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

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

zczhu@bu.edu, jmun@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

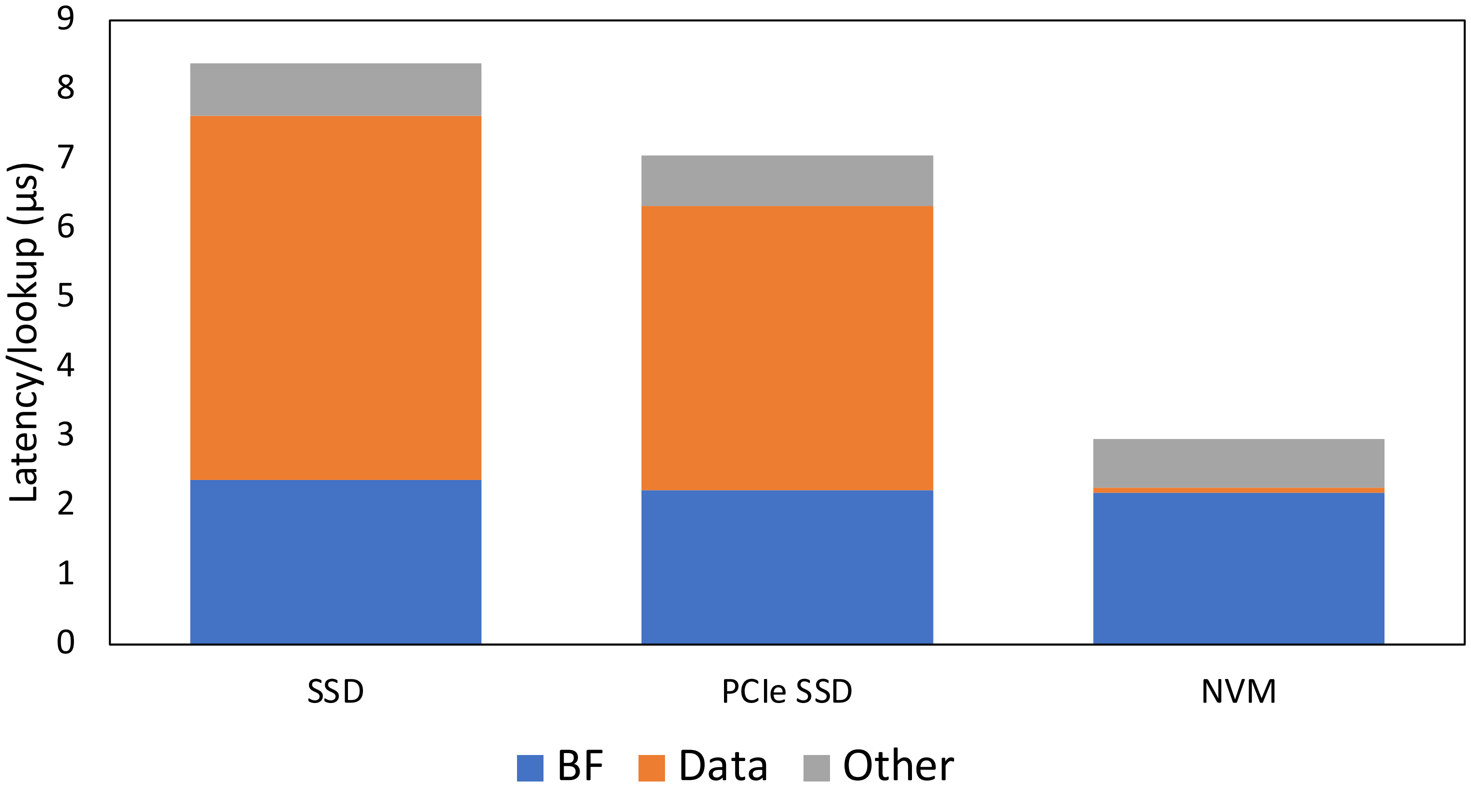
amazon DynamoDB

APACHE HBASE

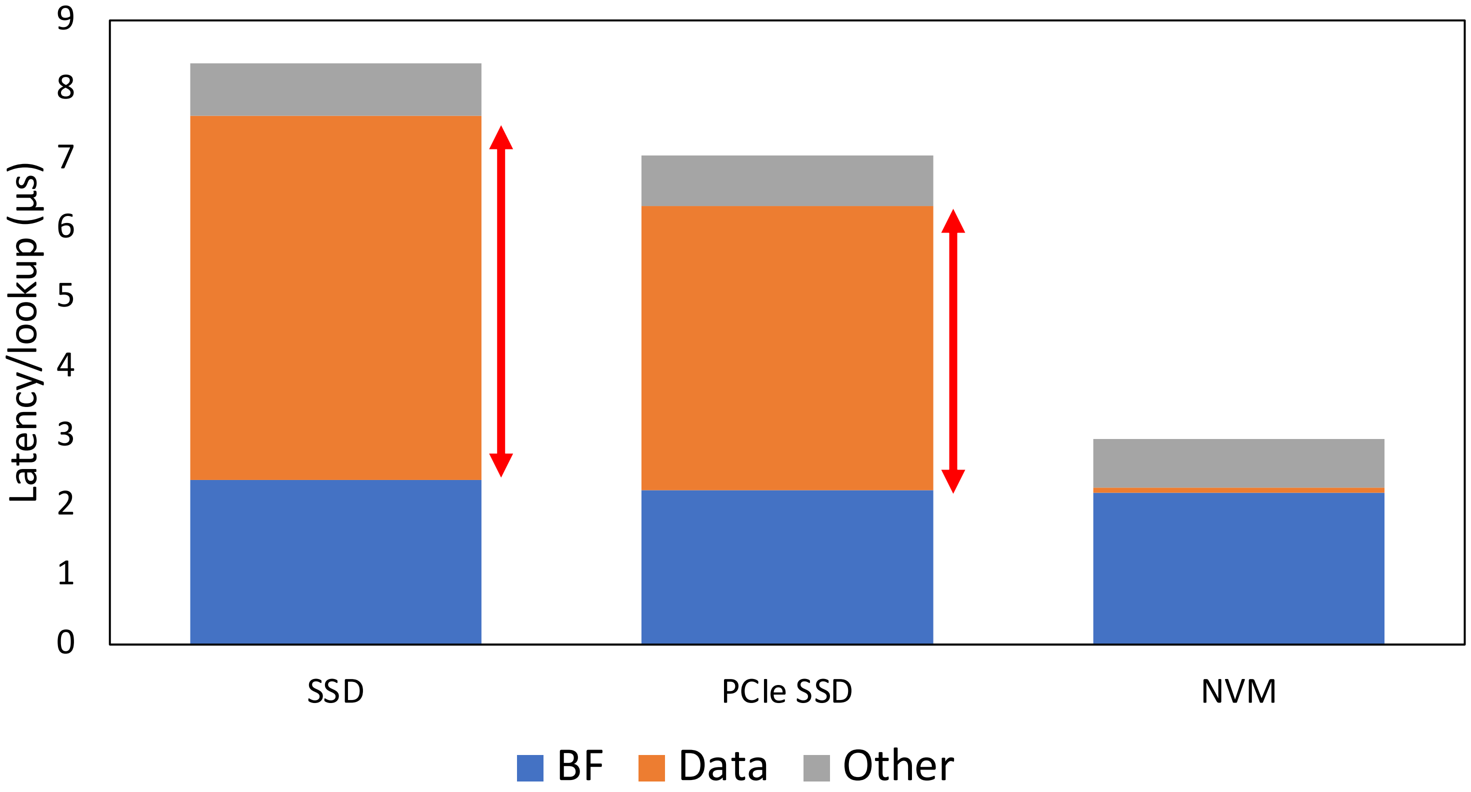
levelDB

cassandra

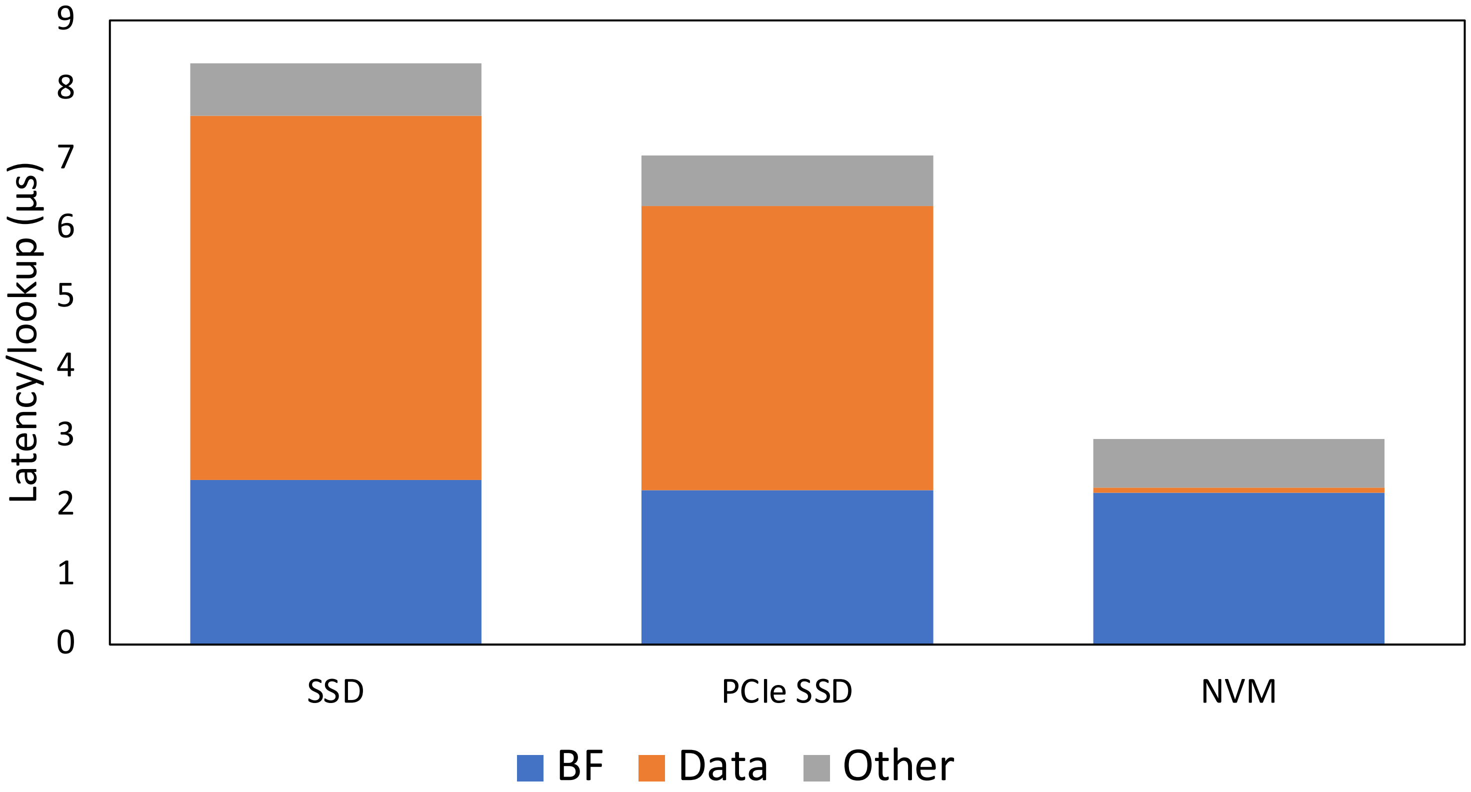
Where does the time go?



Where does the time go?

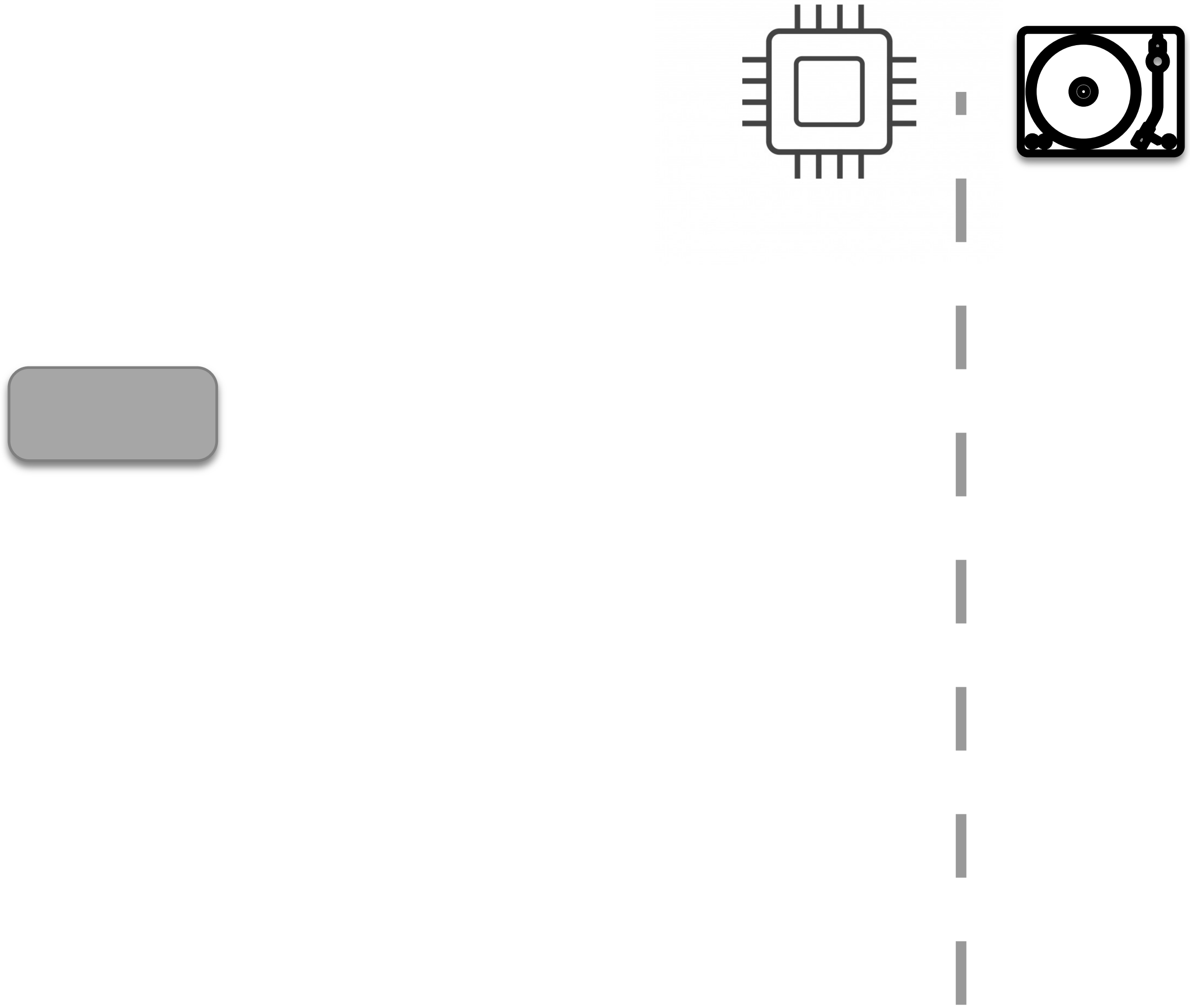


Where does the time go?



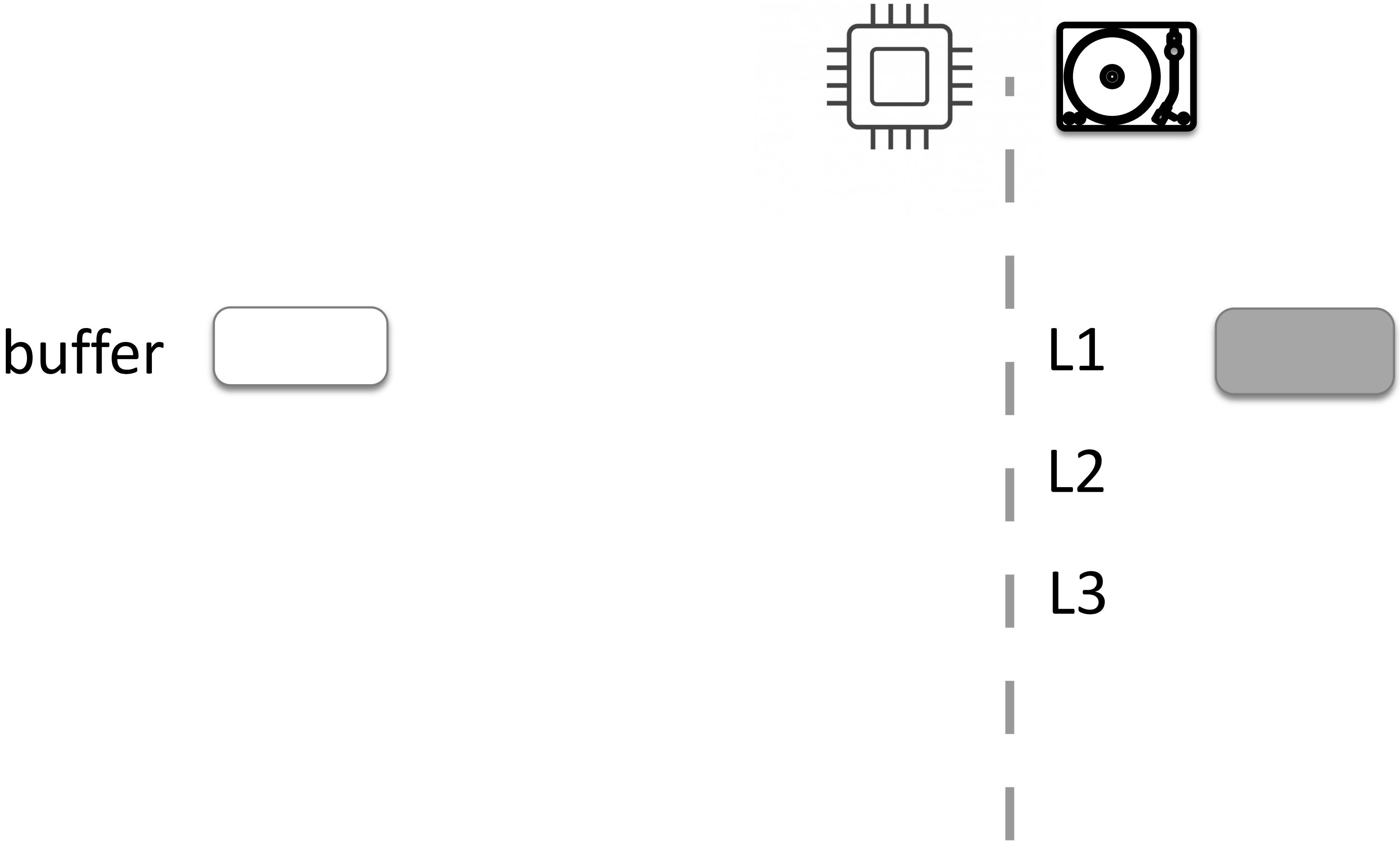
The time spent on Bloom filters dominates for faster storage.

Log-Structured Merge Trees

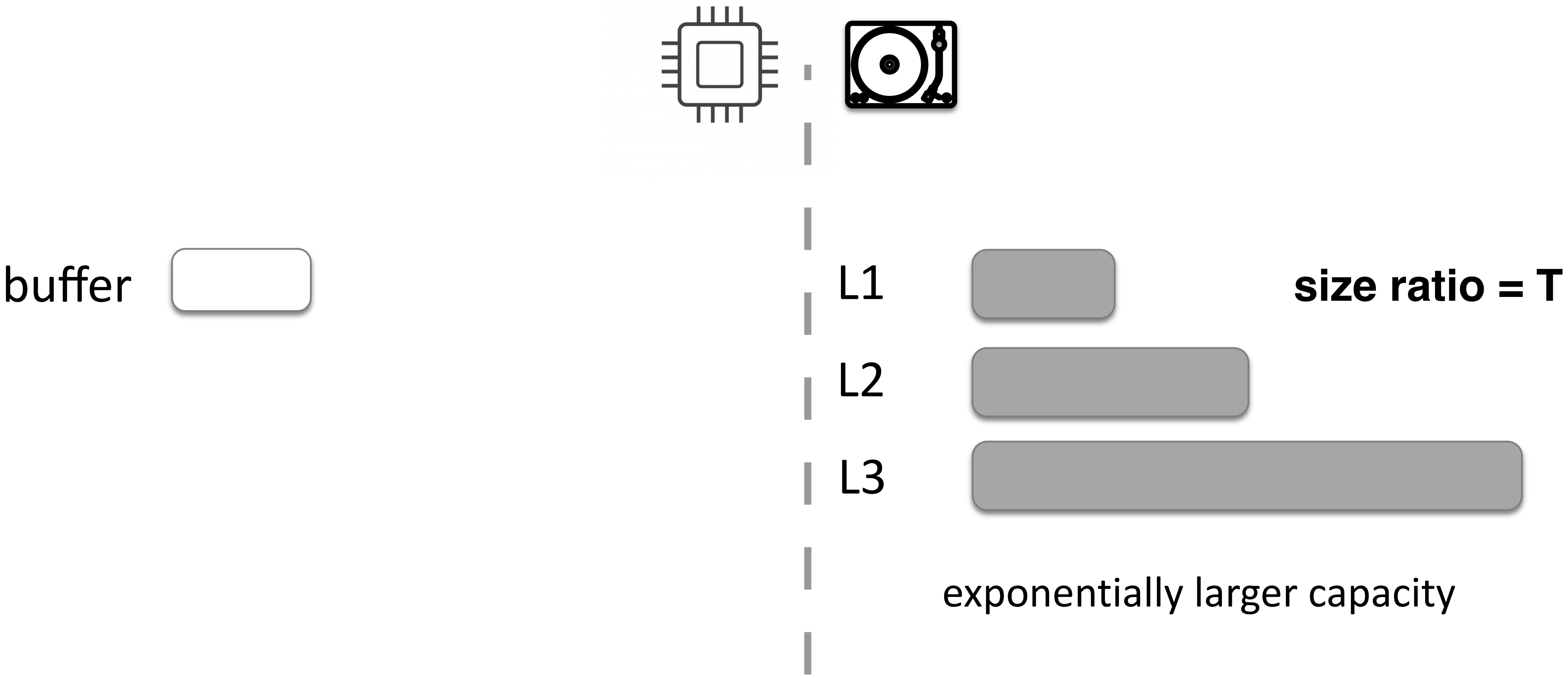


buffer

Log-Structured Merge Trees



Log-Structured Merge Trees



buffer

L1

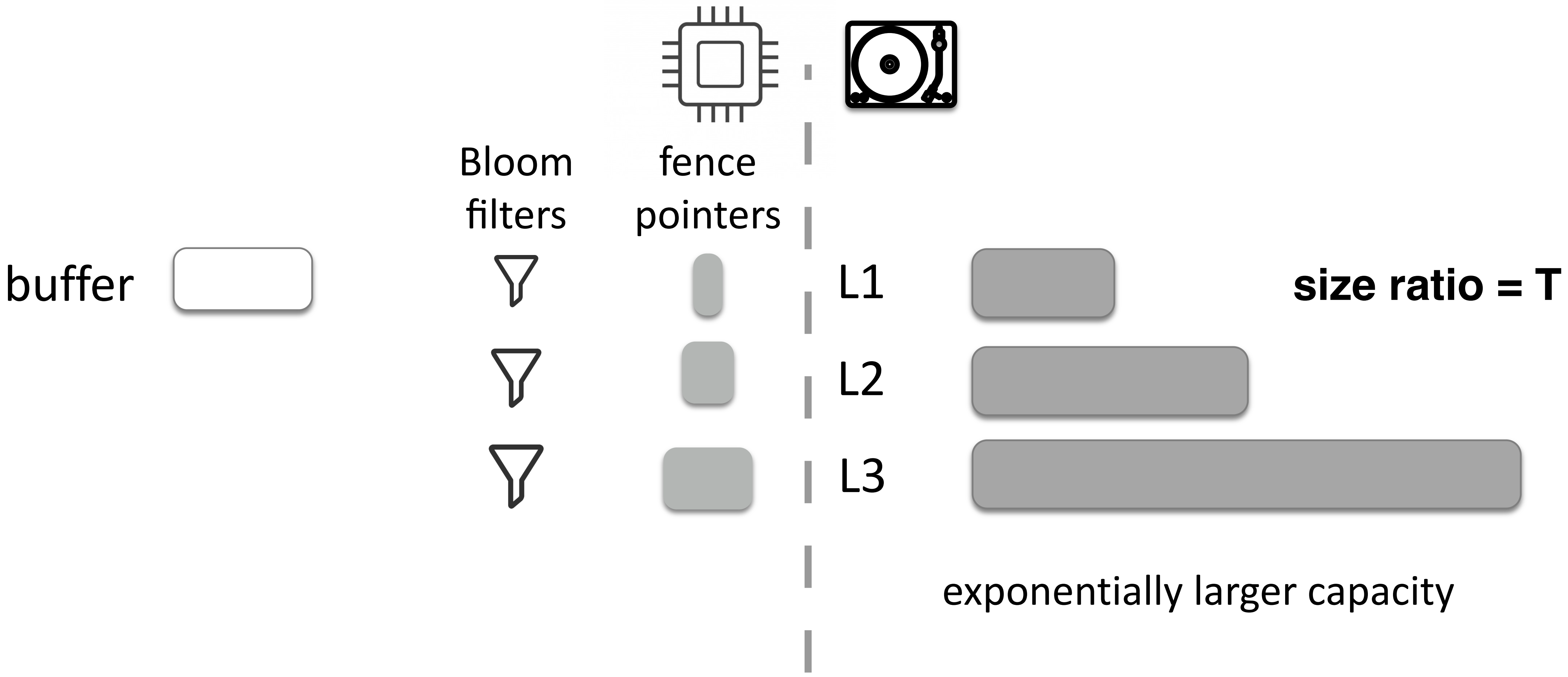
L2

L3

size ratio = T

exponentially larger capacity

Log-Structured Merge Trees



Bloom filters

fence pointers

buffer

L1

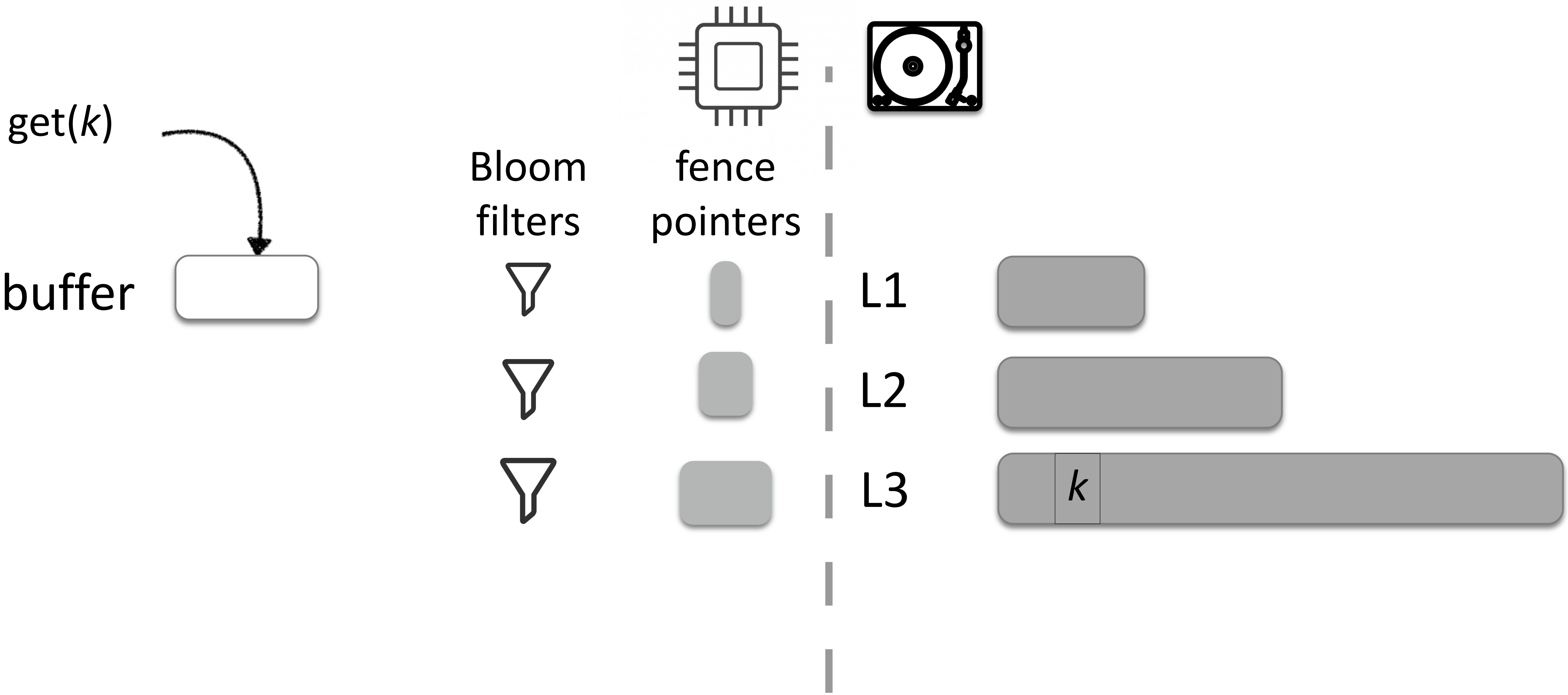
L2

L3

size ratio = T

exponentially larger capacity

Log-Structured Merge Trees



get(*k*)

Bloom filters

fence pointers

buffer

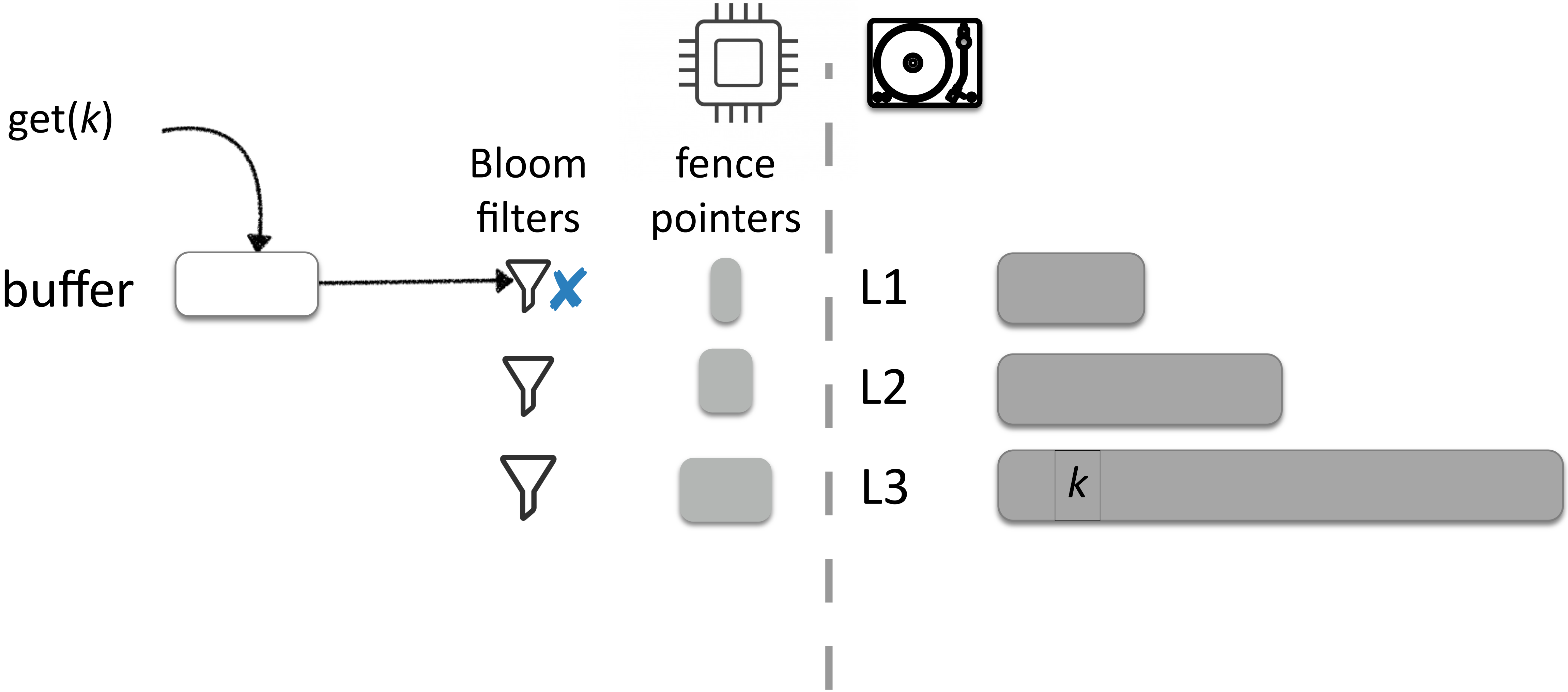
L1

L2

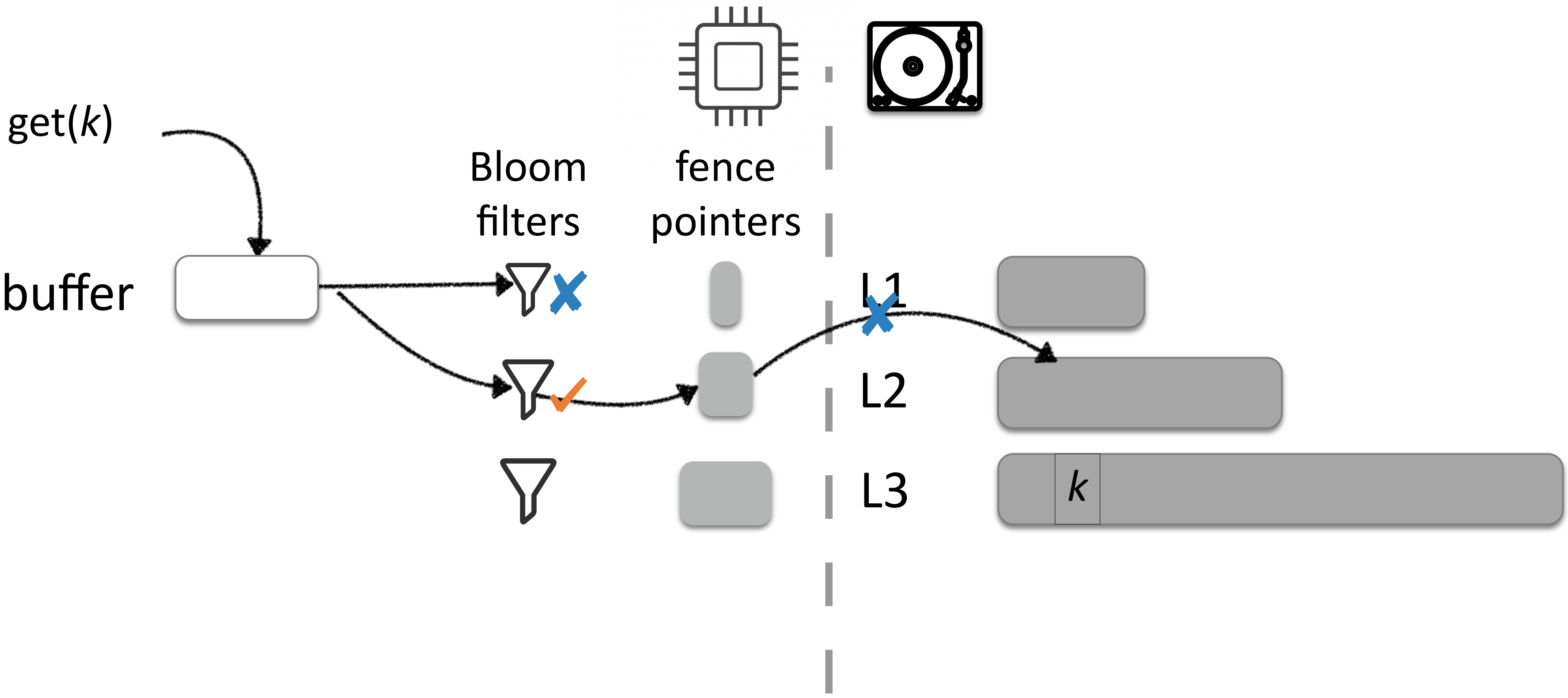
L3

k

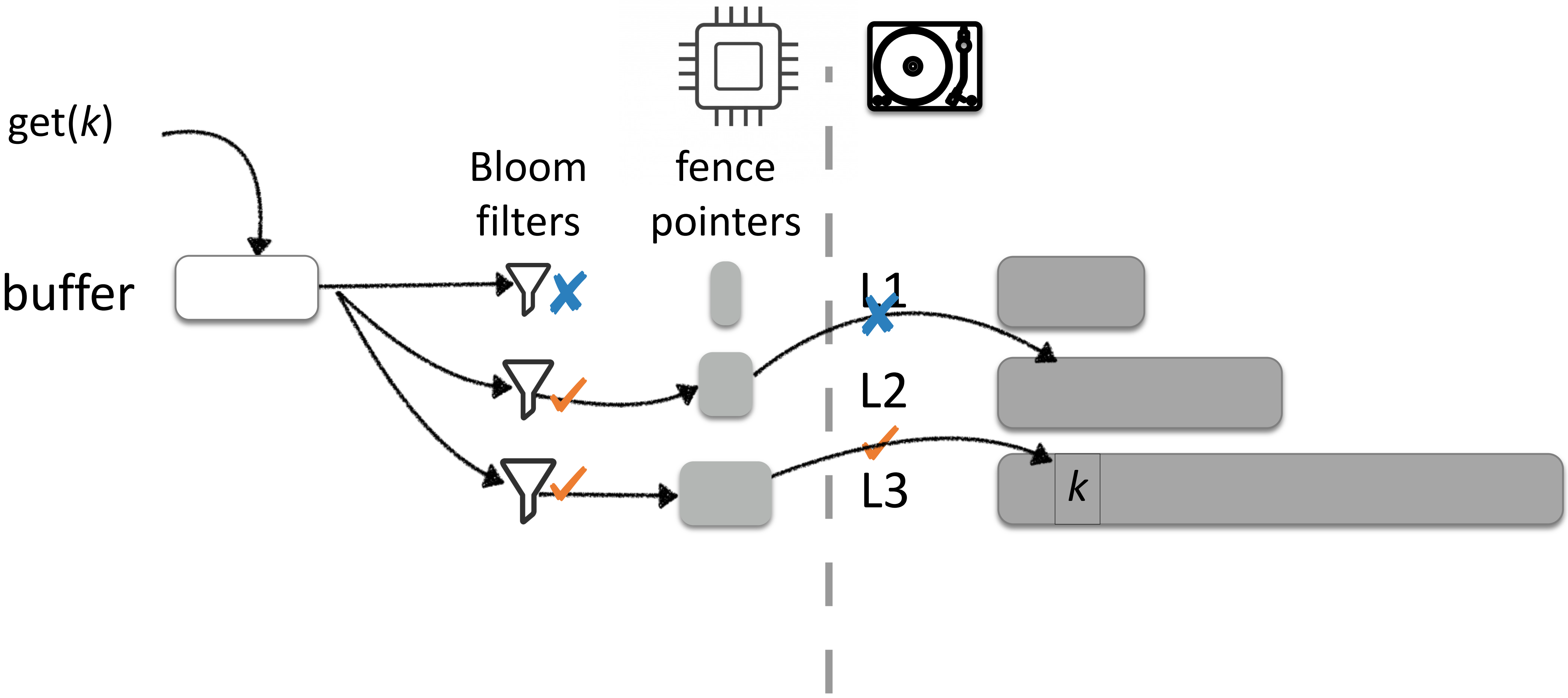
Log-Structured Merge Trees



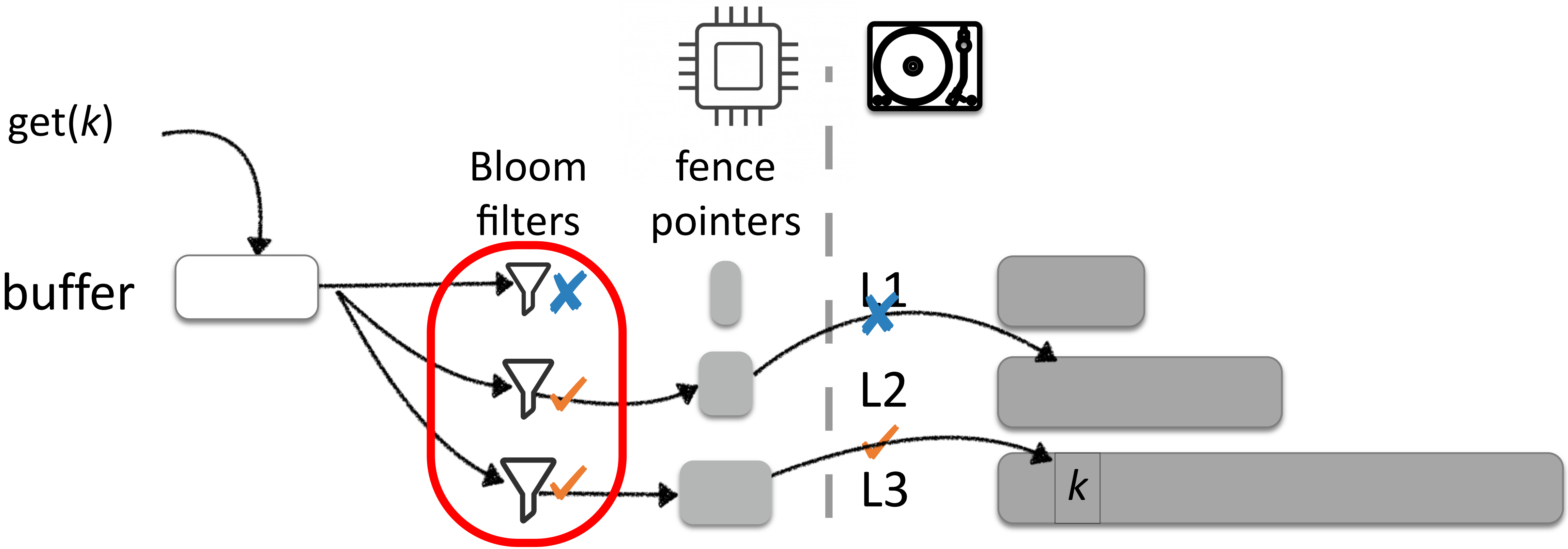
Log-Structured Merge Trees



Log-Structured Merge Trees

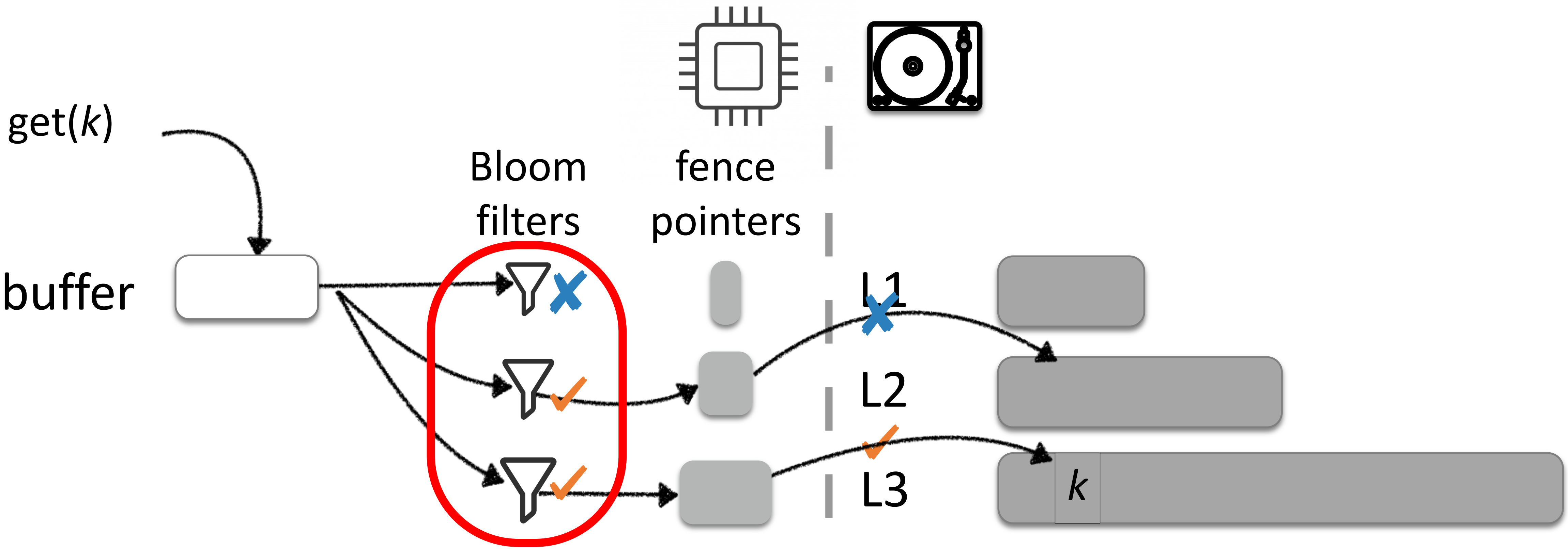


Log-Structured Merge Trees



All queries use a BF!

Log-Structured Merge Trees



All queries use a BF!

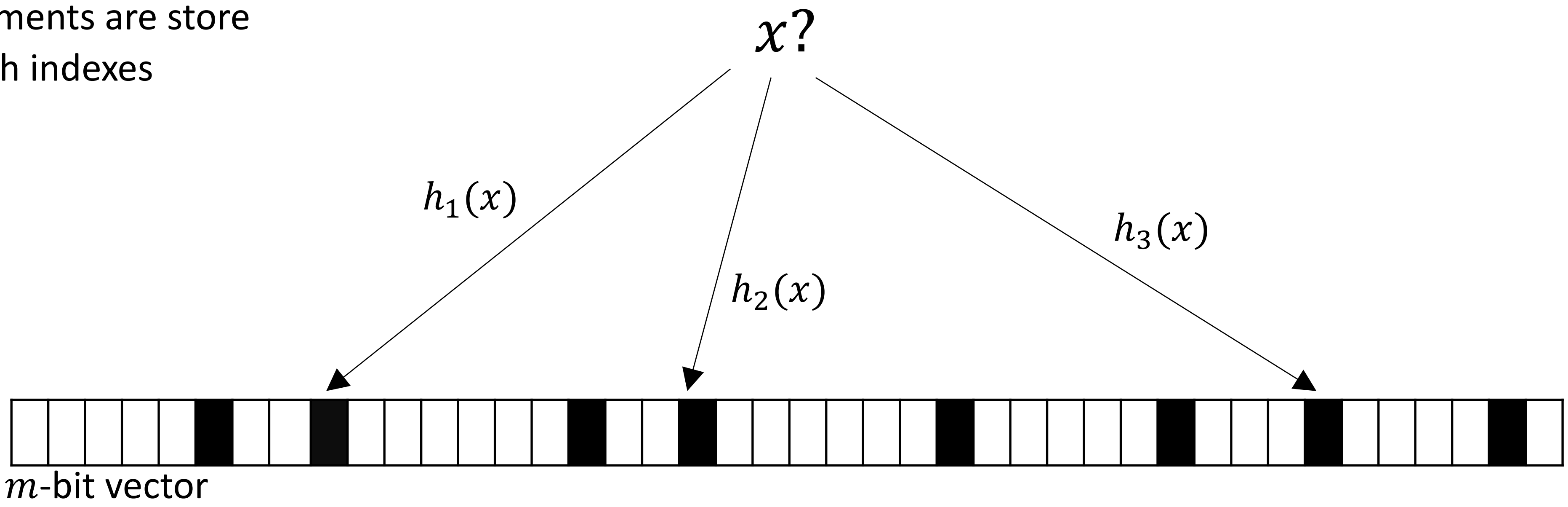
What is the cost of querying a Bloom filter?

Bloom Filter Query Cost

m-bit vector

n elements are store

k hash indexes



the fraction of empty queries

$$k \cdot \underline{T_H} + \underline{T_P} + \alpha \cdot f_p \cdot T_D + (1-\alpha) \cdot \underline{T_D}$$

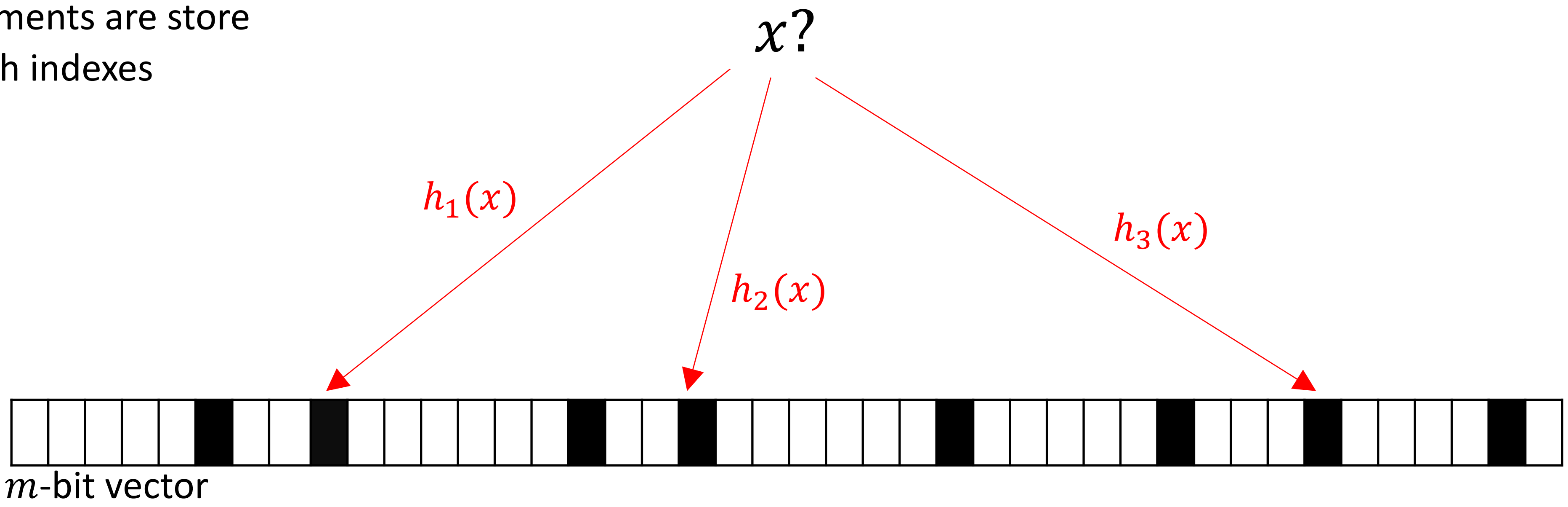
Hashing time Probing time Data access time

Bloom Filter Query Cost

m-bit vector

n elements are store

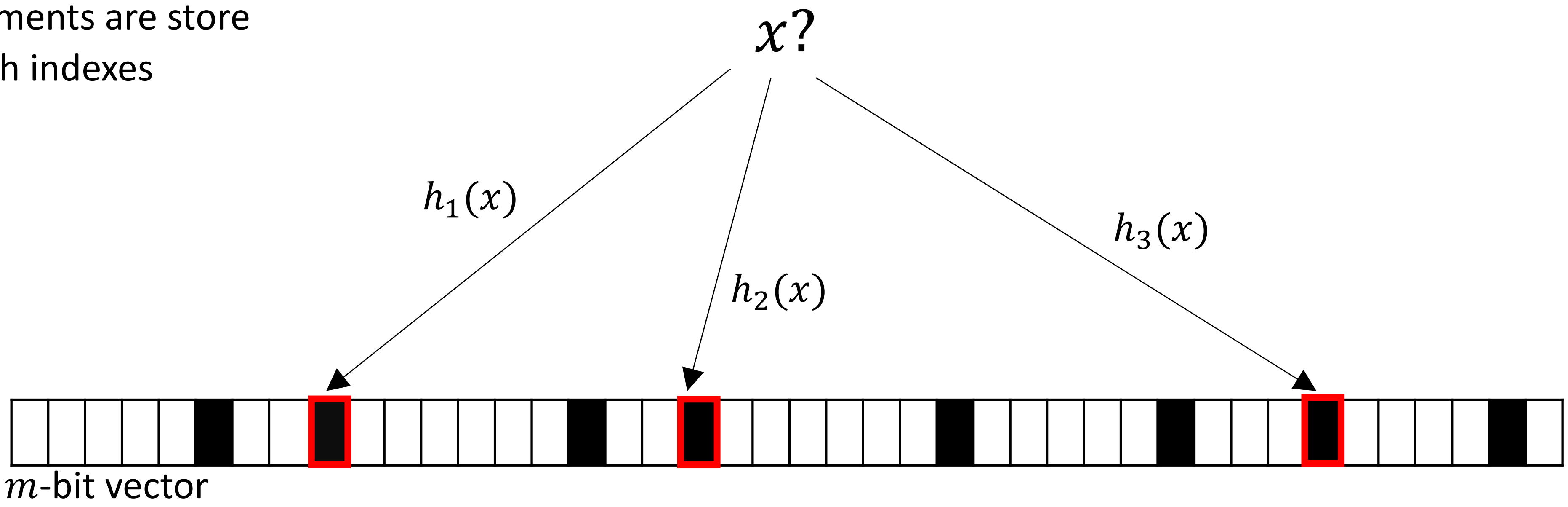
k hash indexes



$$\underline{k \cdot T_H} + T_P + \alpha \cdot f_p \cdot T_D + (1 - \alpha) \cdot T_D$$

Bloom Filter Query Cost

m-bit vector
n elements are store
k hash indexes

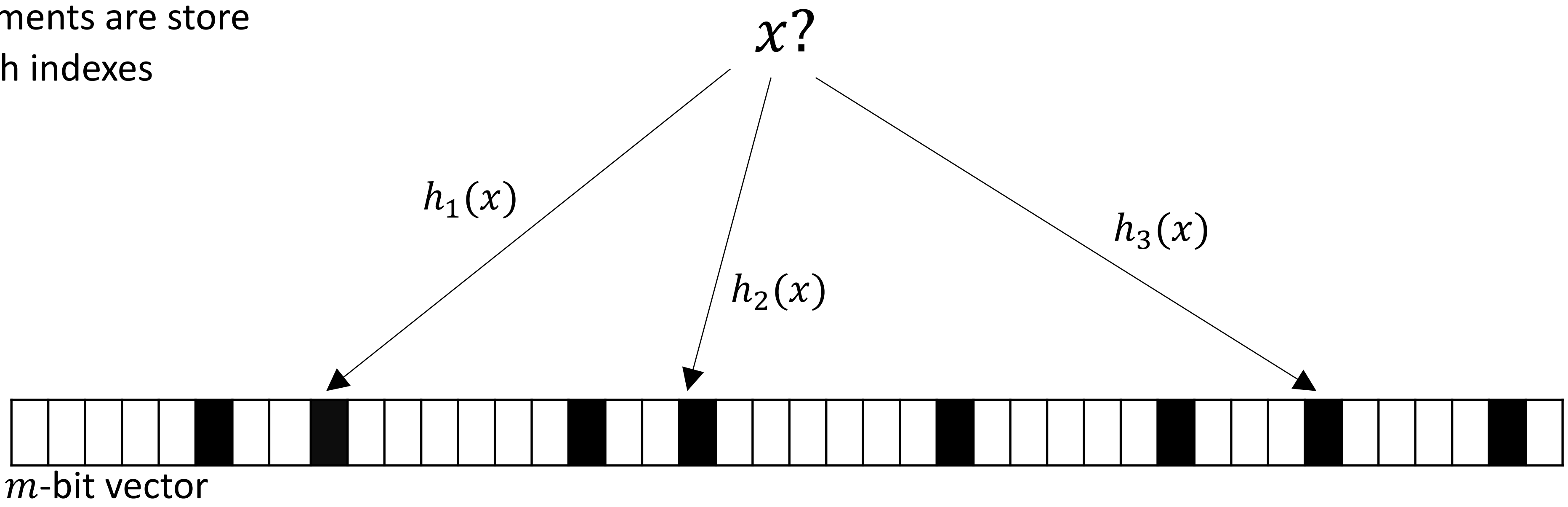


$$k \cdot T_H + \underline{T_P} + \alpha \cdot f_p \cdot T_D + (1-\alpha) \cdot T_D$$

α → the fraction of empty queries

Bloom Filter Query Cost

m-bit vector
n elements are store
k hash indexes

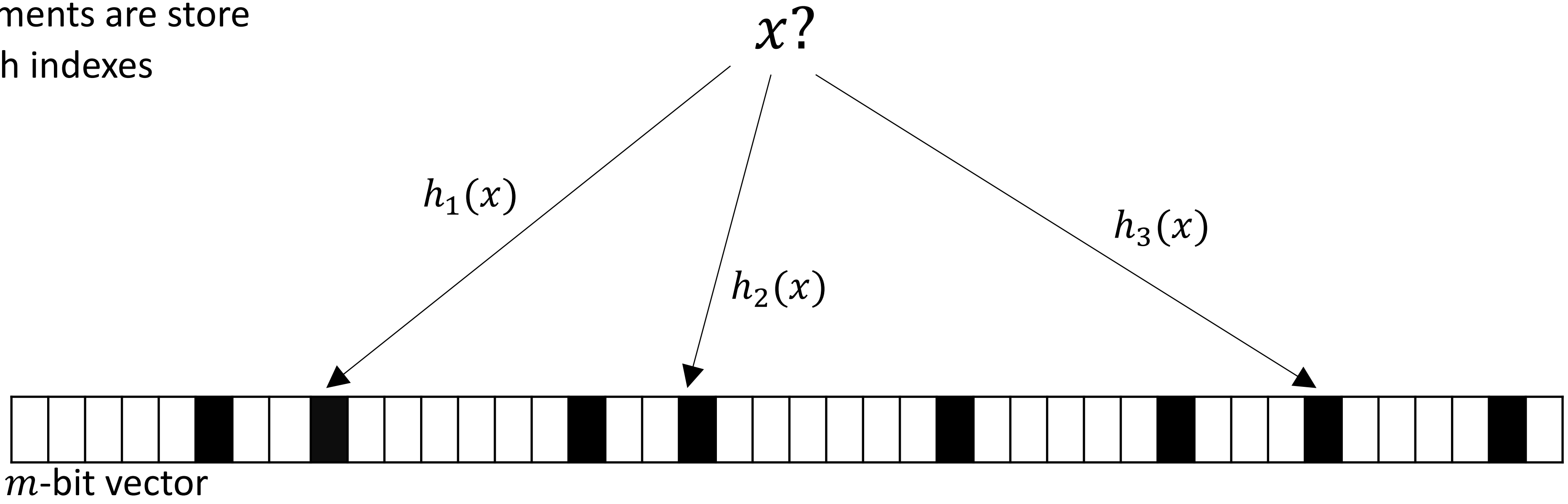


$$k \cdot T_H + T_P + \underline{\alpha \cdot f_p \cdot T_D + (1-\alpha) \cdot T_D}$$

the fraction of empty queries

Bloom Filter Query Cost

m-bit vector
n elements are store
k hash indexes



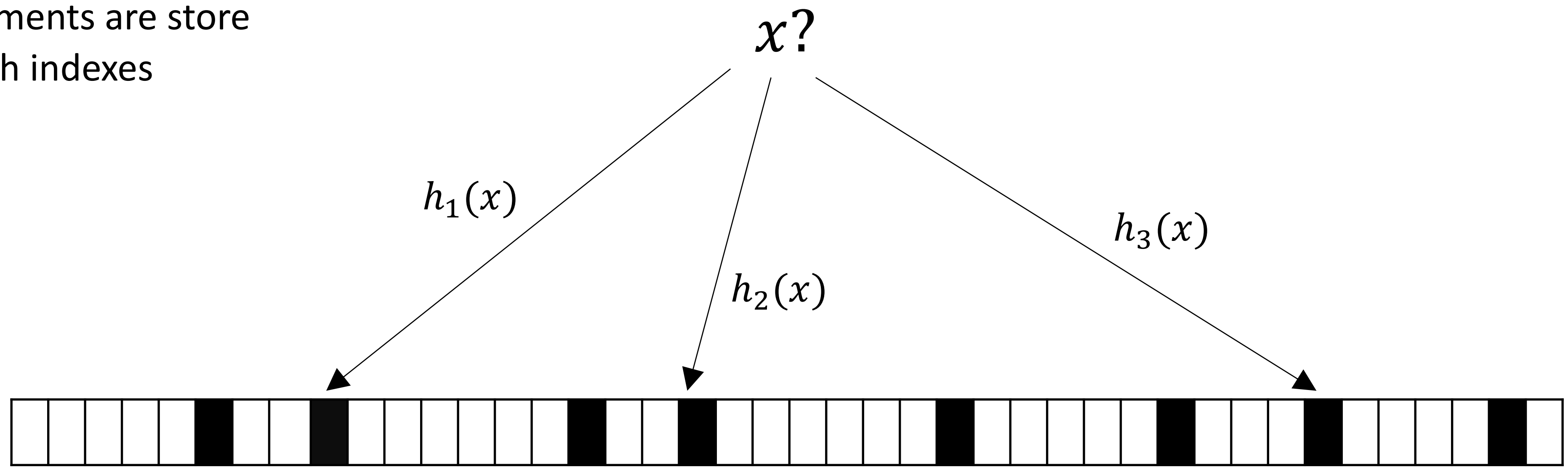
$$k \cdot T_H + T_P + \underbrace{\alpha \cdot f_p \cdot T_D}_{\text{empty}} + \underbrace{(1-\alpha) \cdot T_D}_{\text{non-empty}}$$

Bloom Filter Query Cost

m-bit vector

n elements are store

k hash indexes

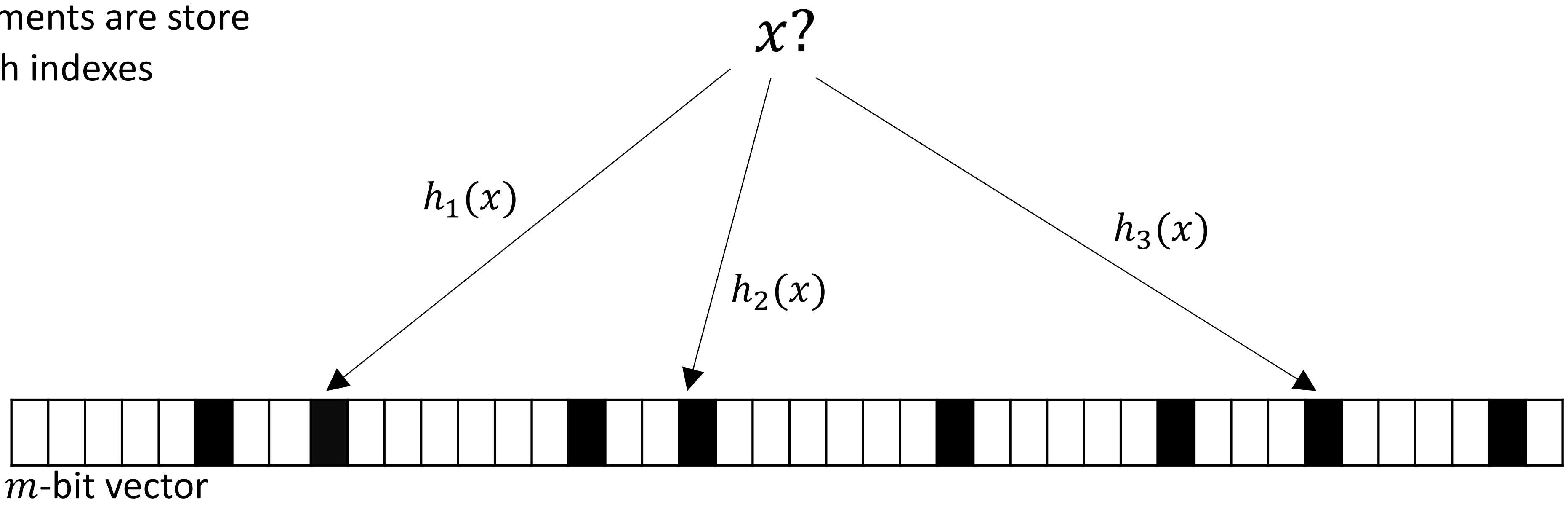


m-bit vector

$$\boxed{k \cdot T_H} + T_P + \alpha \cdot f_p \cdot T_D + (1 - \alpha) \cdot T_D$$

Bloom Filter Query Cost

m-bit vector
n elements are store
k hash indexes



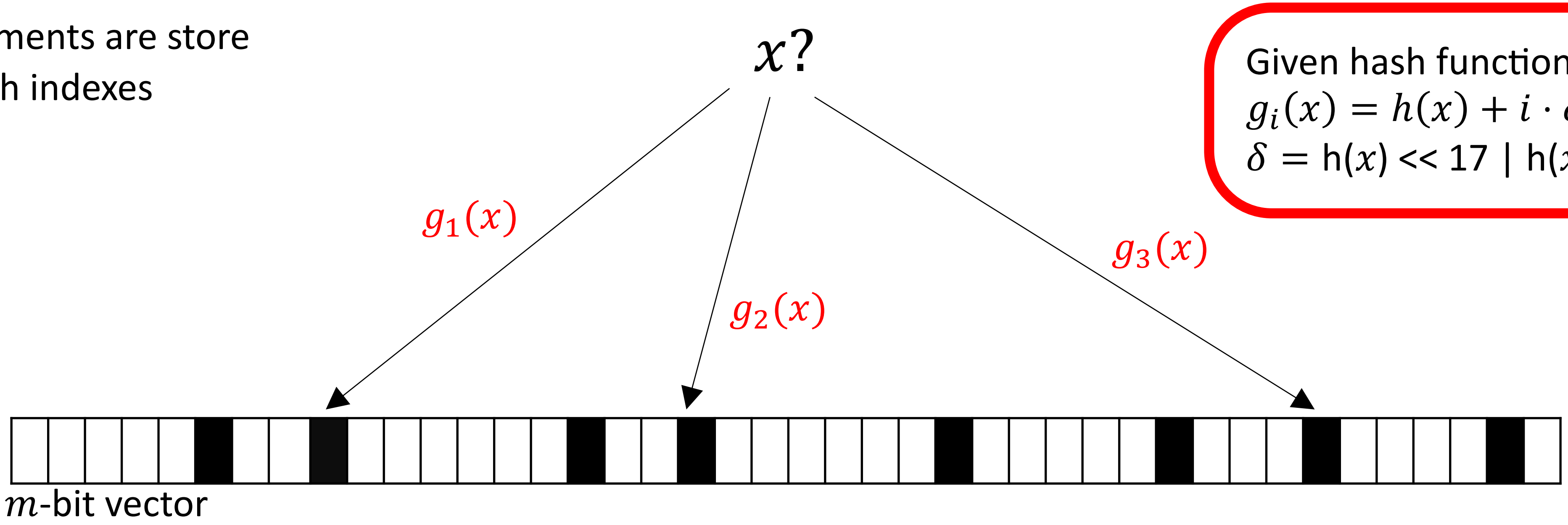
$$\boxed{k \cdot T_H} + T_P + \alpha \cdot f_p \cdot T_D + (1 - \alpha) \cdot T_D$$

→ a single hash function,
followed by much cheaper bitwise operations

Bloom Filter Query Cost

m-bit vector
n elements are store
k hash indexes

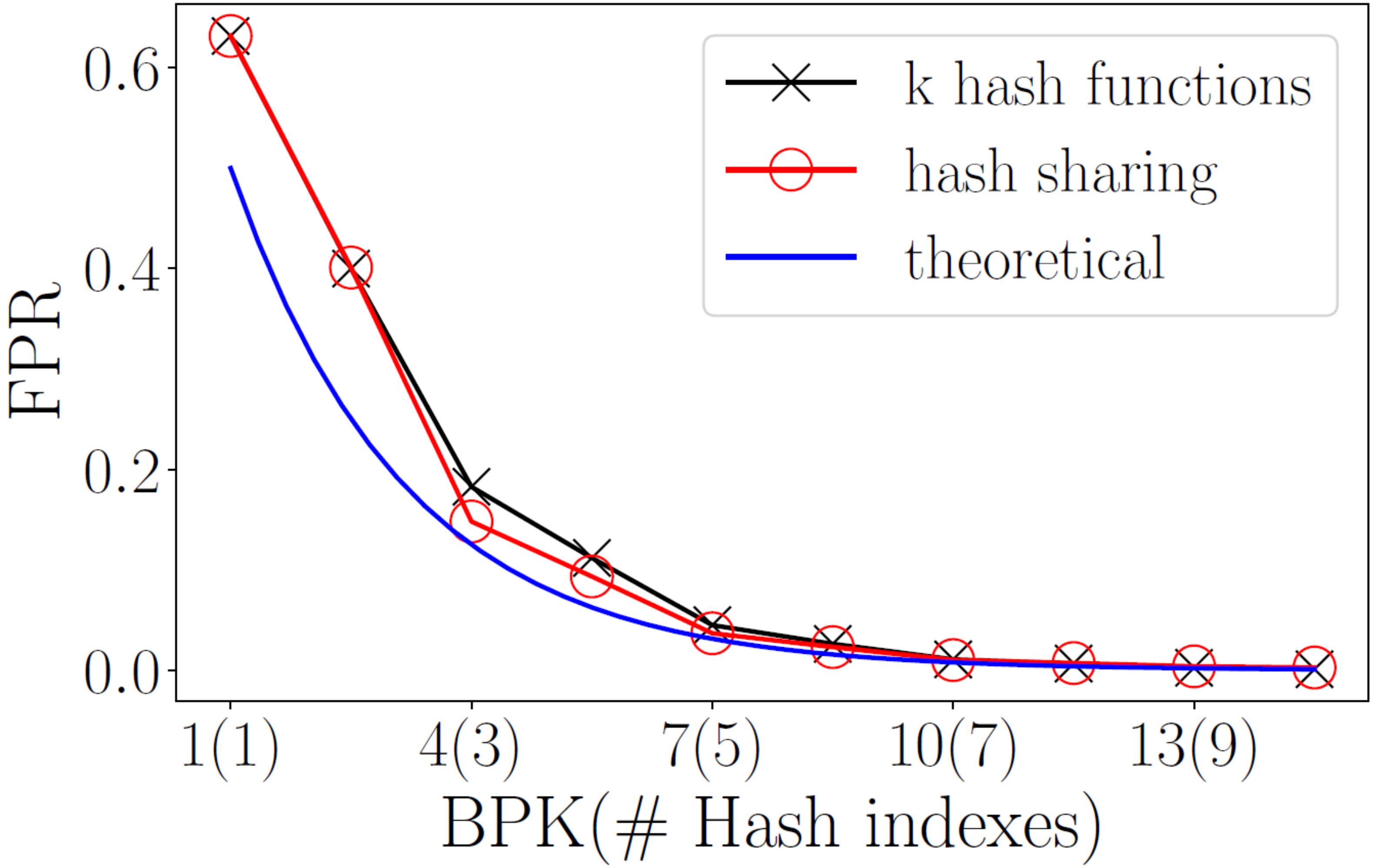
Given hash function, h
$g_i(x) = h(x) + i \cdot \delta$
δ = h(x) << 17 | h(x) >> 15



$$\cancel{k} \cdot T_H + T_P + \alpha \cdot f_p \cdot T_D + (1-\alpha) \cdot T_D$$

Bloom Filter False Positive Rate

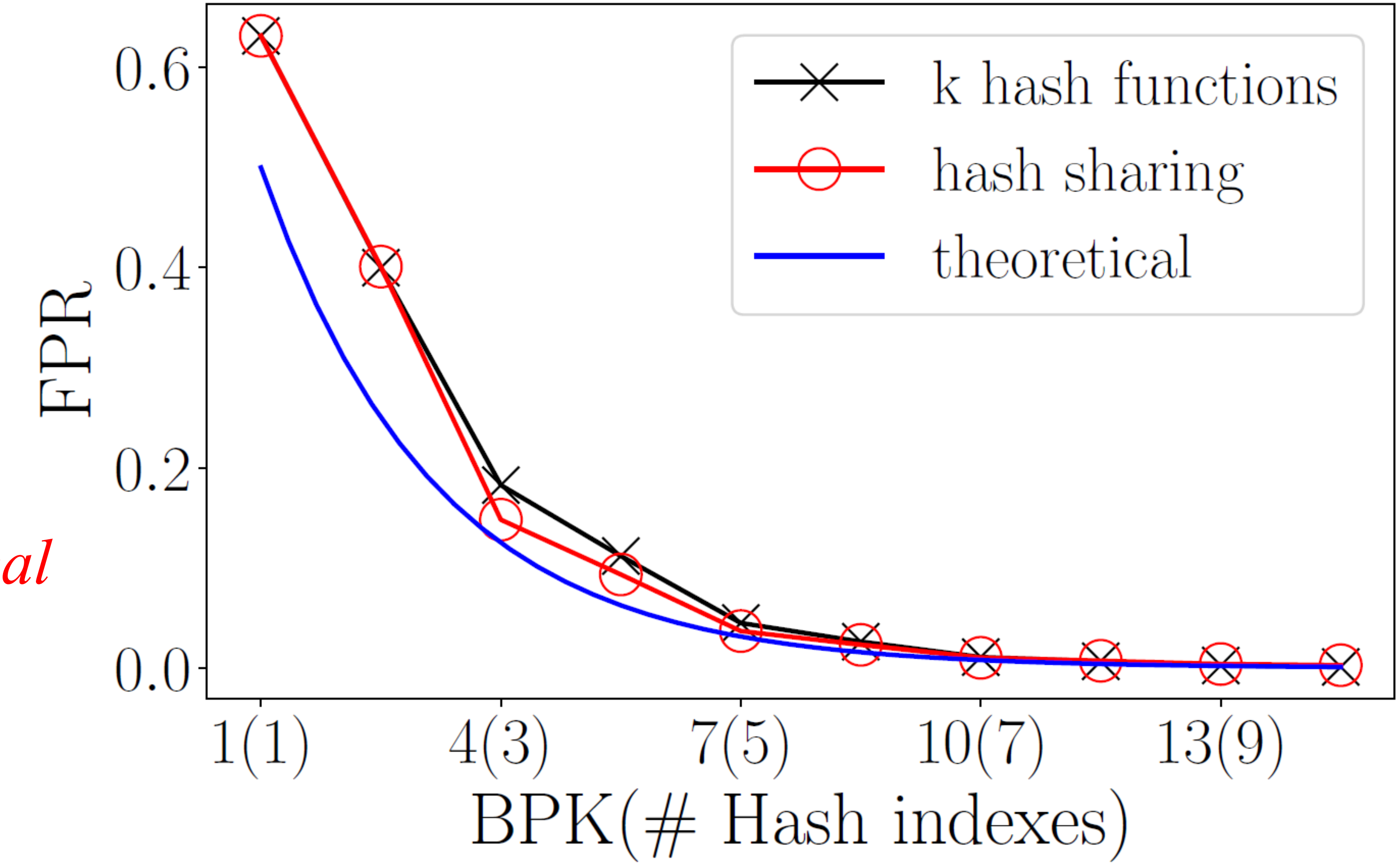
k vs. single hash function



Bloom Filter False Positive Rate

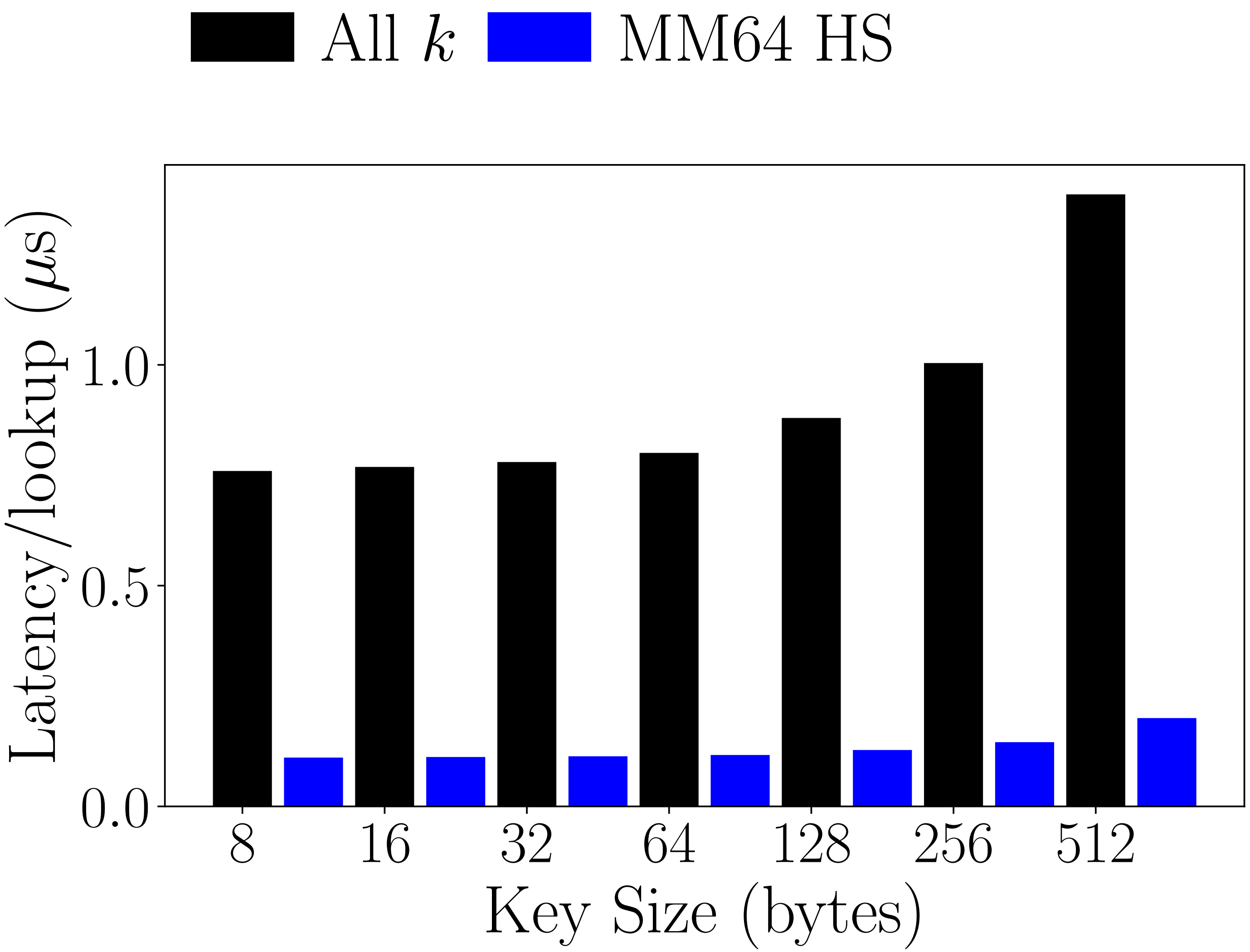
k vs. single hash function

FPR close-to-theoretical



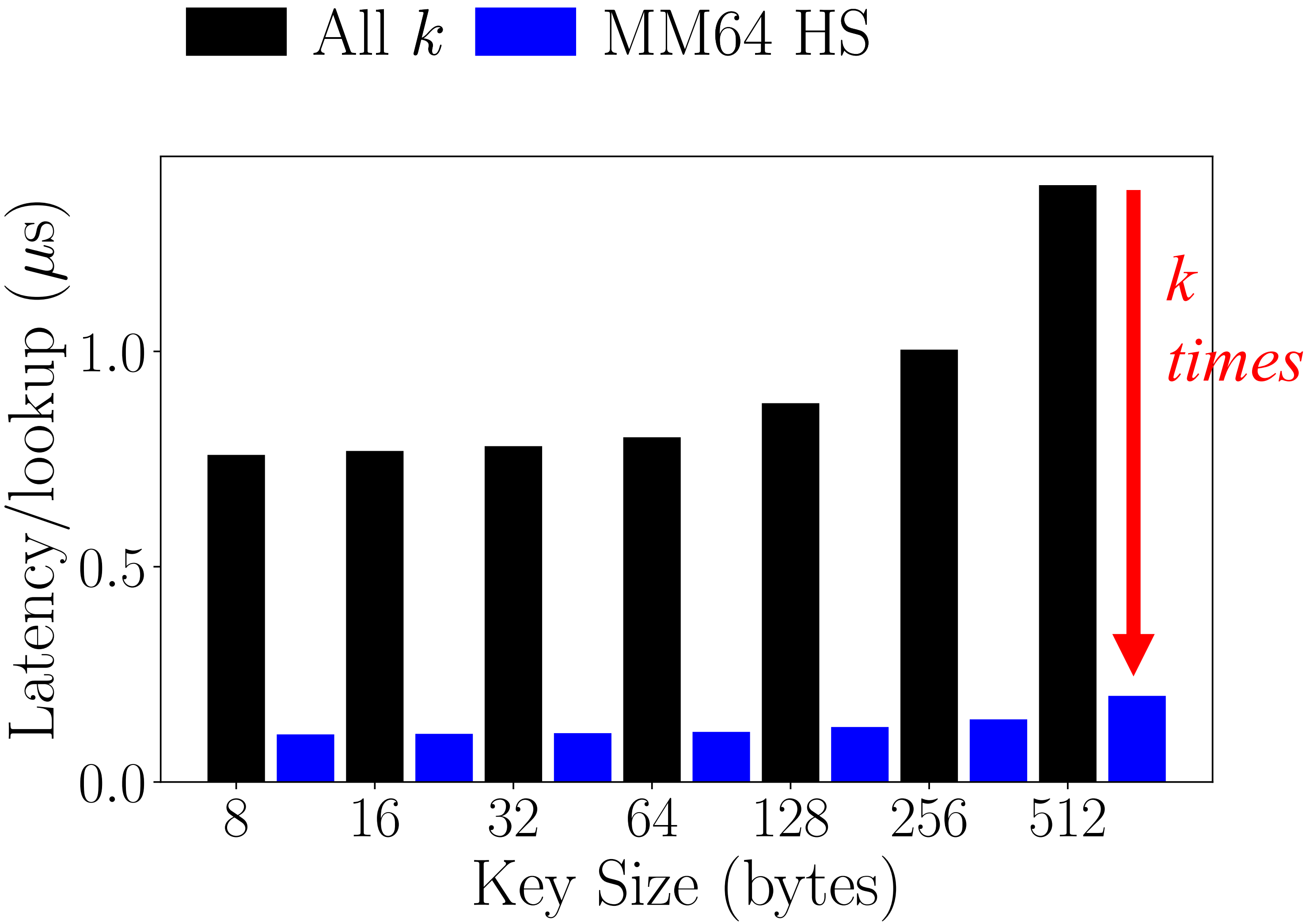
Bloom Filter Lookup Latency

k vs. single hash function



Bloom Filter Lookup Latency

k vs. single hash function



What is the Lookup Cost in LSM-Trees?

What is the Lookup Cost in LSM-Trees?

Leveling

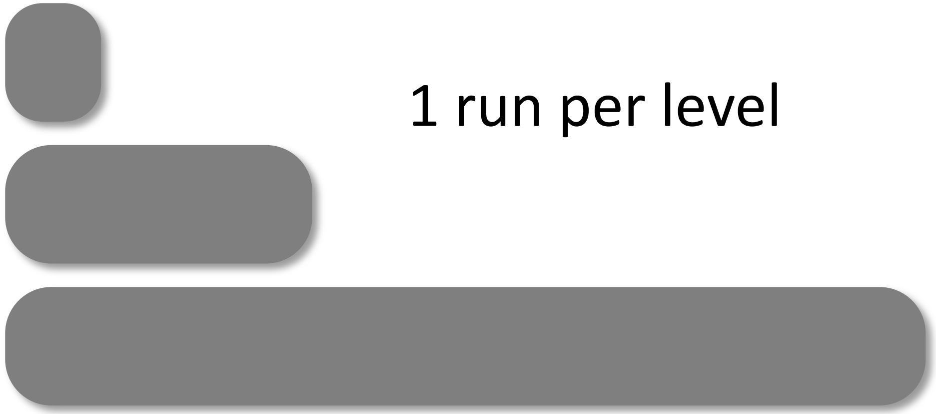
read-optimized

Tiering

write-optimized

Leveling

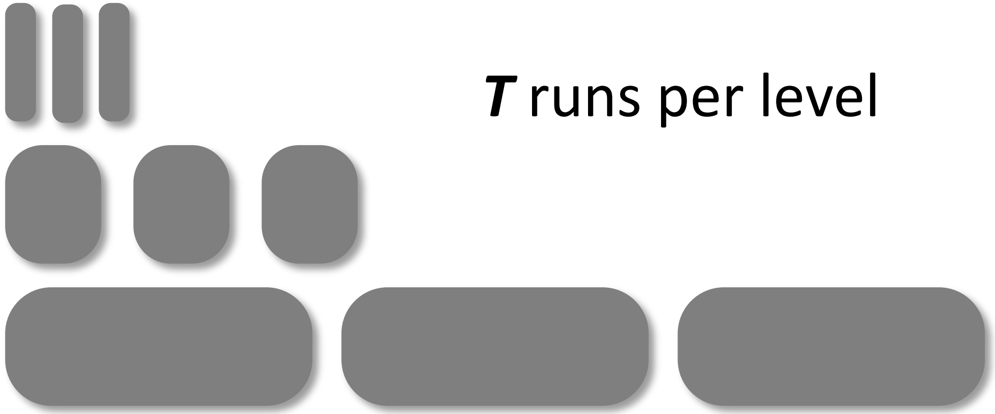
read-optimized



1 run per level

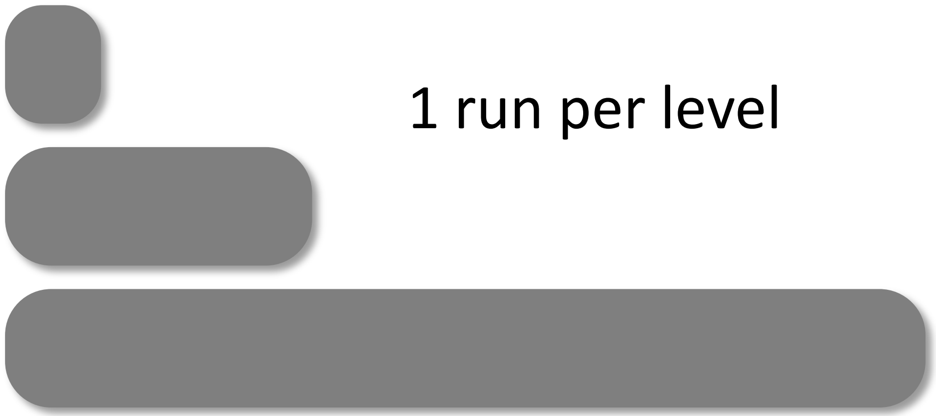
Tiering

write-optimized



T runs per level

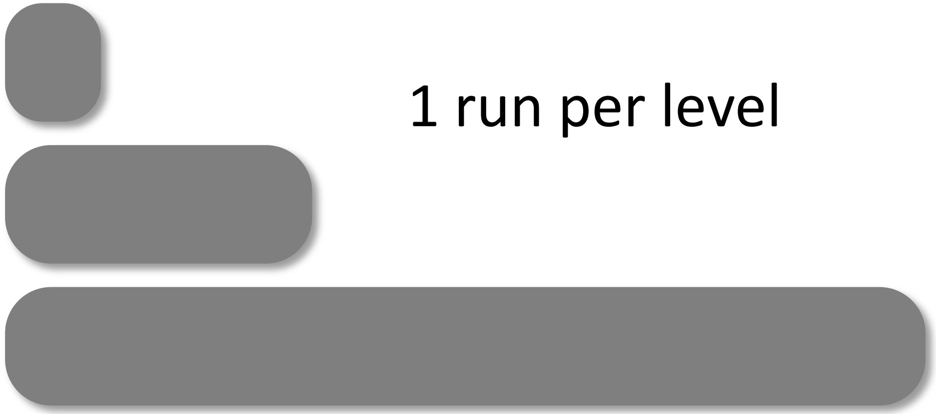
Lookup Cost in a Leveled LSM-Tree



1 run per level

Lookup cost in level i, $\mathcal{T}(i)$

Lookup Cost in a Leveled LSM-Tree



1 run per level

Lookup cost in level i, $\mathcal{T}(i)$

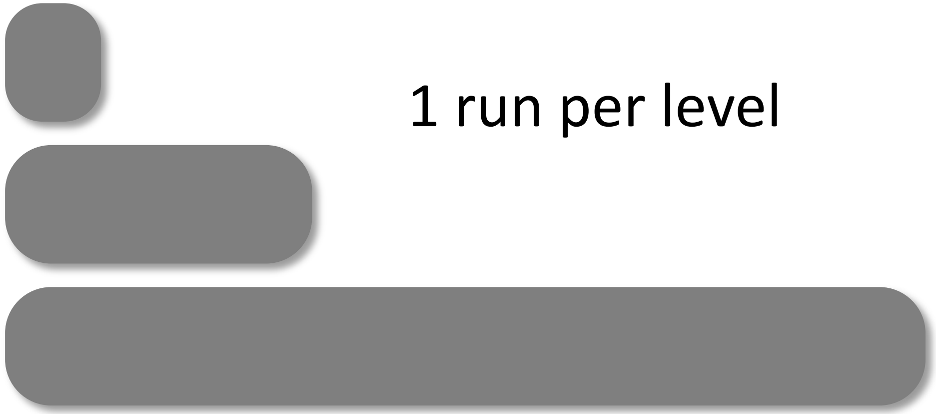
- empty (α_i)

$$\alpha_i \cdot (T_H + T_P + f_p \cdot T_D)$$

- non-empty ($1 - \alpha_i$)

$$(1 - \alpha_i) \cdot (T_H + T_P + T_D)$$

Lookup Cost in a Leveled LSM-Tree



1 run per level

Lookup cost in level i, $\mathcal{T}(i)$

- empty (α_i)

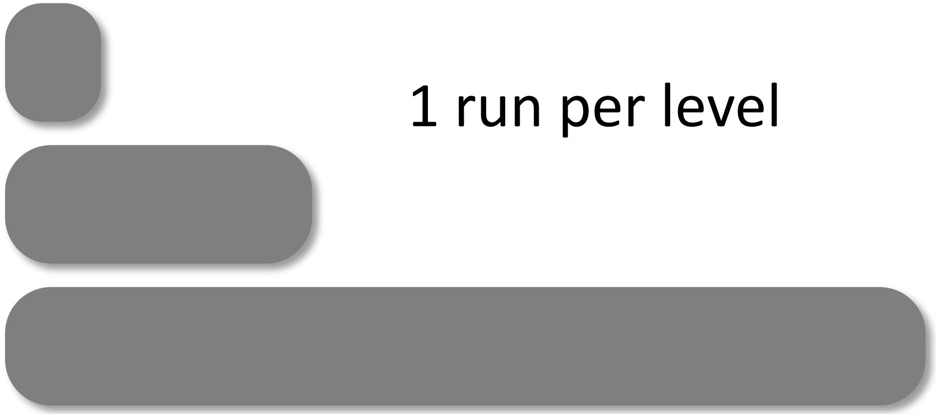
  $\alpha_i \cdot (T_H + T_P + f_p \cdot T_D)$
- non-empty ($1 - \alpha_i$)

  $(1 - \alpha_i) \cdot (T_H + T_P + T_D)$

$\boxed{+}$

$$\mathcal{T}(i) = T_H + T_P + \alpha_i \cdot f_p \cdot T_D + (1 - \alpha_i) \cdot T_D$$

Lookup Cost in a Leveled LSM-Tree



1 run per level

Lookup cost in level i, $\mathcal{T}(i)$

- empty (α_i)

  $\alpha_i \cdot (T_H + T_P + f_p \cdot T_D)$

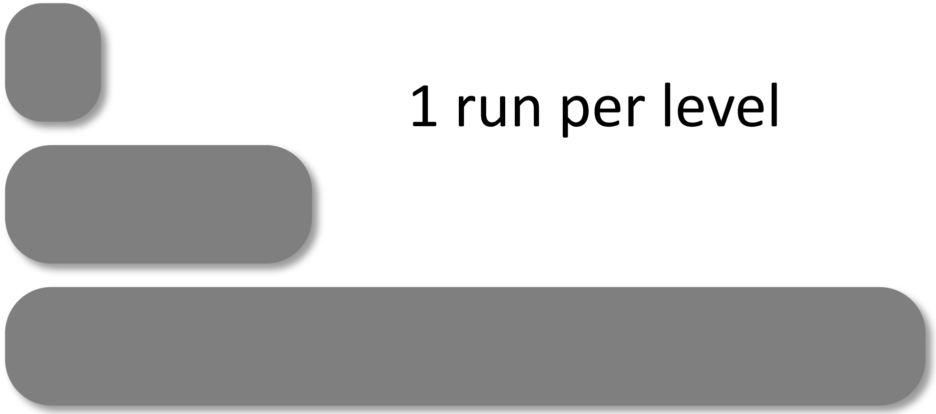
- non-empty ($1 - \alpha_i$)

  $(1 - \alpha_i) \cdot (T_H + T_P + T_D)$

$\boxed{+}$

$$cost \approx \left(L - \frac{1-\alpha}{T-1}\right) \cdot (T_H + T_P) + \left(L - \frac{1-\alpha}{T-1} - 1 + \alpha\right) \cdot (f_p \cdot T_D) + (1-\alpha) \cdot T_D$$

Lookup Cost in a Leveled LSM-Tree



1 run per level

Lookup cost in level i, $\mathcal{T}(i)$

- empty (α_i)

  $\alpha_i \cdot (T_H + T_P + f_p \cdot T_D)$

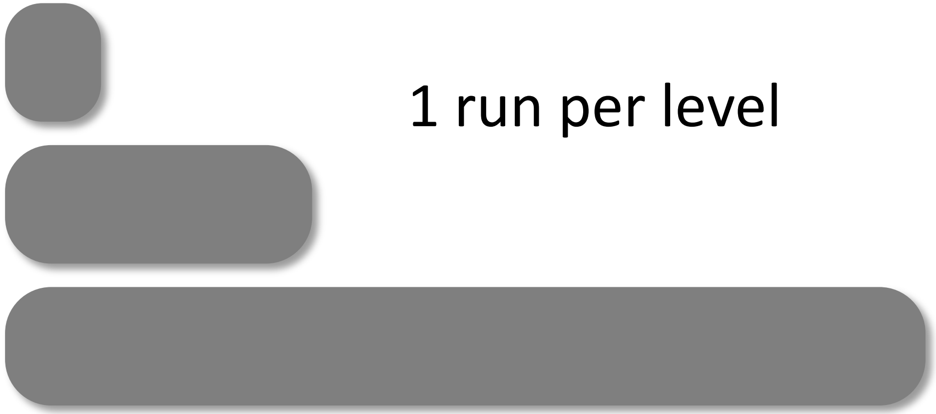
- non-empty ($1 - \alpha_i$)

  $(1 - \alpha_i) \cdot (T_H + T_P + T_D)$

$+$

$$cost \approx \underbrace{\left(L - \frac{1-\alpha}{T-1}\right) \cdot (T_H + T_P)}_{\text{Bloom filter cost}} + \left(L - \frac{1-\alpha}{T-1} - 1 + \alpha\right) \cdot (f_p \cdot T_D) + (1 - \alpha) \cdot T_D$$

Lookup Cost in a Leveled LSM-Tree



1 run per level

Lookup cost in level i, $\mathcal{T}(i)$

- empty (α_i)

  $$\alpha_i \cdot (T_{BF} + f_p \cdot T_D)$$

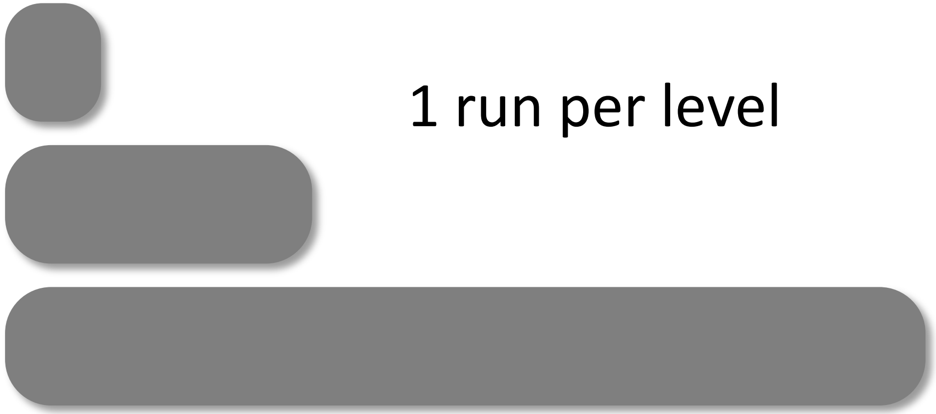
- non-empty ($1-\alpha_i$)

  $$(1-\alpha_i) \cdot (T_{BF} + T_D)$$

$$\boxed{+}$$

$$cost \approx \underbrace{\left(L - \frac{1-\alpha}{T-1}\right) \cdot (T_H + T_P)}_{\text{Bloom filter cost}} + \underbrace{\left(L - \frac{1-\alpha}{T-1} - 1 + \alpha\right) \cdot (f_p \cdot T_D)}_{\text{Data access due to false positives}} + (1-\alpha) \cdot T_D$$

Lookup Cost in a Leveled LSM-Tree



1 run per level

Lookup cost in level i, $\mathcal{T}(i)$

- empty (α_i)

  $\alpha_i \cdot (T_{BF} + f_p \cdot T_D)$

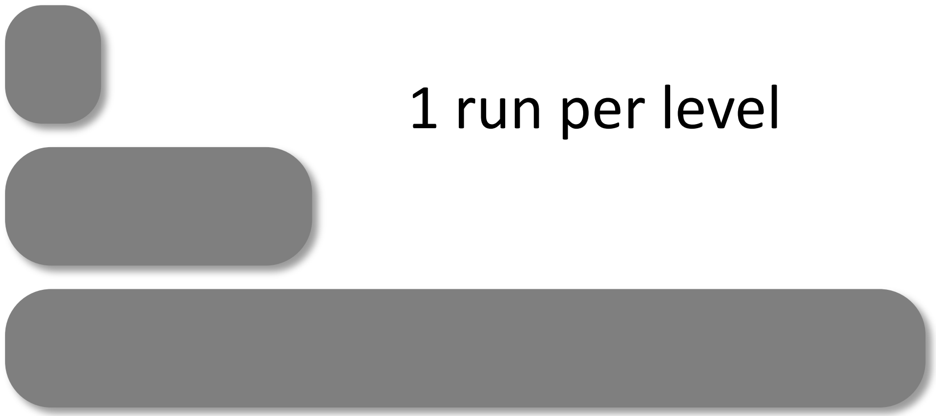
- non-empty ($1 - \alpha_i$)

  $(1 - \alpha_i) \cdot (T_{BF} + T_D)$

$\boxed{+}$

$$cost \approx \underbrace{\left(L - \frac{1-\alpha}{T-1}\right) \cdot (T_H + T_P)}_{\text{Bloom filter cost}} + \underbrace{\left(L - \frac{1-\alpha}{T-1} - 1 + \alpha\right) \cdot (f_p \cdot T_D)}_{\text{Data access due to false positives}} + \underbrace{(1-\alpha) \cdot T_D}_{\text{Data access}}$$

Lookup Cost in a Leveled LSM-Tree



1 run per level

Lookup cost in level i, $\mathcal{T}(i)$

- empty (α_i)

  $\alpha_i \cdot (T_{BF} + f_p \cdot T_D)$

- non-empty ($1 - \alpha_i$)

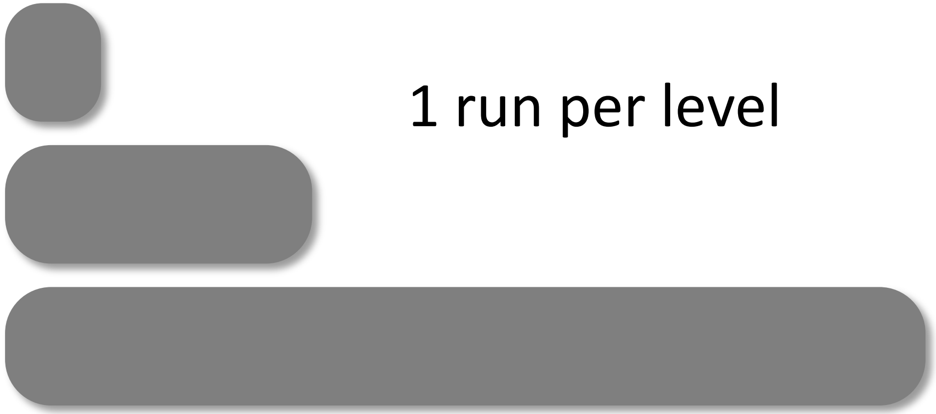
  $(1 - \alpha_i) \cdot (T_{BF} + T_D)$

$\boxed{+}$

$$cost \approx \overset{\propto L}{\boxed{\left(L - \frac{1-\alpha}{T-1}\right) \cdot (T_H + T_P)}} + \left(L - \frac{1-\alpha}{T-1} - 1 + \alpha\right) \cdot (f_p \cdot T_D) + (1 - \alpha) \cdot T_D$$

Hashing accumulates as L grows (larger data size)

Lookup Cost in a Leveled LSM-Tree



1 run per level

Lookup cost in level i, $\mathcal{T}(i)$

- empty (α_i)

$$\alpha_i \cdot (T_{BF} + f_p \cdot T_D)$$

- non-empty ($1 - \alpha_i$)

$$(1 - \alpha_i) \cdot (T_{BF} + T_D)$$

$$\boxed{+}$$

$$cost \approx \left(L - \frac{1-\alpha}{T-1}\right) \cdot (T_{BF}) + \left(L - \frac{1-\alpha}{T-1} - 1 + \alpha\right) \cdot (f_p \cdot T_D) + \boxed{(1-\alpha) \cdot T_D}$$

Hashing is more prominent for empty queries

Storage Access vs. Hashing

Operation	Latency	Normalized
4KB access on SDD	113 μs	706×
4KB access on PCIe SDD	10 μs	62.5×
4KB access on emulated NVM	**250 ns**	**1.56×**
4KB access on Memory	160 ns	1×
Murmur Hash of 1KB	235 ns	1.47×

Storage Access vs. Hashing

Operation	Latency	Normalized
4KB access on SDD	$113\ \mu s$	$706\times$
4KB access on PCIe SDD	$10\ \mu s$	$62.5\times$
4KB access on emulated NVM	$\mathbf{250\ ns}$	$\mathbf{1.56}\times$
4KB access on Memory	$160\ \mathrm{ns}$	$1\times$
Murmur Hash of 1KB	$\mathbf{235}\ \boldsymbol{ns}$	$\mathbf{1.47}\times$

10% faster than NVM

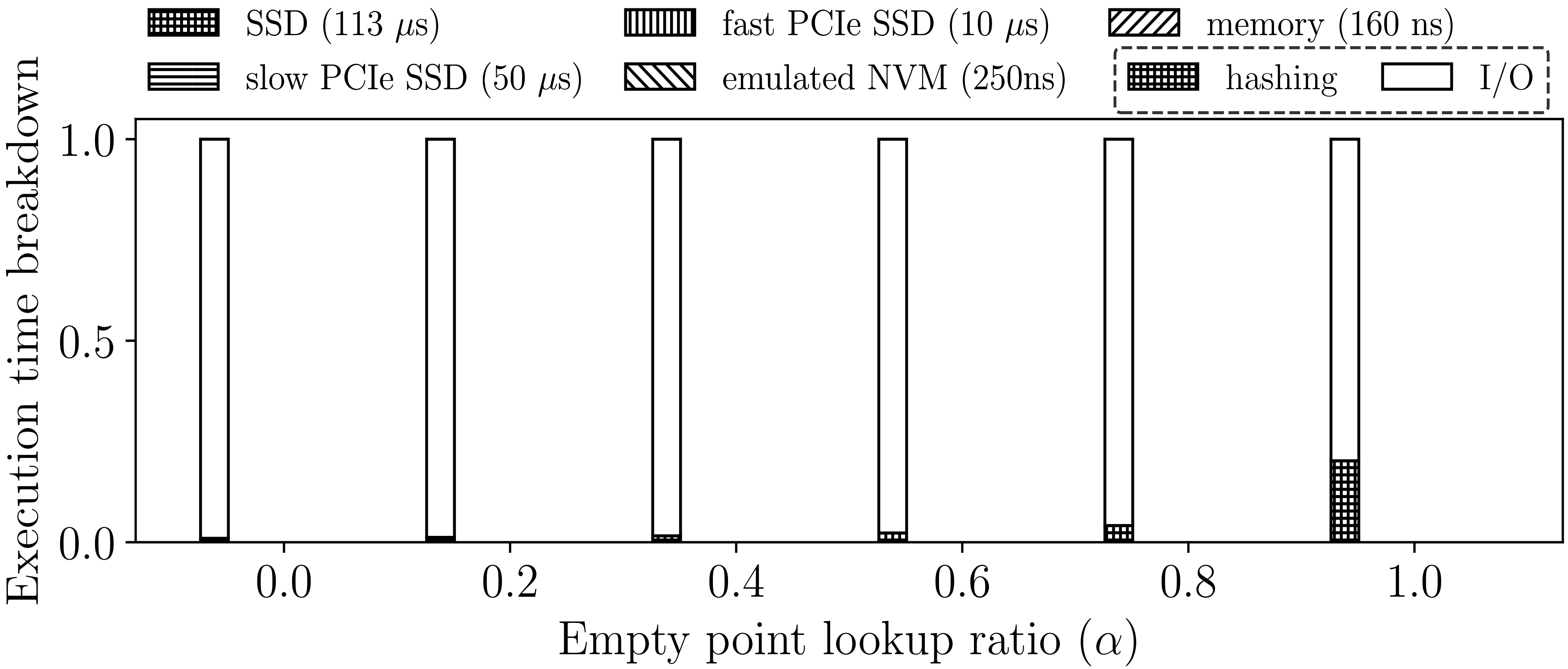
Storage Access vs. Hashing

Operation	Latency	Normalized
4KB access on SDD	$113\ \mu s$	$706\times$
4KB access on PCIe SDD	$10\ \mu s$	$62.5\times$
4KB access on emulated NVM	$\mathbf{250\ ns}$	$\mathbf{1.56\times}$
4KB access on Memory	$160\ \mathrm{ns}$	$1\times$
Murmur Hash of 1KB	$\mathbf{235\ ns}$	$\mathbf{1.47\times}$

10% faster than NVM

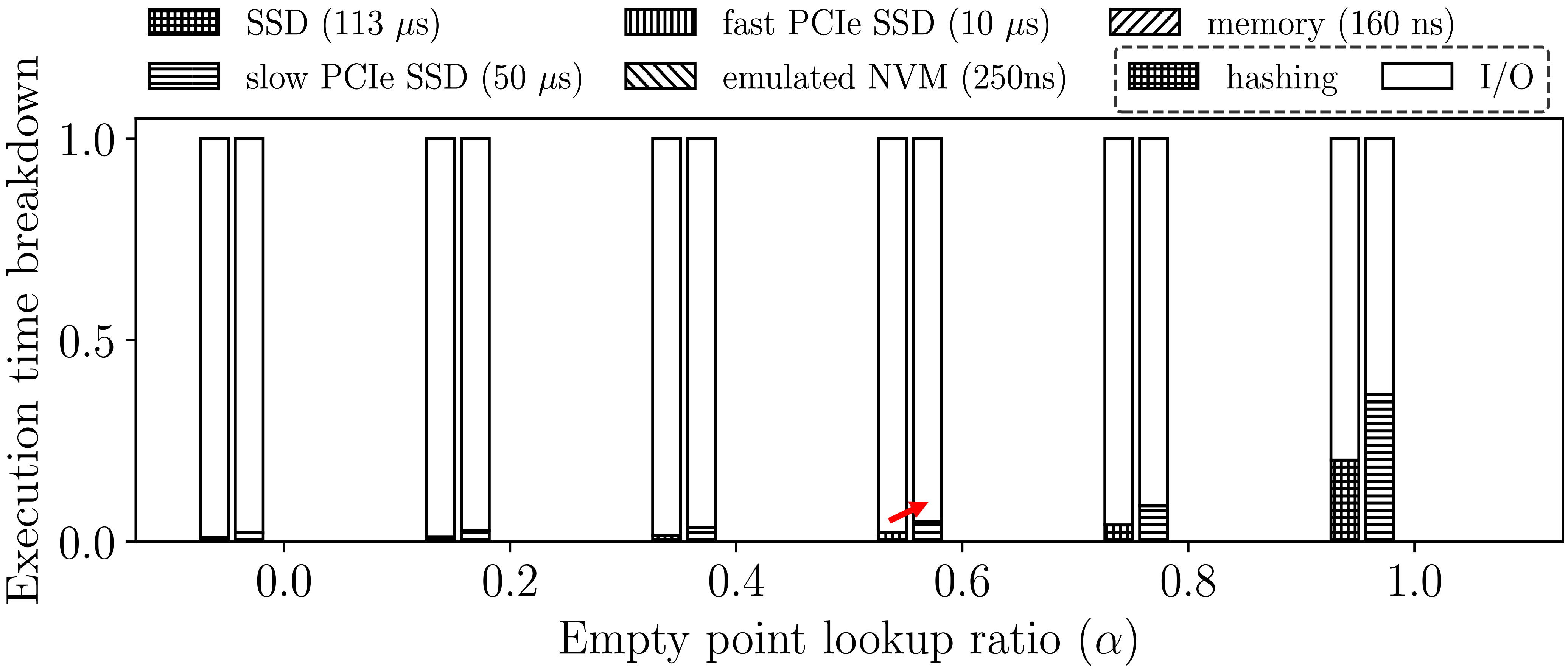
What is the time spent hashing as we move to faster devices?

Hashing Overhead in a Leveled LSM-Tree



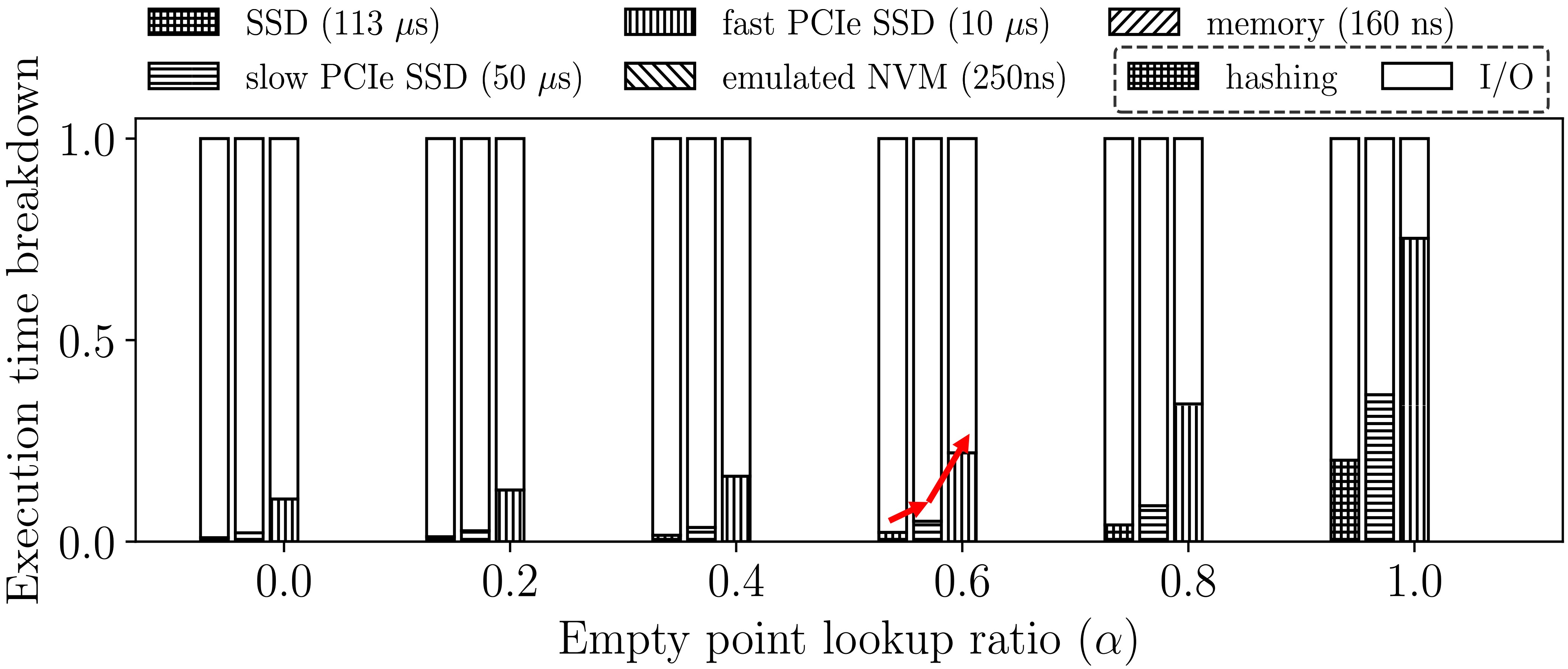
In SSDs, hashing is over 10% for all empty lookups

Hashing Overhead in a Leveled LSM-Tree



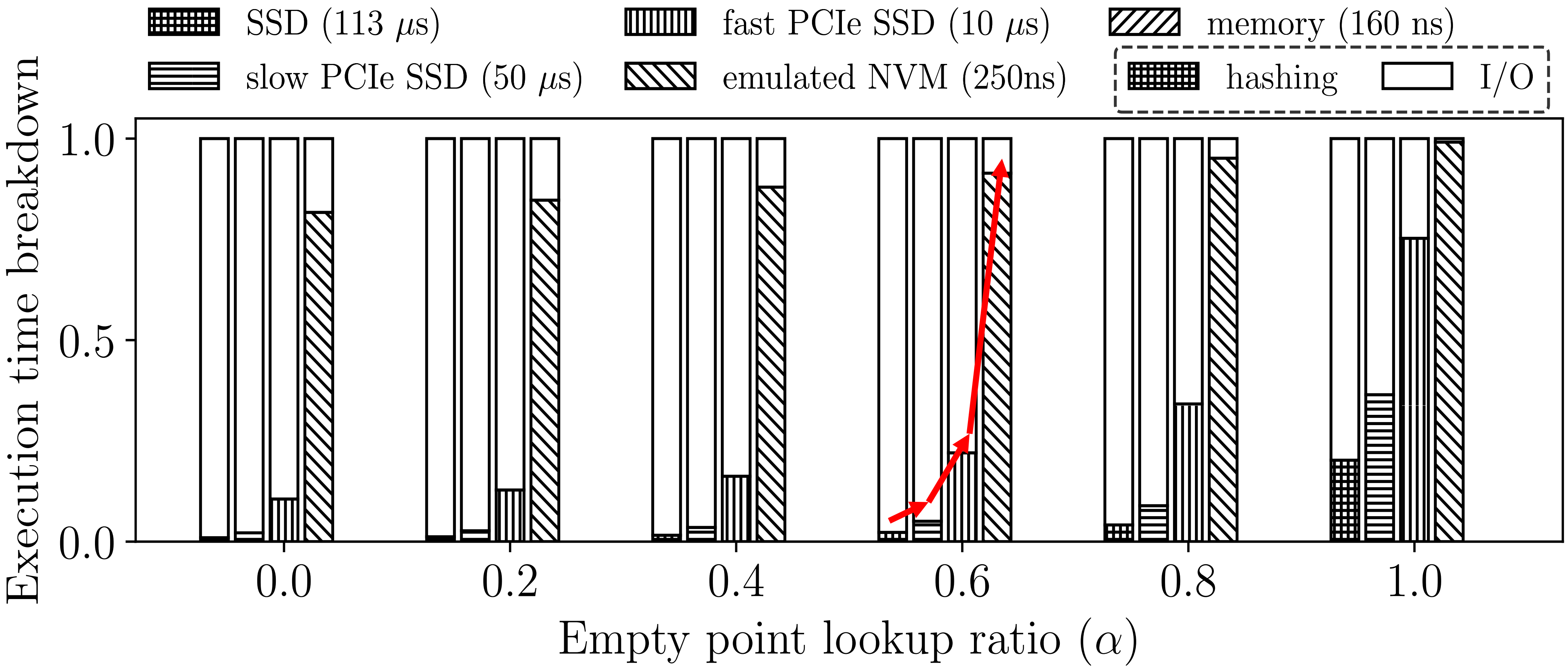
Hashing is getting more dominant

Hashing Overhead in a Leveled LSM-Tree



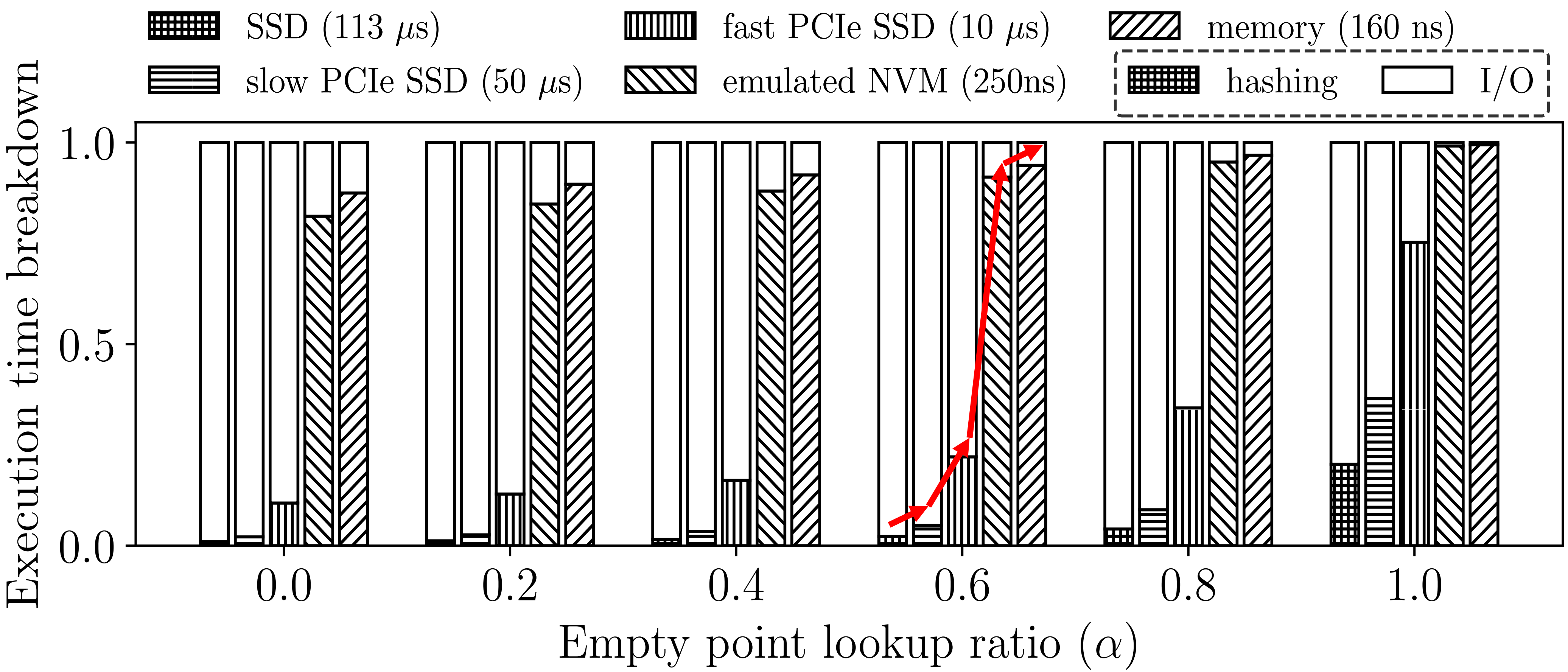
Hashing is getting more dominant

Hashing Overhead in a Leveled LSM-Tree



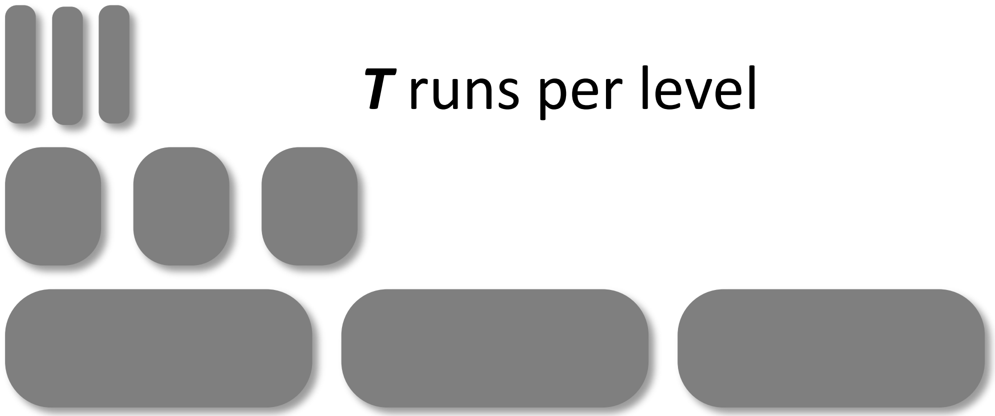
Hashing is getting more dominant

Hashing Overhead in a Leveled LSM-Tree



Hashing is getting more dominant

Lookup Cost in a Tiered LSM-Tree



T runs per level

Lookup cost in level i, $\mathcal{T}(i)$

- empty (α_i)

  $\alpha_i \cdot T \cdot (\,T_H + T_P + f_p \cdot T_D)$

- non-empty $(1 - \alpha_i)$

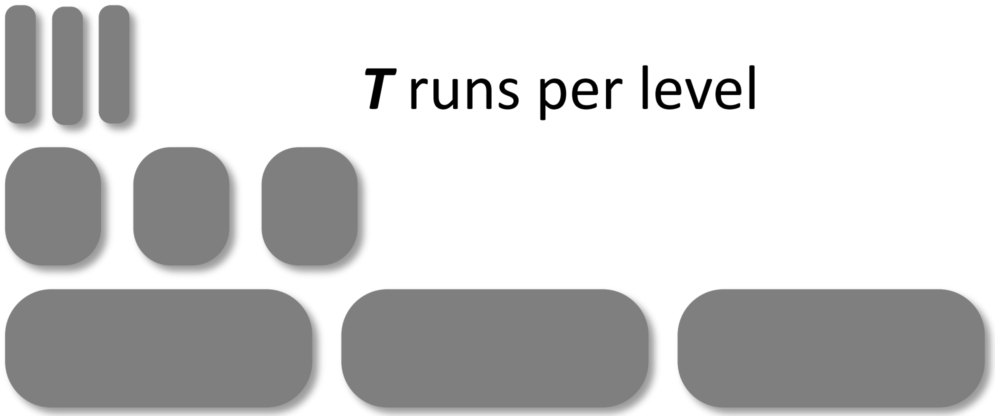
  $(1-\alpha_i) \cdot \dfrac{T+1}{2} \cdot (\,T_H + T_P) + (1-\alpha_i) \cdot T_D$

$cost \;\approx \left(T \cdot L - \frac{T+1}{2}(1-\alpha)\right) \cdot (T_{BF})$ **Bloom filter cost**

$+ \left(T \cdot L - (1-\alpha) \cdot (T+1)\right) \cdot \left(f_p \cdot T_D\right)$ **Data access due to false positives**

$+ (1-\alpha) \cdot\; T_D$ **Data access**

Lookup Cost in a Tiered LSM-Tree



T runs per level

Lookup cost in level i, $\mathcal{T}(i)$

- empty (α_i)

  $\alpha_i \cdot T \cdot (\, T_H + T_P + f_p \cdot T_D)$

- non-empty $(1 - \alpha_i)$

  $(1 - \alpha_i) \cdot \dfrac{T+1}{2} \cdot (\, T_H + T_P) + (1 - \alpha_i) \cdot T_D$

$$cost \approx \overset{\propto T \cdot L}{\boxed{\left(T \cdot L - \frac{T+1}{2}(1-\alpha)\right) \cdot (T_H + T_P)}}$$

Bloom filter cost

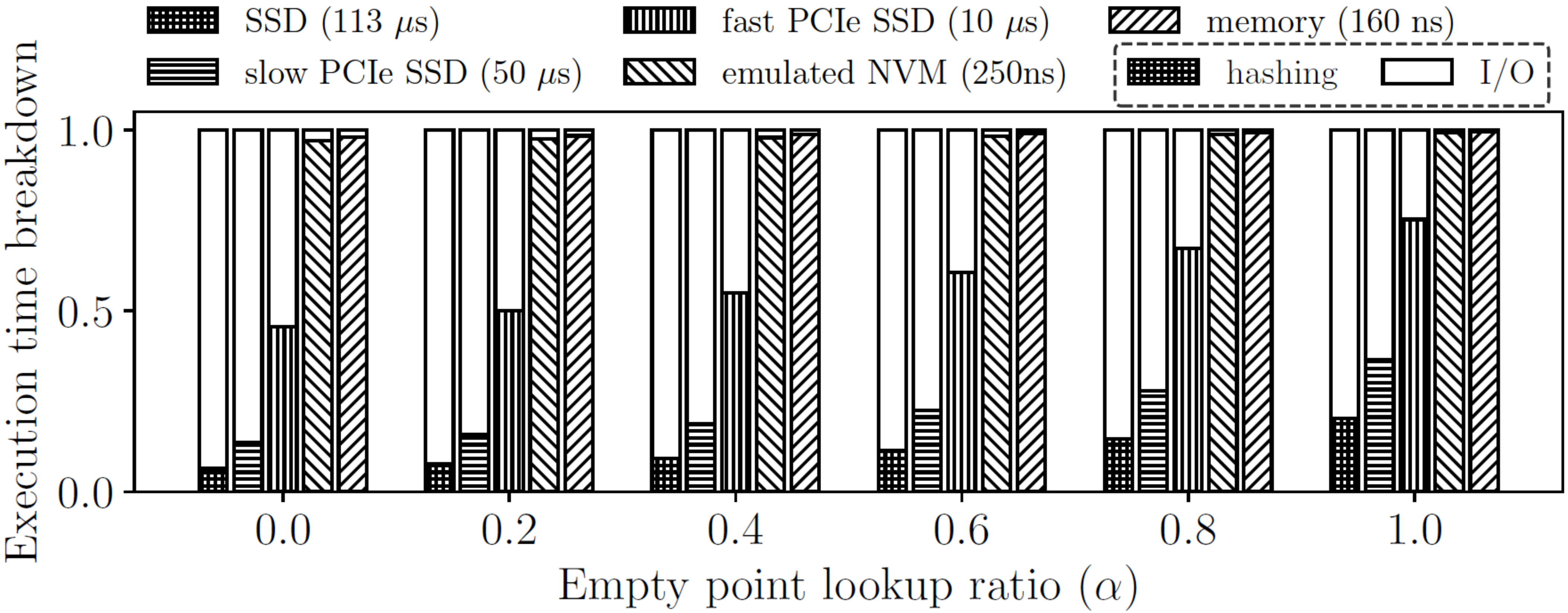
$$+ \left(T \cdot L - (1-\alpha) \cdot (T+1)\right) \cdot \left(f_p \cdot T_D\right)$$

Data access due to false positives

$$+ (1-\alpha) \cdot T_D$$

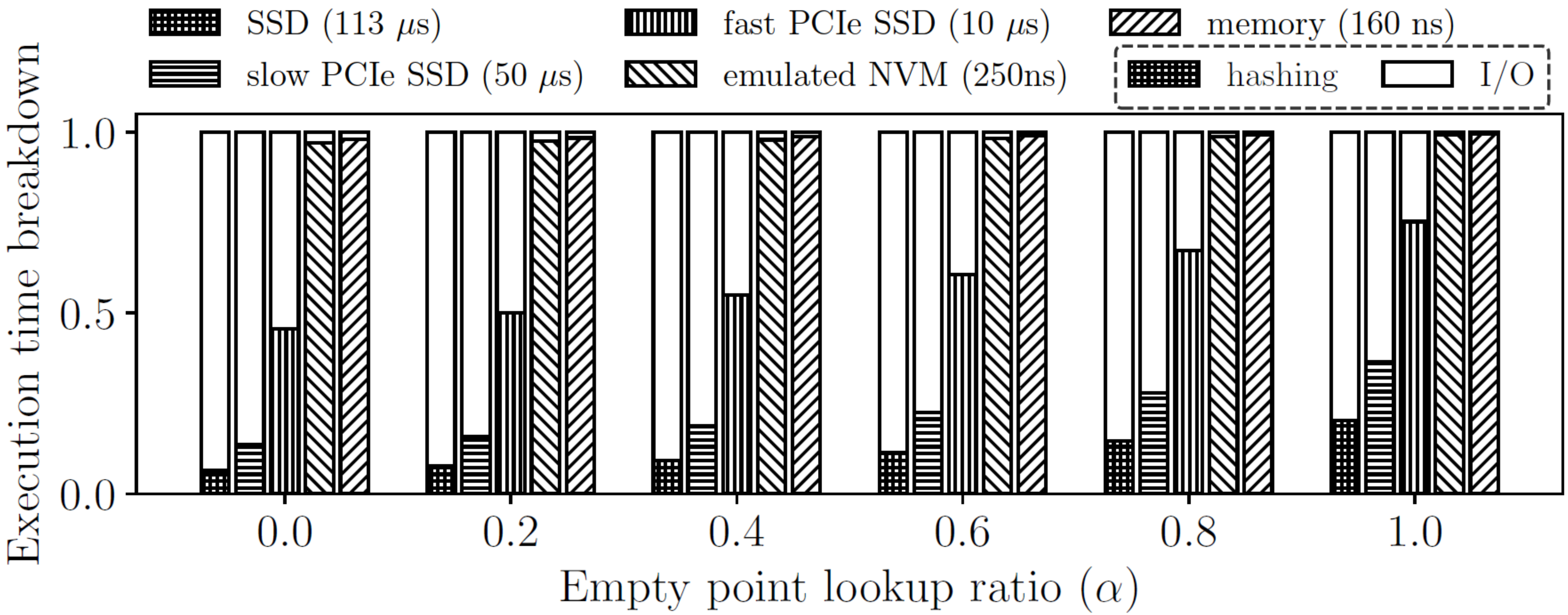
Data access

Hashing Overhead in a Tiered LSM-Tree



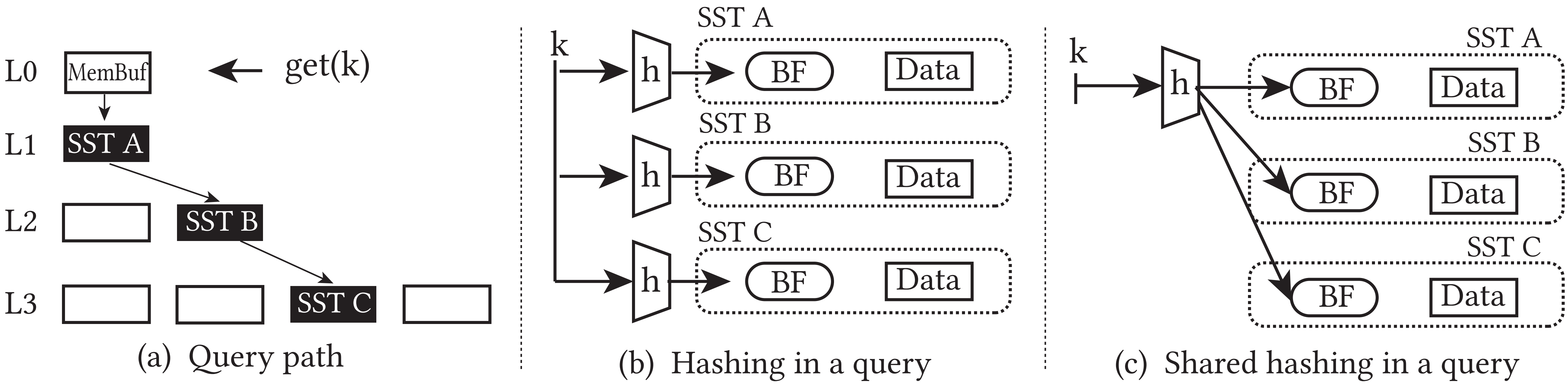
Similar, but hashing is more pronounced.

Hashing Overhead in a Tiered LSM-Tree



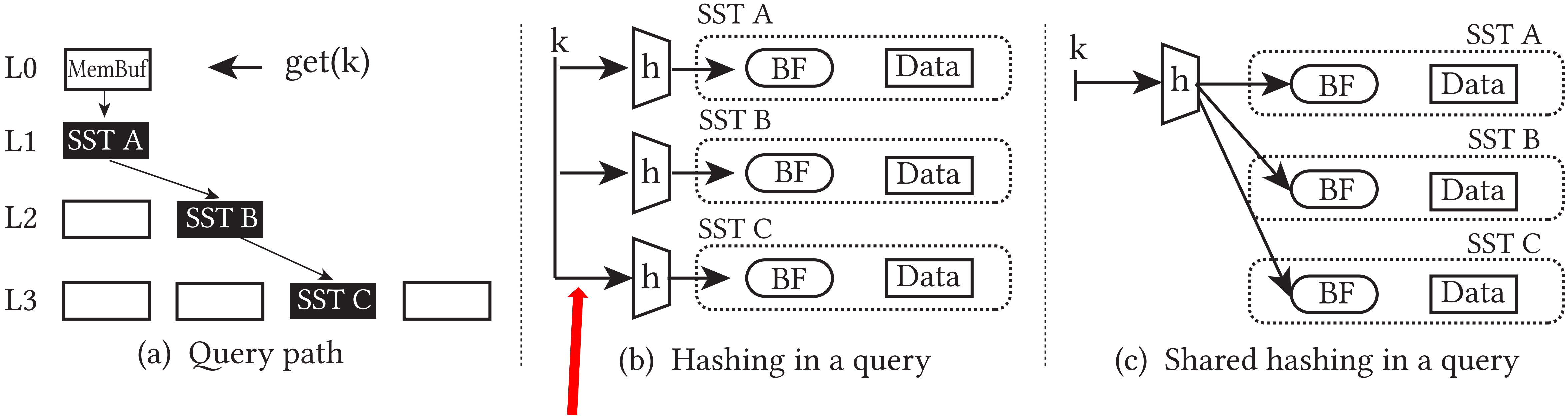
How can we reduce the hashing overhead in LSM-trees?

Hash Sharing in Leveled LSM-Trees



(a) Query path (b) Hashing in a query (c) Shared hashing in a query

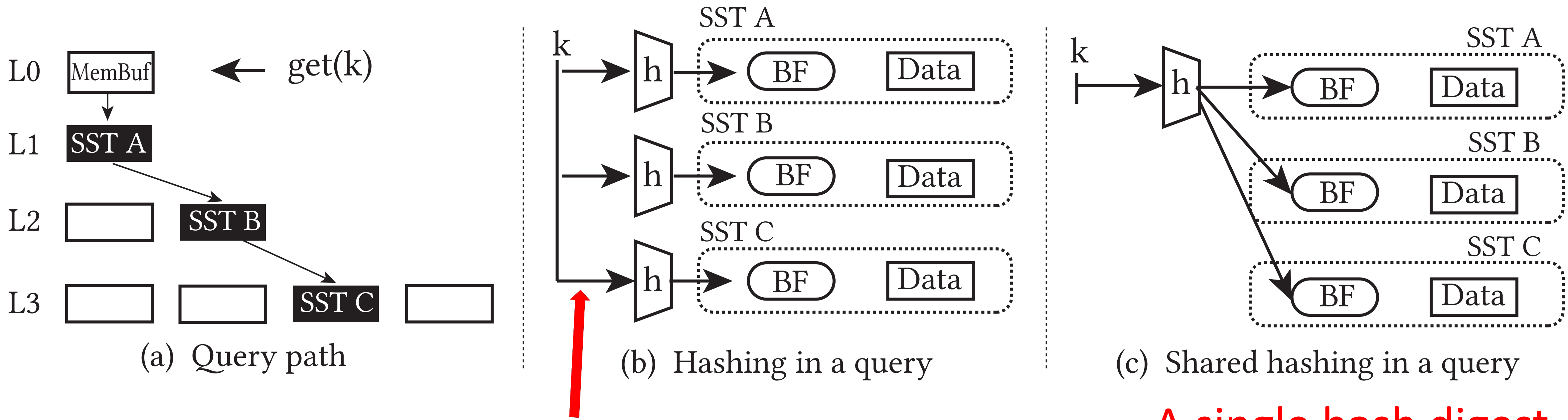
Hash Sharing in Leveled LSM-Trees



(a) Query path (b) Hashing in a query (c) Shared hashing in a query

The same hash function is calculated multiple times which brings CPU overhead.

Hash Sharing in Leveled LSM-Trees



(a) Query path

(b) Hashing in a query

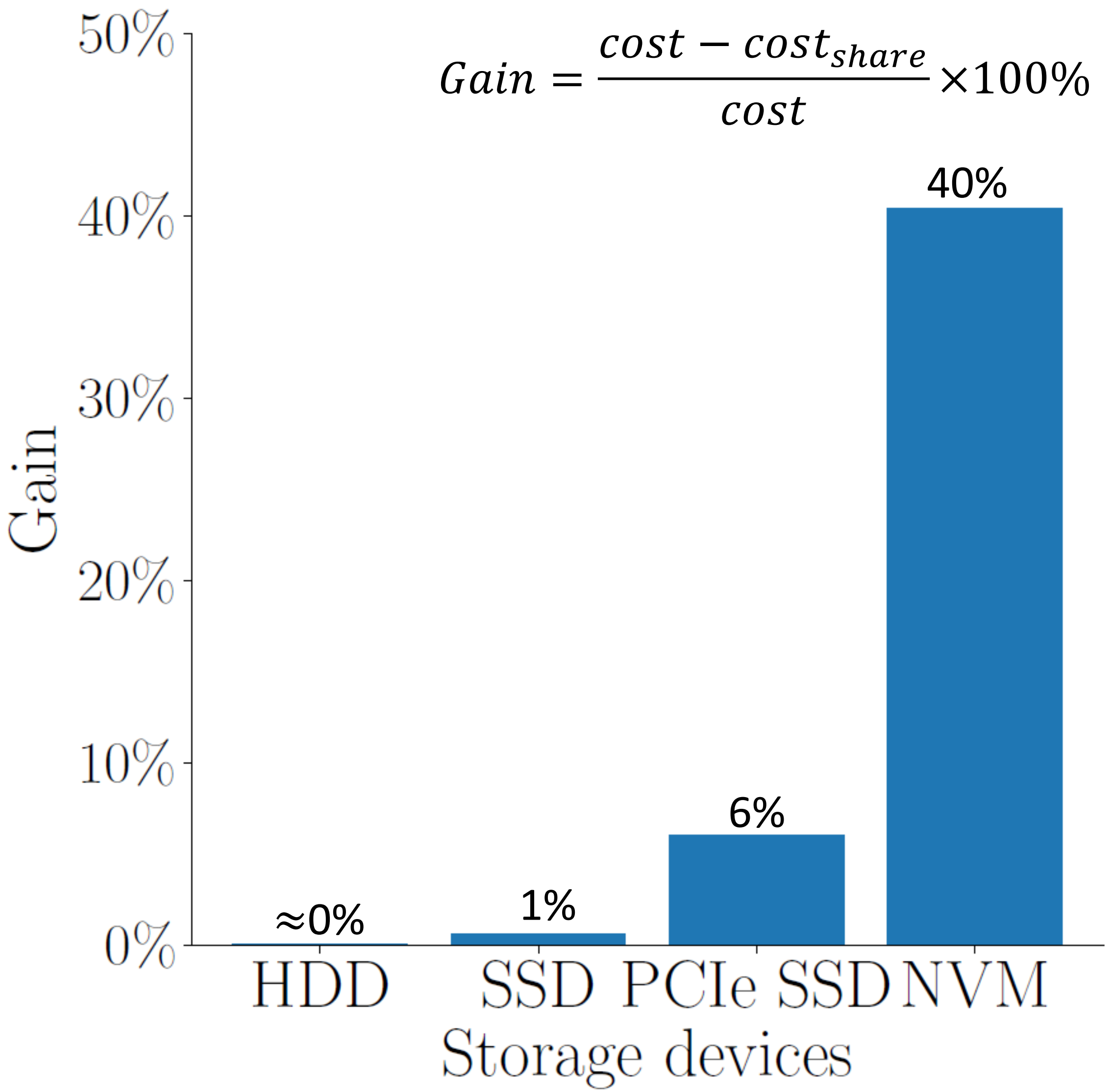
(c) Shared hashing in a query

The same hash function is calculated multiple times which brings CPU overhead.

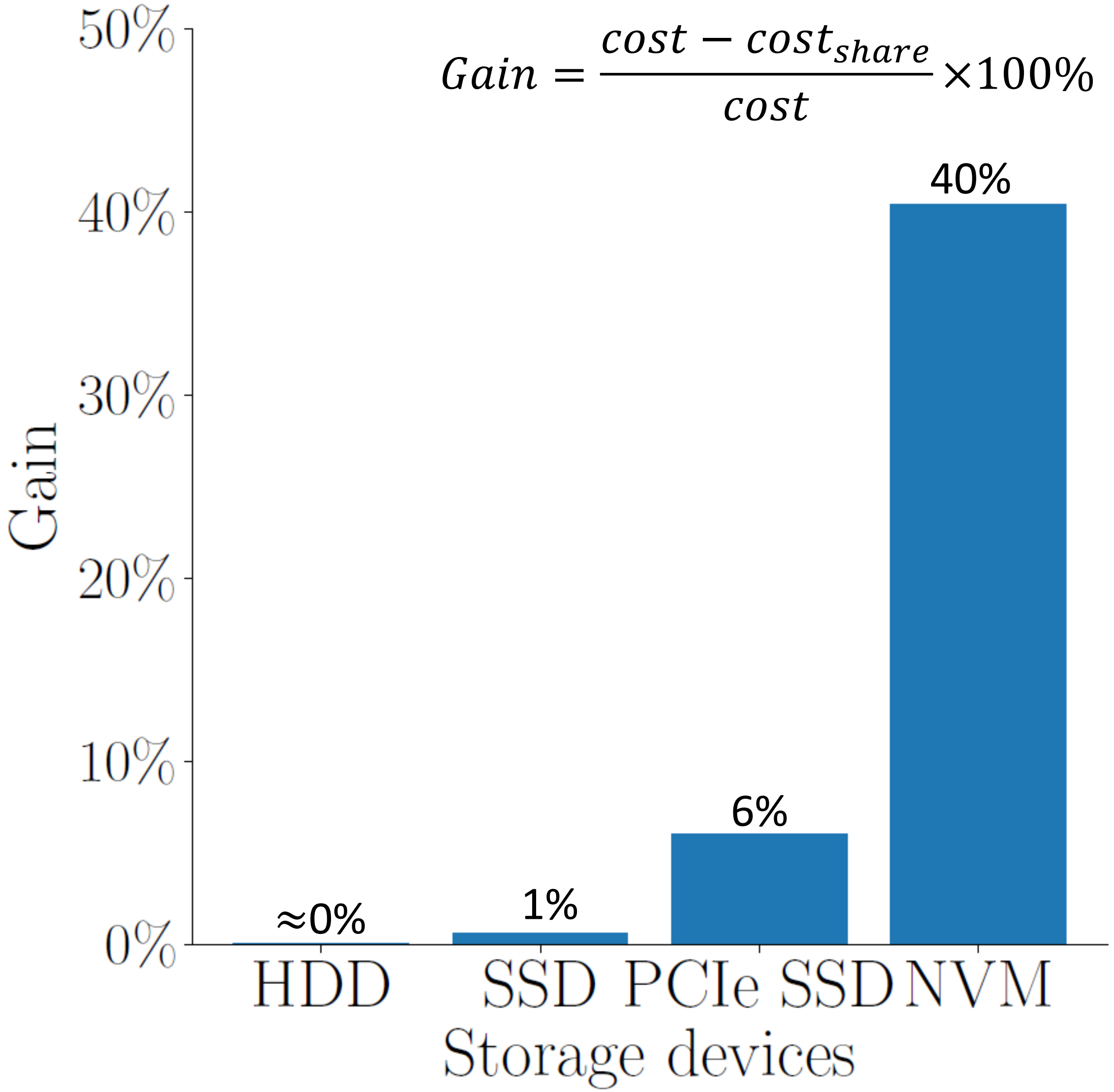
A single hash digest can be reused to avoid expensive hash calculations.

Theoretical Gain w.r.t. Evolving Storage Devices

$$Gain = \frac{cost - cost_{share}}{cost} \times 100\%$$



Theoretical Gain w.r.t. Evolving Storage Devices



$$Gain = \frac{cost - cost_{share}}{cost} \times 100\%$$

The gain increases rapidly for faster storage devices

Experiment Setting

Build an LSM prototype (RocksDB's fast local BF).

1M point queries (report avg latency of 5 experiments)

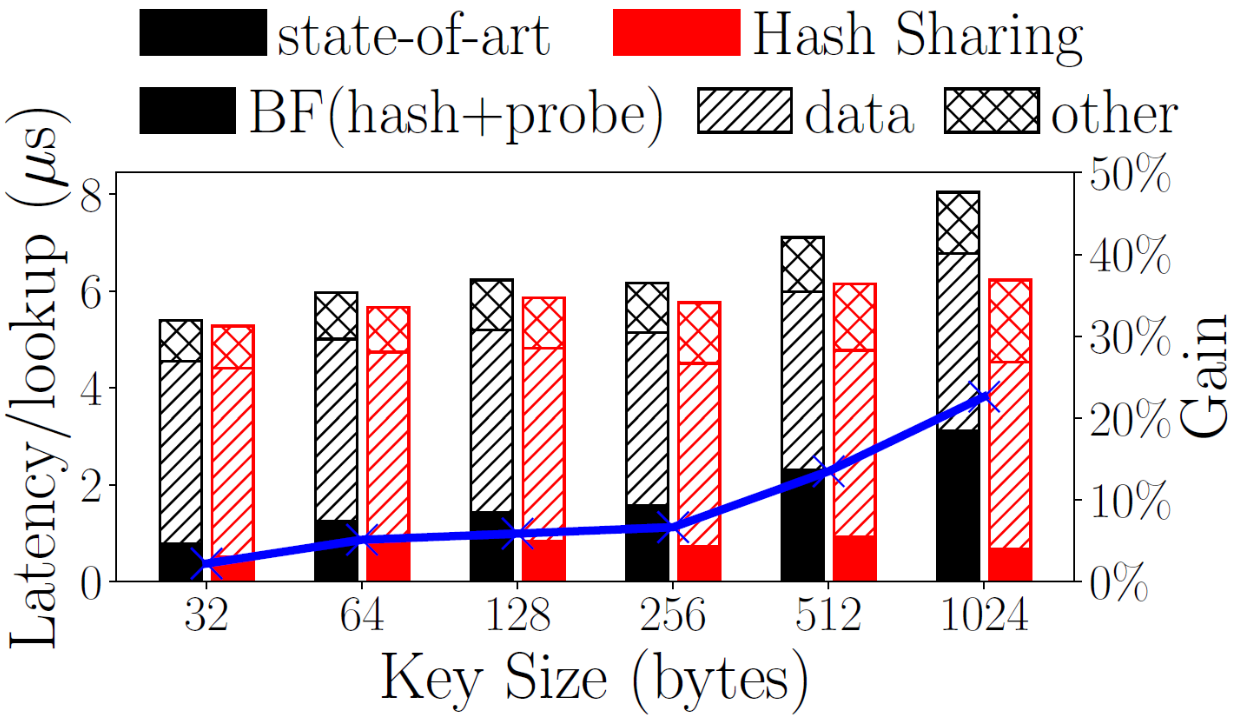
Use PCIe SSD (10 μs) with direct I/O by default

	Parameters	Default Value
Workload	Entry size (key + value)	1KB-2KB
	Data volume	22GB
	Empty query ratio (α)	100%
	Query distribution	Uniform
LSM	File size	2 MB
	Size ratio	10
	Bits per key	10

Impact of the key size

Workload: Entry size: 2KB, #Entries: 11M
Tuning: Bits per key: 10, Size ratio: 10, Storage: PCIe SSD

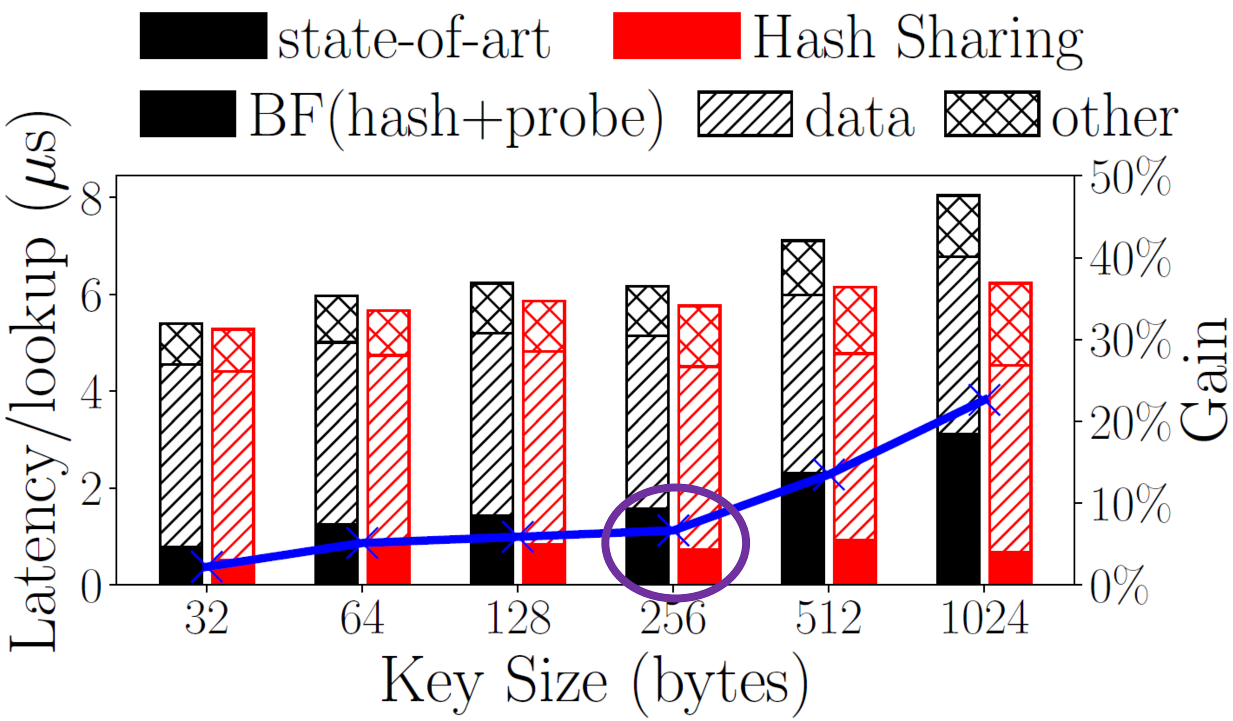
Uniform Query Distribution



Impact of the key size

Workload: Entry size: 2KB, #Entries: 11M
Tuning: Bits per key: 10, Size ratio: 10, Storage: PCIe SSD

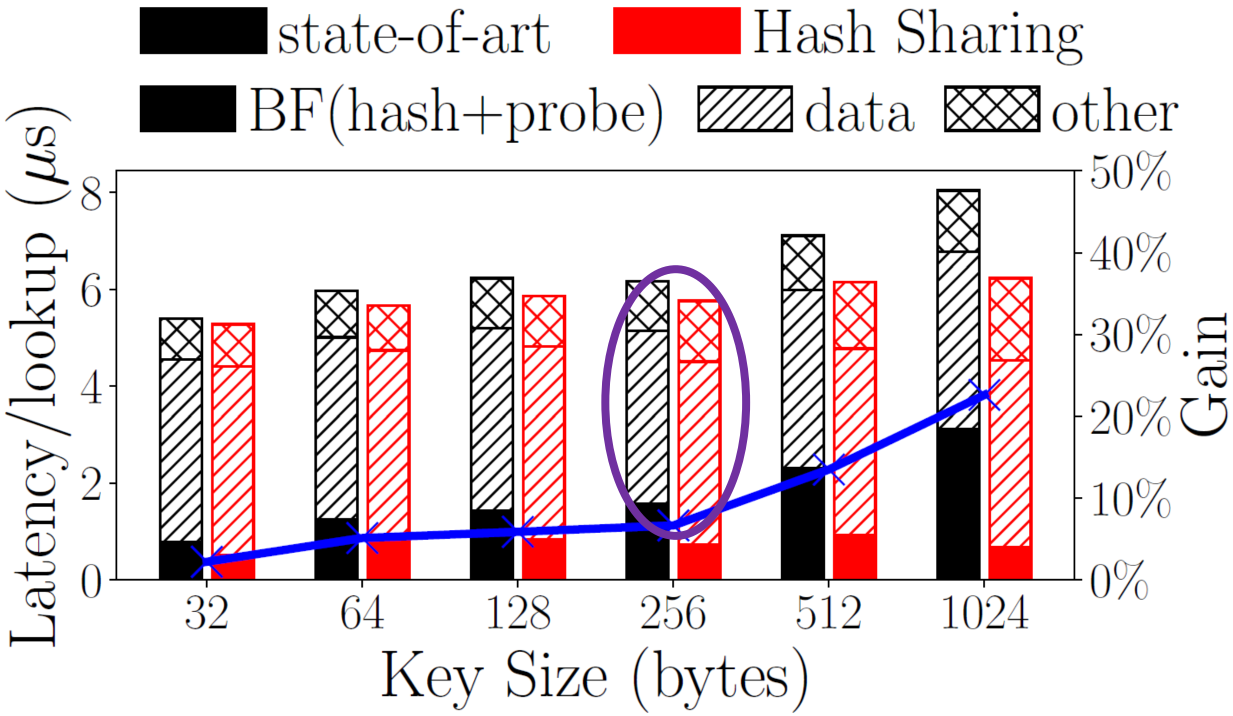
Uniform Query Distribution



Impact of the key size

Workload: Entry size: 2KB, #Entries: 11M
Tuning: Bits per key: 10, Size ratio: 10, Storage: PCIe SSD

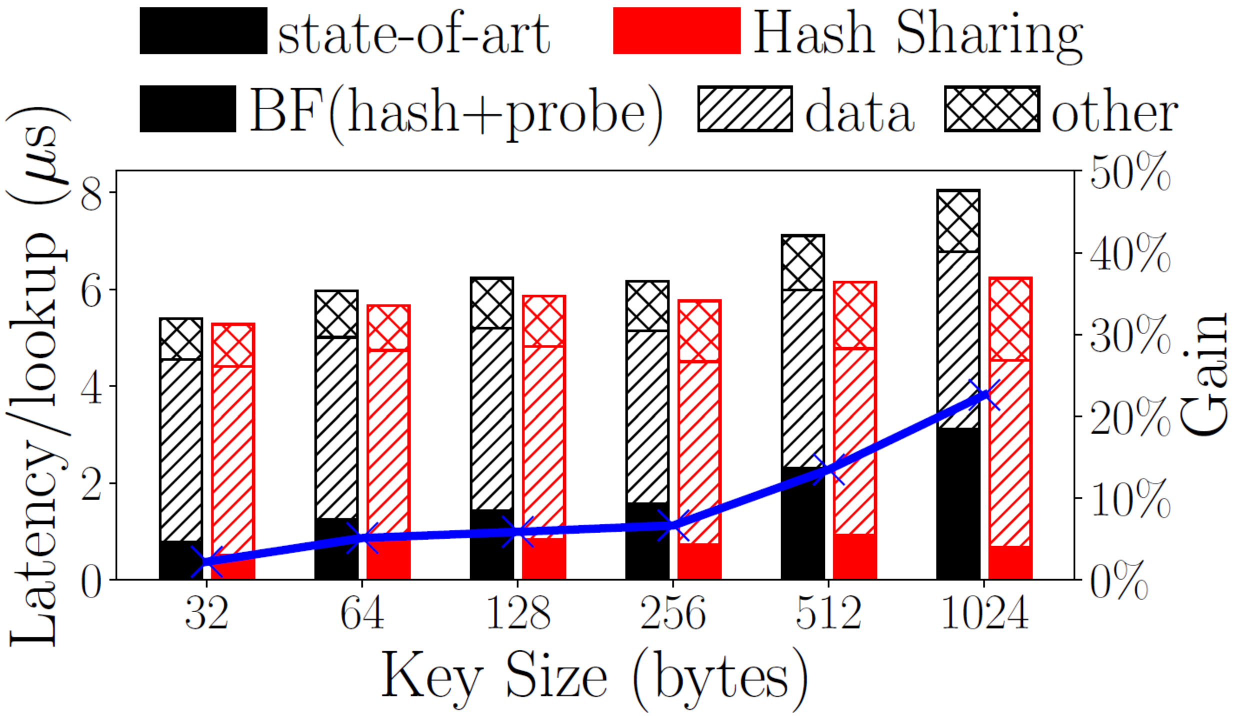
Uniform Query Distribution



Impact of the key size

Workload: Entry size: 2KB, #Entries: 11M
Tuning: Bits per key: 10, Size ratio: 10, Storage: PCIe SSD

Uniform Query Distribution

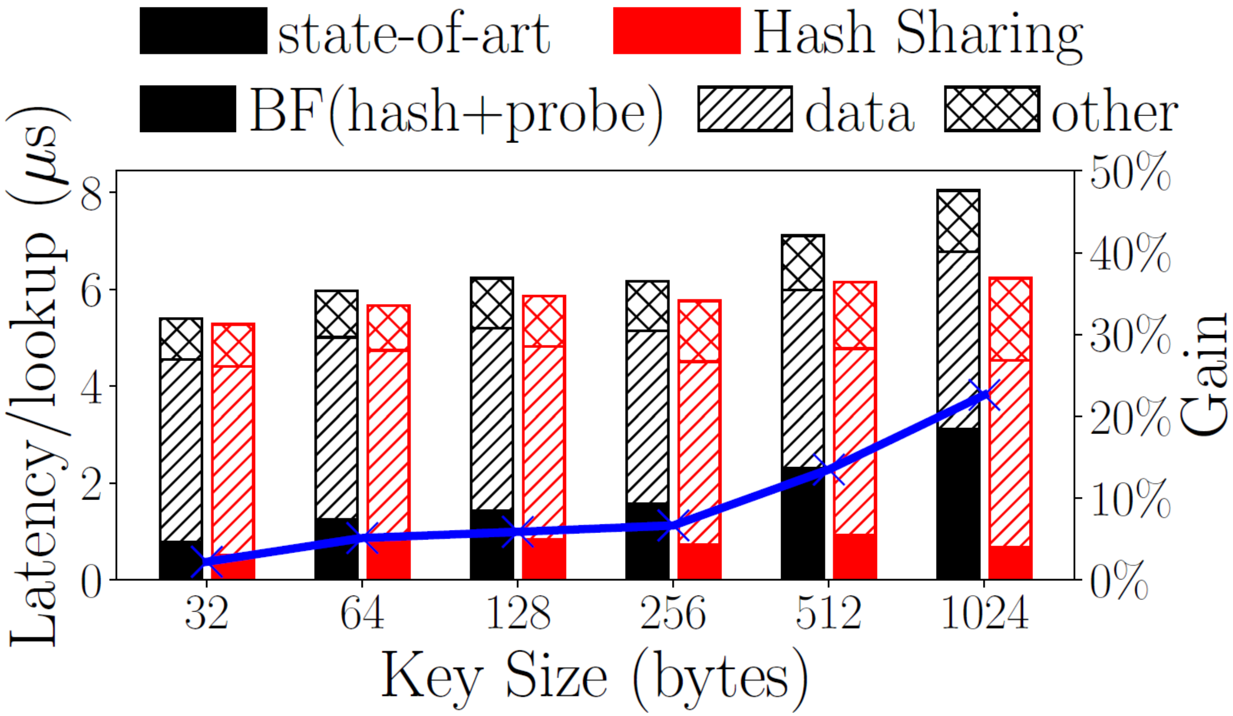


As key size grows, the gain increases up to 20%

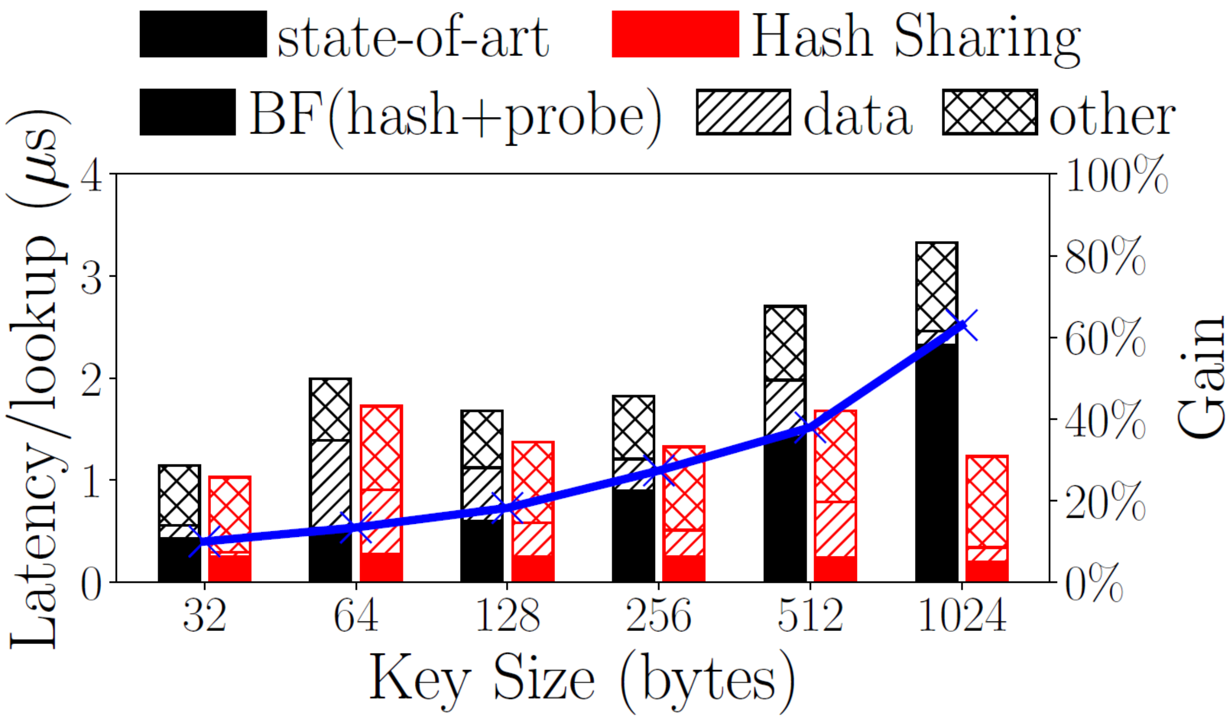
Impact of the key size

Workload: Entry size: 2KB, #Entries: 11M
Tuning: Bits per key: 10, Size ratio: 10, Storage: PCIe SSD

Uniform Query Distribution



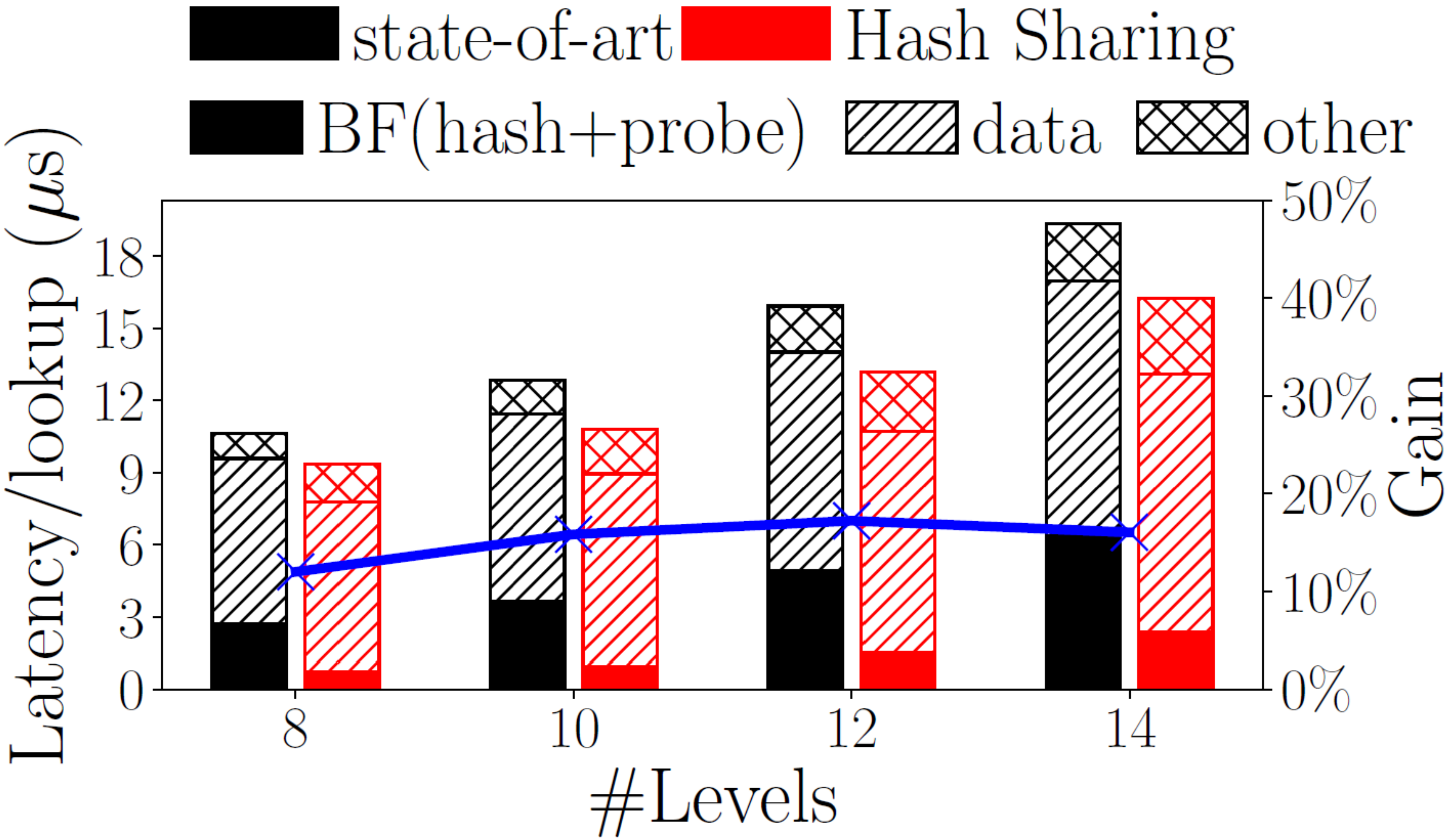
Zipfian Query Distribution



As key size grows, the gain increases up to 20%

For skewed empty query workload, the gain increases up to 60%

Impact of #levels



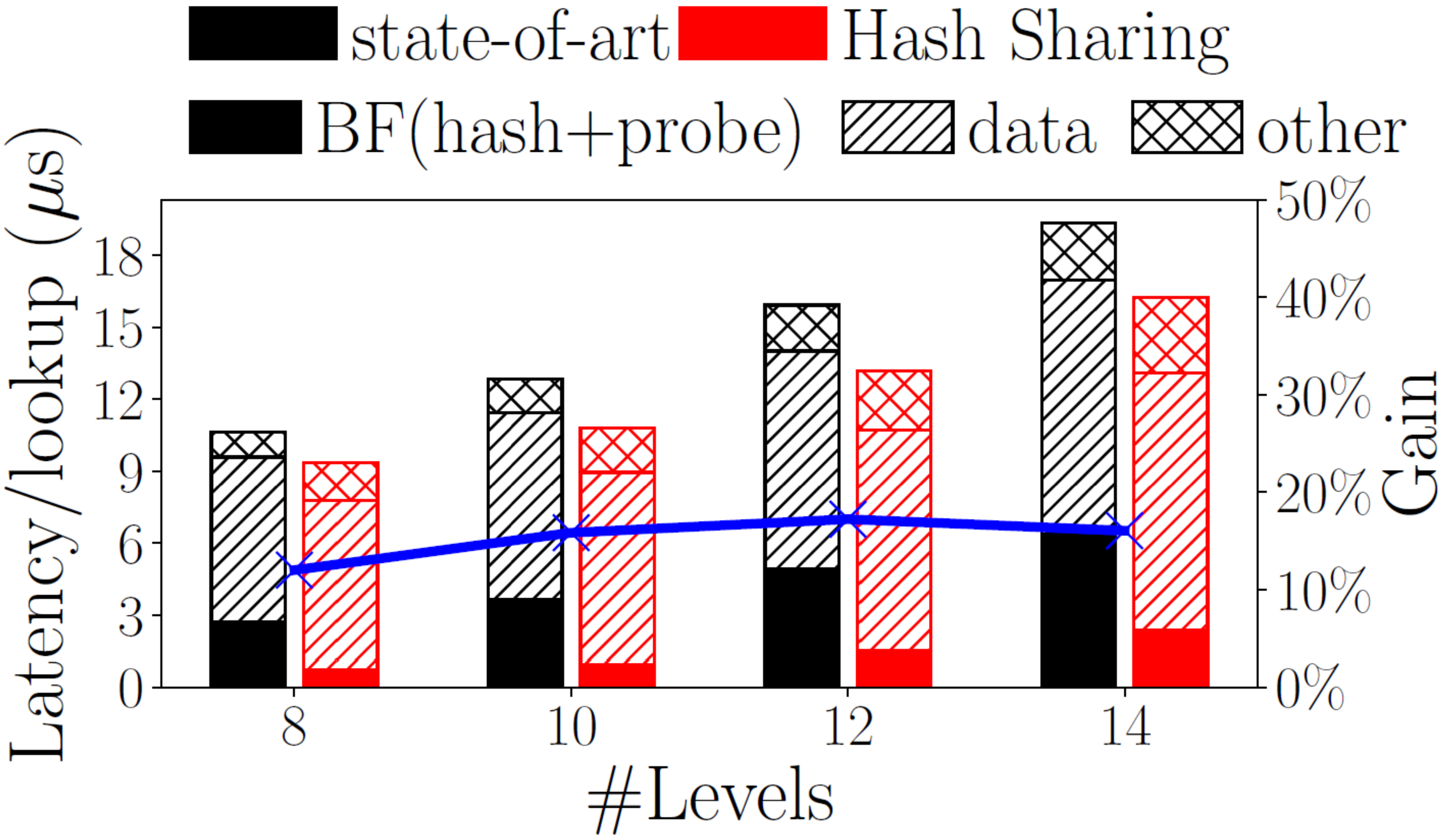
Workload:
- Key size: 64B
- Entry size: 1KB
- #Entries: 22M

Tuning:
- Bits per key: 10
- Size ratio: 2

Storage: PCIe SSD

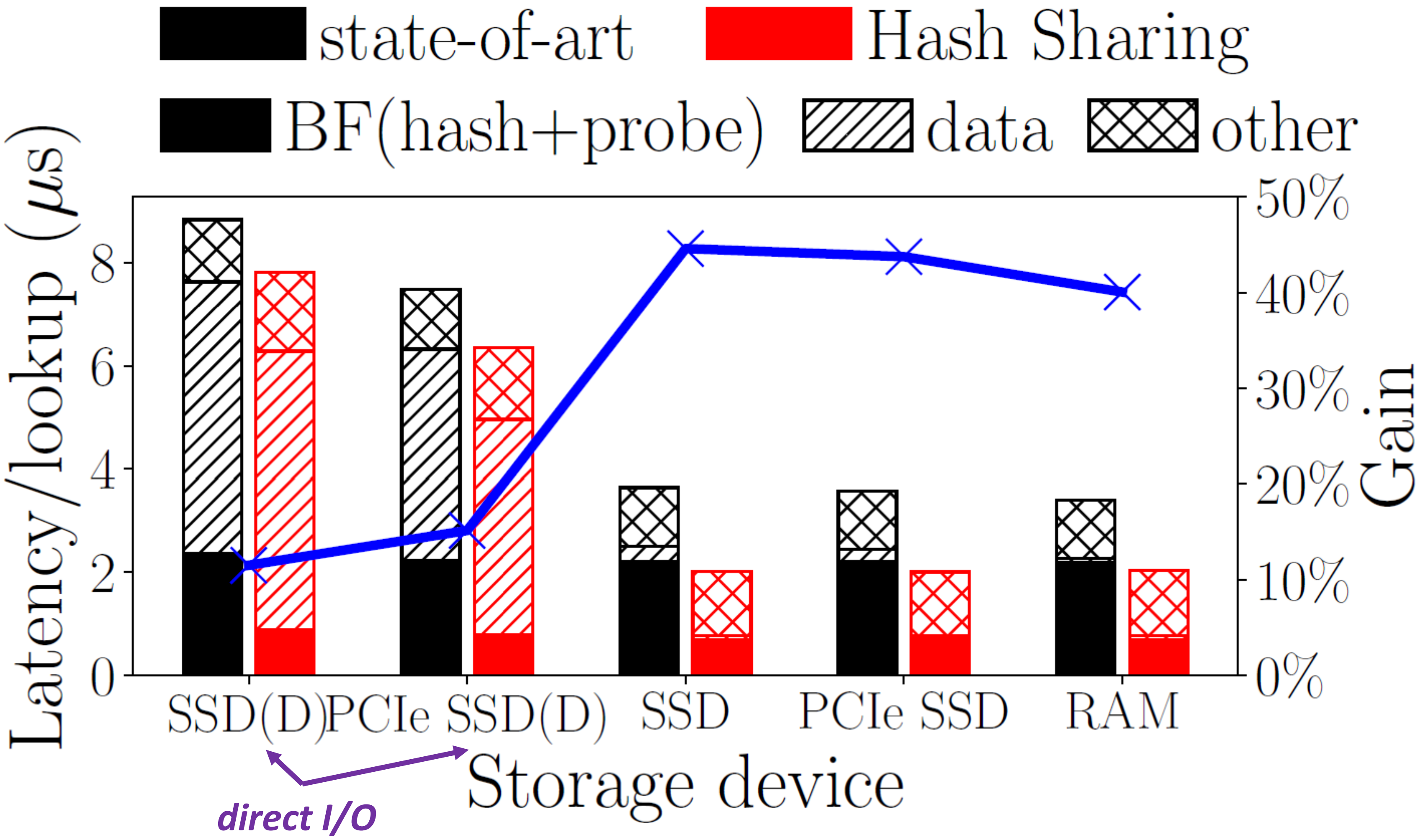
Impact of #levels

Workload:
Key size: 64B
Entry size: 1KB
#Entries: 22M
Tuning:
Bits per key: 10
Size ratio: 2
Storage: PCIe SSD



The gain initially increases as #level grows, and then plateaus

Impact of storage device



Workload:

Key size: 64B

Entry size: 1KB

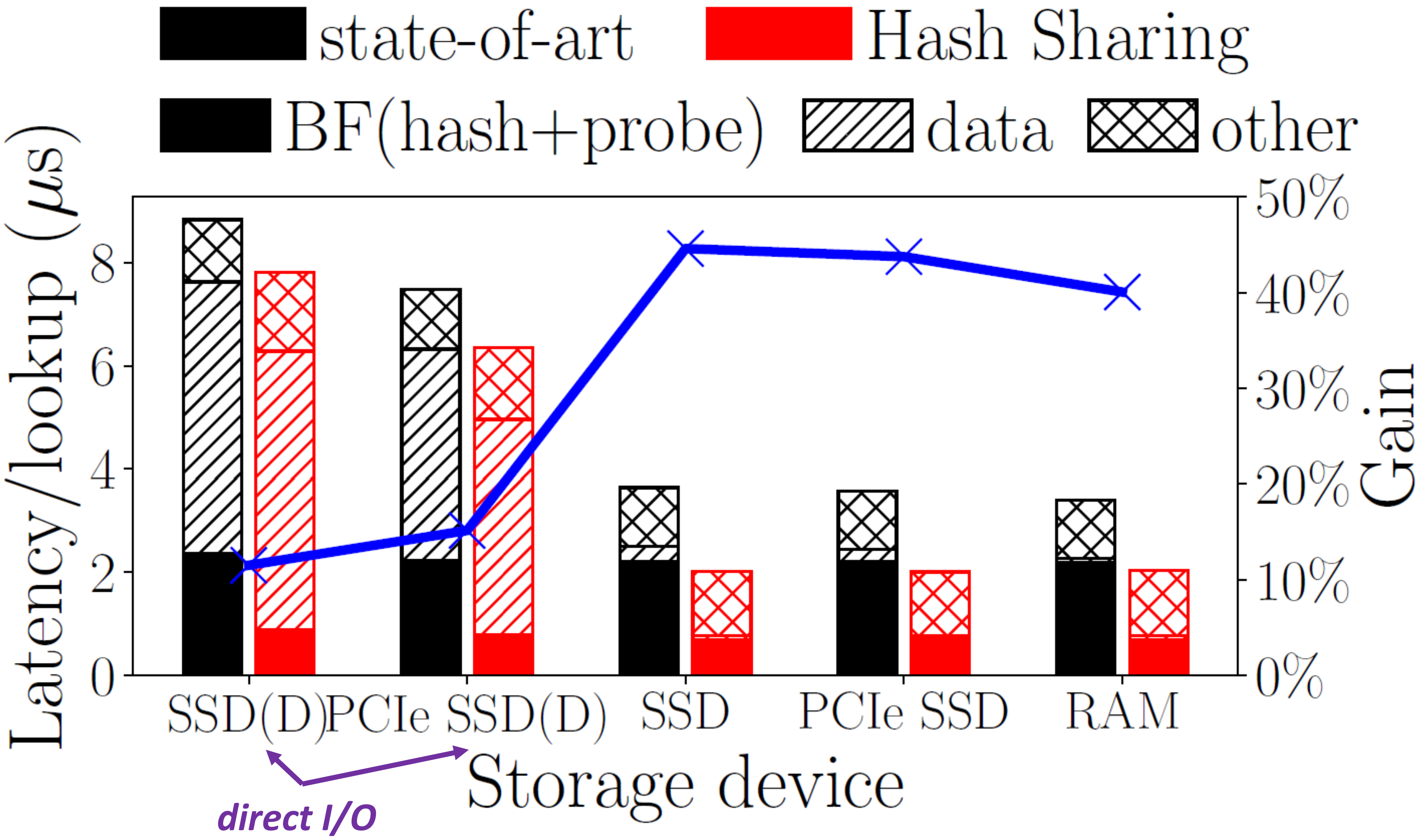
#Entries: 22M

Tuning:

Bits per key: 10

Size ratio: 10

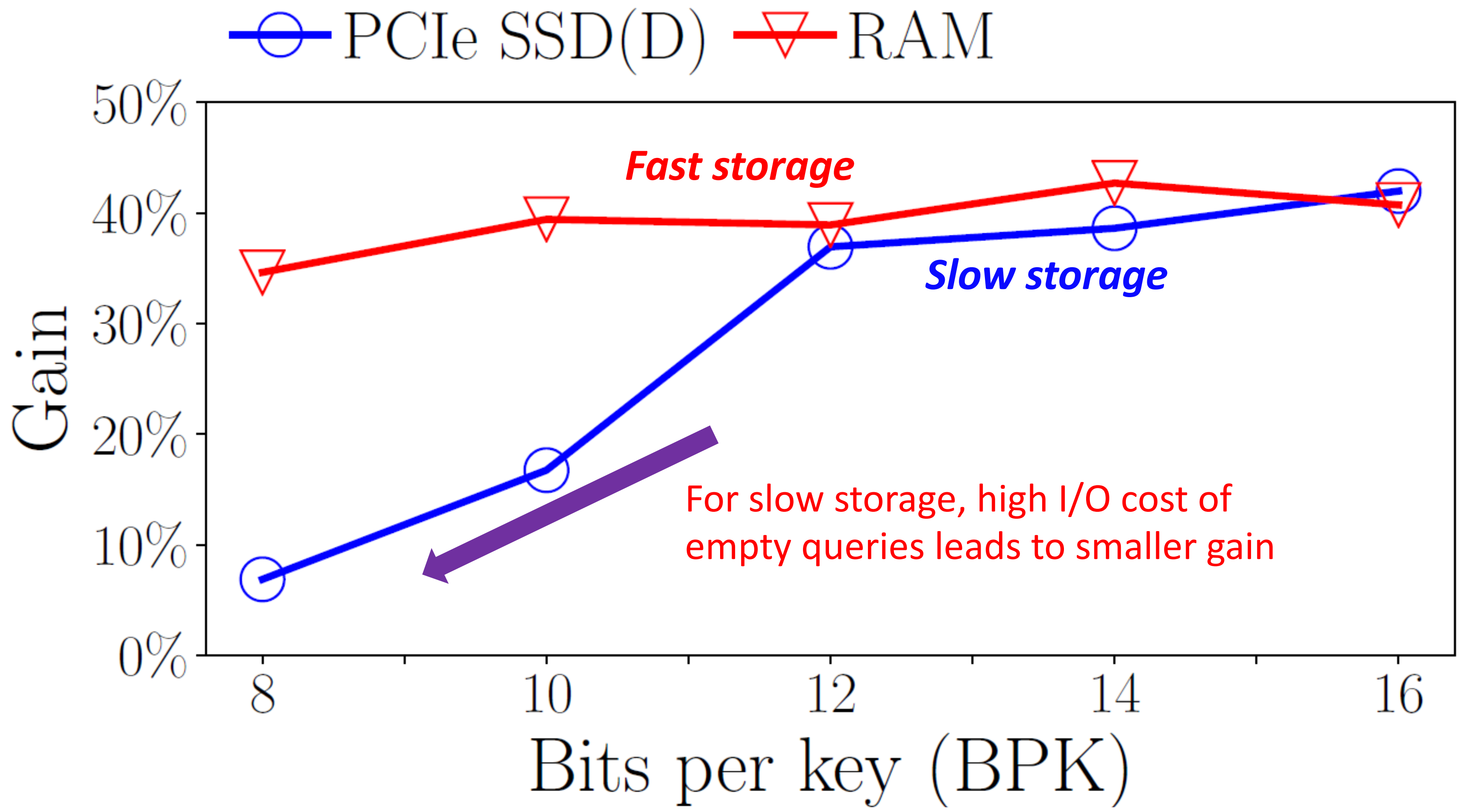
Impact of storage device



Workload:
Key size: 64B
Entry size: 1KB
#Entries: 22M
Tuning:
Bits per key: 10
Size ratio: 10

Hash sharing leads to higher gain for faster storage.

Impact of the I/O cost of empty queries



Fast storage

Slow storage

For slow storage, high I/O cost of empty queries leads to smaller gain

High *I/O cost of empty queries* Low

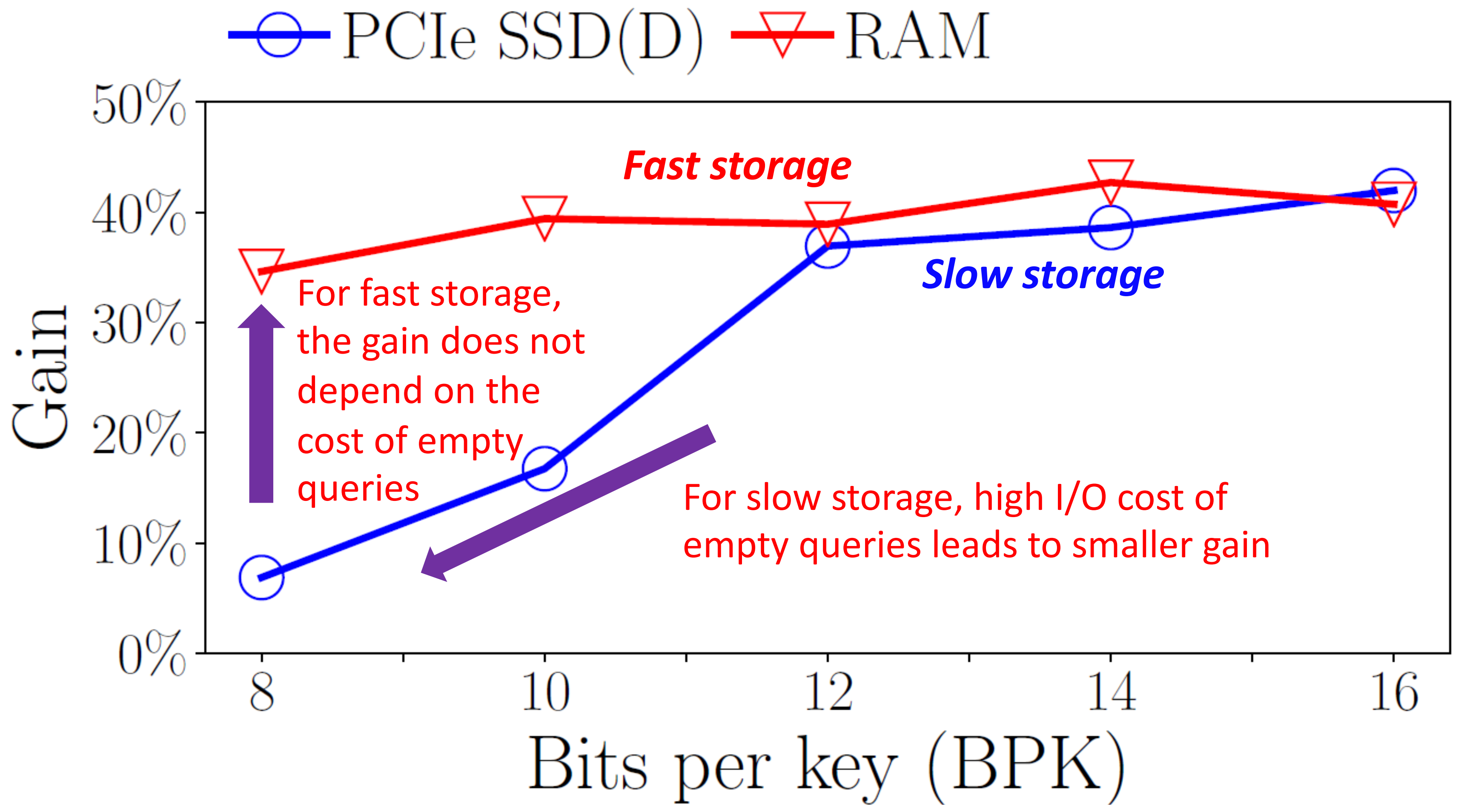
Workload:

- Entry size: 1KB
- #Entries: 22M
- Empty query ratio (α) : 1

Tuning:

- Bits per key: 10
- Size ratio: 10

Impact of the I/O cost of empty queries



Fast storage

Slow storage

For fast storage, the gain does not depend on the cost of empty queries

For slow storage, high I/O cost of empty queries leads to smaller gain

High *I/O cost of empty queries* Low

Workload:

Entry size: 1KB

#Entries: 22M

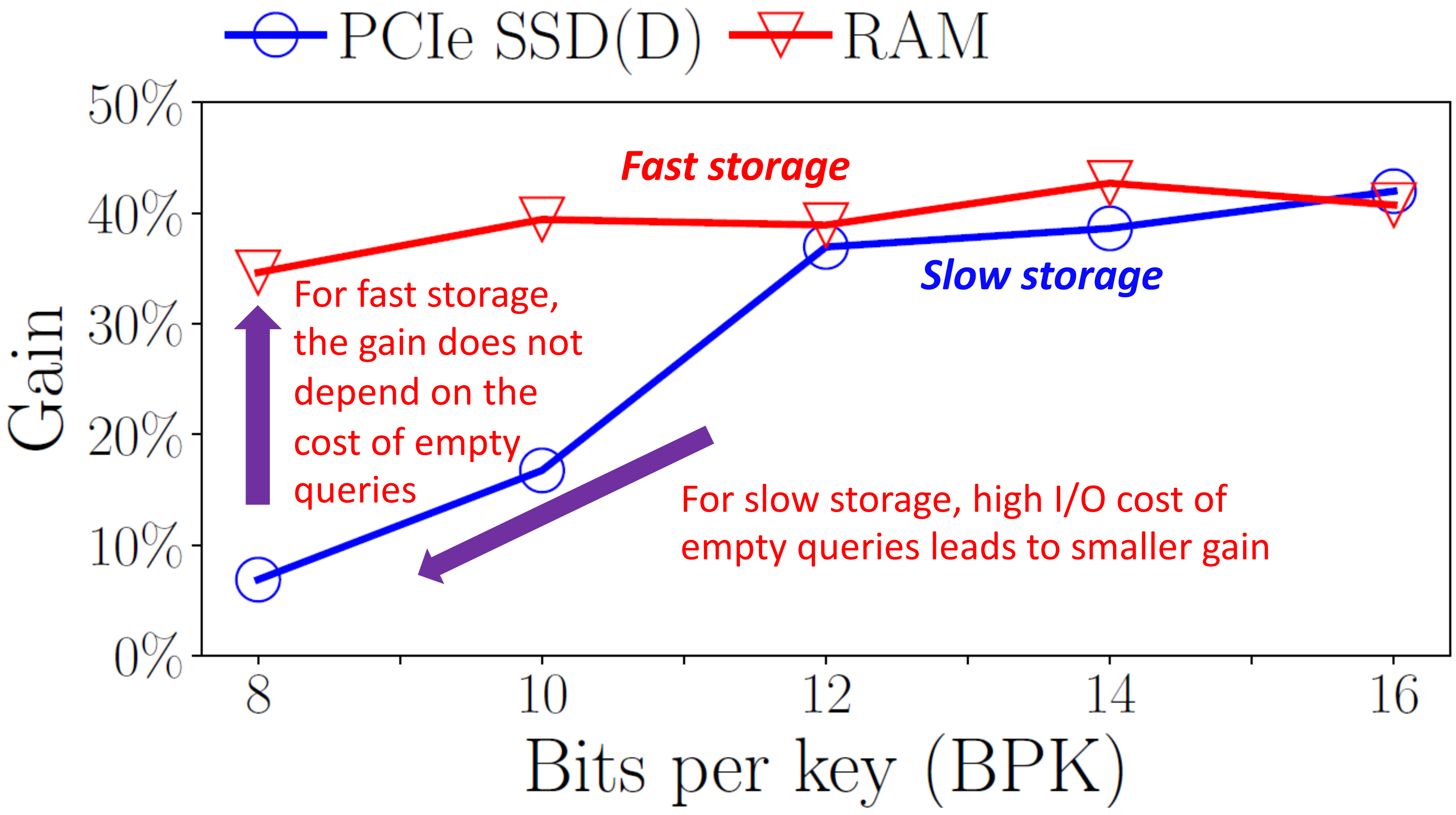
Empty query ratio (α) : 1

Tuning:

Bits per key: 10

Size ratio: 10

Impact of the I/O cost of empty queries



Fast storage

Slow storage

For fast storage, the gain does not depend on the cost of empty queries

For slow storage, high I/O cost of empty queries leads to smaller gain

High *I/O cost of empty queries* Low

Workload:
- Entry size: 1KB
- #Entries: 22M
- Empty query ratio (α) : 1

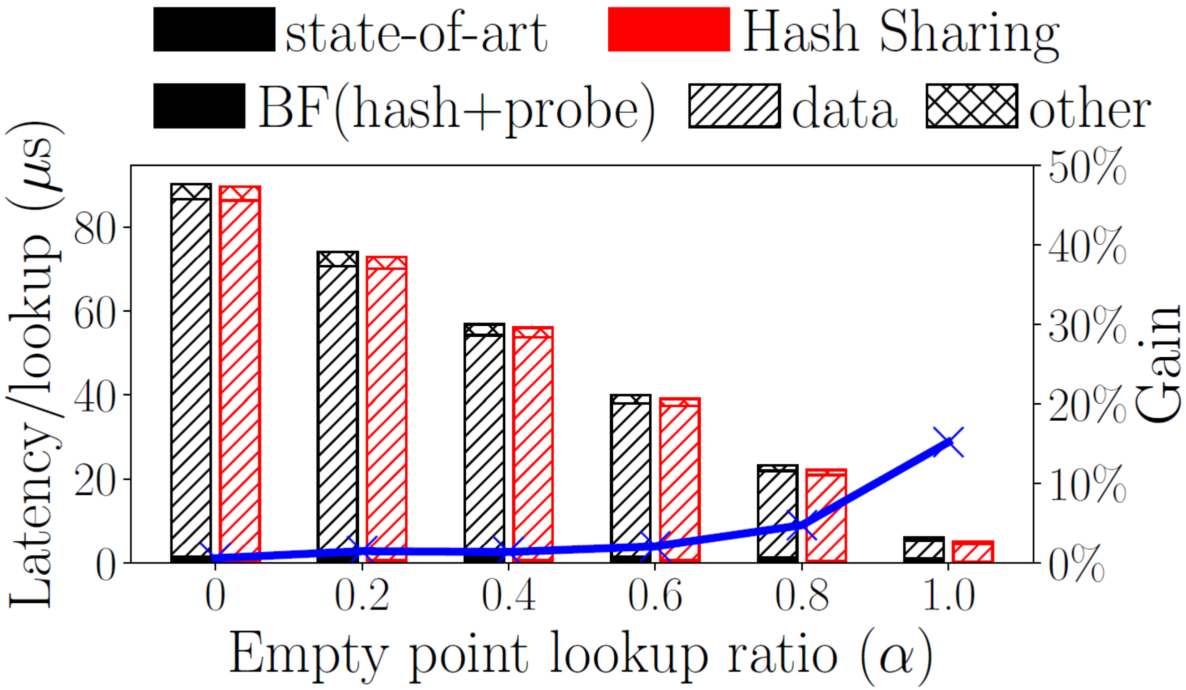
Tuning:
- Bits per key: 10
- Size ratio: 10

Fast storage leads to high gain. Even for slower storage, if the cost of empty queries is low (low FPR), the gain is high

Impact of empty lookup ratio ($\boldsymbol{\alpha}$)

Workload: Entry size: 1KB, #Entries: 22M
Tuning: Bits per key: 10, Size ratio: 10

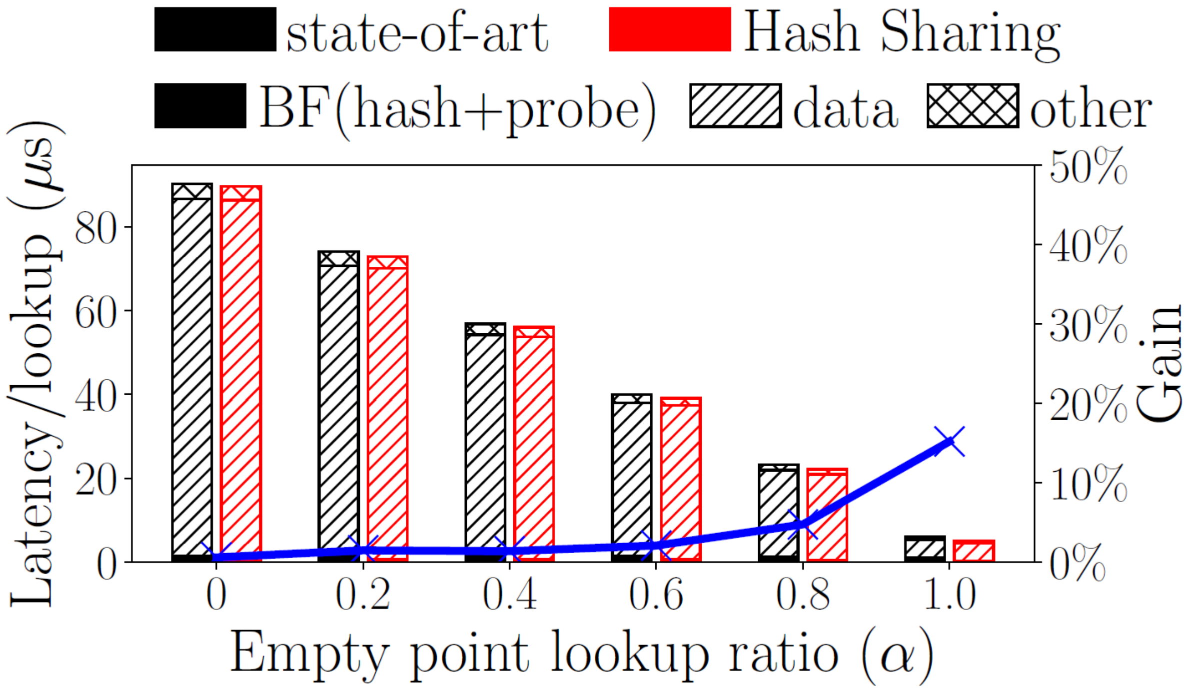
Storage: PCIe SSD (D)



Impact of empty lookup ratio (α)

Workload: Entry size: 1KB, #Entries: 22M
Tuning: Bits per key: 10, Size ratio: 10

Storage: PCIe SSD (D)

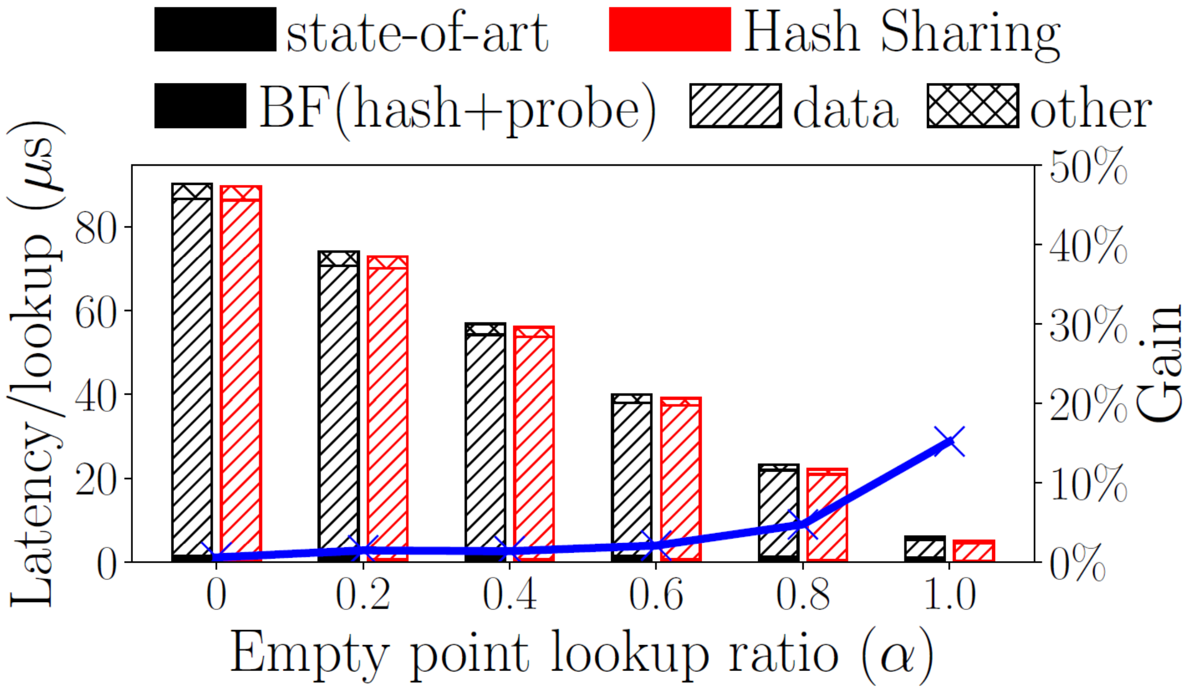


The gain on PCIe SSD increases as α increases

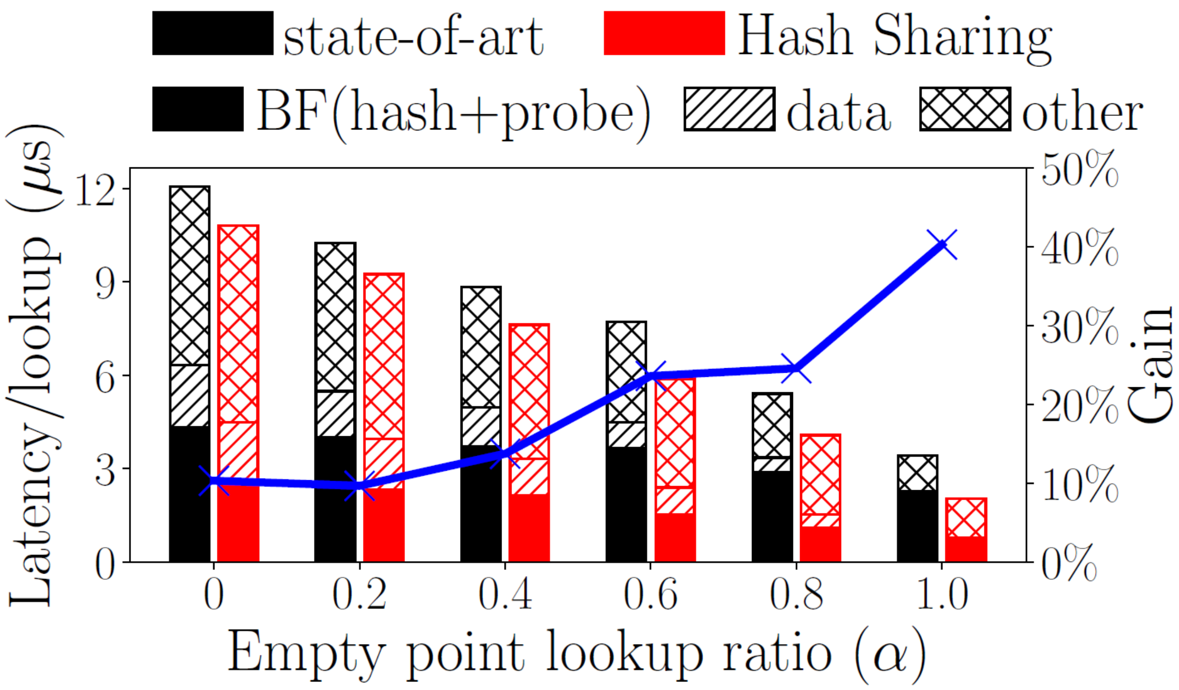
Impact of empty lookup ratio (α)

Workload: Entry size: 1KB, #Entries: 22M
Tuning: Bits per key: 10, Size ratio: 10

Storage: PCIe SSD (D)



Storage: RAM disk



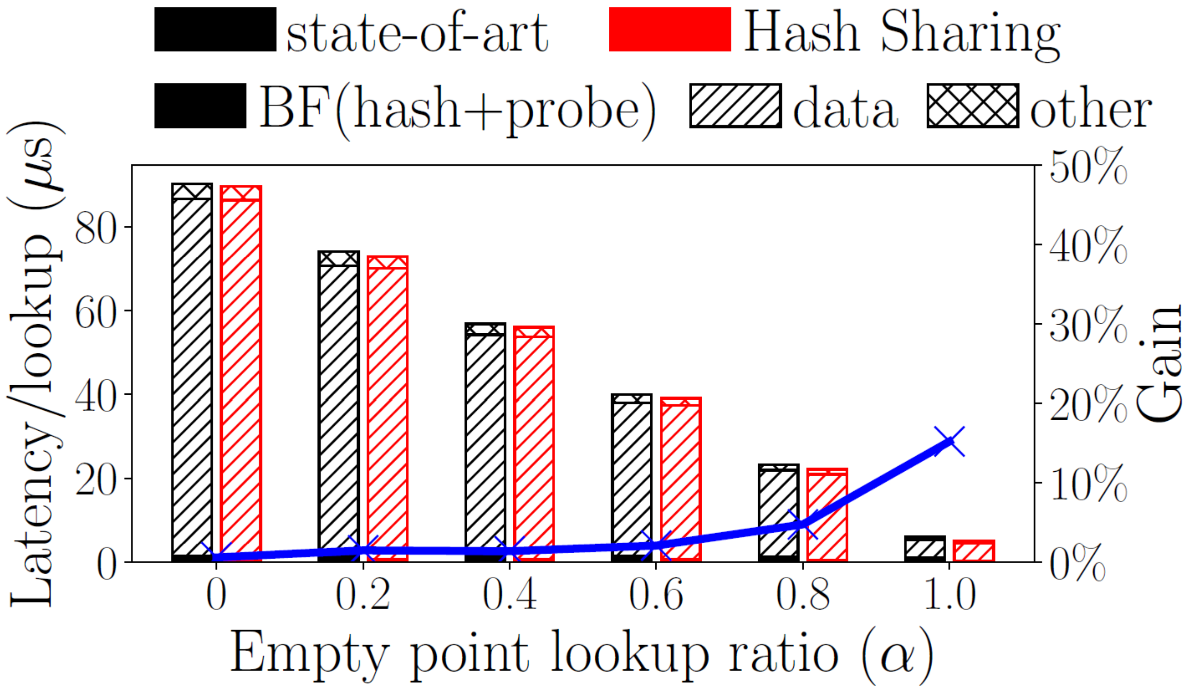
The gain on PCIe SSD increases as α increases

The benefit is pronounced when it comes to a RAM disk

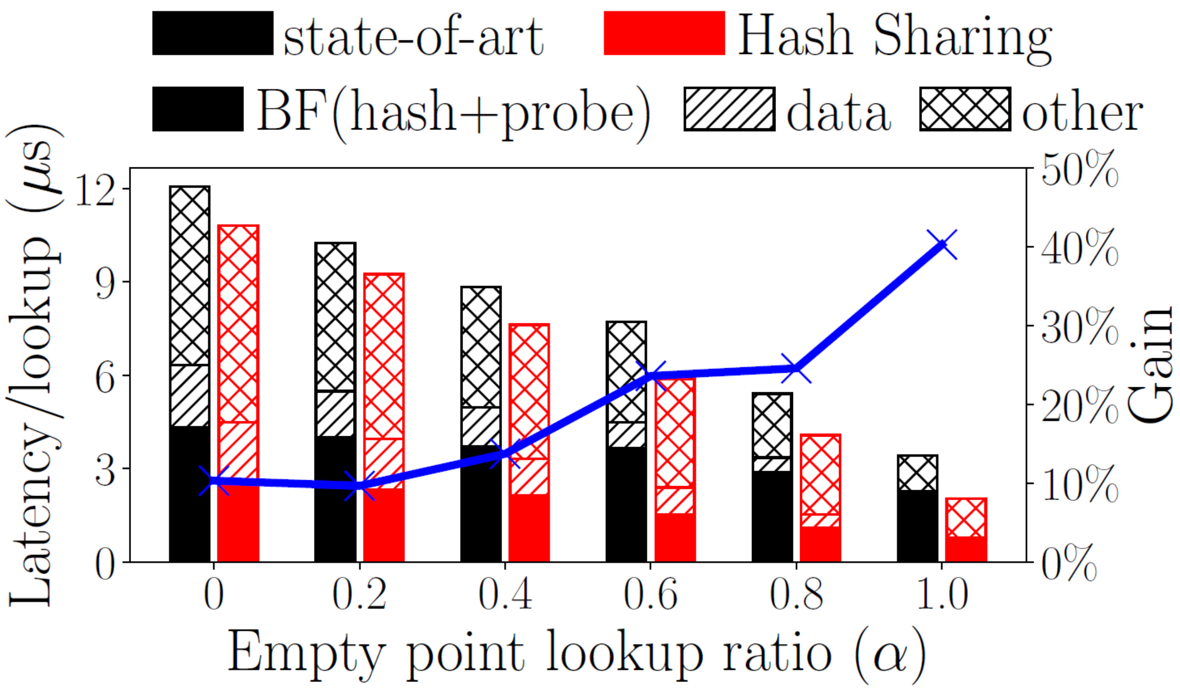
Impact of empty lookup ratio (α)

Workload: Entry size: 1KB, #Entries: 22M
Tuning: Bits per key: 10, Size ratio: 10

Storage: PCIe SSD (D)



Storage: RAM disk



The gain on PCIe SSD increases as α increases

The benefit is pronounced when it comes to a RAM disk

Overall, hash sharing has more impact for faster devices which is further exacerbated for empty queries.

Conclusion

- ☐ BFs dominate LSM query latency for fast storage

Conclusion

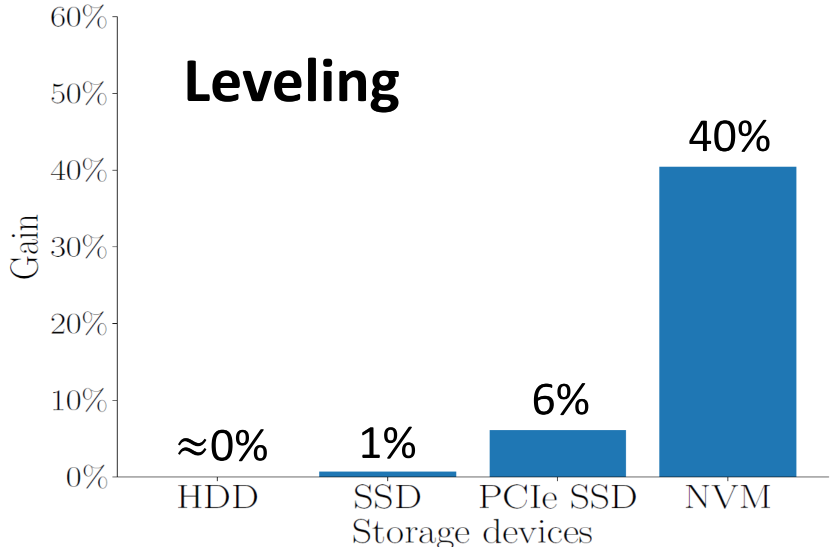
- ❑ BFs dominate LSM query latency for fast storage
- ❑ Develop a query cost model to quantify and predict the amount of time on hashing and data accessing

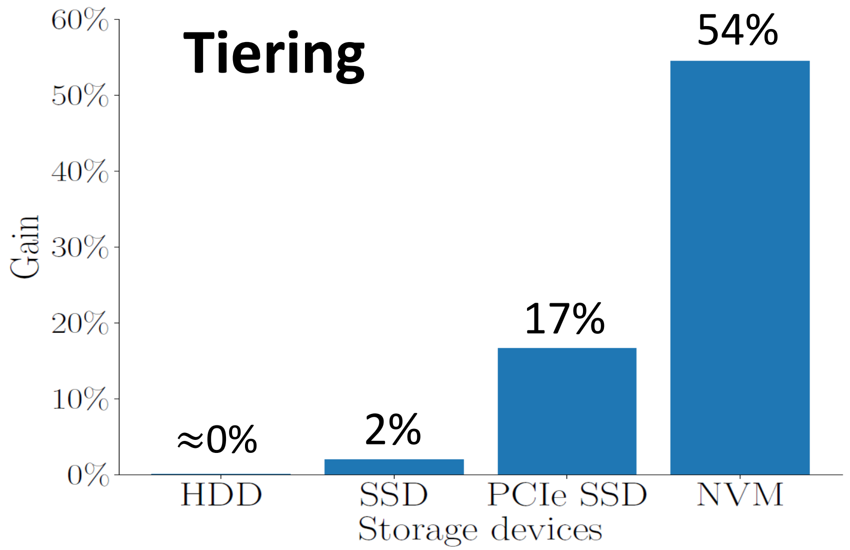
Conclusion

❑ BFs dominate LSM query latency for fast storage

❑ Develop a query cost model to quantify and predict the amount of time on hashing and data accessing

❑ Reduce hashing, by sharing it across BFs and levels, leading to performance gains up to 40%



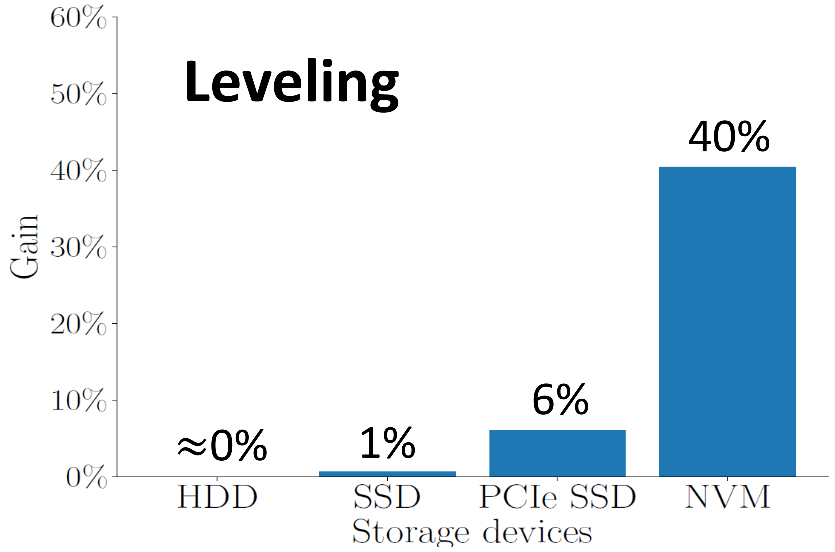


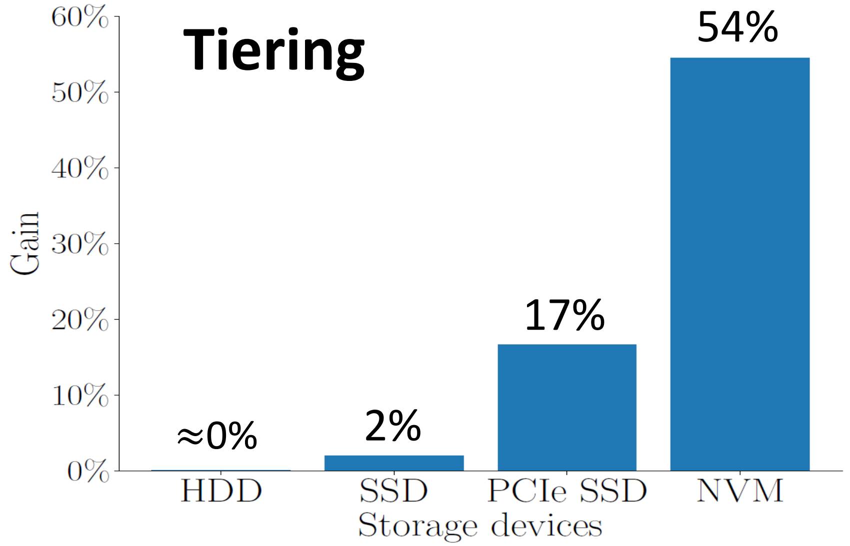
Conclusion

☐ BFs dominate LSM query latency for fast storage

☐ Develop a query cost model to quantify and predict the amount of time on hashing and data accessing

☐ Reduce hashing, by sharing it across BFs and levels, leading to performance gains up to 40%





Thank you!

https://github.com/BU-DiSC/BF-Shared-Hashing