EcoTune: Rethinking Compaction Policies in LSM-trees

Ryan Crosier, Daniel Silla, Will Garlington, Antony Ponomarev, Jackson Gilstrap

Based on: H. Wang, J. Qiu, F. Yuan, H. Zhang — Proc. ACM Manag. Data, Vol. 3, No. 3 (SIGMOD), 2025

Traditional view

- Tune compaction policy to trade write performance for read performance
- Assumes more compaction ⇒ slower writes, faster reads
- Only considers instantaneous query speed after compaction

Reframing

- Real-world writes use only a fraction of available I/O bandwidth on NVMe SSDs ⇒ more resources for compaction!
- Let's focus on average query performance after reserving resources for writes
- Optimize average **query throughput** over a **compaction round**

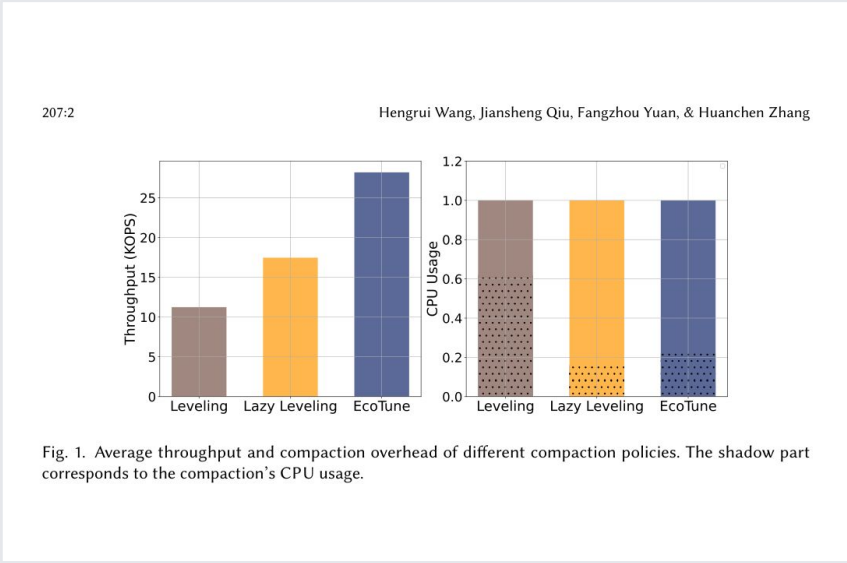
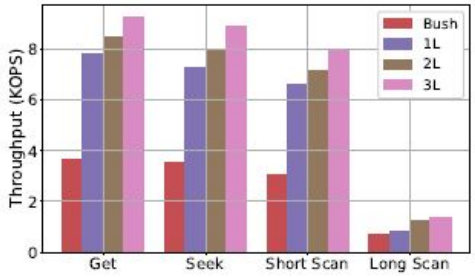


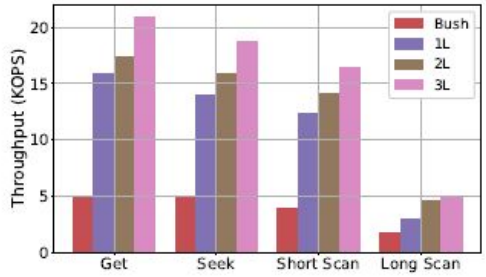
Fig. 1. Average throughput and compaction overhead of different compaction policies. The shadow part corresponds to the compaction's CPU usage.

Key observation: a "better" compaction policy may deliver lower ***average*** throughput if it burns shared CPU / I/O on compaction. We have more I/O to spend!

When does compaction improve query performance?



(a) Query Speed on NVMe SSD.



(b) Query Speed on Optane.

	Bush	1L	2L	3L
Get	1.06	1.009	1.007	1.003
Seek	1.08	1.03	1.02	1.01
Short Scan	1.11	1.07	1.07	1.07

Table 3. I/O Performance of Different Compaction Policies.

Fig. 3 (paper): merges matter most for long range scans; small-run merges mostly affect CPU overhead

Takeaways

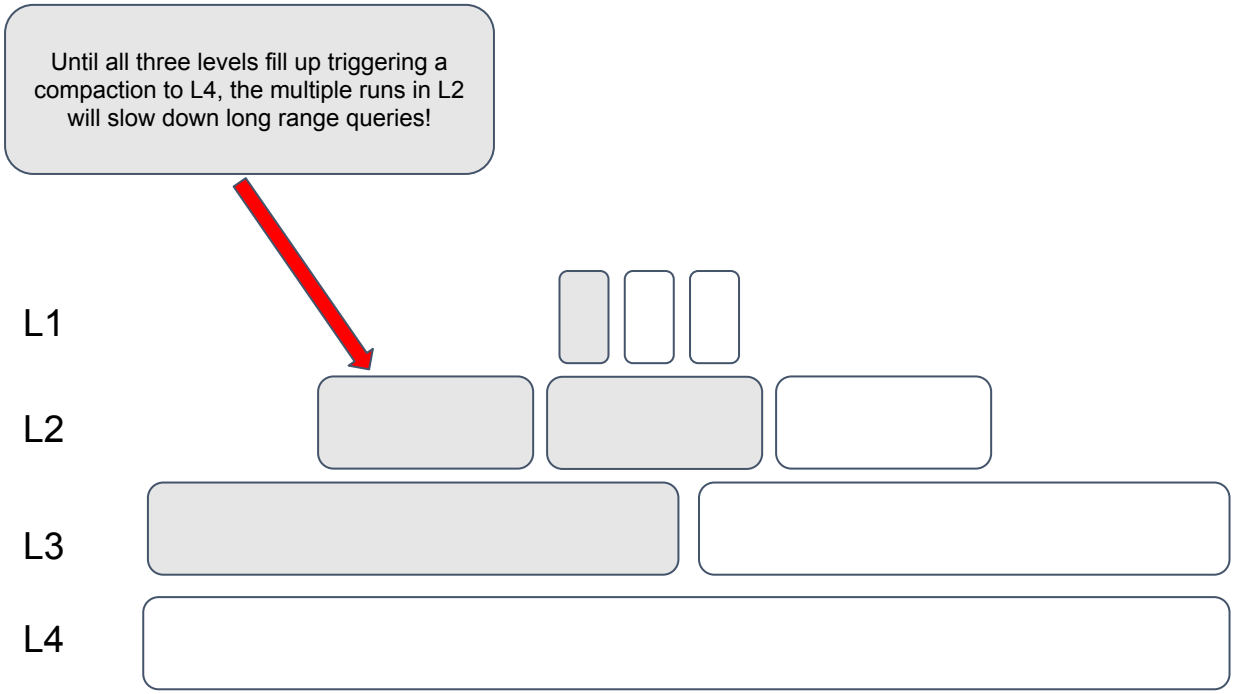
- With filters, Gets / Seeks / short scans are relatively robust to # of runs
- Long range scans are sensitive to the number of large runs
- Too many small runs inflates CPU work (filter probing)

Increasing the number of leveling levels (from 1L to 3L) improves query performance across all types **with diminishing returns**

Increasing the number of leveling levels decreases the I/O overhead because there are fewer total sorted runs

Optane plot pattern is identical to the NVMe SSD plot however achieved ~2x more throughput

Compactions as Investments



Investment view

- Compaction **Investment** = CPU + I/O time now
- **Return** = higher ***average*** query speed over a compaction round
- Earlier compaction ⇒ longer "benefit window" from sortedness before global compaction
- Thus, be aggressive early, and lazier near the end of the round

Implication: fixed "levels" (equal-sized runs) are a poor abstraction when optimal aggressiveness varies over time.

What is the Return on Investment of existing policies?

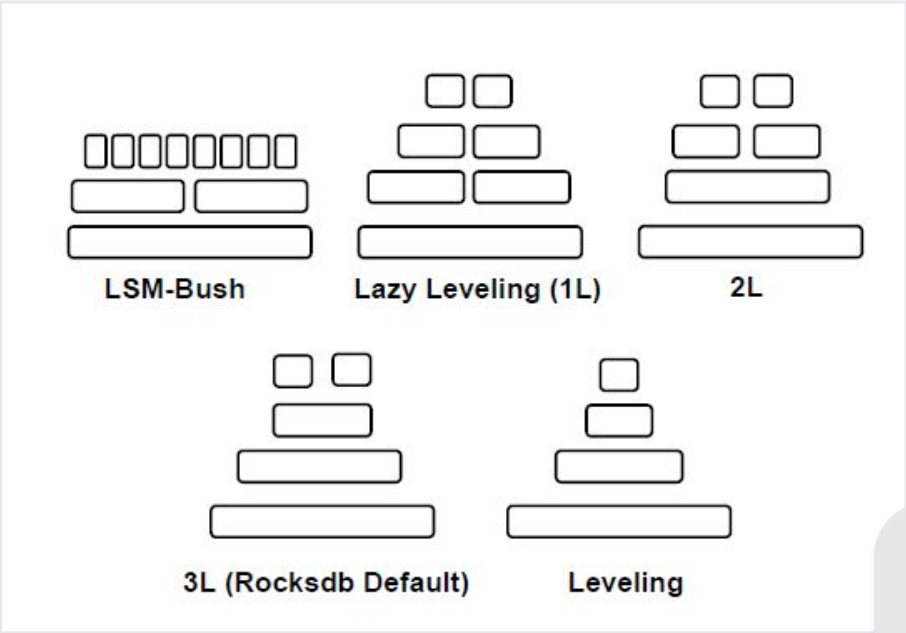


Fig. 2 (paper): Leveling, Lazy Leveling, multi-level variants, LSM-bush

Aggressiveness

Existing Policies

- Leveling: 1 run per level → high compaction cost
- Lazy leveling: tiering above, leveling at last level
- Tiering: multiple runs per level → more runs to probe

Prior tuning work still assumes fixed aggressiveness per “level”. EcoTune asks: given a fixed write/flush rate, how should we schedule merges to maximize average query throughput?

Three-level model: top / main / last

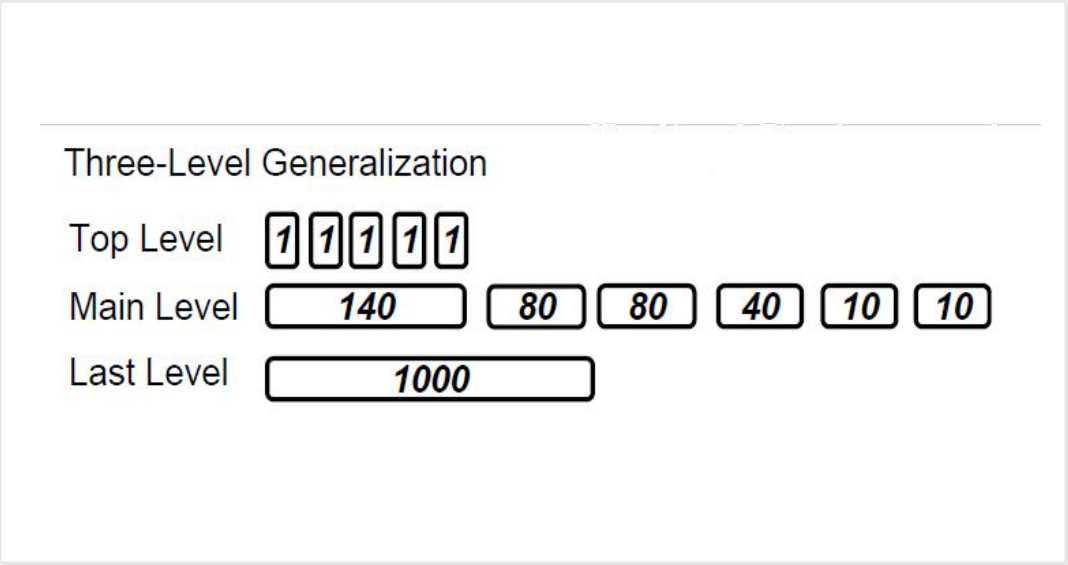


Fig. 4 (paper): time-varying run sizes → flatten "levels" into a main level

The Oracle: EcoTune algorithm computes how many main-level runs to keep over time (aggressive early → lazier later).

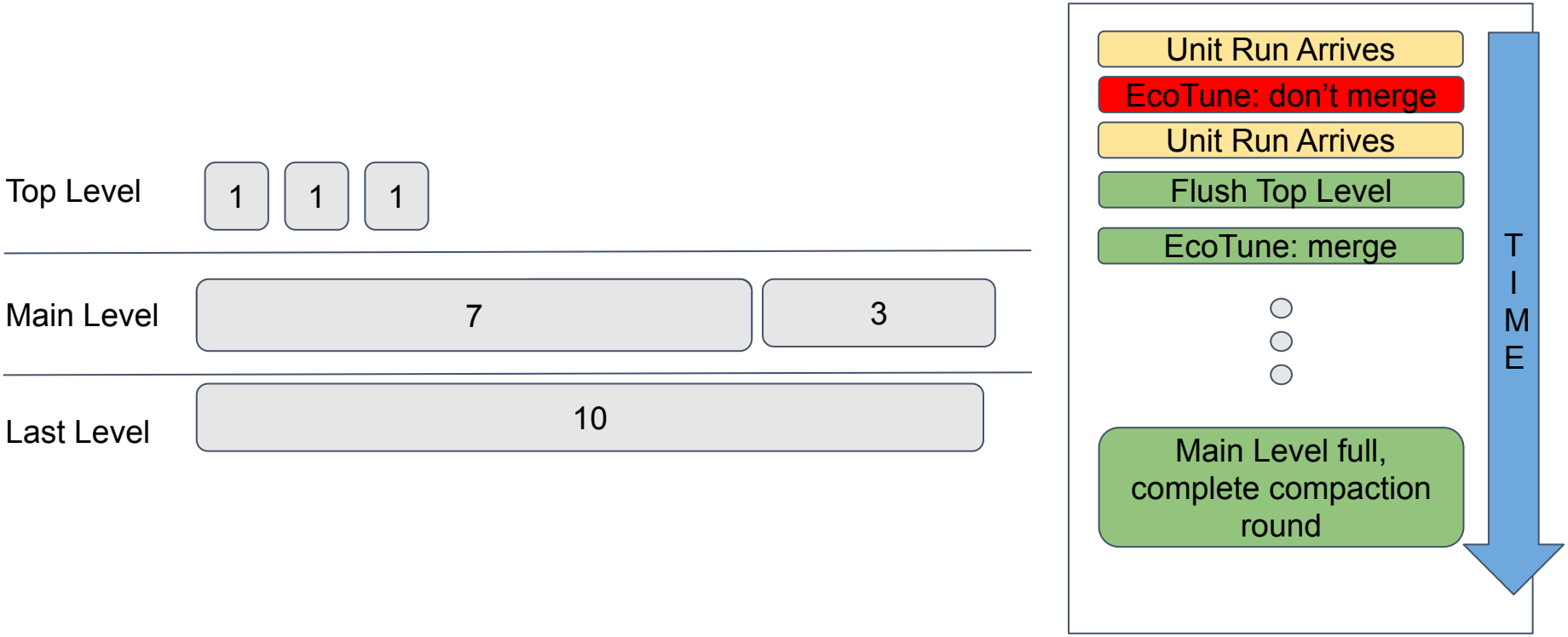
Logical levels

- Top level: SSD write buffer; avoid compaction here to save resources
- Main level: where #large runs dominates long-scan performance
- Last level: single run; global compaction checkpoint

Compaction

- We have an oracle that knows the optimal # of runs to compact at any given time to maximize average throughput
- A compaction round completes when main level is compacted into last level

Example Compaction



EcoTune algorithm: dynamic programming over a compaction round

Model (main level)

- Time is “unit runs” from top→main
- Query speed decreases with more runs
- Score = (query speed) × (time) — maximize total score

Recurrence (Simplified Intuition)

Core Idea: Decide how many unit runs (x) to merge into the next run, then solve subproblems on what remains.

$$f(e,c) = \max_x \left[f(e,x) + (T_v - x \cdot T_C) \cdot q(e+1) + f(e+1, c-x) \right]$$

- **EcoTune extensions:** Allows queries during compaction (β).
- **Pending runs:** Handles long-duration compactions effectively.

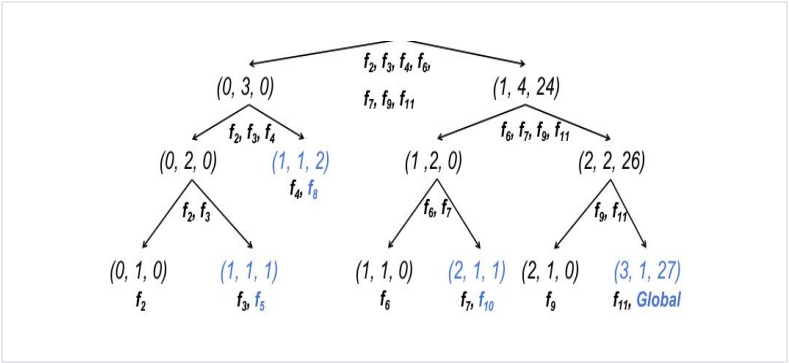


Fig. 6: DP problem partitioning

Resulting Output:

A schedule of main-level merges that adapts aggressiveness over time based on specific workload and hardware parameters.

Implementation sketch

- Keep flush + top→main compaction (TMC) provisioned to avoid write stalls (fixed write speed assumption)
- Use remaining threads as main-level compaction (MLC) threads, and also for queries when idle
- Measure compaction speed T_C on the machine; estimate β = query speed during MLC vs no-MLC
- Solve DP each round to produce a policy (main-level “layout vector”) for the next round
- Global compaction merges everything into the last level (checkpoint)

Why this matters: EcoTune explicitly treats compaction and queries as competitors for the *same* CPU/I/O pool, instead of optimizing an instantaneous RA bound.

Evaluation setup

Baselines

- Leveling (RocksDB default)
- Lazy Leveling
- Moose (workload-aware structure tuning)

Workloads

- Uniform mixes with varying long-range scan ratio (Seek + 100Next)
- Skewed YCSB (30% long scans, 70% point reads)

Hardware (examples)

- Optane SSD (P5800X) and NVMe SSD (D7-P5620)
- Multi-core AMD EPYC test machine (paper setup)

What we measure

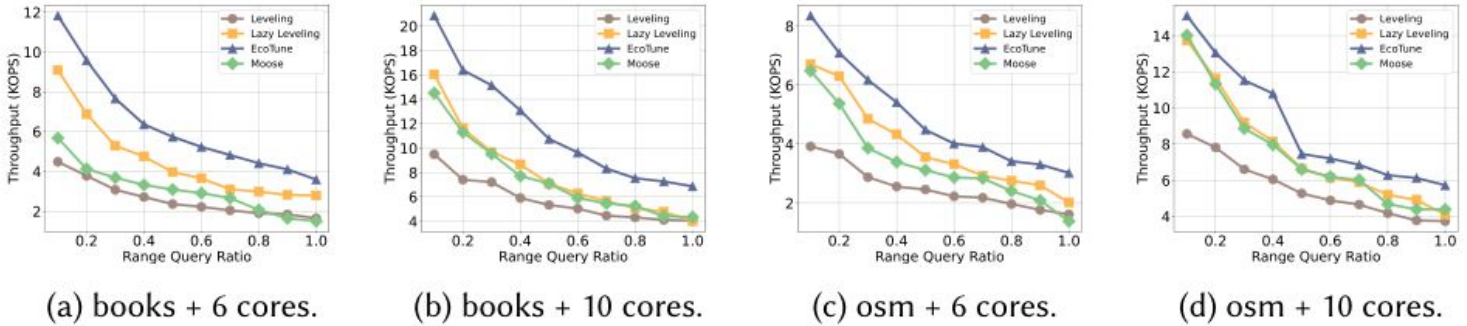
- Average query throughput over a compaction round
- Tail latency including queuing effects under different arrival rates
- Sensitivity to thread allocation, write speed, and parallelism

Why “compaction round”? Global compaction resets the structure, so policies are comparable round-by-round.

Next: results

EcoTune’s biggest gains show up when long-range scans are common and compaction would otherwise steal CPU/I/O from queries.

Results: average throughput vs long-range scan ratio (NVMe)



On books dataset, EcoTune maintains higher performance with varying parallelism

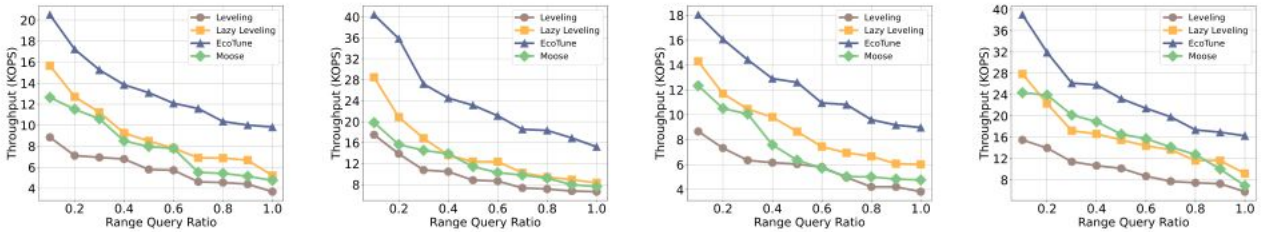
On osm dataset, EcoTune maintains higher performance but the difference is less pronounced

Fig. 7 (paper)

What to notice

- EcoTune is consistently the top curve across datasets and core counts
- EcoTune improves throughput by 30%–150% over leveling and 15%–80% over lazy leveling across ratios

Results: average throughput vs long-range scan ratio (Optane)



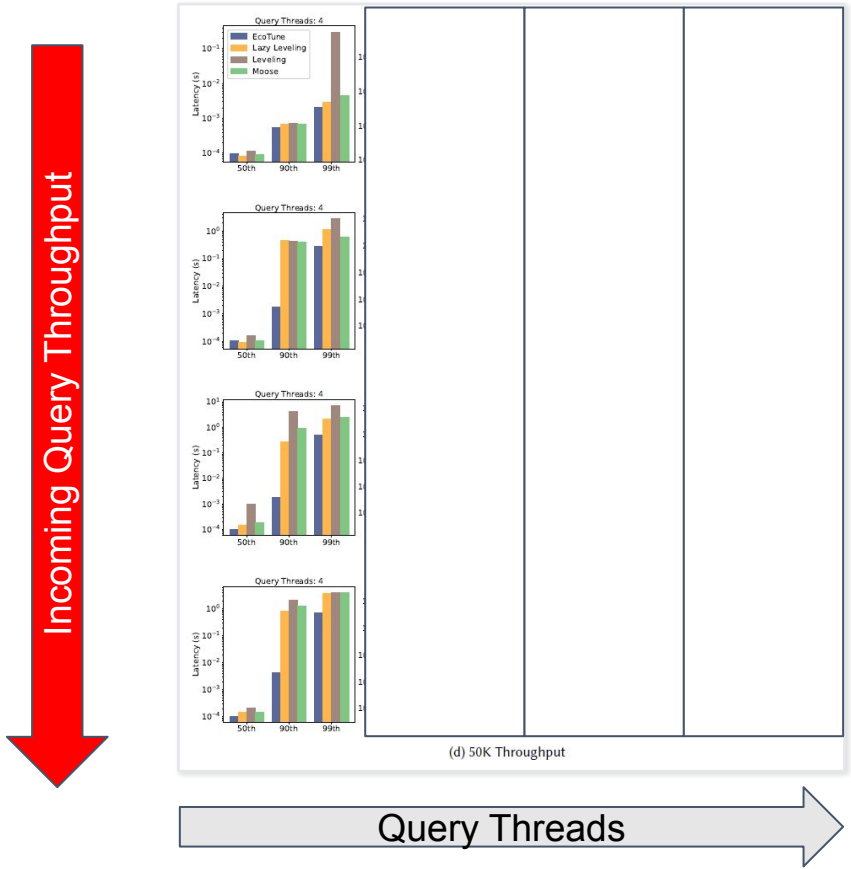
(a) books + 6 cores. (b) books + 10 cores. (c) osm + 6 cores. (d) osm + 10 cores.

Fig. 8. Throughput vs Long Range Scan Ratio on Optane SSD.

Why Optane shows bigger headroom

- Paper notes Optane's write bandwidth + low random-read latency enlarges the policy design space
- EcoTune benefits by spending less time compacting (more time serving reads) while controlling #large runs
- Across ratios, EcoTune reaches up to ~1.8× throughput vs the second-best baseline (paper summary)

EcoTune Improves latency (especially with lots of threads!)



Main message

- Deprioritized large compaction during heavy query loads
- Queuing latency dominates execution latency when compaction temporarily steals resources
- EcoTune's higher average throughput leaves fewer queued requests
- Paper reports up to 4 orders-of-magnitude lower latency vs leveling, and 3 orders vs lazy leveling
- Thread split (MLC vs query threads) affects latency even if throughput rises

Skewed workload (YCSB): EcoTune stays strong with caching effects

Setup

- 30% long-range scans, 70% point reads
- Write speed 100 MB/s (paper setting)

Result

- ≈15% higher throughput than lazy leveling
- ≈6× higher throughput than leveling
- Explanation in paper: aggressive compaction can invalidate cached blocks; EcoTune is aggressive early, not always

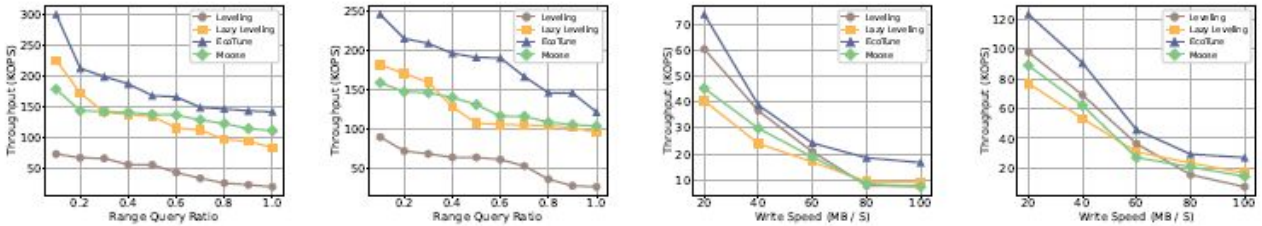
Hengrui Wang, Jiansheng Qiu, Fangzhou Yuan, & H

	Optane (KOPS)	NVMe (KOPS)
EcoTune	174.80	122.86
Lazy Leveling	157.85	109.90
Leveling	29.52	23.09
Moose	142.11	104.07

Table 9 (paper): average throughput (KOPS)

Even when cache effects matter, EcoTune's "invest early, then ease off" avoids perpetual churn while still controlling run counts.

Scaling: high parallelism and varying write speed



(a) books + Optane. (b) osm + Optane. (c) books + NVMe. (d) books + Optane.

Fig. 10. (a) and (b) are results of 64 threads (16 *MLC* threads and 48 query threads) experiments on Optane SSD. (c) and (d) shown query throughput under different write speed. We use 10 cores and the workload contains 30% long range scan.

- Queuing effect from high write speeds can decrease KOPS throughput
- At high parallelism, throughput can become SSD-bandwidth bound; EcoTune still leads (paper result)
- When write speed is low, compaction occupies a smaller fraction of the round; leveling improves, but EcoTune remains best by choosing the right aggressiveness
- Takeaway: EcoTune adapts across bottlenecks without separately modeling CPU vs I/O (paper note)

What EcoTune adds

- A throughput-first objective: maximize average query throughput over a compaction round
- A time-aware policy: aggressive early compactions, lazier later compactions (timing as “ROI amplifier”) from dynamic programming
- A simplified LSM view: three logical levels where only main-level run count is centrally optimized
- Measured gains in RocksDB: up to ~1.8× throughput vs second-best in sweep experiments and large tail-latency reductions due to less queuing
- Better caching
- Mitigates queuing effects

Open questions

- Query-stream-dependent compaction: adapt to arrival process to reduce under-utilization and queuing
- Tighter modeling of CPU scheduling overheads (especially at low core counts)

EcoTune re-solves its DP at every global compaction using the previous round's workload statistics. What happens when the workload changes significantly between rounds?

- EcoTune re-solves its DP and updates its compaction policy at every global compaction checkpoint using statistics collected from the previous round
  - A sharp shift in long range scan ratio would force EcoTune to run a fundamentally different compaction schedule than what the DP would have computed.
- Is EcoTune a general purpose solution or a highly tuned, overfitted, theoretical approach to specific stable workloads?

EcoTune re-solves its DP at every global compaction using the previous round's workload statistics. What happens when the workload changes significantly between rounds?

The authors state “We found that our EcoTune is also robust to workload shift. The solved policy is not sensitive to different workloads. For long range scan ratio varies from 30% to 80%, the optimal compaction policy remains unchanged. Therefore, workload shift can not easily violate the performance of EcoTune.”

The paper compares against leveling, lazy leveling, and Moose, but RocksDB has other compaction strategies like universal compaction. Why weren't those included?

- Section 4.2 of the paper states “[They] view the LSM-tree as a set of sorted runs, similar to RocksDB’s universal compaction.”
  - If this conceptual model already exists, what does EcoTune add over a well tuned universal compaction?
  - Why are comparisons in the paper not made to the most similar compaction method already present in RocksDB - wouldn’t this be the most effective comparison?

The paper compares against leveling, lazy leveling, and Moose, but RocksDB has other compaction strategies like universal compaction. Why weren't those included?

The authors didn't include universal compaction as a baseline because EcoTune's structural design actually mimics it. The authors explicitly state that they view the LSM-tree as a set of sorted runs, similar to RocksDB's universal compaction. Instead, they chose Leveling, Lazy Leveling, and Moose because these represent normal design space where sorted runs are restricted grouped into rigid physical levels. By comparing EcoTune against these rigid models, the authors can clearly demonstrate the advantage of their dp algorithm, which determines compaction aggressiveness based on the timing of a sorted run's creation rather than a fixed physical level

The paper fixes the memory write buffer at just 8MB throughout all experiments. How might EcoTune's advantage change with a larger cache?

- With a proportionally small write buffer, these experiments are bound by I/O costs which EcoTune is optimized for.
- With a larger buffer, the number of sorted runs from disk decreases and with it, EcoTune’s advantage.
    - Section 5.4 illustrates that under a skewed workload, EcoTune’s advantage shrinks from 80% to merely 15% over lazy levelling.
    - Could a larger buffer shrink EcoTune’s advantage even more drastically?

The paper fixes the memory write buffer at just 8MB throughout all experiments. How might EcoTune's advantage change with a larger cache?

Due to EcoTune's three-level design, the system should scale the size of the in-memory write buffer proportionally with the overall data size. By fixing the number of sorted runs in the top and main levels, and deliberately avoiding compaction at the top level, EcoTune's performance advantages remain intact and scales effectively even as the cache and data grow on a fixed write buffer size.

EcoTune views write speed as a constant variable defined by the user, when could this affect system performance?

- What happens when write speed becomes a bottleneck or the actual write throughput is considerably below the defined speed?
- Is there any effect on resource usage?

EcoTune views write speed as a constant variable defined by the user, when could this affect system performance?

- Threads for writing the the buffer and flushing the buffer are fixed, so resources are being wasted when reads are low and the CPU could actually become underutilized if the thread count is not enough to saturate the cores
- Potential benefits of making write speed a dynamic value -> what effects this would have on the complexity of the EcoTune algorithm?

Answers to questions in slide 17

- EcoTune re-solves its DP at every global compaction using the previous round's workload statistics. What happens when the workload changes significantly between rounds?
    - “We found that our EcoTune is also robust to workload shift. The solved policy is not sensitive to different workloads. For long range scan ratio varies from 30% to 80%, the optimal compaction policy remains unchanged. Therefore, workload shift can not easily violate the performance of EcoTune.”
- The paper compares against leveling, lazy leveling, and Moose, but RocksDB has other compaction strategies like universal compaction. Why weren't those included?
    - The authors didn’t include those other compaction strategies due to the close similarity universal compaction had with the way they were also viewing their model of the LSM tree. On the other hand the aforementioned compaction policies all restrict the size of their sorted runs, which provides a good benchmark for their DP algorithm against.
- The paper fixes the memory write buffer at just 8MB throughout all experiments. How might EcoTune's advantage change with a larger cache?
    - The advantage wouldn’t change since Ecotune doesn’t do any compaction at the top level, increasing the write buffer simply allows for less frequent flushes to Ecotune’s top level, all of EcoTune’s benefits come from the main level.