Buffered Persistence in B+ Trees (Abstract)*

Mingzhe Du, Michael L. Scott
{mdu5,scott}@cs.rochester.edu
Department of Computer Science, University of Rochester
Rochester, NY, USA

Abstract

Existing concurrent B+ trees for Non-Volatile Memory (NVM) persist every update immediately, incurring high persistence overhead and excessive NVM writes, even though such strict durability is often unnecessary. B+ trees are commonly used to index disk- and flash-based databases, where updates are typically buffered and persistence occurs on a millisecond timescale. We propose BD+Tree, a Buffered Durable B+ tree that defers and batches NVM writes. BD+Tree improves performance and reduces NVM wear-out by lowering persistence overhead and improving cache efficiency.

1 Introduction

Concurrent B+ trees are among the most widely used database indexing structures and have been extensively optimized. The advent of Non-Volatile Memory (NVM), with latency and bandwidth only a small constant factor worse than that of DRAM but with higher capacity and lower cost per byte, offers the intriguing opportunity to build persistent structures with both high performance in the absence of crashes and low post-crash recovery time. Existing B+ trees for NVM follow a strict correctness criterion—*Durable Linearizability (DL)*—which requires that (1) operations appear to occur in some total order consistent with sequential semantics and (2) each operation appear to take place *and to persist* at some single point in time between its call and return [1, 3].

Achieving DL requires immediately persisting each critical change using cache line write-back instructions—typically a flush followed by a fence. These instructions can require scores or even hundreds cycles to complete. Additionally, some flush instructions (e.g., clflush, clflushopt, and even clwb on certain Intel machines) invalidate cache lines, requiring subsequent accesses to miss in the last-level cache. Moreover, repeated flushes to the same cache line contribute to wear-out in NVM with limited endurance.

Our work is founded on two key observations. First, cache reuse plays a critical role in persistent applications, enhancing performance through latency hiding and reducing wear-out through in-cache write combining. Second, disk- and flash-based databases have relied on buffered persistence for decades, making DL an unnecessarily strong correctness criterion for a B+ tree serving as an index. What matters in post-crash recovery is consistency between the tree and the database; loss of the last few updates is appropriate if the corresponding transactions have been lost from the buffer cache (sync operations should of course persist both).

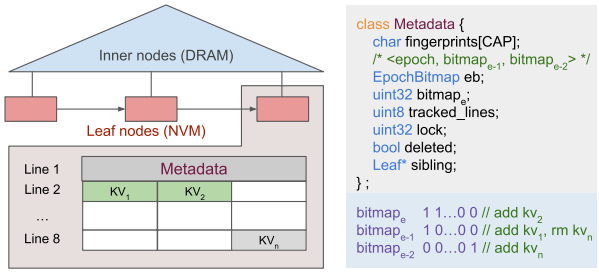

Figure 1: BD+Tree structure with 512B nodes (left), metadata structure (top right), and a bitmap example (bottom right).

We present a persistent B+ Tree (BD+Tree) that implements *Buffered Durable Linearizability (BDL)* [3]. This correctness criterion ensures that (1) all operations appear to occur in some total order consistent with sequential semantics, (2) each operation (in the absence of crashes) appears to take place at some single point in time between its call and return, and (3) post-crash recovery restores a consistent prefix of this pre-crash execution history. The implementation amortizes persistence overhead over large operation batches, optimizes cache reuse, and reduces NVM writes through in-cache write-combining.

2 BD+Tree Design

BD+Tree achieves BDL by implementing an epoch system and a specialized leaf node design. The epoch system serves two key functions: it (1) maintains a global clock that ticks every few milliseconds, segmenting execution into distinct epochs; (2) tracks and persists writes to NVM at epoch boundaries. Managed by a background thread, the epoch system ensures that maintenance and persistence overhead are off the critical path. At any given point in normal (crash-free) execution, epochs can be categorized as

- the *active* epoch e, in which new operations start. Operations in this epoch will not survive a crash that occurs before epoch $e+2$.
- the *in-flight* epoch $e-1$, in which threads can continue their ongoing operations but are prohibited from starting new ones. Operations in this epoch will not survive a crash that occurs before $(e-1)+2 = e+1$.
- *valid* epochs i, for $i \leq e-2$. Threads are not allowed to execute or initiate new operations in these epochs. Content in valid epochs has been securely persisted.

BD+Tree operations–including *insertion*, *deletion*, *lookup*, and *range query*–are designed to be fully contained with boundaries

*Full paper published in *Proceedings of the ACM on Management of Data 2*:6 (Dec. 2024), and presented at the SIGMOD/PODS International Conference on Management of Data, Berlin, Germany, June 2025.

HOPC '25, Portland, OR, USA

ACM ISBN 979-8-4007-2039-0/2025/07
https://doi.org/10.1145/3746238.3746245

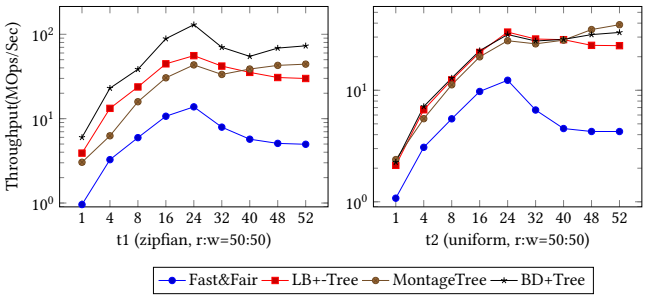

Figure 2: Throughput of persistent B+ trees (x-axis indicates the number of threads; y-axis scale is logarithmic).

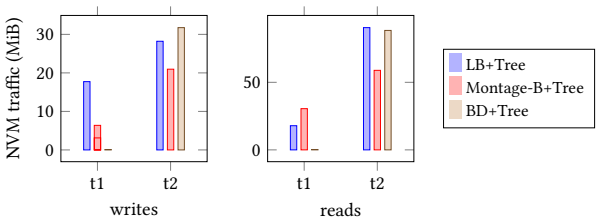

Figure 3: NVM traffic generated per million operations (in MiB). Experiments were conducted with 24 threads. Fast &Fair is excluded because of its fully persistence.

of a single epoch, ensuring no operation spans across multiple epochs. Writers coordinate with the epoch system to track and persist updates, while readers, which do not alter the tree, execute independently. Before progressing to the next epoch, the epoch system ensures that all *in-flight* operations are complete and updates are persisted. Waiting for *in-flight* operations to complete does not cause global quiescence, as new operations can begin in the *active* epoch. Once all in-flight data have been written back to NVM, the epoch system issues a fence and increments the epoch number. At this point the formerly *active* epoch becomes *in-flight*, and the *in-flight* epoch becomes *valid*. In the event of a crash, only operations from *valid* epochs are recovered.

To preserve BDL, updates from newer epochs must be applied to separate copies of data created in earlier epochs, ensuring the execution history remains intact and recoverable. In-place updates are permitted only for data created within the same epoch. BD+Tree employs a specialized design to track the execution epoch of each operation. As illustrated in Fig. 1, BD+Tree stores inner nodes in DRAM (recoverable from leaf nodes), and leaf nodes in NVM. Inner nodes retain the standard B+ tree structure, whereas leaf nodes adopt a specialized design: the first cache line stores metadata, and the remaining cache lines comprise key-value (KV) pairs. The metadata includes two key components to support BDL:

- **EpochBitmap** is a composite field comprising an epoch number and two bitmaps. The epoch number indicates when the node was last updated. If its value is e, the two bitmaps bitmap_{e-1} and bitmap_{e-2} indicate which KV pairs were visible in and before epoch $e-1$, respectively.
- **bitmap$_e$** is a bit array that identifies KV pairs that are visible in epoch e, where e is the epoch number stored in *EpochBitmap*. A KV pair is considered inserted/deleted into the node when the associated bit is set/cleared.

Each KV pair may have up to three copies, corresponding to three epochs. Only the most recent copy is visible, as determined by bitmaps. The global epoch number, node epoch number, and three bitmaps together determine the state of a leaf node. Consider the example shown in the bottom right of Fig. 1. If the global epoch number is 5, the same as the node epoch number stored in EpochBitmap, and $\text{bitmap}_e = \langle 1, 1, \ldots, 0, 0\rangle$, then there are KV pairs in slots 0 and 1. Similarly, $\text{bitmap}_{e-1} = \langle 1, 0, \ldots, 0, 0\rangle$, indicates that there was one KV pair, in slot 0, during epoch 4, and $\text{bitmap}_{e-2} = \langle 0, 0, \ldots, 0, 1\rangle$, indicates that there was one KV pair, in the last slot, two epochs ago. Together, the three bitmaps imply the execution history: a KV pair was inserted into slot $n-1$ in epoch 3 or earlier; in epoch 4, the KV pair in the last slot was removed, and a new KV pair was inserted into slot 0 (it may be a copy of the KV pair in slot $n-1$, clearing the last bit in bitmap_{e-1} ensures that only the new copy is visible); in epoch 5, a KV pair was inserted into slot 1. Upon a crash in epoch 5, the node would be restored to its state two epochs ago, with only the KV pair in slot $n-1$ remaining. KV pairs in slots 0 and 1 would be discarded. If instead the node epoch number were 4, indicating the last update to the node occurred one epoch ago, bitmap_e and bitmap_{e-1} would identify visible KV pairs from epochs 3 and 2, respectively. Only the KV pair in slot 0 should be recovered after a crash. Similarly, if the epoch number were 3, the recovery procedure would restore KV pairs in the first two slots, as indicated by bitmap_e.

3 Evaluation

We benchmarked BD+Tree against three persistent B+ trees: Fast & Fair (fully persisted, DL) [2], LB+Tree (hybrid, DL) [4], and Montage-B+Tree (hybrid, BDL) [5], using Memcached traces with varying workload characteristics. Fast & Fair and LB+Tree incur high persistence overhead and excessive NVM traffic due to strict persistence. Montage-B+Tree reduces these at a substantial memory cost, consuming 10× more DRAM than BD+Tree by keeping the bulk of the tree in DRAM, with only KV pairs in NVM. As shown in Fig. 2, under a high-locality scenario (t1), BD+Tree achieves a 1.5–2.4× speedup, by minimizing persistence overhead and maximizing cache efficiency. It also reduces NVM traffic by up to 99% through in-cache write combining (Fig. 3). Even under a challenging, uniformly distributed workload with a large working set (t2), which negates cache reuse benefits, BD+Tree remains highly competitive.

In the event of a crash, LB+Tree and BD+Tree need only a fast traverssal of the leaf layer in NVM to rebuild the inner tree. Montage-B+Tree, however, must scan the NVM heap to collect valid KV pairs—a significantly more costly process. BD+Tree counteracts the space overhead of maintaining KV duplicates with effective in-place updates and memory recycling, for a negligible increase in NVM usage. The advantage of BD+Tree becomes more pronounced as cache sizes grow, aligning well with trend in CPU design. Please see the full paper for details.

References

[1] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A Persistent Lock-Free Queue for Non-Volatile Memory. In *23rd ACM Symp. on Principles and Practice of Parallel Programming*. 28–40.

[2] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *16th USENIX Conf. on File and Storage Technologies*. 187–200.

[3] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under A Full-system-crash Failure Model. In *30th Intl. Symp. on Distributed Computing*. Springer, 313–327.

[4] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+ Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *Proceedings of the VLDB Endowment* 13, 7 (2020), 1078–1090.

[5] Haosen Wen, Wentao Cai, Mingzhe Du, Louis Jenkins, Benjamin Valpey, and Michael L. Scott. 2021. A fast, General System for Buffered Persistent Data Structures. In *50th Intl. Conf. on Parallel Processing*. 1–11.