

B-Trees in Modern SSD-Backed Systems: The Effects of Parallel I/O and Write Amplification

Thang Sian Kop
New York University
Email: ts5783@nyu.edu

Abstract—B-trees are still common in databases because they support fast searches, updates, and ordered scans. However, traditional B-trees were designed for hard disks, not modern solid-state drives (SSDs). SSDs can process many requests at the same time, but they also suffer from write amplification because flash memory cannot update data in place. These properties change how a B-tree should be designed. A modern B-tree should use asynchronous I/O, batching, caching, smaller update units, and better data placement. This paper explains how block-level parallelism and write amplification affect B-tree design. It also reviews systems such as PIO B-tree, FD-tree, Bw-tree, Bf-Tree, and compression-aware B⁺-trees. Finally, it presents a larger evaluation plan using up to one billion keys.

Index Terms—B-tree, B⁺-tree, SSD, NVMe, write amplification, parallel I/O, database index

I. INTRODUCTION

B-trees have been used in database systems for many years. They store many keys inside each node, which keeps the tree shallow and reduces storage accesses [1], [2]. B⁺-trees are especially useful because all records are stored at the leaf level. This makes point searches and range scans efficient.

The basic B-tree idea still works on modern hardware. The problem is that SSDs behave differently from hard disks. A hard disk is slow mainly because its mechanical head must move before reading data. An SSD has no moving parts. It contains several flash channels and can process many requests at once.

This sounds like an easy improvement. Replace the hard disk with an SSD and the B-tree becomes faster. In practice, that is only partly true. A normal B-tree search often reads one page, waits for the result, and then decides which page to read next. This serial behavior does not fully use the parallel hardware inside an SSD.

Writes create another problem. Flash memory cannot directly overwrite an old page. The SSD writes the new version somewhere else and later erases a larger flash block. During this cleaning process, valid data may need to be copied again. As a result, one small database update can cause much more physical writing than expected.

This paper argues that modern B-trees should not treat one full page as the only unit for reading, caching, logging, and writing. These operations have different needs. Reads benefit from high fanout and large pages, while updates often benefit from smaller units and buffered writes.

II. PARALLELISM IN MODERN SSDS

A. Why One Request Is Not Enough

Modern NVMe SSDs can process many I/O requests at the same time. They use multiple queues, flash channels, dies, and controllers. However, the SSD can use this hardware only when the database sends enough independent requests.

A single B-tree lookup contains dependencies. The database must read a parent node before it knows which child node to read. This makes one root-to-leaf search difficult to parallelize.

Fortunately, the root and upper levels of a B-tree are normally kept in memory. Most storage requests therefore occur at the leaf level. Requests for different keys can be processed independently.

A database can expose more parallelism by:

- processing many transactions at the same time;
- using asynchronous I/O;
- grouping several updates into one batch;
- prefetching pages during range scans;
- keeping internal B-tree nodes in memory.

Roh *et al.* proposed the PIO B-tree, which groups several I/O operations so the SSD can process them in parallel [4]. Their results showed large improvements for insertions and range scans. The exact speedup depends on the hardware, but the main lesson is still useful: SSD performance depends on the number of active requests, not only on the number of total requests.

Haas and Leis also showed that modern NVMe devices need asynchronous I/O and enough queue depth to reach high throughput [8]. A blocking program that waits after every request may leave most of the SSD unused.

B. The Cost of More Parallelism

More parallelism is not always better. A very deep queue can increase throughput, but it can also increase response time. An individual request may spend more time waiting behind other requests.

Large batches create a similar trade-off. A batch of 128 updates may use the SSD efficiently, but the first update in the batch may wait until the batch is full.

For this reason, queue depth and batch size should be tested instead of fixed at the largest possible value. The best setting depends on whether the system cares more about total throughput or individual request latency.

III. WRITE AMPLIFICATION

A. Database-Level Write Amplification

A small logical update may cause a full database page to be written. For example, suppose a B-tree uses an 8-KB page and changes a 32-byte value. Writing the full page creates:

$$\frac{8192}{32} = 256 \quad (1)$$

times more page traffic than the logical update.

The database may also write the same update to a write-ahead log. Page splits, checkpoints, and recovery files can add even more writes.

Database-level write amplification can be written as:

$$WA_{DB} = \frac{\text{bytes written by the database}}{\text{logical bytes changed}}. \quad (2)$$

B. SSD-Level Write Amplification

The SSD creates another layer of write amplification. Flash memory is written in pages but erased in larger blocks. Before erasing a block, the SSD may copy its remaining valid pages to a new location.

SSD-level write amplification can be written as:

$$WA_{SSD} = \frac{\text{bytes written to flash memory}}{\text{bytes received from the database}}. \quad (3)$$

The total physical cost can be approximated as:

$$WA_{total} \approx WA_{DB} \times WA_{SSD}. \quad (4)$$

This distinction matters. A design may reduce database writes while still creating poor flash placement inside the SSD. It may also reduce SSD cleaning while continuing to write too many full database pages.

IV. HOW B-TREE DESIGNS REDUCE WRITES

A. Buffering Updates

One method is to collect several updates before writing them to storage. The FD-tree keeps a small B⁺-tree at the top and stores updates in sorted lower levels [3]. This changes many small random writes into larger sequential writes.

Buffered trees reduce write cost, but reads become more complicated. A search may need to check several levels instead of following one clean B-tree path.

This is the same basic trade-off seen in LSM-trees. Writes become cheaper, but reads and background merging become more expensive.

B. Delta Records

The Bw-tree avoids changing a full page for every update. Instead, it creates small delta records that describe the change [5]. These deltas are later combined with the original page.

This works well when there are many small updates. However, long delta chains slow down reads. The system must decide when to combine the deltas into a new page.

Combining too often brings back full-page write costs. Waiting too long makes reads slower. There is no perfect merge point for every workload.

C. Smaller Cache and Update Units

Bf-Tree separates the in-memory cache unit from the disk page [10]. It uses smaller mini-pages in memory while keeping larger pages on storage.

This approach makes sense because a page can contain many records, but only a few records may be active. Caching and changing a small hot section avoids moving the entire page every time.

The main cost is complexity. The system needs extra metadata to track mini-pages and later combine them into disk pages.

D. Per-Page Logging and Compression

Qiao *et al.* showed that transparent storage compression can help reduce B⁺-tree write amplification [7]. Their design uses sparse pages and small local logs instead of rewriting complete pages for every change.

The pB⁺-tree uses a small append-only log for each page [9]. Updates first enter the local log. The page is rewritten only after enough changes have collected.

This reduces small random writes and can make recovery faster. Still, the method depends on careful crash recovery and storage support for sparse or compressed data.

E. Out-of-Place Writes

Traditional B-trees update the same logical page address. The SSD then performs its own out-of-place write internally. This means both the database and the SSD may move data.

Recent work on LeanStore suggests writing B-tree pages out of place at the database level [12]. The database can group pages with similar lifetimes and place them together. This can reduce the amount of valid data copied during SSD cleaning.

Interfaces such as Zoned Namespaces and Flexible Data Placement give software more control over where data are written [13]. The downside is that the database must handle more mapping and cleaning work.

V. COMPARISON

Table I summarizes the main approaches.

The strongest design depends on the workload. A read-heavy database may prefer a normal B⁺-tree with asynchronous I/O. A write-heavy database may benefit more from buffering, deltas, or an LSM-style design.

VI. LARGE-SCALE EVALUATION PLAN

Small experiments can give misleading results. If the complete B-tree fits in memory, the test mainly measures CPU and cache performance. It does not show how the tree behaves when it must read from the SSD.

A stronger experiment should use several input sizes:

- 1 million keys for correctness testing;
- 10 million keys for a small baseline;
- 100 million keys for cache-pressure testing;
- 500 million keys for an out-of-memory workload;
- 1 billion keys for long write and garbage-collection tests.

TABLE I
SSD-AWARE B-TREE DESIGN CHOICES

Method	Benefit	Cost
Async I/O	Uses SSD parallelism	Can increase latency
Batching	Groups small operations	Individual requests may wait
Update buffers	Reduces random writes	Adds merge work
Delta records	Avoids full-page updates	Slows reads if chains grow
Mini-pages	Caches only hot data	Needs more metadata
Per-page logs	Reduces page rewrites	Makes recovery more complex
Out-of-place writes	Improves flash placement	Needs mapping and cleaning

Using 16-byte keys and 112-byte values gives a logical record size of about 128 bytes. One billion records would therefore require about 128 GB before index and metadata overhead.

The experiment should compare page sizes of 4 KB, 8 KB, 16 KB, and 32 KB. It should also test queue depths of 1, 4, 16, 64, and 256.

Important workloads include:

- point reads;
- random updates;
- random inserts;
- sequential inserts;
- mixed read and write workloads;
- short and long range scans;
- uniform and skewed key access.

Each test should run long enough for the SSD to enter a steady state. A short test on an empty SSD may end before garbage collection begins. Write-heavy tests should run for at least several minutes after a warm-up period. Longer runs are better when measuring SSD write amplification [11].

The evaluation should report:

- operations per second;
- average and 99th-percentile latency;
- cache hit rate;
- database bytes written;
- SSD or NAND bytes written;
- page splits and merge work;
- CPU use;
- recovery time after a simulated crash.

The main comparison should include a normal B⁺-tree, an asynchronous B⁺-tree, a buffered or delta-based tree, and an LSM-tree such as RocksDB.

VII. DISCUSSION

The research shows that SSDs do not make B-tree design unimportant. They make the design choices more visible.

Parallelism helps only when the database creates independent work. One search path is still mostly serial. High

throughput comes from running many searches or updates at the same time.

Write amplification also cannot be solved at only one layer. Reducing full-page database writes is useful, but the SSD may still perform heavy internal copying. Better placement inside the SSD is useful, but it does not fix unnecessary log writes or page rewrites created by the database.

There is also a cost behind every improvement. Buffering delays writes. Delta records add read work. Per-page logs make recovery harder. Out-of-place storage needs cleaning and mapping.

For simple, read-heavy workloads, a traditional B⁺-tree may still be the best choice. For write-heavy workloads, separating the search page from the update and placement units is usually more effective.

VIII. CONCLUSION

Modern SSDs change B-tree design in two major ways. First, they reward many parallel and asynchronous requests. Second, they punish repeated small updates through database-level and SSD-level write amplification.

A modern B-tree should keep upper levels in memory, process leaf requests in parallel, batch updates carefully, and avoid rewriting full pages for small changes. Delta records, mini-pages, local logs, compression, and out-of-place placement are all useful methods.

The main trade-off is clear. Reducing writes usually adds metadata, background work, or read overhead. Increasing parallelism usually increases queueing and may hurt latency.

For this reason, B-tree designs should be evaluated with large datasets, long-running workloads, and measurements from both the database and the SSD. Small tests alone cannot show the full cost of garbage collection, cache misses, and background maintenance.

REFERENCES

- [1] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indexes," *Acta Informatica*, vol. 1, pp. 173–189, 1972.
- [2] G. Graefe, "Modern B-tree techniques," *Foundations and Trends in Databases*, vol. 3, no. 4, pp. 203–402, 2011.
- [3] Y. Li, B. He, R. J. Yang, Q. Luo, and K. Yi, "Tree indexing on solid state drives," *Proc. VLDB Endow.*, vol. 3, no. 1–2, pp. 1195–1206, 2010.
- [4] H. Roh, S. Park, S. Kim, M. Shin, and S.-W. Lee, "B⁺-tree index optimization by exploiting internal parallelism of flash-based solid state drives," *Proc. VLDB Endow.*, vol. 5, no. 4, pp. 286–297, 2011.
- [5] J. J. Levandoski, D. B. Lomet, and S. Sengupta, "The Bw-tree: A B-tree for new hardware platforms," in *Proc. IEEE Int. Conf. Data Engineering*, 2013, pp. 302–313.
- [6] A. Kakaraparthi, J. M. Patel, K. Park, and B. P. Kroth, "Optimizing databases by learning hidden parameters of solid state drives," *Proc. VLDB Endow.*, vol. 13, no. 4, pp. 519–532, 2019.
- [7] Y. Qiao, X. Chen, N. Zheng, J. Li, Y. Liu, and T. Zhang, "Closing the B⁺-tree vs. LSM-tree write amplification gap on modern storage hardware with built-in transparent compression," in *Proc. USENIX Conf. File and Storage Technologies*, 2022, pp. 69–82.
- [8] G. Haas and V. Leis, "What modern NVMe storage can do, and how to exploit it," *Proc. VLDB Endow.*, vol. 16, no. 9, pp. 2090–2102, 2023.
- [9] K. Huang, Z. Shen, Z. Shao, T. Zhang, and F. Chen, "Breathing new life into an old tree: Resolving logging dilemma of B⁺-tree on modern computational storage drives," *Proc. VLDB Endow.*, vol. 17, no. 2, pp. 134–147, 2023.

- [10] X. Hao and B. Chandramouli, “Bf-Tree: A modern read-write-optimized concurrent larger-than-memory range index,” *Proc. VLDB Endow.*, vol. 17, no. 11, pp. 3442–3455, 2024.
- [11] G. Haas, B. Lee, P. Bonnet, and V. Leis, “SSD-iq: Uncovering the hidden side of SSD performance,” *Proc. VLDB Endow.*, vol. 18, no. 11, pp. 4295–4308, 2025.
- [12] B. Lee, T. Ziegler, and V. Leis, “How to write to SSDs,” *Proc. VLDB Endow.*, vol. 19, no. 7, pp. 1469–1483, 2026.
- [13] NVM Express, “Flexible Data Placement: State of the Union,” *Future of Memory and Storage*, 2023.