Mnemosyne: Dynamic Workload-Aware BF Tuning via Accurate Statistics in LSM trees

Zichen Zhu

Yanpeng Wei

Juhyoung Mun

Manos Athanassoulis

Log-Structured Merge-tree (LSM-tree)

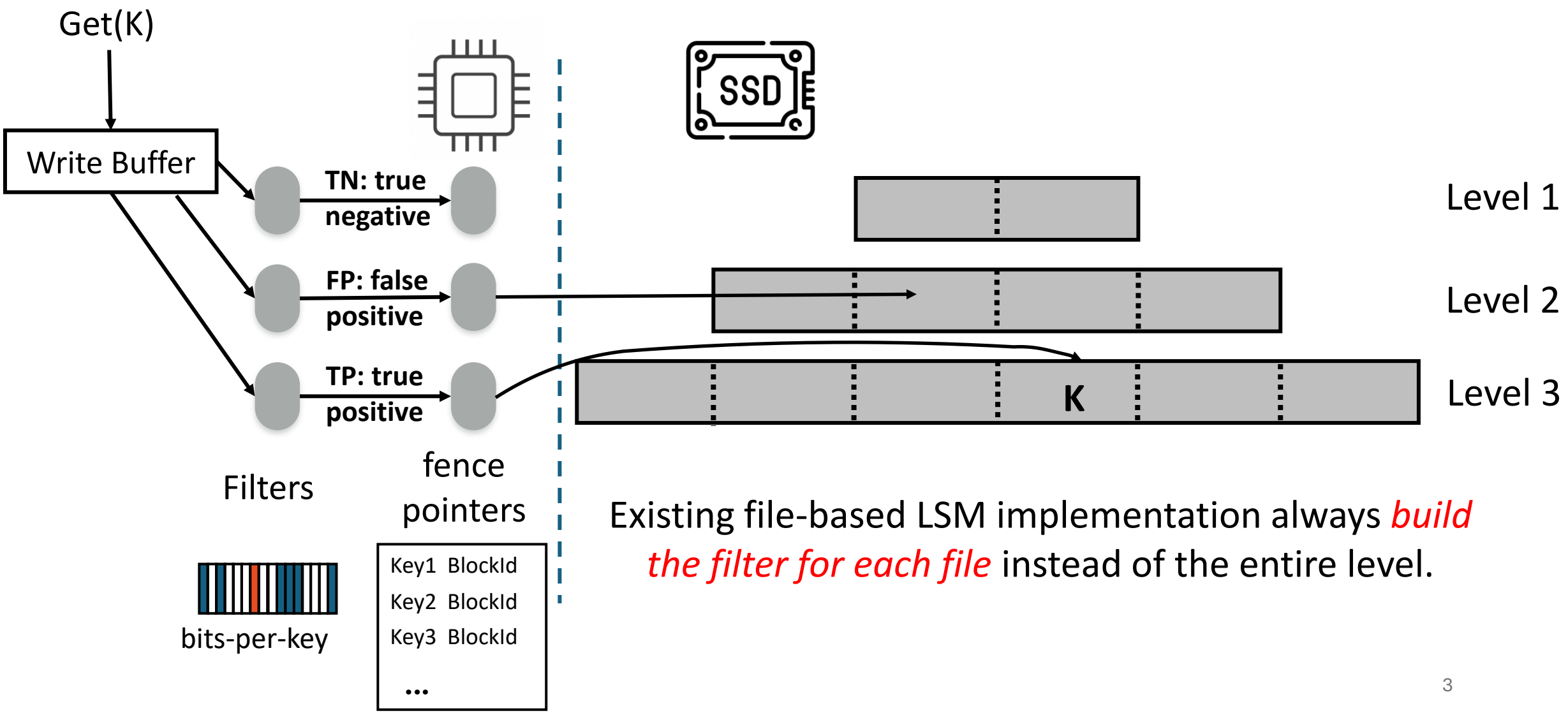
Widely adopted because it offers fast ingestion rate and competitive read latency

NoSQL

Relational

Time-series

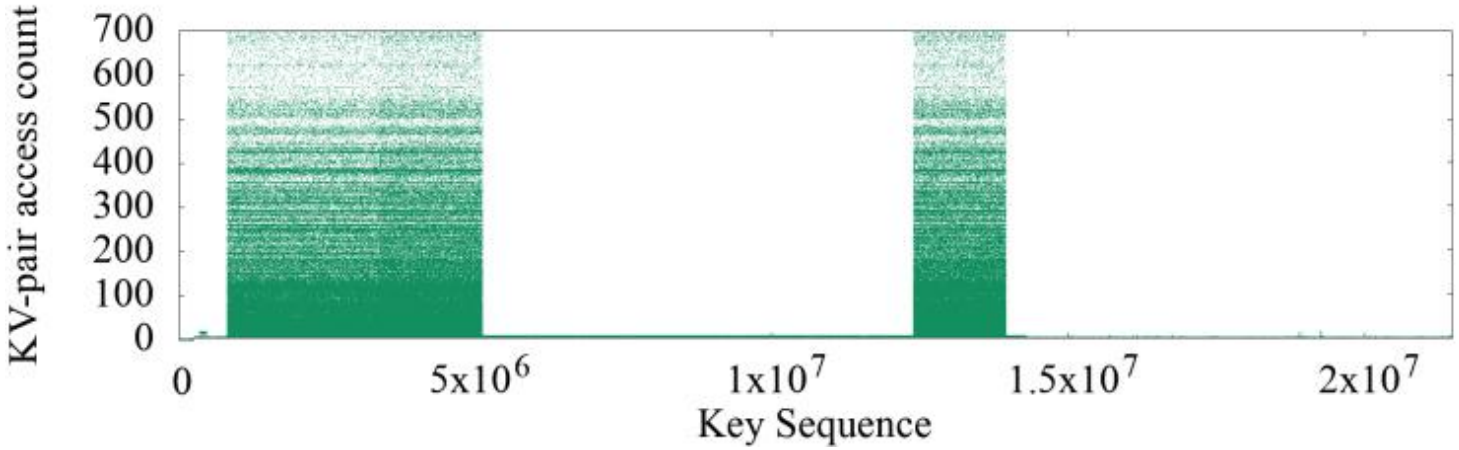
LSM Basics - Get(K)



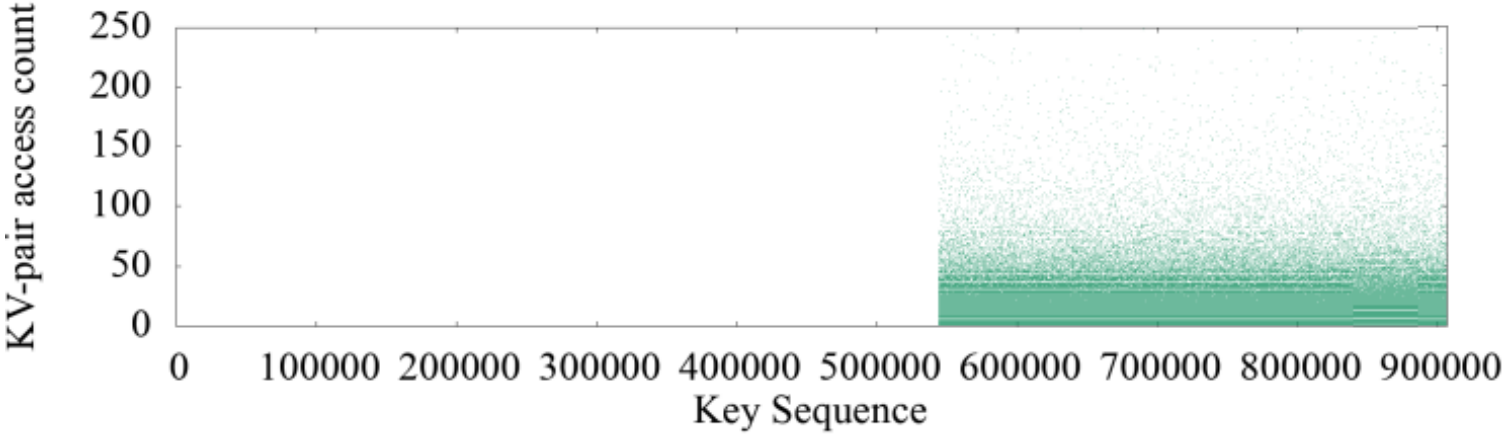
Existing file-based LSM implementation always *build the filter for each file* instead of the entire level.

Access Skew

ZippyDB



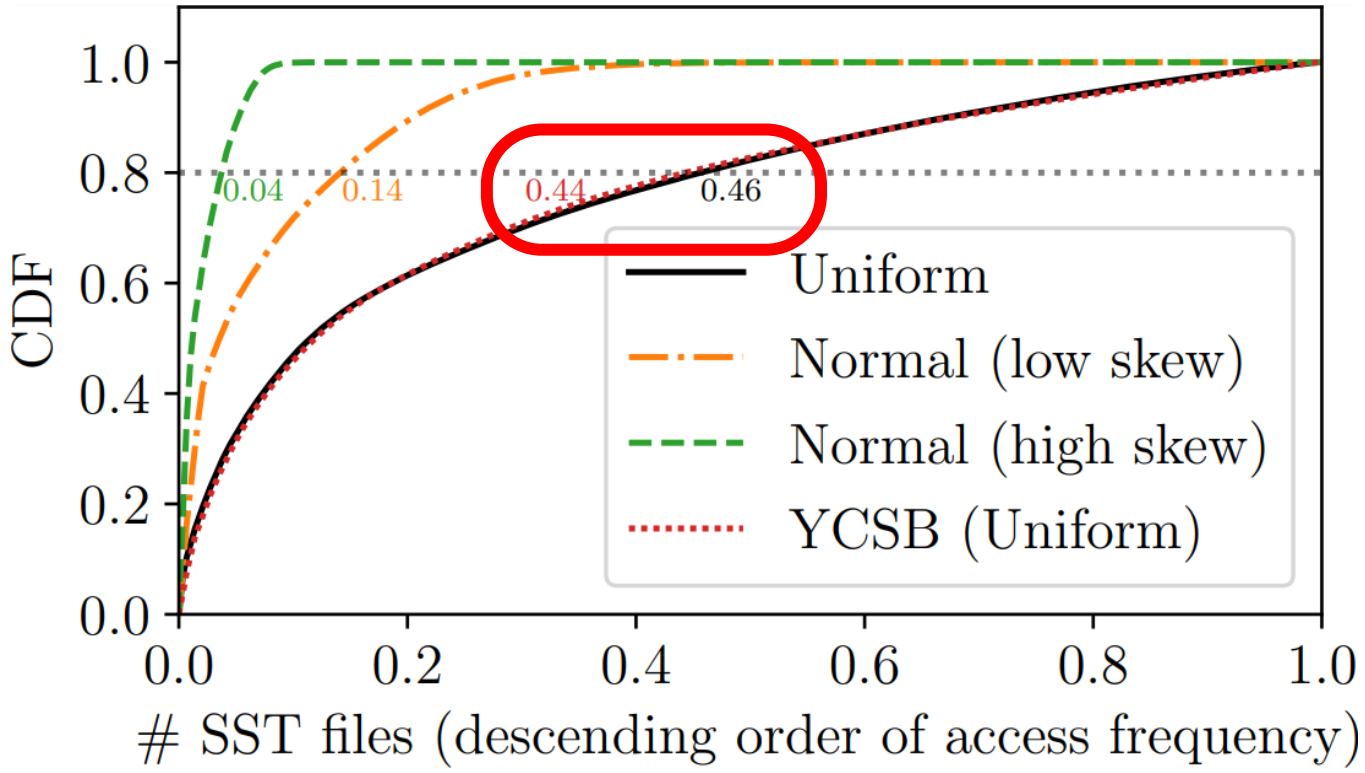
UP2X

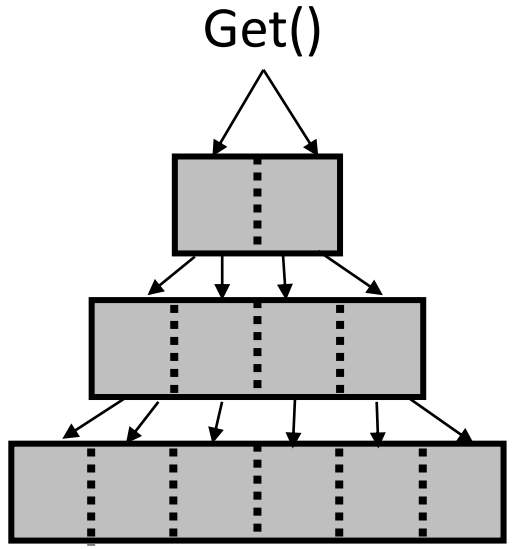


Real-life workloads exhibit access skew

Cao et al. *Characterizing, modeling, and benchmarking RocksDB Key-Value workloads at Facebook*. FAST 2020.

Access Frequency Patterns





Even in a perfectly uniform workload,

80% of the lookups are directed to **44~46% of the files**

How We Exploit the Access Skew

For Bloom Filter,

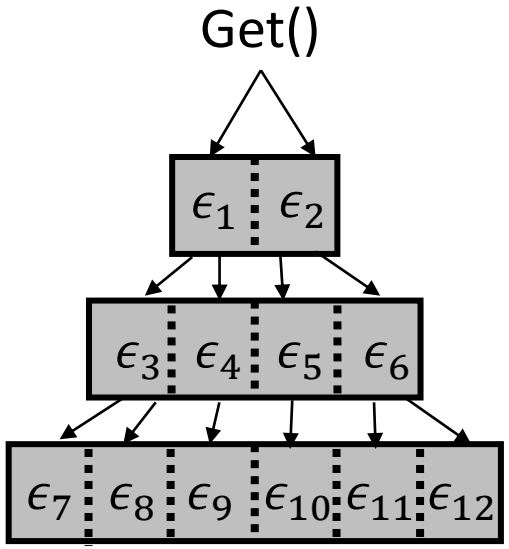
$$g(\epsilon, n) = -\frac{n \cdot \log \epsilon}{(\ln 2)^2}$$

For Cuckoo Filter,

$$g(\epsilon, n) = n \cdot (\log_2 \frac{1}{\epsilon} + 1 + \log_2 b)/\alpha$$

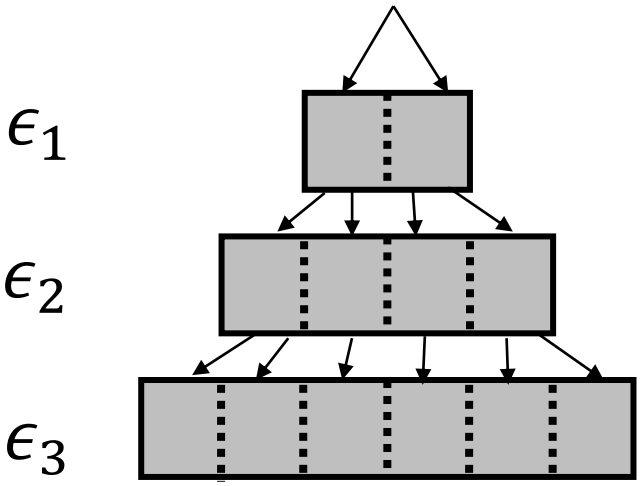
$$\min_{\{\epsilon_i\}} \sum_{i=1}^{F} z_i \cdot \epsilon_i$$

$$s.t. \sum_{i=1}^{F} g(\epsilon_i, n_i) \leq M$$



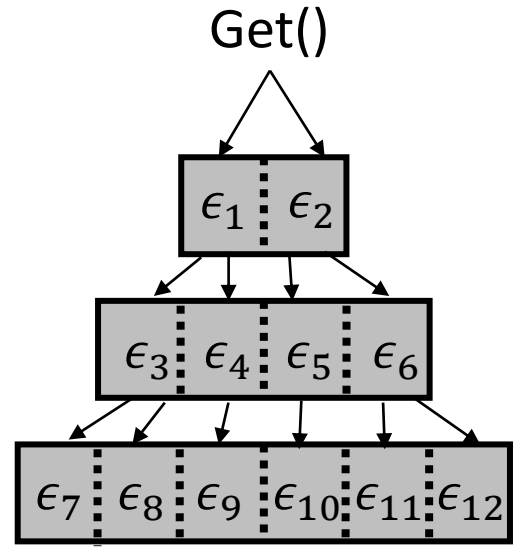
Notation	Meaning
F	The total number of files
z_i	The frequency of empty point queries for Filter i
ϵ_i	The false positive rate of Filter i
n_i	The number of elements in Filter i
$g(\epsilon, n)$	A function that returns the space for a filter using ϵ and n
M	The overall memory for filters

Monkey vs. Ours



$$\min_{\{\epsilon_j\}} \sum_{j=1}^{L} \epsilon_j \qquad s.t. \sum_{j=1}^{L} g(\epsilon_j, n_j) \le M$$

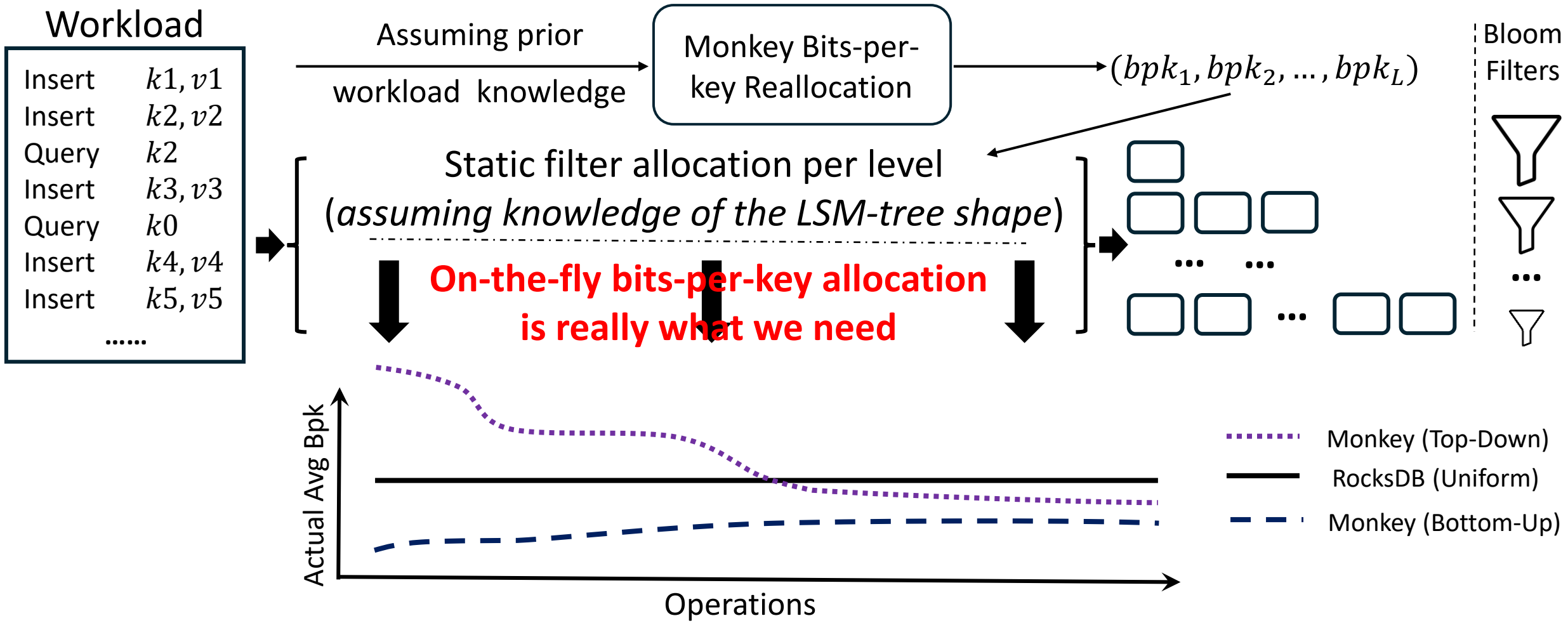
$$\min_{\{\epsilon_i\}} \sum_{i=1}^{F} z_i \cdot \epsilon_i \qquad s.t. \sum_{i=1}^{F} g(\epsilon_i, n_i) \le M$$



Notation	Meaning
F	The total number of files
z_i (z_j)	The frequency of empty point queries for Filter i (Level j)
ϵ_i (ϵ_j)	The false positive rate of Filter i (Level j)
n_i (n_j)	The number of elements in Filter i (Level j)
$g(\epsilon, n)$	A function that returns the space for a filter using ϵ and n
M	The overall memory for filters
L	The number of levels

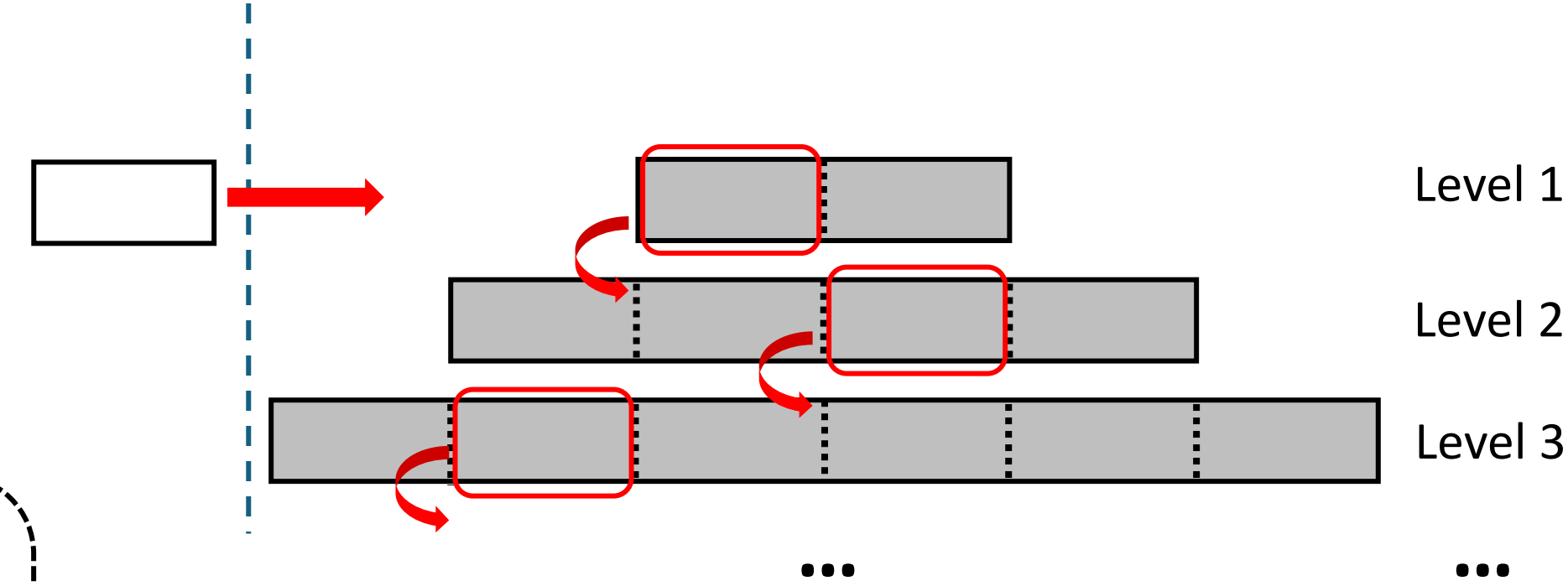
Dayan et al. *Monkey: Optimal Navigable Key-Value Store*. SIGMOD 2017.

Monkey Workflow



Static filter allocation in Monkey makes the actual bits-per-key either overutilized or underutilized compared to the user-defined one.

Solution: Mnemosyne



$$\min_{\{\epsilon_i\}} \sum_{i=1}^{F'} z_i \cdot \epsilon_i$$

$$s.t. \sum_{i=1}^{F'} g(\epsilon_i, n_i') \leq M'$$

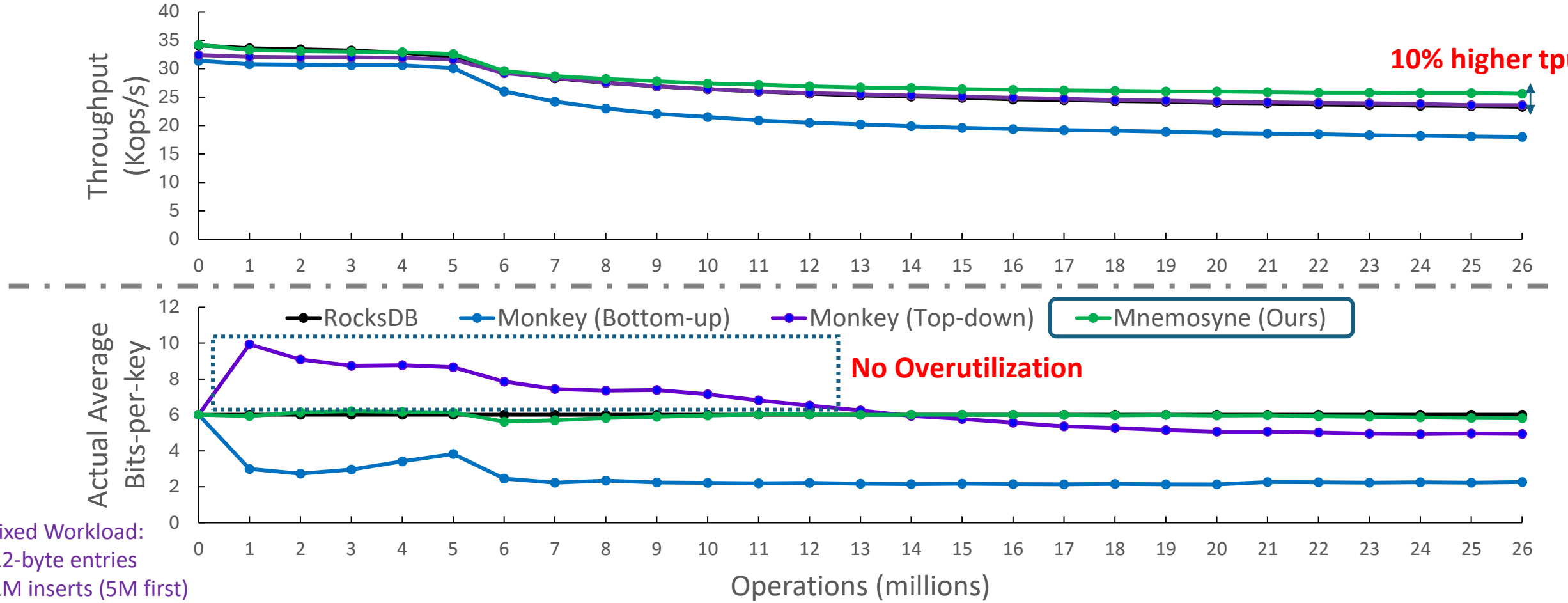
$$\epsilon_i \leq 1$$

Mnemosyne: *Solve the problem during each compaction*.

Challenge 1: Efficiency? ⇨ A fast solver based on dual theory

Challenge 2: How to get z_i? ⇨ Merlin: window-based tracking

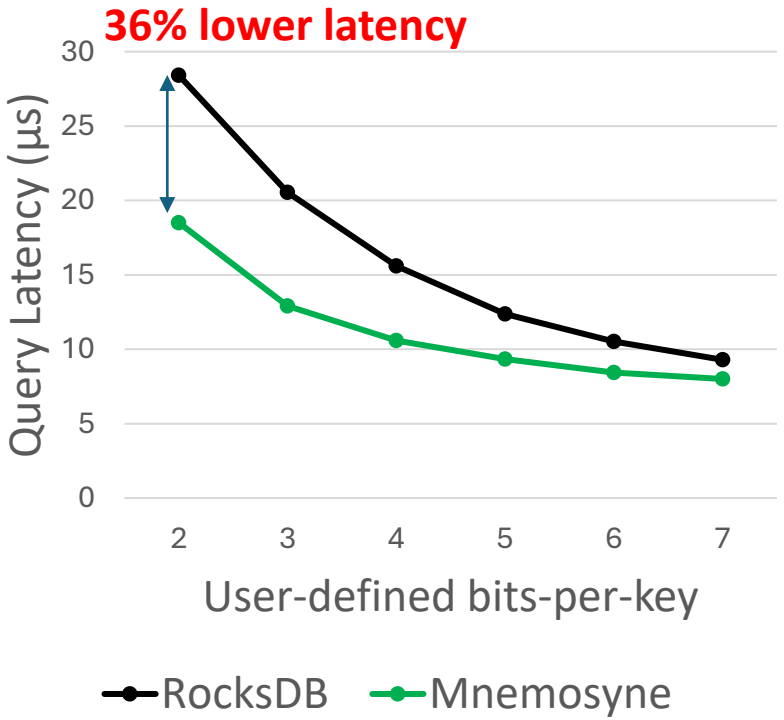
Experimental Results



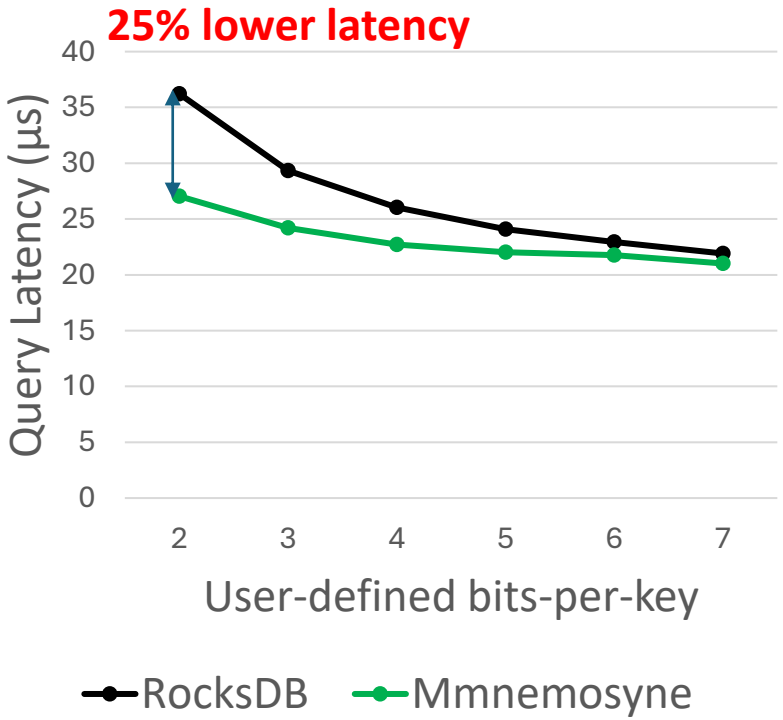
Mixed Workload:
512-byte entries
11M inserts (5M first)
5M updates
10M empty queries
Uniform distribution
6 bits-per-key

Mnemosyne has **higher throughput** than RocksDB and Monkey **without overusing** bits-per-key.

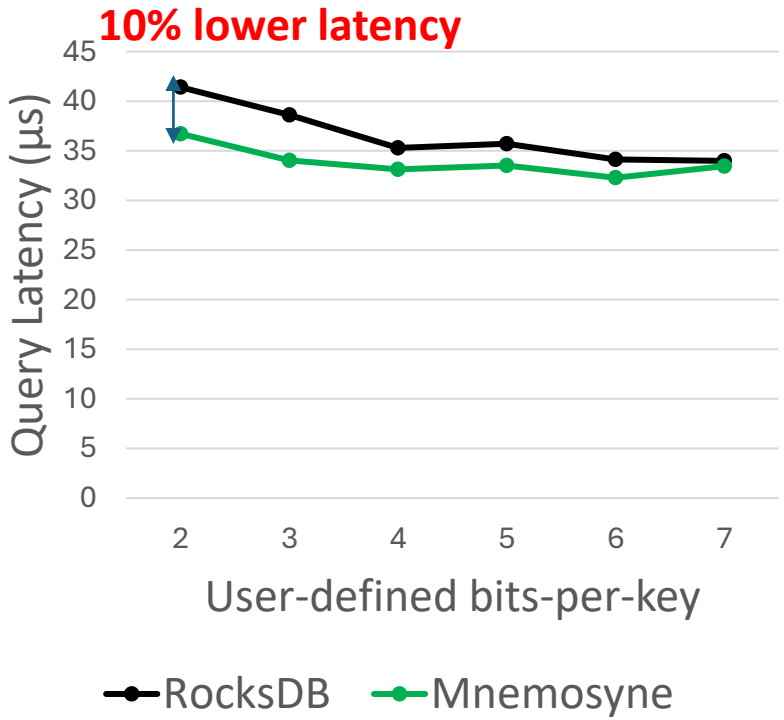
Experimental Results



Empty Query Ratio $Z = 1.0$



Empty Query Ratio $Z = 0.5$



Empty Query Ratio $Z = 0.0$

Mixed Workload:
512-byte entries
Preload 20M inserts
10M updates
31M queries
Normal distribution

Mnemosyne does **not need to have prior knowledge** and achieves much **lower query latency** for skewed accesses.

Summary of Mnemosyne

Mnemosyne outperforms RocksDB by up to 36% without prior workload knowledge or over-allocating memory (bits-per-key).