The Log-Structured Merge-Bush & the Wacky Continuum

Niv Dayan, Stratos Idreos
Harvard University

ABSTRACT

Data-intensive key-value stores based on the Log-Structured Merge-Tree are used in numerous modern applications ranging from social media and data science to cloud infrastructure. We show that such designs exhibit an intrinsic contention between the costs of point reads, writes and memory, and that this trade-off deteriorates as the data size grows. The root of the problem is that in all existing designs, the capacity ratio between any pair of levels is fixed. This causes write cost to increase with the data size while yielding exponentially diminishing returns for point reads and memory.

We introduce the Log-Structured Merge-Bush (LSM-Bush), a new data structure 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. By using a doubly-exponential ratio growth rate, LSM-bush brings write cost down from $O(\log N)$ to $O(\log \log N)$, and it can trade this gain to either improve point reads or memory. Thus, it enables more scalable trade-offs all around.

We further introduce Wacky, a design continuum that includes LSM-Bush as well as all state-of-the-art merge policies, from laziest to greediest, and can assume any of them within a single implementation. Wacky encompasses a vast space of performance properties, including ones that favor range reads, and it can be searched analytically to find the design that performs best for a given workload in practice.

ACM Reference Format:
Niv Dayan, Stratos Idreos. 2019. The Log-Structured Merge-Bush & the Wacky Continuum. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, Article 4, 18 pages. https://doi.org/10.1145/3299869.3319903

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3319903

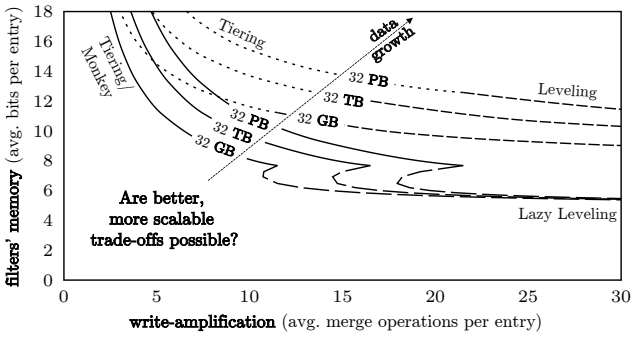

Figure 1: Existing designs need to sacrifice increasingly more write performance or more memory to hold point read performance fixed as the data grows.

1 SCALING WITH DATA

Key-Value Stores are Everywhere. A key-value store (KV-store) is a database that maps from key identifiers to their corresponding data values. KV-stores are a driving force behind the NoSQL revolution, and they are used by numerous and diverse applications including graph processing [8, 15], crypto-currencies [49], online transaction processing [27], time-series data management [36, 37, 52], flash translation layer design [22], and spatial data management [58].

Driving Industry Trends. There are three trends that drive the design of KV-stores. First, the proportion of write operations in many applications is increasing [53]; we are collecting more data more quickly. Second, for data-intensive applications, the size of the raw data is larger than the available memory (DRAM), and so most data resides in storage devices, which are orders of magnitude slower than memory devices. Third, the advent of flash-based solid state drives (SSDs) has made write operations costlier than read operations [1]. Due to these trends, many KV-stores increasingly optimize for input/output (I/O) operations in storage, trying to rapidly ingest application writes while still enabling fast application reads under a tight memory budget.

LSM-Tree. To accommodate these trends, many modern KV-stores rely on the Log-Structured Merge-tree (LSM-tree) [46] as their storage engine. LSM-tree buffers writes in memory, flushes them as sorted runs to storage, and merges similarly-sized runs across multiple levels of exponentially increasing capacities. Modern designs use in-memory fence pointers to allow reads to find the relevant key range at each run

quickly, and they use in-memory Bloom filters to allow point reads to skip runs that do not contain a target entry. Such designs are used by LevelDB [29] and BigTable [19] at Google, RocksDB [28] at Facebook, Cassandra [38], HBase [6] and Accumulo [4] at Apache, Voldemort [41] at LinkedIn, Dynamo [26] at Amazon, WiredTiger [59] at MongoDB [44], and bLSM [53] at Yahoo.

Problem 1: Three-Way Trade-Off. Existing LSM-tree designs exhibit an intrinsic trade-off between the costs of reads, writes, and memory. It is not possible to simultaneously achieve the best case on all three metrics [10]. For example, in order to enable faster point reads, we either need to (1) use more memory to enlarge the Bloom filters and thereby reduce read I/Os due to false positives, or to (2) increase write cost by compacting runs more greedily to restrict the number of runs across which false positives can be incurred. With existing designs, tuning to improve on one of these three metrics makes either one or both of the other metrics worse. As a result, no single design/tuning performs well universally as the trade-off has to be managed for the given workload and memory budget.

Problem 2: Worse Trade-Offs as the Data Grows. To exacerbate the problem, Figure 1 shows that the trade-off deteriorates as the data grows. The curves in the figure represent different LSM-tree variants, each with a distinct trade-off domain. We include Tiering and Leveling, which are used broadly in industry. We also include newer designs that optimize memory across the Bloom filters, namely Tiering/Monkey [20, 21] and Lazy Leveling [23]. The x-axis measures write cost as write-amplification (WA), defined as the average number of times a data entry gets rewritten due to compactions. The y-axis measures memory as the average number of bits per entry needed across all Bloom filters to fix the expected number of false positives to a small constant (0.1 in this example). This allows plotting the write/memory trade-offs for each design while holding point read cost fixed, though the curves look similar relative to each other for any two of these metrics plotted against each other while holding the third fixed. We generate the curve for each design by varying the capacity ratio, which dictates merge greediness across levels, to enumerate all possible trade-offs. The curves are drawn using models that we explain in detail later on.

The curves for Tiering/Monkey and Lazy Leveling complement each other and thus comprise the Pareto frontier of best existing trade-offs with respect to writes, memory and point reads (their curves meet at the point where the capacity ratio is set to 2, in which case their behaviors become identical [23]). The figure shows that *as the data grows, the Pareto frontier moves outwards leading to worse trade-offs.* For example, assume a starting point of Tiering/Monkey with 10 bits per entry available for its Bloom filters. With 32GB of data, WA is ≈ 6. To support data growth to 32PB while holding point read cost fixed, we either have to pay over 2x in WA (e.g., moving rightwards in Figure 1) or have to reserve $\approx 50\%$ more bits per entry for the Bloom filter (e.g., moving upwards in Figure 1). The trade-off deteriorates at an even faster rate for plain Tiered and Leveled designs.

Write and Point Read Intensive Workloads. The potential for enabling more scalable trade-offs depends on the workload. While frequent compactions are needed to support fast range reads, many modern applications ranging from e-commerce [26] and blockchain [49] to multi-player gaming [25] exhibit write-intensive workloads with mostly or only point reads. We refer to such workloads as **w**rite and **p**oint read **i**ntensive, or WPI for short. Many LSM-tree designs have been built specifically to support such workloads. Examples include Dynamo [26], Voldemort [41], and LSM-trie [60], which index entries in the LSM-tree based on hashes of keys. For WPI workloads, there are untapped opportunities for improving the scalability of the three-way read/write/memory trade-off.

Insight: Not all Compactions are Created Equal. In this paper, we analyze how compaction overheads emanate from across different levels and the amount by which each compaction helps to curb the memory footprint and/or point read cost. We show that not all compactions are as impactful: some improve point reads and memory significantly while others improve them negligibly. The root cause is a cost emanation asymmetry. Larger levels contribute the most to the cost of point reads and memory. On the other hand, compaction overheads at smaller levels increase logarithmically with the data size while yielding exponentially diminishing returns. Existing designs are unable to address this problem because of a core in-built inflexibility: they assign fixed, uniform capacity ratios between any pair of adjacent levels.

The Log-Structured Merge-Bush. We introduce LSM-bush, a new data structure for WPI applications that rids them of non-impactful compactions. It does this by setting increasing capacity ratios between smaller pairs of adjacent levels. As a result, smaller levels get lazier as they gather more runs before merging them. In this way, non-impactful compactions are eliminated, and so overall compaction costs grow more slowly with the data size. By exposing the ratios' growth rate as a manipulable knob, LSM-bush allows the gain in write cost to be traded for either memory or point reads. Hence, it unravels a superior and more scalable Pareto frontier of read/write/memory trade-offs. Our previous work has improved the scalability of this trade-off by optimizing memory among the Bloom filters with Monkey [20, 21] and by controlling merge greediness *within* levels with Lazy Leveling [23]. LSM-bush builds on this work by further allowing to control of merge greediness *across* levels. We show that this

is a necessary next step to continue pushing the scalability envelope towards its theoretical and practical limits.

The Wacky Continuum. We further introduce **W**acky: **A**morphous **C**alculable **K**e**y**-Value Store. Wacky is *amorphous* in that it includes LSM-bush along with the whole spectrum of merge policies, from laziest to greediest, and it can instantiate any of them using a small and finite set of knobs within a single unified implementation. Wacky is *calculable* in that it includes a set of analytical models that allow searching for the merge policy that performs best for a given workload. In the spirit of our long-term vision on distilling a Periodic Table of Data Structures [33, 34], Wacky organizes a rich, vast and complex space into a design continuum [31] that allows to tractably reason about the impact of every design decision on the overall system's behavior.

Contributions. We summarize our contributions as follows.

- We show that existing LSM-tree designs are subject to a three-way read/write/memory trade-off that scales sub-optimally for WPI workloads.
- For WPI workloads with some range reads, we show how to adjust the capacity ratio of the largest level to improve write cost from $O(\log N)$ to $O(\sqrt{\log N})$ while keeping the same read and memory costs. We call this design Capped Lazy Leveling (CLL).
- For WPI workloads with no range reads, we generalize CLL to set increasing capacity ratios between smaller levels so that newer data is merged more lazily. By using a doubly-exponential ratio growth rate, write cost decreases to $O(\log \log N)$ without hurting point read or memory costs. We further show how to use hash-tables instead of Bloom filters at smaller levels to control CPU overheads. We call this design LSM-bush.
- We introduce Wacky, a design continuum that includes the whole spectrum of merge policies and can be navigated to find the best design for a given workload.
- Using an implementation and experimental analysis over RocksDB, we show that Wacky (1) significantly outperforms existing designs for WPI workloads, and (2) matches their performance for other workloads without requiring manual tuning.

2 BACKGROUND: MERGE POLICIES

In this section, we give the necessary background on existing LSM-tree designs and their cost properties.

High-Level Overview. LSM-tree organizes runs in storage into L levels of exponentially increasing capacities by using a fixed capacity ratio T between each pair of levels. As a result, the number of levels L is $O(\log_T(N/F))$, where N is the data size and F is the buffer size. Application writes are first buffered in memory and get flushed to Level 1 in storage as a sorted run each time the buffer fills up. Whenever Level i reaches capacity, all runs within that level get sort-merged[1] and the resulting run is pushed to Level i+1. Thus, data moves to larger levels as it ages.

Fence Pointers. To facilitate read performance, modern designs use in-memory fence pointers with the min/max key in every block of every run [28]. The fence pointers enable reads to find the block that contains the relevant key range at each run with one I/O.

Bloom Filters. To further facilitate point read performance, modern designs use an in-memory Bloom filter for every run to allow point reads to skip runs that do not contain a target entry. As each Bloom filter leads to one wasted I/O with a probability given by its false positive rate (FPR), the expected number of wasted I/Os per point read can be measured as the sum of FPRs across all filters, which we denote as p. In this way, the worst-case cost of a point read can be approximated as 1+p (one I/O to fetch the target entry plus p I/Os due to false positives). We continue to use p as a measure of point read cost throughout the paper, and we show how to model point read cost more precisely in Section 5.

Merge Trade-Offs. The greediness with which LSM-tree merges runs controls an important read/write/memory trade-off. In particular, reducing merge greediness decreases the amortized cost of writes while increasing the number of runs in the system. Having more runs, in turn, either causes the Bloom filters' sum of FPRs p to increase and to thereby increase point read cost, or it requires assigning more memory to the filters to keep point read cost fixed. In this way, the merge policy dictates the read/write/memory cost balances that can be achieved. For readability throughout the paper, we focus on the write vs. memory cost trade-off while holding point read cost fixed, though we continually point out how achieving a gain in any one of these metrics can be traded for a gain in either one or both of the other two.

Merge Policies. Figure 2 illustrates the three state-of-the-art merge policies, which we now analyze. We measure merge overheads in terms of write-amplification (WA), the average number of merge operations that each entry participates in while traveling from the smallest to largest level. We measure memory in terms of the average number of bits per entry needed across all Bloom filters to achieve a given value for p, the sum of FPRs. This allows us to study memory/merge trade-offs while holding the cost of point reads fixed. Our analysis is in terms of worst-case costs.

Leveling. With Leveling [28, 29, 46], a merge is triggered at a level as soon as a new run comes in. This leads to each entry getting merged on average $O(T)$ times within each level

[1]While we refer to merge operations as occurring at the granularity of whole runs, some designs partition each run into files and merge smaller groups of files at a time to avoid performance slumps [28]. The designs discussed in this paper are compatible with both approaches.

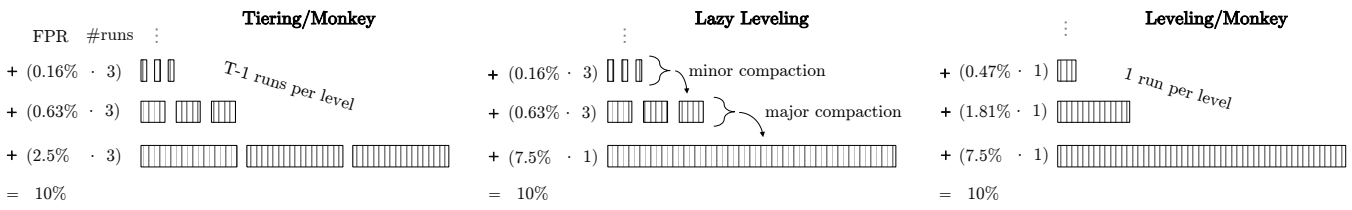

Figure 2: An illustration of the largest three levels with Tiered, Lazy Leveled and Leveled designs, all with a base ratio T of 4 and with Bloomptimized filters set to have an overall FPR sum p of 0.1, or 10%.

before the level fills up. WA is therefore $O(T \cdot L)$. As each level contains at most one run, there are $O(L)$ runs in the system. Leveling is the most greedy merge policy, and as such its write cost is highest. We analyze its memory footprint below.

Tiering. With Tiering [5, 35], each level gathers runs and merges them only when it reaches capacity. As each entry participates in one merge operation at each level, WA is $O(L)$. At the same time, each level contains at most $O(T)$ runs and so there are $O(T \cdot L)$ runs in the system. Tiering is a lazier merge policy than Leveling.

Uniform Bloom Filters Memory. The number of bits per entry needed for a Bloom filter to have an FPR of ϵ is $\frac{\ln(1/\epsilon)}{\ln(2)^2}$ [56], or more simply $O(\ln(1/\epsilon))$. LSM-tree designs in industry set the same FPR to filters at all levels [5, 6, 28, 29]. This means that in order to keep the FPR sum p fixed, the FPR for each individual filter has to decrease at a rate of p/L with Leveling or $p/(L \cdot T)$ with Tiering with respect to the number of levels L. As most of the memory overhead emanates from the largest level (since it contains exponentially more entries), we derive the memory complexity based on the largest level as $O(\ln(L/p))$ bits per entry with Leveling or $O(\ln((L \cdot T)/p))$ with Tiering. These expressions show that having to decrease the FPR for filter/s at the largest level requires more bits per entry as the data size increases to keep point read cost fixed.

Bloomptimized Filters Memory. A better approach is to keep the FPR at the largest level fixed as the data grows and to instead set decreasing FPRs to filters at smaller levels to prevent the sum of FPRs p from increasing. The intuition is that each false positive costs one I/O regardless of the level (due to the fence pointers), yet it is much cheaper in terms of memory to achieve lower FPRs at smaller levels as they contain exponentially fewer entries. We showed that the optimal approach is to set exponentially decreasing FPRs to smaller levels relative to the largest level's FPR [20, 21]. While this entails having *linearly* more bits per entry at smaller levels, the number of entries for smaller levels decreases at a much faster exponential rate, and this results in smaller levels still requiring an exponentially decreasing proportion of the overall memory footprint. As a result, the memory footprint becomes $O(\ln 1/p)$ bits per entry with Leveling or $O(\ln T/p)$ with Tiering. We coin this approach *Bloomptimization*, and we illustrate it in action in Figure 2.

Cost Emanation Asymmetry. Having exponentially decreasing FPRs for Bloom filters at smaller levels opens up a new avenue for improved merge policies for WPI workloads. The reason is that it creates an asymmetry in how the costs of writes, point reads and memory are derived from across different levels. In particular, the sum of FPRs p (and thus point read cost) mostly emanates from larger levels, which have the highest FPRs. Similarly, the Bloom filters' memory footprint mostly emanates from the largest levels as they contain most of the entries. On the other hand, write-amplification emanates equally from all levels (i.e., $O(1)$ per level with Tiering and $O(T)$ per level with Leveling). This means that the merge operations at smaller levels entail a lot of overhead, which does not significantly help to curb the costs of point reads or memory.

Lazy Leveling. With Lazy Leveling, we showed that a merge policy can leverage this asymmetry to reduce merge overheads by using Tiering for Levels 1 to $L-1$ and Leveling at Level L [23]. In this way, WA is $O(T+L)$ as an entry gets merged $O(1)$ time at each of Levels 1 to $L-1$ and $O(T)$ times at Level L. Since there is one run at Level L with a fixed FPR while smaller levels' filters are assigned exponentially decreasing FPRs as the data grows, the memory footprint remains $O(\ln 1/p)$ bits per entry. As Lazy Leveling reduces WA relative to Leveling while having the same memory complexity, it enables superior memory/merge trade-offs[2]. The broader principle is that the means of exploiting the cost emanation asymmetry is to *merge newer data more lazily*.

Minor vs. Major Compactions. We refer to merge operations at Levels 1 to $L-1$ as *minor compactions* and to merge operations at Level L as *major compactions*. In the next two sections, we present a series of techniques to alleviate their overheads in distinct ways. The new designs that we present generalize and augment our previous work on Bloomptimized filters and Lazy Leveling with the ability to set non-uniform and carefully-adjusted capacity ratios across different levels. The overarching goal is to further push the read/write/memory scalability envelope for WPI applications to maintain more stable performance as the data grows.

[2]We show later that Leveling is still a good choice for workloads with many range reads or for applications without enough memory for Bloom filters.

Term	Definition	Unit
N	total data size	blocks
F	buffer size	blocks
B	block size	entries
L	number of levels	levels
M	average bits per entry across all Bloom filters	bits
p	sum of FPRs across all Bloom filters	
N_i	data size at Level i	blocks
a_i	maximum number of runs at Level i	runs
r_i	capacity ratio between Levels i and $i-1$	
p_i	Bloom filter FPR at Level i	
T	base capacity ratio	
C	capping ratio (for largest level)	
X	ratio growth exponential	
K	Levels 1 to $L-1$ merge greediness	
Z	Levels 1 to L merge greediness	

Table 1: List of terms used throughput the paper.

3 CAPPED LAZY LEVELING

To devise a more scalable merge policy, we start by taking a closer look at Lazy Leveling. As we have seen in Section 2, WA complexity with Lazy Leveling is $O(T+L)$, and it can alternatively be expressed as $O(T+\log_T(N/F))$. Minor compactions contribute the $O(\log_T(N/F))$ term while major compactions contribute the $O(T)$ term. In order to prevent the overhead of minor compactions from increasing as the data grows, the only existing tuning option is to increase the capacity ratio T to curb the value of $O(\log_T(N/F))$. The problem, however, is that this causes the cost of major compactions $O(T)$ to increase. In this way, Lazy Leveling exhibits an intrinsic contention between the costs of minor and major compactions as the costs of both are controlled by the same knob. To address this, we introduce Capped Lazy Leveling (CLL), a generalization of Lazy Leveling that decouples major and minor compaction costs. It does this by allowing to control the largest level's capacity independently.

High-Level Design. Figure 3 illustrates the high-level design of CLL, and Table 1 lists terms that we use throughout the paper. Similarly to Lazy Leveling, CLL performs Tiered merging at Levels 1 to $L-1$ and Leveled merging at Level L. The novelty is that CLL introduces a new design parameter called the *capping ratio* C that allows varying the largest level's capacity ratio independently. As shown in Figure 3, we define C as the ratio between the data size at the largest level to the cumulative capacity across all smaller levels. On the other hand, the term T, coined the *base ratio*, denotes the capacity ratio between any pair of adjacent smaller levels.

The capping ratio C can be tuned to assume any value from one and above. When C is set to $T-1$, CLL becomes identical to Lazy Leveling. We further use Equation 1 to more generically denote r_i as the ratio between the capacities at Level i and Level $i-1$. Throughout the section, we show

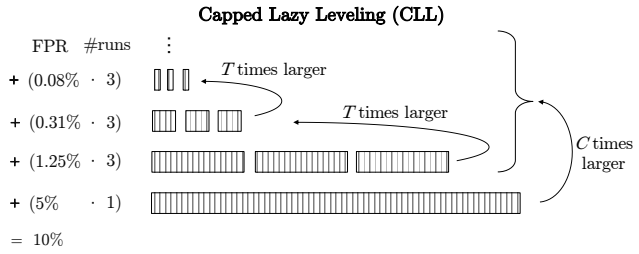

Figure 3: CLL allows the largest level's capacity ratio to be tuned independently (T is set to 4, C is set to 1, and p is set to 0.1 or 10% in this figure).

how to co-tune the knobs C and T to enable more scalable read/write/memory cost trade-offs for WPI workloads.

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

Level Capacities. To derive the cost properties for CLL, we start by formalizing its structure. Equation 2 denotes N_i as the capacity at Level i. We derive it in Appendix A by observing that 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.

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

Number of Levels. Equation 3 gives the number of levels L, derived in Appendix B by observing that the largest level's capacity is larger than the buffer size by a factor of the product of all capacity ratios.

$$L = \left\lceil \log_T \left(\frac{N_L}{F} \cdot \frac{T-1}{C} \right) \right\rceil \tag{3}$$

Runs at Each Level. Equation 4 denotes a_i as the number of runs that Level i gathers before a merge is triggered. Levels 1 to $L-1$ each have at most $T-1$ runs (the T^{th} run triggers a minor compaction), while Level L has one run since a major compaction is triggered whenever a new run comes in.

$$a_i = \begin{cases} T-1, & 1 \le i \le L-1 \\ 1, & i = L \end{cases} \tag{4}$$

Bloom Filters. Equation 5 denotes p_i as the FPR that CLL assigns to filters at Level i. These FPRs are derived in Appendix C by generalizing Bloomptimized filters for CLL such that the largest level's FPR is fixed as the data grows while smaller levels are set decreasing FPRs to keep the sum of FPRs p fixed (note that by definition $p = \sum_1^L a_i \cdot p_i$, and that Equation 5 is defined for $p < \frac{C+1}{C}$).

$$p_i = \begin{cases} p \cdot \frac{1}{C+1} \cdot \frac{1}{T^{L-i}}, & 1 \le i \le L-1 \\ p \cdot \frac{C}{C+1}, & i = L \end{cases} \tag{5}$$

Top-Down Capacity Determination. LSM-tree designs typically set the capacity at each level to be a multiple of the

buffer size [29]. The problem with applying this approach to CLL is that whenever a new largest level L gets created, it would contain only a single run of the same size as the runs that would later fill up Level $L-1$ as more data arrives. Hence, CLL would effectively degenerate into a Tiered LSM-tree whenever a new largest level is created (and thus violate the performance properties that we establish later in this section). In order to maintain an invariant that the data size at Level L always exceeds the cumulative data size across all smaller levels by a factor of at least C, CLL sets level capacities *top-down* based on the data size at Level L. It does this by 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. We describe this process in detail in Appendix E.

Memory. We now continue to derive the cost properties for CLL. We derive the memory footprint by summing up the standard memory equation for a Bloom filter's memory across all levels (i.e., $\sum_{i=1}^{L} (B \cdot N_i \cdot \ln(1/p_i))/\ln(2)^2$). We simplify this expression into closed-form in Appendix D and express it in terms of the average number of bits per entry M needed across all filters in Equation 6. In Equation 7, we express the 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.

$$M = \frac{1}{\ln(2)^2} \cdot \ln\left(\frac{1}{p} \cdot \frac{C+1}{C^{\frac{C}{C+1}}} \cdot T^{\frac{T}{(C+1)\cdot(T-1)}}\right) \tag{6}$$

$$M \in O\left(\ln \frac{T^{\frac{1}{C}}}{p}\right) \tag{7}$$

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. WA with CLL is therefore $O(C+L)$, which can also be expressed as $O(C+\log_T(N/P))$.

Squared CLL. Based on CLL's cost properties, we now show how to co-tune the capping ratio C and the base ratio T to enable the best possible trade-offs. We introduce Squared CLL (SCLL), a variant of CLL that sets the capping ratio C to be equal to the number of levels L[3]. We call it *squared* because it causes the cost of major and minor compactions to emanate equally from the largest level, i.e., the width, and from the smaller levels, i.e., the height. WA complexity therefore simplifies to $O(\log_T(N/F))$, while the memory requirement M becomes $O(\ln(\frac{T^{1/L}}{p}))$ and thus decreases as the data grows. The intuition is that as the data grows, SCLL increases the fraction of data at the largest level, which has the highest FPR and thus requires the fewest bits per entry.

Asymptotic Win. We summarize the properties of SCLL as well as for CLL in Figure 8, which holds the cost of point reads fixed while detailing the cost complexities for the rest of the cost metrics in terms of the base ratio T, the capping ratio C, and the sum of FPRs p. Figure 8 also includes range read cost, derived by analyzing the maximum number of runs $\sum_1^L a_i$. We observe that relative to Tiering, the memory complexity for SCLL is better while all other cost complexities are the same. In this way, SCLL achieves a net asymptotic win.

Better Trade-Offs. While SCLL improves memory relative to Tiering, the gain can be traded for either point read cost or write cost. Trading for point read cost requires keeping M fixed as the data grows. This causes the sum of FPRs p to decrease at a rate of $O(T^{1/L} \cdot e^{-M})$ (derived by rearranging Equation 7 in terms of p). On the other hand, trading for write cost while keeping point reads and memory fixed requires increasing T at a rate of $O(e^{\sqrt{\ln(N/F)}})$ as the data grows (derived by rearranging Equation 7 in terms of T, substituting L for C, and simplifying). We plug this function for T into SCLL's WA complexity, which becomes $O(\sqrt{\ln(N/F)})$, a slower asymptotic growth rate with respect to the data size than with a traditional LSM-tree. In this way, SCLL enables more scalable trade-offs all around.

New Pareto Frontier. Figure 4 Part (A) visualizes the memory/merge trade-offs that CLL enables relative to existing designs (all the designs from Figure 1 for 32PB for included as well as Monkey/Leveling). Each of the four green curves for CLL fixes C to a different value (i.e., 1, 2, 4 and L for SCLL) while varying T to enumerate different trade-offs. We give the precise cost models used to generate this figure in Section 5. The figure demonstrates that SCLL delineates the best memory/merge trade-offs across all instances of CLL. It further demonstrates that SCLL complements the cost curve for Lazy Leveling (i.e., it connects to it at the point where $T=L$). In this way, Lazy Leveling and SCLL delineate a new and improved Pareto frontier while rendering the Tiered merge policy suboptimal for WPI workloads.

Figure 4 Part (B) illustrates the scalability of the new Pareto frontier (we omit the points for Lazy Leveling that are not along the Pareto frontier for clarity). Subsequent curves of the same color and line style in the figure correspond to the trade-offs enabled by a given design for different data sizes (32GB, 32TB and 32PB). As the data grows, the new Pareto frontier moves outwards more slowly and thus enables more stable cost properties.

Analyzing Range Reads. Figure 4 (C) illustrate the trade-offs between range reads and write-amplification that SCLL enables. The log-scale y-axis measures range read cost as the

[3]Since the number of levels L is defined in terms of C in Eq. 3, setting C to L creates a circular dependency. We avoid this dependency by setting C to $\log_T(N/P)$, as this function approximates the number of levels well enough.

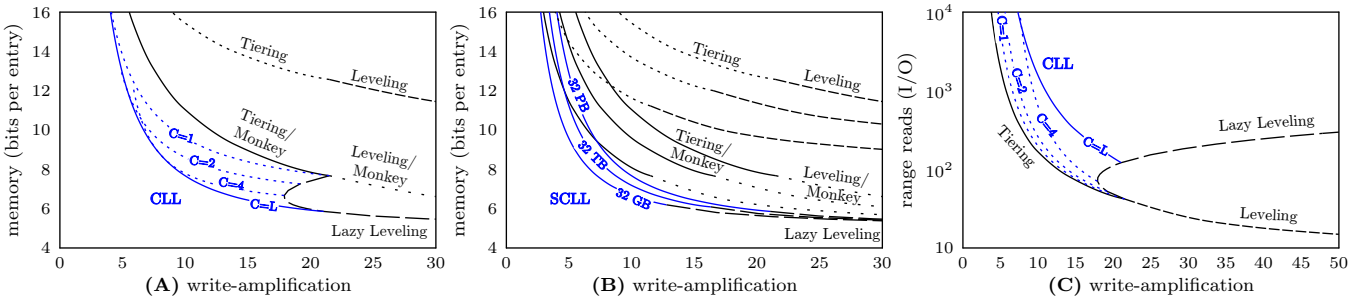

Figure 4: CLL provides better and more scalable trade-offs for WPI workloads at the expense of range reads.

maximum number of runs that a range read has to access. SCLL exhibits worse range/write trade-offs than with Tiering. The reason is that while major compactions with SCLL are crucial for bounding the costs of memory and point reads, they double WA relative to Tiering without significantly helping to curb the number of runs that a range read has to access. Hence, SCLL is best suited for WPI workloads. We return to this point in Section 5.

4 THE LOG-STRUCTURED MERGE-BUSH

While SCLL enables new and more scalable trade-offs between WA, memory and point reads, is it possible to do even better? To gain insight, Figure 5 illustrates a cumulative breakdown of how the different cost metrics emanate from different levels with SCLL[4]. The vertical distance between two adjacent points i and $i-1$ on a given curve represents the percentage contribution of the i^{th} level to the total cost of the corresponding metric. We observe that while minor compaction overheads emanate equally from Levels 1 to $L-1$, smaller levels contribute negligibly to point read cost and memory. In this way, the cost emanation asymmetry discussed in Section 2 still lingers with SCLL for Levels 1 to $L-1$. 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.* To address this cost asymmetry, we now introduce LSM-bush, a generalization of CLL that curbs the overhead of minor compactions as the data grows by setting increasing capacity ratios between smaller levels.

High-Level Design. Figure 6 illustrates the high-level design. Similarly to CLL, LSM-bush performs Tiered merging at Levels 1 to $L-1$ and Leveled merging at Level L. It also uses top-down capacity determination, and it allows tuning the capping ratio C independently to be able to restrict major compaction costs. The core innovation is that LSM-bush introduces a new parameter called the *growth exponential* X to allow growing the capacity ratios between pairs of adjacent smaller levels. As formalized in Equation 8, for all Levels 1 to $L-1$, the capacity ratio r_i at Level i is greater by a power of X than the capacity ratio at Level $i+1$. As a 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.

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

Level Capacities. To derive the cost properties of LSM-bush, we first formalize its structure. Equation 9 denotes N_i as the capacity at Level i, derived in Appendix A by observing that it is smaller than Level L by a factor of the inverse product of the capacity ratios between these levels.

$$N_i = \begin{cases} \frac{N}{C+1} \cdot \left(\frac{T}{r_i}\right)^{\frac{1}{X-1}} \cdot \frac{r_i - 1}{r_i}, & 1 \le i \le L-1 \\ N \cdot \frac{C}{C+1}, & i = L \end{cases} \tag{9}$$

Number of Levels. We derive the number of levels in Appendix B by observing that the largest level is larger than the buffer by a factor of the product of all capacity ratios.

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

Runs at Each Level. Equation 11 denotes a_i as the maximum number of runs at Level i. For Levels 1 to $L-1$, Level i gathers at most $r_i - 1$ runs from Level $i-1$ before reaching capacity (the r_i^{th} 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.

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

Bloom Filters. Equation 5 denotes p_i as the FPR assigned to filters at Level i. Derived in Appendix C, 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. Equation 12 is defined for $p < \frac{C+1}{C} \cdot a_L$.

[4]While the figure is drawn for an instance with $T = 12$ and $L = 10$, the shapes of the curves look similar for any instance of SCLL.

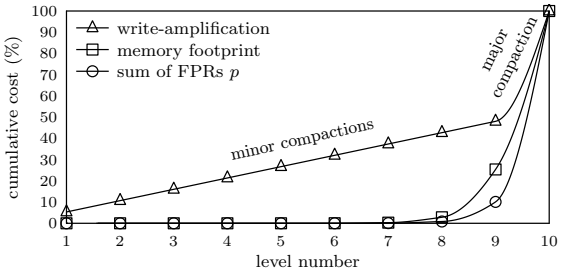

Figure 5: Minor compactions significantly increase WA while yielding exponentially diminishing returns for point reads and memory.

$$p_i = \begin{cases} \frac{p}{a_i} \cdot \frac{1}{C+1} \cdot \frac{r_{i-1}}{r_i} \cdot \left(\frac{T}{r_i}\right)^{\frac{1}{X-1}}, & 1 \le i \le L-1 \\ \frac{p}{a_L} \cdot \frac{C}{C+1}, & i = L \end{cases} \quad (12)$$

Assuming CLL. While Equations 9, 10 and 12 are undefined when the growth exponential X is set to exactly one (e.g., due to division by zero), we note that as X approaches one and becomes arbitrarily close to it, these equations approach and converge to Equations 2, 3, and 5 respectively from the previous section on CLL. The same applies to all equations in this section. In this way, LSM-bush is a superset of CLL.

Quadratic LSM-bush. We now continue to analyze the cost properties of LSM-bush. Since LSM-bush is extremely versatile structurally, we highlight a particular instance called *Quadratic* LSM-bush (QLSM-bush), which fixes the growth exponential X to 2. Thus, for Levels 1 to $L-1$, the number of runs at Level i is the *square* of the number of runs at Level $i+1$. We use QLSM-bush to enable a comparison of cost complexities against other designs. We also study the trade-offs provided by other values of the growth exponential.

Memory. The memory footprint is derived by summing up the equation for a Bloom filter's memory across all levels (i.e., $\sum_{i=1}^{L} (B \cdot N_i \cdot \ln(1/p_i))/\ln(2)^2$). While this expression is harder to simplify for LSM-bush into closed-form than it is for CLL (we discuss why in Appendix D), we can derive its complexity. We recall that the capacity at smaller levels, the term $B \cdot N_i$, decreases at a doubly-exponential rate. On the other hand, the number of bits per entry for smaller levels, the term $\ln(1/p_i)/\ln(2)^2$, increases at an exponential rate. 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 (particularly the largest three levels[5]). We summarize their memory complexity as $O(\ln(\frac{T^{1/C}}{p}) + X)$ bits per entry to highlight the impact of all four parameters on memory. As this

[5]The memory (in bits) needed for the largest three levels with respect to Equations 12 and 9 is $O(N \cdot \ln(\frac{1}{p}))$, $O(\frac{N}{C} \cdot \ln(\frac{C \cdot T}{p}))$ and $O(\frac{N}{C \cdot T} \cdot \ln(\frac{C \cdot T^X}{p}))$ while the smaller levels' footprints get dominated by these terms. We sum up these terms to an overall complexity of $O(\ln(\frac{T^{1/C}}{p}) + X)$ bits per entry.

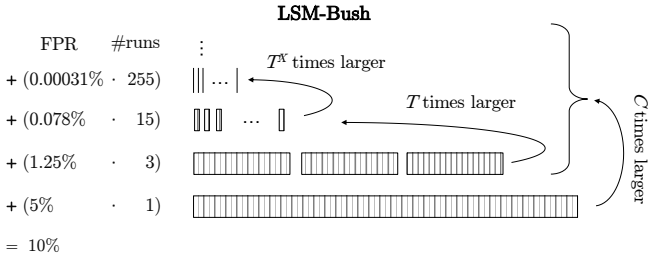

Figure 6: LSM-bush sets increasing capacity ratios between smaller levels (T is set to 4, C is set to 1, p is set to 0.1 or 10%, and X is set to 2 in this figure).

expression is independent of the data size, *the number of bits per entry for LSM-bush needed to guarantee a given point read cost does not increase as the data grows.*

Write-Amplification. On average, an entry participates in one minor compaction at each of Levels 1 to $L-1$ and in $O(C)$ major compactions at Level L. WA is therefore $O(C+L)$. For QLSM-bush, this can be expressed as $O(C + \log_2 \log_T N/F)$.

Range Reads. We derive range read cost by analyzing the number of runs across all levels. When the growth exponential X is close to 1, LSM-bush becomes identical to CLL. In this case, the number of runs is $O(1 + T \cdot (L-1))$, i.e., 1 run at Level L and $O(T)$ runs at each of Levels 1 to $L-1$. For higher values of X, the number of runs a_1 at Level 1 quickly comes to dominate the overall number of runs in the system as it contains doubly-exponentially more runs than at larger levels. Thus, a range read generally has to access $O(1 + T \cdot (L-1) + a_1)$ runs. The complexity for a_1 can be analyzed using Equation 8 as $O(T^{X^{L-2}})$. By subbing in L from Equation 10, we can simplify to $O((\frac{N}{F \cdot C})^{\frac{X-1}{X}} \cdot T^{\frac{1}{X}})$. For QLSM-bush where X is set to 2, this further simplifies to $O(\sqrt{(N \cdot T)/(F \cdot C)})$.

Bloom Filter CPU Overheads. As LSM-bush can contain a large number of runs at its smaller levels on account of merging more lazily, having to perform 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. As a hash table requires more bits per entry than a Bloom filter does, in Appendix F we show (1) how to pick which levels to use a hash table for such that most filter checks are eliminated while the memory footprint stays modest, and (2) how to index probabilistic key signatures rather than actual keys in each hash table to restrict its size while at the same time ensuring that the expected number of false positives per point read stays fixed.

Comparison to SCLL. We summarize the properties of QLSM-bush against SCLL in Figure 8. As we have seen,

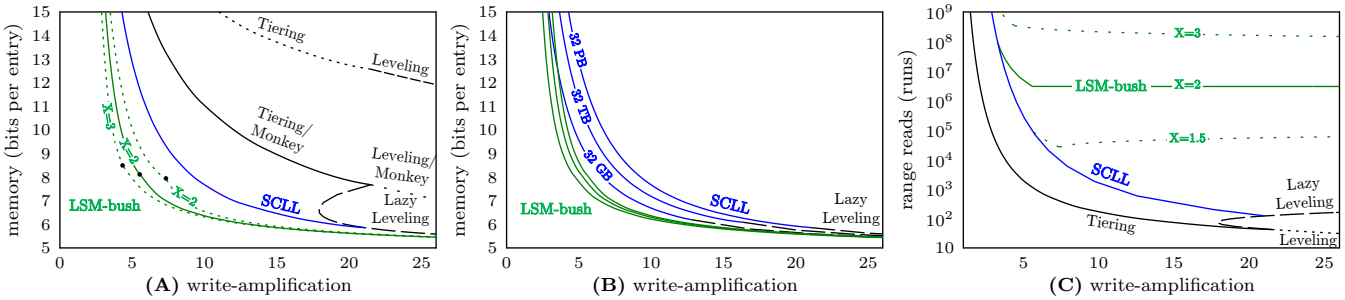

Figure 7: LSM-bush is more scalable than SCLL for WPI workloads at a further expense to range reads.

SCLL exhibits a decreasing memory footprint with respect to the data size (i.e., by pegging the capping ratio to the number of levels[6]). Furthermore, SCLL is able to trade the gain in memory to reduce WA down to $O(\sqrt{\ln(N/F)})$ while holding point reads and memory fixed as the data grows. QLSM-bush, on the other hand, offers a superior write cost of $O(C + \log_2 \log_T N/F))$ while holding point reads and memory fixed as the data grows. For WPI workloads, this is a net improvement over SCLL. As we show next, QLSM-bush can trade the gain in WA for either memory or point reads by co-tuning its knobs.

QLSM-Bush Pareto Frontier. In Figure 7 Part (A), the solid green curve represents the best memory/merge trade-offs that QLSM-bush enables by co-tuning the base ratio T and the capping ratio C. The black dot indicates the point where T is set to 2 and C is set to 1. To the right of each dot, we increase T while setting C to $T - 1$. While this increases the overhead of major compactions and thus WA, it increases the proportion of data at the largest level and thus reduces memory. To the left of the dot, we increase T while fixing C to 1. While this decreases the overhead of minor compactions and thus WA, it increases the memory requirement as there are more runs in the system. In this way, QLSM-bush can be tuned to assume different memory/merge trade-offs along a new Pareto frontier that dominates SCLL. In Figure 7 Part (B), we show that this curve moves outwards more slowly than SCLL's curve as minor compaction overheads increase more slowly with QLSM-bush. In this way, QLSM-bush offers increasingly dominating memory/merge trade-offs than existing designs as the data grows. As any gain in memory can be traded for point read cost by holding memory fixed, QLSM-bush allows to trade among all three metrics along a superior three-dimensional Pareto frontier, of which we only illustrate two dimensions in the figure. We return to this point with experiments in Section 6.

[6]We experimented with pegging the capping ratio C to the number of levels L with QLSM-bush as well and observed that in practice this leads to worse trade-offs (e.g., compared with fixing C to 1) as the gain in memory is slow and does not offset the increase in major compaction overheads.

Farther Pareto Frontier. Figure 7 Part (A) illustrates two more instances of LSM-bush with different tunings of the growth exponential X (1.5 and 3). We observe that each value of the growth exponential defines a distinct Pareto frontier. Higher values lead to superior frontiers, though each subsequent frontier incurs diminishing returns as minor compaction overheads decrease more slowly as X increases. The trade-off is that with higher values of X, range reads become more expensive as there are more runs in the system. We illustrate this in Figure 4 Part (C), which compares the range/write trade-offs that these instances of LSM-bush enable. In this way, LSM-bush allows optimizing more for WPI workloads by using higher values of X or optimizing more for workloads with range reads using lower values of X.

5 THE WACKY CONTINUUM

As we have seen throughout the paper, different designs enable a wide range of trade-offs between the costs of writes, point reads, range reads, and memory. For example, while LSM-bush enables the best point/write/memory trade-offs, the Leveled and Tiered LSM-tree designs delineate the best range/write trade-offs (shown in Figure 7). Overall, no single point in the design space performs as well as possible across all workloads. This begs the question of how to pick the best design and tuning for a particular application workload?

To answer this question, we introduce Wacky, a generalization that can assume any of the Bloomptimized designs discussed in this paper within a single implementation. Wacky is a design continuum [31] with a small and finite set of knobs that can be tweaked to transition across designs. It further includes cost models that predict how changing each knob would impact overall system behavior. As a result, Wacky's design space can be searched analytically and navigated in small, judicious, and informed steps to converge to the instance that performs best in practice.

Controlling Merging Within Levels. Wacky inherits the LSM-bush design space as a starting point and augments it with the ability to assume Bloomptimized Tiered and Leveled LSM-tree designs. The goal is to also be able to optimize for range reads. To this end, Wacky adds the ability to control

		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)	$O(p)$	$O(p)$	$O(p)$	$O(p)$	$O(p)$	$O(p)$
	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)	$O(\ln \frac{T^{1/C}}{p})$	$O(\ln \frac{T^{1/L}}{p})$	$O(\ln \frac{T^{1/C}}{p})$	$O(\ln \frac{T}{p})$	$O(\ln \frac{1}{p})$	$O(\ln \frac{1}{p})$
	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.

the greediness of merge operations *within* levels (similarly to Dostoevsky [23]). Each level consists of one *active run* and zero or more *static runs*. Incoming data into a level gets merged into the active run. Once this run reaches a configurable fraction of the level's capacity, called the *merge threshold*, it becomes static and a new active run is initialized. When the merge threshold is set to 1, we have Leveled merging. When it is set to $1/(r_i-1)$, we have Tiered merging.

Wacky allows to fine-tune the merge threshold across different levels by introducing two new parameters: K controls the merge threshold at Levels 1 to $L-1$ while Z controls the merge threshold at Level L. Equation 13 denotes a_i as the maximum number of runs that each level can have in terms of these parameters. Generally, the merge threshold for Level i is $a_i/(r_i-1)$, where a_i comes from Equation 13. In this way, when K and Z are both set to zero, we have Leveled merging at all levels. When they are both set to one, we have Tiered merging across all levels. Wacky allows to transition between Leveling and Tiering at different levels in small steps by tweaking these knobs.

$$a_i = \begin{cases} (r_i-1)^K, & 1 \le i \le L-1 \\ C^Z, & i = L \end{cases} \tag{13}$$

Structural Universality. Wacky inherits the parameters X, T and C from LSM-bush to set capacity ratios across levels. As a result, it is able to use Equations 9 and 10 from Section 4 out-of-the-box to determine the overall number of levels as well as individual level capacities with respect to these parameters. To set assign Bloomptimized FPRs to its filters, Wacky uses Equation 12 from Section 4 while plugging in the number of runs at each level from Equation 13. In this way, Wacky has a total of five parameters, K, Z, X, T and C, which fully dictate the overall structure. At the top part of Figure 8, we show how to set each of these parameters to assume any of the designs discussed so far.

Searchable Cost Model. We now continue to derive a generalized cost model for Wacky with the goal of being able to search its design space for the best design for a given application. While it proves difficult to derive a precise closed-form model for each of the metrics with respect to such a versatile design space, we instead derive models that are fast to compute in terms of the structural properties of Wacky, which Wacky inherits in closed-form as Equations 9, 10, 12 and 13.

Write Cost. We derive the amortized I/O cost of an application write by dividing write-amplification by the block size B as each write I/O during a merge operation handles B entries. To model WA across the different levels, we observe that each entry gets copied an average of $\frac{C}{a_L}$ times at Level L and an average of $\frac{r_i-1}{a_i+1}$ times at Level i. This model 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 [23]. As a result, a proportion of $1/r_i$ of each levels' capacity skips Level i and thus discounts write cost.

$$W = \frac{1}{B}\cdot\left(\frac{C}{a_L} + \sum_{i=1}^{L-1}\frac{r_i-1}{a_i+1}\right) \tag{14}$$

Point Reads. Equation 15 denotes R_{zero} as the I/O cost of a point read to a non-existing entry and R as the cost of a worst-case point read (to an entry at the largest level). R_{zero} incurs an average of $\sum_{i=1}^{L} p_i \cdot a_i$ false positives, and so its average I/O cost is equal to p, the sum of all FPRs. On the other hand, R entails (1) one I/O to fetch the target entry from Level L, (2) an average of $p - p_L \cdot a_L$ false positives while searching Levels 1 to $L-1$, and (3) an average of $\frac{p_L \cdot a_L}{2}$ false positives while searching Level L (this assumes the target entry is on average in the middle run at Level L). We sum up these three terms to obtain the expression for R in Equation 15, which predicts the I/O cost of point reads given the Bloom filters' assignment.

$$\begin{aligned} R_{zero} &= p \\ R &= 1 + (p - p_L \cdot \frac{(a_L+1)}{2}) \end{aligned} \tag{15}$$

Range Reads. We model the I/O cost of range reads V as the total number of runs in the system: $V = \sum_1^L a_i$.

Finding the Best Design. To search Wacky for the best design, we model the average worst-case operation cost Θ in Equation 16 based on the proportion of different types of operations in the workload. In particular, we denote r, z, w and v as the proportion of point reads, zero-result point reads, writes, and range reads, respectively, and we multiply each of them by the corresponding I/O cost from the equations derived above. Equation 16 allows iterating over different configurations in search of the design that minimizes the average operation cost.

$$\Theta = r \cdot R + z \cdot R_{zero} + w \cdot W + v \cdot V \tag{16}$$

Tractable Search. As searching all combinations of Wacky's parameters can be computationally intensive, we propose a practical approach is to fix some of the parameters as in Figure 8 so as to search projections of the space. For each projection, we iterate over the base ratio T to enumerate its different trade-offs. Since the cost properties for the different metrics become increasingly extreme as we increase T, the design that strikes the best balance usually occurs for values of T with fewer than three digits. We therefore use an iterative approach that continues to increment T until finding a local minimum for the average operation cost Θ. We then pick the design from across all projections that minimizes the average operation cost Θ. This approach allows finding a good design that approximates the optimal choice for a given workload in a fraction of a second. Future directions include (1) extending the cost models to take workload idiosyncrasies into account (e.g., skew), (2) weighting the different costs based on hardware characteristics (e.g., sequential vs. random access), and (3) adapting during runtime as workloads change to identify the best design [30].

6 EVALUATION

We now evaluate Wacky against existing designs, and we highlight how it uses SCLL and LSM-bush to enable better and more scalable cost trade-offs.

Implementation. We implemented Wacky on top of RocksDB [28], an extensible open-source KV-store. At its core, RocksDB is a Leveled LSM-tree, though it also supports an API that enables a client application to select and merge runs using customizable user-programmed logic. We used this API to implement Wacky by triggering merging *across* and *within* levels based on Equations 8 and 13, respectively. By default, RocksDB maintains an in-memory Bloom filter for every run, and it sets the same FPR to runs at all levels. We extended RocksDB to allow setting FPRs to different levels based on Equation 12. **Baselines.** We compare Wacky against seven baselines to represent different parts of the design space. In particular, we use plain Leveled and Tiered LSM-trees with uniform FPRs to reflect designs in industry, as well as Leveled, Tiered and Lazy Leveled designs with Bloomptimized filters to reflect the toughest competition from research. We tune all these designs with a capacity ratio of 10 (i.e., the default capacity ratio in RocksDB).

How to Evaluate Wacky. As Wacky can assume numerous forms with different cost properties, we search the Wacky design space prior to each experiment for the best design given the application setting (i.e., the workload, data size, and memory budget). We do so using the analytical method outlined in Section 5 to iterate over different values of the base ratio T for each of the five projections of Wacky from Figure 8 and picking the design that minimizes the average operation cost. We also evaluate SCLL ($T = 30$) and QLSM-bush ($T = 2, C = 1$) to focus in more detail on the new parts of the design space introduced in this paper.

Default Workload and Setup. Each experimental trial commences by default from a fresh clone of a 256GB data set consisting of 128B entries. To focus on worst-case performance, we measure throughput across workloads consisting of uniformly randomly distributed insertions and reads. Since the cost of an application write is incurred indirectly after the buffer flushes, we measure write cost by running multiple writes across long time windows and measuring the number of writes the system was able to process. We ensure that the time windows are long enough in each experiment to account for the full amortized write cost (e.g., by waiting for at least one major compaction to take place). We assign a default budget of 5 bits per entry to the Bloom filters of each baseline. In this way, we allow memory-optimized designs to manifest their improvement in terms of superior point read performance.

Experimental Infrastructure. We use a machine with a 2TB SSD connected through a PCI express bus, 32GB DDR4 main memory, four 2.7 GHz cores with 8MB L3 caches, running 64-bit Ubuntu 16.04 LTS on an ext4 partition. For all baselines, we used half of the threads for merging and the other half to each issue reads and writes.

Point Reads vs. Writes. In Figure 9 Parts (A) and (B), we use a workload consisting of point reads and insertions, and we increase the proportion of point reads on the x-axis from 0% to 100%. Part (A) uses only zero-result point reads, while Part (B) uses only worst-case point reads where the target entry is at the largest level. On the y-axis, we measure throughout by normalizing it for all baselines with respect to Wacky to enable a clear comparison. We report the actual throughput that Wacky achieves for each workload above the top x-axis (thousands of read/write operations per second).

First, we observe that the plain Leveled and Tiered designs perform sub-optimally across the board because they do

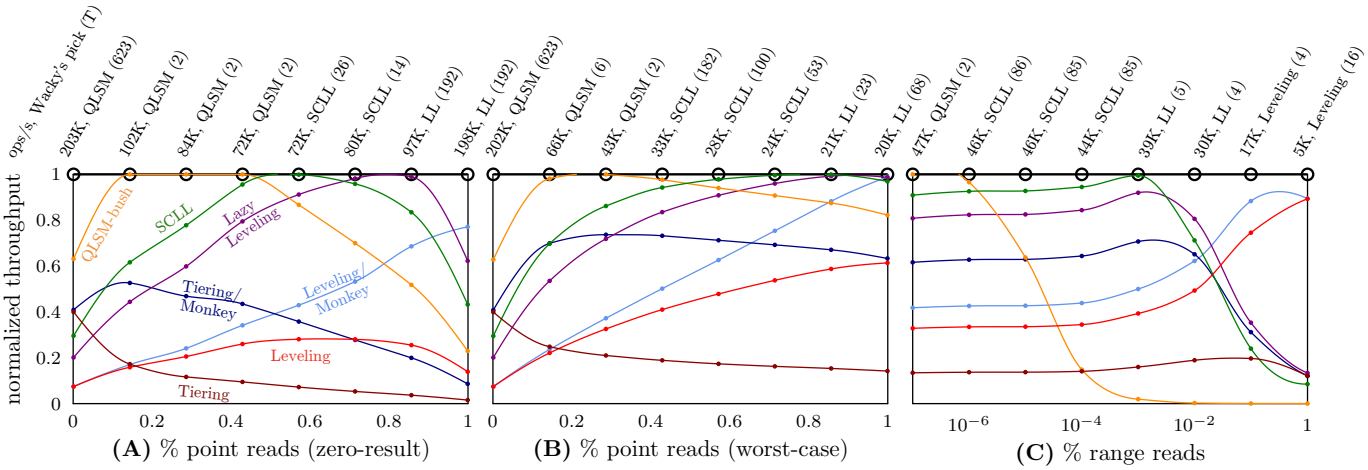

Figure 9: Wacky chooses the design that maximizes throughput for every workload, and it leverages SCLL and QLSM-bush to enable superior performance for WPI workloads.

not allocate their memory in a manner that minimizes the sum of FPRs. Point reads therefore incur significantly more false positives and thus I/O than with the other designs. Monkey/Tiered and Monkey/Leveled also do not exhibit the best possible performance for any of the workloads. The reason is that the Tiered Monkey variant incurs many false positives at the largest level as it contains many runs, while the Leveled Monkey variant suffers from high compaction overheads across all levels. When the workload is write-dominated, the best design is LSM-bush as it optimizes the most for writes. As the proportion of reads increases, the best design becomes SCLL and then Lazy Leveling, each of which optimizes more for reads by merging more greedily.

On top of the figure, we report the design that Wacky picks for each workload point as well as the chosen base ratio T, which fine-tunes each design. While no fixed design is best for every workload, Wacky's ability to search for the best design allows it to either match or outperform each of the fixed designs across the board.

Range Reads vs. Writes. Figure 9 Part (C) uses a mixed workload with 60% writes and 40% reads and where the x-axis varies the proportion between point reads and range reads (the range reads are all short accessing ≈ 1 block per run, while the point reads are 50% zero-result and 50% targeting an entry at the largest level). Similarly to the previous experiments, no single design performs well across all workload combinations. As we increase the proportion of range reads, it pays off to merge more greedily to restrict the number of runs that need to be accessed. The optimal choice therefore switches from QLSM-bush to SCLL, then to Lazy Leveling, and finally to Leveling. Wacky dominates all fixed designs across the board by being able to assume the best design for each workload point.

Data Size Scalability. Figure 10 Part (A) shows how the different designs scale as the data size grows. We fix the workload to 60% insertions and 40% point reads. Half of the point reads are zero-result and half target entries at the largest level. The key take-away is that the optimal data structure choice changes with respect to the data size. Wacky first uses SCLL, but as the data size grows the merge overheads of SCLL grow too. Therefore, Wacky switches from SCLL to QLSM-bush to allow itself to scale better as the data size continues to grow. In this way, Wacky scales better than existing systems by (1) having fundamentally more scalable designs to chose from, and (2) knowing how to choose and adopt the best design for the workload.

Memory Scalability. Figure 10 Part (B) uses the same workload from Part (A) while varying the memory budget for the Bloom filters on the x-axis from 0 to 20 bits per entry. With 0 bits per entry, Wacky uses Leveling as in this case each read accesses every run. The merge greediness of Leveling therefore pays off. As the memory increases, however, Wacky switches to using SCLL and then to increasingly more write-optimized variants of QLSM-bush. A crucial observation is that for most of the designs, performance eventually flattens out as the number of false positives becomes negligible. On the other hand, Wacky is able to continue improving throughout with respect to memory. The reason is that as memory increases, Wacky picks more write-optimized designs as the increasingly powerful Bloom filters make up for having more runs in the system. In this way, Wacky scales better with respect to memory.

The figure includes a dashed curve labeled SK corresponding to SkimpyStash [25], which represents log-structured hash-table (LSH-table) designs. SkimpyStash (1) logs entries in storage, (2) maps them using a hash table in memory, and (3) chains entries in storage belonging to the same hash

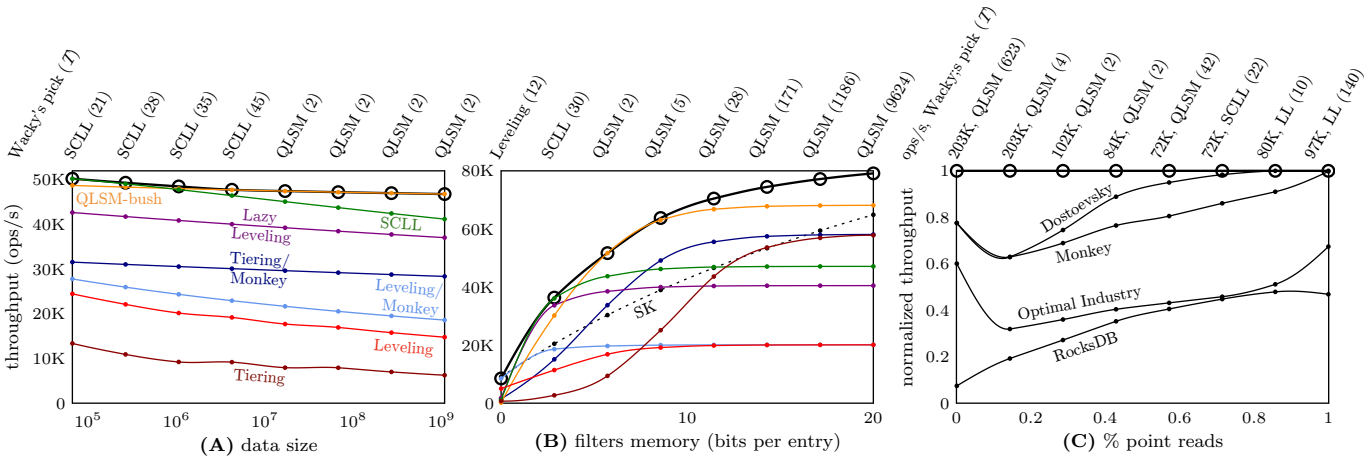

Figure 10: Wacky scales better with respect to data size (A) and memory (B), and as its design space is more versatile it contains better design choices, especially for write-heavy workloads (C).

bucket as a linked list. As memory increases, more hash buckets fit in memory and so the average linked list length decreases. Point reads therefore cost fewer I/Os. Nevertheless, SkimpyStash requires ≈ 40 bits per entry to reduce the average linked list length to one. With 20 or fewer bits per entry as with this experiment, each point read for SK entails multiple I/Os and thus compromises throughput. The general observation is therefore that Wacky dominates LSH-table designs such as SK when memory is scarce.

Systems Comparison. In Figure 10 Part (B), we compare Wacky to Monkey and Dostoevsky as we vary the proportion between writes and point reads in the workload (half of the point reads are zero-result while the other half target the largest level). Monkey includes Bloomptimized Tiered and Leveled designs, while Dostoevsky is a superset of Monkey that also includes Lazy Leveling. We search each of their design spaces and fine-tune the base ratio prior to every experimental trial as we do for Wacky, and we use the analytically chosen instance for the experiment. The figure also includes plain Leveling to represent the default behavior of RocksDB, as well as a more general curve labeled Optimal Industry, which picks the best plain Tiered or Leveled design (with uniform FPRs across all levels) and fine-tunes the base ratio. This experiment demonstrates by how much the inclusion of SCLL and QLSM-bush in the design space of Wacky allows it to push performance relative to the best design that each of the other design spaces includes. Dostoevsky provides the toughest competition. We observe that Wacky significantly outperforms Dostoevsky for write-intensive workloads as it is able to use SCLL and QLSM-bush to optimize far more for writes than any design within Dostoevsky.

Summary. Overall, the experiments show that while each fixed design may be best for one particular application setting, no fixed design performs well universally for all workloads, data sizes, or memory budgets. In this way, we demonstrate the potential of Wacky as a versatile and searchable design continuum that can identify and use the best design instance for the target application. We further demonstrate that SCLL and QLSM-bush allow Wacky to outperform and scale better than existing designs for WPI workloads with respect to both memory and data size.

7 RELATED WORK

Over the past decade, a growing body of work has emerged with the goal of optimizing LSM-tree based KV-stores for the increasing write rates of applications. In this section, we position LSM-bush along this arch.

LSM-Tree. With write-intensive workloads, LSM-tree's merge operations become a performance bottleneck, and so a lot of complementary work from the past ten years focuses on mitigating merge overheads. A widespread approach is to partition individual runs into multiple independent files and to only merge files with a small key range overlap with files in the next level [11, 55, 57]. Other approaches store the value components of all entries outside of LSM-tree to avoid repeatedly merging them [17, 42]. Accordion packs entries more densely into the buffer to reduce the frequency with which the buffer flushes and triggers merging [13], and TRIAD proposes to keep frequently updated entries in the buffer to avoid their continual inclusion in merging [11]. One recent design opportunistically merges parts of runs that have recently been scanned and are already in memory [47]. Several recent approaches use Tiered merging while controlling the data's layout within a level [48] or using cached Bloom filters [60] or Cuckoo filters [51] to keep the

access cost of point reads low. In this work, we show that some many operations in existing LSM-tree designs are fundamentally unimpactful and that we can remove them by applying increasing capacity ratios across smaller levels to scale write cost better than with existing LSM-tree designs.

LSH-Table. The Log-structured hash table (LSH-Table) logs entries in storage and maps their locations in memory using a hash table. In this way, it exhibits optimum write performance at the expense of having a high memory footprint for the hash table. Design variants such as BitCask store all keys in memory [54], though most modern designs use a smaller key-signature (e.g., 1-2 bytes per entry) to represent each entry in the hash table [2, 3, 18, 24]. To be able to function with little memory, some variants allow having k fewer hash buckets than data entries, and so data entries within the log that map to the same bucket are chained as a linked list in storage [18, 25]. As a result, every point read is more expensive as traversing a linked list costs $O(k)$ I/Os. In this way, LSH-table designs generally require ample memory to perform well while the designs we focus on in this work perform better when memory is scarce. In our work on design continuums [32], we show a path towards unifying the design spaces of LSM-tree and LSH-table to be able to perform as well as possible under any memory constraints. Furthermore, our design decision in Section 4 to use hash tables for smaller levels of LSM-bush is inspired by LSH-table designs. Smaller levels of LSM-bush can in fact be viewed as a series of LSH-tables that evolve as a bush.

LSM-Tree designs with fractional cascading embed fence pointers within the runs in storage to as opposed to storing them in memory [12, 39]. Read cost is generally $O(\log_T(N/F))$ I/O and write cost is $O(\frac{T}{B} \cdot \log_T(N/F))$ I/O, where T is the capacity ratio between levels. Such designs exhibit the same asymptotic properties as Buffer-tree. Our focus in this work is rather on data structures that use more memory to obtain cheaper point reads in $O(1)$.

B-Tree. BerkeleyDB uses B-tree to persist data in storage [45]. B-tree generally performs reads and writes in $O(\log_B(N))$ I/O if only the root node is in memory or in $O(1)$ I/O if all internal nodes are in memory. While B-tree is a part of the read/write/memory trade-off spectrum considered in this paper, our focus here is on the more write-optimized part of the spectrum to facilitate write-intensive applications.

Buffer-tree is a write-optimized B-tree variant for which each internal node contains a write buffer that gets spilled onto its children when it fills up [7, 14, 16]. With α being the number of children each node has and the rest of the space being used as buffer, Buffer-tree performs writes in $O(\frac{\alpha}{B} \cdot \log_\alpha(N))$ I/O while read cost is $O(\log_\alpha(N))$ I/O. While it is also a part of the read/write/memory trade-off spectrum that we consider, our focus in this work is on how to best leverage memory to achieve point reads in $O(1)$ I/O.

Unbounded 2-Level Designs. Some KV-store designs log a fixed number of data batches and then merge them into a single larger run [9, 40]. With such designs, the cost of merging into the larger run grows linearly with respect to data size. In contrast, LSM-bush fixes the capacity ratio between the largest and second largest levels to prevent write cost from growing linearly in this way.

Further KV-Store Paradigms. We examine other related KV-store designs and data models in Appendix H.

8 CONCLUSION

We show that existing key-value stores backed by an LSM-tree exhibit deteriorating performance/memory trade-offs as the data grows. The reason is that compaction overheads at smaller levels grow without significantly benefiting point reads or memory. We introduce LSM-bush, a new data structure that eliminates non-impactful compactions by setting increasing capacity ratios between smaller levels so that newer data is merged more lazily. In this way, LSM-bush improves the scalability of writes, and it can trade this gain for either point reads or memory to enable more scalable trade-offs all around. We embed LSM-bush within Wacky, a design continuum that generalizes all state-of-the-art merge policies within a single searchable implementation. Wacky can be utilized and navigated by any storage application, from key-value stores and beyond, to find the best and most scalable design for any workload.

Acknowledgments. We thank the anonymous reviewers for their valuable feedback, and we thank Zichen Zhu for helping with implementation. This work is supported by the National Science Foundation under grant IIS-1452595.

REFERENCES

[1] Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J. D., Manasse, M., and Panigrahy, R. Design Tradeoffs for SSD Performance. *ATC* (2008).
[2] Anand, A., Muthukrishnan, C., Kappes, S., Akella, A., and Nath, S. Cheap and Large CAMs for High Performance Data-Intensive Networked Systems. *NSDI* (2010).
[3] Andersen, D. G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., and Vasudevan, V. FAWN: A Fast Array of Wimpy Nodes. *SOSP* (2009).
[4] Apache. Accumulo. *https://accumulo.apache.org/*.
[5] Apache. Cassandra. *http://cassandra.apache.org*.
[6] Apache. HBase. *http://hbase.apache.org/*.
[7] Arge, L. The Buffer Tree: A Technique for Designing Batched External Data Structures. *Algorithmica 37*, 1 (2003), 1–24.
[8] Armstrong, T. G., Ponnekanti, V., Borthakur, D., and Callaghan, M. LinkBench: a Database Benchmark Based on the Facebook Social Graph. *SIGMOD* (2013).

[9] Athanassoulis, M., Chen, S., Ailamaki, A., Gibbons, P. B., and Stoica, R. Online Updates on Data Warehouses via Judicious Use of Solid-State Storage. *TODS 40*, 1 (2015).

[10] Athanassoulis, M., Kester, M. S., Maas, L. M., Stoica, R., Idreos, S., Ailamaki, A., and Callaghan, M. Designing Access Methods: The RUM Conjecture. *EDBT* (2016).

[11] Balmau, O., Didona, D., Guerraoui, R., Zwaenepoel, W., Yuan, H., Arora, A., Gupta, K., and Konka, P. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. *ATC* (2017).

[12] Bender, M. A., Farach-Colton, M., Fineman, J. T., Fogel, Y. R., Kuszmaul, B. C., and Nelson, J. Cache-Oblivious Streaming B-trees. *SPAA* (2007).

[13] Bortnikov, E., Braginsky, A., Hillel, E., Keidar, I., and Sheffi, G. Accordion: Better Memory Organization for LSM Key-Value Stores. *PVLDB 11*, 12 (2018), 1863–1875.

[14] Brodal, G. S., and Fagerberg, R. Lower Bounds for External Memory Dictionaries. *SODA* (2003).

[15] Bu, Y., Borkar, V. R., Jia, J., Carey, M. J., and Condie, T. Pregelix: Big(ger) Graph Analytics on a Dataflow Engine. *PVLDB 8*, 2 (2014), 161–172.

[16] Buchsbaum, A. L., Goldwasser, M. H., Venkatasubramanian, S., and Westbrook, J. On External Memory Graph Traversal. *SODA* (2000).

[17] Chan, H. H. W., Li, Y., Lee, P. P. C., and Xu, Y. HashKV: Enabling Efficient Updates in KV Storage via Hashing. *ATC* (2018).

[18] Chandramouli, B., Prasaad, G., Kossmann, D., Levandoski, J. J., Hunter, J., and Barnett, M. FASTER: A Concurrent Key-Value Store with In-Place Updates. *SIGMOD* (2018).

[19] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A Distributed Storage System for Structured Data. *OSDI* (2006).

[20] Dayan, N., Athanassoulis, M., and Idreos, S. Monkey: Optimal Navigable Key-Value Store. *SIGMOD* (2017).

[21] Dayan, N., Athanassoulis, M., and Idreos, S. Optimal Bloom Filters and Adaptive Merging for LSM-Trees. *TODS (to appear* (2018).

[22] Dayan, N., Bonnet, P., and Idreos, S. GeckoFTL: Scalable Flash Translation Techniques For Very Large Flash Devices. *SIGMOD* (2016).

[23] Dayan, N., and Idreos, S. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. *SIGMOD* (2018).

[24] Debnath, B., Sengupta, S., and Li, J. FlashStore: high throughput persistent key-value store. *PVLDB 3*, 1-2 (2010), 1414–1425.

[25] Debnath, B., Sengupta, S., and Li, J. SkimpyStash: RAM space skimpy key-value store on flash-based storage. *SIGMOD* (2011).

[26] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's Highly Available Key-value Store. *SIGOPS Op. Sys. Rev. 41*, 6 (2007), 205–220.

[27] Dong, S., Callaghan, M., Galanis, L., Borthakur, D., Savor, T., and Strum, M. Optimizing Space Amplification in RocksDB. *CIDR* (2017).

[28] Facebook. RocksDB. *https://github.com/facebook/rocksdb.*

[29] Google. LevelDB. *https://github.com/google/leveldb/*.

[30] Idreos, S., Athanassoulis, Dayan, N., Guo, D., Kester, M. S., Maas, L., and Zoumpatianos, K. Past and future steps for adaptive storage data systems: From shallow to deep adaptivity. In *BIRTE* (2016).

[31] Idreos, S., Dayan, N., Qin, W., Akmanalp, M., Hilgard, S., Ross, A., Lennon, J., Jain, V., Gupta, H., Li, D., and Zhu, Z. Design continuums and the path toward self-designing key-value stores that know and learn. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)* (2019).

[32] Idreos, S., Dayan, N., Qin, W., Akmanalp, M., Hilgard, S., Ross, A., Lennon, J., Jain, V., Gupta, H., Li, D., and Zhu, Z. Design continuums and the path toward self-designing key-value stores that know and learn. In *Biennial Conference on Innovative Data Systems Research (CIDR)* (2019).

[33] Idreos, S., Zoumpatianos, K., Athanassoulis, M., Dayan, N., Hentschel, B., Kester, M. S., Guo, D., Maas, L. M., Qin, W., Wasay, A., and Sun, Y. The Periodic Table of Data Structures. *IEEE DEBULL 41*, 3 (2018), 64–75.

[34] Idreos, S., Zoumpatianos, K., Hentschel, B., Kester, M. S., and Guo, D. The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models. *SIGMOD* (2018).

[35] Jagadish, H. V., Narayan, P. P. S., Seshadri, S., Sudarshan, S., and Kanneganti, R. Incremental Organization for Data Recording and Warehousing. *VLDB* (1997).

[36] Kondylakis, H., Dayan, N., Zoumpatianos, K., and Palpanas, T. Coconut: A scalable bottom-up approach for building data series indexes. *Proceedings of the VLDB Endowment 11*, 6 (2018), 677–690.

[37] Kondylakis, H., Dayan, N., Zoumpatianos, K., and Palpanas, T. Coconut palm: Static and streaming data series exploration now in your palm. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2019).

[38] Lakshman, A., and Malik, P. Cassandra - A Decentralized Structured Storage System. *SIGOPS Op. Sys. Rev. 44*, 2 (2010), 35–40.

[39] Li, Y., He, B., Yang, J., Luo, Q., Yi, K., and Yang, R. J. Tree Indexing on Solid State Drives. *PVLDB 3*, 1-2 (2010), 1195–1206.

[40] Lim, H., Fan, B., Andersen, D. G., and Kaminsky, M. SILT: A Memory-Efficient, High-Performance Key-Value Store. *SOSP* (2011).

[41] LinkedIn. Voldemort. *http://www.project-voldemort.com.*

[42] Lu, L., Pillai, T. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. WiscKey: Separating Keys from Values in SSD-conscious Storage. *FAST* (2016).

[43] Memcached. Reference. *http://memcached.org/*.

[44] MongoDB. Online reference. *http://www.mongodb.com/*.

[45] Olson, M. A., Bostic, K., and Seltzer, M. I. Berkeley DB. *ATC* (1999).

[46] O'Neil, P. E., Cheng, E., Gawlick, D., and O'Neil, E. J. The log-structured merge-tree (LSM-tree). *Acta Informatica 33*, 4 (1996), 351–385.

[47] Pilman, M., Bocksrocker, K., Braun, L., Marroquin, R., and Kossmann, D. Fast Scans on Key-Value Stores. *PVLDB 10*, 11 (2017), 1526–1537.

[48] Raju, P., Kadekodi, R., Chidambaram, V., and Abraham, I. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. *SOSP* (2017).

[49] Raju, P., Ponnapalli, S., Kaminsky, E., Oved, G., Keener, Z., Chidambaram, V., and Abraham, I. mlsm: Making authenticated storage faster in ethereum. In *Proceedings of the USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)* (2018).

[50] Redis. Online reference. *http://redis.io/*.

[51] Ren, K., Zheng, Q., Arulraj, J., and Gibson, G. SlimDB: A Space-Efficient Key-Value Storage Engine For Semi-Sorted Data. *PVLDB 10*, 13 (2017), 2037–2048.

[52] Rhea, S., Wang, E., Wong, E., Atkins, E., and Storer, N. Littletable: A time-series database and its uses. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2017).

[53] Sears, R., and Ramakrishnan, R. bLSM: A General Purpose Log Structured Merge Tree. *SIGMOD* (2012).

[54] Sheehy, J., and Smith, D. Bitcask: A Log-Structured Hash Table for Fast Key/Value Data. *Basho White Paper* (2010).

[55] Shetty, P., Spillane, R. P., Malpani, R., Andrews, B., Seyster, J., and Zadok, E. Building Workload-Independent Storage with VT-trees. *FAST* (2013).

[56] Tarkoma, S., Rothenberg, C. E., and Lagerspetz, E. Theory and Practice of Bloom Filters for Distributed Systems. *IEEE Communications Surveys & Tutorials 14*, 1 (2012), 131–155.
[57] Thonangi, R., and Yang, J. On Log-Structured Merge for Solid-State Drives. *ICDE* (2017).
[58] Vo, A.-V., Konda, N., Chauhan, N., Aljumaily, H., and Laefer, D. F. Lessons learned with laser scanning point cloud management in hadoop hbase. In *Proceedings of EG-ICE* (2019).
[59] WiredTiger. Source Code. *https://github.com/wiredtiger/wiredtiger*.
[60] Wu, X., Xu, Y., Shao, Z., and Jiang, S. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. *ATC* (2015).

A LEVEL CAPACITIES

In this appendix, we derive Equation 9 from Section 4, which gives the *maximum stable data size* that each level can hold (i.e., before filling up and getting flushed to the next level).

Level *L*. By definition, the data size N_L at Level L is larger by a factor of C than the cumulative data size at Levels 1 to $L-1$. It follows that N_L comprises a fraction of $\frac{C}{C+1}$ of the overall data size (i.e., $N_L = N \cdot \frac{C}{C+1}$).

Levels *1* to *L-1*. For Levels 1 to $L-1$, every r_i^{th} run that arrives at Level i causes it to fill up and get flushed. Therefore, a fraction of at most $\frac{r_i-1}{r_i}$ of Level i can be full before it gets flushed. We derive the ratio by which the maximum stable data size at Level i is greater than at Level $i-1$ by multiplying the maximum stable data size at Level i by $\frac{r_i}{r_i-1}$ to obtain the full capacity at Level i, dividing by r_i to get the full capacity at Level $i-1$, and multiplying by $\frac{r_{i-1}-1}{r_{i-1}}$ to get the maximum stable capacity at Level $i-1$. Hence, the maximum stable data size at Level i is greater by a factor of $\frac{r_i}{r_i-1} \cdot \frac{1}{r_i} \cdot \frac{r_{i-1}-1}{r_{i-1}}$, or more simply $\frac{1}{r_i-1} \cdot \frac{r_{i-1}-1}{r_{i-1}}$, than the maximum stable data size at Level $i-1$. We frame this in Equation 17 with respect to the capacity at Level L and simplify to obtain N_i, which denotes the maximum stable capacity at Level i.

$$
\begin{aligned}
N_i &= \frac{N_L}{r_L} \cdot \prod_{j=i+1}^{L-1} \frac{1}{r_j - 1} \cdot \frac{r_{j-1} - 1}{r_{j-1}} && \text{framing} \\
&= \frac{N_L}{C} \cdot \frac{r_i - 1}{r_i} \cdot \prod_{j=i+1}^{L-1} T^{-X^{L-i-1}} && \text{use Eq. 8 for } r_i \text{ \& simplify product} \\
&= \frac{N_L}{C} \cdot \frac{r_i - 1}{r_i} \cdot T^{-\frac{X^{L-i-1}-1}{X-1}} && \text{apply geometric sum for ratios' exponents} \\
&= \frac{N_L}{C} \cdot \frac{r_i - 1}{r_i} \cdot \left(\frac{T}{r_i}\right)^{\frac{1}{X-1}} && \text{simplify} \qquad (17)
\end{aligned}
$$

By expressing this result in terms of the data size N, we obtain Equation 9 in Section 4 for LSM-bush, which also applies more generally to Wacky as explained in Section 5. By further setting the growth exponential X to approach its limit value of 1, we obtain Equation 2 for CLL in Section 3.

B NUMBER OF LEVELS

We derive the number of levels by observing that the largest level's capacity is larger than the buffer by a factor of the product of all capacity ratios. We simplify and solve for L.

$$
\begin{aligned}
N_L &= F \cdot \prod_{i=1}^{L} r_i && \text{framing} \\
N_L &= F \cdot C \cdot \frac{T}{T-1} \cdot \prod_{i=1}^{L-1} T^{X^{L-i-1}} && \text{use Eq. 8 for } r_i \\
N_L &= F \cdot C \cdot \frac{T}{T-1} \cdot T^{\frac{X^{L-1}-1}{X-1}} && \text{use geometric sum} \\
L &= \left\lceil 1 + \log_X((X-1) \cdot \log_T(\frac{N}{F} \cdot \frac{1}{C+1} \cdot \frac{T-1}{T}) + 1) \right\rceil && \text{rearramge} \qquad (18)
\end{aligned}
$$

C OPTIMIZING THE BLOOM FILTERS

In this Appendix, we derive the optimal false positive rates (FPRs) to assign Bloom filters at different levels. We frame the problem as a constrained multivariate optimization problem whereby the objective function to minimize is the sum of FPRs p, the constraint is the memory available, and the variables to optimize are the FPRs at each of the levels. To solve the problem, we use the method of Lagrange Multipliers (LM). We frame the problem in standard form in Equation 19, where g denotes the objective function, derived by summing at the FPRs across all levels (i.e., $p = \sum_1^L a_i \cdot p_i$), and where y denotes the constraint, derived by summing up the memory footprint across all levels (i.e., $\sum_{i=1}^{L} (B \cdot N_i \cdot \ln(1/p_i))/\ln(2)^2$).

$$
\begin{aligned}
g &= \Sigma_{i=1}^{L}(a_i \cdot p_i) - p \\
y &= B \cdot N_L \cdot \left(\frac{\ln(1/p_L)}{ln(2)^2} + \sum_{i=1}^{L-1} \frac{1}{C} \cdot \frac{r_i - 1}{r_i} \cdot \left(\frac{T}{r_i}\right)^{\frac{1}{X-1}} \cdot \frac{\ln(1/p_i)}{ln(2)^2} \right) \qquad (19)
\end{aligned}
$$

The Lagrangian $\mathcal{L}$ is defined in terms of the objective function and the constraint as $\mathcal{L} = y + \lambda \cdot g$. We differentiate the Lagrangian with respect to the FPR at the largest level p_L, set the partial derivative to zero, and rearrange in terms of p_L in Equation 20. We do the same for p_i in Equation 21.

$$p_L = \frac{1}{a_L} \cdot \frac{B \cdot N_L}{\ln(2)^2 \cdot \lambda} \qquad (20)$$

$$p_i = \frac{1}{a_i} \cdot \frac{1}{C} \cdot \frac{r_i - 1}{r_i} \cdot \left(\frac{T}{r_i}\right)^{\frac{1}{X-1}} \cdot \frac{B \cdot N_L}{\ln(2)^2 \cdot \lambda} \qquad (21)$$

Next, we equate Equations 20 and 21 to to express p_i in terms of p_L in Equation 22.

$$p_i = \frac{p_L \cdot a_L}{a_i} \cdot \frac{1}{C} \cdot (\frac{T}{r_i})^{\frac{1}{X-1}} \cdot \frac{r_i - 1}{r_i} \qquad (22)$$

We use Equation 22 to derive the value of p_L with respect to the overall sum of FPRs p in Equation 23.

$$
\begin{aligned}
p &= \sum_{i=1}^{L} p_i \cdot a_i && \text{definition of } p \\
p &= a_L \cdot p_L \cdot (1 + \sum_{i=1}^{L-1} \frac{1}{C} \cdot (\frac{T}{r_i})^{\frac{1}{X-1}} \cdot \frac{r_i - 1}{r_i}) && \text{use Eq. 22 for } p_i \\
p &= a_L \cdot p_L \cdot (1 + 1/C) && \text{sum simplifies to } 1/C. \\
p_L &= \frac{p}{a_L} \cdot \frac{C}{C+1} && \text{rearrange in terms of } p_L \qquad (23)
\end{aligned}
$$

By plugging in Equation 23 for p_L into Equation 22, we obtain the value of p_i in terms of the sum of FPRs p as well. The overall result is given in Equation 12 in Section 4. By further taking X to its limit value of 1 and plugging 1 for a_L and $T-1$ for a_i, we obtain Equation 5 for CLL in Section 3.

D MEMORY FOOTPRINT DERIVATION

In this Appendix, we derive the memory footprint needed by Bloom filters across all levels to guarantee a given point read cost, given by the sum of FPRs p across all filters. We start by focusing on the case where the growth exponential X approaches one. Equation 24 sums up the memory for filters across all levels with respect to each levels' capacity N_i and FPR p_i (derived in Appendices A and C). We simplify this sum by arranging the exponents within the logarithm in the third step as geometric and arithmetico-geometric series, both of which converge into closed-form expressions.

$$
\begin{aligned}
&\textstyle\sum_{i=1}^{L} B\cdot N_i\cdot\frac{\ln(1/p_i)}{\ln(2)^2}\\
&= -\tfrac{N\cdot B\cdot C}{C+1}\cdot\left(\ln(\tfrac{p}{C^Z}\cdot\tfrac{C}{C+1})+\tfrac{1}{C}\cdot\tfrac{T-1}{T}\cdot\textstyle\sum_{i=1}^{L-1}\left(\frac{1}{T^{L-i-1}}\cdot\ln\left(\frac{p}{a_i}\cdot\frac{T-1}{T}\cdot\frac{1}{C+1}\cdot\frac{1}{T^{L-i-1}}\right)\right)\right)\\
M &= -\tfrac{C}{C+1}\cdot\left(\ln\left(\tfrac{p}{C^Z}\cdot\tfrac{C}{C+1}\right)+\tfrac{1}{C}\cdot\tfrac{T-1}{T}\cdot\ln\left(\left(\tfrac{p}{(T-1)^K}\cdot\tfrac{T-1}{T}\cdot\tfrac{1}{C+1}\right)^{\left(\sum_{i=1}^{L}\frac{1}{T^{L-1}}\right)}\cdot\left(\tfrac{1}{T}\right)^{\left(\sum_{i=1}^{L}\frac{i}{T^i}\right)}\right)\right)\\
&= -\tfrac{C}{C+1}\cdot\left(\ln\left(\tfrac{p}{C^Z}\cdot\tfrac{C}{C+1}\right)+\tfrac{1}{C}\cdot\tfrac{T-1}{T}\cdot\ln\left(\left(\tfrac{p}{(T-1)^K}\cdot\tfrac{T-1}{T}\cdot\tfrac{1}{C+1}\right)^{\frac{T}{T-1}}\cdot\left(\tfrac{1}{T}\right)^{\frac{T}{(T-1)^2}}\right)\right)\\
&= \ln\left(\frac{1}{p}\cdot C^{\frac{Z\cdot C}{C+1}}\cdot\frac{C+1}{C^{\frac{C}{C+1}}}\cdot(T-1)^{\frac{K-1}{C+1}}\cdot T^{\frac{T}{(C+1)\cdot(T-1)}}\right)
\end{aligned}
\tag{24}
$$

The final result is expressed in terms of M, the average number of bits per entry across all filters needed to guarantee a given value of p. For CLL, Z is set to 0 while K is set to 1. This allows to further simplify into Equation 6 in Section 3.

In Equation 25, we rearrange Equation 24 in terms of the sum of FPRs p to be able to predict point read cost with respect to a memory budget M.

$$
p = e^{-M\cdot ln(2)^2}\cdot C^{\frac{Z\cdot C}{C+1}}\cdot\frac{C+1}{C^{\frac{C}{C+1}}}\cdot(T-1)^{\frac{K-1}{C+1}}\cdot T^{\frac{T}{(C+1)\cdot(T-1)}} \tag{25}
$$

We now continue to the more general case where the growth exponential X can assume values higher than one. It is harder in this case to derive a closed-form expression as the terms from across different levels not add up to well-known convergent series. Nevertheless, as discussed in Section 4, the largest three levels with Wacky always dominate the memory footprint. Therefore, we can approximate the memory requirement (to within 10%) based on the memory footprint for the largest three levels, given in Equation 26.

$$
\begin{aligned}
M &\approx M_L + M_{L-1} + M_{L-2}\\
M_L &= -\frac{N}{\ln(2)^2}\cdot\frac{C}{C+1}\cdot\ln\left(p\cdot\frac{C}{C+1}\right)\\
M_{L-1} &= -\frac{N}{\ln(2)^2}\cdot\frac{1}{C+1}\cdot\frac{T-1}{T}\cdot\ln\left(\frac{p}{a_{L-1}}\cdot\frac{1}{C+1}\cdot\frac{T-1}{T}\right)\\
M_{L-2} &= -\frac{N}{\ln(2)^2}\cdot\frac{1}{C+1}\cdot\frac{1}{T}\cdot\frac{T^X-1}{T^X}\cdot\ln\left(\frac{p}{a_{L-2}}\cdot\frac{1}{C+1}\cdot\frac{1}{T}\cdot\frac{T^X-1}{T^X}\right)
\end{aligned}
\tag{26}
$$

E SETTING CAPACITIES TOP-DOWN

In this appendix, we describe how CLL, LSM-bush and Wacky set capacities to Levels 1 to $L-1$ top-down based on the data size at Level L. After every major compaction, we use the size of the resulting run N_L to compute the overall required capacity N across all levels (i.e., $N = N_L\cdot\frac{C+1}{C}$). We compute the number of levels L based on Equation 10 with respect to N (the number of levels may stay the same, or it may grow or shrink depending on the volume if insertions/deletions in the workload). We then set capacities to the different levels with respect to these values of N and L using Equation 9. Effectively, this approach adjusts the widths of Levels 1 to $L-1$ to ensure that their cumulative data size is smaller by at least a factor of C than the data size at Level L.

F HASH TABLES FOR SMALLER LEVELS

As LSM-bush can contain a larger number of runs than with traditional LSM-tree designs, having to check a Bloom filter for every run during a point read can become a CPU bottleneck. To prevent this bottleneck, LSM-bush replaces Bloom filters at smaller levels by one hash table per level that maps from each key in the level to the physical block that contains the key's entry on a run in storage. This replaces numerous Bloom filter checks by a single hash table probe. To control memory overheads for these hash tables, they do not store actual keys but instead use a b bits hash digest for each key called a key signature, which can lead to a false positive at a rate of 2^{-b} [3, 18, 24].

Key Signature Size. The first question is how to set key signature sizes so that the expected number of false positives (and thus I/Os) does not increase. With only Bloom filters, the expected I/O cost at Level i is $p_i\cdot a_i$. To ensure this bound continues to hold, we set the size of key signatures at Level i to each be $\lceil(p_i\cdot a_i)\rceil$ bits, where p_i and a_i come from Equations 12 and 13. This ensures that the FPR for one hash table probe at Level i is the same as the sum of FPRs across all the Bloom filters that it replaces, and so point read cost remains the same. As a result, each key signature amounts to 2-3 bytes. With an additional 4-5 byte pointer to storage and some extra hash table capacity to avoid collisions, a hash table requires $\approx 64-128$ bits per entry.

Choosing Levels. As a hash table requires more bits per entry than Bloom filters, the next question is how to choose which levels to use hash tables for so as to keep memory modest while still eliminating most filter checks. To this end, we observe that smaller levels, which contain the most runs in the system, also contain far fewer entries than at larger

QLSM-bush

level	runs a_i	buffer multiple N_i / P	FPR per level $a_i \cdot p_i$	HT or BFs?	HT or BFs bits per entry	FP bits per entry
1	255	510	0.04%	HT	60.0	4
2	15	7680	0.59%	HT	56.0	4
3	3	24576	1.88%	BFs	10.8	4
4	1	32768	2.50%	BFs	7.7	4
5	1	65536	5.00%	BFs	6.2	4
totals:	275	131070	**10.00%**		**6.8**	**4**

Figure 11: A per-level overview of QLSM-bush.

levels. This means that using a hash table for these smaller levels, especially for Level 1, would increase the memory footprint by a modest amount while still eliminating most filter checks. While our design exposes this choice to the application, we observe that as a rule of thumb using Bloom filters for the largest three levels and hash tables for smaller levels is a good compromise. For example, with QLSM-bush this increases memory by $\approx 10\%$ while reducing the number of filter checks from $O(\sqrt{N})$ to 10-11 checks as there is a small number of runs at the largest levels.

Recovery. Unlike Bloom filters, hash tables are tricky to recover after a crash because they are not immutable but continue changing as more runs get added to a level. To be able to recover a hash table without relying on checkpoints or on rereading base data, we persist a small amount of metadata in storage to allow us to rebuild a hash table after recovery from scratch. In particular, whenever a new run gets created at a level that uses a hash table, we persist an additional array of hash-pointer pairs for all entries within the new run. The pointer identifies the block whereon the corresponding entry resides. For the hash, the first $\lceil \log_2(N_i \cdot B) \rceil$ bits are used to identify the matching bucket, while the remaining $\lceil \log_2(1/(a_i \cdot p_i)) \rceil$ bits comprise the key signature, which gets reinserted into the bucket during recovery along with the pointer. The size of these hash-pointer pairs is 12-13 bytes per entry, and so this does not significantly amplify storage overheads relative to base data.

G EXAMPLE OF QLSM-BUSH INSTANCE

Figure 11 provides a per-level breakdown of QLSM-bush (T=2, C=1) for a 1TB data size of 128B entries with 8B keys over a storage device with 4KB blocks. The buffer size is 8MB, and the design is tuned to assume a sum of false positive rates of 10% (at most 0.1 extra I/O per point read). For each level, the figure expresses the number of runs a_i, the capacity of each level as a multiple of the buffer size N_i/F, the false positive rate per level $a_i \cdot p_i$, whether a level uses a hash table (HT) or Bloom filters (BFs), and the total number of bits per entry needed for hash tables or Bloom filters, and fence pointers. For levels with hash tables, the number of bits per entry includes a 48 bit pointer while the rest of the bits comprise a key signature (Level 1 for LL-bush has $61 - 48 = 13$ bit key signatures). For the fence pointers, each of them consists of 16B key-pointer pair that includes one 8B key. Since a storage block can fit 4KB/128B= 32 entries and we need one fence pointer per block, the fence pointers require $(16/32) \cdot 8 = 4$ additional bits per entry. The bottom row shows statistics for the total number of runs, the total data size as a multiple of the buffer size, the overall false positive rate, and the average number of bits per entry needed across all levels

H OTHER KEY-VALUE STORE DESIGNS

In-Memory Key-Value Stores. In-memory key-value stores such as Memcached [43] and Redis [50] store the entire dataset in memory to avoid the I/O overheads of storage. In this work, we focus on applications for which the data size is much larger than the available memory.

BigTable Model. While we focus on a key-value store data model in this paper that reflects systems such as RocksDB, Voldemort, and Dynamo, Wacky may also be applied within the BigTable model for systems such as HBase, Cassandra or Cloud BigTable. The BigTable model is a sparse three-dimensional map whereon (1) data is indexed based on row ID, column ID, and timestamp, and (2) data can be divided across multiple column families, each of which is an independent physical structure. A possible performance problem that can arise within this model is that data belonging to a given row can be fragmented across multiple runs. This is a general problem for log-structured designs, and structures such as LSM-bush can exacerbate this problem by having more rows over which data can be fragmented. We can thereby leverage Wacky to use LSM-bush only for column families that are subject to puts over entire rows at a time so that data cannot be fragmented, and/or for workloads where reads are based on row+column ID pairs so that they do not require materializing a row from across multiple runs. On the other hand, we can leverage Wacky to use more merge-greedy LSM-tree designs for column-families for which rows are fragmented and there are many row+column gets.