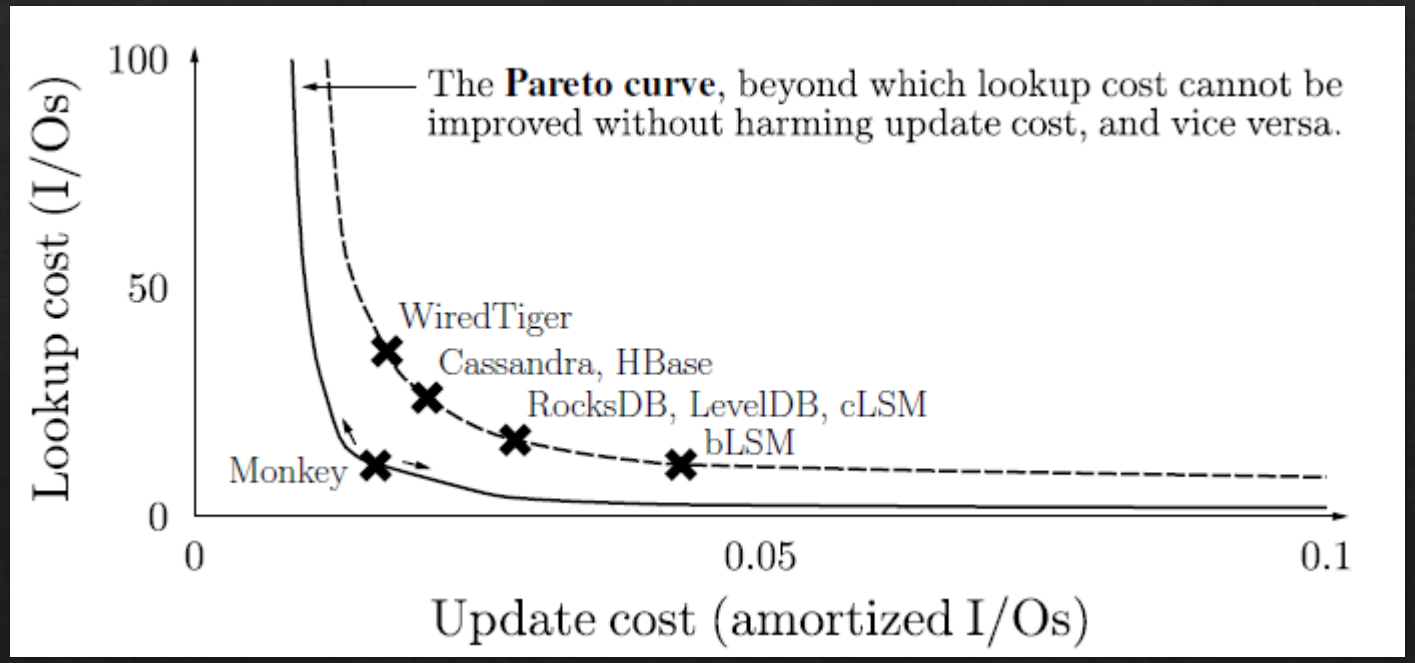
Monkey: Optimal Navigable Key-Value Store

By
Efstathios Karatsiolis

Monkey: Optimal Navigable Key-Value Store

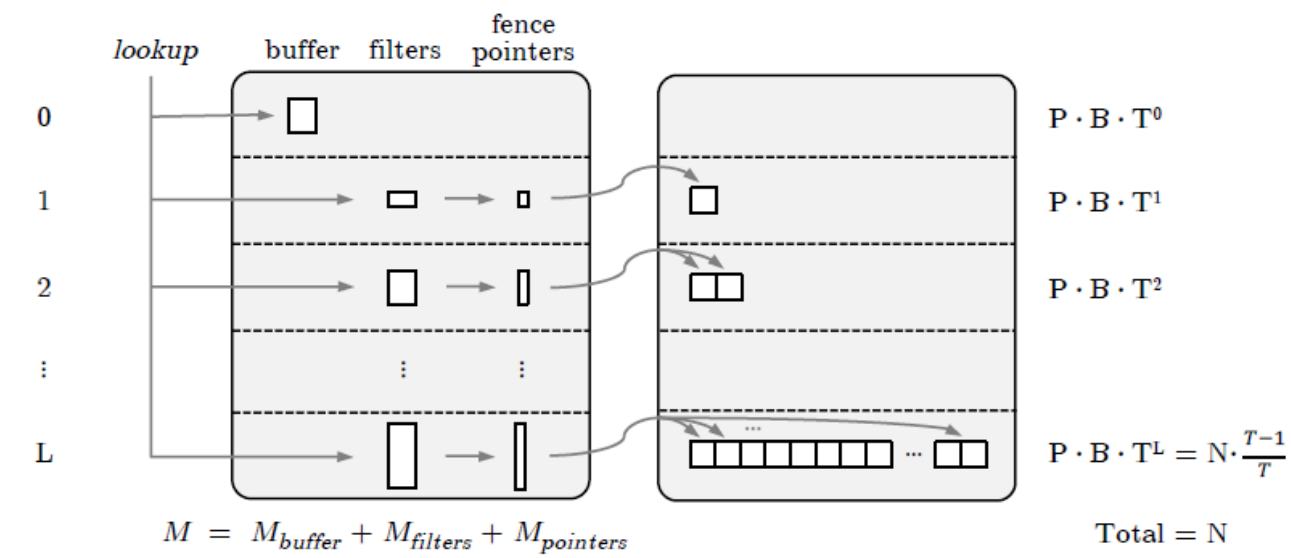
- Monkey maps the design of LSM trees and is able to tune critical design knobs
- Monkey is able to navigate the curve and find the best trade-off for a given application



LSM Trees Background
Key Words

- **Runs**:
  - A block of data as stored in the LSM Tree
- **BloomFilter**:
  - A probabilistic structure that can answer maybe/no. Has a False Positive Rate (FPR), which describes how often the bloomfilter was wrong in its “Yes” prediction.
- **FencePointer**:
  - Metadata presenting the minimum and the maximum value of a page
- **Tier/Level**:
  - A level of the LSM tree. Levels start at 0
- **Lookup**:
  - Either a single value lookup or a ranged query
- **Update**:
  - Either an insert or update or delete

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



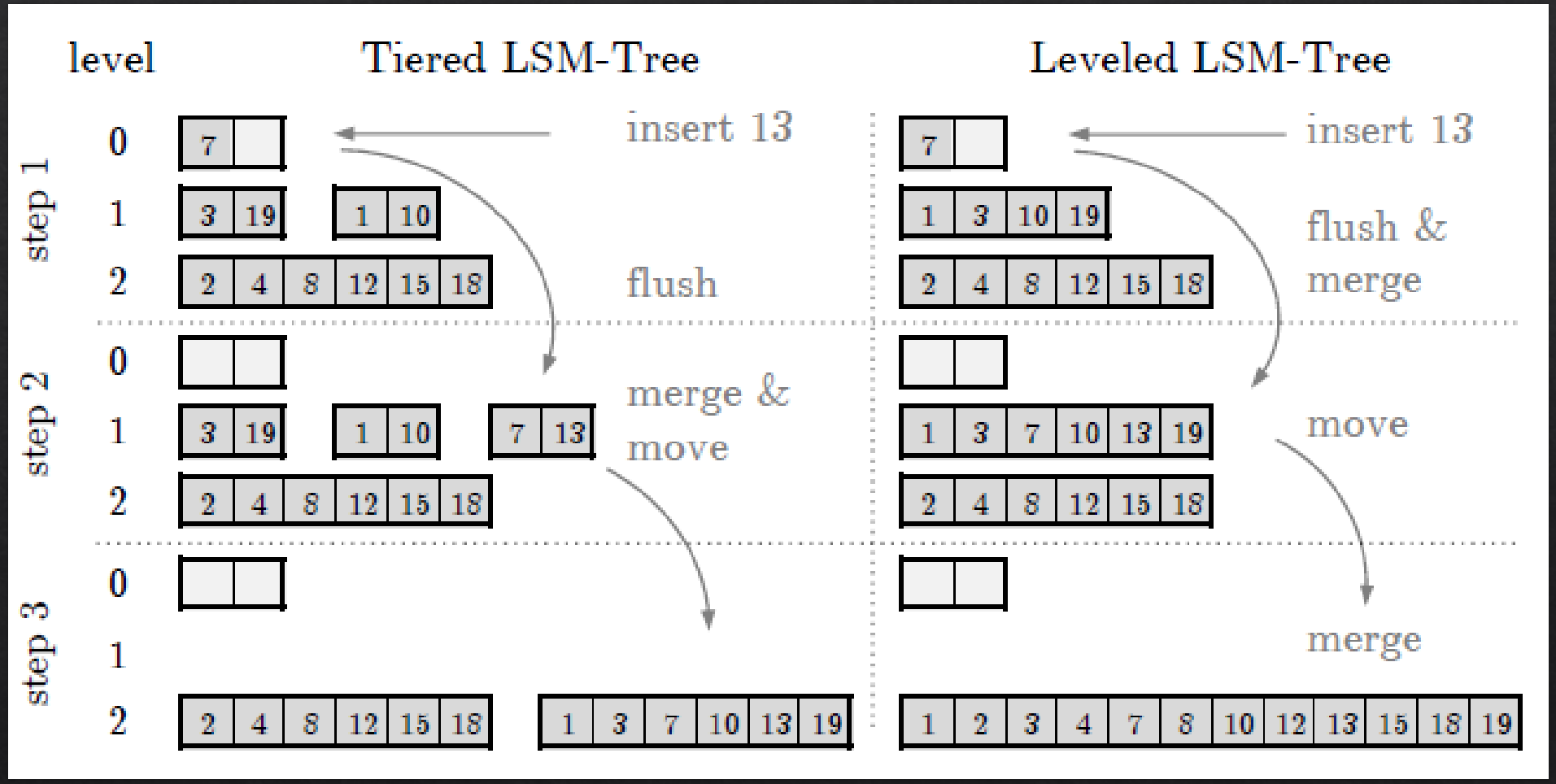
Term	Description	Unit
N	Total number of entries	entries
L	Number of levels	levels
B	Number of entries that fit into a disk page	entries
E	Size of an entry	bits
P	Size of the buffer in disk pages	pages
T	Size ratio between adjacent levels	
T_{lim}	Size ratio value at which point L converges to 1	
M	Total 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

LSM Trees Background
Key Concepts

- **Buffering Updates** :
  - A buffer in memory receives all the updates.
  - The buffer is in Level 0.
  - When buffer fills → empty contents to disk.
  - Disk levels start at 0.
  - The size of the buffer M_{buffer} = P*B*E
  - The overall number of levels. $L = \left\lceil \log_T \left(\frac{N \cdot E}{M_{buffer}} \cdot \frac{T-1}{T} \right) \right\rceil$

- **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.
  - **Two** merging policies : lelveling and tiering

Tier vs Level



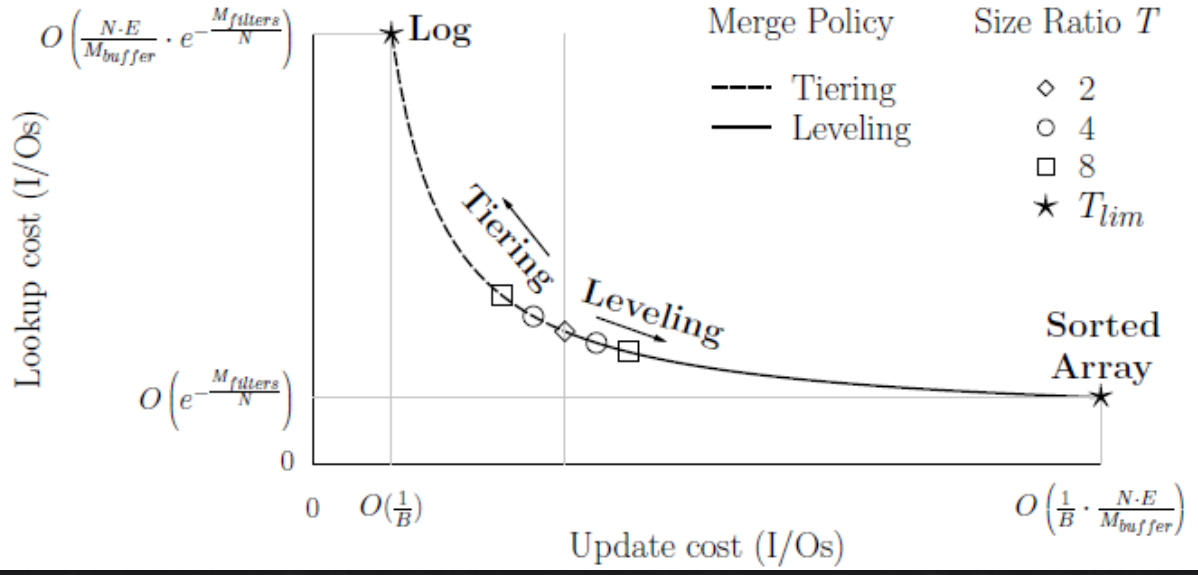
LSM Trees Background
Key Concepts

- **Lookups**:
  - A lookup starts from the memory buffer and traverses the other levels. After finding the first matching entry it terminates. Range lookups require merging of runs on different levels.
- **Probing a Run**:
  - Fence pointers allow to read almost exactly as many pages as we need
- **Bloom Filters**:
  - Each run has a Bloomfilter to facilitate lookups. False positive rate is derived from the number of bits allocated to each entry and the total number of entries

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

Tier vs Level – Round 2

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	$O(e^{\frac{M_{filters}}{N}})$	$O(\frac{1}{B} \cdot \frac{N \cdot E}{M_{buffer}})$



Cost Analysis
Tier vs Level – Round 3

Tier Lookup Cost $O(L \cdot T \cdot e^{-\frac{M_{filters}}{N}})$ I/Os

Tier Update Cost $O(\frac{L}{B})$ I/Os

Level Lookup Cost $O(L \cdot e^{-\frac{M_{filters}}{N}})$ I/Os

Level Update Cost $O(\frac{T \cdot L}{B})$

Design space contentions

- **Contention 1:**

  How do we allocate a given amount of main memory of M-filters among the different Bloom filters?

  → Reallocating memory from one filter to another reduces/increases false positive rate, what is the optimal ?

- **Contention 2:**

  How to allocate the available main memory between the buffer and the filters?

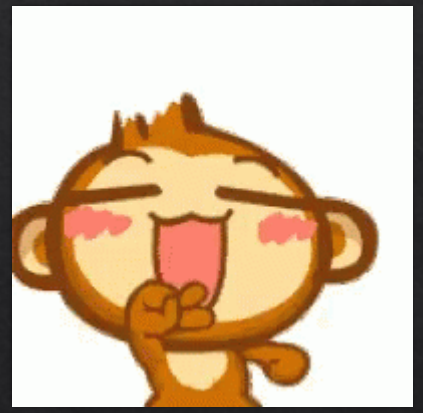
  →Larger buffer == faster lookup AND smaller update cost
  BUT smaller Bloom Filters == slower lookup

- **Contention 3:**

  How to tune the size ratio and merge policy?

  →Decreasing/increasing size ratio on tiering/leveling improves lookup but degrades update.
  With different workloads, how do we optimize ?

That’s where MONKEY comes in



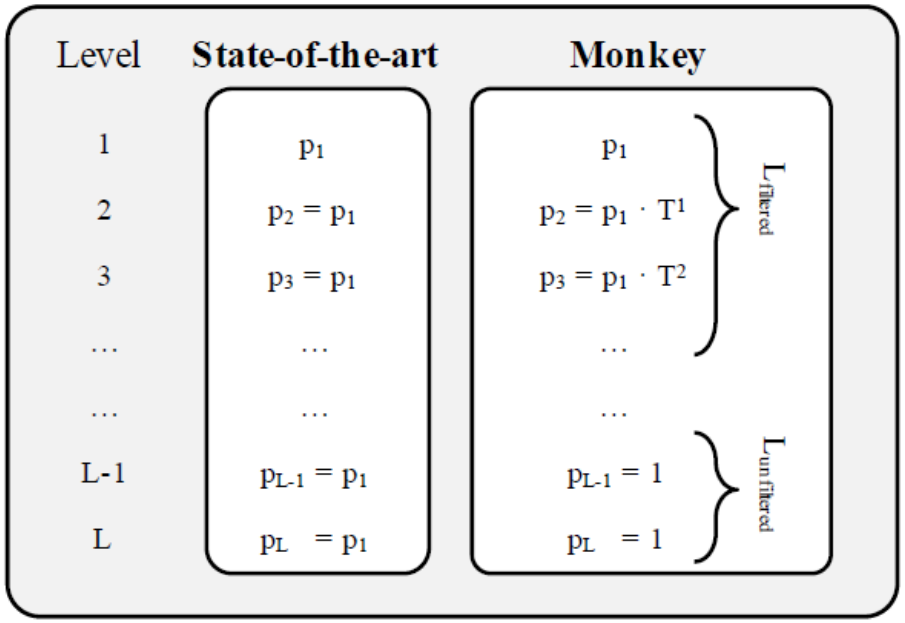
Monkey

- Monkey is an LSM tree based key-value store
- Is able to reach the Pareto curve and find the best possible balance between lookups and updates
- Maximized throughput for uniformly random workloads

Key features:

- Design Knobs
- Minimizing Lookup Cost
- Performance Prediction
- Autotuning

Term describing Monkey



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 $p_1, p_2 \ldots p_L$ to Bloom filters in proportion to the number of key-value pairs in a given level, and a further list of terms used to describe Monkey.

Minimizing Lookup Cost

- Average worst case lookup cost:

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

- Main Memory Footprint

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

Leveling

$$p_i = \begin{cases} 1, & \text{if } i > L_{filtered} \\ \frac{(R - L_{unfiltered}) \cdot (T-1)}{T^{L_{filtered}+1-i}}, & \text{else} \end{cases}$$

for $0 < R \leq L$

and $1 \leq i \leq L$

and $L_{filtered} = L - \max(0, \lfloor R - 1 \rfloor)$

(5)

Tiering

$$p_i = \begin{cases} 1, & \text{if } i > L_{filtered} \\ \frac{R - L_{unfiltered} \cdot (T-1)}{T^{L_{filtered}+1-i}}, & \text{else} \end{cases}$$

for $0 < R \leq L \cdot (T-1)$

and $1 \leq i \leq L$

and $L_{filtered} = L - \max(0, \lfloor \frac{R-1}{T-1} \rfloor)$

(6)

Bloomfilters in Monkey

- Monkey does not implement BloomFilters to all the levels. Why ?
  - → Because of the FPR
  - Because of the fence pointer, probe cost is the same no matter the level
  - Bloomfilters are T times bigger in every level compared to the i-1 level
  - Maintaining low FPR requires much more space

- Monkey finds optimal number of levels to have a BloomFilter
- Monkey sets the FPR proportional to their size

Bloomfilters everywhere

Bloomfilters where it is useful

Predicting Lookup and Update Cost

- Modeling zero result lookup cost (R)

$$R = R_{filtered} + R_{unfiltered}$$

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

$$R_{unfiltered} = \begin{cases} L_{unfiltered}, & \text{with leveling} \\ L_{unfiltered} \cdot (T-1), & \text{with tiering} \end{cases}$$

Predicting Lookup and Update Cost

- Number of deeper levels that have no filter

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

$$M_{threshold} = \frac{N}{\ln(2)^2} \cdot \frac{\ln(T)}{(T-1)} \tag{8}$$

Modeling Worst-Case Non-Zero-Result Lookup Cost (V)

$$V = R - p_L + 1$$

ModelingWorst-Case Update Cost (W)

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

Modeling Worst-Case Range Lookup Cost (Q).

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

Time to compare

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

Merge policy	Monkey Lookup Cost (R) $\frac{M_{threshold}}{T^L} \leq M_{filters} \leq M_{threshold}$ (d)	Monkey Lookup Cost (R) $M_{threshold} \leq M_{filters}$ (e)
(1) Tiering ($T = T_{lim}$)	$O(\frac{N \cdot E}{M_{buffer}})$	$O(\frac{N \cdot E}{M_{buffer}} \cdot e^{-\frac{M_{filters}}{N}})$
(2) Tiering ($2 \leq T < T_{lim}$)	$O(T \cdot \log_T(\frac{N}{M_{filters}}))$	$O(T \cdot e^{-\frac{M_{filters}}{N}})$
(3) Leveling ($2 \leq T < T_{lim}$)	$O(\log_T(\frac{N}{M_{filters}}))$	$O(e^{-\frac{M_{filters}}{N}})$
(4) Leveling ($T = T_{lim}$)	$O(1)$	$O(e^{-\frac{M_{filters}}{N}})$

Time to compare

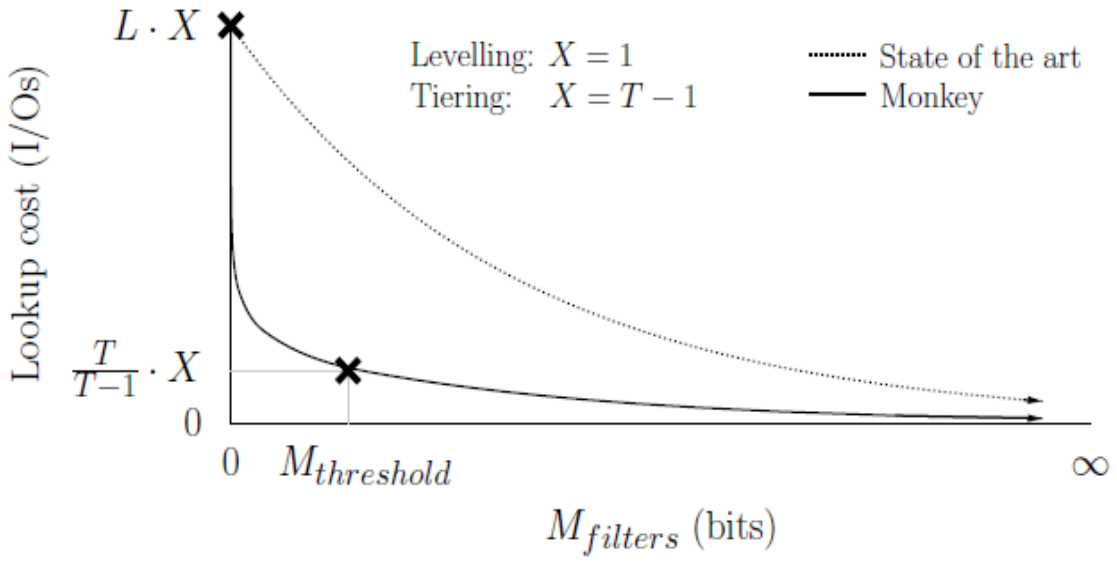


Figure 7: Monkey dominates the state of the art in terms of lookup cost R for all values of $M_{filters}$.

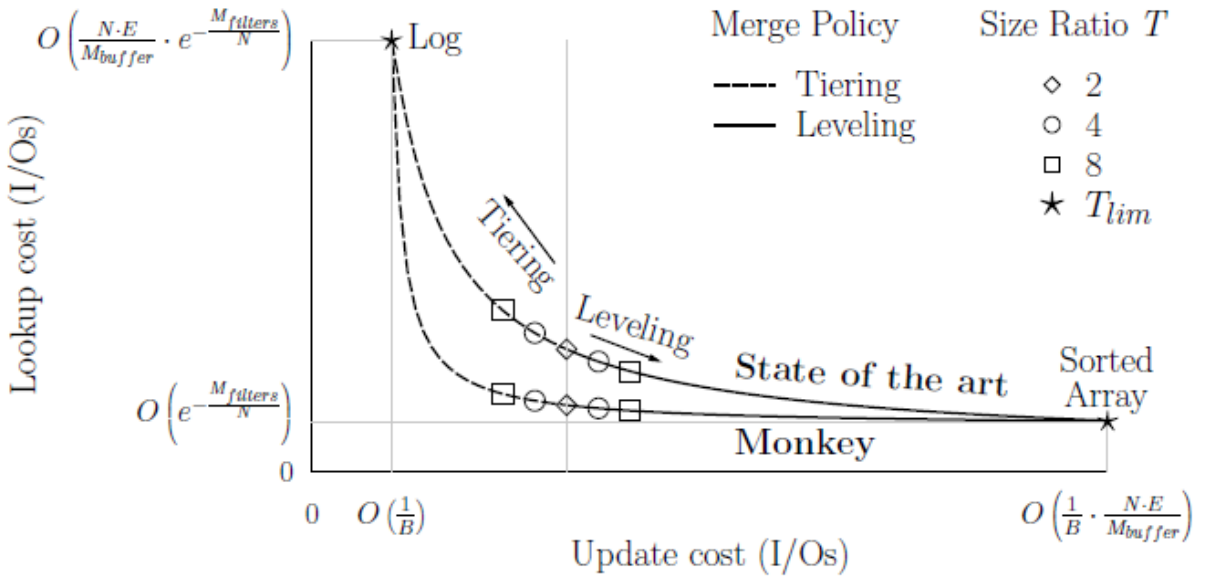
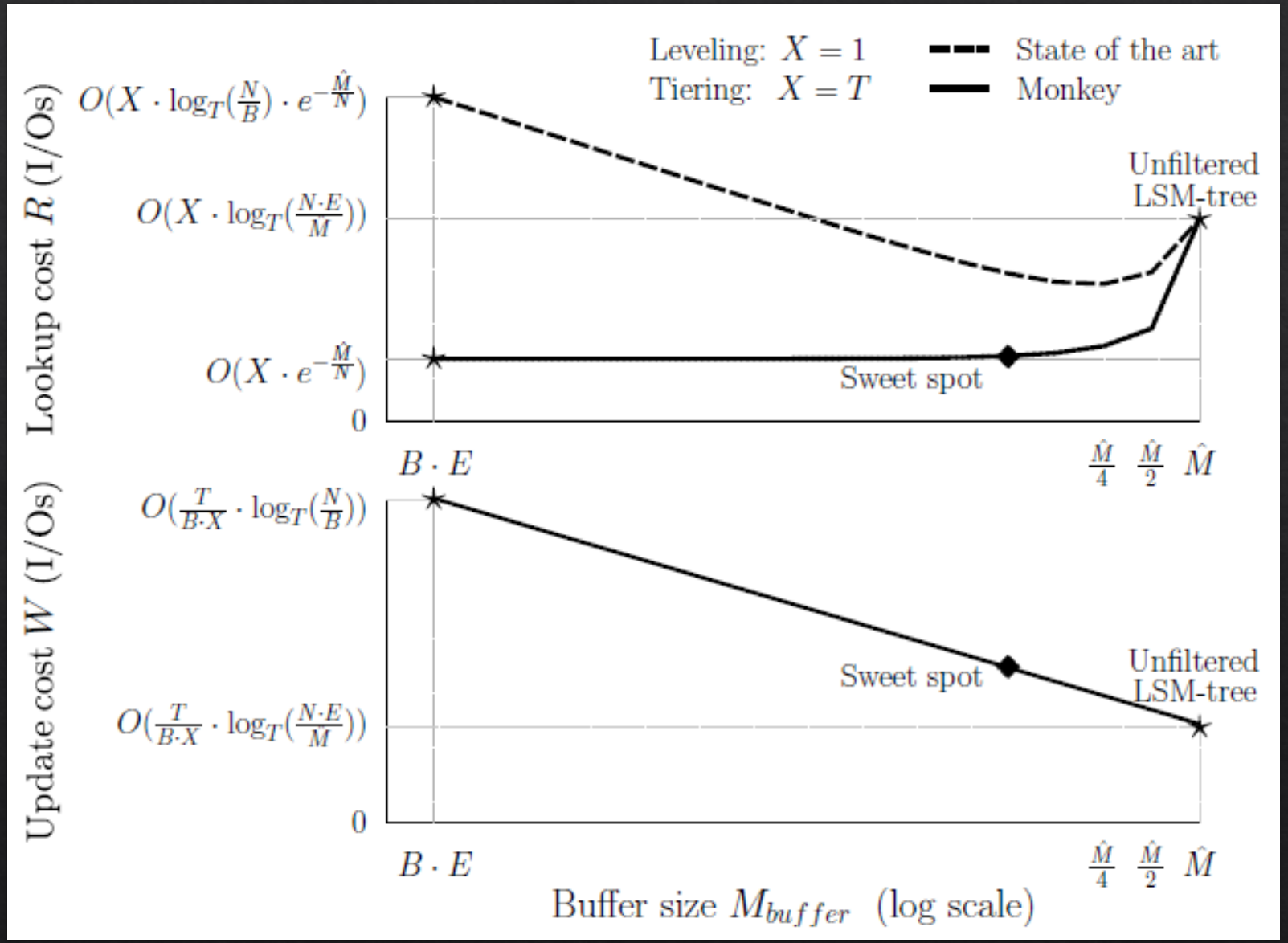


Figure 8: Monkey dominates the state of the art for any merge policy and size ratio.

Scalability and Tunability

- Scaling Bloom filters' footprint with the number of data, lookup cost remains fixed whereas in the state of the art it increases at a logarithmic rate.
- Lookup cost is independent of the entry size, and so it does not increase for data sets with larger entry sizes.
- Lookup cost is independent of the buffer size. This simplifies tuning relative to the state of the art → no need to carefully balance main memory allocation between the buffer and filters to optimize lookup performance.

Scalability and Tunability

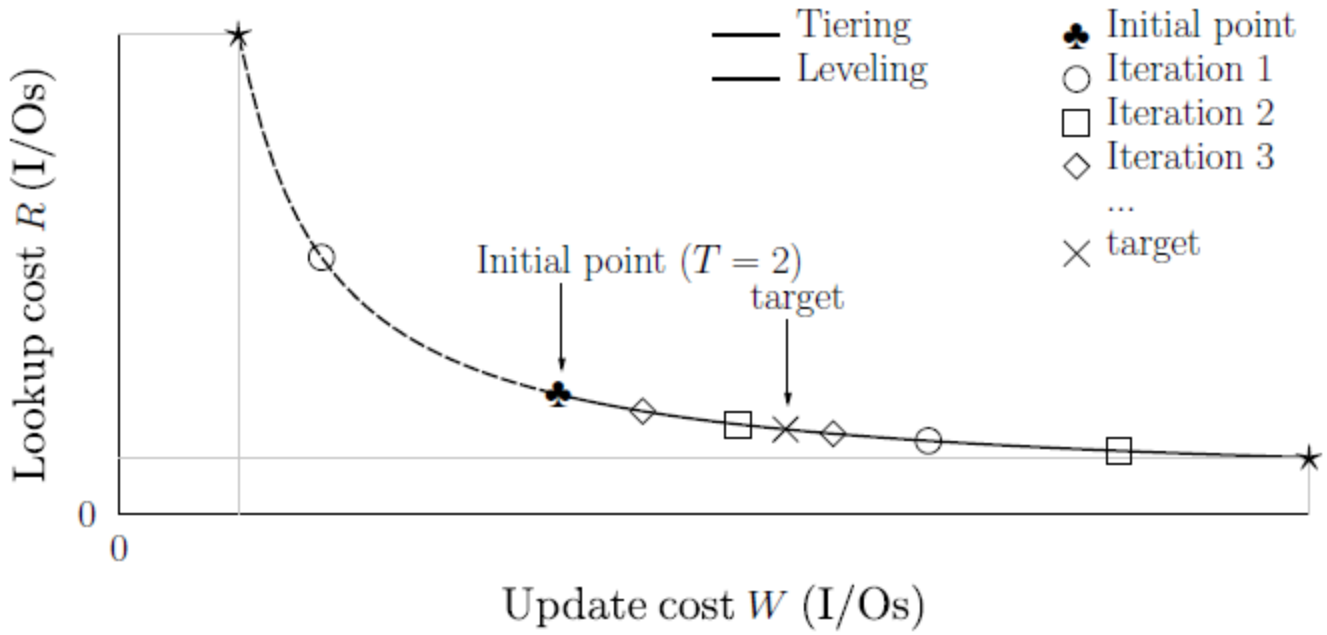


Tuning the Size Ratio and Merge Policy

- The merge policy and size ratio are complementary means of navigating the same tradeoffcontinuum.
- Monkey a divide and conquer algorithm that linearizes this continuum into a single imension and finds the tuning that maximizes throughput.

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.



Tuning Main Memory Allocation

- Three-step strategy
  - Alocate first M bytes to buffer, since there are levels where bloomfilters might have no use.
  - Allocate 5% to the buffer and the rest 95% to the bloomfilters
  - Continue following step two until the I/O overhead due to false positives becomes negligible
  - If there is more main memory in our budget beyond step two → allocate it to the buffer to further reduce update cost.

Experiments

- Monkey Scales Better with Data Volume
  - 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 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.
- Monkey Needs Less Main Memory
  - Repeat 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).
- Monkey Improves Lookup Cost for Different Workloads
  - Experiment shows that Monkey significantly improves lookup latency for non-zero-result lookups across a wide range of temporal locality in the query workload. The results are shown in Figure 11 (D).

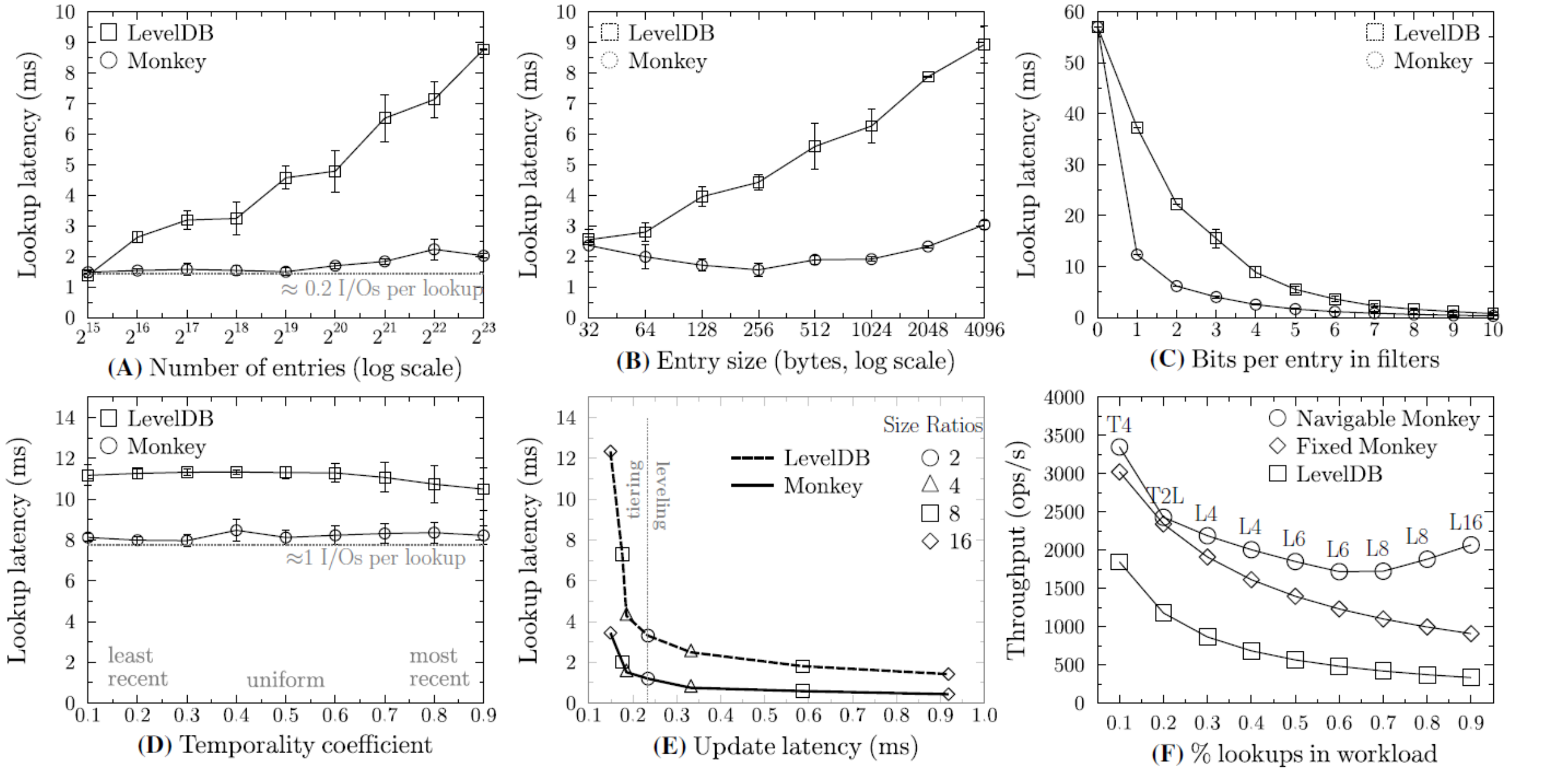


Figure 11: Monkey improves lookup cost under any (A) number of entries, (B) entry size, (C) amount of memory, (D) lookup locality, and (F) merge policy and size ratio. It navigates the design space to find the design that maximizes throughput (F).

Experiments

- Monkey Reaches the Pareto Curve
  - Repeat the experimental setup multiple times, each time using a different configuration of size ratio and merge policy. 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).
- Monkey Navigates the Design Space to Maximize Throughput.
  - 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%. Results are shown in Figure 11 (F).

Discussion

Thank you