Analysis: The Log-Structured Merge-Bush & theWacky Continuum

`https://stratos.seas.harvard.edu/files/stratos/files/wackyandthebush.pdf`

Anwesha Saha

October 2, 2023

1 Formulae Used

1. Defines the capacity ratio r_i between levels i and i-1. Allows tuning ratio for largest level independently.

$$r_i = \begin{cases} T, & \text{if } 1 \leq i \leq L-1 \\ \frac{C \cdot T}{T-1}, & \text{if } i = L \end{cases} \tag{1}$$

2. Calculates capacity Ni of each level i based on overall data size N and ratios.

$$N_i = \begin{cases} N \cdot \frac{1}{C+1} \cdot \frac{T-1}{T} \cdot \frac{1}{T^{L-i-1}}, & \text{if } 1 \leq i \leq L-1 \\ N \cdot \frac{C}{C+1}, & \text{if } i = L \end{cases} \tag{2}$$

3. Calculates total number of levels L based on data size N, buffer size F, and ratios.

$$L = \left\lceil \log_T \left(\frac{N_L}{F} \cdot \frac{T-1}{C} \right) \right\rceil$$

4. Gives maximum number of runs a_i per level based on merge policy.

$$a_i = \begin{cases} T-1, & \text{if } 1 \leq i \leq L-1 \\ 1, & \text{if } i = L \end{cases} \tag{3}$$

5. Defines false positive rate pi for bloom filter at each level i. Sets decreasing rates for smaller levels.

$$p_i = \begin{cases} p \cdot \frac{1}{C+1} \cdot \frac{1}{T^{L-i}}, & \text{if } 1 \leq i \leq L-1 \\ p \cdot \frac{C}{C+1}, & \text{if } i = L \end{cases} \tag{4}$$

6. Average number of bits per entry M needed across all filters:

$$M = \frac{1}{\ln(2)^2} \cdot \ln\left(\frac{1}{p} \cdot \frac{C+1}{C^{\frac{C}{C+1}}} \cdot T^{\frac{T}{(C+1)\cdot(T-1)}}\right)$$

The next equation portrays cost complexity of M to highlight how the different parameters impact it. The core new finding is that as we increase the capping ratio C, the memory requirement decreases. The intuition is that increasing the capping ratio brings a higher proportion of the data to the largest level, which has the highest FPR and thus requires the fewest bits per entry.

$$M \in O\left(\ln \frac{T^{\frac{1}{C}}}{p}\right)$$

2 What is the problem this paper is solving?

1. Key-value stores based on log-structured merge trees (LSM-trees) are widely used but face challenges in balancing read, write, and memory costs as data grows.
2. Existing LSM-tree designs have fixed capacity ratios between levels, causing write cost to increase and read/memory benefits to diminish with data size.
3. **The goal of this paper therefore is finding optimal tradeoffs between read, write, and memory costs in LSM-tree based key-value stores.**

3 Why is it important?

1. There is a tradeoff between read, write, and memory costs based on merge policy greediness.
2. Smaller LSM-tree levels contribute significantly to write cost but negligibly to read/memory costs.
3. **LSM-trees are widely used but face scalability issues. Improving their efficiency is important for many applications.**

4 Why is it challenging?

There are many interdependent tuning knobs which are difficult to optimally tune. The task is mainly challenging because:

1. Problem 1: In existing LSM-tree designs, there is a fundamental trade-off between the costs associated with reads, writes, and memory usage. Optimizing one of these aspects often leads to degradation in the others. For instance, to enhance point reads' speed, you must either allocate more memory to expand Bloom filters and minimize read I/Os due to false positives or increase write costs by aggressively compacting data to reduce the number of runs with false positives. These trade-offs are inherent, making it challenging to achieve optimal performance universally. The design and tuning of LSM-trees must be tailored to the specific workload and memory constraints.

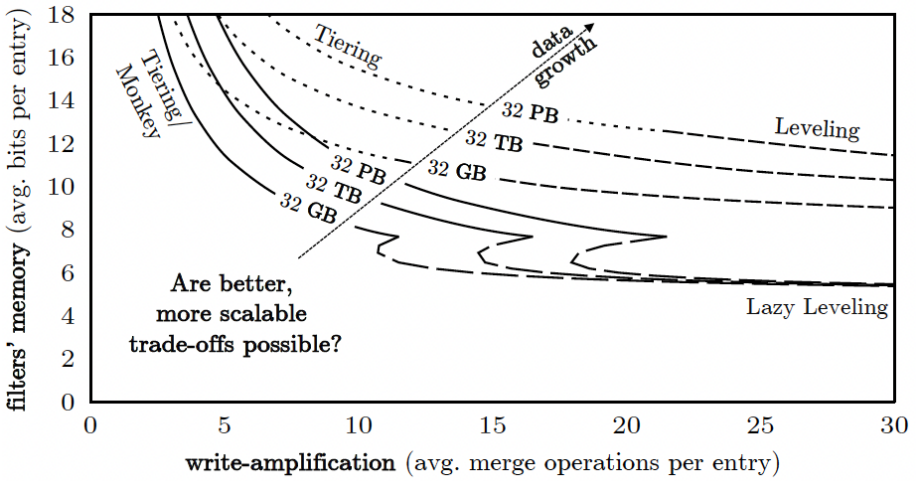

Figure 1: tradeoff graph from paper

2. Problem 2: The trade-off dilemma worsens as the volume of data grows. This is illustrated in the figure in the paper by plotting different LSM-tree variants, each with its trade-off characteristics. These variants include widely used approaches like Tiering and Leveling, as well as newer designs like Tiering/Monkey and Lazy Leveling that optimize memory usage in Bloom filters. The x-axis represents write cost as write-amplification (WA), measuring how many times a data entry gets rewritten due to compactions. The y-axis represents memory usage as the number of bits needed per entry across all Bloom filters to control the expected number of false positives. This allows us to visualize trade-offs while keeping point read costs constant. As shown in the figure, as the data size increases, the Pareto frontier of best trade-offs (represented by the Tiering/Monkey and Lazy Leveling curves) shifts outward, indicating worse trade-offs. To accommodate substantial data growth, you either need to accept significantly higher write amplification or allocate substantially more memory per entry for Bloom filters. This deterioration in trade-offs is even more

pronounced for traditional Tiered and Leveled designs.

5 Describe the paper gist in 1-2 sentences.

Wacky adapts LSM-tree structure to find optimal read/write/memory tradeoffs for different workloads and data sizes.

6 What is important to remember? What did we learn?

Varying capacity ratios across levels enables better scalability. Controlling merge operations is important.

7 Solution description: Explain how the solution works

Uses analytical models to search design space. Grows capacity ratios for smaller levels to merge lazily. Controls largest level capacity. Replaces bloom filters with hash tables for small levels.

8 Describe the experimental setup

Implemented in RocksDB. Compared against Leveled, Tiered, Lazy Leveled LSM-trees. Evaluated read/write throughput, scalability.

9 Summarize the main results

There are 2 cases for the main results: Outperforms other designs across range of workloads. Scales better with data size and memory due to LSM-Bush.

10 When doesn't it work?

Not as optimized for workloads with many range queries.

11 What assumptions does the paper make and when are they valid?

Assumes uniform random workload distribution. Focused on read/write/memory costs, not latency.

12 Related work: List the main competitors and describe how they differ

13 Follow-up research ideas

Integrate with adaptive indexing structures like Adaptive Radix Tree. Expand design space, e.g. optimize for latency, skew, temporal locality. Machine learning to predict optimal configurations.