LSM-Trees Under Memory Pressure

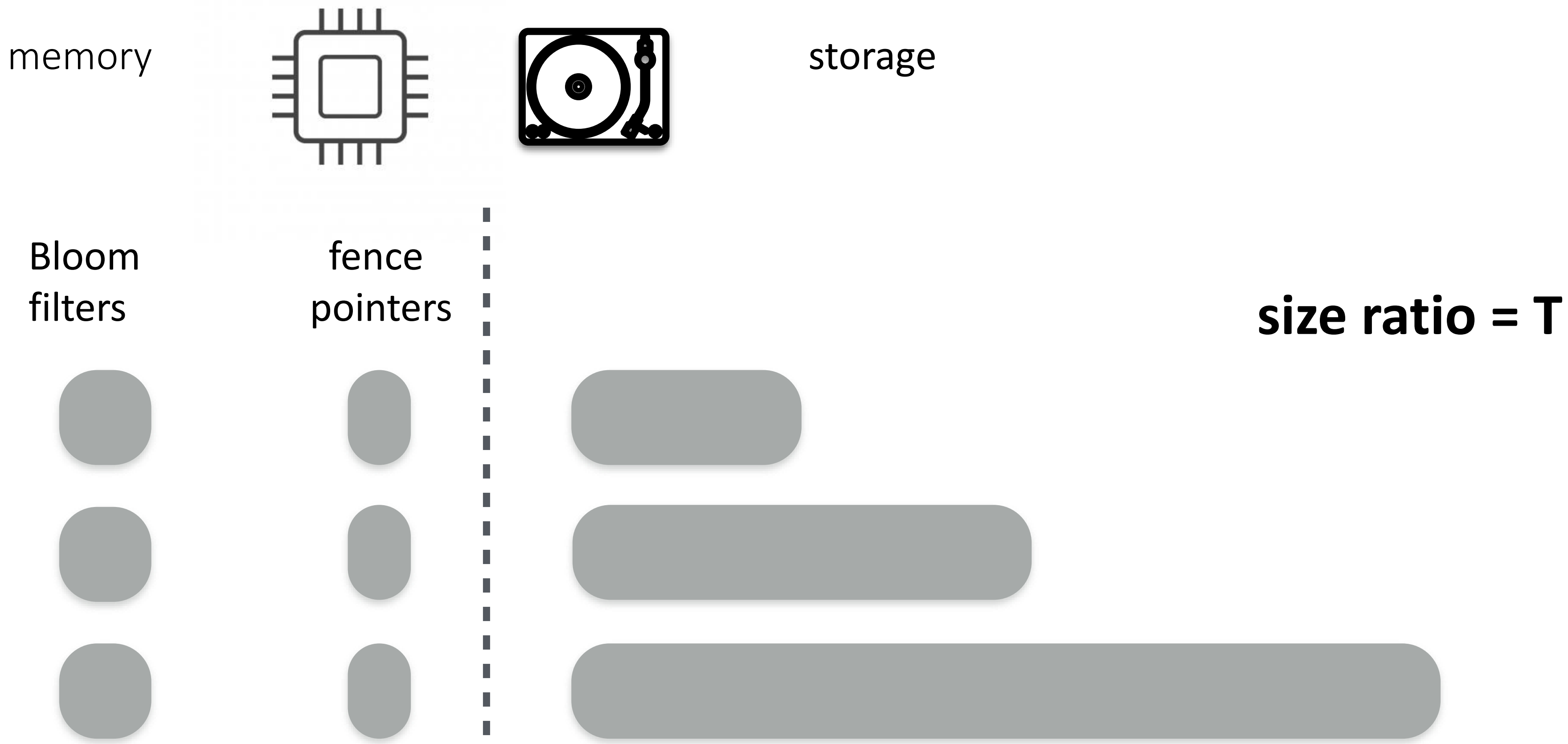
Presenter: Zichen Zhu

Slides credited to ***Juhyoung Mun*** and ***Manos Athanassoulis***

Log-Structured Merge Trees

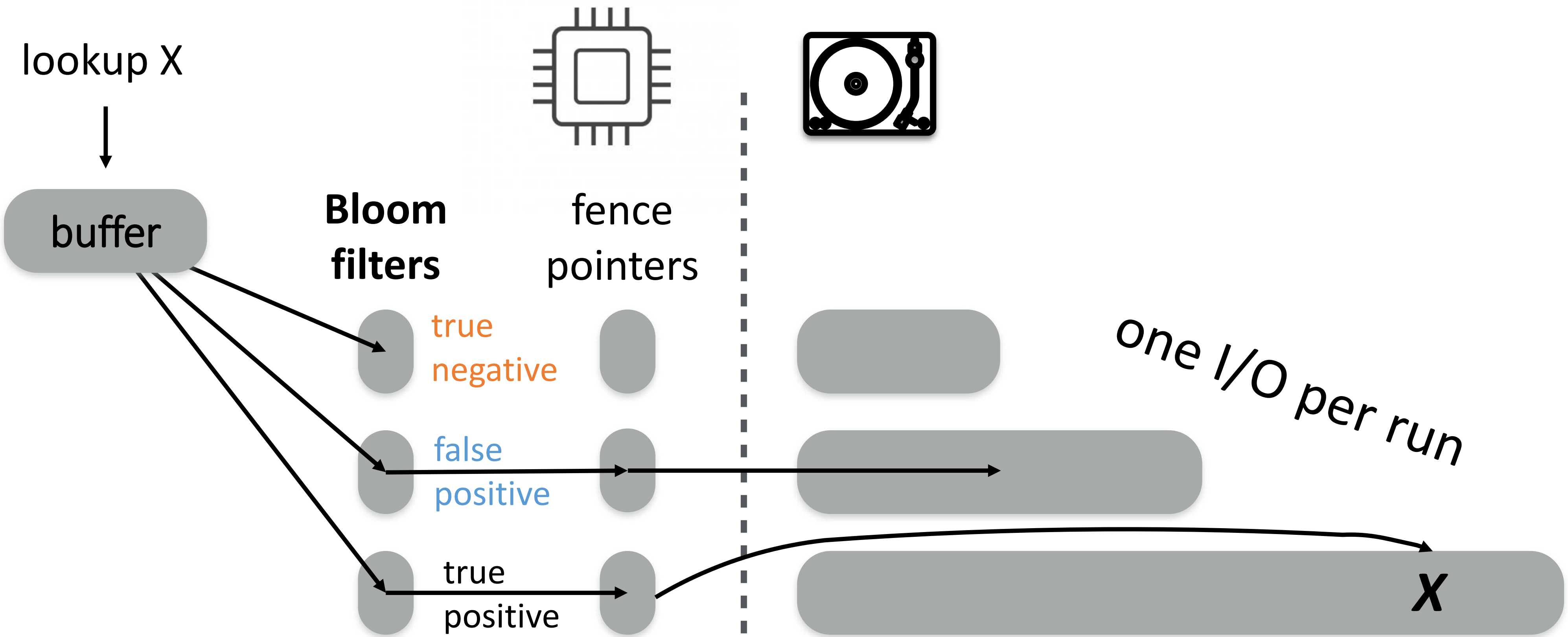
Widely adopted because they balance read performance and ingestion

Log-Structured Merge Trees



exponentially larger capacity

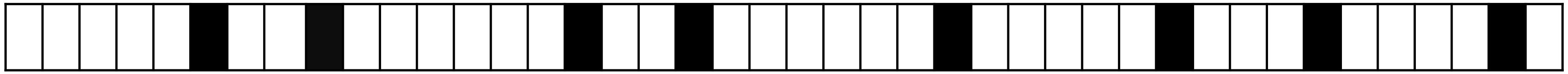
Log-Structured Merge Trees



Bloom Filter

m-bit vector
n elements are store
k hash indexes

y?



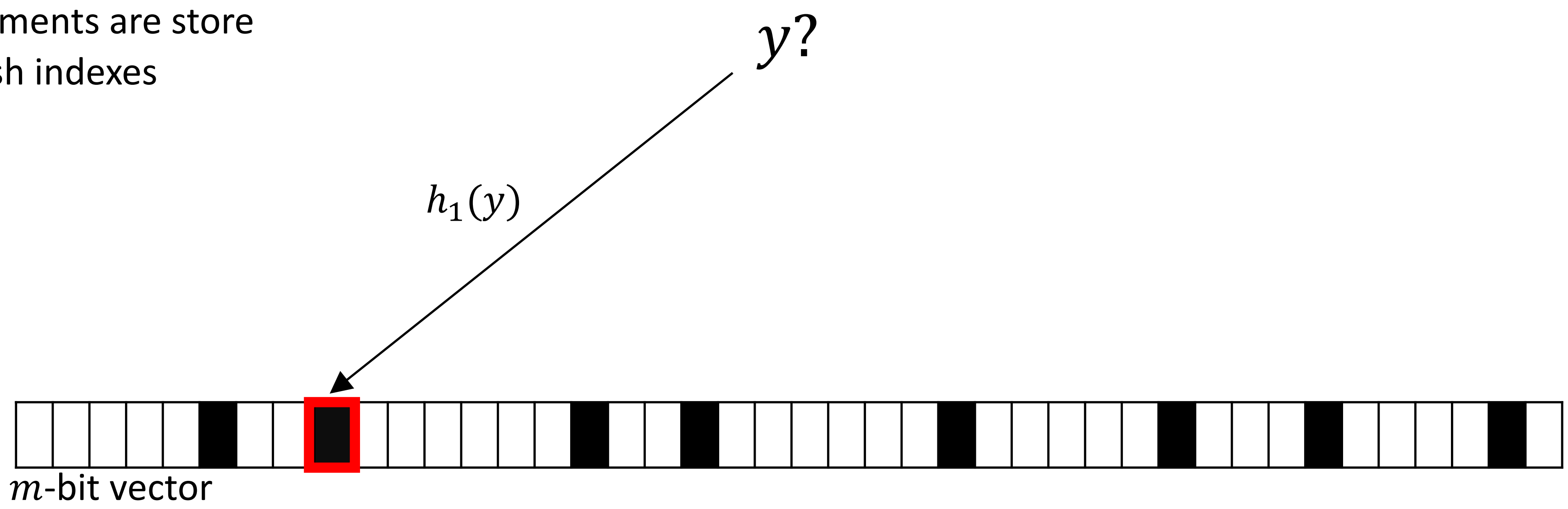
m-bit vector

Bloom Filter

m-bit vector

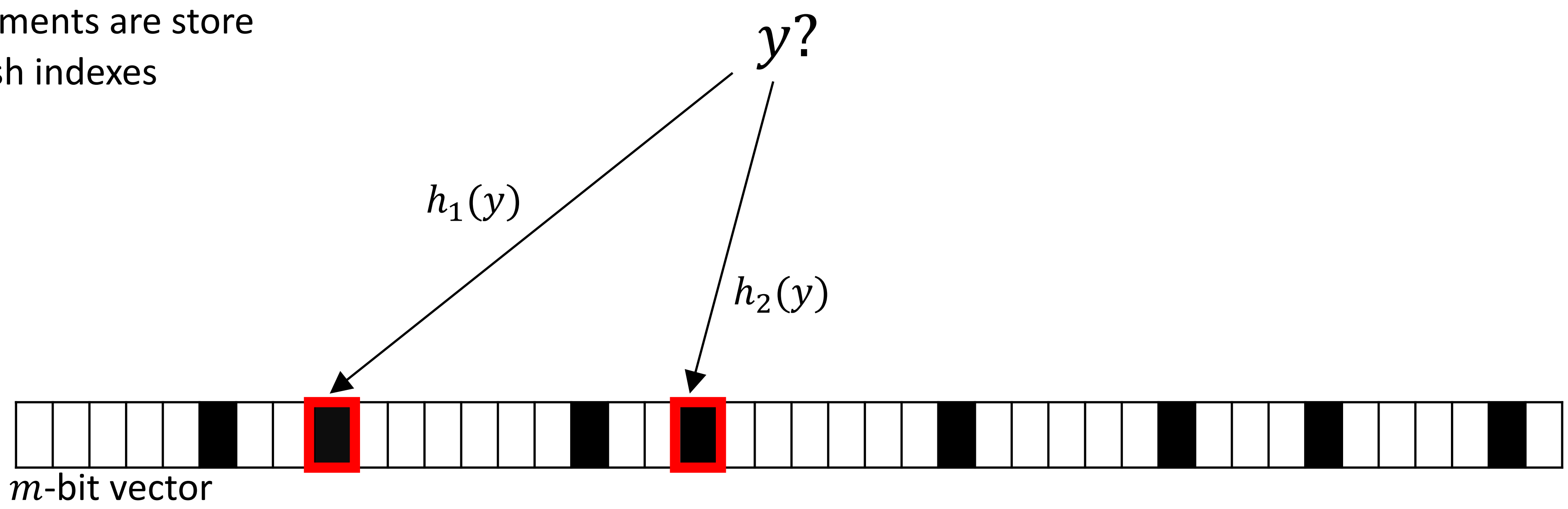
n elements are store

k hash indexes



Bloom Filter

m-bit vector
n elements are store
k hash indexes

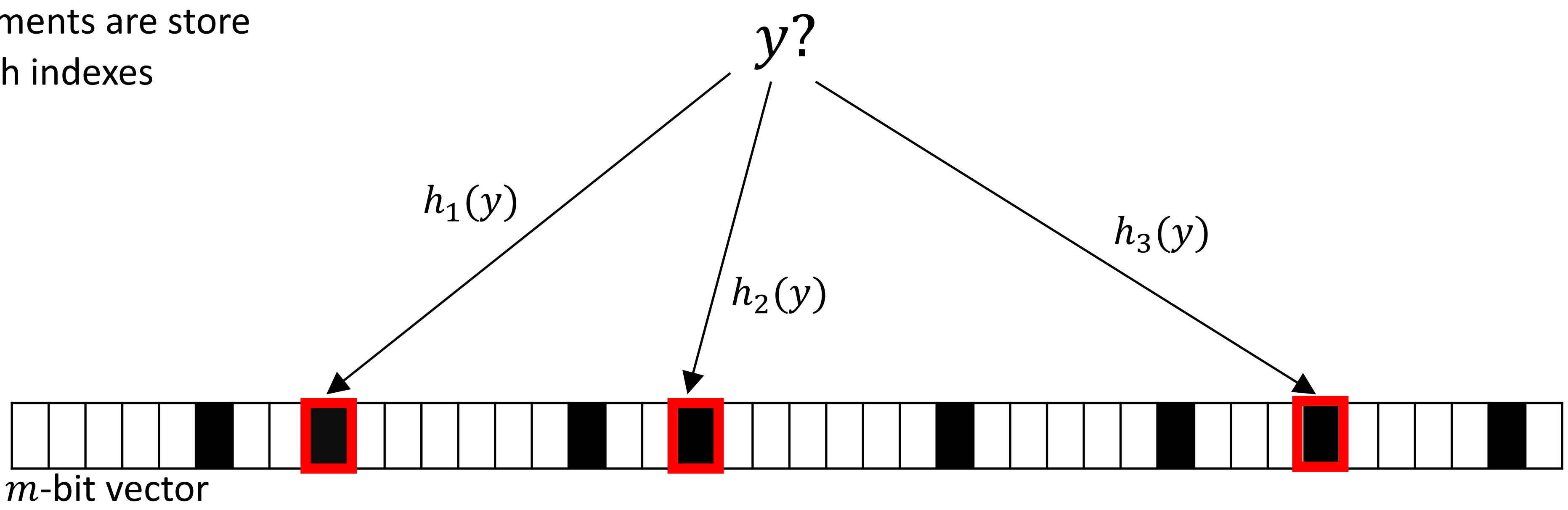


Bloom Filter

m-bit vector

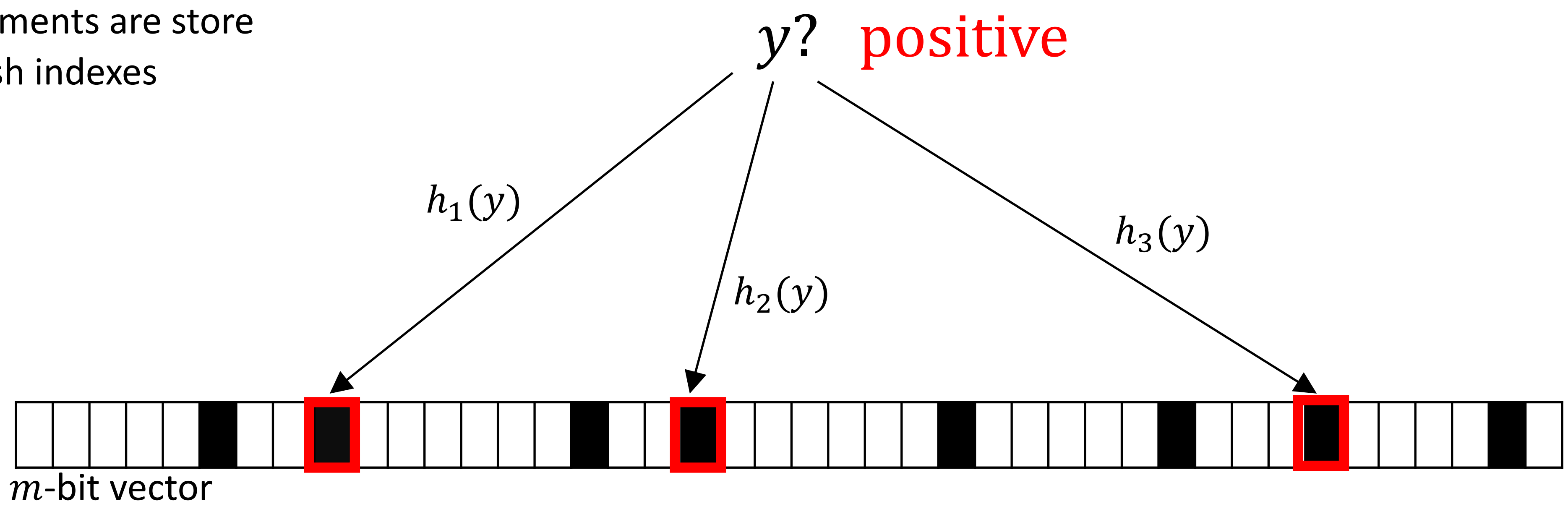
n elements are store

k hash indexes



Bloom Filter

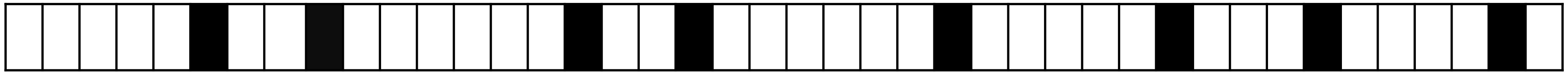
m-bit vector
n elements are store
k hash indexes



Bloom Filter

m-bit vector
n elements are store
k hash indexes

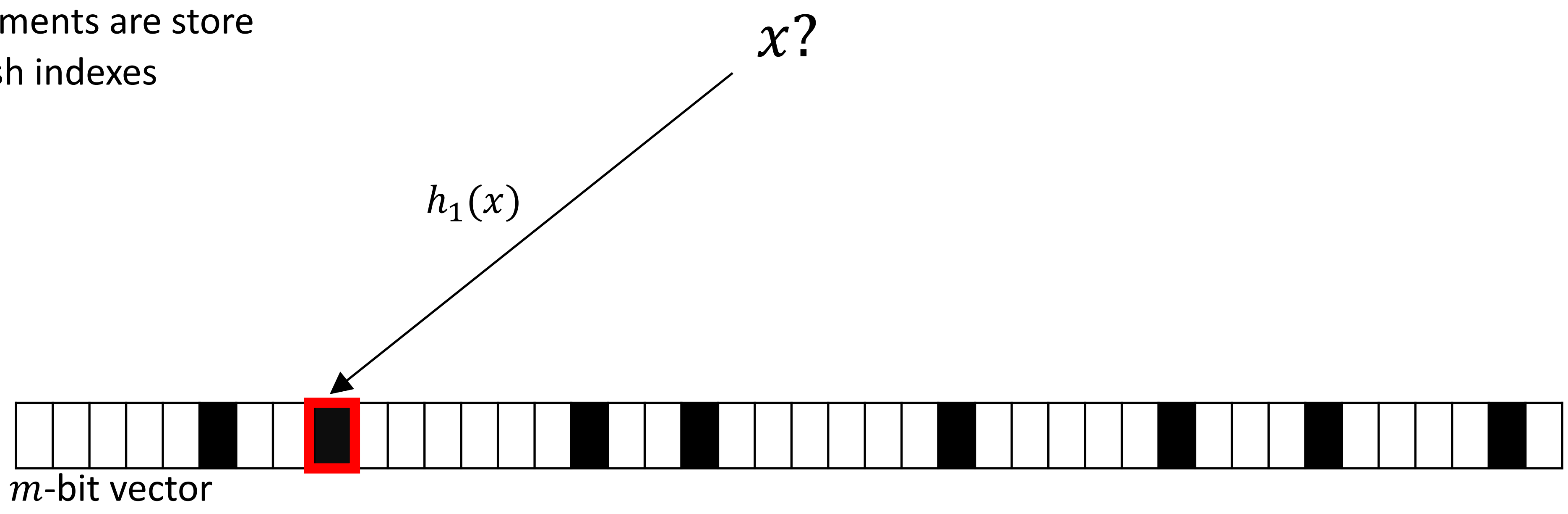
$x?$



m-bit vector

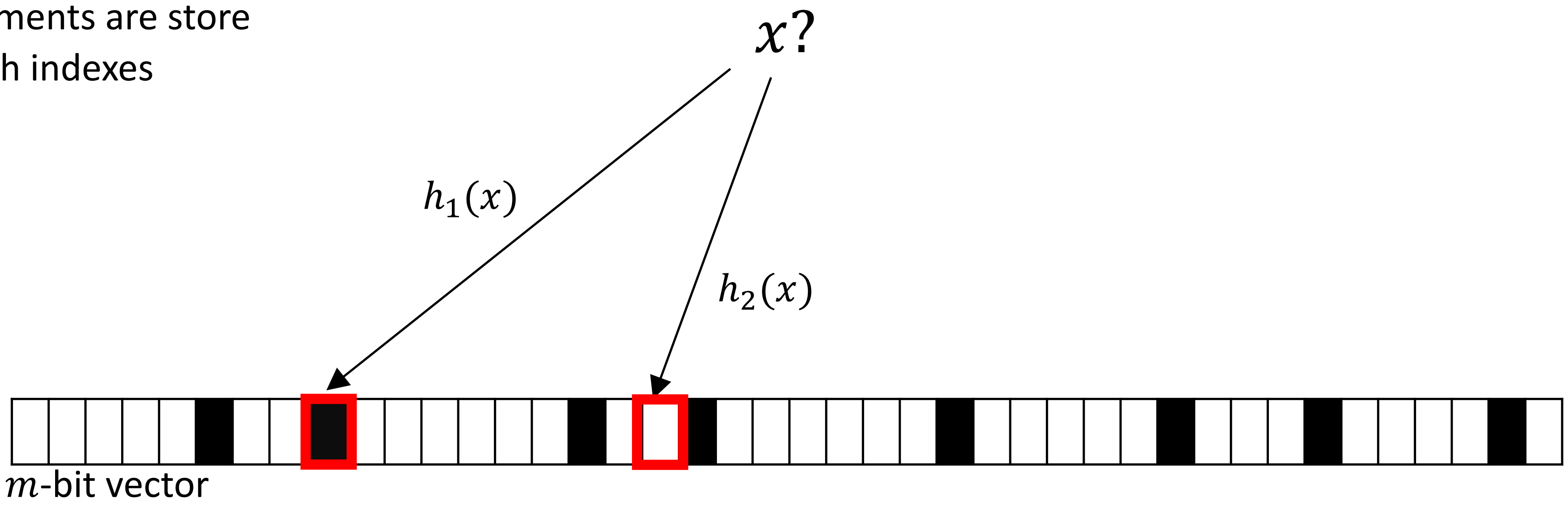
Bloom Filter

m-bit vector
n elements are store
k hash indexes



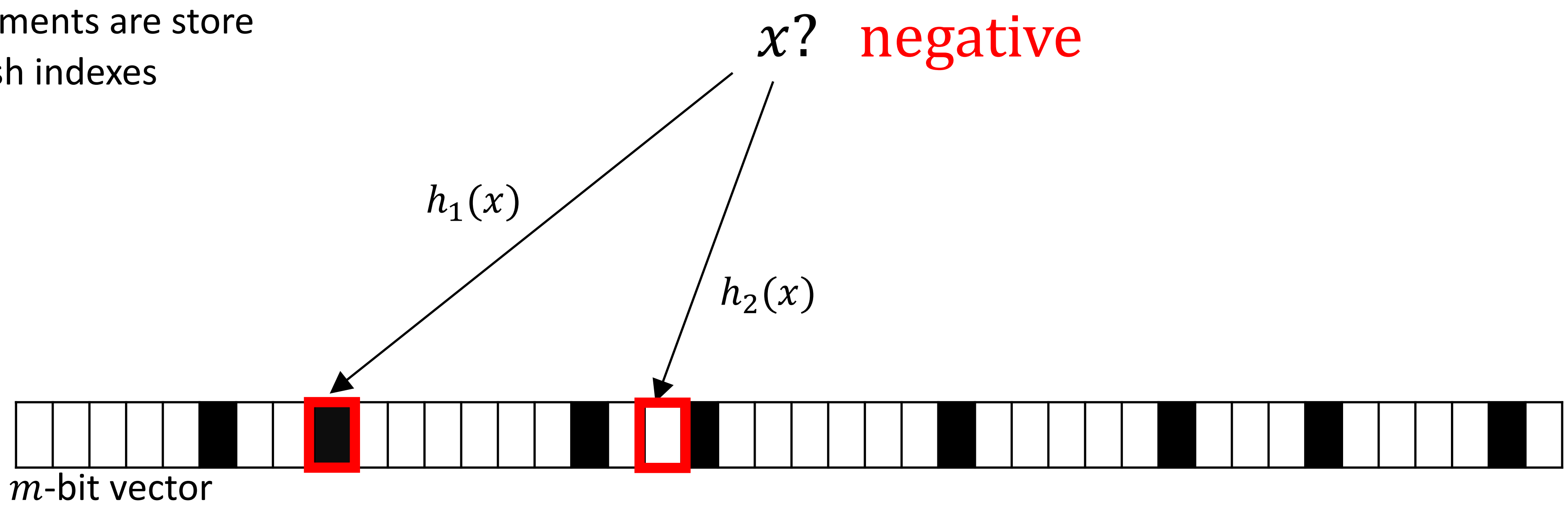
Bloom Filter

m-bit vector
n elements are store
k hash indexes



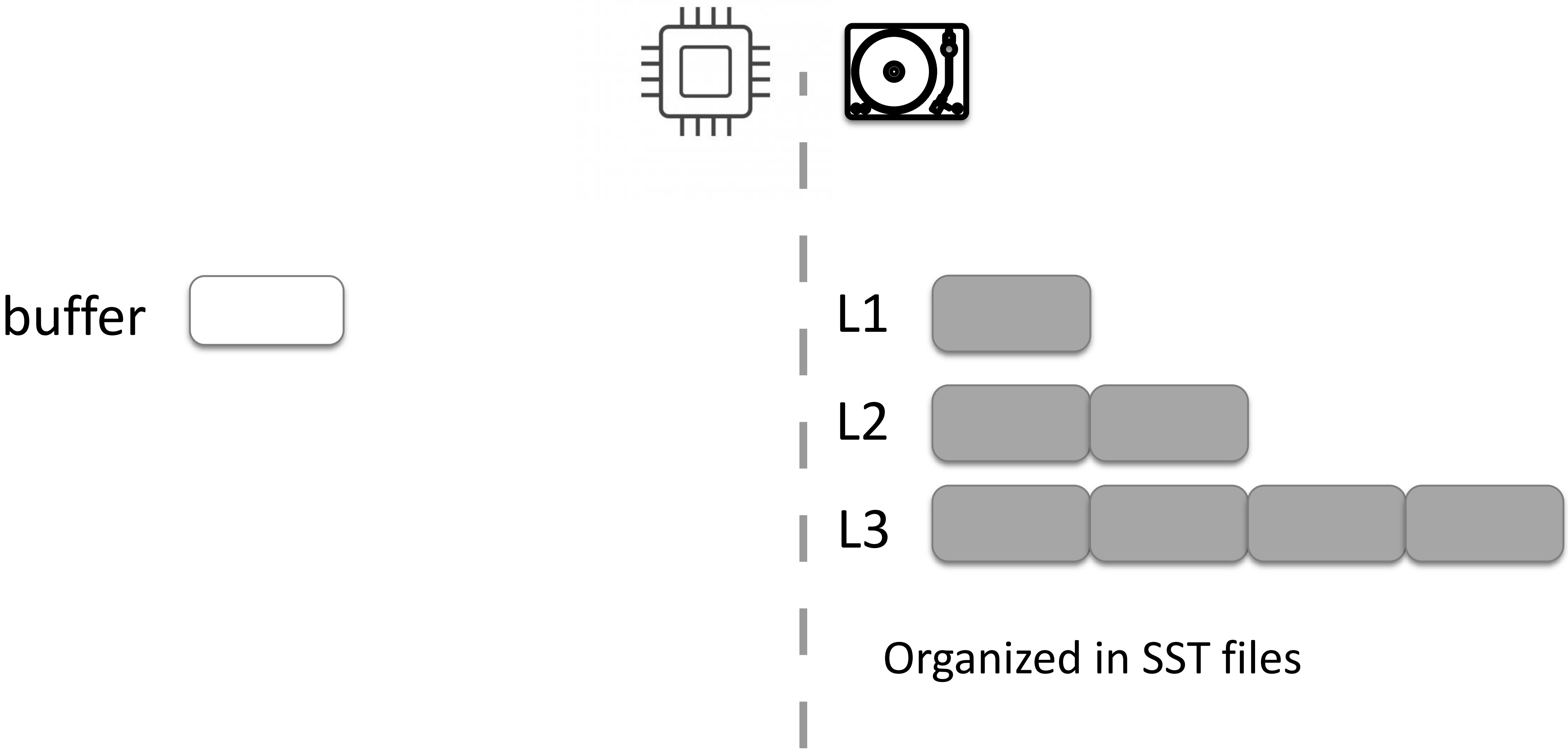
Bloom Filter

m-bit vector
n elements are store
k hash indexes

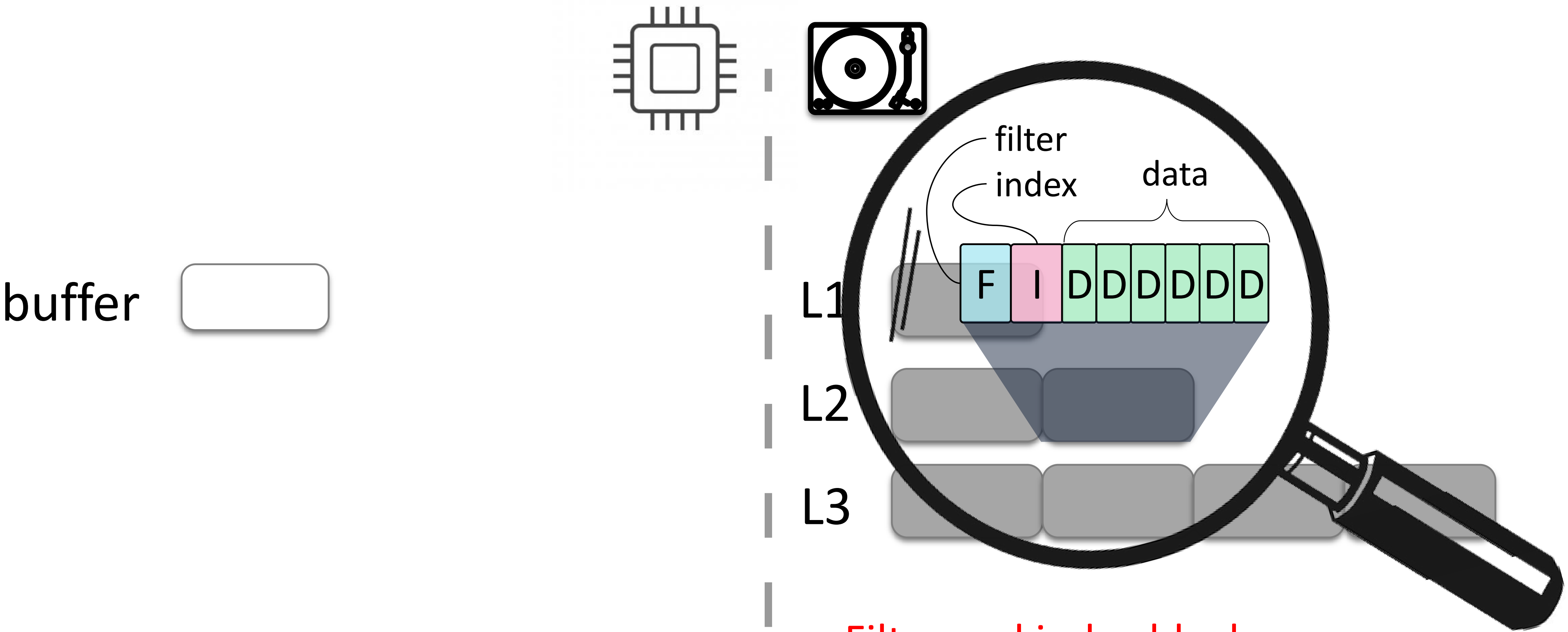


$$\text{false positive } p = e^{-\frac{\text{bits } M}{\text{entries } N} \cdot ln(2)^2}$$

Log-Structured Merge Trees



Log-Structured Merge Trees



Filter and index blocks
to enhance the lookup performance

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

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

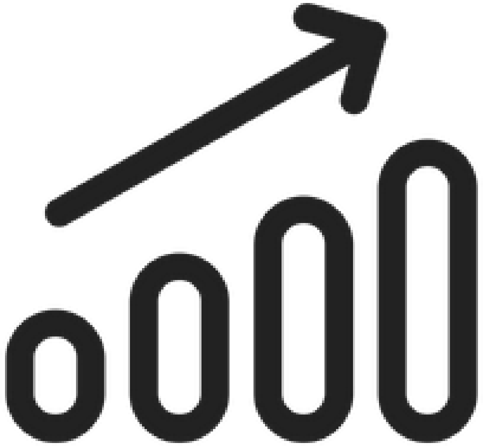
Memory vs. Storage

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

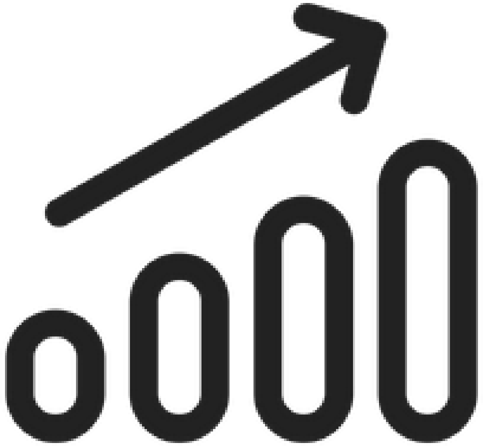
The price drop in memory has been slower than storage making it hard to maintain the same memory-to-data ratio

Memory Pressure in LSM-trees



Data size ↑

Memory Pressure in LSM-trees

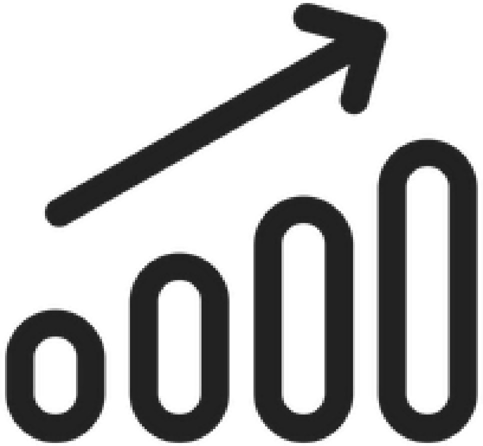


Data size ↑

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

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

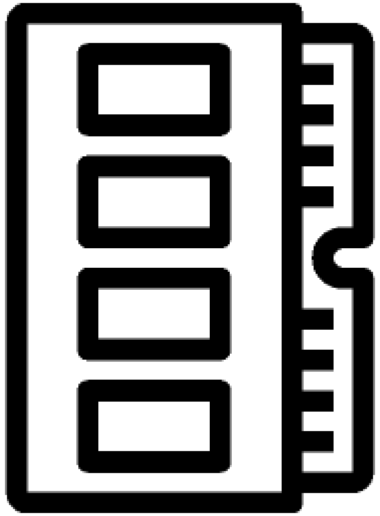
Memory Pressure in LSM-trees



Data size ↑

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

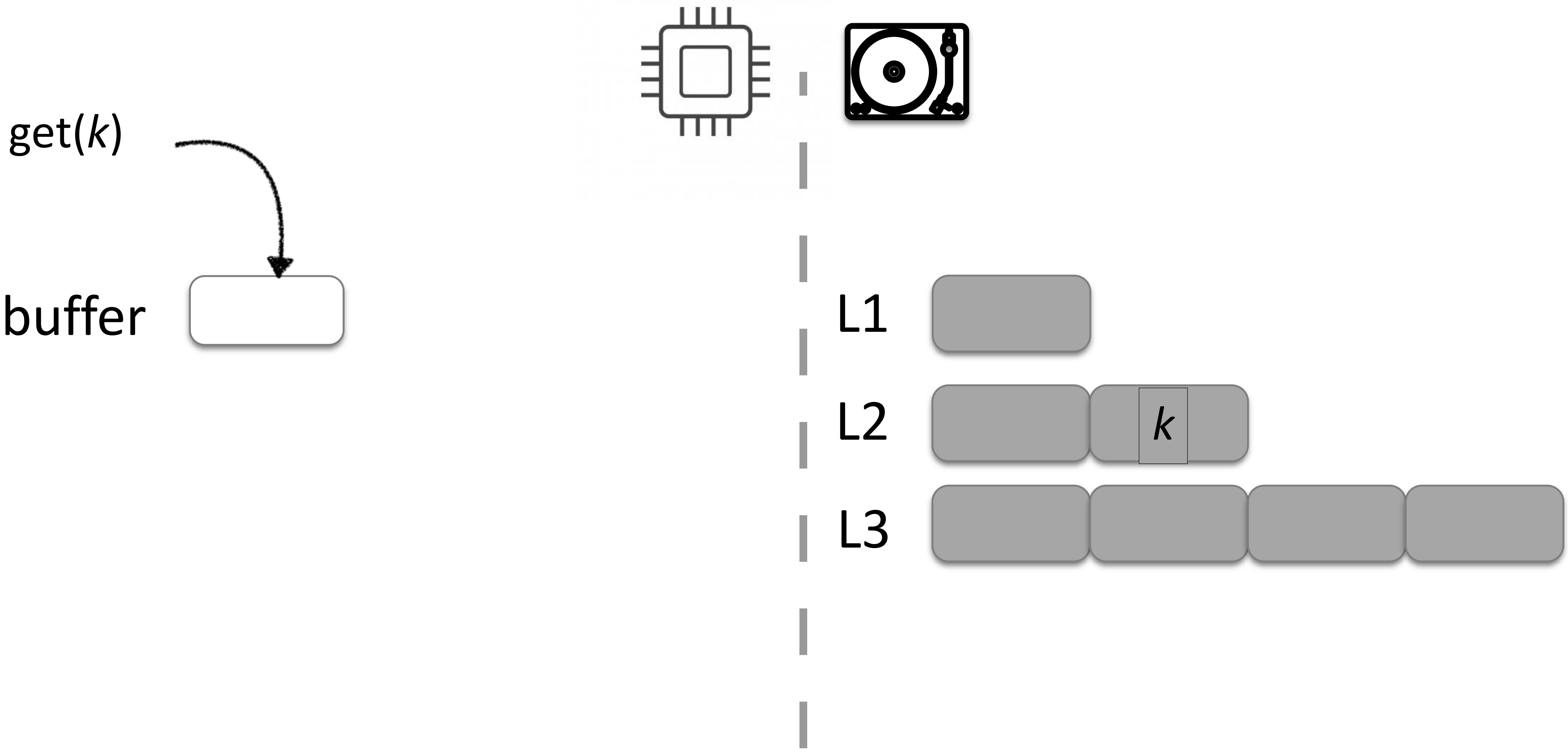
11% space amplification,
1KB entry, 64B key, bpk 10



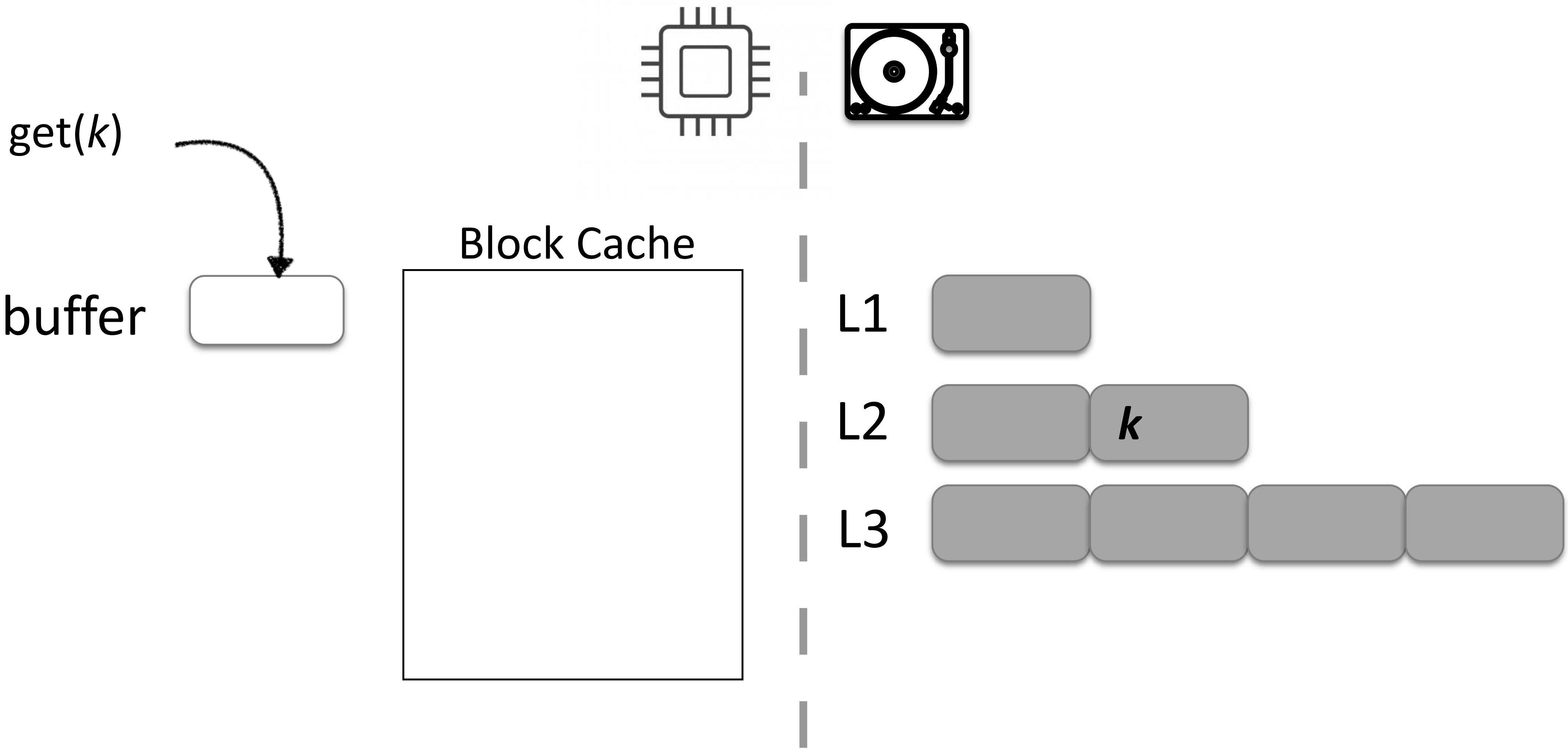
memory-to-data ratio ↓

Memory pressure

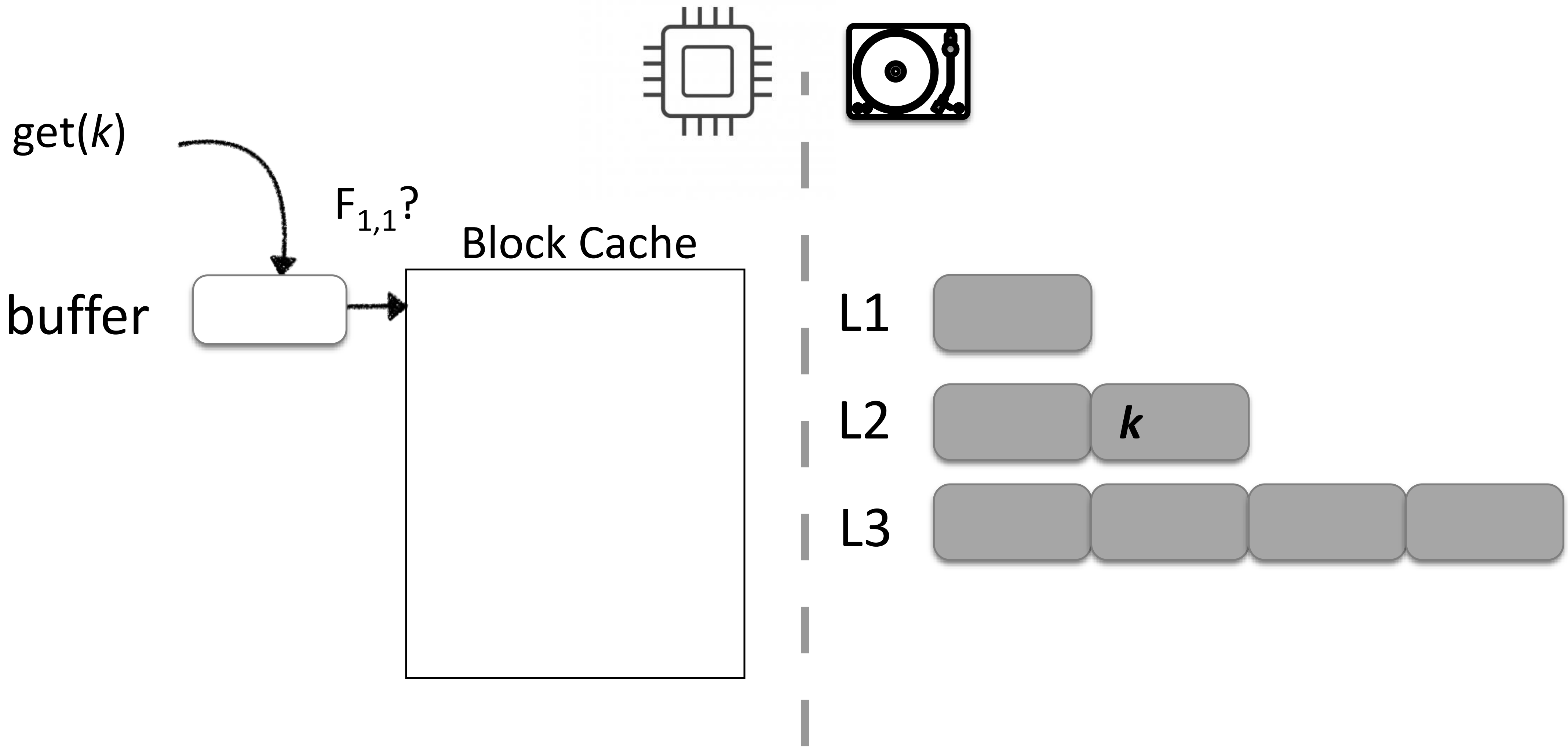
Log-Structured Merge Trees



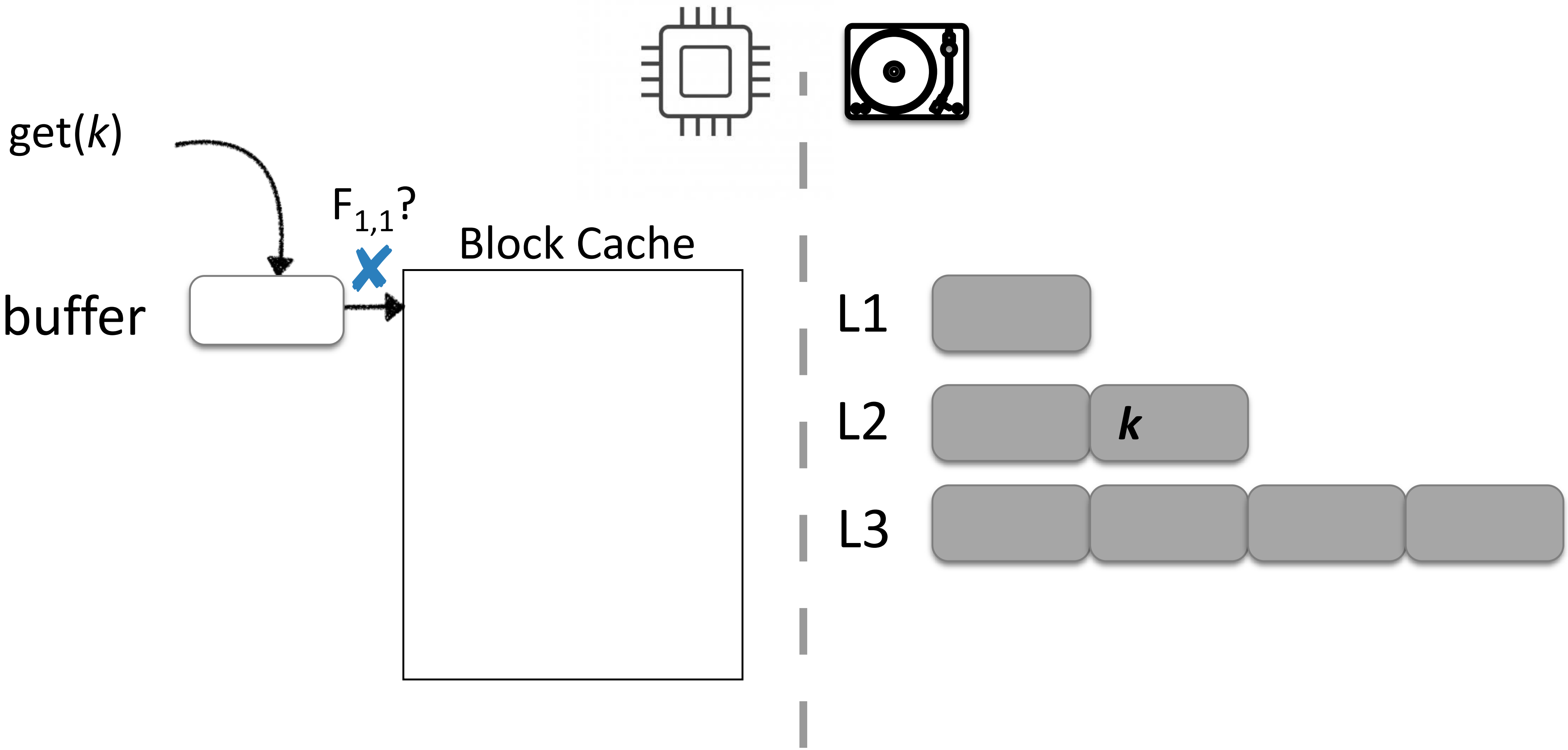
Log-Structured Merge Trees



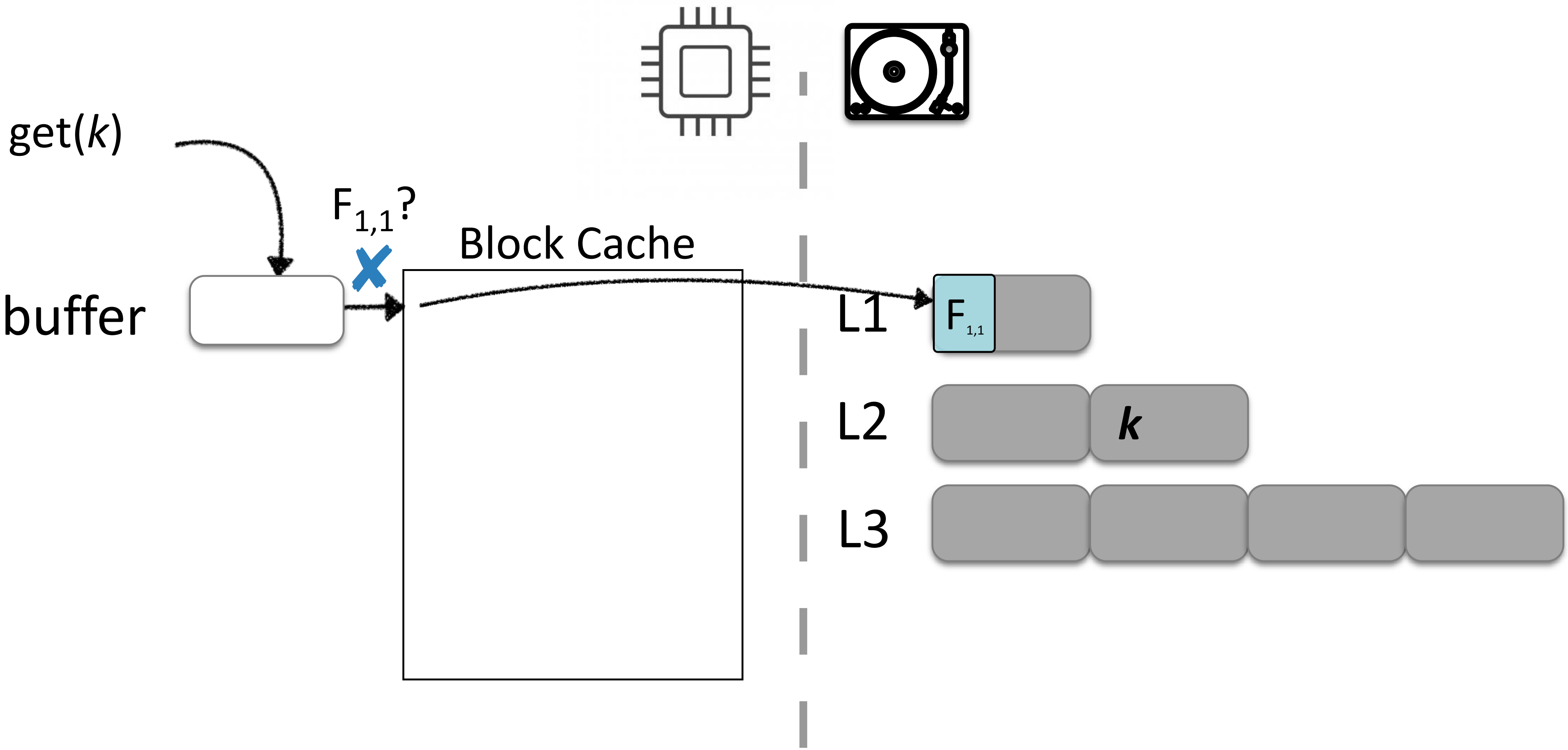
Log-Structured Merge Trees



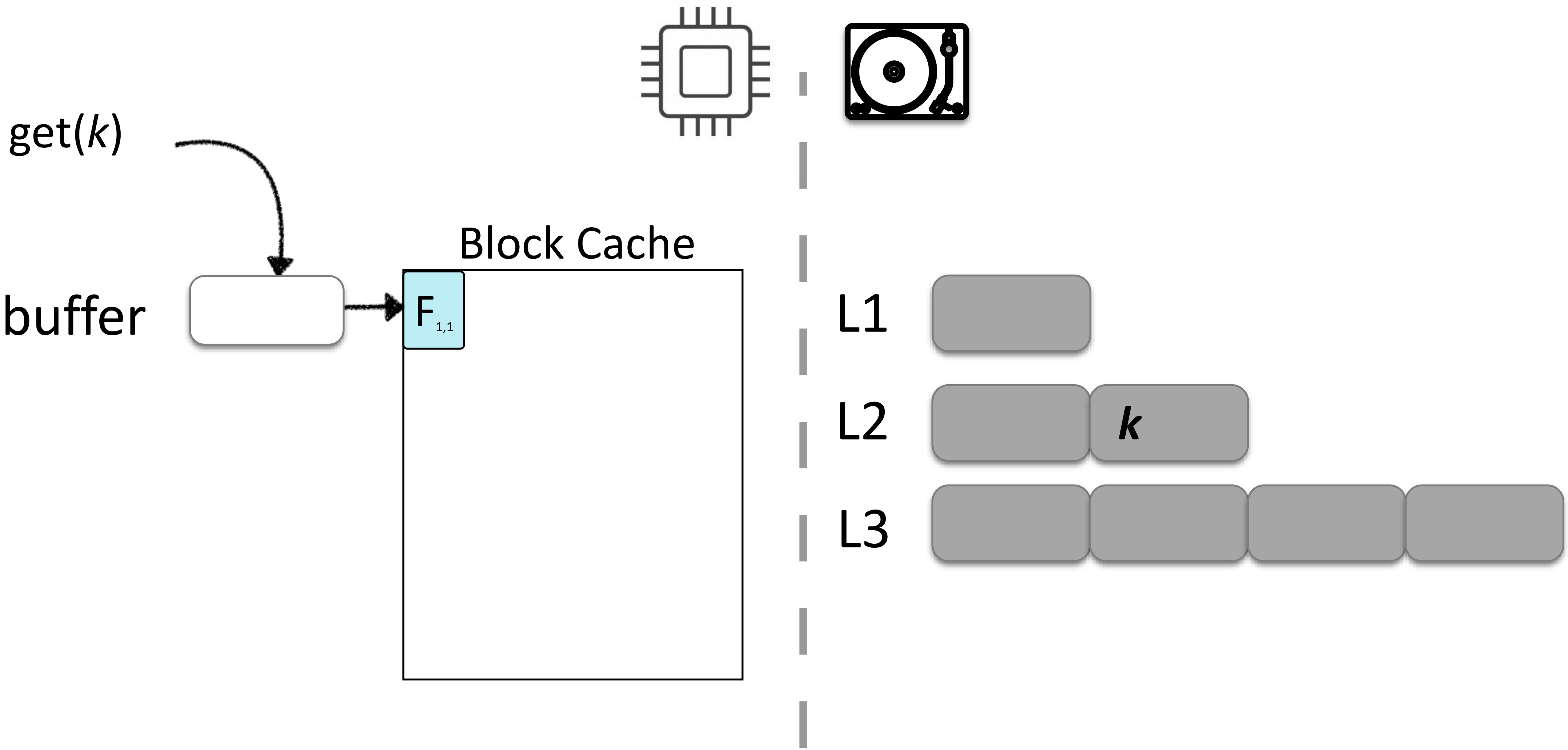
Log-Structured Merge Trees



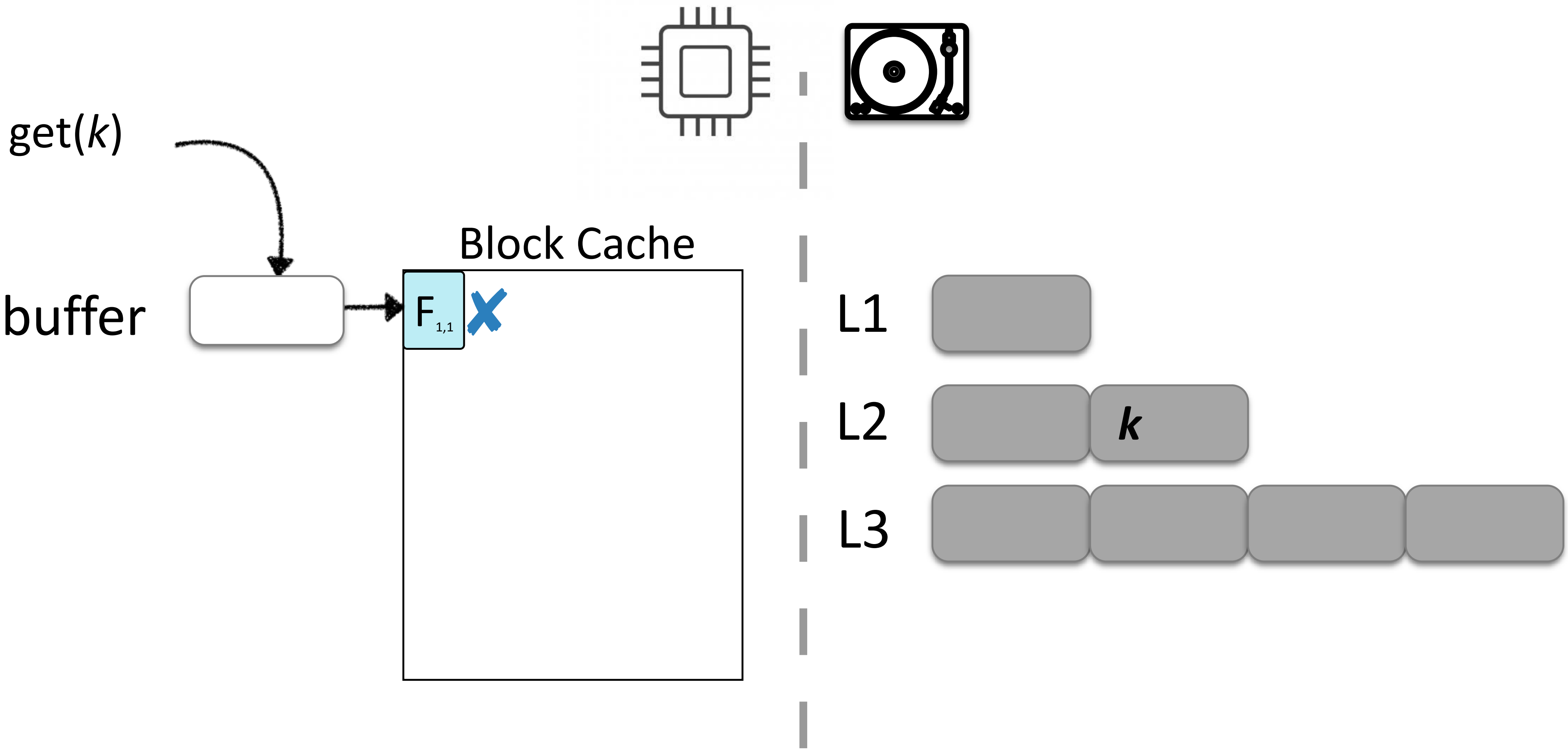
Log-Structured Merge Trees



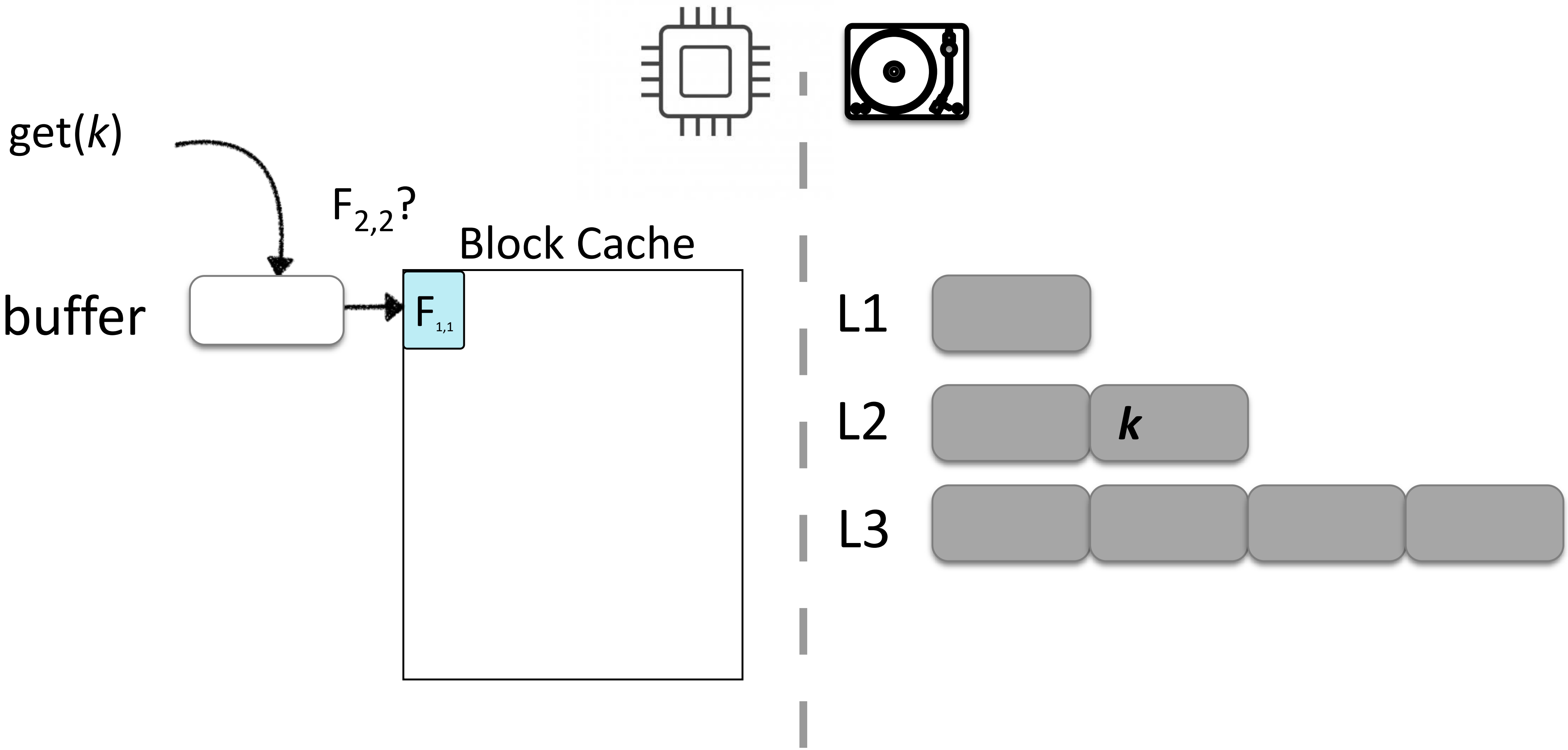
Log-Structured Merge Trees



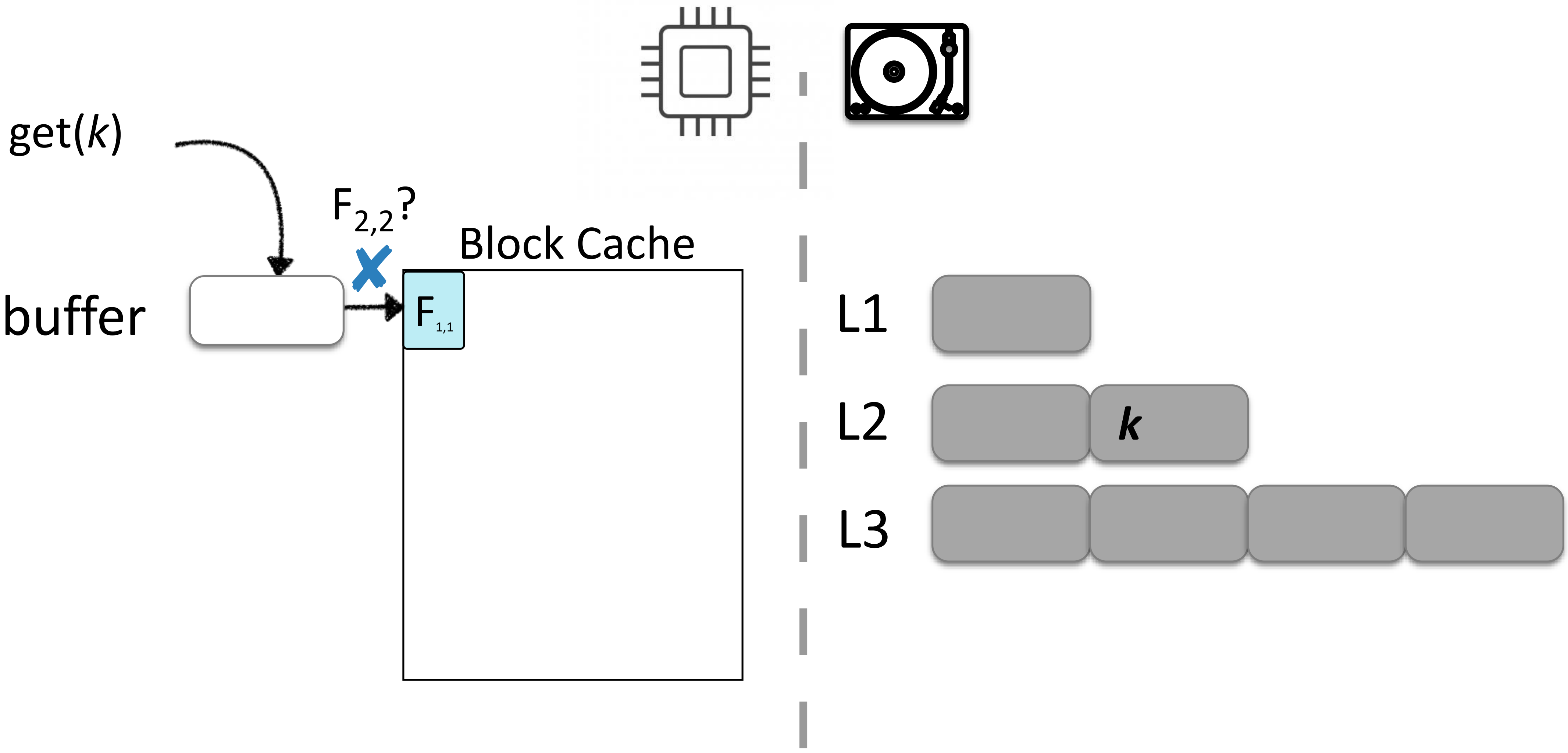
Log-Structured Merge Trees



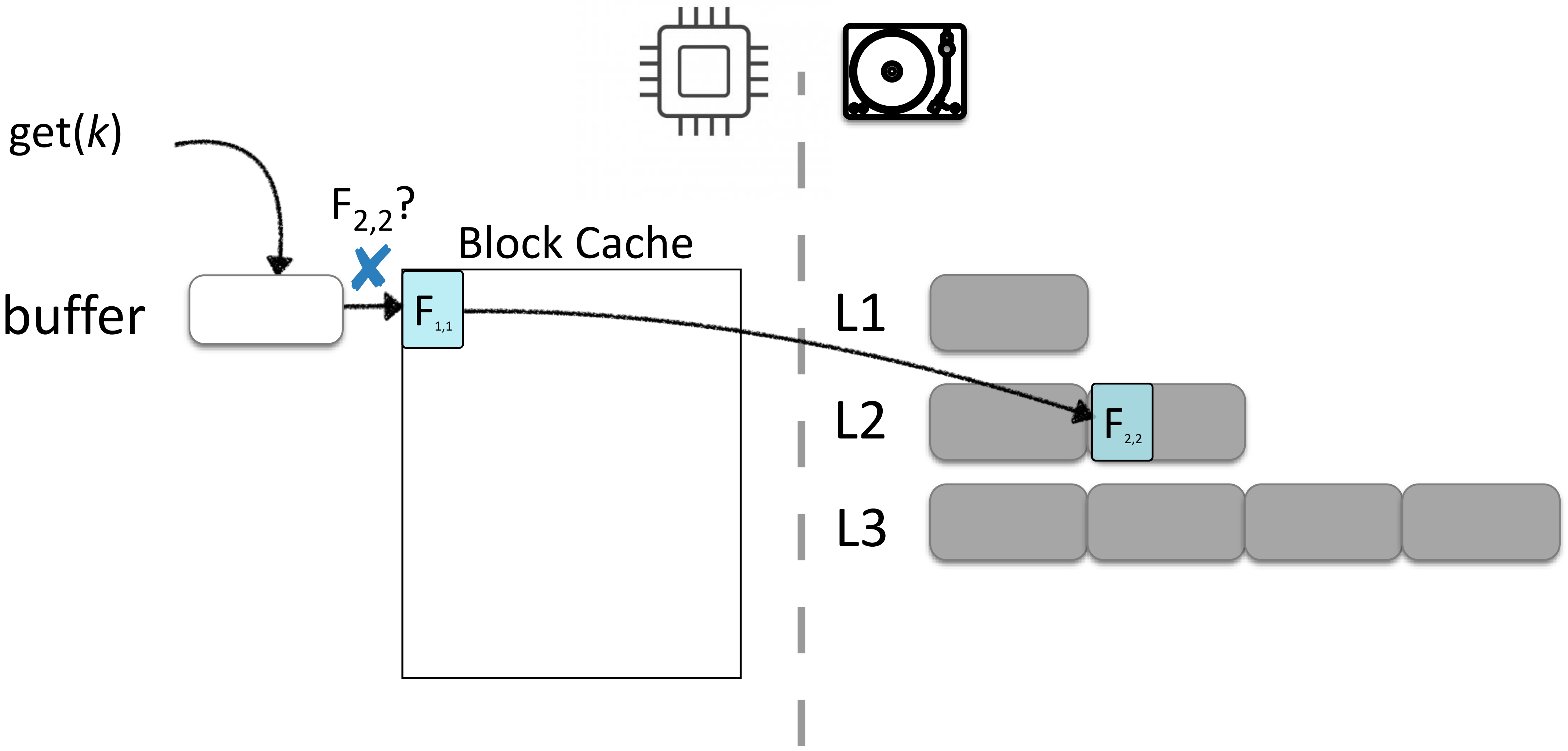
Log-Structured Merge Trees



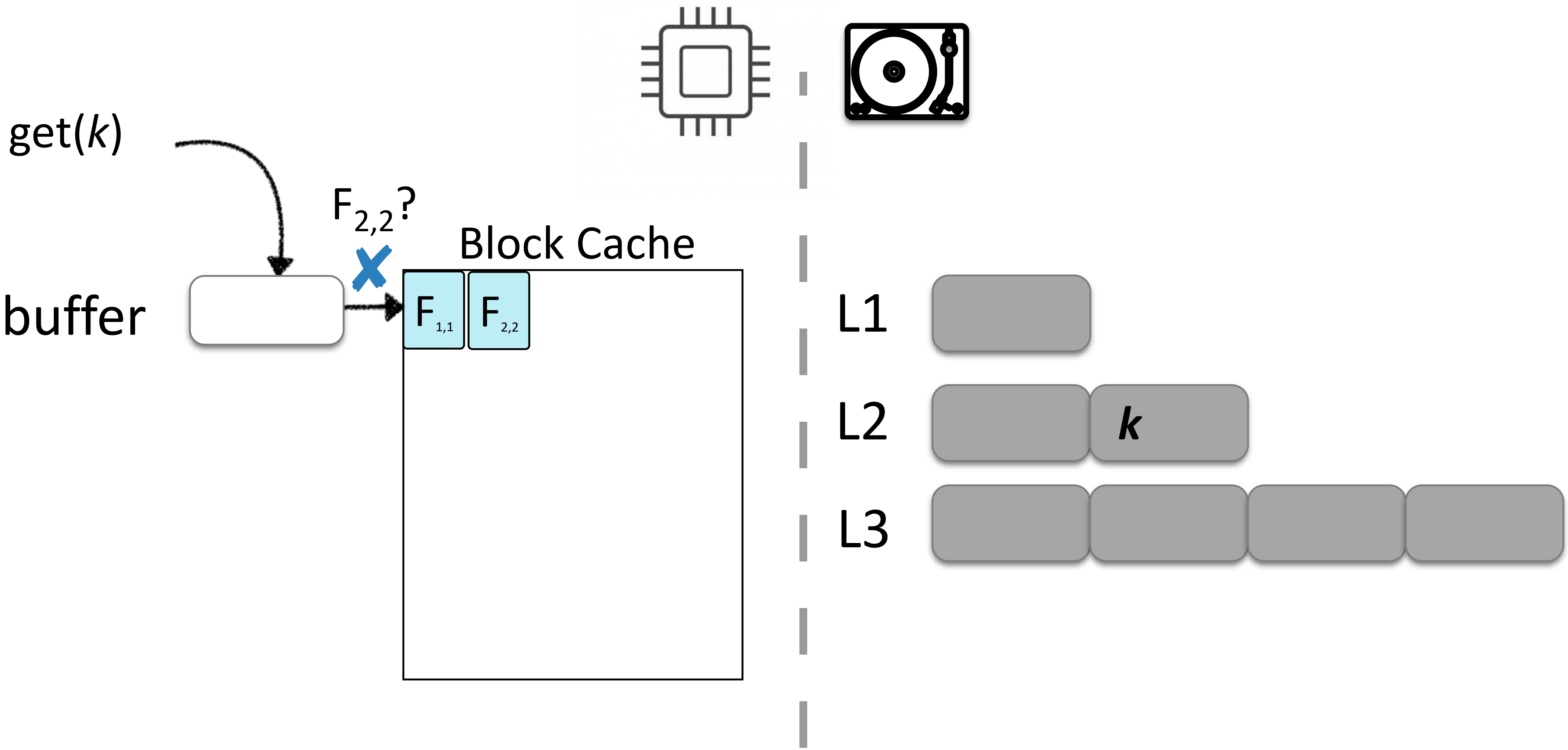
Log-Structured Merge Trees



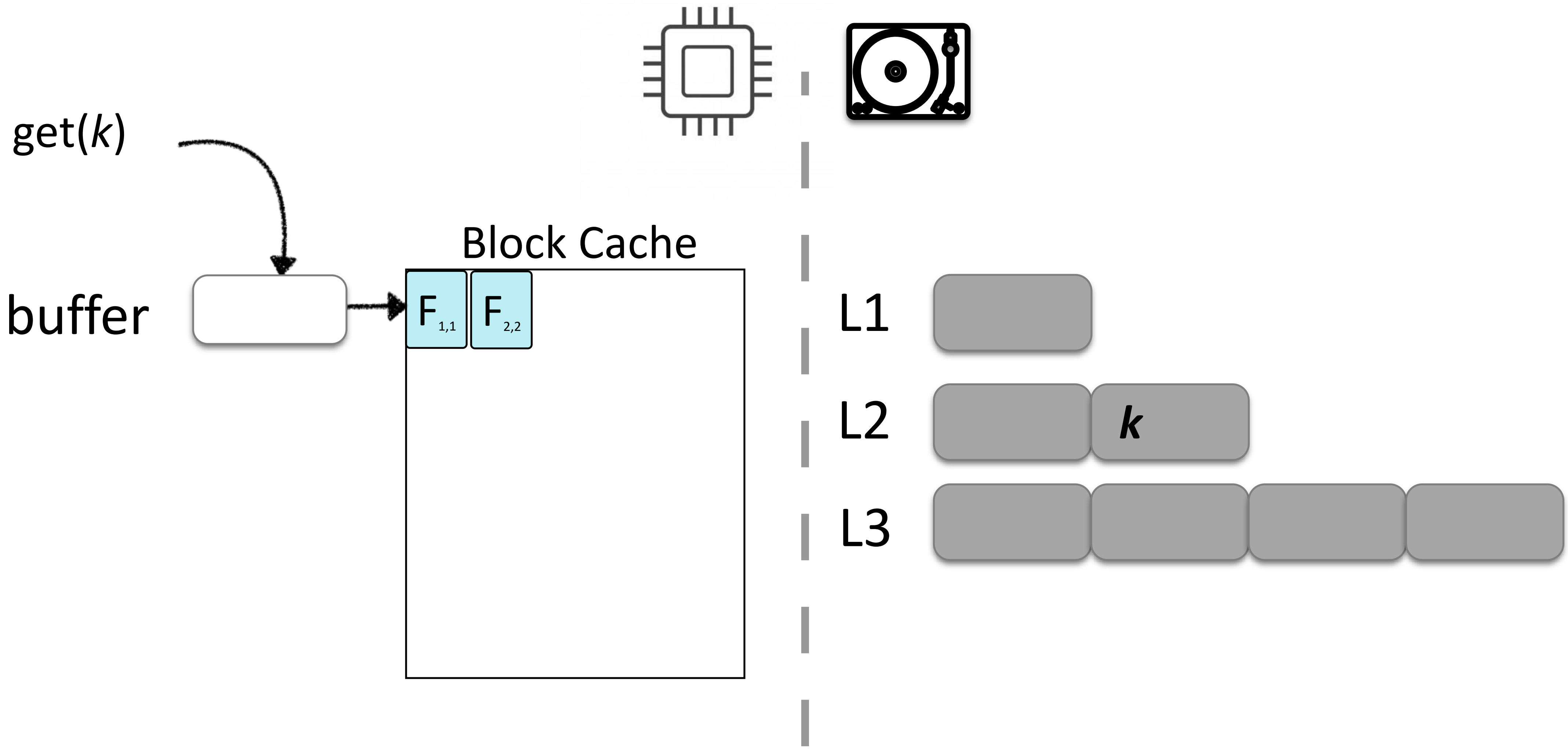
Log-Structured Merge Trees



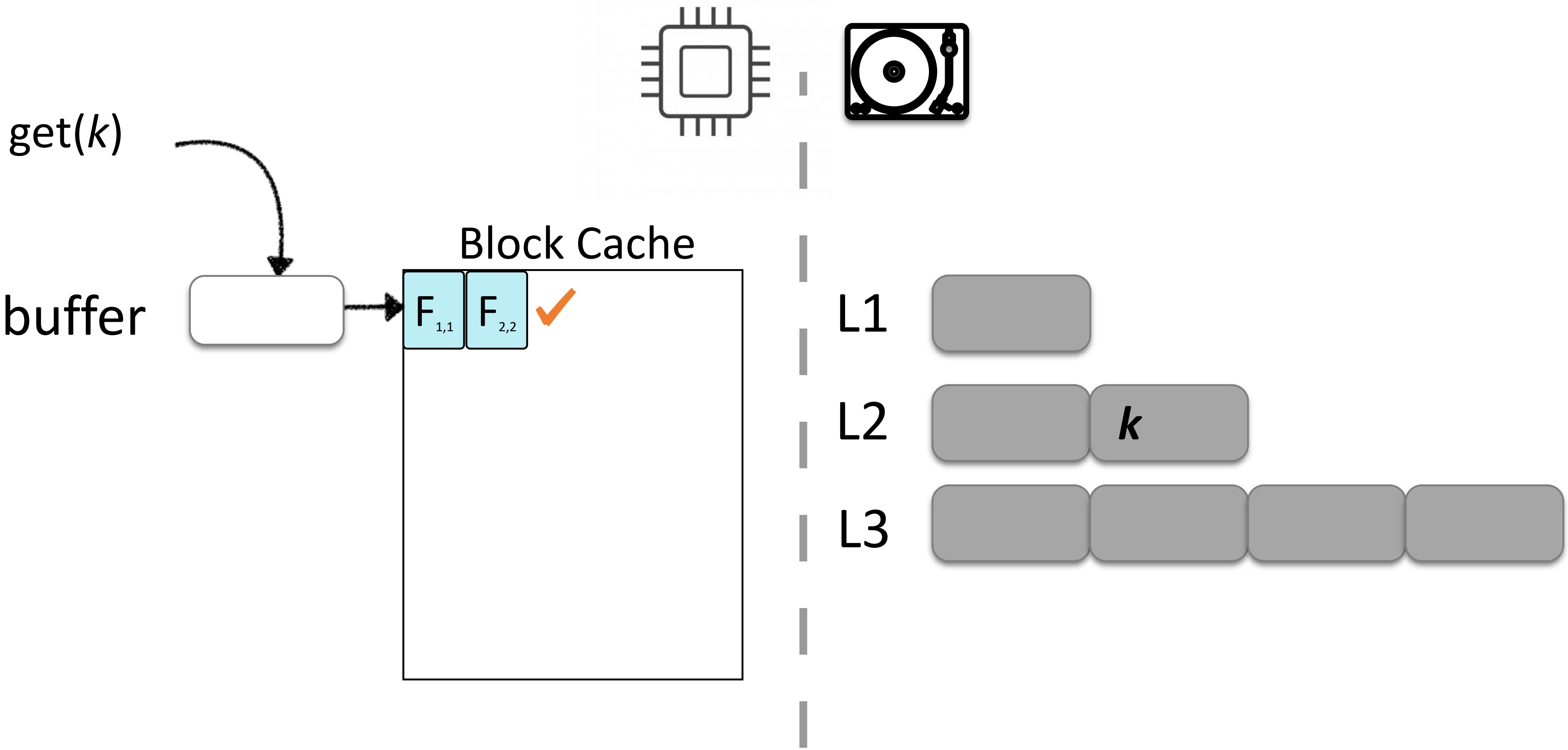
Log-Structured Merge Trees



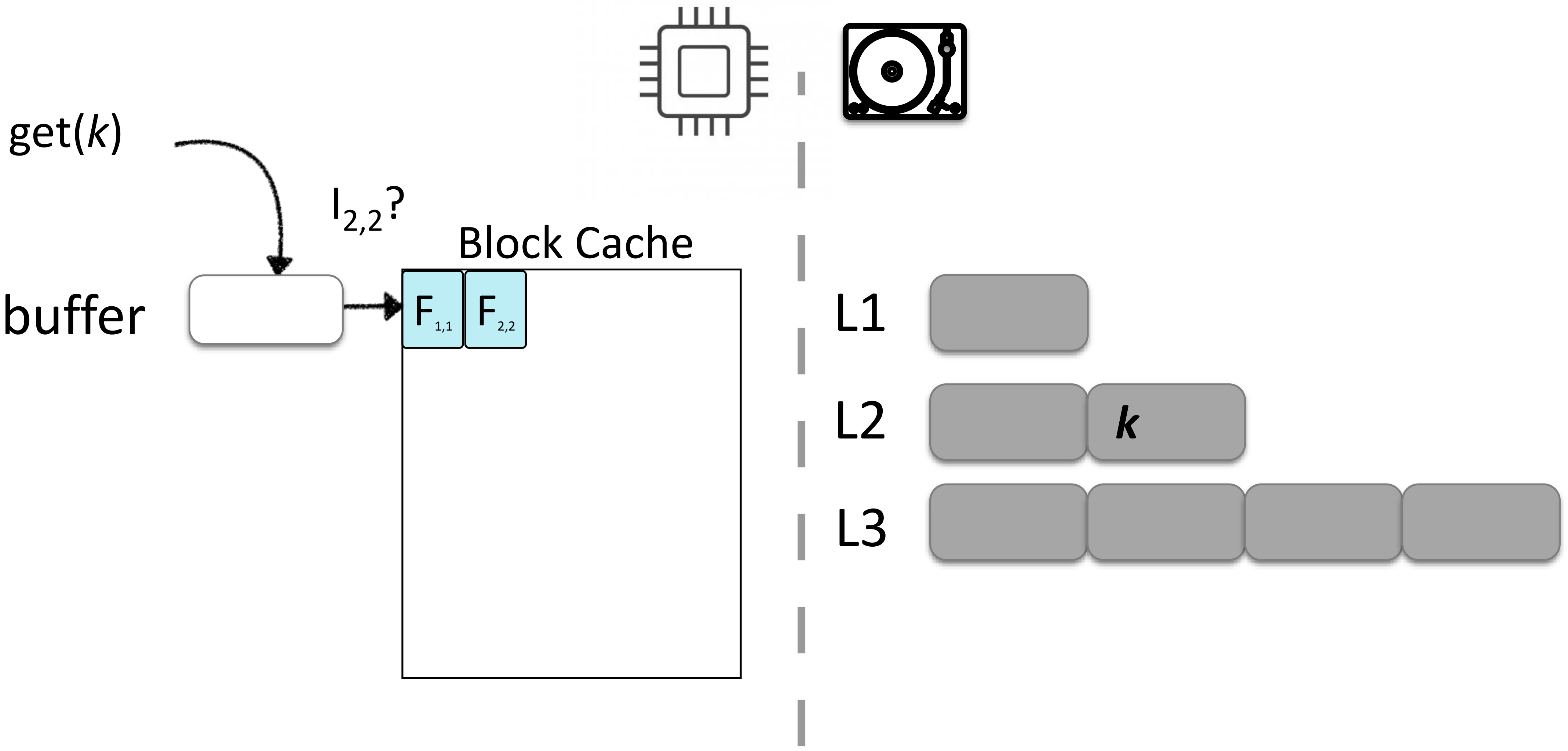
Log-Structured Merge Trees



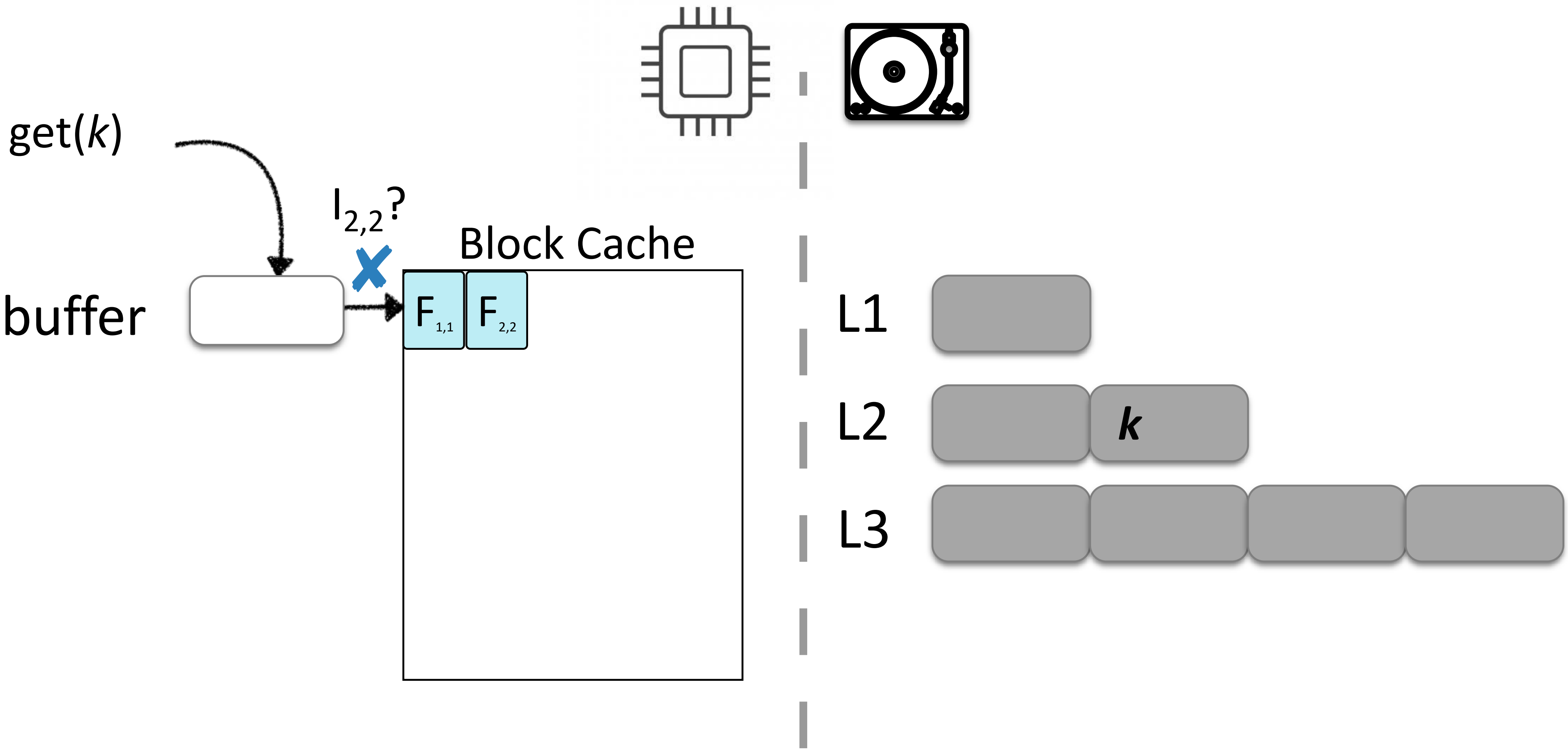
Log-Structured Merge Trees



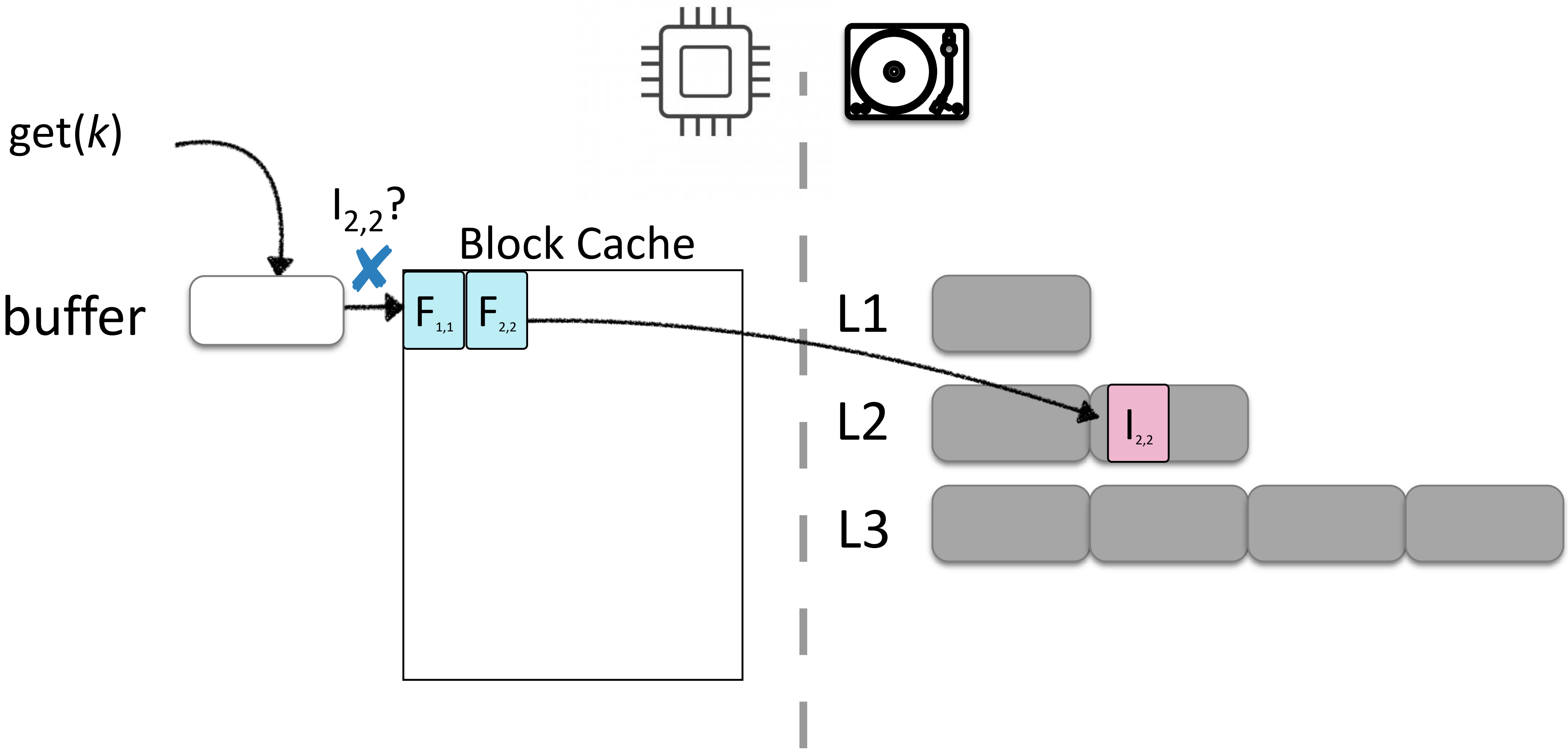
Log-Structured Merge Trees



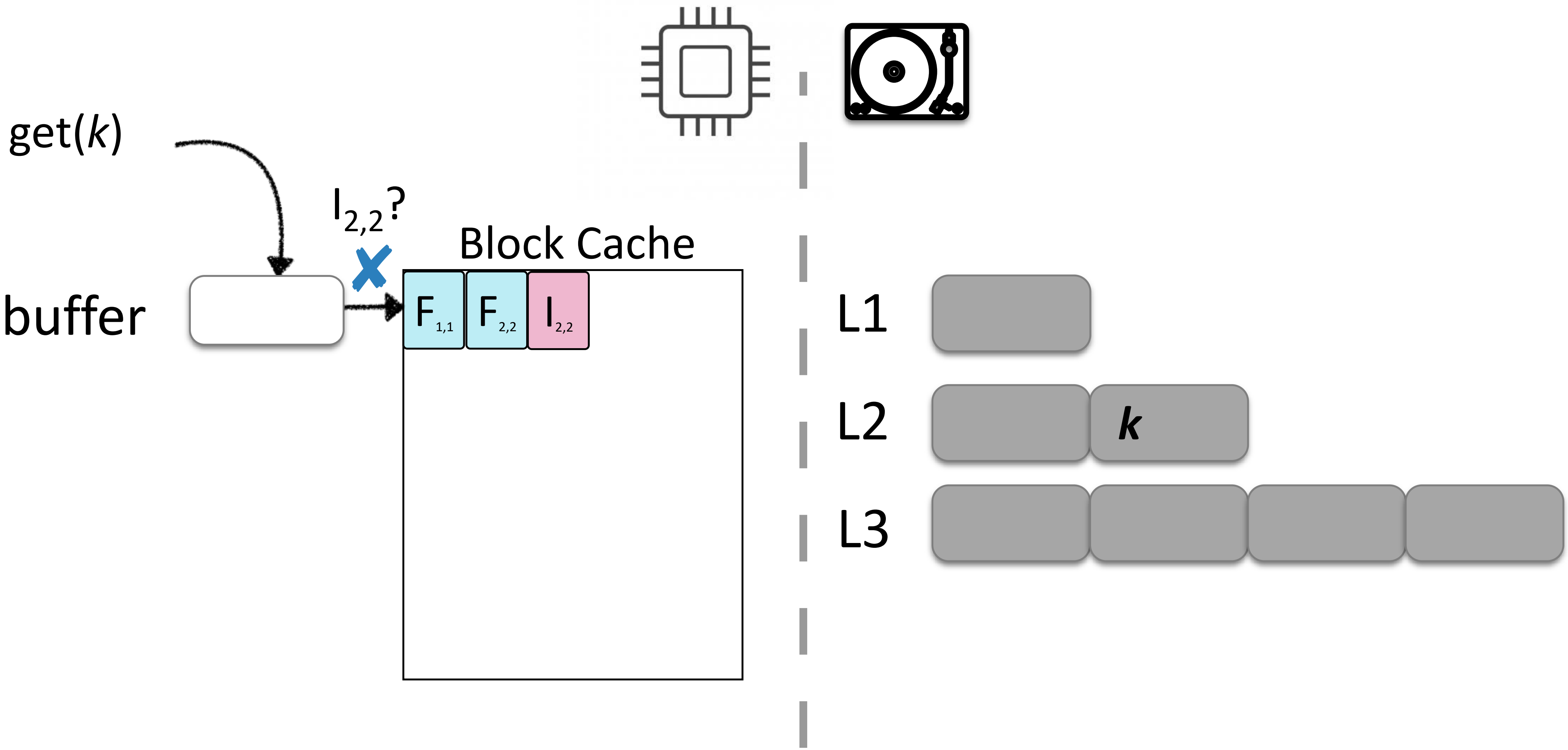
Log-Structured Merge Trees



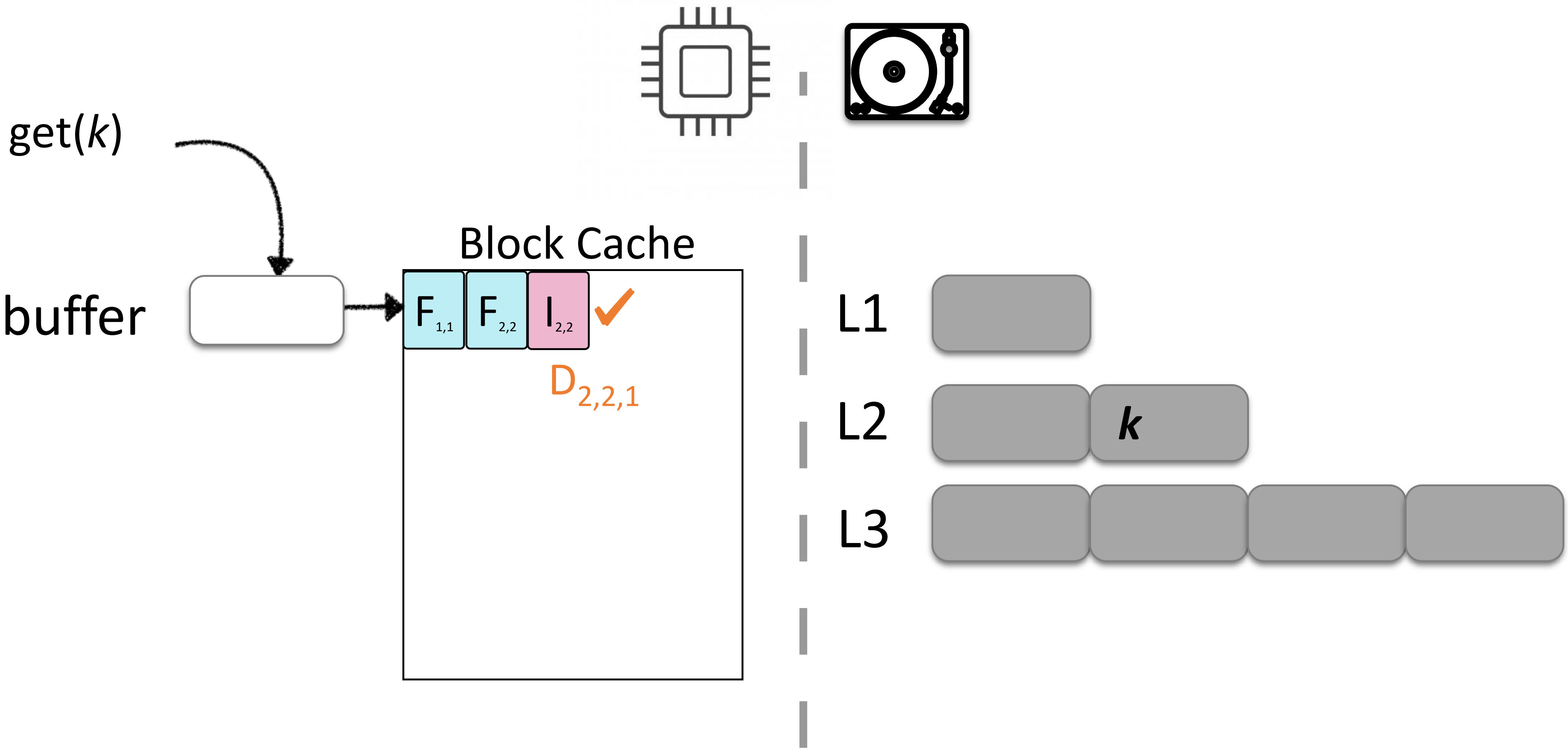
Log-Structured Merge Trees



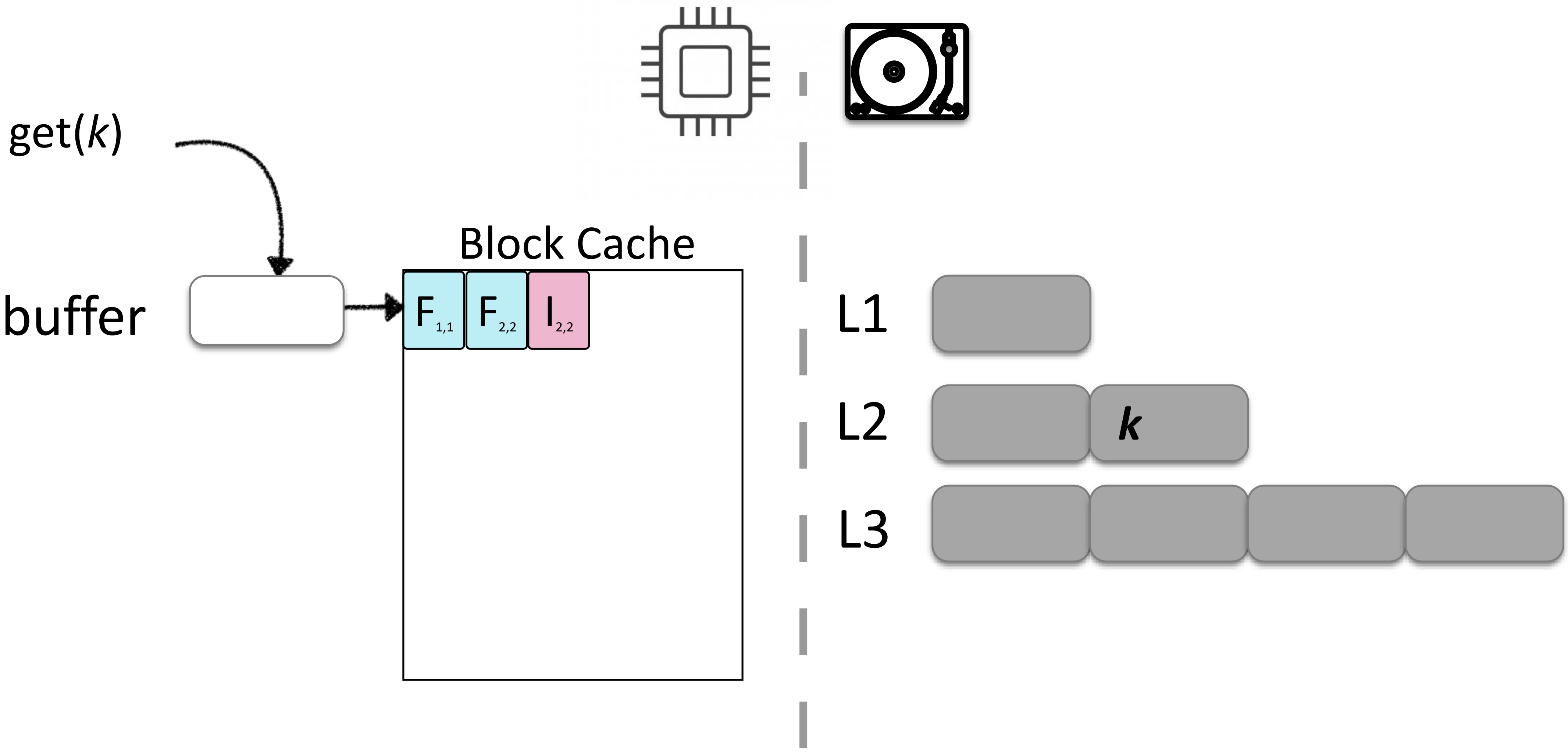
Log-Structured Merge Trees



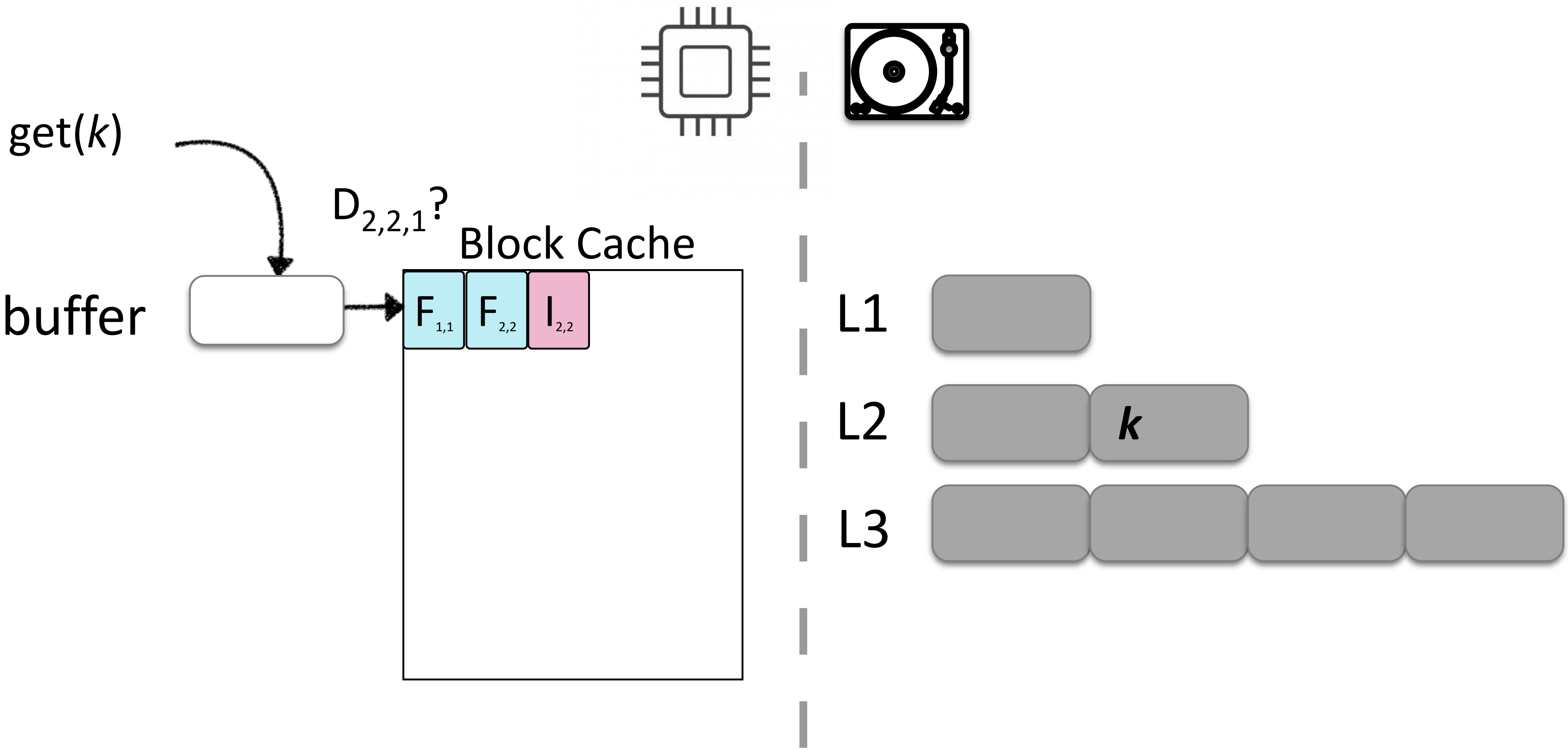
Log-Structured Merge Trees



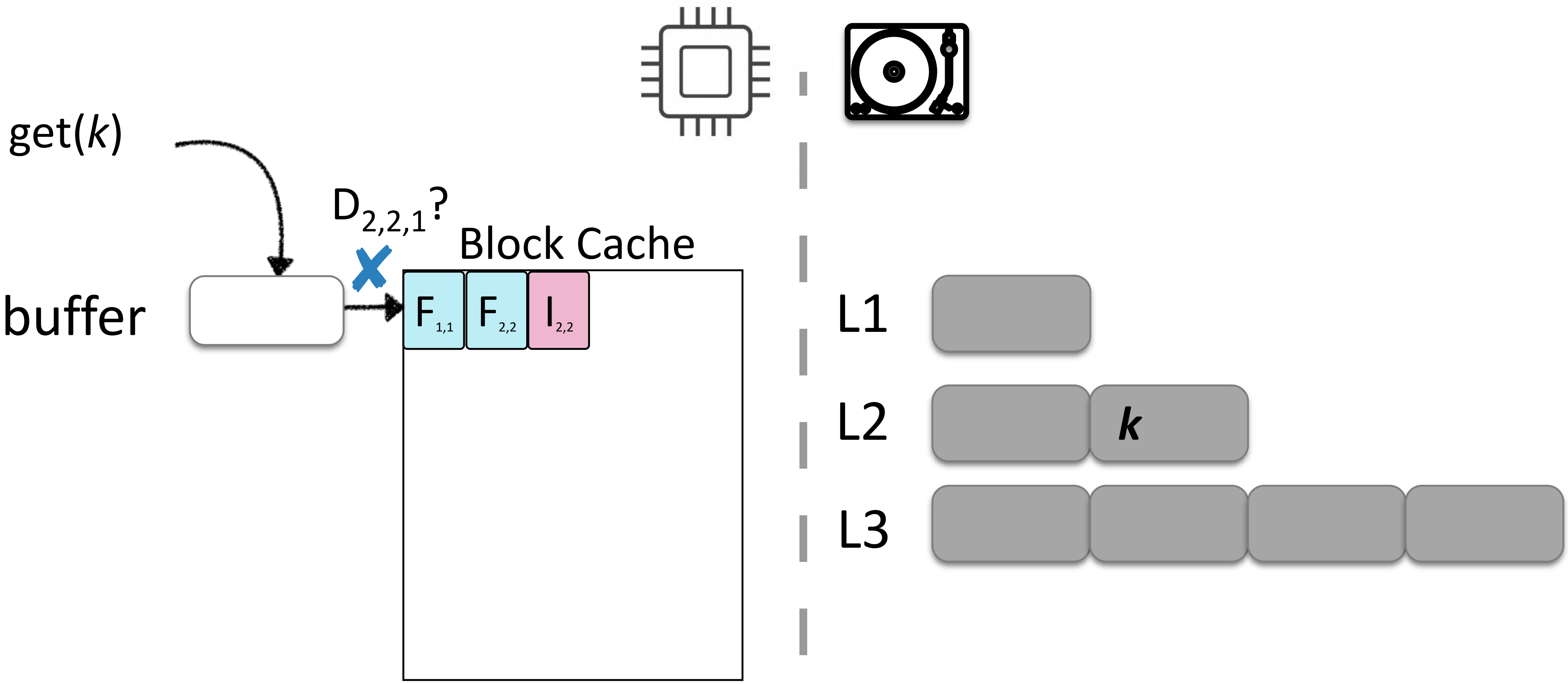
Log-Structured Merge Trees



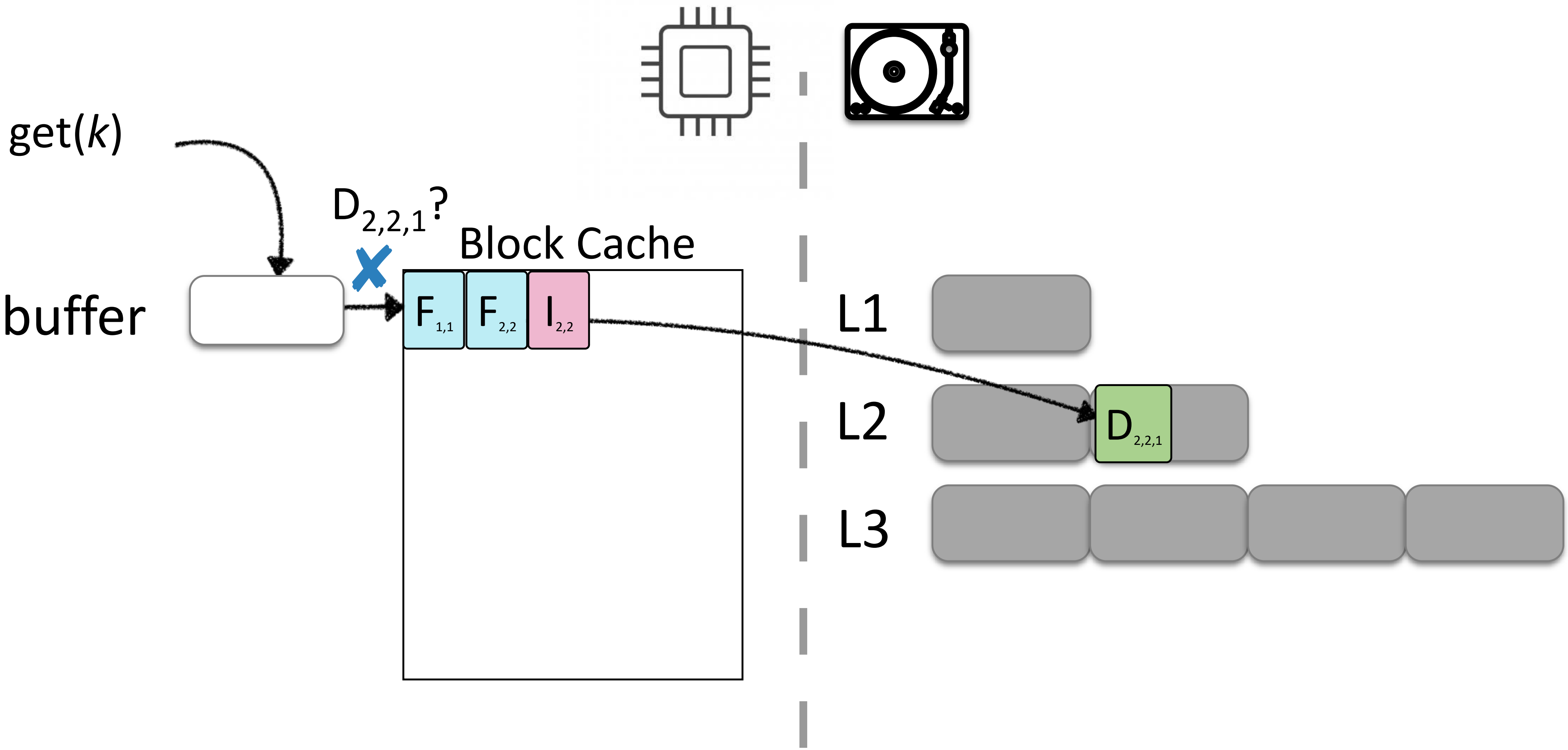
Log-Structured Merge Trees



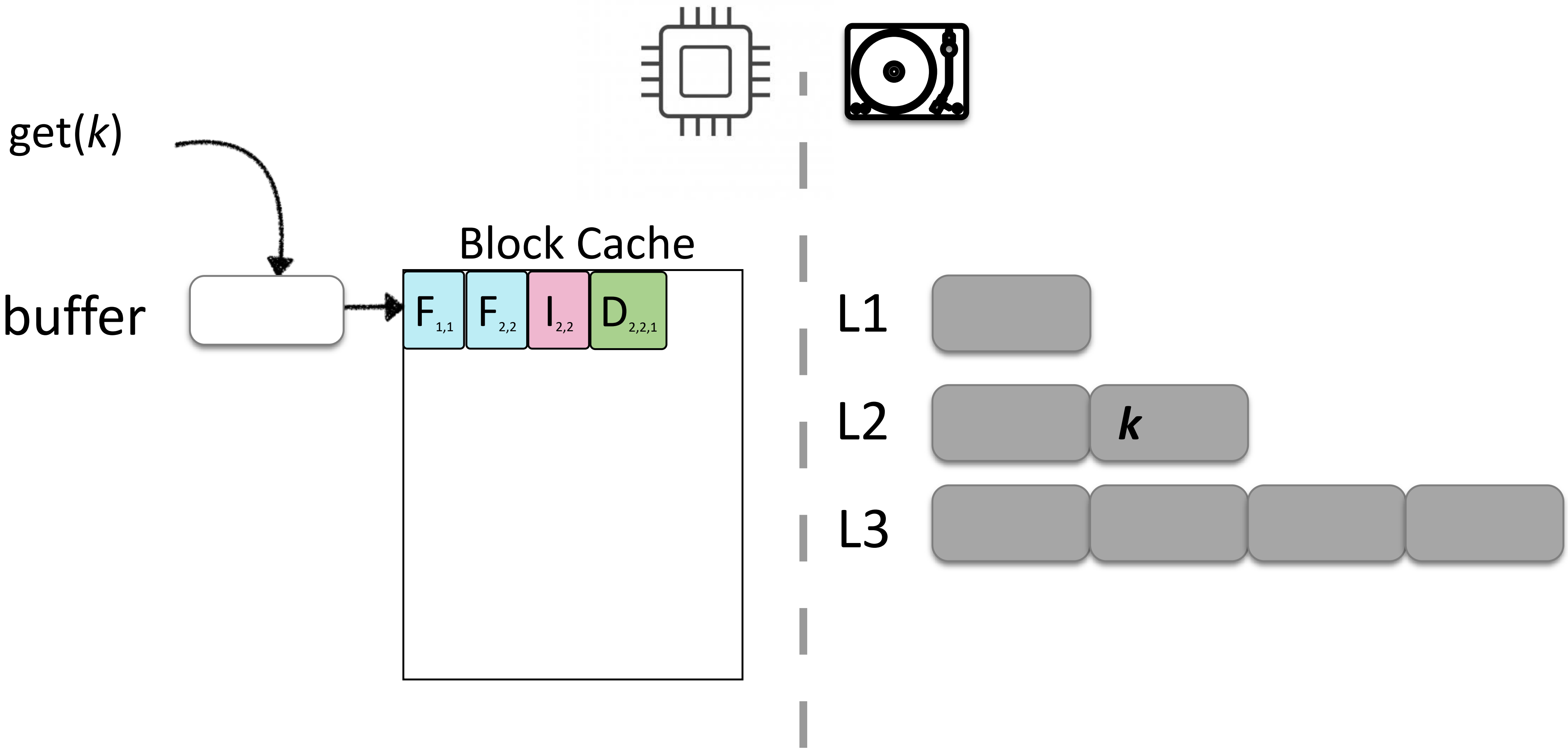
Log-Structured Merge Trees



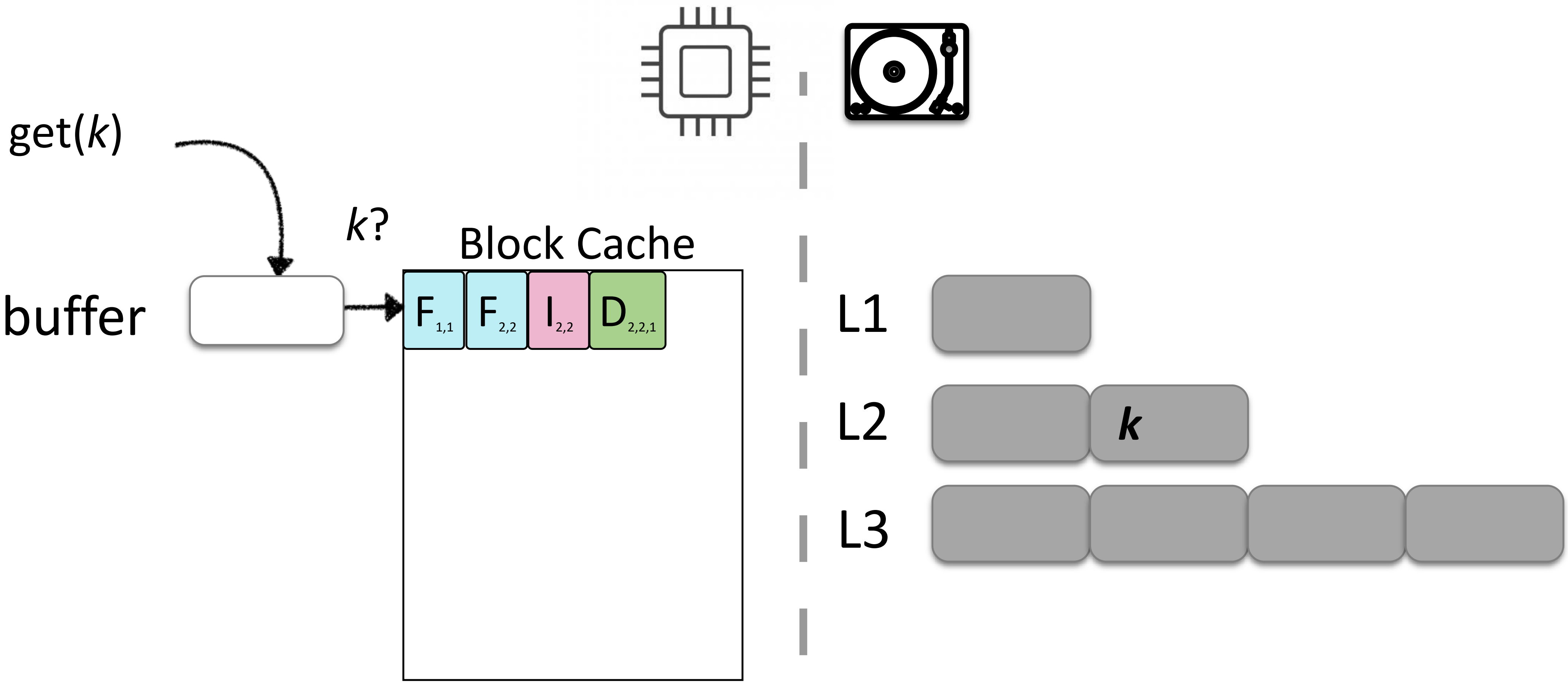
Log-Structured Merge Trees



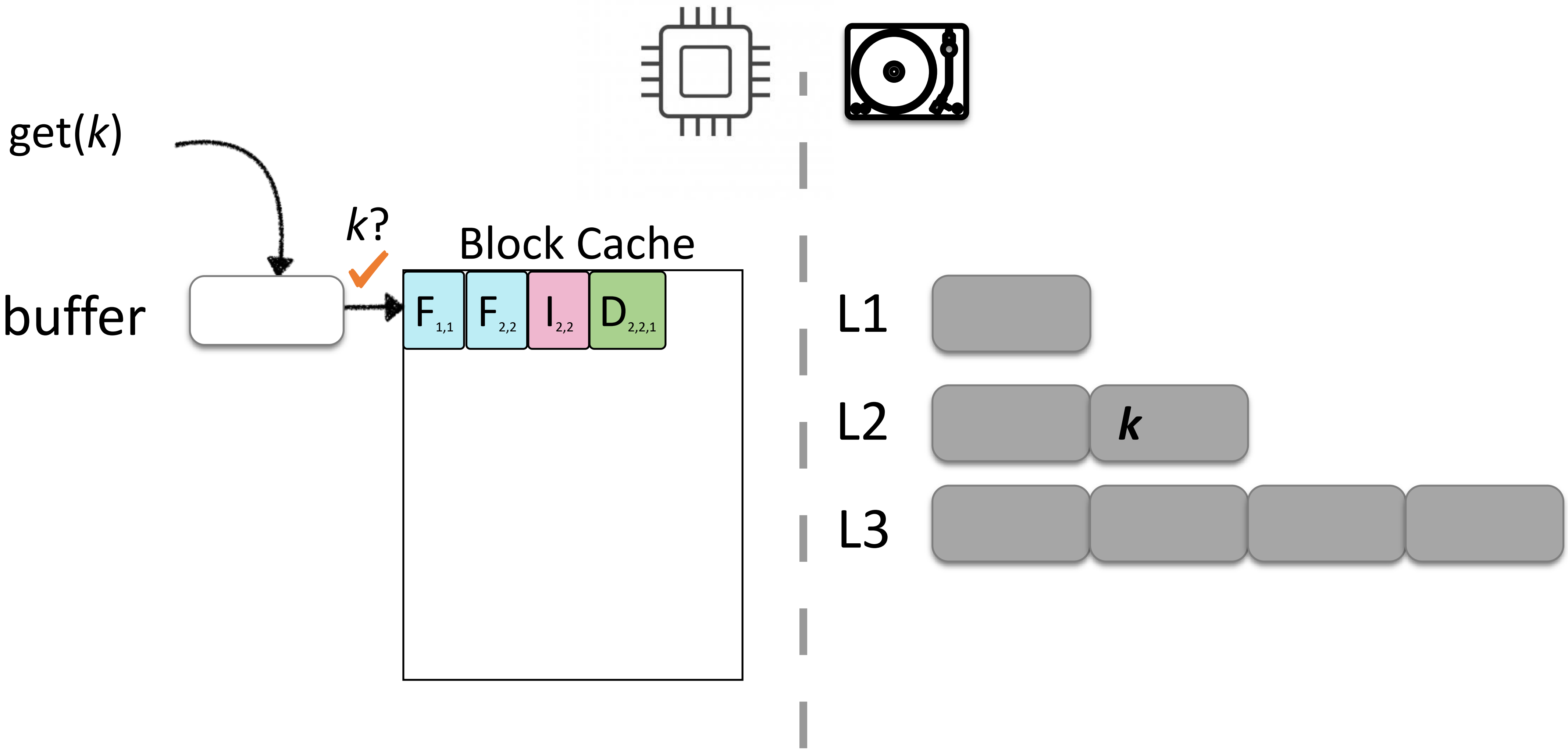
Log-Structured Merge Trees



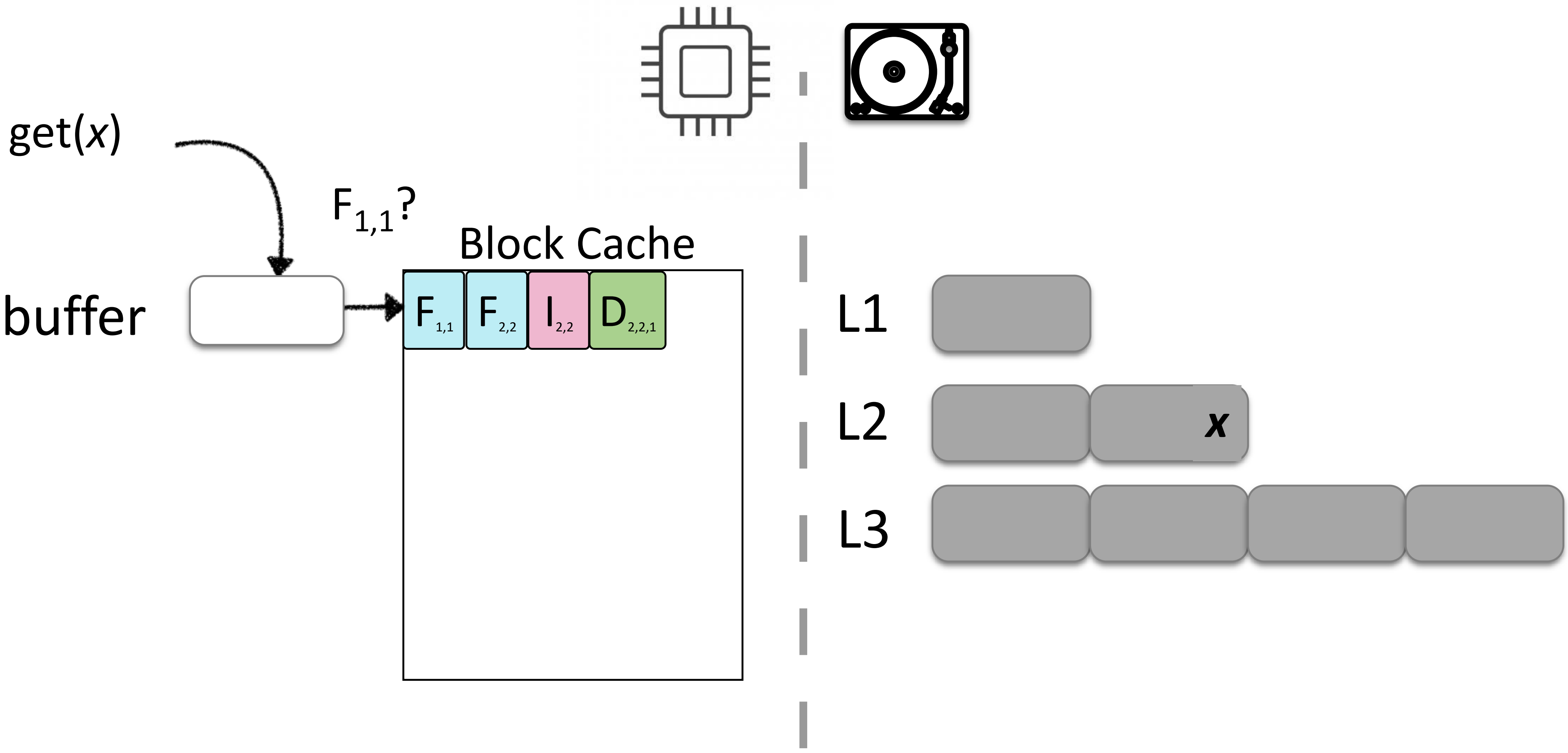
Log-Structured Merge Trees



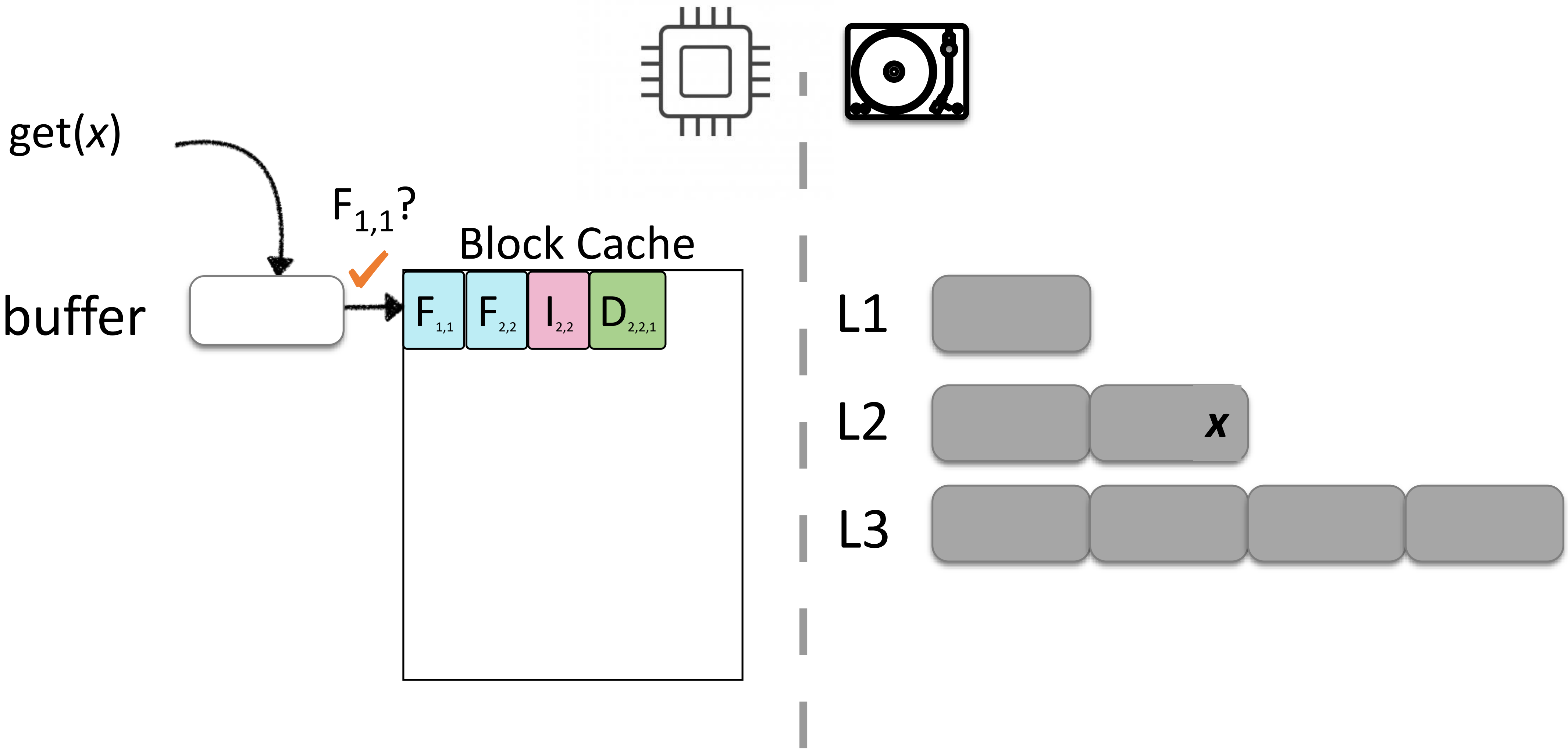
Log-Structured Merge Trees



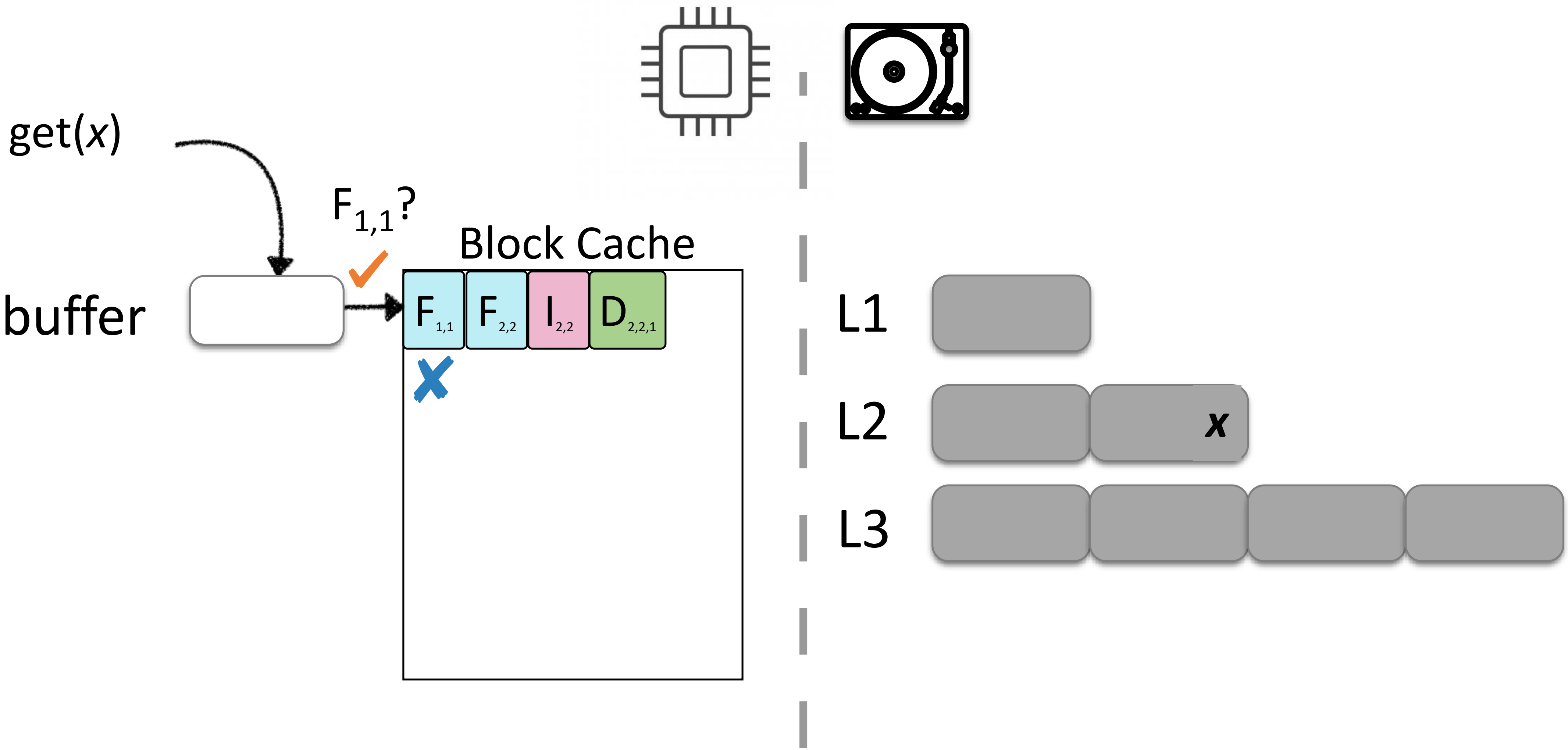
Log-Structured Merge Trees



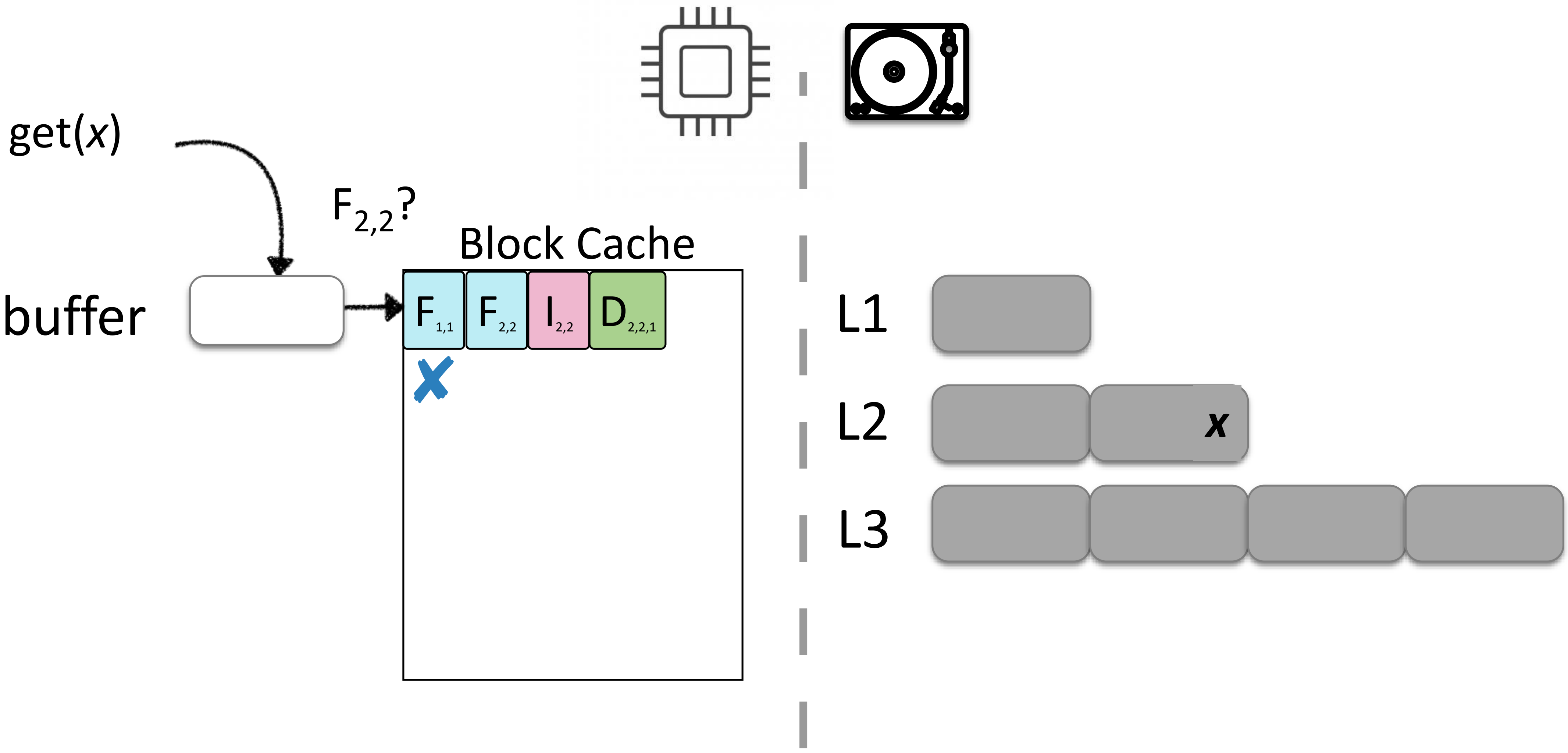
Log-Structured Merge Trees



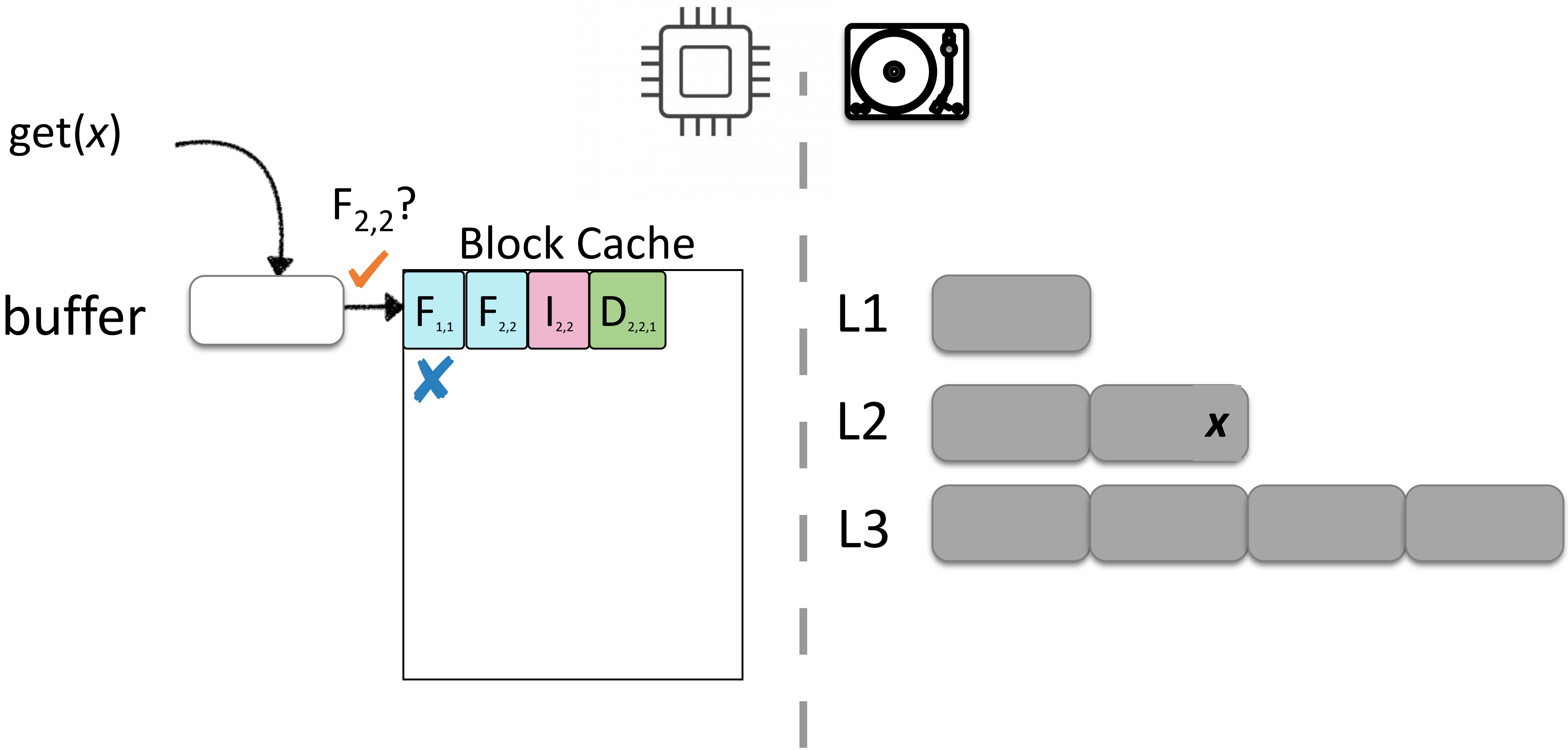
Log-Structured Merge Trees



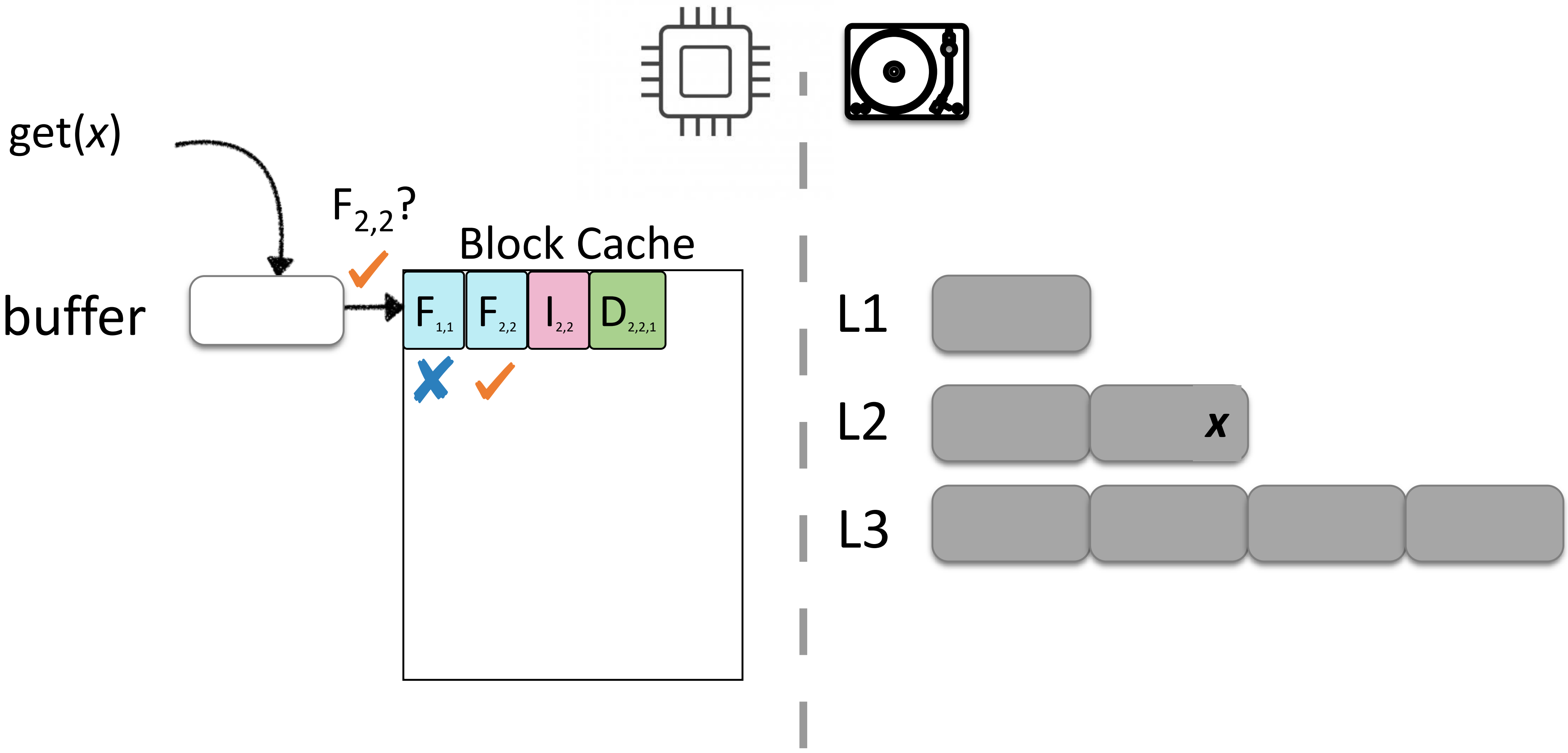
Log-Structured Merge Trees



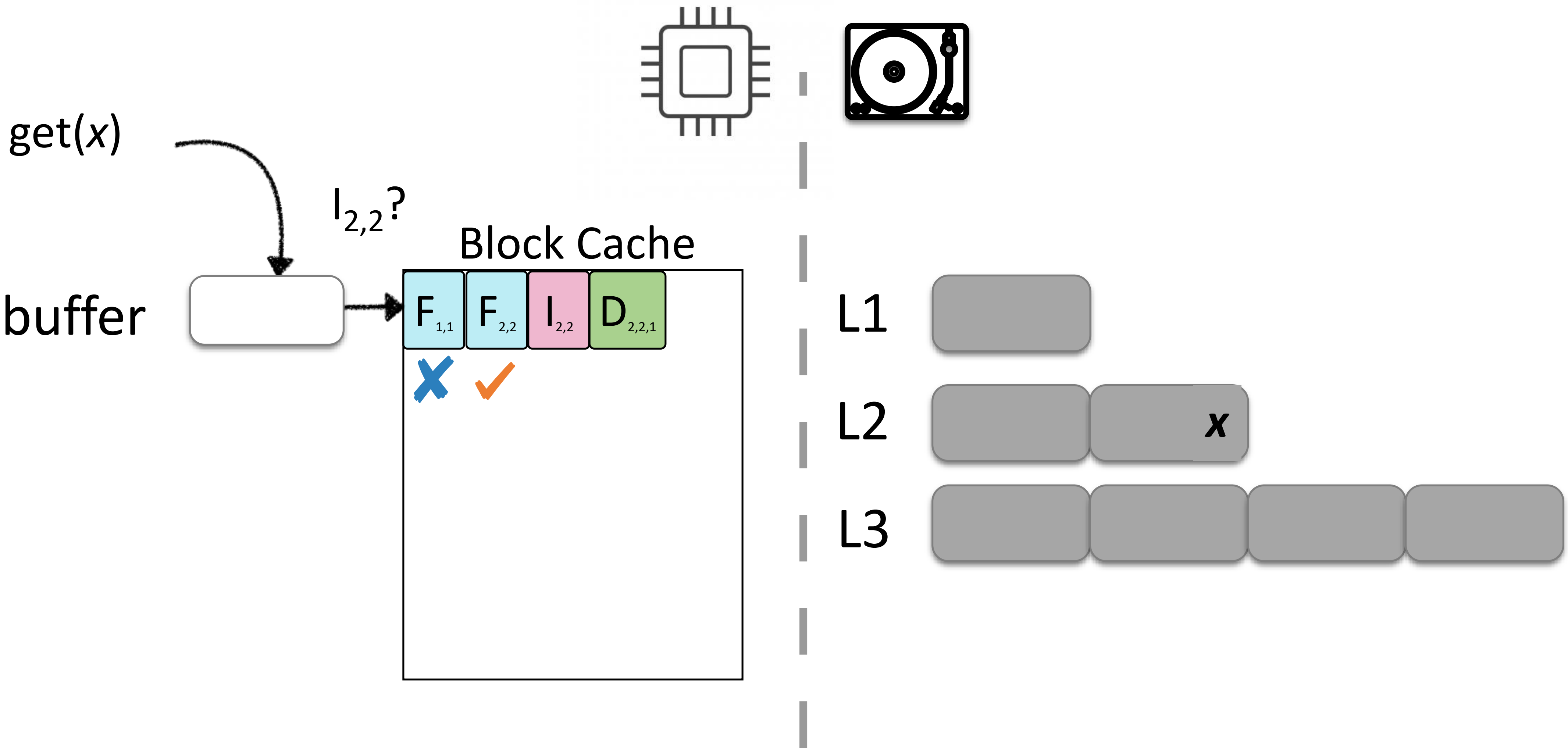
Log-Structured Merge Trees



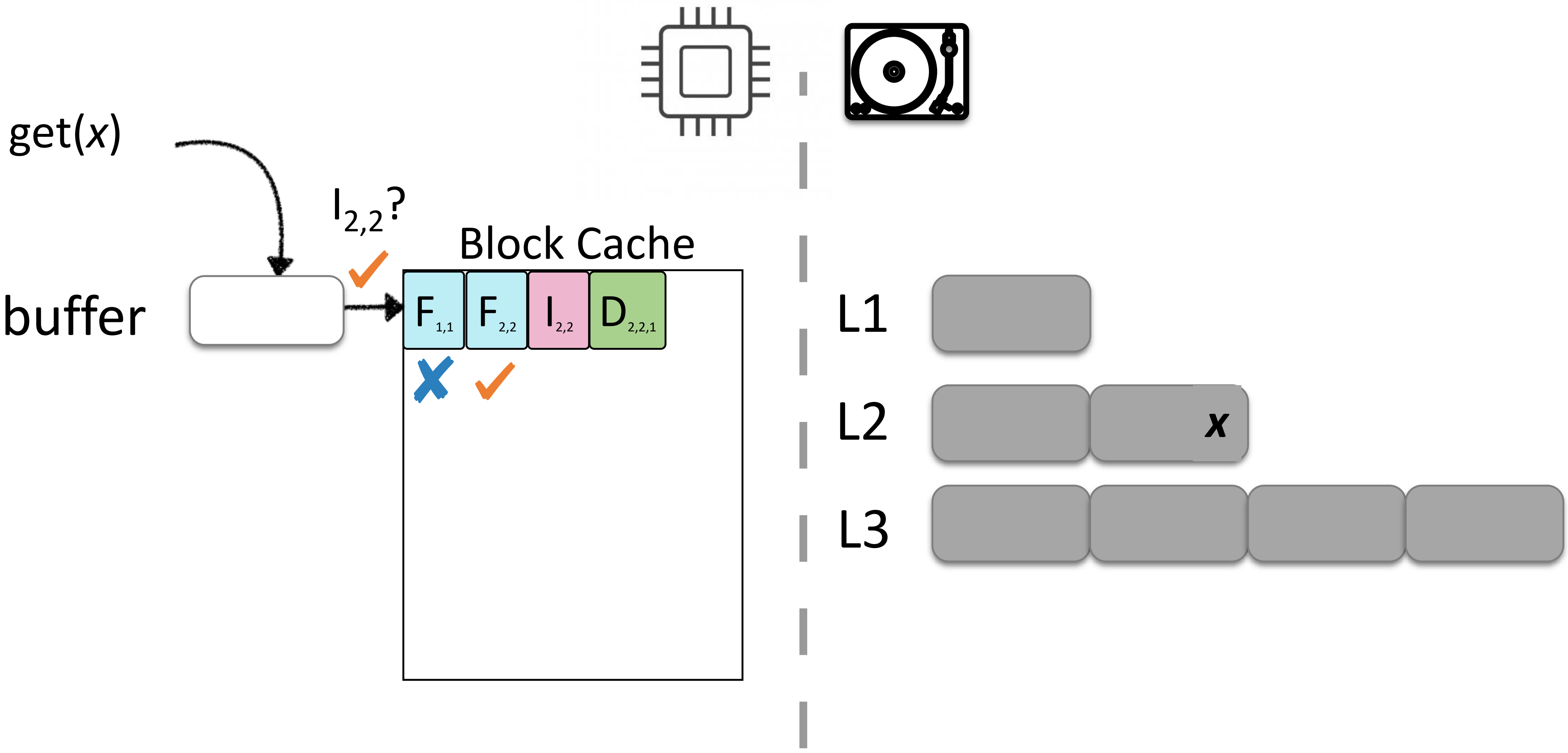
Log-Structured Merge Trees



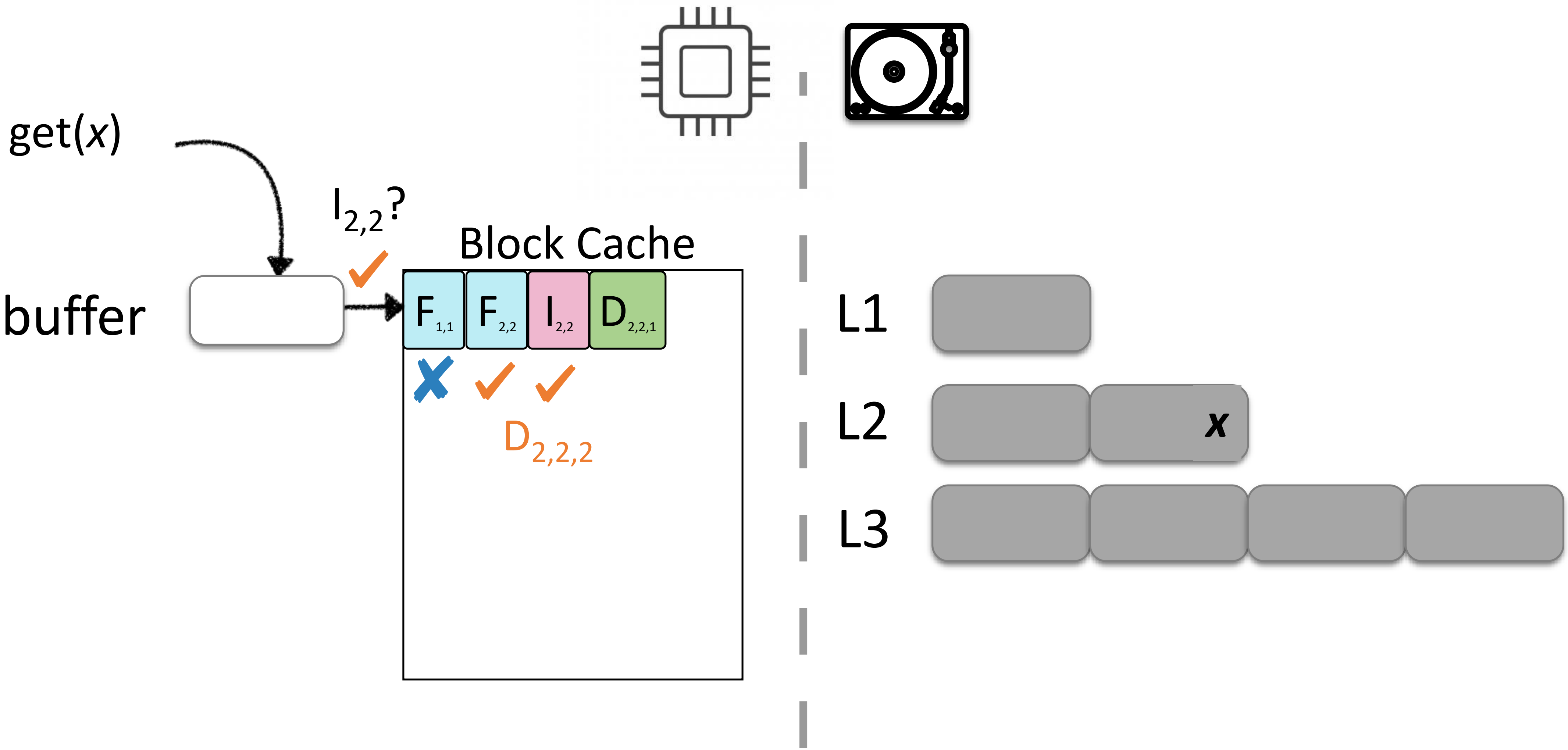
Log-Structured Merge Trees



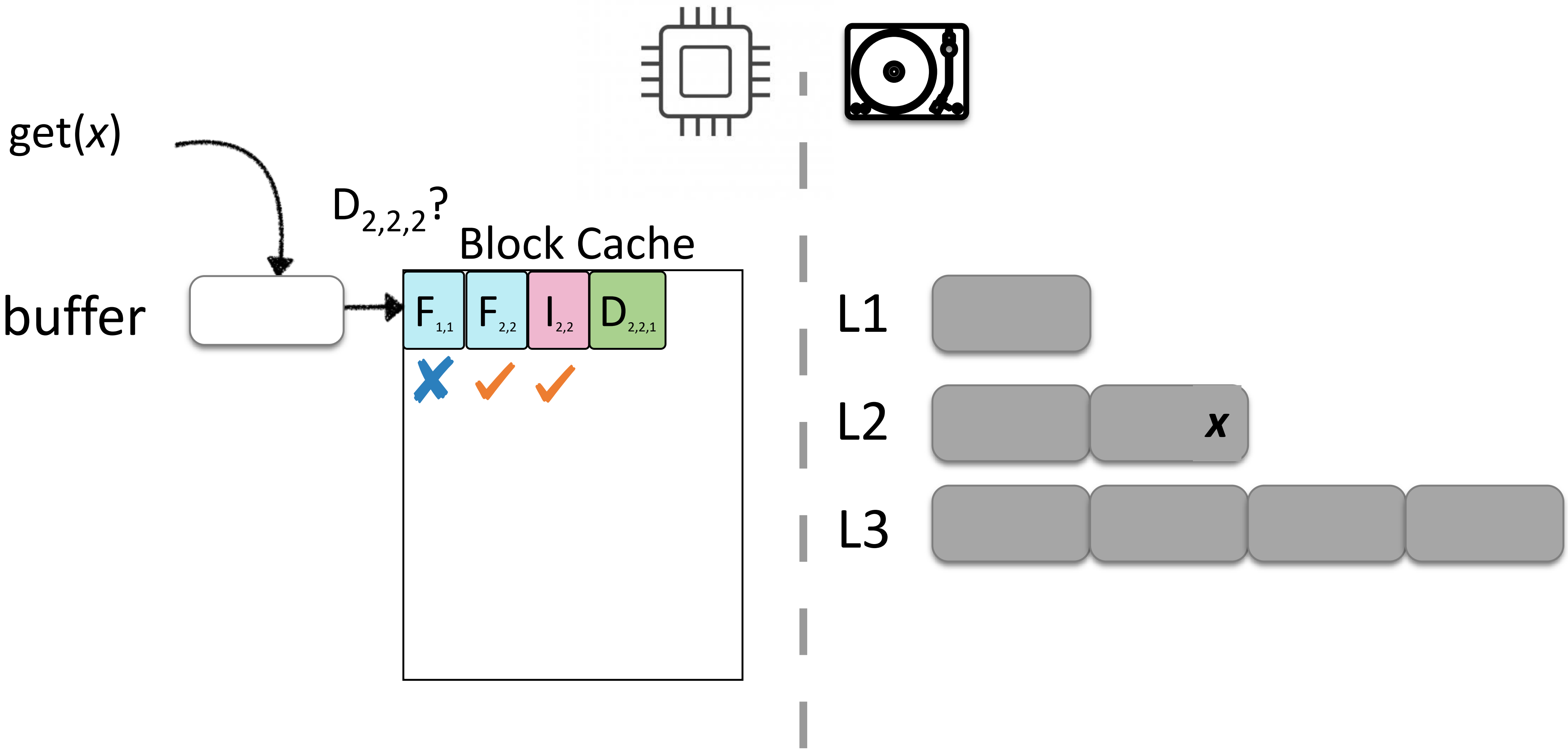
Log-Structured Merge Trees



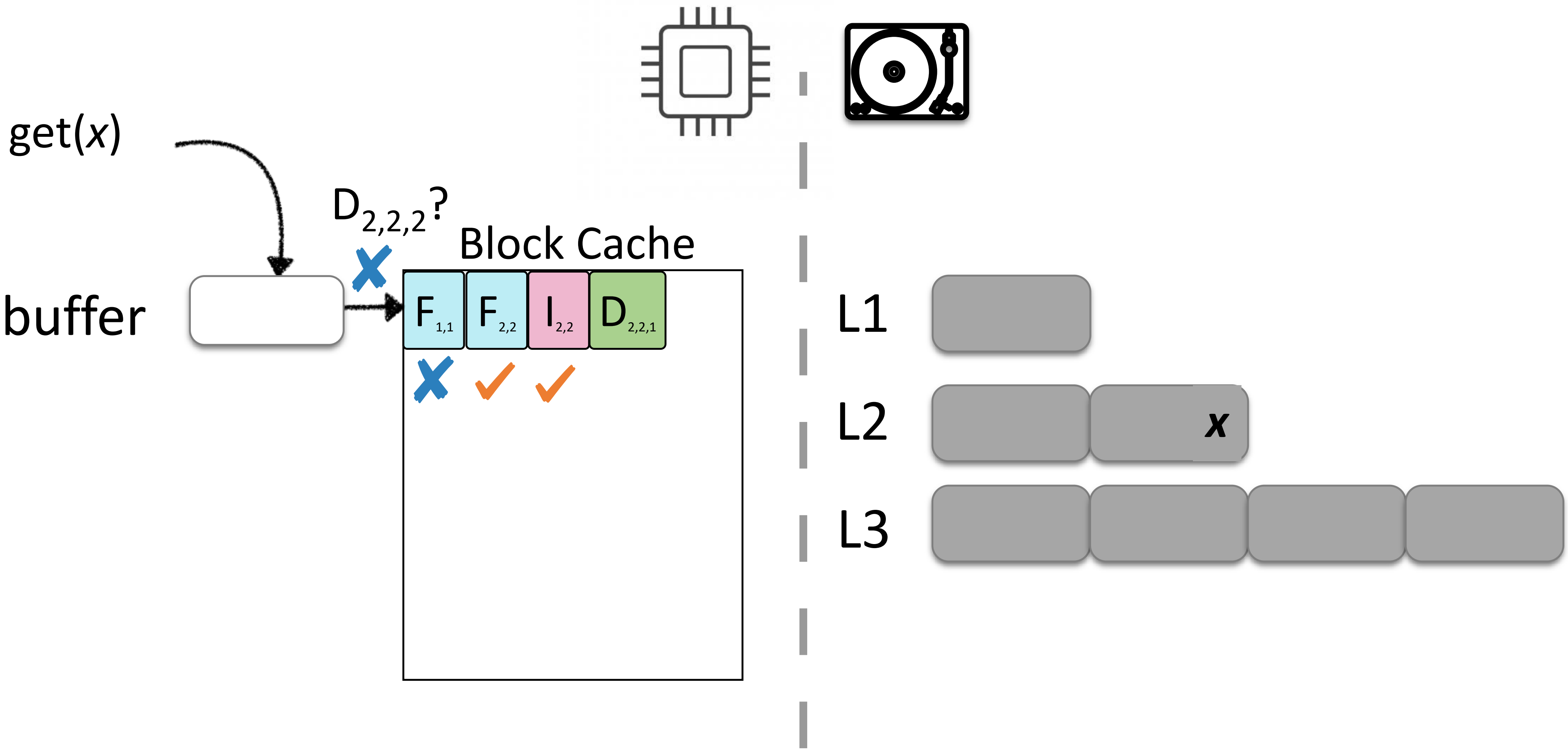
Log-Structured Merge Trees



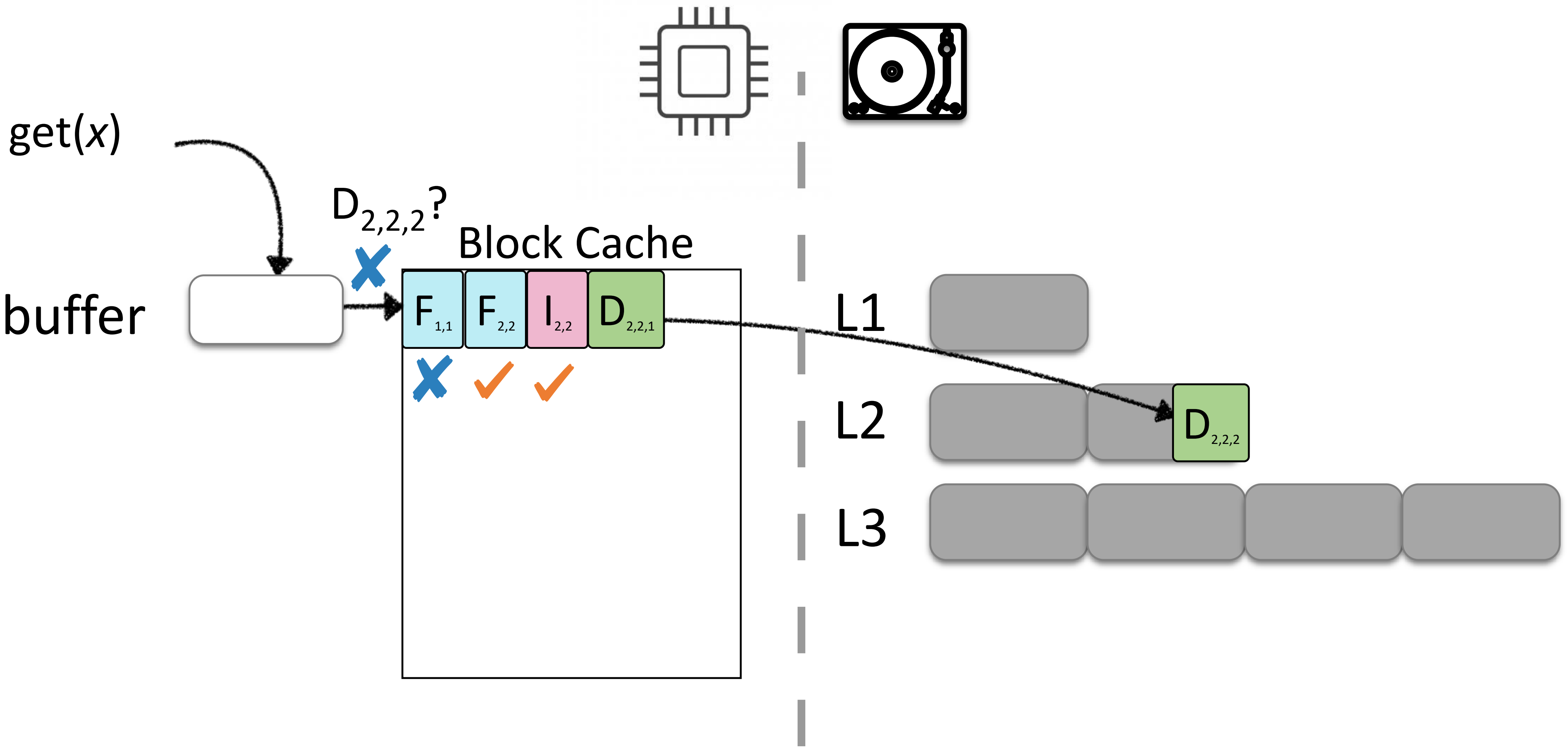
Log-Structured Merge Trees



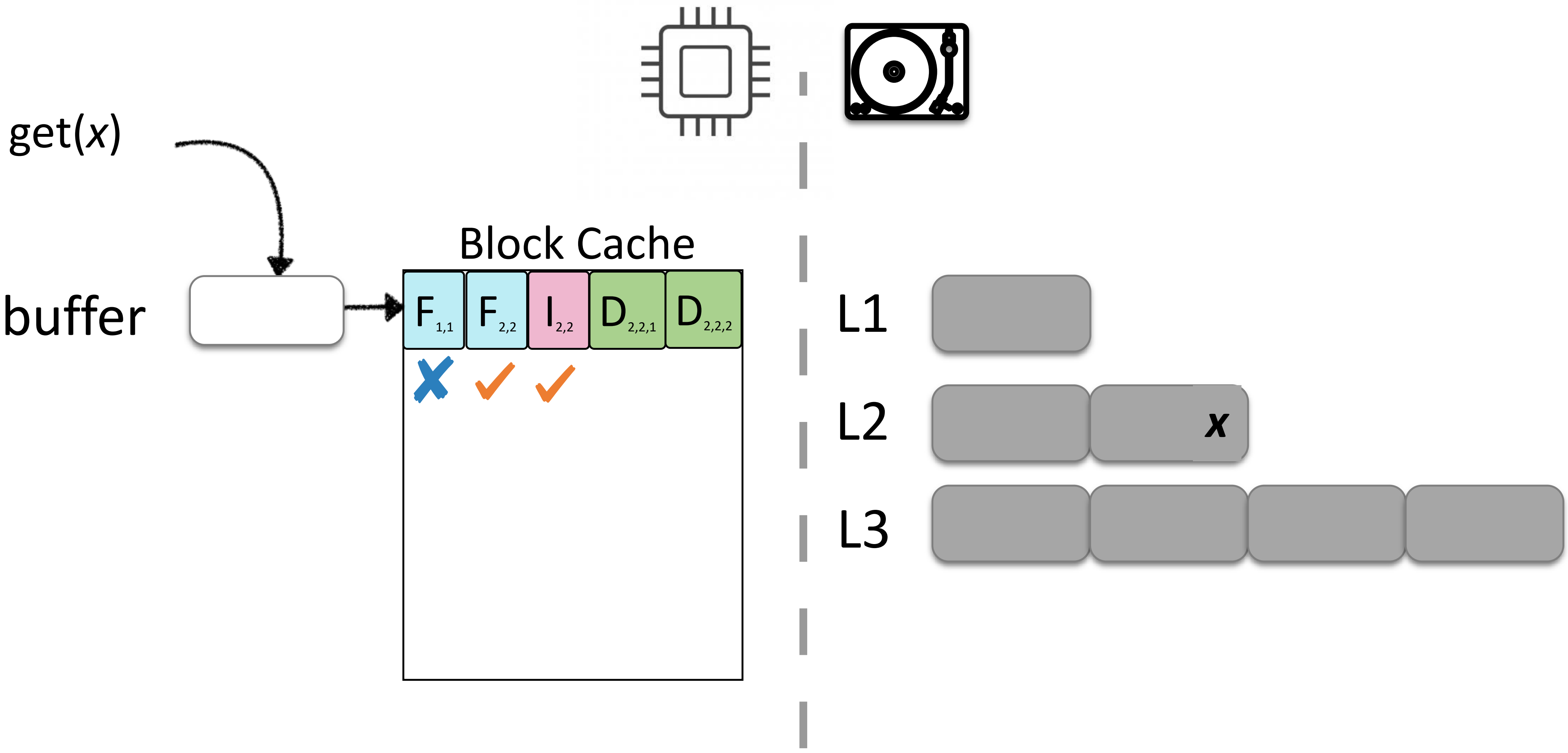
Log-Structured Merge Trees



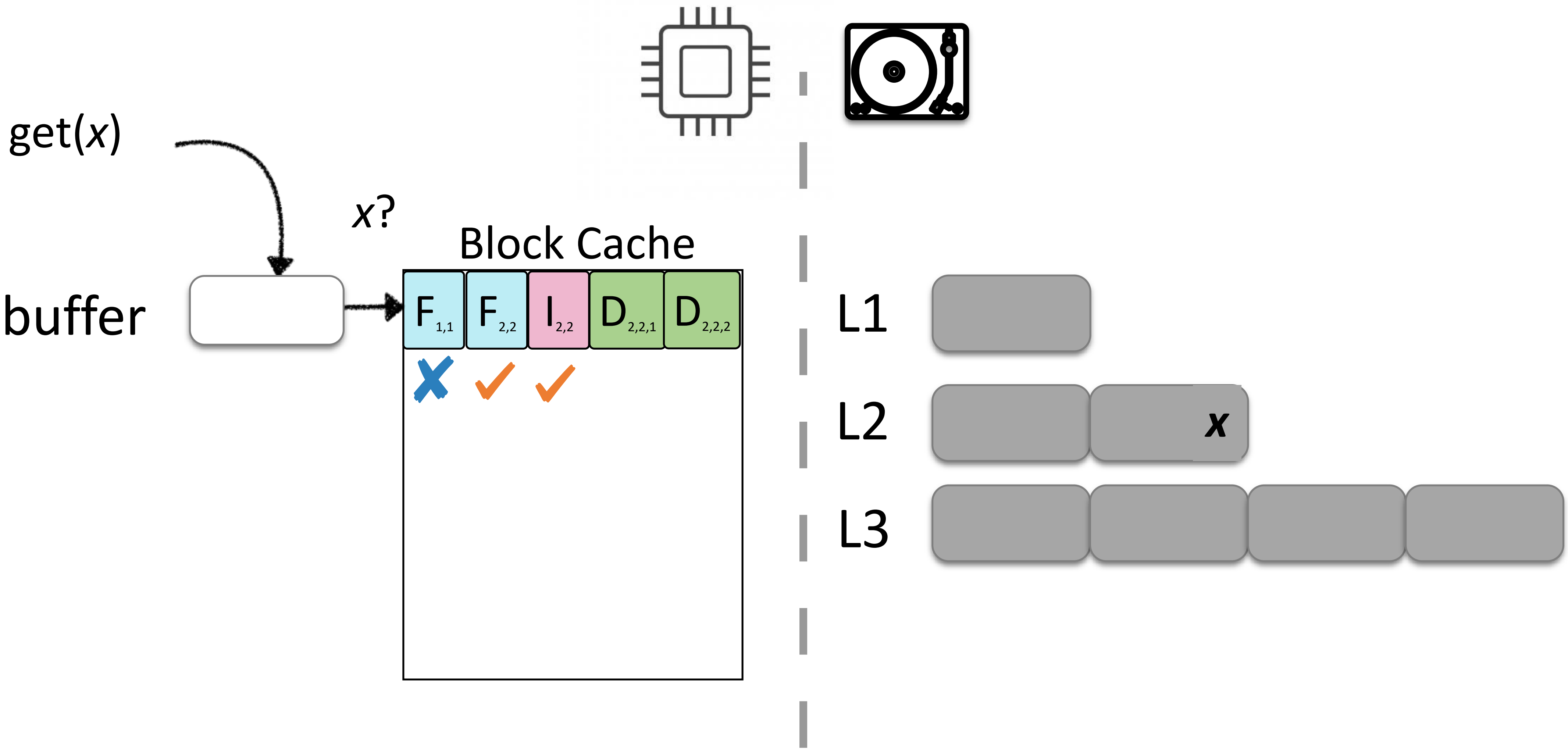
Log-Structured Merge Trees



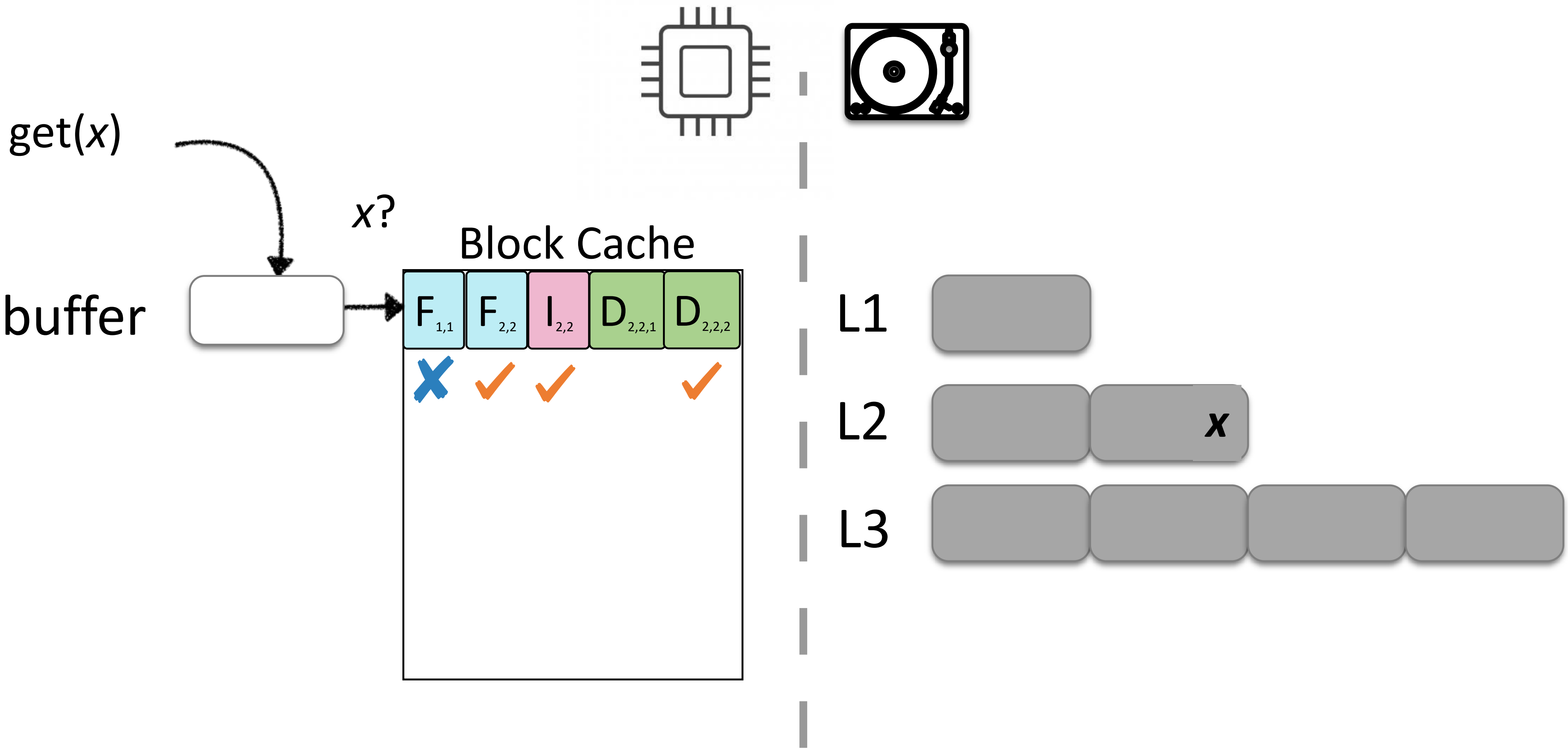
Log-Structured Merge Trees



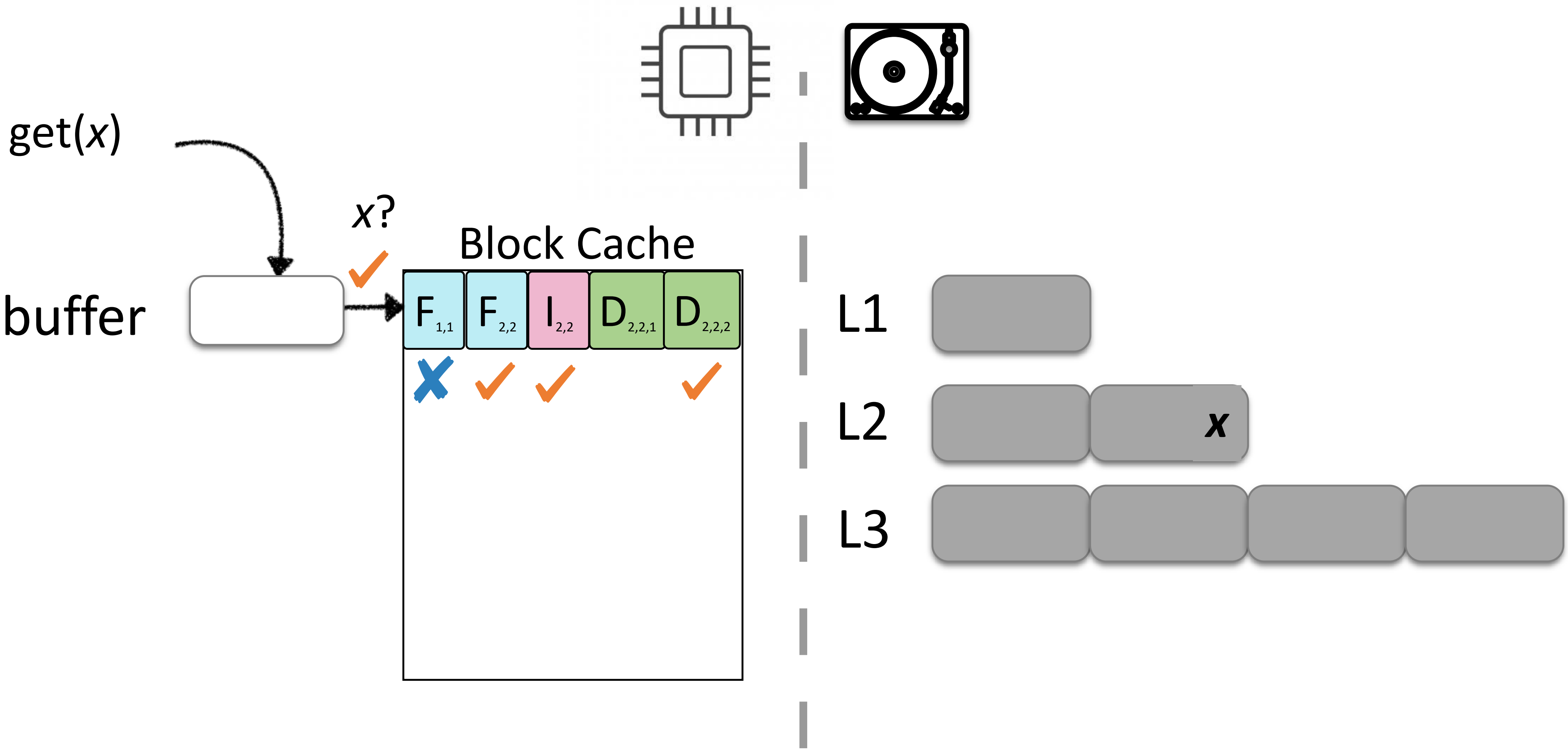
Log-Structured Merge Trees



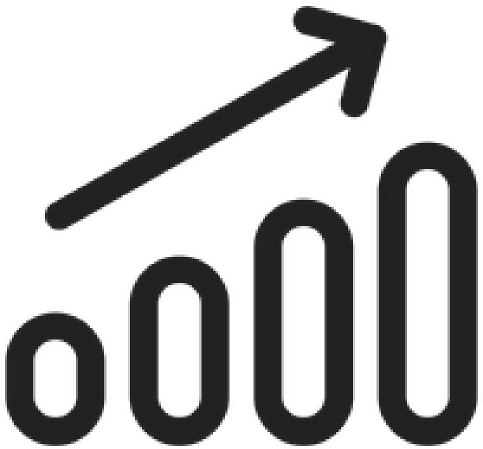
Log-Structured Merge Trees



Log-Structured Merge Trees



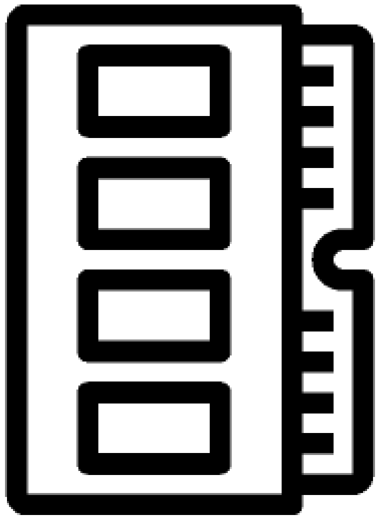
Memory Pressure in LSM-trees



Data size ↑

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

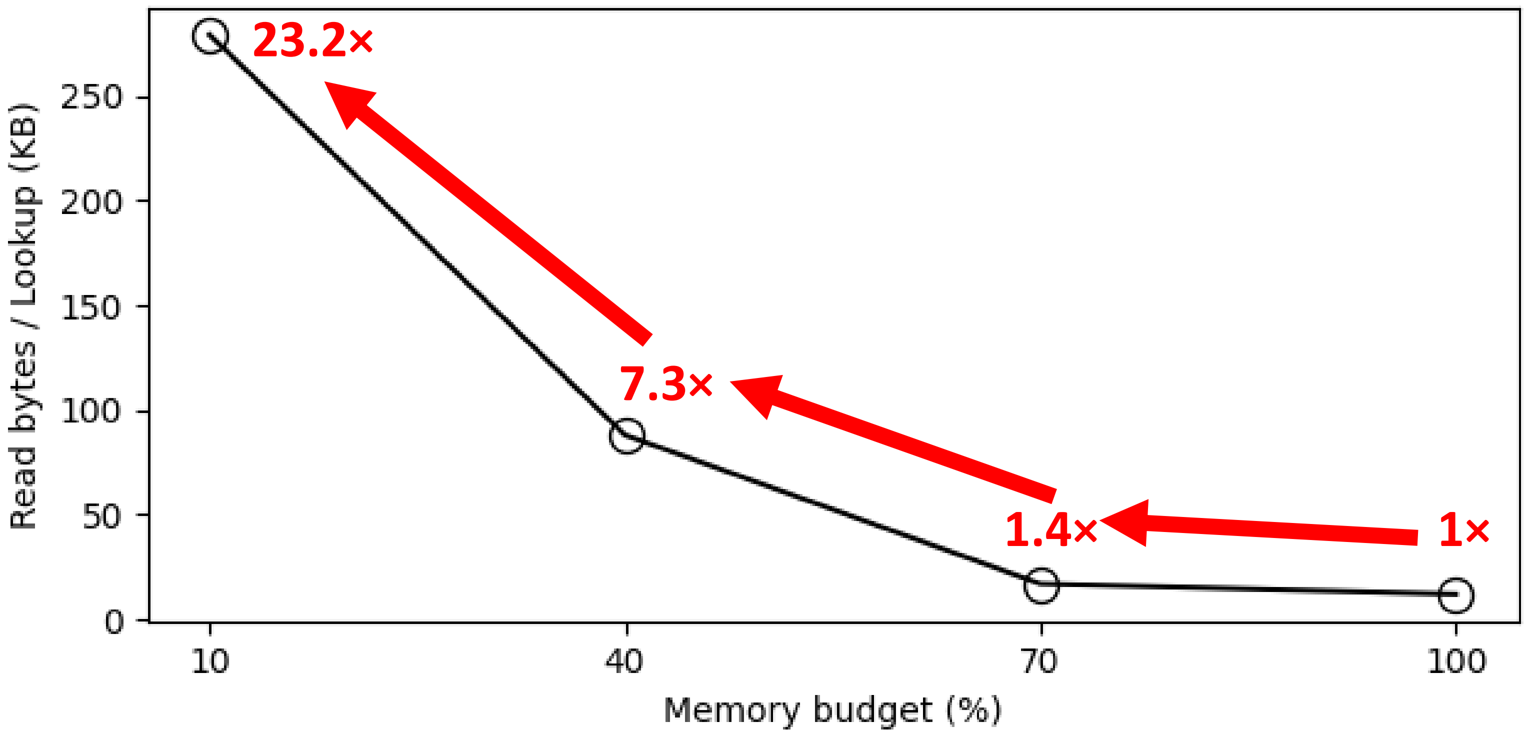
11% space amplification,
1KB entry, 64B key, bpk 10



memory-to-data ratio ↓

Memory pressure

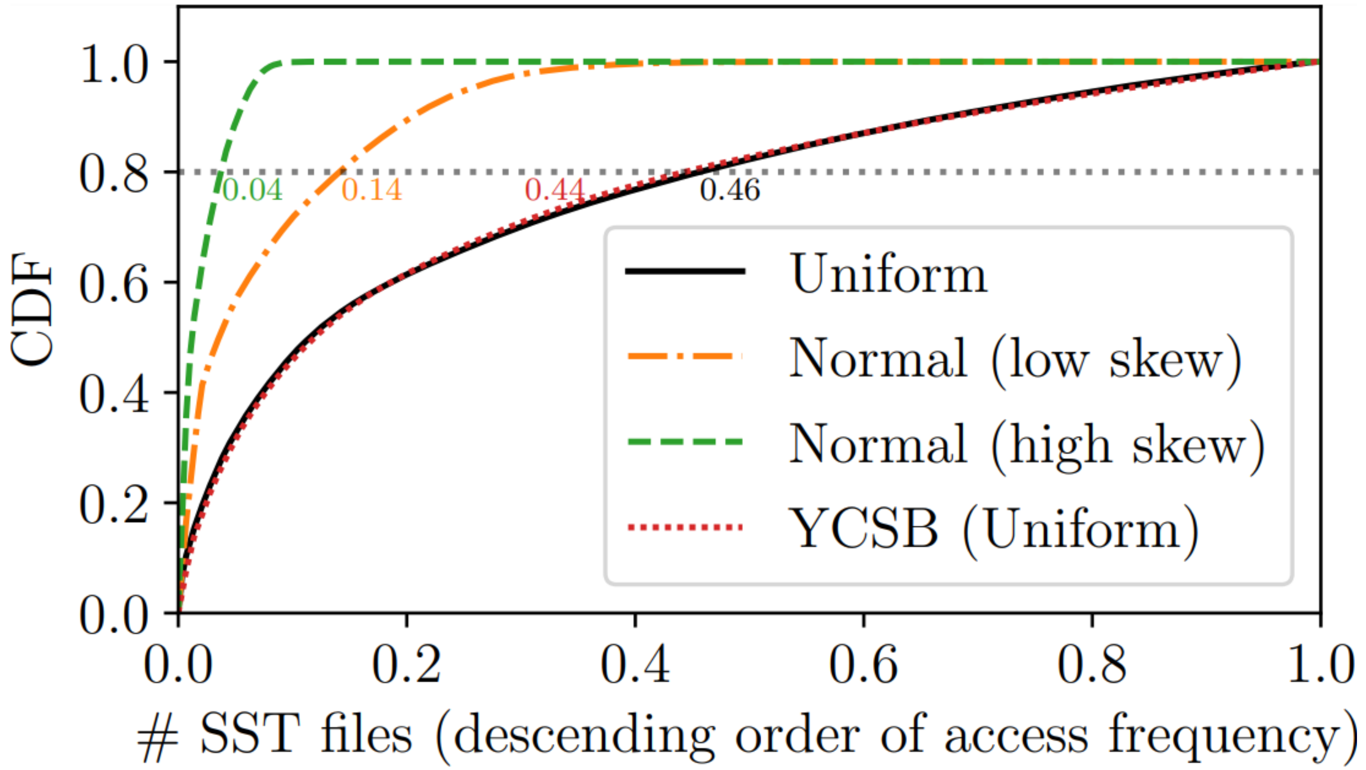
Lookup cost under memory pressure



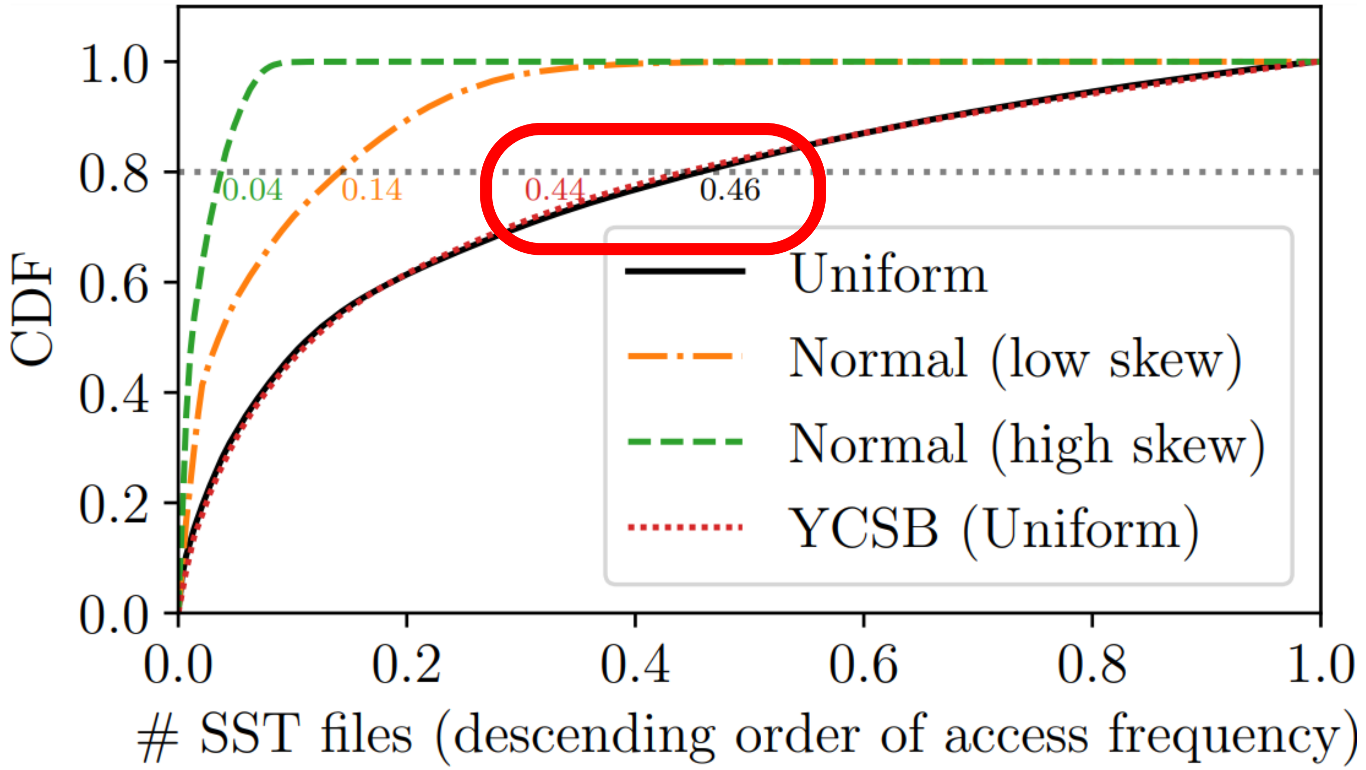
As the available memory decreases, the read bytes per query increase rapidly.

Are all filter blocks equally important?

Access Frequency Patterns



Access Frequency Patterns



Even in a perfectly uniform workload,

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

[Monkey, SIGMOD 2017]

worst-case I/O cost

$$O\left(\sum \boldsymbol{p}\right)$$

memory

Bloom filters

$\boldsymbol{p}$

$\boldsymbol{p}$

$\boldsymbol{p}$

$$\text{false positive } \boldsymbol{p} = e^{-\frac{\text{bits } M}{\text{entries } N} \cdot ln(2)^2}$$

[Monkey, SIGMOD 2017]

memory

Bloom filters

p

p

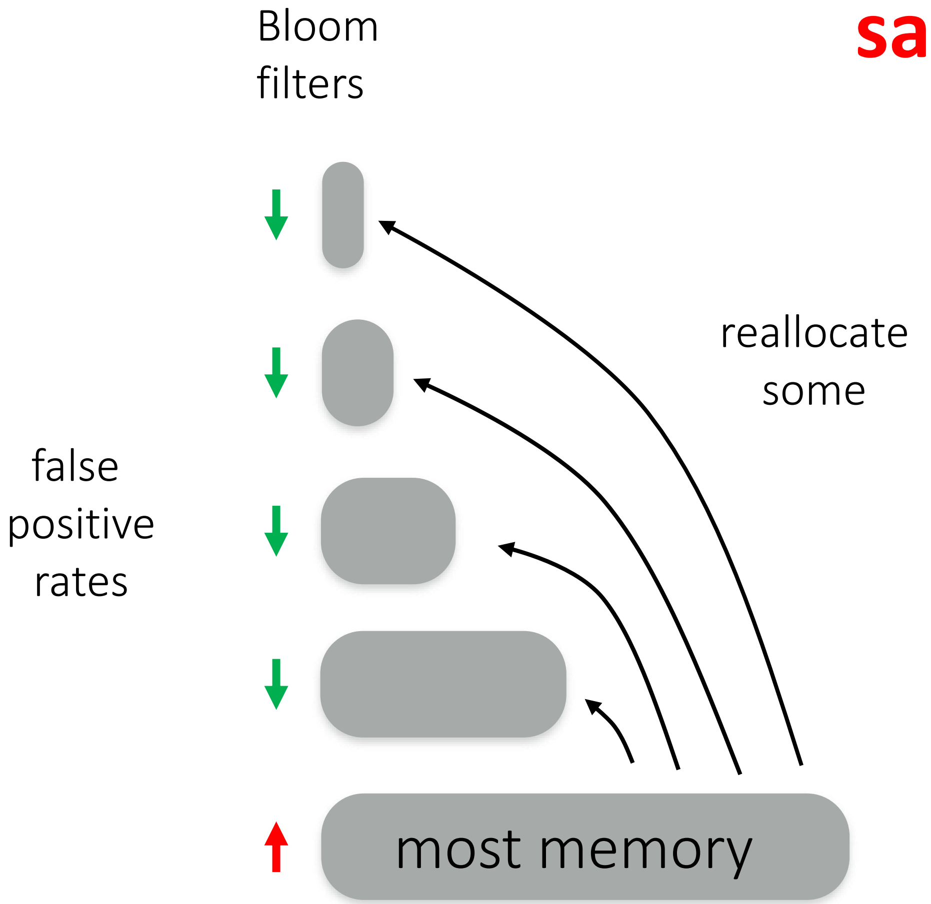
p

worst-case I/O cost

$$O\left(log(N) \cdot e^{-M/N}\right)$$

can we do better?

same memory, fewer I/O



[Monkey, SIGMOD 2017]

relax model optimize

$0 < p_2 < 1$

$0 < p_1 < 1$

$0 < p_0 < 1$

$\text{lookup cost} = \sum p_i$

$\text{memory footprint} = f(p_0, p_1, \dots)$

in terms of $\boldsymbol{p_0}, \boldsymbol{p_1}, \dots$

Bloom filters

memory footprint

[Monkey, SIGMOD 2017]

…

p_2

false positive rates

p_1

p_0

$$\text{false positive } p = e^{-\frac{\text{bits } M}{\text{entries } N} \cdot ln(2)^2}$$

$$\Downarrow$$

$$\text{bits}(\boldsymbol{p}, \boldsymbol{N}) = -\frac{ln(\boldsymbol{p})}{ln(2)^2} \cdot \boldsymbol{N}$$

$$\text{lookup cost} = \sum p_i$$

[Monkey, SIGMOD 2017]

Bloom filters

memory footprint

$$\text{bits}(\boldsymbol{p}, \boldsymbol{N}) = -\frac{ln(\boldsymbol{p})}{ln(2)^2} \cdot \boldsymbol{N}$$

…

…

p_2 $\text{bits}(p_2, N/T^2)$

false positive rates p_1 $\text{bits}(p_1, N/T)$

p_0 $\text{bits}(p_0, N\ )$

$$\text{lookup cost} = \sum p_i$$

$$\textbf{memory} = -\frac{\boldsymbol{N}}{ln(2)^2} \cdot \sum \frac{ln(p_i)}{T^i}$$

74

[Monkey, SIGMOD 2017]

Bloom filters

minimize: $$\text{lookup cost} = \sum p_i$$

w.r.t. $$\boldsymbol{M} = -\frac{\boldsymbol{N}}{ln(2)^2} \cdot \sum \frac{ln(p_i)}{\boldsymbol{T}^i}$$

...

p_2

false positive rates p_1

p_0

[Monkey, SIGMOD 2017]

Monkey

Bloom filters

...

p_0/T^2

p_0/T

p_0

false positive rates

exponential decrease

State-of-the-art

Bloom filters

...

p

p

p

same

[Monkey, SIGMOD 2017]

Monkey
Bloom filters

...

false positive rates

State-of-the-art
Bloom filters

...

Monkey		State-of-the-art
...	$<$	...
p_0/T^2	$<$	p
p_0/T	$<$	p
p_0	$>$	p

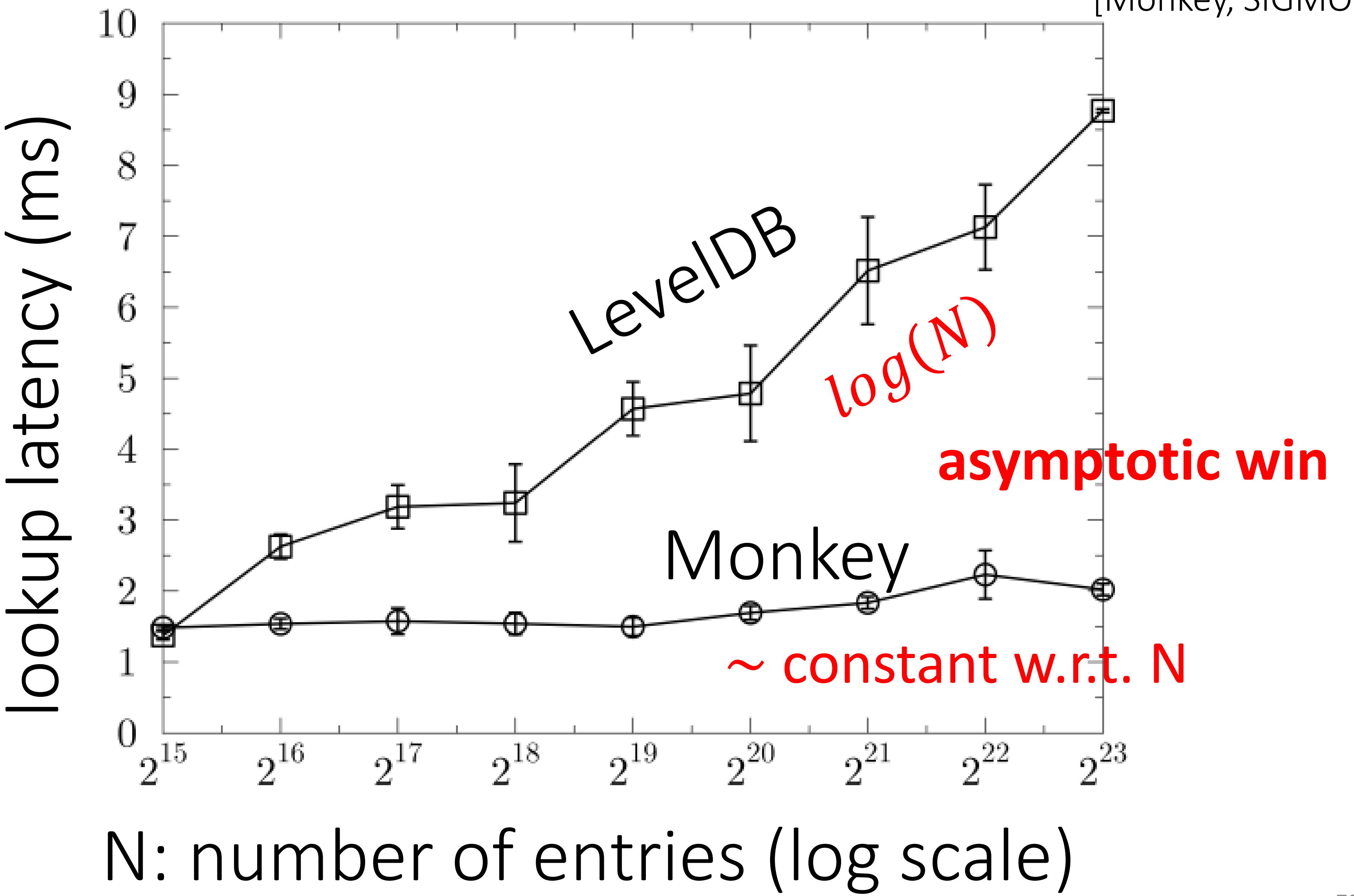
$$\text{lookup cost} = \sum p_i \quad < \quad = \sum p$$

$$= O\left(e^{-M/N}\right) \qquad = O\left(log(N) \cdot e^{-M/N}\right)$$

asymptotic win

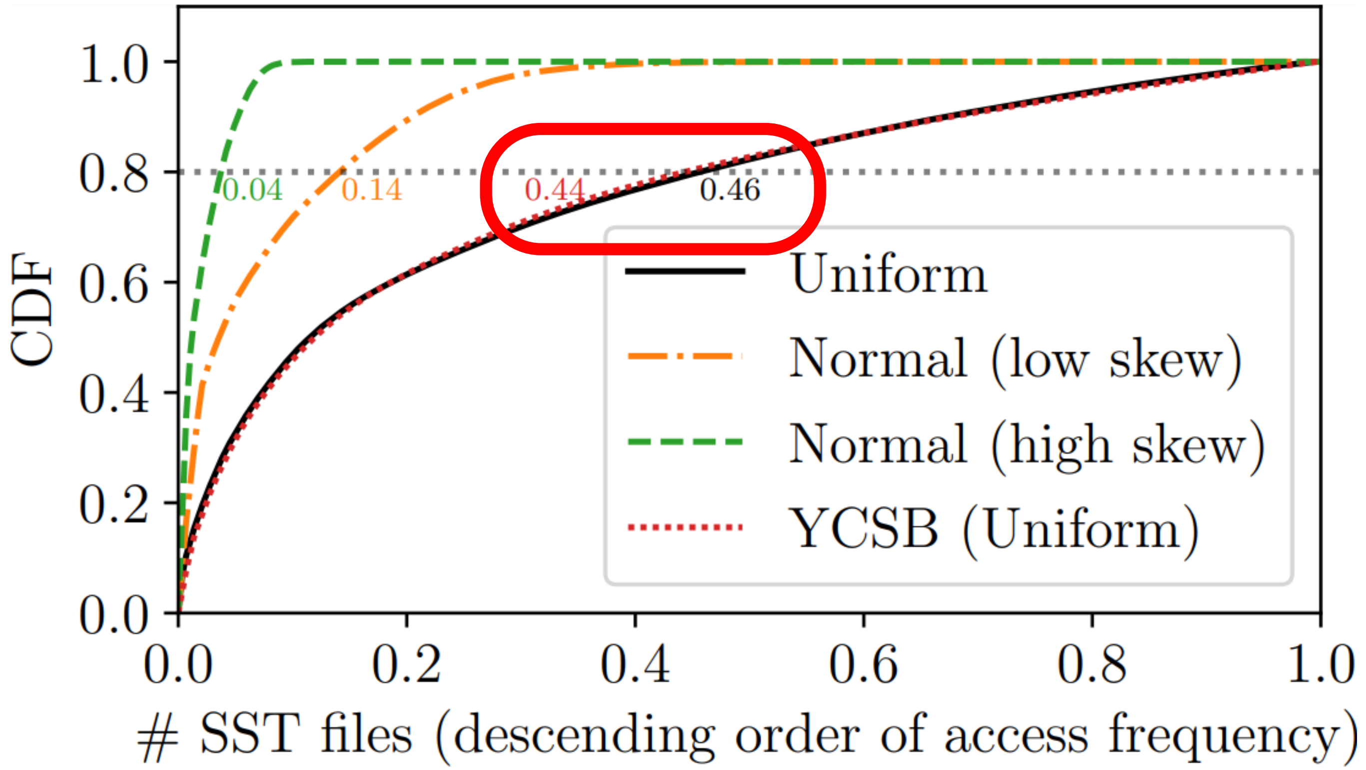
lookup cost increases at slower rate as data grows

[Monkey, SIGMOD 2017]



78

Access Frequency Patterns

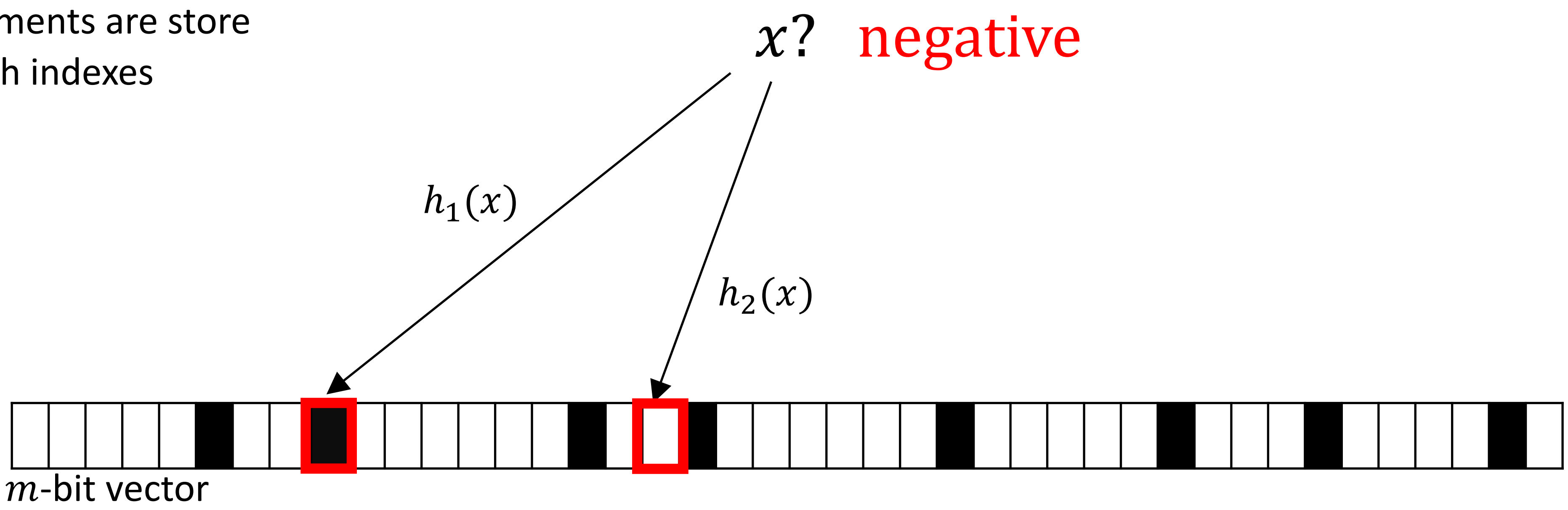


Even in a perfectly uniform workload,

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

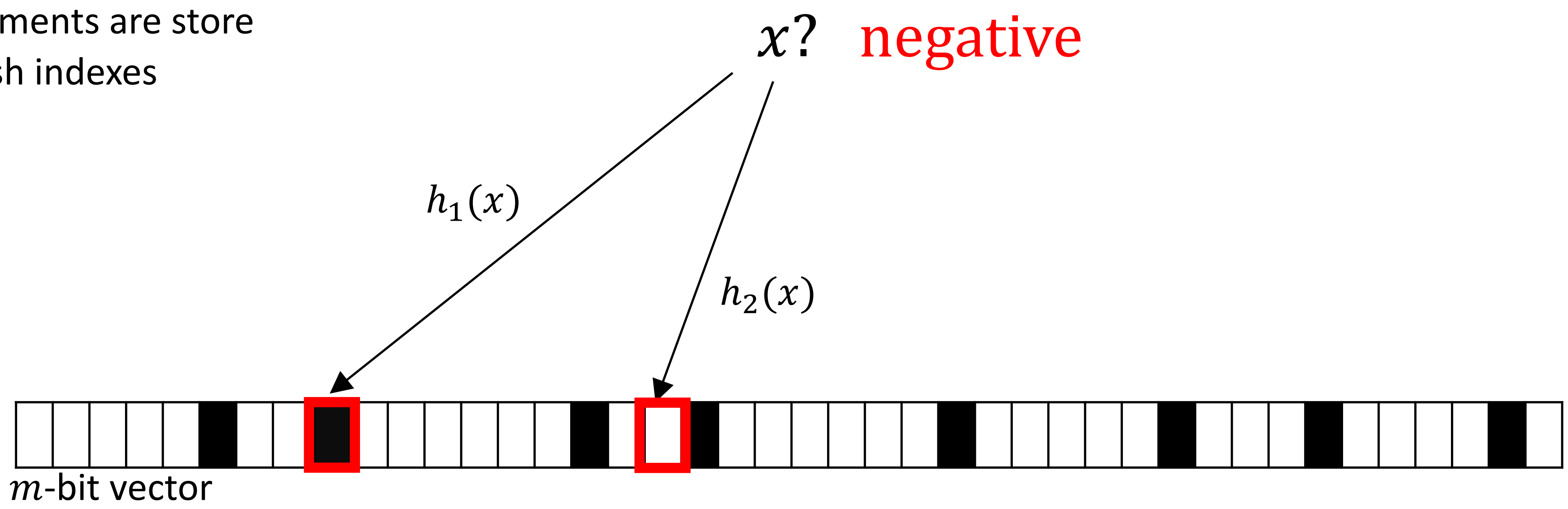
Bloom Filter

m-bit vector
n elements are store
k hash indexes



Bloom Filter

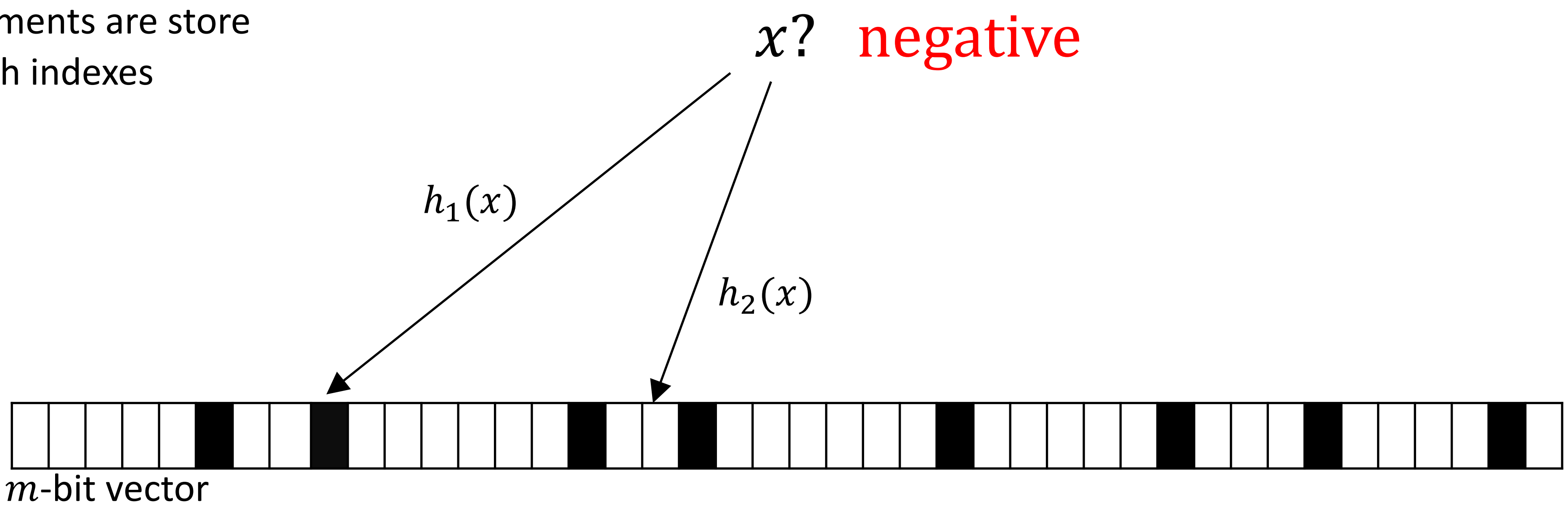
m-bit vector
n elements are store
k hash indexes



Is the entire filter useful?

Bloom Filter

m-bit vector
n elements are store
k hash indexes

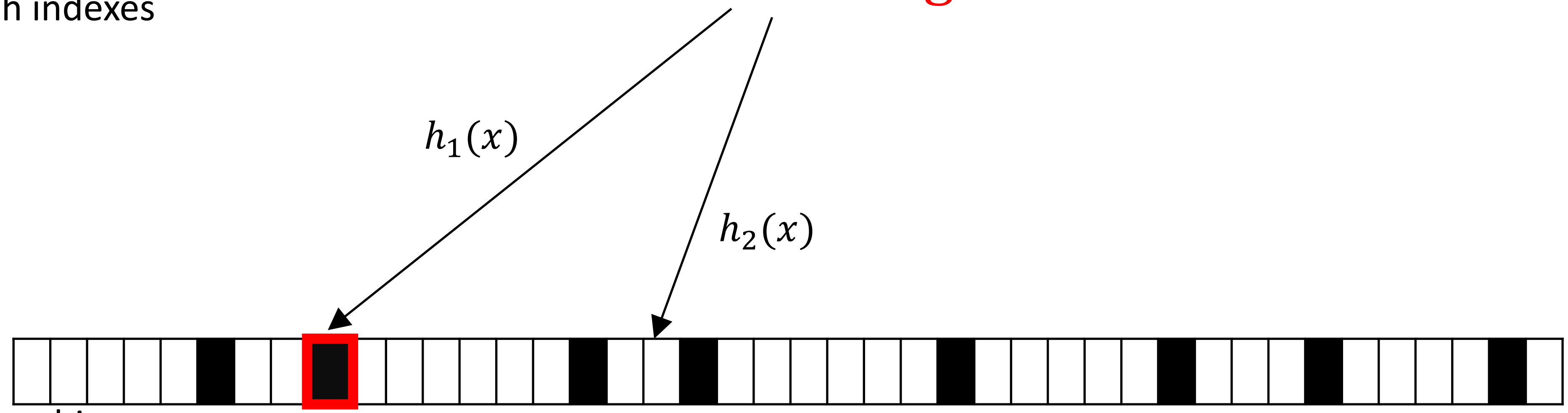


$$probes_{empty} = 1 + \frac{1}{2} + \frac{1}{2^2} + \cdots + \frac{1}{2^{k-1}} = \sum_{d=1}^{k} \frac{1}{2^{k-1}} = 2 - \frac{1}{2^{k-1}}$$

Bloom Filter

m-bit vector
n elements are store
k hash indexes

$x?$ negative



m-bit vector

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

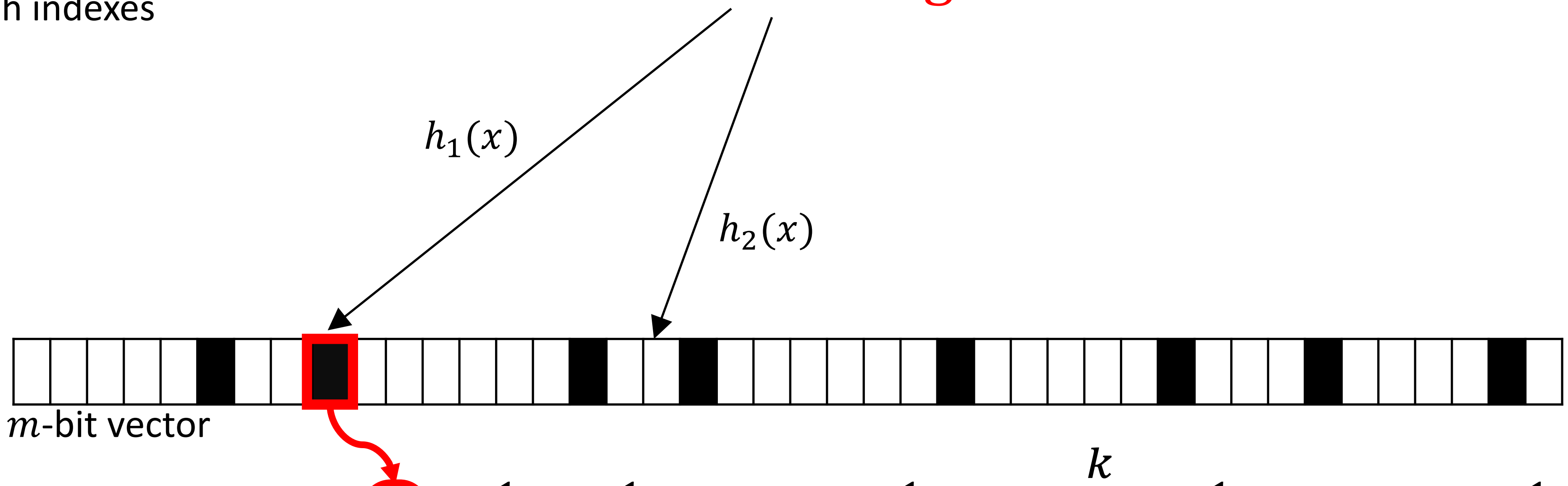
Bloom Filter

m-bit vector

n elements are store

k hash indexes

x? negative



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

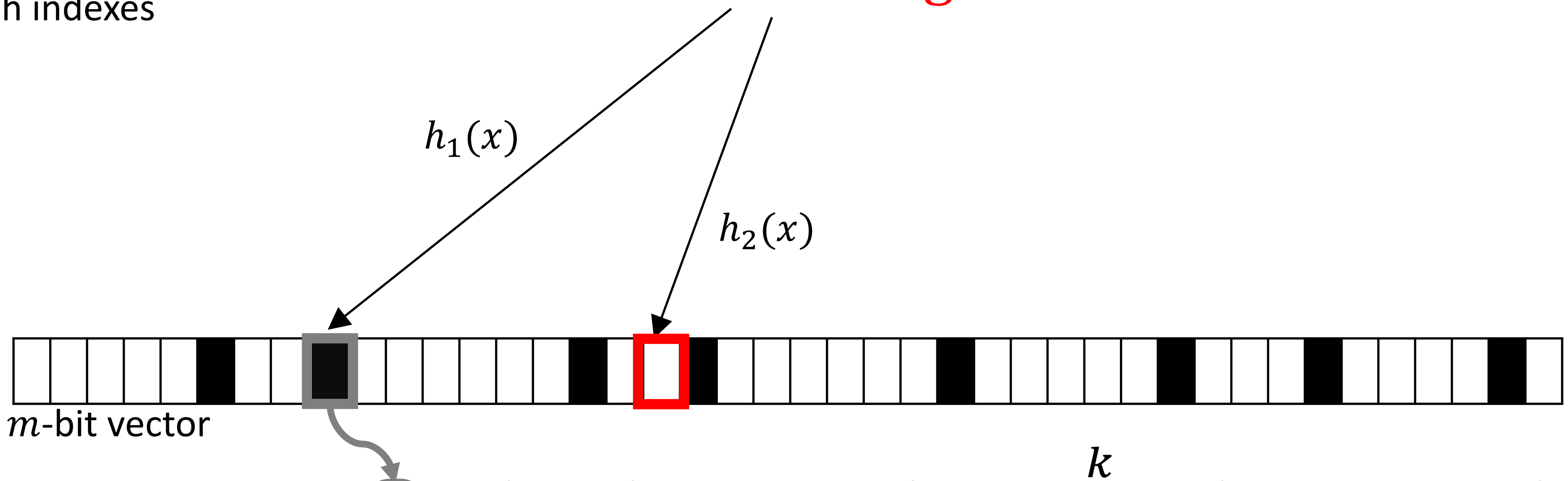
Bloom Filter

m-bit vector

n elements are store

k hash indexes

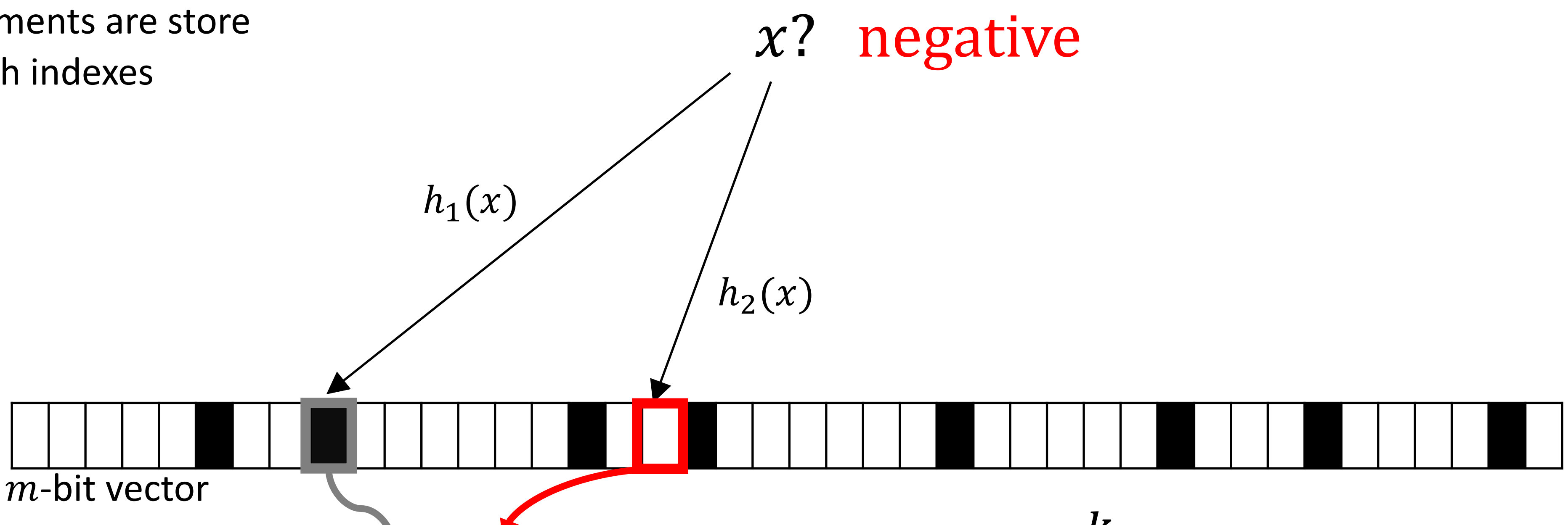
x? negative



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

Bloom Filter

m-bit vector
n elements are store
k hash indexes



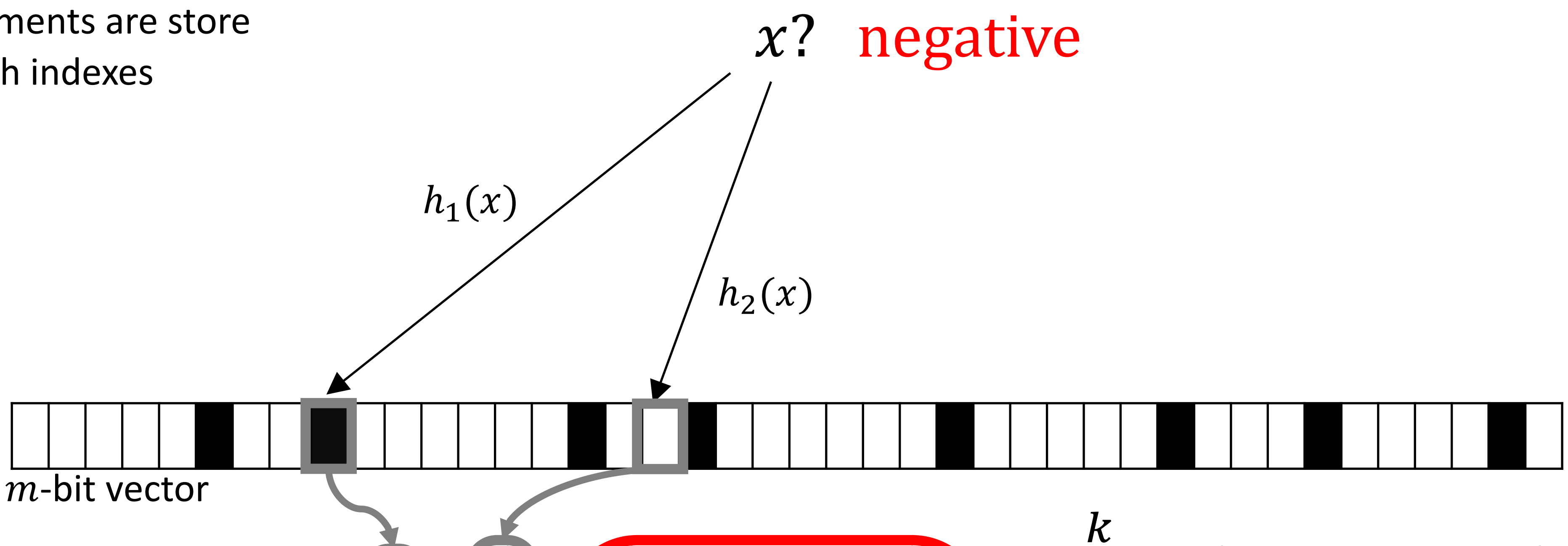
$$probes_{empty} = 1 + \frac{1}{2} + \frac{1}{2^2} + \cdots + \frac{1}{2^{k-1}} = \sum_{d=1}^{k} \frac{1}{2^{k-1}} = 2 - \frac{1}{2^{k-1}}$$

Bloom Filter

m-bit vector

n elements are store

k hash indexes

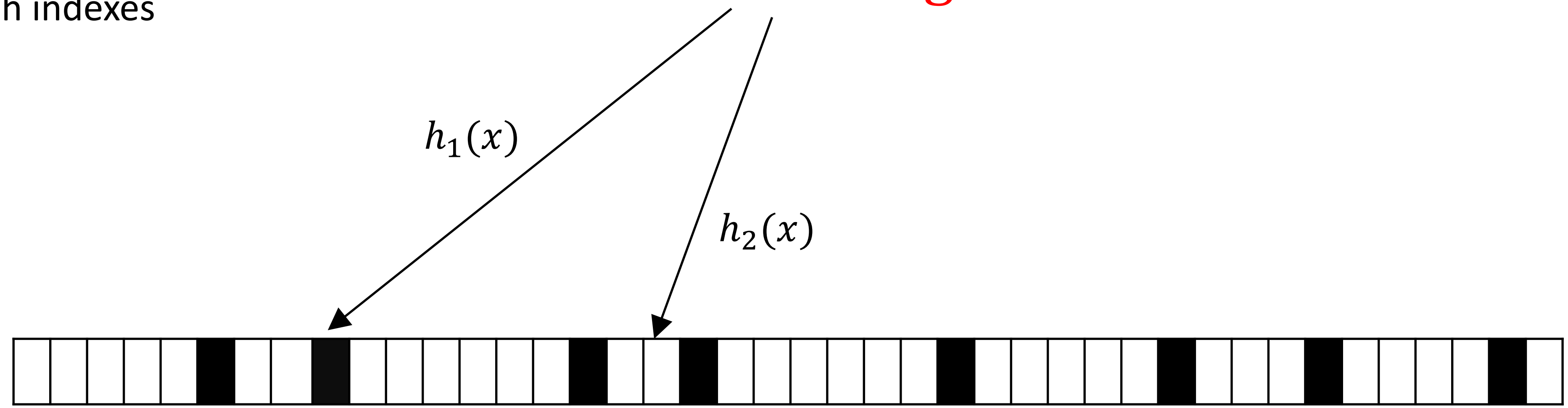


$$probes_{empty} = 1 + \frac{1}{2} + \underbrace{\frac{1}{2^2} + \cdots + \frac{1}{2^{k-1}}}_{\text{for all } k \text{ hash indexes}} = \sum_{d=1}^{k} \frac{1}{2^{k-1}} = 2 - \frac{1}{2^{k-1}}$$

Bloom Filter

m-bit vector
n elements are store
k hash indexes

x? negative

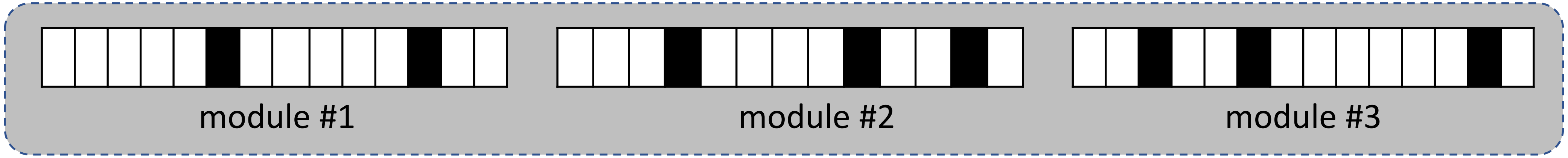


m-bit vector

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

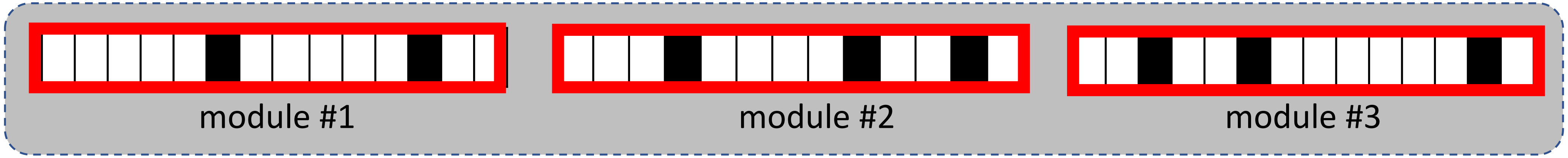
Modular Bloom Filter

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



Modular Bloom Filter

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

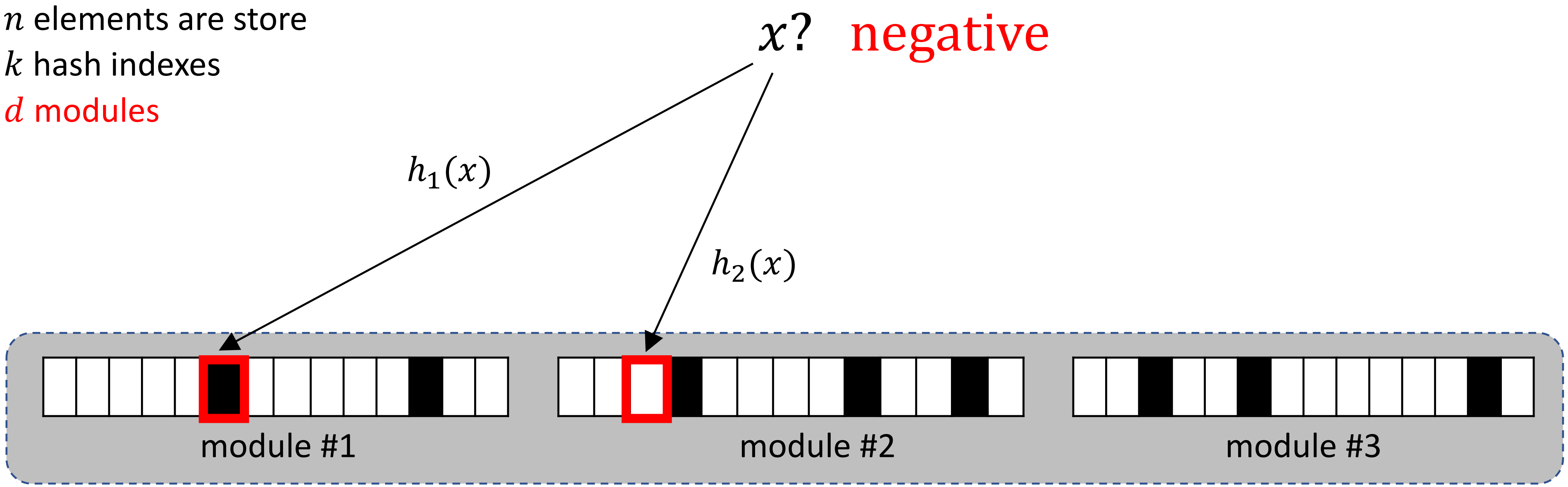


An MBF is a collection of D Bloom filters

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

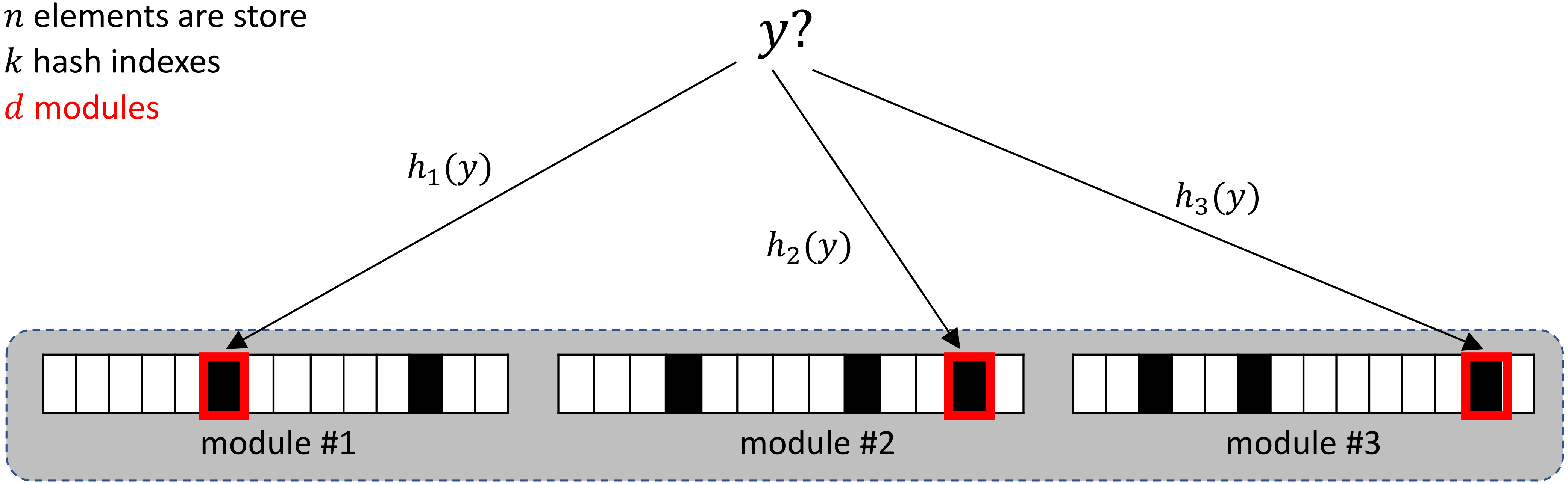
Modular Bloom Filter

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



Modular Bloom Filter

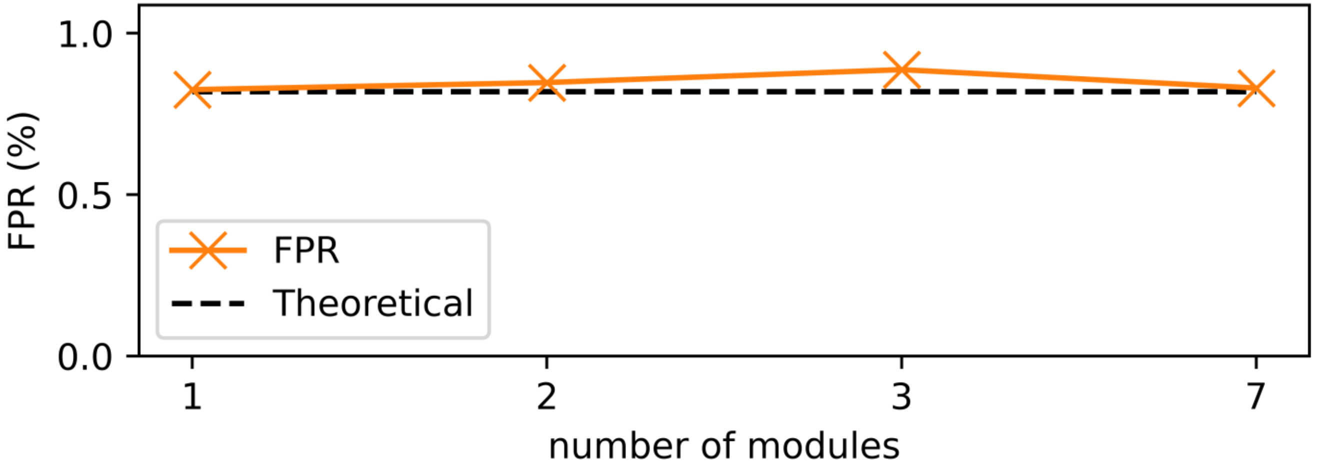
m-bit vector
n elements are store
k hash indexes
d modules



Modular Bloom Filter

False positive rate

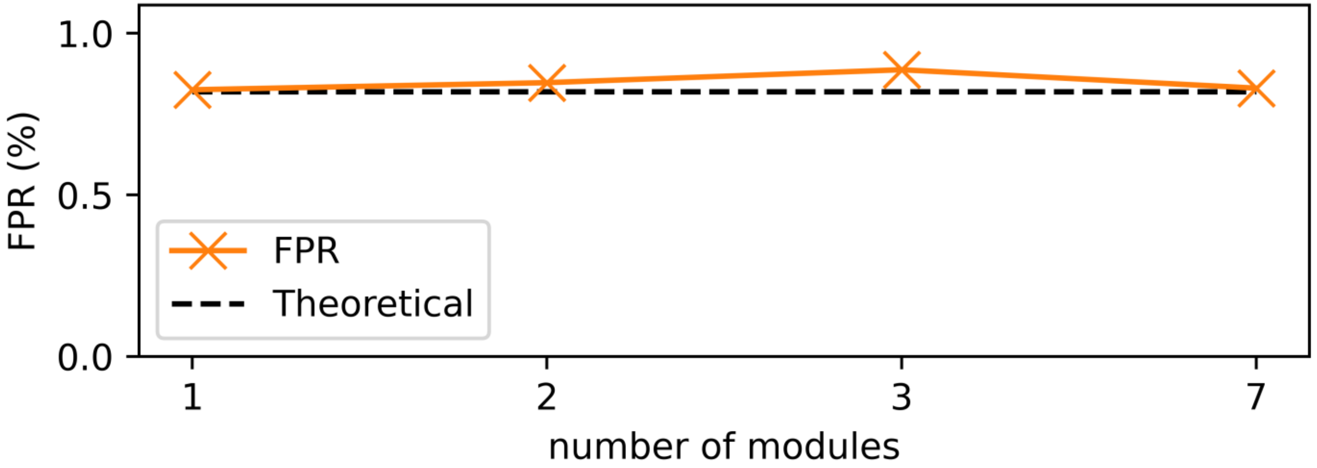
FPR close-to-theoretical



Modular Bloom Filter

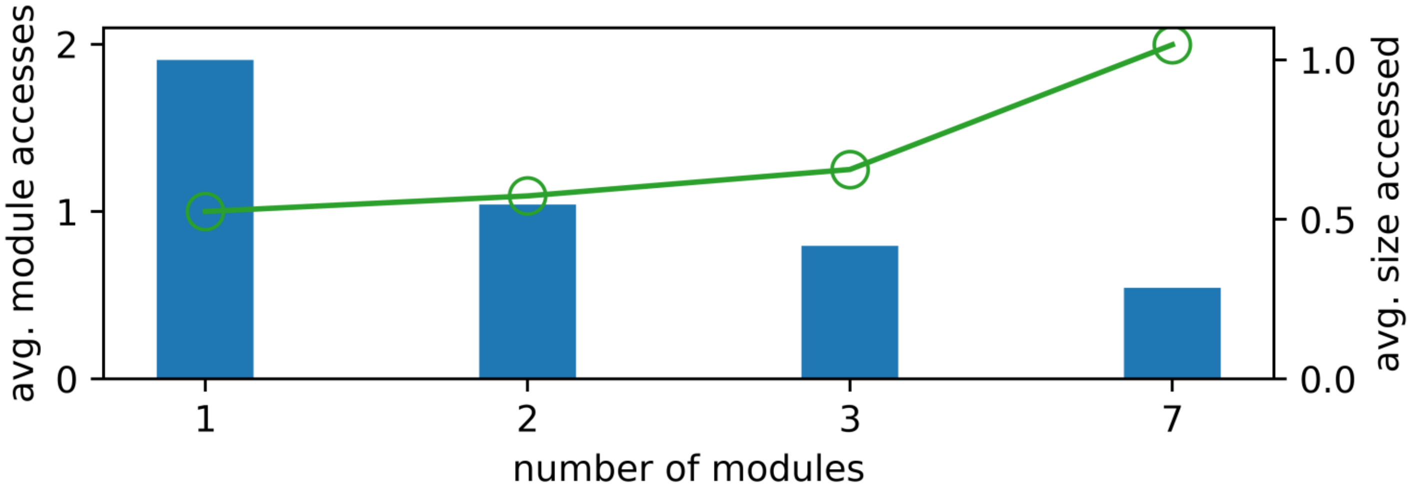
False positive rate

FPR close-to-theoretical



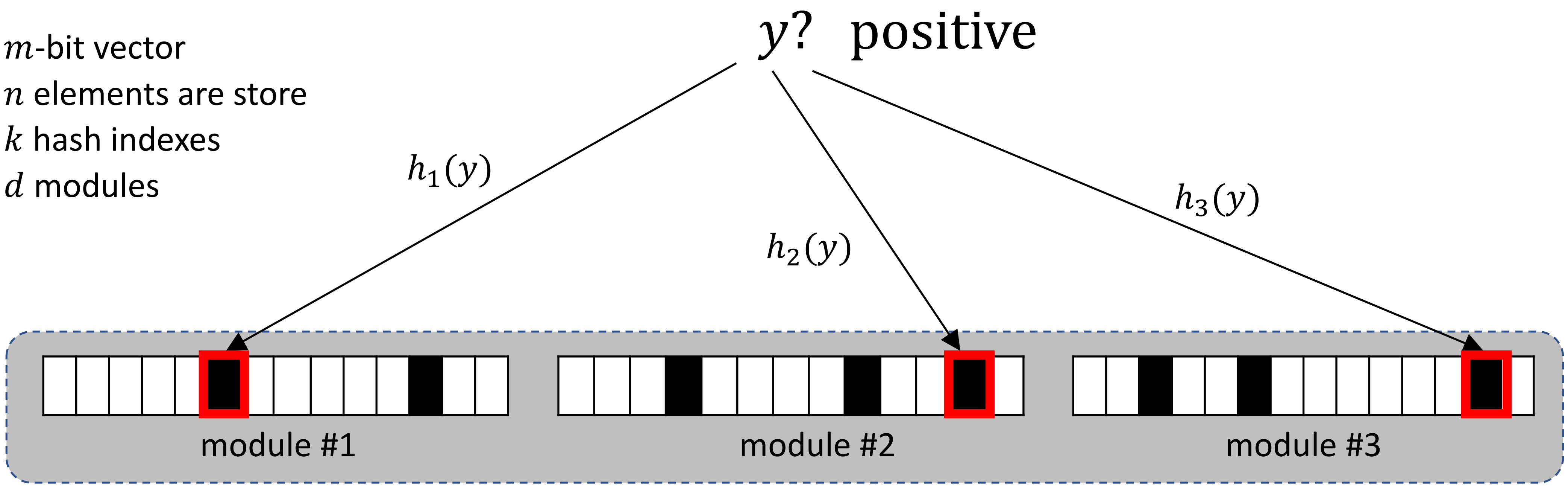
Avg. # of module accesses
vs.
Avg. size accessed

Less memory requirement



Modular Bloom Filter

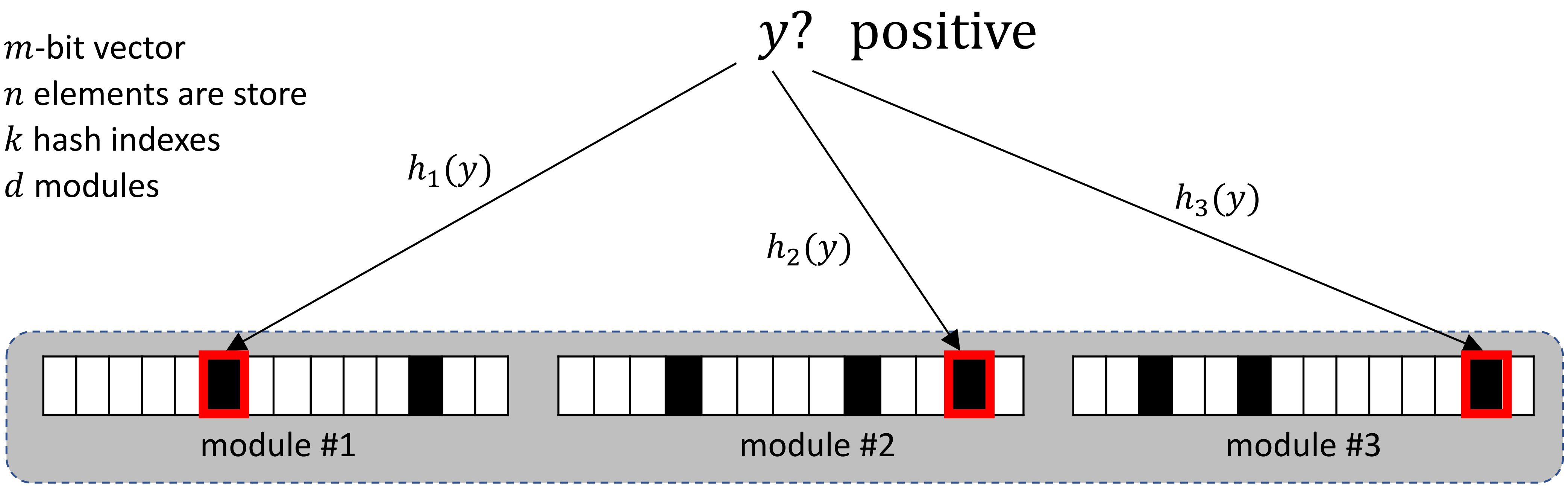
m-bit vector
n elements are store
k hash indexes
d modules



MBFs are not useful for positive queries.

Modular Bloom Filter

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



MBFs are not useful for positive queries.

What if we know something more about the queries?

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

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

access frequency

$$expIO_{l,i,d} = \underbrace{\beta_{l,i}} \cdot \left(\underbrace{\alpha_{l,i}} + \underbrace{\left(1 - \alpha_{l,i}\right) \cdot f_{sm}^{d}}\right)$$

non-empty queries

false positives from empty queries

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

Skipping Modules

Skipping Modules *based on their utilities*

Skipping Modules

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

$u_{l,i,d}$ = expIO(l,i,d) – expIO(l,i,d-1)

if $u_{l,i,d} < threshold_d$ **then**
    return *true*

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

Modular Bloom filter
&
Skipping Algorithm
&
Sharing Hashing

\+

LSM-tree

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

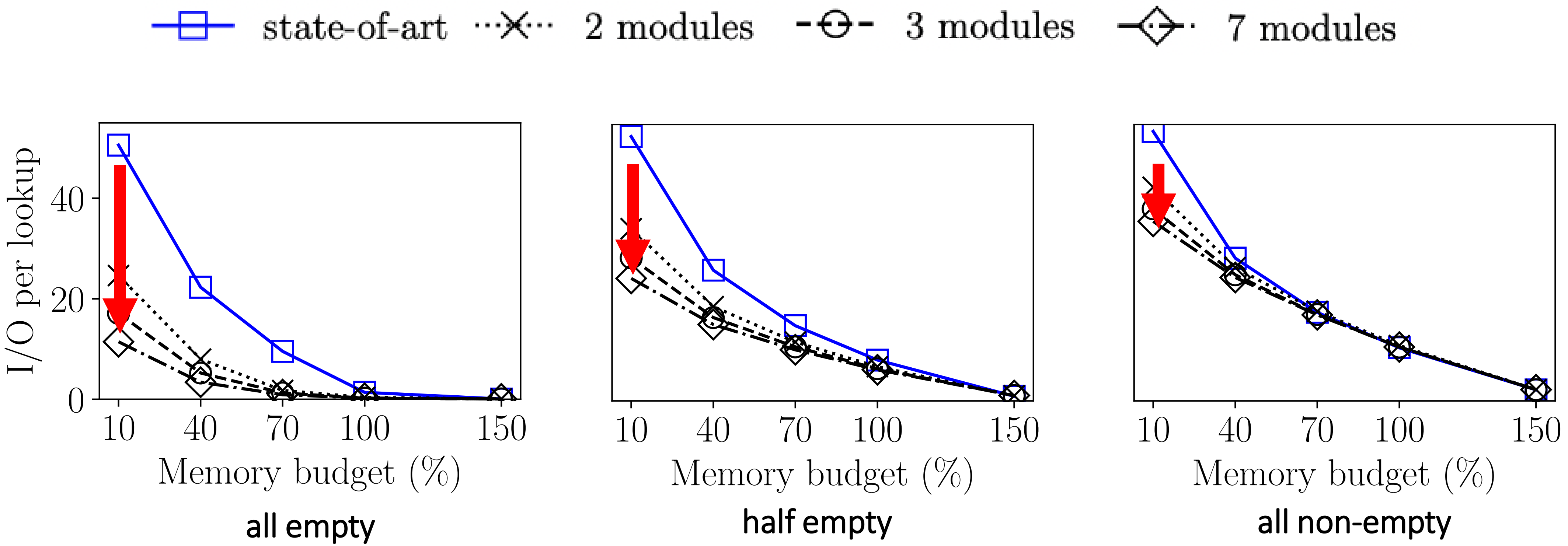
Term	Value
number of modules	1, ***2***, 3, or 7
Size of each module	***equal***
skipping algorithm	none, partial ($\mathcal{P}$), or ***full ($\mathcal{F}$)***

Approaches Tested

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

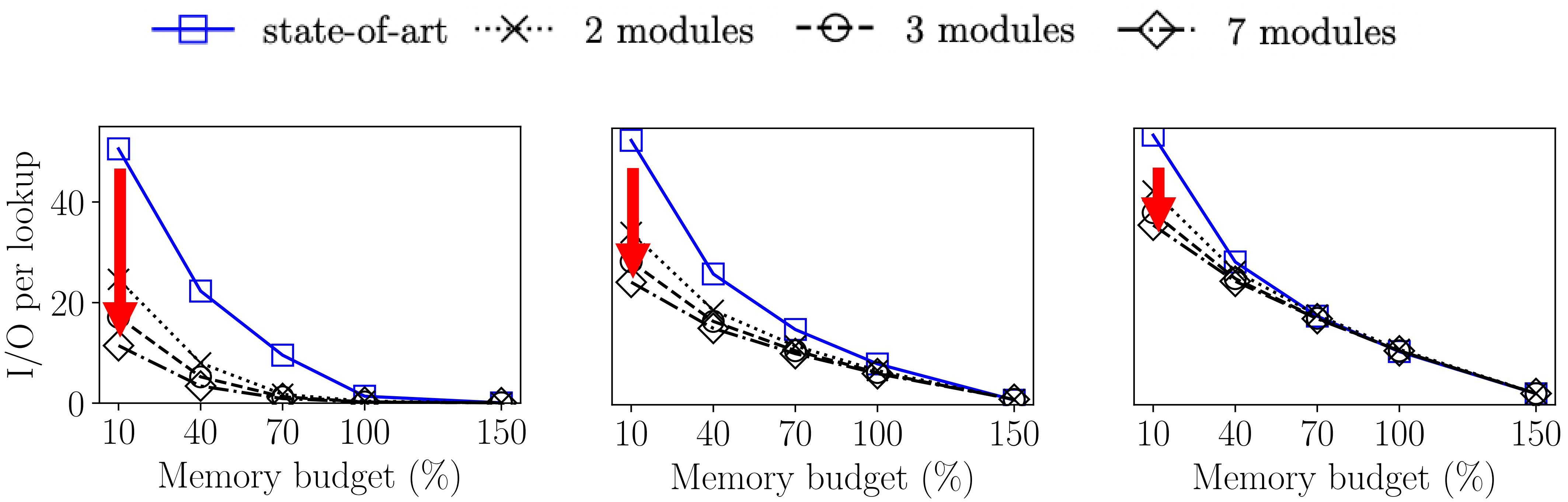
Impact of number of modules

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



Impact of number of modules

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

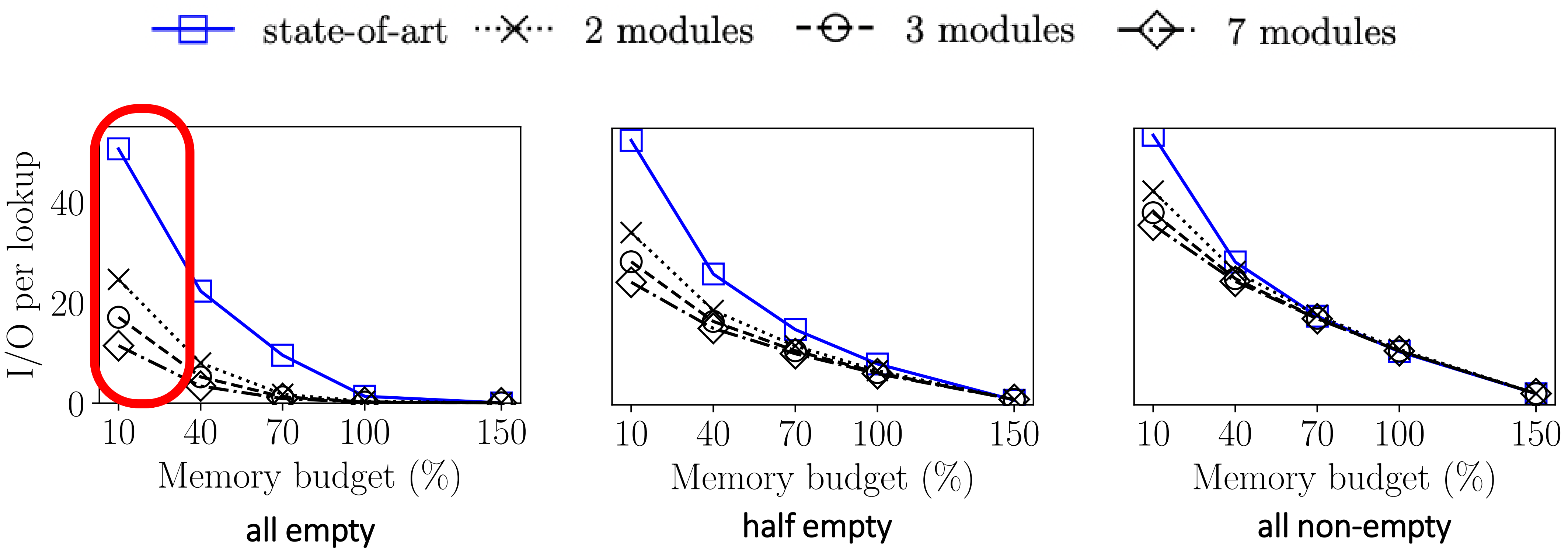


all empty — half empty — all non-empty

SHaMBa enhances the lookup performance for empty queries

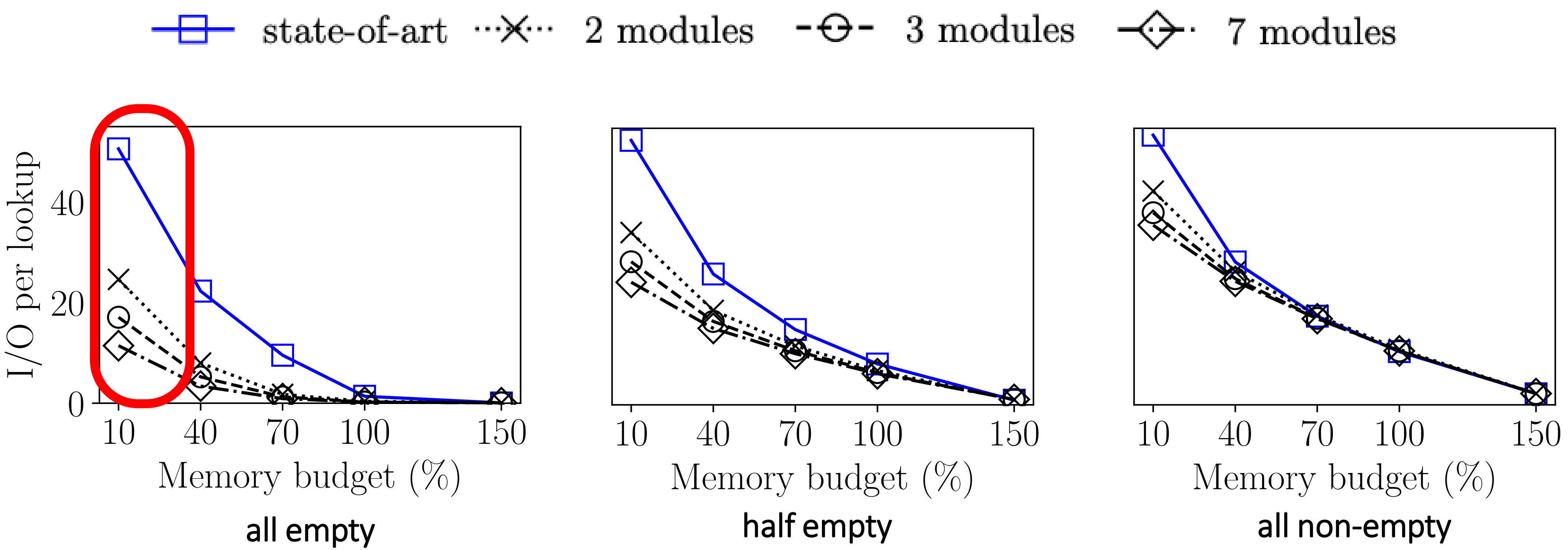
Impact of number of modules

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



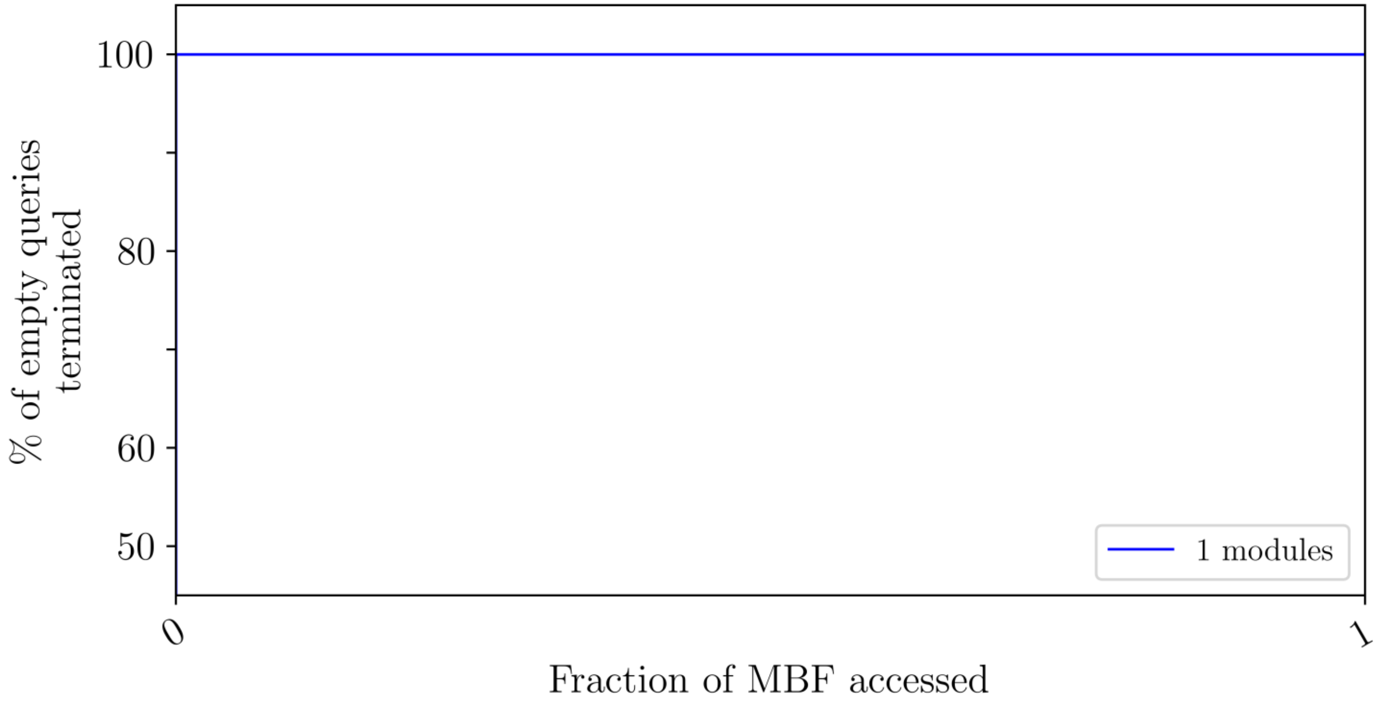
Impact of number of modules

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

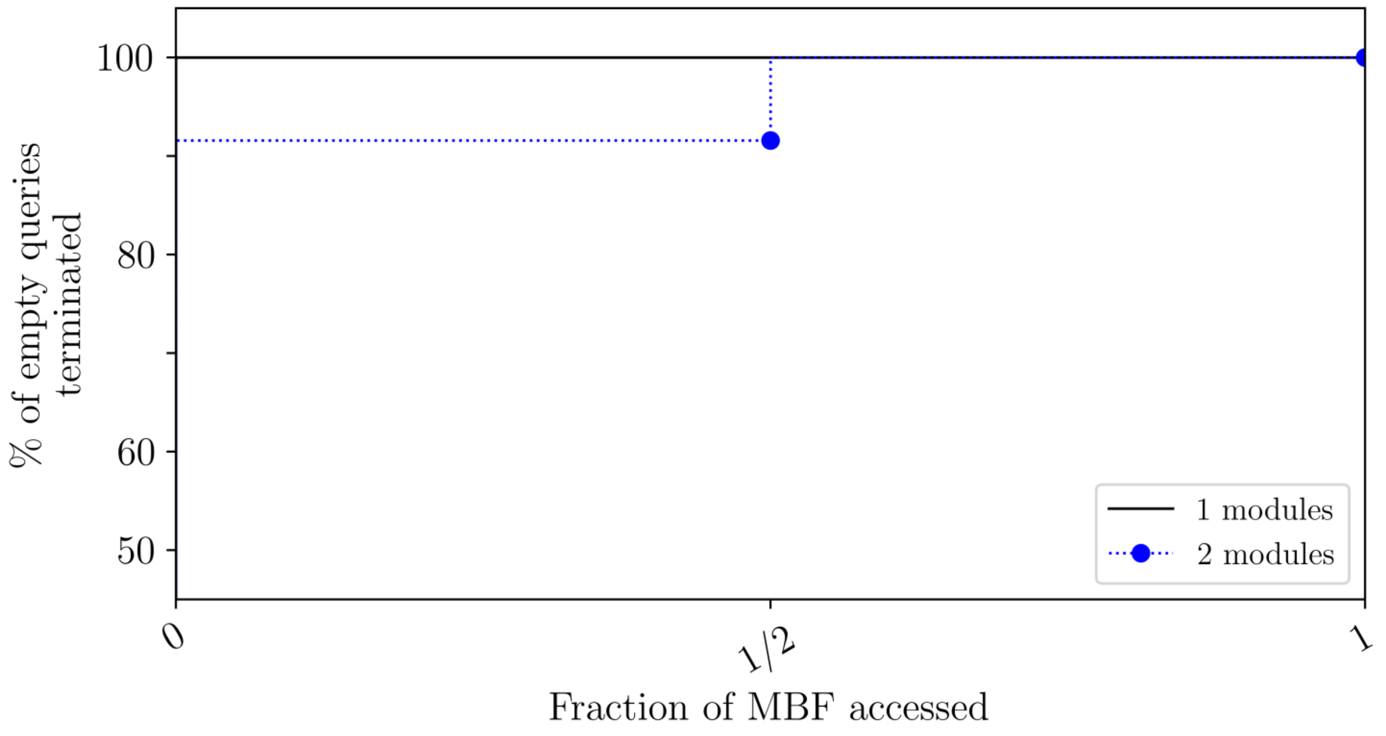


SHaMBa performs best with smaller modules

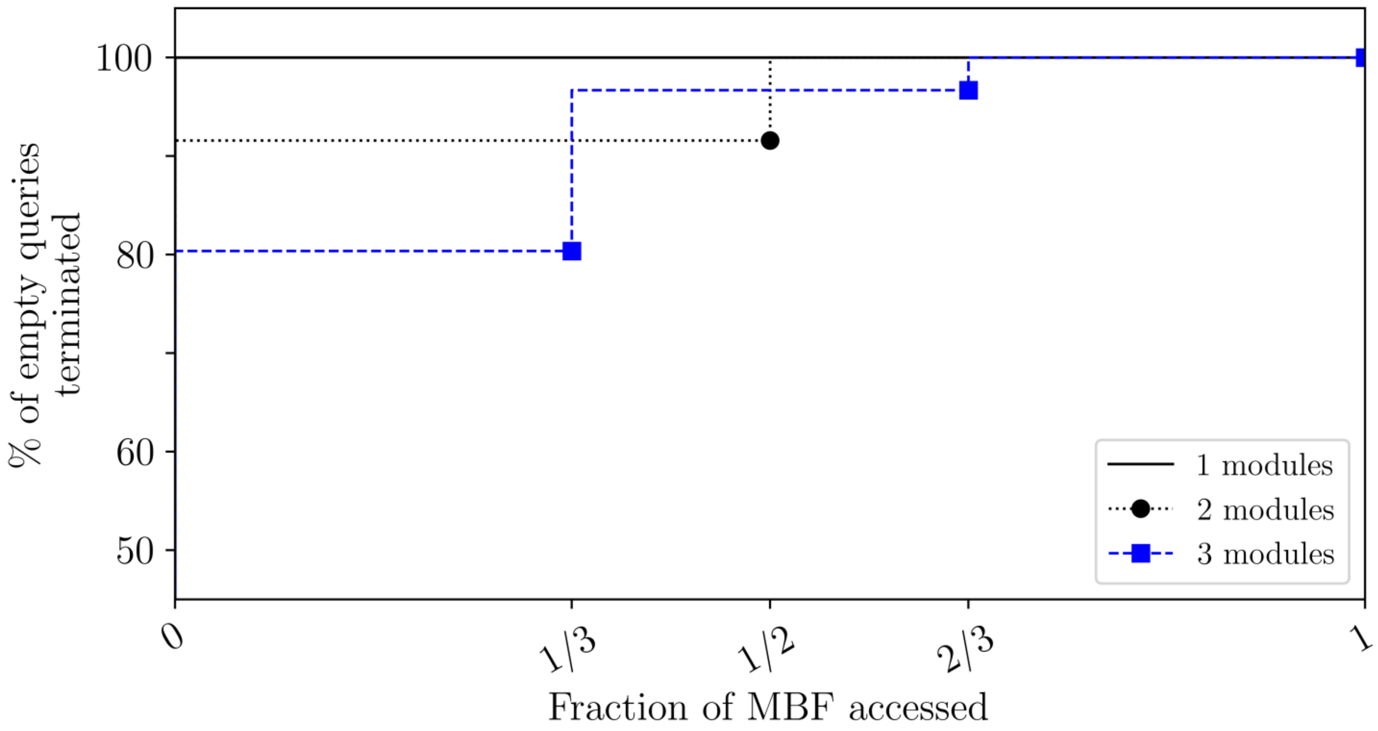
Impact of number of modules



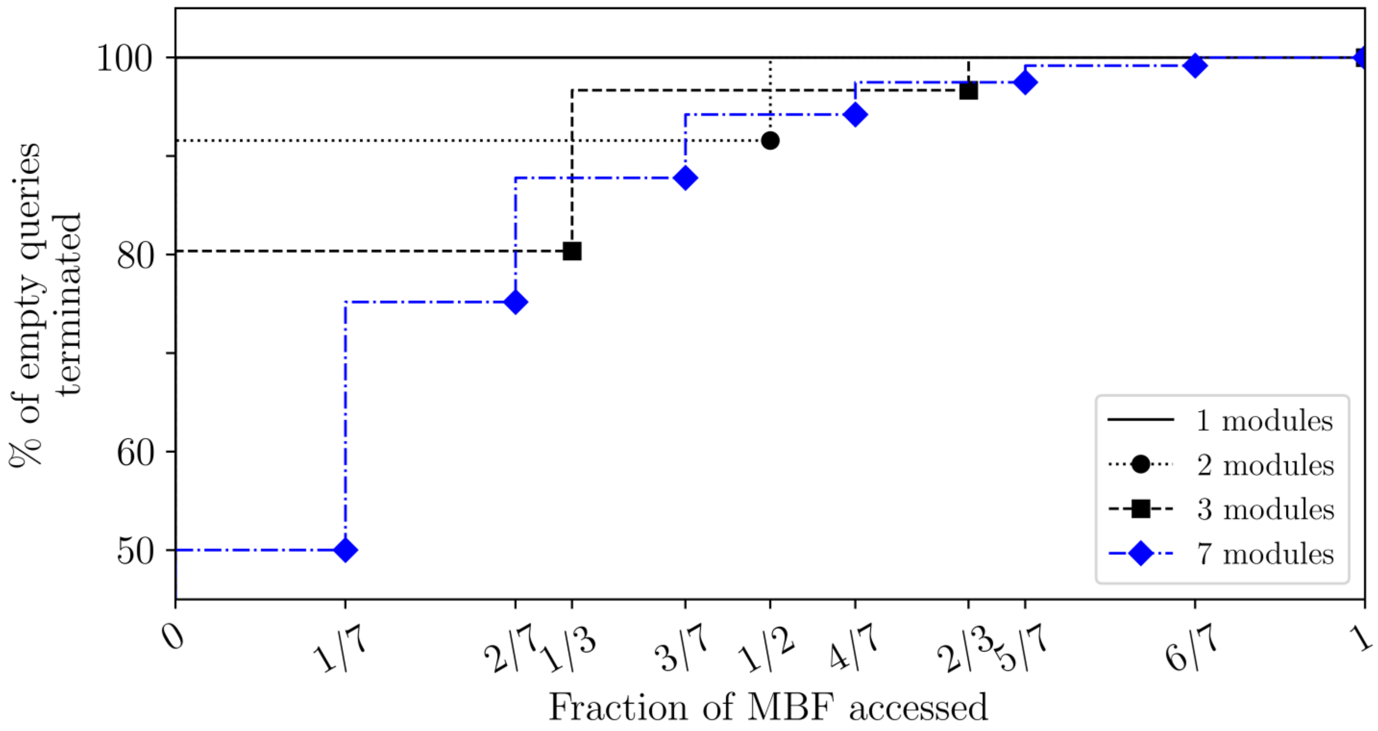
Impact of number of modules



Impact of number of modules



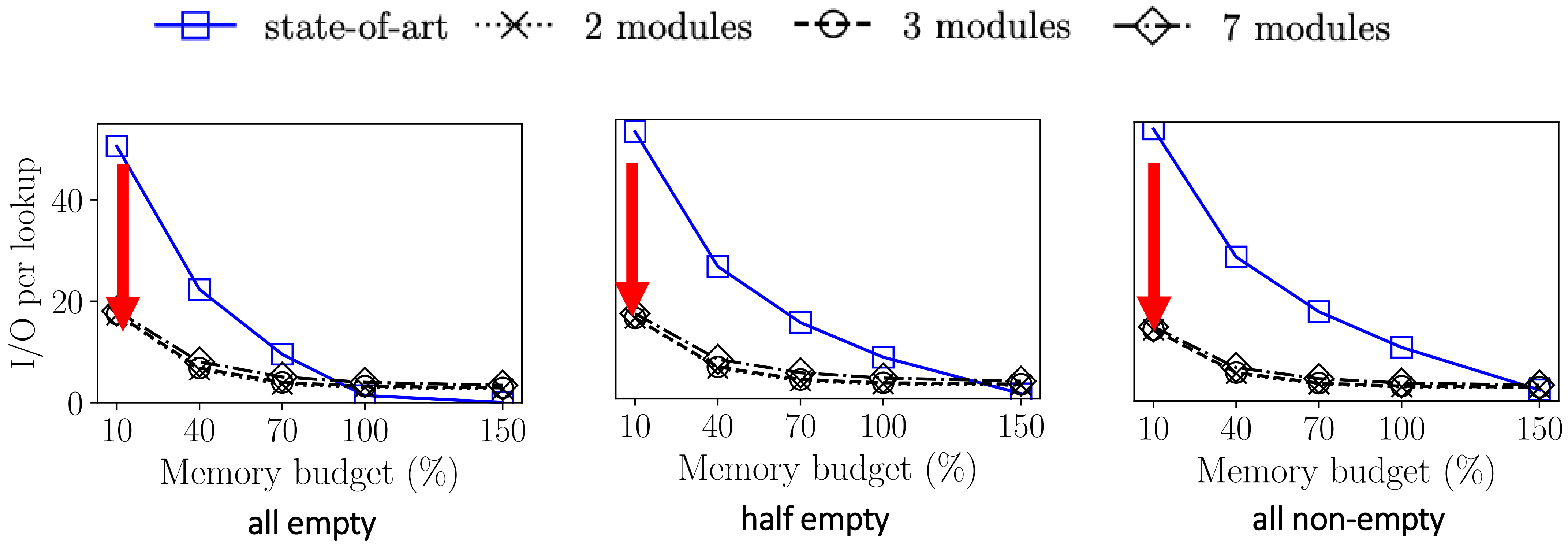
Impact of number of modules



More queries can be terminated earlier with less memory.

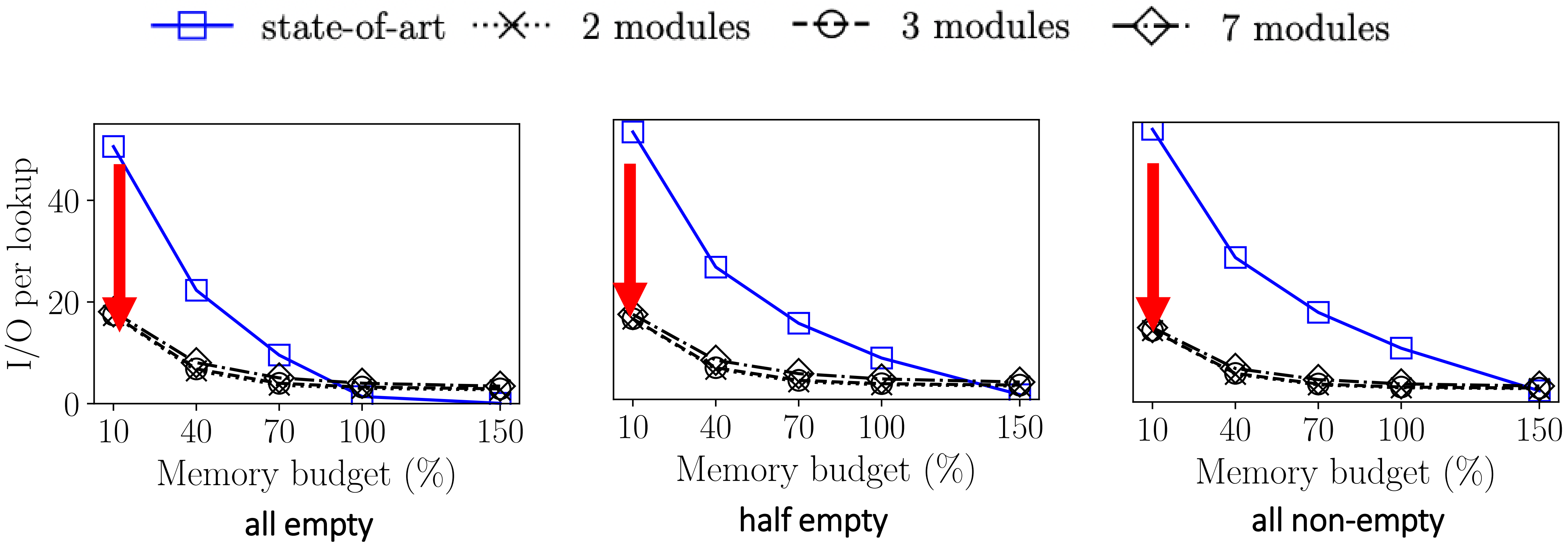
Impact of number of modules

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



Impact of number of modules

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



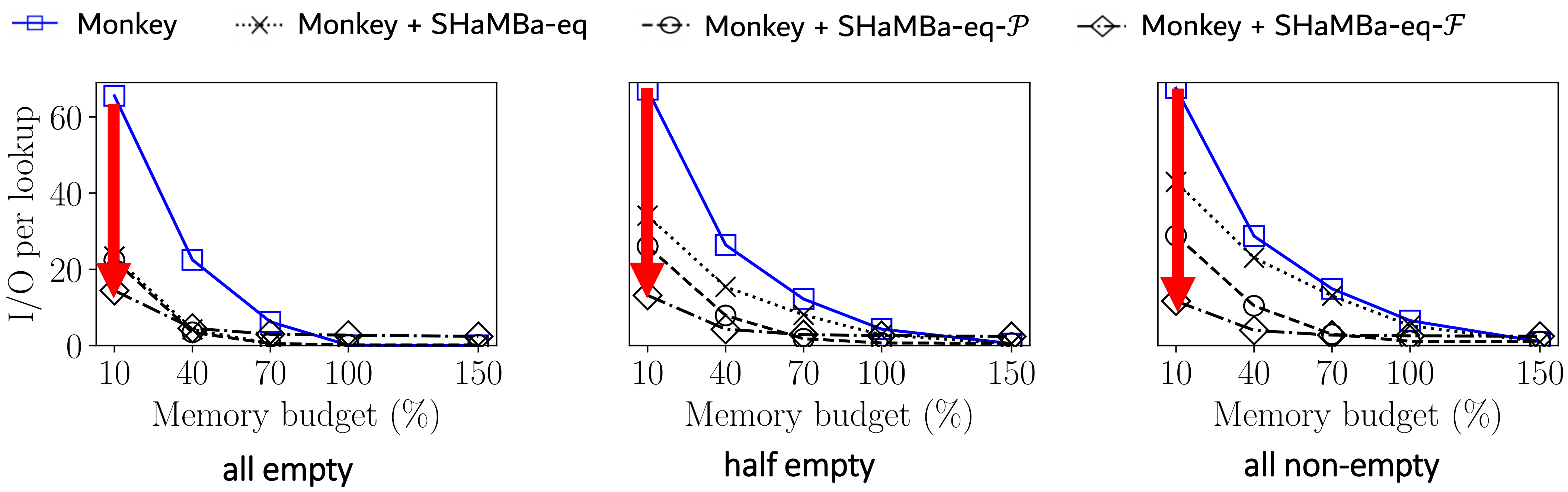
Skipping modules effectively skips unnecessary filters/modules.

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

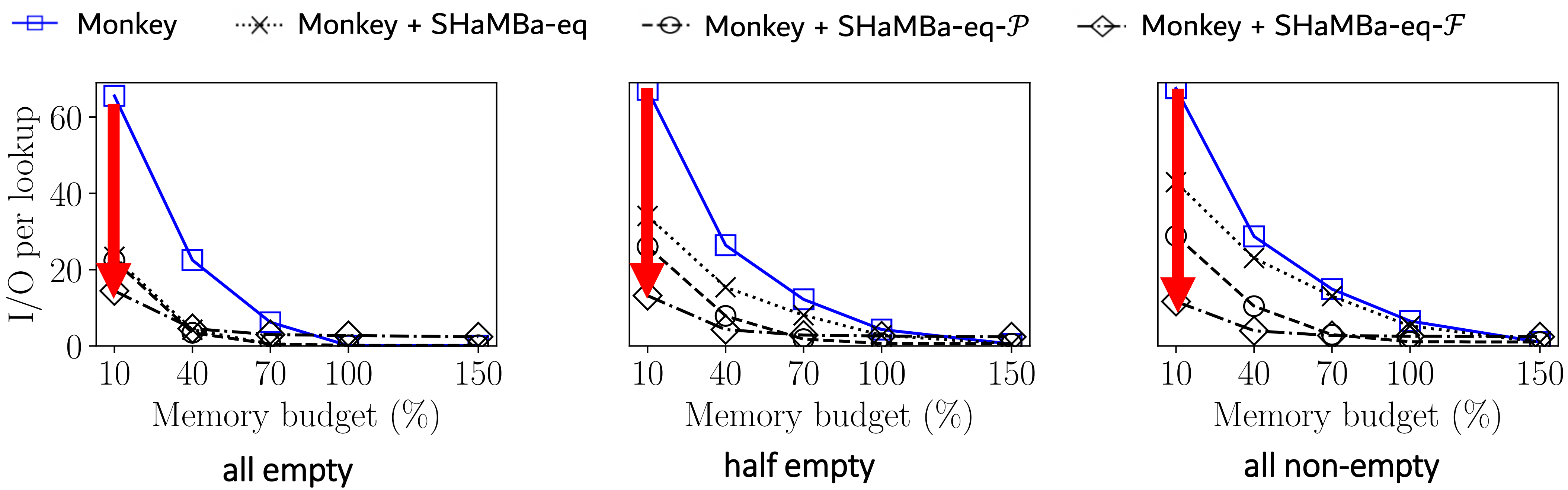


SHaMBa with Monkey

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

Conclusion

- Modular Bloom filters (MBFs)
  - a BF variant that consists of multiple modules
  - 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

Conclusion

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

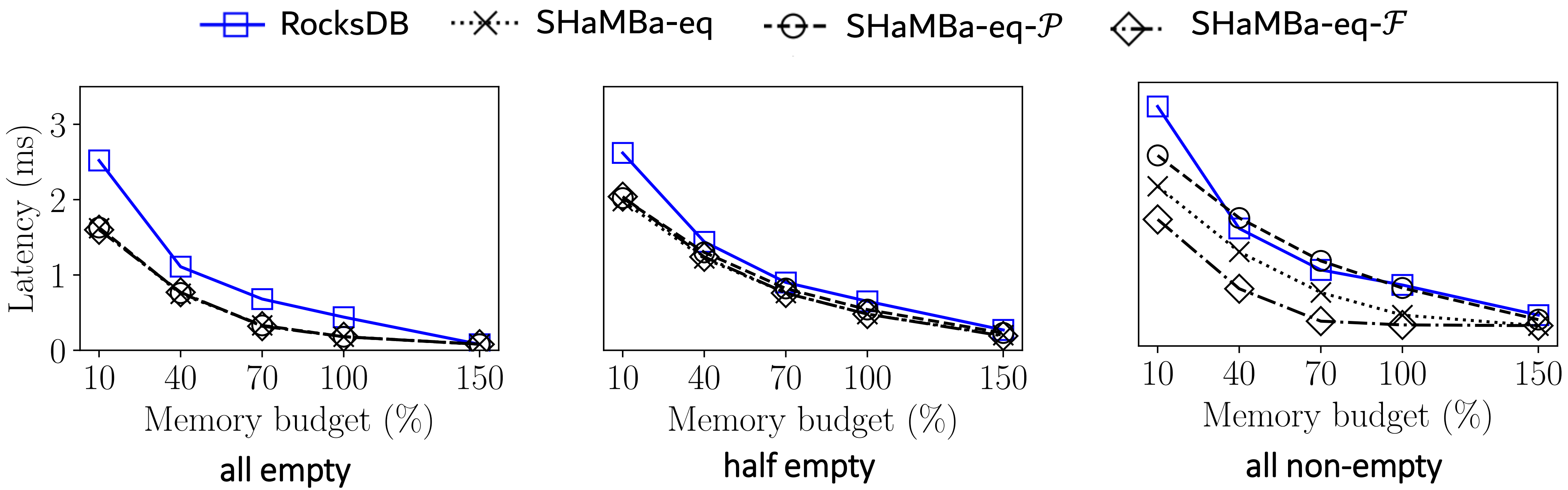
- SHaMBa
  - a novel LSM-based key-value engine
  - specifically addresses performance loss due to memory pressure
  - the same average number of I/Os, with 1/3 of the memory by the state of the art

Thank you!

Q&A

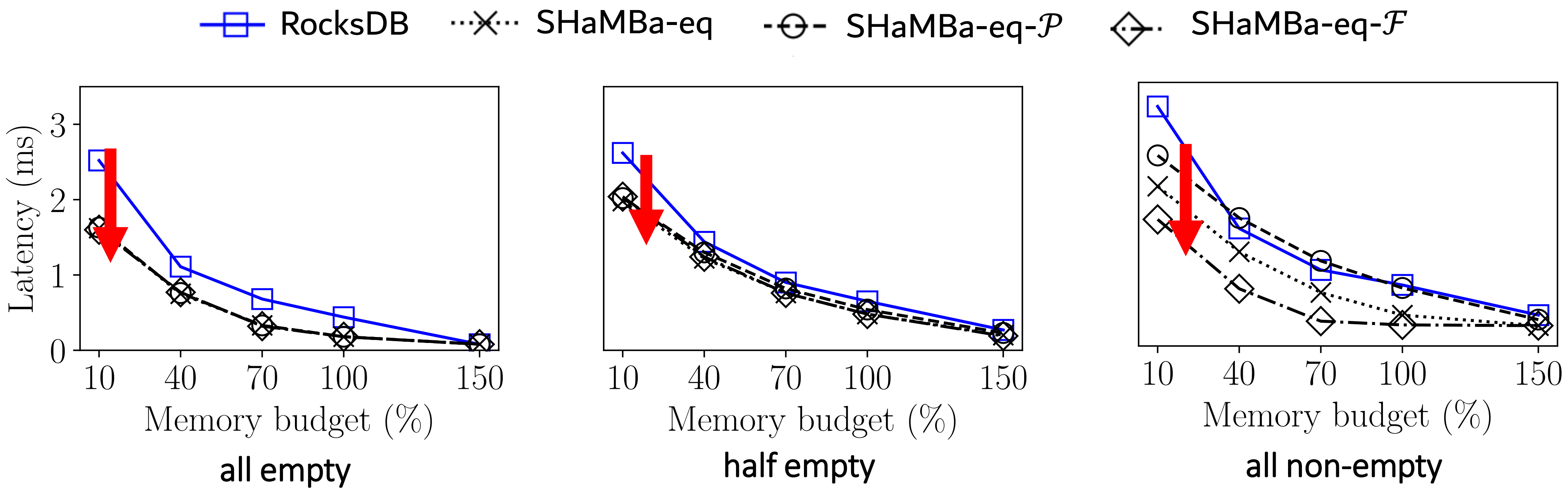
SHaMBa with RocksDB

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



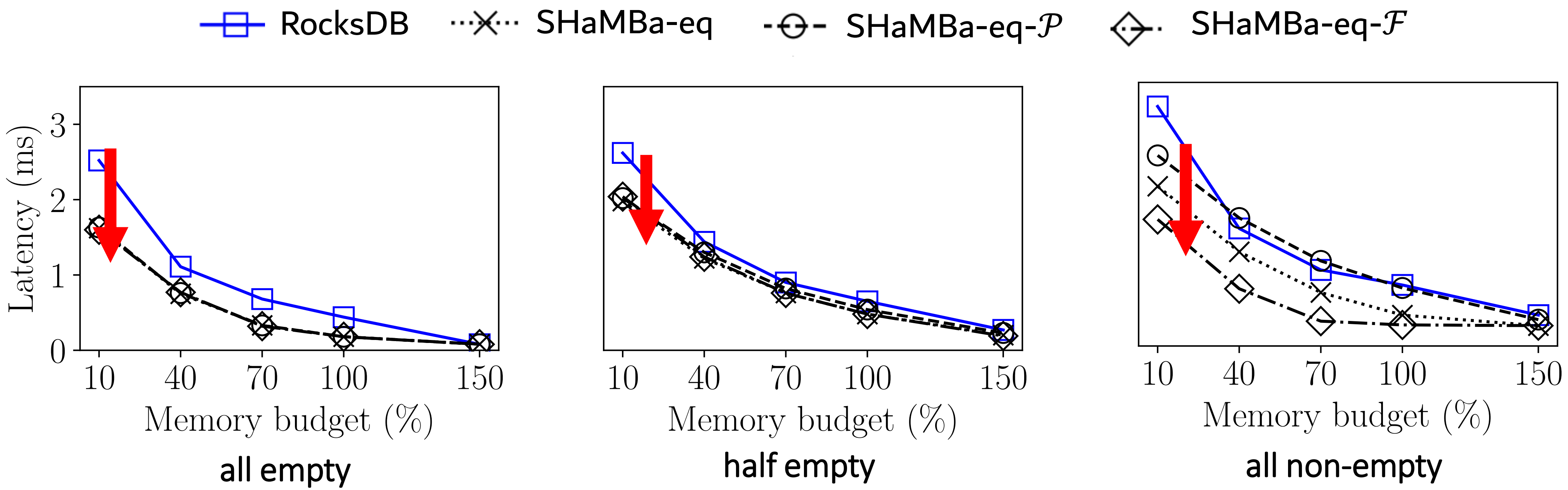
SHaMBa with RocksDB

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



SHaMBa with RocksDB

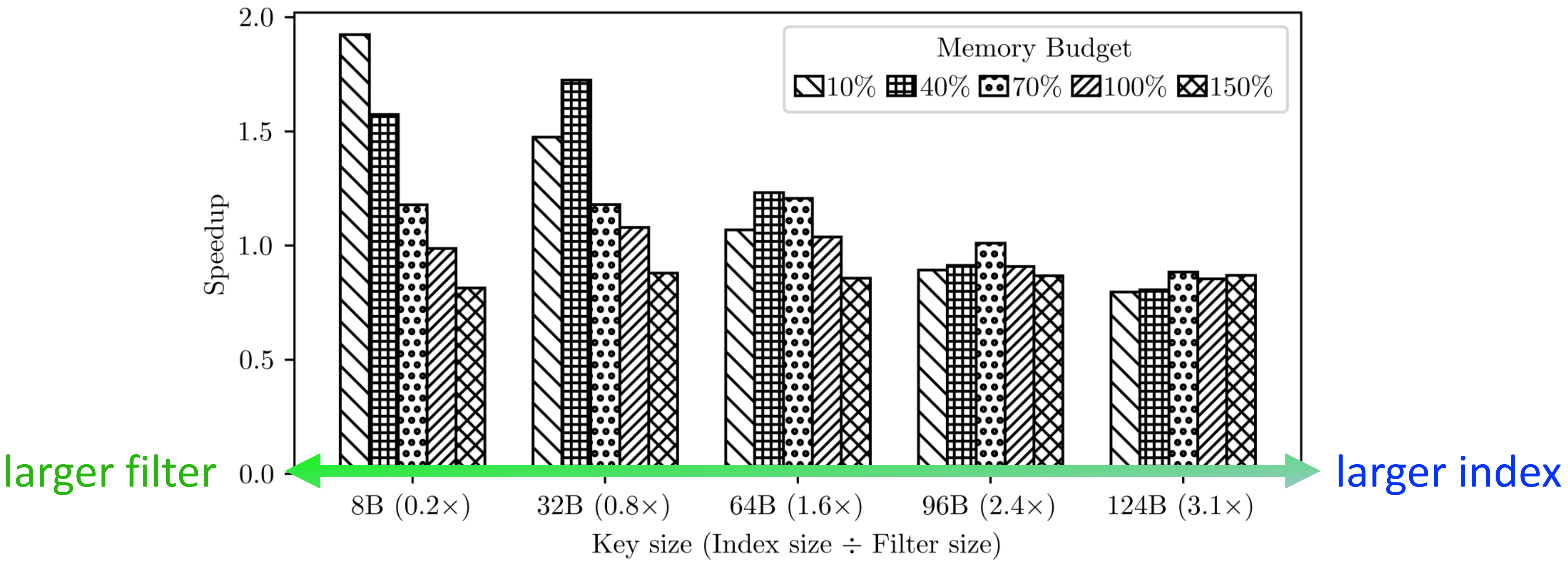
Workload: Uniform, Entry size: 64B, #Entries: 30K
Tuning: 2 equal sized modules, RocksDB version 6.19.3



SHaMBa-eq accelerates point lookups

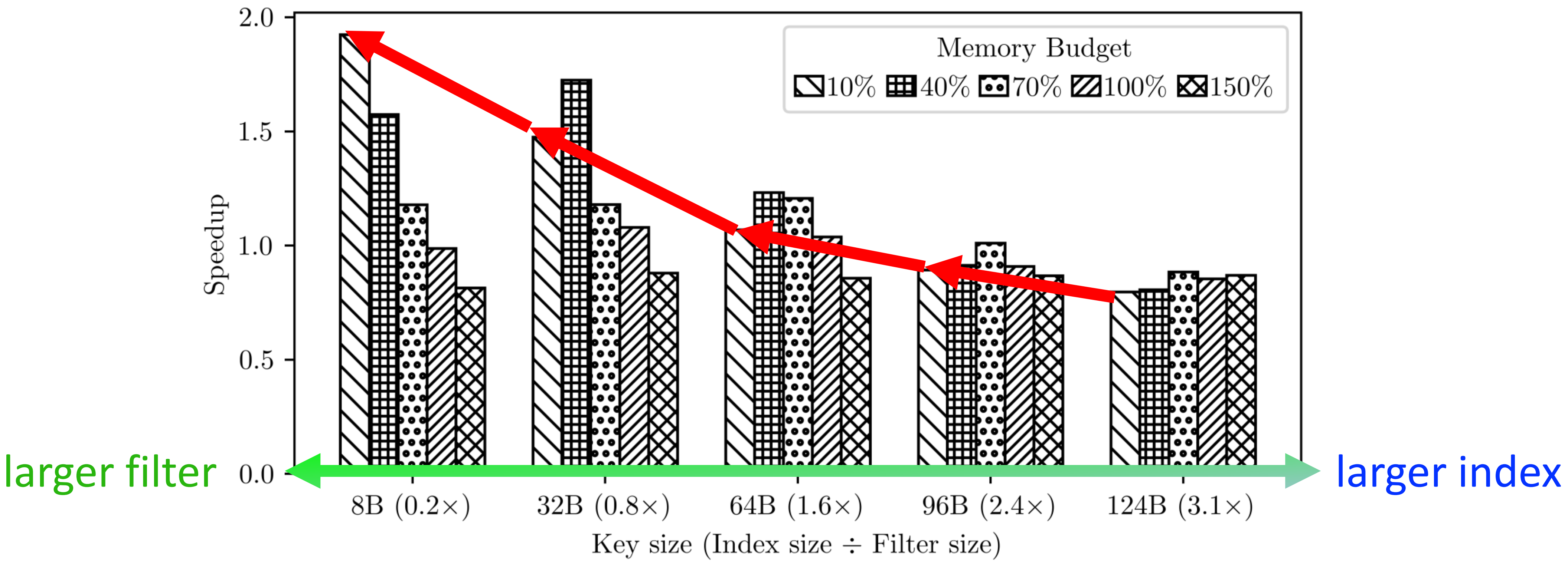
SHaMBa with larger index

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



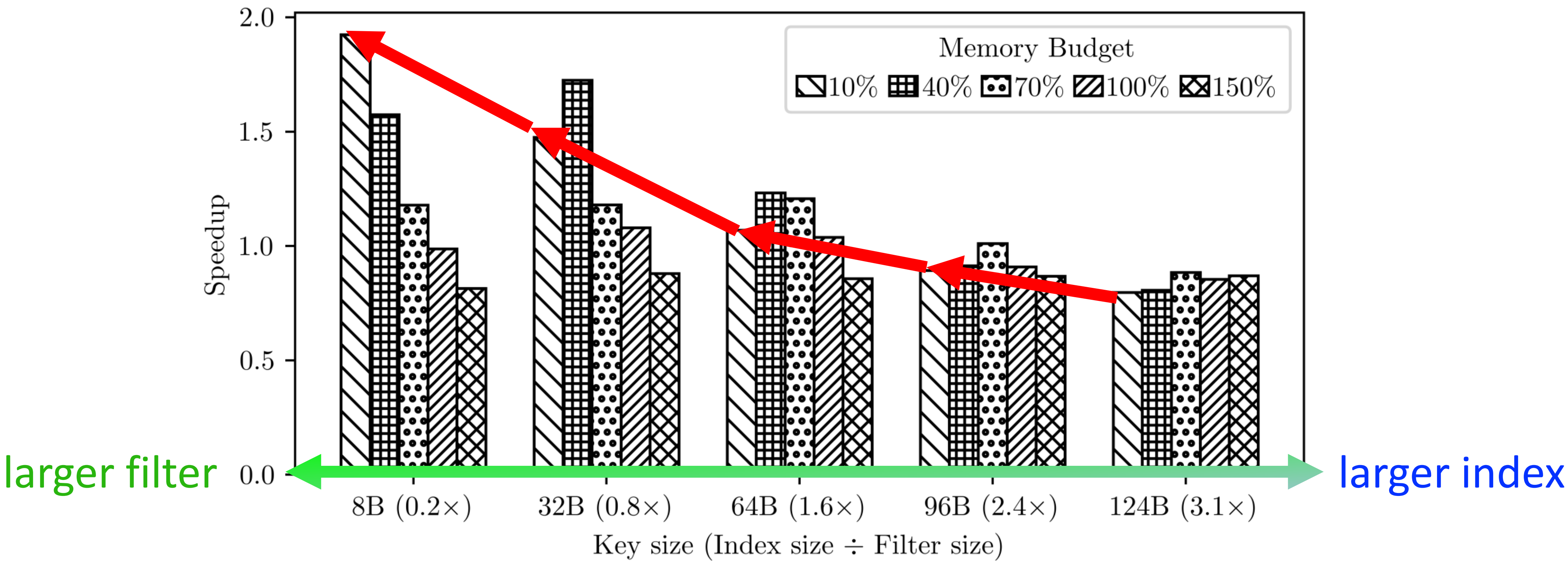
SHaMBa with larger index

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



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