The Write Path

Free Sample · One Chapter

A Field Comparison of LSM-Tree and B-Tree Storage Engines

Md Nasim Sheikh

www.nasimstg.dev

About this sample

This is one complete chapter from *The Write Path*, a short, data-driven field guide to how database storage engines actually write to disk. The numbers in this chapter are not estimates. They come from a reproducible benchmark, an open-source Docker harness that you can run on your own machine in a few minutes.

The full book traces a single write through both a B-tree (PostgreSQL-style) and an LSM-tree (RocksDB-style), explains where the bytes really go, and turns the analysis into a practical framework for choosing an engine for a given workload. The chapter reproduced here, "The Three Amplifications," is the analytical core: it defines the three ratios that govern every storage engine and reports the measured comparison between the two designs.

4

The Three Amplifications

To compare two storage engines independently of any particular machine, the useful quantities are ratios rather than rates: how much more does the engine do than the logical work the application requested? There are three such ratios, and together they form the analytical core of this book and the basis of the RUM conjecture.

4.1 Defining the amplifications

Definition | Write Amplification (WA)

The ratio of bytes physically written to storage to bytes logically written by the application. $\mathrm{WA} = \dfrac{\text{bytes written to device}}{\text{bytes written by application}}$. A WA of 10 means a 1 KB row costs 10 KB of device writes over its lifetime.

Definition | Read Amplification (RA)

The number of physical reads (or I/O operations) required to satisfy one logical read. $\mathrm{RA} = \dfrac{\text{I/Os per query}}{1}$. A point lookup that must probe four SSTables has, in the worst case, an RA of four.

Definition | Space Amplification (SA)

The ratio of bytes occupied on disk to the bytes of live logical data. SA $= \frac{\text{bytes on disk}}{\text{bytes of live data}}$. Obsolete versions, tombstones, and half-full pages all push SA above 1.

These three quantities cannot be minimized all at once. They are coupled, and that coupling is the central idea in storage-engine design.

4.2 The RUM conjecture

In 2016, Athanassoulis and colleagues stated the trade-off precisely: for **R**ead overhead, **U**pdate (write) overhead, and **M**emory (space) overhead, an access method can minimize at most two at the expense of the third. They named it the RUM conjecture [2].

Key Insight

Read, Update, Memory: choose two. Every storage engine is a particular bet on which of the three amplifications to sacrifice. A B-tree bets against *write* overhead under random inserts; a log-structured engine bets against *read* overhead, accepting that a key may live in several files at once. No configuration escapes the triangle; the most one can do is choose the corner that matches the workload.

This is why the question "which engine is faster?" is not well posed. The useful question is "which amplification can your workload afford to pay?", and that has a definite answer once the workload is known.

4.3 A measured comparison

Rather than assert ratios, this book includes a benchmark that can be run directly. The harness (in the `bench/` directory) drives an *identical* Go workload (the same key size, value size, operation counts, batch size, and per-phase durability) through two production embedded engines, so that the only variable is the data structure:

- **Pebble** [6], the LSM-tree storage engine behind CockroachDB; and
- **bbolt** [7], the copy-on-write B$^+$-tree behind etcd.

Running RocksDB against PostgreSQL directly would be unfair, because one is an embedded key–value library and the other a full SQL server, so the comparison uses two embeddable engines of the same shape as the designs in Chapters 2 and 3. Compression is disabled in Pebble so that the contest is purely structural. Table 4.1 reports one million random 16-byte keys with 100-byte values on the reference machine described in the appendix; the raw output is in `bench/results.json`.

Table 4.1. Measured results: 1,000,000 random keys, 100-byte values, batched durable writes (AMD Ryzen 7 5700X, Docker, 2026). Absolute numbers are hardware-specific; reproduce them with the included harness.

Engine	Write (ops/s)	Read (ops/s)	Mixed (ops/s)	Write amp.	Space amp.
Pebble (LSM)	**319,942**	66,205	**119,458**	**2.5**×	2.6×
bbolt (B⁺-tree)	27,982	**1,151,946**	47,916	44×	**1.9**×

Three results stand out, and all three appear in Figure 4.1 (the axis is logarithmic, because the numbers span two orders of magnitude).

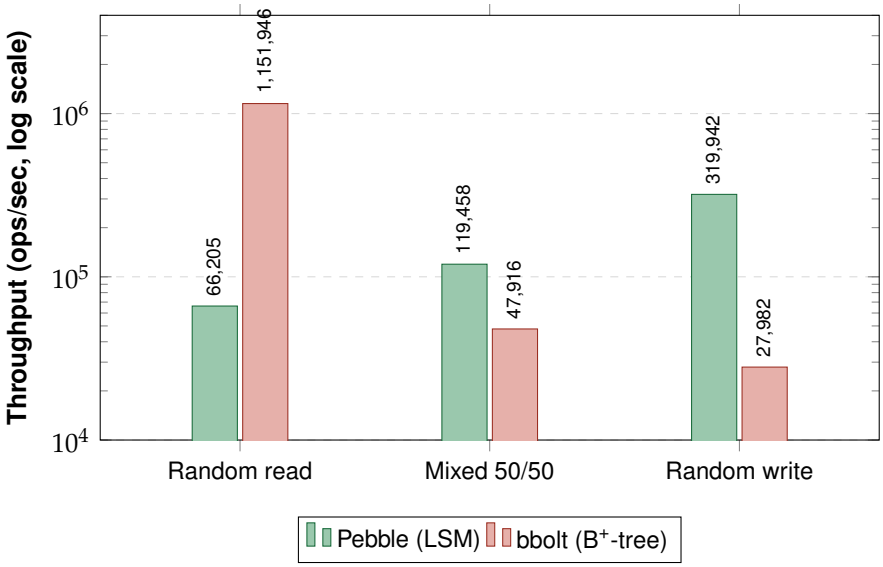

Figure 4.1. Measured throughput by workload (log scale). The B⁺-tree dominates point reads; the LSM-tree dominates writes and the mixed workload. Same hardware, same harness; see the appendix.

Reading Figure 4.1

Reads: the B⁺-tree is roughly *17*× faster (1.15 M versus 66 k ops/s). A memory-mapped B⁺-tree resolves a point read with one tree walk over cached pages, whereas the LSM engine may consult its memtable and several SSTables.

Writes: the LSM-tree is roughly *11*× faster (320 k versus 28 k ops/s). Each insert the B⁺-tree absorbs may split and rewrite pages, whereas the LSM-tree only appends.

Mixed (50/50): the LSM-tree leads (119 k versus 48 k), because the write half of the workload dominates the B⁺-tree's cost.

This is the read/write crossover the book is concerned with, now measured rather than assumed: below some write fraction the B⁺-tree's read advantage prevails; above it, the LSM-tree's write advantage prevails.

4.4 Write amplification is regime-dependent

The most surprising column in Table 4.1 is write amplification: the B⁺-tree wrote *44*× its logical data, the LSM-tree only *2.5*×. This contradicts the common shorthand that an LSM-tree trades high write amplification for write speed. The reality is more specific, and getting it right matters before a hardware decision.

Key Insight

There is no universal ranking of write amplification between B-trees and LSM-trees. It depends on the *workload regime*, the *scale*, and the specific *engine design*. The measurement above is the random-insert regime, in which the LSM-tree's central purpose, turning random writes into sequential appends, is most advantageous.

Three factors explain the numbers, each with a citable basis.

Why the B⁺-tree wrote so much. This was the original motivation for the LSM-tree. O'Neil and colleagues observed in 1996 that a B-tree "will effectively double the I/O cost of the transaction" to keep an index current under a high insert rate [1]. bbolt adds to this: it is a *copy-on-write* B⁺-tree, so each committed transaction writes modified pages to new locations and rewrites interior nodes and the freelist. Under random keys with an fsync per batch, that overhead compounds into the 44× measured here. In-place B-trees with a buffer pool, such as PostgreSQL and InnoDB, amortize random page writes far better,

but still pay for page-granularity writes, the WAL, and full-page writes [5] (Chapter 2).

Why the LSM-tree wrote so little here. At one million keys the dataset is only a few memtable flushes deep, so leveled compaction has barely begun to rewrite data. This is the LSM-tree at its most favorable.

Why that reverses at scale. As an LSM-tree grows, data migrates through more levels, and each level rewrites it again. WiscKey reports that I/O amplification in production LSM-trees "can reach a factor of 50× or higher" [3] for large, update-heavy datasets. The leveled LSM-tree's write amplification therefore grows with scale, while a B-tree's is roughly scale-independent. Running the harness at 100 k keys and then at 1 M shows the LSM figure beginning to climb (2.1× to 2.5×) while the B-tree's stays high: the first stretch of two curves that eventually cross.

Engineering Pitfall | "Write-optimized" refers to throughput, not bytes written

An LSM-tree maximizes write *throughput* by sequentializing I/O. Whether it also writes *fewer total bytes* than a B-tree depends on the regime: far fewer under random inserts (measured above), but potentially far more under heavy updates at large scale [3]. If the constraint is SSD endurance or per-write cloud billing, measure bytes-to-device on *your* workload, and do not assume either engine wins.

4.5 What this means for the RUM triangle

The measurement refines the RUM picture rather than overturning it. Both engines spend to keep data sorted; they differ in when, and in what currency. The B$^+$-tree pays eagerly, in random write amplification, to keep reads at an amplification near one, which is why it read 17× faster. The LSM-tree defers the cost into background compaction, keeping writes cheap now and paying in read amplification (several files per lookup) and, at scale, in write amplification later. Neither escaped the triangle [2]; each chose a different corner. The next chapter turns that choice into a decision you can defend.

Works cited in this chapter

[1] P. O'Neil, E. Cheng, D. Gawlick, E. O'Neil. "The Log-Structured Merge-Tree (LSM-Tree)." *Acta Informatica*, 1996.

[2] M. Athanassoulis et al. "Designing Access Methods: The RUM Conjecture." EDBT 2016.

[3] L. Lu et al. "WiscKey: Separating Keys from Values in SSD-Conscious Storage." USENIX FAST 2016.

[5] PostgreSQL Documentation, "WAL Internals" (`full_page_writes`).

[6] CockroachDB Pebble (LSM engine benchmarked here).

[7] etcd bbolt (B^+-tree engine benchmarked here).

Get the full book

The complete *Write Path* traces both write paths step by step, adds the B-tree and LSM-tree internals, the engine-selection framework, and a benchmarking protocol that will not mislead you, and ships the reproducible Docker harness behind every figure.

www.nasimstg.dev

Open-source benchmark harness included.