Reducing Bloom Filter CPU Overhead in LSM-Trees on Modern Storage Devices

Ju Hyoung Mun, Zichen Zhu, Aneesh Raman, Manos Athanassoulis

jmun@bu.edu, zczhu@bu.edu, aneeshr@bu.edu, mathan@bu.edu

presentation at DAMON 2021

Log-Structured Merge Trees

Widely adopted because they balance read performance and ingestion

RocksDB

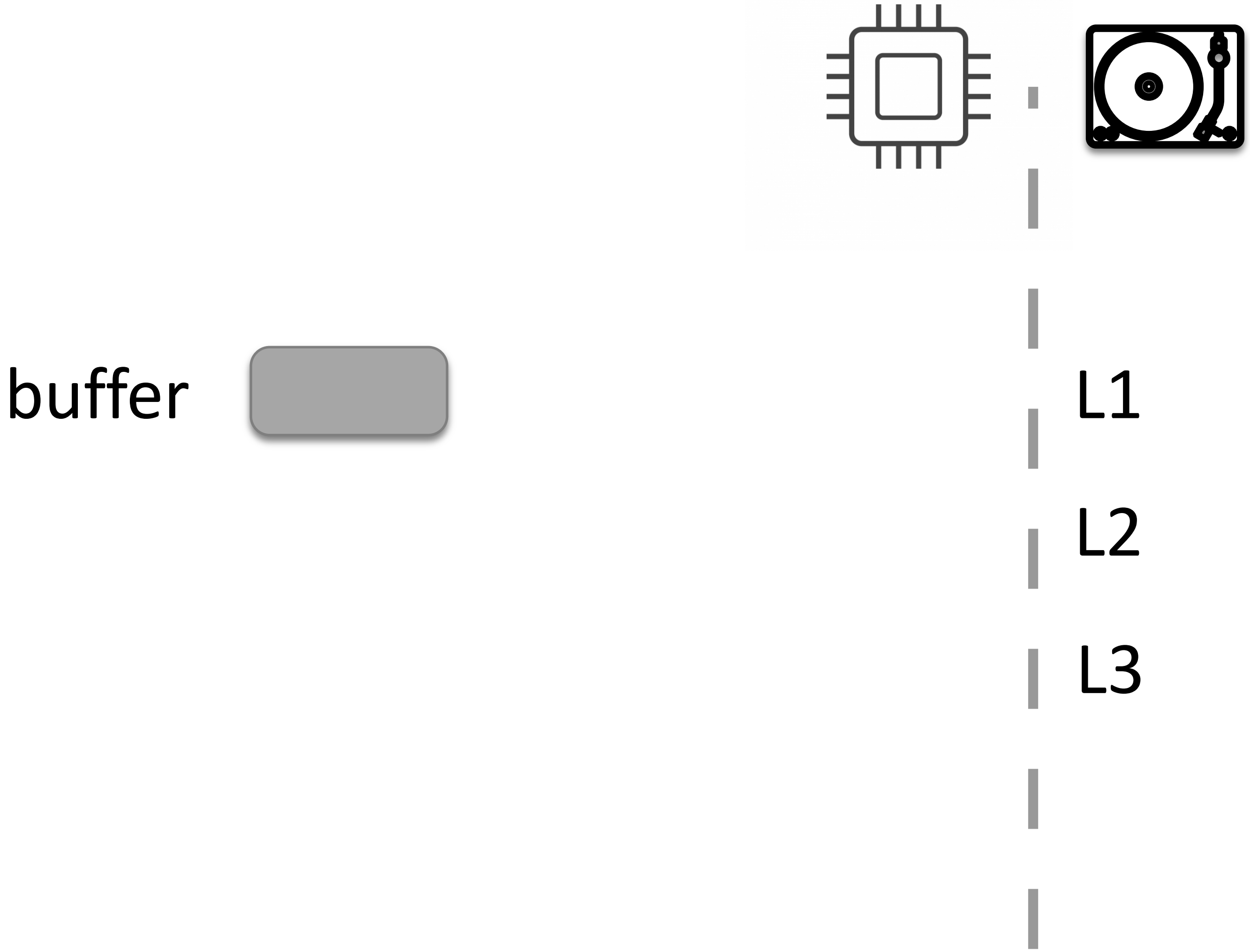
amazon DynamoDB

APACHE HBASE

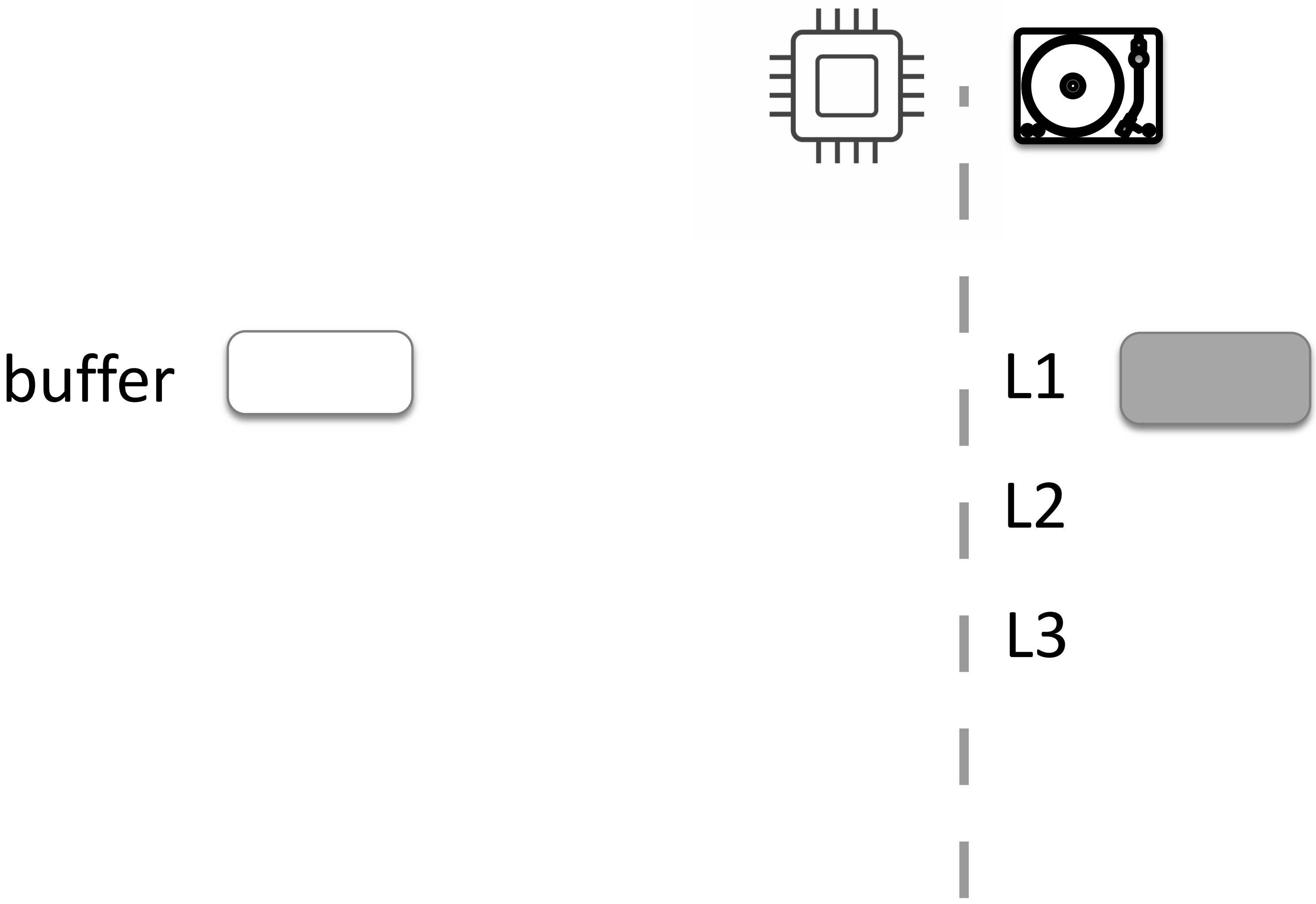
levelDB

cassandra

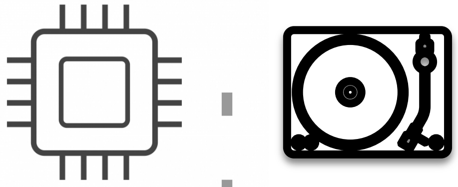
Log-Structured Merge Trees



Log-Structured Merge Trees



Log-Structured Merge Trees



buffer

L1

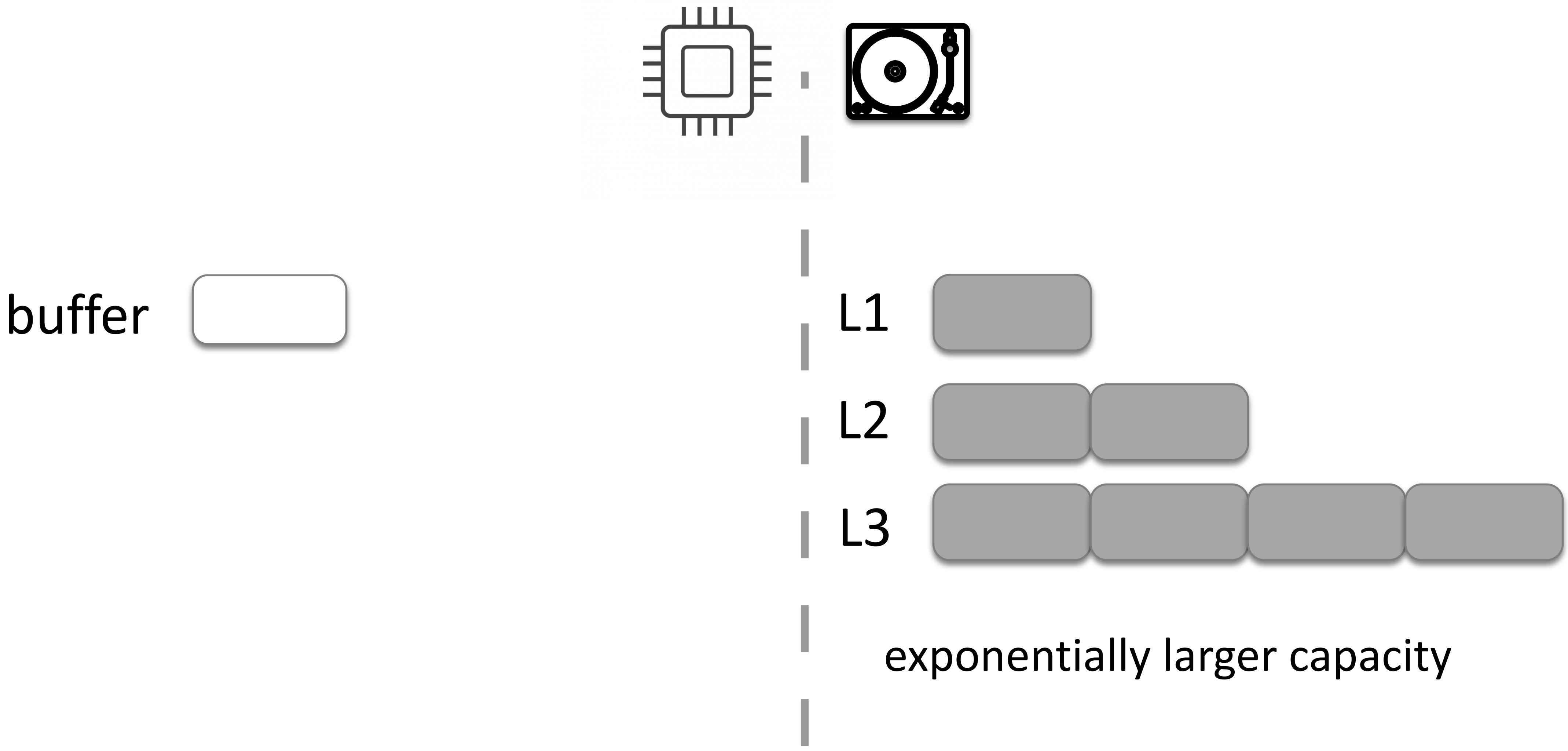
L2

L3

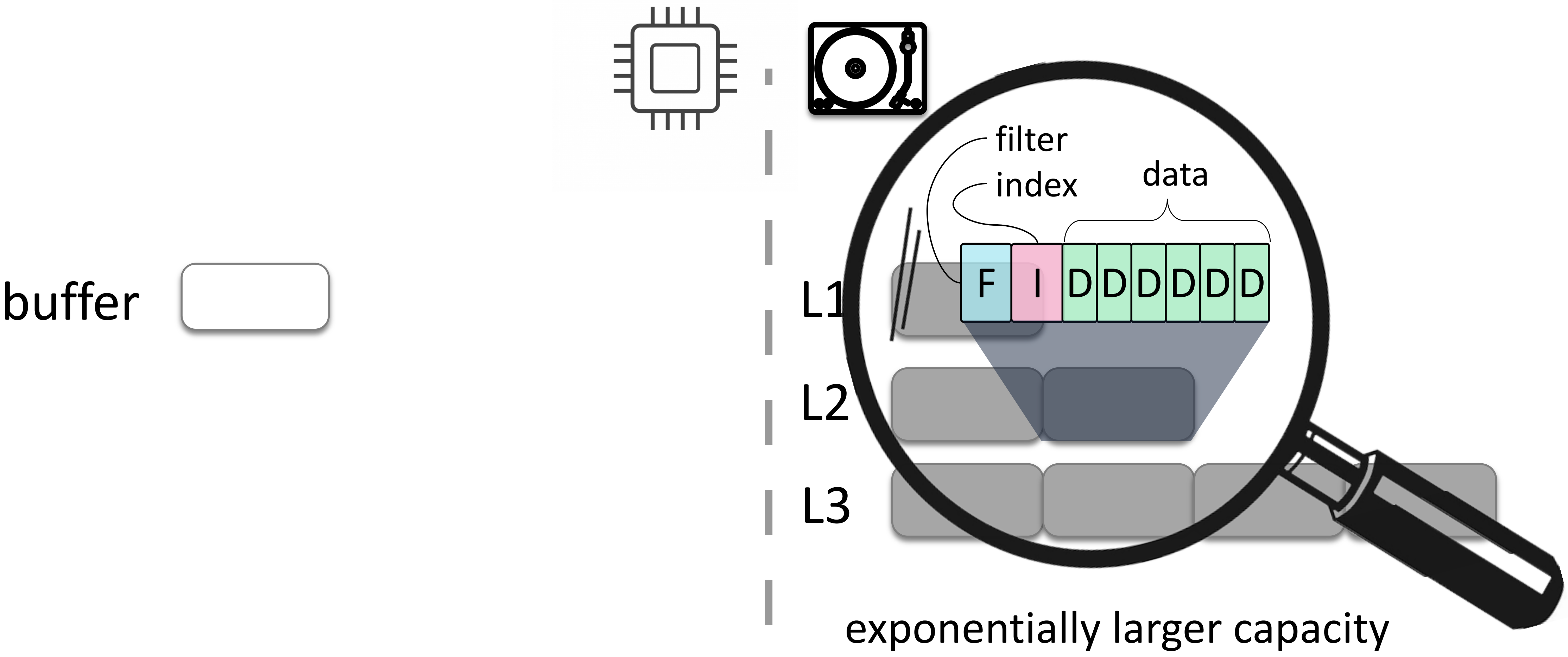
size ratio = T

exponentially larger capacity

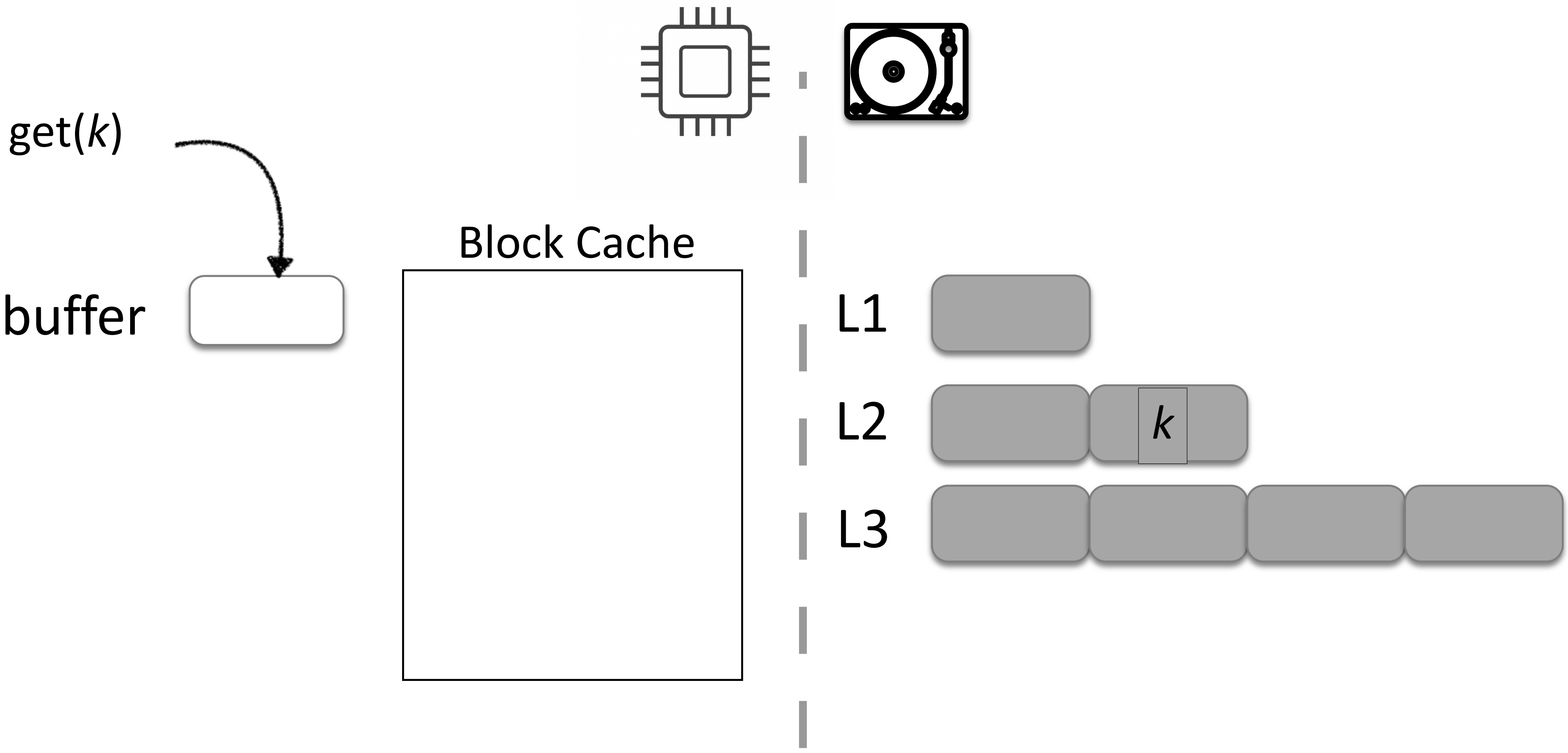
Log-Structured Merge Trees



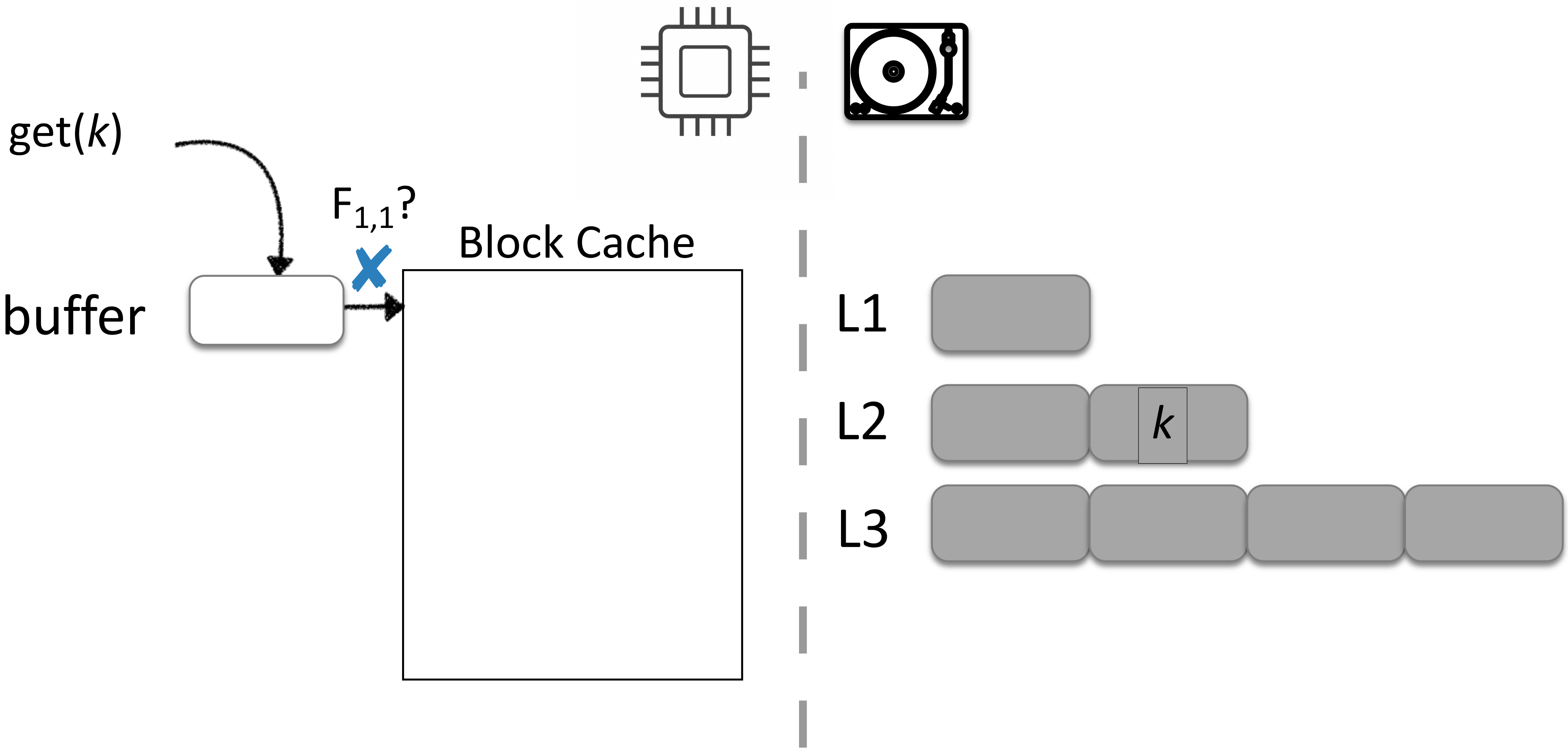
Log-Structured Merge Trees



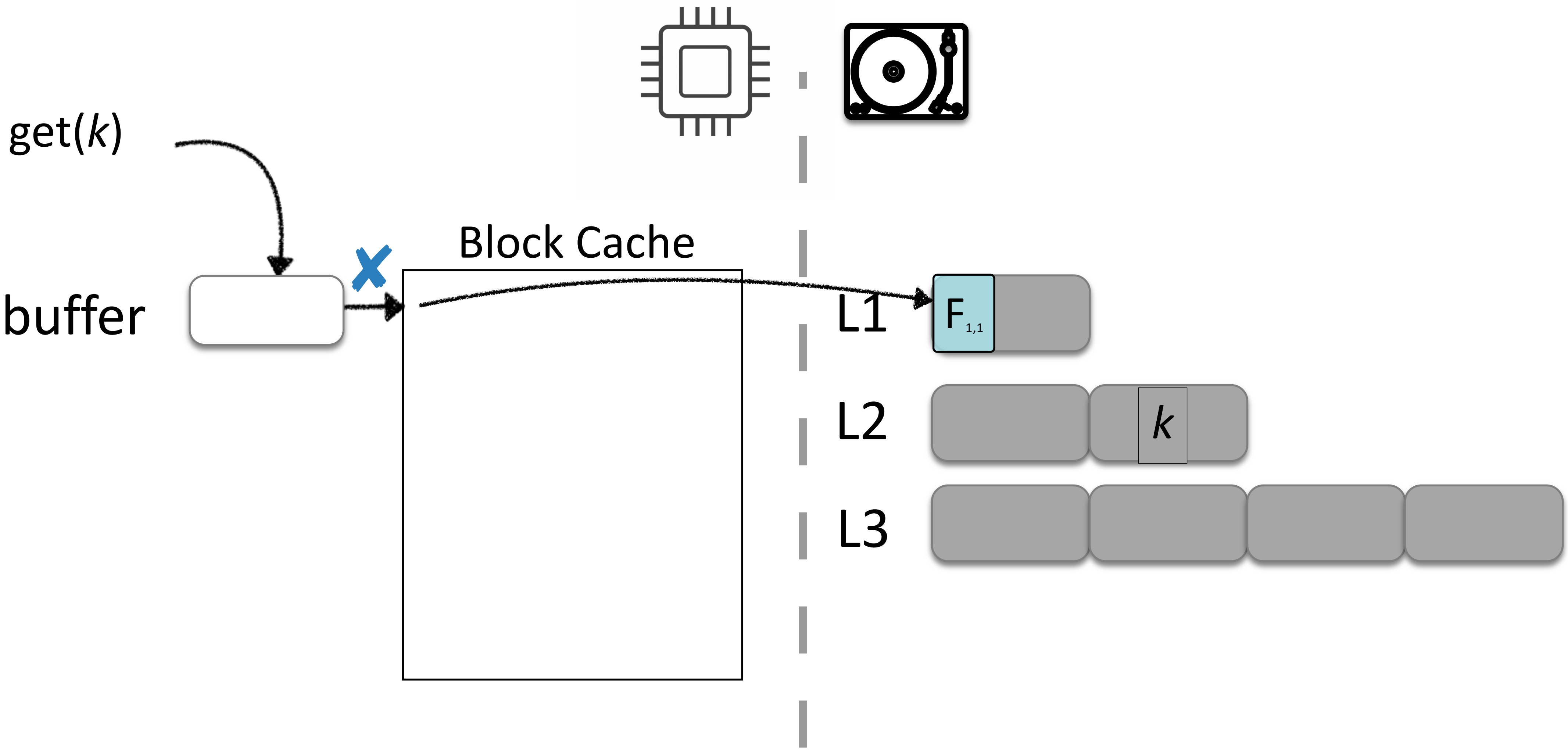
Log-Structured Merge Trees



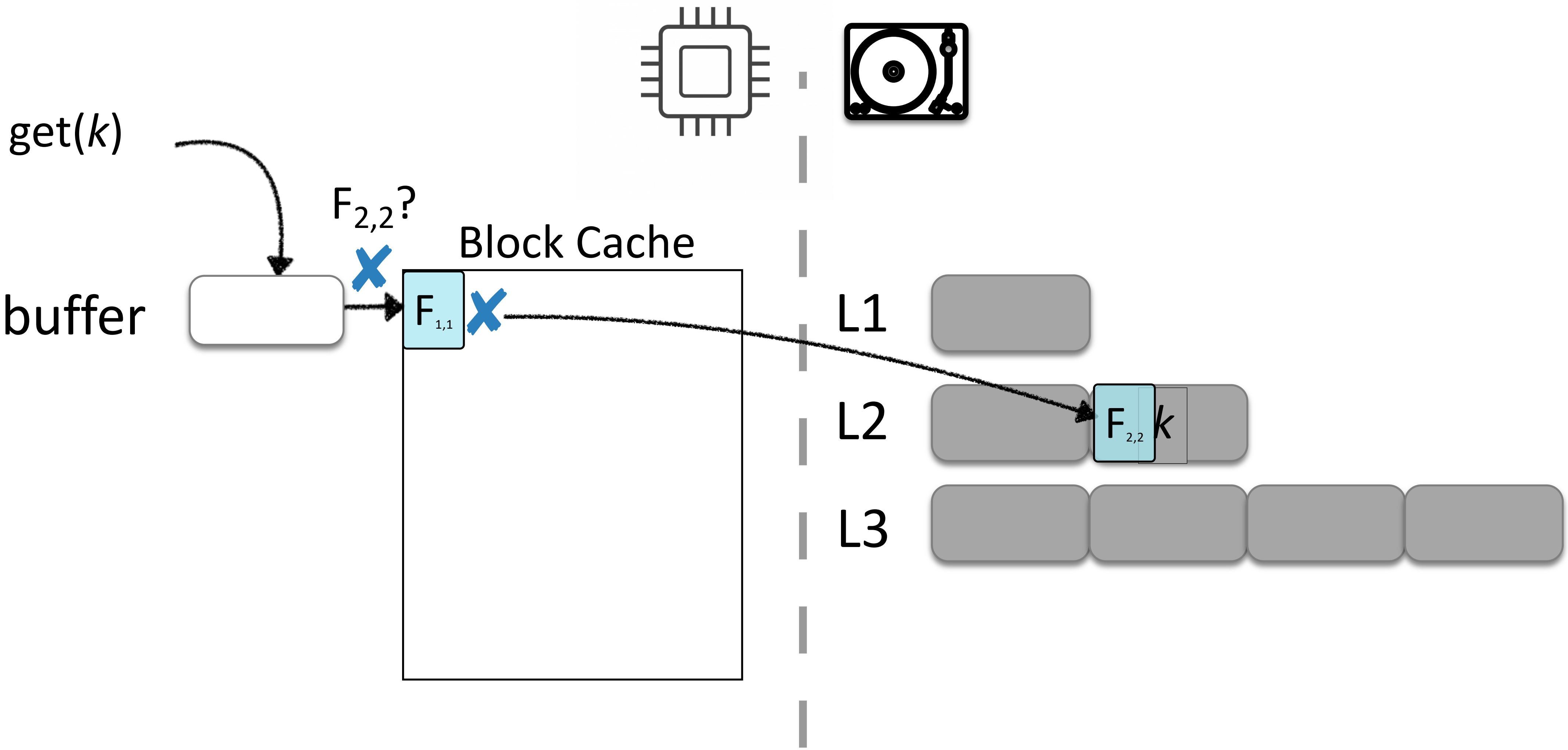
Log-Structured Merge Trees



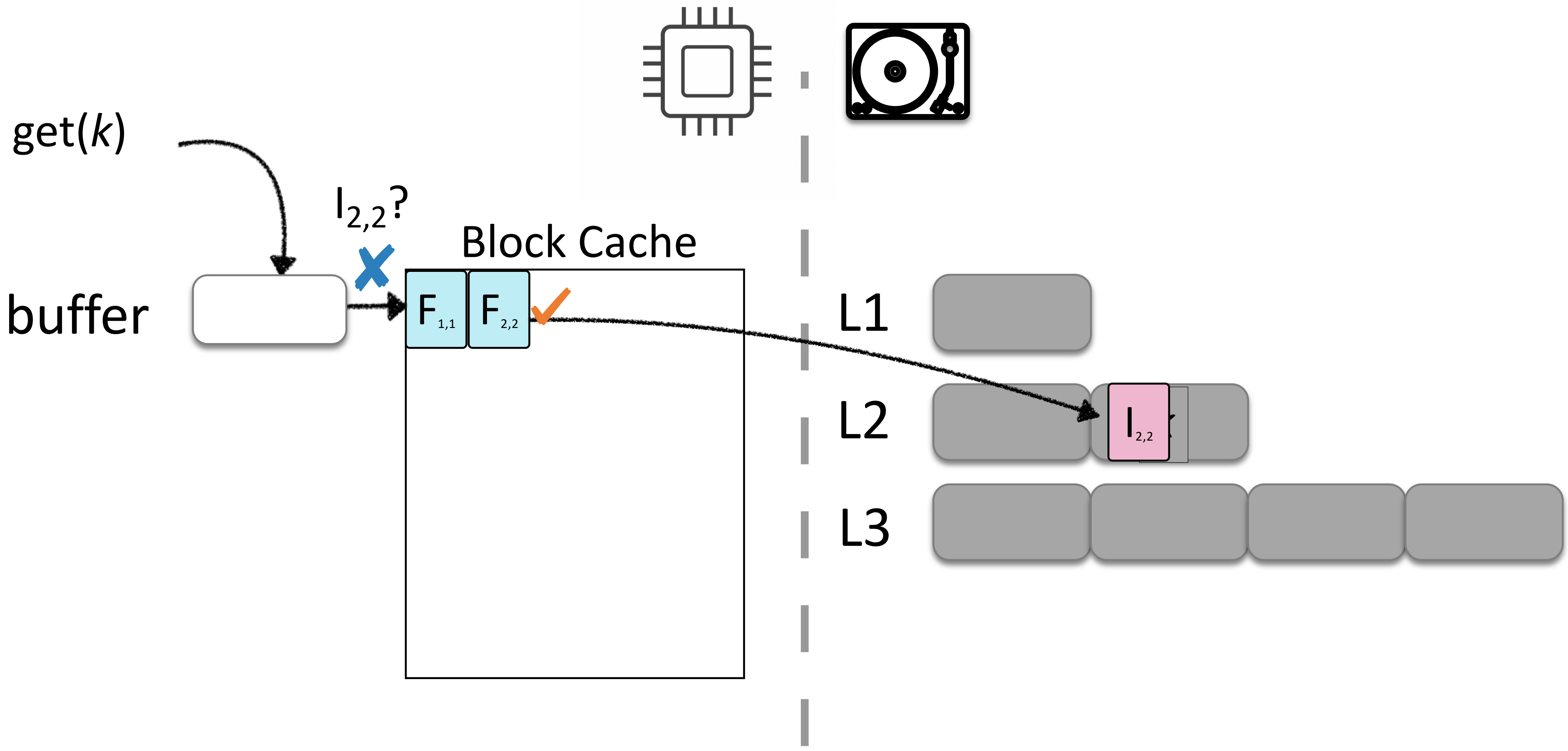
Log-Structured Merge Trees



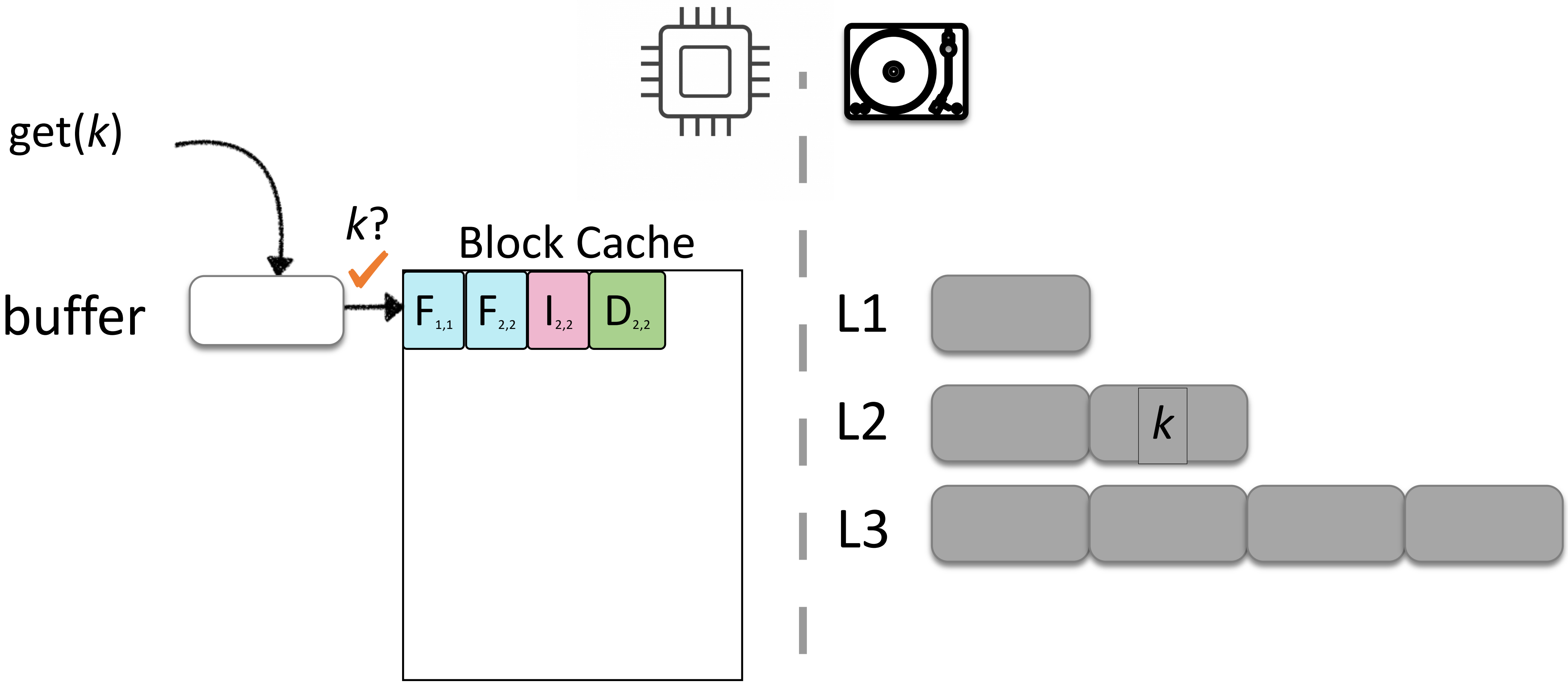
Log-Structured Merge Trees



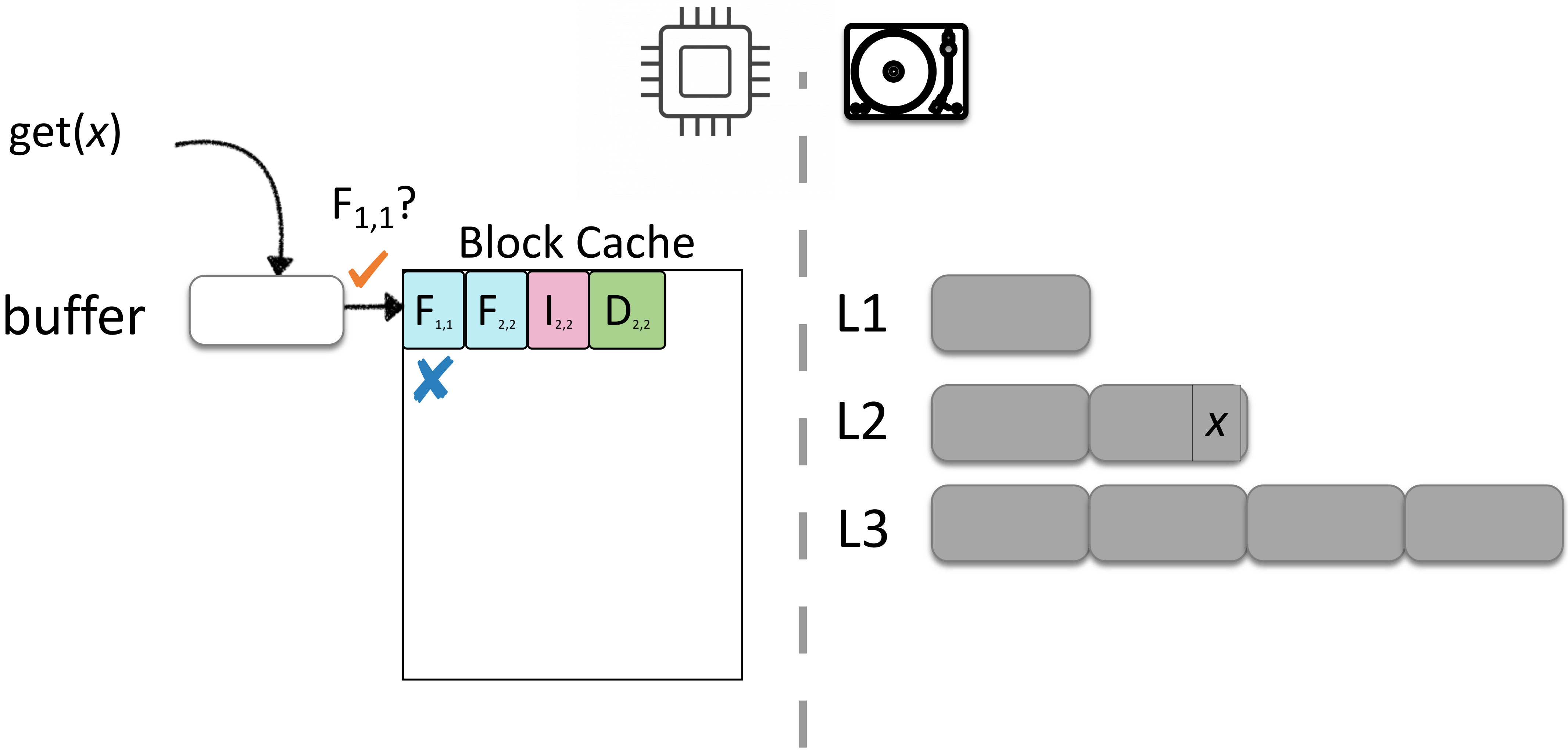
Log-Structured Merge Trees



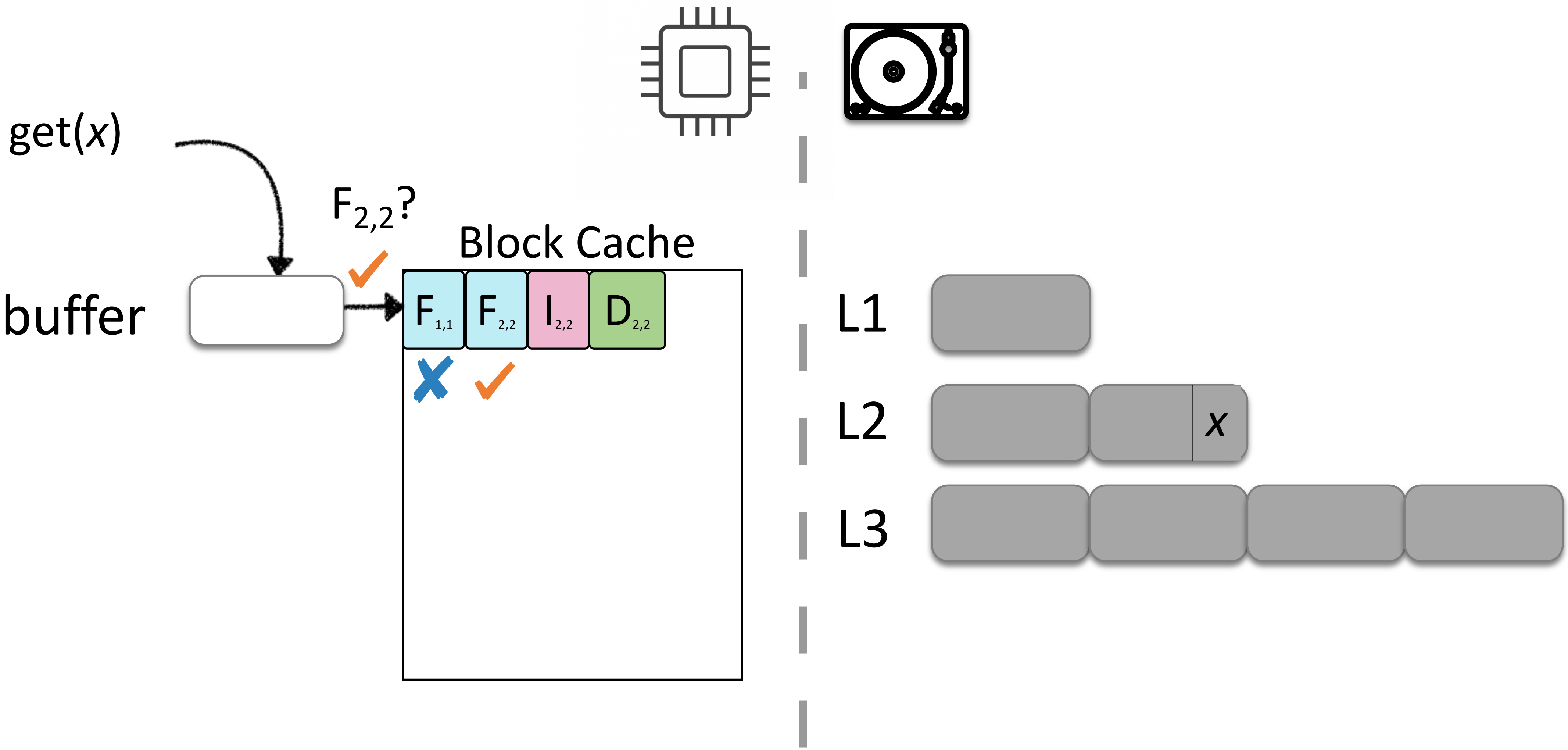
Log-Structured Merge Trees



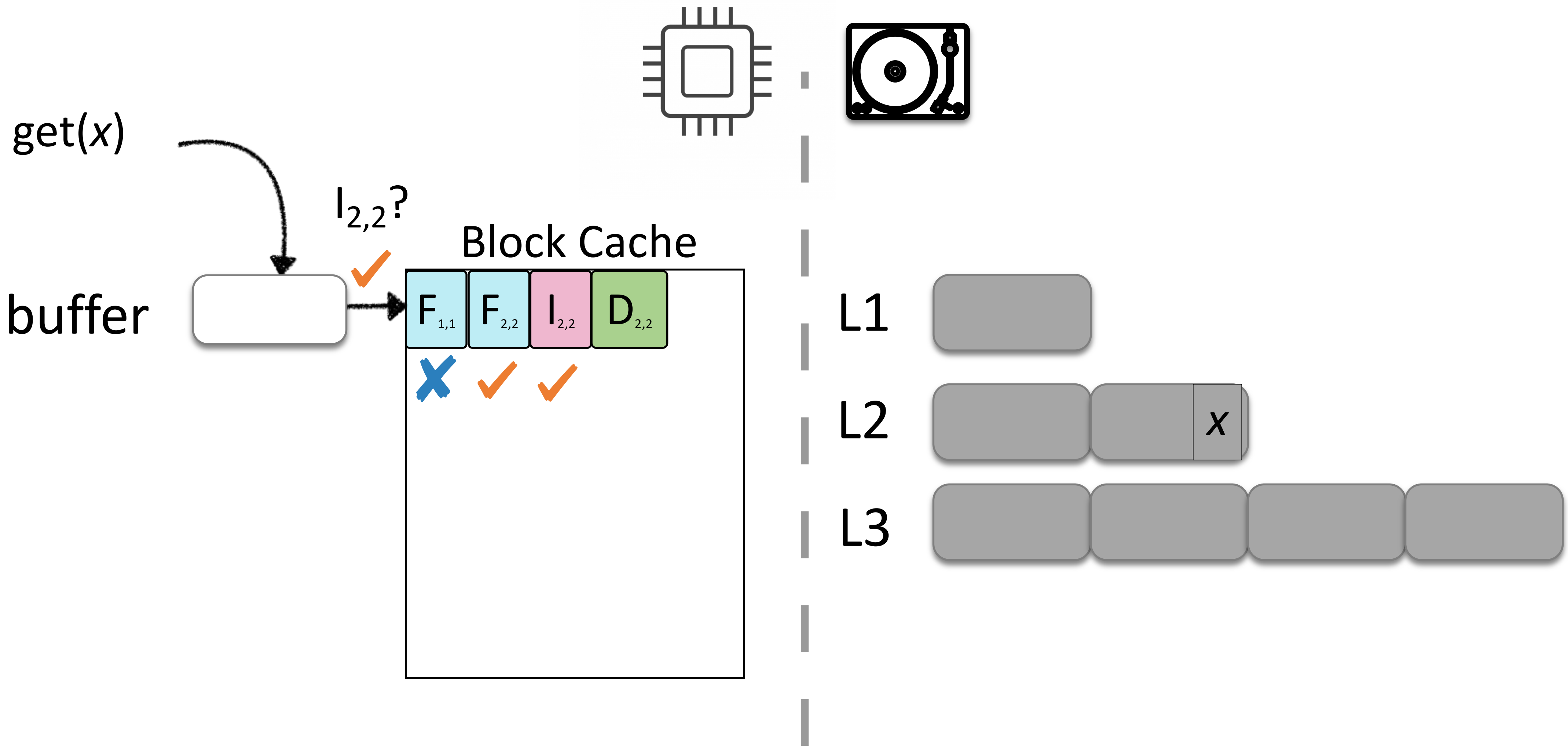
Log-Structured Merge Trees



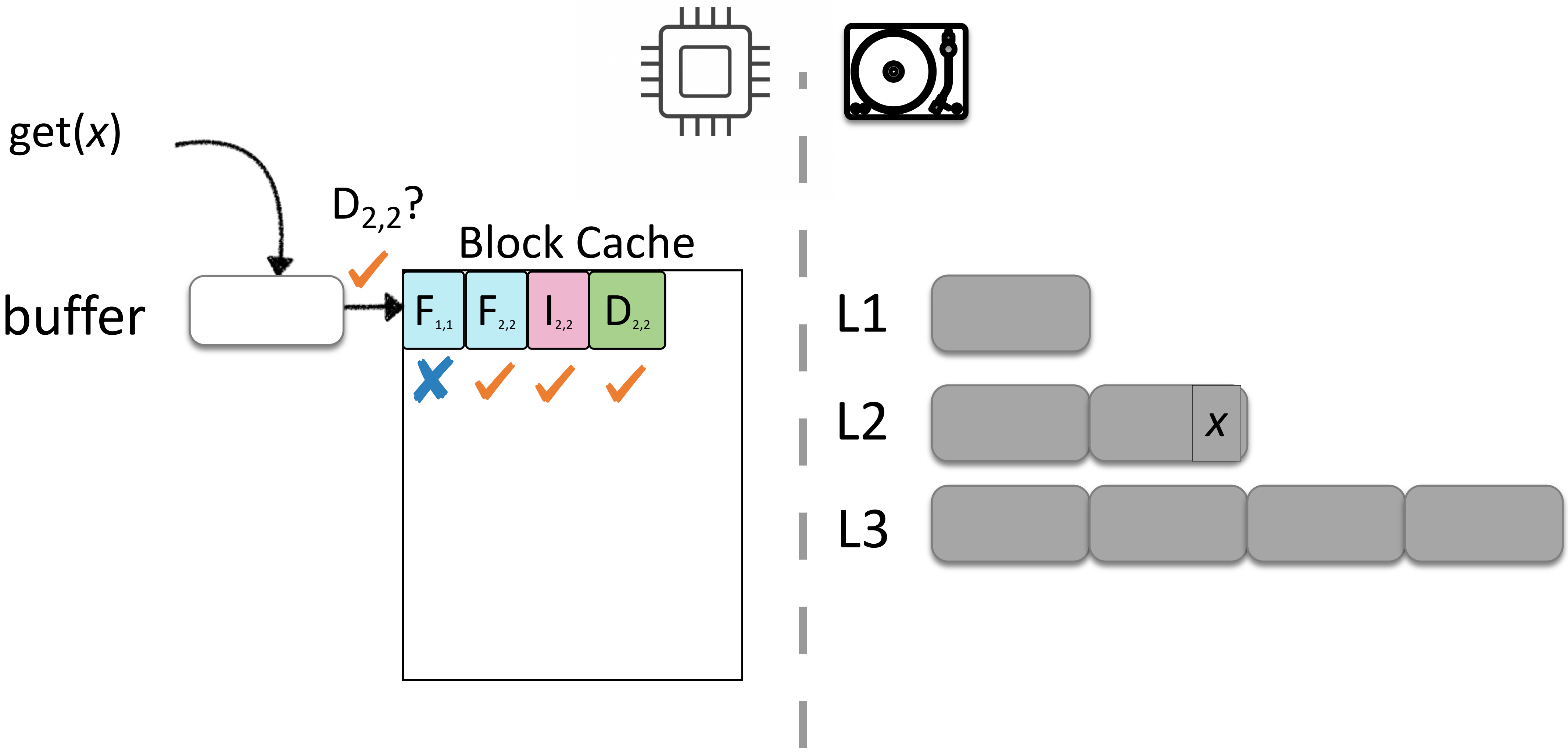
Log-Structured Merge Trees



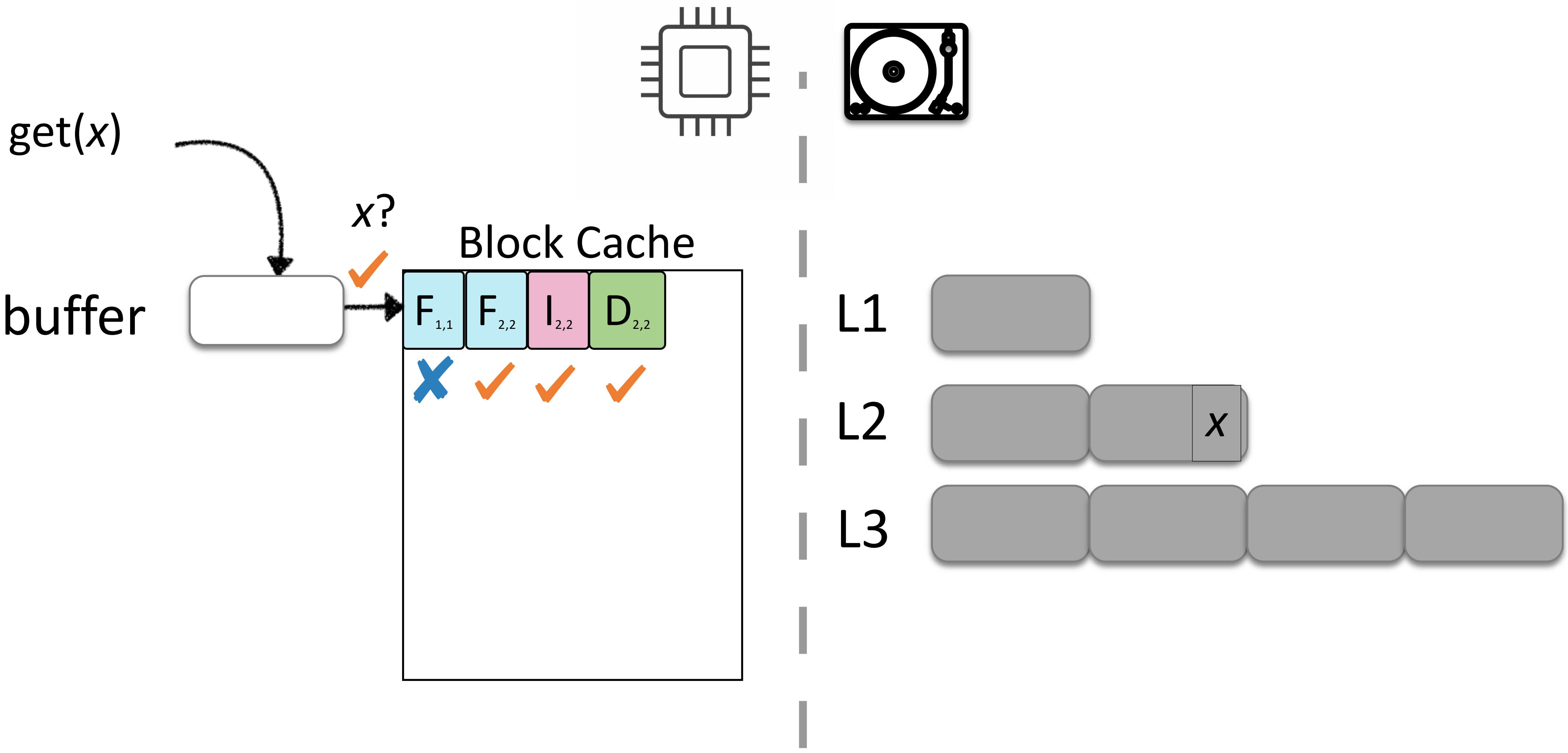
Log-Structured Merge Trees



Log-Structured Merge Trees



Log-Structured Merge Trees



Memory Pressure in LSM-trees

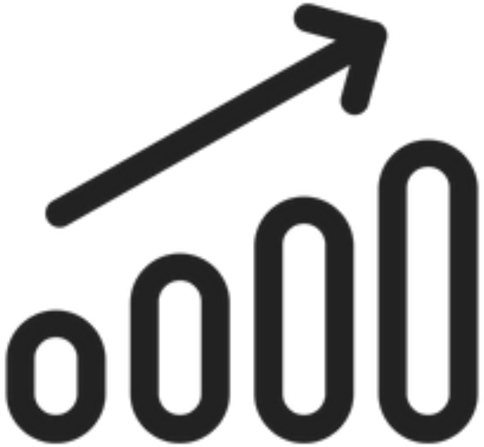
Memory vs. Storage

Metric	DRAM				HDD				SATAFlash SSD	
	1987	1997	2007	2018	1987	1997	2007	2018	2007	2018
Unit price($)	5k	15k	48	80	30k	2k	80	49	1k	415
Unit capacity	1MB	1GB	1GB	16GB	180MB	9GB	250GB	2TB	32GB	800GB
$/MB	5k	14.6	0.05	0.005	83.33	0.22	0.0003	0.00002	0.03	0.0005
Random IOPS	–	–	–	–	5	64	83	200	6.2k	67k (r)/20k (w)
Sequential b/w (MB/s)	–	–	–	–	1	10	300	200	66	500 (r)/460 (w)

The Five-Minute Rule 30 Years Later and Its Impact on the Storage Hierarchy, Communications of the ACM, 2019

The price drop in memory has been slower than storage making it hard to maintain the same memory-to-data ratio

Memory Pressure in LSM-trees



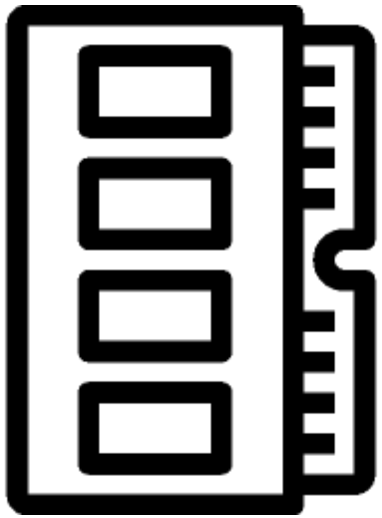
Data size ↑

For 1TB data,
1.3GB filter &17.2GB index

11% space amplification,
1KB entry, 64B key, bpk 10

File size ↑

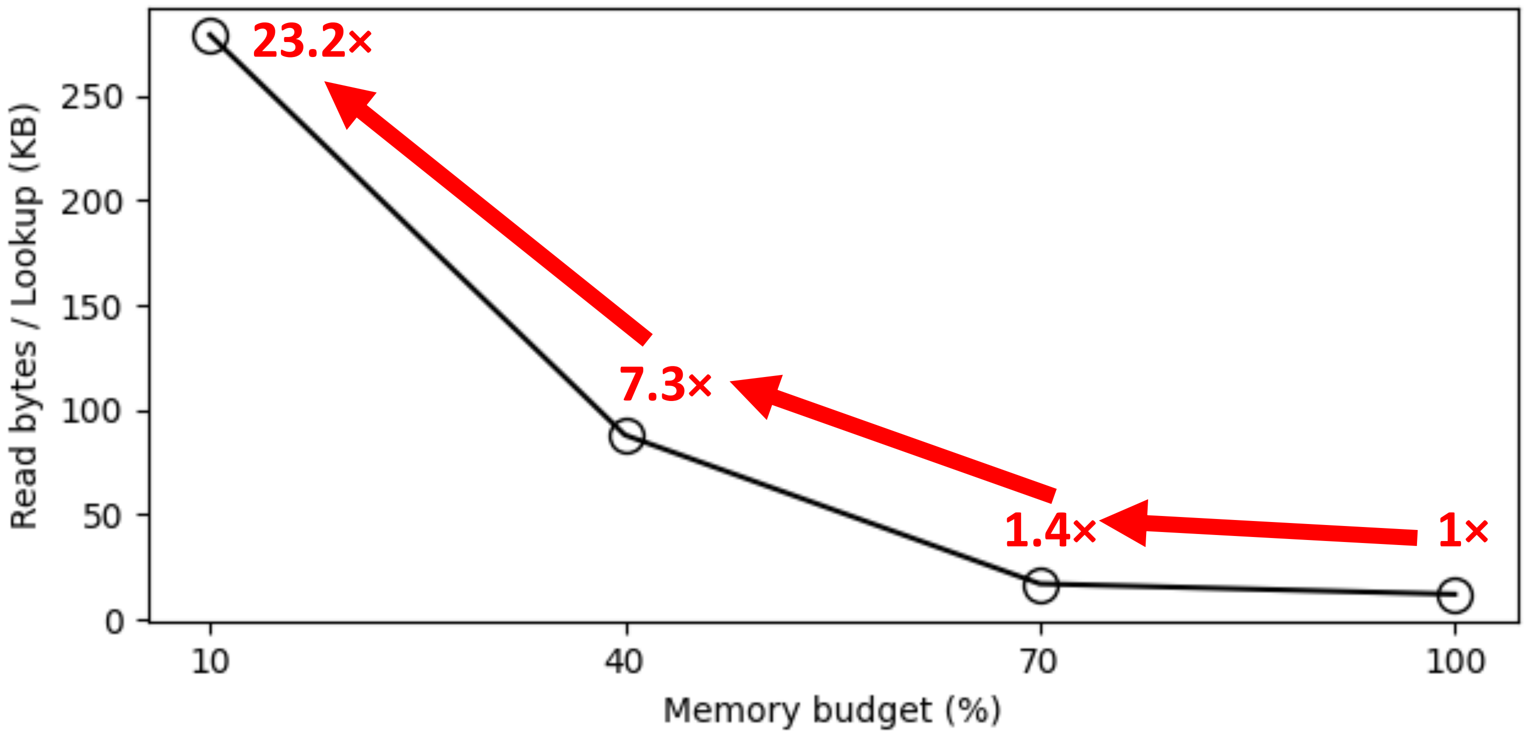
Size of each block increases



memory-to-data ratio ↓

Memory pressure

Lookup cost under memory pressure

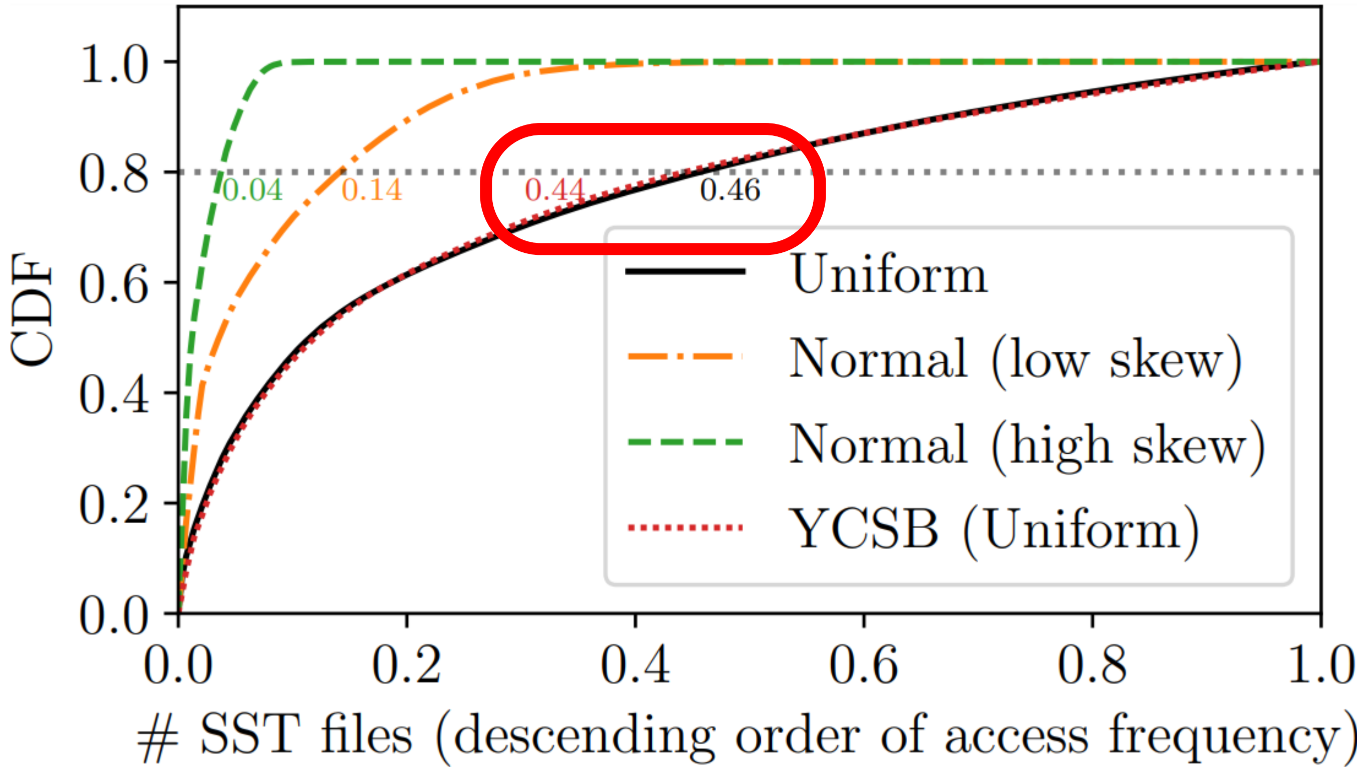


As the available memory decreases, the read bytes per query increase rapidly.

Are all filter blocks equally important?

State-of-the-art LSM designs treat all BFs equally

Access Frequency Patterns

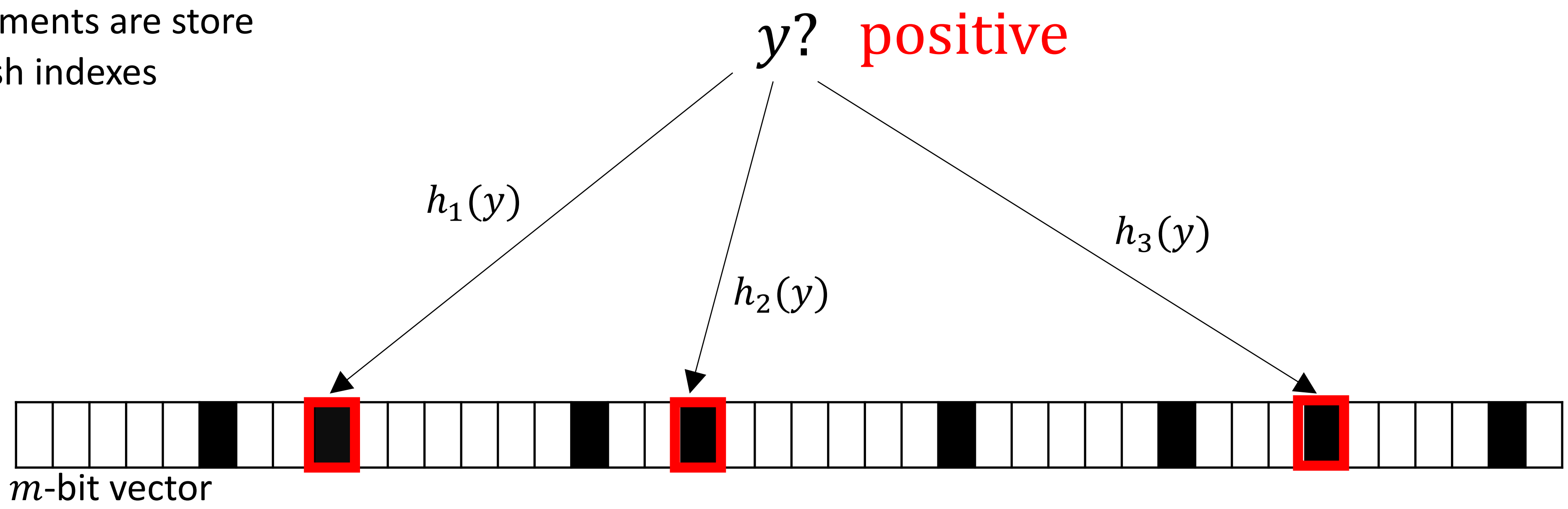


Even in a perfectly uniform workload,

<u>80% of the lookups</u> *are directed to* ***<u>44~46% of the SST files</u>***

Bloom Filter

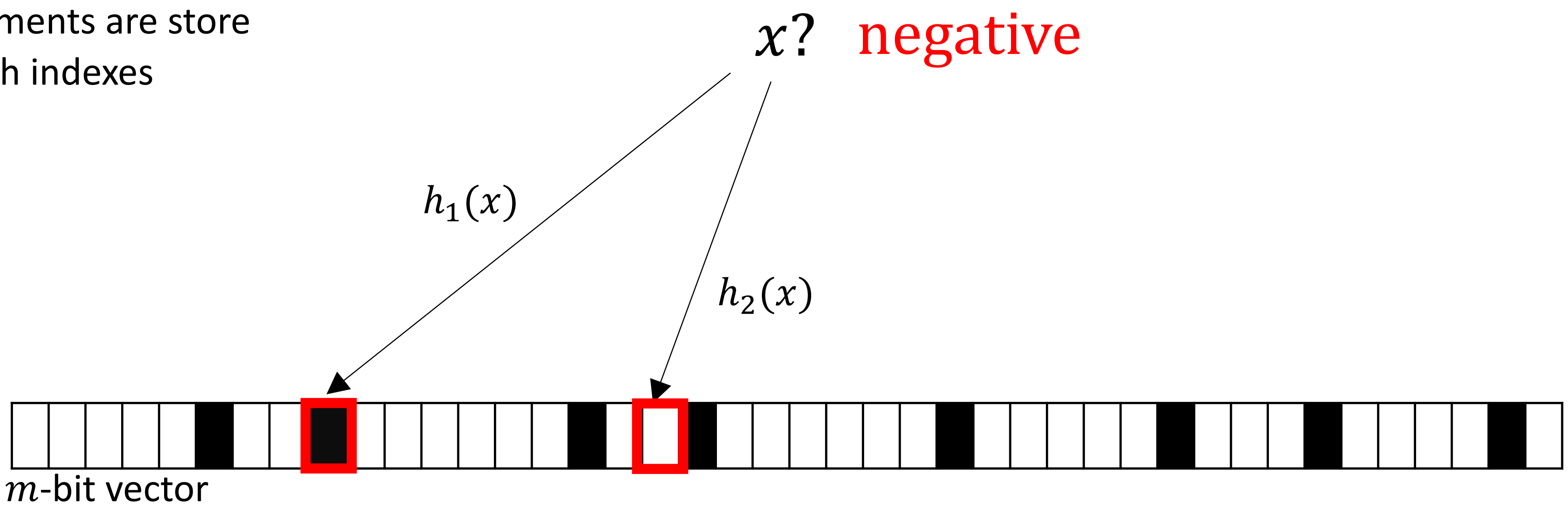
m-bit vector
n elements are store
k hash indexes



Always access all k indexes for positive queries

Bloom Filter

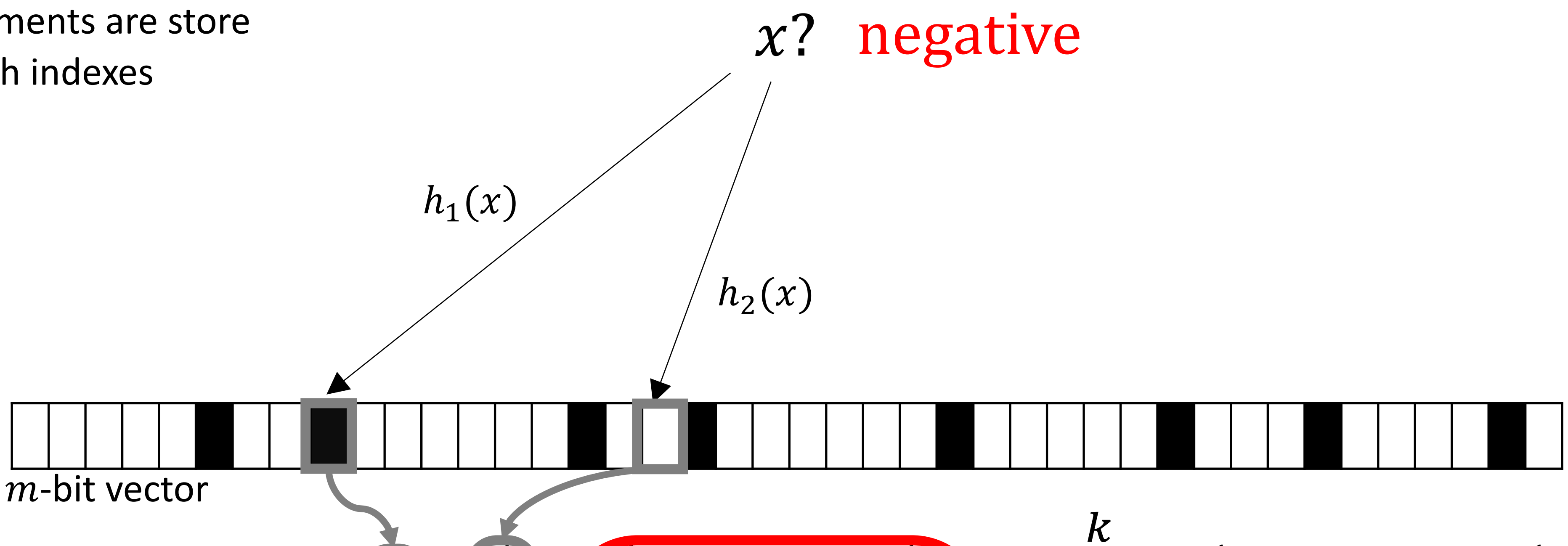
m-bit vector
n elements are store
k hash indexes



Is the entire filter useful?

Bloom Filter

m-bit vector
n elements are store
k hash indexes

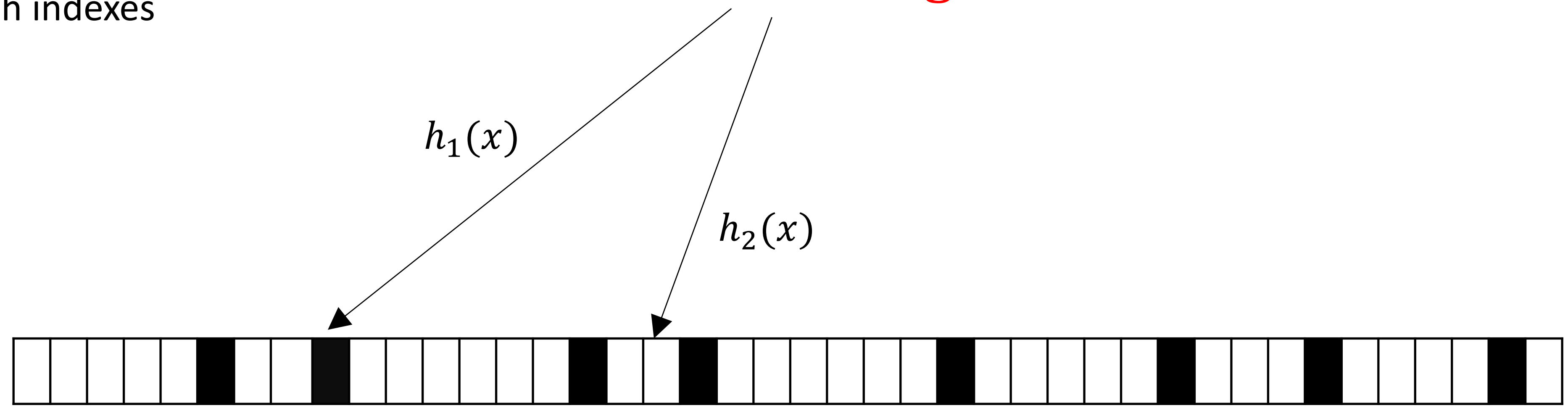


$$probes_{empty} = 1 + \frac{1}{2} + \underbrace{\frac{1}{2^2} + \cdots + \frac{1}{2^{k-1}}}_{\text{for all } k \text{ hash indexes}} = \sum_{d=1}^{k} \frac{1}{2^{k-1}} = 2 - \frac{1}{2^{k-1}}$$

Bloom Filter

m-bit vector
n elements are store
k hash indexes

x? negative

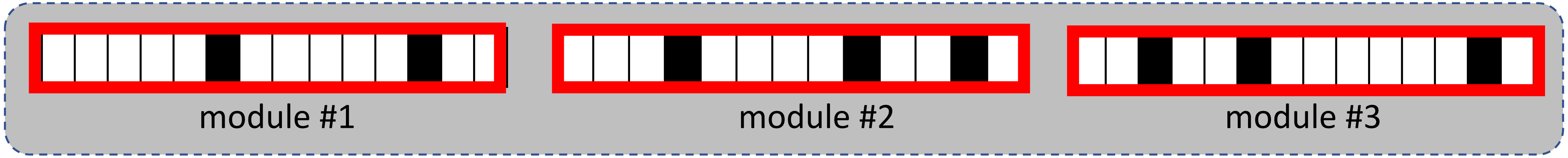


m-bit vector

$$probes_{empty} = 1 + \frac{1}{2} + \frac{1}{2^2} + \cdots + \frac{1}{2^{k-1}} = \sum_{d=1}^{k} \frac{1}{2^{k-1}} = 2 - \frac{1}{2^{k-1}}$$

Modular Bloom Filter

m-bit vector
n elements are store
k hash indexes
d modules

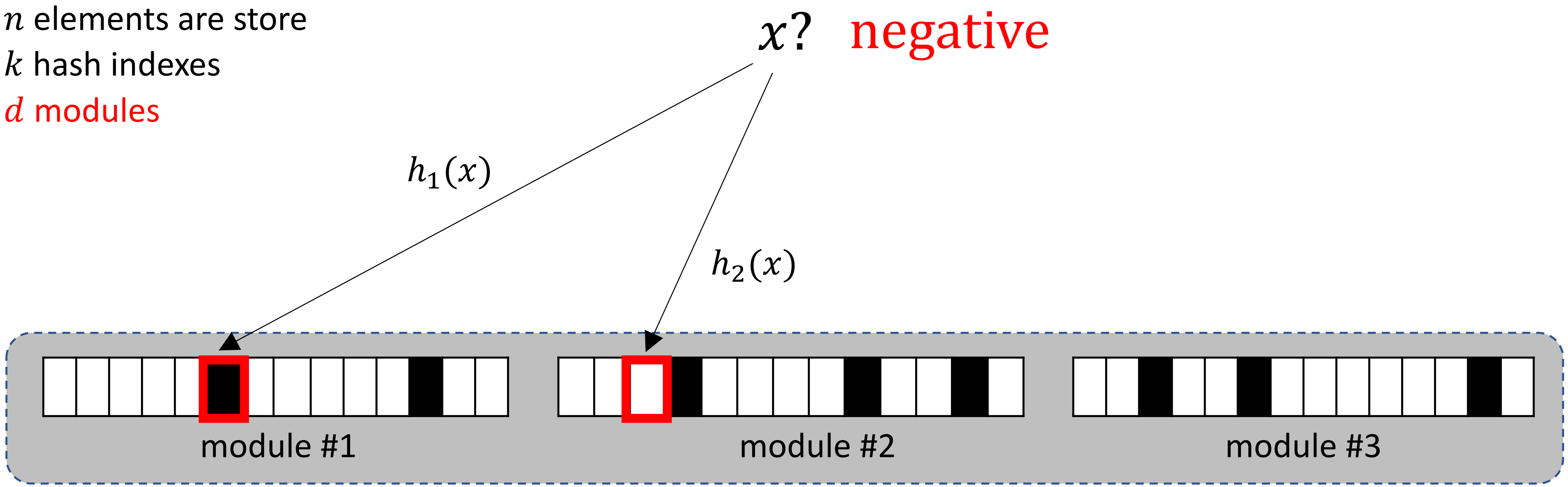


An MBF is a collection of D Bloom filters

- m_d-bit vector
- n elements
- k_d hash indexes

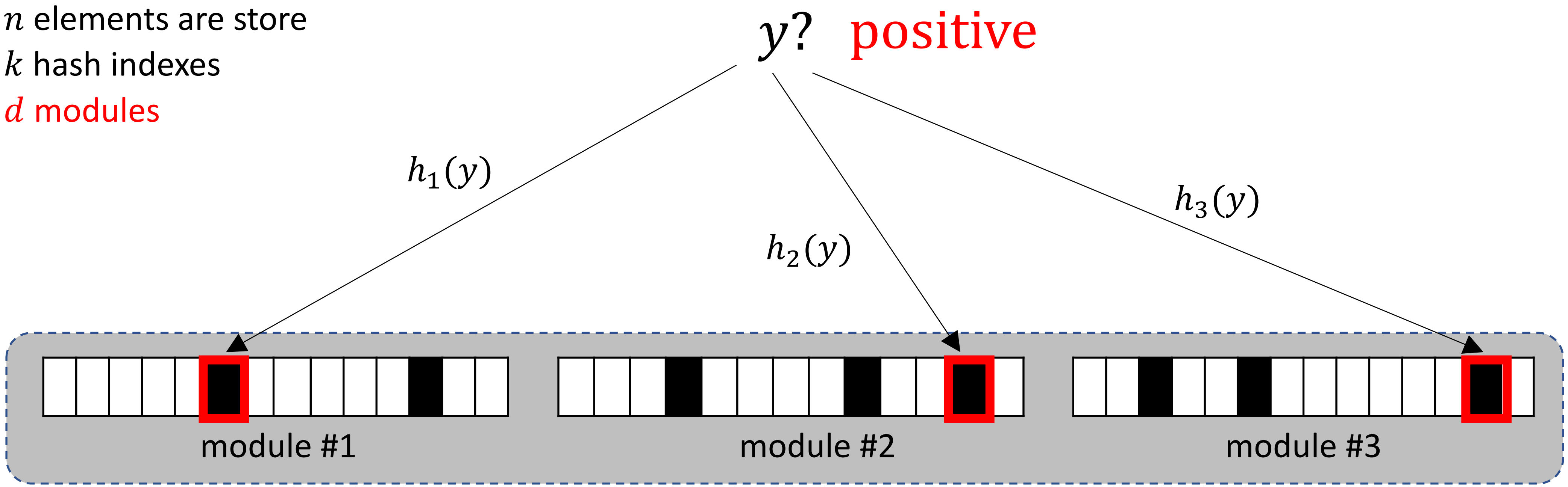
Modular Bloom Filter

m-bit vector
n elements are store
k hash indexes
d modules



Modular Bloom Filter

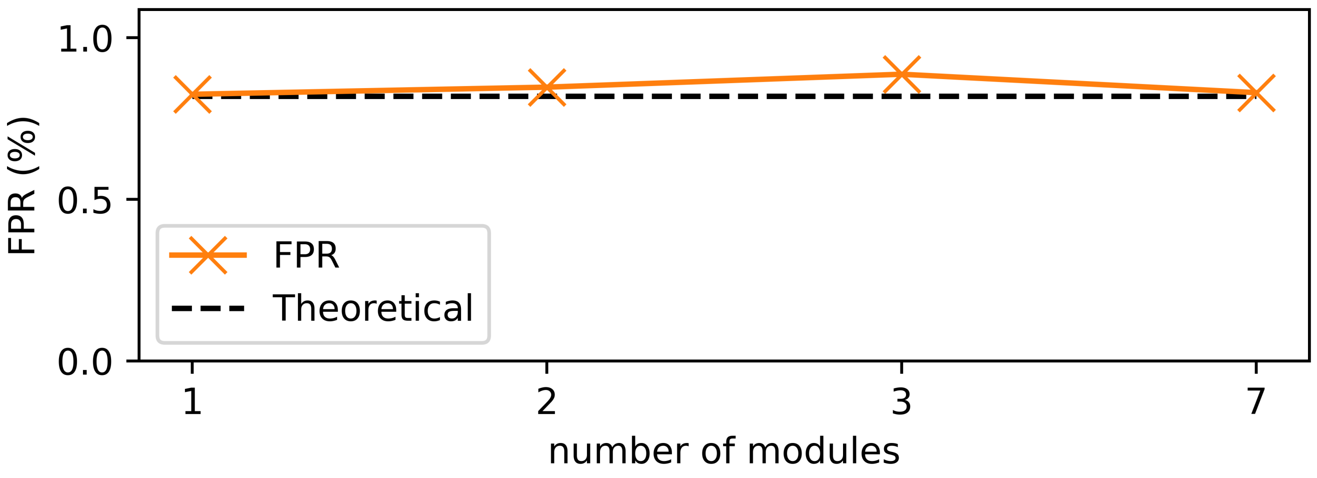
m-bit vector
n elements are store
k hash indexes
d modules



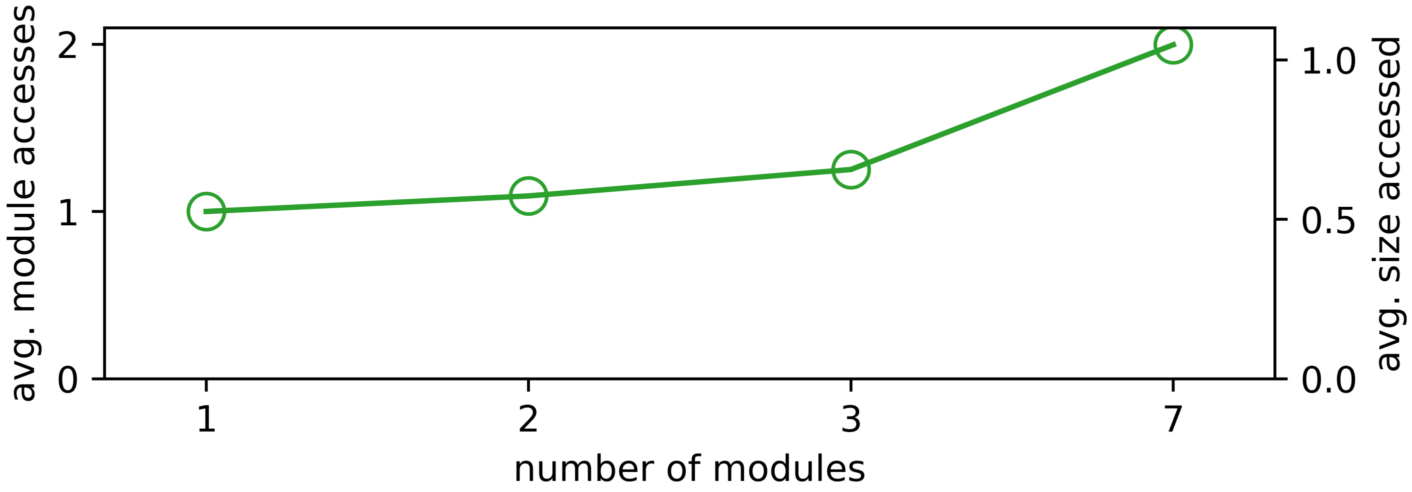
Modular Bloom Filter

False positive rate

FPR close-to-theoretical



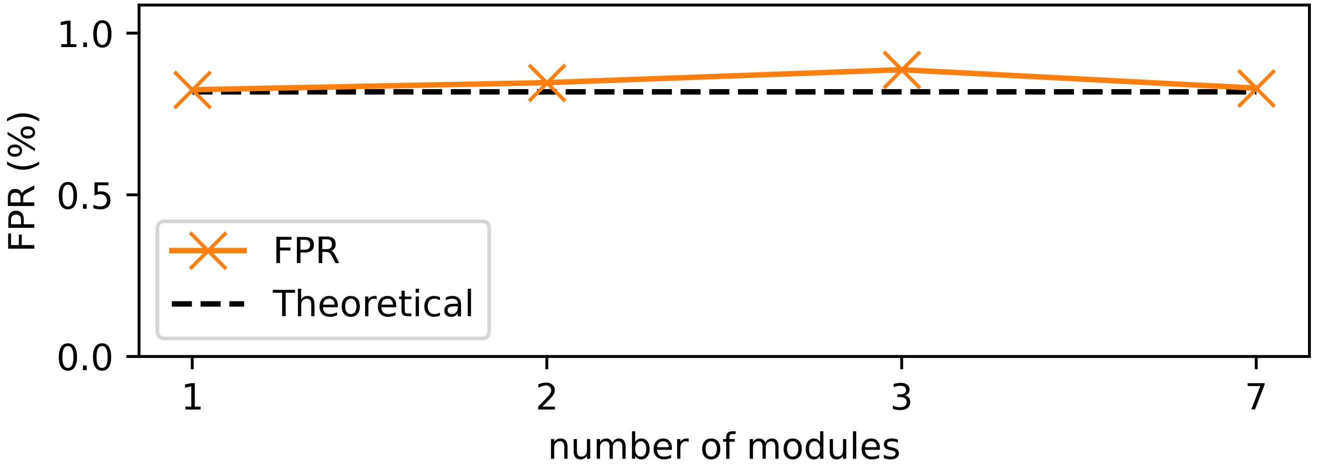
Avg. number of module accesses



Modular Bloom Filter

False positive rate

FPR close-to-theoretical

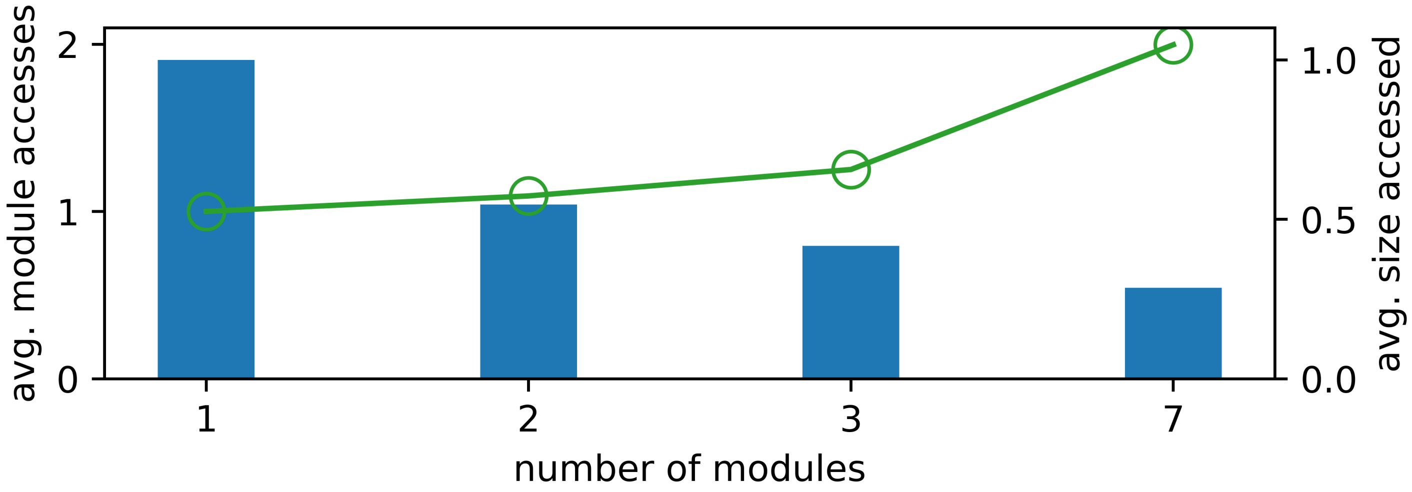


Avg. # of module accesses

vs.

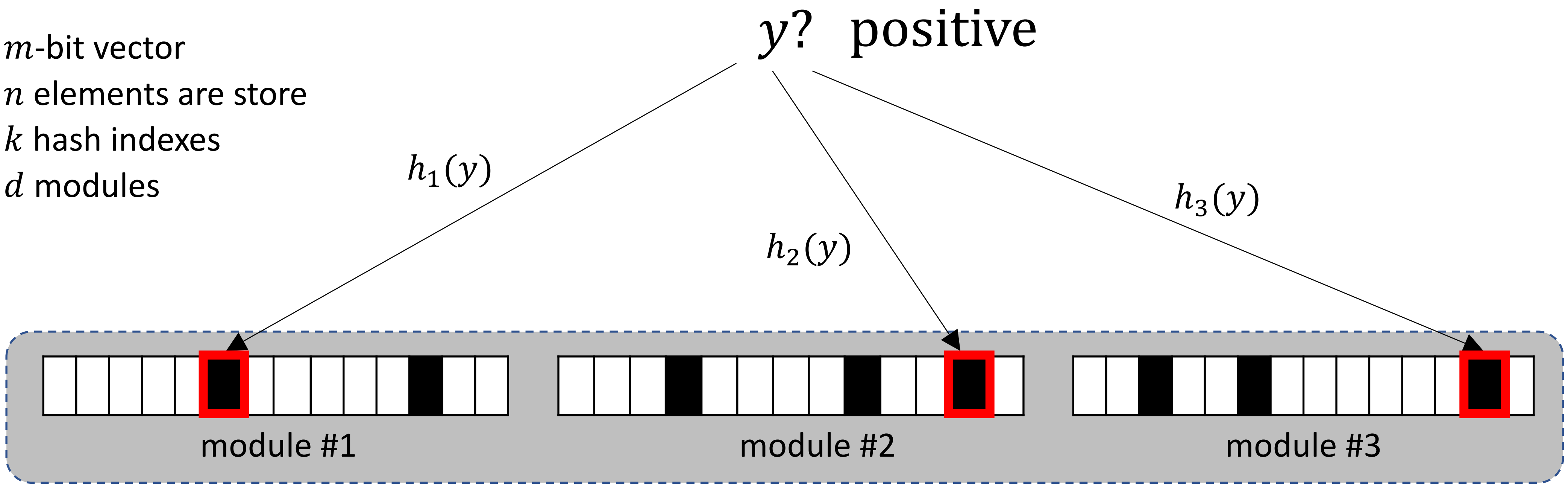
Avg. size accessed

Less space requirement



Modular Bloom Filter

m-bit vector
n elements are store
k hash indexes
d modules



MBFs are not useful for non-empty queries.

What if we know the queries would be non-empty in advance?

Skipping Modules

Utility : a measure of the benefit of a filter or a module

$$u_{l,i,d} = expIO_{l,i,d} - expIO_{l,i,d-1}$$

The expected number of I/Os that can be reduced by using d-th module

Expected number of I/Os

$$expIO_{l,i,d} = \underbrace{\beta_{l,i}}_{\text{access frequency}} \cdot \left(\underbrace{\alpha_{l,i}}_{\text{non-empty queries}} + \underbrace{(1 - \alpha_{l,i}) \cdot f_{sm}^{d}}_{\text{false positives from empty queries}}\right)$$

l-th level
i-th SST file
d-th module
f_{sm}: FPR of a single module

Utility is high if file is frequently accessed, or queries are empty

Skipping Modules

***Skipping Modules** based on their utilities*

$u_{l,i,d}$ = expIO(l,i,d) - expIO(l,i,d-1) // calc module's utility

if $u_{l,i,d} < threshold_d$ **then** // skip module when there's no benefit

    return *true*

else // otherwise, keep querying modules

    result = QueryModule(key, $module_{l,i,d}$)

Modular Bloom filter
&
Skipping Algorithm
&
Sharing Hashing

+ LSM-tree

Sharing Hashing with Modular Bloom filters (SHaMBa)

Experimental Evaluation

Experiment Settings

LSM-tree tuning

Term	Value	Explanation
E	64	entry size (B)
K	32	key size (B)
B	64	block size (#entries)
P	1024	buffer size/file size (#blocks)
T	4	size ratio
b	10	bits per key for filters

Size of blocks

Term	Value	Explanation
S_D	4	data block size (KB)
S_I	32	index block size (KB)
S_F	80	filter block size (KB)

Approaches Tested

Tuning knobs of SHaMBa

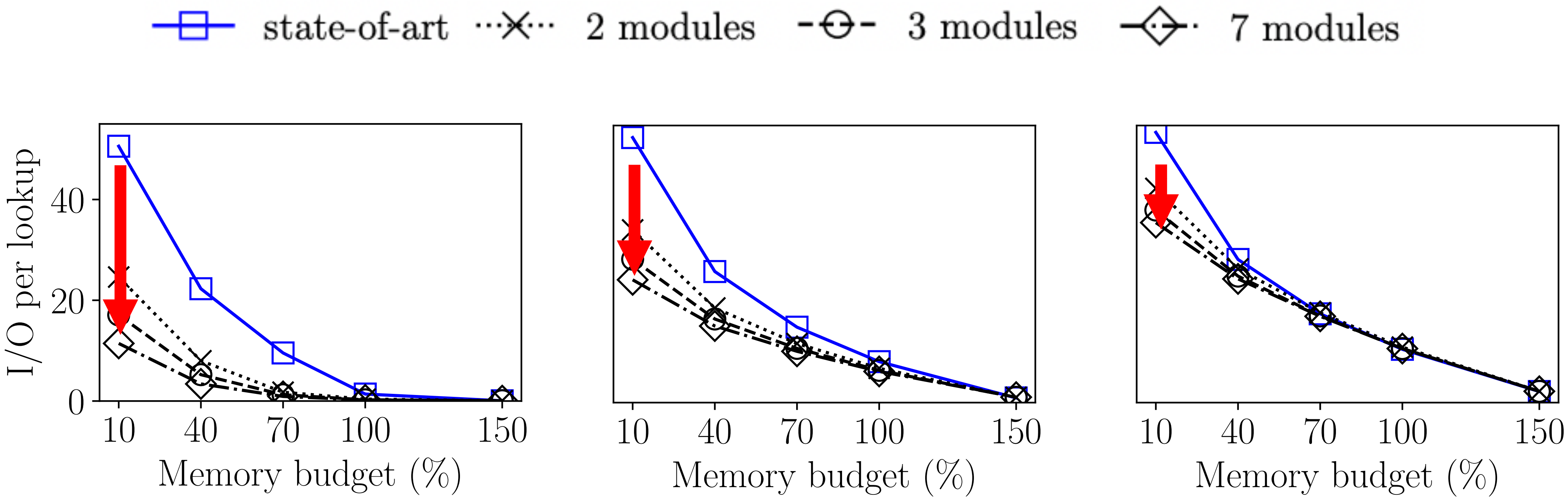
Term	Value
number of modules	1, ***2***, 3, or 7
Size of each module	***equal*** or proportional
skipping algorithm	none, partial ($\mathcal{P}$), or ***full ($\mathcal{F}$)***

Approaches Tested

- *state-of-the-art*
- *SHaMBa-eq*
- *SHaMBa-eq-$\mathcal{P}$*
- *SHaMBa-eq-$\mathcal{F}$*
- *SHaMBa-prop*
- *SHaMBa-prop-$\mathcal{P}$*
- *SHaMBa-prop-$\mathcal{F}$*

Impact of number of modules

Workload: Uniform, Entry size: 64B, #Entries: 30K
Tuning: no skipping algorithm, equal sized modules



all empty

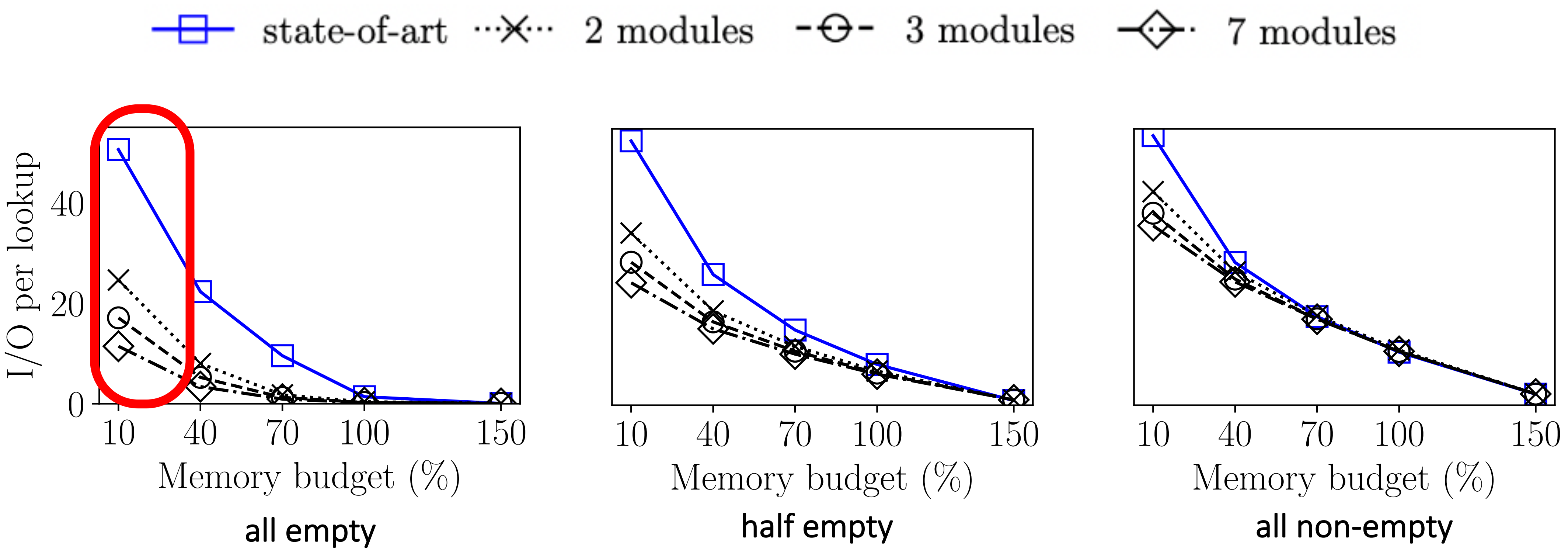
half empty

all non-empty

SHaMBa enhances the lookup performance for empty queries

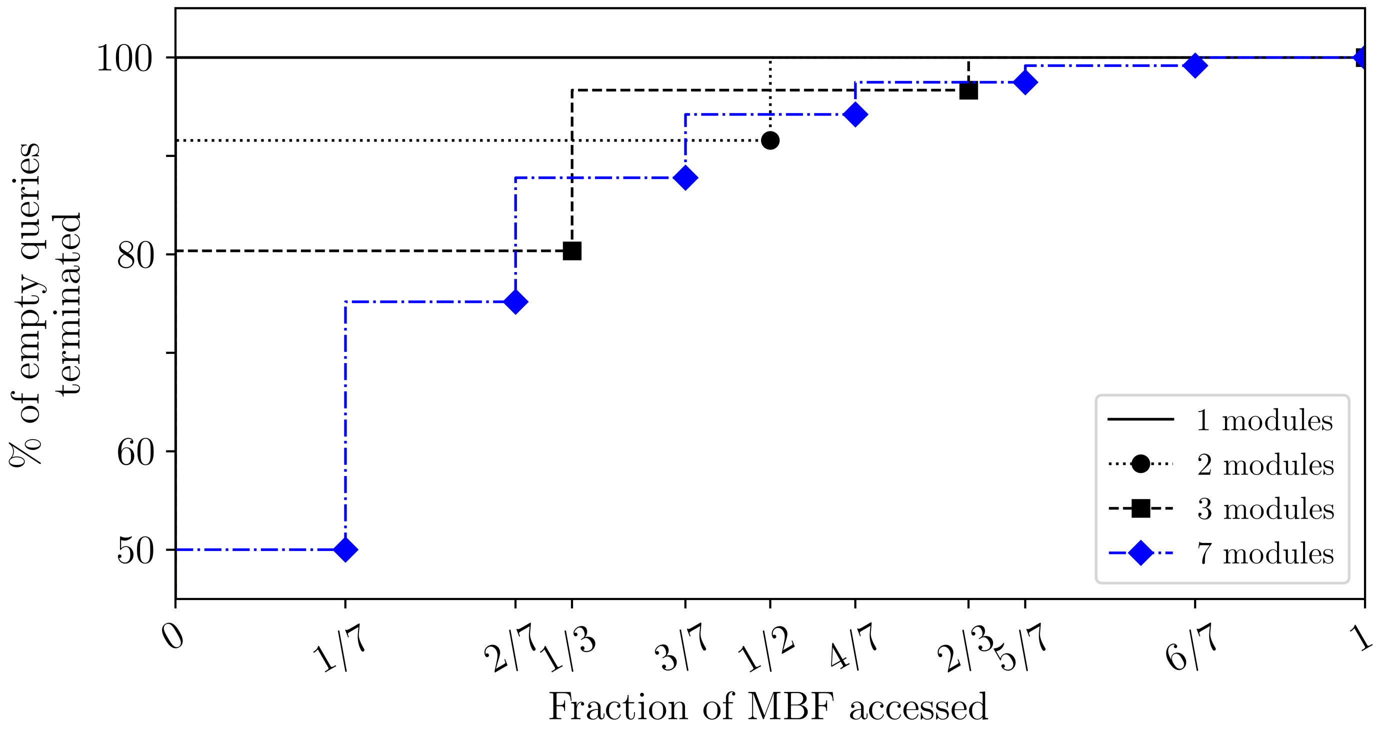
Impact of number of modules

Workload: Uniform, Entry size: 64B, #Entries: 30K
Tuning: no skipping algorithm, equal sized modules



SHaMBa Performs Best with Smaller Modules

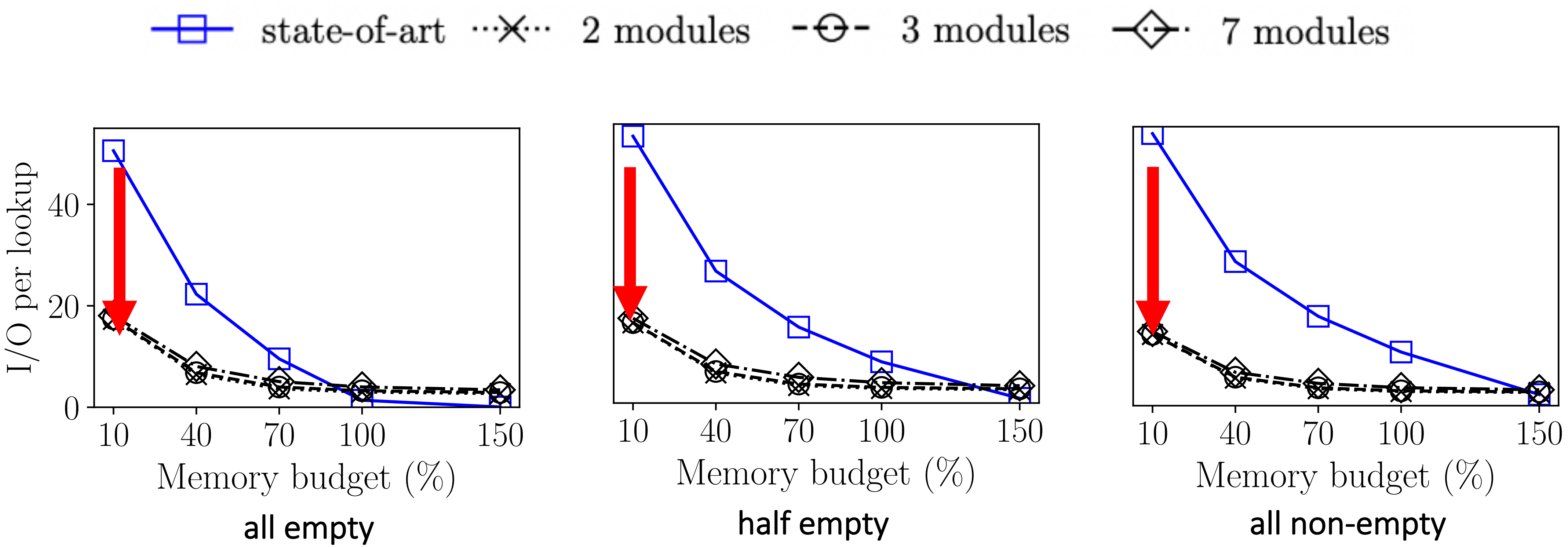
Impact of number of modules



Smaller modules are more beneficial

Impact of number of modules

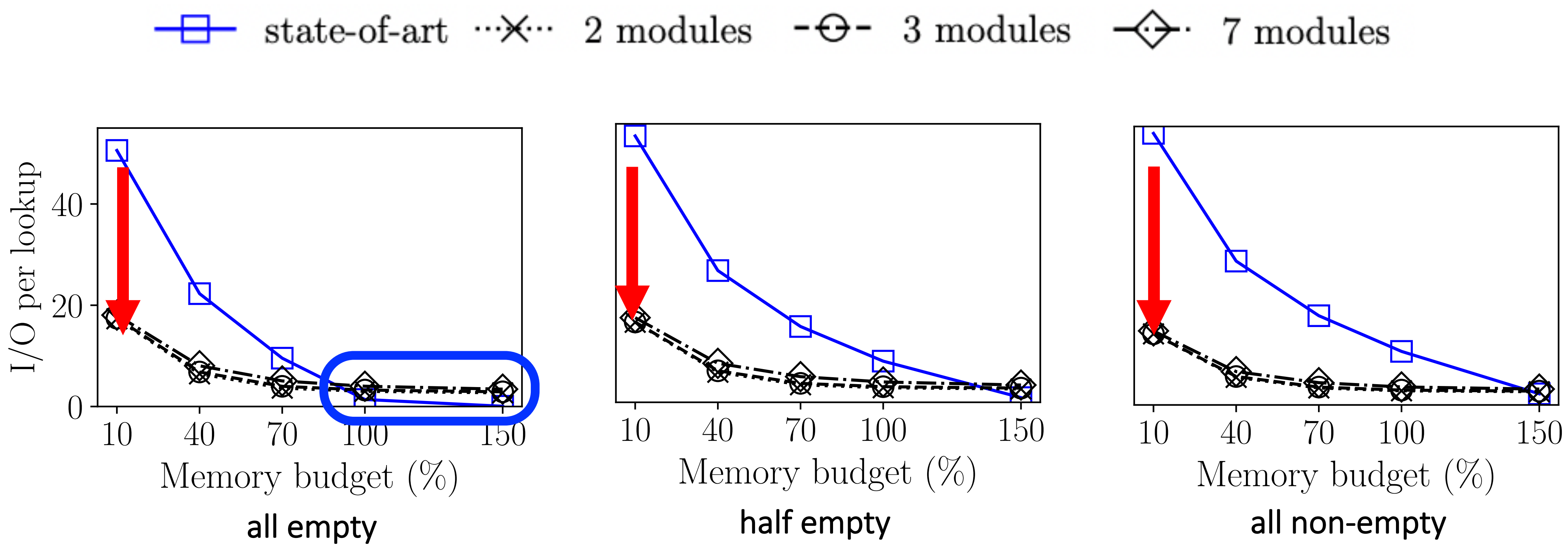
Workload: Uniform, Entry size: 64B, #Entries: 30K
Tuning: full skipping algorithm, equal sized modules



Skipping modules reduces the impact of the number of the modules

Impact of number of modules

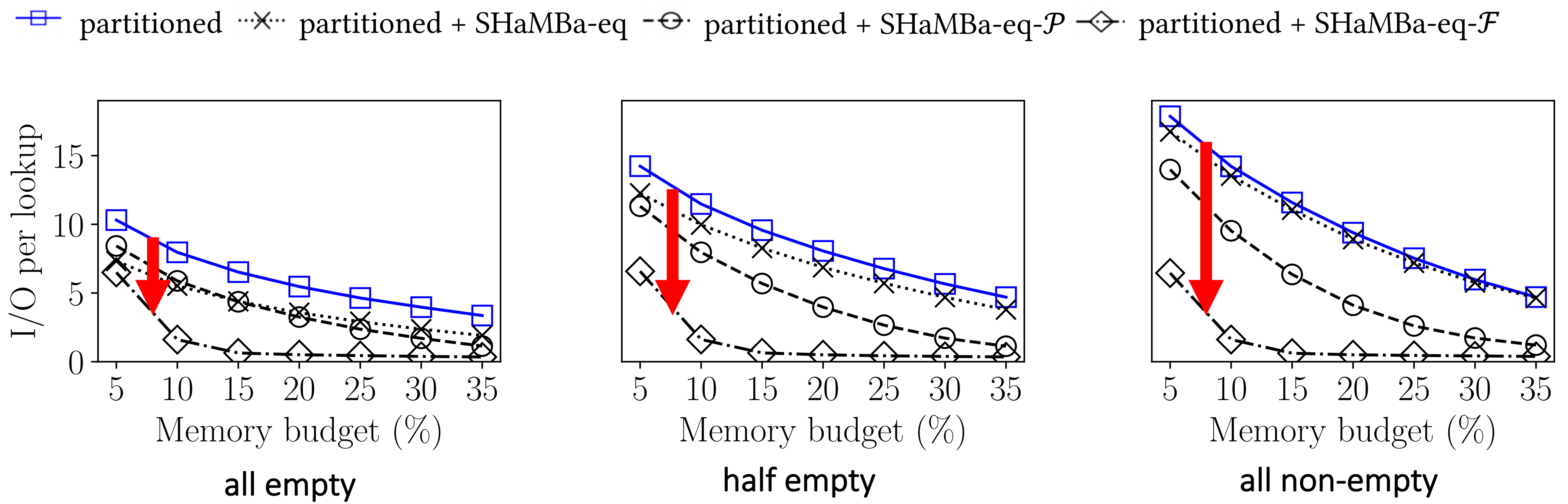
Workload: Uniform, Entry size: 64B, #Entries: 30K
Tuning: full skipping algorithm, equal sized modules



Skipping modules reduces the impact of the number of the modules

SHaMBa with Partitioned Index/Filter

Workload: Uniform, Entry size: 64B, #Entries: 30K
Tuning: 2 equal sized modules



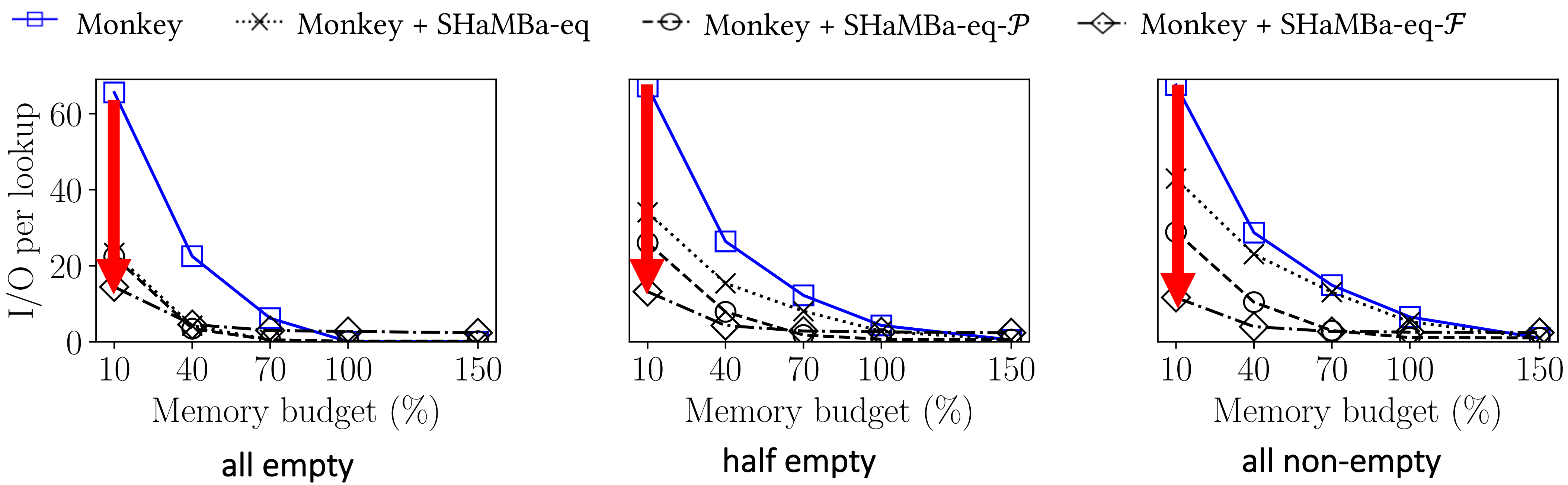
SHaMBa boosts partitioned index/filter under severe memory pressure

SHaMBa with Monkey

Workload: Uniform, Entry size: 64B, #Entries: 30K
Tuning: 2 equal sized modules

Monkey allocates more bits per element in the shallower levels to aggressively reduce their false positives

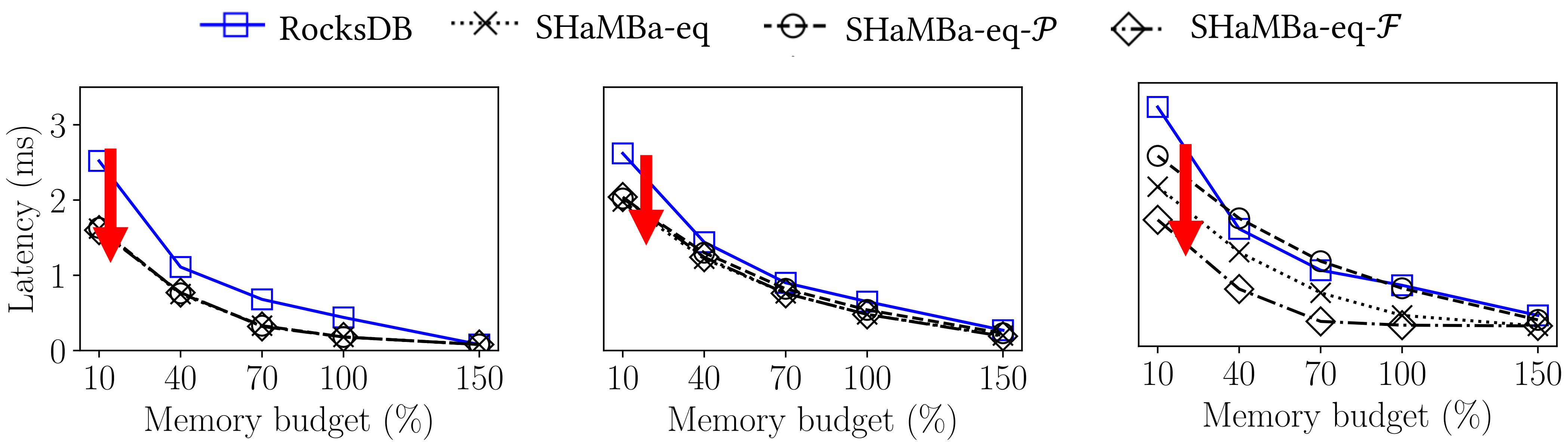
Monkey: Optimal Navigable Key-Value Store, ACM SIGMOD 2017



SHaMBa further improves performance of Monkey

SHaMBa-eq with RocksDB

Workload: Uniform, Entry size: 64B, #Entries: 30K
Tuning: 2 equal sized modules, RocksDB version 6.19.3



all empty

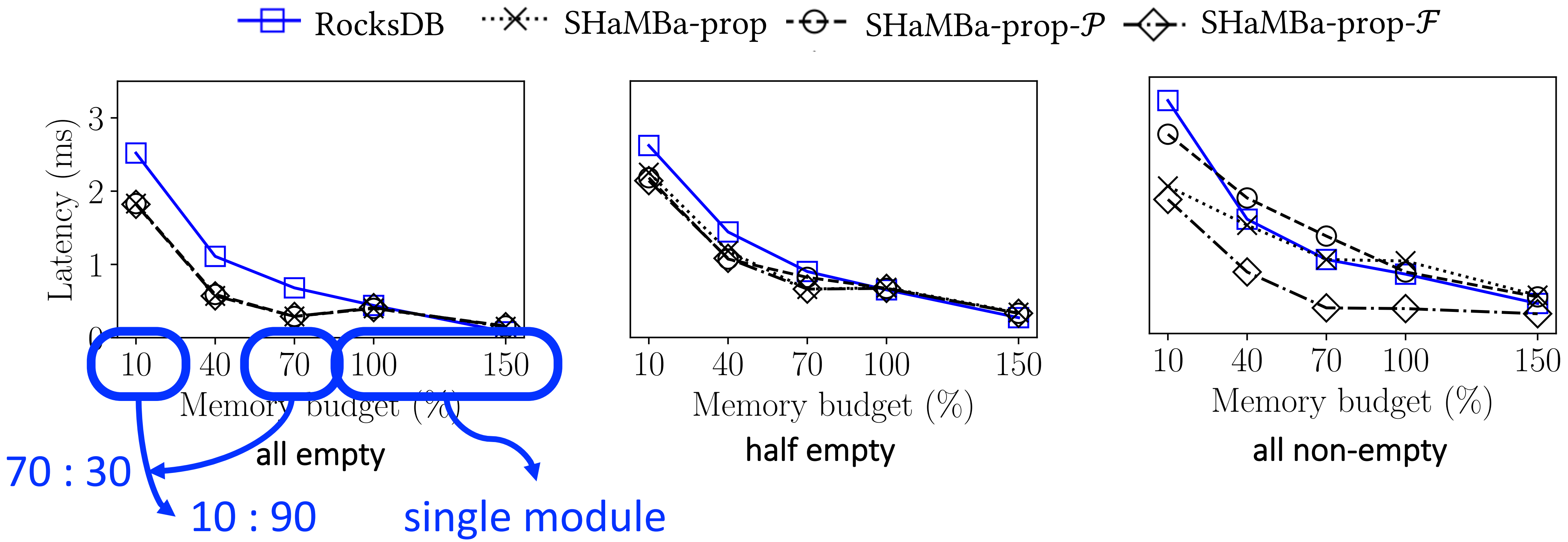
half empty

all non-empty

SHaMBa-eq accelerates point lookups

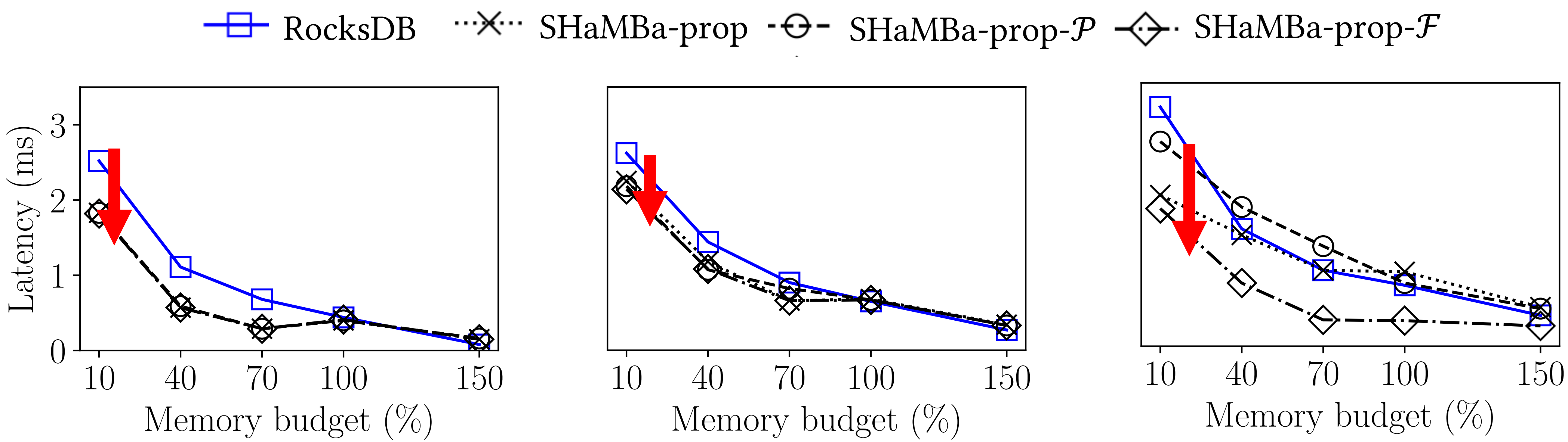
SHaMBa-prop with RocksDB

Workload: Uniform, Entry size: 64B, #Entries: 30K
Tuning: 2 proportional sized modules, RocksDB version 6.19.3



SHaMBa-prop with RocksDB

Workload: Uniform, Entry size: 64B, #Entries: 30K
Tuning: 2 proportional sized modules, RocksDB version 6.19.3



all empty

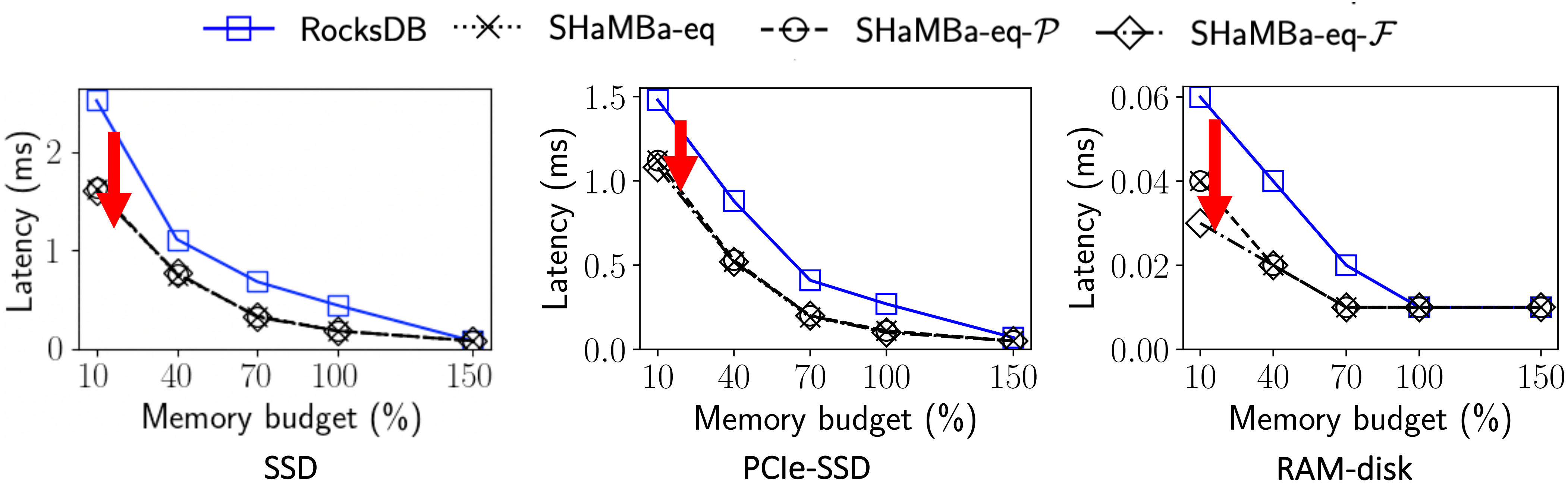
half empty

all non-empty

SHaMBa-prop accelerates point lookups

SHaMBa on various Devices

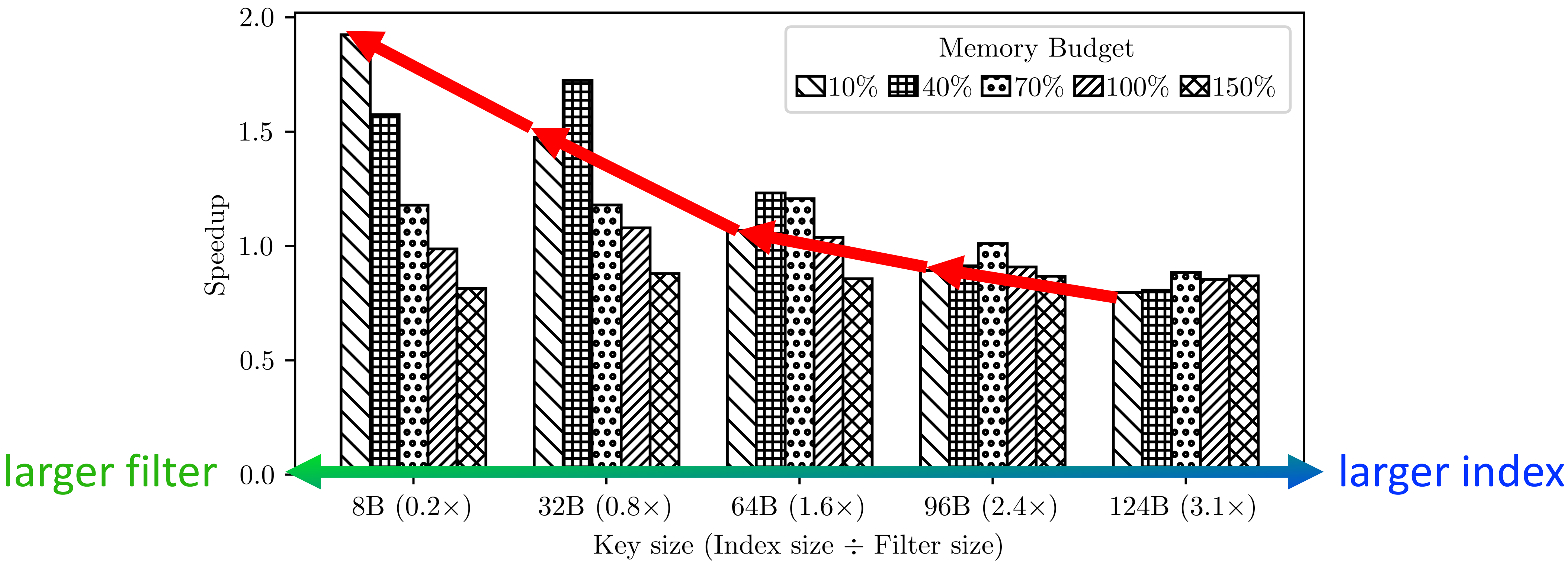
Workload: Uniform, Entry size: 64B, #Entries: 30K
Tuning: 2 equal sized modules, RocksDB version 6.19.3



SHaMBa also benefits faster storage devices

SHaMBa with larger index

Workload: Uniform (all empty), Entry size: 128B, #Entries: 30K
Tuning: 2 equal sized modules, RocksDB version 6.19.3



SHaMBa performs best when filters are larger than indexes

Conclusion

- Modular Bloom filters (MBFs)
  - a BF variant that consists of multiple module
  - enable smooth navigation of the memory vs. performance trade-off

- SHaMBa
  - a novel LSM-based key-value engine
  - specifically addresses performance loss due to memory pressure
  - the same average number of I/Os, with 1/3 of the memory by the state of the art

Thank you!