffer at any given time.

Furthermore, the frequency of the target internal node is checked using a predefined constant, `HOT_NODE_THRESHOLD`. If this threshold is reached, the target node is migrated to DRAM with the help of a migration mechanism (see Section 5.3.7).

When the shared buffer in NVM becomes full, a flush operation is triggered. Buffered messages are then propagated to the corresponding child nodes. During this process, coalescing rules are applied to prevent redundant message propagation.

Update Operation

The update operation follows the structure of the insert operation (see Algorithm 5.2). When a key is updated, the system first checks whether a buffered message for the same key already exists in the shared buffer and coalesces the messages if necessary. In cases where the update reaches a leaf node in SSD, the system attempts an in-place update by appending the message to the local buffer of the leaf on disk. If the leaf buffer is full, the leaf is normalized to apply pending messages.

Delete Operation

Similarly to insert and update, when a delete operation is triggered, it will be inserted into the NVM shared buffer. However, if a buffered message already exists for the same key (either an insert or an update), the delete operation coalesces with the existing message. Algorithm 5.3 shows how the delete operation is implemented.

5.3.4 Persistent Memory Management

In this thesis, PMDK, as mentioned and explained in Section 5.3.1, is used for the PMem structure of the unified B$^\epsilon$-tree. The proposed unified B$^\epsilon$-tree design leverages NVM to persist internal nodes, shared buffer, and auxiliary structures, while minimizing unnecessary memory transfers by directly processing nodes in NVM. The layout is registered under the name `bepsilon_layout` and is composed of several persistent data structures that organize the tree.

The persistent memory layout is registered using the `POBJ_LAYOUT` macros provided by PMDK. These macros are required by PMDK to define which data structures are part of the PMem pool and to track their relationships.

Algorithm 5.1 Insert operation in unified B$^\epsilon$-tree

```
Require: Key k, Value v
Log the insert operation.
if Tree is empty then
    Create SSD leaf root and store key-value pair.
    return Success
end if
if Root is an SSD leaf then
    if SSD leaf exists on disk then
        Load SSD leaf into memory.
        Try to append message to SSD leaf buffer.
        if Buffer is full then
            Normalize SSD leaf buffer.
            if Leaf is still full after normalization then
                Split SSD leaf and promote root.
            end if
        end if
        Save SSD leaf to disk.
        return Success
    else
        Fallback to buffered insert.
    end if
end if
Locate target internal node for buffering.
if Target node is hot (in DRAM) then
    if DRAM buffer is full then
        Flush DRAM buffer.
    end if
    Insert message into DRAM buffer.
    return Success
else
    Insert message into shared NVM buffer.
    return Success
end if
```

Algorithm 5.2 Update operation in unified B^{ϵ}-tree

```
Require: Key k, New Value v
Log the update operation.
if Tree is empty then
    Abort with error: key does not exist.
    return KeyNotFound
end if
if Root is an SSD leaf then
    if SSD leaf file exists then
        Load SSD leaf from disk.
        if Key not found in SSD leaf or buffer then
            Abort with error: key does not exist.
            return KeyNotFound
        end if
        Try to append update message to SSD leaf buffer.
        if Buffer is full then
            Normalize SSD leaf buffer.
            Retry append.
            if Buffer still full after normalization then
                Save SSD leaf to disk.
                Split SSD leaf.
                Fallback to buffered update.
            end if
        end if
        Save SSD leaf to disk.
        return Success
    else
        Fallback to buffered update.
    end if
end if
Buffered Update Path:
Locate target internal node for buffering.
if Target node is hot (in DRAM) then
    if DRAM buffer is full then
        Flush DRAM buffer.
    end if
    Insert update message into DRAM buffer.
    return Success
else
    Insert update message into shared NVM buffer.
    return Success
end if
```

Algorithm 5.3 Delete operation in unified B$^{\epsilon}$-tree

```
Require: Key k
Log the delete operation.
if Tree is empty then
    Abort with error: key does not exist.
    return KeyNotFound
end if
if Root is an SSD leaf then
    if SSD leaf file exists then
        Load SSD leaf from disk.
        if Key not found in SSD leaf or buffer then
            Abort with error: key does not exist.
            return KeyNotFound
        end if
        Try to append delete message to SSD leaf buffer.
        if Buffer is full then
            Normalize SSD leaf buffer.
            Retry append.
            if Buffer still full after normalization then
                Save SSD leaf to disk.
                Split SSD leaf.
                Fallback to buffered delete.
            end if
        end if
        Save SSD leaf to disk.
        return Success
    else
        Fallback to buffered delete.
    end if
end if
Buffered Delete Path:
Locate target internal node for buffering.
if Target node is hot (in DRAM) then
    if DRAM buffer is full then
        Flush DRAM buffer.
    end if
    Insert delete message into DRAM buffer.
    return Success
else if Node was recently evicted then
    Insert delete message into shared NVM buffer.
    return Success
else
    Insert delete message into shared NVM buffer.
    return Success
end if
```

Listing 5.7: PMDK Layout

```
#include <libpmemobj.h>
#include <libpmemobj/pool_base.h>
#include <libpmemobj/base.h>
#include <libpmemobj/atomic_base.h>

POBJ_LAYOUT_BEGIN(bepsilon_layout);
POBJ_LAYOUT_ROOT(bepsilon_layout, PMEMRoot);
POBJ_LAYOUT_TOID(bepsilon_layout, PersistentNode);
POBJ_LAYOUT_TOID(bepsilon_layout, message);
POBJ_LAYOUT_TOID(bepsilon_layout, KeyList);
POBJ_LAYOUT_TOID(bepsilon_layout, NodeMessageEntry);
POBJ_LAYOUT_TOID(bepsilon_layout, MessageToNodeEntry);
POBJ_LAYOUT_TOID(bepsilon_layout, NodeFrequencyEntry);
POBJ_LAYOUT_END(bepsilon_layout);
```

Listing 5.4 shows how all persistent types that will be stored in NVM are registered. It includes the layout name, `PMEMRoot`, which represents the root node of the unified B$^\epsilon$-tree and the entry point of the PMem structure. It also includes:

- **PersistentNode**: Represents internal nodes of the tree.
- **message**: Represents buffered operations (insert, update, delete). Each message stores the operation type, key, and value, and is used in the shared buffer and auxiliary maps
- **KeyList**: Used to track the list of keys buffered for each internal node.
- **MessageToNodeEntry**: Maps each buffered key to the internal node it currently belongs to.
- **NodeMessageEntry**: Maps node IDs to their corresponding `KeyList`, enabling fast lookup of buffered keys for each node.
- **NodeFrequencyEntry**: Tracks the frequency of node accesses, which is used to detect hot nodes that should be migrated to DRAM.

`PMEMRoot` is defined as follows:

Listing 5.8: Persistent Memory Root Structure

```
struct PMEMRoot
{
    TOID(PersistentNode)
        persistentRoot;

    TOID(message)
        messages[MAX_NVM_MESSAGES];
    int messageCount;

    TOID(NodeMessageEntry)
        nodeMessageMap[MAX_NODES];
    int nodeMessageMapCount;

    TOID(MessageToNodeEntry)
        messageToNodeMap[MAX_NODES];
    int messageToNodeMapCount;

    TOID(NodeFrequencyEntry)
        nodeFrequencyMap[MAX_NODES];
    int nodeFrequencyCount;

    int nextLeafFileID;
};
```

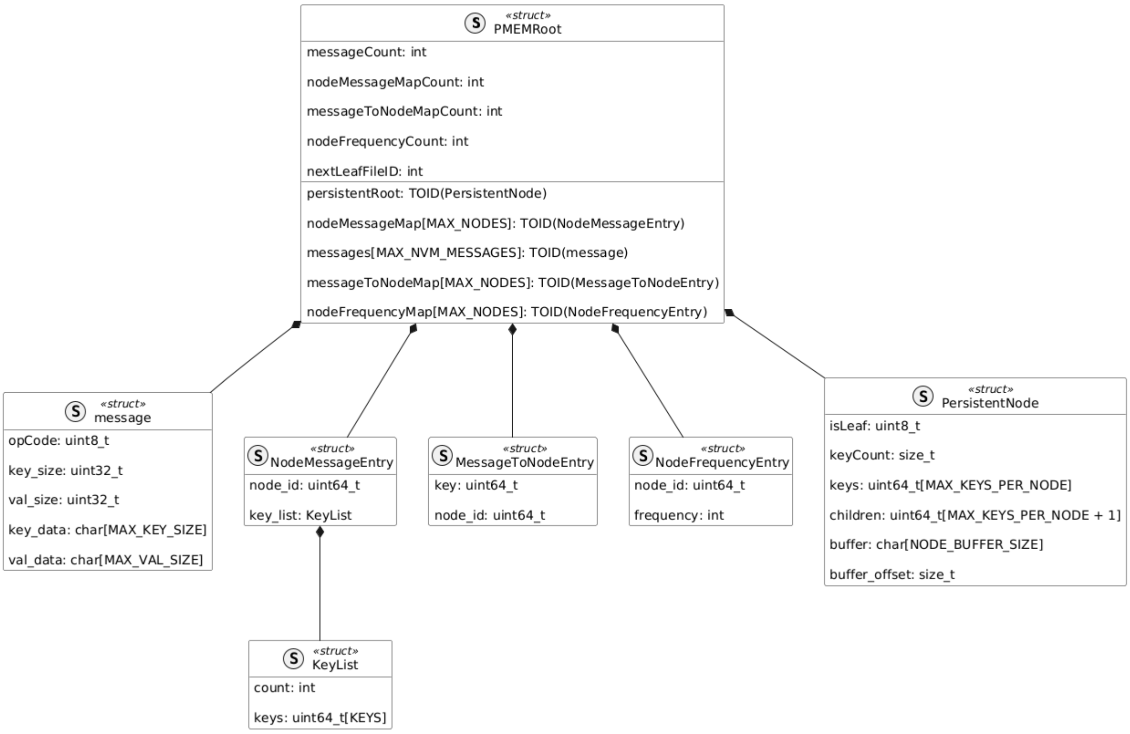


Figure 5.1: Persistent memory layout

Figure 5.1 shows the structure of the defined PMem. The `PMEMRoot` structure includes a persistent pointer to the root node of the B$^{\epsilon}$-tree, the centralized shared buffer, the auxiliary hash tables, the access frequency of internal nodes, and a persistent counter (`nextLeafFileID`) that identifies each leaf node stored on SSD.

5.3.5 Shared Buffer Logic

As shown in Figure 5.1 and Listing 5.8, the shared buffer is implemented as a persistent fixed size array of `message` structures within the `PMEMRoot`. Each `message` entry stores the type of operation (i.e., insert, update, or delete), the size of both the key and value, and their corresponding contents. A separate counter called `messageCount` is also defined to track the number of currently buffered messages in the shared buffer.

Furthermore, as the design suggests, three persistent auxiliary hash maps are defined. The `nodeMessageMap` assigns each internal node ID to the list of keys buffered for that node. The `messageToNodeMap` maps each buffered key to the internal node to which it belongs. The `nodeFrequencyMap` is used to track the access frequency of each node in the tree, enabling node migration to DRAM. All of these auxiliary hash maps are always kept up to date and support both the flushing and migration processes.

Insert into Shared Buffer and Auxiliary Hash Map Updates

The insertion of an operation into the shared buffer is performed using the `insertToNVMSharedBuffer` function. As shown in Figure 5.2, when an operation

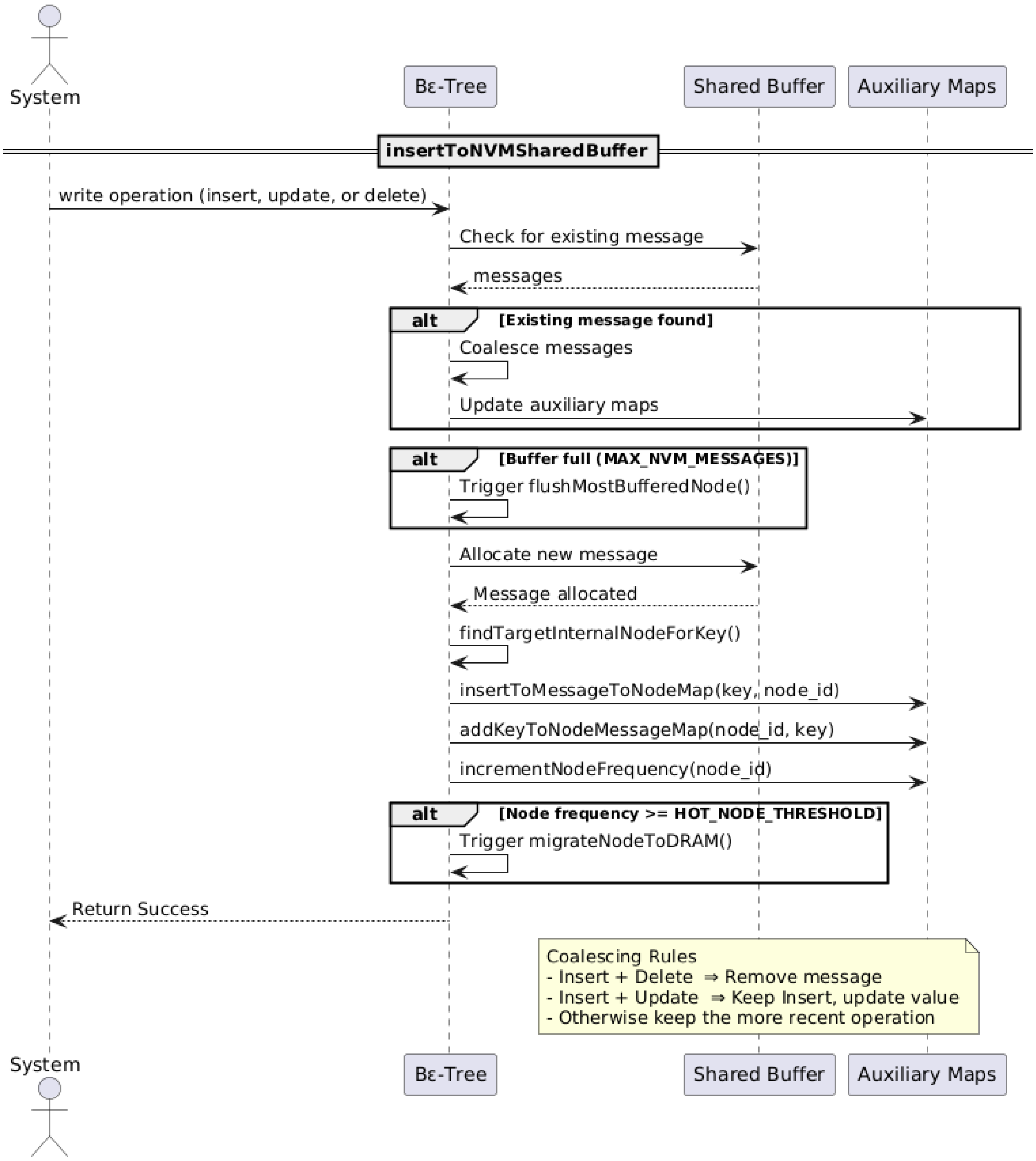


Figure 5.2: Sequence diagram of insertion of an operation into the shared buffer and updating of auxiliary map in NVM

arrives, the tree scans the shared buffer to check whether a message for the same key already exists for possible coalescence. If the message is merged, both auxiliary hash maps (i.e., `nodeMessageMap` and `messageToNodeMap`) are updated. The size of the buffer is then checked for possible flush operation. For each message, a new `message` object in NVM is created using `pmemobj_alloc` and has following components:

- The type of operation.
- The key and its size.
- The value and its size (except for delete operations).

The message is persisted using `pmemobj_persist` to guarantee crash consistency. Furthermore, `findTargetInternalNodeForKey` determines which internal node is responsible for the key in the message. Afterwards, both `nodeMessageMap` and `messageToNodeMap` are updated using `insertToMessageToNodeMap` and `addKeyToNodeMessageMap`. Finally, `incrementNodeFrequency` updates the access frequency of the node.

Flushing Shared Buffer

When the shared buffer reaches its maximum capacity (`MAX_NVM_MESSAGES`), the `flushMostBufferedNode` function is triggered. Unlike the classic B$^\epsilon$-tree, the unified B$^\epsilon$-tree has the feature that only the auxiliary hash maps need to be updated during flushing. This reduces I/O and computational expenses.

As shown in Figure 5.3, the flush mechanism first selects the most buffered internal node, which is the node with the largest number of buffered messages. This is done using `nodeMessageMap` and `nodeFrequencyMap`. All messages buffered for that node are retrieved from the buffer, and their corresponding pointers in the auxiliary hash tables are updated (the correct child of the internal node is selected). When a leaf node is reached, the messages are inserted into the local small buffer of the leaf node. If the in-place buffer of the corresponding leaf node is full, it will be normalized to apply pending messages and reclaim space, and the leaf node is split if necessary. This ensures that the SSD leaves remain balanced and that buffered operations are efficiently managed. All changes to the shared buffer and auxiliary hash maps are persisted using `pmemobj_persist`.

5.3.6 Tree Initialization and Shutdown

The creation of the tree is done using PMDK. Listing 5.9 shows a simple code snippet demonstrating the initialization process. First, a pool file is created (`shared_buffer_pool.pmem`) or opened if an existing pool is present. This is done using `initializePMEMPool`. When the pool is successfully opened, it assigns `pmemPoolHandle` to the loaded pool and reads the `persistentRoot` from the `PMEMRoot` structure. If the tree is empty, a new leaf node is created and persisted. Finally, `replayBinaryWAL` is called for crash consistency.

The method `initializePMEMPool` first attempts to open an existing PMem pool. To ensure cross-platform compatibility, based on the operating system, the `pmemobj_open` function on linux or `pmemobj_openW` on windows is used. If the pool is successfully found, the existing pool is used for further operations. A simple code snippet is provided in Listing 5.10. If the pool is not found, a new pool is created using `pmemobj_create` (or `pmemobj_createW`). The pool is created with a fixed size of 11 GB. Afterwards, the root object of the pool is initialized. This includes:

- `messageCount`, which represents the messages in shared buffer, is set to zero.
- Message in shared buffer are set to `TOID_NULL`.
- All auxiliary hash maps.

- `nextLeafFileID` is set to zero.

Furthermore, the pool `UUID` is retrieved and stored.

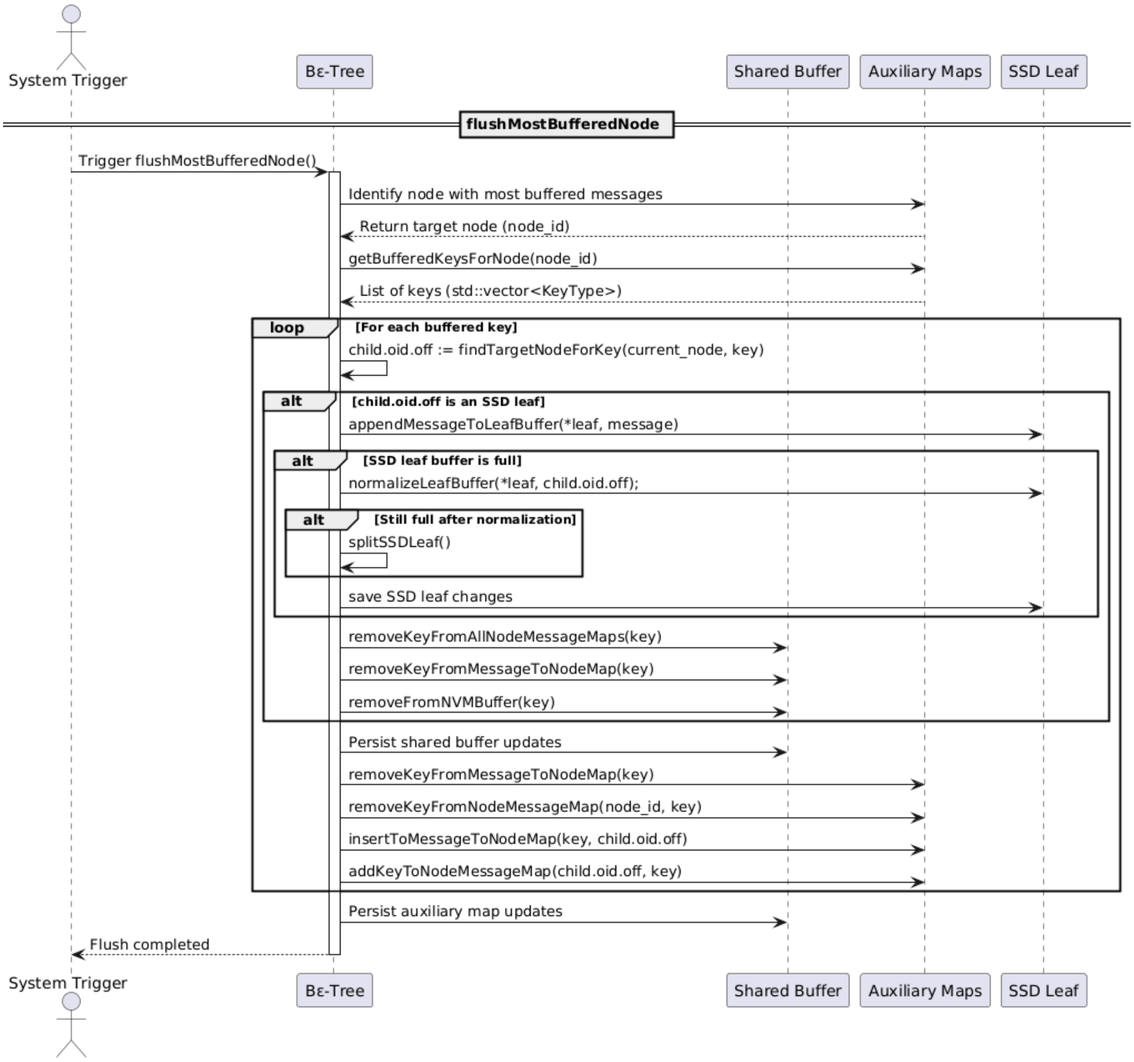


Figure 5.3: Sequence diagram of the `flushMostBufferedNode` operation

For a clean shutdown and future use, a destructor is implemented (see Listing 5.11). First, it checks if any pending operations exist to ensure that buffered changes are persisted in NVM or SSD. Then, the active binary WAL is saved to a timestamped backup file, to support recovery. Furthermore, the main WAL file is truncated to prepare for clean restart scenarios. Finally, the PMem pool is closed using `pmemobj_close` to release the NVM resources and flush any pending changes to disk. Volatile nodes in DRAM are also cleared to prevent memory leak.

Both the constructor and destructor ensure that the tree is persisted and can be recovered, and that all data structures in both volatile and non-volatile memory are properly managed. The recovery strategy is presented in detail in Section 5.4.

Listing 5.9: Tree initialization

```
BEpsilonTree(const std::string& filename, int checkpointFreq = -1)
    {
        if (checkpointFreq > 0)
            this->checkpointFrequency = checkpointFreq;
        const std::string pmemPath = getPlatformPath("shared_buffer_pool.pmem");
        initializePMEMPool(pmemPath);

        pmemRoot = POBJ_ROOT(pmemPoolHandle, PMEMRoot);
        persistentRoot = D_RW(pmemRoot)->persistentRoot;

        if (TOID_IS_NULL(persistentRoot))
        {
            persistentRoot = allocatePersistentNode(true);
            D_RW(pmemRoot)->persistentRoot = persistentRoot;
            pmemobj_persist(pmemPoolHandle, D_RW(pmemRoot), sizeof(PMEMRoot));
        }
        else
        {
            std::cout << "Loaded persistentRoot from PMEM pool.\n";
        }

        replayBinaryWAL();
    }
```

5.3.7 Migration of Hot Nodes to DRAM

As mentioned in Section 4.3, nodes that are frequently accessed are migrated to DRAM. This feature is implemented with the help of `migrateNodeToDRAM`. The `nodeFrequencyMap` tracks the number of accesses for each internal node. When the frequency of a node exceeds a predefined threshold (`HOT_NODE_THRESHOLD`), the node is marked as a hot node and migrated to DRAM. The migration process consists of the following steps:

- Identify hot nodes.
- Copy the keys and children of the corresponding node into a new `VolatileNode` structure in DRAM.
- Transfer all buffered messages that belong to the node from the shared NVM buffer into the local buffer of the migrating node.
- Remove the migrated messages from the shared buffer, `MessageToNodeMap`, and `NodeMessageMap`.

5.3.8 Eviction of Hot Nodes from DRAM

When the maximum number of hot nodes in DRAM is reached (MAX_HOT_NODES), one of the nodes must be evicted to make space for new nodes. The eviction mechanism is handled by `evictVolatileNodeToNVM`. Also here, the node structure needs to be changed. The current eviction strategy follows a first-in first-out (FIFO) policy. The first node inserted into the `volatileNodes` map, which represents the hot nodes in DRAM, is selected for migration.

The method checks whether there are pending operations in the local buffer of the node. When this is the case, the `flushVolatileNodeBuffer` is called to flush these messages to the persistent shared buffer or to SSD leaves. This ensures that no data are lost. The node is then removed from `volatileNodes` and its access frequency is reset to zero. This method does not copy the node back to NVM, as the persistent node is maintained in sync during tree operations which reduce writing overhead and increase the efficiency.

Listing 5.10: PMEMPool initialization

```
    void initializePMEMPool(const std::string& pmemPath)
    {
#ifdef _WIN32
        std::wstring pmemPathW(pmemPath.begin(), pmemPath.end());
        pmemPoolHandle = pmemobj_openW(pmemPathW.c_str(), LAYOUT_NAME);
#else
        pmemPoolHandle = pmemobj_open(pmemPath.c_str(), LAYOUT_NAME);
#endif

        if (!pmemPoolHandle)
        {
#ifdef _WIN32
            pmemPoolHandle = pmemobj_createW(pmemPathW.c_str(), LAYOUT_NAME,
                1ull * 1024 * 1024 * 1024, 0666);
#else
            pmemPoolHandle = pmemobj_create(pmemPath.c_str(), LAYOUT_NAME,
                1ull * 1024 * 1024 * 1024, 0666);
#endif

            if (!pmemPoolHandle)
            {
                perror("pmemobj_create");
                exit(1);
            }

            TOID(PMEMRoot)
                root = POBJ_ROOT(pmemPoolHandle, PMEMRoot);
            auto* ptr = D_RW(root);

            ptr->messageCount = 0;
            for (int i = 0; i < MAX_NVM_MESSAGES; ++i)
            {
                ptr->messages[i] = TOID_NULL(message);
            }
            ptr->nodeFrequencyCount = 0;
            ptr->nodeMessageMapCount = 0;
            ptr->messageToNodeMapCount = 0;
            ptr->nextLeafFileID = 0;

            pmemobj_persist(pmemPoolHandle, ptr, sizeof(PMEMRoot));
        }
        else
        {
            std::cout << "[NVM] Existing PMEM pool opened successfully.\n";
        }
        TOID(PMEMRoot)
            root = POBJ_ROOT(pmemPoolHandle, PMEMRoot);
        PMEMoid rootOID = pmemobj_oid(D_RW(root));
        this->poolUUID = rootOID.pool_uuid_lo;
    }
```

5.3.9 Read Buffer for Frequently Accessed Values

As discussed in Section 4.5, the unified B$^\epsilon$-tree has a fixed size read buffer to make search operation for frequently accessed keys more efficient and faster. The read buffer, which resides in DRAM, is defined as `std::unordered_map<KeyType, ValueType>`. It directly stores key-value pairs in DRAM. The tree also defines a linked list called `lruList` to manage the access order of keys. The cache insertion works as follows:

- The key is added to `lruList`.
- The key-value pair is inserted into `readCache`.
- When the buffer exceeds the predefined maximum capacity (`maxReadCacheSize`), the least recently used key is evicted.

Listing 5.11: Tree Destructor

```
~BEpsilonTree()
{
    if (opCounter > 0)
    {
        checkpoint();
    }
    const std::string binWal = getPlatformPath("nvm_tree_bin.wal");
    const std::string bakName = getPlatformPath("wal_bin_backup")
                                + timestampename() + ".bak";

    std::ifstream src(binWal, std::ios::binary);
    std::ofstream dst(bakName, std::ios::binary);
    if (src && dst)
    {
        dst << src.rdbuf();
    }
    src.close();
    dst.close();

    std::ofstream clear(binWal, std::ios::trunc | std::ios::binary);
    clear.close();

    if (pmemPoolHandle)
    {
        pmemobj_close(pmemPoolHandle);
        pmemPoolHandle = nullptr;
    }
}
```

5.3.10 Point Query

The unified B$^\epsilon$-tree is distributed across DRAM, NVM, and SSD, which means that the search operation needs to check all these layers. Figure 5.4 presents a sequence diagram of the search operation. Furthermore, Algorithm 5.4 provides simple pseudocode on how the read operation works on the different storage types.

When a point query is triggered, the tree first checks if the key exists in the `readCache` in DRAM. If not, it checks the hot nodes in DRAM and their corresponding local buffers. If a delete message for the queried key is found, "a key not found" is returned. If an insert or update message is found, the key and its associated value are returned. If the key is not present in DRAM, the tree proceeds to NVM and checks the centralized shared buffer for messages that have the queried key. Same as before, if a delete message is found, the search ends. If an insert or update message is found, the key and its value are returned. Finally, if the key is still not found, the tree uses the `searchRecursive` method to navigate through internal nodes and reach the appropriate SSD leaf node. The tree checks the small local buffer of the leaf and the leaf itself. If the key is not found, the search operation is aborted, and the tree returns that the key does not exist.

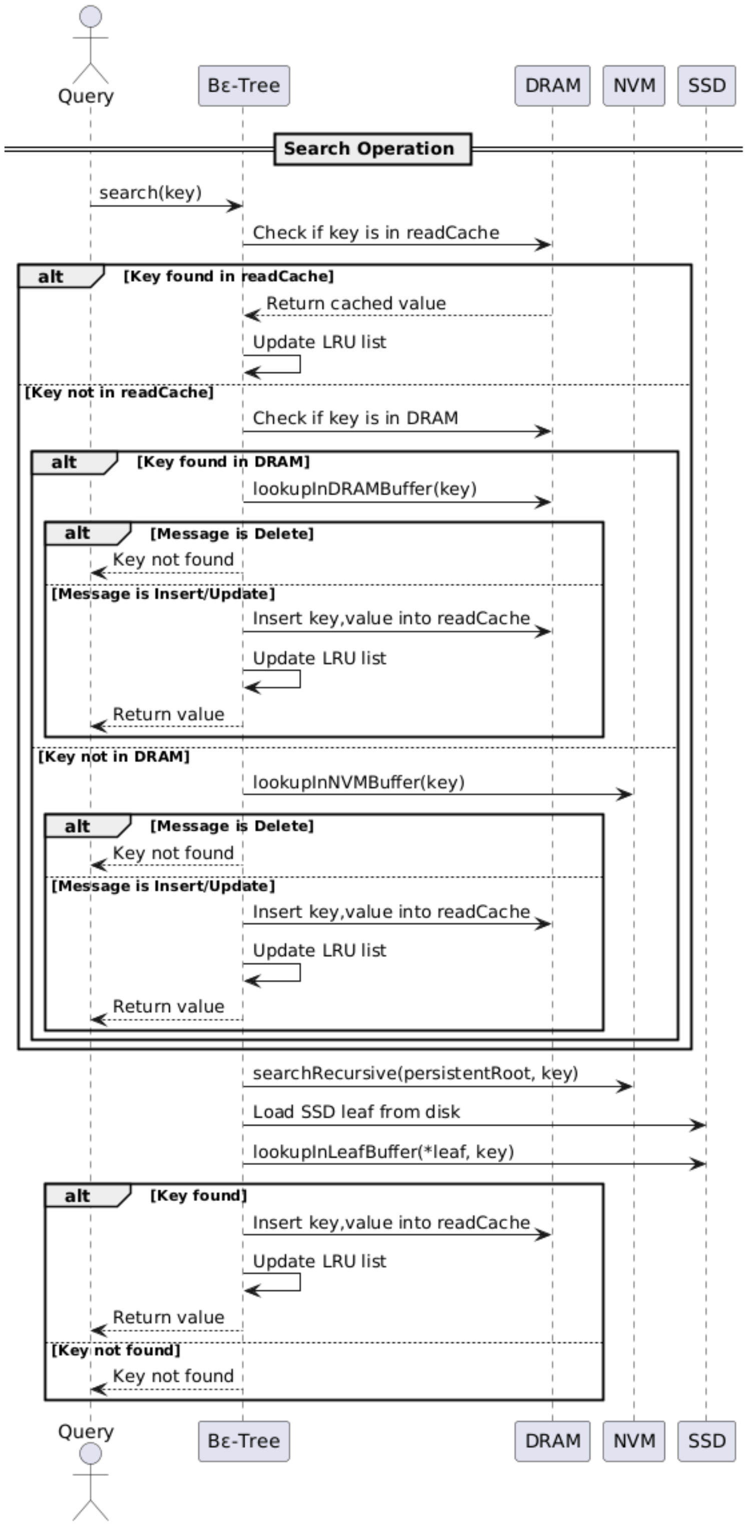


Figure 5.4: Sequence diagram of the search operation in the unified B$^\epsilon$-tree

Algorithm 5.4 Search operation in unified B^ϵ-tree

Require: Key k

Ensure: Return value if key exists, else return "not found"

```
Check DRAM read cache:
if key k is in readCache then
    return cached value
end if
Check DRAM node buffer:
if node containing key k is in DRAM then
    if key k is in DRAM node buffer then
        if buffered operation is Delete then
            return "not found"
        else
            Insert key and value into readCache
            Update LRU list
            return value
        end if
    end if
end if
Check NVM shared buffer:
if key k is in NVM shared buffer then
    if buffered operation is Delete then
        return "not found"
    else
        Insert key and value into readCache
        Update LRU list
        return value
    end if
end if
Traverse tree to SSD leaf:
Perform recursive search starting from persistent root
Load SSD leaf from disk
if key k is found in SSD leaf (including leaf buffer) then
    Insert key and value into readCache
    Update LRU list
    return value
else
    return "not found"
end if
```

DRAM Read Buffer

The tree begins by checking the read buffer in DRAM, which contains the most frequently accessed keys. It performs a hash map lookup to determine whether the queried key is present. If it is found, its associated value is returned, and an LRU policy is used to ensure that only the most recently accessed keys are kept in the buffer. If the key is not found in the buffer, the search continues to the next step.

DRAM Node Buffers

If the key was not found in the DRAM read buffer, the tree, as shown in Figure 5.4 and Algorithm 5.4, moves to check the buffers of hot nodes in DRAM. The method responsible for this step is `lookupInDRAMBuffer`. It returns the operation and the value associated with the searched key if found, otherwise a `std::nullopt`. If an insert or update operation for the queried key is found, the corresponding value is returned. If a delete operation is found, the search terminates and reports that the key does not exist.

NVM Shared Buffer

If the key is still not found in DRAM, the next step in the search operation is to call `lookupInNVMBuffer`. This method is responsible to check if the key is in the centralized shared buffer in NVM. It first checks that the persistent memory pool is available and then iterates through all messages currently stored in the buffer. If a match is found, the method decodes the operation type and extracts the associated value. If the queried key is not found, the method returns a `std::nullopt`, which indicates that the tree should proceed to the next layer.

SSD Leaves

If the key was not found in the previous steps, the tree recursively iterates over the internal nodes in NVM using the `searchRecursive` method. It starts at the root and goes to the appropriate child using binary search. This helps to determine which child pointer to follow. Once the appropriate child is selected, the method checks whether it is an internal or a leaf node. If it is internal, a recursion is done to select the next appropriate child. If the child is an SSD leaf node, the local buffer is checked for recent updates to the key via a linear scan. If a message for the searched key is found, the operation type is retrieved. Like in the step of DRAM node buffer, if an insert or update is found, the associated value is returned. On the other hand, if the message represents a delete operation, the query returns that the key does not exist in the tree. Furthermore, if the key is not found in the leaf buffer, a binary search is performed over the keys in the leaf to continue the search

Complexity

The read operation works on the different storage layers of the unified B$^{\epsilon}$-tree and ensures that frequently accessed keys can be retrieved faster. Table 5.1 presents the different search methods applied by the read operation at the corresponding storage level.

Layer	Lookup Method	Complexity
DRAM Read Buffer	Hash map lookup `readCache.find(key)`	$O(1)$
DRAM Node Buffers	Linear scan of internal node local buffer `lookupInDRAMBuffer`	$O(n)$
NVM Shared Buffer	Linear scan of centralized shared buffer `lookupInNVMBuffer`	$O(n)$
SSD Leaves	buffer scan + binary search in leaf keys	$O(b)+$ $O(\log k)$

Table 5.1: Lookup method and complexity per layer in the unified B$^{\epsilon}$-tree

5.3.11 Range Query

In addition to point queries, the unified B$^{\epsilon}$-tree also supports range queries. In contrast to the classic B$^{\epsilon}$-tree, which only traverses the tree to retrieve key-value pairs within the specified range, a range query in the unified B$^{\epsilon}$-tree must check all relevant storage layers. The operation begins by checking the read buffer in DRAM, which stores frequently accessed key-value pairs. If a relevant key that is within the given range is found, its corresponding value is added to the result. The query checks the nodes in DRAM afterward. The node buffers are scanned to find any buffered messages for the keys. The next step is to check the centralized shared buffer in NVM. Furthermore, the operation traverses the internal nodes in NVM to reach the leaf nodes in SSD. It checks both the small local buffer and the leaf itself to find relevant keys and add them to the result. Finally, all retrieved results are merged, and redundant entries are removed.

5.4 Crash Recovery

The unified B$^{\epsilon}$-tree employs a hybrid crash recovery mechanism. It combines PMem durability with operation log replay. As described previously, the NVM part of the tree, which includes internal nodes (including the root), a centralized shared buffer, and auxiliary hash maps, is stored and updated using the `pmemobj_persist` provided by PMDK. When the PMEM pool is reopened, persistent structures are automatically restored to their last consistent state without requiring additional manual reconstruction.

Furthermore, all write operations are stored in a binary WAL file `nvm_tree_bin.wal`. Each operation is written to the WAL via the method `logOperationBinary` before it is applied to the tree. The method is called in each write operation before it is applied to the tree. First, it opens the WAL file in append mode and serializes the operation type, key, and the value (only for insert and update). Listing 5.12 shows how the method `logOperationBinary` is implemented. This mechanism ensures that all operations are recoverable after a crash. The WAL is written in a binary format to minimize logging overhead and disk space consumption.

Listing 5.12: `logOperationBinary`: writing operations to the binary WAL

```
void logOperationBinary(Operations op, const KeyType& key,
                        const ValueType& value = ValueType{})
{
    if (isReplaying)
        return;
    std::ofstream log(getPlatformPath("nvm_tree_bin.wal"),
        std::ios::binary | std::ios::app);
    if (!log.is_open())
    {
        std::cerr << "Failed to open binary WAL.\n";
        return;
    }

    uint8_t opCode = static_cast<uint8_t>(op);
    log.write(reinterpret_cast<const char*>(&opCode), sizeof(opCode));
    log.write(reinterpret_cast<const char*>(&key), sizeof(KeyType));

    if (op == Operations::Insert || op == Operations::Update)
    {
        log.write(reinterpret_cast<const char*>(&value), sizeof(ValueType));
    }
    log.close();
}
```

The parameter `checkpointFrequency` specifies the number of write operations to be inserted into the binary WAL file before automatically triggering a checkpoint. This mechanism ensures that the log is persistently backed up. When reaching this threshold, the `checkpoint()` method is triggered. It creates a timestamped backup of the auxiliary textual write-ahead log file `nvm_tree_testn.log` to maintain a record of executed operations. Furthermore it truncates the original log file to remove already persisted entries.

Figure 5.5 shows the steps performed during crash recovery in the unified B$^{\epsilon}$-tree. First, the PMEM pool is opened using the `pmemobj_open` defined in `initialize-PMEMPool`. This allows loading the `PMEMRoot` which contains all internal nodes and persistent data structures that reside in NVM. Then, the method `replayBinaryWAL` is called to restore all unflushed operations in the file `nvm_tree_bin.wal`, and re-execute them sequentially based on the type of the operation. During this process, the flag `isReplaying` is set to true to suspend new logging and prevent recursion. This step is repeated until the end of file (EOF) is reached. Furthermore, SSD leaf files are reconstructed by reapplying buffered operations to normalized leaf structures, using the recovered `nextLeafFileID` for allocation.

This hybrid mechanism, which combines PMem consistency and WAL-based durability, ensures that the tree is crash safe and can always be recovered to its last consistent state. The efficiency and performance of this mechanism are discussed later in Section 6.4.

5.5 Summary of Implementation

In this chapter, the implementation of the unified B$^{\epsilon}$-tree proposed by [KWB+24] is presented and discussed. First, a baseline design based on the classic B$^{\epsilon}$-tree [BFCJ+15] structure was implemented. It included buffered internal nodes, message coalescing, and recursive flushing logic. Furthermore, the different features discussed in Chapter 4 were integrated into the baseline to meet the design goals of the unified tree architecture. The main features implemented in this thesis include:

- A centralized shared buffer in PMem for all internal nodes, eliminating the need for individual node buffers in NVM.
- Three different persistent auxiliary hash maps (i.e., `nodeMessageMap`, `messageToNodeMap`, and `nodeFrequencyMap`) to efficiently track message-to-node relationships and node frequencies.
- A dynamic node migration mechanism that promotes frequently accessed internal nodes from NVM to DRAM.
- A read buffer for frequently accessed values.
- A crash recovery strategy that combines PMem consistency with a binary WAL.

Furthermore, the different read and write operations, including `insert`, `update`, `delete`, `point`, and `range queries`, were explained in detail. This implementation was done with the help of PMDK, which efficiently manages NVM, and with C++ as the programming language.

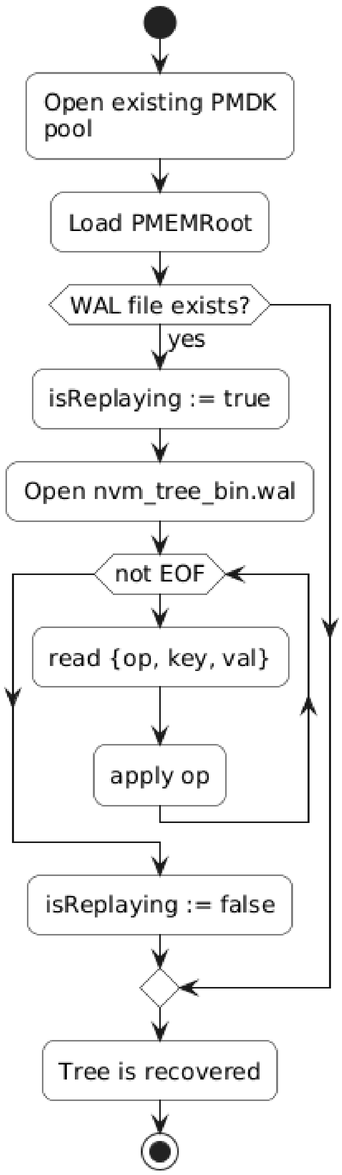


Figure 5.5: Recovery strategy of the unified B^{ϵ}-tree

6. Evaluation

In this chapter, after presenting and implementing the unified B^ϵ-tree, its performance is evaluated using different benchmarks. First, the characteristics of the server on which all evaluations are performed are presented. Furthermore, an analysis is conducted on the impact of different features introduced in Chapter 4, such as the local leaf buffer located in SSD and the centralized shared buffer in NVM, on the performance of the tree. Moreover, the unified B^ϵ-tree is compared against the classic B^ϵ-tree [BFCJ+15] and other persistent memory indexing structures, including FPTree, wB^+-tree, and `FAST_FAIR`. For each evaluation, the strengths and limitations of the proposed prototype are discussed in detail. Finally, the performance of the crash recovery mechanism described in Section 5.4 is evaluated. This helps to evaluate the durability and correctness of the unified B^ϵ-tree when a crash occurs.

6.1 Experimental Setup

All experiments were performed on a server equipped with two Intel® Xeon® Gold 5220R CPUs[4], each running at 2.20 GHz. Table 6.1 presents the characteristics of the environment used. The server provides a total of 96 logical processors, which consist of 24 physical cores per socket and 2 hardware threads per core through simultaneous multithreading (SMT). The server architecture is x86_64 and supports both 32-bit and 64-bit operating modes.

Each socket has 37.75 MiB of shared L3 cache, and each core includes 1 MiB of L2 cache and 46 KiB of L1 cache per core pair (split evenly between instruction and data caches). The server is equipped with 376 GiB of RAM. Each CPU socket has its own integrated memory controller (IMC) with three DDR4 memory channels per controller, resulting in a non-uniform memory access (NUMA) architecture.

Furthermore, the server includes persistent memory (PMem) in the form of Intel® Optane™ DC DIMMs installed alongside DDR4 RAM on the same memory bus. Two

[4] https://www.intel.com/content/www/us/en/products/sku/199354/intel-xeon-gold-5220r-processor-35-75m-cache-2-20-ghz/specifications.html

namespaces were configured: one in `fsdax` mode and one in `devdax` mode, providing a combined capacity of approximately 1 TiB[5].

The operating system installed on the server was Ubuntu 20.04.6 LTS (Focal Fossa)[6], and GCC version 8.4.0[7] was used to compile the implementation described in Chapter 5.

Characteristic	Description
CPU	
Type	2× Intel® Xeon® Gold 5220R
Number of logical processors	96 (24 cores × 2 threads × 2 sockets)
Frequency	2.20 GHz, up to 4.00 GHz
Caches	L1: 32KiB I + 32KiB D per core
	L2: 1MiB per core (48MiB total)
	L3: 35.75MiB per socket (71.5MiB total)
Memory	
DRAM Capacity	376 GiB (2 NUMA nodes)
PMEM Capacity	1 TiB (2 namespaces: 541GB + 532GB)
PMEM Mode	fsdax and devdax
OS	
Release Version	Ubuntu 20.04.6 LTS (Focal Fossa)
Kernel	Linux 5.4.0-216-generic
Compiler	GCC 8.4.0

Table 6.1: Description of experimental setup (following the format [CLF+20])

6.2 Benchmarking Methodology

6.2.1 Impact of Leaf Node Buffer

As discussed in Chapter 4, leaf nodes in the unified B$^\epsilon$-tree include a small local buffer to store minor updates before applying them. This delays the splitting and merging of leaf nodes, which reduces write amplification and improves overall performance, especially on SSDs where random writes are costly.

To evaluate the impact of this feature on the performance of the tree, a series of micro benchmarks were performed, comparing the performance of the tree with (up to 600 bytes) and without leaf buffer enabled. The chosen buffer size does not reflect real-world configurations, but serves only to illustrate the effect of enabling the buffer. The micro benchmarks were performed under identical properties: the maximum number of buffered messages in the NVM shared buffer is ten, internal and leaf nodes can store up to four keys.

To ensure fairness across different benchmarks, different fixed size lists of random 8-byte keys were generated and stored in binary files. The following script was used to generate and shuffle keys:

[5] Reported via `ndctl list -N`
[6] https://releases.ubuntu.com/focal/
[7] https://gcc.gnu.org/onlinedocs/8.4.0/

Listing 6.1: Key generation

```
#include <iostream>
#include <fstream>
#include <vector>
#include <numeric>
#include <algorithm>
#include <random>
#include <string>

int main() {
    std::vector<size_t> datasetSizes = {10000, 50000, 100000, 250000, 500000
                                        , 1000000};

    for (size_t datasetSize : datasetSizes) {
        std::vector<int64_t> keys(datasetSize);
        std::iota(keys.begin(), keys.end(), 1);
        std::shuffle(keys.begin(), keys.end(), std::default_random_engine(42));

        std::string filename = "keys_" + std::to_string(datasetSize) + ".bin";
        std::ofstream out(filename, std::ios::binary);
        out.write(reinterpret_cast<char*>(keys.data()), keys.size()
                  * sizeof(int64_t));
        out.close();
    }
    return 0;
}
```

The benchmark measures the throughput of three write operations: insert, update, and delete for different lists of keys. First, all keys are inserted into an empty tree. The second phase updates each key by multiplying its current value by 100. In the third phase, one-third of the keys are deleted from the tree. Each operation is timed using `std::chrono::high_resolution_clock`.

As shown in Figure 6.1, the tree was more efficient when using the local leaf buffer. For the insert operation, the buffered version of the tree had a lower total execution time across all dataset sizes. For example, the insertion of 500,000 keys without using the local leaf buffer resulted in a total execution time of 1.65 seconds. In contrast, the version with the local leaf buffer had 0.56 seconds, which leads to a decrease in the total execution time for insertion of 66%. The same improvement was observed for the delete operation. The operations were initially saved into the local buffers of the leaf nodes and not directly applied to the tree, which reduces the number of expensive SSD writes and structural modifications (splits and merges). However, the total execution time of the update with buffering was slightly higher. The reason behind this was that the buffer could already contain pending inserts or deletes. When an update arrives and the buffer becomes full, it triggers a flush, which causes these buffered messages to be applied to the leaf nodes. Despite this, the features of the local buffers in leaf nodes proposed in the design reduce overall I/O by coalescing operations and avoiding unnecessary disk writes.

As shown in Figure 6.2, the local buffer of internal nodes has an impact on the amount of SSD traffic generated by the different operations. For example, when inserting one million keys, the buffered version of the tree performed 19% fewer reads and 29% fewer writes compared to the tree without local buffers. This implies that introducing a local buffer for each leaf node improves I/O efficiency in write-heavy workloads.

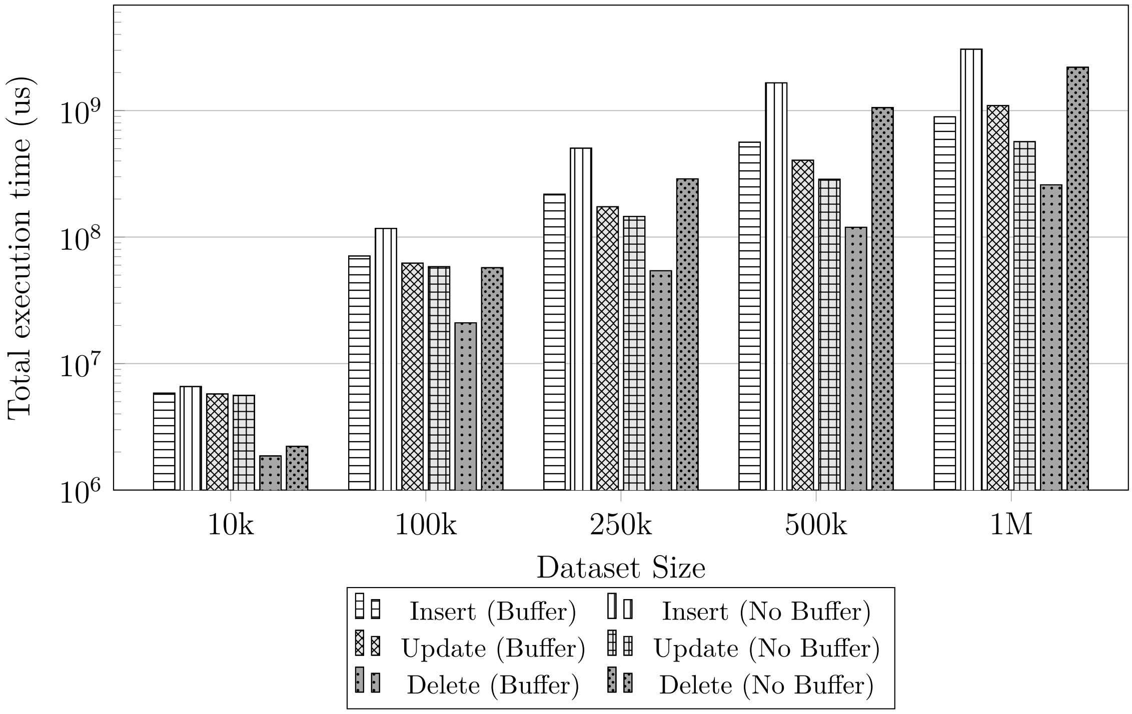


Figure 6.1: Total execution time comparison: with vs. without leaf buffer

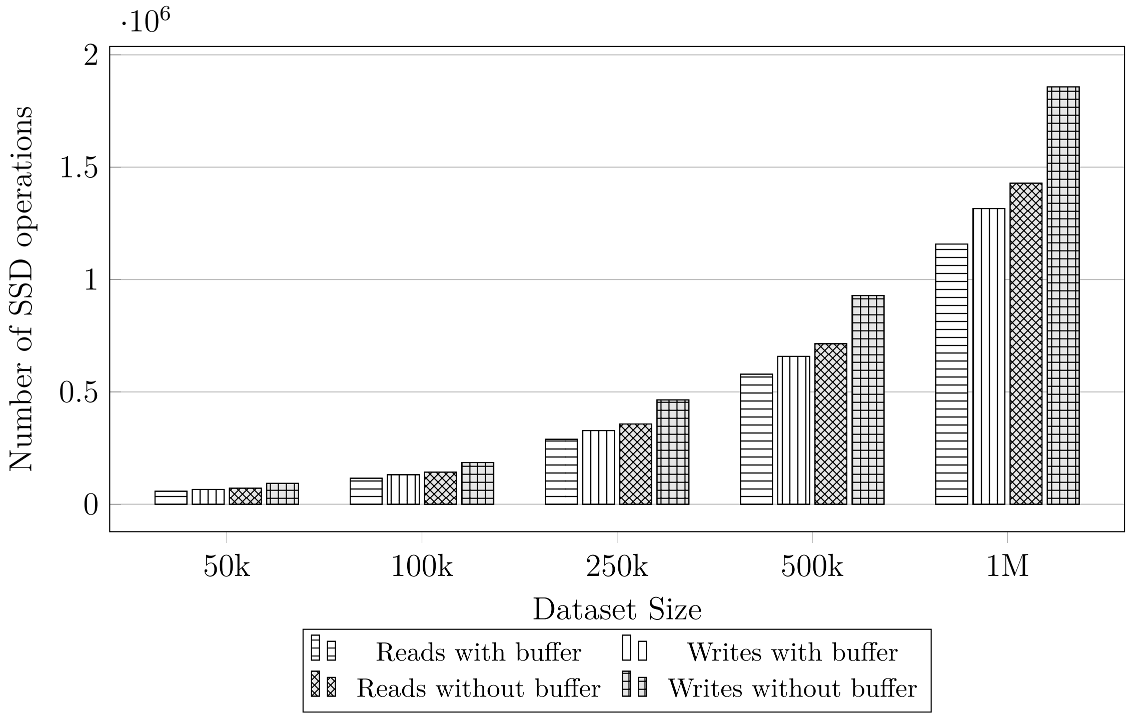
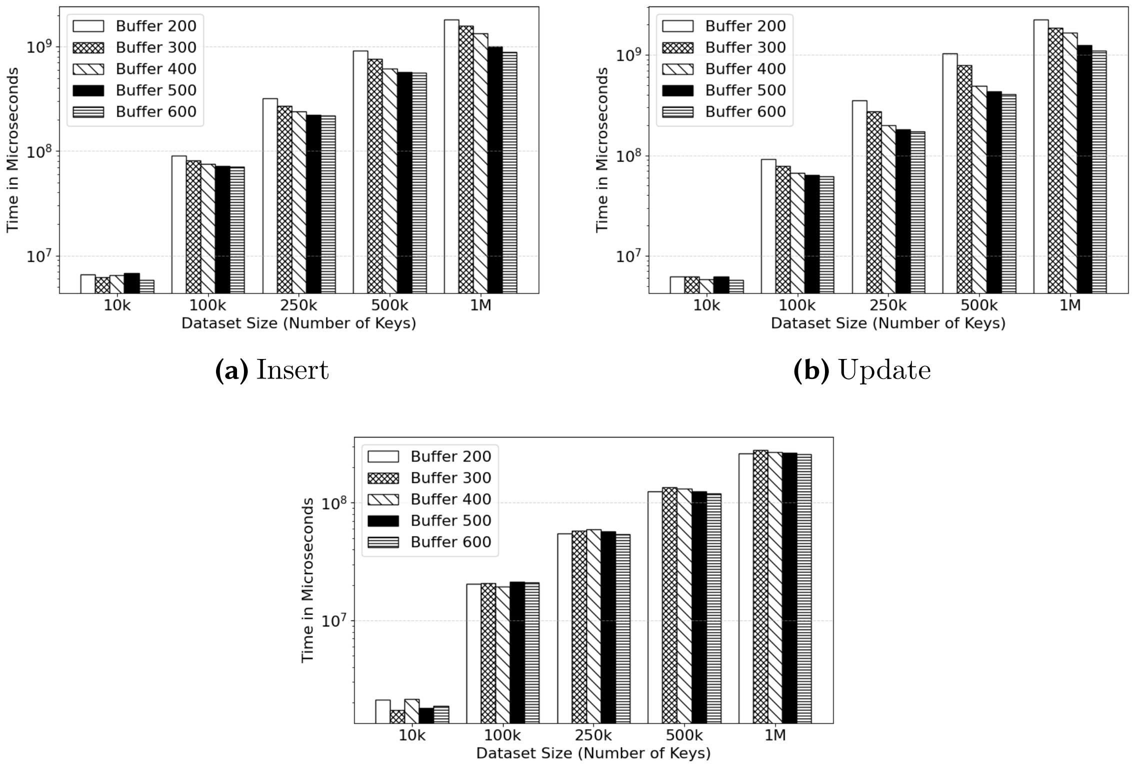


Figure 6.2: SSD I/O during inserts: with vs. without leaf buffer

Furthermore, a similar benchmark was conducted to evaluate the impact of the size of the local leaf buffer on tree performance. The benchmark was performed using various dataset sizes and five different buffer sizes: 200, 300, 400, 500, and 600 bytes.

The results shown in Figure 6.3 indicate that increasing the size of the local buffer in leaf nodes generally reduces total execution time by delaying flushes and structural operations, such as splits and merges. For update operations, for example, the local leaf buffer size has a significant impact on performance. For small datasets (e.g., 10,000 keys), the buffer size has a minimal effect. However, for larger datasets, larger buffer sizes ($\geq$ 400 bytes) reduce update total execution time, as the buffer can store more operations before flushing them to the corresponding leaf node. However, for this tree configuration, a local buffer size of 400 bytes is the most efficient across all operations, offering the best balance across insert, update, and delete workloads. These findings confirm that a sufficiently sized local buffer significantly improves write-heavy performance in large workloads.



(a) Insert

(b) Update

(c) Delete

Figure 6.3: Impact of leaf buffer size on total execution time of different write operations

6.2.2 Impact of Shared NVM Buffer Size

The unified B$^\epsilon$-tree introduces an NVM centralized shared buffer used by internal nodes. This buffer temporarily holds messages before they are flushed to their respective target nodes. As discussed in Section 4.4, this buffer is accompanied by auxiliary hash tables to allow efficient flushing and to make the tree part in NVM consistent. In this subsection, the impact of the buffer size on the performance of the tree is evaluated by measuring the total execution time of insert, delete, update, and search operations. The tree has the following properties: each node can have a maximum of 64 keys, and the leaf node buffer size is 2,000 bytes.

First, 100,000 random 8-byte keys are inserted as a warm-up phase to populate the tree. This is followed by 100,000 search operations to simulate a read-heavy workload. Then, 100,000 additional inserts and updates are performed. Finally, 100,000 deletes are performed. In each run, the size of the shared buffer is varied.

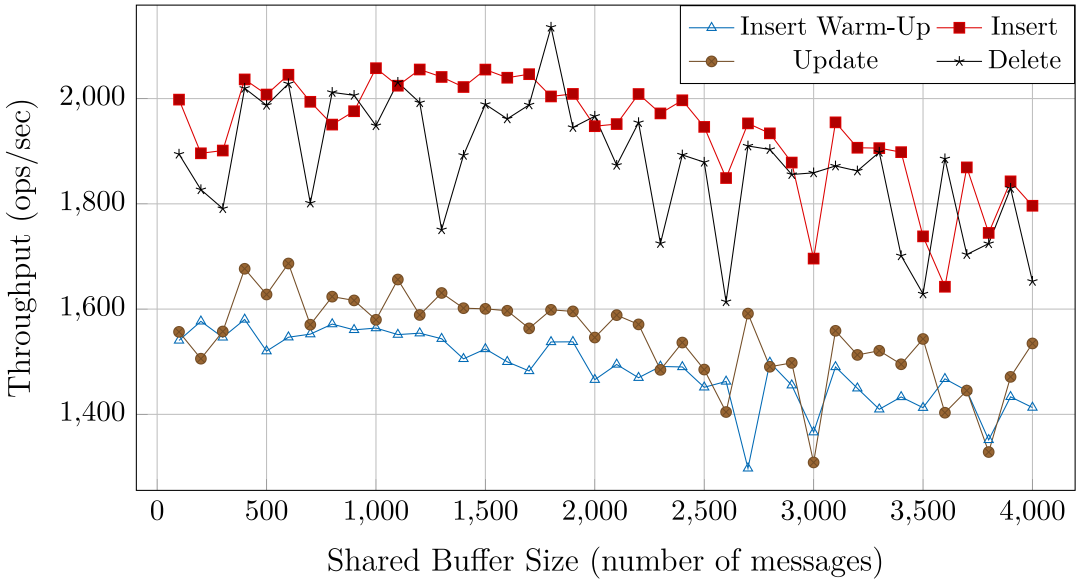


Figure 6.4: Write operations throughput vs shared buffer size

First, the impact of the buffer size is analyzed for different write operations. As shown in Figure 6.4, the size of the shared buffer affects the performance of the different operations. For the warm-up phase, the size of the buffer in this case slightly affects the throughput. As the buffer size increases, the throughput decreases and increases slightly, but it stays in the range (1,400-1,580) ops/sec. This is because the warm-up phase involves sequential inserts into an initially empty tree, where performance is dominated by raw write bandwidth. On the other hand, the insert operation is affected by the size of the shared buffer. Increasing the buffer size from 100 to 1,700 generally improves the throughput. This is because the buffer decreases the frequency of flushes to leaf nodes. However, beyond this range, the throughput decreases significantly, which introduces a cost of managing a larger buffer. For the update operation, a pattern similar to the insert behavior is observed. The optimal range in this case was between 400 and 1,100. The delete operation also follows the pattern of the insert. However, the range of buffer sizes that provided the most efficient throughput was between 400 and 1800. In conclusion, Figure 6.4 shows that larger buffers initially improve write throughput, but exceeding a certain size results in less efficient throughput.

Furthermore, the impact of the size of the shared buffer on the search operation throughput is shown in Figure 6.5. In this case, the throughput is significantly affected by the buffer size. Increasing the buffer size reduces the performance of the search operation. Throughput drops from over 100,000 ops/sec to around 30,000 as

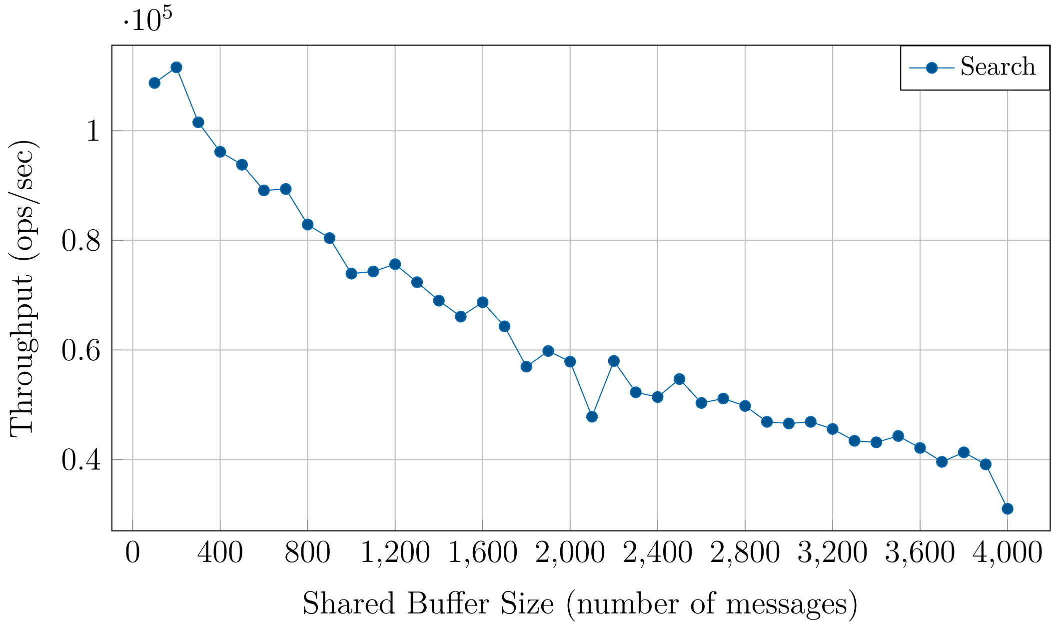


Figure 6.5: Impact of the size of the shared buffer on the throughput of the search operation

the buffer grows. As discussed in Section 5.3.10, the search operation uses a linear search in the centralized shared buffer in NVM. This implies that for larger sizes, the search requires more time to iterate over all entries in the buffer, which reduces the read amplification of the tree.

6.2.3 Performance Comparison

To evaluate the performance of the unified B$^\epsilon$-tree, a comparison was conducted against the classic B$^\epsilon$-tree and different persistent indexing structures. This includes the FPTree [OLN$^+$16] presented in Section 3.5, `FAST_FAIR` [HKWN18] (see Section 3.6), and wB$^+$-tree [CJ15], which is a B$^+$-tree optimized for NVM and was presented in Section 3.3. The benchmark has the following steps:

- Warm up an empty tree by inserting a given set of fixed size n 8-byte key-value pairs.
- Search for n random keys.
- Insert an additional n key-value pairs.
- Update n randomly chosen keys.
- Perform n random deletions.

The different datasets were generated with Listing 6.1. Performance was measured in terms of the total execution time (in microseconds) for each operation.

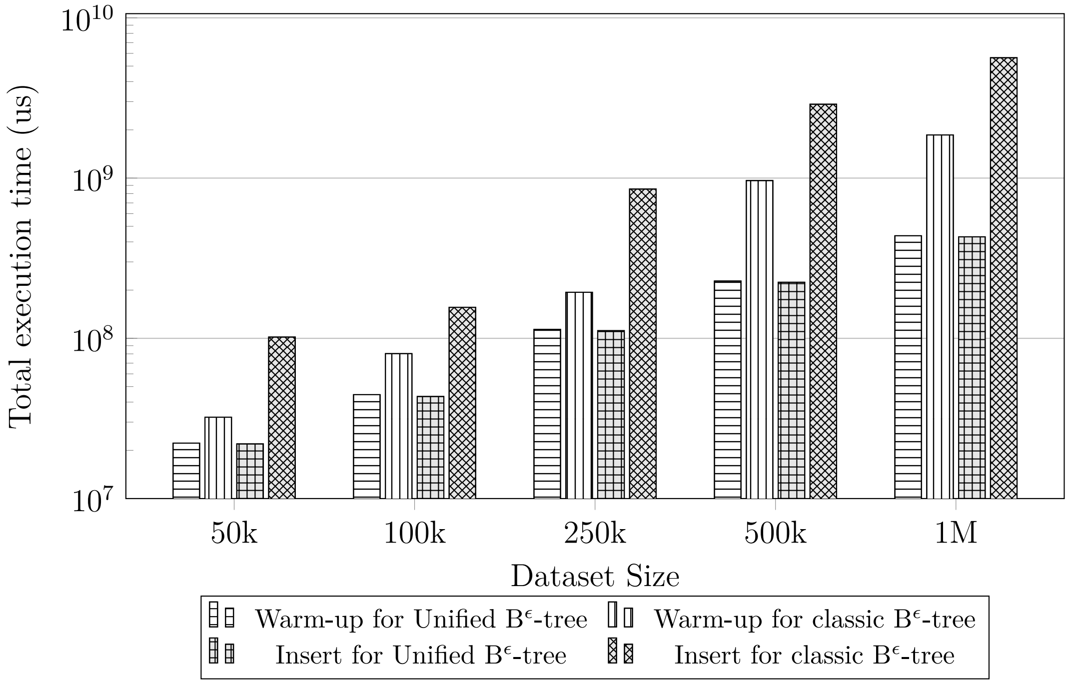


Figure 6.6: Total execution time of warm-up and insert phases for unified vs. classic B$^\epsilon$-trees

6.2.3.1 Insert Execution Time Comparison Between Unified and Classic B$^\epsilon$-Tree

After evaluating the impact of the key features introduced in the unified tree, its performance is compared with that of the classic B$^\epsilon$-tree. Unlike the classic implementation, the unified B$^\epsilon$-tree integrates several key enhancements discussed in Chapter 4. It has a local buffer at each leaf node, a centralized shared buffer, auxiliary hash maps in NVM, and other features.

The same benchmark strategy described previously is used to compare the performance of the unified and the classic B$^\epsilon$-tree. As the classic variant uses DRAM and to ensure fairness, a delay variable is added for the different operations to simulate the additional total execution time cost of NVM. For this benchmark, the total read and write execution time of PMem is set to 500 ns.

In both warm-up and insert benchmarks (see Figure 6.6), the proposed unified B$^\epsilon$-tree had a lower total execution time than the classic B$^\epsilon$-tree across various dataset sizes. For instance, for a dataset of size 500,000, the warm-up performance of the unified tree was more than four times faster than that of the classic variant, and the total execution time of the insert operation was 12 times lower. This is primarily due to the buffering mechanism, which delays structural changes and applies the different insert operations as a block change. Moreover, the total execution time difference between the warm-up and insert phases in the unified tree is small. In contrast, the classic B$^\epsilon$-tree shows a significant increase in total execution time during the insert phase following warm-up, which is caused by structural operations such as node splitting and flushing the buffer.

6.2.3.2 Unified B^{ϵ}-Tree Against NVM-Optimized Persistent B^{+}-Tree Structures

After evaluating the performance of the unified B^{ϵ}-tree against the classic variant, a comparison with `FAST_FAIR` [HKWN18], wB^{+}-tree [CJ15], and FPTree [OLN+16] is conducted. These data structures are optimized B^{+}-trees for byte-addressable PMem. Various open-source implementations of the different trees, publicly available on GitHub [8] [9] [10], are used and adapted for this evaluation to ensure compatibility with the benchmarking framework.

The `FAST_FAIR` implementation provides support for insert, search, and delete operations, but it does not implement an update operation. Therefore, an update method was added. The operation checks whether the given key already exists in the tree and performs an insertion using the same key with the new value to replace the old one.

Figure 6.7, Figure 6.8, Figure 6.9, Figure 6.10, and Figure 6.11 present the runtime performance [11] for dataset sizes ranging from 10,000 to one million key-value pairs.

In both warm-up and insert phases (see Figure 6.7 and Figure 6.8), the unified B^{ϵ}-tree had lower total execution time than `FAST_FAIR` and wB^{+}-tree across all dataset sizes. For example, for one million keys, the total execution time of insert was approximately 11.59% faster than wB^{+} and more than 30% faster than `FAST_FAIR`. This confirms that delaying costly structural modifications, such as splits that occur because of insert operations, has a positive impact on the performance of the tree. Furthermore, both wB^{+}-tree and `FAST_FAIR` place all nodes in NVM, which implies a higher total execution time for insert due to structural modifications in PMem. However, FPTree had a lower insert total execution time than the unified B^{ϵ}-tree, as only leaf nodes are in NVM. For larger datasets (e.g., one million keys), the difference of total execution time becomes less, indicating that the unified buffering model scales efficiently with increasing data volume.

In the search benchmark shown in Figure 6.9, the unified B^{ϵ}-tree had higher query total execution time than the compared trees, for small datasets. For datasets of 500,000 and one million keys, it had a lower total execution time than wB^{+}-tree. This is caused by the linear search when traversing the buffer of hot nodes in DRAM and the linear scan of the centralized shared buffer in NVM. This suggests using persistent hash tables for the shared buffer to enable constant-time ($O(1)$) key lookup, which reduces the total execution time of the read operation. This will be discussed in Chapter 8. Additionally, the benchmark strategy used did not take advantage of the read buffer in DRAM, since each key was queried only once. Therefore, in Section 6.3, a macro benchmark is performed with a zipfian distribution [Zip16] to simulate realistic access patterns and evaluate the effectiveness of the read buffer.

The update phase (see Figure 6.10) shows that the unified B^{ϵ}-tree had a lower total execution time than `FAST_FAIR`. However, its update total execution time was higher

[8] https://github.com/DICL/FAST_FAIR
[9] https://github.com/sfu-dis/FPTree
[10] https://github.com/thustorage/nvm-datastructure/tree/master/singleThread/wb+tree
[11] On a logarithmic scale

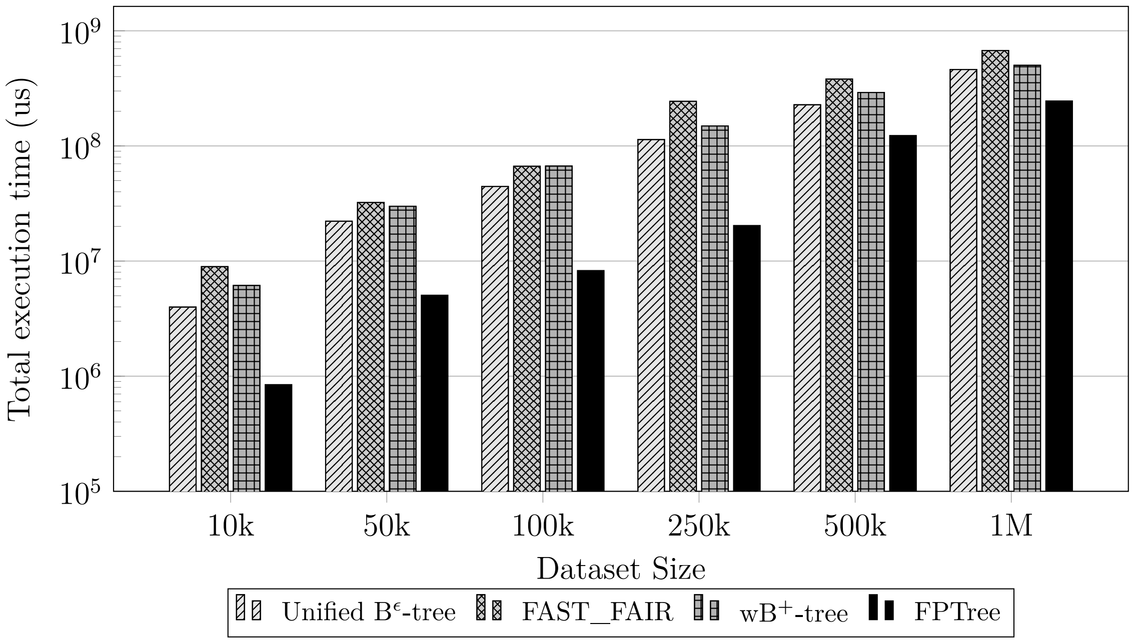


Figure 6.7: Warm-up total execution time of different tree structures

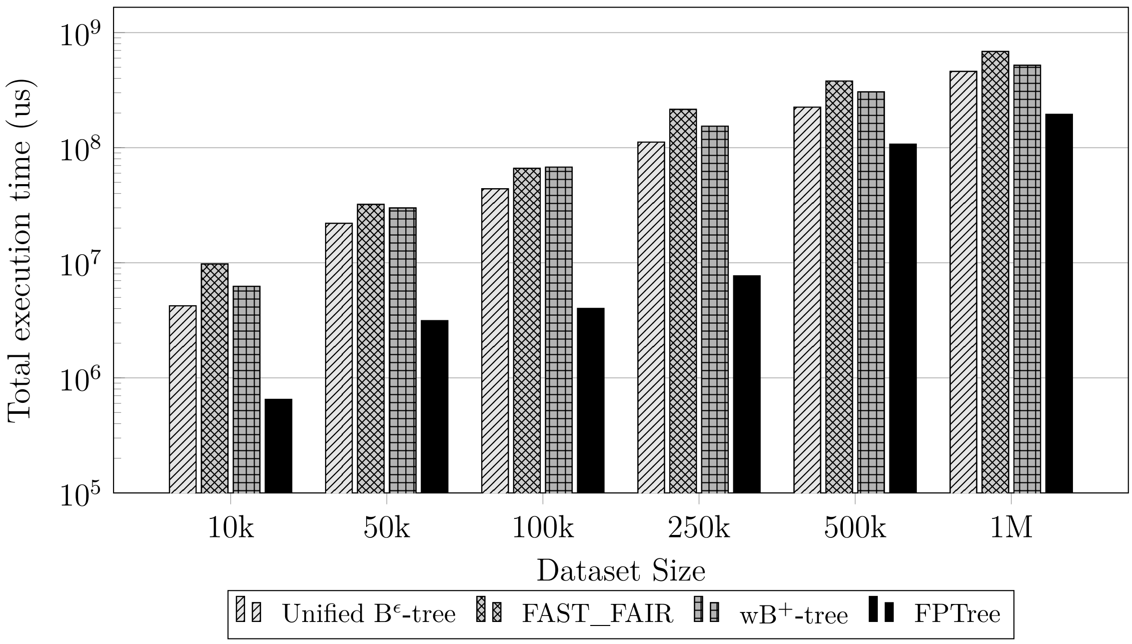


Figure 6.8: Insert total execution time of n new key-value pairs into each tree after warm-up

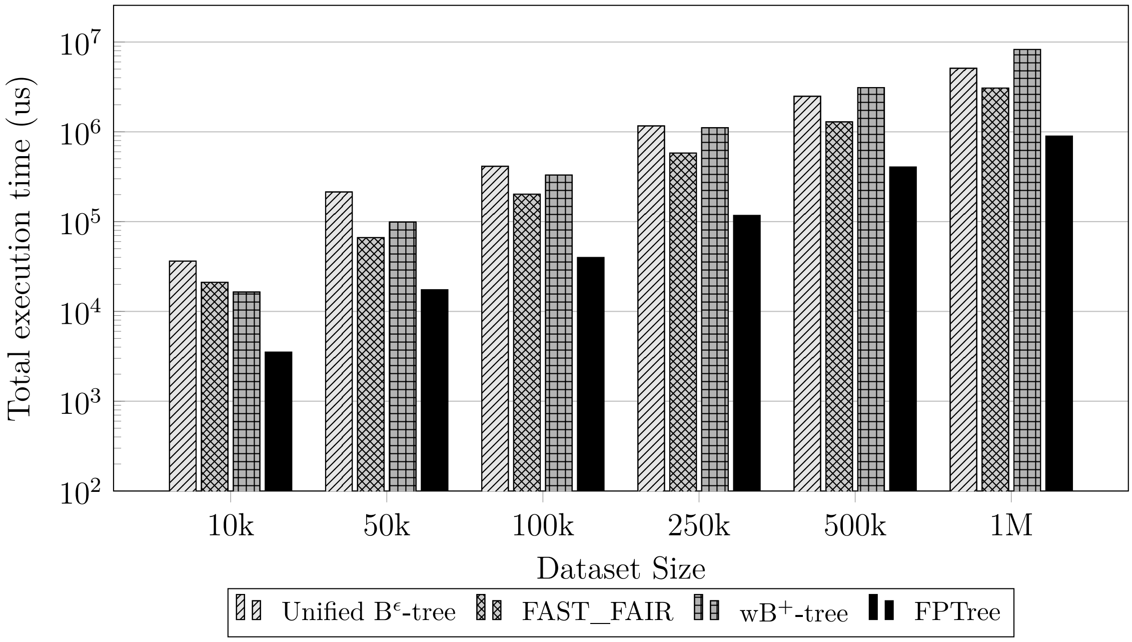


Figure 6.9: Search performance comparison for randomly queried keys across various dataset sizes

than the other compared trees. This behavior can be caused by the local buffer of leaf nodes. This buffer can contain pending inserts. When new updates come and the buffer becomes full, these operations are applied to the tree, which can cause structural changes, such as node splits. In contrast, both FPTree and wB$^+$-tree apply updates more directly at the leaf level with minimal buffering overhead, which can explain their superior performance in this phase.

Figure 6.11 presents the results of the total execution time for delete operations for the different trees. The unified B$^\epsilon$-tree has lower delete total execution time than `FAST_FAIR` and FPTree for most dataset sizes, particularly as the dataset size increases. Both the centralized shared buffer in NVM and the local buffer of leaf nodes in SSD delay the propagation of deletions to the tree structure, which requires costly structural changes, such as deleting or merging nodes. However, the wB$^+$-tree demonstrates a lower total execution time than the unified B$^\epsilon$-tree, as it only marks the targeted key as deleted by updating the bitmap or slot array without moving other keys, since leaf nodes are unsorted. This feature reduces the delete total execution time for the wB$^+$-tree.

To conclude, the evaluation of the performance of the unified B$^\epsilon$-tree against other optimized NVM data structures highlights both its advantages and limitations. For large insert operations, the tree consistently outperforms `FAST_FAIR` and wB$^+$-tree. This is due to its write-optimized architecture and the use of a centralized shared buffer in NVM, which reduces the frequency of structural modifications during large-scale insertions. However, the results also reveal some limitations. For read operations, the unified B$^\epsilon$-tree had higher total execution time for small datasets, which is caused by linear scans over buffers in both DRAM and NVM. This suggests potential optimizations that can be applied to the tree, which will be discussed in Chapter 8.

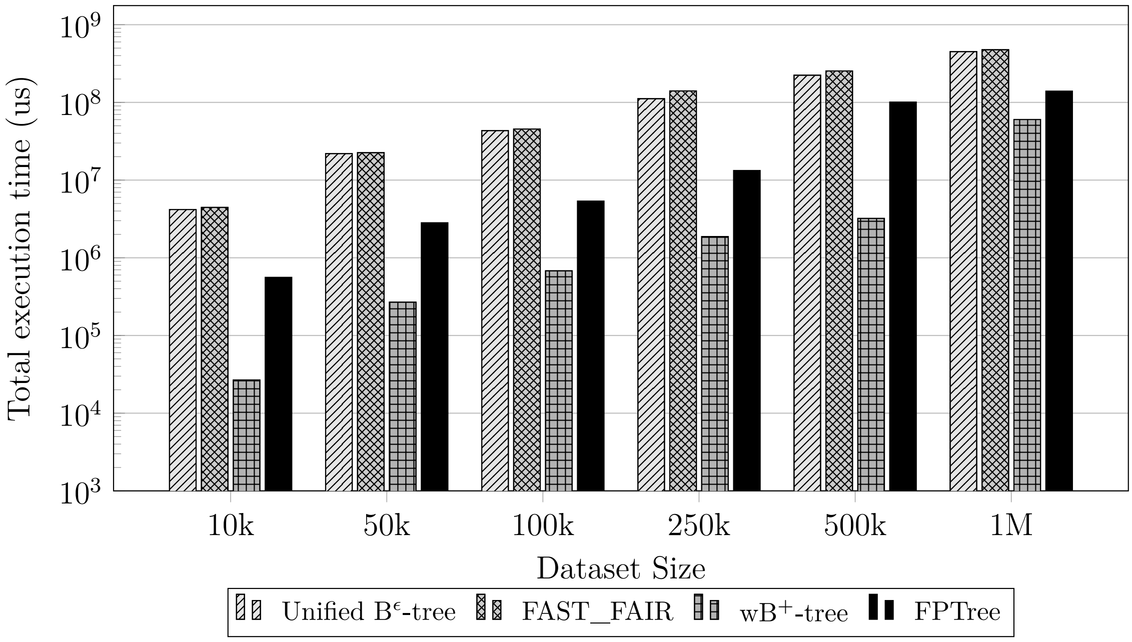


Figure 6.10: Update performance based on randomly selected key

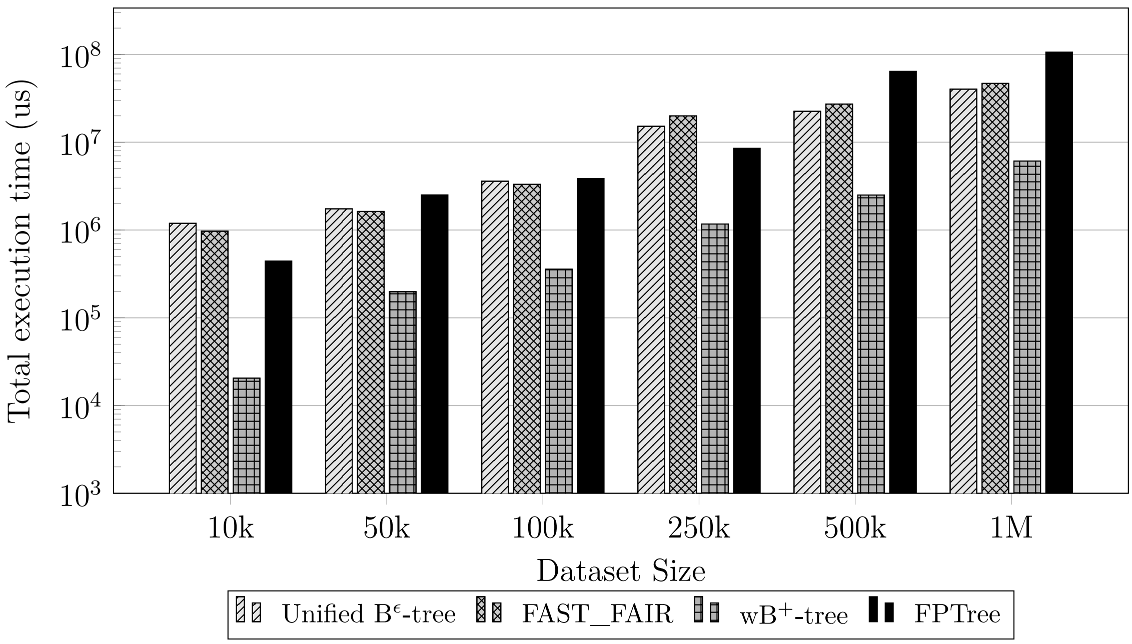


Figure 6.11: Delete total execution time comparison on randomly selected keys

6.3 Macro Benchmark with YCSB Workloads

A further evaluation methodology used in this thesis to measure the performance of the unified B^{ϵ}-tree against the different NVM-optimized data structures is a yahoo! cloud serving benchmarking (YCSB)[12] approach, a popular benchmarking methodology [CST+10] that uses different read/write ratios and skew/uniform key access. In addition to the previously benchmarked NVM-optimized B^{+}-tree, the NoveLSM [KBG+18], presented in Section 3.1, is also included in this macro benchmark evaluation. For this, the open-source implementation of NoveLSM[13] was adapted to integrate with the benchmarking framework. This enables the evaluation of the performance of the unified B^{ϵ}-tree in comparison to both NVM-optimized B^{+}-trees and LSM-trees across different workload characteristics.

First, the YCSB benchmark populates an empty tree with five million records (8-byte key, 8-byte value). This load phase is followed by a run phase of ten million operations, where the read/write ratio is determined by the selected workload. Table 6.2 presents the different workloads used in this evaluation. To model realistic access patterns, a zipfian distribution [Zip16] with 99% skewness was used, which helps to assess the impact of the read buffer in DRAM on the read performance of the unified B^{ϵ}-tree.

Workload	Characteristic	Read	Update	Insert	Scan	RMW
YCSB-A	Update-heavy	50%	50%	0%	0%	0%
YCSB-B	Read-mostly	95%	5%	0%	0%	0%
YCSB-C	Read-only	100%	0%	0%	0%	0%
YCSB-D	Read-latest	95%	0%	5%	0%	0%
YCSB-E	Short-ranges	0%	0%	5%	95%	0%
YCSB-F	Read-modify-write	50%	0%	0%	0%	50%

Table 6.2: Used YCSB workloads in the evaluation

Figure 6.12[14] presents the results of the YCSB macro benchmark. During the load phase, the unified B^{ϵ}-tree had a higher throughput than both `FAST_FAIR` and wB^{+}-tree, while achieving a throughput that is approximately 14% lower than `FPTree`, which highlights the positive impact of the centralized shared buffer in NVM, as the delay of costly structural modifications, such as splits, reduces the overhead of the load phase. However, compared to NoveLSM, the unified B^{ϵ}-tree had a throughput that was about 28% lower. This is because NoveLSM benefits from its log-structured design, which favors sequential writes during bulk loading.

In YCSB-A, the update-heavy workload with an even 50% read / 50% update ratio, the unified B^{ϵ}-tree had 21,379 ops/sec. This is higher than the wB^{+}-tree (20,264 ops/sec) and slightly lower than `FAST_FAIR`, FPTree, and NoveLSM. The lower throughput is caused by the update operation. As discussed in subsubsection 6.2.3.2, the local buffers of leaf nodes can contain pending insert operations from the load phase. Once the buffer is full, it is flushed, causing structural changes.

[12] https://github.com/brianfrankcooper/YCSB
[13] https://github.com/sudarsunkannan/lsm_nvm
[14] On a logarithmic scale

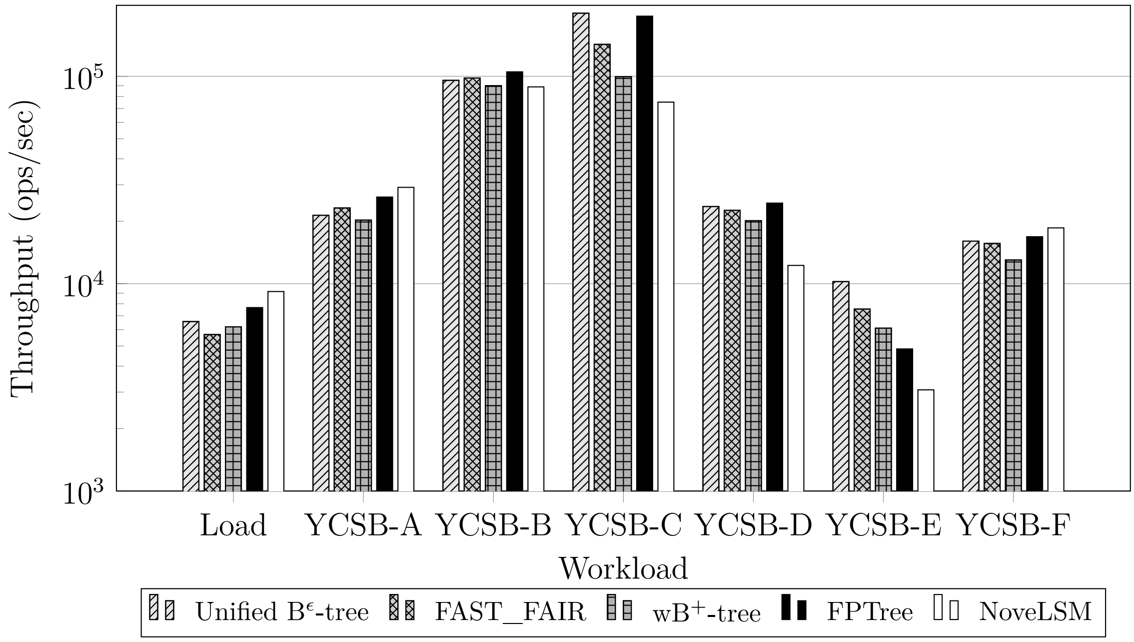


Figure 6.12: Throughput on the YCSB workloads

For YCSB-B, which is read-heavy with 95% reads and 5% updates, the unified B^ϵ-tree outperforms both wB^+-tree and NoveLSM. Compared to YCSB-A, the performance gap to `FAST_FAIR` and FPTree is reduced because the lower number of update operations reduces buffer flushes and structural modifications. As a result, the read buffer in DRAM becomes more effective.

In contrast to previous workloads, the unified B^ϵ-tree had the highest throughput among the other NVM-optimized B^+-trees and LSM-tree in YCSB-C, which presents a read-only workload (100% reads). The proposed tree had a throughput of 201,642 ops/sec, which is 41% higher than `FAST_FAIR`, more than 100% higher than the wB^+-tree, and 168% higher than NoveLSM. Furthermore, it outperforms FPTree by 3%. As discussed previously, the zipfian distribution with 99% skewness allows the presence of more hot keys. This increases the effectiveness of the read buffer in DRAM, which stores frequently accessed keys and thereby saves the cost of traversing the tree in NVM or SSD.

Moreover, in YCSB-D, which is the read-latest workload with 95% reads and 5% inserts, the unified B^ϵ-tree achieved 23,547 ops/sec. This is higher than `FAST_FAIR` (22,598 ops/sec), wB^+-tree (20,135 ops/sec), and NoveLSM (12,247 ops/sec). It nearly matches the throughput of `FPTree` (24,475 ops/sec).

Furthermore, in YCSB-E, which is characterized by short-range queries (95% scans and 5% inserts), the unified B^ϵ-tree outperformed other trees. It had a throughput, which is 36% higher than `FAST_FAIR`, 68% higher than wB^+-tree, 111% higher than FPTree, and more than 230% higher than NoveLSM.

Finally, in YCSB-F, each operation consists of reading a key, modifying its value, and then writing the updated value back to the tree. The proposed design had a higher throughput than `FAST_FAIR` and wB^+-tree.

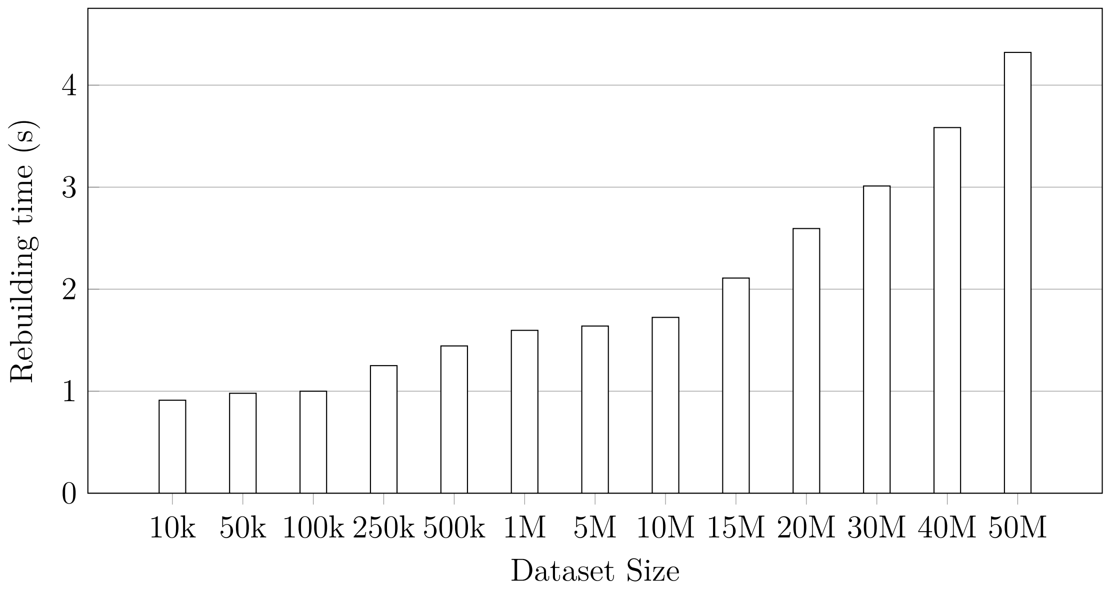


Figure 6.13: Recovery performance of the unified B$^\epsilon$-tree with various dataset sizes

6.4 Crash Recovery Performance Evaluation

In the previous chapter (Chapter 5), Section 5.4 described the crash mechanism implemented by the unified B$^\epsilon$-tree when a crash occurs. The goal is to measure how efficiently the tree can restore a consistent and usable state after a crash.

This hybrid mechanism leverages WAL and NVM consistency to ensure the correctness and durability of the tree. A benchmark scenario, which is repeated for various dataset sizes to evaluate scalability and recovery overhead, was designed with the following steps:

1. Insert a given set of 8-byte fixed size key-value pairs n.
2. Simulate a crash by terminating the process using `std::exit(1)`.
3. Re-initialize the tree using the recovery strategy.
4. Measure the time required to restore the tree and the number of recovered operations.

Figure 6.13 presents the time (in seconds) required for rebuilding the unified B$^\epsilon$-tree for various dataset sizes after a crash occurs. The recovery process required for all tested sizes is less than 4.4 seconds. For instance, for a dataset of one million entries, the recovery phase took approximately 0.96 seconds to rebuild the tree, while datasets of five million and thirty million keys took around 1.04 seconds and 3.01 seconds, respectively.

This shows that the hybrid crash recovery mechanism described in Section 5.4, which combines persistent memory for internal metadata and binary WAL based replay for operations, scales efficiently and provides both reliability and a low total recovery execution time.

6.5 Summary

This chapter evaluates the performance and crash recovery characteristics of the unified B$^\epsilon$-tree. First, the impact of the different features introduced in Chapter 4 on the performance of the unified B$^\epsilon$-tree was analyzed. The experiments showed that the leaf node buffer significantly reduced the total execution time of different write operations , especially for larger datasets, by reducing the frequency of costly structural operations. It also improves I/O efficiency in write-heavy workloads. Moreover, the size of the centralized shared buffer in NVM had an impact on the total execution time of different operations performed. However, for each tree configuration, there was an optimal buffer size that offered the best balance between read and write performance.

Second, the unified B$^\epsilon$-tree was compared against the classic B$^\epsilon$-tree and other NVM-optimized B$^+$-tree variants such as `FAST_FAIR`, wB$^+$-tree, and FPTree. The benchmarks showed that the unified design achieved lower warm-up and insertion latencies than the classic B$^\epsilon$-tree. In large-scale insert workloads, it also outperformed both `FAST_FAIR` and wB$^+$-tree. However, for read operations on small datasets, it had a higher total execution time compared to the other structures. This is caused by the linear scans over buffers in both DRAM and NVM.

Furthermore, a macro benchmark evaluation with different YCSB workloads was conducted, including comparisons with NoveLSM in addition to the NVM optimized B$^+$-tree variants. The results show that the unified B$^\epsilon$-tree had higher throughput than `FAST_FAIR` and wB$^+$-tree in most workloads and remained within 14% of FPTree during the load phase. It achieved the highest throughput in the read-only workload and (YCSB-C). Moreover, the proposed design delivered the best performance in the short-ranges workload (YCSB-E). Compared to NoveLSM, the unified B$^\epsilon$-tree outperformed it in read-dominated workloads.

Finally, the crash recovery mechanism, described in Section 5.4, was evaluated. The experiment showed that the unified B$^\epsilon$-tree is capable of quickly restoring a consistent state, even when handling large datasets.

7. Conclusion

In this thesis, the design of a unified B$^{\epsilon}$-tree, originally proposed by Karim et al. [KWB+24], was presented, implemented, and evaluated. The design is optimized for heterogeneous memory hierarchies, including DRAM, non-volatile memory (NVM), and SSD storage. As the amount of data continues to grow each year, DRAM alone, which is volatile, cannot handle this amount of data and will not be able to meet the needs of modern data-rich applications. To overcome these limitations, NVM, which is a modern memory type, offers near DRAM speed and persistent storage. However, modern computing systems employ heterogeneous storage hierarchies. To address the limitations of traditional indexing structures, which are not well suited to exploit the unique performance and persistence characteristics of modern memory technologies, a unified B$^{\epsilon}$-tree was implemented. Furthermore, despite the advantages of B$^{\epsilon}$-tree in comparison with B-trees or LSM-trees, only a limited number of researches have optimized the B$^{\epsilon}$-tree for NVM.

The unified B$^{\epsilon}$-tree implemented in this thesis introduces several features that extend beyond those of the classic B$^{\epsilon}$-tree. Instead of the local buffer that each internal node has, the proposed design introduces a centralized shared persistent buffer in NVM, which temporarily stores incoming write operations. This buffer is accessible by all internal nodes located in NVM and is accompanied by three auxiliary hash tables to enable efficient tracking of both buffered messages and the frequency of node accesses. This feature increases flush efficiency and minimizes write amplification, as only the auxiliary hash tables need to be updated when flushing the shared buffer. Moreover, in the unified B$^{\epsilon}$-tree, internal nodes are stored and accessed in NVM. The design enables direct reads and in-place updates on NVM without copying internal nodes to DRAM, which reduces memory overhead. Furthermore, nodes that are frequently accessed are migrated to DRAM by creating working copies of them there. This feature requires transforming the layout of the migrated node to maintain tree consistency. A further feature that the unified B$^{\epsilon}$-tree introduces is a read buffer in DRAM, which increases the performance of read operations for frequently accessed values. The buffer stores recently accessed nodes to prevent redundant reads from other storage layers. On the other hand, leaf nodes are stored in SSD. Each leaf node includes a small buffer, where messages are stored before being applied to the node. This feature reduces write amplification.

To ensure tree consistency after a crash, a recovery mechanism was also implemented. It provides durability through a WAL and structural recovery of SSD leaf nodes and NVM-resident metadata. All operations are logged in a binary file. After a crash occurs, the recovery mechanism replays these operations in order to restore the tree to a consistent state.

The unified B^ϵ-tree was implemented in C++ using PMDK to manage non-volatile memory structures and ensure crash consistency. This combination enables fine-grained control over memory allocation, persistence, and recovery logic across heterogeneous storage layers.

After the implementation was completed, an evaluation was performed to assess the performance of the unified B^ϵ-tree across various operations. Multiple benchmarks were performed to evaluate the advantages of the key features introduced in the unified B^ϵ-tree. Both the impact of local leaf node buffering and the effect of centralized shared NVM buffer size were examined. Operations such as insert, update, delete, and read were tested using different configurations. The results confirmed that both features improved throughput and reduced I/O overhead, demonstrating the effectiveness of the proposed hybrid design of the unified B^ϵ-tree.

In addition, the performance of the unified B^ϵ-tree was compared with the classic B^ϵ-tree and other data structures presented in Chapter 3. The evaluation showed that the proposed design had a lower warm-up and insertion total execution time than the classic B^ϵ-tree, `Fast_Fair`, and wB^+-tree. Furthermore, its total delete execution time was lower than that of `FAST_FAIR` and FPTree, while its total update execution time was lower than that of `FAST_FAIR`. Moreover, a macro benchmark using YCSB was performed to evaluate the unified B^ϵ-tree under different workloads. The results showed that it achieved higher throughput than `FAST_FAIR` and wB^+-tree in most workloads and achieved the highest throughput in the read-only workload (YCSB-C). In addition, it had the best performance in the short-range workload (YCSB-E) and outperformed NoveLSM in read-dominated workloads.

Moreover, to evaluate the effectiveness of the recovery strategy used by the tree, a series of different benchmarks was conducted for various dataset sizes. The results confirmed that the crash recovery mechanism operates efficiently, as for a dataset of size 50 million entries, the tree was restored in less than 4.5 seconds. This confirms that the hybrid crash recovery mechanism, which combines PMem consistency and WAL-based durability, scales efficiently across dataset sizes and provides both reliability and low recovery overhead.

8. Future Work

The unified B^ϵ-tree presented in this thesis introduces a new design that targets heterogeneous storage systems with NVM. As discussed in the previous chapter, the evaluation showed that the tree did not consistently outperform existing structures across all micro and macro benchmarks. As this thesis represents the first prototype of the unified design, there are several improvements that can be made in the future. This chapter presents potential directions for future work that aim to optimize the performance of the current implementation.

8.1 Reducing Lookup Total Execution Time in Persistent Data Structures

The centralized shared buffer and the auxiliary hash maps have a lookup complexity of $O(n)$ (see Table 5.1). The current implementation uses a linear scan for key lookups in the persistent NVM buffer, which is efficient for small buffer sizes. However, as the system scales or under workloads with many buffered messages, lookup total execution time can increase linearly with buffer size, which reduces the performance of the read operation. To overcome this limitation, persistent hash tables can be used to enable constant-time ($O(1)$) key lookup, even for larger buffer sizes. Dash[15], which was introduced by Lu et al. [LHWL20], is a dynamic and scalable hash table that can be used to accelerate lookups. Furthermore, Rewo-Hash[16] [HYH20] uses a dual buffer scheme with DRAM-based caching and log-free atomic writes to reduce overhead.

Other persistent hash maps, such as Plush [VVRI+22], CCEH [NCC+19], PCLHT [LMK+19], and Clevel [CHDZ20], may also be used to increase the throughput of read operations and improve scalability. A comparative study by Hu et al. [HCW+21] provides a detailed evaluation of the different NVM-optimized hash tables, which is helpful in choosing the most appropriate hash table design for the unified B^ϵ-tree.

[15]https://github.com/baotonglu/dash
[16]https://github.com/Meditator-hkx/Rewo-Hash

8.2 Multithreading and Parallel Operation Support

The current unified B$^{\epsilon}$-tree does not support multithreading. Various operations are performed sequentially, which makes the implementation single-threaded. A possible direction for future work is to extend the implementation with multithreading support, as the current implementation limits throughput and scalability. Using thread-safe mechanisms, multiple threads can insert or update keys into the shared NVM buffer simultaneously.

One possible optimization is to enable parallel access to the centralized shared buffer in NVM. In the current implementation, the buffer is accessed sequentially as an array of messages in the `PMEMRoot` structure, which introduces additional overhead. Therefore, the use of multiple threads to insert or modify messages inside the buffer could reduce the total execution time. PMDK supports multithreading by using a persistent mutex (`pmem::obj::mutex`). Furthermore, flushing the buffer of leaf nodes can be done using background threads.

8.3 Partitioned Buffers

In the current implementation of the unified B$^{\epsilon}$-tree, the centralized shared buffer in NVM is accessed as a single block by a single thread. In addition to the optimizations discussed in Section 8.2, the buffer could be logically divided into multiple blocks. For each block, a specific thread or thread group is assigned to it. This allows multiple threads to insert and manage messages in parallel, which can improve throughput. This feature should also be considered for different auxiliary hash maps (i.e., `nodeMessageMap` and `messageToNodeMap`). This reduces synchronization overhead and avoids cross thread locking bottlenecks.

8.4 Dynamic Management of the Shared Buffer

As mentioned in Section 5.3.5, the shared buffer in NVM has a fixed size, which is determined by the predefined macro `MAX_NVM_MESSAGES` in `PMEMRoot`. When this threshold is reached, a flush operation is triggered. A potential future improvement is to introduce a dynamic buffer management strategy for this buffer. The tree could monitor the message arrival rate, for example, and dynamically adjust the most effective capacity needed for the buffer.

Appendix

A.1 Different Algorithms for B$^{\epsilon}$-Tree

Algorithm A.1 Insert operation in classic B$^{\epsilon}$-tree

Require: Key k, Value v
Ensure: Insert (k, v) into the Bε-tree

```
if root is null then
    Create a new leaf node and set it as root
    Insert (k, v) into root
    return Success
else
    current ← root
    if current is an internal node then
        Buffer (k, v) in current
        if buffering fails then
            return Error
        end if
        if buffer size ≥ threshold then
            Flush buffer of current
            if flushing fails then
                return Error
            end if
        end if
        return Success
    else
        Find sorted position i for k in current.keys
        Insert k and v at position i in current
        if current is overfull then
            parent ← FindParent(root, current)
            Split current leaf node
            if splitting fails then
                return Error
            end if
        end if
        return Success
    end if
end if
```

Algorithm A.2 Delete operation in classic B$^\epsilon$-tree

Require: Key k

Ensure: Remove the key k from the Bε-tree

```
if root is null then
    return KeyDoesNotExist
end if
current ← root
if current is an internal node then
    Buffer a delete operation (k, _) in current
    if buffering fails then
        return Error
    end if
    if buffer size ≥ threshold then
        Flush the buffer
        if flushing fails then
            return Error
        end if
    end if
    return Success
else
    Search for k in current.keys
    if k is found then
        Remove k and its value from current
    else
        return KeyDoesNotExist
    end if
    if current is underfull then
        parent ← FindParent(root, current)
        Handle underflow in current
        if underflow handling fails then
            return Error
        end if
    end if
    return Success
end if
```

Algorithm A.3 Update operation in classic B$^\epsilon$-tree

Require: Key k, New value v
Ensure: Update the value of key k in the B$^\epsilon$-Tree

```
current ← root
if current is an internal node then
    Buffer an update operation (k, v) in current
    if buffering fails then
        return Error
    end if
    if buffer size ≥ threshold then
        Flush the buffer
        if flushing fails then
            return Error
        end if
    end if
    return Success
else
    Search for k in current.keys
    if k is found then
        Update the corresponding value to v
    else
        return KeyDoesNotExist
    end if
    return Success
end if
```

Algorithm A.4 Search operation in classic B$^\epsilon$-tree

Require: Key k

Ensure: Retrieve the value associated with key k if it exists

```
if root is null then
    return KeyDoesNotExist
end if
current ← root
Initialize an empty list collectedMessages
while current is an internal node do
    for all messages in current.buffer do
        if message key = k then
            Add message to collectedMessages
        end if
    end for
    i ← index of first key greater than k in current.keys
    if i ≥ number of children then
        return KeyDoesNotExist
    end if
    current ← current.children[i]
end while
Search for k in current.keys
if k is found then
    v ← corresponding value from current
else
    Initialize keyInserted ← false
    for all messages in collectedMessages do
        if message is Insert then
            v ← value from message
            keyInserted ← true
            break
        else if message is Delete then
            return KeyDoesNotExist
        end if
    end for
    if keyInserted = false then
        return KeyDoesNotExist
    end if
end if
for all remaining messages in collectedMessages do
    if message is Delete then
        return KeyDoesNotExist
    else if message is Update then
        v ← updated value from message
    end if
end for
return Success
```

Algorithm A.5 Range query in classic B$^\epsilon$-tree

```
Require: Lower bound low, upper bound high
Ensure: Return all (k, v) pairs with low ≤ key ≤ high
Initialize empty list result
current ← root
while current is an internal node do
    for all messages in current.buffer do
        if buffered key ∈ [low, high] then
            if message is Insert then
                Add (k, v) to result
            else if message is Delete then
                Remove (k) from result if present
            end if
        end if
    end for
    i ← index of first key greater than low in current.keys
    if i ≥ number of children then
        break
    end if
    current ← current.children[i]
end while
while true do
    for i = 0 to current.keys.size() do
        if current.keys[i] ∈ [low, high] then
            Add (k, v) to result
        else if current.keys[i] > high then
            return sorted result without duplicates
        end if
    end for
    parent ← FindParent(root, current)
    while parent exists do
        index ← position of current in parent.children
        if index + 1 < parent.children.size() then
            current ← parent.children[index + 1]
            while current is not a leaf do
                current ← current.children[0]
            end while
            break
        else
            current ← parent
            parent ← FindParent(root, parent)
        end if
    end while
    if parent is null then
        break
    end if
end while
Sort result and remove duplicates by key
return result
```

Algorithm A.6 Flushing buffer of internal node

```
if node is a leaf or buffer is empty then return Success
end if
bufferCopy ← node.buffer
Clear node.buffer
for all operation in bufferCopy do
    (opType, key, value) ← operation
    i ← index such that key < node.keys[i] (using upper_bound)
    if i ≥ node.children.size() then return Error
    end if
    child ← node.children[i]
    if opType = Insert then
        if child is not a leaf then
            propagate insert to child's buffer
        else
            insert key and value into child at correct position
            if child is overfull then
                result ← splitLeaf(node, child)
                if result ≠ Success then return result
                end if
                node ← findParent(root, child)
            end if
        end if
    else if opType = Delete then
        if child is not a leaf then
            propagate delete to child's buffer
        else
            if key exists in child then
                remove key and corresponding value
                if child underflows then
                    result ← handleUnderflow(node, child)
                    if result ≠ Success then
                        return result
                    end if
                end if
            end if
        end if
    else if opType = Update then
        if child is not a leaf then
            propagate update to child's buffer
        else
            if key exists in child then
                update value of key to new value
            end if
        end if
    else
        return Error
    end if
end for
return Success
```

A.2 Micro and Macro Benchmarking Results

Dataset Size	Insert	
	With Leaf Node Buffer	Without Leaf Node Buffer
10,000	5,839,632	6,582,278
100,000	70,896,184	117,061,720
250,000	218,154,348	504,483,625
500,000	562,369,546	1,655,855,186
1,000,000	892,577,308	3,056,496,372

Table A.1: Total execution time (in microseconds) comparison of insert operation with and without leaf node buffer for various dataset sizes

Dataset Size	Update	
	With Leaf Node Buffer	Without Leaf Node Buffer
10,000	5,754,240	5,613,937
100,000	62,208,776	58,327,217
250,000	173,680,500	145,704,923
500,000	405,107,632	286,556,637
1,000,000	1,095,637,621	568,113,254

Table A.2: Total execution time (in microseconds) comparison of update operation with and without leaf node buffer for various dataset sizes

Dataset Size	Delete	
	With Leaf Node Buffer	Without Leaf Node Buffer
10,000	1,862,199	2,218,285
100,000	21,005,613	57,276,537
250,000	54,178,025	288,558,378
500,000	119,473,369	1,055,979,064
1,000,000	258,760,641	2,201,689,130

Table A.3: Total execution time (in microseconds) comparison of delete operation with and without leaf node for various dataset sizes

Dataset Size	With Buffer		Without Buffer	
	Reads	Writes	Reads	Writes
50,000	57,868	65,746	71,429	92,868
100,000	115,749	131,508	142,900	185,810
250,000	289,027	328,064	357,209	464,428
500,000	578,915	657,840	714,418	928,846
1,000,000	1,157,999	1,316,008	1,428,715	1,857,440

Table A.4: SSD I/O during insert operations with and without a leaf buffer across various dataset sizes

Buffer Size (byte)	Key Count	Inserts	Updates	Deletes
200	10,000	6,623,583	6,215,828	2,124,953
	100,000	90,614,580	91,735,072	20,443,860
	250,000	320,424,045	351,270,913	54,622,211
	500,000	918,608,702	1,032,204,327	124,884,953
	1,000,000	1,815,978,167	2,250,037,011	261,598,763
300	10,000	6,207,684	6,168,306	1,731,890
	100,000	81,494,134	78,808,564	20,671,474
	250,000	268,912,337	272,488,053	57,582,223
	500,000	757,273,543	786,985,620	134,728,633
	1,000,000	1,570,319,834	1,859,564,797	281,141,124
400	10,000	6,531,440	5,877,998	2,131,643
	100,000	74,665,949	66,354,864	19,407,574
	250,000	239,453,566	197,762,600	59,034,241
	500,000	612,562,054	490,199,632	131,950,737
	1,000,000	1,335,796,716	1,666,027,562	270,395,476
500	10,000	6,803,422	6,245,085	1,795,673
	100,000	72,165,675	63,891,383	21,390,693
	250,000	220,240,100	180,106,753	56,761,586
	500,000	573,094,905	435,004,226	124,250,461
	1,000,000	1,001,962,418	1,255,537,809	264,624,443
600	10,000	5,839,632	5,754,240	1,862,199
	100,000	70,896,184	62,208,776	21,005,613
	250,000	218,154,348	173,680,500	54,178,025
	500,000	562,369,546	405,107,632	119,473,369
	1,000,000	892,577,308	1,095,637,621	258,760,641

Table A.5: Benchmark performance (in microseconds) for different leaf node buffer sizes and key counts for write operations

Buffer Size (messages)	Insert Warmup	Search	Insert	Update	Delete
100	1,540.47	108,720.00	1,998.03	1,556.75	1,894.42
200	1,576.75	111,584.00	1,896.04	1,505.75	1,826.93
300	1,546.29	101,543.00	1,901.28	1,557.47	1,791.00
400	1,580.29	96,157.00	2,036.39	1,676.62	2,019.62
500	1,520.12	93,806.50	2,007.70	1,627.76	1,987.32
600	1,546.55	89,119.80	2,045.26	1,686.66	2,028.23
700	1,552.43	89,375.80	1,993.88	1,570.34	1,801.52
800	1,571.53	82,880.70	1,950.58	1,623.64	2,011.79
900	1,560.56	80,419.90	1,975.96	1,616.48	2,006.20
1,000	1,563.97	73,929.60	2,057.73	1,579.68	1,948.57
1,100	1,551.54	74,320.90	2,024.46	1,656.23	2,030.90
1,200	1,554.21	75,645.30	2,055.25	1,588.84	1,992.19
1,300	1,543.71	72,367.80	2,041.06	1,630.63	1,750.88
1,400	1,505.48	69,004.10	2,021.96	1,601.73	1,892.24
1,500	1,524.25	66,086.20	2,055.10	1,600.51	1,989.01
1,600	1,499.84	68,702.90	2,039.61	1,596.85	1,961.52
1,700	1,482.48	64,316.30	2,046.37	1,563.37	1,988.06
1,800	1,537.57	56,955.10	2,004.05	1,598.61	2,135.67
1,900	1,537.83	59,815.50	2,008.87	1,595.63	1,945.14
2,000	1,465.83	57,868.20	1,947.52	1,545.98	1,966.49
2,100	1,494.80	47,825.00	1,951.55	1,588.54	1,873.28
2,200	1,469.61	57,995.00	2,008.67	1,570.94	1,954.34
2,300	1,491.01	52,278.70	1,971.81	1,484.54	1,724.78
2,400	1,490.07	51,408.00	1,996.75	1,536.44	1,892.92
2,500	1,451.42	54,693.30	1,946.20	1,485.14	1,879.14
2,600	1,462.53	50,328.10	1,849.01	1,404.42	1,614.16
2,700	1,297.45	51,138.00	1,952.86	1,591.52	1,909.92
2,800	1,498.46	49,788.10	1,933.93	1,490.43	1,903.30
2,900	1,455.35	46,886.70	1,878.44	1,497.80	1,855.77
3,000	1,366.16	46,585.90	1,695.97	1,308.62	1,858.92
3,100	1,490.09	46,887.00	1,954.83	1,558.74	1,871.97
3,200	1,449.58	45,581.00	1,906.57	1,512.92	1,862.48
3,300	1,409.64	43,424.30	1,905.86	1,520.88	1,898.09
3,400	1,433.09	43,149.60	1,898.22	1,495.18	1,701.35
3,500	1,412.64	44,309.70	1,738.19	1,543.21	1,628.87
3,600	1,467.52	42,120.30	1,642.46	1,403.10	1,885.62
3,700	1,445.01	39,583.60	1,869.25	1,445.22	1,703.77
3,800	1,351.50	41,324.30	1,744.77	1,328.41	1,724.43
3,900	1,433.31	39,123.90	1,842.73	1,471.22	1,829.54
4,000	1,412.99	31,021.50	1,796.33	1,534.75	1,652.80

Table A.6: Throughput (ops/sec) for different operations across various NVM shared buffer sizes

Dataset Size	Unified B^{ϵ}-Tree		Classic B^{ϵ}-Tree	
	Warm-up	Insert	Warm-up	Insert
50,000	22,179,334	21,936,947	32,202,880	101,849,987
100,000	44,506,942	43,352,203	80,256,500	156,215,233
250,000	113,619,872	111,659,861	193,829,770	854,190,885
500,000	227,804,536	223,866,254	965,022,455	2,889,955,712
1,000,000	436,468,956	429,932,993	1,859,044,819	5,635,864,424

Table A.7: Total execution time (in microseconds) comparison of warm-up and insert phases for the unified and classic B^{ϵ}-trees

Dataset Size	WAL Size (MB)	Recovery Time (s)
10,000	0.166	0.9117
50,000	0.830	0.9796
100,000	1.660	1.0002
250,000	4.150	1.2506
500,000	8.301	1.4437
1,000,000	16.602	1.5965
5,000,000	83.008	1.6388
10,000,000	166.016	1.7236
15,000,000	249.023	2.1092
20,000,000	332.031	2.5946
30,000,000	498.047	3.0116
40,000,000	664.057	3.5844
50,000,000	830.078	4.3211

Table A.8: Recovery performance of the unified B^{ϵ}-tree with various dataset sizes

	Unified B^{ϵ}-tree	FAST_FAIR	wB^{+}-tree	FPTree	NoveLSM
Load	6,567	5,685	6,193	7,668	9,164
A	21,379	23,180	20,264	26,175	29,145
B	95,658	98,143	90,127	105,132	88,953
C	201,642	142,642	99,645	195,130	75,114
D	23,547	22,598	20,135	24,475	12,247
E	10,233	7,543	6,103	4,844	3,075
F	16,037	15,634	13,009	16,847	18,584

Table A.9: Throughput (ops/sec) on the YCSB workloads

	Number of Operations	Warm-Up Insert	Search	Insert	Update	Delete
Unified B$^{\epsilon}$-Tree	10,000	3,985,447	36,313	4,228,718	4,180,967	1,190,785
	50,000	22,179,334	213,969	22,044,243	21,936,947	1,743,534
	100,000	44,506,942	413,339	43,952,147	43,352,203	3,595,649
	250,000	113,619,872	1,163,640	111,789,261	111,659,861	15,187,971
	500,000	227,804,536	2,488,787	225,174,944	223,866,254	22,503,608
	1,000,000	461,268,863	5,104,098	460,103,924	450,517,612	40,261,798
FAST-FAIR [HKWN18]	10,000	8,942,880	21,118	9,763,855	4,449,755	968,929
	50,000	32,322,740	66,333	32,199,408	22,555,377	1,627,044
	100,000	66,609,064	202,193	66,204,890	45,438,930	3,311,334
	250,000	244,023,714	579,897	215,430,890	140,322,798	19,974,459
	500,000	381,041,784	1,291,197	379,368,888	253,670,609	27,219,651
	1,000,000	674,536,630	3,071,027	685,963,943	476,462,988	46,818,184
wB^{+}-tree [CJ15]	10,000	6,145,132	16,446	6,245,338	26,757	20,514
	50,000	29,896,279	98,956	29,958,645	268,485	198,267
	100,000	66,890,001	330,024	67,780,176	679,600	357,493
	250,000	149,124,664	1,110,839	153,863,596	1,876,318	1,170,425
	500,000	291,286,477	3,107,125	305,485,637	3,211,345	2,504,398
	1,000,000	501,589,637	8,256,786	520,448,883	60,236,894	6,105,753
FPTree [OLN+16]	10,000	844,739	3,525	650,867	560,289	443,254
	50,000	5,051,010	17,448	3,143,998	2,824,099	2,512,938
	100,000	8,269,475	39,887	4,014,589	5,370,953	3,866,659
	250,000	20,356,824	117,327	7,704,646	13,289,653	8,531,482
	500,000	123,135,151	406,130	107,435,372	101,020,070	64,277,376
	1,000,000	245,782,332	896,790	195,104,875	139,691,420	106,422,075

Table A.10: Total execution time (in microseconds) comparison of read and write operations for different NVM optimized trees

Bibliography

[ABB$^+$14] Sattam Alsubaiee, Alexander Behm, Vinayak Borkar, Zachary Heilbron, Young-Seok Kim, Michael J. Carey, Markus Dreseler, and Chen Li. Storage management in asterixdb. *Proc. VLDB Endow.*, 7(10):841–852, June 2014. Accessed: 30.04.2025. doi:10.14778/2732951.2732958. (cited on Page 14)

[ADS16] Dmytro Apalkov, Bernard Dieny, and J. M. Slaughter. Magnetoresistive random access memory. *Proceedings of the IEEE*, 104(10):1796–1830, 2016. Accessed: 01.04.2025. URL: https://hal.science/hal-01834195, doi:10.1109/JPROC.2016.2590142. (cited on Page 17)

[AGEM24] Milad Ashtari Gargari, Nima Eslami, and Mohammad Hossein Moaiyeri. A reconfigurable nonvolatile memory architecture for prolonged wearable health monitoring devices. *IEEE Transactions on Consumer Electronics*, 70(2):4717–4728, 2024. Accessed: 13.04.2025. doi:10.1109/TCE.2024.3399223. (cited on Page 19)

[Akr21] Shoaib Akram. Performance evaluation of intel optane memory for managed workloads. *ACM Transactions on Architecture and Code Optimization*, 18:1–26, 04 2021. Accessed: 14.04.2025. doi:10.1145/3451342. (cited on Page 18)

[Apaa] Apache Software Foundation. Apache cassandra official website. https://cassandra.apache.org/_/index.html. Accessed: 29.04.2025. (cited on Page 14)

[Apab] Apache Software Foundation. Apache hbase official website. https://hbase.apache.org/. Accessed: 30.04.2025. (cited on Page 14)

[BCMV03] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti. Introduction to flash memory. *Proceedings of the IEEE*, 91(4):489–502, 2003. Accessed: 11.04.2025. doi:10.1109/JPROC.2003.811702. (cited on Page 17)

[BF03] Gerth Stolting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 546–554, USA, 2003. Society for Industrial and Applied Mathematics. Accessed: 03.04.2025. (cited on Page 10)

[BFCJ+15] Michael A. Bender, Martín Farach-Colton, William K. Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan. An introduction to bϵ-trees and write-optimization. *login Usenix Mag.*, 40, 2015. Accessed: 03.04.2025. URL: https://api.semanticscholar.org/CorpusID:2305387. (cited on Page xi, xiii, 10, 11, 28, 33, 34, 35, 39, 62, and 65)

[BGVW00] Adam Buchsbaum, Michael Goldwasser, Suresh Venkatasubramanian, and Jeffery Westbrook. On external memory graph traversal. *SODA '00: Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, 11 2000. Accessed: 04.04.2025. doi:10.1145/338219.338650. (cited on Page 11)

[BKS+08] Geoffrey Burr, Bulent Kurdi, Campbell Scott, Chung Lam, Kailash Gopalakrishnan, and Rohit Shenoy. Overview of candidate device technologies for storage-class memory. *IBM Journal of Research and Development*, 52:449–464, 07 2008. Accessed: 04.04.2025. doi:10.1147/rd.524.0449. (cited on Page xiii and 18)

[BM70] 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*, SIGFIDET '70, pages 107–141, New York, NY, USA, 1970. Association for Computing Machinery. Accessed: 15.04.2025. doi:10.1145/1734663.1734671. (cited on Page 5 and 8)

[BO17] Jalil Boukhobza and Pierre Olivier. 10 - emerging non-volatile memories. In Jalil Boukhobza and Pierre Olivier, editors, *Flash Memory Integration*, pages 203–224. Elsevier, 2017. Accessed: 15.04.2025. URL: https://www.sciencedirect.com/science/article/pii/B9781785481246500141, doi:10.1016/B978-1-78548-124-6.50014-1. (cited on Page xiii and 16)

[Bre04] J.E. Brewer. Lest we forget: Nvsm from origins to the "beyond cmos" era. In *Proceedings. 2004 IEEE Computational Systems Bioinformatics Conference*, pages 1–, 2004. Accessed: 11.04.2025. doi:10.1109/NVMT.2004.1380787. (cited on Page 17)

[CBB21] Zixian Cai, Stephen M. Blackburn, and Michael D. Bond. Understanding and utilizing hardware transactional memory capacity. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2021, pages 1–14, New York, NY, USA, 2021. Association for Computing Machinery. Accessed: 20.04.2025. doi:10.1145/3459898.3463901. (cited on Page 24)

[CCA+11] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. *ACM SIGARCH Computer Architecture News*, 39(1):105–118, 2011. Accessed: 11.04.2025. (cited on Page 17)

[CCM+10] Adrian M. Caulfield, Joel Coburn, Todor Mollov, Arup De, Ameen Akel, Jiahua He, Arun Jagatheesan, Rajesh K. Gupta, Allan Snavely, and Steven Swanson. Understanding the impact of emerging non-volatile memories on high-performance, io-intensive computing. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2010. Accessed: 11.04.2025. doi:10.1109/SC.2010.56. (cited on Page 19)

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), June 2008. Accessed: 29.04.2025. doi:10.1145/1365815.1365816. (cited on Page 14)

[CGN11] Shimin Chen, Phillip Gibbons, and Suman Nath. Rethinking database algorithms for phase change memory. In *Cidr*, pages 21–31, 04 2011. Accessed: 15.04.2025. (cited on Page 22)

[CHDZ20] Zhangyu Chen, Yu Hua, Bo Ding, and Pengfei Zuo. Lock-free concurrent level hashing for persistent memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 799–812, 2020. Accessed: 18.07.2025. (cited on Page 83)

[Che20] Yangyin Chen. Reram: History, status, and future. *IEEE Transactions on Electron Devices*, 67(4):1420–1433, 2020. Accessed: 04.04.2025. doi:10.1109/TED.2019.2961505. (cited on Page 18)

[CHRG23] Preeti Chaudhary, Meghana A. Hasamnis, and Himanshu Rai Goyal. Non-volatile random access memory based on memristors: Design and analysis. In *2023 Second International Conference on Augmented Intelligence and Sustainable Systems (ICAISS)*, pages 1666–1671, 2023. Accessed: 28.04.2025. doi:10.1109/ICAISS58487.2023.10250534. (cited on Page 17)

[CJ15] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proc. VLDB Endow.*, 8(7):786–797, February 2015. Accessed: 15.04.2025. doi:10.14778/2752939.2752947. (cited on Page 1, 22, 31, 42, 71, 73, and 95)

[CLF+20] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. utree: A persistent b+-tree with low tail latency. *Proceedings of the VLDB Endowment*, 13(12):2634–2648, 2020. Accessed: 25.07.2025. (cited on Page xiii and 66)

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009. Accessed: 16.04.2025. (cited on Page 6)

[Com79] Douglas Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979. Accessed: 18.04.2025. doi:10.1145/356770.356776. (cited on Page 8)

[CST+10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010. Accessed: 21.08.2025. (cited on Page 77)

[CWH+14] Renhai Chen, Yi Wang, Jingtong Hu, Duo Liu, Zili Shao, and Yong Guan. Virtual-machine metadata optimization for i/o traffic reduction in mobile virtualization. In *2014 IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–2, 2014. Accessed: 10.04.2025. doi:10.1109/NVMSA.2014.6927204. (cited on Page 19)

[DAT+14] Dhananjoy Das, Dulcardo Arteaga, Nisha Talagala, Torben Mathiasen, and Jan Lindström. {NVM}{Compression-Hybrid}{Flash-Aware} application level compression. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 14)*, 2014. Accessed: 11.04.2025. (cited on Page 19)

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007. Accessed: 29.04.2025. doi:10.1145/1323293.1294281. (cited on Page 14)

[DPV21] Toan Dao Thanh, Viet-Thanh Pham, and Christos Volos. Chapter 17 - implementation of organic rram with ink-jet printer: from design to using in rfid-based application. In Christos Volos and Viet-Thanh Pham, editors, *Mem-elements for Neuromorphic Circuits with Artificial Intelligence Applications*, Advances in Nonlinear Dynamics and Chaos (ANDC), pages 347–360. Academic Press, 2021. Accessed: 15.04.2025. URL: https://www.sciencedirect.com/science/article/pii/B9780128211847000268, doi:10.1016/B978-0-12-821184-7.00026-8. (cited on Page xi and 17)

[DSST89] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989. Accessed: 03.05.2025. URL: https://www.sciencedirect.com/science/article/pii/0022000089900342, doi:10.1016/0022-0000(89)90034-2. (cited on Page 27)

[DZC+13] Penglin Dai, Qingfeng Zhuge, Xianzhang Chen, Weiwen Jiang, and Edwin H.-M. Sha. Effective file data-block placement for different types of page cache on hybrid main memory architectures. *Design Automation for Embedded Systems*, 17(3):485–506, 2013. Accessed: 11.04.2025. doi:10.1007/s10617-014-9148-3. (cited on Page 19)

[EBFCK12] John Esmet, Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. The tokufs streaming file system. In *Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'12, page 14, USA, 2012. USENIX Association. Accessed: 13.04.2025. (cited on Page 11)

[ECKY13] Hyeonsang Eom, Chanho Choi, Shin-gyu Kim, and Heon Y. Yeom. Evaluation of dram power consumption in server platforms. In Youn-Hee Han, Doo-Soon Park, Weijia Jia, and Sang-Soo Yeo, editors, *Ubiquitous Information Technologies and Applications*, pages 799–805, Dordrecht, 2013. Springer Netherlands. Accessed: 06.07.2025. (cited on Page 1)

[EN10] Ramez Elmasri and Shamkant Navathe. *Fundamentals of Database Systems*. Addison-Wesley Publishing Company, USA, 6th edition, 2010. Accessed: 18.04.2025. (cited on Page 8)

[FHJ18] Wang Fuzong, Guo Helin, and Zhao Jian. Dynamic data compression algorithm selection for big data processing on local file system. In *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence*, CSAI '18, pages 110–114, New York, NY, USA, 2018. Association for Computing Machinery. Accessed: 01.05.2025. doi:10.1145/3297156.3297233. (cited on Page 27)

[FNW17] Scott W. Fong, Christopher M. Neumann, and H.-S. Philip Wong. Phase-change memory-towards a storage-class memory. *IEEE Transactions on Electron Devices*, 64(11):4374–4385, 2017. Accessed: 01.04.2025. doi:10.1109/TED.2017.2746342. (cited on Page 18)

[Goo21] Google. Leveldb on github. https://github.com/google/leveldb, 2021. Accessed: 30.04.2025. (cited on Page 14)

[GS20] Manuel Gallo and Abu Sebastian. An overview of phase-change memory device physics. *Journal of Physics D: Applied Physics*, 53, 03 2020. Accessed: 03.04.2025. doi:10.1088/1361-6463/ab7794. (cited on Page 18)

[GvRL+18] Philipp Götze, Alexander van Renen, Lucas Lersch, Viktor Leis, and Ismail Oukid. Data management on non-volatile memory: A perspective. *Datenbank-Spektrum*, 18, 10 2018. Accessed: 25.04.2025. doi:10.1007/s13222-018-0301-1. (cited on Page xiii and 31)

[HCLH20] Jiangkun Hu, Youmin Chen, Youyou Lu, and Xubin He. Understanding and analysis of b+ trees on nvm towards consistency and efficiency. *CCF Transactions on High Performance Computing*, 2, 03 2020. Accessed: 23.04.2025. doi:10.1007/s42514-020-00022-z. (cited on Page 24)

[HCW+21] Daokun Hu, Zhiwen Chen, Jianbing Wu, Jianhua Sun, and Hao Chen. Persistent memory hash indexes: An experimental evaluation. *Proceedings of the VLDB Endowment*, 14(5):785–798, 2021. Accessed: 20.07.2025. (cited on Page 83)

[HKWN18] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in Byte-Addressable persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 187–200, Oakland, CA, February 2018. USENIX

Association. Accessed: 07.07.2025. URL: https://www.usenix.org/conference/fast18/presentation/hwang. (cited on Page 1, 24, 25, 31, 42, 71, 73, and 95)

[HR24] Ishrag Hamid and MM Rahman. A comprehensive literature review on volatile memory forensics. *Electronics (2079-9292)*, 13(15), 2024. Accessed: 16.04.2025. (cited on Page 16)

[HT20] Takahiro Hirofuchi and Ryousei Takano. A prompt report on the performance of intel optane dc persistent memory module. *IEICE Transactions on Information and Systems*, E103.D(5):1168–1172, May 2020. Accessed: 13.04.2025. URL: http://dx.doi.org/10.1587/transinf.2019EDL8141, `doi:10.1587/transinf.2019edl8141`. (cited on Page 19)

[HYH20] Kaixin Huang, Yan Yan, and Linpeng Huang. Revisiting persistent hash table design for commercial non-volatile memory. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 708–713. IEEE, 2020. Accessed: 15.07.2025. (cited on Page 83)

[HYZ+25] Xianyu He, Chaoshu Yang, Runyu Zhang, Huizhang Luo, Zhichao Cao, and Jeff Zhang. Optimizing both performance and tail latency for b + tree on persistent memory. *Journal of Systems Architecture*, 163:103406, 04 2025. Accessed: 18.04.2025. `doi:10.1016/j.sysarc.2025.103406`. (cited on Page 8)

[Hö21] Till Höppner. Design and implementation of an object store with tiered storage. Bachelor's thesis, Otto-von-Guericke Universität Magdeburg, Magdeburg, Germany, 2021. Accessed: 01.05.2025. (cited on Page xi, 1, 27, 29, 33, 34, and 42)

[IMJA20] Sumio Ikegawa, Frederick B. Mancoff, Jason Janesky, and Sanjeev Aggarwal. Magnetoresistive random access memory: Present and future. *IEEE Transactions on Electron Devices*, 67(4):1407–1419, 2020. Accessed: 01.04.2025. `doi:10.1109/TED.2020.2965403`. (cited on Page 17)

[Inta] Intel. libpmem. Accessed: 15.05.2025. URL: https://pmem.io/pmdk/libpmem/. (cited on Page 42)

[Intb] Intel. Libpmemblk. Accessed: 15.05.2025. URL: https://pmem.io/pmdk/libpmemblk/. (cited on Page 42)

[Intc] Intel. Libpmemlog. Accessed: 15.05.2025. URL: https://pmem.io/pmdk/libpmemlog/. (cited on Page 42)

[Intd] Intel. Libpmemobj. Accessed: 15.05.2025. URL: https://pmem.io/pmdk/libpmemobj/. (cited on Page 42)

[Inte] Intel. Libpmempool. Accessed: 15.05.2025. URL: https://pmem.io/pmdk/libpmempool/. (cited on Page 42)

[Int17] Intel. Announcing the persistent memory development kit, 12 2017. Accessed: 15.05.2025. URL: https://pmem.io/2017/12/11/NVML-is-now-PMDK.html. (cited on Page 42)

[Int22] Intel Corporation. Intel® optane™ persistent memory 200 series brief, 2022. Accessed: 13.04.2025. URL: https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-persistent-memory-200-series-brief.html. (cited on Page 19)

[IRLP16] Nusrat Islam, Md Rahman, Xiaoyi Lu, and D.K. Panda. High performance design for hdfs with byte-addressability of nvm and rdma. In *Proceedings of the 2016 International Conference on Supercomputing*, pages 1–14, 06 2016. Accessed: 11.04.2025. doi:10.1145/2925426.2926290. (cited on Page 19)

[IS14] Soojun Im and Dongkun Shin. Differentiated space allocation for wear leveling on phase-change memory-based storage device. *IEEE Transactions on Consumer Electronics*, 60(1):45–51, 2014. Accessed: 10.04.2025. doi:10.1109/TCE.2014.6780924. (cited on Page 19)

[Ish12] Hiroshi Ishiwara. Ferroelectric random access memories. *Journal of nanoscience and nanotechnology*, 12:7619–27, 10 2012. Accessed: 01.04.2025. doi:10.1166/jnn.2012.6651. (cited on Page 17)

[IYZ+19] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane dc persistent memory module, 2019. Accessed: 13.04.2025. URL: https://arxiv.org/abs/1903.05714, arXiv:1903.05714. (cited on Page 19)

[JK93] Theodore Johnson and Padmashree Krishna. Lazy updates for distributed search structure. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 337–346, New York, NY, USA, 1993. Association for Computing Machinery. Accessed: 20.04.2025. doi:10.1145/170035.170085. (cited on Page 8)

[JTK+12] Doo Seok Jeong, Reji Thomas, Ram Katiyar, J Scott, H. Kohlstedt, Adrian Petraru, and Cheol Hwang. Emerging memories: Resistive switching mechanisms and current status. *Reports on progress in physics. Physical Society (Great Britain)*, 75:076502, 07 2012. Accessed: 15.04.2025. doi:10.1088/0034-4885/75/7/076502. (cited on Page xi and 17)

[JWC+13] Myoungsoo Jung, Ellis H. Wilson, Wonil Choi, John Shalf, Hasan Metin Aktulga, Chao Yang, Erik Saule, Umit V. Catalyurek, and Mahmut Kandemir. Exploring the future of out-of-core computing with compute-local non-volatile memory. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2013. Accessed: 11.04.2025. doi:10.1145/2503210.2503261. (cited on Page 19)

[JYY+18] Huei-Yau Jeng, Tzu-Chien Yang, Li Yang, James G. Grote, Hsin-Lung Chen, and Yu-Chueh Hung. Non-volatile resistive memory devices based on solution-processed natural dna biomaterial. *Organic Electronics*, 54:216–221, 2018. Accessed: 13.04.2025. URL: https://www.sciencedirect.com/science/article/pii/S1566119917306535, doi:10.1016/j.orgel.2017.12.048. (cited on Page 19)

[JYZ+15] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Betrfs: Write-optimization in a kernel file system. *ACM Trans. Storage*, 11(4), November 2015. Accessed: 12.04.2025. doi:10.1145/2798729. (cited on Page xi, 9, and 11)

[Kaz18] Uranus Kazemi. A survey of big data: challenges and specifications. *CiiT International Journal of Software Engineering and Technology*, 10(5):87–93, 2018. Accessed: 01.07.2025. (cited on Page 1)

[KB10] Petra Koruga and Miroslav Baca. Analysis of b-tree data structure and its usage in computer forensics. In *Central European Conference on Information and Intelligent Systems*, page 423. Faculty of Organization and Informatics Varazdin, 2010. Accessed: 15.04.2025. (cited on Page 6)

[KBG+18] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning lsms for nonvolatile memory with novelsm. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 993–1005, 2018. Accessed: 27.08.2025. (cited on Page xi, 1, 21, 22, 31, and 77)

[KGT+14] Manu Komalan, J.I. Gomez, Christian Tenllado, José Montañana, Antonio Artés, Francisco Tirado, and Francky Catthoor. Design exploration of a nvm based hybrid instruction memory organization for embedded platforms. *Design Automation for Embedded Systems*, 17, 10 2014. Accessed: 10.04.2025. doi:10.1007/s10617-014-9151-8. (cited on Page 19)

[KITO12] T. Kawahara, K. Ito, R. Takemura, and H. Ohno. Spin-transfer torque ram technology: Review and prospect. *Microelectronics Reliability*, 52(4):613–627, 2012. Advances in non-volatile memory technology. Accessed: 01.04.2025. URL: https://www.sciencedirect.com/science/article/pii/S002627141100446X, doi:10.1016/j.microrel.2011.09.028. (cited on Page 18)

[KJ92] P. Krishna and T. Johnson. Implementing distributed search structures. Technical Report TR94009, University of Florida, 1992. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=1e98d9ded9b55813b370d7fc44389896dfcd6308, Accessed: 20.04.2025. (cited on Page 6)

[Knu98] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching.* Addison Wesley Longman Publishing Co., Inc., USA, 1998. Accessed: 16.04.2025. (cited on Page 6)

[KR98] Robert L. Kruse and Alexander J. Ryba. *Data structures and program design in C++.* Prentice-Hall, Inc., USA, 1998. Accessed: 16.04.2025. (cited on Page 6)

[KWB+23] Sajad Karim, Johannes Wünsche, David Broneske, Michael Kuhn, and Gunter Saake. Assessing non-volatile memory in modern heterogeneous storage landscape using a write-optimized storage stack. 3714, 06 2023. Accessed: 02.05.2025. (cited on Page xi, 28, 29, 33, 35, and 42)

[KWB+24] Sajad Karim, Johannes Wünsche, David Broneske, Michael Kuhn, and Gunter Saake. A design proposal for a unified b-epsilon-tree: Embracing nvm in memory hierarchies. 3710:43–50, 05 2024. Accessed: 01.04.2025. (cited on Page xi, 1, 2, 3, 17, 29, 33, 34, 35, 36, 42, 62, and 81)

[KWK+25] Sajad Karim, Johannes Wünsche, Michael Kuhn, Gunter Saake, and David Broneske. Nvm in data storage: A post-optane future. *ACM Transactions on Storage*, 2025. Accessed: 19.08.2025. (cited on Page xiii and 18)

[Lam10] Chung H. Lam. Storage class memory. In *2010 10th IEEE International Conference on Solid-State and Integrated Circuit Technology*, pages 1080–1083, 2010. Accessed: 21.04.2025. doi:10.1109/ICSICT.2010.5667551. (cited on Page 17)

[LC19] Chen Luo and Michael J. Carey. Lsm-based storage techniques: a survey. *The VLDB Journal*, 29(1):393–418, July 2019. Accessed: 29.04.2025. doi:10.1007/s00778-019-00555-y. (cited on Page 14 and 15)

[LC25] Hui-Tang Luo and Tseng-Yi Chen. Rethinking b-epsilon tree indexing structure over nvm with the support of multi-write modes. In *Proceedings of the 30th Asia and South Pacific Design Automation Conference*, ASPDAC '25, pages 251–257, New York, NY, USA, 2025. Association for Computing Machinery. Accessed: 15.04.2025. doi:10.1145/3658617.3697566. (cited on Page xi, 1, 29, 30, 33, 34, and 42)

[LCJ+20] Haikun Liu, Di Chen, Hai Jin, Xiaofei Liao, Bingsheng He, Kan Hu, and Yu Zhang. A survey of non-volatile main memory technologies: State-of-the-arts, practices, and future directions, 2020. Accessed: 13.04.2025. URL: https://arxiv.org/abs/2010.04406, arXiv:2010.04406. (cited on Page 19)

[LCT+22] Fengbei Liu, Yuanhong Chen, Yu Tian, Yuyuan Liu, Chong Wang, Vasileios Belagiannis, and Gustavo Carneiro. Nvum: Non-volatile unbiased memory for robust medical image classification, 2022. Accessed: 13.04.2025. URL: https://arxiv.org/abs/2103.04053, arXiv:2103.04053. (cited on Page 19)

[LCW20] Jihang Liu, Shimin Chen, and Lujun Wang. Lb+trees: optimizing persistent index performance on 3dxpoint memory. *Proceedings of the VLDB Endowment*, 13:1078–1090, 03 2020. Accessed: 25.04.2025. doi:10.14778/3384345.3384355. (cited on Page xi, 1, 26, 27, 31, and 42)

[LHWL20] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable hashing on persistent memory. *arXiv preprint arXiv:2003.07302*, 2020. Accessed: 15.07.2025. (cited on Page 83)

[LKE18] Minho Lee, Dong Hyun Kang, and Young Ik Eom. M-clock: Migration-optimized page replacement algorithm for hybrid memory architecture. *ACM Trans. Storage*, 14(3), October 2018. Accessed: 10.04.2025. doi:10.1145/3216730. (cited on Page 19)

[LMK+19] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: Converting concurrent dram indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 462–477, 2019. Accessed: 18.07.2025. (cited on Page 83)

[LPF19] Tobias Latzo, Ralph Palutke, and Felix Freiling. A universal taxonomy and survey of forensic memory acquisition techniques. *Digital Investigation*, 28:56–69, 2019. Accessed: 16.04.2025. URL: https://www.sciencedirect.com/science/article/pii/S1742287618304535, doi:10.1016/j.diin.2019.01.001. (cited on Page 16)

[LPG+17] Lanyue Lu, Thanumalayan Pillai, Hariharan Gopalakrishnan, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage*, 13:1–28, 03 2017. Accessed: 23.04.2025. doi:10.1145/3033273. (cited on Page 14)

[LY81] Philip L. Lehman and s. Bing Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4):650–670, December 1981. Accessed: 20.04.2025. doi:10.1145/319628.319663. (cited on Page 6 and 8)

[LZL+13] Jian Li, Qingfeng Zhuge, Duo Liu, Huizhang Luo, and Edwin H.-M. Sha. A content-aware writing mechanism for reducing energy on non-volatile memory based embedded storage systems. *Des. Autom. Embedded Syst.*, 17(3-4):711–737, September 2013. Accessed: 10.04.2025. doi:10.1007/s10617-014-9150-9. (cited on Page 19)

[Mar] MariaDB Foundation. Mariadb official website. https://mariadb.org/. Accessed: 12.04.2025. (cited on Page 11)

[MDH+01] T. Mikolajick, C. Dehm, W. Hartner, I. Kasko, M.J. Kastner, N. Nagel, M. Moert, and C. Mazure. Feram technology for high density applications. *Microelectronics Reliability*, 41(7):947–950, 2001. Accessed: 01.04.2025. URL: https://www.sciencedirect.com/science/article/pii/

S002627140100049X, doi:10.1016/S0026-2714(01)00049-X. (cited on Page 17)

[Met22] Meta. Rocksdb official website. https://rocksdb.org/, 2022. Accessed: 30.04.2025. (cited on Page xi, 14, and 15)

[Mic] Microsoft Corporation. Ntfs overview. Accessed: 18.04.2025. URL: https://learn.microsoft.com/en-us/windows-server/storage/file-server/ntfs-overview. (cited on Page 8)

[Mis24] Supriya Mishra. A survey of lsm-tree based indexes, data systems and kv-stores. In *2024 IEEE International Students' Conference on Electrical, Electronics and Computer Science (SCEECS)*, pages 1–6, 2024. Accessed: 25.04.2025. doi:10.1109/SCEECS61402.2024.10482249. (cited on Page 14)

[MSCT14] Jagan Meena, Simon Sze, Umesh Chand, and Tseung-Yuen Tseng. Overview of emerging non-volatile memory technologies. *Nanoscale Research Letters*, 9:1–33, 09 2014. Accessed: 11.04.2025. doi:10.1186/1556-276X-9-526. (cited on Page 17)

[MyS] MySQL. Mysql official website. https://www.mysql.com/. Accessed: 15.04.2025. (cited on Page 11)

[NCC+19] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. {Write-Optimized} dynamic hashing for persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 31–44, 2019. Accessed: 17.07.2025. (cited on Page 83)

[OCGO96] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, June 1996. Accessed: 25.04.2025. doi:10.1007/s002360050048. (cited on Page 14)

[OLN+16] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 371–386, New York, NY, USA, 2016. Association for Computing Machinery. Accessed: 15.04.2025. doi:10.1145/2882903.2915251. (cited on Page xi, 1, 23, 25, 31, 42, 71, 73, and 95)

[Ora] Oracle Corporation. *MySQL 8.4 Reference Manual: The InnoDB Storage Engine*. Accessed: 18.04.2025. URL: https://dev.mysql.com/doc/refman/9.3/en/innodb-storage-engine.html. (cited on Page 8)

[Per22] Percona. Tokudb storage engine in percona server for mysql. https://docs.percona.com/percona-server/5.7/tokudb/tokudb_intro.html, 2022. Accessed: 12.04.2025. (cited on Page 11)

[POL$^+$19] Georgios Psaropoulos, Ismail Oukid, Thomas Legler, Norman May, and Anastasia Ailamaki. Bridging the latency gap between nvm and dram for latency-bound operations. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, DaMoN'19, New York, NY, USA, 2019. Association for Computing Machinery. Accessed: 10.04.2025. doi:10.1145/3329785.3329917. (cited on Page 36)

[Pos] PostgreSQL Global Development Group. Postgresql documentation. Accessed: 18.04.2025. URL: https://www.postgresql.org/docs/current/index.html. (cited on Page 6)

[Pug90] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, June 1990. Accessed: 23.04.2025. doi:10.1145/78973.78977. (cited on Page 15)

[PZDR$^+$19] Gianlucca Puglia, Avelino Zorzo, Cesar De Rose, Taciano Perez, and Dejan Milojicic. Non-volatile memory file systems: A survey. *IEEE Access*, PP:1–1, 02 2019. Accessed: 04.04.2025. doi:10.1109/ACCESS.2019.2899463. (cited on Page xiii, 16, and 18)

[Ram24] Siddhartha Raman. *A Review on Non-Volatile and Volatile Emerging Memory Technologies*, page 23. BoD–Books on Demand, 01 2024. Accessed: 28.04.2025. doi:10.5772/intechopen.110617. (cited on Page 17)

[RKCA17] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 497–514, New York, NY, USA, 2017. Association for Computing Machinery. Accessed: 23.04.2025. doi:10.1145/3132747.3132765. (cited on Page 14)

[RRB10] Ugo Russo, Andrea Redaelli, and Roberto Bez. Non-Volatile Memory Technology Overview. WTAI: Workshop on Technology Architecture Interaction, June 2010. Accessed: 01.04.2025. URL: https://inria.hal.science/inria-00514448. (cited on Page xiii, 17, and 18)

[Sca20a] Steve Scargall. *Programming Persistent Memory: A Comprehensive Guide for Developers*. Apress, Berkeley, CA, 2020. Accessed: 14.04.2025. doi:10.1007/978-1-4842-4932-1. (cited on Page 19)

[Sca20b] Steve Scargall. *Programming Persistent Memory: A Comprehensive Guide for Developers*, chapter 5, pages 63–72. Apress, 01 2020. Accessed: 10.04.2025. doi:10.1007/978-1-4842-4932-1. (cited on Page 42)

[SDD$^+$02] J. M. Slaughter, R. W. Dave, M. DeHerrera, M. Durlam, B. N. Engel, J. Janesky, N. D. Rizzo, and S. Tehrani. Fundamentals of mram technology. *Journal of Superconductivity*, 15(1):19–25, 2002. Accessed: 01.04.2025. doi:10.1023/A:1014018925270. (cited on Page 17)

[SKP+22] Raj Shree, Ashwani Kant Shukla, Ravi Prakash Pandey, Vivek Shukla, and Diksha Bajpai. Memory forensic: Acquisition and analysis mechanism for operating systems. *Materials Today: Proceedings*, 51:254–260, 2022. CMAE'21. Accessed: 16.04.2025. URL: https://www.sciencedirect.com/science/article/pii/S2214785321038633, doi:10.1016/j.matpr.2021.05.270. (cited on Page 16)

[SQL] SQLite Consortium. Sqlite documentation. Accessed: 18.04.2025. URL: https://www.sqlite.org/docs.html. (cited on Page 6)

[ST08] Alan Sexton and Hayo Thielecke. Reasoning about b+ trees with operational semantics and separation logic. *Electronic Notes in Theoretical Computer Science*, 218:355–369, 10 2008. Accessed:18.04.2025. doi:10.1016/j.entcs.2008.10.021. (cited on Page 8)

[TC09] Pallavi Tadepalli and H. Conrad Cunningham. Incrementally distributed b+ trees: approaches and challenges. In *Proceedings of the 47th Annual ACM Southeast Conference*, ACMSE '09, New York, NY, USA, 2009. Association for Computing Machinery. Accessed: 20.04.2025. doi:10.1145/1566445.1566451. (cited on Page 6 and 8)

[VDB+22] Janet Vorobyeva, Daniel Delayo, Michael Bender, Martin Farach-Colton, Prashant Pandey, Cynthia Phillips, Shikha Singh, Eric Thomas, and Thomas Kroeger. Using advanced data structures to enable responsive security monitoring. *Cluster Computing*, 25, 08 2022. Accessed: 05.04.2025. doi:10.1007/s10586-021-03463-5. (cited on Page xi and 15)

[VRLK+18] Alexander Van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. Managing non-volatile memory in database systems. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1541–1555, New York, NY, USA, 2018. Association for Computing Machinery. Accessed: 28.04.2025. doi:10.1145/3183713.3196897. (cited on Page 17)

[VTRC11] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*, pages 61–75, 01 2011. Accessed: 10.04.2025. (cited on Page 1, 22, 31, and 42)

[VVRI+22] Lukas Vogel, Alexander Van Renen, Satoshi Imamura, Jana Giceva, Thomas Neumann, and Alfons Kemper. Plush: A write-optimized persistent log-structured hash-table. *Proceedings of the VLDB Endowment*, 15(11):2895–2907, 2022. Accessed: 17.07.2025. (cited on Page 83)

[WH24] Daniel C. Worledge and Guohan Hu. Spin-transfer torque magnetoresistive random access memory technology status and future directions. *Nature Reviews Electrical Engineering*, 1(11):730–747, November 2024. Accessed: 01.04.2025. doi:10.1038/s44287-024-00111-z. (cited on Page 18)

[Wie18] Felix Wiedemann. Modern storage stack with key-value store interface and snapshots based on copy-on-write bε-trees. Master's thesis, Universität Hamburg, 2018. Accessed: 05.04.2025. (cited on Page xi, 1, 27, 28, 29, 33, 34, and 42)

[WKK+19] Yue Wang, Kyung-Mun Kang, Minjae Kim, Hong-Sub Lee, Rainer Waser, Dirk Wouters, Regina Dittmann, J. Joshua Yang, and Hyung-Ho Park. Mott-transition-based rram. *Materials Today*, 28:63–80, 2019. Accessed: 04.04.2025. URL: https://www.sciencedirect.com/science/article/pii/S1369702119303347, doi:10.1016/j.mattod.2019.06.006. (cited on Page 18)

[WWC+15] Chin-Hsien Wu, Po-Han Wu, Kuo-Long Chen, Wen-Yen Chang, and Kun-Cheng Lai. A hotness filter of files for reliable non-volatile memory systems. *IEEE Transactions on Dependable and Secure Computing*, 12(4):375–386, 2015. Accessed: 10.04.2025. doi:10.1109/TDSC.2015.2399313. (cited on Page 19)

[YKH+20] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association. Accessed: 13.04.2025. URL: https://www.usenix.org/conference/fast20/presentation/yang. (cited on Page 19)

[YWW+16] Jun Yang, Qingsong Wei, Chundong Wang, Cheng Chen, Khai Leong Yong, and Bingsheng He. Nv-tree: A consistent and workload-adaptive tree structure for non-volatile memory. *IEEE Transactions on Computers*, 65(7):2169–2183, 2016. Accessed: 18.04.2025. doi:10.1109/TC.2015.2479621. (cited on Page xi, 1, 23, 24, 31, and 42)

[ZD21] Baoquan Zhang and David HC Du. Nvlsm: A persistent memory key-value store using log-structured merge tree with accumulative compaction. *ACM Transactions on Storage (TOS)*, 17(3):1–26, 2021. Accessed: 25.08.2025. (cited on Page 1)

[Zip16] George Kingsley Zipf. *Human behavior and the principle of least effort: An introduction to human ecology.* Ravenio books, 2016. Accessed: 21.08.2025. (cited on Page 73 and 77)

[ZLCW20] Ling Zhan, Kai Lu, Zhilong Cheng, and Jiguang Wan. Rangekv: An efficient key-value store based on hybrid dram-nvm-ssd storage structure. *IEEE Access*, 8:154518–154529, 2020. Accessed: 29.08.2025. (cited on Page 21)

[ZMR+23] Fuheng Zhao, Zach Miller, Leron Reznikov, Divyakant Agrawal, and Amr El Abbadi. Autumn: A scalable read optimized lsm-tree based key-value stores with fast point and range read speed. *arXiv preprint arXiv:2305.05074*, 2023. Accessed: 25.08.2025. (cited on Page 1)

[ZMRA21] Zichen Zhu, Ju Hyoung Mun, Aneesh Raman, and Manos Athanassoulis. Reducing bloom filter cpu overhead in lsm-trees on modern storage devices. In *Proceedings of the 17th International Workshop on Data Management on New Hardware*, DAMON '21, New York, NY, USA, 2021. Association for Computing Machinery. Accessed: 05.04.2025. doi:10.1145/3465998.3466002. (cited on Page 16)

[ZSW21] Yuanhui Zhou, Taotao Sheng, and Jiguang Wan. Hbtree: an efficient index structure based on hybrid dram-nvm. In *2021 IEEE 10th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–6, 2021. Accessed: 25.04.2025. doi:10.1109/NVMSA53655.2021.9628870. (cited on Page xi, 1, 25, 26, 31, and 42)

[ZWC+20] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, Zhongdong Huang, and Jianling Sun. Fpga-accelerated compactions for lsm-based key-value store. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies*, FAST'20, pages 225–238, USA, 2020. USENIX Association. Accessed: 05.04.2025. (cited on Page xi and 15)

[ZWF+23] Xiaomin Zou, Fang Wang, Dan Feng, Tianjin Guan, and Nan Su. Rowe-tree: A read-optimized and write-efficient b+-tree for persistent memory. In *Proceedings of the 51st International Conference on Parallel Processing*, pages 1–11, 01 2023. Accessed: 23.04.2025. doi:10.1145/3545008.3545043. (cited on Page xiii, 24, and 31)

Statement of Authorship

Name:	______________	Surname:	______________
Date of birth:	______________	Matriculation no.:	______________

I herewith assure that I have written the present thesis independently, that the thesis has not been partially or fully submitted as graded academic work and that I have used no other means than the ones indicated. I have indicated all parts of the work in which sources are used according to their wording or according to their meaning.

I am aware of the fact that violations of copyright can lead to injunctive relief and claims for damages of the author as well as a penalty by the law enforcement agency.

Furthermore, I confirm that I am aware that the use of content (including but not limited to text, figures, images and code) generated by artificial intelligence (AI) in the thesis must be disclosed. In those cases I have specified the AI system used, I have marked the specific sections of the thesis where AI-generated content was used, and I have provided a brief explanation of the level of detail at which the AI system was used to generate the content. I also stated the reason for using the tools.

I assure that even if a generative AI system has been used, the scientific contribution has been made entirely by myself.

______________________	______________________
Signature	Place, Date