Monkey: Optimal Navigable Key-Value Store

Niv Dayan Manos Athanassoulis Stratos Idreos

Harvard University

{dayan, manos, stratos}@seas.harvard.edu

ABSTRACT

In this paper, we show that key-value stores backed by an LSM-tree exhibit an intrinsic trade-off between lookup cost, update cost, and main memory footprint, yet all existing designs expose a suboptimal and difficult to tune trade-off among these metrics. We pinpoint the problem to the fact that all modern key-value stores suboptimally co-tune the merge policy, the buffer size, and the Bloom filters' false positive rates in each level.

We present Monkey, an LSM-based key-value store that strikes the optimal balance between the costs of updates and lookups with any given main memory budget. The 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 this sum. 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 LevelDB 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 LSM-tree design space onto a closed-form model that enables co-tuning the merge policy, the buffer size and the filters' false positive rates to trade among lookup cost, update cost and/or 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 various LSM-tree design elements accordingly.

1. INTRODUCTION

LSM-Based Key-Value Stores. Modern key-value stores that maintain application data persistently typically use a Log-Structured-Merge-tree (LSM-tree) [27] as their storage layer. In 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 [29] and (2) good space-efficiency as data is stored compactly [14] (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 disk as a sorted run whenever it fills up, and organizing disk-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 deeper level when the current one becomes full. To speed up point lookups, which are common in practice [6], 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 of a run) 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 [19] and BigTable [12] at Google, RocksDB [15] at Facebook, Cassandra [21], HBase [5] and Accumulo [3] at Apache, Voldemort [25] at LinkedIn, Dynamo [13] at Amazon, WiredTiger [34] at MongoDB, and bLSM [29] and cLSM [18] at Yahoo. Various relational stores today also support this design. For example, MySQL can run on RocksDB using MyRocks [14], which prefixes keys to indicate which table they belong to and maps from SQL commands to RocksDB commands. Moreover, SQLite recently changed its storage engine from a B-tree in SQLite3 to an LSM-tree in SQLite4, meaning that every table is now an LSM-tree [31].

The Problem: Suboptimal Design Tuning. In this paper, we closely investigate the design space of LSM-trees and show that LSM-based key-value stores exhibit an intrinsic trade-off among lookup cost, update cost, and main memory footprint (as discussed for various data structures in [8, 7]). Existing systems strike a sub-

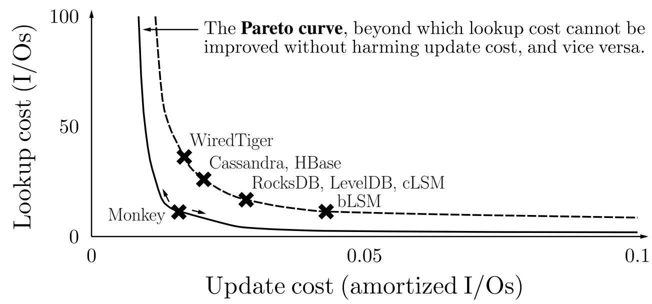


Figure 1: State-of-the-art LSM-tree based key-value stores are not tuned along the Pareto curve. As a result, they are unable to maximize throughput. In contrast, Monkey is tuned along the Paretro Frontier, and it can navigate it to find the best trade-off for a given application to maximize throughput.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD '17, May 14–19, 2017, Chicago, IL, USA.

© 2017 ACM. ISBN 978-1-4503-4197-4/17/05...$15.00

DOI: http://dx.doi.org/10.1145/3035918.3064054

optimal trade-off 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 are not tuned along the optimal Pareto curve beyond which it is impossible to improve lookup cost without harming update cost and vice versa with a given main memory budget. 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-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 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 disk page). As a result, the runs at the largest level take up the majority of the overall memory budget allocated to the Bloom filters 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 (which implies better lookup performance as there are fewer levels to probe and better update performance as there are fewer levels 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 levels and by switching the merge policy between *leveling* and *tiering*, which support one run per level vs. multiple runs per level respectively. Again, however, it is difficult to predict how a given change would impact performance given the complexity of the design space.

The Solution: Monkey. In this paper, we map the design space of LSM-trees. This allows us (1) to holistically tune critical design knobs thereby achieving performance along the Pareto curve (as shown in Figure 1), and (2) to accurately navigate the Pareto curve to find the best trade-off for a given application.

We introduce **M**onkey: **O**ptimal **N**avigable **Key**-Value Store. Monkey reaches the Pareto curve 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 shallower 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 way 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. The number of levels is $O\log(\frac{N \cdot E}{M_{buffer}}))$, where N is the number of entries, E is the size of entries, and M_{buffer} is the size of the buffer. This has two important benefits: (1) lookup cost scales better with the number and size of entries, and (2) lookup cost is 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.

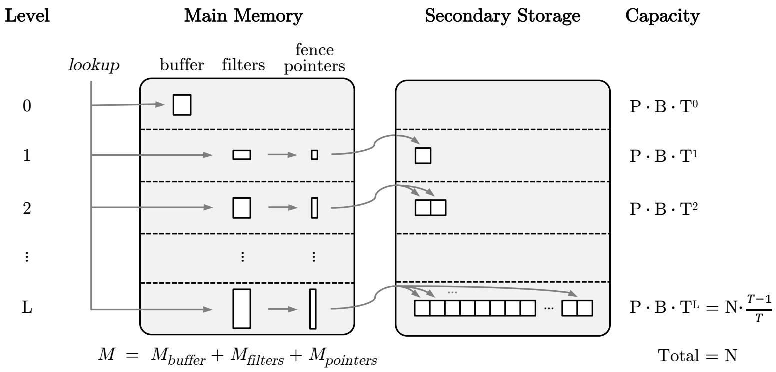
The second key point in Monkey is navigating the Pareto curve to find the optimal balance between lookup cost and update cost under a given main memory budget and application workload (lookup over update ratio). To do so, we map the design space and environmental parameters that affect this balance, and we capture the worst-case cost of lookups and updates as concise closed-form expressions. While all existing major key-value store implementations expose numerous tuning knobs, Monkey is novel in that it allows precise navigation of the design space with predictable results, making it easy to set-up or adapt a key-value store installation.

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 trade-off among lookup cost, update cost, and main memory footprint; yet state-of-the-art key-value stores are not tuned along the optimal trade-off 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 Monkey's Bloom filter tuning (1) reduces 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 tuning.
- We identify the tuning and environment parameters that determine worst-case performance and model worst-case lookup and update costs as closed-form expressions. We show how to use these models to find the holistic tuning that (1) maximizes throughput under a uniformly random workload with any lookup/update ratio, and (2) maximize the lower-bound on throughput for any other workload.
- 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 LevelDB 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.

Online Demo. To provide further understanding of the impact of Monkey we provide an online interactive demo[1].

[1] http://daslab.seas.harvard.edu/monkey/



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_{buffer}	Main memory allocated to the buffer	bits
$M_{filters}$	Main memory allocated to the Bloom filters	bits
$M_{pointers}$	Main memory allocated to the fence pointers	bits

Figure 2: Overview of an LSM-tree and list of terms used throughout the paper.

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 the discussion in this paper both keys and values are stored inside the LSM-tree, although it is also possible to store large values outside of the LSM-tree and point to them using fixed-width pointers attached to keys [26]. For ease of presentation, all figures in this paper 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 paper. 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 which 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 such arrays as *runs*. We denote the number of bits of main memory allocated to the buffer as M_{buffer}, and we define it as $M_{buffer} = P \cdot B \cdot E$, where B is the number of entries that fit into a disk page, P is the amount of main memory in 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 2 MB.

The runs at Level 1 and higher are immutable. Each Level i has a capacity threshold of $B \cdot P \cdot T^i$ entries, where T is a tuning parameter denoting the size ratio between the capacities of adjacent levels. Thus, levels have exponentially increasing capacities by a factor of T. The overall number of levels is given by Equation 1.

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

The size ratio T has a limiting value of T_{lim}, where $T_{lim} = \frac{N \cdot E}{M_{buffer}}$. 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 and tiering [20]. The former optimizes more for lookups and the latter more for updates. 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

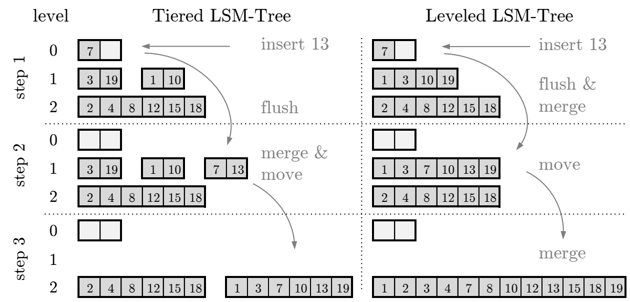


Figure 3: Before and after a recursive merge with tiered and leveled LSM-trees where the size ratio T is set to 3, and $B \cdot P$, the number of entries that fit into the buffer, is set to 2.

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, 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 lowest to highest (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 [27], each run is structured as a compact B-tree. Over the past two decades, however, main memory has become cheaper, so modern designs simply store an array of fence pointers in main memory with min/max information for every disk page of every run [15, 19]. Maintaining a flat array structure is much simpler and leads to good search performance in memory (binary search as each run is sorted). Given that the LSM-tree is on disk, these in-memory binary searches are not in the critical path of performance (I/O is). Thus, a lookup initially searches the fence pointers. If it is a point lookup, it then reads the appropriate disk page with one I/O, or if it is a range lookup it begins a scan from this page. The size

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

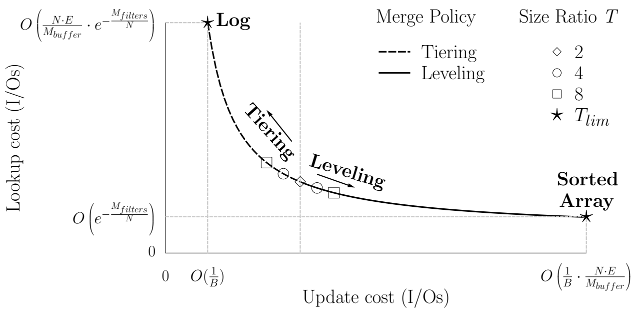


Figure 4: LSM-tree design space: from a log to a sorted array.

of the fence pointers is modest. For example, with 16 KB disk pages and 4 byte pointers, the fence pointers are smaller by ≈ 4 orders of magnitude than the raw data size. Stated formally, we denote the amount of main memory occupied by the fence pointers as $M_{pointers}$, and we assume throughout this work that $M_{pointers}$ is $O(\frac{N}{B})$ thereby guaranteeing that probing a run takes $O(1)$ disk I/O for point lookups.

Bloom Filters. To speed up point queries, every run has a corresponding Bloom filter [10] in main memory. A point lookup probes a run's filter before accessing the run in secondary storage. If the filter returns negative, 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, 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 [32][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 $O(e^{-\frac{M_{filters}}{N}})$ of the runs, where $M_{filters}$ is the overall amount of main memory allocated to the filters. As $M_{filters}$ approaches 0 or infinity, the term $O(e^{-\frac{M_{filters}}{N}})$ approaches 1 or 0 respectively. All implementations that we know of use 10 bits per entry for their Bloom filters by default [3, 4, 5, 15, 19, 29]. The corresponding false positive rate is $\approx 1\%$. With this tuning, an average entry size of 128 bytes (typical in practice [2]) entails the Bloom filters being ≈ 2 orders of magnitude smaller than the raw data size.

Cost Analysis. We now analyze the worst-case I/O cost complexity of updates and lookups for both a tiered and a leveled LSM-tree. First, we explain how we measure these costs. 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 very common in practice [11, 29] (e.g., insert-if-not-exist queries [29]), and (2) they incur the maximum pure I/O overhead (i.e., read I/Os that do not find relevant entries to a lookup). For the rest of the paper, our use of the terms worst-case lookups and updates follows these definitions unless otherwise specified. We later also model the worst-case costs of non-zero-result lookups and range lookups for completeness.

For a tiered LSM-tree, the worst-case lookup cost is given by $O(L \cdot T \cdot e^{-\frac{M_{filters}}{N}})$ 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 $O(e^{-\frac{M_{filters}}{N}})$ of the runs. The worst-case update cost is $O(\frac{L}{B})$ 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 $O(\frac{1}{B})$, 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 $O(L \cdot e^{-\frac{M_{filters}}{N}})$ 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 $O(e^{-\frac{M_{filters}}{N}})$ of the runs on average. The worst-case update cost is $O(\frac{T \cdot L}{B})$ because each update is copied $O(T)$ times per level and through $O(L)$ levels overall.

3. LSM-TREE DESIGN SPACE

In this section, we describe critical trade-offs and tuning contentions in the LSM-tree design space. Our contributions in this paper 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 tuning 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 shown in Figure 4. These costs depend on multiple tuning 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_{buffer} and the Bloom filters, and (4) the allocation of $M_{filters}$ among each of the different Bloom filters. The main observation from Figure 4 is that:

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

Tuning the 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 trade-off continuum.

[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.

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 (left top in the graph of Figure 4) while a leveled LSM-tree degenerates to sorted array (right bottom in the graph of Figure 4). A log is an update-friendly data structure while a sorted array is a read-friendly one. In this way, an 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 paper.

Tuning Main Memory Allocation. The limits of the curve in Figure 4 are determined by the allocation of main memory among the filters $M_{filters}$ and the buffer M_{buffer}. 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 [24]. Moreover, DRAM consumes ≈ 4 times more power per bit than disk during runtime [33]. 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 $M_{filters}$ 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 $M_{filters}$ 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 complexity table in Figure 4, 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 [3, 4, 5, 15, 19, 29]. We refer to these 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 whose novelty is being able to reach and navigate the Pareto curve to find the best possible balance between the costs of lookups and updates for any given main memory budget, workload and storage medium. It maximizes throughput for

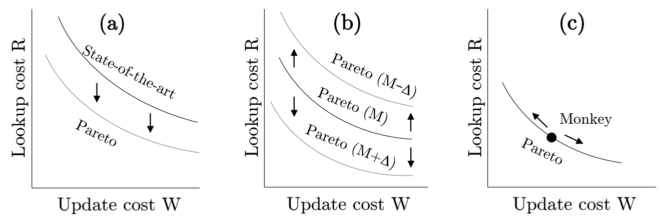


Figure 5: The design of Monkey impacts performance in three ways: (a), it reaches the Pareto curve by optimally tuning the Bloom filters, (b) it predicts how changes in environmental parameters and main memory utilization reposition the Pareto curve, and (c) it finds the point on the Pareto curve that maximizes worst-case throughput for a given application.

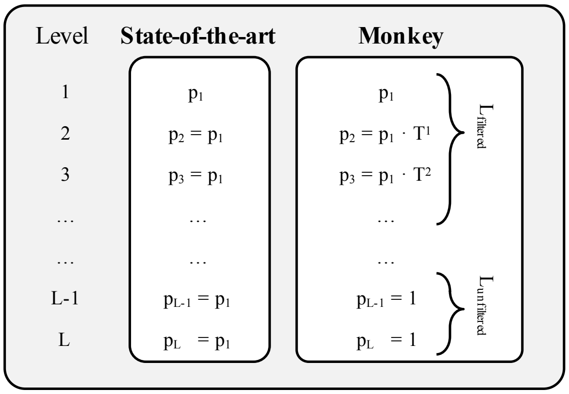
uniformly random workloads, and it maximizes the lower-bound on throughput for all other workloads. Monkey achieves this by (1) resolving the contentions in the LSM-tree design space, and (2) using models to optimally trade among lookup cost, update cost, and main memory footprint. Below we give a high level summary of the main design elements and performance impact of Monkey before we move forward to describe each one of them in depth.

Design Knobs. Monkey transforms the design elements that impact worst-case behavior into tuning 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 $p_1 ... p_L$ assigned to Bloom filters across different levels, and (4) the allocation of main memory M between the buffer M_{buffer} and the filters $M_{filters}$. Monkey can co-tune these knobs to optimize throughput, or in order 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 this sum. 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 trade-off curve vertically down to the Pareto curve.

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 Pareto curve, as shown Figure 5 (b). We derive an analogous model for the state of the art as a baseline and show that Monkey dominates it.

Autotuning. The third design element in Monkey is the ability to holistically self-tune 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 4.4 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 Pareto curve for the balance between lookup



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	
$\hat{M}$	Amount of main memory to divide among filters and buffer	bits

Figure 6: Overview of how Monkey allocates false positive rates $p_1, p_2...p_L$ 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.

and update cost that maximizes the worst-case throughput. This property is visually shown in Figure 5 (c).

Using Monkey. We present Monkey as a key-value store. Our work, though, is applicable to any kind of application where one would use an LSM-tree (see discussion on this in Appendix A).

4.1 Minimizing Lookup Cost

We now continue to discuss how Monkey minimizes the worst-case lookup cost. We focus for now on zero-result lookups since they incur the highest possible pure I/O overhead (i.e., I/Os that do no useful work), and so minimizing their worst-case cost allows achieving robust performance, and we later show experimentally that it also significantly improves performance in other cases (e.g., non-zero-result lookups and workload skew).

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 show that R is equal to the sum of the false positive rates (FPRs) of all Bloom filters. We then show how to tune the Bloom filters' FPRs across different levels to minimize this sum, subject to a constraint on the overall amount of main memory $M_{filters}$. We assume a fixed entry size E throughout this section; in Appendix C we give a iterative optimization algorithm that quickly finds the optimal FPR assignment even when the entry size is variable or changes over time.

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 $p_1...p_L$ 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 T^{th} 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.

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

Modeling Main Memory Footprint. We now model the total main memory footprint for the bloom filters in terms of the FPRs $p_1, p_2...p_L$ 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 Bloom filters is the sum of this expression over all levels, as captured by Equation 4.

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

Minimizing Lookup Cost with Monkey. Using Equations 3 and 4, we can calculate the average worst-case lookup cost R and main memory footprint $M_{filters}$ for any assignment of FPRs across the different levels. To minimize R with respect to $M_{filters}$, we first tackle the converse yet equivalent problem for ease of exposition: finding the optimal assignment of FPRs $p_1...p_L$ across the different levels that minimizes $M_{filters}$ for any user-specified value of R. This amounts to a multivariate constrained optimization problem. In Appendix B, 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[3].

Leveling

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

Tiering

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

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),

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

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-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 shallowest $L_{filtered}$ levels based on 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).

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 tuning 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 B.1, and the result is Equation 7. This equation assumes a fixed entry size (we lift this restriction in Appendix C).

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

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.

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

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 B.1. The optimal value of $L_{unfiltered}$ given by Equation 8 can be plugged into Equation 7 to compute R.

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 writes. Thus, no entry is eliminated before getting merged into the largest level. Using arithmetic series, we 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 this by L since each entry moves through L levels, and we divide by B since each write I/O moves B entries from the original runs to the resulting run. Finally, we account for reading the original runs in order 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 $p_1, p_2 ... p_L$ in Equation 3 to be equal to each other. The complete derivation and resulting closed-form Equation 26 are in Appendix E.

Monkey Dominates Existing Designs. We now have two cost models for lookup cost with Monkey (Equations 7 and 8) and with the state of the art (Equation 26). In Figure 7, we compare Monkey against the state of the art by plotting the zero-result lookup cost as we vary $M_{filters}$ with these equations. Although we keep the terms in the figure general, the curves are drawn for an LSM-tree with 512 TB 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 $M_{filters}$ from 0 to 35 GB.

Monkey dominates the state of the art in terms of lookup cost for any overall amount of main memory allocated to the filters. The

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

Table 1: 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.

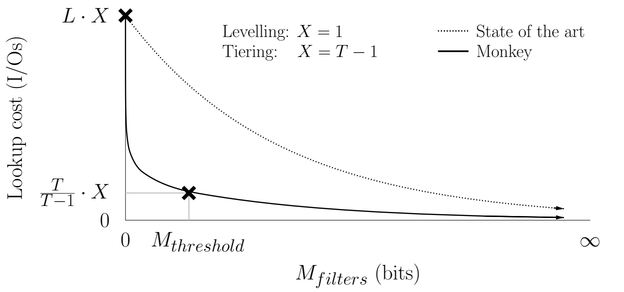


Figure 7: Monkey dominates the state of the art in terms of lookup cost R for all values of $M_{filters}$.

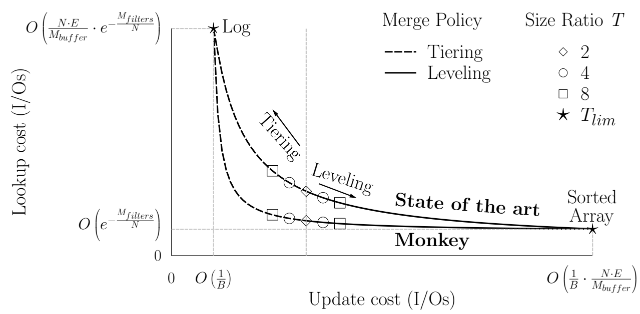


Figure 8: Monkey dominates the state of the art for any merge policy and size ratio.

reason is that it allocates this memory optimally among the Bloom filters to minimize the average number of read I/Os per lookup.

As $M_{filters}$ 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.

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 tuning.

Complexity Analysis. In Table 1, we express the models for worst-case lookup cost and update cost using big O notations. We do so by applying standard simplification rules on Equations 7 and 8 for R in Monkey, on Equation 26 for lookup cost in the state of the art, and on Equation 10 for update cost in both.

The complexity of worst-case lookup cost for Monkey is different depending on whether $M_{filters}$ 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 $M_{filters} > M_{threshold}$, 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 $M_{filters}$ 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 $M_{filters}$, eventually $M_{filters}$ 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 $M_{filters} = M_{threshold}$, 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 $M_{filters} > M_{threshold}$, and otherwise it is $O(R_{unfiltered})$.

The condition $M_{filters} > M_{threshold}$ can be equivalently stated as having the number of bits per element $\frac{M_{filters}}{N} > \frac{1}{\ln(2)^2} \cdot \frac{\ln(T)}{T-1}$. 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 $M_{filters}$ is lower and greater than $M_{threshold}$ (Column b and c respectively in Table 1). We observe that when $M_{filters}$ 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 $M_{filters} \geq M_{threshold}$ (Columns c and e in Table 1). Monkey shaves a factor of $O(L)$ from the complexity of lookup cost for both tiering and leveling (Rows *2* and *3* in Table 1). Note that we express $O(L)$ in Table 1 as $O(\log_T(\frac{N \cdot E}{M_{buffer}}))$ 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 $\frac{M_{filters}}{N}$ 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.

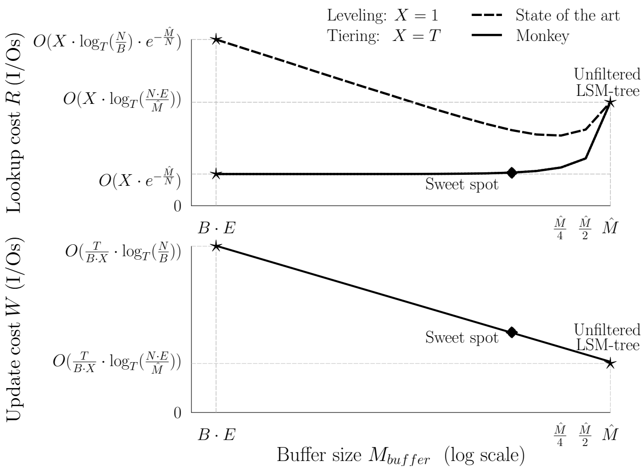


Figure 9: Monkey simplifies tuning by eliminating the dependence of lookup cost on the buffer size.

2. Lookup cost is independent of the entry size, and so it does not increase for data sets with larger entry sizes.

3. Lookup cost is independent of the buffer size. This simplifies tuning 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 $M_{filters} \leq M_{threshold}$. In the state of the art (Column *b*, Rows *2* and *3* in Table 1), lookup cost decreases at a logarithmic rate as M_{buffer} 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 $M_{filters}$ 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 data sets with larger entries.

Monkey dominates the state of the art for all data sets and tunings 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 1. 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 dominates the state of the art by reducing the lookup cost, hence reaching the Pareto curve.

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[4]. 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 $\hat{M} = M_{buffer} + M_{filters}$. On the x-axis (log scale), we increase M_{buffer} at the expense of $M_{filters}$ from one disk page ($B \cdot E$ bits) to $\hat{M}$ bits, in which case the Bloom filters cease to exist (i.e., $M_{filters} = 0$). 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 tuning by making lookup cost independent of the buffer size, provided that $M_{filters} > \frac{M_{threshold}}{T^L}$ 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 the next subsection, we give a strategy for how to allocate main memory among the filters and buffer using this insight.

4.4 Tuning Based on Workload, Hardware and SLAs

We now show how to navigate the LSM-tree design space. Monkey achieves this by being able to precisely trade among lookup cost, update cost, and main memory footprint by controlling four tuning 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 tuning that maximizes throughput. In Table 2 we list new terms.

Modeling Throughput. First, we model the average operation cost θ by weighting the zero-result point lookup cost R, the non-zero result point lookup cost V, the range lookup cost Q, and the update cost W from Equations 7, 9, 10 and 11 by their proportion in the workload represented by the terms r, v, q and w respectively (note that $r + v + q + w = 1$). The result is Equation 12.

$$\theta = r \cdot R + v \cdot V + q \cdot Q + w \cdot W \tag{12}$$

To obtain the worst-case throughput τ, we take the inverse of the average operation cost θ multiplied by Ω, the amount of time to perform a read to secondary storage. The result is Equation 13.

$$\tau = 1/(\theta \cdot \Omega) \tag{13}$$

Tuning the Size Ratio and Merge Policy. The merge policy and size ratio are complementary means of navigating the same trade-off continuum. We devise a divide and conquer algorithm that linearizes this continuum into a single dimension and finds the tuning that maximizes throughput. Figure 10 illustrates an example of the first three iterations of the algorithm to find the optimal tuning marked as "target". In Appendix D we give the full algorithm

The algorithm first sets T to 2 and computes θ for this tuning. It then iteratively divides the space into two and computes θ_1 and θ_2 for the center of each. If either θ_1 or θ_2 is smaller than θ, we narrow down on that partition of the space in the next iteration. Otherwise, we remain at the same point, but we probe points that are twice as

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

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

Table 2: Table of terms used for tuning.

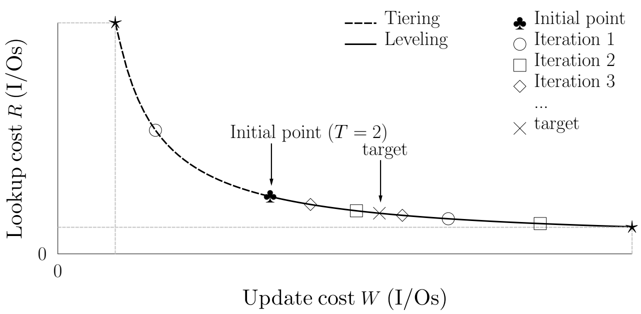


Figure 10: Monkey divides and conquers the design space to find the tuning that maximizes throughput.

close to the current point. This algorithm runs in $O(\log_2(T_{lim}))$ time since there are $O(T_{lim})$ possible tunings and since we halve the size of the problem in each iteration. In practice, it runs in milliseconds.

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. This is analogous to restricting the search space by setting a horizontal limit for lookup cost or a vertical limit for update cost in Figure 10.

Tuning Main Memory Allocation. We introduce a three-step strategy for allocating the available main memory $\hat{M}$ between the buffer and filters. Equation 8 shows that when $M_{filters} < \frac{M_{threshold}}{T^L}$ the filters are too small to yield any benefit, and so step one is to always allocate the first $\min(\hat{M}, \frac{M_{threshold}}{T^L})$ bits of $\hat{M}$ to the buffer. In step two, we allocate 5% to the buffer and 95% to the filters of the remaining main memory. This method approximates the sweet spot in Figure 9. We continue following step two until the I/O overhead due to false positives becomes negligible (e.g., compared to the CPU and RAM overhead of probing a Bloom filter, which we approximate to be in the order of a microsecond). In a disk-based system, seek latency is ≈ 10 milliseconds. This means that when the sum of false positives R in Equation 7 reaches the order of 10^{-4}, the average contribution of I/O to lookup latency becomes ≈ 1 μs, which is insignificant compared to other system overheads. The analogous threshold value of R for a flash-based system is 10^{-2} since a read I/O to a flash device is in the order of tens to hundreds of microseconds. If there is more main memory in our budget beyond step two, we allocate it to the buffer to further reduce update cost.

5. EXPERIMENTAL ANALYSIS

We now proceed to experimentally evaluate Monkey against state-of-the-art designs. We first show that Monkey's method of tuning 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.

Experimental Setup. For experimentation we use a machine with a 500GB 7200RPM disk, 32GB DDR4 main memory, and 4 2.7GHz cores with 8MB L3 cache. The machine is running 64-bit Ubuntu 16.04 LTS, and we run the experiments on an ext4 partition with journaling turned off.

Implementation. We implemented Monkey on top of LevelDB, 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 LevelDB comprises a fixed point in the design space. It only supports leveling as a merge policy, it has a fixed, hard-coded size ratio, and it assigns the same FPR to filters across all levels. We implemented Monkey by adding support for tiering, different size ratios, and optimal FPRs for filters across different levels (by embedding our algorithm from Appendix C). To enable an apples to apples comparison, in all experiments Monkey only differs from LevelDB in how it allocates Bloom filters, and we use the same default values for all other parameters. The default configuration is: size ratio is 2 (the size ratio at which leveling and tiering behave identically); buffer size is 1 MB; the overall amount of main memory allocated to all Bloom filters is $\frac{M_{filters}}{N} = 5$ bits per element (though Monkey allocates these bits differently across different levels). We vary these parameters one at a time to compare across a wide range of the design space. Similarly to recent versions of RocksDB we implemented direct I/O to be able to fully control the memory budget. In addition, all reported experiments in the main part of the paper are set with the block cache of LevelDB and Monkey disabled. 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 F, we show that Monkey maintains its advantages when there is enough memory to devote to a block cache.

Default Set-up. Unless otherwise mentioned the default experimental set-up is as follows. The database is initially empty. We insert 1GB of key-value entries where each entry is 1KB in size. The entries are uniformly randomly distributed across the key space and inserted at a random order. After the initialization phase, we issue 16K zero-result point lookups which are uniformly randomly distributed across the key space. (We repeat several variations of this set-up with different data sizes and query workloads).

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

Monkey Scales Better with Data Volume. In the first experiment, we show that Monkey improves lookup latency by an increasing margin as the data volume grows. We set up this experiment by repeating the default experimental setup multiple times, each time using more data entries.

Results are shown in Figure 11 (A). Lookup latency for the LevelDB 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 LevelDB 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 lower 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 LevelDB by up to 80%, and its margin of improvement increases as the number of entries grows. The horizontal dotted line in Figure 11 (A) shows how lookup latency corresponds to disk I/Os. The average number of I/Os per lookup is lower than one because the

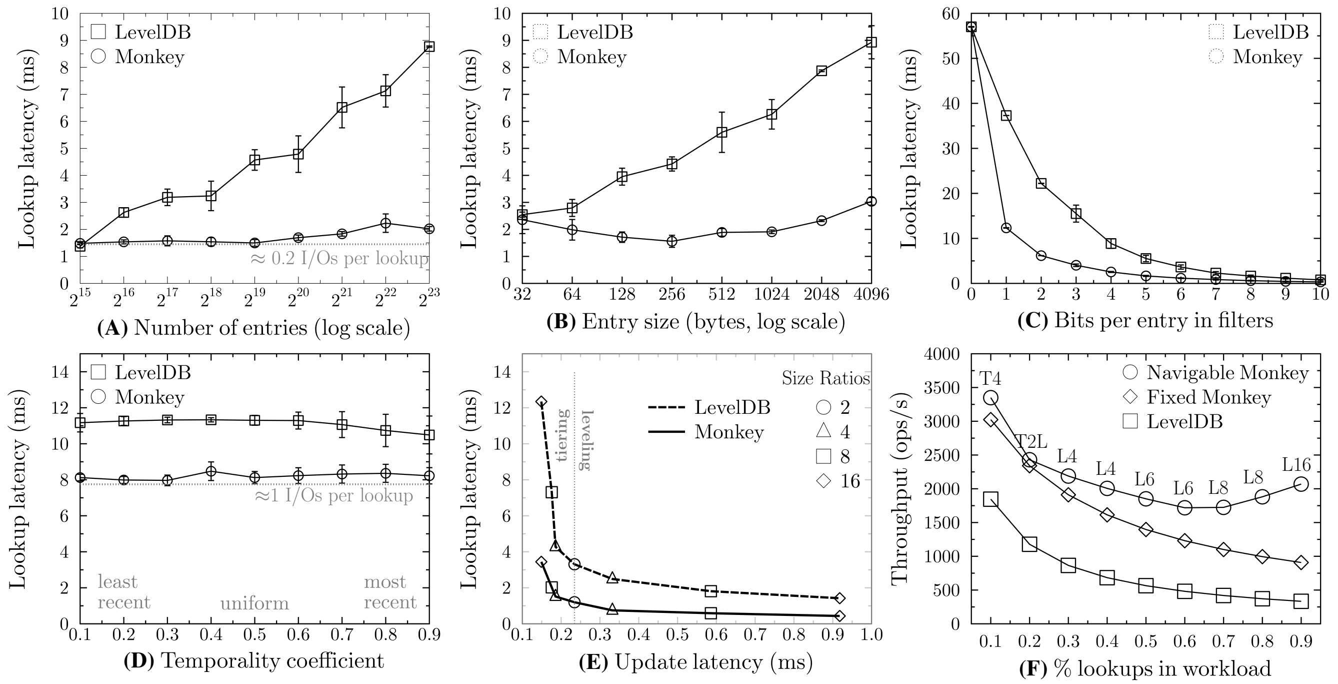


Figure 11: Monkey improves lookup cost under any (A) number of entries, (B) entry size, (C) amount of memory, (D) lookup locality, and (F) merge policy and size ratio. It navigates the design space to find the design that maximizes throughput (F).

lookups target non-existing keys, and so they do not issue I/Os most of the time due to the filters.

Figure 11 (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 experiment, we show that Monkey can match the performance of LevelDB 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. The results are shown in Figure 11 (C). When the number of bits per entry is set to 0, both Monkey and LevelDB 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 I/Os drops to nearly 0, at which point the curves nearly converge. Overall, with the exception of the extreme case of no memory budget for the filters, Monkey can match the performance of LevelDB with a smaller memory footprint (up to $\approx 60\%$ smaller in this experiment, though the asymptotic improvement of Monkey implies that the margin of improvement increases as a function of the number of entries).

Monkey Improves Lookup Cost for Different Workloads. 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. To control temporal locality, we define a coefficient c ranging from 0 to 1 whereby c percent of the most recently updated entries receive $(1-c)$ percent of the lookups. When c is set to 0.5, the workload is uniformly randomly distributed. When it is above 0.5, recently updated entries receive most of the lookups, and when it is below 0.5 the least recently updated entries receive most of the lookups. We set up this experiment by repeating the experimental setup multiple times, with the difference that during the query phase we issue lookups to existing keys based on the temporality coefficient.

The results are shown in Figure 11 (D). For both Monkey and LevelDB, each lookup involves at least one I/O for the target key, and so lookup latency comprises at least one disk seek. We mark this source of latency using the dotted gray line, which represents the approximate time to perform one seek on our hard disk. Any contribution to latency above this line arises due to false positives.

A key observation in Figure 11 (D) is that lookup latency for both Monkey and LevelDB is largely insensitive to temporal locality. The reason is that in an LSM-tree the most recently updated entries are at the shallower levels, which have exponentially lower capacities than the largest level. Hence, even a lookup for the most recently updated 10% of the entries has to probe most levels on average. The curve for LevelDB slightly decreases as temporal locality increases because a lookup traverses fewer levels on average before finding the target key and terminating, and so fewer false positives take place. For Monkey, lookup latency is even less sensitive to temporal locality. The reason is that all but the last level have significantly lower FPRs than for LevelDB. Even though a lookup traverses fewer levels on average before terminating as temporal locality increases, the low FPRs at these lower levels mean that false positives are rare, and so they contribute very modestly to latency in all cases. In this way, Monkey improves lookup latency by up to 30% in this experiment for non-zero-result lookups across a wide range of temporal locality in the query workload.

Monkey Reaches the Pareto Curve. In this experiment, we show that Monkey reaches the Pareto frontier and is therefore able to navigate a better trade-off 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. We measure the average latencies of lookups and updates and plot them against each other for Monkey and LevelDB. The result is shown in Figure 11 (E). The key observation is that for any configuration, Monkey achieves a significantly lower lookup cost than LevelDB due the tuning of its Bloom filters, as predicted by our analysis in Section 4.3. Hence, Monkey shifts the trade-off curve downwards to the Pareto frontier. As a result, Monkey improves lookup cost by up to 60%, 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 experiment 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 this 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%. We compare two instances of Monkey to LevelDB. The first, named Fixed Monkey, is the default configuration of Monkey that we have experimented with so far. The second, named Navigable Monkey, is the full version that navigates the design space to find the merge policy to size ratio that gives the balance between lookup cost and update cost (and the corresponding optimal Bloom filters' allocation) that maximizes throughput. LevelDB and Fixed Monkey have a fixed size ratio of 2 (i.e., the point in the space at which leveling and tiering behave identically).

Results are shown in Figure 11 (F). For each data point in Navigable Monkey, we show the configuration that it adopts (T stands for tiering, L for leveling, and the number is the size ratio). A key observation is that the curve for Navigable Monkey is bell-shaped. The reason is that the more extreme the workload tends towards one operation type, the more possible it is to achieve a higher throughput as a single configuration handles most operations well. Overall, Fixed Monkey significantly improves upon LevelDB, and Navigable Monkey significantly improves on Fixed Monkey. Navigable Monkey more than doubles throughput relative to LevelDB by being able to reach the Pareto curve and to navigate the design space.

6. RELATED WORK

LSM-Based Key-Value Stores. As state-of-the-art LSM-tree based key-value stores [4, 12, 15, 18, 19, 21, 29, 34], 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 [4, 12, 15, 18, 19, 21, 29, 34]. 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 positives 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 shallower levels have exponentially lower false positive rates). This reduces the asymptotic complexity of worst-case lookup cost, and in practice it reduces lookup latency by at least $50\% - 80\%$ without losing anything.

Holistic Tuning. Existing LSM-tree based key-value stores do not allow to easily trade among lookup cost, update cost and main memory footprint. For example, LevelDB [19] hard-codes the size ratio between levels to 10 with leveled compaction, RocksDB [15] and LSM-trie [35] only enable leveled or tiered compaction respectively, and they use a fixed number of bits per element for all Bloom filters. WiredTiger [34] supports dynamic values for size ratio having as starting point 15, however, it also uses a fixed number of bits per element for all Bloom filters (setting as default 16 bits per element). In addition, the balance between lookup cost, update cost and main memory footprint depends on a combination of interdependent tuning and environmental parameters that must be precisely co-tuned. In this way, tuning in existing key-value stores becomes 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 easy. It exposes critical tuning knobs that influence worst-case lookup cost, update cost, and main memory footprint. Moreover, it uses novel worst-case closed-form models that enable optimizing throughput and answering what-if questions regarding how changes in environmental parameters affect performance.

Recent complementary work [23] 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 tuning over the entire design space.

De-amortizing Merge Operations. To maintain stable performance, all LSM-tree based key-value stores spread the work done by merge operations over time. Some stores pace merge operations directly with respect to application updates [9, 22, 29]. 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 [4, 12, 15, 18, 19, 21, 29, 34]. Other recent work [1] proposes merging runs on dedicated servers. 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. To reduce the volume of data that is copied during merge operations, WiscKey [26] decouples values from keys and stores values on a separate log. This technique is compatible with Monkey's core design, but it would require adapting the cost models to account for (1) only merging keys, and (2) having to access the log during lookups.

VT-tree [30] proposes to avoid including portions of runs that do not overlap during merge operations. This technique is used in major key-value stores, where an SSTable is simply moved to the next level if there are no SSTables in the next level with overlapping key-ranges [4, 12, 15, 19, 21, 29]. Our implementation on top of LevelDB takes advantage of this technique.

In-Memory Stores. Key-value stores such as Redis [28] and Memcached [17] 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 paper, and so this work is orthogonal to in-memory efforts. However, given that similar trade-offs exist in a pure in-memory environment in order 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 trade-off among lookup cost, update cost, and main memory footprint. However, all existing designs strike a suboptimal and difficult to tune balance among these metrics. We present Monkey, an LSM-based key-value store that reaches the Pareto performance curve by allocating the Bloom filters so as to minimize the worst-case lookup cost. Monkey uses a closed-form model to navigate the design space to find the holistic tuning that maximizes throughput under a given main memory budget, application workload, and storage medium.

Acknowledgments. We thank the anonymous reviewers for their valuable feedback. This work is supported by the National Science Foundation under grant IIS-1452595.

8. REFERENCES

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

[2] M. R. Anderson, D. Antenucci, V. Bittorf, M. Burgess, M. J. Cafarella, A. Kumar, F. Niu, Y. Park, C. Ré, and C. Zhang. Brainwash: A Data System for Feature Engineering. In *CIDR*, 2013.

[3] Apache. Accumulo. *https://accumulo.apache.org/*.

[4] Apache. Cassandra. *http://cassandra.apache.org*.

[5] Apache. HBase. *http://hbase.apache.org/*.

[6] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. LinkBench: a Database Benchmark Based on the Facebook Social Graph. In *SIGMOD*, 2013.

[7] M. Athanassoulis and S. Idreos. Design Tradeoffs of Data Access Methods. In *SIGMOD*, 2016.

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

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

[10] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *CACM*, 13(7):422–426, 1970.

[11] N. 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. TAO: Facebook's Distributed Data Store for the Social Graph. In *ATC*, 2013.

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

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

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

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

[16] Facebook. MyRocks. *http://myrocks.io/*.

[17] B. Fitzpatrick and A. Vorobey. Memcached: a distributed memory object caching system, 2011.

[18] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar. Scaling Concurrent Log-Structured Data Stores. In *EuroSys*, 2015.

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

[20] B. C. Kuszmaul. A Comparison of Fractal Trees to Log-Structured Merge (LSM) Trees. *Tokutek White Paper*, 2014.

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

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

[23] H. Lim, D. G. Andersen, and M. Kaminsky. Towards Accurate and Fast Evaluation of Multi-Stage Log-structured Designs. In *FAST*, 2016.

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

[25] LinkedIn. Online reference. *http://www.project-voldemort.com*.

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

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

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

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

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

[31] SQLite4. Online reference. *https://sqlite.org/src4/*.

[32] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz. Theory and Practice of Bloom Filters for Distributed Systems. *IEEE Communications Serveys & Tutorials*, 14(1):131–155, 2012.

[33] D. Tsirogiannis, S. Harizopoulos, and M. A. Shah. Analyzing the energy efficiency of a database server. In *SIGMOD*, 2010.

[34] WiredTiger. WiredTiger. *https://github.com/wiredtiger/wiredtiger*.

[35] X. Wu, Y. Xu, Z. Shao, and S. Jiang. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. In *ATC*, 2015.

APPENDIX

A. HOW TO USE MONKEY

In this paper, we present Monkey as a key-value store. Our work, though, is applicable to any kind of application where one would use an LSM-tree. We summarize below the kinds of scenarios where our study can prove useful.

Any LSM-Based Key-Value Store. Any LSM-based key-value store can adopt the Monkey way of assigning false positive rates across its bloom-filters so that it can reach the optimal Pareto curve.

Tuning of LSM-Trees. The closed form formulas provided in this paper can be used to statically tune key-value stores for a given scenario (read/write ratio). As long as the tuning knobs are exposed, any key-value store can use these formulas to be tuned along the Pareto curve.

Adaptive Key-Value Stores. A future class of key-value stores may adaptively switch from one tuning setting to another one. The formulas provided in this paper can be the seed for taking these decisions (along with the transformation costs).

Any LSM-Tree Application. LSM-trees are used in many applications. Any scenario can benefit from our analysis to improve throughput if the performance of the LSM-tree for point lookups is in the critical path.

B. OPTIMAL FALSE POSITIVE RATES

In this appendix, we derive the optimal false positive rates (FPRs) $p_1 ... p_L$ by optimizing lookup cost R in Equation 3 with respect to the main memory footprint $M_{filters}$ 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 straight-forward 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:

$$g(p_L ... p_1, R) = p_L + p_{L-1} + ... + p_1 - R$$

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

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

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

Next, we differentiate the Lagrangian with respect to each of the FPRs $p_1 ... p_L$, and we set every partial derivative to 0. Thus, we arrive to 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.

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

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.

$$\begin{aligned} R &= \frac{p_L}{T^0} + \frac{p_L}{T^1} + ... + \frac{p_L}{T^{L-1}} \\ &= p_L \cdot \frac{(\frac{1}{T})^L - 1}{\frac{1}{T} - 1} \end{aligned} \tag{14}$$

We now rearrange in and express generically in terms of the FPR for Level i. The result appears in Equations 15 and 16 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 $M_{filters}$ is used to achieve the user-specified value of R.

Leveling

$$p_i = \frac{R}{T^{L-i}} \cdot \frac{T^{L-1}}{T^L - 1} \cdot (T-1)$$
$$\text{for } 0 < R \leq \frac{T^L - 1}{T^{L-1}} \cdot \frac{1}{T-1}$$
$$\text{and } 1 \leq i \leq L \tag{15}$$

Tiering

$$p_i = \frac{R}{T^{L-i}} \cdot \frac{T^{L-1}}{T^L - 1}$$
$$\text{for } 0 < R \leq \frac{T^L - 1}{T^{L-1}}$$
$$\text{and } 1 \leq i \leq L \tag{16}$$

The key difference between Equations 15 and 16 is that the optimal false positive rate prescribed to any Level i is $(T-1)$ 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 15 and 16 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 15 and 16 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 deepest to shallowest. Hence, we can denote $L_{unfiltered}$ as the number of levels from level L and down whose FPRs converged to 1, whereas $L_{filtered}$ is the number of levels from Level 1 and up 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 15 and 16 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 17 and 18 respectively. We use L_u to denote $L_{unfiltered}$ in these equations for brevity.

Leveling

$$p_i = \begin{cases} 1, & \text{if } i > L - L_u \\ \frac{R - L_u}{T^{(L-L_u)-i}} \cdot \frac{T^{(L-L_u)-1} \cdot (T-1)}{T^{(L-L_u)} - 1}, & \text{else} \end{cases}$$
$$\text{for } 0 < R \leq L$$
$$\text{and } 1 \leq i \leq L$$
$$\text{and } L_u = \max(0, \lfloor R-1 \rfloor) \tag{17}$$

Tiering

$$p_i = \begin{cases} 1, & \text{if } i > L - L_u \\ \frac{R - L_u \cdot (T-1)}{T^{(L-L_u)-i}} \cdot \frac{T^{(L-L_u)-1}}{T^{(L-L_u)} - 1}, & \text{else} \end{cases}$$
$$\text{for } 0 < R \leq L \cdot (T-1)$$
$$\text{and } 1 \leq i \leq L$$
$$\text{and } L_u = \max(0, \lfloor \tfrac{R-1}{T-1} \rfloor) \tag{18}$$

Simplification. As the number of levels L grows, Equations 15 and 16 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 ≈ 5 or above. We can use this insight to simplify Equations 17 and 18 into Equations 5 and 6, which appear in Section 4.1. For practical analysis and implementations we recommend using Equations 17 and 18.

B.1 Modeling Memory Footprint and Lookup Cost

We now show how to derive a closed-form model for main memory utilization for the filters $M_{filters}$ 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.

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

We then apply logarithm operations to get the following:

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

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 arithmetico-geometric series to infinity on the exponents of the denominator. We get the following after some further simplification.

$$M_{filters} = \begin{cases} \frac{N}{\ln(2)^2} \cdot \ln \left(\frac{T^{\frac{T}{T-1}}}{R \cdot (T-1)} \right) & \text{with leveling} \\ \frac{N}{\ln(2)^2} \cdot \ln \left(\frac{T^{\frac{T}{T-1}}}{R} \right) & \text{with tiering} \end{cases} \tag{19}$$

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 shallower $L_{filtered}$ levels. To do so, we adjust Equation 19 by applying it on a smaller version of the problem with $N/T^{L_{unfiltered}}$ entries (i.e., the number of entries in the shallower $L_{filtered}$ levels), and we discount the I/Os to the unfiltered levels by subtracting the number of runs in those levels from R.

$$M_{filters} = \begin{cases} \dfrac{N}{\ln(2)^2 \cdot T^{L_{unfiltered}}} \cdot \ln\left(\dfrac{T^{\frac{T}{T-1}}}{(R - L_{unfiltered}) \cdot (T-1)}\right) & \text{for leveling} \\ \dfrac{N}{\ln(2)^2 \cdot T^{L_{unfiltered}}} \cdot \ln\left(\dfrac{T^{\frac{T}{T-1}}}{R - L_{unfiltered} \cdot (T-1)}\right) & \text{for tiering} \end{cases} \tag{20}$$

Lookup Cost. We now rearrange Equation 19 to be in terms of lookup cost R.

$$R_{filtered} = \begin{cases} \dfrac{T^{\frac{T}{T-1}}}{T-1} \cdot e^{-\frac{M_{filters}}{N} \cdot \ln(2)^2} & \text{with leveling} \\ T^{\frac{T}{T-1}} \cdot e^{-\frac{M_{filters}}{N} \cdot \ln(2)^2} & \text{with tiering} \end{cases} \tag{21}$$

The above equation is still not adapted to the case where $M_{filters}$ is so low that some of the filters at deeper levels cease to exist. To adapt it to this case, we first of all 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 which are $\frac{T}{T-1}$ and T under tiering and leveling respectively into Equation 19. 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 22.

$$L_{unfiltered} = \begin{cases} 0, & M_{threshold} \le M_{filters} \\ \left\lceil \log_T\left(\frac{M_{threshold}}{M_{filters}}\right)\right\rceil, & \frac{M_{threshold}}{T^L} \le M_{filters} \le M_{threshold} \\ L, & 0 \le M_{filters} \le \frac{M_{threshold}}{T^L} \end{cases} \tag{22}$$

Now, in the deepest levels with no filters, we need to probe every runs, which is $L_{unfiltered}$ with leveling and $(T-1) \cdot L_{unfiltered}$ 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 21 on a smaller version of the problem with $N/T^{L_{unfiltered}}$ levels. This becomes Equation 7 in Section 4.2.

C. AUTO-TUNING 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, 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 1 (and auxiliary Algorithms 2 and 3). Algorithm 1 takes as parameters (1) the overall amount of main memory $M_{filters}$ 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.

```
AutotuneFilters (M_filters, runs)
    Δ = M_filters;
    runs[0] = M_filters;
    R = runs.length − 1 + eval(runs[0].bits, runs[0].entries);
    while Δ ≥ 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], Δ, min);
                R_new = TrySwitch(runs[j], runs[i], Δ, min);
            end
        end
        if R_new == R then
            Δ = Δ/2;
        R = R_new;
    end
    return R;
```

Algorithm 1: Allocate $M_{filters}$ to minimize the sum of FPRs.

```
TrySwitch (run1, run2, Δ, R)
    R_new = R − eval(run1.bits, run1.entries) − eval(run2.bits, run2.entries) +
        eval(run1.bits + Δ, run1.entries) + eval(run2.bits − Δ, run2.entries);
    if R_new < R and run2.bits − Δ > 0 then
        R = R_new;
        run1.bits += Δ;
        run2.bits −= Δ;
    return R;
```

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

```
eval (bits, entries)
    return e^(−(bits/entries)·ln(2)^2);
```

Algorithm 3: Returns the false positive rate of a Bloom filter.

D. AUTO-TUNING THE SIZE RATIO AND MERGE POLICY

In this Appendix, we give the complete auto-tuning algorithm. This algorithm linearizes the tuning space into one dimension and performs a binary search over it. It consists of Algorithm 5, which computes the normalized operation cost θ. The body of the algorithm is in Algorithm 4, where we perform the binary search over the tuning space.

E. 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 false positive rates to filters under both leveling and tiering. We do so by setting all false positive rates $p_1, p_2 ... p_L$ in Equation 3 to be equal to each other, simplifying, and rearranging. The results are Equations 23 and 24.

Leveling

$$p_i = \frac{R_{art}}{L} \quad \text{for } 0 < R_{art} \le L \text{ and } 1 \le i \le L \tag{23}$$

Tiering

$$p_i = \frac{R_{art}}{L \cdot (T-1)} \quad \text{for } 0 < R_{art} \le L \cdot (T-1) \text{ and } 1 \le i \le L \tag{24}$$

Second, we plug the false positive rates in Equations 23 and 24 into Equation 4 and simplify by applying logarithm operations and sums of geometric series. The result is shown in Equation 25.

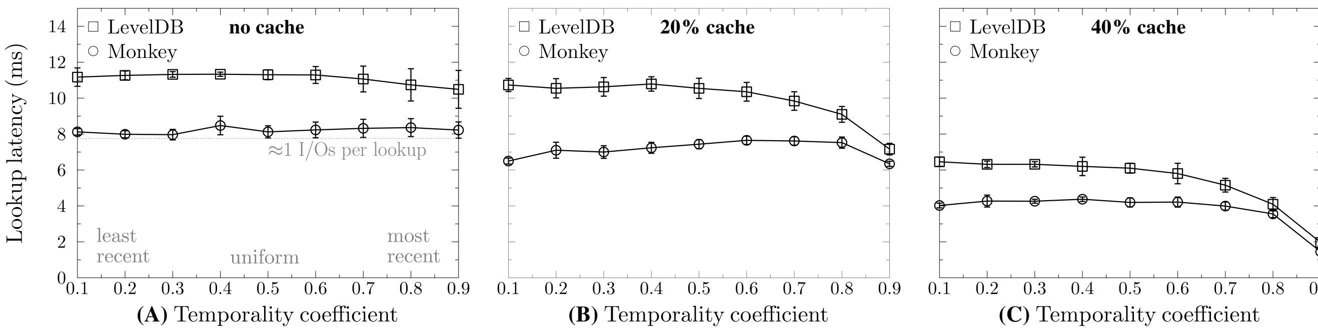


Figure 12: Monkey maintains its advantages with a block cache, and it can also utilize a cache to query recently touched keys.

Autotune *()*
- $i = 0$;
- $\theta = compute(i)$;
- $\Delta = \frac{1}{2} \cdot \frac{N \cdot E}{M_{buffer}}$;
- **while** $\Delta \geq 1$ **do**
  - $\theta_1 = compute(i + \Delta)$;
  - $\theta_2 = compute(i - \Delta)$;
  - **if** $\theta_1 < \theta$ ***and*** $\theta_1 < \theta_2$ **then**
    - $\theta = \theta_1$;
    - $i = i + \Delta$;
  - **else if** $\theta_2 < \theta$ **then**
    - $\theta = \theta_2$;
    - $i = i - \Delta$;
  - $\Delta = \Delta/2$;
- **end**
- $T_{best} = \ | \, i \, | + 2$;
- $merge_{best} =$ **if** $i > 0$ **then** tiering **else** leveling;

Algorithm 4: Finding the best size ratio and merge policy.

compute *(i)*
- $T = | \, i \, | + 2$;
- $merge =$ **if** $i > 0$ **then** tiering **else** leveling;
- $\theta = \theta(T, merge)$; // Equation 12
- **return** θ ;

Algorithm 5: Linearizing the tuning space.

$$M_{filters} = \frac{N}{\ln(2)^2} \cdot (1 - T^{-L}) \cdot \ln\left(\frac{L}{R_{art}}\right) \tag{25}$$

As L grows, the Equation converges into the following:

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

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

$$R_{art} = \begin{cases} L \cdot e^{-\frac{M_{filters}}{N} \cdot \ln(2)^2}, & \text{with leveling} \\ L \cdot (T-1) \cdot e^{-\frac{M_{filters}}{N} \cdot \ln(2)^2}, & \text{with tiering} \end{cases} \tag{26}$$

$$\text{for } 0 < M_{filters} \leq N \cdot E$$

The corresponding big O notations of R_{art} are $O(L \cdot e^{-\frac{M_{filters}}{N}})$ for leveling and $O(L \cdot T \cdot e^{-\frac{M_{filters}}{N}})$ for tiering.

F. CACHING

In all experiments in the main part of the paper both LevelDB and Monkey do not use a cache. We do that by explicitly disabling the block cache from within LevelDB. We use this setup in order 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 LevelDB, which caches recently accessed data blocks (16 KB 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 paper. Once data is 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 11 (D) in Section 5) whereby c percent of the most recently updated entries receive $(1-c)$ percent 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 12. Figure 12 (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 LevelDB across the whole workload range. We observe the same overall behavior with Monkey outperforming LevelDB across the two other graphs in Figure 12 (B) and (C) when the cache is enabled. The difference here is that when we query very recently accessed keys, these keys will likely exist in the cache and so they will not be processed through the LSM-tree. As a result, as we query more recently accessed items, both Monkey and LevelDB nearly converge. Monkey can utilize the cache for frequently accessed items in the same way as LevelDB. 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 12 (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 16 KB data block for every recently accessed 1 KB key-value entry.[5]

Overall, Monkey can utilize 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 largely different sources of latency: false positives and true positives respectively. In this work we focused on tuning the Bloom filters to reduce and control the rate of I/Os due to false positives. Further investigation could be beneficial to also tune the cache to reduce and control the rate of I/Os due to true positives.

[5]Further research to design a cache that caches key-value pairs instead of blocks could be more effective although there might be side-effects with cache maintenance.