The Log-Structured Merge-Bush & the Wacky Continuum

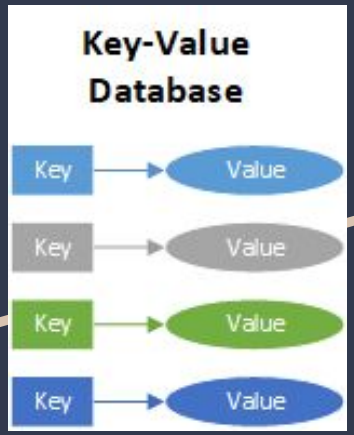
Sean Chun and Kaijie Zhou

Presentation Overview

1. Introduction
    a. Problem (WPI)
    b. Solution(Wacky with LSM-Bush)
2. Background
3. Capped Lazy Leveling
4. Log Structured Merge Bush
5. Wacky Continuum
6. Experiment Results
    a. Evaluations
7. Related Works
8. Conclusion
9. Q&A

Introduction

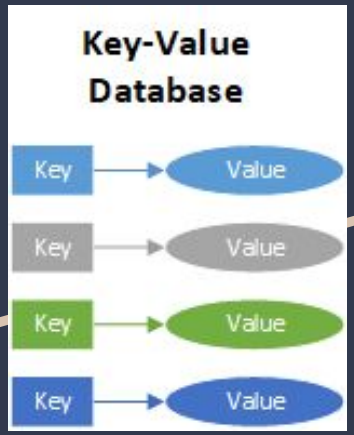
Key-Value Store (KVS)



- **What is a Key Value Store?**
- **What are some issues with KVS?**

Introduction

Key-Value Store (KVS)



Some Issues that current Key-Value Store is facing:

- Increase in write proportions in many application
- Large raw data cannot be all fit into memory
- Solid State Drive (SSD) make write operations more costly than read operations

Introduction

Problem

Write and Point Read Intensive Workloads (WPI)

- Write intensive workloads with mostly only point reads workload
- Need to rapidly ingest write and enable fast application read

Existing designs keep capacity ratio fixed between levels.

Introduction

Existing Design to Keep fixed Point Read Cost

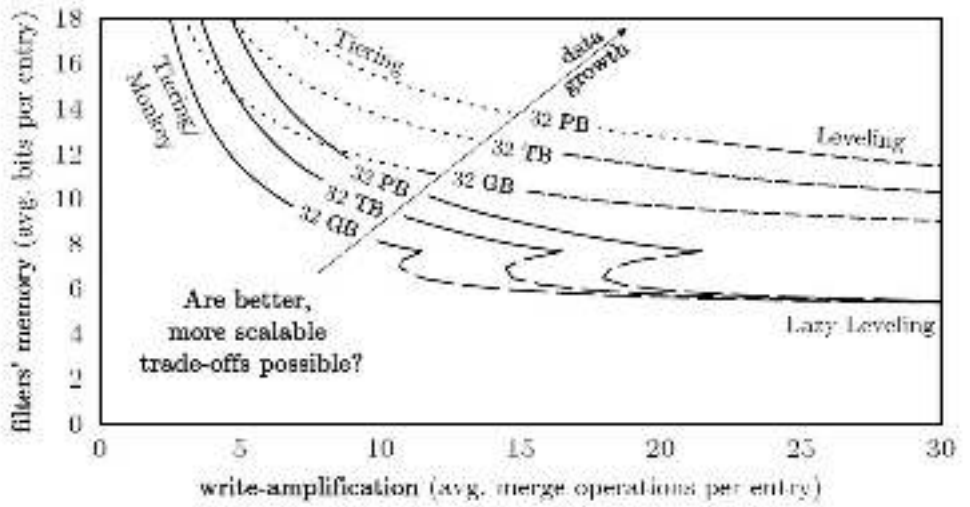


Figure 1: Existing designs need to sacrifice increasingly more write performance or more memory to hold point read performance fixed as the data grows.

Introduction

Solution

The Wacky Continuum

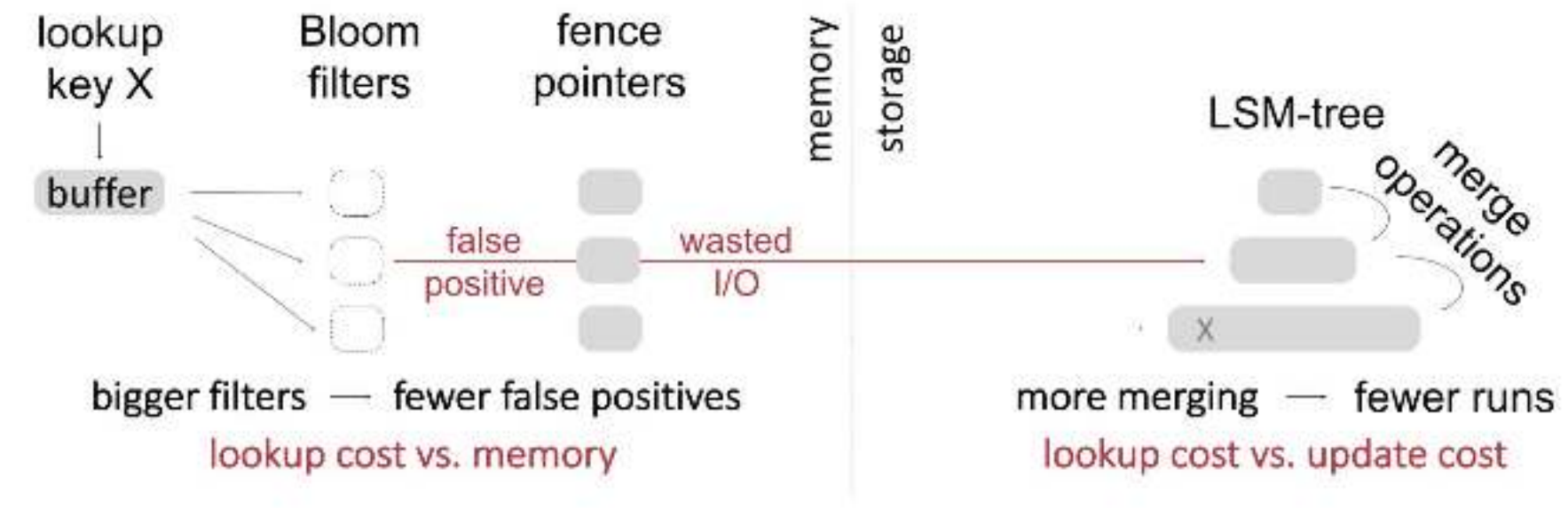
- LSM-Bush with whole spectrum of merge policies, from laziest to greediest.
- Provide analytical model that allow searching for best merge policy for a given workload
- With some range read, use CLL to adjust the capacity ratio of largest level
- With no range read, increase capacity ratio between smaller level and use LSM-Bush

LSM-Bush

- Setting increasing capacity ratios between smaller pairs of adjacent levels, which allow for more runs before merging

Background

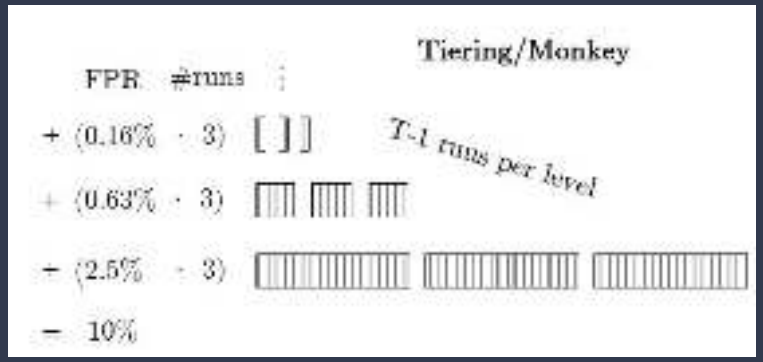
Bloom Filters and Fence Pointers

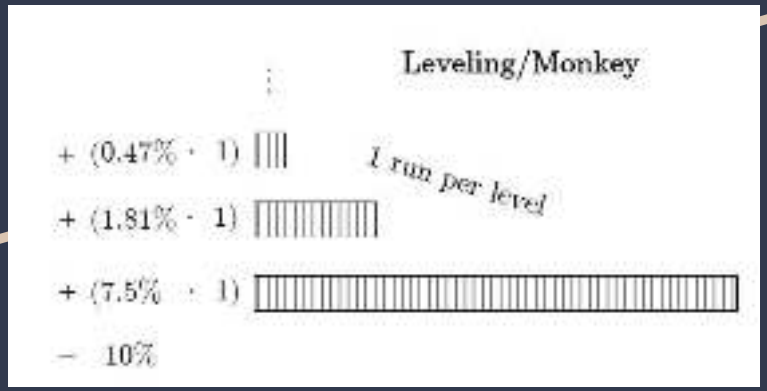


Background

Difference between Tiering and Leveling?

Background Continued





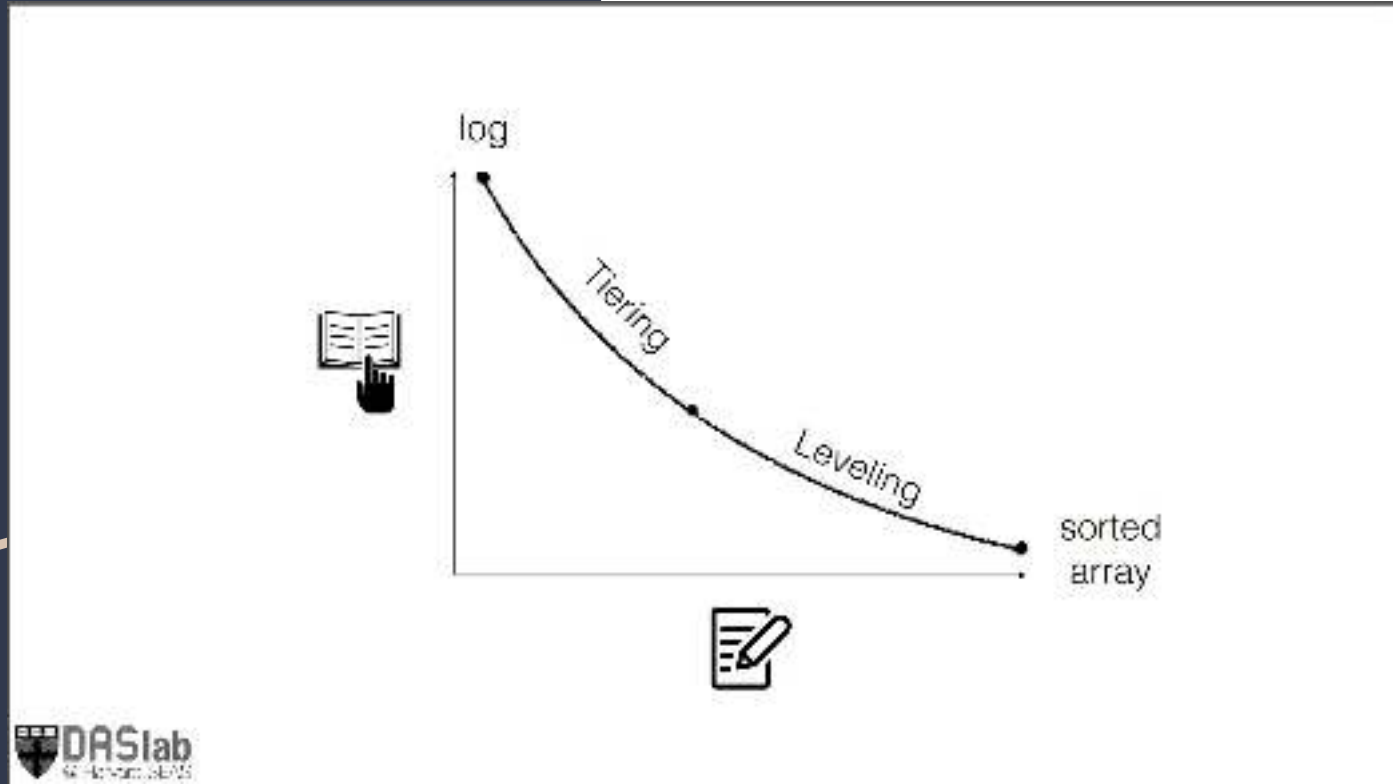
Merge Policies

- **Leveling**
  - A merge is triggered at a level as soon as a new run comes in
- **Tiering**
  - Gather runs and merges them only when it reaches capacity.
- **Uniform Bloom Filters Memory**
  - Require more bits per entry for the largest level as the data size grows in order to keep the point read cost fixed
-

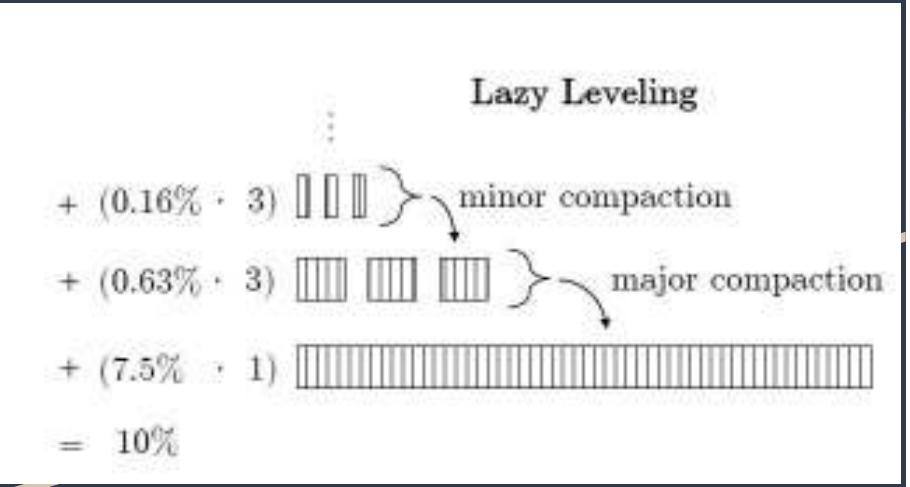
Background Continued

Sorted Array best for read (minimize read cost)

Log best for write (minimize write cost)



Background Continued:



- **Bloomptimized Filters Memory**
  - Set exponentially decreasing False Positive Rates to smaller levels relative to the largest level's FPR
  - Require linearly more bits per entry at smaller levels, the number of entries for smaller levels decreases at a much faster exponential rate
- **Cost Emancipation Asymmetry**
  - Sum of FPR (read cost) mostly come from larger levels, but write cost come equally from every level
- **Lazy Leveling**
  - Reduce merge overhead by using Tiering on L-1 levels and Leveling on level L
  - WA is O(T+L) (T from largest level and L from 1 to L-1 levels)
  -

Background Continued:

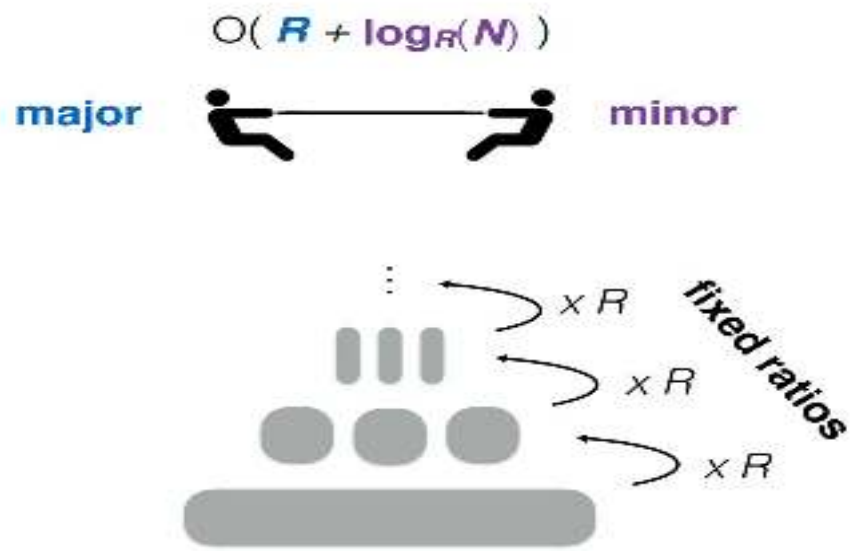
Problem with Lazy Leveling:

- What can be a problem with optimizing the lazy leveling policy?

Background Continued:

Problem with Lazy Leveling:

- What can be a problem with optimizing the lazy leveling policy?



DASlab @ Harvard SEAS

Capped Lazy Leveling

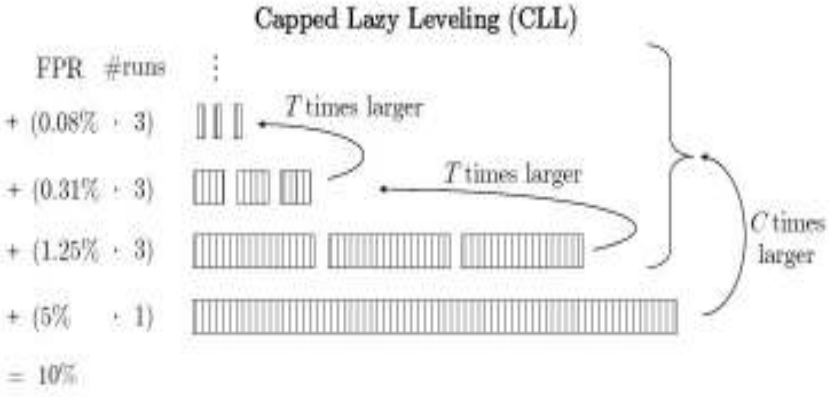


Figure 3: CLL allows the largest level's capacity ratio to be tuned independently (T is set to 4, C is set to 1, and p is set to 0.1 or 10% in this figure).

- **High Level Design**
  - Allow decoupling control of major and minor cost
  - Tiering in L-1 levels
  - Introduce new parameter capping ratio C, this allow varying the largest level capacity independently.
- **Level Capacities**
  - the capacity at Level i is smaller than at Level L by a factor of the inverse product of the capacity ratios between these levels
- **Number of Levels**

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

Capped Lazy Leveling Continued

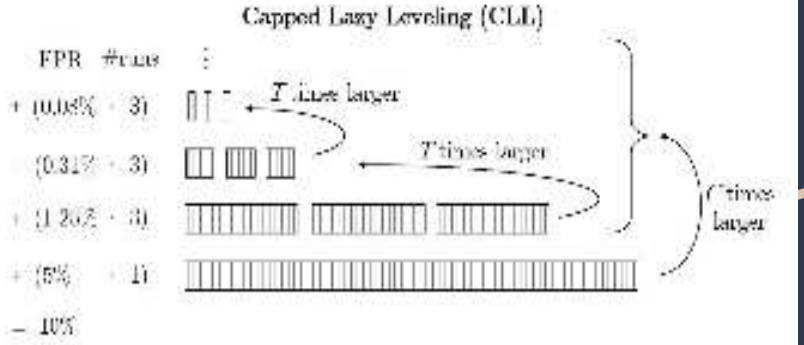


Figure 3: CLL allows the largest level's capacity ratio to be tuned independently (T is set to 4, C is set to 1, and p is set to 0.1 or 10% in this figure).

- **Runs at each level**

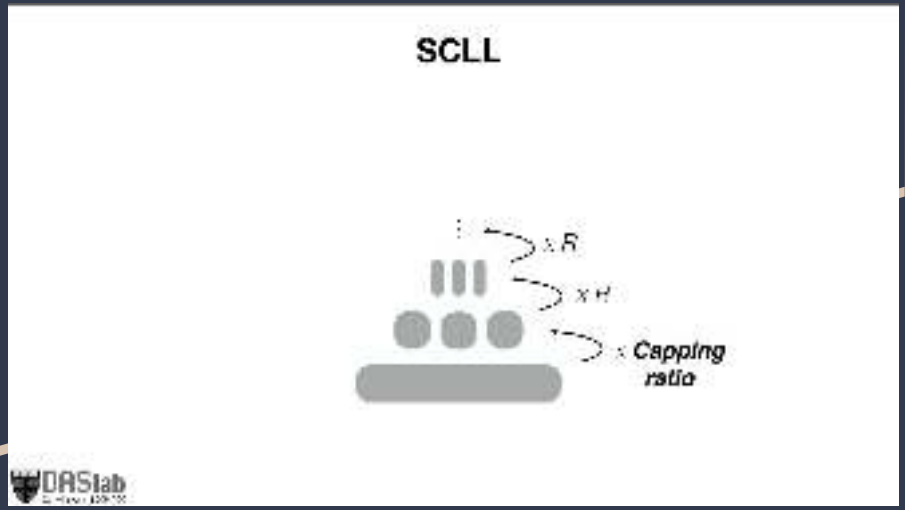
$$a_i = \begin{cases} T-1, & 1 \le i \le L-1 \\ 1, & i = L \end{cases}$$

- **Bloom Filters**
  - Fixed largest level and exponentially decreasing FPRs in the smaller levels

$$p_i = \begin{cases} p \cdot \frac{1}{C+1} \cdot \frac{1}{T^{L-i}}, & 1 \le i \le L-1 \\ p \cdot \frac{C}{C+1}, & i = L \end{cases}$$

- **Top-Down Capacity Determination**
  - adjusting the widths of Levels 1 to L−1 after every major compaction to ensure that their cumulative capacity is smaller by a factor of C than the size of the new run at Level L.

Capped Lazy Leveling Continued



- **Memory**
  - Increase capping ratio C, memory requirement decreases because it brings data to the largest level.
- **Write Amplification**
  - An entry on average participates in one minor compaction at each of Levels 1 to L−1, and in O(C) major compactions at Level L
- **Squared CLL**
  - Variant of CLL that sets the capping ratio C to be equal to the number of levels L.
  - Increase fraction of data on the largest level, which has the highest FPR, and requires fewest bits per entry

Capped Lazy Leveling Continued

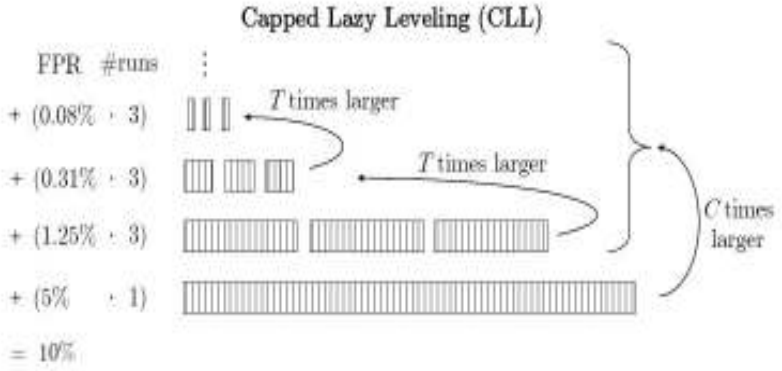


Figure 3: CLL allows the largest level's capacity ratio to be tuned independently (T is set to 4, C is set to 1, and p is set to 0.1 or 10% in this figure).

- **Better Trade-Offs:**
  - More all around scalable tradeoffs
  - **Trading memory gain for point read**, need to hold M(average bits per entry) fixed, which will cause the sum of FPRs to decrease
  - **Trading memory gain for write cost keeping point read cost fixed and memory fixed**, require to increase T, which still performs better than a LSM Tree
- **Analyzing Range Reads:**
  - Not good for range reads, good for WPI workload
  - Double WA does not reduce the number of runs a range read has to access

Discussion!

1. What is the difference between major and minor compactions?
2. What are the pros and cons of using CLL?
3. What is the effect of having less merging on write and read costs?

Is it possible to do even better than SCLL?

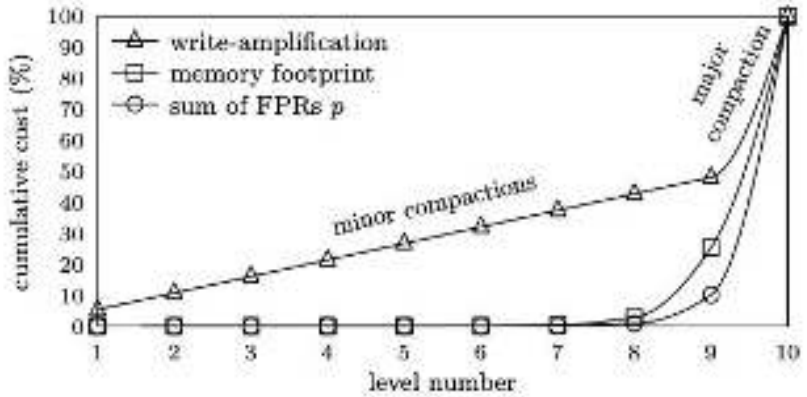
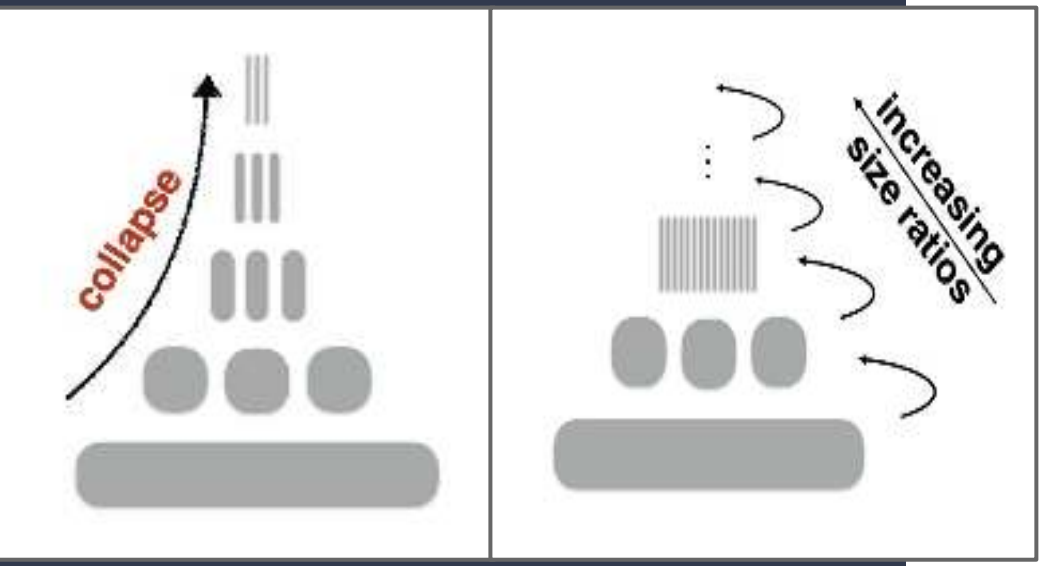


Figure 5: Minor compactions significantly increase WA while yielding exponentially diminishing returns for point reads and memory.

Illustrates a cumulative breakdown of how the different cost metrics emanate from different levels with SCLL

- The core insight is that while minor compaction overheads increase logarithmically with the data size, they lead to exponentially diminishing returns with respect to memory and point reads

Solution: THE LOG-STRUCTURED MERGE-BUSH



- To address this cost asymmetry, we introduce LSM-BUSH
  - A new data structure, generalization of CLL, that sets increasing capacity ratios between adjacent pairs of smaller levels
  - As a result, smaller levels get lazier by gathering more runs before merging them
  - more scalable tradeoffs all around

Discussion!

What key implementation does this paper introduce for LSM bush? And why?

LSM Continued

Key Innovation

- LSM-bush introduces a new parameter called the **growth exponential X** to s pairs of adjacent smaller levels

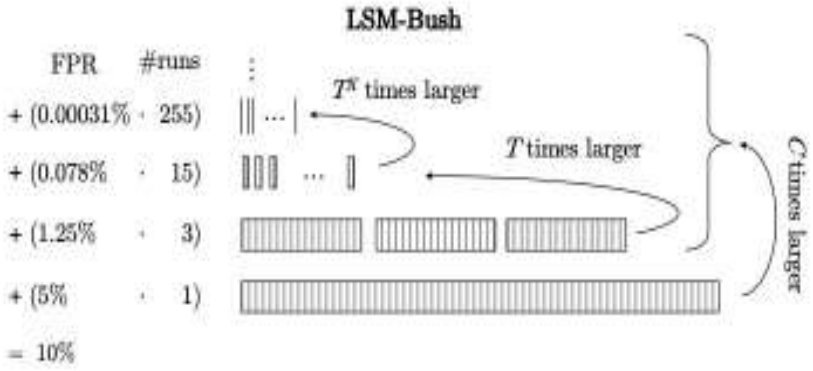


Figure 6: LSM-bush sets increasing capacity ratios between smaller levels (T is set to 4, C is set to 1, p is set to 0.1 or 10%, and X is set to 2 in this figure).

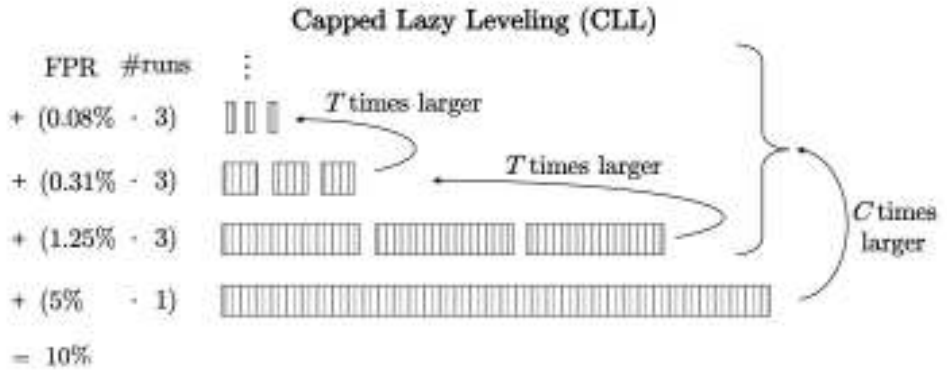
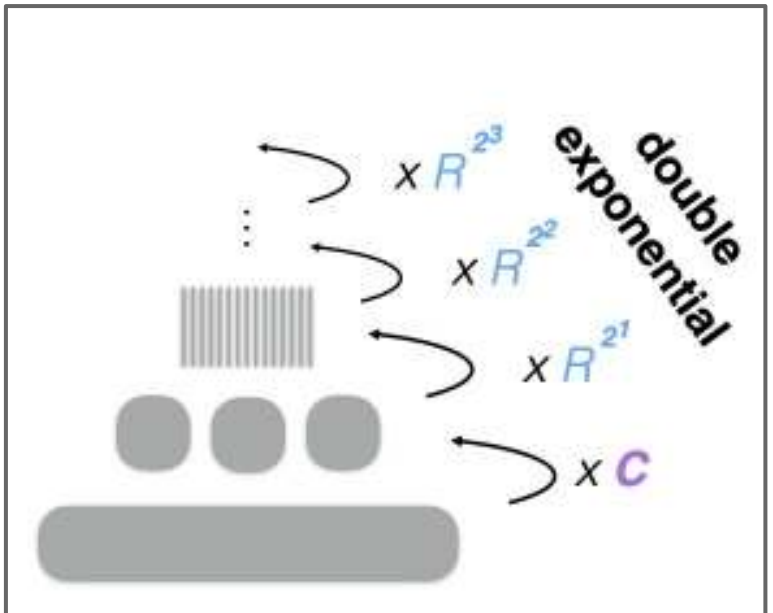


Figure 3: CLL allows the largest level's capacity ratio to be tuned independently (T is set to 4, C is set to 1, and p is set to 0.1 or 10% in this figure).

LSM Continued: Capacity Ratio

$$r_i = \begin{cases} T^{X^{L-i-1}}, & 1 \le i \le L-1 \\ C \cdot \frac{T}{T-1}, & i = L \end{cases} \quad (8)$$

- For all Levels 1 to to L-1,, the capacity ratio ri at Level i is greater by a power of X than the capacity ratio at Level i+1
- Result:
  - The capacity at smaller levels decreases at a doubly-exponential rate.
  - When X is set close to 1, LSM-bush becomes identical to CLL. As we increase X, smaller levels become increasingly lazier by gathering more runs before merging them.



LSM Continued: Level Capacities and Number Of Levels

$$N_i = \begin{cases} \frac{N}{C+1} \cdot \left(\frac{T}{r_i}\right)^{\frac{1}{X-1}} \cdot \frac{r_i - 1}{r_i}, & 1 \le i \le L-1 \\ N \cdot \frac{C}{C+1}, & i = L \end{cases}$$

Level Capacity: Ni is the the capacity at Level i, derived i by observing that it is smaller than Level L by a **factor of the inverse product of the capacity ratios between these levels**.

$$L = \left\lceil 1 + \log_X((X-1) \cdot \log_T(\frac{N}{F} \cdot \frac{1}{C+1} \cdot \frac{T-1}{T}) + 1) \right\rceil \qquad (10)$$

Number of Levels: The number of levels is found by observing that the **largest level is larger than the buffer by a factor of the product of all capacity ratios.**

LSM Continued: Runs at Each Level

$$a_i = \begin{cases} r_i - 1, & 1 \le i \le L-1 \\ 1, & i = L \end{cases} \quad (11)$$

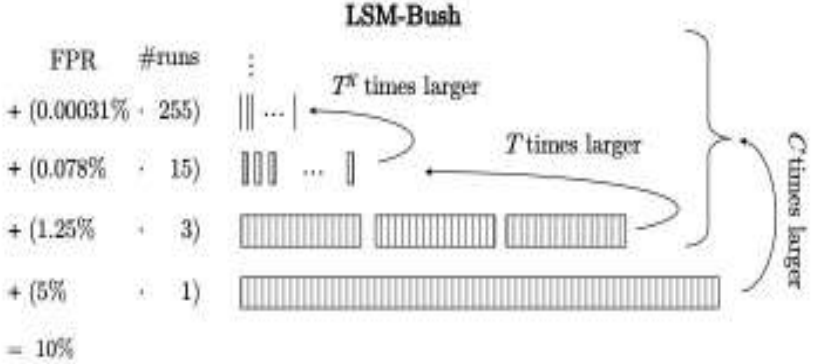


Figure 6: LSM-bush sets increasing capacity ratios between smaller levels (T is set to 4, C is set to 1, p is set to 0.1 or 10%, and X is set to 2 in this figure).

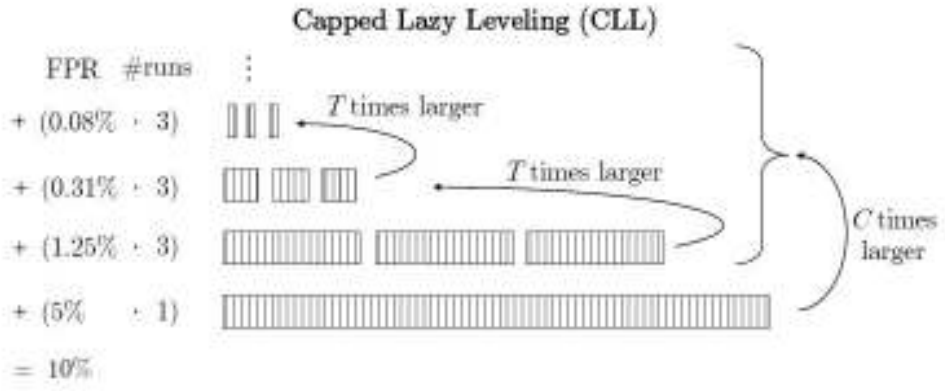


Figure 3: CLL allows the largest level's capacity ratio to be tuned independently (T is set to 4, C is set to 1, and p is set to 0.1 or 10% in this figure).

Ai is the maximum number of runs at Level i

For Levels 1 to L−1, Level i gathers at most ri−1 runs from Level i − 1 before reaching capacity (the r th i run triggers a minor compaction). On the other hand, Level L has one run since a major compaction is triggered whenever a new run comes in.

LSM Continued: Bloom Filters

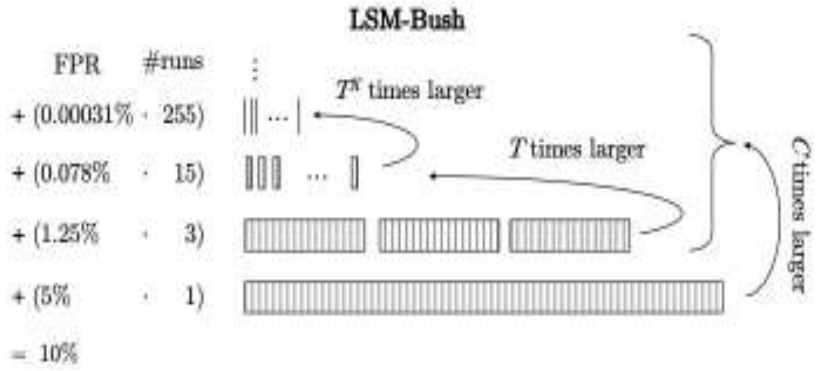


Figure 6: LSM-bush sets increasing capacity ratios between smaller levels (T is set to 4, C is set to 1, p is set to 0.1 or 10%, and X is set to 2 in this figure).

$$p < \frac{C+1}{C} \cdot a_L.$$

$$p_i = \begin{cases} \frac{p}{a_L} \cdot \frac{1}{C+1} \cdot \frac{r_{i-1}}{r_i} \cdot \left(\frac{T}{r_i}\right)^{\frac{1}{X-1}}, & 1 \le i \le L-1 \\ \frac{p}{a_L} \cdot \frac{C}{C+1}, & i = L \end{cases}$$

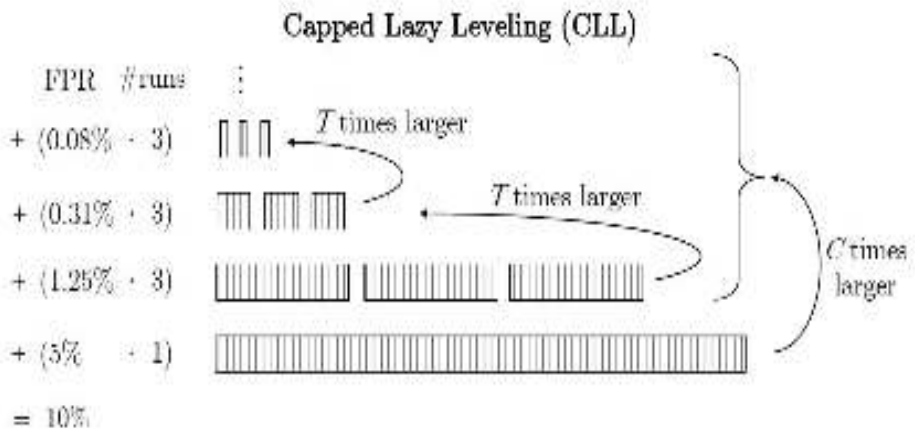
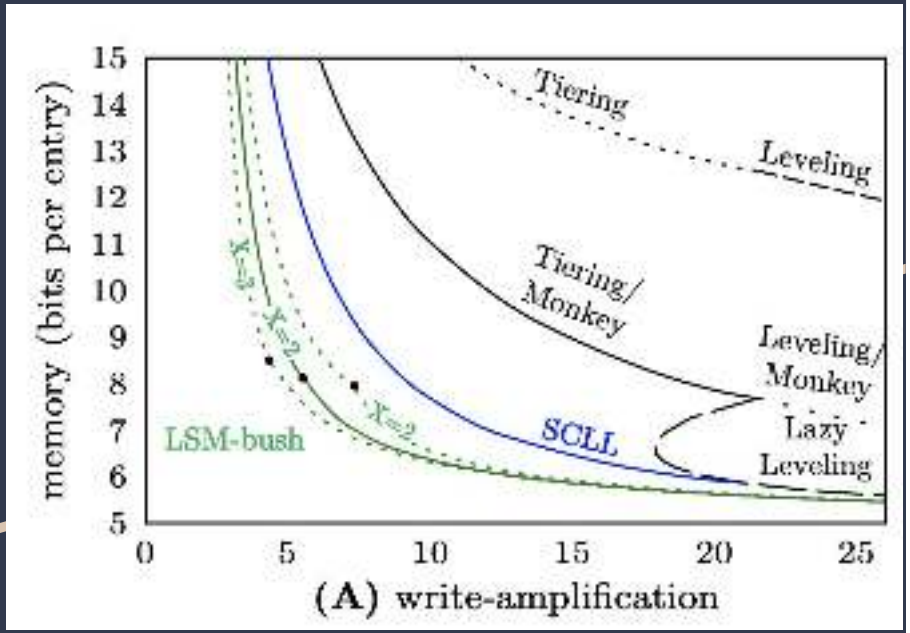


Figure 3: CLL allows the largest level's capacity ratio to be tuned independently (T is set to 4, C is set to 1, and p is set to 0.1 or 10% in this figure).

This equation Bloomptimizes the filters such that the largest level's filter has a fixed FPR while smaller levels are assigned decreasing FPRs as the data grows to keep the sum of FPRs p fixed

Performs one hash table check rather than numerous Bloom filter checks for smaller levels

Properties of LSM Bush: Memory



$$\sum_{i=1}^{L} (B \cdot N_i \cdot \ln(1/p_i))/\ln(2)^2).$$

The memory footprint is derived by summing up the equation for a Bloom filter's memory across all levels

Capacity at smallers levels decreases at a doubly exponential rate

Defined by: $B \cdot N_i$,

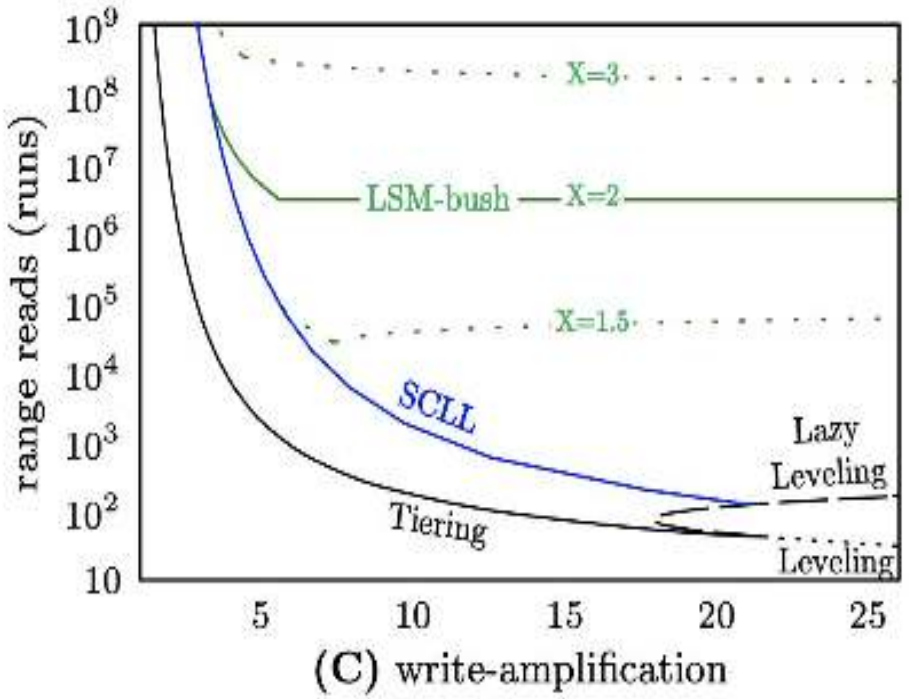
On the other hand, the number of bits per entry for smaller levels increases at an exponential rate

Defined by: $\ln(1/p_i)/\ln(2)^2$,

Thus, while smaller levels require exponentially more bits per entry, they have doubly-exponentially fewer entries

As a result, larger levels dominate the overall memory footprint. The number of bits per entry for LSM-bush needed for a given point read cost does not increase as the data grows.

Properties of LSM: Range Reads

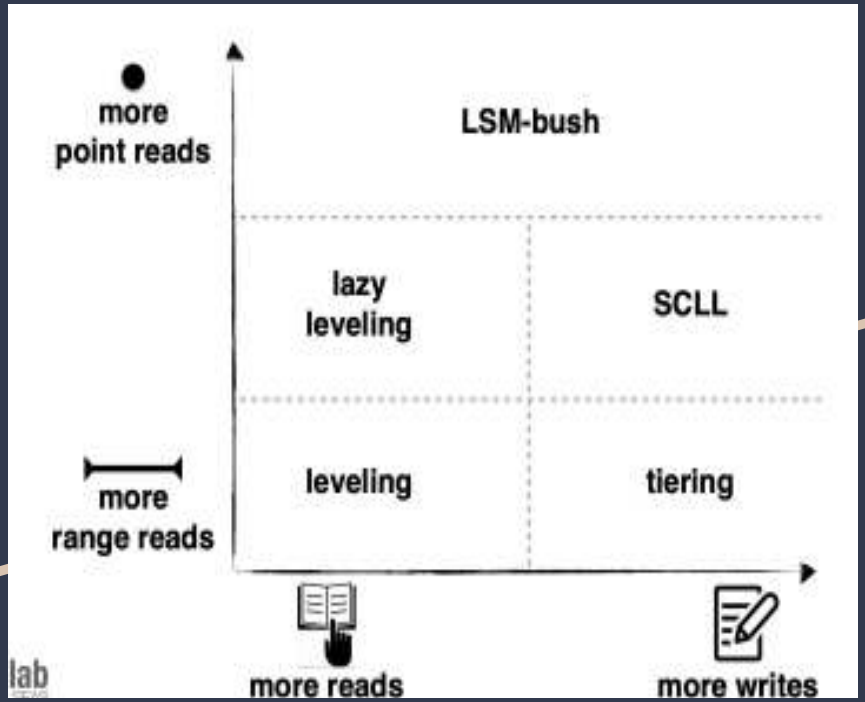


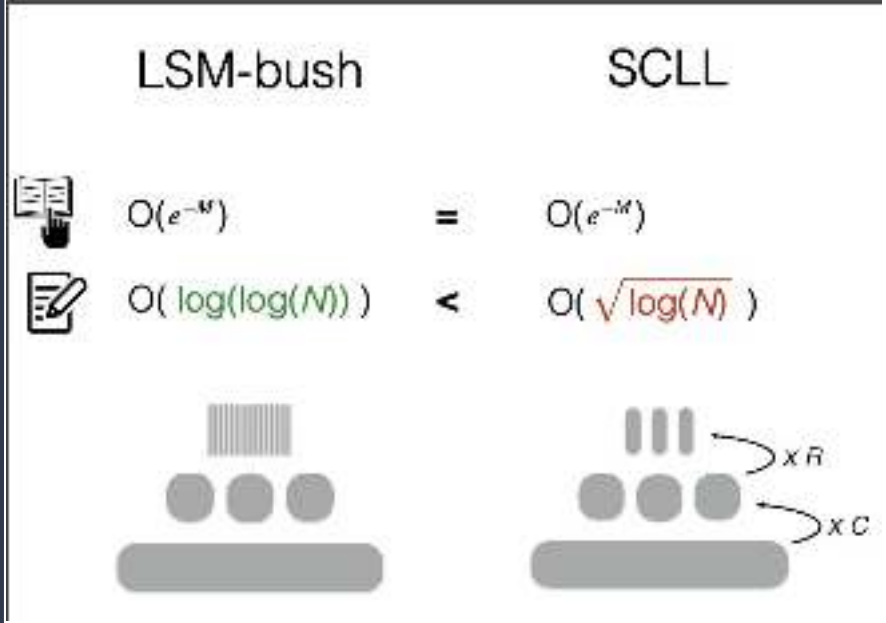
- Range read costs are derived analyzing the number of runs across all levels.
  - When the growth exponential X is close to 1, LSM-bush becomes identical to CLL
    - number of runs is O(1 +T · (L−1))
  - For higher values of X, the number of runs at Level 1 quickly comes to dominate the overall number of runs in the system since it contains doubly-exponentially more runs than at larger levels
- Thus, range reads generally has to access O(1+T · (L−1)+a1) runs.
- The complexity is analyzed by O(T^X^L−2)

Properties of LSM: Bloom Filter CPU Overheads

- As LSM-bush can contain a large number of runs at its smaller levels on account of merging more lazily
  - Performing a Bloom filter check for each of these runs during a point read can become a CPU bottleneck
  - To prevent this bottleneck, LSM-bush replaces the Bloom filters at smaller levels (typically at Levels 1 to L−3) by a hash table that maps from each entry to its physical location in storage
  - As a result, LSM-bush performs one hash table check rather than numerous Bloom filter checks for smaller levels
    - hash table requires more bits per entry than a Bloom filter does

LSM-Bush: Summary





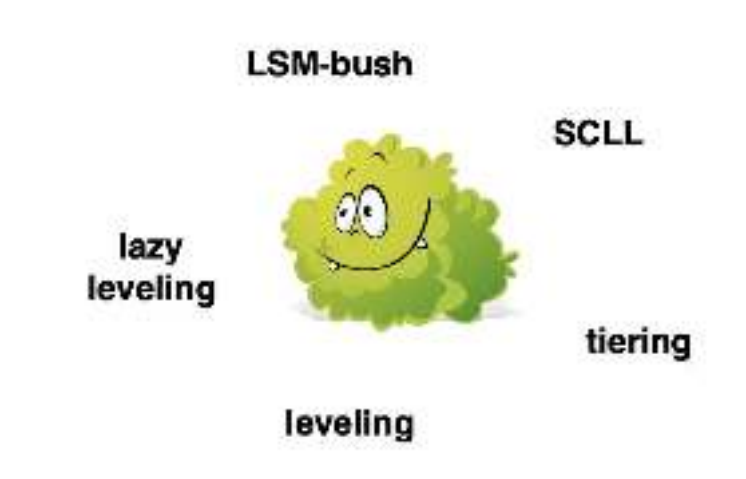
Overall, a faster write time can be achieved using the LSM-bush structure by introducing the new variable (Growth Exponential X)

Discussion!

What model is best for high number of point reads?

What model is best for more write and more range reads?

The Wacky Continuum



Wacky: **A**morphous **C**alculable **K**e**y**-Value Store

- Wacky is a generalization that can assume any of the Bloomptimized designs discussed so far.
  - A design continuum that can be tweaked to transition across designs, includes cost models to predict how tweaks impact overall system behavior
- Wacky's design space can be searched analytically and navigated in small and informed steps to converge to the best performformance model

Wacky Continuum: Controlling Merging within Levels

$$a_i = \begin{cases} (r_i - 1)^K, & 1 \le i \le L-1 \\ C^Z, & i = L \end{cases}$$

- Wacky inherits the LSM-bush design space with the goal to be able to optimize for range reads
- Incoming data into a level gets merged into the active run. Once the run reaches the merge threshold, it becomes static and a new active run is initialized.
    - When merge threshold = 1, we have **Leveled merging**.
    - When merge threshold= 1/(ri−1), we have **Tiered merging**
- Wacky introduces new two parameters:
    - K controls the merge threshold at Levels 1 to L - 1
    - Z controls the merge threshold at Level L
    - Allows Wacky to fine-tune the merge threshold across different levels
- Equation: the maximum number of runs that each level can have
    - When K and Z are both 0 then there is Leveled Merging; When both are 1 then Tiered Merging

Wacky Continuum

		QLSM-bush	SCLL	CLL	Tiering/Monkey	Lazy Leveling	Leveling/Monkey
Wacky	capping ratio C	C	L	C	$T-1$	$T-1$	$T-1$
	growth exponential X	2	$\to 1$	$\to 1$	$\to 1$	$\to 1$	$\to 1$
	small levels merge greed K	1	1	1	1	1	0
	largest level merge greed Z	0	0	0	1	0	0
Costs	zero-result point read (I/O)	$O(\rho)$	$O(\rho)$	$O(\rho)$	$O(\rho)$	$O(\rho)$	$O(\rho)$
	point read (I/O)	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
	write-amplification	$O(C + \log_2 \log_T \frac{N}{F})$	$O(\log_T \frac{N}{F})$	$O(C + \log_T \frac{N}{F})$	$O(\log_T \frac{N}{F})$	$O(T + \log_T \frac{N}{F})$	$O(T \cdot \log_T \frac{N}{F})$
	filters memory (bits/entry)	$O(\ln \frac{T^{1/C}}{\rho})$	$O(\ln \frac{T^{1/L}}{\rho})$	$O(\ln \frac{T^{1/C}}{\rho})$	$O(\ln \frac{T}{\rho})$	$O(\ln \frac{1}{\rho})$	$O(\ln \frac{1}{\rho})$
	range read cost (runs)	$O(\sqrt{\frac{N}{F} \cdot \frac{T}{C}})$	$O(T \cdot \log_T \frac{N}{F})$	$O(1 + T \cdot \log_T \frac{N}{F \cdot C})$	$O(T \cdot \log_T \frac{N}{F})$	$O(1 + T \cdot \log_T \frac{N}{F \cdot T})$	$O(\log_T \frac{N}{F})$

Figure 8: A list of Wacky's parameters and how to tune them to assume both existing and new designs including CLL, SCLL and QLSM-bush, which provide increasingly more scalable WPI performance.

- Wacky has a total of five parameters, K, Z, X, T and C, which fully dictate the overall structure.
  - Figure 8 shows how these parameters can assume any of the designs discussed so far

Wacky Continuum: Searchable Cost Model

A generalized cost model for Wacky with the goal of being able to search for the best design for a given application.

- Write Cost: divide write-amplification by the block size B
  - each entry gets copied an average of (C/a_L) times at Level L and $(r_i - 1)/(a_L + 1)$ at level i
  - assumes preemptive merging, whereby we include all runs at Levels 1 to i in a merge operation if these levels are all just below capacity

$$W = \frac{1}{B} \cdot \left(\frac{C}{a_L} + \sum_{i=1}^{L-1} \frac{r_i - 1}{a_i + 1} \right)$$

Wacky Continuum: Searchable Cost Model

$$R_{zero} = p$$

$$R = 1 + (p - p_L \cdot \frac{(a_L + 1)}{2})$$

$$V = \sum_1^L a_i.$$

- Point Reads:
  - Rzero = the I/O cost of a point read to a non-existing entry
    - P = the sum of all FPRs
  - R = the cost of a worst-case point read
    - (1) one I/O to fetch the target entry from Level L
    - (2) an average of p −pL ·aL false positives while searching Levels 1 to L−1
    - (3) an average of pL ·aL/ 2 false positives while searching Level L
    - sum up these three terms to obtain the expression for R
- Range Reads:
  - V = the total number of runs in the system

Wacky Continuum: Searchable Cost Model

Finding the best design:

- Find average worst-case operation cost Θ in different types of operations in the workload
- r= point reads
- z= zero-result point reads
- w= writes
- v= range reads
- Multiply each of them by the corresponding I/O cost
- Allows the search of the design to minimizes the average operation cost

$$\Theta = r \cdot R + z \cdot R_{zero} + w \cdot W + v \cdot V$$

Alternative: Searching all combinations of Wacky's parameters can be computationally intensive

- Sse an iterative approach to increment T to find a local minimum for the average operation cost Θ

Evaluation

Implementation:

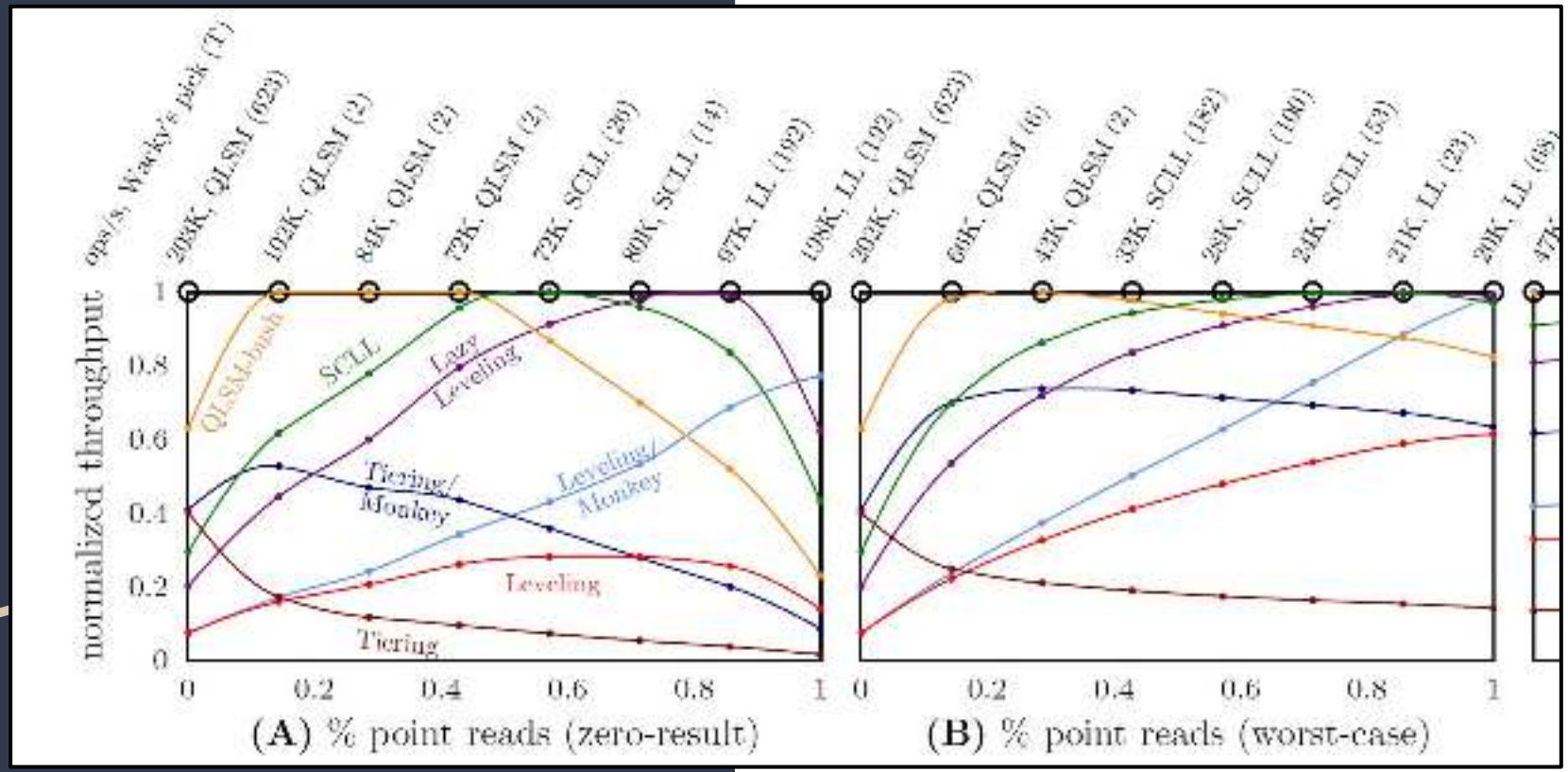
- Wacky on top of Rocks-DB
- Use RocksDB API to trigger merging across levels and within levels
- Extend RocksDB to set different FPRs to different levels

Default Workload:

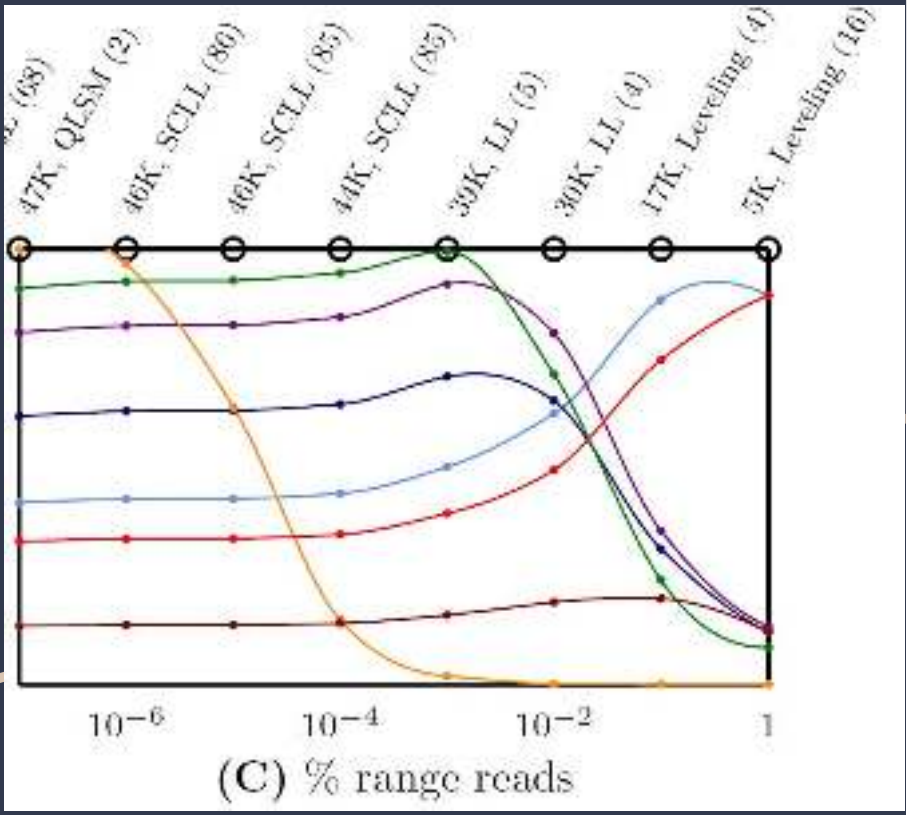
- 256GB data with 128B entries
- Default 5 bits per entry to the Bloom Filters of each baseline

Evaluation

Point Reads vs Writes



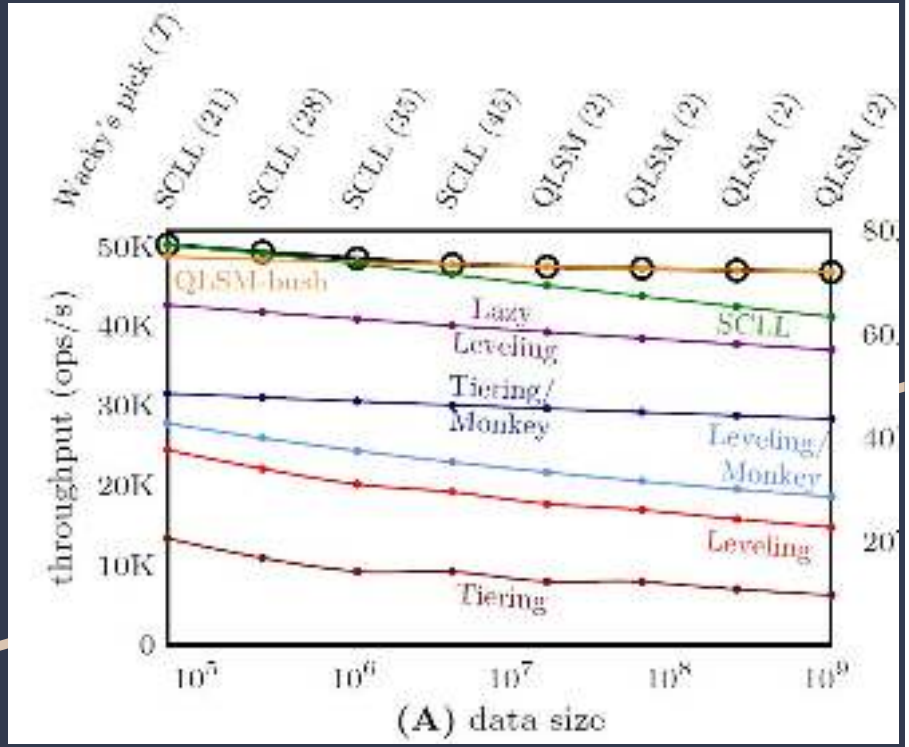
Evaluation



Range Read vs Writes

- 60% writes and 40% read
- QLSM-bush to SCLL to lazy leveling and finally to Leveling

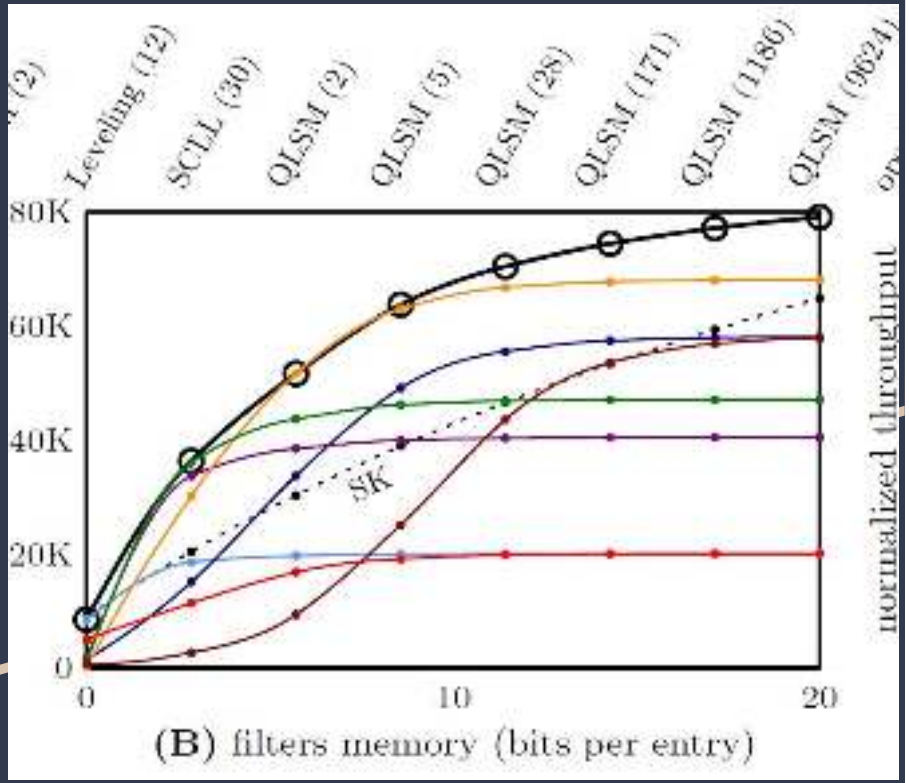
Evaluation



Data Size Scalability:

- Workload of 60% insertions and 40% point reads(half zero results and half at largest level)
- Switch from SCLL to QLSM-Bush

Evaluation

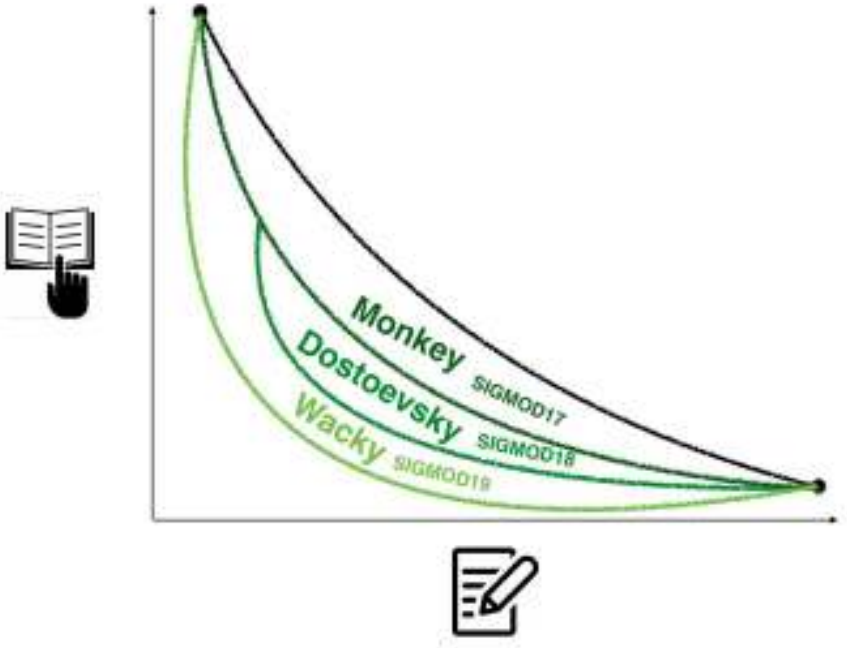


Memory Scalability

- Dash curve is SK (log-structured hash table design)

- SK performs better than Wacky given more than 40 bits per entry for bloom filter

Evaluation



Systems Comparison

- Wacky vs Monkey(Bloomptizied Tierd and leveled) vs Dostoevsky(Monkey with LL)
- Wacky outperform Dostoevsky in write intensive workload due to SCLL and QLSM-Bush

Related Works: LSM Tree

- Write-intensive workloads -> LSM-tree's merge operations become a performance bottleneck
- Solutions
  a. Partition individual runs
  b. Store the value components of all entries
  c. Pack entries more densely into the buffer
- Many operations in existing LSM-tree designs are fundamentally unimpactful and can be removed by applying increasing capacity ratios across smaller levels to scale write cost better than existing LSM-tree designs (LSM BUSH)

Related Works: LSH-Table

- (LSH-Table) logs entries in storage and maps their locations in memory using a hash table
  - Exhibits optimum write performance at the expense of having a **high memory footprint**
- Solution:
  - Unify the design spaces of LSM-tree and LSH-table to be able to perform as well as possible under any memory constraints
  - -> Smaller levels of LSM-bush are a series of LSH-tables that evolve as a bush

1. Existing key-value stores backed by an **LSM tree** exhibit deteriorating performance/memory trade-offs as the data grows
2. Introduce **LSM-bush**
3. Embed LSM-bush within **Wacky**