READ MODE · ANSWER FIRST

# How LSM Trees Work in RocksDB and LevelDB

An interactive, visual explanation of How LSM Trees Work in RocksDB and LevelDB, from setup through the complete system.

CHEAT SHEET · 6 ESSENTIAL IDEAS

## The whole story in 6 lines

How LSM Trees Work in RocksDB and LevelDB becomes easier to reason about when every stage is connected as one system.

1. Teach simultaneity at the front door: one logical write becomes one durable log record stream plus one in-memory sorted structure.
2. Separate the foreground write path from the background flush queue, then use engine controls to show why RocksDB tolerates more...
3. Make the learner see that L0 is overlap-heavy while L1+ are partitioned by key range.
4. Show probe order and why bloom filters matter before the learner sees any compaction nuance.
5. Teach file picking, merge ordering, drop rules, and why RocksDB can parallelize what LevelDB usually runs serially.
6. Close with a structural comparison: shared core on top, LevelDB and RocksDB capability envelopes below, tuned by workload.

STAGE 1 OF 8

# Setup


Welcome. Before we start, let us cover the big picture. An LSM tree is the storage engine behind databases like RocksDB and LevelDB. Think of it like a multi-layer filing system: new papers land on your desk first, and you sort them into drawers later.

An LSM tree stands for Log-Structured Merge tree. Instead of updating data in place, it always writes new entries first and merges them in the background. This makes writes very fast.

A WAL is a Write-Ahead Log. Before any change reaches memory, it gets written to this log file on disk. If the machine crashes, the WAL lets the database recover everything that was in progress.

A memtable is a sorted buffer that lives in RAM. New writes land here first so the database can serve fresh data without touching disk. When the memtable fills up, its contents get flushed to a file.

An SST file (Sorted String Table) is an immutable, sorted file on disk. Once written, it never changes. Compaction is the background job that merges older SST files together so reads stay fast. A bloom filter is a small lookup structure that can quickly say "this key is definitely not in this file," saving a lot of unnecessary disk reads.

Over the next six stages we will trace the full lifecycle: how data enters, how memory flushes to disk, how levels are shaped, how reads find keys, how compaction cleans up, and how LevelDB and RocksDB differ. Let us start with the very first moment a write arrives.

STAGE 2 OF 8

# Write Beat


02 WRITE BEAT

You just met the vocabulary. Now let us watch what happens the moment a write arrives at the LSM tree.

The writer thread groups keys into one write batch. Try changing the Batch control to see how many keys travel together.

That single batch fans out to two places at once. The WAL appends the data on disk while the mutable memtable inserts the same keys in sorted order in memory.

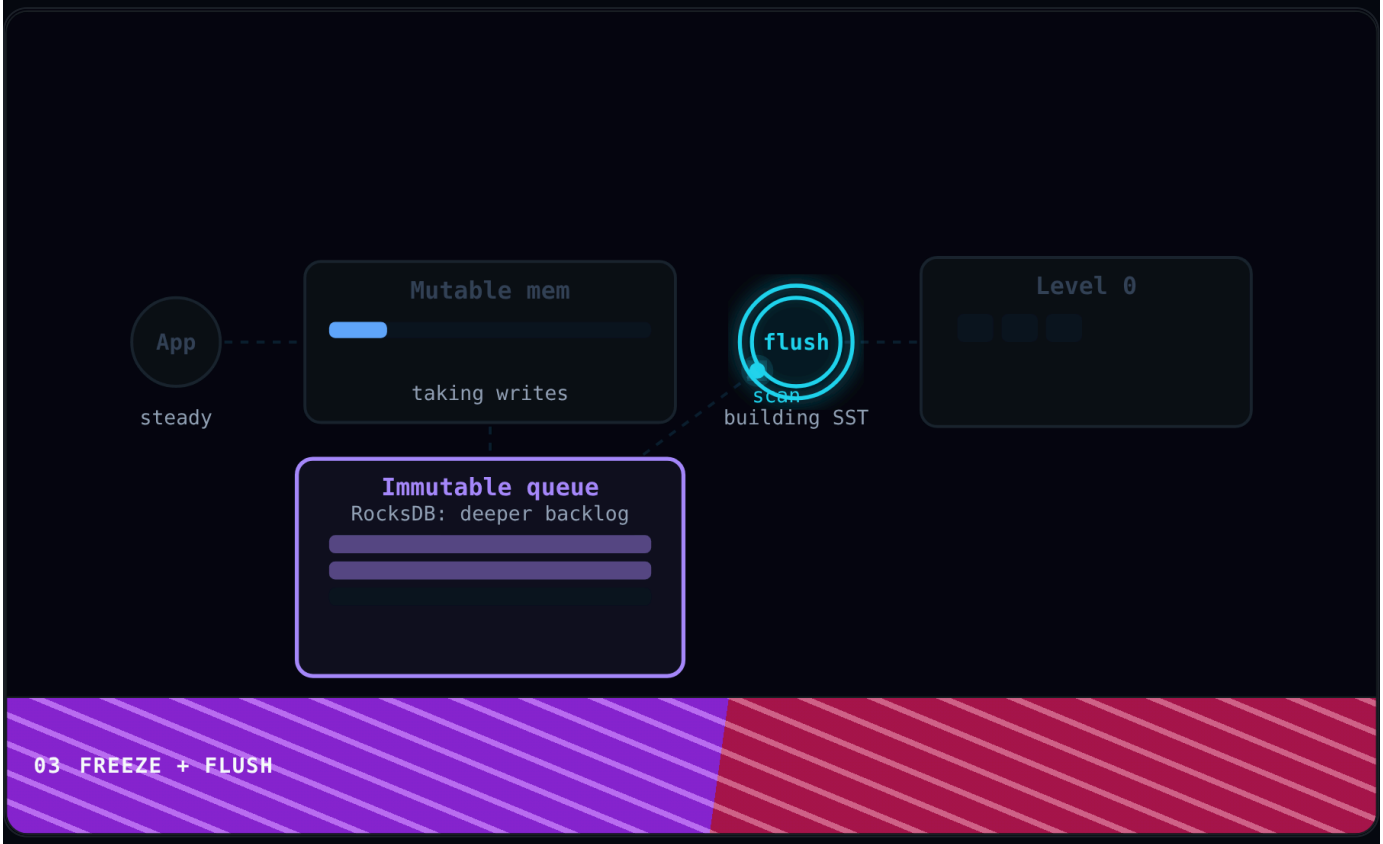
The WAL is your crash safety net. If the machine restarts, the database replays this log to recover every write that was in progress.

The memtable keeps keys sorted so that a later flush can produce a clean, ordered SST file without extra sorting work.

Once the write is visible in the memtable and durable in the WAL, the client gets its acknowledgment. Next we will see what happens when that memtable fills up.

STAGE 3 OF 8

# Freeze + Flush




The memtable from the last stage is still filling with writes. Eventually it hits its size limit and something has to give.

When the memtable is full, the engine freezes it. That frozen copy becomes immutable, and a fresh mutable memtable takes its place so new writes keep flowing.

Switch the Engine control between LevelDB and RocksDB. LevelDB allows only one immutable table at a time, while RocksDB can queue several before it stalls writers.

A background flush worker reads the frozen table in key order and writes a new sorted SST file into Level 0.

Because RocksDB tolerates multiple immutable tables, it can keep the mutable memtable accepting writes while older frozen tables drain in parallel.

Once the flush finishes, the engine discards the WAL range that the frozen table covered. Level 0 now has a fresh file. Next we will look at how those Level 0 files stack up.

# Level Shape


04 LEVEL SHAPE

Each flush from the previous stage dropped a new file into Level 0. Now you can see how those files pile up and form the shape of the tree.

Level 0 files can overlap because each flush captures whatever key range the memtable held. Increase the L0 runs control to add more overlapping files and watch the pile grow.

Overlapping Level 0 files are cheap to create, but they cost you at read time. The engine may need to probe several files just to answer one key lookup.

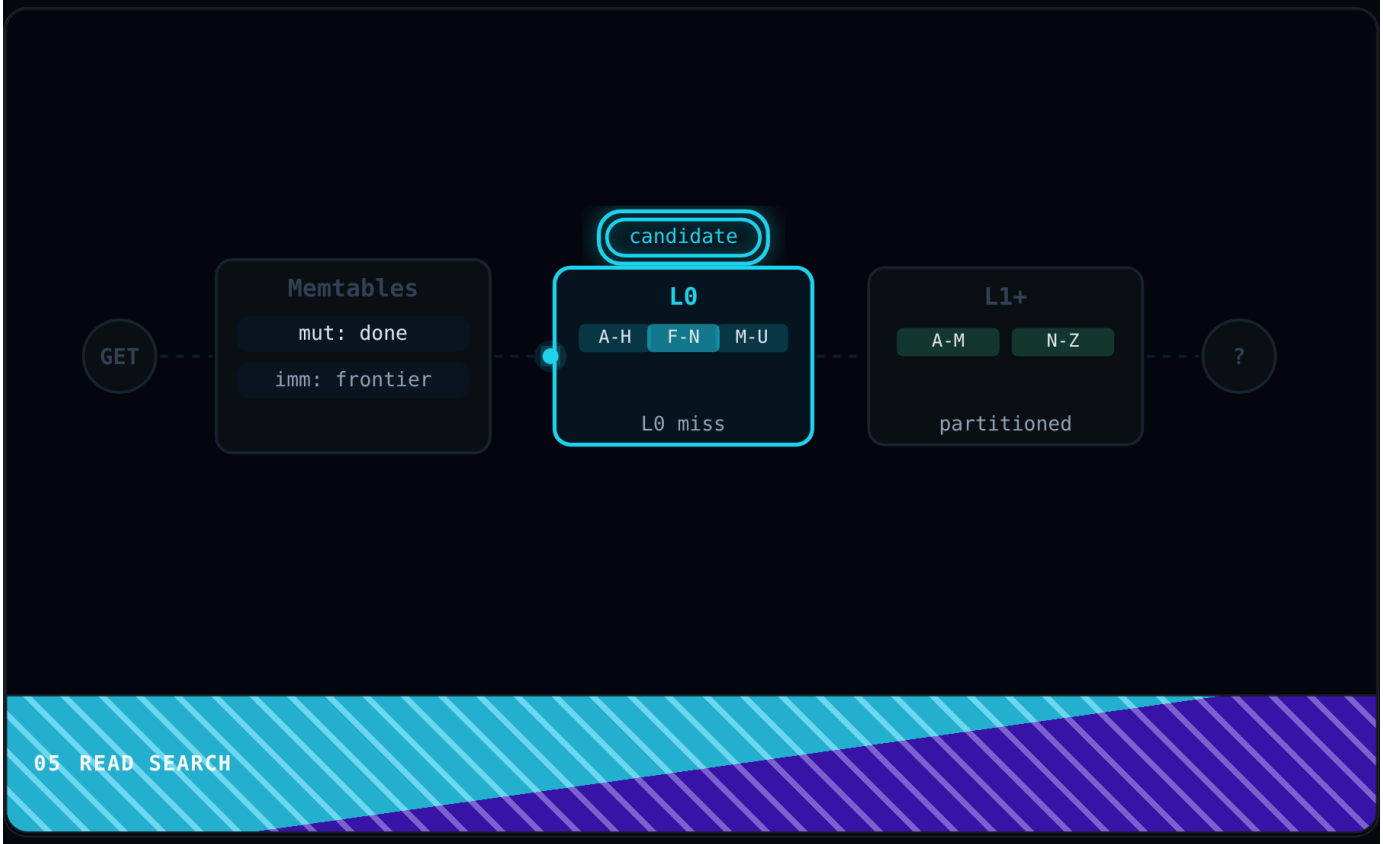
Compaction solves this by merging selected Level 0 files into Level 1, where every file covers a separate key range with no overlap.

Deeper levels follow the same rule: sorted, non overlapping, but larger and holding colder data.

So the top of the tree is write friendly and the bottom is read friendly. Next we will trace a read search through these levels to see why that shape matters.

STAGE 5 OF 8

# Read Search




05 READ SEARCH

You saw how levels are shaped. Now let us follow a single key lookup as it searches from newest data to oldest.

The engine checks the mutable memtable first because it holds the freshest keys. If your key was just written, the answer comes back immediately from memory.

If the memtable does not have the answer, the engine moves on to immutable memtables and then the overlapping Level 0 files, always checking newest first.

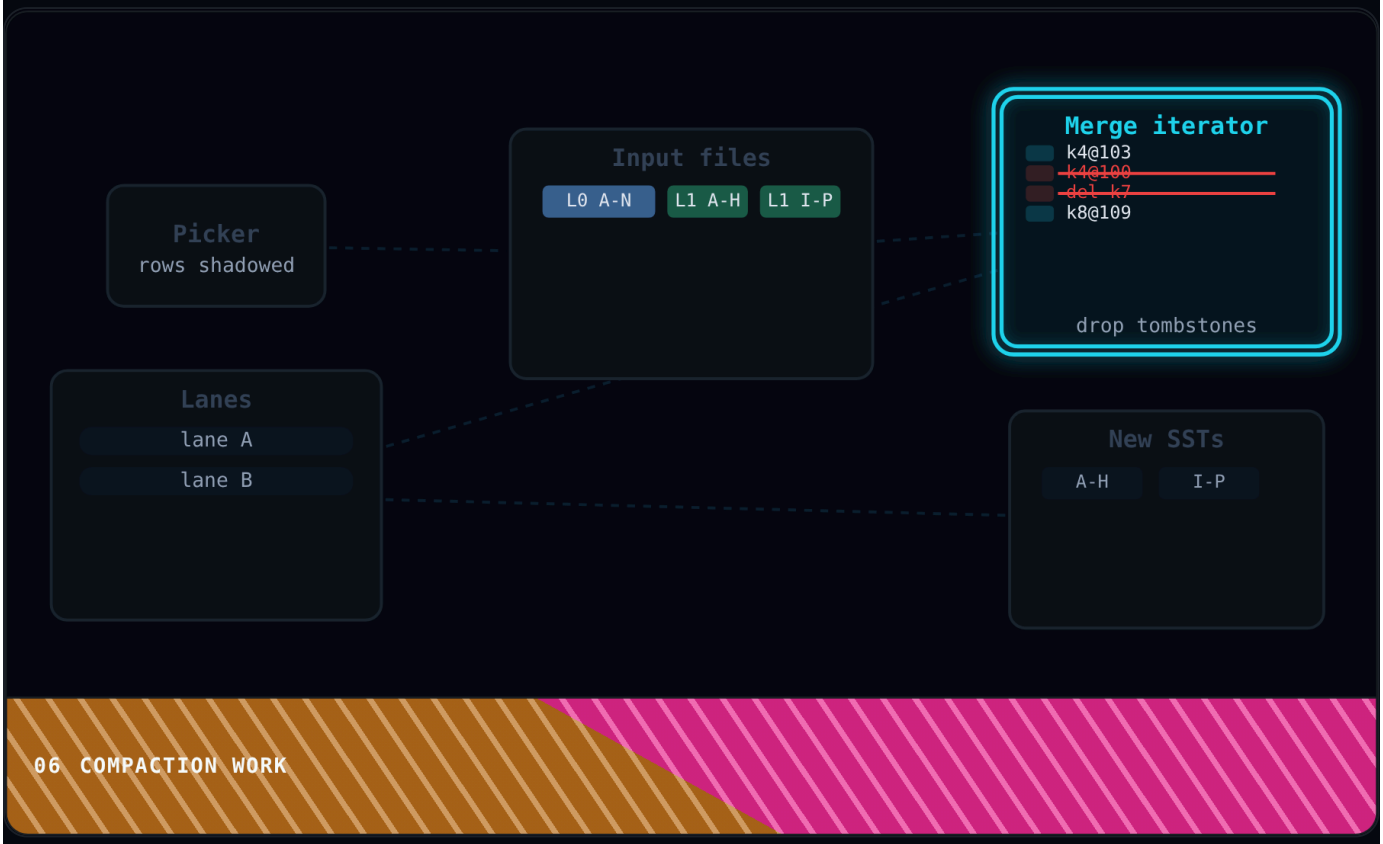
Bloom filters are the secret weapon here. They let a file say "this key is definitely not in me" without reading any data blocks, saving disk reads. Use the Lookup control to try a Recent key, an Older key, or a Miss.

In lower levels each file covers a separate key range, so the engine only needs to check one file per level.

The search stops at the first structure that proves the newest value, a tombstone, or a confirmed miss. Next we will see how compaction keeps this search path short.

STAGE 6 OF 8

# Compaction Work




06 COMPACTION WORK

Reads searched through multiple levels. Compaction is the background job that cleans up those levels so reads stay fast.

The compaction picker selects one Level 0 file and every overlapping Level 1 file that might hold the same keys.

The engine reads those input files in key order so newer versions shadow older ones in a single merge pass.

During the merge, tombstones and overwritten values are dropped when no active snapshot still needs them.

Switch the Engine control to compare. LevelDB runs one compaction lane at a time, while RocksDB can split the key range into parallel subcompaction lanes.

Fresh output files replace the old inputs, leaving fewer files and a flatter read path. Next we will zoom out to compare LevelDB and RocksDB side by side.

STAGE 7 OF 8

# LevelDB vs RocksDB


07 LEVELDB VS ROCKSDB

You have seen compaction in action. Now let us step back and compare the two engines that share this same LSM core: WAL, memtable, flush, SST levels, and compaction.

LevelDB stays intentionally small. It offers one database handle, lean configuration knobs, and a single leveled compaction path.

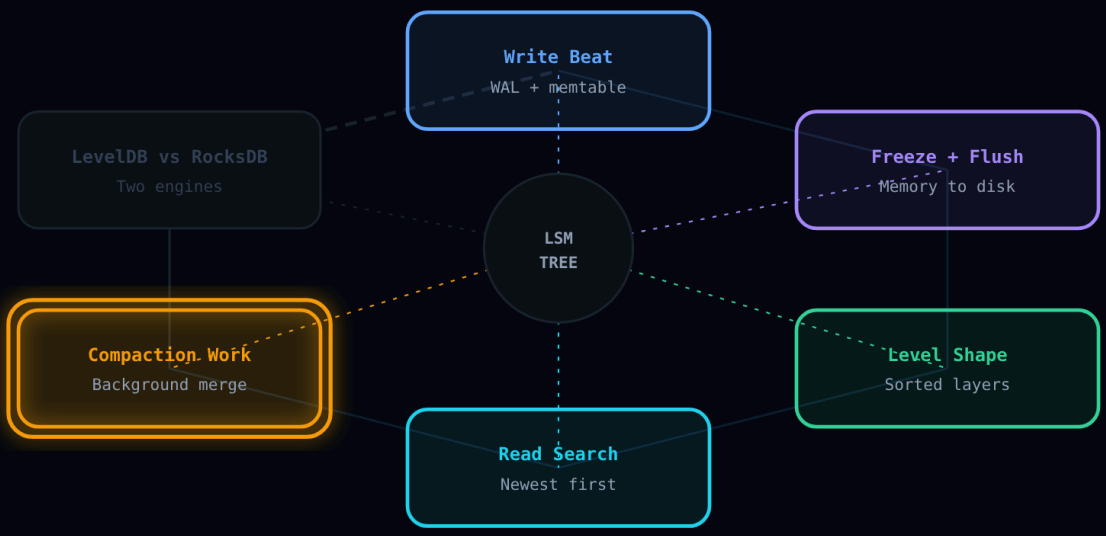
RocksDB wraps the same core with more write buffers, background threads, block caches, bloom filters, and rate limiters. Use the Focus control to spotlight each engine.

Under write heavy loads, RocksDB keeps more immutable memtables and compaction jobs in flight so the front end stalls less often.

Under read heavy loads, RocksDB gives you more tools to prune disk IO, while LevelDB remains the simpler, lower overhead baseline.

Think of LevelDB as the minimal leveled LSM and RocksDB as the production tuned branch of the same idea. In the Recap we will connect all six stages into one loop.

# Recap




08 RECAP

We started at the front door. A write batch lands in the WAL for crash safety and the memtable for fast lookups, both at the same time. That dual-write trick is what makes LSM trees durable without being slow.

When the memtable fills up, the engine freezes it and a background worker flushes it into a new Level 0 file on disk. Remember, RocksDB can queue several frozen memtables while LevelDB allows only one.

Level 0 files can overlap because each flush is independent. Lower levels are partitioned by key range so reads only need to check one file per level. That layered shape is the "tree" in LSM tree.

A point read walks from newest to oldest: memtable first, then immutable tables, then L0, then deeper levels. Bloom filters let most files say "not here" without touching data blocks, which is why they matter so much for read speed.

Compaction is the janitor. It picks overlapping files, merge-sorts them, drops old versions, and writes clean output. Without it, reads would get slower and slower as files pile up.

LevelDB keeps the design simple with minimal knobs. RocksDB layers on more write buffers, parallel compaction lanes, rate limiters, and richer bloom and cache options for production workloads.

And the cycle repeats. New writes feed the memtable, flushes create files, levels grow, reads search down, and compaction keeps the whole machine balanced. That is the complete LSM story.