Optimal Bloom Filters and Adaptive Merging for LSM-Trees

NIV DAYAN, MANOS ATHANASSOULIS, and STRATOS IDREOS, Harvard University, USA

In this article, we show that key-value stores backed by a log-structured merge-tree (LSM-tree) exhibit an intrinsic tradeoff between lookup cost, update cost, and main memory footprint, yet all existing designs expose a suboptimal and difficult to tune tradeoff among these metrics. We pinpoint the problem to the fact that modern key-value stores suboptimally co-tune the merge policy, the buffer size, and the Bloom filters' false-positive rates across the LSM-tree's different levels.

We present Monkey, an LSM-tree based key-value store that strikes the optimal balance between the costs of updates and lookups with any given main memory budget. The core insight is that worst-case lookup cost is proportional to the sum of the false-positive rates of the Bloom filters across all levels of the LSM-tree. Contrary to state-of-the-art key-value stores that assign a fixed number of bits-per-element to all Bloom filters, Monkey allocates memory to filters across different levels so as to minimize the sum of their false-positive rates. We show analytically that Monkey reduces the asymptotic complexity of the worst-case lookup I/O cost, and we verify empirically using an implementation on top of RocksDB that Monkey reduces lookup latency by an increasing margin as the data volume grows (50–80% for the data sizes we experimented with). Furthermore, we map the design space onto a closed-form model that enables adapting the merging frequency and memory allocation to strike the best tradeoff among lookup cost, update cost and main memory, depending on the workload (proportion of lookups and updates), the dataset (number and size of entries), and the underlying hardware (main memory available, disk vs. flash). We show how to use this model to answer what-if design questions about how changes in environmental parameters impact performance and how to adapt the design of the key-value store for optimal performance.

CCS Concepts: • **Information systems → Point lookups**; **Hierarchical storage management**; • **Theory of computation → Data structures and algorithms for data management**;

Additional Key Words and Phrases: LSM-tree, Bloom filters, key-value stores, NoSQL, system design

ACM Reference format:
Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2018. Optimal Bloom Filters and Adaptive Merging for LSM-Trees. *ACM Trans. Database Syst.* 43, 4, Article 16 (December 2018), 48 pages.
https://doi.org/10.1145/3276980

1 INTRODUCTION

LSM-Tree Based Key-Value Stores. Modern key-value stores that maintain application data persistently typically use a Log-Structured-Merge-tree (LSM-tree) [57] as their storage layer. In

This is an invited submission for the Best of ACM SIGMOD 2017 issue. It is an extended version of the ACM SIGMOD 2017 paper titled "Monkey: Optimal Navigable Key-Value Store" [31] that includes mew material on optimizing Bloom filter allocation for general workloads (Section 4.4) and holistically optimizing all design knobs of an LSM-tree based key-value store (Section 4.5). We further added Appendices C and F-L that elaborate on the analyses in the paper.

Authors' addresses: Niv Dayan, Manos Athanassoulis, Stratos Idreos, Harvard University; emails: {dayan, manos, stratos}@seas.harvard.edu.

0362-5915/2018/12-ART16 $15.00
https://doi.org/10.1145/3276980

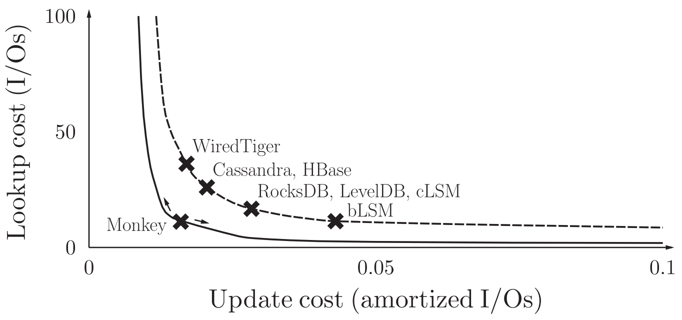


Fig. 1. Monkey is designed along a better performance tradeoff curve than the state of the art. It navigates this curve to find the design that maximizes throughput for a given application workload and hardware.

contrast to traditional storage paradigms that involve in-place updates to persistent storage, LSM-trees perform out-of-place updates thereby enabling (1) high throughput for updates [63] and (2) good space-efficiency as data is stored compactly [38] (as they do not need to keep free space at every node to allow in-place updates). They do so by buffering all updates in main memory, flushing the buffer to secondary storage as a sorted run whenever it fills up, and organizing storage-resident runs into a number of levels of increasing sizes. To bound the number of runs that a lookup probes to find a target key, runs of similar sizes (i.e., at the same level) are sort-merged and pushed to the next level when the current one becomes full. To speed up point lookups, which are common in practice [10], every run has an associated Bloom filter in main memory that probabilistically allows to skip a run if it does not contain the target key. In addition, every run has a set of fence pointers in main memory that map values to disk pages of the run (effectively maintaining min-max information for each page) and thereby allow searching a run for a target key in a single I/O. This design is adopted in a wide number of modern key-value stores including LevelDB [44] and BigTable [27] at Google, RocksDB [39, 40] at Facebook, Cassandra [51], HBase [8], AsterixDB [2] and Accumulo [6] at Apache, Voldemort [55] at LinkedIn, Dynamo [36] at Amazon, WiredTiger [70] at MongoDB, and bLSM [63] and cLSM [43] at Yahoo. Relational databases today such as MySQL (using MyRocks [39]) and SQLite4 support this design as a storage engine by mapping primary keys to rows as values.

The Problem: Suboptimal Design. In this article, we closely investigate the design space of LSM-trees and show that LSM-tree based key-value stores exhibit an intrinsic tradeoff among lookup cost, update cost, and main memory footprint (as discussed for various data structures in [13, 14, 45]). Existing systems strike a suboptimal tradeoff among these metrics. Figure 1 shows this graphically using cost models (described later) and the default configuration settings for several state-of-the-art systems as found in their latest source code and documentation. Existing systems do not strike optimal performance tradeoffs, i.e., beyond which it is impossible to improve lookup cost without harming update cost and vice versa. As a result, they cannot maximize throughput for a given main memory budget and application workload.

We pinpoint the problem to the fact that all LSM-tree based key-value stores suboptimally co-tune the core design choices in LSM-trees: the merge policy, the size ratio between levels, the buffer size, and, most crucially, the Bloom filters' false-positive rates.

The first problem is that existing designs assign the same false-positive rate (i.e., number of bits per element) to every Bloom filter regardless of the size of the run that it corresponds to. Our insight is that the worst-case point lookup cost over the whole LSM-tree is proportional to the sum of the false-positive rates of all filters. Assigning equal false-positive rates to all filters, however, does not minimize this sum. The reason is that maintaining the same false-positive rate across all runs means that larger runs have proportionally larger Bloom filters, whereas the I/O cost of probing any run is the same regardless of its size (due to each run having fence pointers in main memory that allow direct access to the relevant secondary storage page). As a result, the Bloom filters at the largest level take up the majority of the overall memory budget without yielding a significant reduction in lookup cost.

The second problem is that the relationship between the different design knobs and performance is non-linear, and so it is difficult to co-tune the various design options in the LSM-tree design space to optimize performance. For example, the available main memory budget cannot only be used to increase Bloom filter accuracy but it can also be used to increase the size of the LSM-tree buffer. Since the levels' capacities are multiples of the buffer size, this reduces the number of levels thereby improving lookup and update performance as there are fewer levels to probe and to merge entries through. However, it is unclear how to allocate a fixed main memory budget among these structures to strike the best balance. In addition, it is possible to trade between lookup cost and update cost by varying the size ratio between the capacities of levels as well as by switching the merge policy between *leveling* and *tiering*, which support one run per level vs. multiple runs per level, respectively. Here again, it is difficult to predict how a given change would impact performance given the complexity of the design space.

The Solution: Monkey. We introduce **M**onkey: **O**ptimal **N**avigable **Key**-Value Store. Monkey achieves a better performance tradeoff curve than state-of-the-art designs by using a novel analytical solution that minimizes lookup cost by allocating main memory among the Bloom filters so as to minimize the sum of their false-positive rates. The core idea is setting the false-positive rate of each Bloom filter to be proportional to the number of entries in the run that it corresponds to (meaning that the false-positive rates for smaller levels are exponentially decreasing). The intuition is that any given amount of main memory allocated to Bloom filters of larger runs brings only a relatively minor benefit in terms of how much it can decrease their false-positive rates (to save I/Os). On the contrary, the same amount of memory can have a higher impact in reducing the false-positive rate for smaller runs. We show analytically that this method shaves a factor of $O(L)$ from the worst-case lookup cost, where L is the number of LSM-tree levels. The intuition is that the false-positive rates across the levels form a geometric series, and its sum converges to a constant that is independent of L. Since the number of levels L can be expressed in more detail as $O(\log((N \cdot E)/M_{buf}))$, where N is the number of entries, E is the size of entries, and M_{buf} is the size of the buffer, shaving this term out of point lookup cost has important benefits: (1) lookup cost scales better with the number and size of entries and (2) lookup cost becomes independent of the buffer size thereby removing the contention in how to allocate main memory between the filters and the buffer. Therefore, Monkey scales better and is easier to tune.

Moreover, while existing designs are either rigid or difficult to tune points in the design space, Monkey autonomously navigates its tradeoff curve to find the optimal balance between lookup cost and update cost under a given main memory budget, application workload (lookup over update ratio), and storage device. To do so, we map the design space and environmental parameters that affect this balance, and we capture the worst-case cost of lookups and updates in a concise closed-form model. We then adapt (1) the memory allocation between the Bloom filters and buffer and (2) the merging frequency within and across levels by controlling the size ratio and the merge policy within levels, respectively. These runtime adaptations allow us to find the best balance between the costs of lookups and updates that maximizes throughput. While main-stream key-value stores expose various tuning knobs, Monkey is novel in that it allows precise navigation of the design space with predictable results, making it easy to select the optimal key-value store design for a particular workload and hardware.

Contributions. In summary, our contributions are as follows.

- In Section 3, we show that key-value stores backed by an LSM-tree exhibit a navigable tradeoff among lookup cost, update cost, and main memory footprint; yet state-of-the-art key-value stores are not tuned along the optimal tradeoff curve, because they do not allocate main memory optimally among the Bloom filters and the LSM-tree's buffer.
- In Section 4, we introduce Monkey, an LSM-tree based key-value store that optimizes lookup cost by allocating main memory among the Bloom filters so as to minimize the sum of their false-positive rates. We show analytically that this (1) improves the asymptotic complexity of lookup cost thereby scaling better for large datasets and (2) removes the dependence of lookup cost on the LSM-tree's buffer size thereby simplifying system design.
- We show how to allocate main memory among the Bloom filters to optimize for any balance between zero and non-zero-result point lookups in the workload.
- We identify the critical design knobs of the LSM-tree based key-value store design space as well as the environmental parameters that determine worst-case performance. We then use them to model the worst-case lookup and update costs as closed-form expressions. We show how to use these models to adapt the merging frequency and memory allocation to maximize the worst-case throughput.
- We also show how to use the model to answer what-if design and environmental questions. For instance, if we change (i) the main memory budget, (ii) the proportion of reads and writes in the workload, (iii) the number and/or size of data entries, or (iv) the underlying storage medium (e.g., flash vs. disk), how should we adapt the LSM-tree design, and what is the impact on performance?
- In Section 5, we evaluate Monkey using an implementation on top of RocksDB by applying a wide range of application lookup patterns (i.e., targeting existing vs non-existing keys and varying temporal localities). Monkey improves lookup latency by $50 - 80\%$ in these experiments.

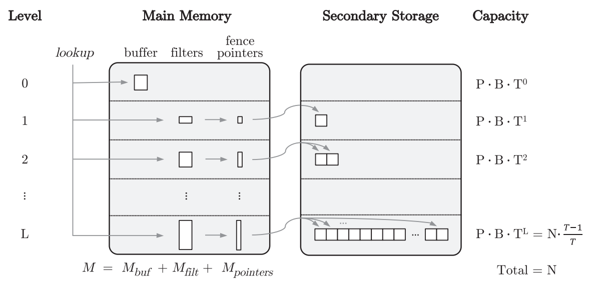
Interactive Design. To provide further understanding of the impact of Monkey we provide an interactive LSM-tree based key-value store design tool at http://daslab.seas.harvard.edu/monkey/.

2 BACKGROUND

This section provides the necessary background on LSM-trees. An LSM-tree stores key-value pairs. A key identifies an object of the application and allows retrieving its value. For ease of presentation, all figures in this article show only keys but can be thought of as key-value pairs.

Buffering Updates. Figure 2 illustrates an LSM-tree and a list of terms used throughout the article. An LSM-tree consists conceptually of L levels. Level 0 refers to an in-memory buffer, and the rest of the levels refer to data in secondary storage. An LSM-tree optimizes for inserts, updates, and deletes (henceforth just referred to as updates) by immediately storing them in the buffer at Level 0 without having to access secondary storage (there is a flag attached to each entry to indicate if it is a delete). If an update refers to a key that already exists in the buffer, then the original entry is replaced in-place and only the latest one survives.

When the buffer's capacity is reached, its entries are sorted by key into an array and flushed to Level 1 in secondary storage. We refer to these sorted arrays in storage as *runs*. We denote the number of bits of main memory allocated to the buffer as M_{buf}, and we define it as $M_{buf} = P \cdot B \cdot E$, where B is the number of entries that fit into a disk page, P is the amount of main memory in



Term	Description	Unit
N	Total number of entries	entries
L	Number of levels	levels
B	Number of entries that fit into a disk page	entries
E	Size of an entry	bits
P	Size of the buffer in disk pages	pages
T	Size ratio between adjacent levels	
T_{lim}	Size ratio value at which point L converges to 1	
M	Total amount of main memory in system	bits
M_{buf}	Main memory allocated to the buffer	bits
M_{filt}	Main memory allocated to the Bloom filters	bits
$M_{pointers}$	Main memory allocated to the fence pointers	bits

Fig. 2. Overview of an LSM-tree and list of terms used throughout the article.

terms of disk pages allocated to the buffer, and E is the average size of data entries. For example, in LevelDB the default buffer size is 2MB.

The runs at Level 1 and beyond are immutable. Each Level i has a capacity threshold of $B \cdot P \cdot T^i$ entries, where T is a design parameter denoting the size ratio between the capacities of adjacent levels. Thus, levels have exponentially increasing capacities by a factor of T. The total number of levels L is given by Equation (1), where N is the total number of physical entries across all levels including updates and deletions,

$$L = \left\lceil \log_T \left(\frac{N \cdot E}{M_{buf}} \cdot \frac{T-1}{T} \right) \right\rceil . \tag{1}$$

The size ratio T has a limiting value of T_{lim}, where $T_{lim} = \frac{N \cdot E}{M_{buf}}$. The value of T can be set anywhere between 2 and T_{lim}. As T approaches T_{lim}, the number of levels L approaches 1.

Merge Operations. To bound the number of runs that a lookup has to probe, an LSM-tree organizes runs among the different levels based on their sizes, and it merges runs of similar sizes (i.e., at the same level). There are two possible merge policies: leveling [57] and tiering [7, 47]. The former optimizes more for lookups and the latter more for updates [50]. With leveling, there is at most one run per Level i, and any run that is moved from Level $i-1$ to Level i is immediately sort-merged with the run at Level i, if one exists. With tiering, up to T runs can accumulate at Level i, at which point these runs are sort-merged. The essential difference is that a leveled LSM-tree merges runs more greedily and therefore gives a tighter bound on the overall number of runs that a lookup has to probe, but this comes at the expense of a higher amortized update cost. Figure 3 compares the behavior of merge operations for tiering and leveling when the size ratio T is set to 3.

If multiple runs that are being sort-merged contain entries with the same key, then only the entry from the most recently-created (youngest) run is kept, because it is the most up-to-date. Thus, the resulting run may be smaller than the cumulative sizes of the original runs. When a merge operation is finished, the resulting run moves to Level $i+1$ if Level i is at capacity.

Lookups. A point lookup starts from the buffer and traverses the levels from smallest to largest (and the runs within those levels from youngest to oldest in the case of tiering). When it finds the first matching entry it terminates. There is no need to look further, because entries with the same key at older runs are superseded. A zero-result lookup (i.e., where the target key does not exist) incurs a potentially high I/O cost, because it probes all runs within all levels. In contrast, a range lookup requires sort-merging all runs with an overlapping key range to identify and ignore superseded entries.

Probing a Run. In the original LSM-tree design from 1996 [57], each run is structured as a compact B-tree. Over the past two decades, however, main memory has become cheaper, so modern designs

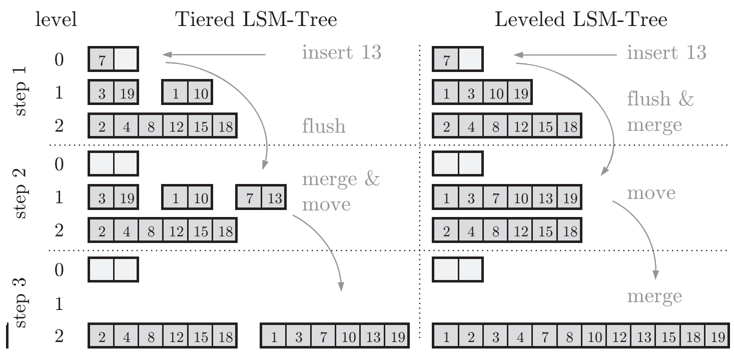


Fig. 3. Before and after a recursive merge with tiered and leveled LSM-trees where the size ratio T is set to 3, and $B \cdot P$, the number of entries that fit into the buffer, is set to 2.

store fence pointers in main memory with min/max information for every disk page of every run[1] [40, 44]. In Appendix I, we show that the size of the fence pointers is modest: typically 3 or more orders of magnitude smaller than the raw data size. The fence pointers enable a lookup to find the relevant key range in a run with just 1 I/O. Given that the LSM-tree is on disk while the fence pointers are in main memory, the overhead of searching fence pointers is not on the critical path of performance (I/O is). Thus, a lookup initially searches the fence pointers before accessing a run in storage. If it is a point lookup, it then reads the appropriate disk block with one I/O, or if it is a range lookup it begins a scan from this block.

Bloom Filters. To speed up point queries, every run has a corresponding Bloom filter [19] in main memory. A point lookup probes a run's filter before accessing the run in secondary storage. If the filter returns negative, then the target key does not exist in the run, and so the lookup skips accessing the run and saves one I/O. If a filter returns positive, then the target key may exist in the run, so the lookup probes the run at a cost of one I/O. If the run actually contains the key, then the lookup terminates. Otherwise, we have a "false positive," and the lookup continues to probe the next run. False positives increase the I/O cost of lookups. The false-positive rate (FPR) depends on (1) the number of *entries* in a run and (2) the number of *bits* in main memory allocated to the run's filter. This relationship is captured by the following equation [66][2]:

$$FPR = e^{-\frac{bits}{entries} \cdot \ln(2)^2}. \tag{2}$$

To the best of our knowledge, all LSM-tree based key-value stores use the same number of bits-per-entry across all Bloom filters. This means that a lookup probes on average $O(e^{-M_{filt}/N})$ of the runs, where M_{filt} is the overall amount of main memory allocated to the filters. As M_{filt} approaches 0 or infinity, the term $O(e^{-M_{filt}/N})$ approaches 1 or 0, respectively. All implementations that we know of use 10 bits per entry for their Bloom filters by default[3] [6–8, 40, 44, 63]. The corresponding false-positive rate is ≈1%. With this tuning, an average entry size of 128 bytes (typical in practice [5]) entails the Bloom filters being ≈2 orders of magnitude smaller than the raw data size.

[1]The fence pointers may be implemented as a sorted array or as a tree.
[2]Equation (2) assumes a Bloom filter that uses the optimal number of hash functions ($\frac{bits}{entries} \ln(2)$) that minimizes the false-positive rate for a given number of bits and entries.
[3]The false-positive rate for a Bloom filter drops at an exponential rate with respect to the number of bits allocated to it. With 10, 20, 30 and 40 bits per entry, the FPR for a Bloom filter in the order of 10^{-2}, 10^{-4}, 10^{-7}, and 10^{-9}, respectively. Therefore, we incur diminishing returns as we allocate more space to the filters: each additional bit wins less performance as the false-positive rate becomes increasingly small.

Cost Analysis. We now analyze the worst-case I/O cost complexities for updates and lookups with tiered and a leveled LSM-trees. Our focus on worse-case analysis is motivated by the fact that many applications today, especially in cloud settings, require stable performance [37]. For updates, we measure the *amortized worst-case I/O cost*, which accounts for the merge operations that an entry participates in after it is updated. For lookups, measure the *zero-result average worst-case I/O cost*, which is the expected number of I/Os performed by a lookup to a key that does not exist in the LSM-tree. We focus on zero-result lookups, because (1) they are common in practice [22, 63] (e.g., insert-if-not-exist queries [63]), (2) they incur the maximum pure I/O overhead (i.e., read I/Os that do not find relevant entries to a lookup), and (3) their analysis is easy to extend to non-zero-result point lookups, as we do later in the article. For the rest of the article, our use of the terms update cost and lookup cost follow these definitions unless otherwise mentioned. We later also model the cost of range lookups for completeness.

For a tiered LSM-tree, the worst-case lookup cost is given by $O(L \cdot T \cdot e^{-M_{filt}/N})$ I/Os, because there are $O(L)$ levels, $O(T)$ runs per level, the cost of probing each run is one I/O due to the fence pointers, and we probe on average only $O(e^{-M_{filt}/N})$ of the runs. The worst-case update cost is $O(L/B)$ I/Os, because each entry participates in $O(L)$ merge operations, i.e., one per level, and the I/O cost of copying one entry during a merge operation is $O(1/B)$, since each write I/O copies $O(B)$ entries into the new run.

For a leveled LSM-tree, the worst-case lookup cost is given by $O(L \cdot e^{-M_{filt}/N})$ I/Os, because there are $O(L)$ levels, there is one run per level, the cost of probing a run is one I/O due to the fence pointers, and we probe $O(e^{-M_{filt}/N})$ of the runs on average. The worst-case update cost is $O(T \cdot L/B)$, because each update is copied $O(T)$ times per level and through $O(L)$ levels overall.

Cost Analysis Extensions. Throughout the article, our analysis assumes that a key is stored adjacently to its value within a run [40, 44]. Our work also applies to applications where there are no values (i.e., the LSM-tree is used to answer set-membership queries on keys), where the values are pointers to data objects stored outside of LSM-tree [56], or where LSM-tree is used as a building block for solving a more complex algorithmic problem (e.g., graph analytics [23], flash translation layer design [32]). We restrict the scope of analysis to the basic operations on LSM-tree so that it can easily be applied to each of these other cases.

Other Log-Structured Designs. Our analysis and optimizations apply out-of-the-box to any LSM-tree based key-value store with (1) sorted runs in storage, (2) fence pointers and a Bloom filter in main memory for each run, (3) a fixed size ratio between capacities at adjacent levels, and (4) either tiering or leveling as a merge policy within levels. We survey other log-structured designs in Section 6.2.

3 LSM-TREE DESIGN SPACE

In this section, we describe critical tradeoffs and design contentions in the LSM-tree design space. Our contributions in this article are enabled by a detailed mapping of this space. We introduce a visualization for the design space, and we use this visualization to identify contentions among design knobs. In the next section, we resolve these contentions with the design of Monkey.

A summary of the design space of LSM-trees and the impact on lookup and update costs is given in Table 1 and illustrated in Figure 4. These costs depend on multiple design parameters: (1) the merge policy (tiering vs. leveling), (2) the size ratio T between levels, (3) the allocation of main memory among the buffer M_{buf} and the Bloom filters, and (4) the allocation of M_{filt} among each of the different Bloom filters. The main observation is that:

The design space of LSM trees spans everything between a write-optimized log to a read-optimized sorted array.

Table 1. Analysis of Existing LSM-tree Designs

Technique	Point Lookup Cost	Update Cost
(1) Log	$O\left(\frac{N\cdot E}{M_{buf}} \cdot e^{-M_{filt}/N}\right)$	$O\left(\frac{1}{B}\right)$
(2) **Tiering**	$O\left(T \cdot \log_T\left(\frac{N\cdot E}{M_{buf}}\right) \cdot e^{-M_{filt}/N}\right)$	$O\left(\frac{1}{B} \cdot \log_T\left(\frac{N\cdot E}{M_{buf}}\right)\right)$
(3) **Leveling**	$O\left(\log_T\left(\frac{N\cdot E}{M_{buf}}\right) \cdot e^{-M_{filt}/N}\right)$	$O\left(\frac{T}{B} \cdot \log_T\left(\frac{N\cdot E}{M_{buf}}\right)\right)$
(4) Sorted Array	$O\left(e^{-M_{filt}/N}\right)$	$O\left(\frac{1}{B} \cdot \frac{N\cdot E}{M_{buf}}\right)$

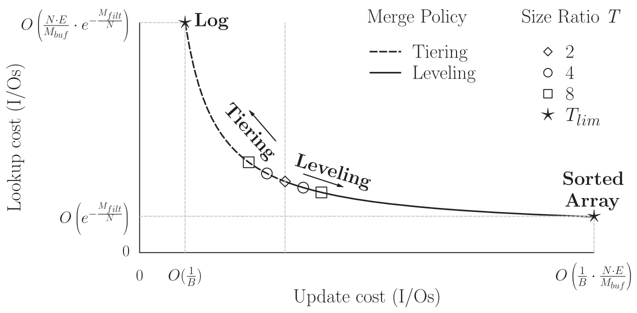


Fig. 4. LSM-tree design space: from a log to a sorted array.

The question is how can we accurately navigate this design space and what is the exact impact of each design decision? To approach an answer to these questions we move on to discuss Figure 4 and the contention among the various design decisions in more detail.

Merge Policy and Size Ratio. The first insight about the design space is that when the size ratio T is set to 2, the complexities of lookup and update costs for tiering and leveling become identical. As we increase T with tiering/leveling, respectively, update cost decreases/increases whereas lookup cost increases/decreases. To generate Figure 4, we plugged all combinations of the merge policy and the size ratio into the complexity equations in Rows *2* and *3* of the table, and we plotted point lookup cost against update cost for corresponding values of the merge policy and size ratio. We did not plot the curve to scale, and in reality the markers are much closer to the graph's origin. However, the shape of the curve and its limits are accurate. The dotted and solid lines correspond to partitions of the design space that are accessible using tiering and leveling, respectively. These lines meet when the size ratio T is set to 2, and they grow farther apart in opposing directions as T increases. This shows that tiering and leveling are complementary methods for navigating the same tradeoff continuum.

As the size ratio T approaches its limit value T_{lim}, the number of levels L approaches 1. When L is 1, a tiered LSM-tree degenerates to log (Row 1 of Table 1) while a leveled LSM-tree degenerates to sorted array (Row 4 of Table 1). We illustrate both of these points as the edges of the curve in Figure 4. To derive the lookup and update costs for a log shown in the figure, we plugged in T_{lim} for T in the cost equations in Table 1 for tiering, and to get the costs for a sorted array we plugged T_{lim} for T in the corresponding cost equations in Table 1 for leveling. We give further intuition for these costs in Appendix F. A log is an update-friendly data structure while a sorted array is a read-friendly one. In this way, LSM-tree can be tuned anywhere between these two extremes in terms of their performance properties. This is a characteristic of the design space we bring forward and heavily utilize in this article.

Main Memory Allocation. The limits of the curve in Figure 4 are determined by the allocation of main memory among the filters M_{filt} and the buffer M_{buf}. Getting the memory allocation right is of critical importance. Main memory today is composed of DRAM chips, which cost ≈2 orders of magnitude more than disk in terms of price per bit, and this ratio is increasing as an industry trend [54]. Moreover, DRAM consumes ≈4 times more power per bit than disk during runtime [68]. As a result, the main memory occupied by the Bloom filters and buffer accounts for a significant portion of a system's (infrastructure and running) cost and should be carefully utilized.

Design Space Contentions. Overall, we identify three critical performance contentions in the LSM-tree design space.

Contention 1 arises in how we allocate a given amount of main memory M_{filt} among the different Bloom filters. By reallocating main memory from one filter to another, we reduce and increase the false-positive rates of the former and latter filters, respectively. How do we optimally allocate M_{filt} among the different Bloom filters to minimize lookup cost?

Contention 2 arises in how to allocate the available main memory between the buffer and the filters. As indicated by the cost complexities in Table 1, allocating a more main memory to the buffer on one hand decreases both lookup cost and update cost, but on the other hand it decreases the Bloom filters' accuracy thereby increasing lookup cost. How do we strike the best balance?

Contention 3 arises in how to tune the size ratio and merge policy. This is complicated, because workloads consist of different proportions of (1) updates, (2) zero-result lookups, (3) non-zero-result lookups, and (4) range lookups of different selectivities. Decreasing the size ratio under tiering and increasing the size ratio under leveling improves lookup cost and degrades update cost, but the impact and rate of change on the costs of different operation types is different. How do we find the best size ratio and merge policy for a particular application workload?

The State of the Art. All LSM-tree based key-value stores that we know of apply static and suboptimal decisions regarding the above contentions. The Bloom filters are all tuned the same, the Buffer size relative to the Bloom filters size is static, and the size ratio and merge policy are also static [6–8, 40, 44, 63]. We refer to these key-value stores collectively as the state of the art. Although they differ from each other in various respects (e.g. centralized vs. decentralized architectures, different consistency guarantees, different data models, etc.), these design aspects are orthogonal to this work. In the next section, we introduce Monkey, which improves upon the state of the art by resolving these contentions and being able to quickly, accurately, and optimally navigate the LSM-tree design space.

4 MONKEY

In this section, we present Monkey in detail. Monkey is an LSM-tree based key-value store that achieves a better performance tradeoff curve than state-of-the-art design, and it navigates this tradeoff curve to find the best possible balance between the costs of lookups and updates for any given main memory budget, application workload, and storage medium. It maximizes throughput for uniformly random workloads, and it maximizes the lower-bound on throughput for all other workloads. Monkey achieves this by (1) resolving the contentions in the LSM-tree design space and (2) using models to optimally trade among lookup cost, update cost, and main memory footprint. Below we give a high level summary of the main design elements and performance impact of Monkey before we move forward to describe each one of them in depth.

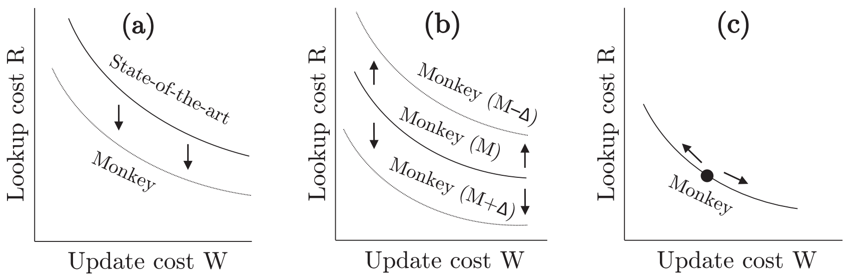


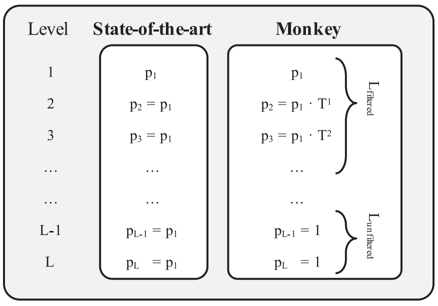
Fig. 5. The design of Monkey impacts performance in three ways: (a), it achieves a better tradeoff curve than the state of the art by optimally allocating the Bloom filters, (b) it predicts how changes in environmental parameters and main memory utilization reposition its tradeoff curve, and (c) it finds the point on its tradeoff curve that maximizes worst-case throughput for a given application.

Design Knobs. Monkey transforms the design elements that impact worst-case behavior into design knobs, and it can alter its behavior by adjusting them. Those knobs comprise: (1) the size ratio among levels T, (2) the merge policy (leveling vs. tiering), (3) the false-positive rates $p_1 \ldots p_L$ assigned to Bloom filters across different levels, and (4) the allocation of main memory M between the buffer M_{buf} and the filters M_{filt}. Monkey can co-tune these knobs to optimize throughput, or to favor one performance metric over another if needed by the application (e.g., a bound on average lookup or update latency). Figure 5 shows the performance effects achieved by Monkey.

Minimizing Lookup Cost. The first core design element in Monkey is optimal allocation of main memory across Bloom filters to minimize lookup cost. The key insight is that lookup cost is proportional to the sum of the false-positive rates (FPR) of all the Bloom filters. In Section 4.1, we show how to tune filters across levels differently to minimize the cost of zero-result point lookups, and in Section 4.4 we generalize this result to optimize for any balance between zero and non-zero-result point lookups in the workload. Figure 5(a) shows visually the impact this change brings. It achieves faster reads than the state of the art for any main memory budget, and so it shifts the entire tradeoff curve vertically down.

Performance Prediction. The second design element in Monkey is the ability to predict how changing a design decision or an environmental parameter would impact worst-case performance. We achieve this in Section 4.2 by deriving closed-form models for the worst-case I/O costs of lookups and updates in terms of the LSM-tree design space knobs. For instance, the models predict how changing the overall amount of main memory or its allocation would reposition the performance tradeoff curve, as shown in Figure 5(b). We derive an analogous model for the state of the art as a baseline and show that Monkey dominates it.

Self-design. The third design element in Monkey is the ability to holistically self-design to maximize the worst-case throughput. We achieve this in two steps. First, in Section 4.3 we use asymptotic analysis to map the design space and thereby devise a rule for how to allocate main memory between the buffer and the filters. Second, in Section 4.5 we model worst-case throughput with respect to (1) our models for lookup cost and update cost, (2) the proportion of lookups and updates in the workload, and (3) the costs of reads and writes to persistent storage. We introduce an algorithm that quickly searches the performance tradeoff curve for the balance between lookup and update cost that maximizes the worst-case throughput. This property is visually shown in Figure 5(c).



Term	Description	Unit
p_i	False positive rate (FPR) for filters at level i	
R	Worst-case zero-result point lookup cost	I/O
$R_{filtered}$	Worst-case zero-result point lookup cost to levels with filters	I/O
$R_{unfiltered}$	Worst-case zero-result point lookup cost to levels with no filters	I/O
V	Worst-case non-zero-result point lookup cost	I/O
W	Worst-case update cost	I/O
Q	Worst-case range lookup cost	I/O
$M_{threshold}$	Value of $M_{filters}$ below which p_L (FPR at level L) converges to 1	bits
ϕ	Cost ratio between a write and a read I/O to persistent storage	
s	Proportion of entries in a range lookup	
$L_{filtered}$	Number of levels with Bloom filters	
$L_{unfiltered}$	Number of levels without Bloom filters	
$\widehat{M}$	Amount of main memory to divide among filters and buffer	bits

Fig. 6. Overview of how Monkey allocates false-positive rates $p_1, p_2 \ldots p_L$ to Bloom filters in proportion to the number of key-value pairs in a given level, and a further list of terms used to describe Monkey.

4.1 Minimizing Lookup Cost

We now continue to discuss how Monkey minimizes the worst-case lookup cost. We start by focusing on zero-result point lookups, and we extend the method and analysis to non-zero-result point lookups in Section 4.4. Figure 6 gives a list of terms that we use to describe Monkey. We denote R as the worst-case expected I/O cost of a zero-result point lookup. We first model R and M_{filt} in terms of the false-positive rates (FPRs) of the Bloom filters across all levels of LSM-tree. We then show how to minimize R by adjusting the FPRs across different levels while keeping M_{filt} and everything else fixed.[4]

Modeling Average Worst-Case Lookup Cost. The average number (i.e., expected value) of runs probed by a zero-result lookup is the sum of the FPRs of all Bloom filters. Equation (3) expresses this sum in terms of the FPRs $p_1 \ldots p_L$ assigned to filters at different levels. With leveling every level has at most one run and so R is simply equal to the sum of FPRs across all levels. With tiering there are at most $T - 1$ runs at every level (when the Tth run arrives from the previous level it triggers a merge operation and a push to the next level). The FPR for all runs at the same level in tiering is the same because they have the same size,

$$R = \begin{cases} \sum_{i=1}^{L} p_i, & \text{with leveling} \\ (T-1) \cdot \sum_{i=1}^{L} p_i, & \text{with tiering} \end{cases} \qquad (3)$$
$$\text{where } 0 < p_i \leq 1.$$

Modeling Main Memory Footprint. We now model the total main memory footprint for the bloom filters in terms of the FPRs $p_1, p_2 \ldots p_L$ of the different levels. To do so, we first rearrange Equation (2) in terms of the number of bits in a filter: $bits = -entries \cdot \frac{\ln(FPR)}{\ln(2)^2}$. This equation captures the cumulative size of any number of Bloom filters that have the same FPRs, and so we can apply it out-of-the-box for both leveling (one Bloom filter per level) and for tiering ($T - 1$ Bloom filters per level). We do so by identifying and plugging in the number of entries and the FPR for each level. As shown in Figure 2, the last level of an LSM-tree has at most $N \cdot \frac{T-1}{T}$ entries, and in general Level i has at most $\frac{N}{T^{L-i}} \cdot \frac{T-1}{T}$ entries, because smaller levels have exponentially smaller capacities by a factor of T. Thus, the amount of main memory occupied by filters at Level i is at most $-\frac{N}{T^{L-i}} \cdot \frac{T-1}{T} \cdot \frac{\ln(p_i)}{\ln(2)^2}$ bits. The overall amount of main memory allocated cumulatively to all

[4]We assume a fixed entry size E throughout this section; in Appendix B we give a iterative optimization algorithm that quickly finds the optimal FPR assignment even when the entry size is variable or changes over time.

Bloom filters is the sum of this expression over all levels, as captured by Equation (4),

$$M_{filt} = -\frac{N}{\ln(2)^2} \cdot \frac{T-1}{T} \cdot \sum_{i=1}^{L} \frac{\ln(p_i)}{T^{L-i}}. \tag{4}$$

Minimizing Lookup Cost with Monkey. Using Equations (3) and (4), we can calculate the average worst-case lookup cost R and main memory footprint M_{filt} for any assignment of FPRs across the different levels. To minimize R with respect to M_{filt}, we first tackle the converse yet equivalent problem for ease of exposition: finding the optimal assignment of FPRs $p_1 \ldots p_L$ across the different levels that minimizes M_{filt} for any user-specified value of R. This amounts to a multivariate constrained optimization problem. In Appendix A, we solve it by applying the method of Lagrange Multipliers on Equations (3) and (4). The result appears in Equations (5) and (6) for leveled and tiered designs, respectively.[5] The lookup cost R is a parameter to Equations (5) and (6). Any value for R within the domain defined in the equations can be plugged in, and the equations return the optimal false-positive rate for each of the levels of LSM-tree to achieve a lookup cost of R using the least possible amount of main memory. Note that the equations are undefined for R being set to 0, the reason being that it is impossible for a Bloom filter to have a false-positive rate of 0.

Leveling

$$p_i = \begin{cases} 1, \text{if } i > L_{filtered} \\ \frac{(R-L_{unfiltered})\cdot(T-1)}{T^{L_{filtered}+1-i}}, \text{else} \end{cases}$$
$$\text{for } 0 < R \leq L$$
$$\text{and } 1 \leq i \leq L$$
$$\text{and } L_{filtered} = L - \max(0, \lfloor R-1 \rfloor), \tag{5}$$

Tiering

$$p_i = \begin{cases} 1, \text{if } i > L_{filtered} \\ \frac{R-L_{unfiltered}\cdot(T-1)}{T^{L_{filtered}+1-i}}, \text{else} \end{cases}$$
$$\text{for } 0 < R \leq L \cdot (T-1)$$
$$\text{and } 1 \leq i \leq L$$
$$\text{and } L_{filtered} = L - \max(0, \lfloor \tfrac{R-1}{T-1} \rfloor). \tag{6}$$

Figure 6 illustrates how Monkey optimally assigns FPRs to Bloom filters across different levels using Equations (5) and (6). In general, the optimal FPR at Level i is T times higher than the optimal FPR at Level $i-1$. In other words, the optimal FPR for level i is proportional to the number of elements at level i. The intuition is that the I/O cost of probing any run is the same regardless of its size (due to the fence pointers we only fetch the qualifying disk page), yet the amount of main memory needed for achieving a low FPR at deeper levels is significantly higher, since they have exponentially more entries. It is therefore better to set relatively more bits per entry (i.e., a lower FPR) to the filters at smaller levels. In contrast, state-of-the-art LSM-tree based key-value stores assign the same FPR to Bloom filters across all the different levels.

The higher we set the lookup cost R, the less main memory for the Bloom filters we need. As shown in Figure 6, the mechanism through which this works in Monkey is that for higher values of R more of the Bloom filters at the deepest levels cease to exist as their optimal FPRs converge to 1. We denote the number of levels with and without filters as $L_{filtered}$ and $L_{unfiltered}$, respectively (note that $L = L_{filtered} + L_{unfiltered}$). Equations (5) and (6) are adapted to find the optimal division between $L_{filtered}$ and $L_{unfiltered}$ and to prescribe FPRs to the smaller $L_{filtered}$ levels based on a smaller version of the problem with $L_{filtered}$ levels.

In summary, Monkey minimizes the main memory footprint for the Bloom filters for a given lookup cost R by (1) finding the optimal number of levels $L_{filtered}$ to which Bloom filters should be allocated and (2) setting the FPR for each of these levels to be proportional to its capacity. Through these steps it achieves the performance effect shown in Figure 5(a).

[5]In the next subsection, we show how to express Equations (5) and (6) in terms of M_{filt} rather than R.

4.2 Predicting Lookup and Update Costs

We now move forward to map the design space of LSM-trees in a way that allows to accurately predict the impact of the various design decisions. To do so, we model lookup and update cost in closed-form expressions with respect to all the design knobs in the Monkey design space. We also demonstrate analytically that Monkey dominates the state-of-the-art designs.

Modeling Zero-Result Lookup Cost (R). To derive a closed-form expression for the zero-result lookup cost R in Monkey, we plug the optimal false-positive rates in Equations (5) and (6) into Equation (4), simplify, and rearrange. The complete derivation is in Appendix A.1, and the result is Equation (7). This equation assumes a fixed entry size (we lift this restriction in Appendix B).

To demystify Equation (7), note that the additive terms $R_{filtered}$ and $R_{unfiltered}$ correspond to the average number of runs probed in the levels with and without filters, respectively. Also recall that when the size ratio T is set to 2, tiering and leveling behave identically, and so the two versions of the equation for tiering and leveling produce the same result. For most practical applications using mainstream key-value stores from industry, $R_{unfiltered}$ is equal to 0 and so the equations simplify to just the $R_{filtered}$ component, as we show in Section 4.3.

$$
\begin{aligned}
R &= R_{filtered} + R_{unfiltered}\\
R_{filtered} &= \begin{cases} \frac{T^{\frac{T}{T-1}}}{T-1} \cdot e^{-\frac{M_{filt}}{N} \cdot \ln(2)^2 \cdot T^{L_{unfiltered}}} & \text{with leveling}\\ T^{\frac{T}{T-1}} \cdot e^{-\frac{M_{filt}}{N} \cdot \ln(2)^2 \cdot T^{L_{unfiltered}}} & \text{with tiering} \end{cases}\\
R_{unfiltered} &= \begin{cases} L_{unfiltered}, & \text{with leveling}\\ L_{unfiltered} \cdot (T-1), & \text{with tiering} \end{cases}.
\end{aligned} \tag{7}
$$

Next, we derive Equation (8), which gives the number of deeper levels for which there are no filters. To do so, we first derive the threshold value $M_{threshold}$ of main memory at which the FPR of filters at the last level (i.e., Level L) converge to 1 (see bottom of Equation (8)). The complete derivation is in Appendix A.1. The optimal value of $L_{unfiltered}$ given by Equation (8) can be plugged into Equation (7) to compute R,

$$
\begin{aligned}
L_{unfiltered} &= \begin{cases} 0, & M_{threshold} \leq M_{filt}\\ \left\lceil \log_T \left(\frac{M_{threshold}}{M_{filt}} \right) \right\rceil, & \frac{M_{threshold}}{T^L} \leq M_{filt} \leq M_{threshold}\\ L, & 0 \leq M_{filt} \leq \frac{M_{threshold}}{T^L} \end{cases}\\
M_{threshold} &= \frac{N}{\ln(2)^2} \cdot \frac{\ln(T)}{(T-1)}.
\end{aligned} \tag{8}
$$

Modeling Worst-Case Non-Zero-Result Lookup Cost (V). Using Equation (7) for the average worst-case zero-result lookup cost R, we can now also model the average worst-case cost V of a non-zero-result lookup, which finds the target key in the oldest run. To model this cost, we subtract p_L, the FPR of the oldest run's filter, and instead add 1 to account for reading one page of this run.

$$V = R - p_L + 1. \tag{9}$$

Modeling Worst-Case Update Cost (W). To model the worst-case update cost, we assume a worst-case update pattern where an entry is updated at most once within a period of N application updates. We show that this is indeed the worst-case workload for updates in Appendix G as it means that no entry gets eliminated before getting merged into the largest level and so all entries participate in the highest possible number of merge operations. In Appendix H, we use arithmetic series to model the amortized worst-case number of merge operations that an entry participates in per level as $\approx \frac{T-1}{T}$ and $\approx \frac{T-1}{2}$ with tiering and leveling, respectively . We multiply these costs by L,

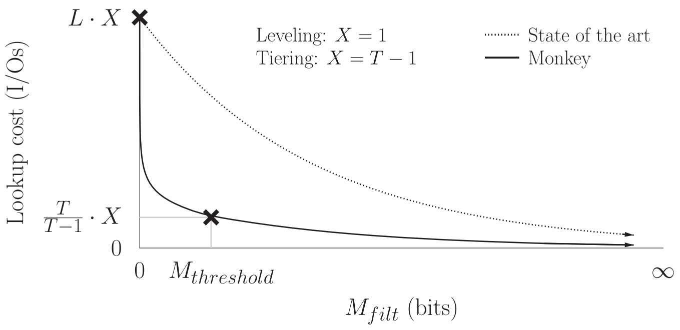


Fig. 7. The lookup cost model for Monkey dominates the state of the art with any main memory budget.

since each entry gets merged across L levels, and we divide by B to get the measurement in terms of block I/Os. Finally, we account for reading the original runs to merge them, and also that write I/Os to secondary storage on some storage devices (e.g., flash) are more expensive than reads, by multiplying by $(1 + \phi)$, where ϕ is the cost ratio between writes and reads. The overall I/O cost is captured by Equation (10). When T is set to 2, tiering and leveling behave identically, so the two parts of the equation produce the same result.

$$W = \begin{cases} \frac{L}{B} \cdot \frac{(T-1)}{2} \cdot (1 + \phi), & \text{with leveling} \\ \frac{L}{B} \cdot \frac{(T-1)}{T} \cdot (1 + \phi), & \text{with tiering.} \end{cases} \tag{10}$$

Modeling Worst-Case Range Lookup Cost (Q). A range lookup involves doing L or $L \cdot (T - 1)$ disk seeks (one per run) for leveling and tiering, respectively. Each seek is followed by a sequential scan. The cumulative number of pages scanned over all runs is $s \cdot \frac{N}{B}$, where s is the average proportion of all entries included in range lookups. Hence, the overall range lookup cost Q in terms of pages reads is as follows.

$$Q = \begin{cases} s \cdot \frac{N}{B} + L, & \text{with leveling} \\ s \cdot \frac{N}{B} + L \cdot (T - 1), & \text{with tiering.} \end{cases} \tag{11}$$

Modeling the State of the Art. We now derive an analogous model for existing state-of-the-art designs. The models for V, W, and Q are the same as for Monkey, namely Equations (9), (10), and (11), because Monkey's core design does not alter these operations. To model the worst-case expected point lookup I/O cost, we set all false-positive rates $p_1, p_2 \ldots p_L$ in Equation (3) to be equal to each other. The complete derivation and resulting closed-form Equation (56) are in Appendix D.

Analytical Comparison. We now have two cost models for lookup cost with Monkey (Equations (7) and (8)) and with the state of the art (Equation (56)). In Figure 7, we compare Monkey against the state of the art by plotting the zero-result lookup cost as we vary M_{filt} with these equations. Although we keep the terms in the figure general, the curves are drawn for an LSM-tree with 512GB of data; the number of entries N is 2^{35}, the entry size E is 16 bytes, the size ratio T is 4, the buffer size $B \cdot P \cdot E$ is 2 MB, and we vary M_{filt} from 0 to 35GB.

The lookup cost model of Monkey dominates that of the state of the art for any overall main memory budget allocated to the filters. The reason is that Monkey allocates this memory optimally among the Bloom filters to minimize the average number of read I/Os per lookup.

As M_{filt} in Figure 7 approaches 0, Monkey and the state of the art both degenerate into an LSM-tree with no Bloom filters, and so their curves meet. The term X in the figure is used to adjust the terms for leveling and tiering. The two curves look identical except that the curve for tiering is vertically stretched upwards by a factor of $T-1$, since there are $T-1$ more runs per level. We verify all of these analytical findings experimentally in Section 5.

4.3 Scalability and Tunability

We now continue by exploring and mapping the design space for Monkey using asymptotic analysis. We show that lookup cost in Monkey scales better than in the state of the art with respect to data volume. We also show that Monkey removes the dependence of lookup cost on the buffer size thereby simplifying design.

Complexity Analysis. Table 2 gives the worst-case lookup and update I/O cost complexities for Monkey and for the state of the art. In Appendix J, we derive lookup complexity for Monkey based on Equations (7) and (8). We do the same for the state of the art in Appendix D based on Equation (56). The update cost is the same for Monkey as for the state of the art, and we analyze it based on Equation (10).

The complexity of worst-case lookup cost for Monkey is different depending on whether M_{filt} is greater or lower than $M_{threshold}$. To understand why, recall that R in Equation (7) is expressed as the sum of two additive terms, $R_{filtered}$ and $R_{unfiltered}$. As long as $M_{filt} > M_{threshold}$, there are filters at all levels and so $R_{unfiltered}$ is zero and $R_{filtered}$ is the dominant term. Moreover, by plugging in $M_{threshold}$ for M_{filt} in Equation (7), we observe that the value of $R_{filtered}$ can be at most $O(1)$ for leveling and at most $O(T)$ for tiering. However, as the number of entries N increases relative to M_{filt}, eventually M_{filt} drops below $M_{threshold}$. At this point $R_{unfiltered}$ becomes non-zero and comes to dominate $R_{filtered}$, because its value is at least $O(1)$ with leveling and $O(T)$ with tiering when $M_{filt} = M_{threshold}$, and it increases up to $O(L)$ with leveling and $O(L \cdot T)$ with tiering as N increases. Thus, the complexity of worst-case lookup cost R is $O(R_{filtered})$ when $M_{filt} > M_{threshold}$, and otherwise it is $O(R_{unfiltered})$.

The condition $M_{filt} > M_{threshold}$ can be equivalently stated as having the number of bits per element $\frac{M_{filt}}{N} > \frac{1}{\ln(2)^2} \cdot \frac{\ln(T)}{T-1}$. The value of $\frac{1}{\ln(2)^2} \cdot \frac{\ln(T)}{T-1}$ is at most 1.44 when T is equal to 2. Hence, we can say more concretely that the complexity of worst-case lookup cost R is $O(R_{filtered})$ when the number of bits-per-element is above 1.44, and otherwise it is $O(R_{unfiltered})$. In modern key-value stores, the number of bits-per-element is typically 10, far above 1.44, and so for most practical purposes the complexity of Monkey is $O(R_{filtered})$.

To enable an apples to apples comparison, we also express the complexity of lookup cost for the state of the art separately for when M_{filt} is lower and greater than $M_{threshold}$ (Column b and c, respectively, in Table 2). We observe that when M_{filt} is lower than $M_{threshold}$, the complexity of lookup cost converges to that of an LSM-tree with no filters.

Comparing Monkey to the State of the Art. We first compare Monkey to the state of the art when $M_{filt} \geq M_{threshold}$ (Columns c and e in Table 2). Monkey shaves a factor of $O(L)$ from the complexity of lookup cost for both tiering and leveling (Rows *2* and *3* in Table 2). Note that we express $O(L)$ in Table 2 as $O(\log_T(N \cdot E/M_{buf}))$ as per Equation (1). In other words, lookup cost R in Monkey is asymptotically independent of the number of levels L of the LSM-tree. The intuition is that the FPRs for smaller levels are exponentially decreasing, and so the expected cost of probing filters across the levels converges to a multiplicative constant. Shaving a factor of $O(L)$ from lookup cost has three important benefits.

(1) As long as we scale the Bloom filters' footprint with the number of data entries (i.e., keep the ratio $\frac{M_{filt}}{N}$ fixed as N increases), lookup cost in Monkey stays fixed, whereas in the state of the art it increases at a logarithmic rate. In this way, Monkey dominates the state of the art by an increasingly large margin as the number of entries increases.

Table 2. Asymptotic Analysis Reveals That (1) Lookup Cost in Monkey Scales Better Than the State of the Art with Respect to the Number and Size of Data Entries, (2) Lookup Cost in Monkey Is Independent of the LSM-tree's Buffer Size, and (3) Monkey and the State of the Art Both Degenerate into a Log and Sorted Array with Tiering and Leveling, Respectively, as the Size Ratio T is Pushed to Its Limit.

Merge policy	**Update**	**State of the Art Lookup Cost**		**Monkey Lookup Cost (R)**	
	Cost (W)	$M_{filt} \leq M_{threshold}$	$M_{threshold} \leq M_{filt}$	$\frac{M_{threshold}}{T^L} \leq M_{filt} \leq M_{threshold}$	$M_{threshold} \leq M_{filt}$
	(a)	*(b)*	*(c)*	*(d)*	*(e)*
(1) Tiering ($T = T_{lim}$)	$O\left(\frac{1}{B}\right)$	$O\left(\frac{N \cdot E}{M_{buf}}\right)$	$O\left(\frac{N \cdot E}{M_{buf}} \cdot e^{-M_{filt}/N}\right)$	$O\left(\frac{N \cdot E}{M_{buf}}\right)$	$O\left(\frac{N \cdot E}{M_{buf}} \cdot e^{-M_{filt}/N}\right)$
(2) Tiering ($2 \leq T < T_{lim}$)	$O\left(\frac{1}{B} \cdot \log_T\left(\frac{N \cdot E}{M_{buf}}\right)\right)$	$O\left(T \cdot \log_T\left(\frac{N \cdot E}{M_{buf}}\right)\right)$	$O\left(T \cdot \log_T\left(\frac{N \cdot E}{M_{buf}}\right) \cdot e^{-M_{filt}/N}\right)$	$O\left(T \cdot \log_T\left(\frac{N}{M_{filt}}\right)\right)$	$O\left(T \cdot e^{-M_{filt}/N}\right)$
(3) Leveling ($2 \leq T < T_{lim}$)	$O\left(\frac{T}{B} \cdot \log_T\left(\frac{N \cdot E}{M_{buf}}\right)\right)$	$O\left(\log_T\left(\frac{N \cdot E}{M_{buf}}\right)\right)$	$O\left(\log_T\left(\frac{N \cdot E}{M_{buf}}\right) \cdot e^{-M_{filt}/N}\right)$	$O\left(\log_T\left(\frac{N}{M_{filt}}\right)\right)$	$O\left(e^{-M_{filt}/N}\right)$
(4) Leveling ($T = T_{lim}$)	$O\left(\frac{1}{B} \cdot \frac{N \cdot E}{M_{buf}}\right)$	$O(1)$	$O\left(e^{-M_{filt}/N}\right)$	$O(1)$	$O\left(e^{-M_{filt}/N}\right)$

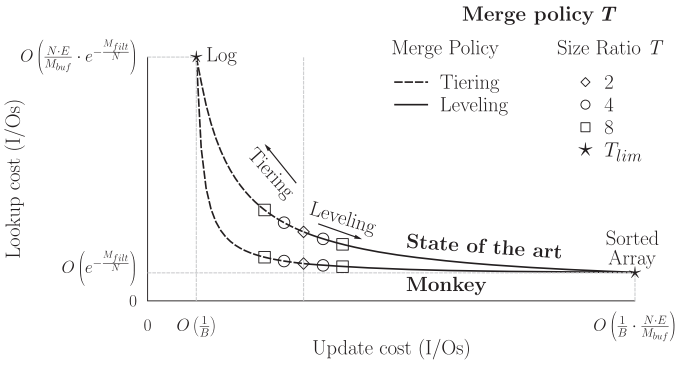


Fig. 8. Monkey dominates the state of the art for any merge policy and size ratio.

(2) Lookup cost is independent of the entry size, and so it does not increase for data with larger entry sizes.
(3) Lookup cost is independent of the buffer size. This simplifies design relative to the state of the art, because we do not need to carefully balance main memory allocation between the buffer and filters to optimize lookup performance.

Next, we compare Monkey to the state of the art when $M_{filt} \leq M_{threshold}$. In the state of the art (Column *b*, Rows *2* and *3* in Table 2), lookup cost decreases at a logarithmic rate as M_{buf} increases, because it absorbs more of the smaller levels. In Monkey (Column *d*, Rows *2* and *3*), lookup cost decreases at a logarithmic rate as M_{filt} increases, since more of the deeper levels have Bloom filters. Monkey improves upon the state of the art by shaving an additive factor of $O(\log_T(E))$ from lookup cost, where E is the entry size. The reason is that Bloom filters are only sensitive to the number of entries rather than their sizes. This means that lookup cost in Monkey does not increase for datasets with larger entries.

Monkey dominates the state of the art for all datasets and designs, because it allocates main memory optimally among the Bloom filters thereby minimizing lookup cost.

Exploring Limiting Behavior. We now focus on the limiting behavior of the lookup cost with respect to the size ratio. This is shown in Rows 1 and 4 of Table 2. We also plot Figure 8 to help this discussion. Figure 8 is an extension of Figure 4 from Section 2 where we mapped the design space of LSM-trees with respect to the impact on lookup and update cost. Figure 4 includes Monkey in addition to the existing state of the art. As T approaches its limit value of T_{lim} for both leveling and tiering, the number of levels L approaches 1. When $T = T_{lim}$, Monkey and the state of the art both degenerate into a log and a sorted array with tiering and leveling, respectively, and so their performance characteristics converge. For all other values of T in-between, Monkey analytically dominates the state of the art by reducing lookup cost, hence achieving a better performance tradeoff curve. We also show these curves experimentally in Section 5.

Analyzing Main Memory Allocation. We now analyze the impact of allocating main memory between the filters and buffer on lookup and update cost. In Figure 9, we plot lookup cost and update cost with Monkey and the state of the art as we vary the relative sizes of the buffer and the filters.[6] We define the amount of main memory excluding the fence pointers that is to be divided between the fence pointers and buffer as $\hat{M}$ bits, and so $\hat{M} = M_{buf} + M_{filt}$. On the x-axis (log scale),

[6]The limits on the y-axis are drawn for $M_{filt} > M_{threshold}$.

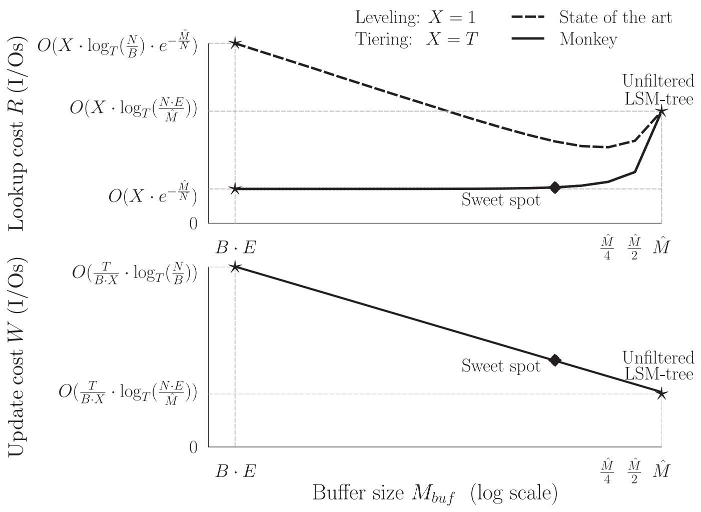


Fig. 9. Monkey simplifies design by eliminating the dependence of lookup cost on the buffer size.

we increase M_{buf} at the expense of M_{filt} from one disk page ($B \cdot E$ bits) to $\hat{M}$ bits, in which case the Bloom filters cease to exist (i.e., $M_{filt} = 0$). While the terms in the figure are general, the curves are drawn using our models in Subsection 4.2 for the configuration outlined at the end of the subsection. The term X is used to adjust the y-axis for leveling and tiering.

The top part of Figure 9 reveals that Bloom filters in the state of the art actually harm lookup performance for a significant portion of the space, because the main memory that they occupy is better-off allocated to the buffer. Monkey removes this performance contention thereby simplifying system design by making lookup cost independent of the buffer size, provided that $M_{filt} > \frac{M_{threshold}}{T^L}$ as per Equation (8). As the buffer continues to grow, however, the Bloom filters shrink and eventually cease to exist, at which point the curves for Monkey and the state-of-the-art converge. The bottom part of Figure 9 illustrates that increasing the buffer size decreases update cost but incurs diminishing returns as the update cost decreases at a logarithmic rate. The overall insight is that it is desirable to set the buffer size as large as possible to reduce update cost while still keeping it below the point where it begins to significantly harm lookup cost. We mark this in Figure 9 as the "sweet spot." In Section 4.5, we give a strategy for how to allocate main memory among the filters and buffer using this insight.

4.4 Optimizing for Non-Zero-Result Point Lookups

Up until now, we only focused on allocating the Bloom filters to minimize the cost for zero-result point lookups. We now generalize Monkey to allocate the Bloom filters so as to optimize for any ratio between non-zero-result and zero-result point lookups in the workload. To optimize for the worst-case non-zero-result point lookups, we focus on lookups for which the target entry is at the largest level as they incur the maximum expected number of false positives before finding the target key.[7] Hence, we treat all non-zero-result point lookups as if they were targeting level L.

[7]Our optimization does not further specialize for non-zero-result point lookups to smaller levels of LSM-tree. The reason is that the false-positive rates at smaller levels of LSM-tree with Monkey are exponentially lower, and so the number of false positives incurred at smaller levels of the tree is not a performance bottleneck (the I/O to the target entry is).

Insight: More Memory for Filters at Smaller Levels. With leveling, the Bloom filter at the largest level does not benefit worst-case point lookups to existing entries as it always returns a true positive when probed. Similarly with tiering, a non-zero-result point lookup probes on average half of the runs at the largest level before finding the target entry and terminating. As a result, half of the Bloom filters at the largest level with tiering on average do not yield a performance benefit.

However, with both tiering and leveling, the Bloom filter(s) at larger levels take up more memory, because they contain more entries. To see this, we denote the amount of memory taken up by filters at Level i as M_i. We analyze M_i for Level i by plugging its capacity and its optimal FPRs from Equation (5) or Equation (6) into Equation (2). We observe that M_i is $O(M_{filt}/T^{L-i})$, or in other words, that filters at larger levels take up exponentially more memory.

Putting these observations together, our design of Monkey so far allocates most of the main memory budget to the largest level to optimize for zero-result point lookups, yet most of this main memory does not significantly benefit non-zero-result point lookups. To optimize for non-zero-result point lookups, it is more beneficial to allocate a higher proportion of the memory budget to filters at smaller levels to reduce the number of false positives that take place before finding the target key. We now continue to introduce $Monkey_{general}$, which allocates a higher proportion of the memory budget to smaller levels as the proportion of non-zero-result point lookups in the workload increases. We also denote the version of Monkey that only optimizes for zero-result point lookups from Section 4 as $Monkey_{zero}$.

Modeling Generalized Point Lookups. We first model the cost of non-zero-result point lookups. With leveling, the expected cost is $1 + R - p_L$, where R is the sum of false-positive rates from Equation (7). The reason is that a lookup issues one I/O to the largest level, and it incurs $R - p_L$ expected I/Os due to false positives at smaller levels. With tiering, the expected cost is $1 + R - \frac{p_L \cdot T}{2}$ I/Os, because a lookup probes on average half of the runs at the largest level before finding the target key. We introduce a constant v in Equation (12) to denote the ratio between non-zero-result lookups to zero-result lookups in the workload.

$$v = \frac{\text{\% non-zero-result point lookups}}{\text{\% zero-result point lookups}}. \tag{12}$$

We then weigh the costs of zero and non-zero-result lookups using v in Equation (13), where Z denotes the expected I/O cost of a lookup.

$$Z = \begin{cases} \frac{v}{v+1} \cdot (1 + R - p_L) + \frac{1}{v+1} \cdot R, & \text{with leveling} \\ \frac{v}{v+1} \cdot \left(1 + R - \frac{p_L \cdot T}{2}\right) + \frac{1}{v+1} \cdot R, & \text{with tiering.} \end{cases} \tag{13}$$

Optimizing FPRs. Next, we optimize the FPRs across the different levels in Equation (13) using Equation (4) as a constraint. As before, we use the method of Lagrange Multipliers. We first derive p_L, the FPR for the largest level. The results are Equations (14) and (15) for leveling and tiering, respectively. As the proportion of non-zero-result point lookups v increases, the optimal value for p_L increases with both leveling and tiering as it becomes less beneficial to allocate memory to the filter at the largest level. When we set v to zero, Equations (14) and (15) converges to Equations (5) and (6), respectively, as we only need to optimize for zero-result point lookups in this case,

Leveling

$$p_L = \begin{cases} (Z \cdot (v+1) - v) \cdot \frac{T-1}{T}, & L_{unfiltered} = 0 \\ 1, & \text{otherwise,} \end{cases} \tag{14}$$

Tiering

$$p_L = \begin{cases} \frac{Z \cdot (v+1) - v}{T \cdot (v+1)} \cdot \left(1 - \frac{v \cdot T}{2 \cdot (v+1) \cdot (T-1)}\right)^{-1}, & L_{unfiltered} = 0 \\ 1, & \text{otherwise.} \end{cases} \tag{15}$$

Next, we derive the optimal FPRs for smaller levels. Again, we use the method of Lagrange Multipliers as in Appendix A. The results are Equations (16) and (17). Each of these equations has three cases. The first case is for when memory is plentiful enough to have Bloom filters across all levels ($L_{unfiltered} = 0$). In this case, the FPRs at smaller levels decrease as v increases as it becomes more beneficial to invest more of the memory budget in smaller levels. The other two cases are for when there is not sufficient memory for having filters at the largest level. In this case, both zero and non-zero-result point lookups issue I/Os to the largest level and need to minimize the sum of FPRs at the smaller levels with filters. As a result, the optimal memory allocation among the filters becomes the same as before in Equations (5) and (6) (though Equations (16) and (17) look slightly different because they are in terms of Z rather than R),

Leveling

$$p_{L-i} = \begin{cases} \left(Z - \frac{v}{v+1}\right) \cdot \frac{T-1}{T} \cdot \frac{1}{T^i}, & L_{unfiltered} = 0 \\ \frac{Z - L_{unfiltered}}{T^{i-L_{unfiltered}}} \cdot \frac{T-1}{T}, & 0 < L_{unfiltered} \leq i, \\ 1, & L_{unfiltered} > i \end{cases} \tag{16}$$

Tiering

$$p_{L-i} = \begin{cases} \frac{1}{T^i} \cdot \frac{Z\cdot(v+1)-v}{T\cdot(v+1)}, & L_{unfiltered} = 0 \\ \frac{Z - \frac{T}{2}\cdot\frac{v}{v+1} - \left(L_{unfiltered} + \frac{1}{v+1} - 1\right)\cdot(T-1)}{T^{1+i-L_{unfiltered}}}, & 0 < L_{unfiltered} \leq i \\ 1, & L_{unfiltered} > i \end{cases} . \tag{17}$$

Next, we find the memory thresholds $M^L_{threshold}$ and $M^{L-1}_{threshold}$ at which the FPRs at the largest and second largest levels converge to one. We derive them by plugging the optimal FPRs from Equations (14), (15), (16), and (17) into Equation (4), rearranging in terms of Z, and equating to Z in Equations (14), (15), (16), and (17) when p_L and p_{L-1} are set to 1. Using these thresholds, we compute the optimal number of levels with no filters using Equation (18).

$$L_{unfiltered} = \begin{cases} 0, & M^L_{threshold} \leq M_{filt} \\ 1, & M^{L-1}_{threshold} \leq M_{filt} \leq M^L_{threshold} \\ 1 + \left\lceil \log_T\left(\frac{M^{L-1}_{threshold}}{M_{filt}}\right)\right\rceil, & \frac{M^{L-1}_{threshold}}{T^L} \leq M_{filt} \leq M^{L-1}_{threshold} \\ L, & 0 \leq M_{filt} \leq \frac{M^{L-1}_{threshold}}{T^{L-1}} \end{cases} \tag{18}$$

$$M^{L-1}_{threshold} = \frac{N}{\ln(2)^2} \cdot \frac{\ln(T)}{T-1} \cdot \frac{1}{T}$$

$$M^L_{threshold} = \begin{cases} \frac{N}{\ln(2)^2} \cdot \left(\frac{\ln(T)}{T-1} - \frac{1}{T} \cdot \ln\left(\frac{1}{v+1}\right)\right) & \text{with leveling} \\ \frac{N}{\ln(2)^2} \cdot \left(\frac{\ln(T)}{T-1} - \frac{1}{T} \cdot \ln\left(1 - \frac{v}{v+1} \cdot \frac{T}{T-1} \cdot \frac{1}{2}\right)\right) & \text{with tiering.} \end{cases}$$

Predicting Performance. Next, we derive a closed-form equation relating the point lookup cost Z to the main main memory budget M_{filt}. We do this by plugging the optimal FPPs from Equations (14), (15), (16), and (17) into Equation (4), simplifying into closed-form, and rearranging in terms of Z. The results are Equations (19) and (20), which enable practitioners to determine precisely how much a given memory budget M_{filt} improves point lookup cost.

Leveling

$$Z = \begin{cases} e^{-\frac{M_{filt}}{N}\cdot\ln(2)^2} \cdot \frac{T^{\frac{T}{T-1}}}{T-1} \cdot \left(\frac{1}{v+1}\right)^{\frac{T-1}{T}} + \frac{v}{v+1}, & M^L_{threshold} \leq M_{filt} \\ e^{-\frac{M}{N}\cdot T^{L_{unfiltered}}\cdot\ln(2)^2} \cdot \frac{T^{\frac{T}{T-1}}}{T-1} + L_{unfiltered}, & \text{otherwise} \end{cases} , \tag{19}$$

Tiering

$$Z = \begin{cases} e^{-\frac{M_{filt}}{N}\cdot\ln(2)^2} \cdot T^{\frac{T}{T-1}} \cdot \left(1 - \frac{v}{v+1} \cdot \frac{T}{T-1} \cdot \frac{1}{2}\right)^{\frac{T-1}{T}} + \frac{v}{v+1}, & M^L_{threshold} \leq M_{filt} \\ e^{-\frac{M}{N}\cdot T^{L_{unfiltered}}\cdot\ln(2)^2} \cdot T^{\frac{T}{T-1}} + \left(L_{unfiltered} - 1 + \frac{1}{v+1}\right) \cdot (T-1) + \frac{T}{2} \cdot \frac{v}{v+1}, & \text{otherwise} \end{cases} . \tag{20}$$

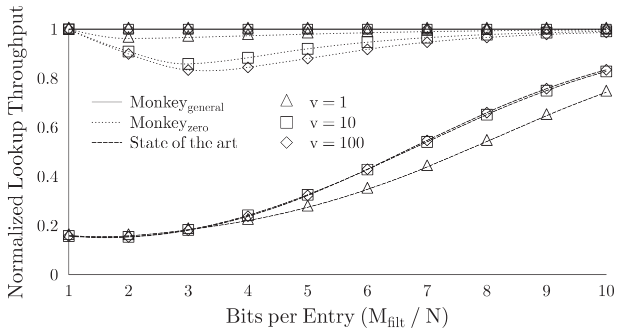


Fig. 10. Optimizing lookups to existing entries further improves lookup throughput by up to 20%.

Analytic Comparison. In Figure 10, we analytically compare $Monkey_{general}$ to $Monkey_{zero}$ (which only optimizes for zero-result point lookups) and to the state of the art in terms of lookup throughput for different amounts of memory and as we increase the proportion of non-zero-result point lookups. For the state of the art and $Monkey_{zero}$, we use Equation (13) for the weighted cost of a mix of zero-result and non-zero-result point lookups in the workload, plugging in different values for R and for p_L. For the state of the art, we plugged in Equation (56) for R and Equation (53) for p_L. For $Monkey_{zero}$, we plugged in Equation (7) for R and Equation (5) for p_L. For $Monkey_{general}$, we used Equation (19). Performance is comparable across curves with the same markers but not for curves with different markers as they are drawn for different workloads (i.e., values of v). The size ratio T is set to 2 for all schemes in this comparison. As we are comparing cost models, we normalize all costs to $Monkey_{general}$ to illustrate the magnitude in performance improvements that our models predict. We evaluate these schemes experimentally in Section 5.

Figure 10 shows that with 1 bit per element, $Monkey_{zero}$ and $Monkey_{general}$ perform the same, because the FPR for the largest level had converged to one and so the assignment of FPRs across all other levels is the same. Otherwise, $Monkey_{general}$ dominates Monkey by up to ≈20%, because it allocates relatively more memory to filters at smaller levels to optimize for non-zero-result point lookups. We also observe that as the proportion of non-zero-result point lookups in the workload increases, $Monkey_{general}$ dominates Monkey by increasing amounts. $Monkey_{zero}$ and $Monkey_{general}$ both outperform the state of the art by up to ≈60% and ≈80%, respectively.

4.5 Holistic Optimal Design of Merge Policy, Size Ratio, and Memory Allocation between Filters and the Buffer

We now show how to navigate the LSM-tree design space to maximize worst-case throughput. Monkey achieves this by controlling four design parameters: the merge policy (tiering vs. leveling), the size ratio, the amount of main memory allocated to the filters, and the amount of main memory allocated to the buffer. We show how to tune these parameters with respect to the dataset (number and size of entries), the workload (proportion of lookups and updates), and the storage medium (cost ratio between reads and writes, and size of disk blocks). We model the worst-case throughput in terms of these parameters, and we devise an algorithm that finds the design that maximizes throughput. In Table 3 we list new terms.

Modeling Throughput. First, we model the average operation cost θ by weighting the point lookup cost Z from Equations (19) and (20) (that model the zero and non-zero-result point lookup cost as discussed in Section 4.4), the range lookup cost Q from Equation (11), and the update cost W from Equation (10) by their proportion in the workload represented by the terms z, q, and w,

Table 3. Table of Terms Used for Self-design

Term	Definition	Units
v	Ratio of non-zero-result point lookups to zero-result point lookups	
Z	Weighted point lookup cost	I/O
z	Percentage of point lookups in workload	%
w	Percentage of updates in workload	%
q	Percentage of range lookups in workload	%
$\hat{M}$	Main memory to divide between the filters and buffer	bits
θ	Average operation cost in terms of lookups	I/O / op
Ω	Time to read a page from persistent storage	sec / I/O
τ	Worst-case throughput	op / sec

respectively. Note that $z + q + w = 1$. The result is Equation (21),

$$\theta = z \cdot Z + q \cdot Q + w \cdot W. \tag{21}$$

To obtain the worst-case throughput τ, we take the inverse of the product of the average operation cost θ multiplied by the secondary storage access latency Ω. The result is Equation (22),

$$\tau = 1/\left(\theta \cdot \Omega\right). \tag{22}$$

For a given workload, dataset, and underlying hardware $(z, v, q, w, s, E, N, \phi, B)$, we express Equation (21) as a function of $\{T, M_{buf}, M_{filt}, pol\}$, where policy *pol* can be either tiering or leveling,

$$cost\left(T, M_{buf}, M_{filt}, pol\right) = z \cdot Z\left(M_{filt}, T, pol\right) + q \cdot Q\left(T, M_{buf}, pol\right) + w \cdot W\left(T, M_{buf}, pol\right). \tag{23}$$

Optimal design is now an optimization problem where we want to find the values of the design parameters that minimize the cost function, which in turn maximizes worst-case throughput.

Size Ratio and Merge Policy. We first introduce an algorithm that optimizes the LSM-tree size ratio and merge policy in a design subspace where main memory allocation is fixed. When main memory allocation is predetermined, the merge policy and size ratio are complementary means of navigating the same tradeoff continuum. We devise a simple brute-force algorithm to check all the possible meaningful values of T, both for tiering and leveling to find the tuning that minimizes the cost for fixed M_{filt} and M_{buf}. Instead of testing for all possible values of T, it suffices to check the value of the cost for all values of T that correspond to an integer number of levels between 1 (maximum value of $T_{lim} = N/B$) and $\lceil \log_2(N/B) \rceil$ (minimum value of T is 2). Hence, we compute the cost function for both tiering and leveling for all T, such that: $T = N^{1/L}$ for $L = 1, 2, \ldots, \lceil \log_2(N/B) \rceil$. The algorithm runs in $O(\log_2(N/B))$ steps.

Main Memory Allocation. The algorithm described above can be used as-is when the main memory allocation is predetermined. State-of-the-art key-value stores frequently use a fixed amount of memory for the buffer, and the remaining of the memory goes to Bloom filters, often targeting using 10 bits per element. Prior research proposed an educated rule-of-thumb where 95% of main memory was allocated for Bloom filters and 5% for the buffer [31]. Here we show how to find the optimal main memory split. We formulate this as an optimization problem where merge policy and frequency is set, and then we optimize solely for main memory allocation. We later combine the two optimization steps by nesting the main memory allocation between the Bloom filters and the buffer with the merge tuning (size ratio and merge policy).

ALGORITHM 1: Holistic Optimal Design

```
**HolisticDesign** $(\hat{M}; z, v, q, w, s, E, N, \phi, B)$
    $min\_cost = \infty$;
    design $\{policy, T, M_{buf}, M_{filt}\}$;
    **for** int $L = 1; L \leq \lceil \log_2(N/B) \rceil; L{+}{+}$ **do**
        $T = N^{1/L}$;
        // Check cost for leveling
        $M_{filt} = optimalMemoryForFilters(\hat{M}, T, \text{“leveling”}; z, v, q, w, s, E, N, \phi, B)$;
        $M_{buf} = \hat{M} - M_{filt}$;
        $current\_cost\_L = cost(T, M_{buf}, M_{filt}, \text{“leveling”})$ // Equation (23) for leveling
        **if** $current\_cost\_L < min\_cost$ **then**
            design.$T = T$; design.$policy$ = “leveling”; design.$M_{buf} = M_{buf}$; design.$M_{filt} = M_{filt}$;
            $min\_cost = current\_cost\_L$;
        **end**
        // Check cost for tiering
        $M_{filt} = optimalMemoryForFilters(\hat{M}, T, \text{“tiering”}; z, v, q, w, s, E, N, \phi, B)$;
        $M_{buf} = \hat{M} - M_{filt}$;
        $current\_cost\_T = cost(T, M_{buf}, M_{filt}, \text{“tiering”})$ // Equation (23) for tiering
        **if** $current\_cost\_T < min\_cost$ **then**
            design.$T = T$; design.$policy$ = “tiering”; design.$M_{buf} = M_{buf}$; design.$M_{filt} = M_{filt}$;
            $min\_cost = current\_cost\_T$;
        **end**
    **end**
    **return** design ;
```

Given the workload, the underlying hardware, the size ratio, and the merge policy we formulate a new optimization problem to decide how to split the available main memory between Bloom filters and the buffer. We consider the cost from Equation (23) as a function of memory for filters M_{filt} where $M_{filt} + M_{buf} = \hat{M}$ and where T and $\hat{M}$ are parameters,

$$cost\left(M_{filt}; T, \hat{M}, pol\right) = z \cdot Z\left(M_{filt}; T, pol\right) + q \cdot Q\left(\hat{M} - M_{filt}; T, pol\right) + w \cdot W\left(\hat{M} - M_{filt}; T, pol\right). \tag{24}$$

Achieving optimal memory allocation is now formalized as finding the number of bits allocated to Bloom filters M_{filt} that minimizes the cost function in Equation (24). The cost functions can be refactored to be easier to manipulate their derivatives with respect to M_{filt} (the full derivation is in Appendix C):

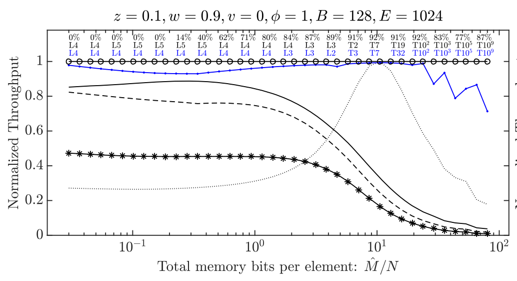
$$cost\left(M_{filt}\right) = \alpha_{pol} \cdot e^{-\beta \cdot M_{filt}} + \gamma_{pol} \cdot ln\left(\frac{\delta}{\hat{M} - M_{filt}}\right) + C, \tag{25}$$

where α and γ have different values for leveling and tiering and β, δ, and C have the same value for both tiering and leveling, as follows:

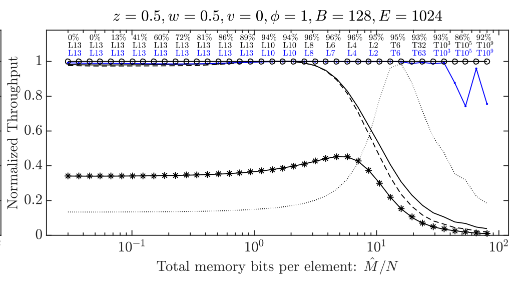
$$\alpha_{leveling} = z \cdot \frac{T^{\frac{T}{T-1}}}{T-1} \cdot \left(\frac{1}{v+1}\right)^{\frac{T-1}{T}}, \quad \alpha_{tiering} = z \cdot T^{\frac{T}{T-1}} \cdot \left(1 - \frac{v}{v+1} \cdot \frac{T}{T-1} \cdot \frac{1}{2}\right)^{\frac{T-1}{T}}, \tag{26}$$

$$\gamma_{leveling} = \frac{q + w \cdot \frac{(T-1)\cdot(1+\phi)}{2 \cdot B}}{\ln(T)}, \quad \gamma_{tiering} = \frac{q \cdot (T-1) + w \cdot \frac{(T-1)\cdot(1+\phi)}{T \cdot B}}{\ln(T)}, \tag{27}$$

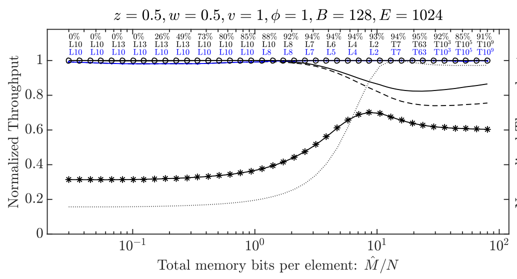
$$\beta = \frac{\ln(2)^2}{N}, \quad \delta = N \cdot E \cdot \frac{T-1}{T}, \quad C = z \cdot \frac{v}{v+1} + q \cdot s \cdot \frac{N}{B}. \tag{28}$$



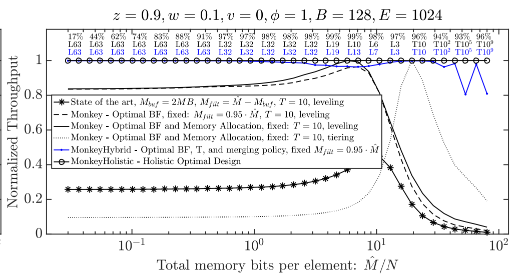
(a) Lookups 10%/Updates 90%



(b) Lookups 50%/Updates 50%, $\upsilon = 0$



(c) Lookups 50%/Updates 50%, $\upsilon = 1$



(d) Lookups 90%/Updates 10%

Fig. 11. Achieving the optimal design comes by combining multiple optimizing steps. Here we see the contribution in terms of normalized performance of optimizing each design dimension of the LSM-tree: Bloom filter allocation, main memory allocation, merge policy, and size ratio. Finally, putting everything together in a multi-parametric optimization, allows us to get the holistically optimal behavior across the whole spectrum of available main memory.

The value of M_{filt} that minimizes Equation (25) cannot be expressed analytically (see Appendix C), hence, we devise an efficient and accurate numerical process that finds for which M_{filt} the cost is minimized using the quickly converging iterative Newton-Raphson method. In Appendix C.1 we discuss more details about this process and its convergence guarantees. This algorithm runs in $O(\log_{10}(\hat{M}/B))$ steps.

Holistic Design. In this section up to now we presented a design process that consists of two steps: first tune for size ratio and merge policy and only then for main memory allocation. This strategy, however, misses opportunities for deeper optimization where the interplay of main memory allocation, size ratio, and merging policy can give us an overall better result. To that end we introduce MonkeyHolistic, shown as Algorithm 1. Every iteration of the algorithm includes the search for the optimal memory allocation so the overall complexity of the holistic tuning process is $O(\log_2(N/B) \cdot \log_{10}(\hat{M}/B))$.

Impact of Holistic Design. Holistic design offers the optimal design across all knobs for any parameterization with respect to workload, underlying hardware, and available resources (main memory and storage devices). Figure 11(a)–(d) compare MonkeyHolistic with the state of the art, as we vary the available memory (bits per element) on the x-axis. The different lines in each figure show how each optimization step towards MonkeyHolistic affects the overall performance. In the top of every figure we show with black the optimal design. For example, in Figure 11(a) the first point is tuned to "L4" and "0%." This means that the optimal design is leveling with size ratio $T = 4$, and the memory allocation for Bloom filters is 0%. The blue labels correspond to the MonkeyHybrid depicted with a blue line. For each of the four figures we see that the performance of the state of the art (black line with marker $*$) can be further optimized first by using optimal Bloom filter allocation (dashed line), and then by optimizing the memory allocation (solid line) while still using state-of-the-art design with respect with size ratio ($T = 10$) and merge policy (leveling). As we have more memory, however, it makes sense to use tiering, because most of the runs that do not contain useful data will not be read by more efficient Bloom filters (the dotted line). This shows that to achieve the optimal we might have to select between leveling and tiering as well. The blue line shows the performance benefit achieved by optimizing for size ratio and merge policy and still maintaining the rule-of-thumb memory allocation. For read-heavy and read-mostly (Figure 11(b)–(d)) workloads the 95–5% rule-of-thumb gives close-to-optimal performance except when the available memory increases to more than 20 bits per element. At this point it is preferable to increase the size ratio aggressively and tune the memory between the buffer and the Bloom filter in a different manner. For a write-mostly workload (Figure 11(a)) the blue line is slightly worse than optimal across the spectrum, since it loses the opportunity to optimize for lower values of memory, where the optimal solution is to reduce memory allocation for filters down to 0% when needed. The key observation of this analysis is that to optimally tune an LSM-tree we need to perform multi-parametric optimization taking into account all possible design decisions at the same time, rather than solving many optimization problems iteratively.

Holistic Design with SLAs. Using this algorithm, we can also impose upper-bounds on lookup cost or update cost to support different service-level agreements (SLAs). To do so, the algorithm discards configurations for which either lookup or update cost exceeds an imposed upper-bound.

5 EXPERIMENTAL ANALYSIS

We now proceed to experimentally evaluate Monkey against state-of-the-art designs. We first show that Monkey's method of optimizing the Bloom filters significantly reduces lookup cost across the whole design space and for various workloads. We then demonstrate Monkey's ability to navigate the design space to find the design that maximizes throughput for a given application workload.

Hardware Platform. Our experimental platform was an AWS EC2 "m4.large" instance, which is equipped with 2 CPUs, 8GB of DRAM, and both an SSD-based and HDD-based storage volume. Specifically, the SSD-based secondary storage volume is an instance of the EBS type "io1" with 25,000 provisioned IOPS, and the HDD-based one, of the EBS type "st1" which is optimized for throughput and can deliver 149–450MB/s. Both storage devices had a bandwidth cap at 450MB/s.

Software Platform. We implemented Monkey on top of RocksDB, which is a well-known and widely used LSM-tree based key-value store, representative of the state of the art. The current implementation of RocksDB supports leveling but not tiering, and it assigns the same FPR to filters across all levels. We used a RocksDB API that enables scheduling merge operations from the application code to implement tiering. Furthermore, we embedded our algorithm from Appendix B) to find and set the optimal FPRs for the different Bloom filters depending on the level that the corresponding run is being created in. The default configuration is as follows: Size ratio is 2 (the size ratio at which leveling and tiering behave identically); buffer size is 1MB; the overall amount of main memory allocated to all Bloom filters is $\frac{M_{filt}}{N} = 5$ bits per element[8] (though Monkey

[8]Though designs in industry typically use 10 bits per entry, we use 5 bits per entry by default, because Monkey with 5 bits per entry achieves approximately the same performance or better than the state of the art with 10 bits per entry. We show this experimentally.

allocates these bits differently across different levels). We vary these configuration knobs throughout the evaluation. Furthermore, we enabled direct I/O and disabled the block cache to be able to fully control the memory budget across all experiments. This represents a worst case scenario where there is not enough memory for a cache and it allows us to measure the impact of Monkey on the pure LSM-tree structure. In Appendix E, we show that Monkey maintains its advantages when there is enough memory to devote to a block cache.

Default Workloads. The goal of our experiments is to evaluate Monkey against the state of the art under worst-case workloads (i.e., that maximize the I/O overhead for both lookups and updates). The reason is that many applications today require predictable performance, and so it is important to be able to provide guarantees that the system will not perform worse than a given SLA under any possible workload. For lookups, the worst-case workload pattern is when the target keys are uniformly randomly distributed both across the key space and across time, because this pattern minimizes the probability that a target entry is in one of the caches along the I/O stack. This maximizes the probability of issuing I/O to storage. For updates, we show in Appendix G that the worst-case pattern can be approximated using uniformly randomly distributed insertions across a much larger key space than the data size, as this causes most inserted entries to participate in the highest possible number of merge operations. Unless otherwise mentioned, all workloads that we use throughout this evaluation are uniformly randomly distributed across the key space and across time. This is in line with the default settings of the Yahoo Cloud Service Benchmark (YCSB) [30], which is widely used today for experimenting with LSM-tree based key-value stores [56, 60–63, 71]. Unless otherwise mentioned, every experiment is preceded by an initialization phase where we insert 1GB of key-value entries to an empty database where each entry is 1KB in size. The domain of keys is approximately one thousand times larger than the number of entries in all of our experiments, so relatively few entries get overwritten during the initialization phase. We begin with experiments that only evaluate point lookup performance to demonstrate the impact of Monkey's Bloom filters' allocation strategy in isolation. We then continue to evaluate mixed workloads consisting of both lookups and updates and demonstrate the impact of Monkey's adaptive merging and holistic memory allocation between the Bloom filters and the buffer.

Metrics. We repeat each experimental trial (data loading and queries) three times. For each trial, we measure the average lookup and update latency and the average number of I/Os per operation. The error bars in our figures represent one standard deviation for latency across trials.

Model Validation. For most of the experiments, we included a side-by-side comparison with the I/O cost models that we derived throughout the article. We did this by multiplying the number of I/Os predicted by our cost models for each of the trials by the average I/O latency as measured on our hardware platform. In Appendix L, we elaborate on the equations we used to predict performance across the different experiments. The reliability of our I/O cost models in general demonstrates that I/O was indeed the bottleneck throughout the experiments with both HDD and SSD.

Monkey Scales Better with Data Volume. In this set of experiments, we show that Monkey improves lookup latency by an increasing margin as the data volume grows. The first experiment doubles the data size for every iteration and measures average latency across 16K zero-result point lookups. Results are shown in Figure 12(A). The curves initially meet with 2^{12} entries as there is only 1 level in LSM-tree, and so the Bloom filters allocation of Monkey is the same as for RocksDB. Lookup latency for the RocksDB increases at a logarithmic rate as the number of data entries increases, as predicted by our cost model in Section 4.3. The reason is that with more data entries the number of levels in the LSM-tree increases at a logarithmic rate, and lookup latency for RocksDB

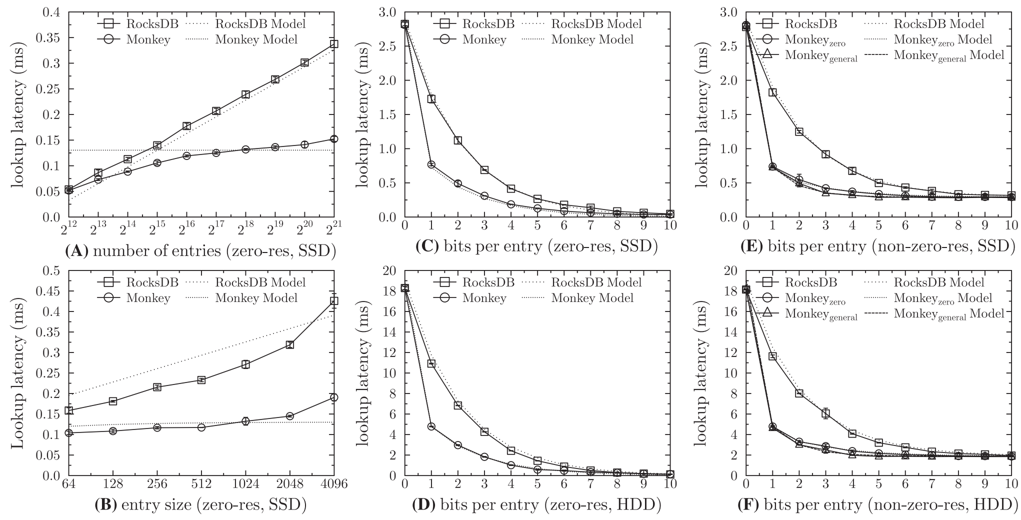


Fig. 12. Monkey improves lookup cost under any (A) number of entries, (B) entry size, and amount of memory for zero-result, (C) for SSD and (D) for HDD, and non-zero-result point lookups, (E) for SSD and (F) for HDD.

is proportional to the number of levels. In contrast, Monkey maintains a stable lookup latency as the number of entries increases, as also predicted by our cost models. The reason is that Monkey assigns exponentially decreasing FPRs to filters at smaller levels, and so the average number of I/Os per lookup converges to a constant that is independent of the number of levels. Overall, Monkey dominates RocksDB by up to 60%, and its margin of improvement increases as the number of entries grows. This demonstrates the asymptotic improvement that the Bloom filters optimization provides, shaving a logarithmic factor from point lookup cost.

Figure 12(B) depicts results for a similar experiment, with the difference that this time we keep the number of data entries fixed and we instead increase the entry size. This has the same impact on performance and for the same reasons as described above.

Monkey Needs Less Main Memory. In this set of experiments, we show that Monkey matches the performance of RocksDB using significantly less main memory. We set up this experiment by repeating the default experimental setup multiple times, each time using a different number of bits-per-entry ratio allocated to the filters, and measuring average latency across 16K point lookups. Figure 12(C), (D), (E), and (F) show the results for both zero-result and non-zero-result point lookups across an SSD and an HDD. Across all experiments, when the number of bits per entry is set to 0, both Monkey and RocksDB degenerate into an LSM-tree with no filters, and so lookup cost is the same. As we increase the number of bits per entry, Monkey significantly reduces lookup latency. Eventually, the filters for both systems become so accurate that the number of false positives approaches zero, at which point the curves nearly converge.

Figure 12(E) and (F) on the right side for non-zero-result point lookups show results for two versions of Monkey, $Monkey_{zero}$ and $Monkey_{general}$. The former is based on Section 4.1 and optimizes for zero-result point lookups, whereas the latter is based on Section 4.4 and finds the optimal Bloom filters allocation for any proportion between zero-result and non-zero-result point lookups in the workload. As there are no zero-result point lookups in the workload, the workload in this experiment is a special case where the Bloom filter at the largest level is not useful, and so $Monkey_{general}$ allocates the main memory budget only to filters at Levels 1 to $L - 1$. With 1 bit per entry, the memory budget M_{filt} with $Monkey_{zero}$ is below $M_{threshold}$, meaning that it has no filter

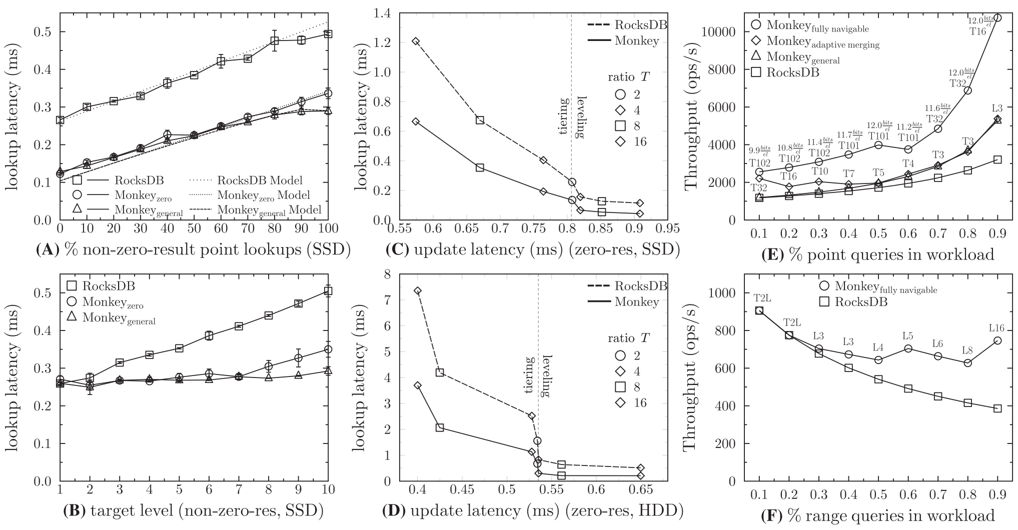


Fig. 13. Monkey improves lookup cost for any (A) mix between zero-result and non-zero-result point lookups and (B) temporal skew for lookups. It enables better tradeoffs between the costs of updates and lookups—(C) for SSD and (D) for HDD—and it navigates the tradeoff to find the design that maximizes throughput for point queries (E) and range queries (F).

for the largest level, and so the behavior of $Monkey_{zero}$ and $Monkey_{general}$ is identical. As main memory increases, $Monkey_{general}$ improves on $Monkey_{zero}$ by allocating its entire memory budget to filters at levels 1 to $L - 1$ thereby eliminating more false positives. By so doing, it improves lookup latency on $Monkey_{zero}$ by ≈10%.

Systems in industry typically use 10 bits per entry across all Bloom filters. The experiments show that Monkey matches this performance with ≈6 bits per entry for zero-result point lookups and with ≈4 bits per entry for non-zero-result point lookups (though the asymptotic improvement of Monkey implies that the margin of improvement increases as a function of the number of entries).

Monkey Is Better for Mixed Point Lookup Workloads. In Figure 13, we show that Monkey dominates the state of the art for any ratio of zero-result to non-zero-result point lookups. With only zero-result point lookups, $Monkey_{zero}$ and $Monkey_{general}$ behave exactly the same, and so their performance characteristics are identical. As the proportion of non-zero-result point lookups increases to 70% and beyond, however, $Monkey_{general}$ outperforms $Monkey_{zero}$ by allocating increasingly more memory to smaller levels and thereby eliminating more false positives for non-zero-result point lookups.

Monkey Improves Lookup Latency across Temporal Skews. This experiment shows that Monkey significantly improves lookup latency for non-zero-result lookups across a wide range of temporal locality in the query workload. In Figure 13(B), we compare RocksDB, $Monkey_{zero}$ and $Monkey_{general}$ as we vary temporal skew for non-zero-result point lookups. The x-axis in the experiment corresponds to the Level i in LSM-tree that the non-zero-result point lookups are targeting. Lookups to Level i target entries whose age is between $B \cdot P \cdot T^i$ and $B \cdot P \cdot T^{i+1}$ in terms of update order. Within a given level, the lookups are uniformly randomly distributed across the key space. When the lookups target Level 1, all schemes perform the same, because there is no opportunity for false positives to occur. As the target level increases, lookup latency for RocksDB degrades linearly, because each additional level increases the probability of false-positive rates by a fixed amount. With $Monkey_{zero}$ and $Monkey_{general}$, lookup performance stays approximately the same until Level 6, because hardly any false positives take place at these smaller levels. Beyond, $Monkey_{general}$ outperforms $Monkey_{zero}$ as it eliminates relatively more false positives at smaller levels than $Monkey_{zero}$ does.

Monkey Achieves a Better Tradeoff Curve. In this experiment, we show that Monkey pushes the tradeoff curve of the state of the art and is therefore able to navigate a better tradeoff continuum between update cost and zero-result lookup cost. We set up this experiment by repeating the experimental setup multiple times, each time using a different configuration of size ratio and merge policy and plotting the average latencies of lookups and updates and against each other for Monkey and RocksDB. To measure the full amortized update cost, we issue 1M uniformly randomly distributed insertions, i.e., the same as the initial number of entries in the LSM-tree. To measure worst-case point lookup cost, we measure latency across 16K zero-result point lookups. The result is shown in Figure 13(C) and (D) for SSD and HDD, respectively. The key observation is that for any configuration, Monkey achieves a significantly lower lookup cost than RocksDB due to the optimization of its Bloom filters, as predicted by our analysis in Section 4.3. Hence, Monkey shifts the tradeoff curve downwards. As a result, Monkey improves lookup cost by up to 50% in this experiment, and this gain can be traded for an improvement of up to 70% in update cost by adjusting the size ratio and merge policy to become more update-friendly.

Monkey Navigates the Design Space to Maximize Throughput. In our next set of experiments, we demonstrate Monkey's ability to navigate the design space to find a configuration that maximizes throughput for a given application workload. We set up the first experiment as follows. We repeat the default experimental setup multiple times, with the difference that during the query processing phase we vary the ratio of zero-result lookups to updates from 10% to 90%, and we perform the experiment by issuing 1M lookup/update operations. We present the results in Figure 13(E) on an SSD. The figure compares three variants of Monkey to show the performance improvement gained from each individual design decision: $Monkey_{general}$ only optimizes the Bloom filters, $Monkey_{adaptive\ merging}$ also adapts the size ratio and merge policy, and $Monkey_{fully\ navigable}$ also adapts the memory allocation between the Bloom filters and buffer. We show the optimal configurations chosen by $Monkey_{adaptive\ merging}$ and $Monkey_{fully\ navigable}$ above the corresponding points in the figure. T stands for tiering, L stands for leveling, the number alongside is the size ratio, and the number on top is the number of bits per entry allocated to the Bloom filters.

$Monkey_{general}$ dominates RocksDB by reducing the number of false positives. Furthermore, Monkey$_{adaptive\ merging}$ dominates $Monkey_{general}$ by merging more or less when the proportion of lookups vs. updates is higher or lower, respectively. $Monkey_{fully\ navigable}$ improves on $Monkey_{adaptive\ merging}$ by using more write-optimized merging and reallocating memory from the buffer to the Bloom filters to make up for having more runs to potentially have to probe. Overall, $Monkey_{fully\ navigable}$ improves throughout by at least 50% in all cases relative to RocksDB by finding the best design.

In Figure 13(F), we vary the proportion between small range lookups (with a selectivity of ≈0.001% of the dataset) and updates in the workload. As there are no point lookups in this experiment, the Bloom filters do not yield any performance benefit, and so $Monkey_{fully\ navigable}$ allocates the space that the Bloom filters would otherwise consume to the buffer to improve update cost. Moreover, $Monkey_{fully\ navigable}$ switches to increasingly read-optimized merging as the proportion of small range lookups increases. An intriguing observation is that the curve for Monkey is bell shaped. The reason is that as a workload tends towards a single operation type, the more possible it is to achieve a higher throughput as a single configuration handles most

Table 4. Default Designs for Mainstream LSM-tree Based Key-value Stores in Industry

	LevelDB	**RocksDB**	**Cassandra**	**HBase**	**Accumulo**
Size ratio (T)	10 *fixed*	10	4	2	4
Merge policy	Leveled *fixed*	Leveled *fixed*	Tiered *semi-fixed*	Leveled	Leveled
Bits per entry (M_{filt}/N)	10	10	10	10	11
Buffer size (M_{buf})	2MB	64MB	$\propto$ M	128MB	1GB

operations well. Overall, Monkey dominates RocksDB by up to ≈50% as the proportion of short range lookups increases.

6 RELATED WORK

6.1 LSM-Tree Based Key Value Stores

As LSM-tree based key-value stores in industry [7, 27, 40, 43, 44, 51, 63, 70], Monkey uses Bloom filters in main memory to probabilistically enable lookups to skip probing runs of the LSM-tree that do not contain a target key. State-of-the-art LSM-tree based key-value stores assign the same false-positive rate to every Bloom filter regardless of the size of the run that it corresponds to [7, 27, 40, 43, 44, 51, 63, 70]. In this work, we observe that worst-case lookup cost is proportional to the sum of the false-positive rates of all filters. Assigning equal false-positive rates to all of them, however, does not minimize this sum.

In contrast, Monkey minimizes the sum of false-positive rates by setting the false-positive rate for each Bloom filter to be proportional to the number of entries in the run that it corresponds to (meaning that runs at smaller levels have exponentially lower false-positive rates). This reduces the asymptotic complexity of worst-case lookup cost by a logarithmic factor of the size of the data.

Holistic Design. Table 4 gives the default design knobs for several mainstream key-value stores in industry: LevelDB, RocksDB, Cassandra, HBase, and Accumulo. We explain in detail how we obtained the values in this table in Appendix K. Some systems restrict the design knobs and thereby only include a subset of the design space. For example, LevelDB hard codes the size ratio to 10 and only enables a leveled merge policy. RocksDB enables tuning the size ratio, but it only allows a leveled merge policy. Cassandra allows tiering with any size ratio, but for leveling it fixes the size ratio to 10. As a result, these systems do not enable the entire tradeoff spectrum between the costs of lookups and updates. Across all these systems, the balance between lookup cost, update cost and main memory footprint depends on a combination of interdependent design and environmental parameters that must be precisely co-tuned.

In this way, designing a key-value store instance for a particular application is effectively a trial and error process that depends on intuition and experience of highly qualified engineers. Monkey represents a step to make this process more automatic and effortless. First, it understands and manages new design knobs that influence worst-case lookup cost, update cost, and main memory footprint, allowing it to assume new design that was not always possible before. Second, it uses worst-case closed-form models that enable optimizing throughput automatically by optimally setting those new knobs. Finally, it answers what-if questions regarding how changes in environmental parameters (workload and hardware) affect performance.

Recent complementary work [53] uses a numeric method to estimate update cost in a leveled LSM-tree when there is a skew in the update pattern. We do one step further here; we model both lookup cost and update costs under both leveled and tiered LSM-trees thereby enabling a holistic design over the entire design space.

Scheduling Merge Operations. To maintain stable performance, all LSM-tree based key-value stores spread the work done by merge operations over time. Some designs pace merge operations directly with respect to application updates [17, 52, 63]. Others partition a run into multiple files (i.e., often called Sorted String Tables or SSTables for short) and merge one file at a time with the set of files in the next level that have overlapping ranges [7, 27, 40, 43, 44, 49, 51, 63, 67, 70]. Other recent work proposes speeding up merging using dedicated servers [1] and SSD parallelism [69]. Since our work focuses on the total amount of work done by merge operations rather than how this work is scheduled, any of the above techniques can be used in conjunction with Monkey.

Reducing Merge Overheads. Various complementary approaches have been devised to reduce merge overheads. Recent designs partition runs into segments (typically called SSTables) and prioritize merging the segment with the least key range overlap with the next level [16, 65, 67]. Another approach opportunistically merges parts of runs that had recently been scanned and are therefore already in memory [59]. Other designs [25, 29, 56] store large entries in a separate log to avoid their inclusion in merging. Accordion packs entries more densely into the buffer to make flushing and merging less frequent [20], while TRIAD keeps frequently updated entries in the buffer to avoid repeatedly merging them [16]. Monkey is able to leverage such techniques while in addition also adapting the size ratio, the merge policy, and the memory allocation among the buffer and other memory resident structures to find the best cost balance between the overheads of merging and lookups

Dostoevsky [33] builds on top of Monkey by introducing Lazy Leveling, a merge policy that relies on having lower false-positive rates at smaller levels to relax merging at those levels. This improves write cost while maintaining the same bound on point lookup cost. In this way, the Bloom filters' optimization opens up new avenues for making merging for key-value stores cheaper.

6.2 Other Key-Value Store Designs

Fractional Cascading. Some variants of LSM-tree use fractional cascading [28] rather than fence pointers to ensure a cost of 1 I/O per run [17, 52]. The key idea is to embed the first key in every storage block X of a run within the next younger run alongside a pointer to block X. This enables a lookup to skip from a given run immediately to the relevant key range in the next run with one I/O. Variants of this method partition runs into individual block-sized nodes thereby de-amortizing merge overheads [9, 21, 24, 48, 58]. Designs using fractional cascading are generally incompatible with in-memory Bloom filters. The reason is that if we use a Bloom filter to skip probing a run, we cannot use the pointers in the run that we skipped to narrow down the search in the next run. As a result, the cost complexity for point lookups for such schemes is $O(L)$ with leveling and $O(L \cdot T)$ with tiering. In contrast, our work relies on using more main memory in the form of fence pointers and Bloom filters to reduce I/O for point lookups to $O(e^{-M_{filt}/N})$ with leveling and $O(T \cdot e^{M_{filt}/N})$ with tiering.

Unbounded Largest Level. Some log-structured designs have fixed capacities for smaller levels and an unbounded capacity for the largest level [11, 12, 54]. Such designs can be thought of as consisting of a small log and a larger sorted array. When the log reaches a fixed size threshold of X entries, it gets merged into the sorted array. In such designs, update cost is $O(N/X)$, meaning it increases linearly with respect to the data size N. In contrast, Monkey uses an LSM-tree based design where the number of levels grows at a logarithmic rate with respect to N. Since every entry gets merged a constant number of times per level, update cost is logarithmic in the data size: $O(\log_T(N))$. In this work, we restrict the scope of analysis to the latter class of designs.

Log-Structured Hash-Tables. Many key-value store designs have been proposed where key-value pairs are stored in a log in secondary storage, and there is a hash table in main memory that maps from application keys to the location of the corresponding key-value entry in the log [3, 4, 15, 26, 34, 35, 64]. Such designs have an update cost of $O(1/B)$, which is cheaper than with LSM-tree based designs. The two main problems with log-based designs are that (1) they do not support efficient range lookups as the entire dataset has to be scanned for every query and (2) storing a mapping for every entry consumes a lot of main memory. By contrast, LSM-tree based designs support efficient range lookups, and the mapping in main memory is at the granularity of blocks (i.e., for the fence pointers) rather than entries, thus taking up significantly less main memory.

In-Memory Stores. Key-value stores such as Redis [61] and Memcached [42] store application data in main memory rather than persistently in secondary storage. We have focused on mapping the design space of persistent key-value stores in this article, and so this work is orthogonal to in-memory efforts. However, given that similar tradeoffs exist in a pure in-memory environment to minimize cache-misses, we expect that a similar study to map the design space of in-memory key-value stores can be beneficial.

7 CONCLUSION

We show that LSM-tree based key-value stores exhibit an intrinsic tradeoff among lookup cost, update cost, and main memory footprint. We show how to achieve a better performance tradeoff curve by allocating the Bloom filters in a way that minimizes the sum of their false-positive rates. In addition, we show that it is possible to devise closed-form models to navigate the design space to find the holistic design that maximizes throughput under a given main memory budget, application workload, and storage medium.

We introduced the particular design knob of variable Bloom filter false-positive rates across different levels to achieve better performance tradeoffs. In Dostoevsky [33], we leveraged having much lower false-positive rates at smaller levels to relax merging and thereby achieving even better tradeoffs. However, the design space of key-value stores is even broader [45, 46], and new types of approximate set membership (ASM) structures continue to emerge [18, 41, 72]. Future work includes introducing and studying further design knobs, showing how to optimize them as well as how they combine with different ASM structures, and thereby continuing to push and improve the performance and space tradeoff curves for key-value stores.

APPENDIX

A OPTIMAL FALSE POSITIVE RATES

In this appendix, we derive the optimal false-positive rates (FPRs) $p_1 \ldots p_L$ by optimizing lookup cost R in Equation (3) with respect to the main memory footprint M_{filt} in Equation (4). To do so, we use the method of Lagrange Multipliers to find the FPRs that minimize Equation (4) subject to Equation (3) as a constraint. We show the detailed derivation for leveling, and we then give the result for tiering as it is a straightforward repetition of the process with the formulation of Equation (3) for tiering. We first express Equation (3) for leveling and Equation (4) in the standard form:

$$g(p_L \ldots p_1, R) = p_L + p_{L-1} + \cdots + p_1 - R$$
$$y(p_L \ldots p_1, N, T) = -\frac{N}{\ln(2)^2} \cdot \frac{T-1}{T} \cdot \sum_{i=1}^{L} \frac{\ln(p_i)}{T^{L-i}}.$$

We can now express the Lagrangian in terms of these functions:

$$\mathcal{L}(p_L \ldots p_1, N, T, R, \lambda) = y(p_L \ldots p_1, N, T) + \lambda \cdot g(p_L \ldots p_1, R).$$

Next, we differentiate the Lagrangian with respect to each of the FPRs $p_1 \ldots p_L$, and we set every partial derivative to 0. Thus, we arrive at the following system of equations:

$$\frac{N}{\ln(2)^2 \cdot \lambda} \cdot \frac{T-1}{T} = P_{L-i} \cdot T^i.$$

We equate these equations to eliminate the constants.

$$P_L \cdot T^0 = P_{L-1} \cdot T^1 = \cdots = P_1 \cdot T^{L-1}.$$

We now express all of the optimal FPRs in terms of the optimal FPR for Level L: P_L.

$$P_{L-i} = \frac{P_L}{T^i}.$$

Next, we express Equation R in terms of only T and P_L by plugging these FPRs into Equation (3). We observe that R is now expressed in terms of a geometric series. We simplify it using the formula for the sum of a geometric series up to L elements.

$$\begin{aligned} R &= \frac{p_L}{T^0} + \frac{p_L}{T^1} + \cdots + \frac{p_L}{T^{L-1}} \\ &= p_L \cdot \frac{(\frac{1}{T})^L - 1}{\frac{1}{T} - 1}. \end{aligned} \tag{29}$$

We now rearrange and express generically in terms of the FPR for Level i. The result appears in Equations (30) and (31) for leveled and tiered LSM-trees, respectively. These equations take R, T and L as parameters, and they return FPR prescriptions for any Level i such that the least possible amount of main memory M_{filt} is used to achieve the user-specified value of R.

Leveling

$$\begin{aligned} p_i &= \frac{R}{T^{L-i}} \cdot \frac{T^{L-1}}{T^L - 1} \cdot (T-1) \\ &\text{for } 0 < R \leq \frac{T^L - 1}{T^{L-1}} \cdot \frac{1}{T-1} \\ &\text{and } 1 \leq i \leq L, \end{aligned} \tag{30}$$

Tiering

$$\begin{aligned} p_i &= \frac{R}{T^{L-i}} \cdot \frac{T^{L-1}}{T^L - 1} \\ &\text{for } 0 < R \leq \frac{T^L - 1}{T^{L-1}} \\ &\text{and } 1 \leq i \leq L. \end{aligned} \tag{31}$$

The key difference between Equations (30) and (31) is that the optimal false-positive rate prescribed to any Level i is $(T-1)$ times lower under tiering than under leveling. The reason is that with tiering each level contains $(T-1)$ runs, and so the false-positive rate has to be $(T-1)$ times lower to keep R fixed.

Supporting the Whole Range for R. The highest possible value of R is the number of runs in the LSM-tree: L and $L \cdot (T-1)$ for leveling and tiering, respectively. Nevertheless, Equations (30) and (31) are undefined when R is set above $\frac{T^L-1}{T^{L-1}} \cdot \frac{1}{T-1}$ under leveling and $\frac{T^L-1}{T^{L-1}}$ under tiering. The reason is that as R grows beyond these bounds, the FPR prescriptions begin to exceed 1. This violates the constraint that a false-positive rate can be at most 1.

We now show how to adapt Equations (30) and (31) to handle larger values of R. The first key insight is that the FPR at level i is strictly greater than the FPR at level $i-1$. This means that as R grows, p_i converges to 1 before p_{i-1}. Therefore, as R increases the FPRs converge to 1 for the different levels in the order of largest to smallest. Hence, we can denote $L_{unfiltered}$ as the number of levels from level L and smaller whose FPRs converged to 1, whereas $L_{filtered}$ is the number of levels from Level 1 and larger with FPRs lower than 1. This partitioning of the levels is shown in Figure 6. Note that $L = L_{filtered} + L_{unfiltered}$.

The second key insight is that the sum of FPRs for the filtered levels $L_{filtered}$ can never be greater than 2 with leveling or $2 \cdot (T-1)$ with tiering, because the FPR at the largest of these levels with filters is at most 1, and the rest of the FPRs are exponentially decreasing. This means that if R is greater than these bounds, then $L_{unfiltered}$ must be non-zero. In fact, it implies that $L_{unfiltered}$ is equal to $\max(0, \lfloor R-1 \rfloor)$ with leveling and to $\max(0, \lfloor \frac{R-1}{T-1} \rfloor)$ with tiering.

The third key insight is that we can now apply Equations (30) and (31) on a smaller version of the problem with $L_{unfiltered}$ levels with Bloom filters and where the sum of false positives for these levels is $R - L_{unfiltered}$ with leveling and $R - L_{unfiltered} \cdot (T-1)$ with tiering. Our adaptations appear in Equations (32) and (33), respectively. We use L_u to denote $L_{unfiltered}$ in these equations for brevity.

Leveling

$$p_i = \begin{cases} 1, \text{if } i > L - L_u \\ \frac{R-L_u}{T^{(L-L_u)-i}} \cdot \frac{T^{(L-L_u)-1}\cdot(T-1)}{T^{(L-L_u)}-1}, \text{else} \end{cases}$$
$$\text{for } 0 < R \leq L$$
$$\text{and } 1 \leq i \leq L$$
$$\text{and } L_u = \max\left(0, \lfloor R-1 \rfloor\right), \tag{32}$$

Tiering

$$p_i = \begin{cases} 1, \text{if } i > L - L_u \\ \frac{R-L_u\cdot(T-1)}{T^{(L-L_u)-i}} \cdot \frac{T^{(L-L_u)-1}}{T^{(L-L_u)}-1}, \quad \text{else} \end{cases}$$
$$\text{for } 0 < R \leq L \cdot (T-1)$$
$$\text{and } 1 \leq i \leq L$$
$$\text{and } L_u = \max\left(0, \left\lfloor \frac{R-1}{T-1} \right\rfloor\right). \tag{33}$$

Simplification. As the number of levels L grows, Equations (30) and (31) converge to $P_i = \frac{R}{T^{L-i+1}} \cdot (T-1)$ with leveling and to $P_i = \frac{R}{T^{L-i+1}}$ with tiering. These simplified equations already accurately approximate the optimal false-positive rates when L is $\approx$5 or above. We can use this insight to simplify Equations (32) and (33) into Equations (5) and (6), which appear in Section 4.1. For practical analysis and implementations, we recommend using Equations (32) and (33).

A.1 Modeling Memory Footprint and Lookup Cost

We now show how to derive a closed-form model for main memory utilization for the filters M_{filt} and for lookup cost R in Monkey. We begin with the assumption that there are filters at all levels, but we later extend the model to also support the case where there are no filters at all levels (i.e., $L_{unfiltered} = 0$). Our step-by-step example is for leveling, but the case for tiering is identical, except we need to use Equation (6) rather than Equation (5). We first plug in the optimal false-positive rates in Equation (5) (for leveling) into Equation (4), which gives the main memory utilization by the Bloom filters with respect to the false-positive rates,

$$M_{filt} = -\frac{N}{\ln(2)^2} \cdot \frac{T-1}{T} \cdot \left(\sum_{i=0}^{L} \frac{1}{T^i} \cdot \ln\left(R \cdot \frac{T-1}{T^{1+i}} \right) \right).$$

We then apply logarithm operations to get the following:

$$M_{filt} = -\frac{N}{\ln(2)^2} \cdot \frac{T-1}{T} \cdot \ln\left(\frac{R^{1+\frac{1}{T}+\frac{1}{T^2}+\cdots} \cdot (T-1)^{1+\frac{1}{T}+\frac{1}{T^2}+\cdots}}{T^{\frac{1}{T^0}+\frac{2}{T^1}+\frac{3}{T^2}+\cdots}} \right).$$

To simplify the above equation, we apply the formula for the sum of geometric series to infinity on the exponents of the numerator, and we apply the sum of an geometric series to infinity on the exponents of the denominator. We get the following after some further simplification.

$$M_{filt} = \begin{cases} \dfrac{N}{\ln(2)^2} \cdot \ln\left(\dfrac{T^{\frac{T}{T-1}}}{R \cdot (T-1)}\right) & \text{with leveling} \\ \frac{N}{\ln(2)^2} \cdot \ln\left(\frac{T^{\frac{T}{T-1}}}{R}\right) & \text{with tiering.} \end{cases} \tag{34}$$

We now extend this equation to the case where $L_{unfiltered} > 0$. In this case, the filters for Bloom filters whose FPRs converged to zero take up no space, and so we only need to find the amount of space occupied by filters in the smaller $L_{filtered}$ levels. To do so, we adjust Equation (34) by applying it on a smaller version of the problem with $N/T^{L_{unfiltered}}$ entries (i.e., the number of entries in the smaller $L_{filtered}$ levels), and we discount the I/Os to the unfiltered levels by subtracting the number of runs in those levels from R,

$$M_{filt} = \begin{cases} \dfrac{N}{\ln(2)^2 \cdot T^{L_{unfiltered}}} \cdot \ln\left(\dfrac{T^{\frac{T}{T-1}}}{(R - L_{unfiltered}) \cdot (T-1)}\right) & \text{for leveling} \\ \frac{N}{\ln(2)^2 \cdot T^{L_{unfiltered}}} \cdot \ln\left(\frac{T^{\frac{T}{T-1}}}{R - L_{unfiltered} \cdot (T-1)}\right) & \text{for tiering.} \end{cases} \tag{35}$$

Lookup Cost. We now rearrange Equation (34) to be in terms of lookup cost R,

$$R_{filtered} = \begin{cases} \dfrac{T^{\frac{T}{T-1}}}{T-1} \cdot e^{-\frac{M_{filt}}{N} \cdot \ln(2)^2} & \text{with leveling} \\ T^{\frac{T}{T-1}} \cdot e^{-\frac{M_{filt}}{N} \cdot \ln(2)^2} & \text{with tiering.} \end{cases} \tag{36}$$

The above equation is still not adapted to the case where M_{filt} is so low that some of the filters at deeper levels cease to exist. To adapt it to this case, we first find the threshold point $M_{threshold}$ at which the FPR for filters at the highest level has converged to 1. To do so, we plug in the bounds for R that are $\frac{T}{T-1}$ and T under tiering and leveling, respectively, into Equation (34). The result simplifies into the following for both leveling and tiering,

$$M_{threshold} = N \cdot \frac{1}{(T-1)} \cdot \frac{\ln(T)}{\ln(2)^2}.$$

As M drops below $M_{threshold}$, every time that it is reduced by a factor of T, the filters at the next deeper level converge to 1. Thus, we can compute the number of levels with no filters as shown in Equation (37),

$$L_{unfiltered} = \begin{cases} 0, & M_{threshold} \leq M_{filt} \\ \left\lceil \log_T\left(\frac{M_{threshold}}{M_{filt}}\right) \right\rceil, & \frac{M_{threshold}}{T^L} \leq M_{filt} \leq M_{threshold} \cdot \\ L, & 0 \leq M_{filt} \leq \frac{M_{threshold}}{T^L} \end{cases} \tag{37}$$

Now, in the largest levels with no filters, we need to probe every run, which costs $L_{unfiltered}$ I/O with leveling and $(T-1) \cdot L_{unfiltered}$ I/O with tiering. In the levels with filters, the average number of runs we must probe is equivalent to the sum of their false positives, and so we can apply Equation (36) on a smaller version of the problem with $N/T^{L_{unfiltered}}$ levels. This becomes Equation (7) in Section 4.2.

B OPTIMIZING THE FILTERS

In Section 4.1 we showed that the optimal FPRs are proportional to the number of entries in each level. Our analysis assumed that the entry size is fixed, and so we could easily infer the number of elements in each level and thereby allocate the optimal FPR to its filters. In practice, however, the average entry size may be variable, or it may change over time. If this is the case, then we can no longer infer the number of elements in each level. To handle this, we extend the implementation of Monkey to record the number of entries for each run as metadata. We then use this metadata to find the optimal false-positive rates using Algorithm 2 (and auxiliary Algorithms 3 and 4). Algorithm 2 takes as parameters (1) the overall amount of main memory M_{filt} to allocate the Bloom filters, and (2) a *runs* vector with one pair for each run where *runs[i].entries* is the number of entries and *runs[i].bits* is the number of bits allocated to the Bloom filter of the *i*th run. The algorithm iteratively moves main memory among the different Bloom filters until the sum of their false positives is minimized. This algorithm does not need to run often, and it takes a fraction of a second to execute on our experimentation platform.

ALGORITHM 2: Allocate M_{filt} to minimize the sum of FPRs.

```
AutotuneFilters ($M_{filt}$, $runs$)
    $\Delta = M_{filt}$;
    $runs[0] = M_{filt}$;
    $R = runs.length - 1 +$ eval $(runs[0].bits, runs[0].entries)$;
    while $\Delta \geq 1$ do
        $R_{new} = R$;
        for int $i = 0; i < runs.length - 1; i$++ do
            for int $j = i + 1; i < runs.length; j$++ do
                $R_{new}$ = TrySwitch$(runs[i], runs[j], \Delta, min)$;
                $R_{new}$ = TrySwitch$(runs[j], runs[i], \Delta, min)$;
            end
        end
        if $R_{new} == R$ then
            $\Delta = \Delta/2$;
        $R = R_{new}$;
    end
    return $R$;
```

ALGORITHM 3: Moves Δ bits to *run*1 from *run*2 if it reduces *R*.

```
TrySwitch ($run1, run2, \Delta, R$)
    $R_{new} = R -$ eval $(run1.bits, run1.entries) -$ eval $(run2.bits, run2.entries) +$ eval $(run1.bits +$
    $\Delta, run1.entries) +$ eval $(run2.bits - \Delta, run2.entries)$;
    if $R_{new} < R$ and $run2.bits - \Delta > 0$ then
        $R = R_{new}$;
        $run1.bits$ += $\Delta$;
        $run2.bits$ −= $\Delta$;
    return $R$;
```

ALGORITHM 4: Returns the false-positive rate of a Bloom filter.

```
eval ($bits, entries$)
    return $e^{-(bits/entries) \cdot ln(2)^2}$;
```

C OPTIMIZING MEMORY ALLOCATION

We want to find the optimal memory allocation between Bloom filters and the buffer that minimizes the workload cost: $cost(x) = z \cdot Z(x) + q \cdot Q(x) + w \cdot W(x)$, where x is the memory given to

Bloom filters. Throughout these derivations, we use the following equations. The terms $Z_{leveling}$, $Q_{leveling}$, and $W_{leveling}$ are used to calculate the leveling cost and the terms $Z_{tiering}$, $Q_{tiering}$, and $W_{tiering}$ are used for the tiering cost.

$$\begin{aligned}
Z_{leveling}(x) &= e^{-\frac{x}{N}\cdot \ln(2)^2} \cdot \frac{T^{\frac{T}{T-1}}}{T-1} \cdot \left(\frac{1}{v+1}\right)^{\frac{T-1}{T}} + \frac{v}{v+1} \\
Z_{tiering}(x) &= e^{-\frac{x}{N}\cdot \ln(2)^2} \cdot T^{\frac{T}{T-1}} \cdot \left(1 - \frac{v}{v+1}\cdot\frac{T}{T-1}\cdot\frac{1}{2}\right)^{\frac{T-1}{T}} + \frac{v}{v+1} \\
Q_{leveling}(x) &= s\cdot\frac{N}{B} + L(x), \quad Q_{tiering} = s\cdot\frac{N}{B} + L(x)\cdot(T-1) \\
W_{leveling}(x) &= \frac{(T-1)\cdot(1+\phi)}{2\cdot B}\cdot L(x), \quad W_{tiering} = \frac{(T-1)\cdot(1+\phi)}{T\cdot B}\cdot L(x) \\
&\text{where } L(x) = \frac{\ln\left(\frac{N\cdot E}{\hat{M}-x}\cdot\frac{T-1}{T}\right)}{\ln(T)}.
\end{aligned} \tag{38}$$

Factoring the Cost Function for Leveling. We first use the above terms for leveling to refactor the cost function with respect to x, the memory given to Bloom filters.

$$\begin{aligned}
cost(x, \text{“leveling”}) &= z\cdot Z_{leveling}(x) + q\cdot\left(s\cdot\frac{N}{B} + L(x)\right) + w\cdot\frac{(T-1)\cdot(1+\phi)}{2\cdot B}\cdot L(x) \\
&= z\cdot\left(e^{-\frac{x}{N}\cdot\ln(2)^2}\cdot\frac{T^{\frac{T}{T-1}}}{T-1}\cdot\left(\frac{1}{v+1}\right)^{\frac{T-1}{T}} + \frac{v}{v+1}\right) + q\cdot s\cdot\frac{N}{B} + \left(q + w\cdot\frac{(T-1)\cdot(1+\phi)}{2\cdot B}\right)\cdot L(x) \\
&= z\cdot\frac{T^{\frac{T}{T-1}}}{T-1}\cdot\left(\frac{1}{v+1}\right)^{\frac{T-1}{T}}\cdot e^{-\frac{x}{N}\cdot\ln(2)^2} + \frac{q + w\cdot\frac{(T-1)\cdot(1+\phi)}{2\cdot B}}{\ln(T)}\cdot\ln\left(\frac{N\cdot E}{\hat{M}-x}\cdot\frac{T-1}{T}\right) + z\cdot\frac{v}{v+1} + q\cdot s\cdot\frac{N}{B}.
\end{aligned} \tag{39}$$

Factoring the Cost Function for Tiering. We proceed to refactor the cost function for tiering with respect to x, the memory given to Bloom filters,

$$\begin{aligned}
cost(x, \text{“tiering”}) &= z\cdot Z_{tiering}(x) + q\cdot\left(s\cdot\frac{N}{B} + L(x)\cdot(T-1)\right) + w\cdot\frac{(T-1)\cdot(1+\phi)}{T\cdot B}\cdot L(x) \\
&= z\cdot\left(e^{-\frac{x}{N}\cdot\ln(2)^2}\cdot T^{\frac{T}{T-1}}\cdot\left(1-\frac{v}{v+1}\cdot\frac{T}{T-1}\cdot\frac{1}{2}\right)^{\frac{T-1}{T}} + \frac{v}{v+1}\right) \\
&\quad + q\cdot s\cdot\frac{N}{B} + \left(q\cdot(T-1) + w\cdot\frac{(T-1)\cdot(1+\phi)}{T\cdot B}\right)\cdot\frac{\ln\left(\frac{N\cdot E}{\hat{M}-x}\cdot\frac{T-1}{T}\right)}{\ln(T)} \\
&= z\cdot T^{\frac{T}{T-1}}\cdot\left(1-\frac{v}{v+1}\cdot\frac{T}{T-1}\cdot\frac{1}{2}\right)^{\frac{T-1}{T}}\cdot e^{-\frac{x}{N}\cdot\ln(2)^2} + z\cdot\frac{v}{v+1} + q\cdot s\cdot\frac{N}{B} \\
&\quad + \frac{\left(q\cdot(T-1) + w\cdot\frac{(T-1)\cdot(1+\phi)}{T\cdot B}\right)}{\ln(T)}\cdot\ln\left(\frac{N\cdot E}{\hat{M}-x}\cdot\frac{T-1}{T}\right).
\end{aligned} \tag{40}$$

Equations (39) and (40) can be written more compactly as

$$cost(x, pol) = \alpha_{pol}\cdot e^{-\beta\cdot x} + \gamma_{pol}\cdot ln\left(\frac{\delta}{\hat{M}-x}\right) + C, \tag{41}$$

where

$$\alpha_{leveling} = z \cdot \frac{T^{\frac{T}{T-1}}}{T-1} \cdot \left(\frac{1}{v+1}\right)^{\frac{T-1}{T}}, \quad \alpha_{tiering} = z \cdot T^{\frac{T}{T-1}} \cdot \left(1 - \frac{v}{v+1} \cdot \frac{T}{T-1} \cdot \frac{1}{2}\right)^{\frac{T-1}{T}}, \tag{42}$$

$$\gamma_{leveling} = \frac{q + w \cdot \frac{(T-1)\cdot(1+\phi)}{2\cdot B}}{\ln(T)}, \quad \gamma_{tiering} = \frac{q\cdot(T-1) + w \cdot \frac{(T-1)\cdot(1+\phi)}{T\cdot B}}{\ln(T)}, \tag{43}$$

$$\beta = \frac{\ln(2)^2}{N}, \quad \delta = N \cdot E \cdot \frac{T-1}{T}, \quad C = z \cdot \frac{v}{v+1} + q \cdot s \cdot \frac{N}{B}. \tag{44}$$

For the remainder of this section, we will refer to both cost functions ($cost_T(x)$ for tiering and $cost_L(x)$ for leveling) as $cost(x)$, and we will differentiate only when needed.

Finding the optimal memory allocation now can be achieved by finding the x_{min} that minimizes the function in Equation (41). To find x_{min} we first prove that the cost function $cost(x)$ is convex in the interval $[0, \hat{M})$. We only care for $x \in [0, \hat{M})$, because the memory used for Bloom filters is necessarily greater or equal to zero and up to the overall memory available, making sure we can use a small portion for the buffer. To prove convexity we calculate the first and second derivatives.

$$cost'(x) = -\alpha \cdot \beta \cdot e^{-\beta \cdot x} + \gamma \cdot \frac{1}{\hat{M} - x}, \tag{45}$$

$$cost''(x) = \alpha \cdot \beta^2 \cdot e^{-\beta \cdot x} + \gamma \cdot \frac{1}{(\hat{M} - x)^2}. \tag{46}$$

For the useful ranges of values for all the parameters of the model, the terms α, β, γ, δ, and C are all positive, hence $cost''(x) > 0$ for $x \in [0, \hat{M})$. As a result, the minimum cost is given when $cost'(x) = 0$. Solving the latter in Equation (47) gives us the optimal memory allocation.

$$cost'(x) = 0 \Rightarrow -\alpha \cdot \beta \cdot e^{-\beta \cdot x} + \gamma \cdot \frac{1}{\hat{M} - x} = 0 \Rightarrow e^{-\beta \cdot x} = \frac{\gamma}{\alpha \cdot \beta} \cdot \frac{1}{\hat{M} - x} \Rightarrow$$

$$x = \frac{1}{\beta} \cdot ln\left(\frac{\alpha \cdot \beta}{\gamma} \cdot (\hat{M} - x)\right). \tag{47}$$

C.1 Finding the Optimal Memory Allocation using the Newton-Raphson Method

Equation (47) does not have an analytical solution. However, solving it is equivalent to finding the root of a function $h(x)$, which is defined as the difference between the left part and right part of Equation (47),

$$\text{find } x_0 \; s.t., \; h(x_0) = 0 \text{ for } h(x) = x - \frac{1}{\beta} \cdot ln\left(\frac{\alpha \cdot \beta}{\gamma} \cdot \left(\hat{M} - x\right)\right). \tag{48}$$

To guarantee that $h(x)$ has a solution in the interval $[0, \hat{M})$ we study its monotonicity using its first derivative, and its values at the end of the interval. The derivative is shown in Equation (49),

$$h'(x) = 1 + 1/\left(\beta \cdot \left(\hat{M} - x\right)\right). \tag{49}$$

Since $\beta > 0$ the derivative is always positive for $x \in [0, \hat{M})$, hence $h(x)$ is monotonically increasing. To guarantee that we have a solution we need $h(x)$ to have opposite sign at the beginning and the end of the interval $h(0) \cdot h(\hat{M}) < 0$. We start from the extreme for $x \approx \hat{M}$. Since x cannot be equal to $\hat{M}$ we consider $h(\hat{M} - \varepsilon)$ for $\varepsilon \to 0$ (from above, i.e., $\varepsilon > 0$),

$$h\left(\hat{M} - \varepsilon\right) = \hat{M} - \varepsilon - \frac{1}{\beta} \cdot \ln\left(\frac{\alpha \cdot \beta}{\gamma} \cdot \left(\hat{M} - \left(\hat{M} - \varepsilon\right)\right)\right) = \hat{M} - \varepsilon - \frac{1}{\beta} \cdot \ln\left(\frac{\alpha \cdot \beta}{\gamma} \cdot \varepsilon\right). \tag{50}$$

We know that $\alpha, \beta, \gamma > 0$, hence $\lim_{\varepsilon \to 0} h(\hat{M} - \varepsilon) = M - (-\infty) = +\infty$. As a result, to have a solution in $[0, \hat{M})$ we need $h(0) < 0$,

$$h(0) = 0 - \frac{1}{\beta} \cdot \ln\left(\frac{\alpha \cdot \beta}{\gamma} \cdot \left(\hat{M} - 0\right)\right) = -\frac{1}{\beta} \cdot \ln\left(\frac{\alpha \cdot \beta}{\gamma} \cdot \hat{M}\right). \tag{51}$$

Since $\alpha, \beta, \gamma > 0$, $h(0) < 0$ means

$$-\frac{1}{\beta} \cdot \ln\left(\frac{\alpha \cdot \beta}{\gamma} \cdot \hat{M}\right) < 0 \Rightarrow \ln\left(\frac{\alpha \cdot \beta}{\gamma} \cdot \hat{M}\right) > 0 \Rightarrow \frac{\alpha \cdot \beta}{\gamma} \cdot \hat{M} > 1 \Rightarrow \hat{M} > \frac{\gamma}{\alpha \cdot \beta}. \tag{52}$$

Furthermore, if $\hat{M} = \frac{\gamma}{\alpha \cdot \beta}$, then $h(0) = 0$, which means that the optimal memory allocation is to give no memory to Bloom filters. Note that we have derived a minimum amount of memory needed to start using Bloom filters, and this minimum amount of memory needed depends on the workload, the underlying hardware, and the current tuning with respect to size ratio and merge policy. With respect to finding the value of x_0 as in Equation (48), we now know:

- if $\hat{M} \leq \frac{\gamma}{\alpha \cdot \beta}$ then $x_0 = 0$, that is all memory goes to the buffer.
- if $\hat{M} > \frac{\gamma}{\alpha \cdot \beta}$ then we need to find the root of $h(x)$. This can be found using bisection or at a much faster convergence rate using the Newton-Raphson method.

Newton-Raphson Method. Now we present how to use the Newton-Raphson numerical method to find the root of $h(x)$ when it exists in the range $[0, \hat{M})$. To avoid numerical instabilities and since $x = \hat{M}$ is not an acceptable solution, we start by considering the maximum possible value of x in $[0, M)$, which in fact is $\hat{M} - \mathit{minimum\_page\_size}$ when we only use one page for the buffer.

If $h(\hat{M} - \mathit{minimum\_page\_size}) \leq 0$, then the solution is in the range $[\hat{M} - \mathit{minimum\_page\_size}, M)$, and we will assign $x_0 = M - \mathit{minimum\_page\_size}$, because this is the maximum amount of main memory that makes sense to use for Bloom filters; we need at least one page for the buffer.

If $h(\hat{M} - \mathit{minimum\_page\_size}) > 0$, then the solution is in the range $[0, \hat{M} - \mathit{minimum\_page\_size}]$, and we use the Newton-Raphson method to find the root, starting from $x_0 = \hat{M} - \mathit{minimum\_page\_size}$. Since the function is convex and monotonically increasing, the Newton-Raphson will converge if it starts from the right side of the root. Finally, now we execute the iterative numerical method using: $x_n = x_{n-1} - \frac{h(x_{n-1})}{h'(x_{n-1})}$. The overall algorithm is shown in Algorithm 5.

It can be shown that the convergence rate of the algorithm is quadratic, which can be pessimistically captured by assuming that with every new iteration the accuracy is increased by one decimal digit. Since our target accuracy is given by splitting memory in the granularity of a page size the algorithm needs $O(\log_{10}(\hat{M}/B))$ steps to reach the desired accuracy.

Algorithm 5 finds the roots of $h(x)$ solving Equation (48), which in turn gives is the value that solves Equation (47). This value minimizes the cost function(s) of Equation (41) and gives us the optimal memory allocation between Bloom filters and the buffer.

D BASELINE

In this Appendix we model the expected zero-result point lookup I/O cost for the state-of-the-art R_{art}. First we derive equations that reflect how the state of the art assigns FPRs to filters with both leveling and tiering. We do so by setting all false-positive rates $p_1, p_2 \ldots p_L$ in Equation (3) to be

ALGORITHM 5: Final Optimal Memory Allocation

```
optimalMemoryForFilters $(\hat{M}, T, policy; z, v, q, w, s, E, N, \phi, B)$
    $error\_tolerance = \frac{minimum\_page\_size}{\hat{M}}$; //stop optimizing when the granularity reaches the page size
    calculate: $\alpha = \alpha_{policy}$ //Equation (42)
    calculate: $\gamma = \gamma_{policy}$ //Equation (43)
    calculate: $\beta, \delta, C$ //Equation (44)
    if $\hat{M} \leq \frac{\gamma}{\alpha \cdot \beta}$ then
        $M_{filt} = 0$;
    else
        $x_0 = \hat{M} - minimum\_page\_size$;
        if $x_0 \leq 0$ then
            //this might happen only $\hat{M}$ is very small
            $M_{filt} = 0$;
        else
            //the iterative Newton-Raphson algorithm
            // using Equation (48)
            if $h(x_0) \leq 0$ then
                //early termination because the solution is very close to $\hat{M}$
                $M_{filt} = \hat{M} - minimum\_page\_size$;
            else
                $error = \hat{M}$;
                while $error > error\_tolerance \cdot \hat{M}$ do
                    $x_1 = x_0 - \frac{h(x_0)}{h'(x_0)}$; // using Equations (48) and (49)
                    $error = |x_1 - x_0|$;
                    $x_0 = x_1$;
                end
                $M_{filt} = x_1$;
            end
        end
    end
    return $M_{filt}$;
```

equal to each other, simplifying, and rearranging. The results are Equations (53) and (54).

Leveling

$$p_i = \frac{R_{art}}{L} \quad \text{for } 0 < R_{art} \leq L, \tag{53}$$

Tiering

$$p_i = \frac{R_{art}}{L \cdot (T-1)} \quad \text{for } 0 < R_{art} \leq L \cdot (T-1). \tag{54}$$

Second, we plug the false-positive rates in Equations (53) and (54) into Equation (4) and simplify by applying logarithm operations and sums of geometric series. The result is shown in Equation (55),

$$M_{filt} = \frac{N}{\ln(2)^2} \cdot \left(1 - T^{-L}\right) \cdot \ln\left(\frac{L}{R_{art}}\right). \tag{55}$$

As L grows, Equation (55) converges into the following:

$$M_{filt} = \frac{N}{\ln(2)^2} \cdot \ln\left(\frac{L}{R_{art}}\right).$$

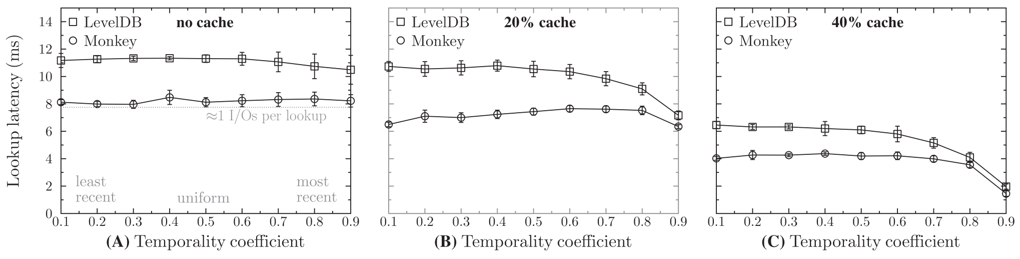


Fig. 14. Monkey maintains its advantages with a block cache. It utilizes it to query recently touched keys.

We can now rearrange in terms of R, for both leveling and tiering.

$$R_{art} = \begin{cases} L \cdot e^{-\frac{M_{filt}}{N} \cdot \ln(2)^2}, & \text{with leveling} \\ L \cdot (T-1) \cdot e^{-\frac{M_{filt}}{N} \cdot \ln(2)^2}, & \text{with tiering.} \end{cases} \tag{56}$$

Hence, the complexity of R_{art} is $O(L \cdot e^{-M_{filt}/N})$ for leveling and $O(L \cdot T \cdot e^{-M_{filt}/N})$ for tiering.

E CACHING

In the main part of the article, both RocksDB and Monkey do not use a cache. We do that by explicitly disabling the block cache from within RocksDB. We use this setup to isolate the impact of the new strategy for Bloom filter allocation to assess the potential of Monkey and its impact on the pure LSM-tree structure. In this experiment, we enable the block cache and we show that Monkey (1) maintains its advantage and (2) can utilize a cache when recently accessed keys are queried.

We set-up this experiment as follows. We activate the block cache feature of RocksDB, which caches recently accessed data blocks (16KB in our setup). We use three different settings whereby the block cache size is 0%, 20%, and 40% of the overall data volume. We repeat the default data loading setup from the main part of the article. Once data are loaded, we warm up the cache using non-zero-result lookups with different temporal localities. To control temporal locality, we use the temporality coefficient c (as in the experiment for Figure 12(D) in Section 5) whereby c percent of the most recently updated entries receive $(1-c)$ percentage of all lookups. When the cache is warm (i.e., full), we continue issuing the same workload and measure average lookup time. In this way, we test for various workloads in terms of how frequently accessed keys we query. By varying this parameter we are able to test various cases in terms of the utility of the cache.

The results are shown in Figure 14. Figure 14(A) serves as a reference by depicting performance when no cache is used. This is the same as we have observed in previous experiments with Monkey outperforming RocksDB across the whole workload range. We observe the same overall behavior with Monkey outperforming RocksDB across the two other graphs in Figure 14(B) and (C) when the cache is enabled. The difference here is that when we query very recently accessed keys, these keys will likely be in the cache. As a result, as we query more recently accessed items, both Monkey and RocksDB nearly converge. Monkey can utilize the cache for frequently accessed items in the same way as RocksDB. An additional property is that even when lookups mostly target a small set of entries with a cumulative size that is much smaller than the overall cache size (e.g., when c is set to 0.9 in Figure 14(B) and (C)), Monkey still maintains a small advantage (that collectively for many queries becomes important). The reason is that the cache cannot store the full set of frequently accessed entries and thereby eliminate I/O for accessing them; it is a block cache and so it stores one 16KB data block for every recently accessed 1KB key-value entry.[9]

Overall, Monkey utilizes the cache and maintains its performance advantage due to better allocation of the false-positive rates across the Bloom filters. The Bloom filters and the cache mitigate different sources of latency: false positives and true positives, respectively. In this work, we focused on optimizing the Bloom filters to reduce and control the rate of I/Os due to false positives. Further investigation could also be beneficial to tune the cache to reduce I/Os due to true positives.

F I/O COSTS FOR LOG AND SORTED ARRAY

In this Appendix, we give intuition for the I/O costs for a log and sorted array in Figures 4 and 8. The zero-result point lookup cost for a sorted array is $O(e^{-M_{filt}/N})$, because the false-positive rate (FPR) is $O(e^{-M_{filt}/N})$ and accessing the sorted array costs one I/O due to the fence pointers. The update cost for a sorted array is $O(\frac{N \cdot E}{M_{buf}} \cdot 1/B)$ I/Os, because every time that the buffer fills up with a data volume of M_{buf} bits and spills to storage, a merge operation involving the entire data volume of $N \cdot E$ bits take place, and we divide by B to give the unit in terms of storage blocks. The zero result point lookup cost for a log is $O(\frac{N \cdot E}{M_{buf}} \cdot e^{-M_{filt}/N})$, because there are $O(\frac{N \cdot E}{M_{buf}})$ runs on disk, and the access probability for each of them is $O(e^{-M_{filt}/N})$. The update cost for a log is $O(1/B)$, because every entry gets written once to disk, and we divide by B to give the unit in blocks.

G WORST-CASE WORKLOAD FOR UPDATES

In this appendix, we provide a proof for why the worst case workload for updates is when an entry gets updated at most once within a period of N updates. If an entry gets updated once per every window of $B \cdot P \cdot T^j$ or fewer updates overall, then it never gets beyond level $j + 1$, because a more recent update with the same key causes it to get discarded. Therefore, the update cost for the entry is $O(j/B)$ I/O with tiering and $O(T \cdot j/B)$ I/O with leveling. For example, if an entry that gets merged once every window of $B \cdot P$ updates (the buffer size), then it always gets eliminated by a more recent entry in level 1, and so the amortized I/O cost is $O(1/B)$ I/O with tiering and $O(T/B)$ I/O with leveling. Now let's suppose all entries updated by the application are updated again every period of $B \cdot P \cdot T^j$ updates. In this case, merge operations in the system never recurse beyond Level $j + 1$, because the resulting run is never large enough to spill to level $j + 1$ as most entries get eliminated by more recent versions. Thus, the overall update cost for any update in is $O(j/B)$ I/O with tiering and $O(T \cdot j/B)$ I/O with leveling. Using this cost model, we observe that update cost is maximized when J is equal to L, as this means that merge operations recurse to the largest level and incur the highest possible overhead. When j is set to L, then every entry gets updated once per window of N updates, which is the worst-case that we initially defined. This concludes the proof.

We can approximate a worst-case workload for updates by issuing uniformly randomly distributed insertions across a much larger key space than the data size. This ensures that the most entries do not get replaced, so most inserted entries incur the highest possible I/O overhead over time.

H DERIVING UPDATE COST

In this Appendix, we show how to derive update cost. With leveling, the jth run that arrives at a level triggers a merge operation involving the existing run at the level, which is the merged outcome of the previous $(T - j)$ runs that arrived since the last time the level was empty. To get

[9]Caching entries instead of blocks may be more effective though there may be side-effects with cache maintenance.

the overall write-amplification per level due to merging before a level runs out of capacity, we thus sum up $\sum_{j=1}^{T-1}(T-j)$. This is an arithmetic sequence that sums up to $\frac{T\cdot(T-1)}{2}$. We divide by the number of runs at a level T to get the average write-amplification across all the runs, thus resulting in $\frac{T-1}{2}$. For tiering, the cost of writing runs at a level before it reaches capacity is $\sum_{j=1}^{T-1} 1$, and we divide by T to get the average write-amplification per level per run: $\frac{T-1}{T}$. We then multiply both expressions by the number of levels L and by the cost ratio between writes and reads to storage $(1+\phi)$, and we divide by the block size B to get the measurement in I/Os. The outcome is Equation (10).

I FENCE POINTERS SIZE

In this Appendix, we derive the proportion between the size of the fence pointers and the raw data size. Every fence pointer is a 2-tuple of size $|FP|$ consisting of (1) an application key and (2) a pointer to some corresponding storage block in some run that begins with this given key. As the total number of blocks in LSM-tree is $\frac{N}{B}$ and there is one fence pointer for every block, the overall amount of space taken by the fence pointers is $\frac{|FP|\cdot N}{B}$. We divide this expression by the raw data size $N\cdot E$ to get the proportion between the fence pointers size and the raw data size as $\frac{|FP|}{B\cdot E}$, where $B\cdot E$ is the block size. Now, suppose that each fence pointer consists of a 16-byte key and an 8-byte pointer to storage, and that each disk block size is 16KB. In this case, the fence pointers take up 0.1% of the raw data size, which is three orders of magnitude smaller than the data size. This ratio may be even lower as the keys in the fence pointers are typically compressed.

J COMPLEXITY ANALYSIS OF POINT LOOKUP COST WITH MONKEY

Here we analyze the cost of zero-result point lookups in Monkey based on Equations (7) and (8) in Section 4.2. We start with the case where $M_{filters} > M_{threshold}$, meaning there are Bloom filters for all levels. In this case, $R_{unfiltered}$ is zero, and so we focus on analyzing the cost contribution of $R_{filtered}$. With leveling, $R_{filtered}$ in Equation (7) is $\frac{T^{\frac{T}{T-1}}}{T-1}\cdot e^{-\frac{M_{filt}}{N}\cdot\ln(2)^2}$ I/O, which simplifies to $O(e^{-M_{filt}/N})$ I/O, because $\frac{T^{\frac{T}{T-1}}}{T-1}$ is a small constant for any value of T. With tiering, $R_{filtered}$ Equation (7) is $T^{\frac{T}{T-1}}\cdot e^{-\frac{M_{filt}}{N}\cdot\ln(2)^2}$ I/O, which simplifies to $O(T\cdot e^{-M_{filt}/N})$ I/O as the expression $T^{\frac{T}{T-1}}$ simplifies to $O(T)$ asymptotically with respect to T. These expressions are maximized when M_{filt} is set to $M_{threshold}$, in which case they become $O(1)$ and $O(T)$ for leveling and tiering, respectively. Hence, $R_{filtered}$ is at most $O(1)$ with leveling and at most $O(T)$ with tiering.

We now analyze Monkey when $M_{threshold}/T^L < M_{filters} \leq M_{threshold}$, the case where we do not have Bloom filters for a subset of the larger levels. First, we note from Equations (7) and (8) that when $M_{filters}$ is equal to $M_{threshold}$, the value of $R_{unfiltered}$ is $O(1)$ with leveling and $O(T)$ with tiering. Second, we note that $R_{unfiltered}$ increases monotonically as $M_{filters}$ decreases. Third, we recall from above that $R_{filtered}$ is in general at most $O(1)$ with leveling and at most $O(T)$ with tiering. These three observations imply that $R_{unfiltered}$ dominates $R_{filtered}$ across this whole range of $M_{threshold}/T^L < M_{filters} \leq M_{threshold}$, and so we can analyze $R_{unfiltered}$ while ignoring $R_{filtered}$ as the contribution of $R_{filtered}$ to the I/O cost, in this case, is asymptotically negligible.

From Equation (7), we observe that $R_{unfiltered}$ is $O(L_{unfiltered})$ with leveling and to $O(T\cdot L_{unfiltered})$ with tiering. From Equation (8), we see that $L_{unfiltered}$ is $O(\log_T(M_{threshold}/M_{filt}))$ when $\frac{M_{threshold}}{T^L} < M_{filters} \leq M_{threshold}$. We proceed to simplify $L_{unfiltered}$ as follows in Equation (57). In part (57a, we plug in the value of $M_{threshold}$ from Equation (8). In part (57b), we apply the product rule on logarithms to split the log into two components, and in part (57c) we eliminate the component $\log_T(\ln(T)/T)$, because it is a small constant for any value of T. The result is that $L_{unfiltered}$

is $O(\log_T(N/M_{filt}))$.

$$\begin{aligned}
O(L_{unfiltered}) &= O\left(\log_T\left(M_{threshold}/M_{filt}\right)\right), && (57a)\\
&= O\left(\log_T\left((N/M_{filt})\cdot(\ln(T)/T)\right)\right), && (57b)\\
&= O\left(\log_T\left(N/M_{filt}\right)+\log_T\left(\ln(T)/T\right)\right), && (57c)\\
&\approx O\left(\log_T\left(N/M_{filt}\right)\right). && (57d)
\end{aligned}$$

As a result, point lookups cost, measured as $R_{unfiltered}$ for the memory range $\frac{M_{threshold}}{T^L} < M_{filters} \leq M_{threshold}$, is $O(\log_T(N/M_{filt}))$ with leveling and $O(T\cdot\log_T(N/M_{filt}))$ with tiering.

K MAPPING TO SYSTEMS IN INDUSTRY

In this appendix, we explain how to map the design space as formalized in this article to the tuning knobs of systems in industry. Thus, we show how we populated Table 4 with the default tuning knobs of mainstream systems in terms of our design space.

RocksDB. RocksDB [40] allows setting the merge policy using the *options.compaction_style* parameter. The default policy is *kCompactionStyleLevel* (leveling) and the size ratio can be set using the parameter *options.target_file_size_multiplier* (10 by default). With this compaction style, every run is partitioned into multiple SSTables. Whenever a Level i reaches capacity, an SSTable from this level is chosen and merged with SSTables with an overlapping range at Level $i+1$.

RocksDB has an additional compaction style called Universal Compaction, which can be turned on by setting the parameter *options.compaction_style* to the option *kCompactionStyleUniversal*. Universal compaction is similar to leveling with two differences. First, every run is a whole file, and so the granularity of merge operations is a run rather than a fragment of a run. Second, merge operations are preemptive across levels: if it is predicted that a merge operation would recurse to Level i as all smaller levels are at capacity, then we merge all the runs from levels 1 to i from the onset. To set the size ratio, RocksDB exposes a parameter *options.compaction_options_universal.size_ratio*, which is defined as the percentage by which a given File j has to be smaller than the next larger file $j+1$ for the merge not to include File $j+1$. This parameter maps to the size ratio T in this article using the formula: *options.compaction_options_universal.size_ratio* $= (T-2)\cdot 100$. The default tuning of this policy sets *options.compaction_options_universal.size_ratio* to 1, which corresponds to a size ratio T of ≈ 2 in our design space.

RocksDB has an additional API that enables monitoring and merging SSTables from the application client code. We used this API in this work to implement tiering on top of RocksDB.

Cassandra. Cassandra [51] offers the option SizeTieredCompactionStrategy, which corresponds to tiering, and it allows configuring the size ratio directly using the parameter *min_threshold*, which directly corresponds to how we define the size ratio T in this article. Cassandra also offers the option LeveledCompactionStrategy, which corresponds to leveling with a fixed, untunable size ratio of 10.

HBase. HBase [8] enables both tiering and leveling with any size ratio. It exposes a ratio parameter: *hbase.store.compaction.ratio*, but it is defined differently than in this article: a file j is included in a merge operation if it is smaller or equal in size to the cumulative sum of all smaller files multiplied by *hbase.store.compaction.ratio*. Setting this ratio above 1 results in a leveled merge policy, whereas setting it below 1 results in tiered merge policy. We map from our design space to this parameter as follows: to enable leveling with size ratio T, set *hbase.store.compaction.ratio* $= T-1$, whereas to enable tiering with a size ratio of T set *hbase.store.compaction.ratio* $= \frac{1}{T-1}$. HBase sets this parameter by default to 1.2, which effectively corresponds to a size ratio T of 2 in our design space.

Accumulo. Accumulo [6] also enables tiering and leveling with any size ratio. Configuring this is analogous to how it is done in HBase via the parameter $table.compaction.major.ratio$. Accumulo sets it by default to 3, which corresponds to leveling with a size ratio of 4 in our design space.

LevelDB. LevelDB [44] hard codes the size ratio to 10 and only enables a leveled merge policy. As it exposes no tuning parameters for merge operations, it is a fixed point in our design space.

L GENERATION OF COST MODELS IN EVALUATION FIGURES

Here, we explain how we generated each of the dotted cost model curves used in Section 5.

Figure 12(A) and (B). The model curves in Figure 12(A) and (B) are generated by multiplying Equations (56) and (7) for lookup cost with RocksDB and Monkey, respectively, by 360 microseconds, the latency that we incurred on average for 1 I/O during this experiment on an Amazon AWS SSD. Note that Equation (7) for lookup cost with Monkey is independent of L, because it is derived by taking L to infinity to enable simplification. Therefore, Equation (7) is flat, giving the latency that Monkey is expected to converge to as the number of levels grows.

Figure 12(C) and (D). The model curves in Figure 12(C) and (D) are generated for the state of the art and for Monkey by multiplying Equations (56) and (7), respectively, by 290ms for SSD and 1.9ms for HDD as we vary the number of bits per entry.

Figure 12(E) and (F). For the state of the art and $Monkey_{zero}$, we used Equation (9) for the cost of a non-zero-result point lookups, plugging in different values for R (the sum of FPRs) and for p_L (the FPR at the largest level). For the state of the art, we plugged in Equation (56) for R and Equation (53) for p_L. For $Monkey_{zero}$, we plugged in Equation (7) for R and Equation (5) for p_L. For $Monkey_{general}$, we used the bottom part of Equation (19) with $L_{unfiltered}$ set to 1.

Figure 13(A). We first obtained the value of v for each point using $\frac{X/100}{1+X/100}$ where X is the percentage of non-zero-result point lookups in the workload. For the state of the art and $Monkey_{zero}$, we used Equation (13) for the weighted cost of a mix of zero-result and non-zero-result point lookups in the workload, plugging in different values for R and for p_L. For the state of the art, we plugged in Equation (56) for R and Equation (53) for p_L. For $Monkey_{zero}$, we plugged in Equation (7) for R and Equation (5) for p_L. For $Monkey_{general}$, we obtained the model using the top part of Equation (19).

REFERENCES

[1] M. Y. Ahmad and B. Kemme. 2015. Compaction management in distributed key-value datastores. *Proc. VLDB Endow.* 8, 8 (2015), 850–861.

[2] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. R. Borkar, Y. Bu, M. J. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y.-S. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. J. Tsotras, R. Vernica, J. Wen, and T. Westmann. 2014. AsterixDB: A scalable, open source BDMS. *Proc. VLDB Endow.* 7, 14 (2014), 1905–1916.

[3] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. 2010. Cheap and large CAMs for high performance data-intensive networked systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'10).* 433–448.

[4] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. 2009. FAWN: A fast array of wimpy nodes. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'09).* 1–14.

[5] M. R. Anderson, D. Antenucci, V. Bittorf, M. Burgess, M. J. Cafarella, A. Kumar, F. Niu, Y. Park, C. Ré, and C. Zhang. 2013. Brainwash: A data system for feature engineering. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR'13).*

[6] Apache. Accumulo. Retrieved from https://accumulo.apache.org/.

[7] Apache. Cassandra. Retrieved from http://cassandra.apache.org.

[8] Apache. HBase. Retrieved from http://hbase.apache.org/.

[9] L. Arge. 2003. The buffer tree: A technique for designing batched external data structures. *Algorithmica* 37, 1 (2003), 1–24.

[10] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. 2013. LinkBench: A database benchmark based on the facebook social graph. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 1185–1196.

[11] M. Athanassoulis, S. Chen, A. Ailamaki, P. B. Gibbons, and R. Stoica. 2011. MaSM: Efficient Online Updates in Data Warehouses. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 865–876.

[12] M. Athanassoulis, S. Chen, A. Ailamaki, P. B. Gibbons, and R. Stoica. 2015. Online updates on data warehouses via judicious use of solid-state storage. *ACM Trans. Database Syst.* 40, 1 (2015).

[13] M. Athanassoulis and S. Idreos. 2016. Design tradeoffs of data access methods. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Tutorial.*

[14] M. Athanassoulis, M. S. Kester, L. M. Maas, R. Stoica, S. Idreos, A. Ailamaki, and M. Callaghan. 2016. Designing access methods: The RUM conjecture. In *Proceedings of the International Conference on Extending Database Technology (EDBT'16).* 461–466.

[15] A. Badam, K. Park, V. S. Pai, and L. L. Peterson. 2009. HashCache: Cache storage for the next billion. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'09).* 123–136.

[16] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka. 2017. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. In *Proceedings of the USENIX Annual Technical Conference (ATC'17).* 363–375.

[17] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. 2007. Cache-Oblivious Streaming B-trees. In *Proceedings of the Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'07).* 81–92.

[18] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. 2012. Don't thrash: How to cache your hash on flash. *Proc. VLDB Endow.* 5, 11 (2012), 1627–1637.

[19] B. H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[20] E. Bortnikov, A. Braginsky, E. Hillel, I. Keidar, and G. Sheffi. 2018. Accordion: Better memory organization for LSM key-value stores. *Proc. VLDB Endow.* 11, 12 (2018), 1863–1875.

[21] G. S. Brodal and R. Fagerberg. 2003. Lower Bounds for External Memory Dictionaries. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'03).* 546–554.

[22] N. G. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. 2013. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the USENIX Annual Technical Conference (ATC'13).* 49–60.

[23] Y. Bu, V. R. Borkar, J. Jia, M. J. Carey, and T. Condie. 2014. Pregelix: Big(ger) graph analytics on a dataflow engine. *Proc. VLDB Endow.* 8, 2 (2014), 161–172.

[24] A. L. Buchsbaum, M. H. Goldwasser, S. Venkatasubramanian, and J. Westbrook. 2000. On external memory graph traversal. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'00).* 859–860.

[25] H. H. W. Chan, Y. Li, P. P. C. Lee, and Y. Xu. 2018. HashKV: Enabling efficient updates in kv storage via hashing. In *Proceedings of the USENIX Annual Technical Conference (ATC'18).* 1007–1019.

[26] B. Chandramouli, G. Prasaad, D. Kossmann, J. J. Levandoski, J. Hunter, and M. Barnett. 2018. FASTER: A concurrent key-value store with in-place updates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 275–290.

[27] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. 2006. Bigtable: A distributed storage system for structured data. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'06).* 205–218.

[28] B. Chazelle and L. J. Guibas. 1986. Fractional cascading: I. A data structuring technique. *Algorithmica* 1, 2 (1986), 133–162.

[29] J. Chen, C. Douglas, M. Mutsuzaki, P. Quaid, R. Ramakrishnan, S. Rao, and R. Sears. 2012. Walnut: A unified cloud object store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 743–754.

[30] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'10).* 143–154.

[31] N. Dayan, M. Athanassoulis, and S. Idreos. 2017. Monkey: Optimal navigable key-value store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 79–94.

[32] N. Dayan, P. Bonnet, and S. Idreos. 2016. GeckoFTL: Scalable flash translation techniques for very large flash devices. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 327–342.

[33] N. Dayan and S. Idreos. 2018. Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 505–520.
[34] B. Debnath, S. Sengupta, and J. Li. 2010. FlashStore: High throughput persistent key-value store. *Proc. VLDB Endow.* 3, 1–2 (2010), 1414–1425.
[35] B. Debnath, S. Sengupta, and J. Li. 2011. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 25–36.
[36] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS Operat. Syst. Review* 41, 6 (2007), 205–220.
[37] J. Dejun, G. Pierre, and C.-H. Chi. 2009. EC2 performance analysis for resource provisioning of service-oriented applications. In *Proceedings of the ICSOC/ServiceWave 2009 WorkshopsService-Oriented Computing*. 197–207.
[38] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum. 2017. Optimizing space amplification in RocksDB. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR'17)*.
[39] Facebook. MyRocks. Retrieved from http://myrocks.io/.
[40] Facebook. RocksDB. Retrieved from https://github.com/facebook/rocksdb.
[41] B. Fan, D. G. Andersen, M. Kaminsky, and M. Mitzenmacher. 2014. Cuckoo filter: Practically better than bloom. In *Proceedings of the ACM International on Conference on emerging Networking Experiments and Technologies (CoNEXT'14)*. 75–88.
[42] B. Fitzpatrick and A. Vorobey. 2011. Memcached: A distributed memory object caching system. White Paper.
[43] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar. 2015. Scaling concurrent log-structured data stores. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys'15)*. 32:1–32:14
[44] Google. LevelDB. Retrieved from https://github.com/google/leveldb/.
[45] S. Idreos, K. Zoumpatianos, M. Athanassoulis, N. Dayan, B. Hentschel, M. S. Kester, D. Guo, L. M. Maas, W. Qin, A. Wasay, and Y. Sun. 2018. The periodic table of data structures. *IEEE Data Eng. Bull.* 41, 3 (2018), 64–75.
[46] S. Idreos, K. Zoumpatianos, B. Hentschel, M. S. Kester, and D. Guo. 2018. The data calculator: Data structure design and cost synthesis from first principles and learned cost models. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 535–550.
[47] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti. 1997. Incremental organization for data recording and warehousing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'97)*. 16–25.
[48] C. Jermaine, A. Datta, and E. Omiecinski. 1999. A novel index supporting high volume data warehouse insertion. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'99)*. 235–246.
[49] C. Jermaine, E. Omiecinski, and W. G. Yee. 2007. The partitioned exponential file for database storage management. *VLDB J.* 16, 4 (2007), 417–437.
[50] B. C. Kuszmaul. 2014. A comparison of fractal trees to log-structured merge (LSM) trees. Tokutek White Paper.
[51] A. Lakshman and P. Malik. 2010. Cassandra—A decentralized structured storage system. *ACM SIGOPS Operat. Syst. Rev.* 44, 2 (2010), 35–40.
[52] Y. Li, B. He, J. Yang, Q. Luo, K. Yi, and R. J. Yang. 2010. Tree indexing on solid state drives. *Proc. VLDB Endow* 3, 1–2 (2010), 1195–1206.
[53] H. Lim, D. G. Andersen, and M. Kaminsky. 2016. Towards accurate and fast evaluation of multi-stage log-structured designs. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'16)*. 149–166.
[54] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. 2011. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'11)*. 1–13.
[55] LinkedIn. 2016. Online reference. Retrieved from http://www.project-voldemort.com.
[56] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. 2016. WiscKey: Separating keys from values in ssd-conscious storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'16)*. 133–148.
[57] P. E. O'Neil, E. Cheng, D. Gawlick and E. J. O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Inf.* 33, 4 (1996), 351–385.
[58] A. Papagiannis, G. Saloustros, P. González-Férez, and A. Bilas. 2016. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *Proceedings of the USENIX Annual Technical Conference (ATC'16)*. 537–550.
[59] M. Pilman, K. Bocksrocker, L. Braun, R. Marroquin, and D. Kossmann. 2017. Fast scans on key-value stores. *Proc. VLDB Endow.* 10, 11 (2017), 1526–1537.
[60] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham. 2017. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'17)*. 497–514.
[61] Redis. Online reference. Retrieved from http://redis.io/.
[62] K. Ren, Q. Zheng, J. Arulraj, and G. Gibson. 2017. SlimDB: A space-efficient key-value storage engine for semi-sorted data. *Proc. VLDB Endow.* 10, 13 (2017), 2037–2048.

[63] R. Sears and R. Ramakrishnan. 2012. bLSM: A general purpose log structured merge tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 217–228.
[64] J. Sheehy and D. Smith. 2010. Bitcask: A log-structured hash table for fast key/value data. Basho White Paper.
[65] P. Shetty, R. P. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok. 2013. Building workload-independent storage with VT-trees. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'13)*. 17–30.
[66] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz. 2012. Theory and practice of bloom filters for distributed systems. *IEEE Commun. Surv. Tutor* 14, 1 (2012), 131–155.
[67] R. Thonangi and J. Yang. 2017. On log-structured merge for solid-state drives. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE'17)*. 683–694.
[68] D. Tsirogiannis, S. Harizopoulos, and M. A. Shah. 2010. Analyzing the energy efficiency of a database server. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 231–242.
[69] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong. 2014. An efficient design and implementation of lsm-tree based key-value store on open-channel SSD. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys'14)*. 16:1–16:14
[70] WiredTiger. Source Code. Retrieved from https://github.com/wiredtiger/wiredtiger.
[71] X. Wu, Y. Xu, Z. Shao, and S. Jiang. 2015. LSM-trie: An LSM-tree-based ultra-large key-value store for small data items. In *Proceedings of the USENIX Annual Technical Conference (ATC'15)*. 71–82.
[72] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo. 2018. SuRF: Practical range query filtering with fast succinct tries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 323–336.

Received December 2017; revised September 2018; accepted September 2018