LSM-Trees &
its **Read Optimizations**

Subhadeep Sarkar   *Niv Dayan*   *Manos Athanassoulis*

BOSTON UNIVERSITY

UNIVERSITY OF TORONTO

Log-**S**tructured **M**erge-tree

LSM-tree

LSM-tree

The Log-Structured Merge-Tree (LSM-Tree)

Patrick O'Neil[1], Edward Cheng[2]
Dieter Gawlick[3], Elizabeth O'Neil[1]
To be published: Acta Informatica

1996

LSM-tree

O'Neil *et al.*

1996

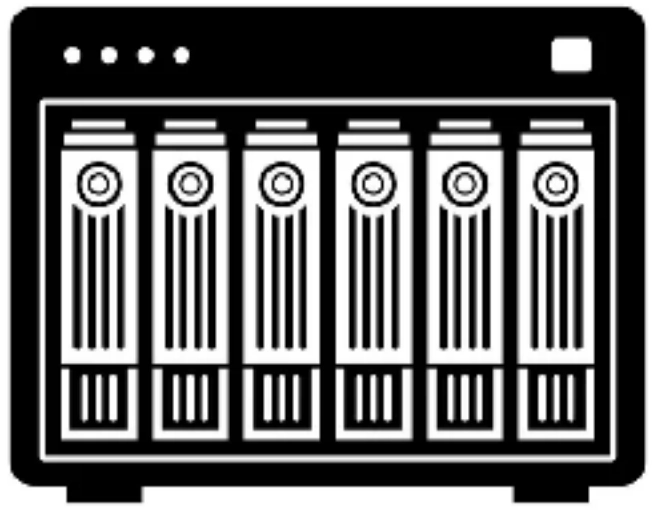
LSM-tree

O’Neil *et al.*

1996

good random writes

good reads



array
of discs

SSD wear-friendly

competitive rand. reads

fast ingestion

LSM-tree
O'Neil *et al.*

poor ingestion perf.

poor query perf.

commodity
hardware

Bigtable

1980s

1996

a decade

2006

LSM-tree

O'Neil *et al.*

1996

Bigtable

2006

APACHE HBASE

2007

LSM-tree
O'Neil *et al.*

1996

Bigtable

2006

APACHE HBASE

2007

cassandra

2010

LSM-tree

O'Neil *et al.*

1996

Bigtable

2006

APACHE HBASE

2007

cassandra

2010

levelDB

2011

LSM-tree
O'Neil *et al.*

1996

Bigtable

2006

APACHE HBASE

2007

cassandra

2010

levelDB

2011

RocksDB

2013

LSM-tree
O'Neil *et al.*

Bigtable

APACHE HBASE

cassandra

levelDB

RocksDB

1996 2006 2007 2010 2011 2013 2023


LSM-tree
O'Neil *et al.*
1996
Bigtable
2006
APACHE HBASE
2007
cassandra
2010
levelDB
2011
RocksDB
2013
2023

LSM-tree

NoSQL

RocksDB
WT
levelDB
SCYLLA
riak
cassandra
tarantool
Bigtable
APACHE HBASE
DynamoDB
accumulo
speedb

SQLite

relational

influxdb
QuasarDB

time-series

2023

LSM-tree

NoSQL

RocksDB
WT
levelDB
SCYLLA
riak
cassandra
tarantool
Bigtable
APACHE HBASE
DynamoDB
accumulo
speedb

SQLite

relational

influxdb
QuasarDB

time-series

2023

Why **LSM** ?

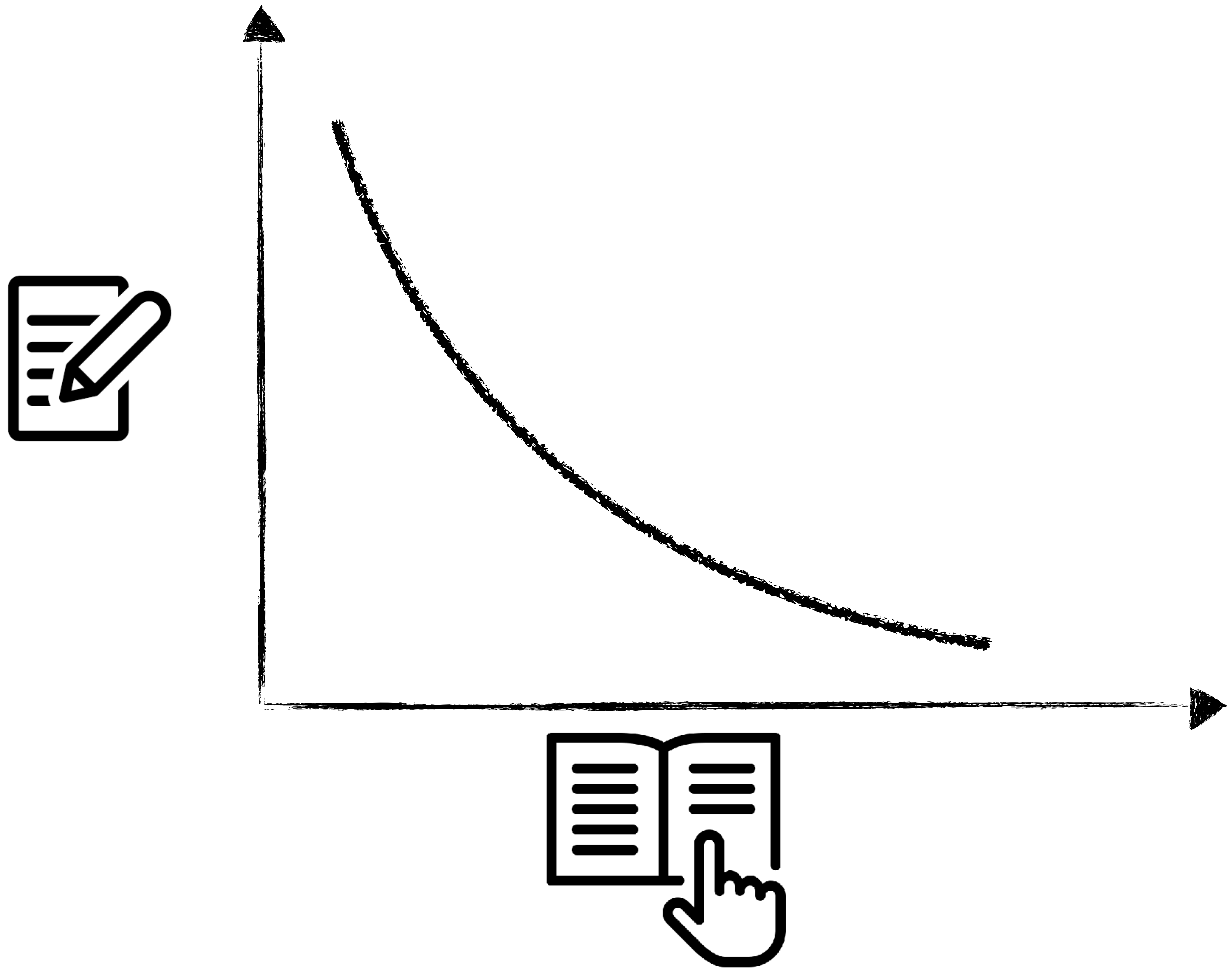
fast ingestion

Why **LSM** ?

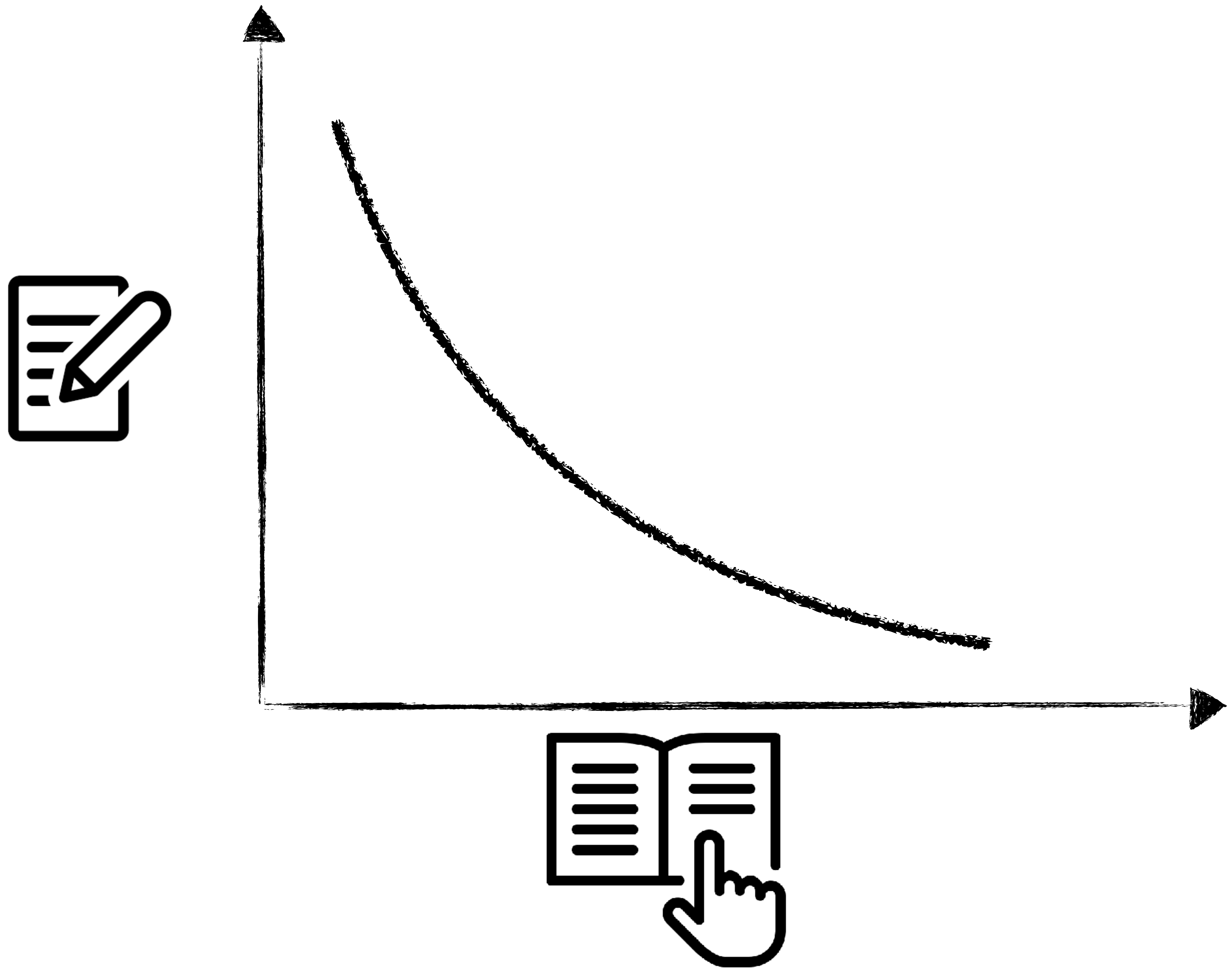
fast ingestion

competitive reads

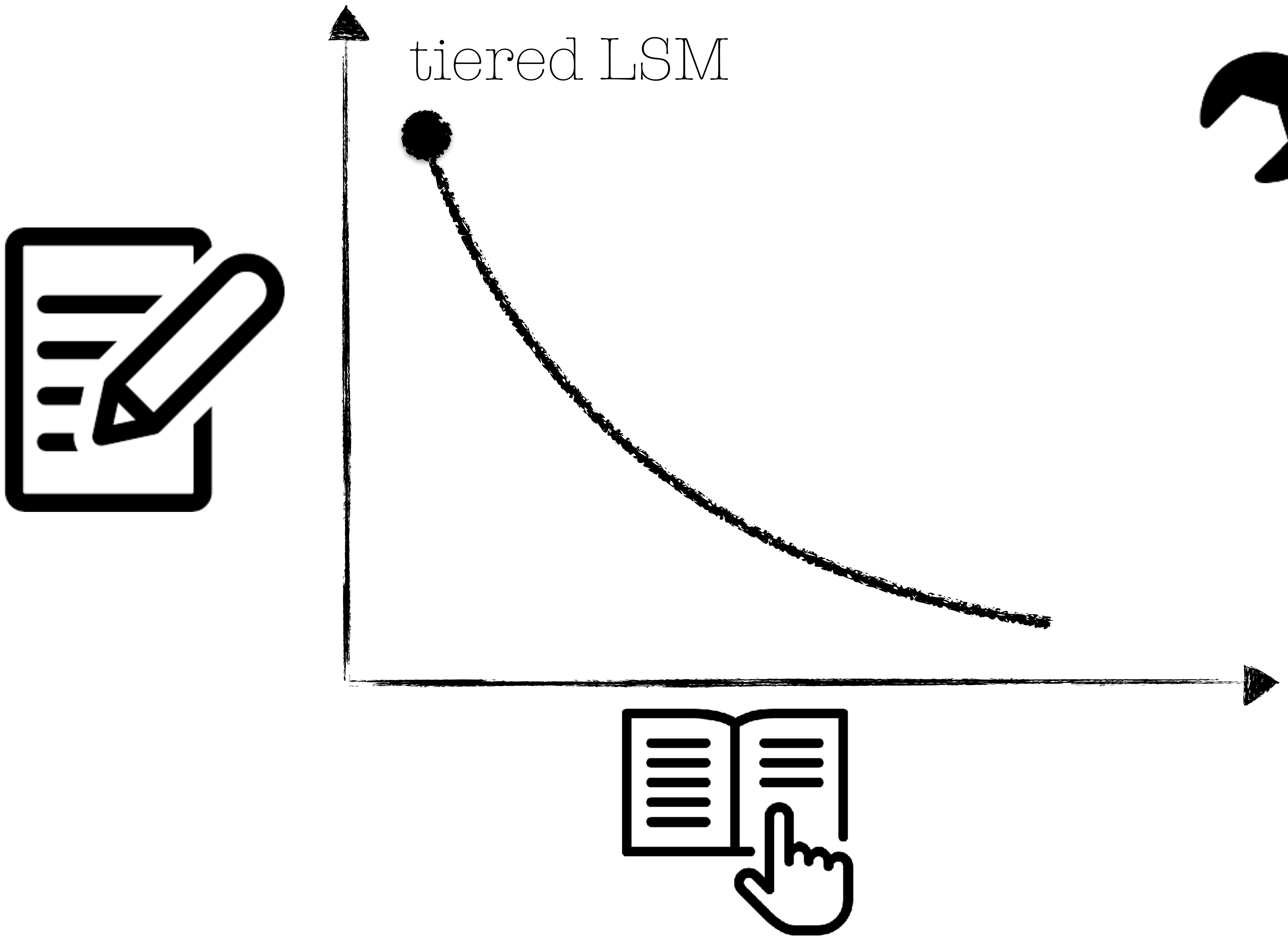
Why **LSM** ?



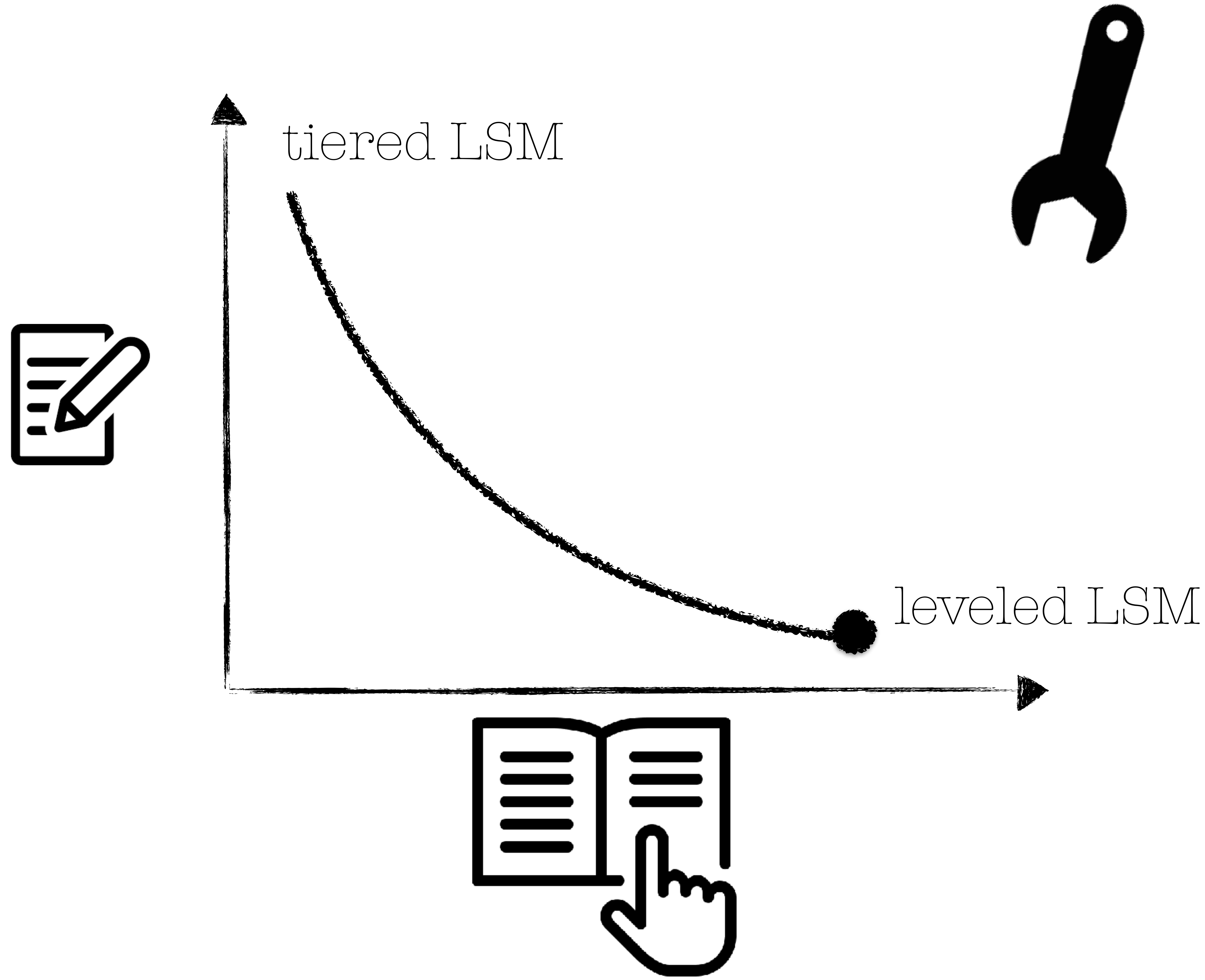
Why **LSM** ?



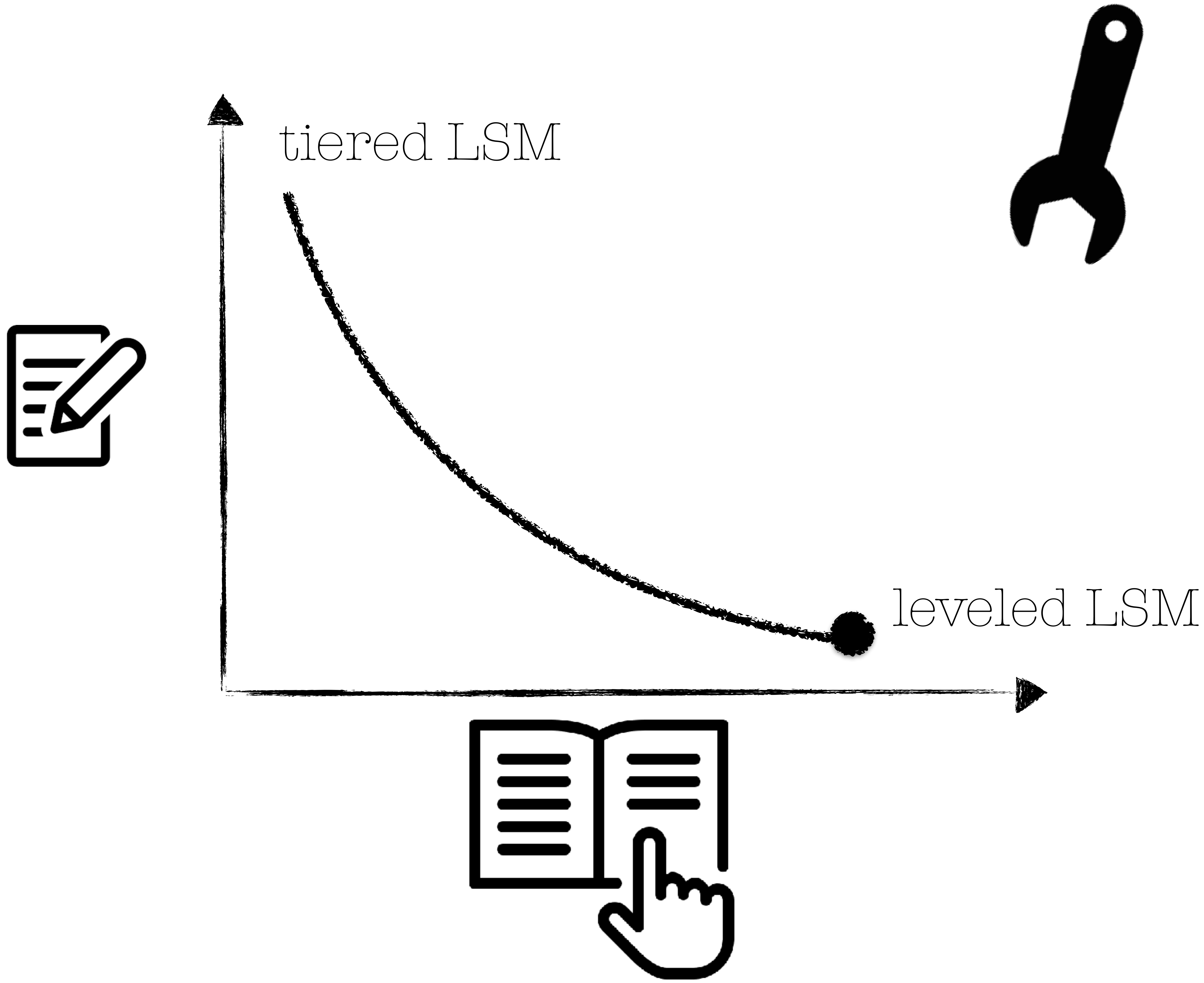
Why **LSM** ?



Why **LSM** ?

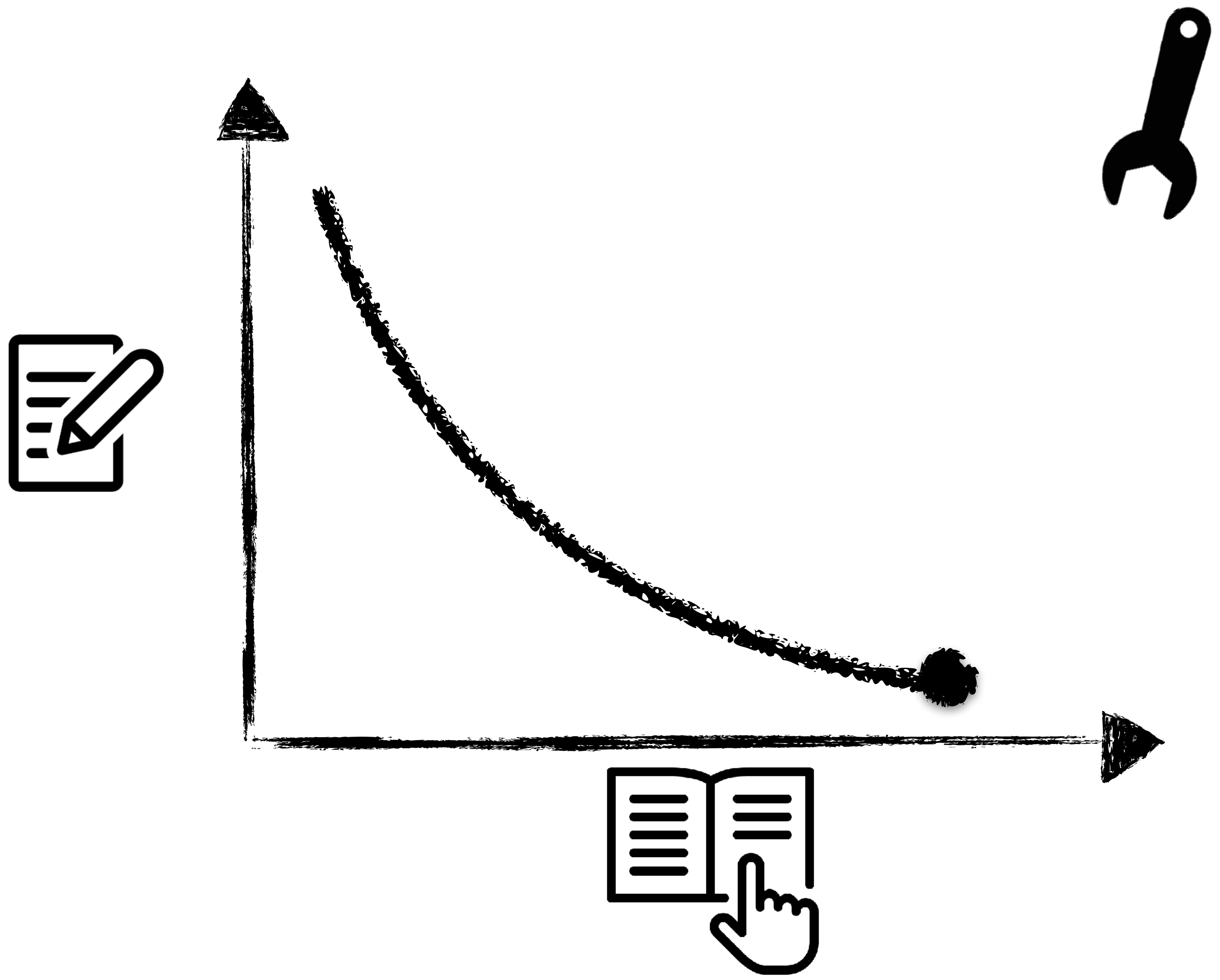


Why **LSM** ?

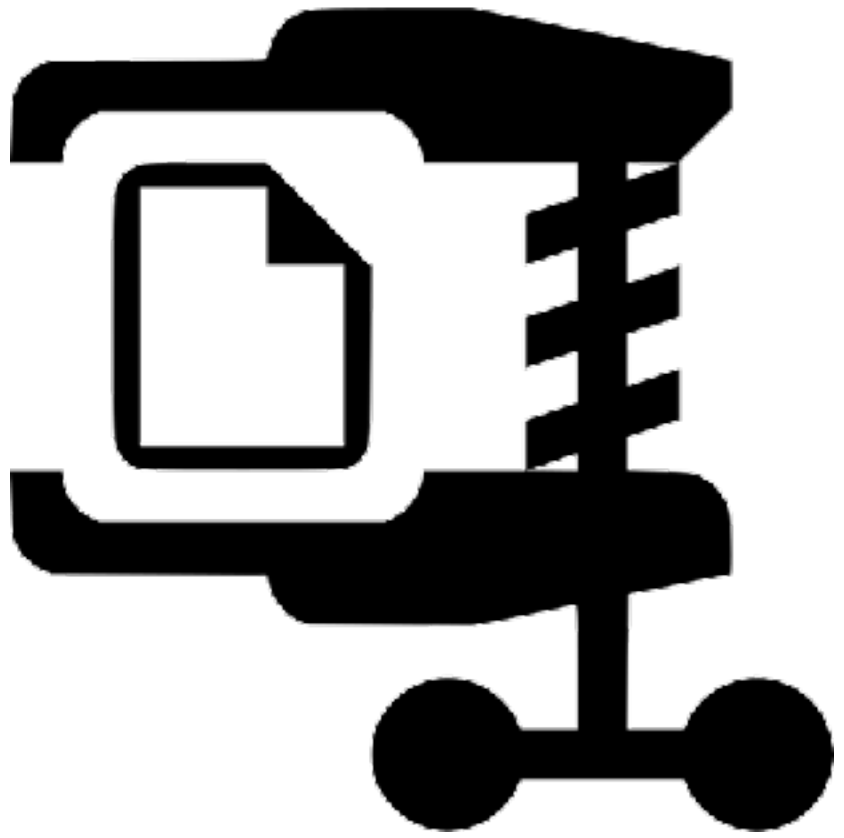


Why **LSM** ?

fast writes

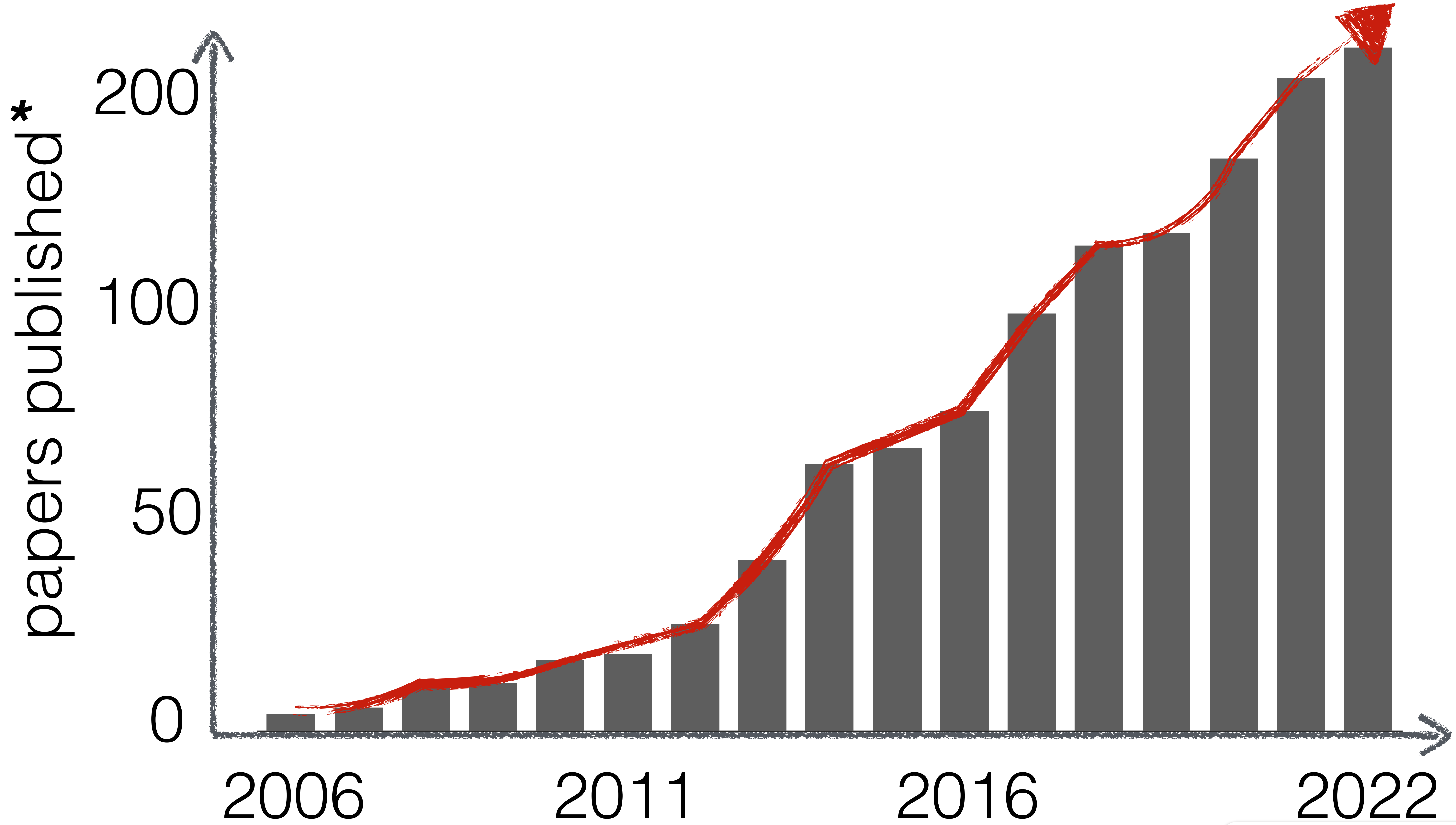


tunable read-write performance

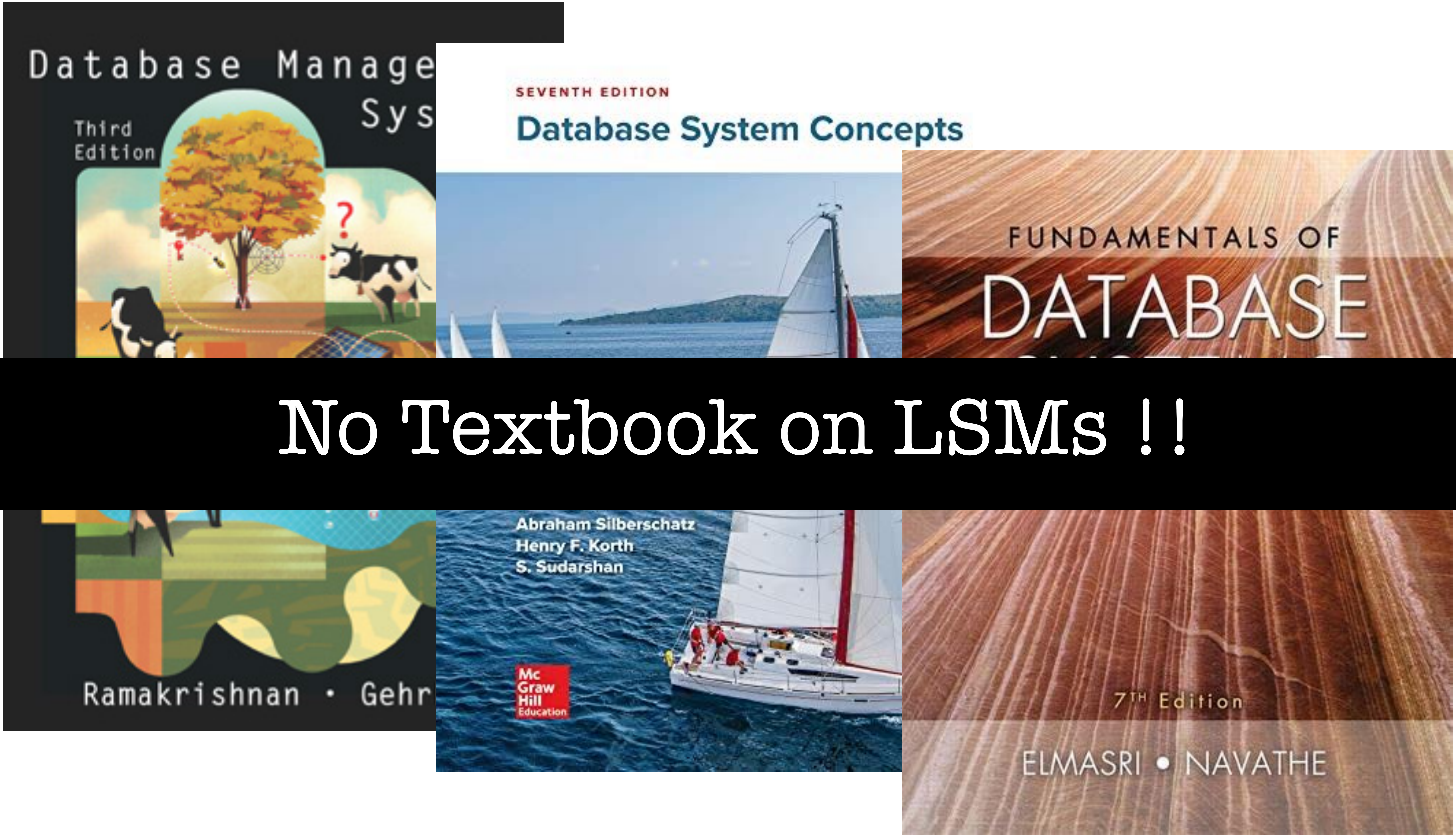


good space utilization

Research Trend



* data from Google scholar



No Textbook on LSMs !!

No Textbook on LSMs !!

explore the LSM paradigm

understand the **read optimizations**

provide useful **insights**

Outline

Part 1: **LSM Basics**

Part 2: **Read Optimizations in LSMs**

Part 3: **Navigating the LSM Design Space**

Outline

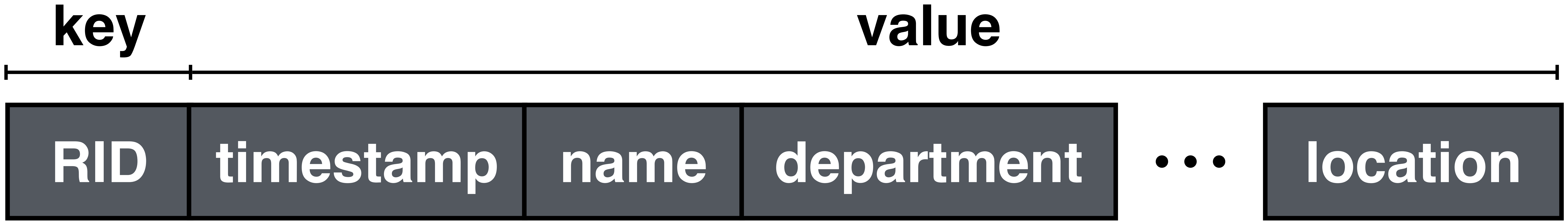
Part 1: **LSM Basics**

Part 2: **Read Optimizations in LSMs**

Part 3: **Navigating the LSM Design Space**

LSM **Basics**

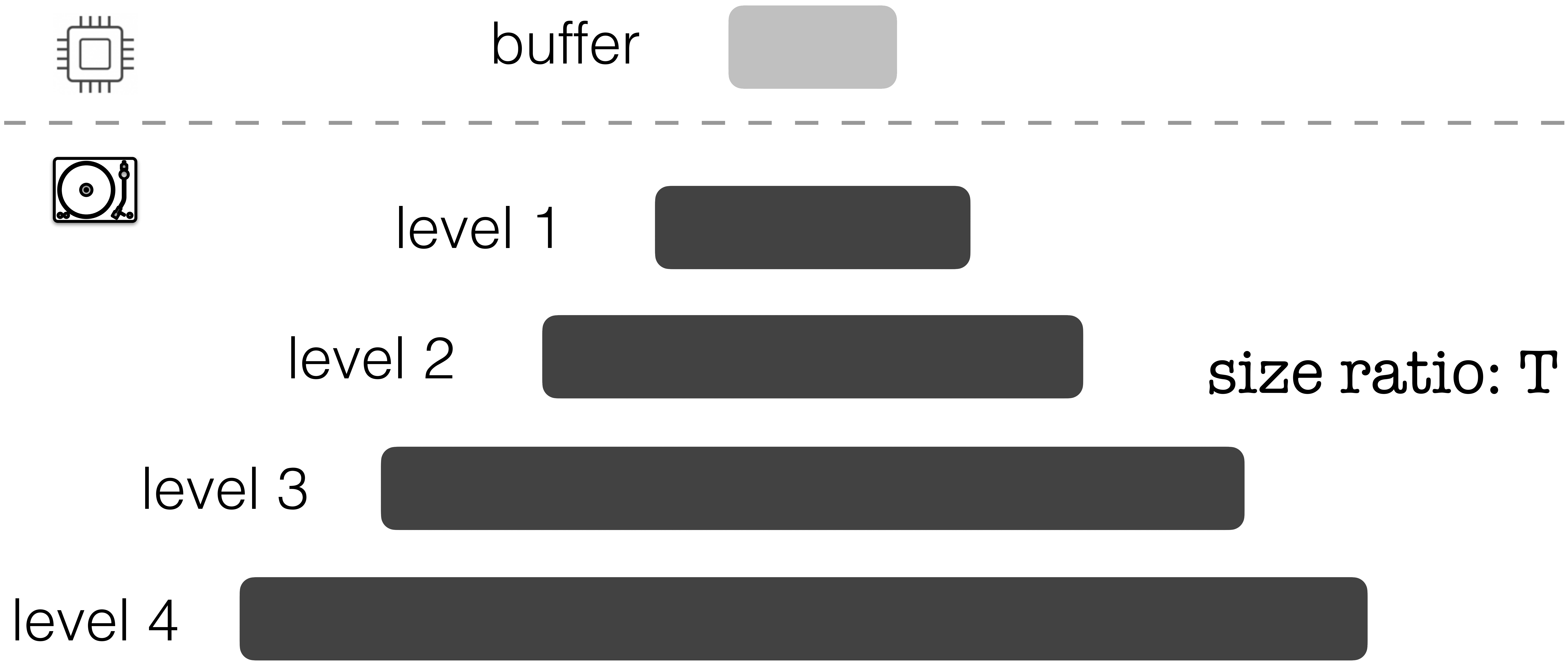
key-value pairs



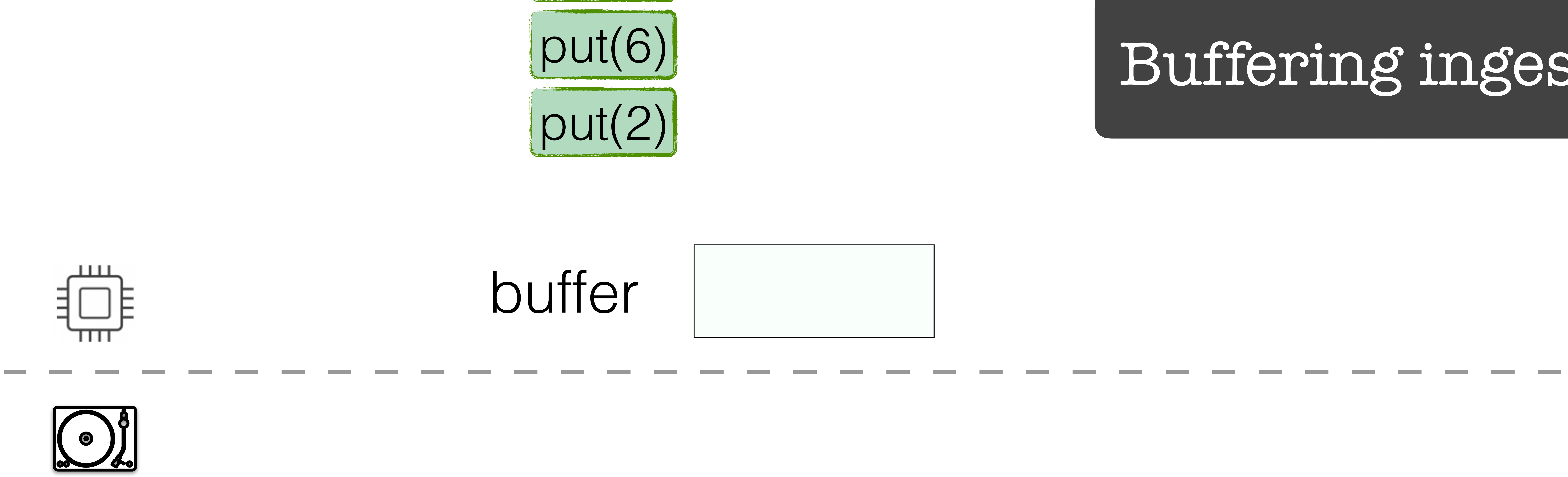
LSM **Basics**

key-value pairs

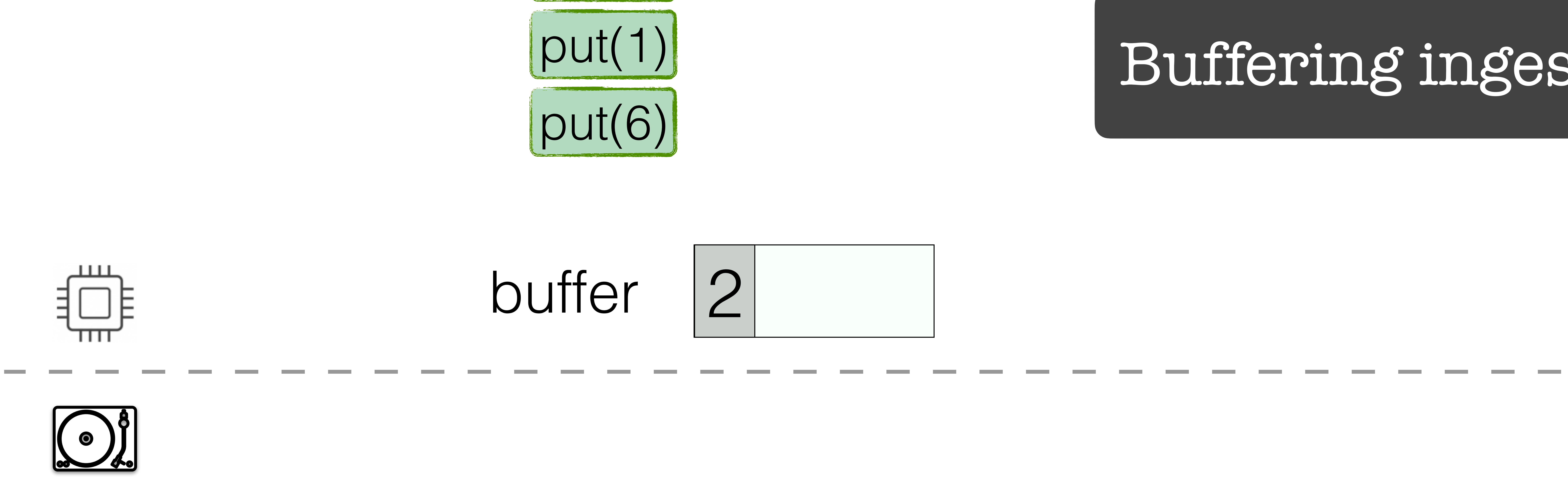
LSM **Basics**



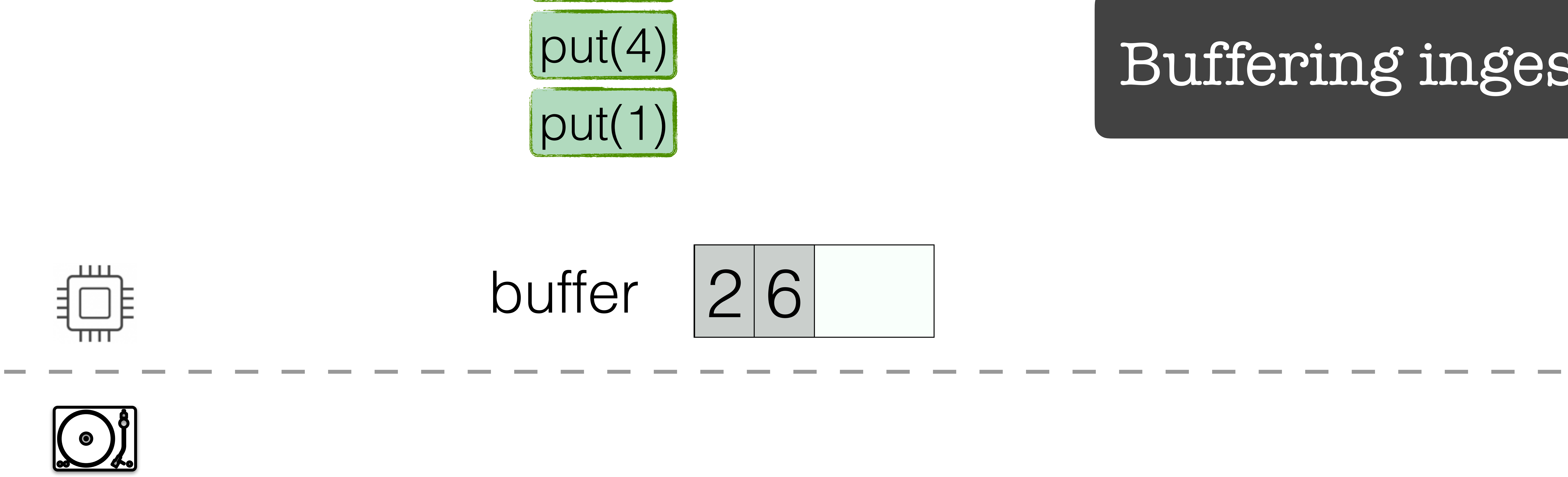
Buffering ingestion



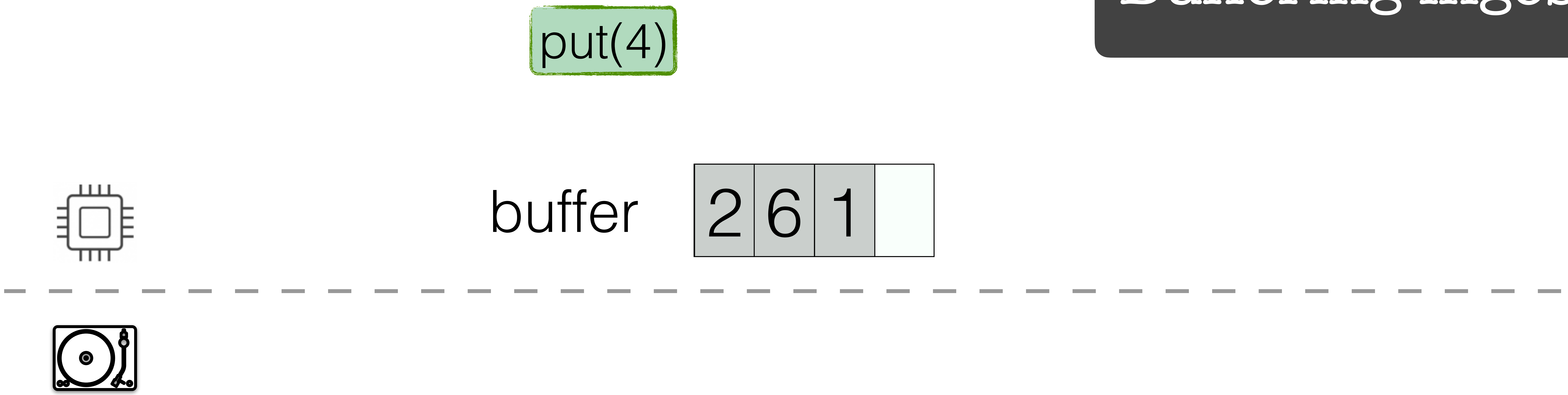
Buffering ingestion



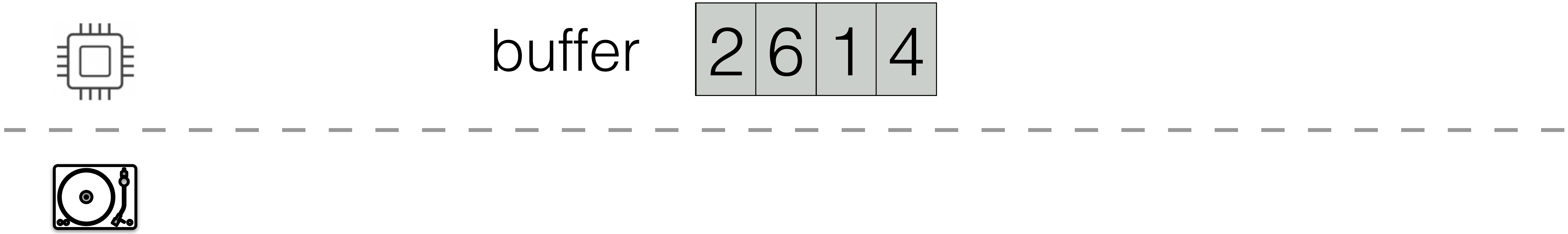
Buffering ingestion



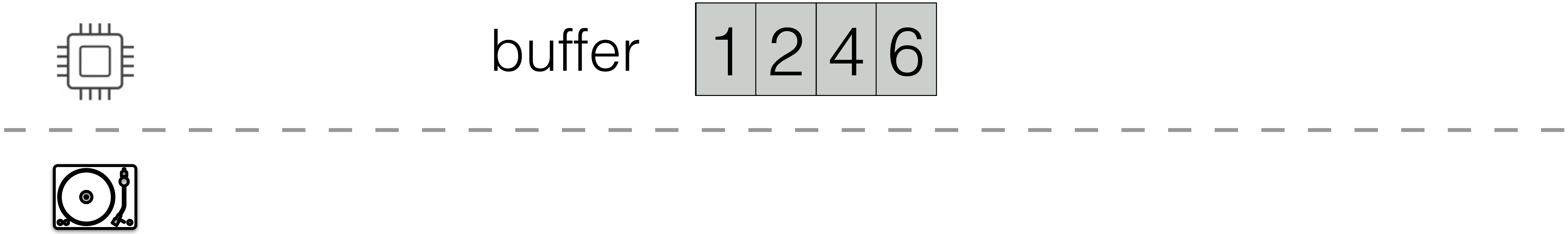
Buffering ingestion



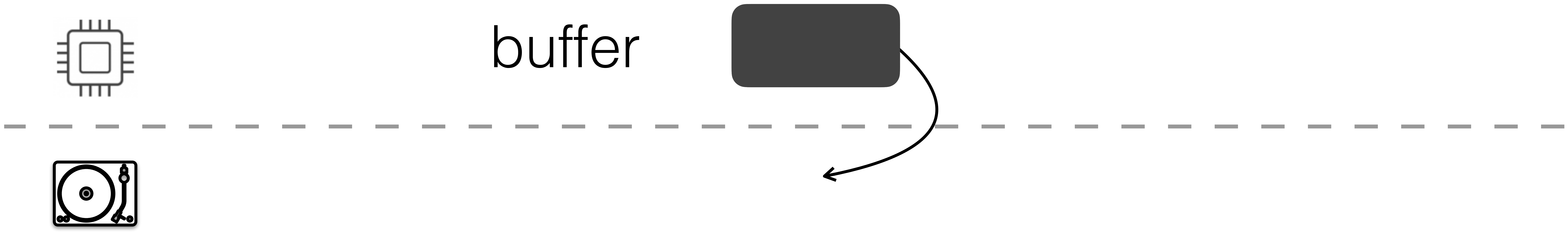
Buffering ingestion



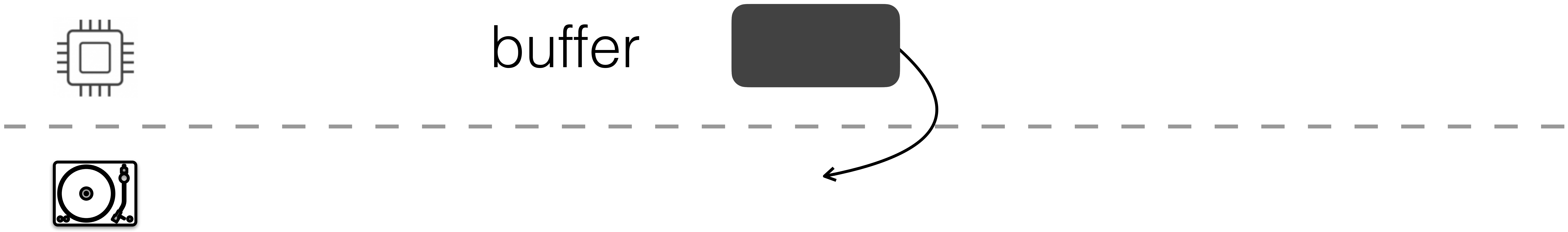
Buffering ingestion



Buffering ingestion

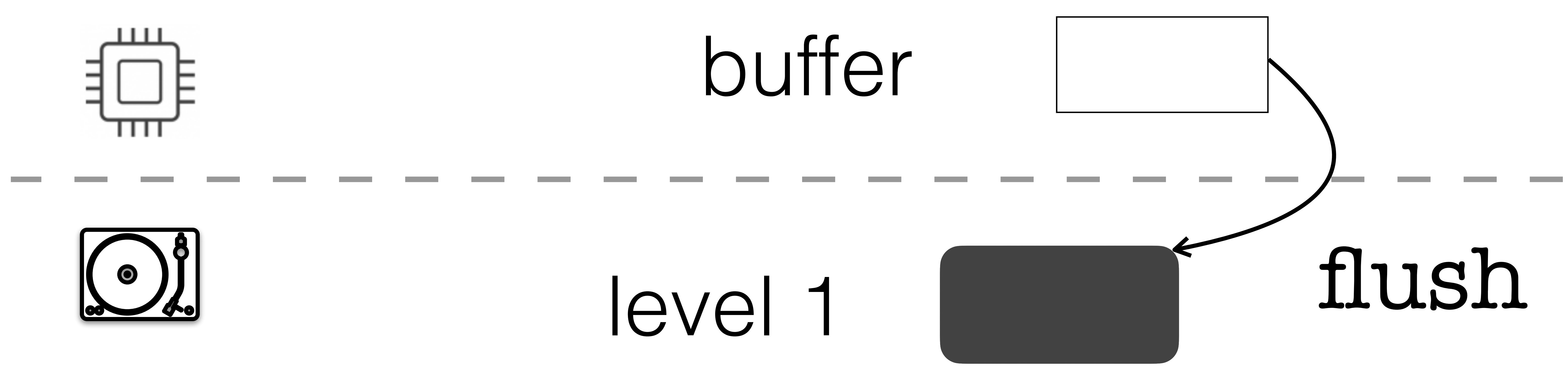


Buffering ingestion

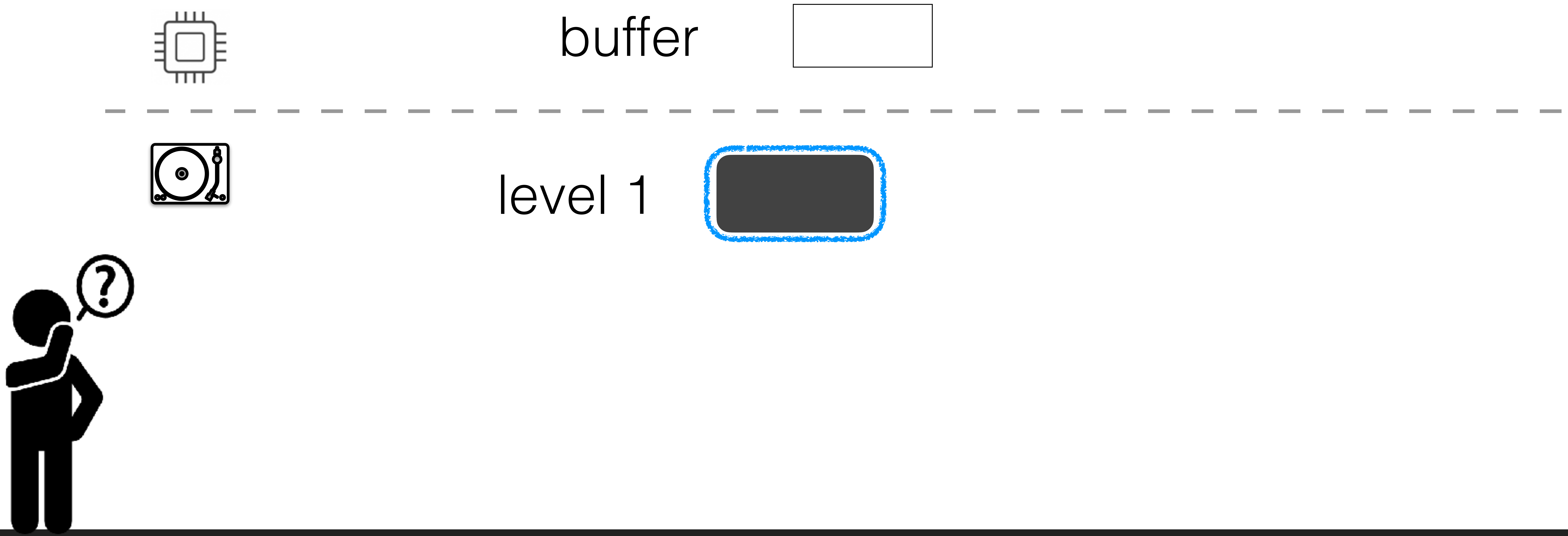


Buffering ingestion

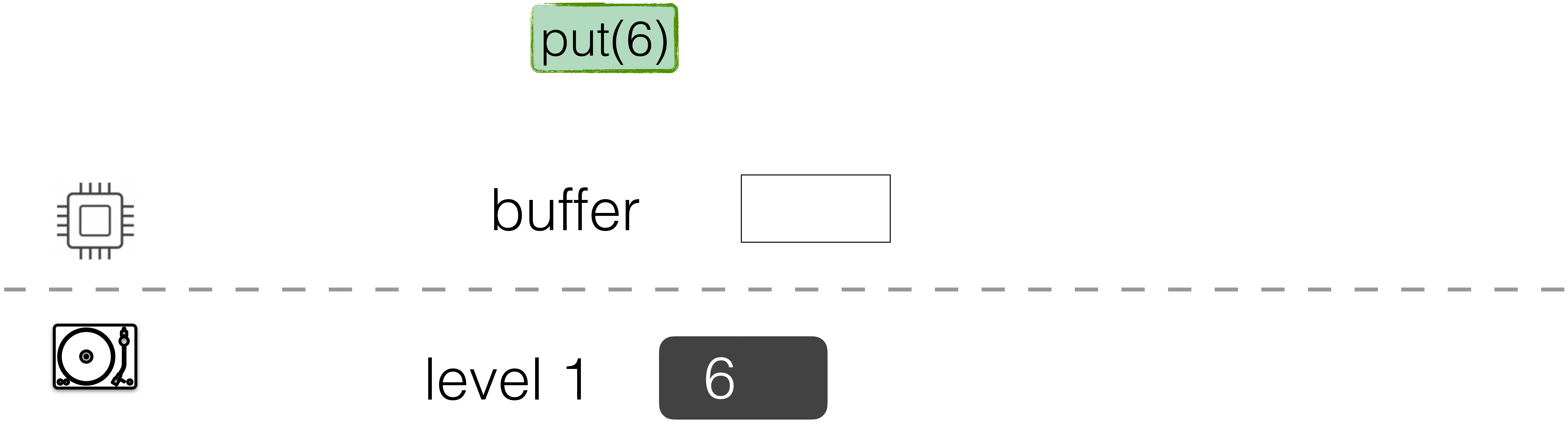
- low ingestion cost
- fast temporal reads

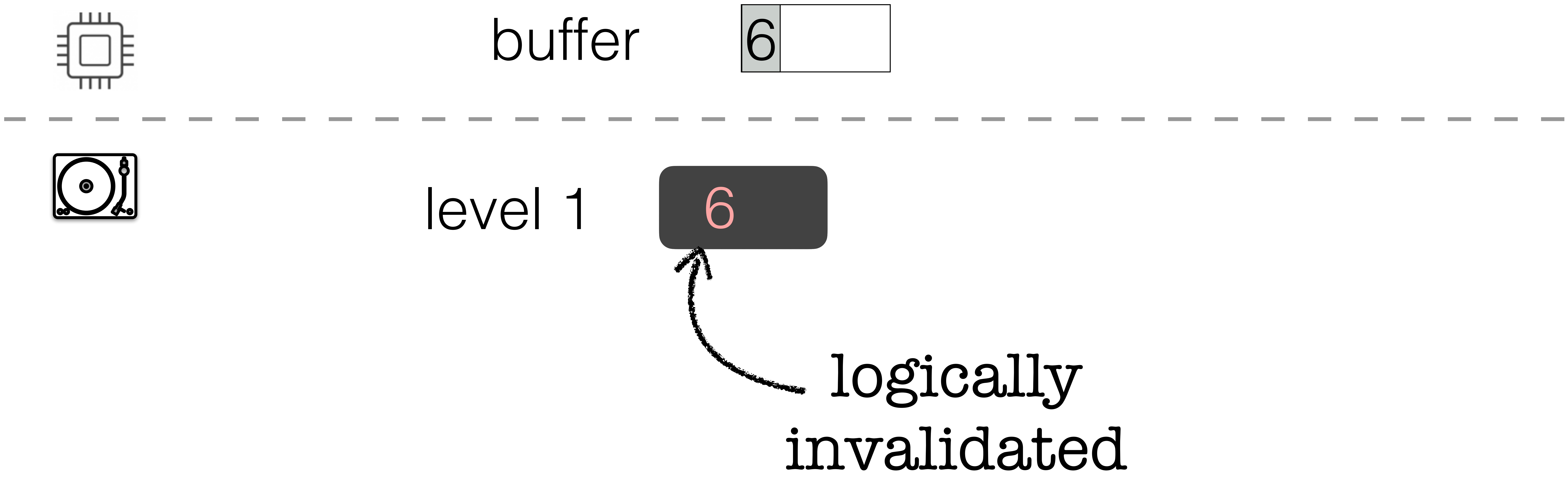


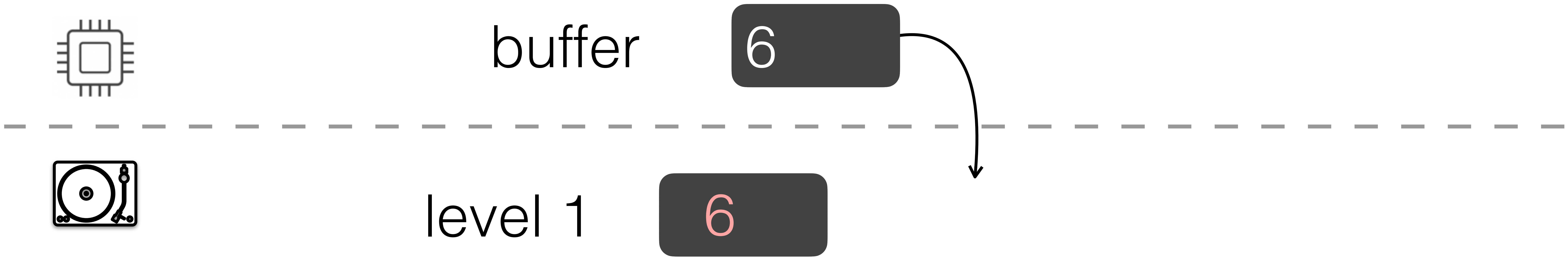
Immutable files on storage

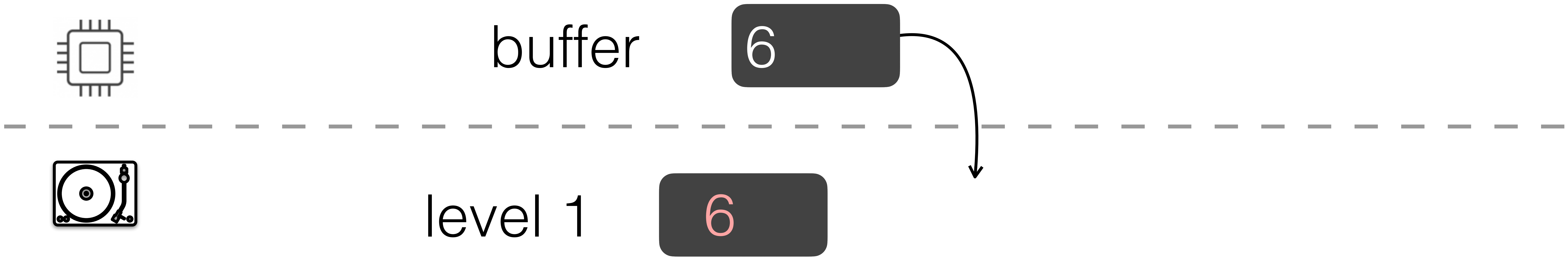


How do we update data?


put(6)
buffer
level 1
6

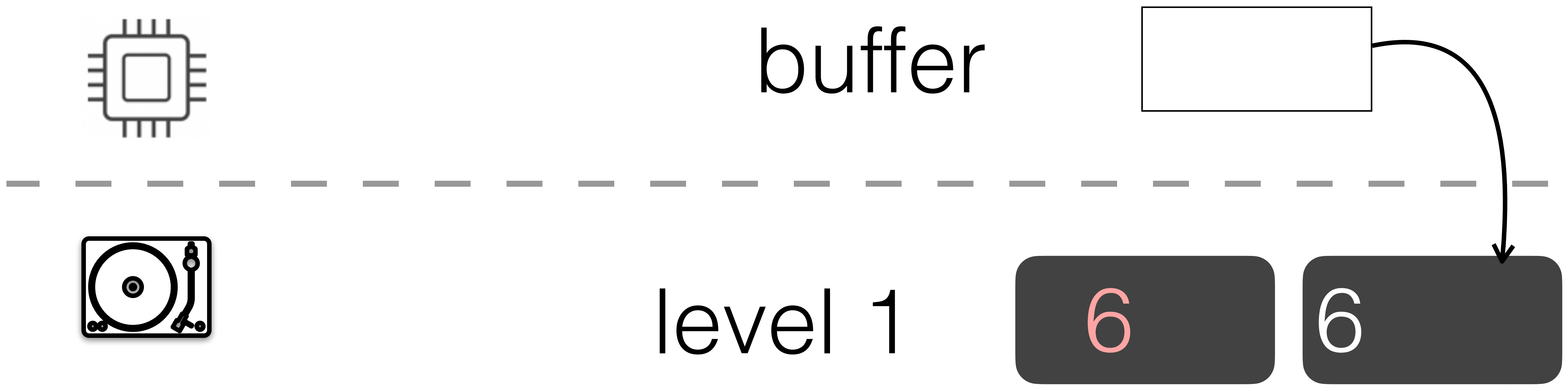

buffer
6
level 1
6
logically
invalidated


buffer
6
level 1
6

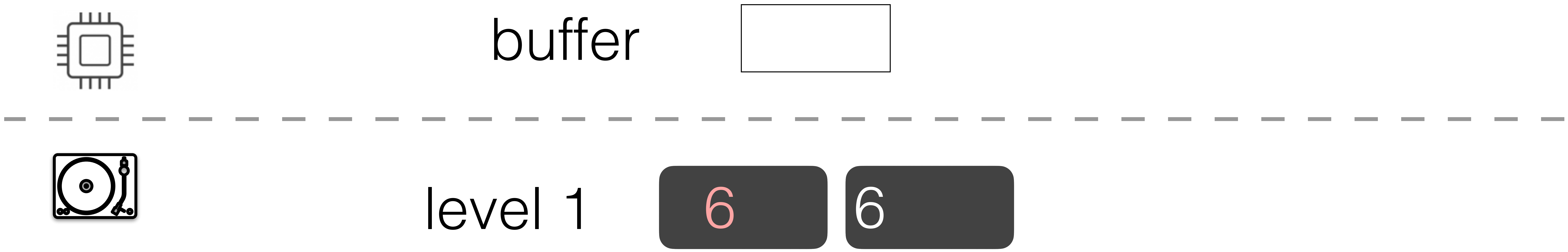

buffer
6
level 1
6

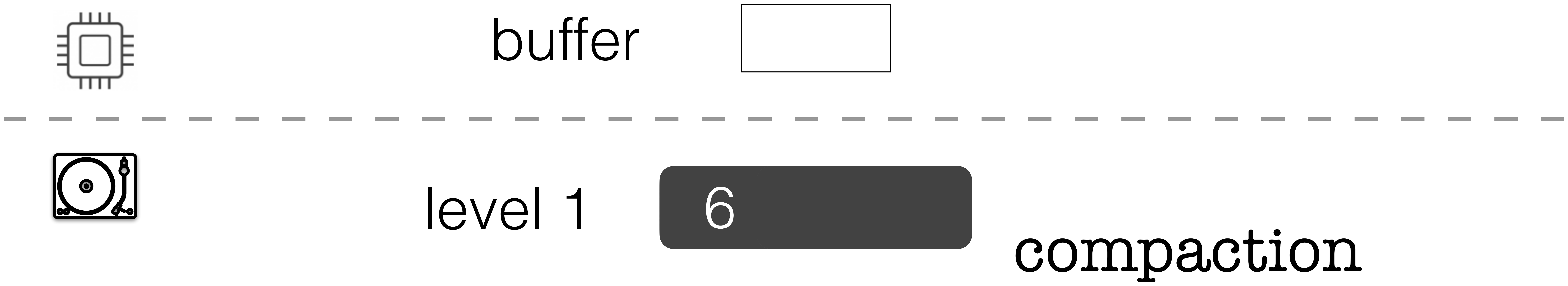
Out-of-place updates

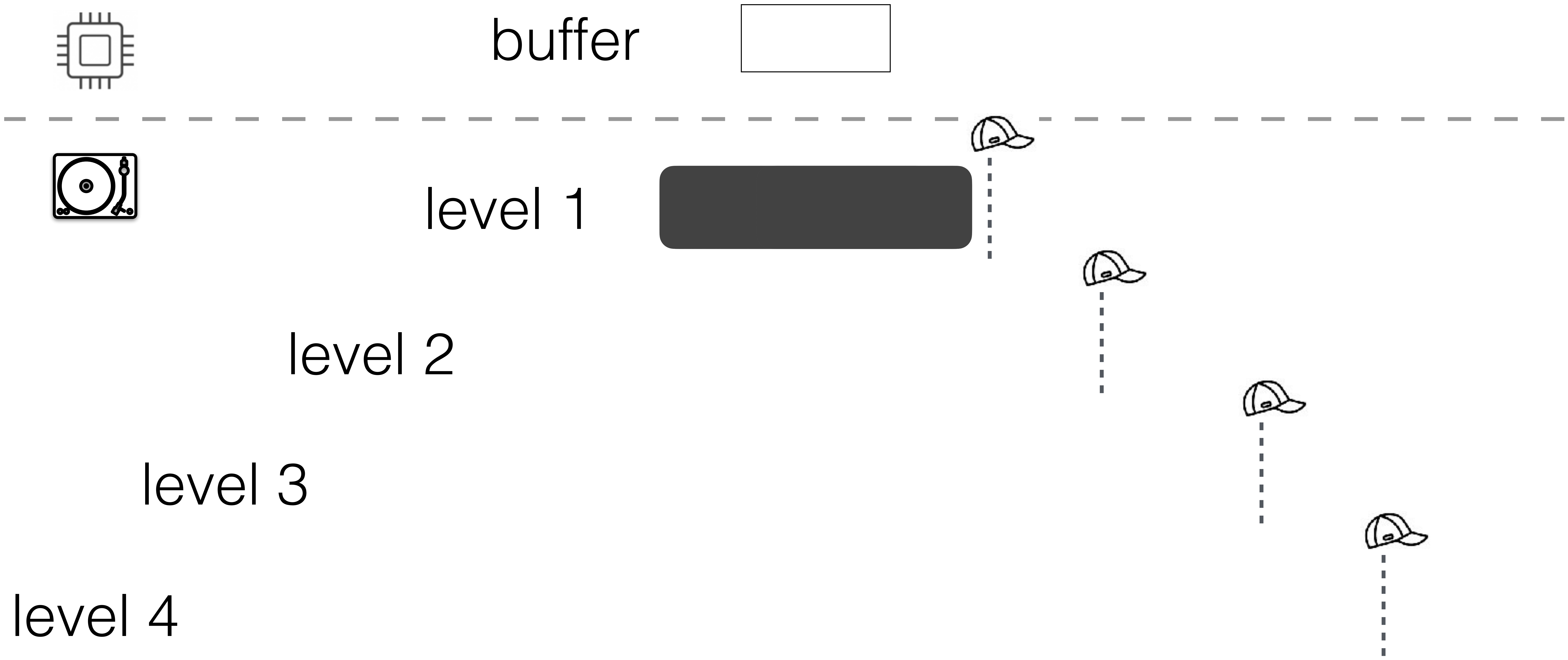
- fast ingestion
- space amplification
- slow reads

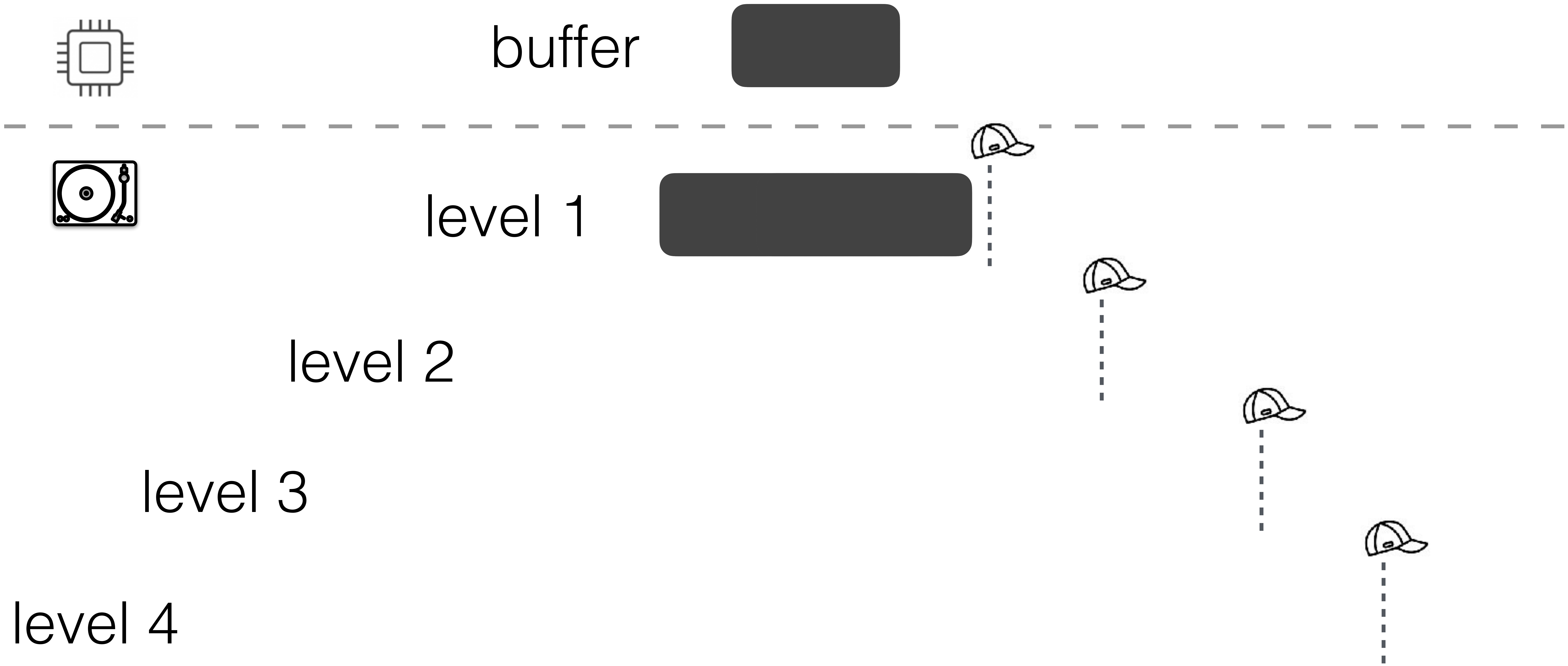


How do we reduce this space amplification?

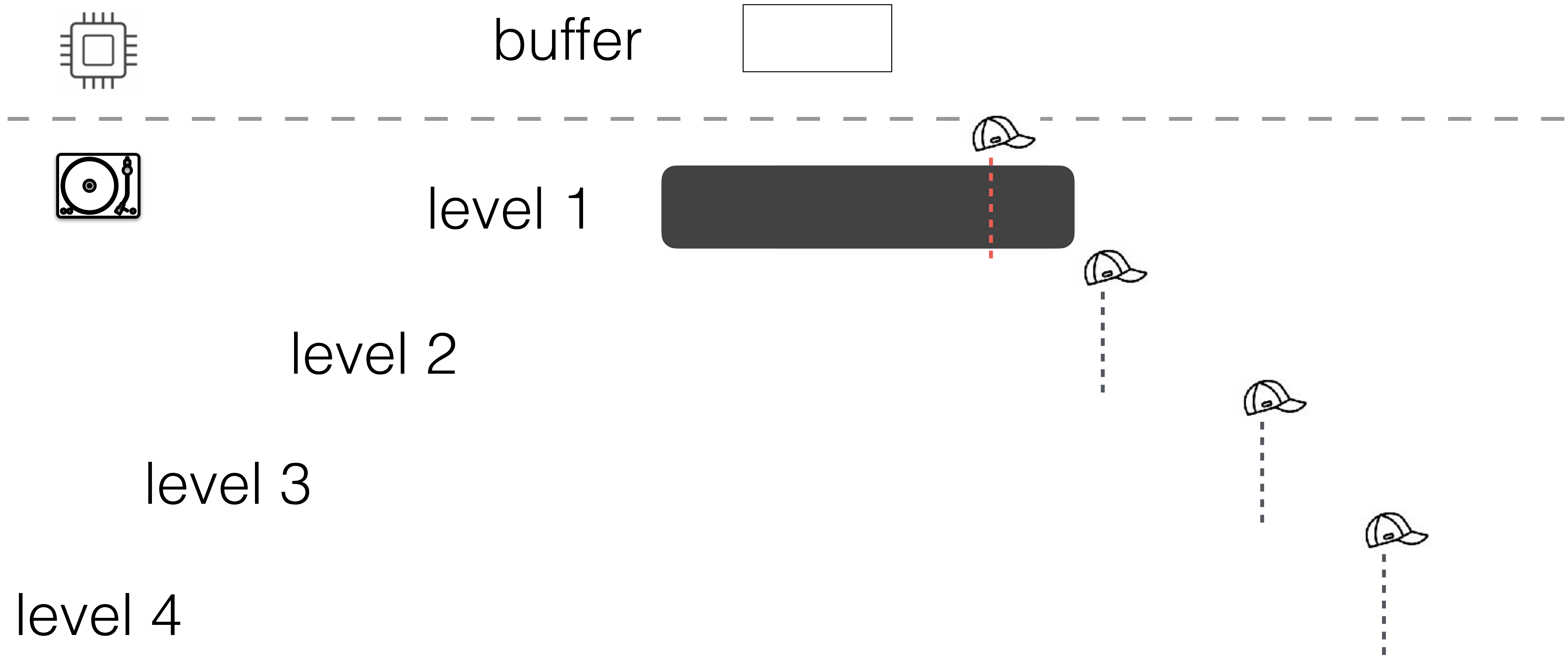

buffer
level 1
6
6

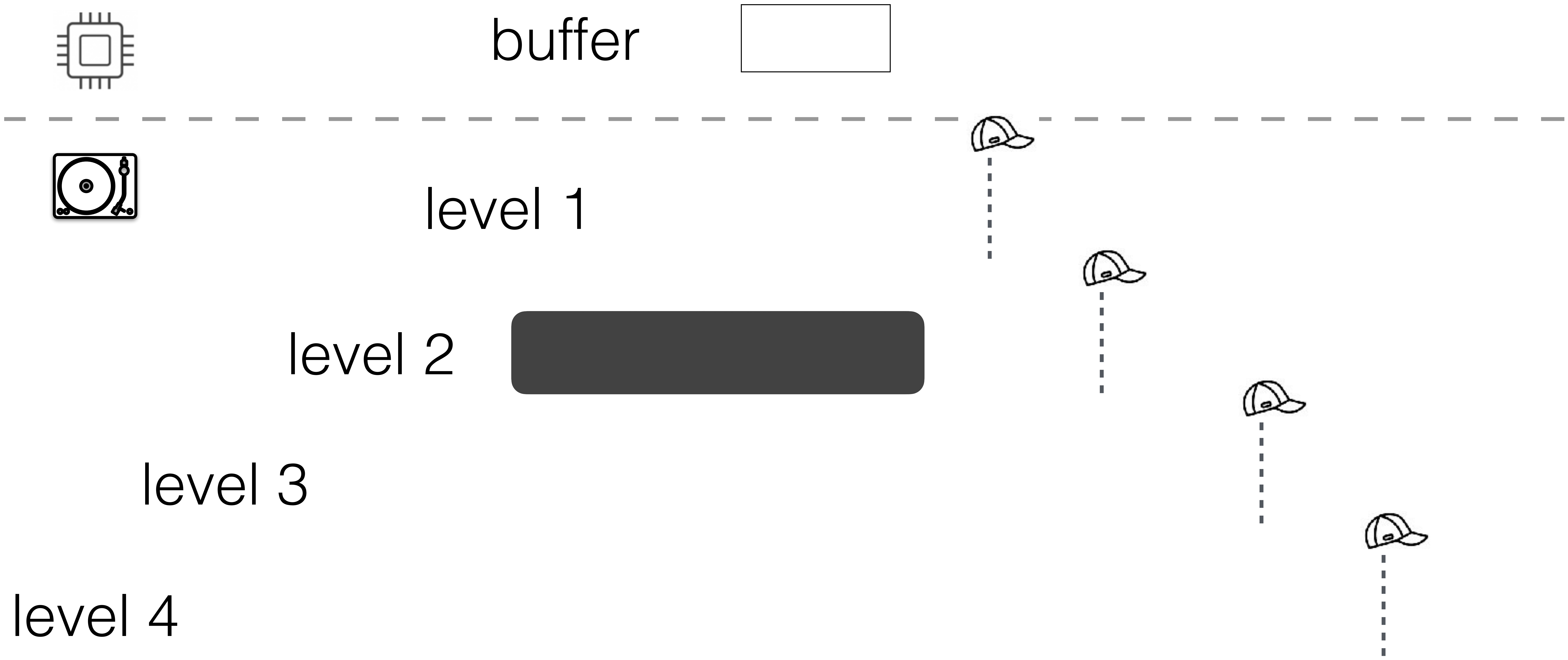

buffer
level 1
6
compaction

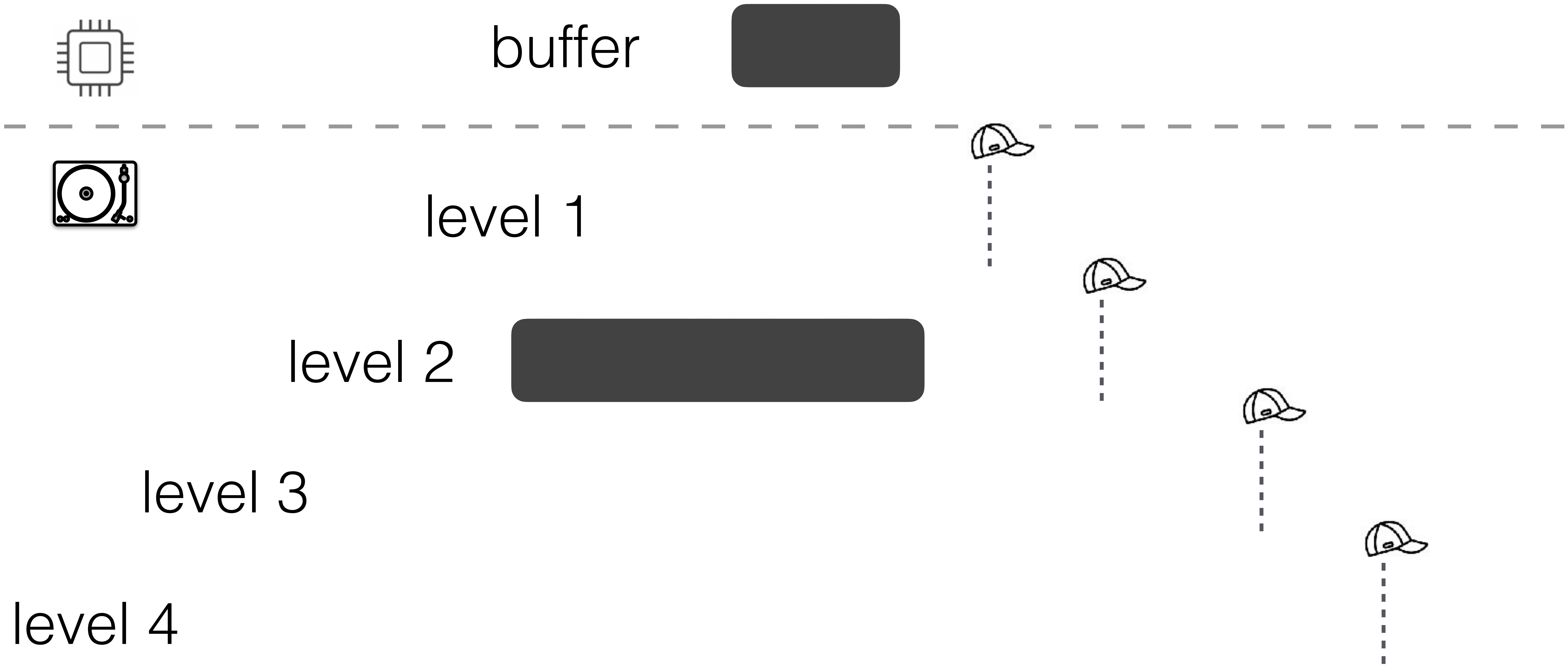

buffer
level 1
level 2
level 3
level 4


buffer
level 1
level 2
level 3
level 4

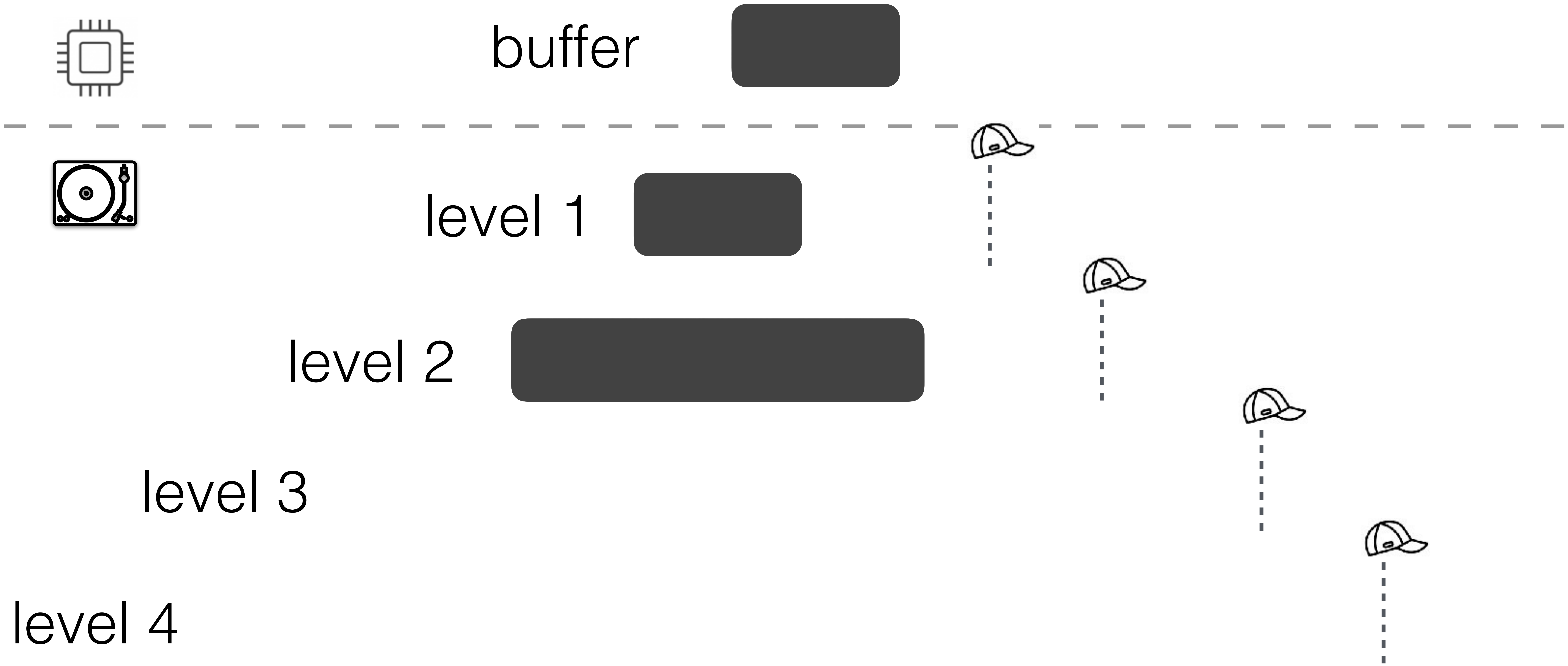

buffer
level 1
level 2
level 3
level 4

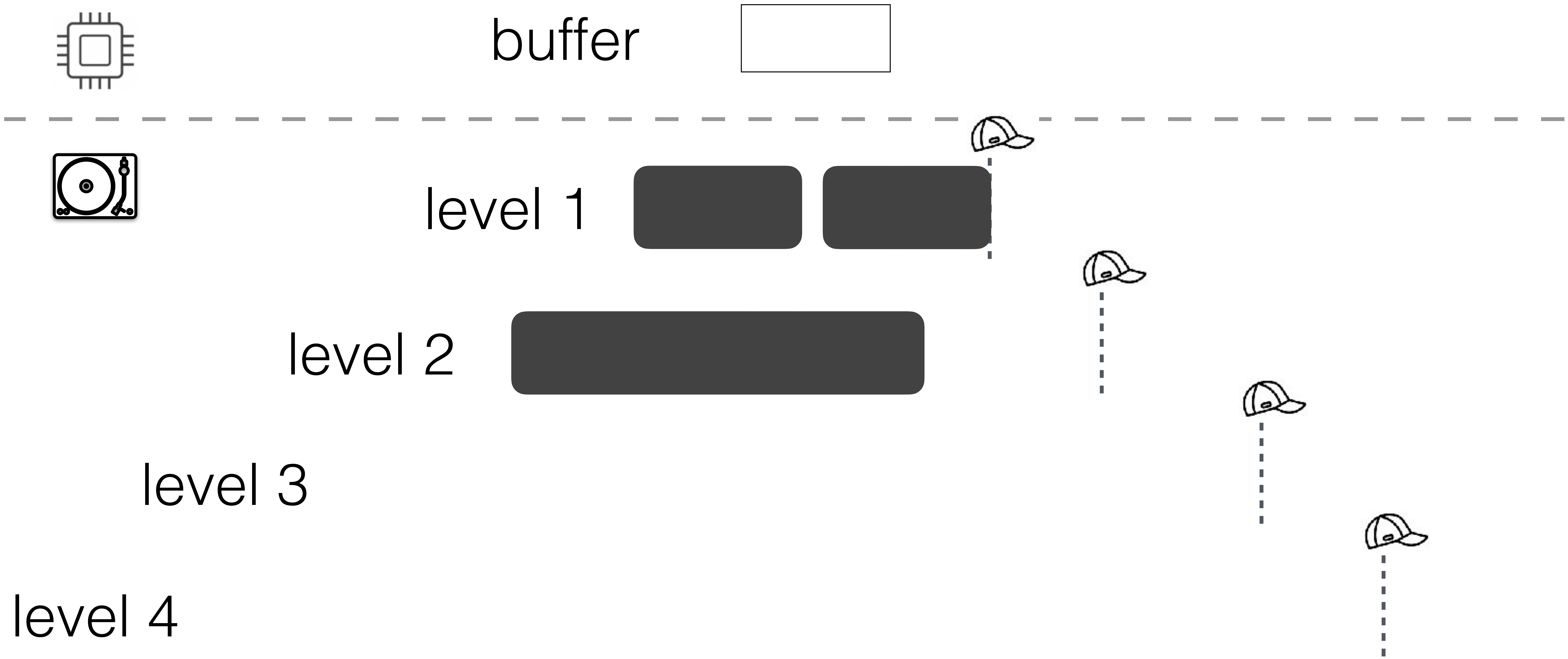

buffer
level 1
level 2
level 3
level 4

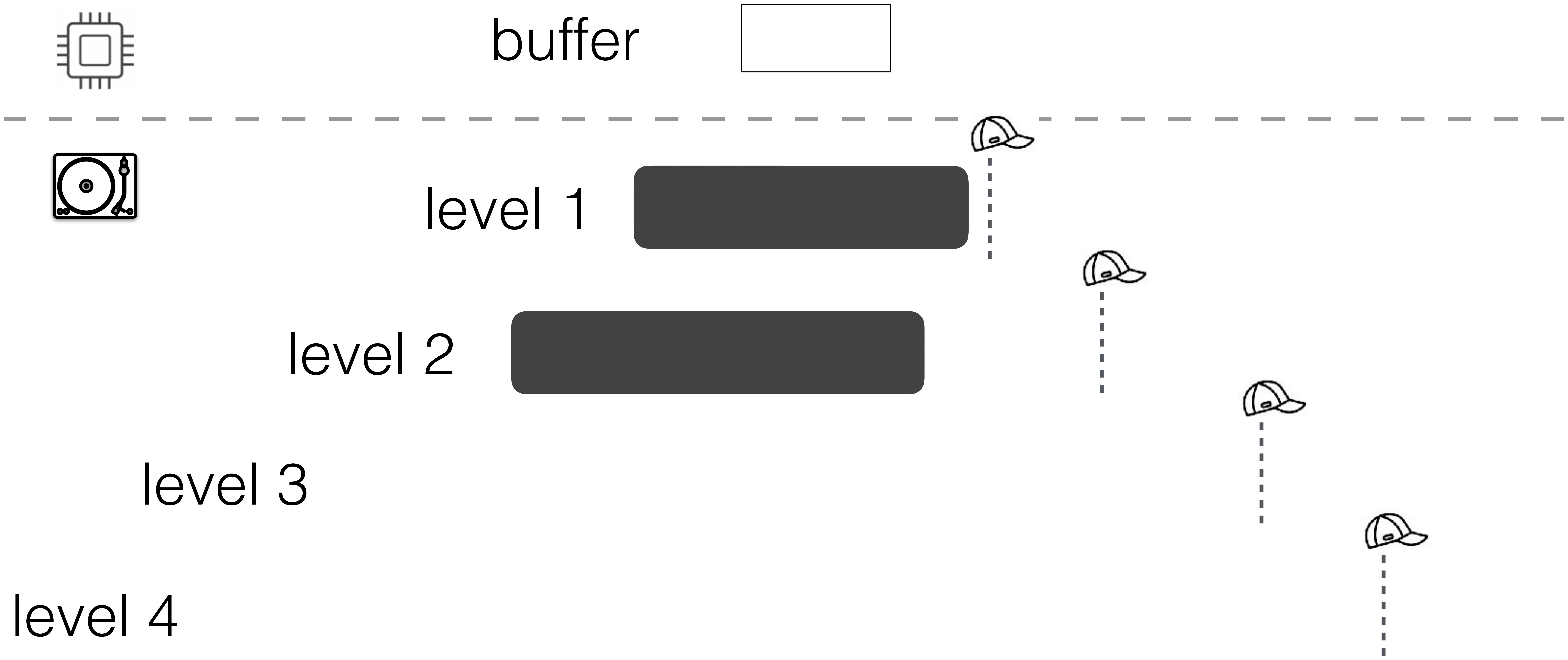

buffer
level 1
level 2
level 3
level 4

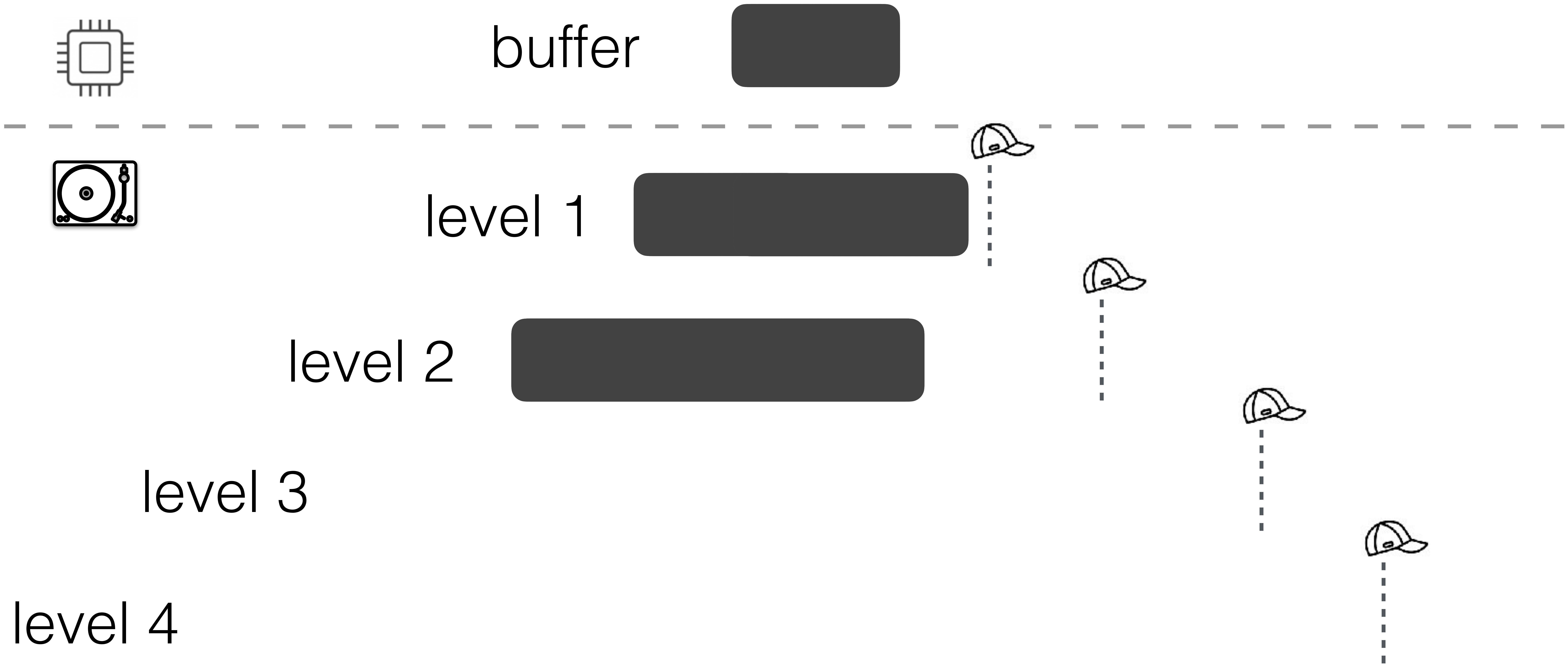

buffer
level 1
level 2
level 3
level 4


buffer
level 1
level 2
level 3
level 4

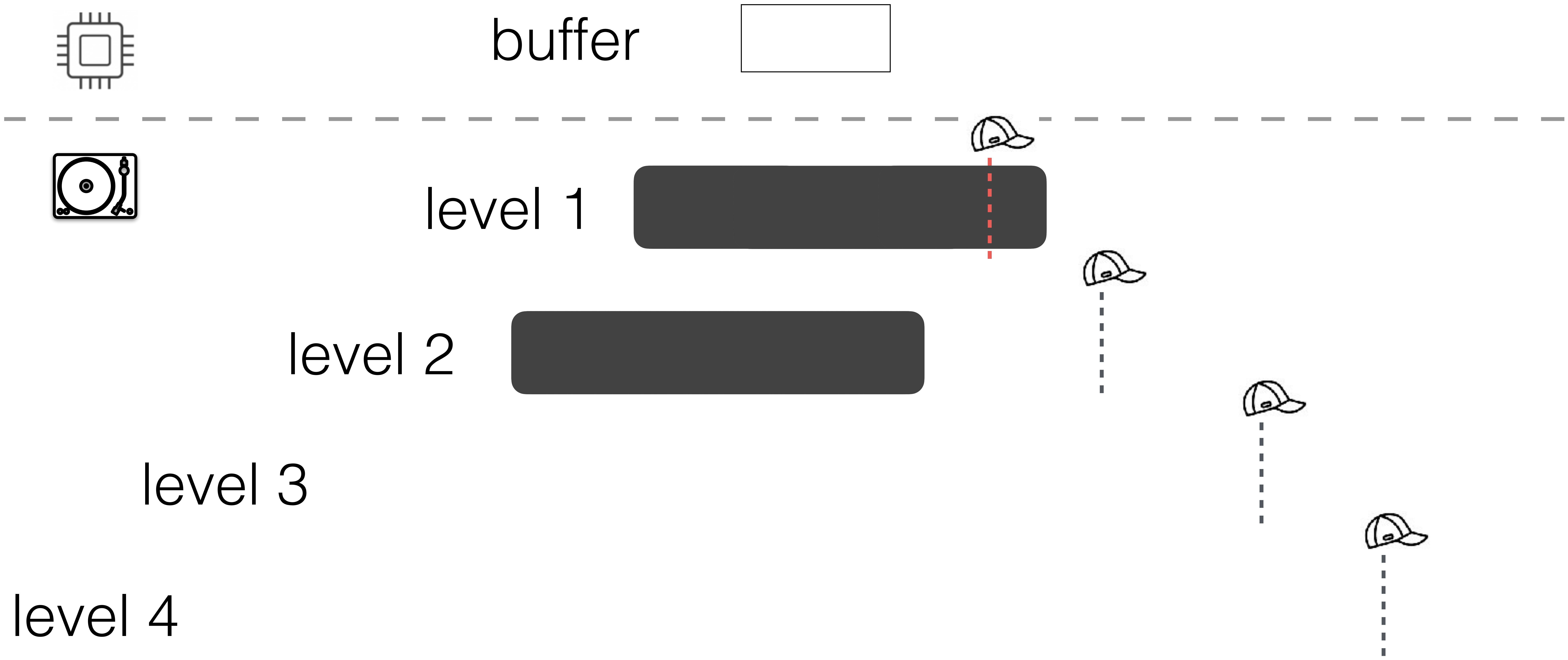

buffer
level 1
level 2
level 3
level 4

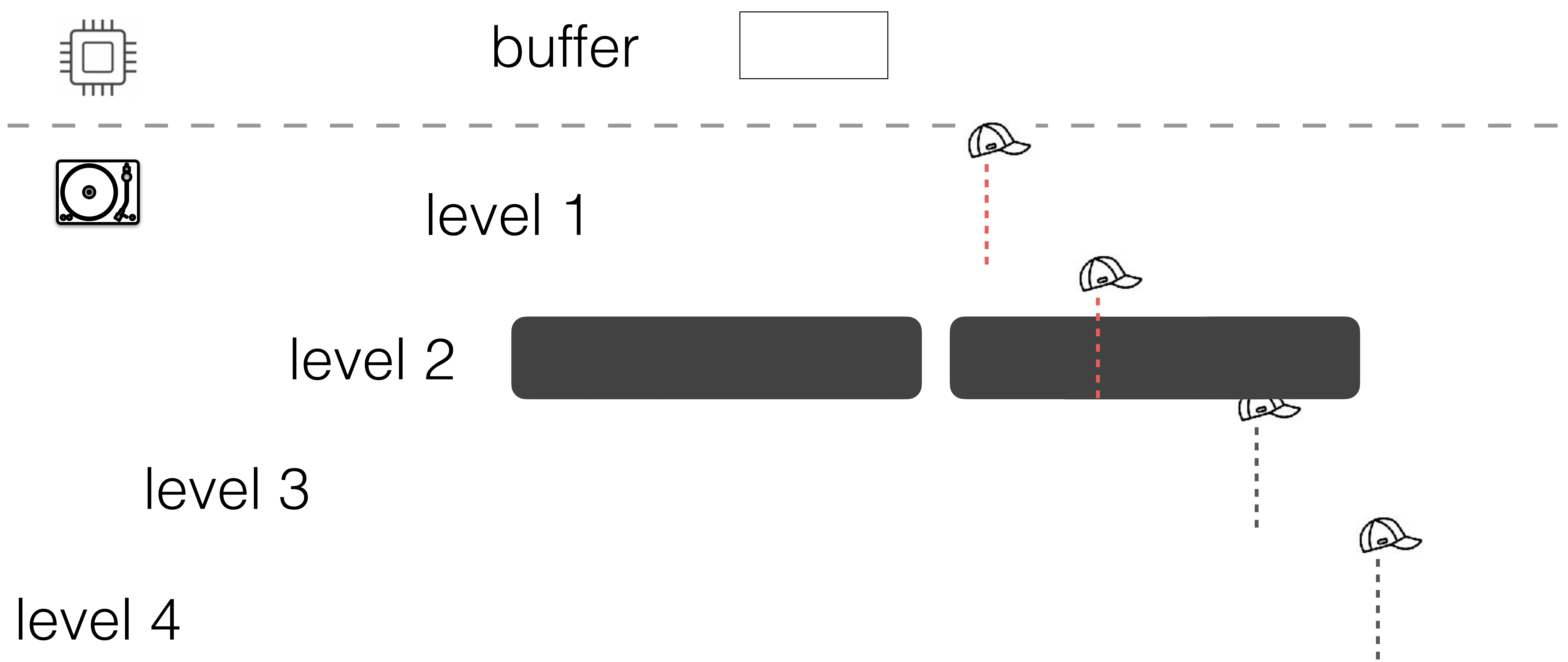

buffer
level 1
level 2
level 3
level 4

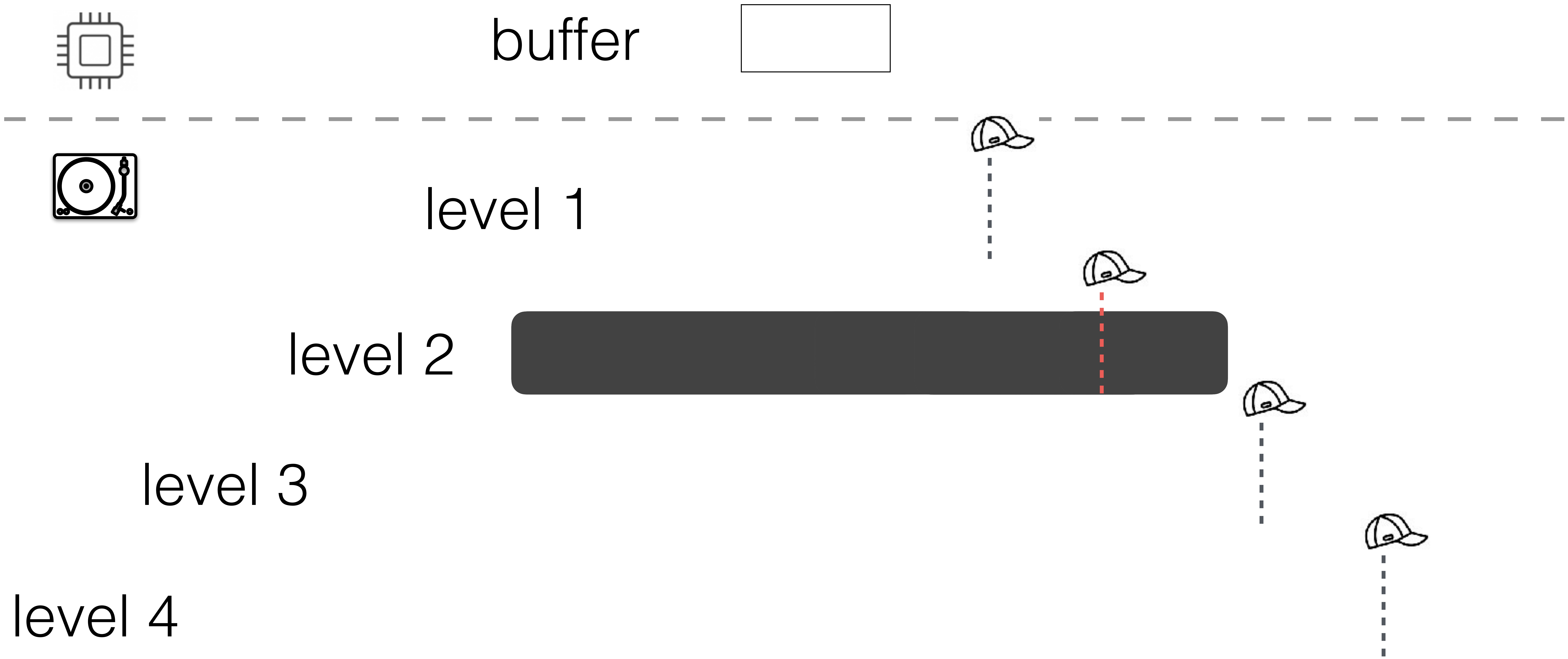

buffer
level 1
level 2
level 3
level 4

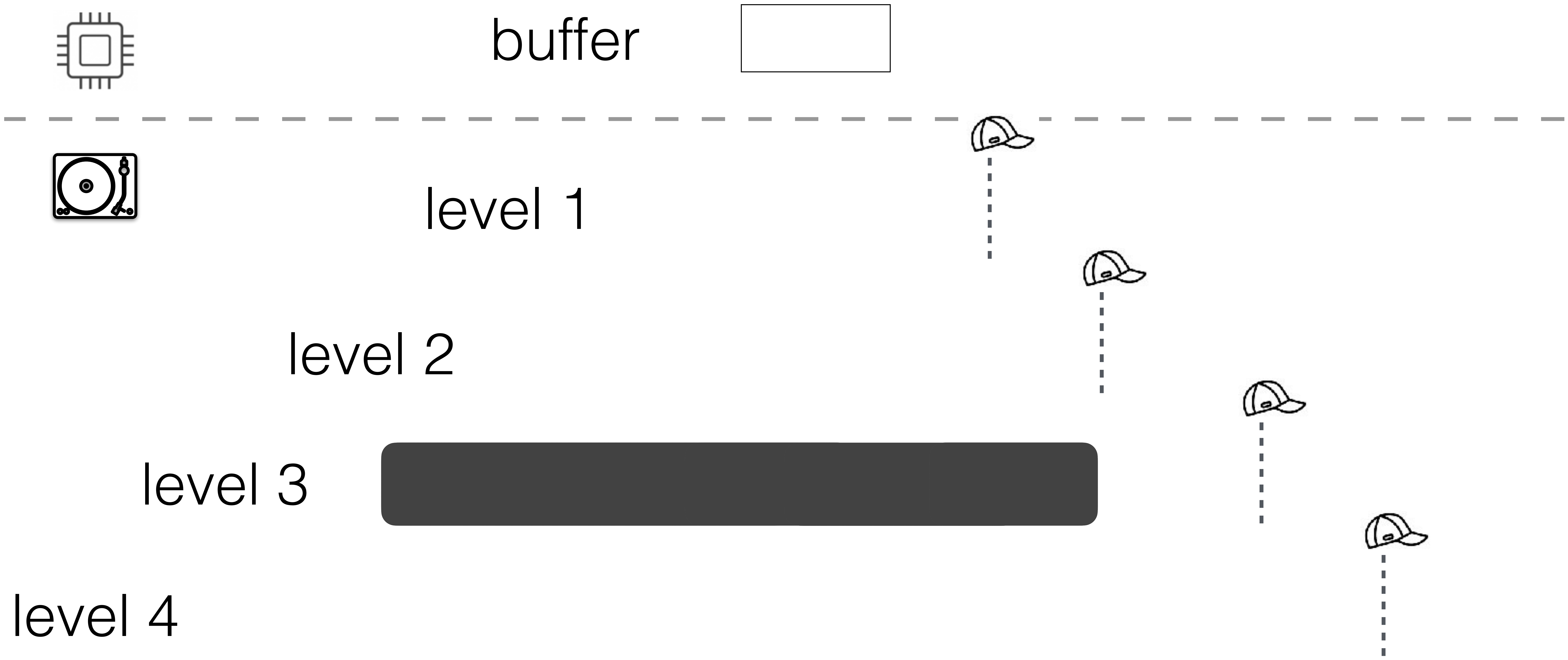

buffer
level 1
level 2
level 3
level 4


buffer
level 1
level 2
level 3
level 4

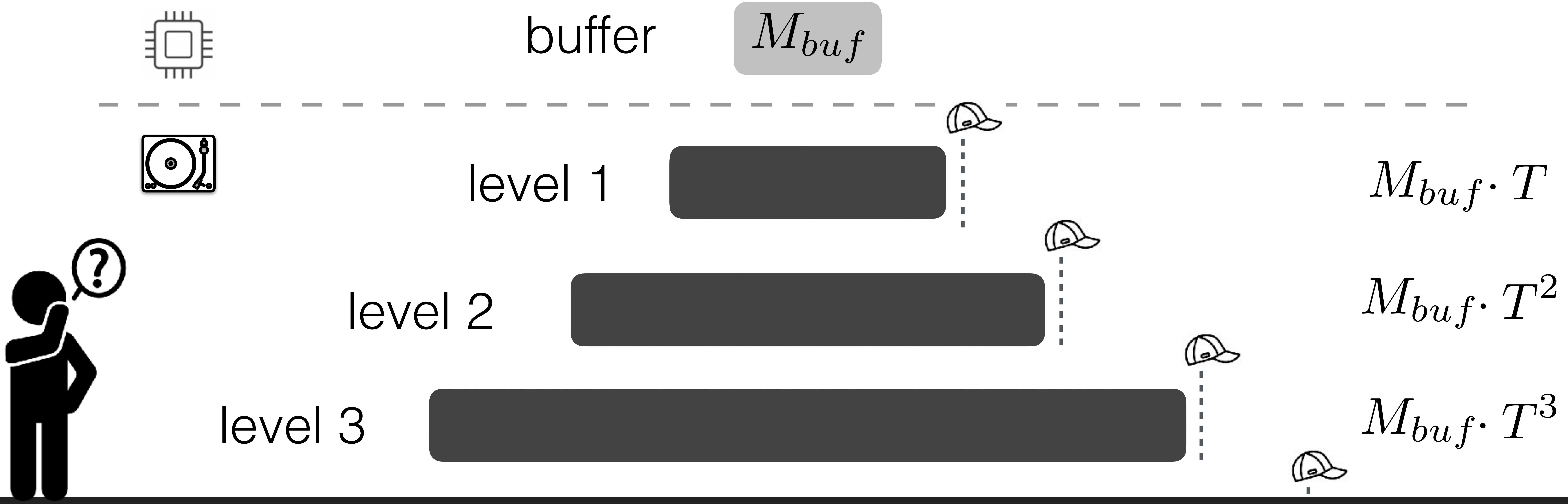

buffer
level 1
level 2
level 3
level 4


buffer
level 1
level 2
level 3
level 4


buffer
level 1
level 2
level 3
level 4


buffer
level 1
level 2
level 3
level 4

M_{buf}: buffer memory
T: size ratio

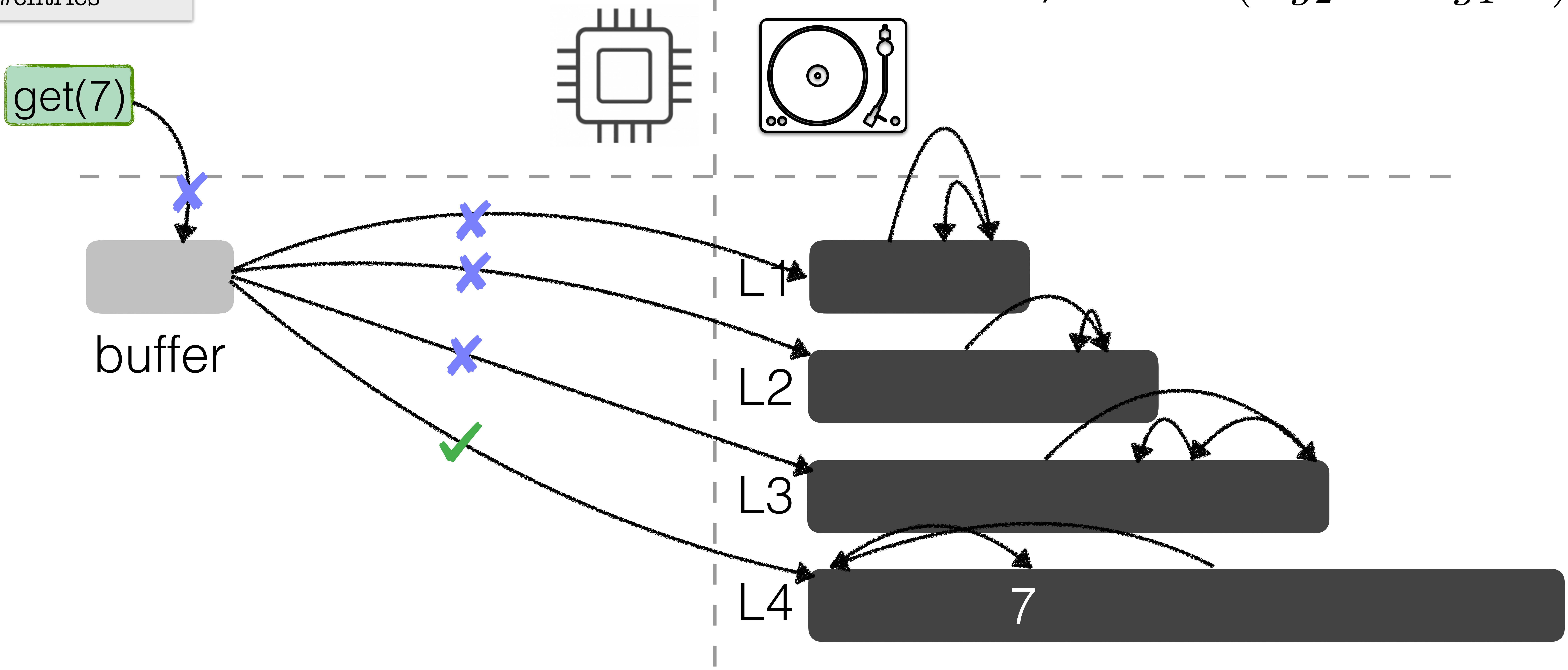


How about queries?

P: pages in buffer
B: entries/page
L: #levels
T: size ratio
N: #entries

Cost analysis

w/o F&I: $\mathcal{O}(log_2 N \cdot log_T N)$

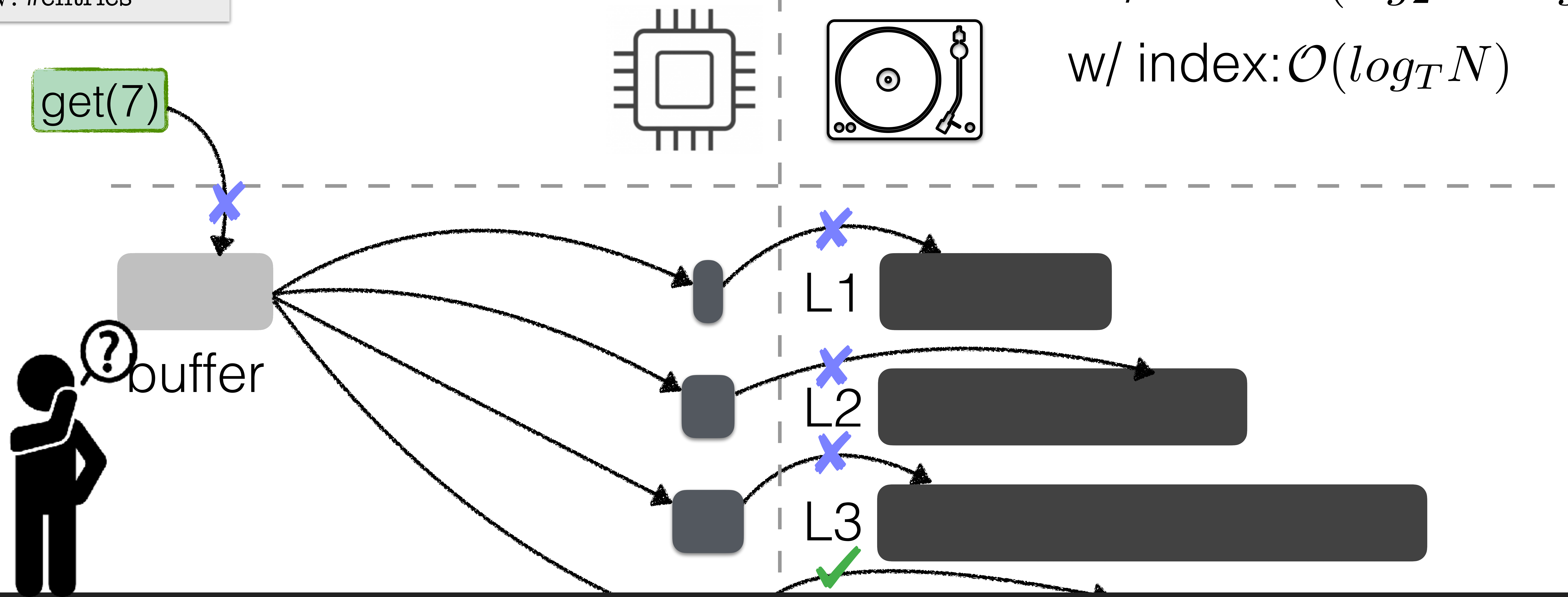


P: pages in buffer
B: entries/page
L: #levels
T: size ratio
N: #entries

Cost analysis

w/o F&I: $\mathcal{O}(log_2 N \cdot log_T N)$

w/ index: $\mathcal{O}(log_T N)$



Can we do better?

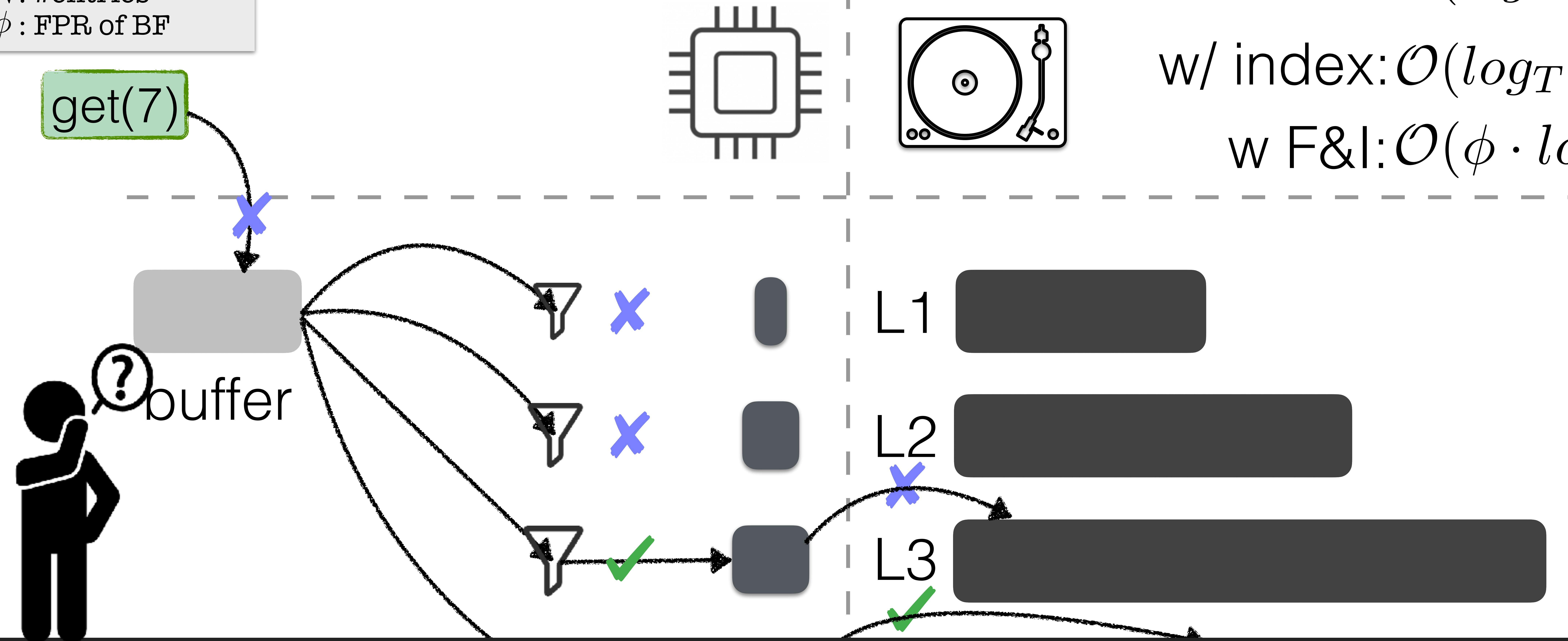
P: pages in buffer
B: entries/page
L: #levels
T: size ratio
N: #entries
ϕ : FPR of BF

Cost analysis

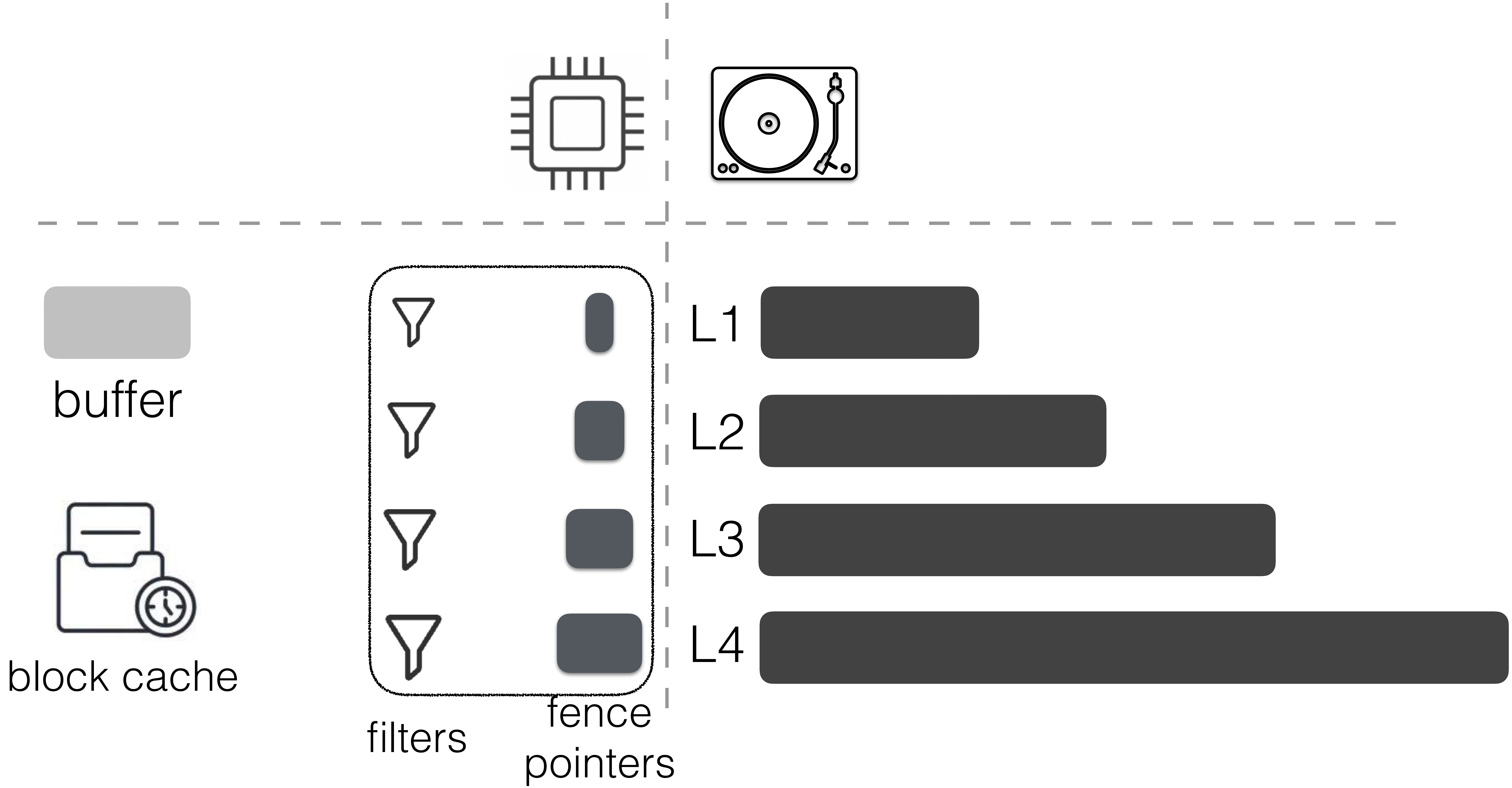
w/o F&I: $\mathcal{O}(log_2 N \cdot log_T N)$

w/ index: $\mathcal{O}(log_T N)$

w F&I: $\mathcal{O}(\phi \cdot log_T N)$

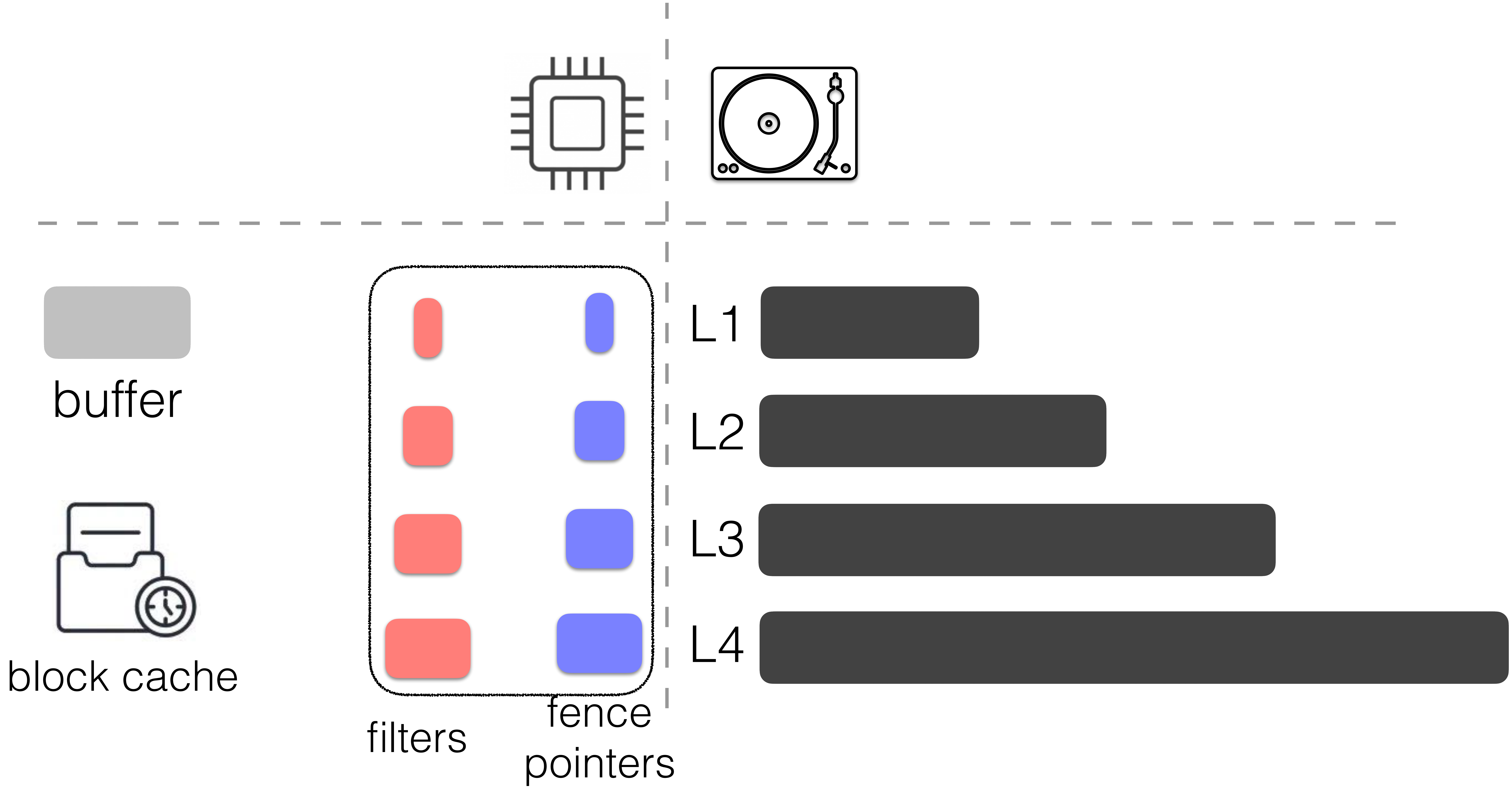


How to manage memory?

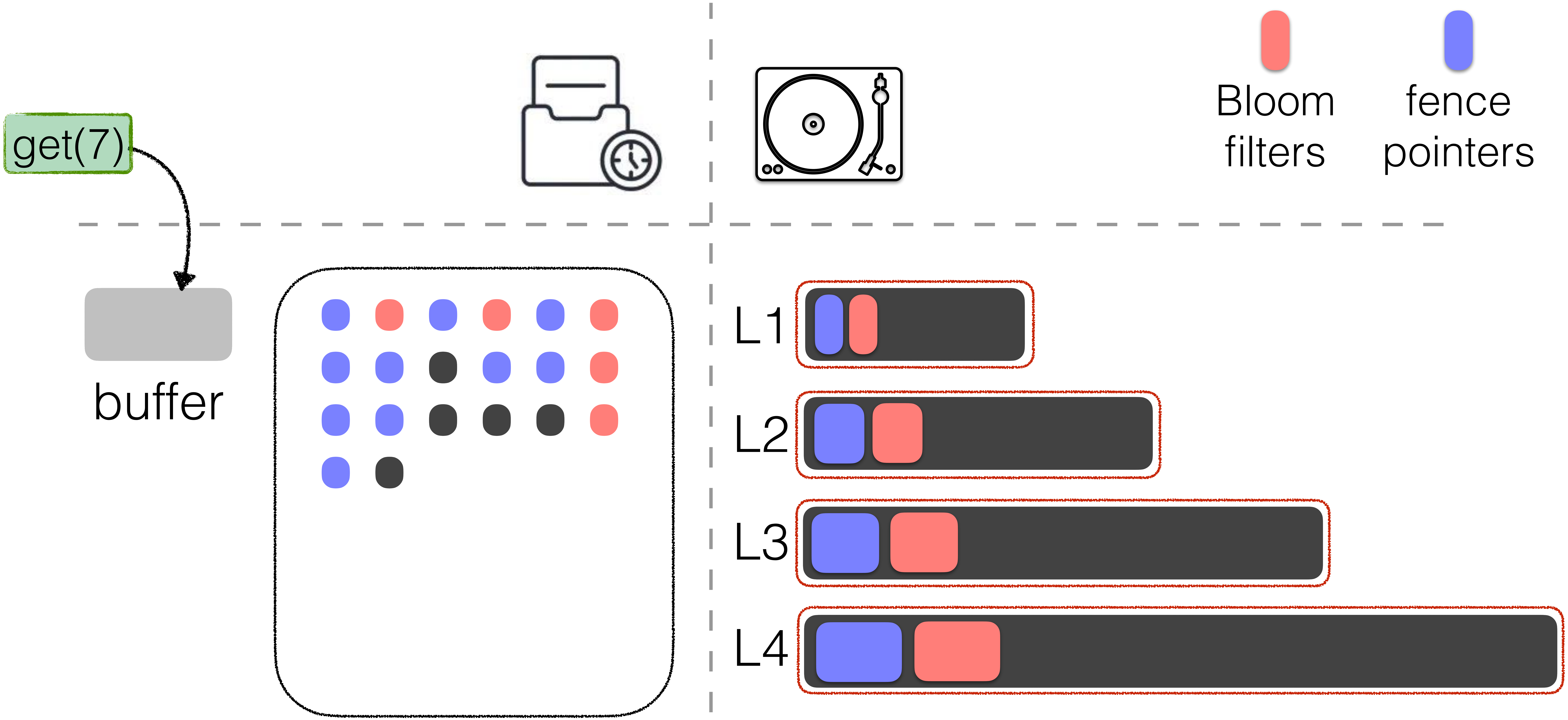


buffer
block cache
filters
fence pointers
L1
L2
L3
L4

Block Cache



Block Cache



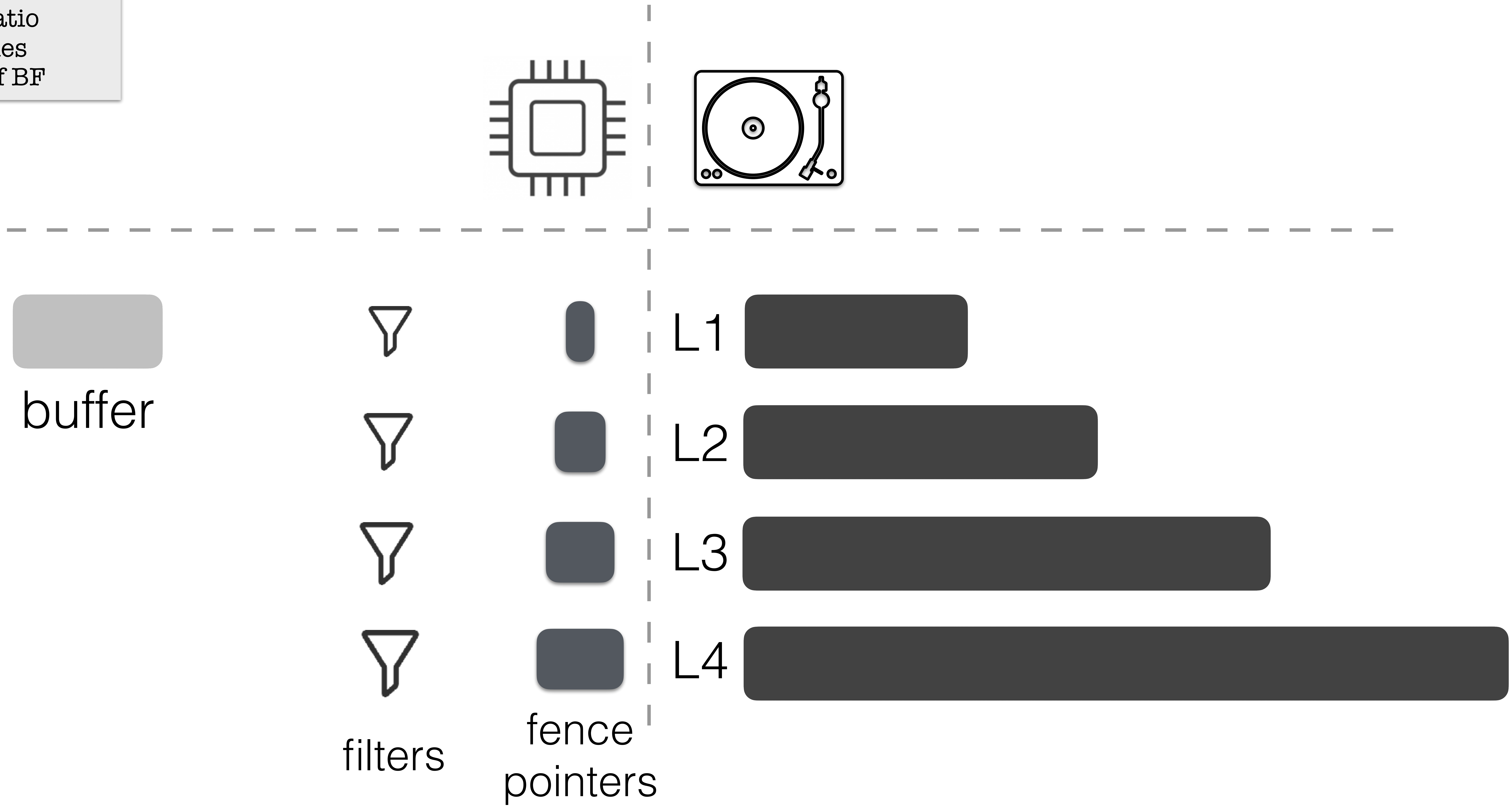
Block Cache

What about range queries?

P: pages in buffer
B: entries/page
L: #levels
T: size ratio
N: #entries
ϕ : FPR of BF

s : selectivity LRQ

Range Queries



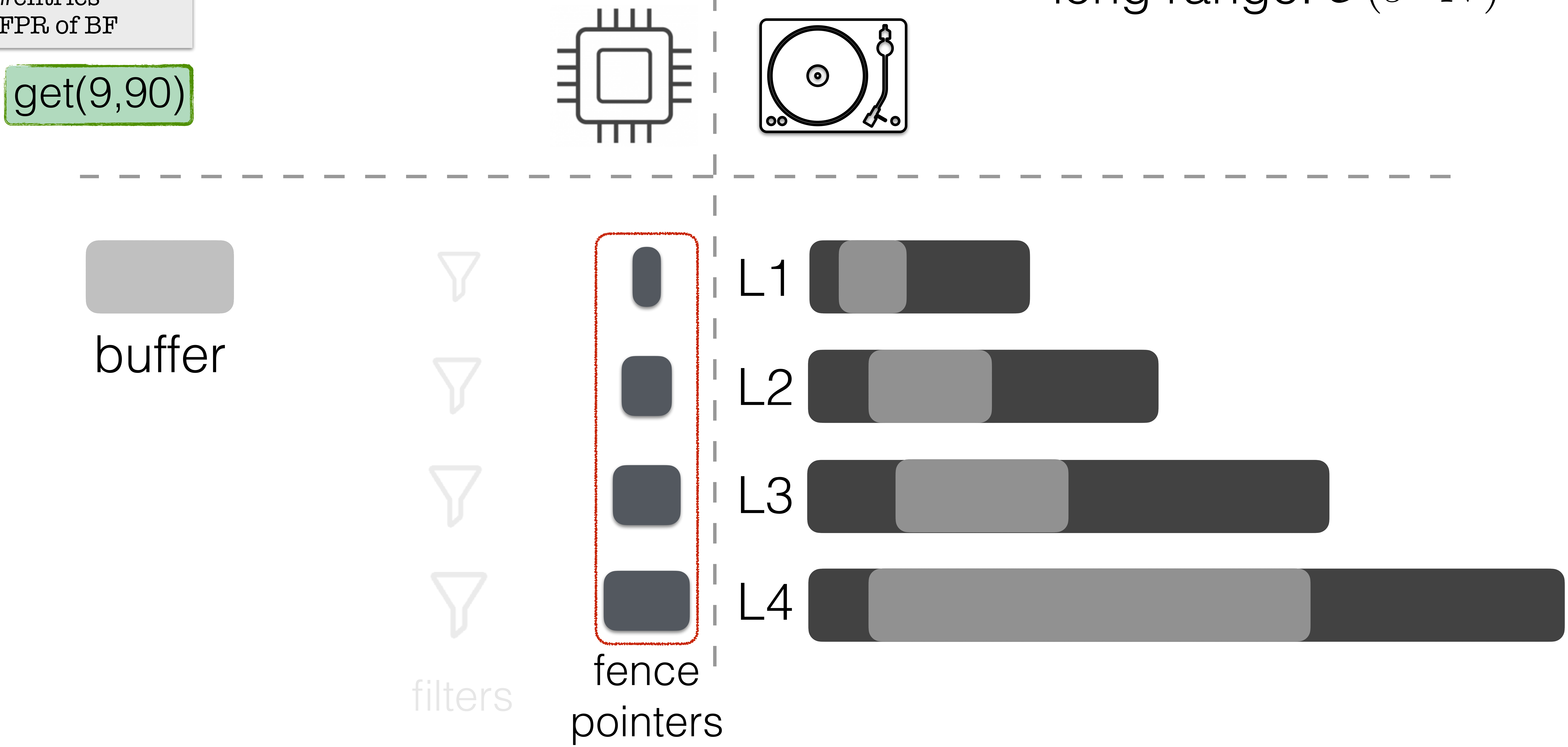
P: pages in buffer
B: entries/page
L: #levels
T: size ratio
N: #entries
ϕ : FPR of BF

s : selectivity LRQ

Range Queries

Cost analysis

long range: $\mathcal{O}(s \cdot N)$



P: pages in buffer
B: entries/page
L: #levels
T: size ratio
N: #entries
ϕ : FPR of BF

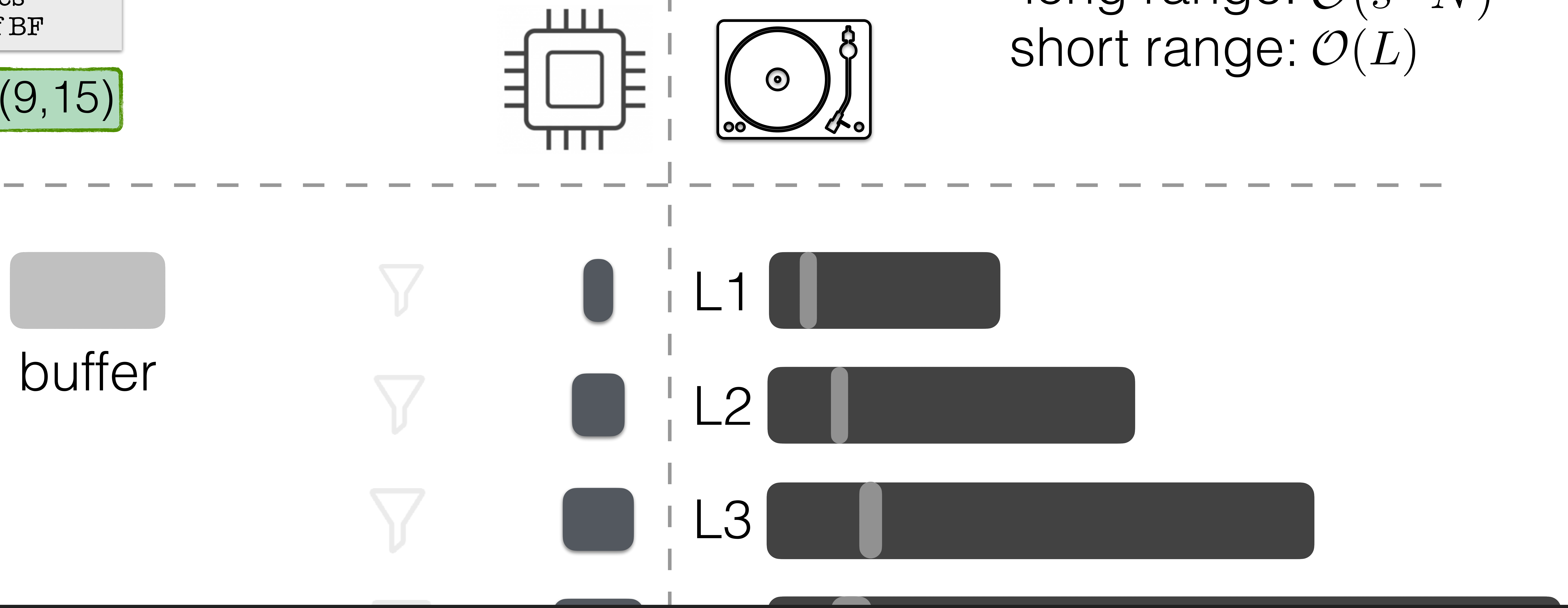
s : selectivity SRQ

Range Queries

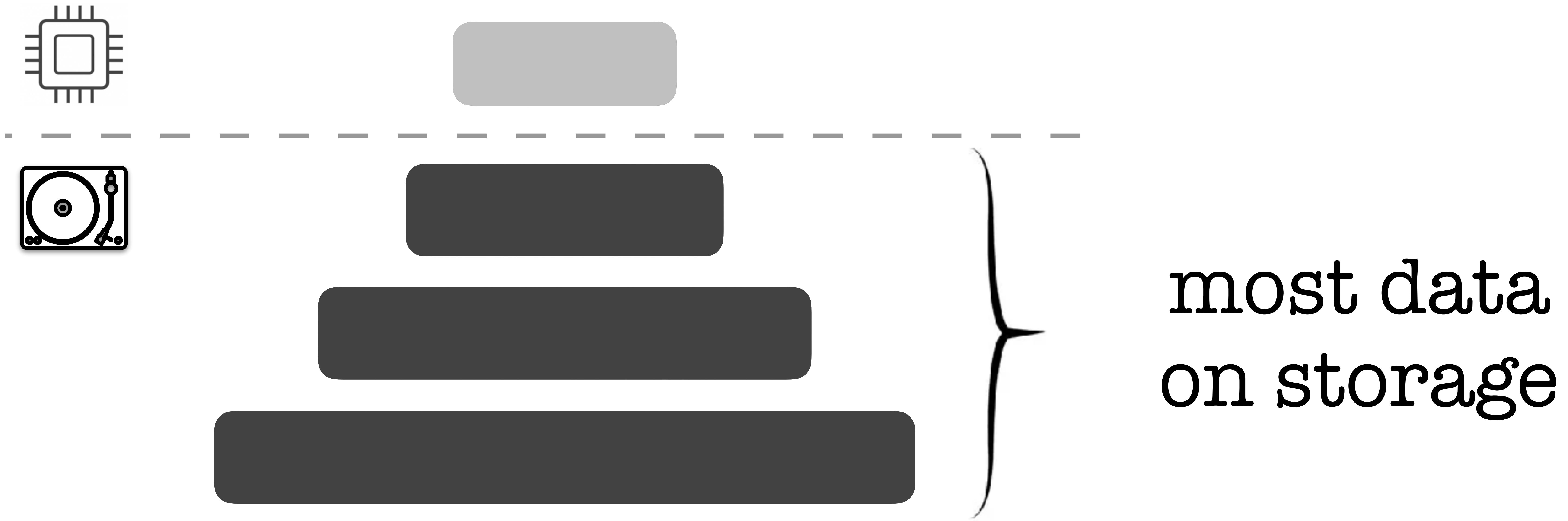
Cost analysis

long range: $\mathcal{O}(s \cdot N)$
short range: $\mathcal{O}(L)$

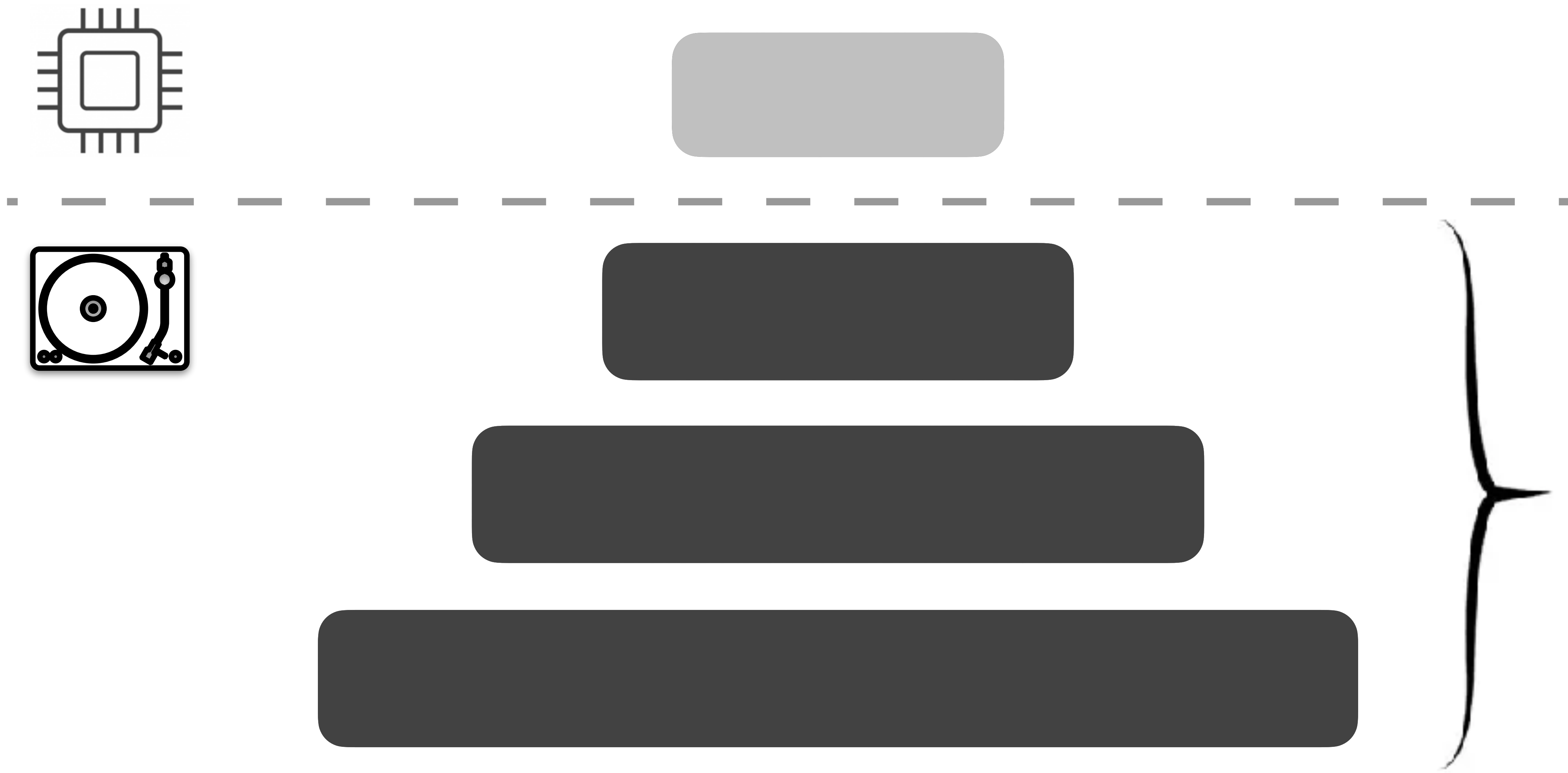
get(9,15)



More on LSM Reads in Part 2.


most data
on storage

L: #levels
T: size ratio

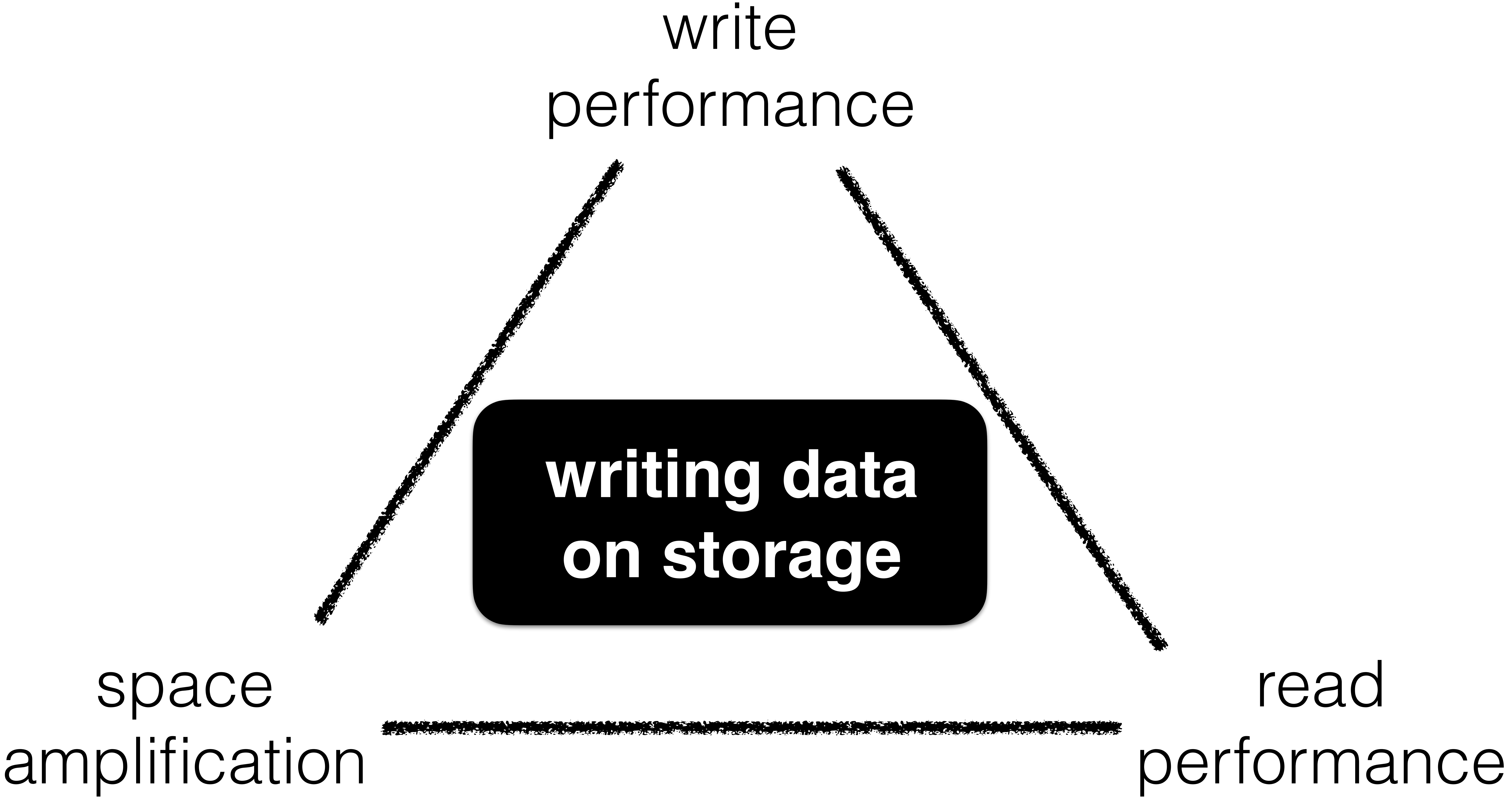


most data
on storage

if $T = 10$ & $L = 4$

99.9% on storage

Performance **Tradeoff**



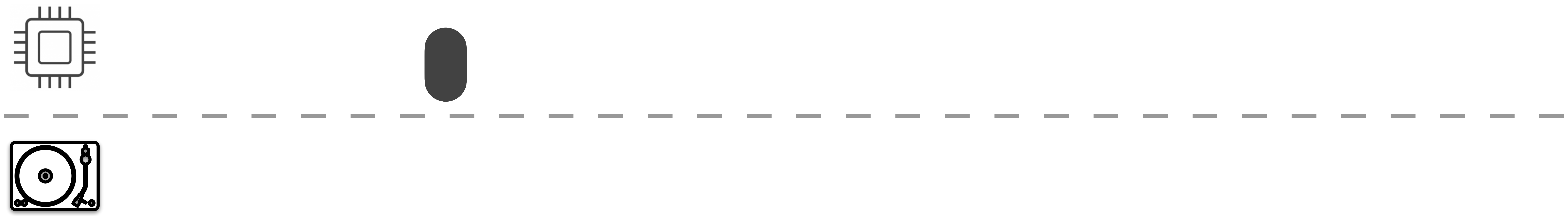
Data **Layout**

Classical LSM design: **leveling**

[eager merging]

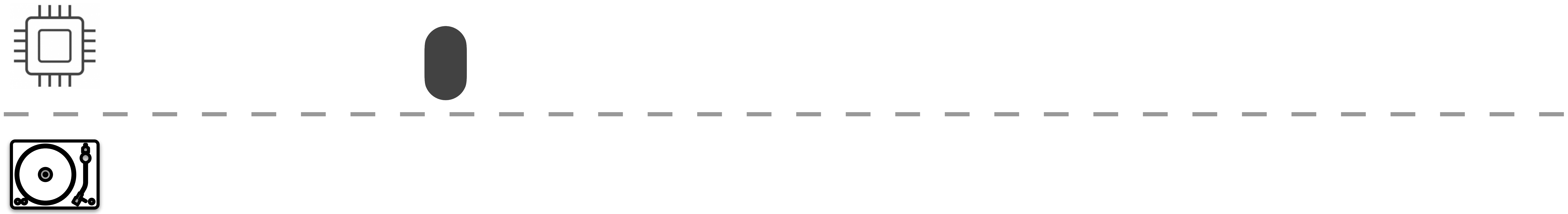
Data **Layout**

leveling [eager]



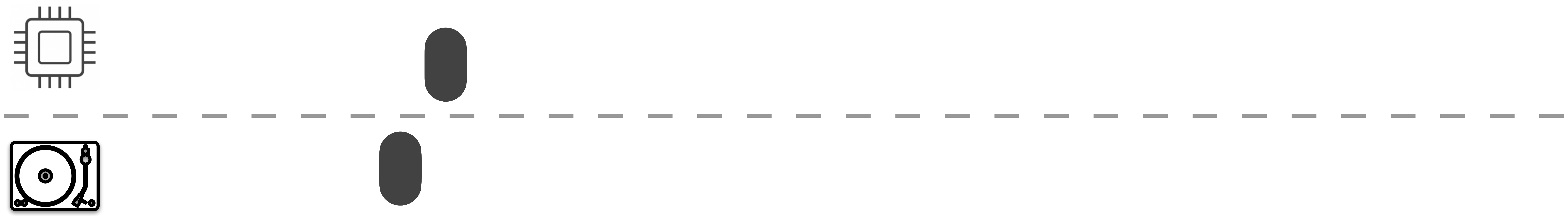
Data **Layout**

leveling [eager]



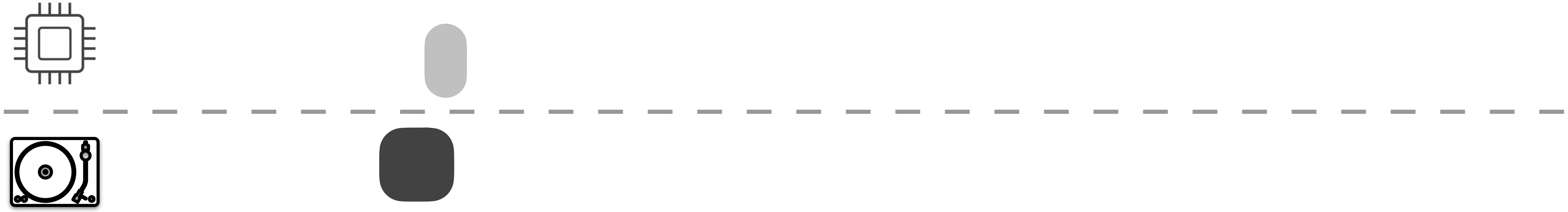
Data **Layout**

leveling [eager]



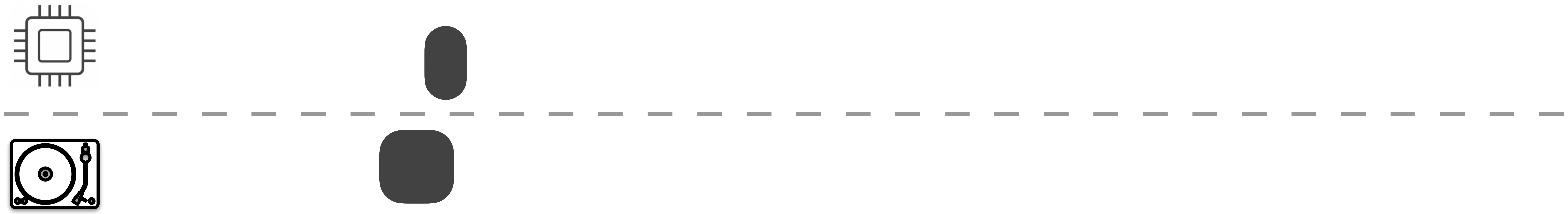
Data **Layout**

leveling [eager]



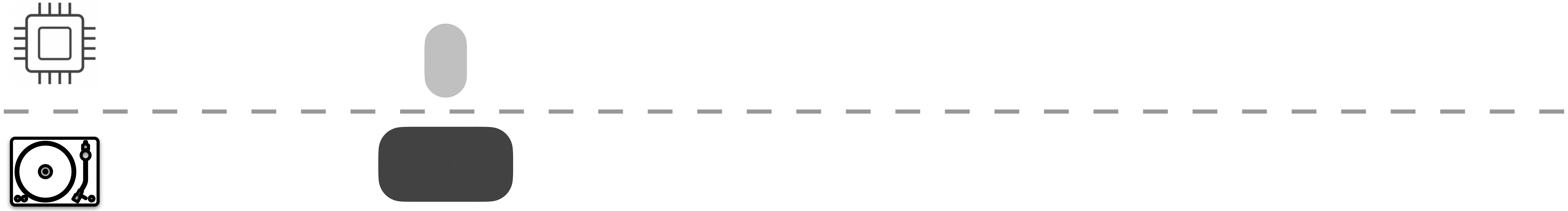
Data **Layout**

leveling [eager]



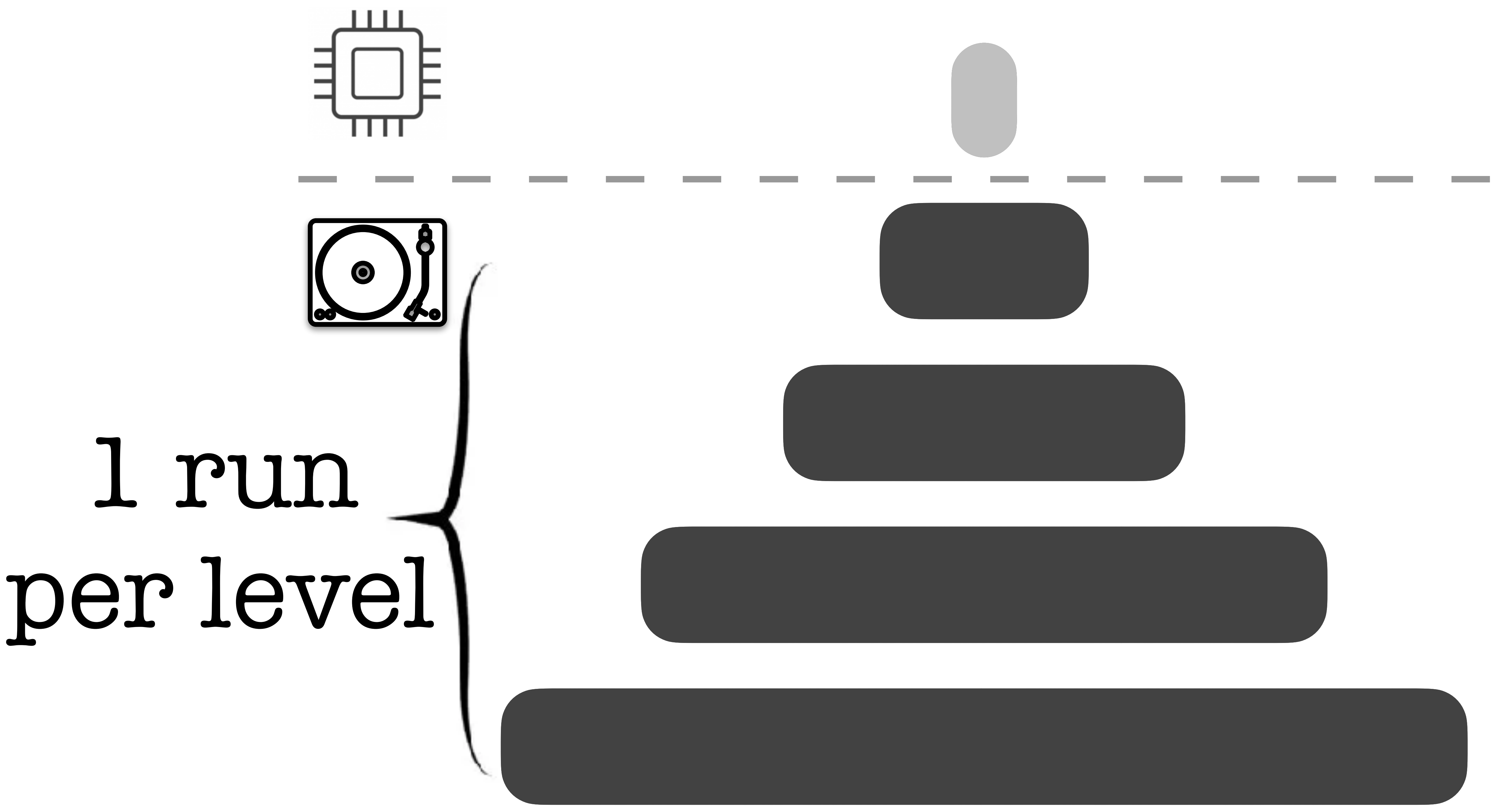
Data **Layout**

leveling [eager]



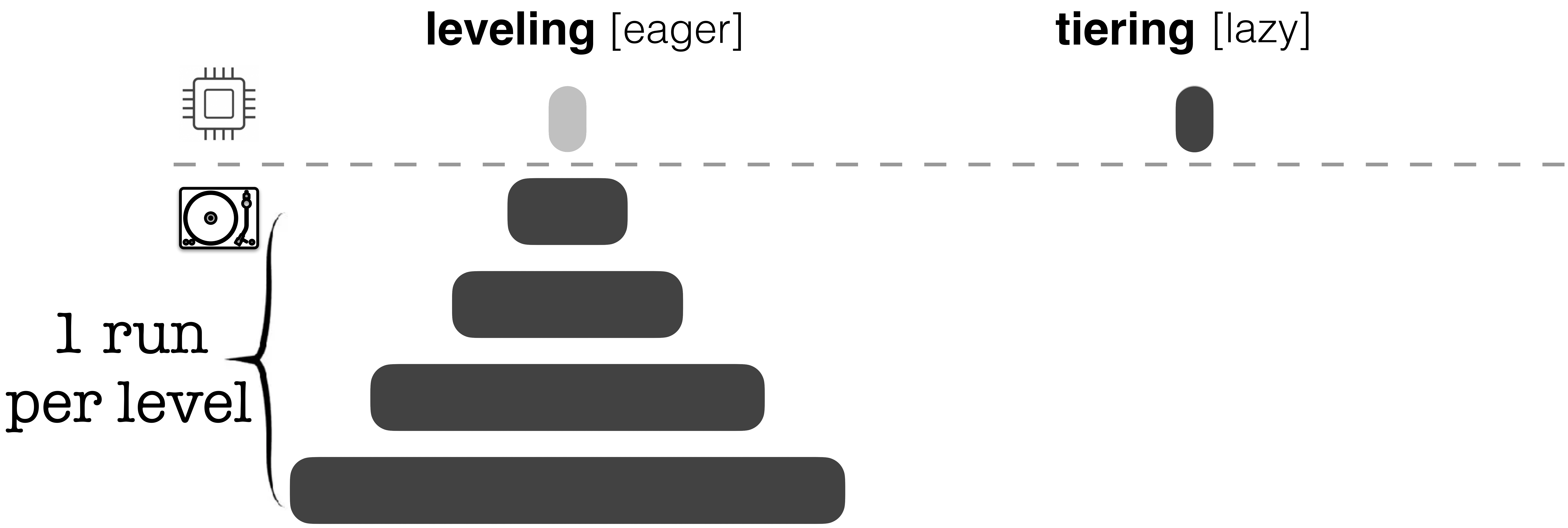
Data **Layout**

leveling [eager]



- good read performance
- good space amplification
- high write amplification

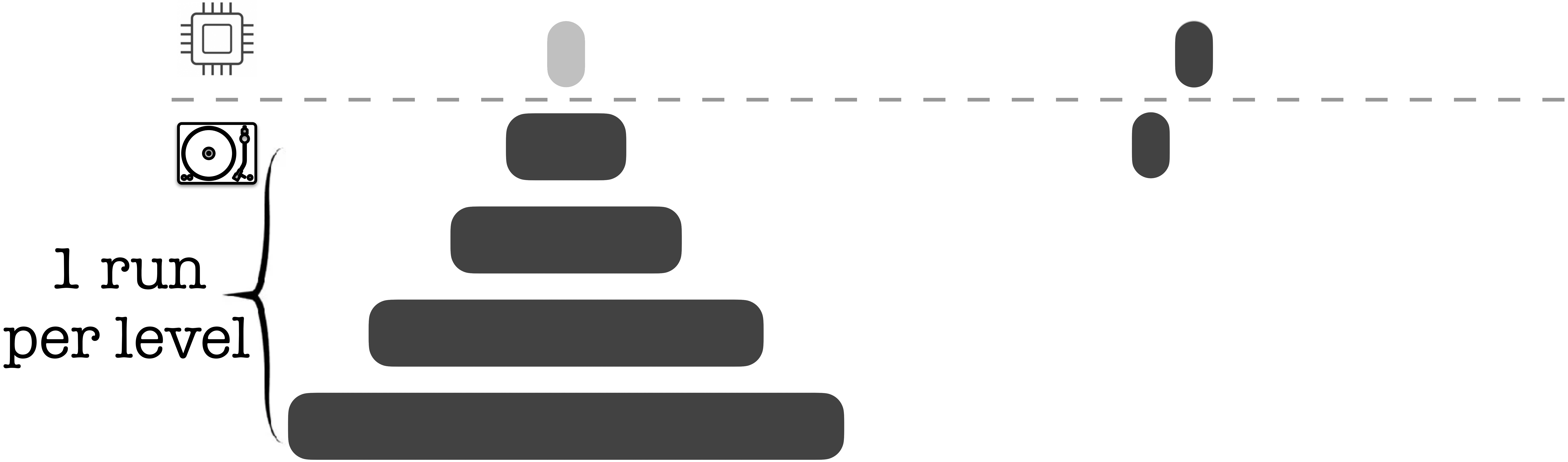
Data **Layout**



- good read performance
- good space amplification
- high write amplification

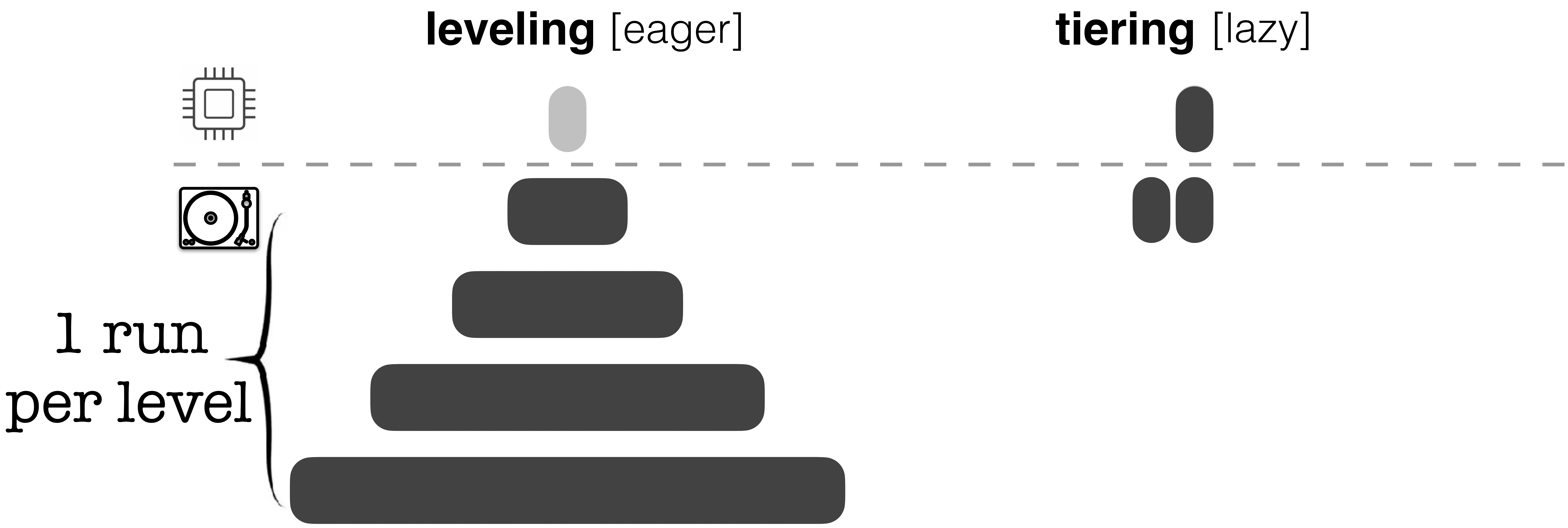
Data **Layout**

leveling [eager] **tiering** [lazy]



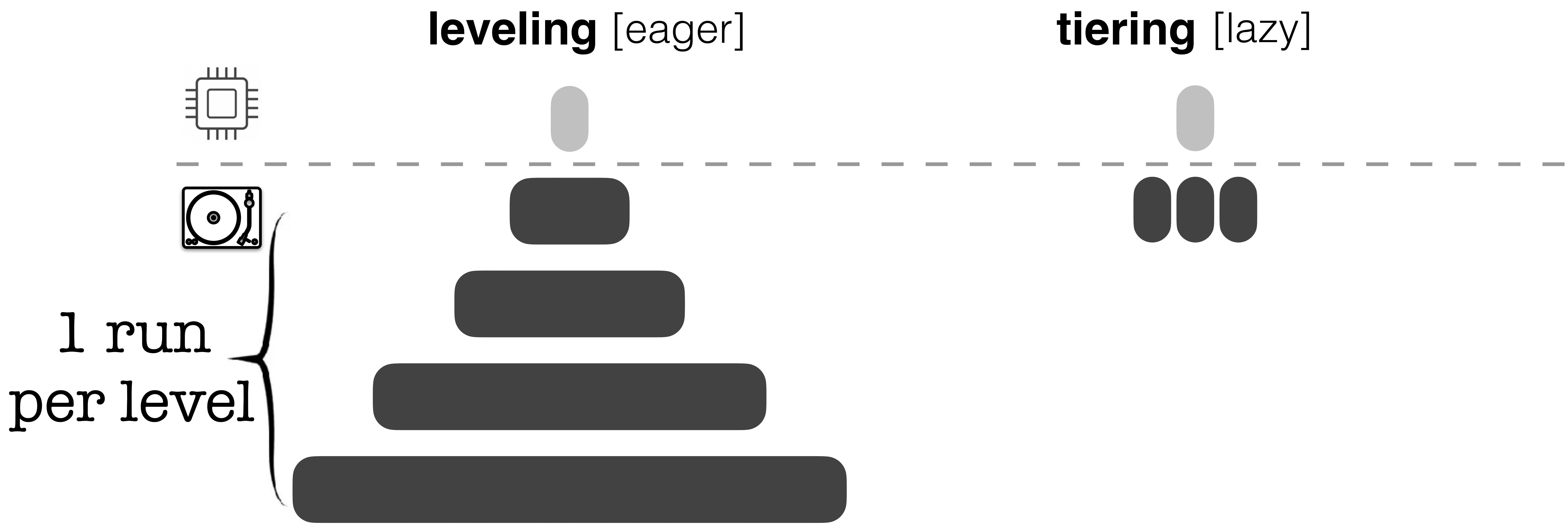
- good read performance
- good space amplification
- high write amplification

Data **Layout**



- good read performance
- good space amplification
- high write amplification

Data **Layout**



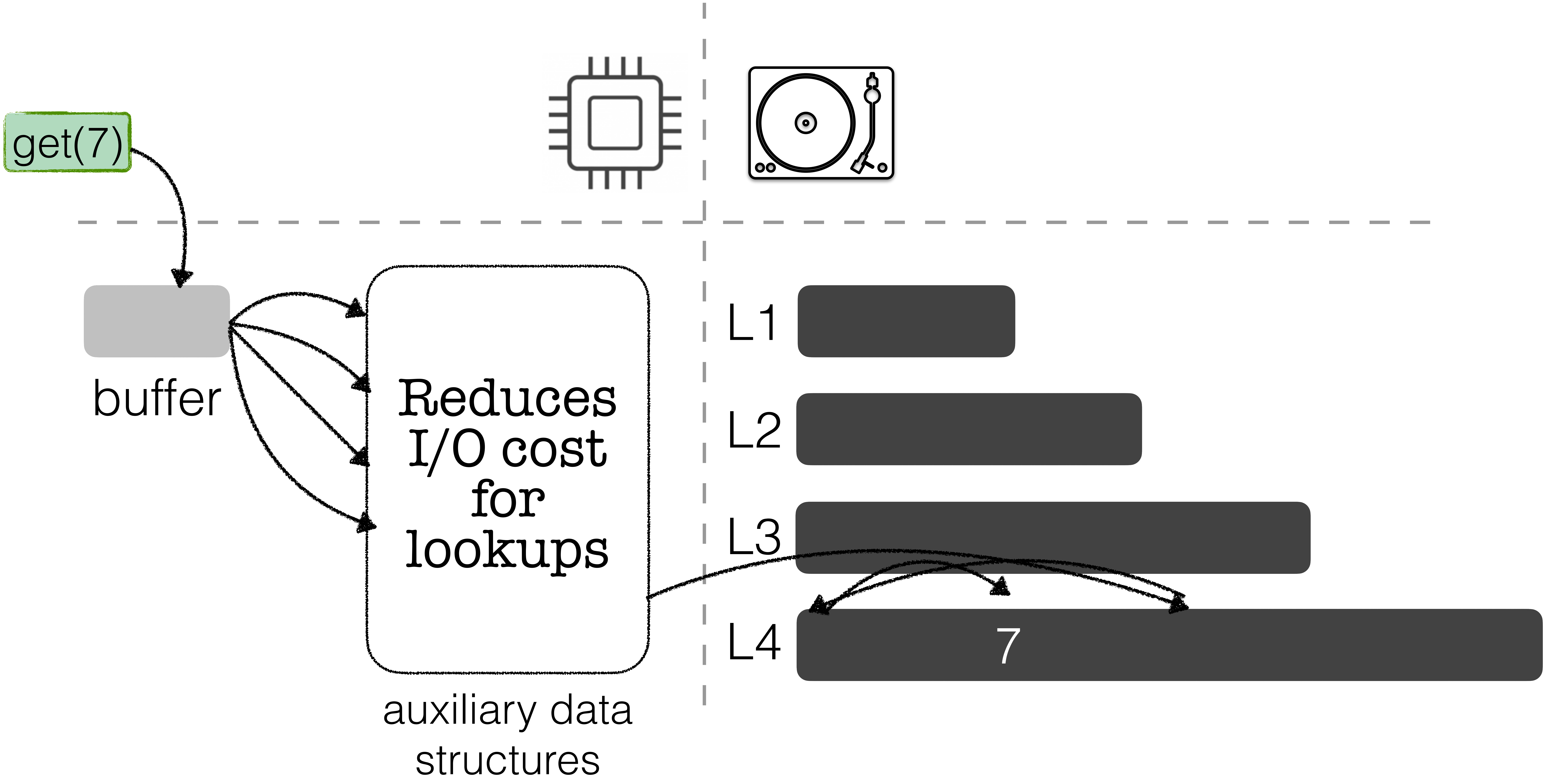
- good read performance
- good space amplification
- high write amplification

Data **Layout**

leveling [eager] | **tiering** [lazy]

- good read performance
- good space amplification
- high write amplification

- poor read performance
- poor space amplification
- good ingestion performance



get(7)
buffer
Reduces I/O cost for lookups
auxiliary data structures
L1
L2
L3
L4
7

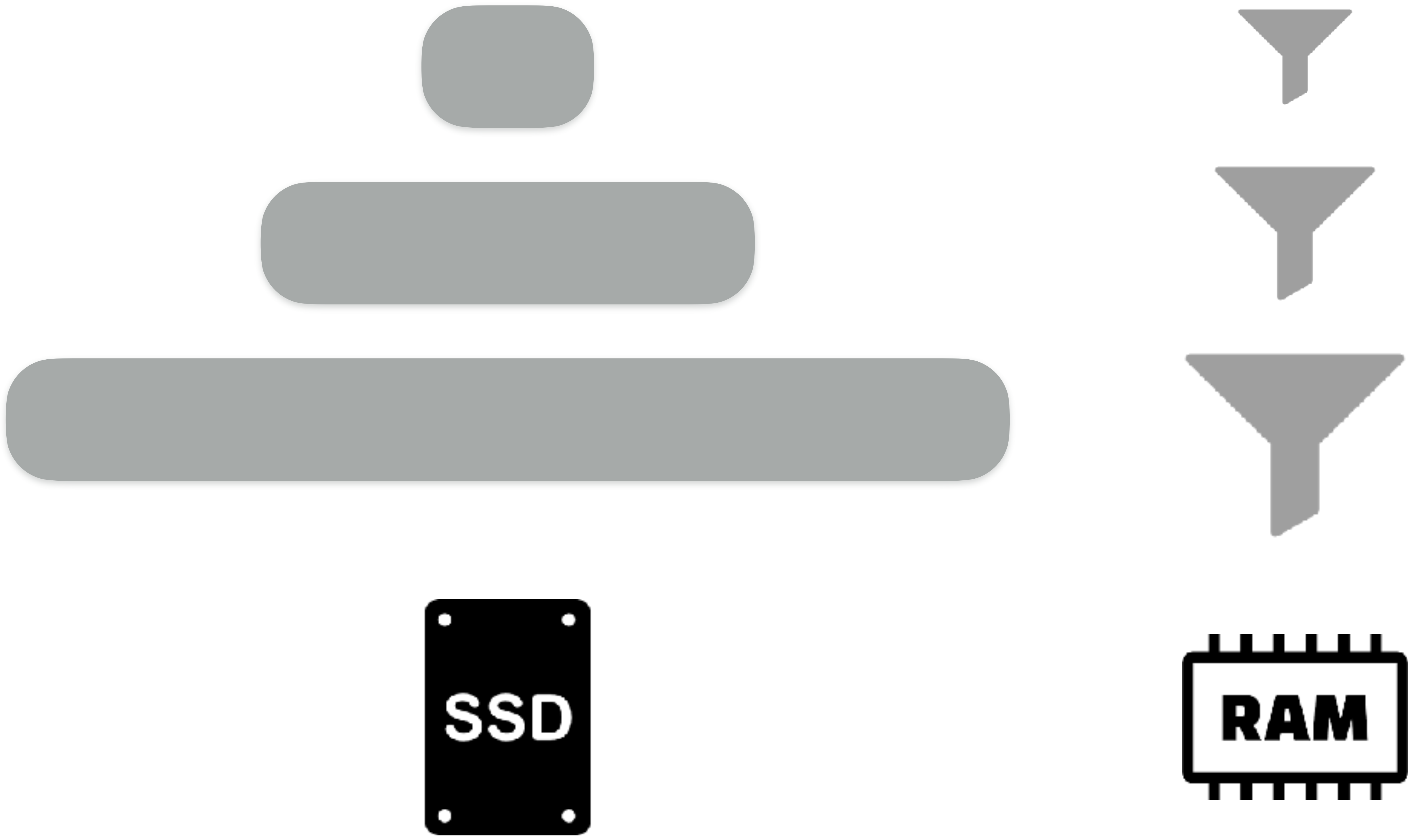
Outline

Part 1: **LSM Basics**

Part 2: **Read Optimizations in LSMs**

Part 3: **Navigating the LSM Design Space**

Filters to the Rescue



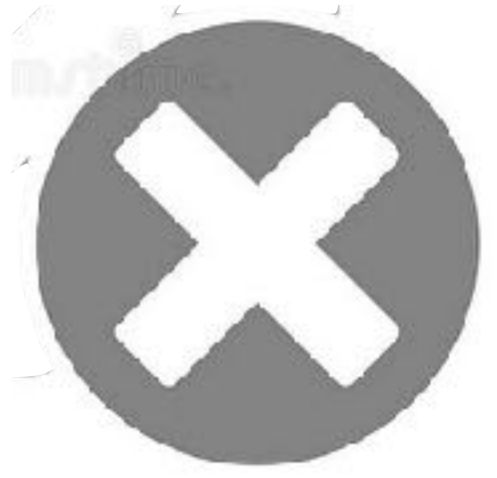
What is a filter

What is a filter

Answers set membership queries

What is a filter

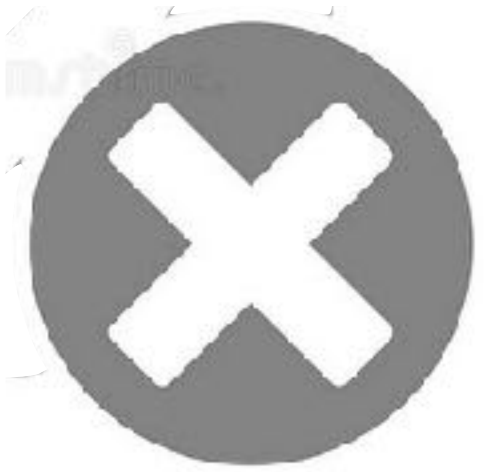
No false negatives



What is a filter

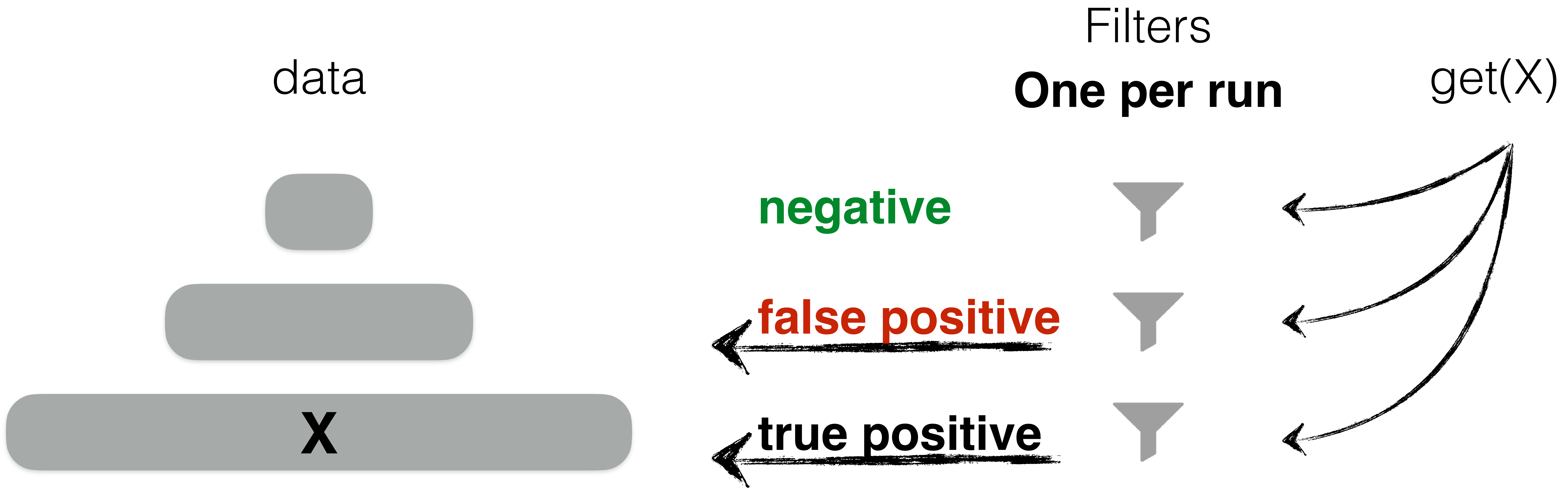
Does Q exist? →  → Set: X Y Z

No false negatives

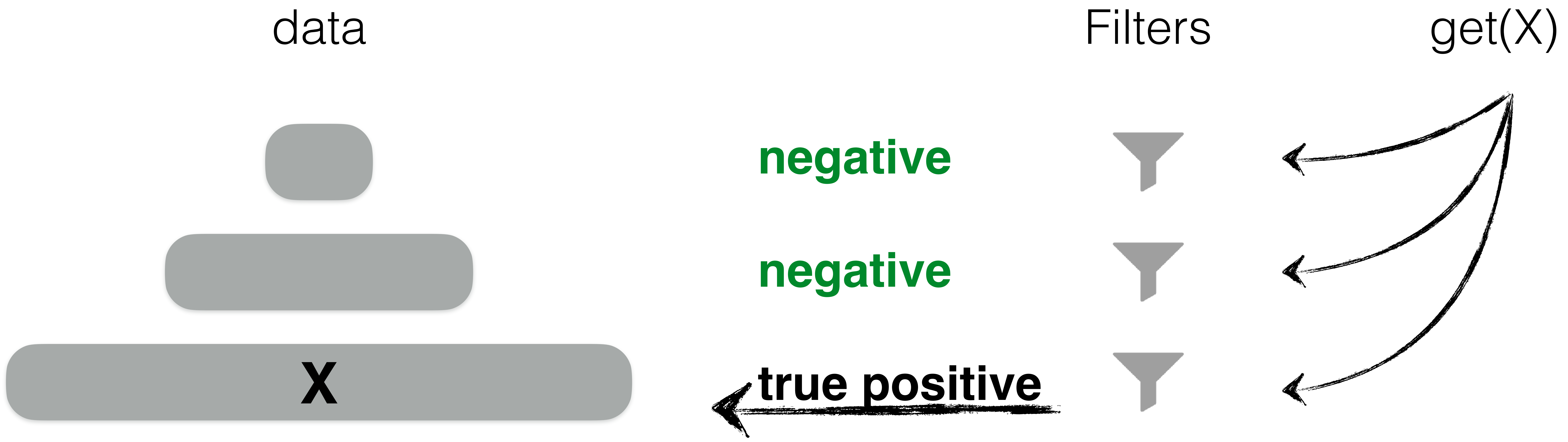


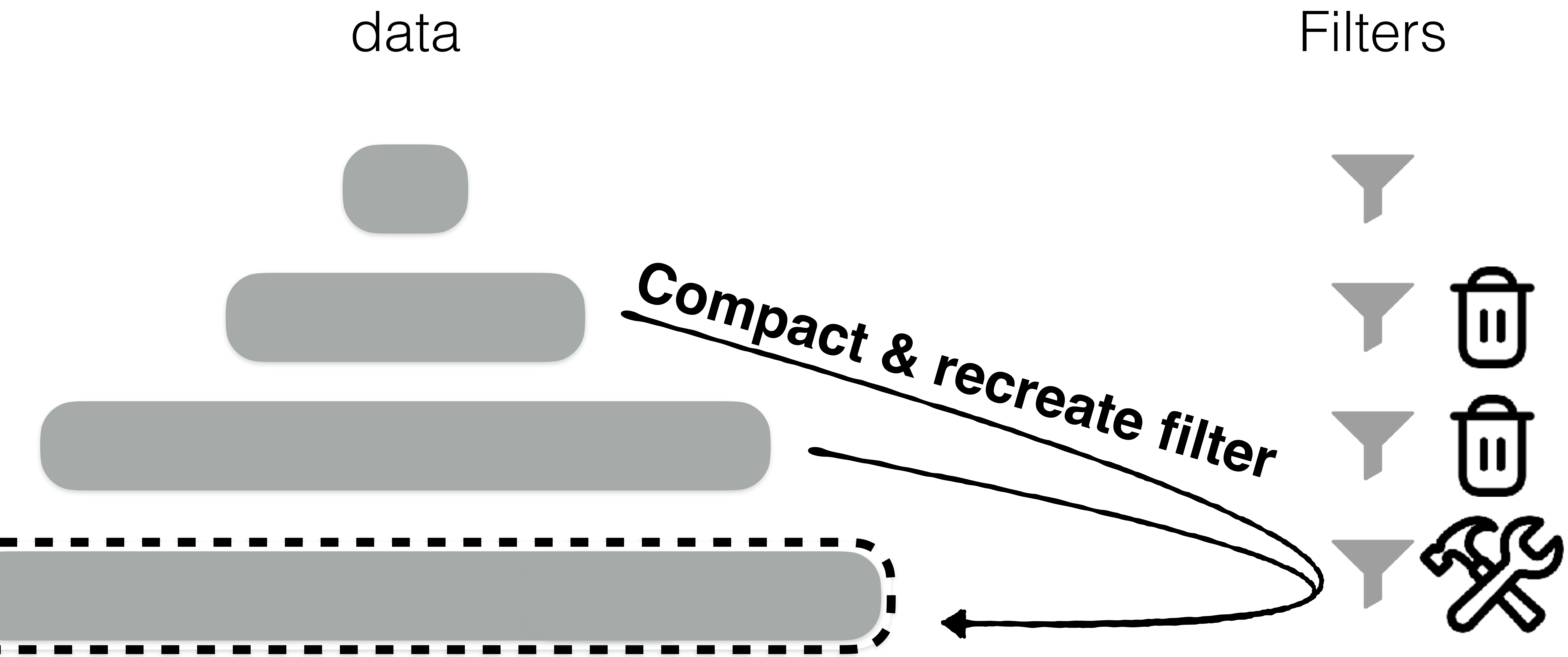
false positives with tunable probability


data
X
Filters
One per run
get(X)

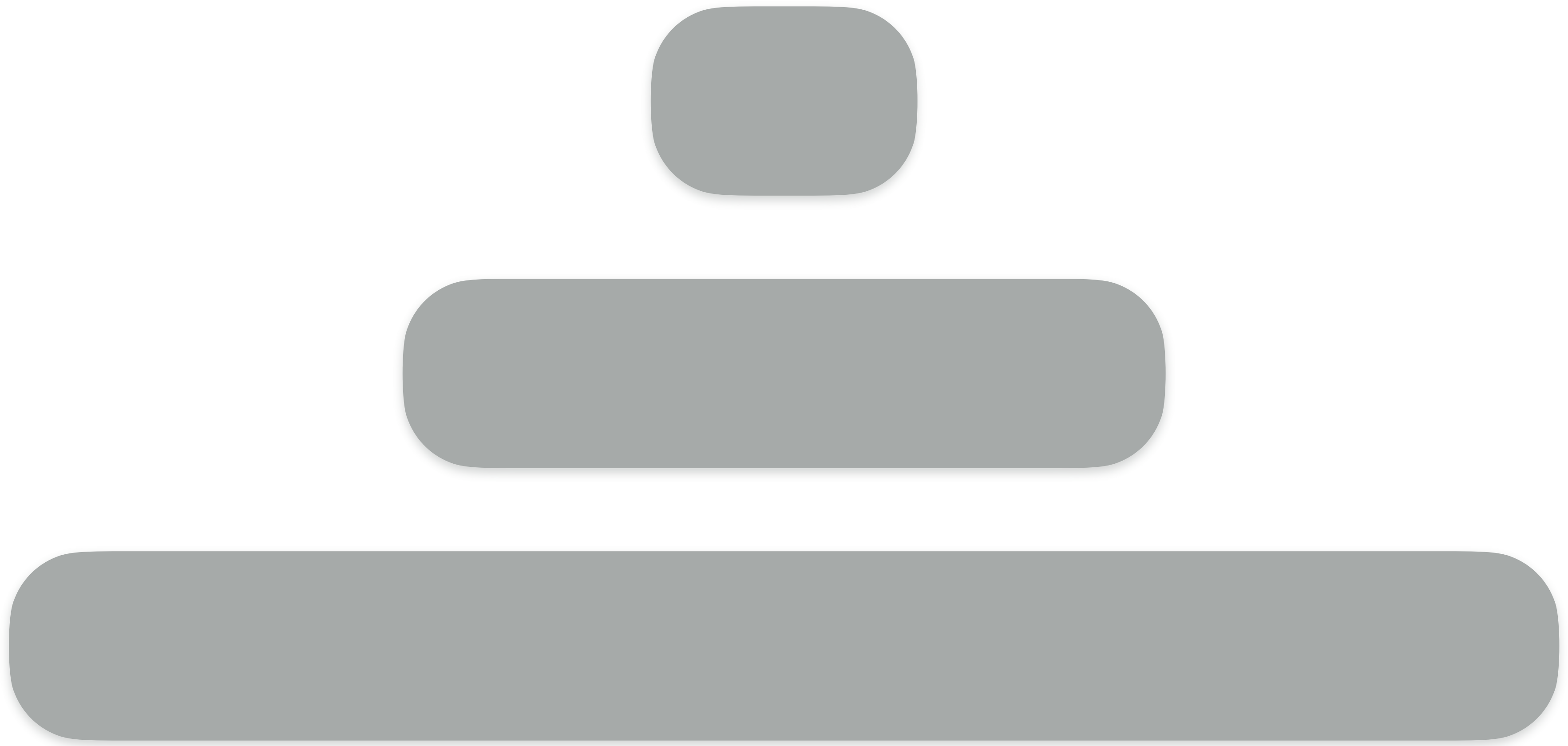

Filters
data
One per run
get(X)
negative
false positive
X
true positive

more memory → fewer false positives




data
Filters
Compact & recreate filter

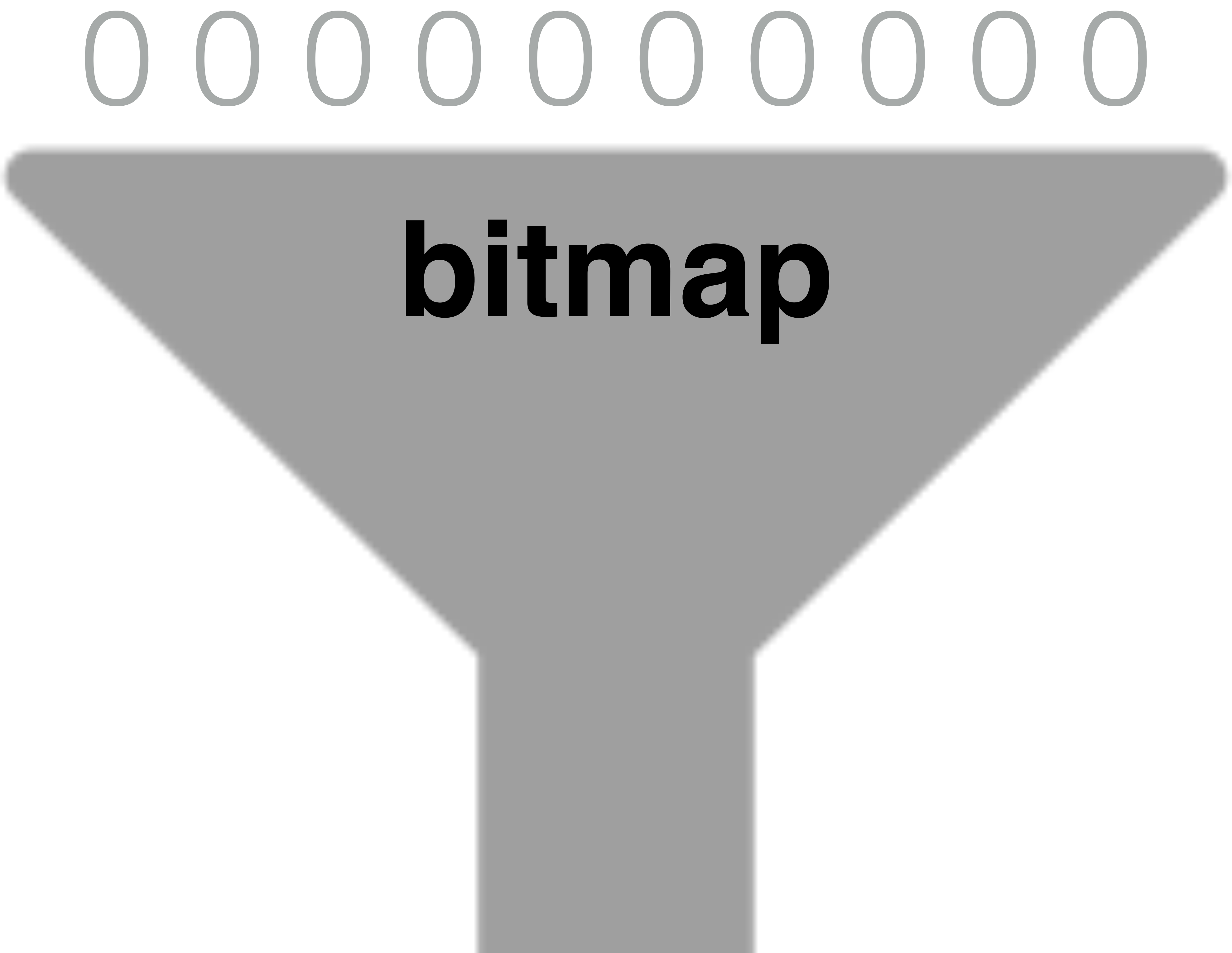
data



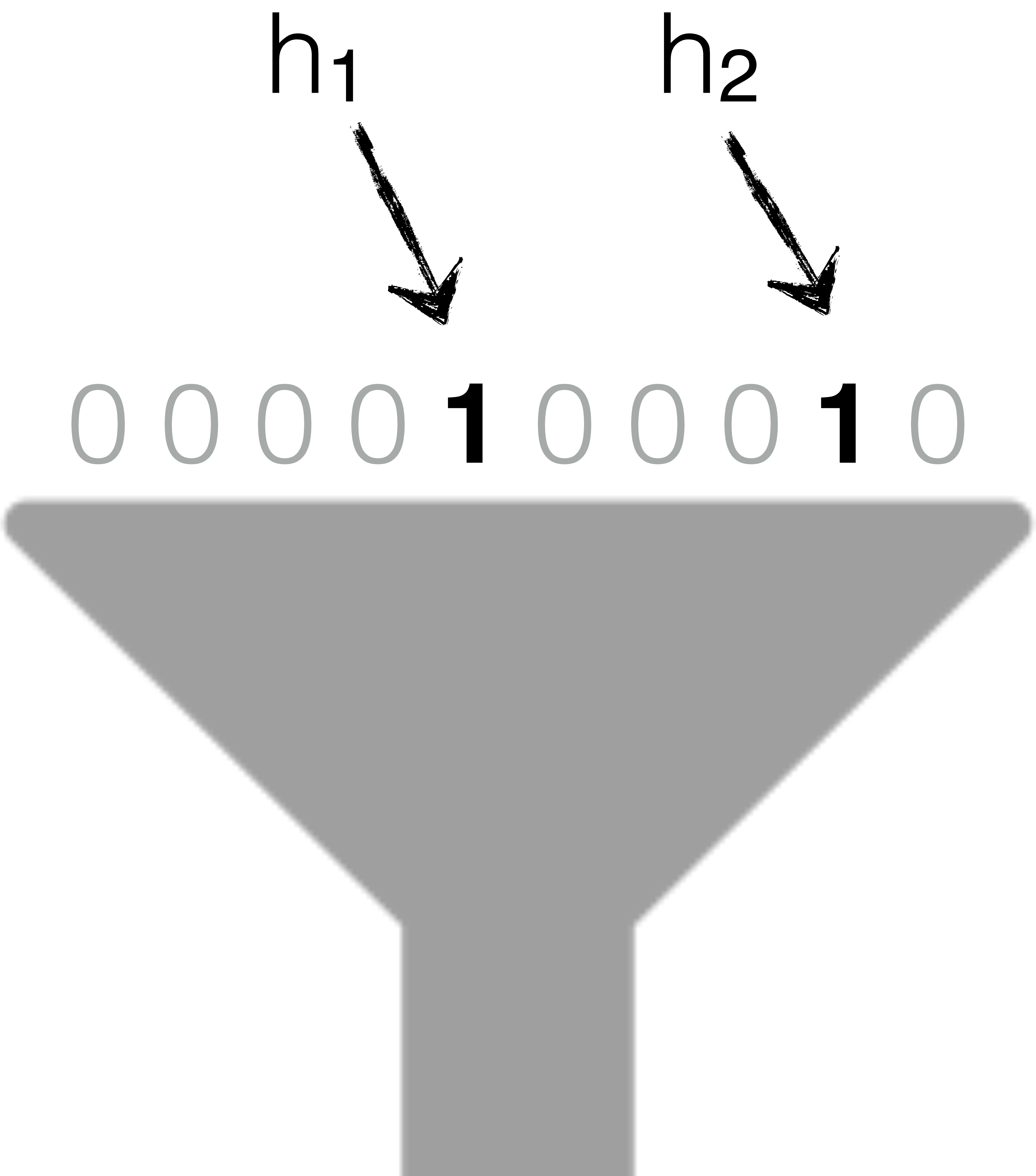
Bloom Filters

BloomCommunACM1970

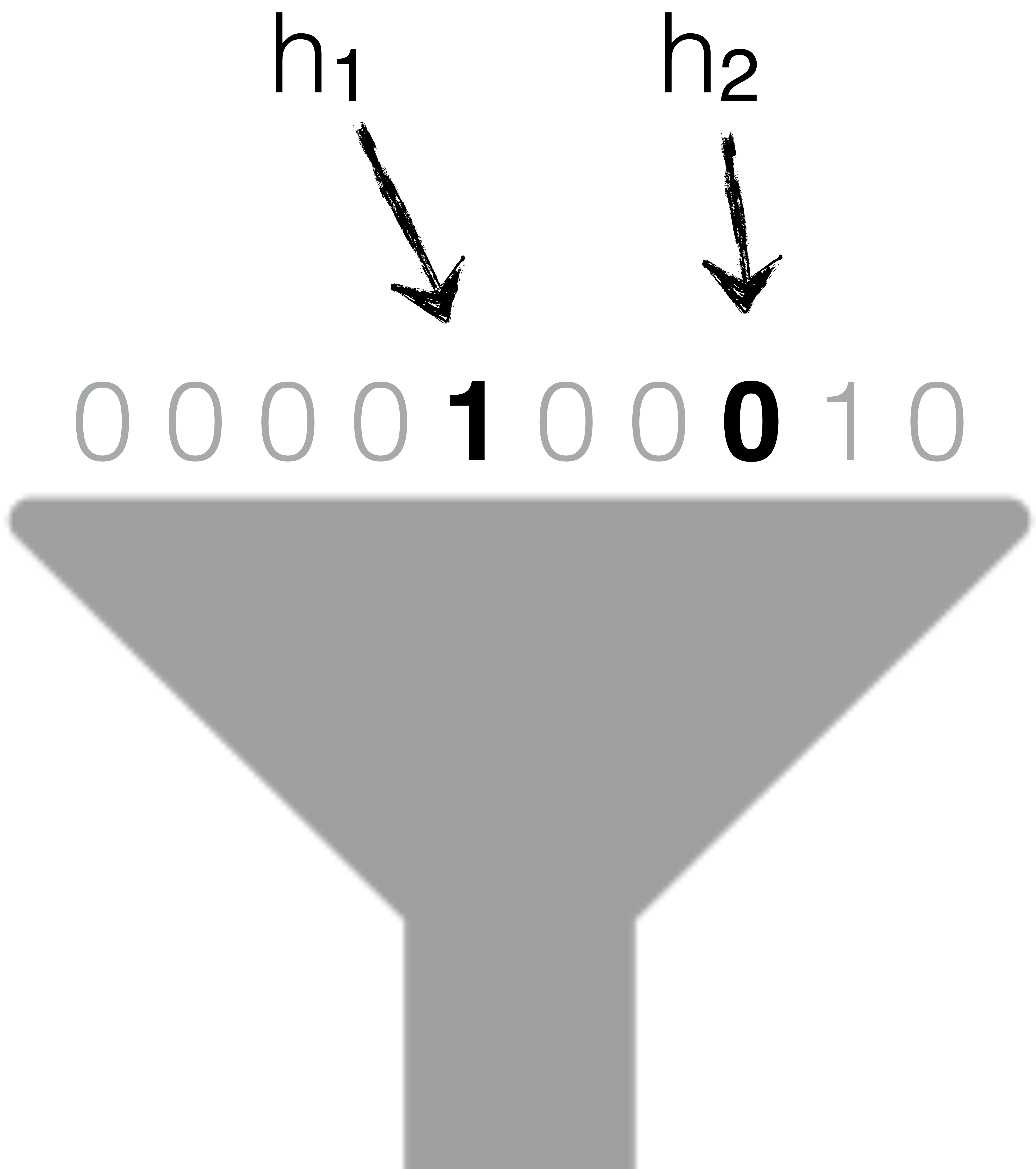
k hash functions



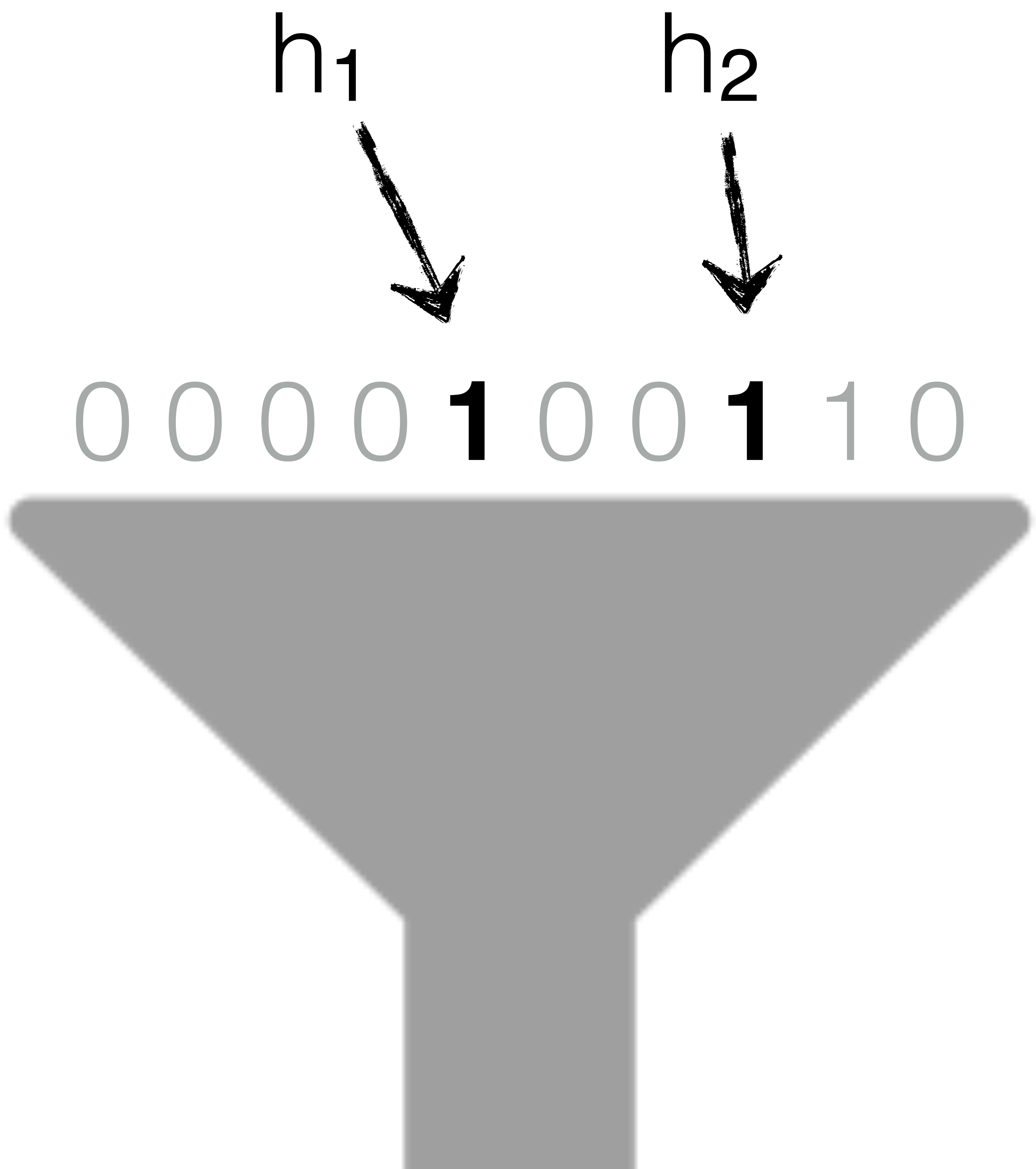
insert: Set from 0 to 1 or keep 1



negative lookup: at least one bit is zero

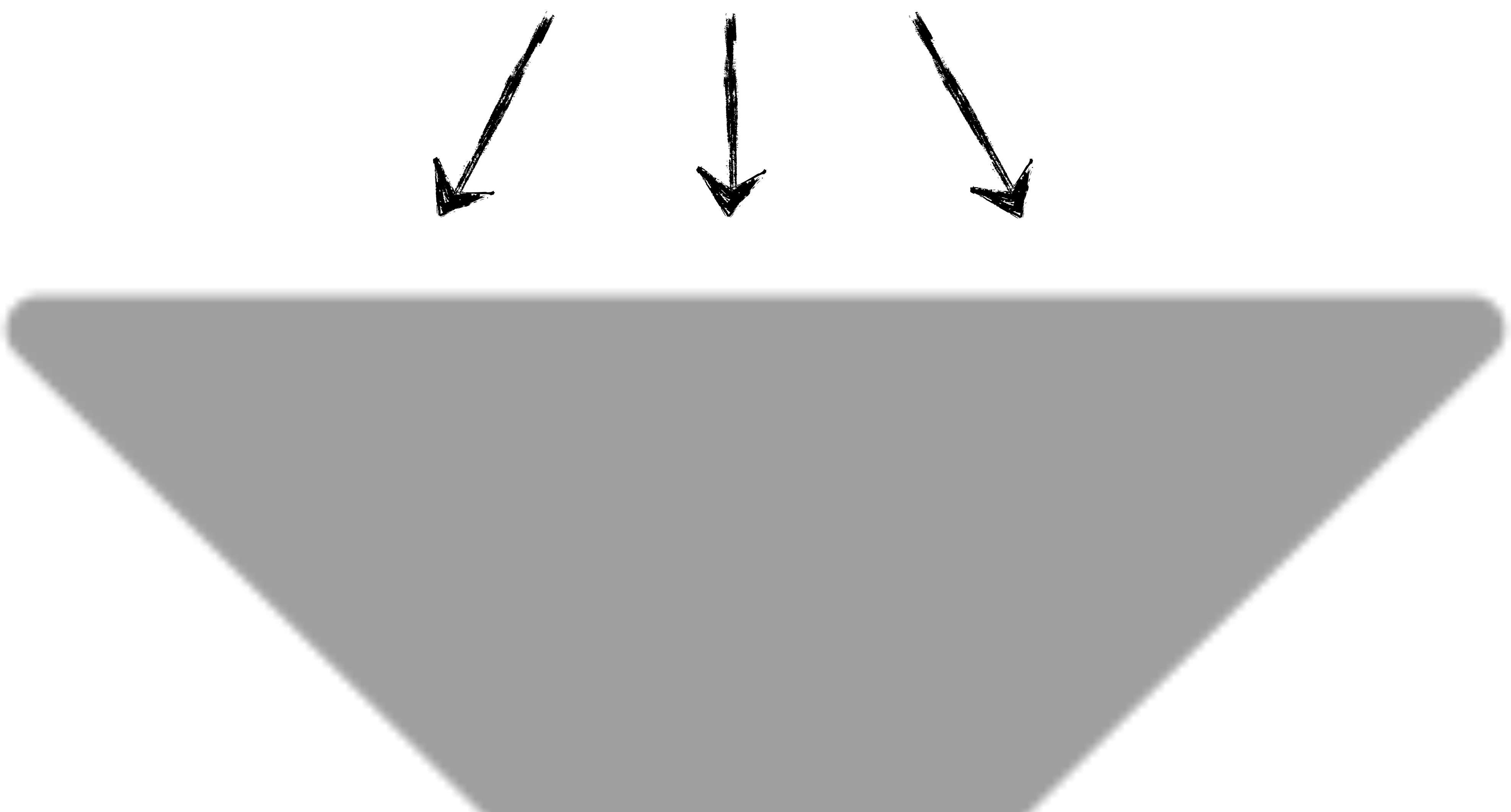


true or false positive lookup



Optimal number of hash functions

$$= \underbrace{\ln(2) \cdot \boldsymbol{M}}_{h_1 \ \ldots \ h_k} \qquad \leftarrow \quad \text{bits / entry}$$



Optimal number of hash functions

$$= \underbrace{\ln(2) \cdot \boldsymbol{M}}_{h_1 \; \ldots \; h_k}$$

bits / entry

With M bits / entry

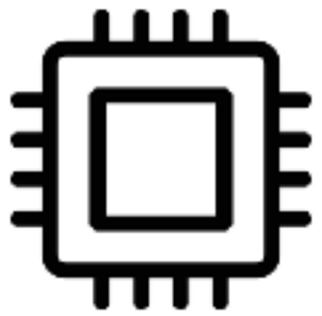
Optimal number of hash functions $= \ln(2) \cdot M$

False positive rate = $\boldsymbol{2^{-M \cdot \ln(2)}}$

5 fronts

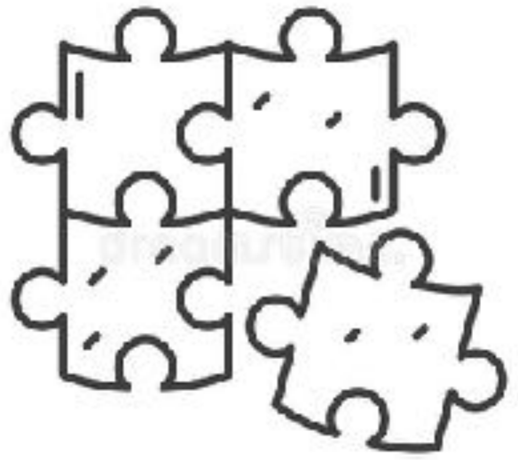
Holistic Tuning

CPU



Lowering Constants

Unification



Range

Holistic Tuning

Monkey

Dostoevsky

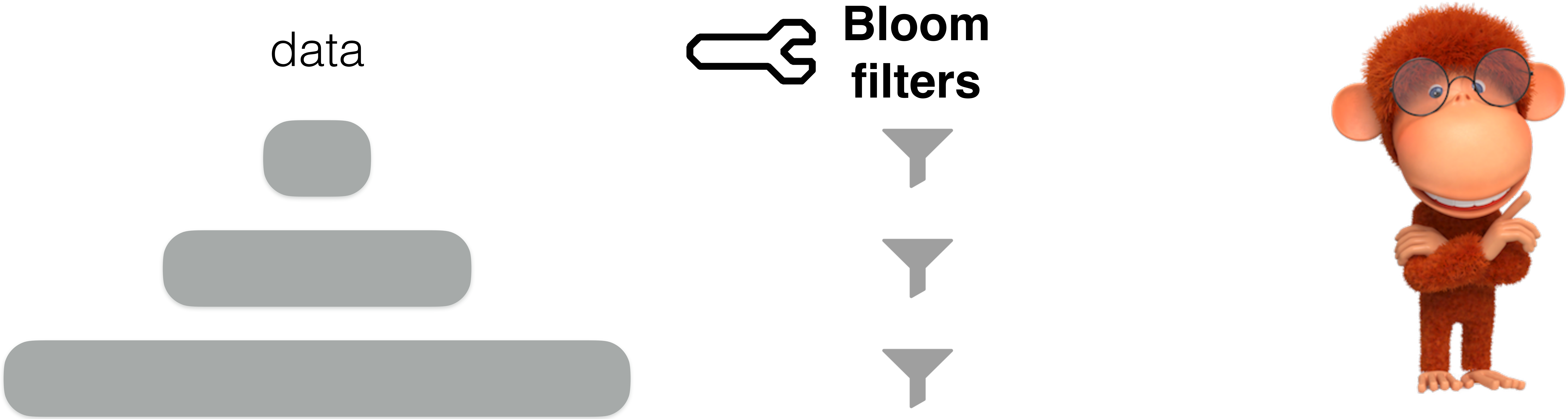
LSM-Bush

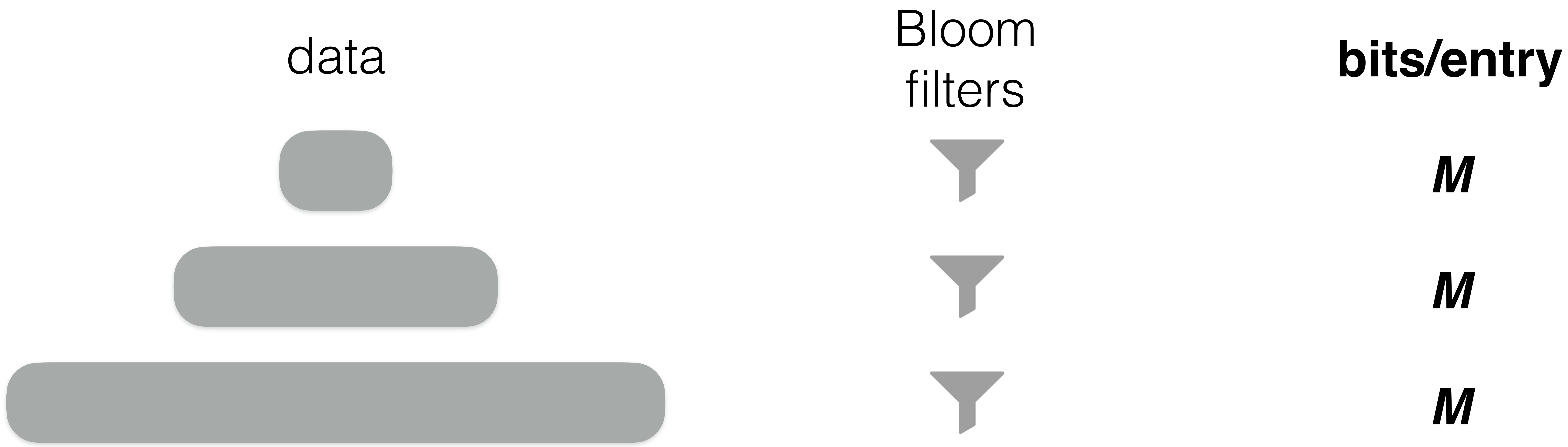
DayanSIGMOD17

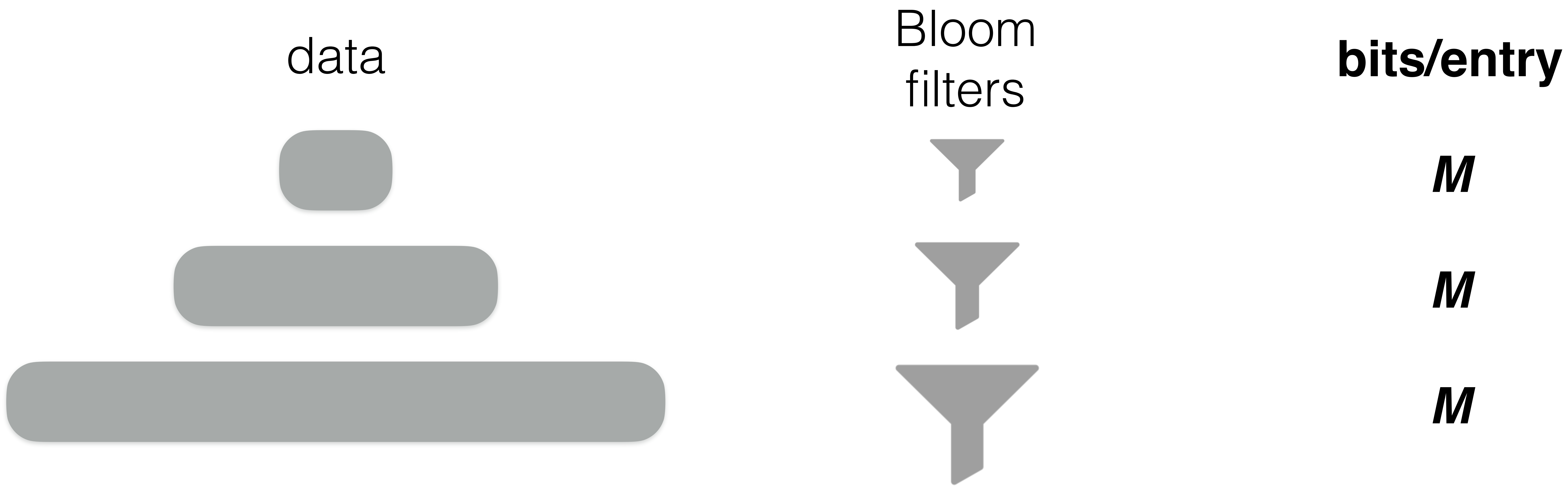
DayanSIGMOD18

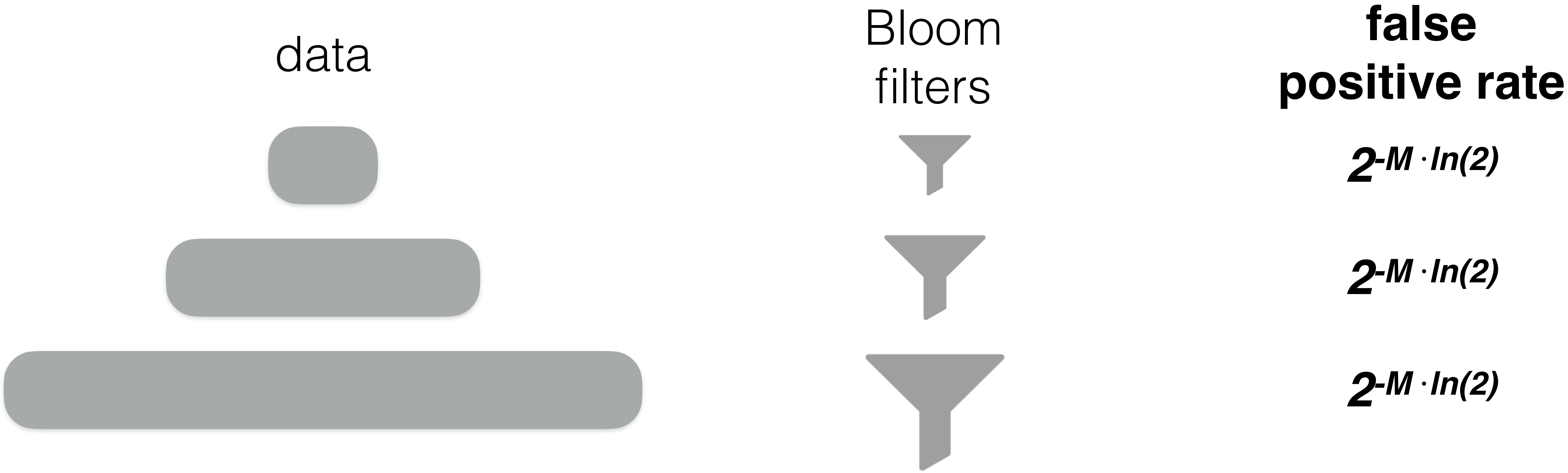
DayanSIGMOD19

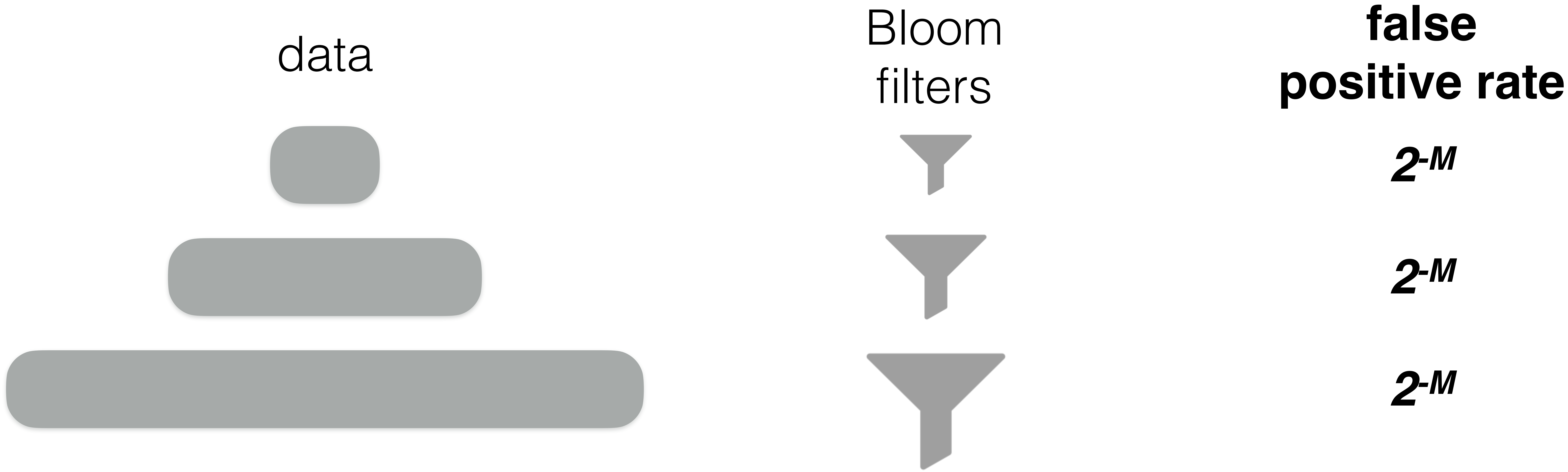
Monkey: **O**ptimal **N**avigable **Key**-Value Store




data
Bloom filters
bits/entry
M
M
M


data
Bloom filters
bits/entry
M
M
M


data
Bloom filters
false positive rate
$2^{-M \cdot ln(2)}$
$2^{-M \cdot ln(2)}$
$2^{-M \cdot ln(2)}$


data
Bloom filters
false positive rate
2^{-M}
2^{-M}
2^{-M}

Bloom filters	false positive rate
	2^{-M} 2^{-M} 2^{-M}

$$\left.\begin{matrix} 2^{-M} \\ 2^{-M} \\ 2^{-M} \end{matrix}\right\} = O(\boldsymbol{2^{-M} \cdot \log_T N})$$

Bloom filters

false positive rate

2^{-M}

2^{-M}

2^{-M}

$$= \quad O(\mathbf{1+2^{-M} \cdot \log_T N})$$

Bloom filters	false positive rate

$$\left.\begin{array}{c}2^{-M}\\[1em]2^{-M}\\[1em]2^{-M}\end{array}\right\} = O(\boldsymbol{2^{-M}} \cdot \mathbf{log}_{\boldsymbol{T}}\, \boldsymbol{N})$$

Bloom filters

false positive rate

2^{-M}

2^{-M}

most memory

2^{-M}

Bloom filters

false positive rate

2^{-M}

2^{-M}

most memory

2^{-M}

saves at most 1 access!

bits / entry

$M + 2$

$M + 1$

$M - 1$

reallocate

false
positive rates

$2^{-(M+2)}$ ↓

$2^{-(M+1)}$ ↓

$2^{-(M-1)}$ ↑

 $\propto 2^{-M} / T^{2}$

 $\propto 2^{-M} / T^{1}$

 $\propto 2^{-M} / T^{0}$

$$\left.\begin{array}{l} 2^{-M} / T^2 \\ 2^{-M} / T^1 \\ 2^{-M} / T^0 \end{array}\right\} = O(\mathbf{2^{-M}})$$

geometric progression

Faster worst case

$$O(2^{-M}) \quad < \quad O(2^{-M} \cdot \log_T N)$$

Monkey opens up new ways of optimizing write performance without sacrificing get performance

Monkey

Dostoevsky

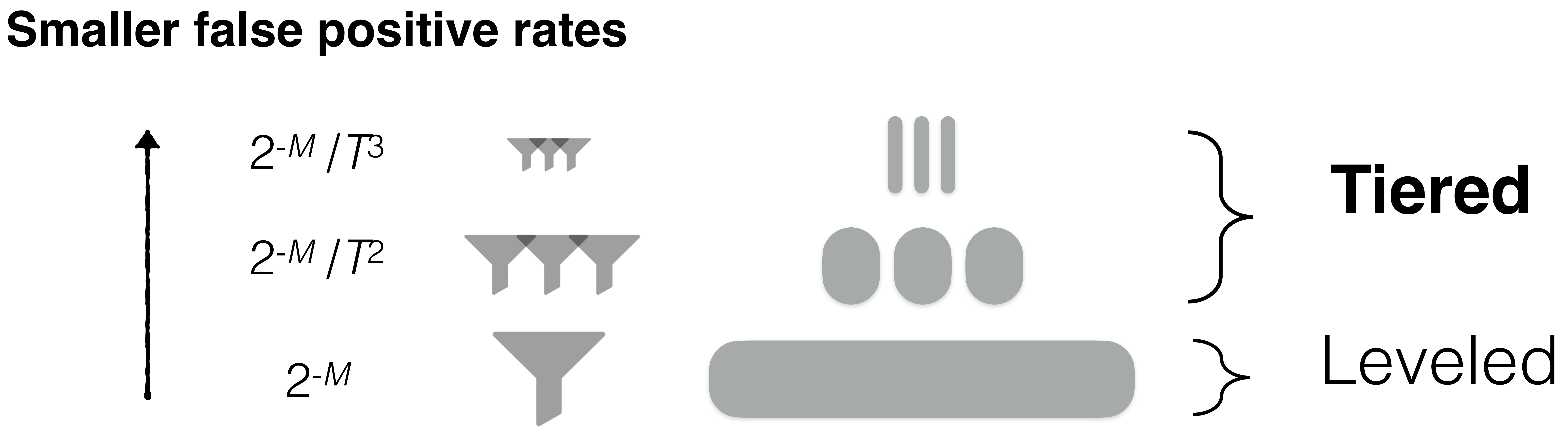
LSM-Bush

DayanSIGMOD17

DayanSIGMOD18

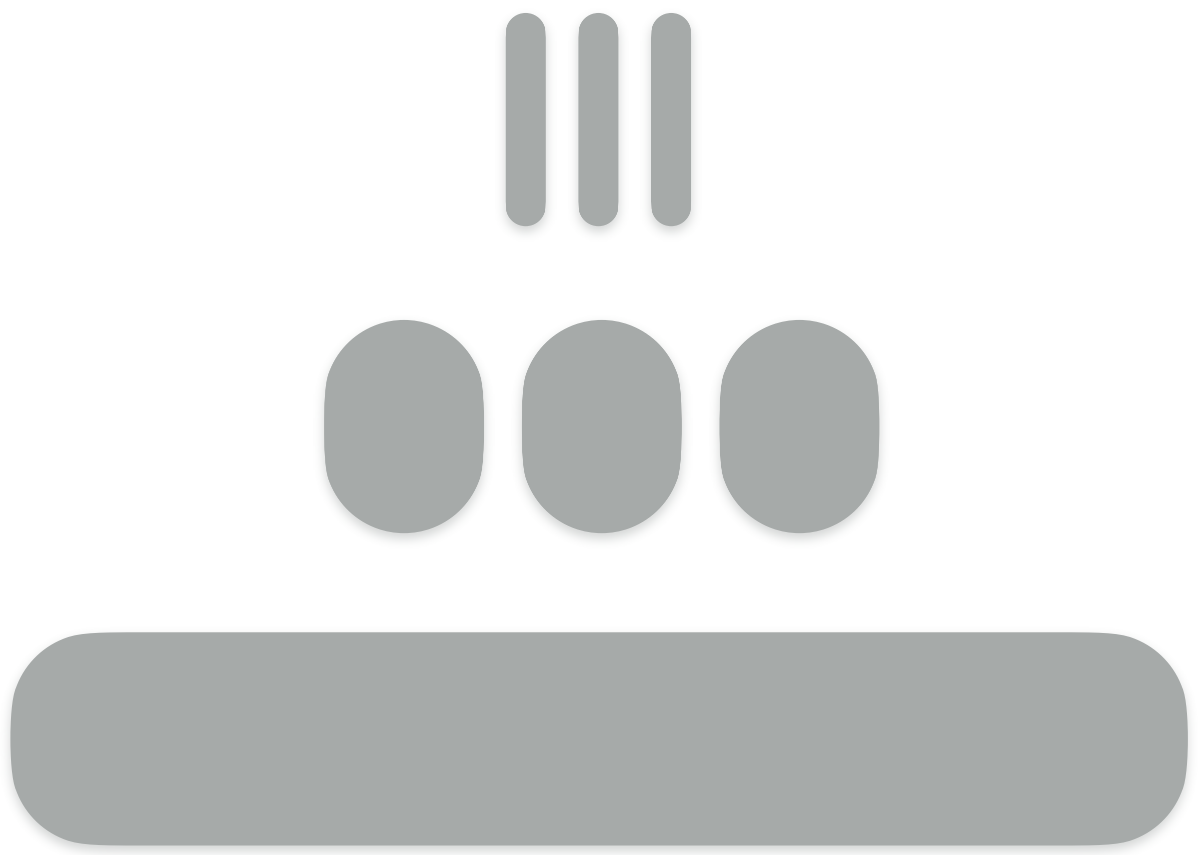
DayanSIGMOD19

Dostoevsky



gets

O($\mathbf{2^{-\mathit{M}}}$)



writes

O($\boldsymbol{T + \log_T N}$)

=

O(1)

+

O(1)

+

O(T)

gets

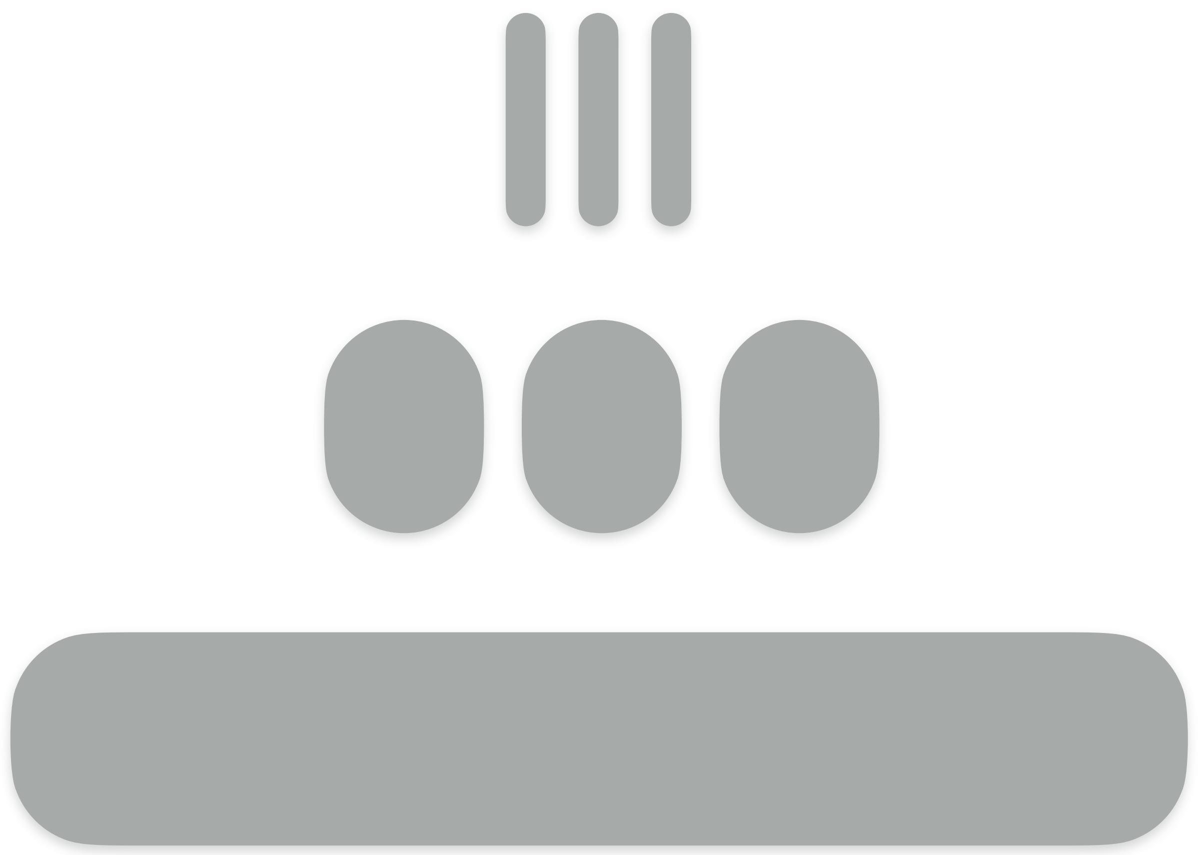
$O(2^{-M})$

writes

$O(T + \log_T N)$ < $O(T \cdot \log_T N)$

leveling

=



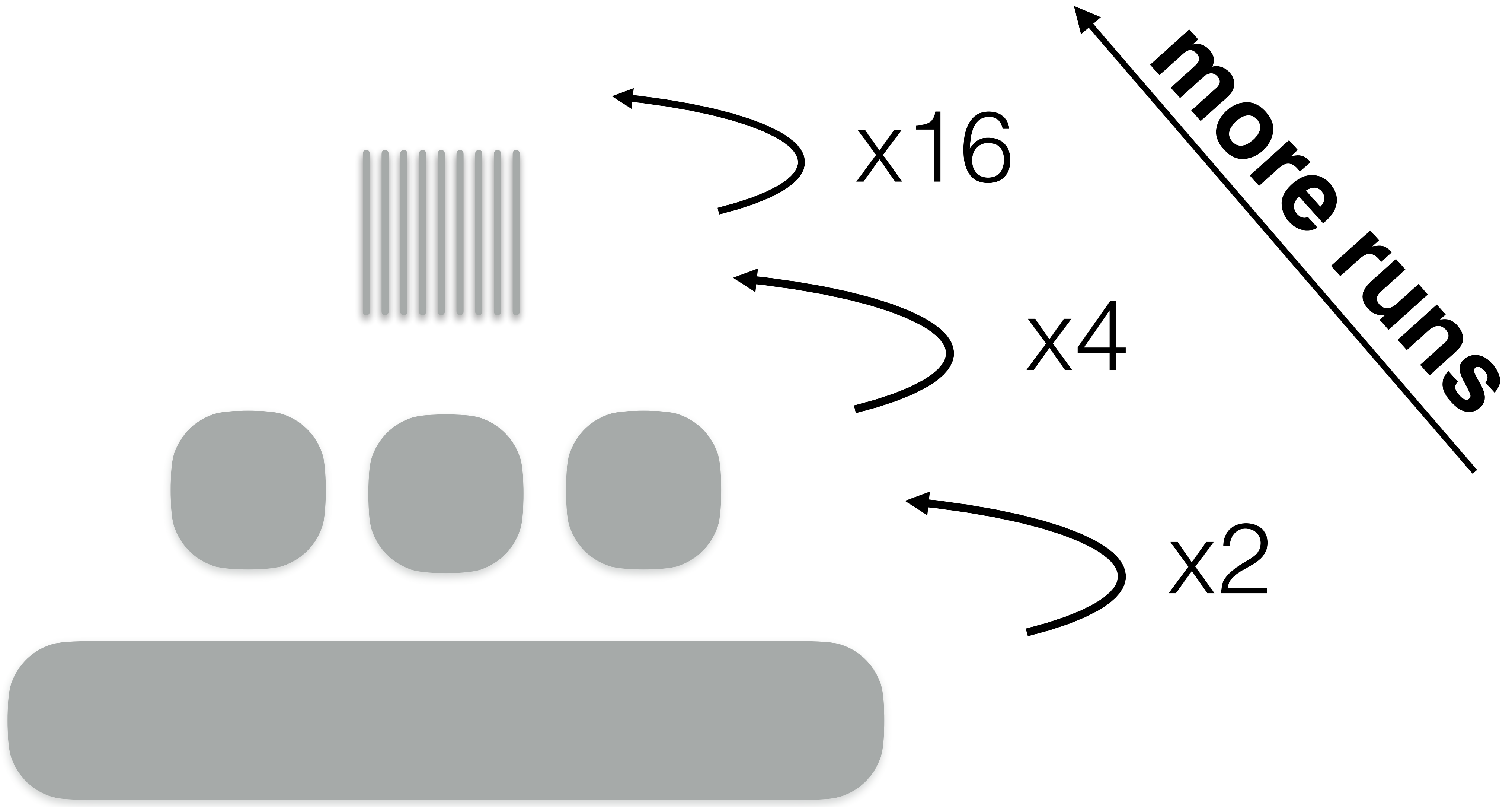
$O(1)$
+
$O(1)$
+
$O(T)$

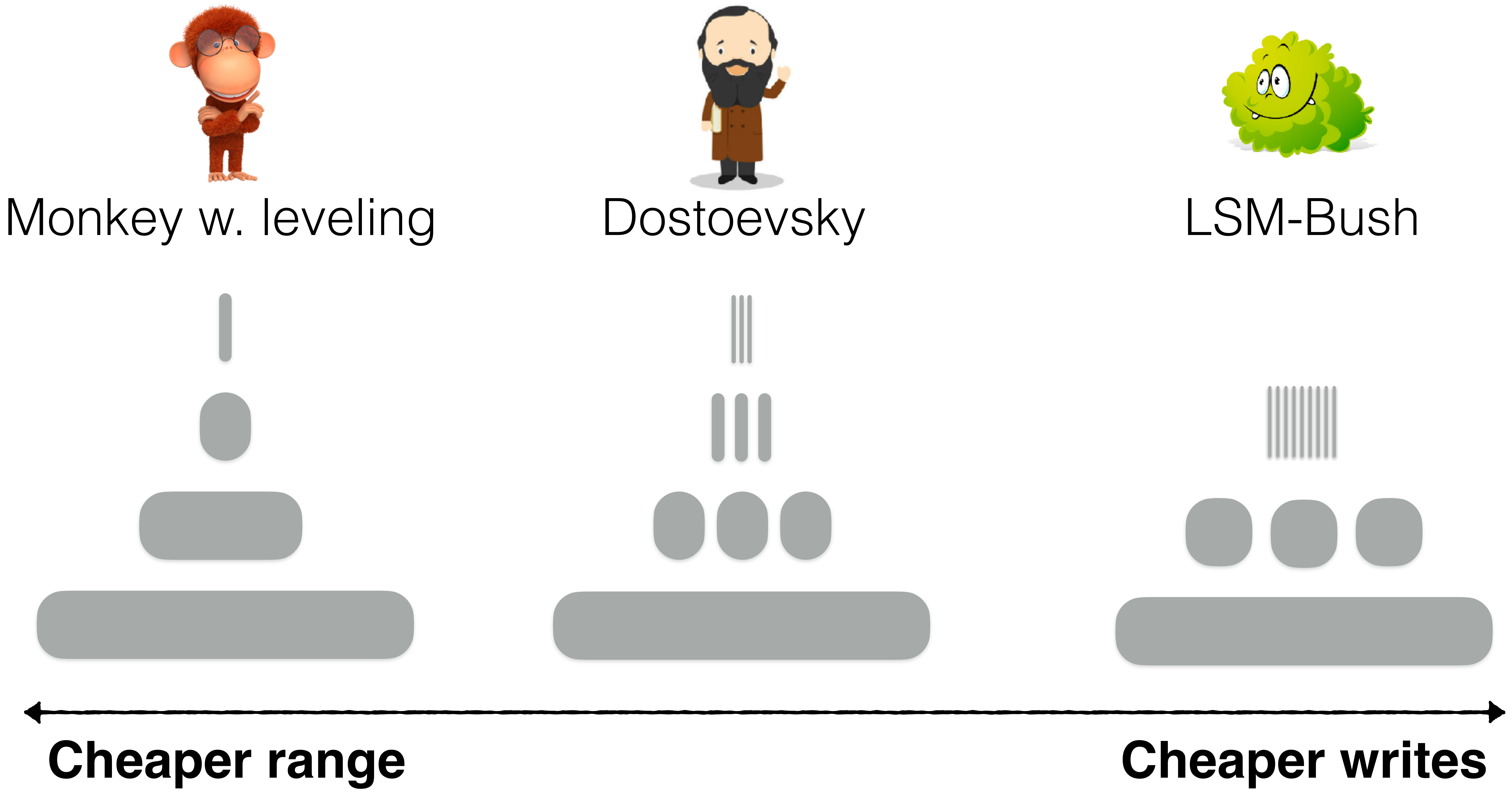
Dostoevsky

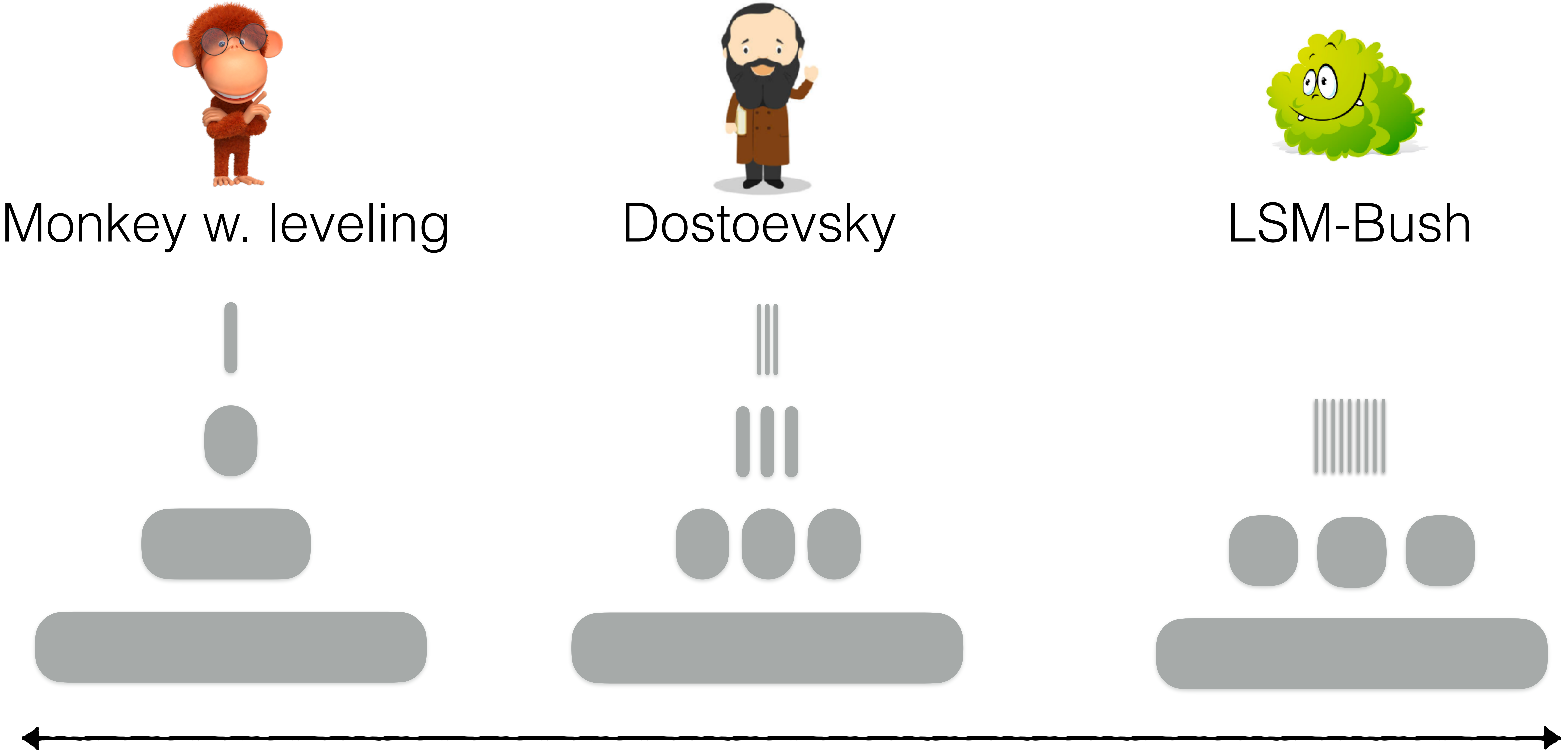
$O(T + \log_T N)$

LSM-Bush

$O(\log_2 \log_T N)$




Monkey w. leveling
Dostoevsky
LSM-Bush
Cheaper range
Cheaper writes

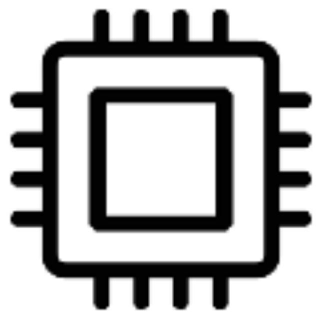


Great point reads all across

5 fronts

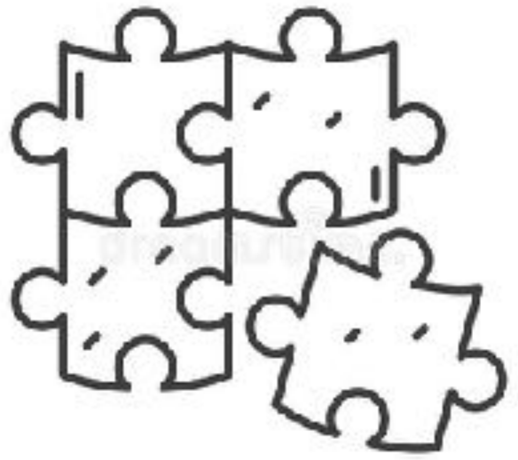
Holistic Tuning

CPU



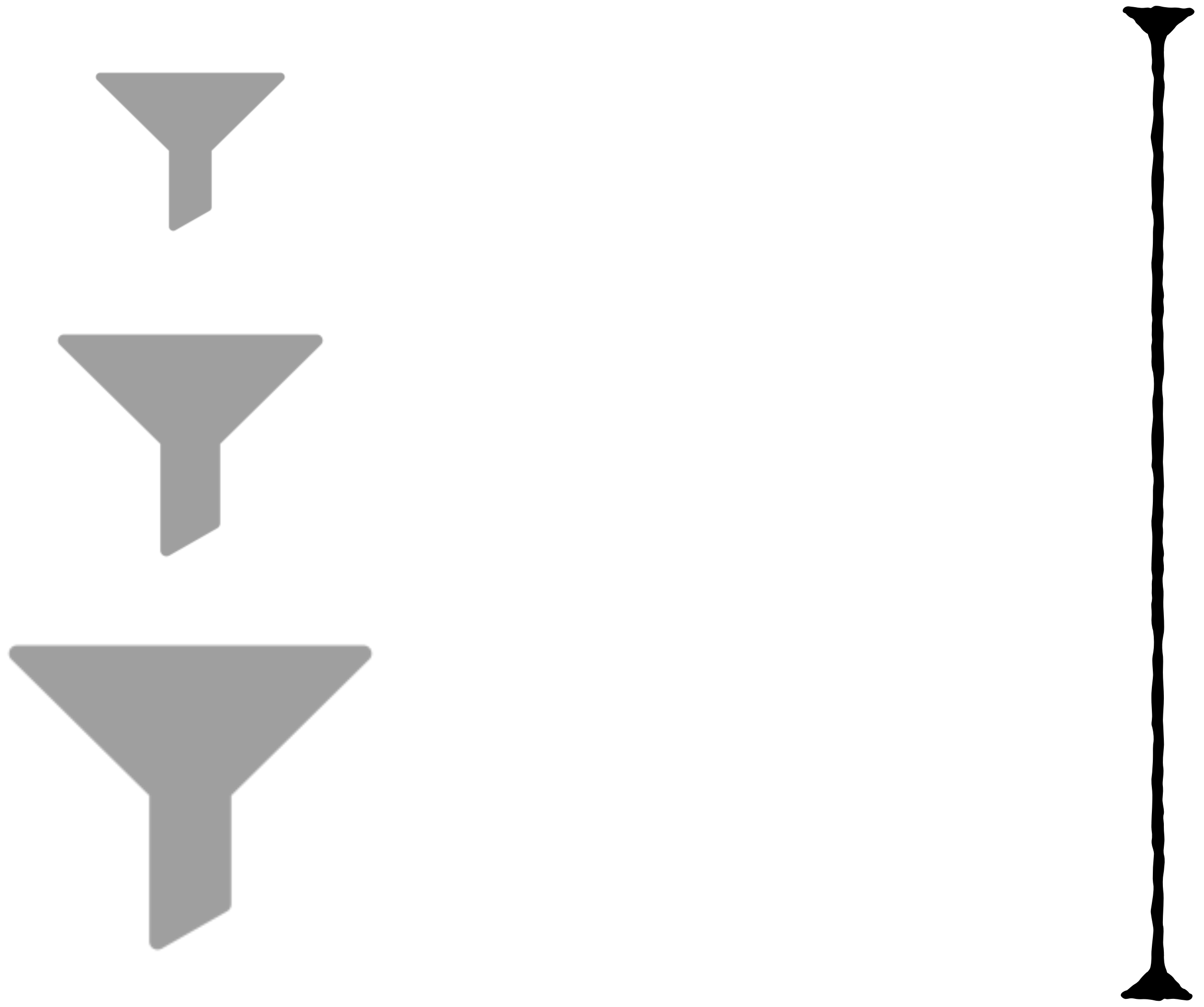
Lowering Constants

Unification



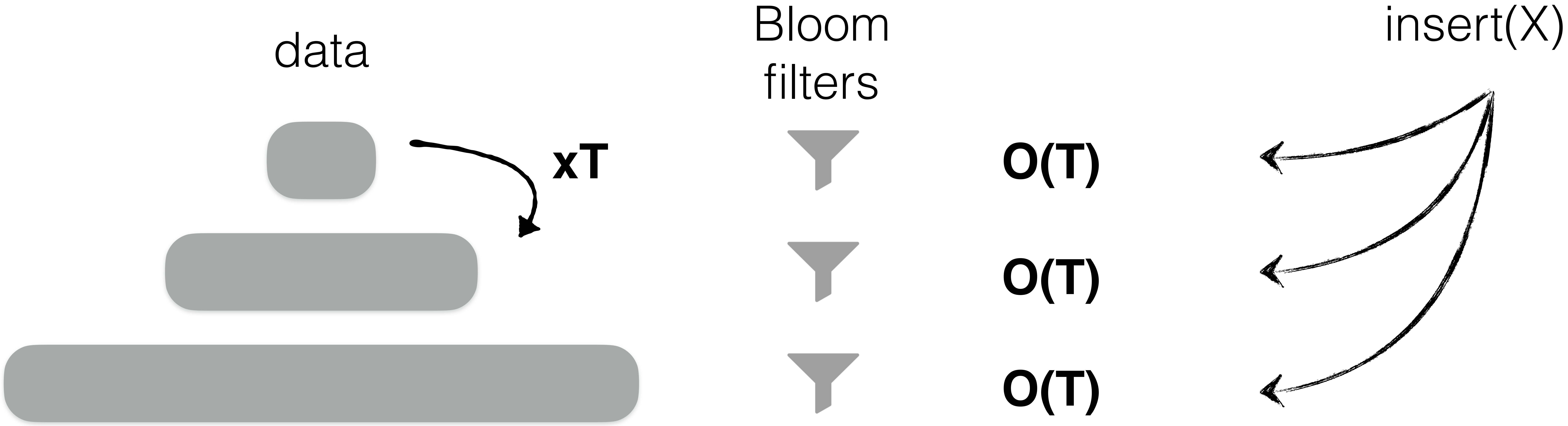
Range

Bloom filters



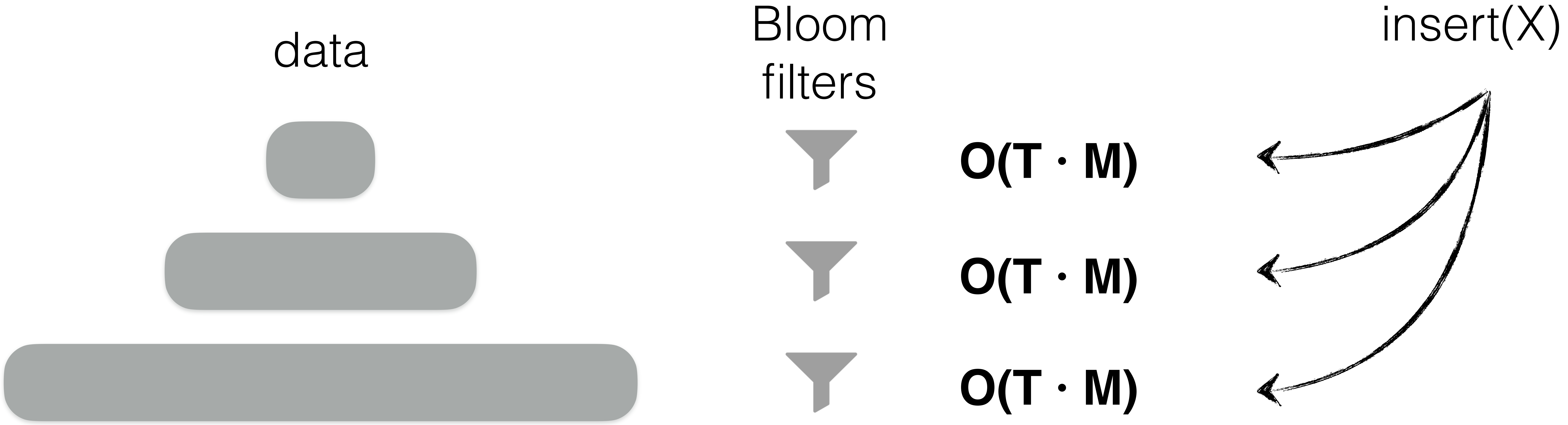
CPU overhead?

Each key is inserted O(T) times per level into a filter



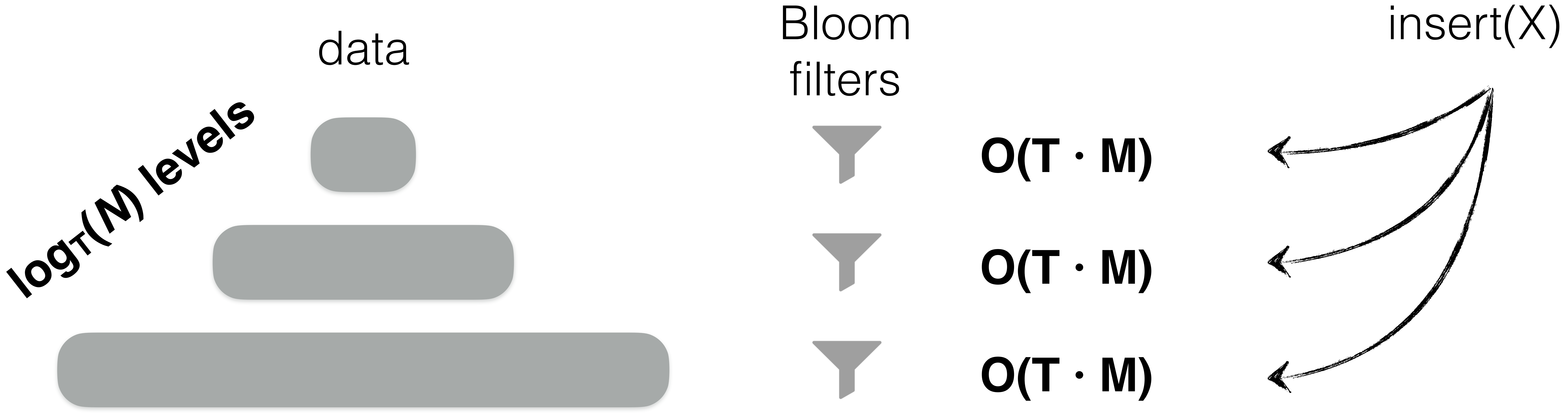
Each key is inserted O(T) times per level into a filter

Each filter insertion uses M · ln(2) hash functions



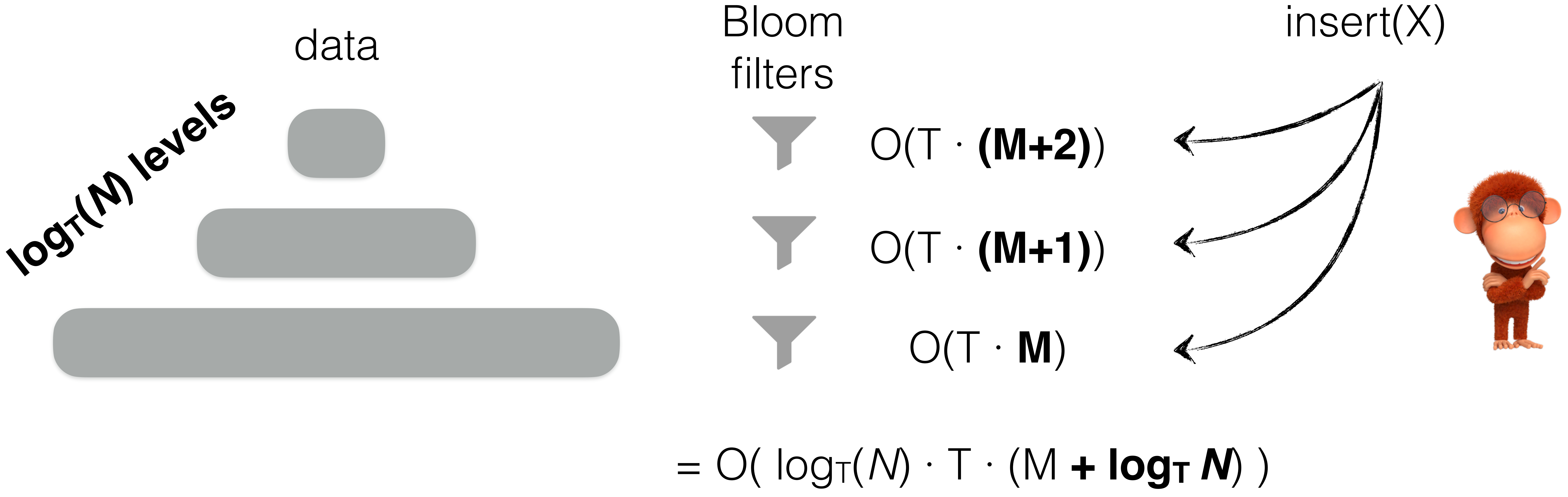
Each key is inserted O(T) times per level into a filter

Each filter insertion uses M · ln(2) hash functions

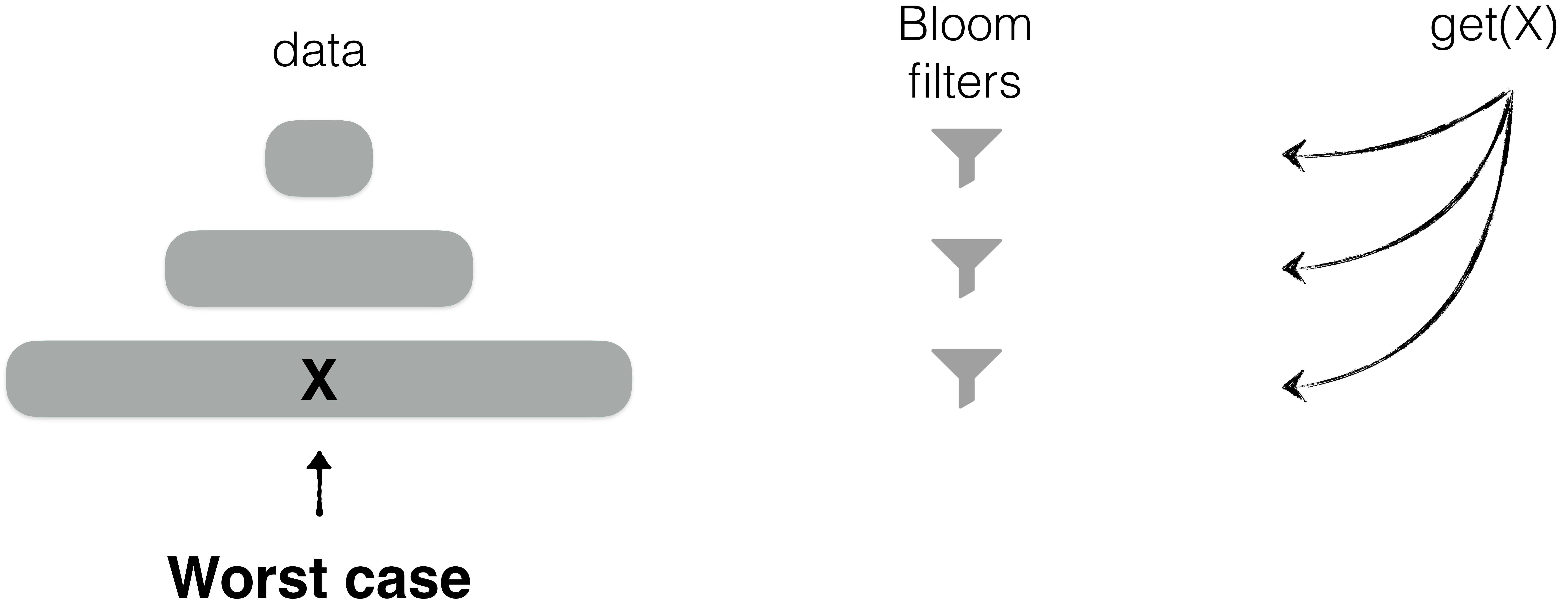


$$= O(\log_T(N) \cdot T \cdot M)$$

With Monkey more hash functions are used

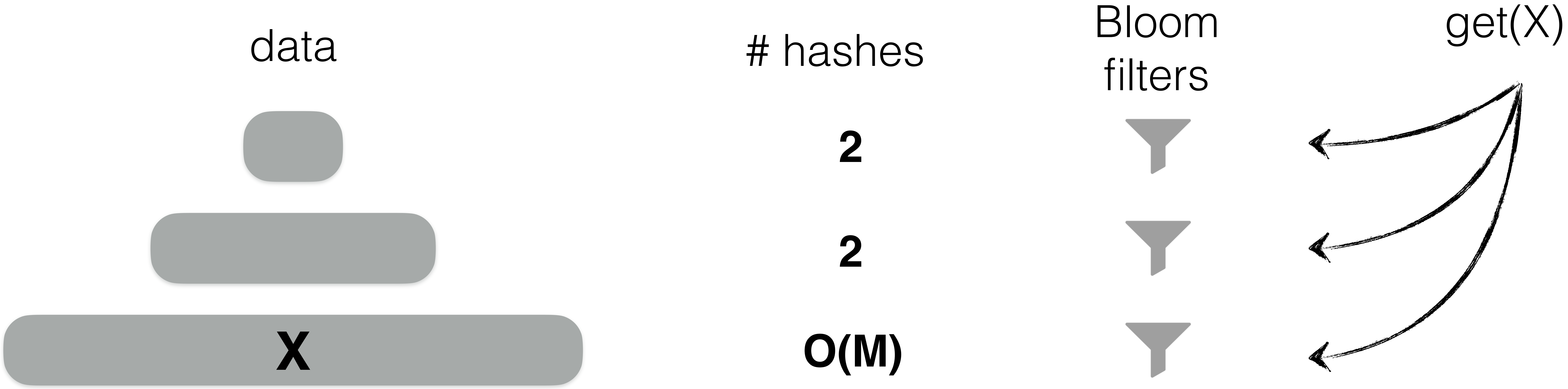


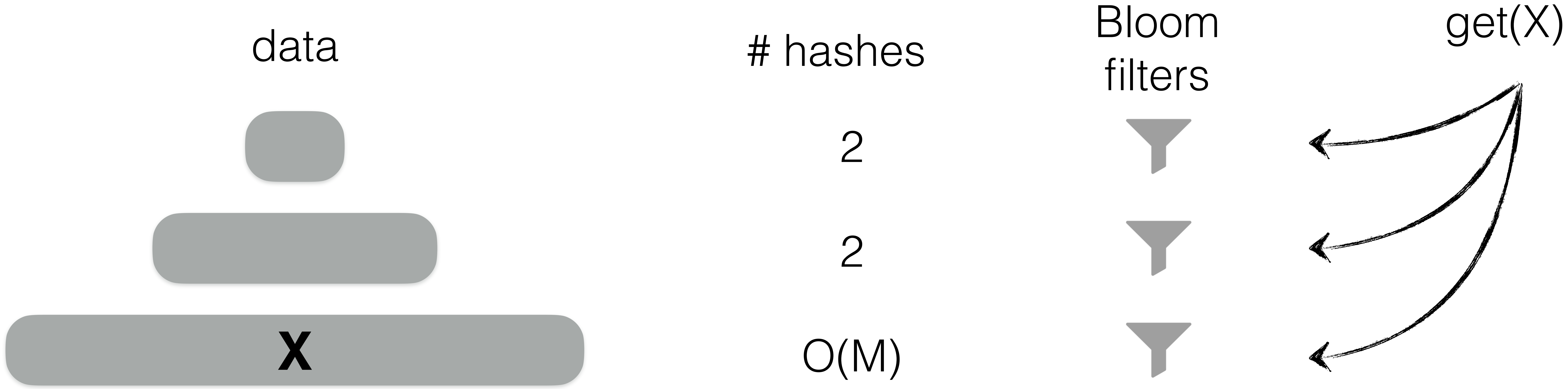
How about get cost?



Expected Negative Query Cost ≈ 2

Positive Query Cost ≈ M · ln(2)



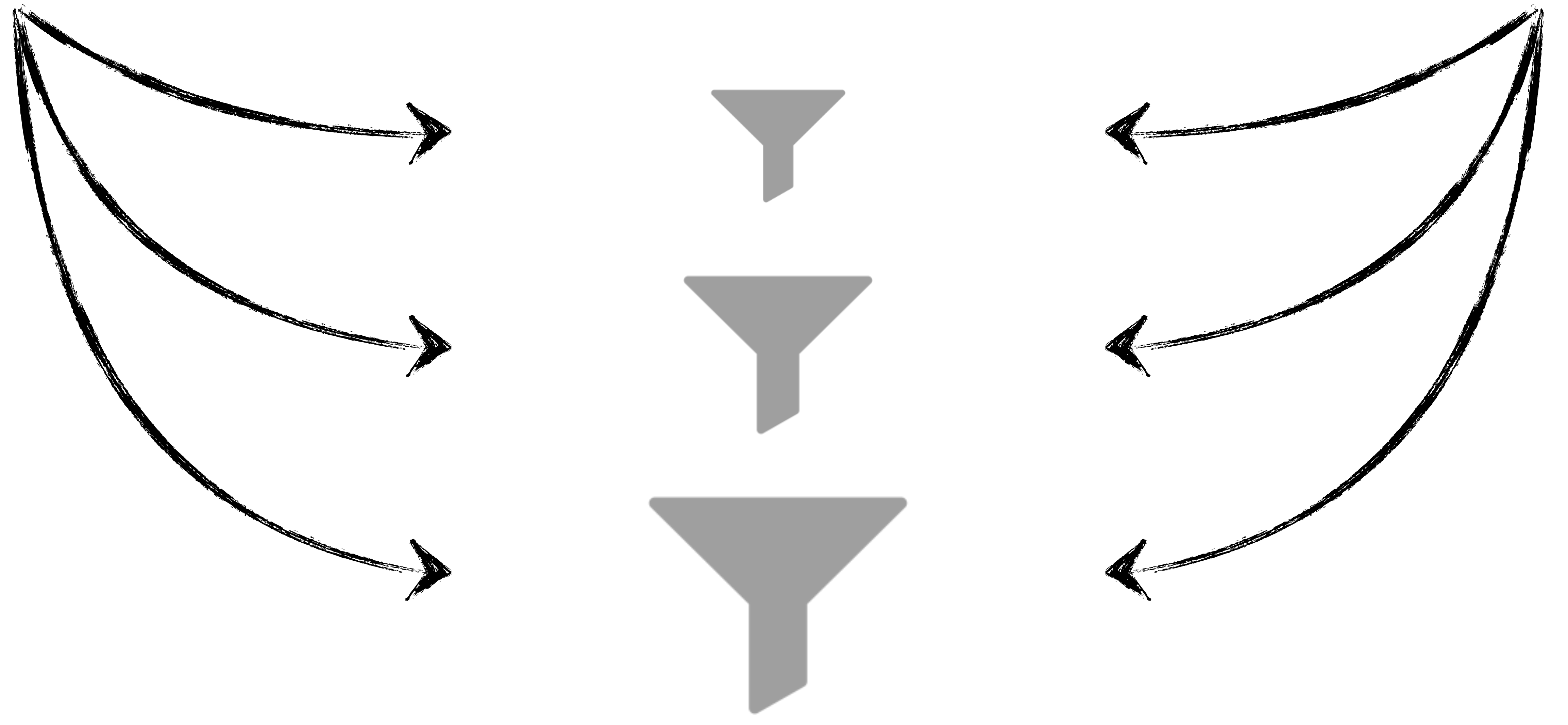


Avg. worst case $= O(\, M + \log_T N \,)$

get
$O(M + \log_T N)$

Bloom
filters

Insert
$O(T \cdot M \cdot \log_T N)$

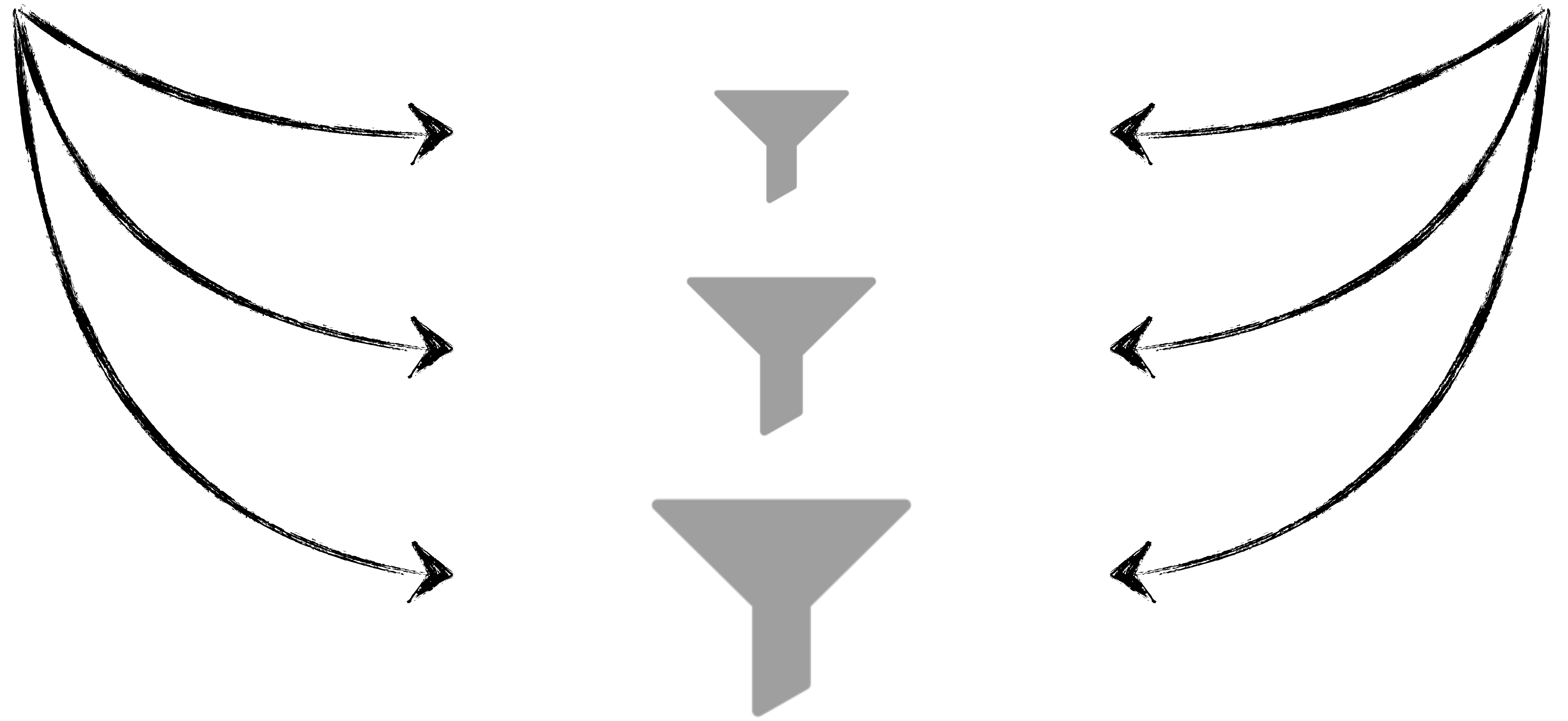


Address Using Blocking and SIMD

get
$O(\ M + \log_T N\ )$

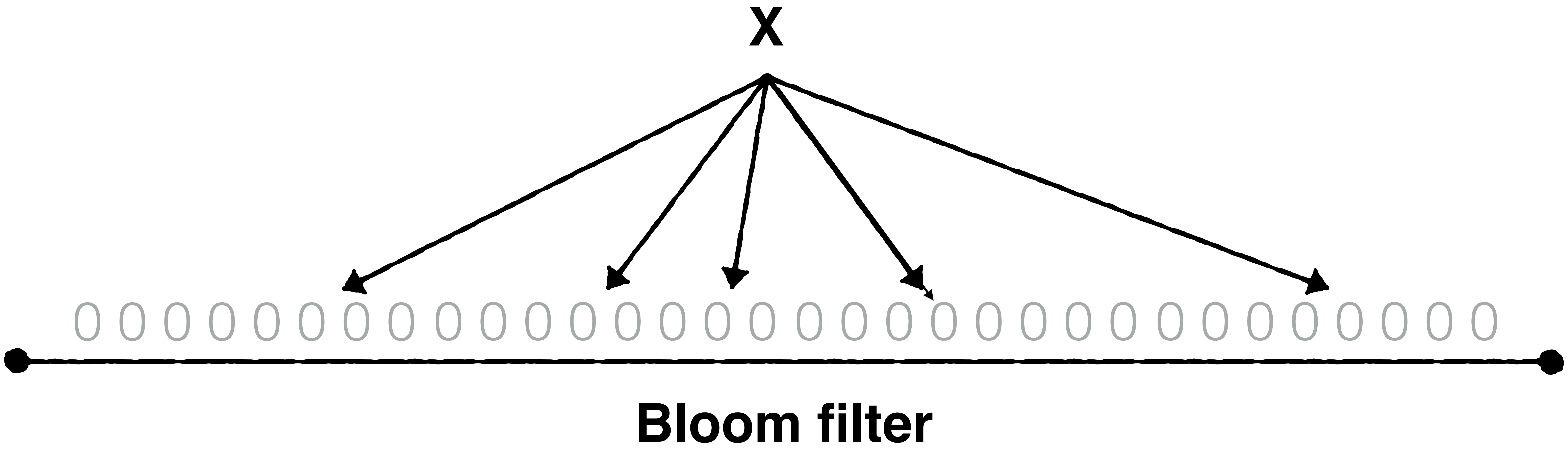
Bloom
filters

Insert
$O(\ T \cdot M \cdot \log_T N\ )$



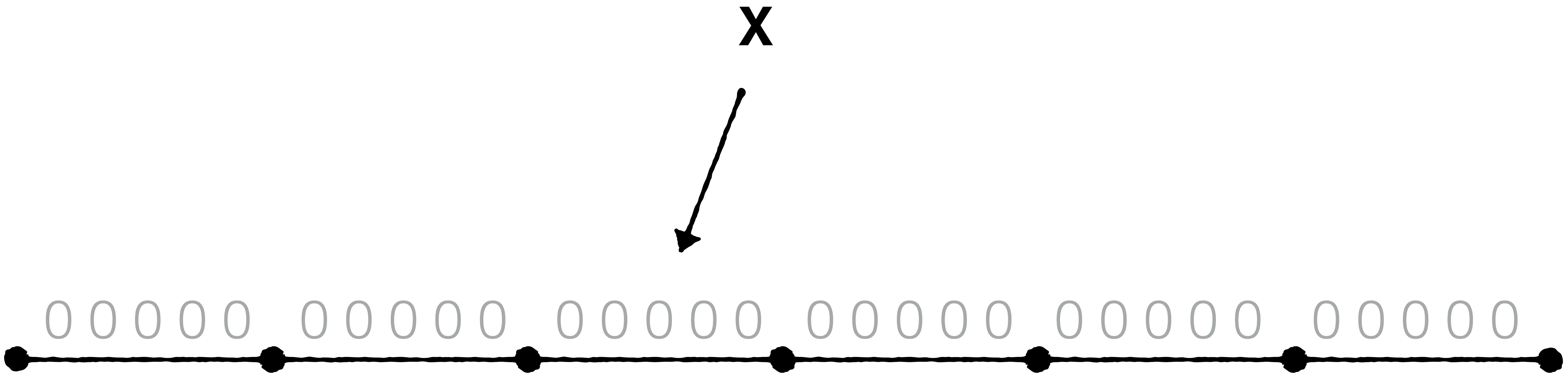
Blocking

PutzeJEA10

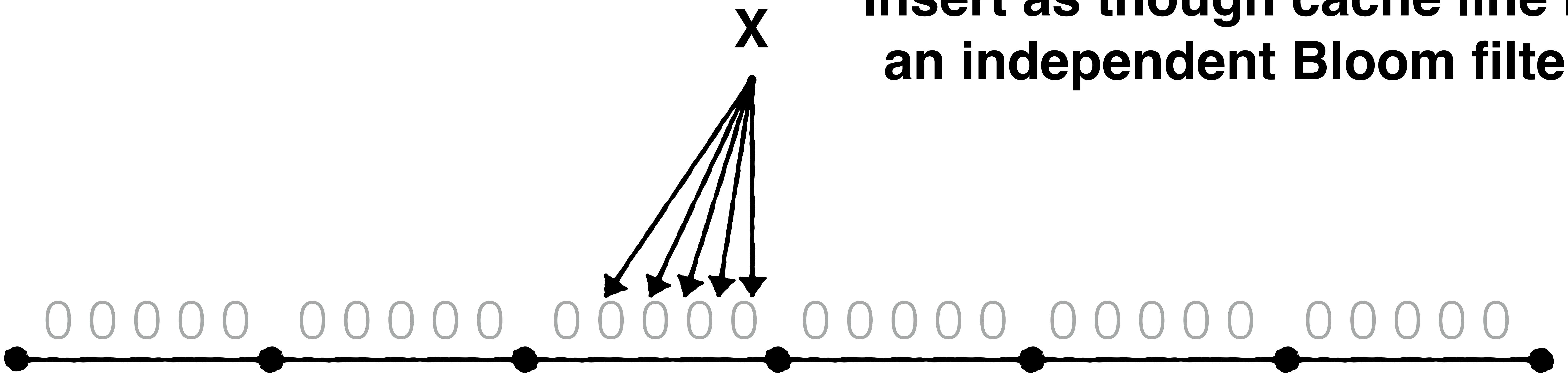


Blocking

Hash to one cache line



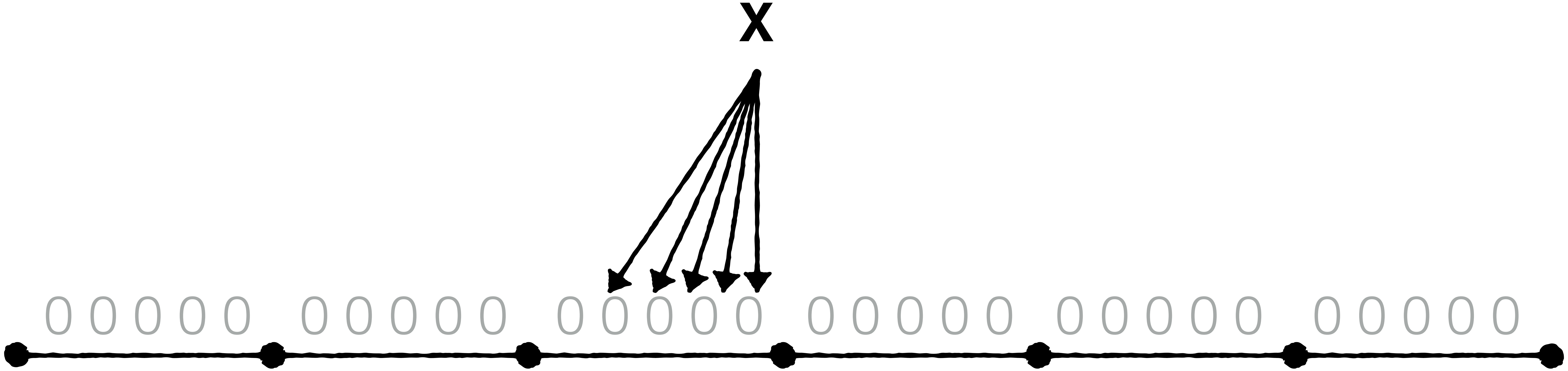
Blocking



Insert as though cache line is an independent Bloom filter

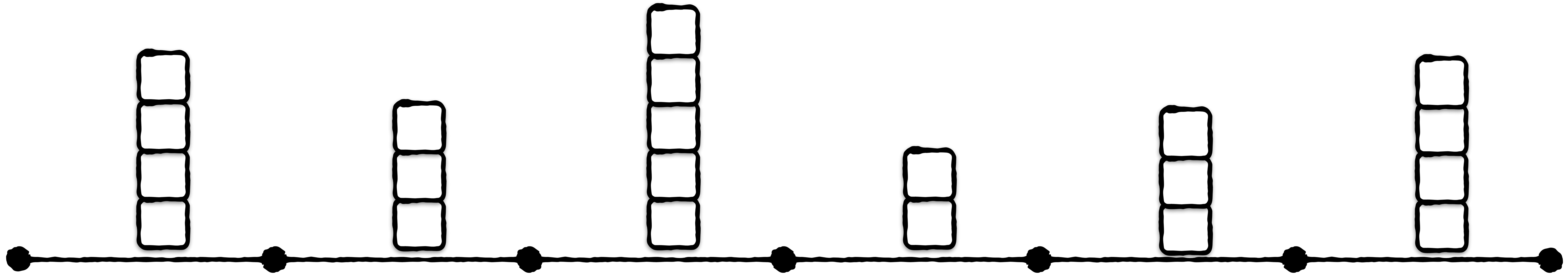
Blocking

Pro: one cache miss per insert/get



Blocking

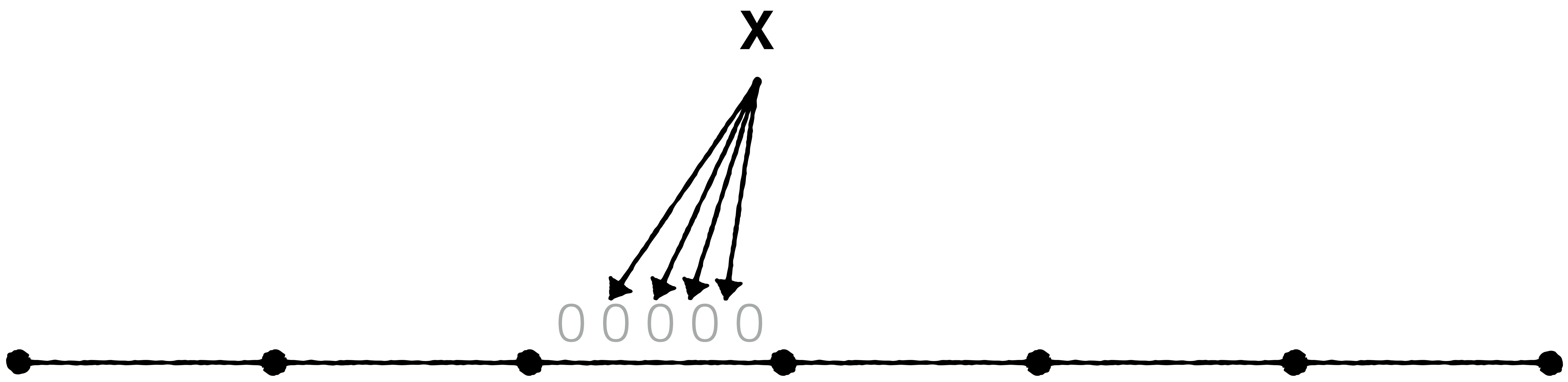
Con 1: uneven distribution of entries across cache lines slightly harms the false positive rate



Blocked Bloom filter

Blocking

Con 2: still need to compute many hash functions per entry



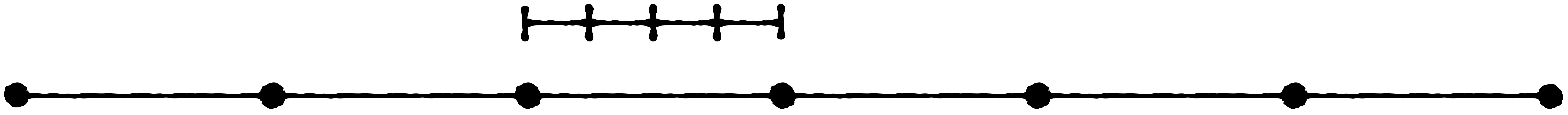
Blocked Bloom filter

SIMD

PolychroniouDAMON14

JianyuanTPDS18

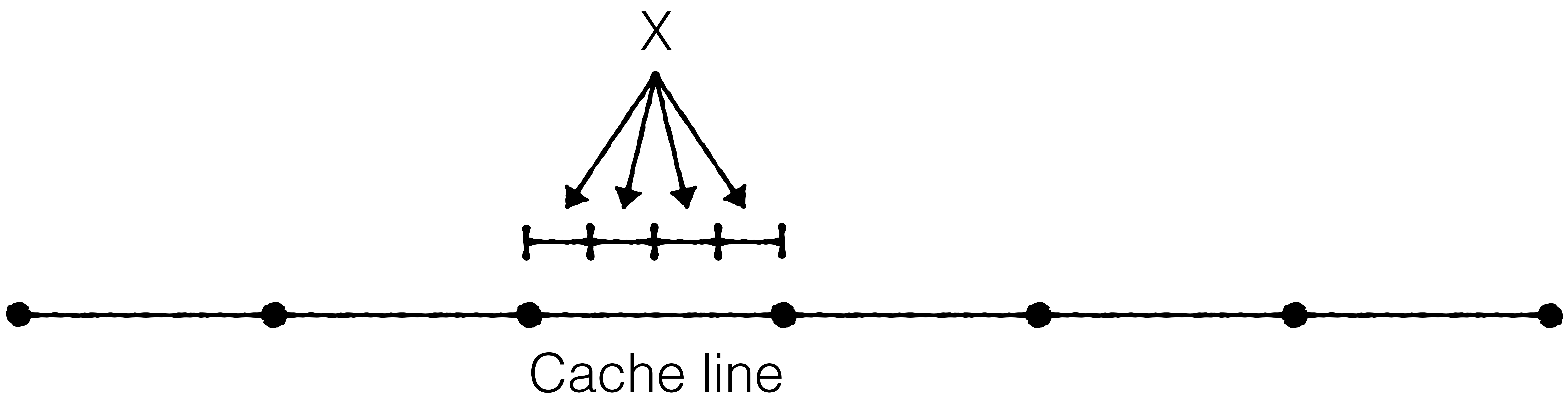
Partition into sub-lines



Cache line

SIMD

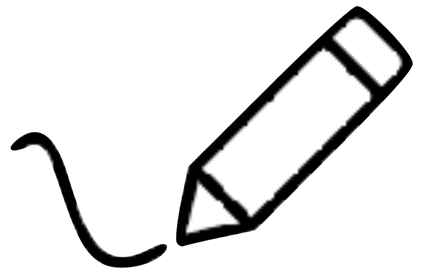
Map one hash per sub-line

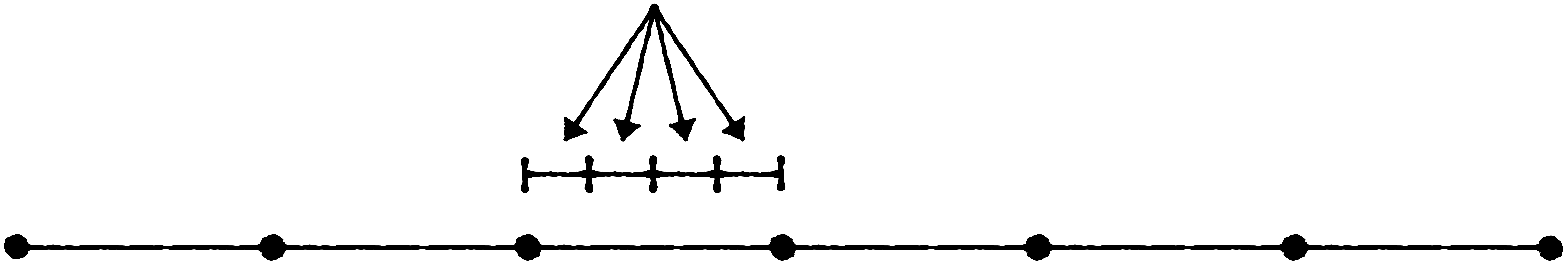


SIMD

Insert(X)

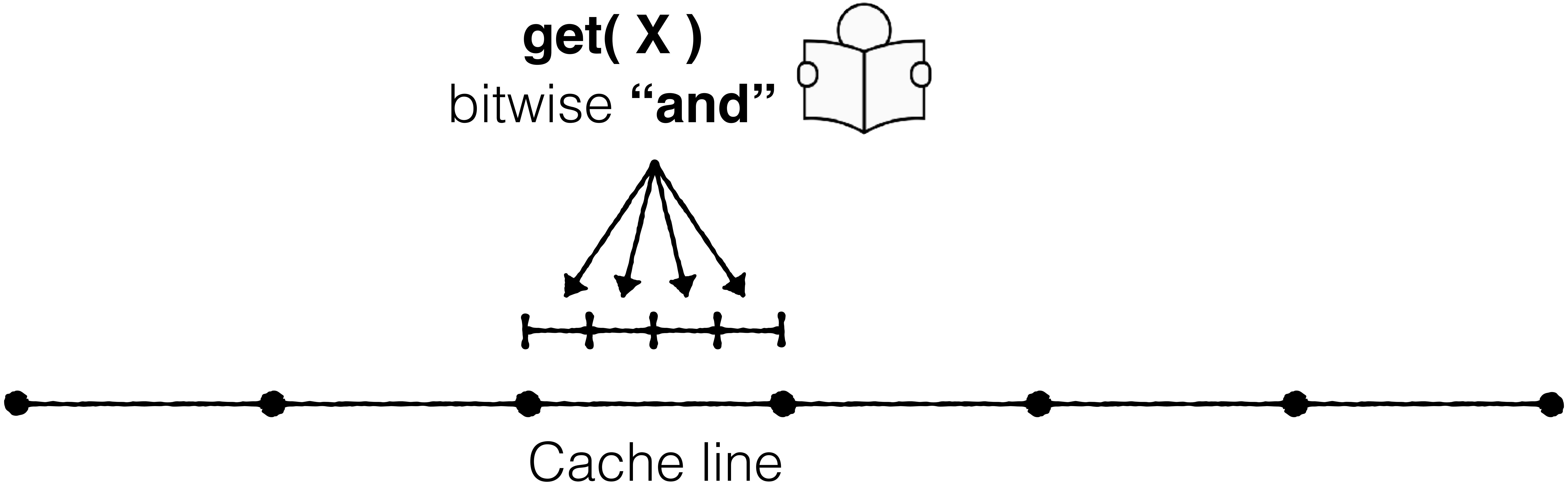
bitwise **“or”**





Cache line

SIMD

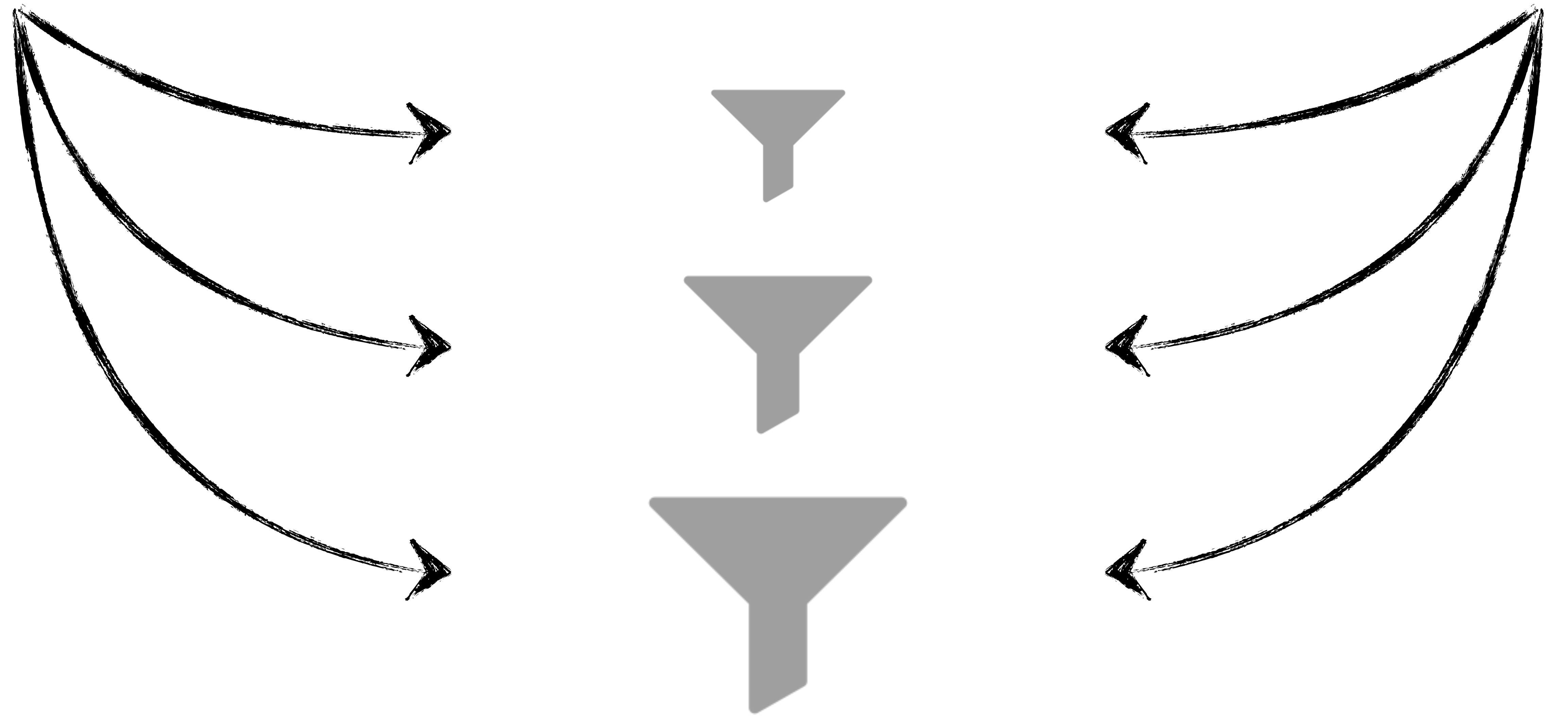


Blocking and SIMD

get
$O(M + \log_T N)$

Bloom
filters

Insert
$O(T \cdot M \cdot \log_T N)$

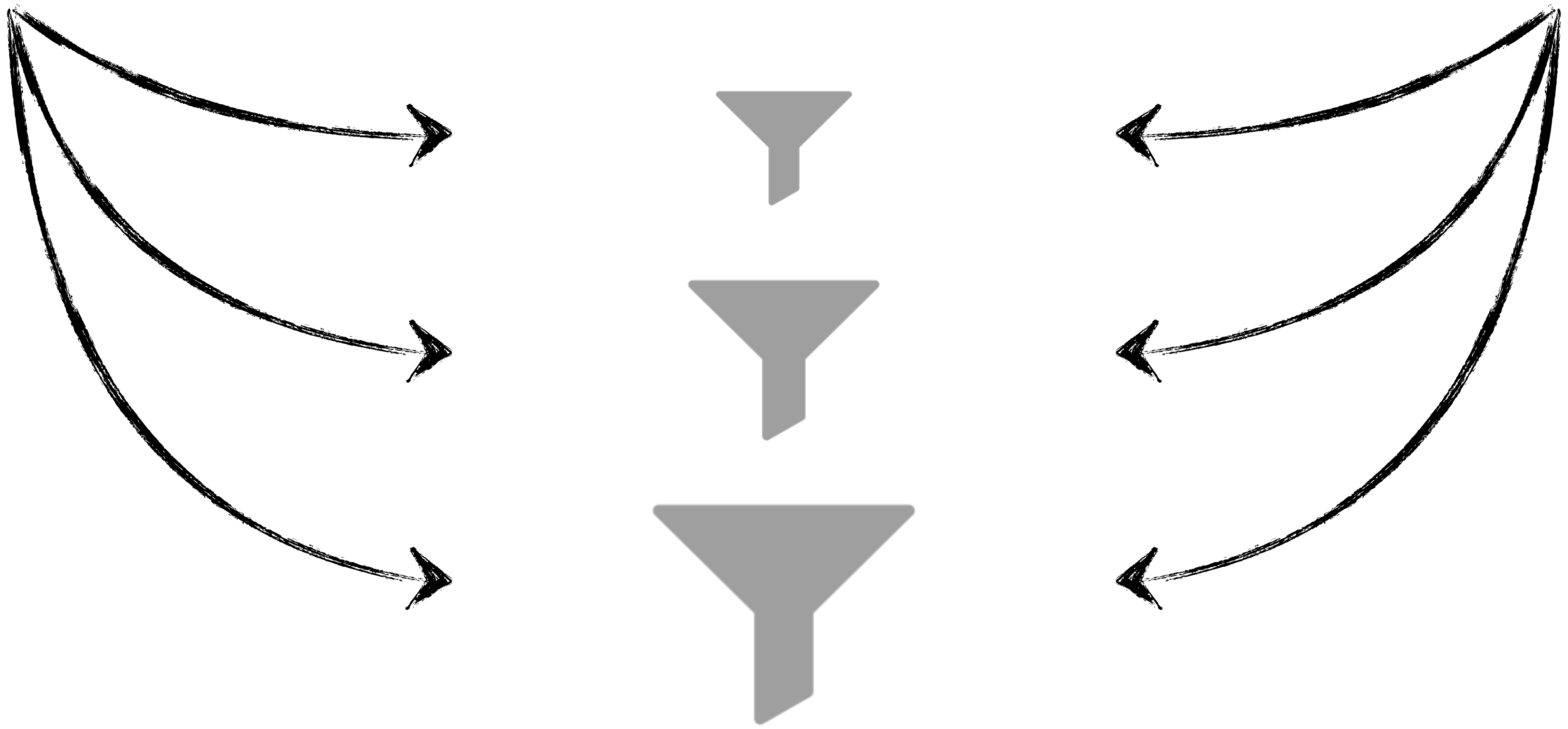


Blocking and SIMD

get
$O(\log_T N)$

Bloom
filters

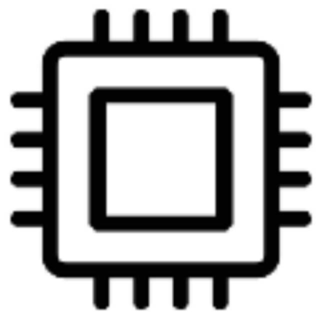
Insert
$O(T \cdot \log_T N)$



5 fronts

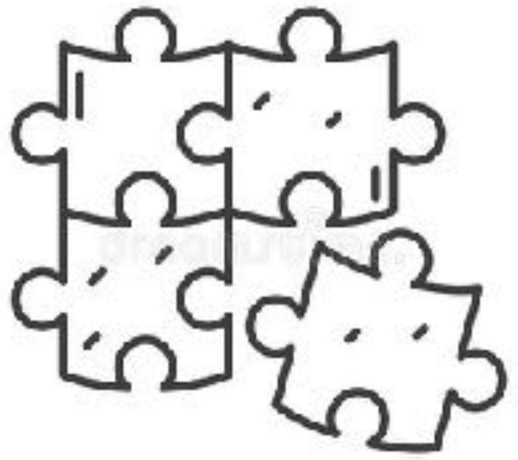
Holistic Tuning

CPU



Improving Constants

Unification



Range

Improving Constants

	Bloom	Ideal
		
False positive rate	$\approx 2^{-M \cdot \ln(2)}$	$\approx \mathbf{2^{-M}}$

	Bloom	Ideal
		
False positive rate	$\approx 2^{-M \cdot \mathbf{0.69}}$	$\boldsymbol{\approx 2^{-M}}$

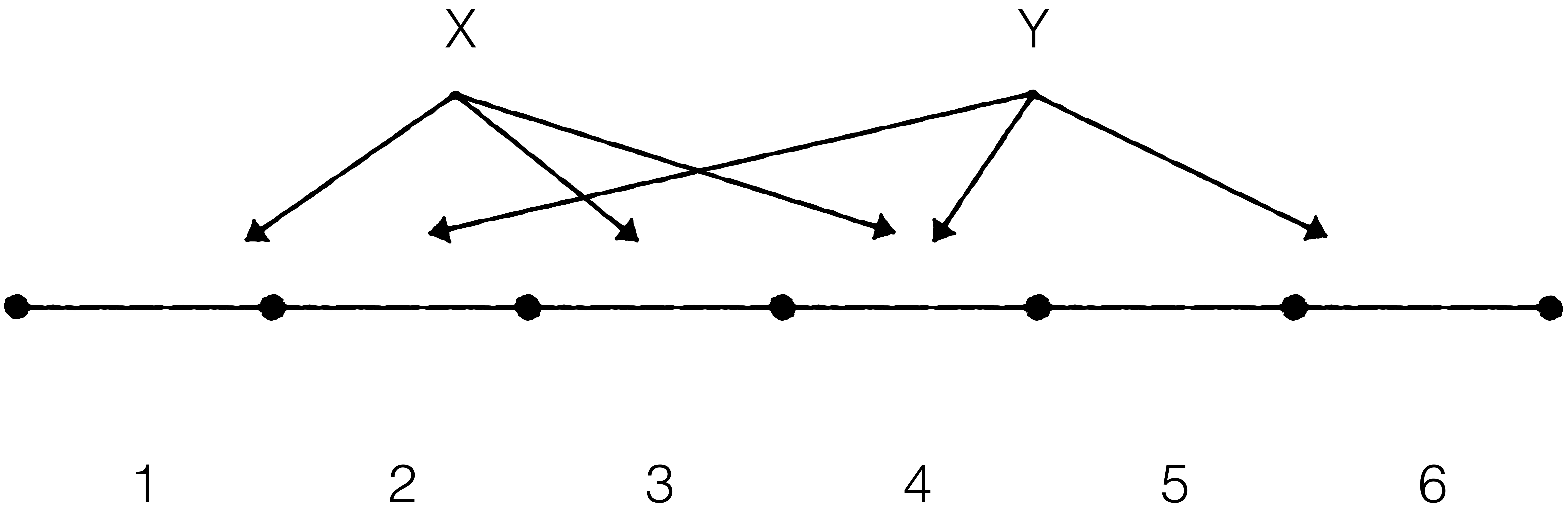
Can we improve this?

Bloom	**XOR**	Ideal
$\approx 2^{-M} \cdot \mathbf{0.69}$		$\boldsymbol{\approx 2^{-M}}$

GrafJEA20

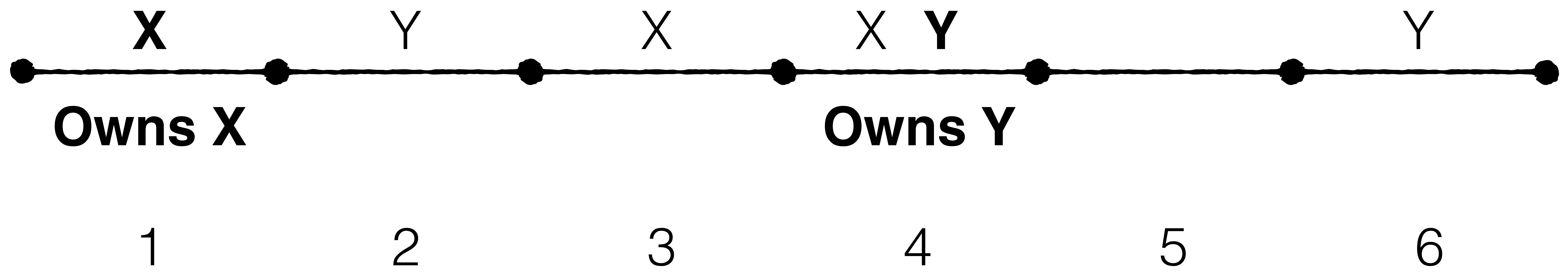
XOR Filter

Hash each entry to three buckets



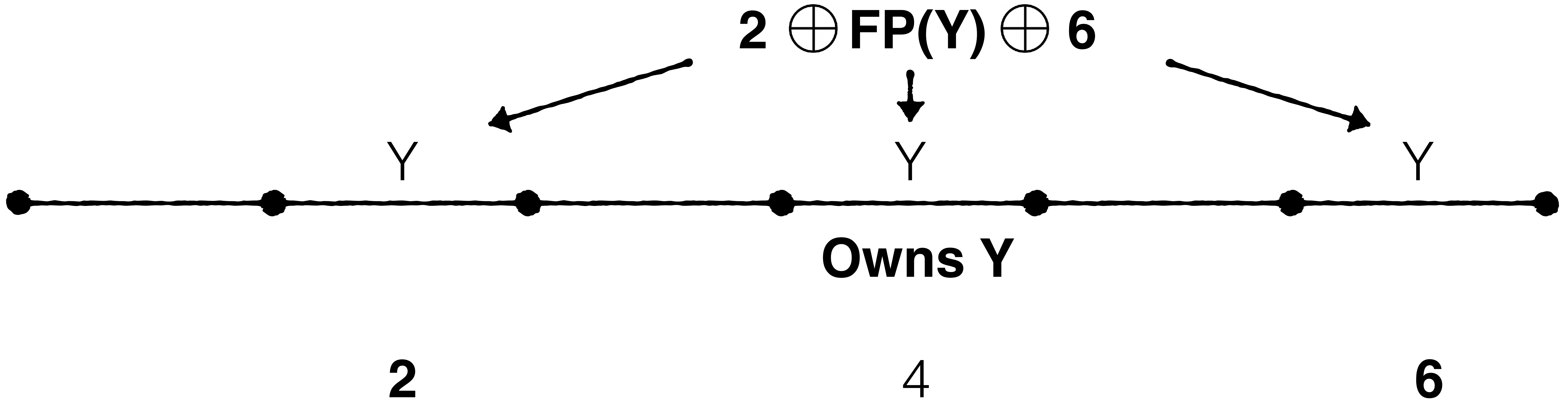
XOR Filter

Assign one bucket to own each entry



XOR Filter

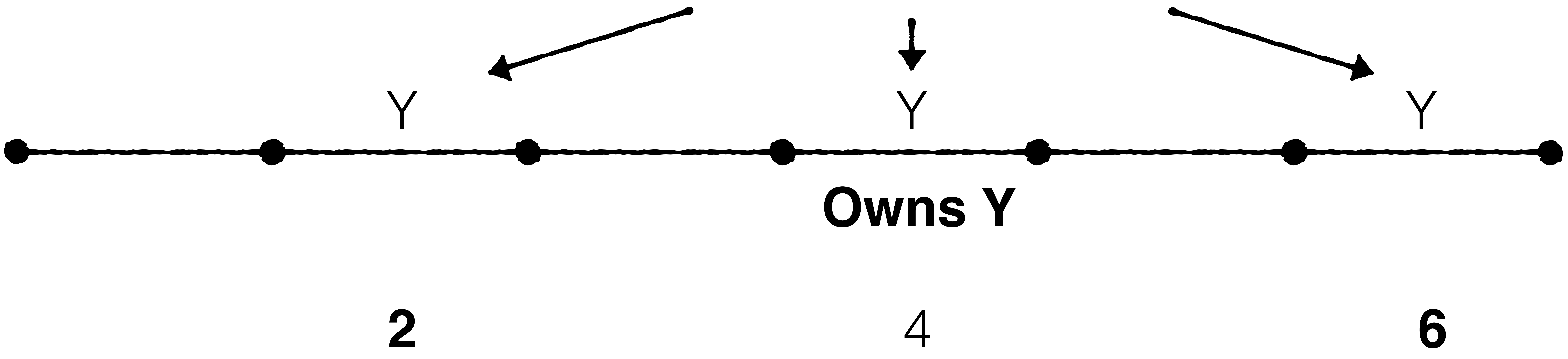
Each bucket stores XOR of fingerprint and other two buckets



XOR Filter

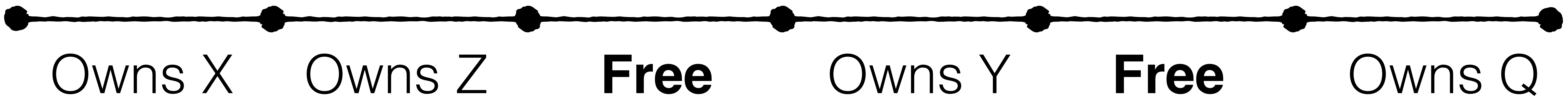
During queries, recover fingerprints by xoring three buckets

get(Y) returns true if FP(Y) = $2 \oplus 4 \oplus 6$



XOR Filter

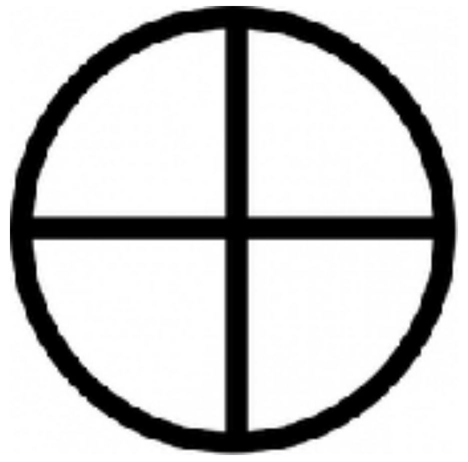
free space ensures each bucket can own one entry



Bloom

$\approx 2^{-M \cdot 0.69}$

XOR



$\boldsymbol{\approx 2^{-M \cdot 0.81}}$

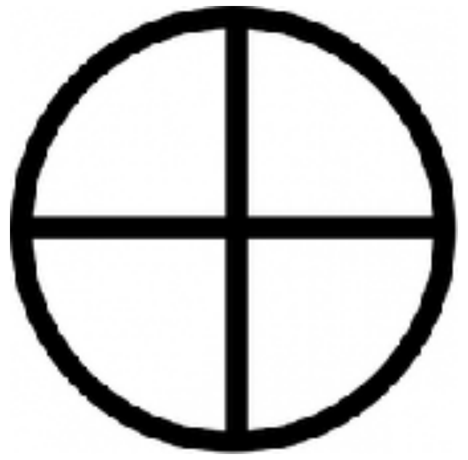
Idealized

$\approx 2^{-M}$

Bloom	XOR	**Ribbon**	Idealized
$\approx 2^{-M \cdot 0.69}$	$\approx 2^{-M \cdot 0.81}$	$\boldsymbol{\approx 2^{-M \cdot 0.92}}$	$\approx 2^{-M}$

Denser XOR filter

DillingerSEA22

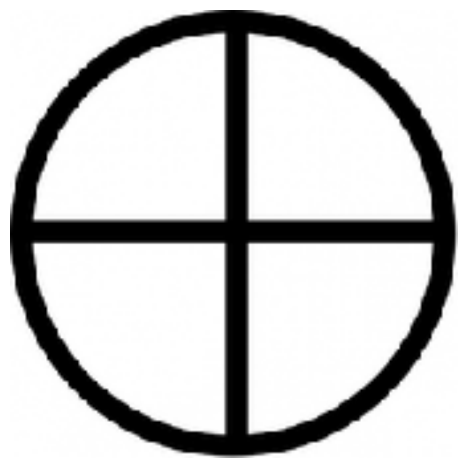
Bloom	XOR	**Ribbon**	Idealized
			
$\approx 2^{-M \cdot 0.69}$	$\approx 2^{-M \cdot 0.81}$	$\boldsymbol{\approx 2^{-M \cdot 0.92}}$	$\approx 2^{-M}$

Denser XOR filter

In RocksDB since 2020

Bloom

XOR



Ribbon

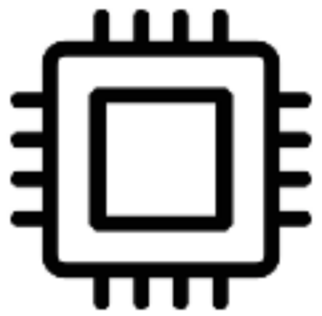
Lower CPU

Lower false positive rate

5 fronts

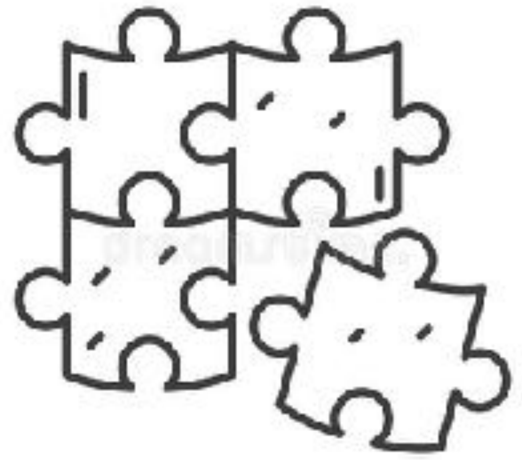
Holistic Tuning

CPU



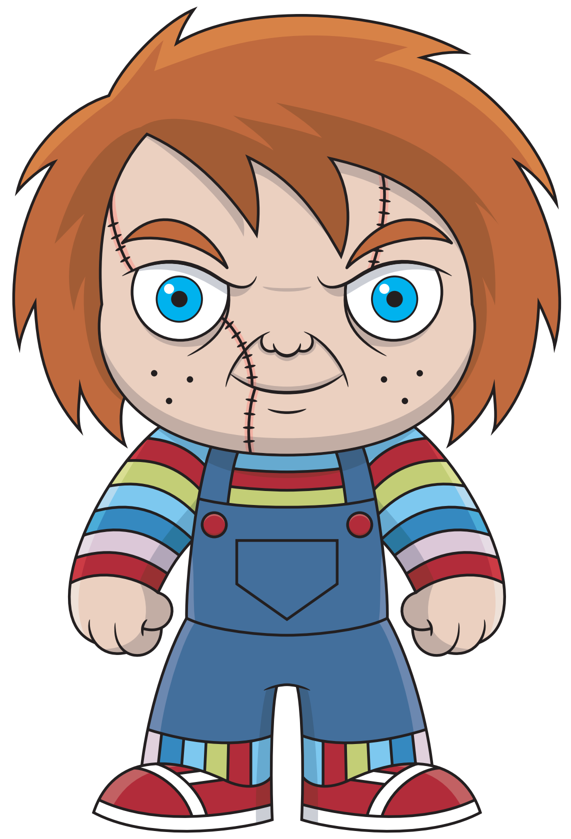
Improving Constants

Unification



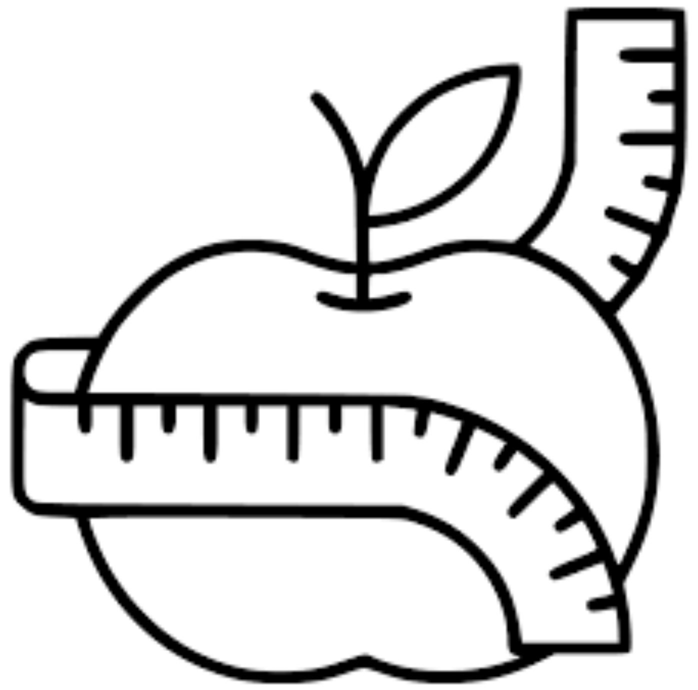
Range

Unification



Chucky

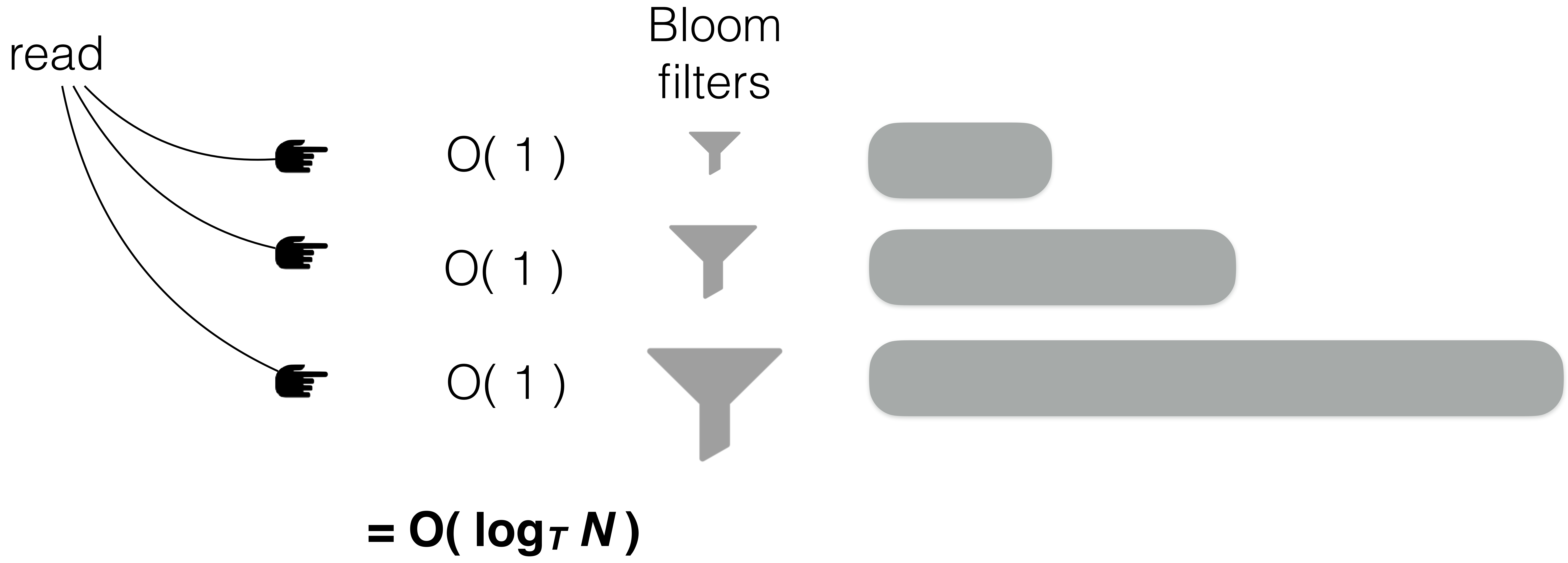
DayanSIGMOD21



SlimDB

RenVLDB17

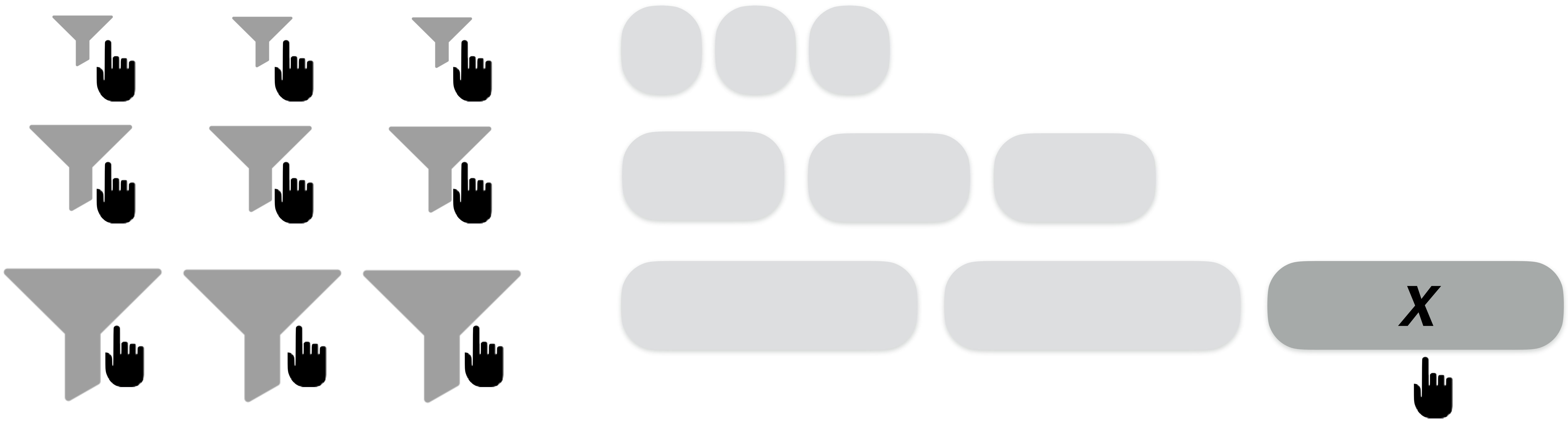
Unification



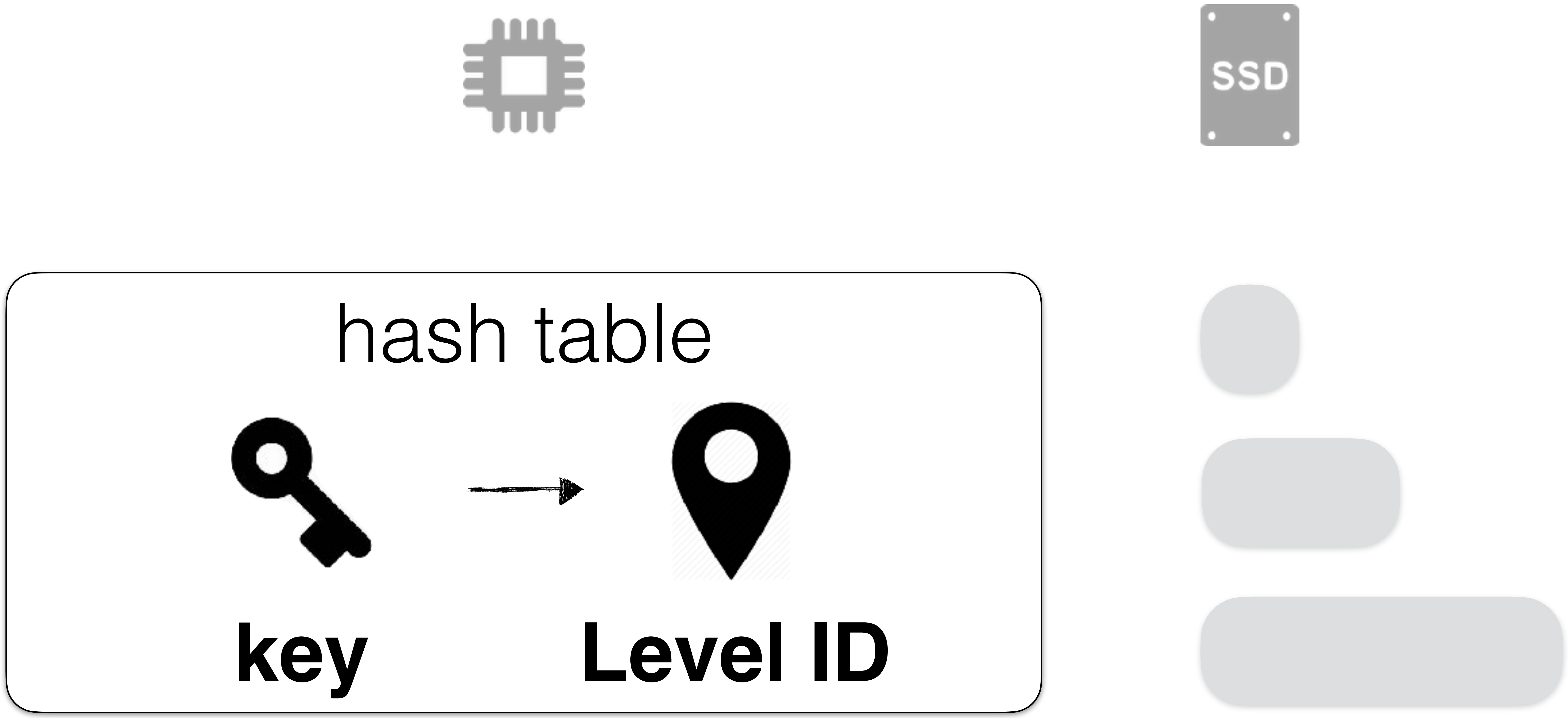
$O(\ T\ )$

$O(\ T\ )$

$O(\ T\ )$

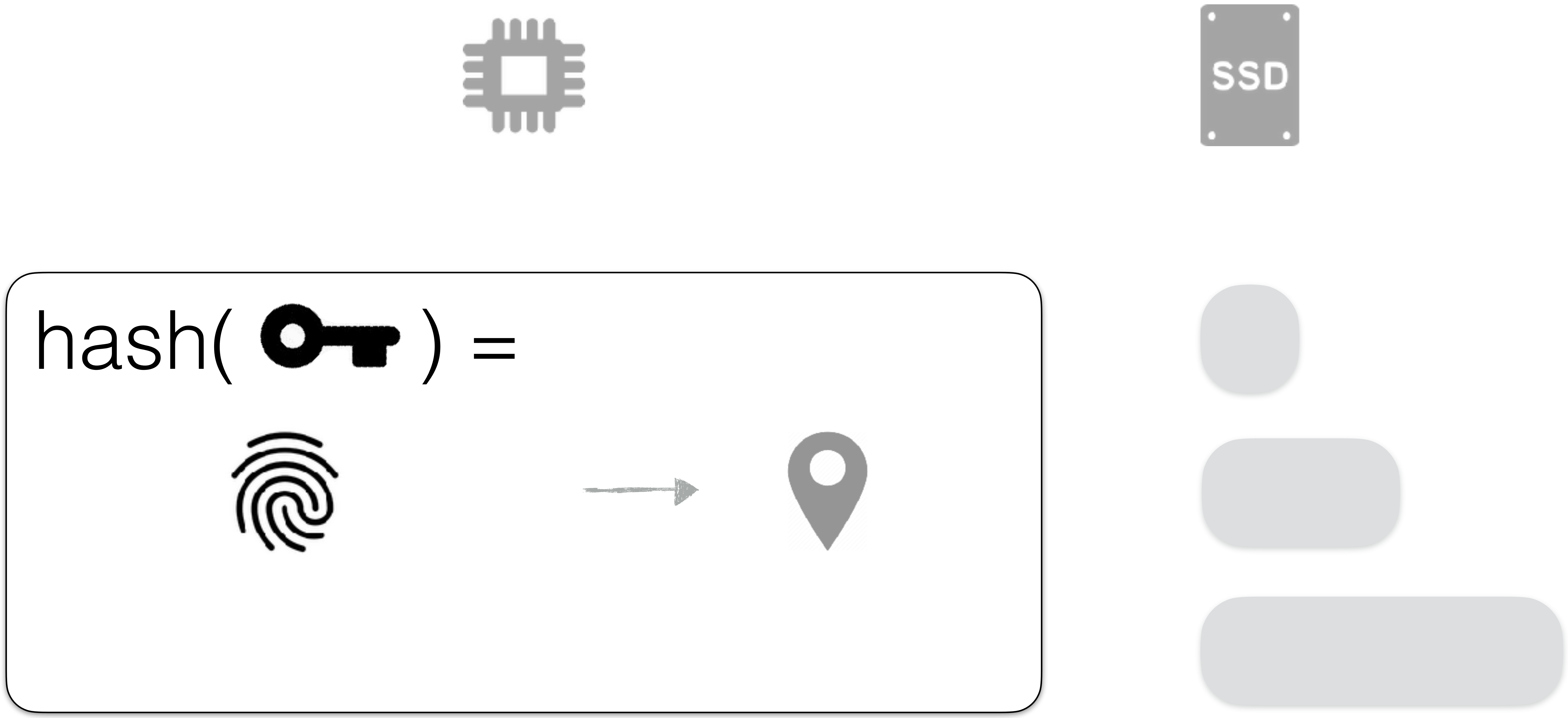


$= \mathbf{O(\ T \cdot \log_{\mathit{T}} \mathit{N}\ )}$



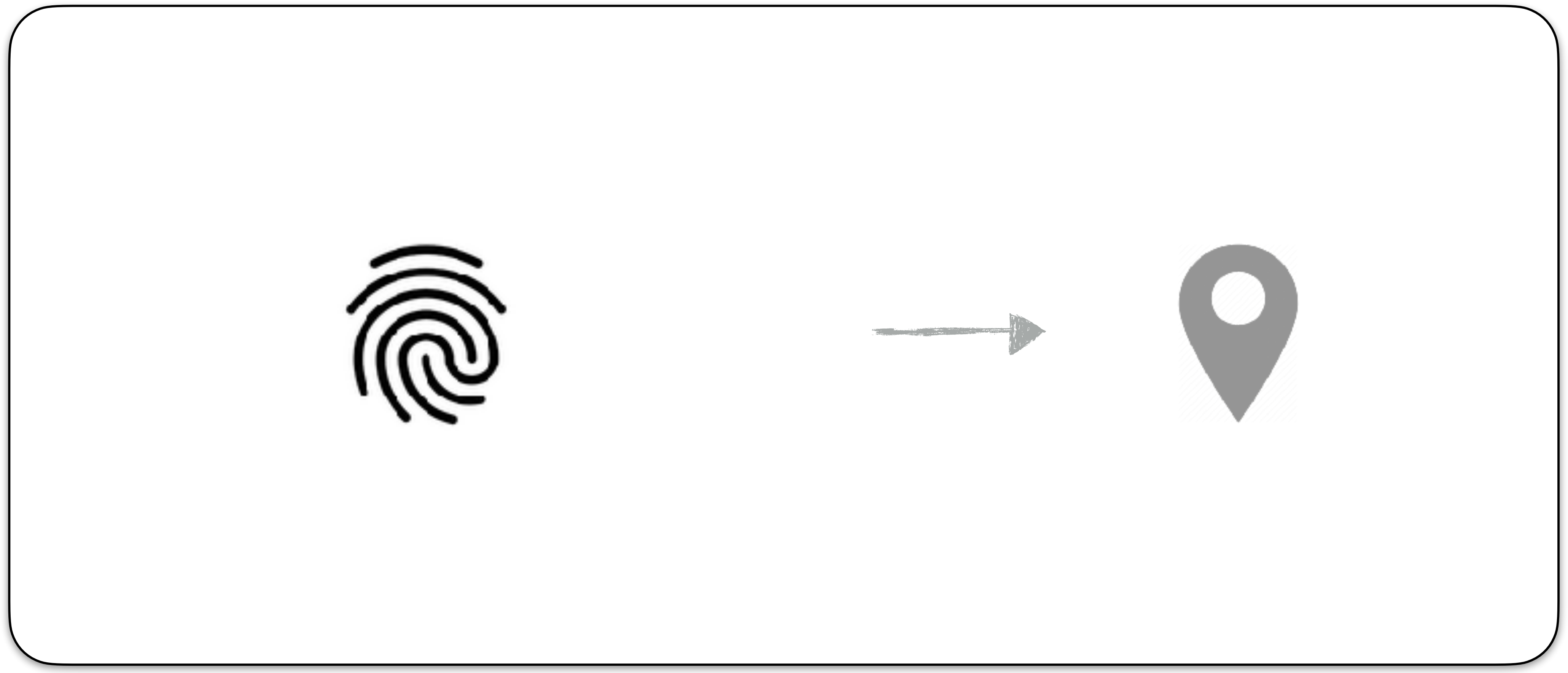
SSD
hash table
key
Level ID


SSD
hash( ) =



cuckoo filter


SSD
$O(\log_2 \log_T N)$



Binary encoding

…	…

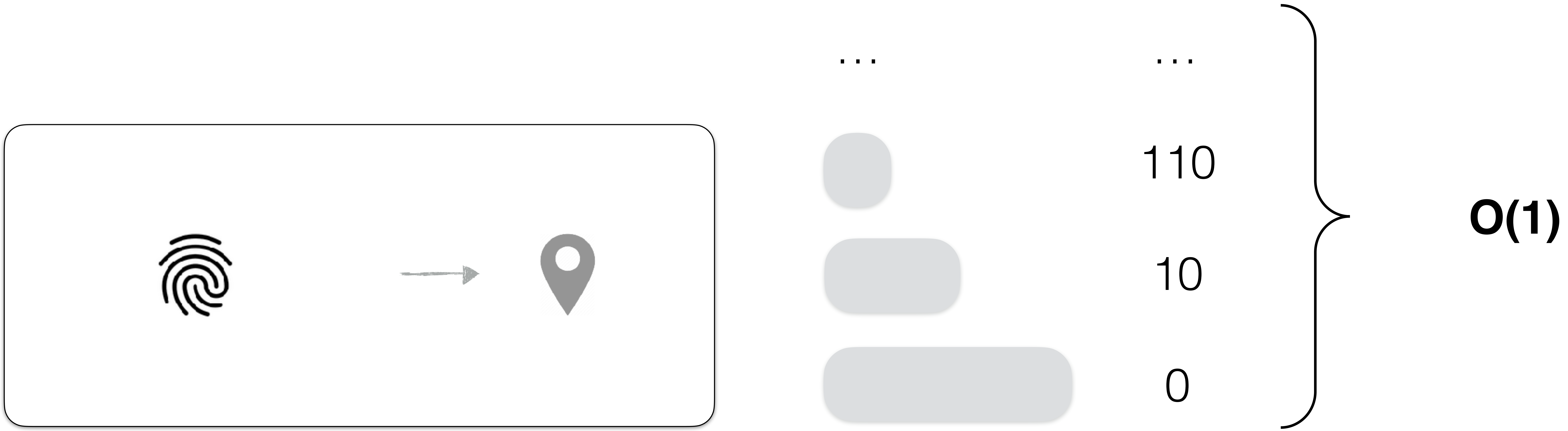
010

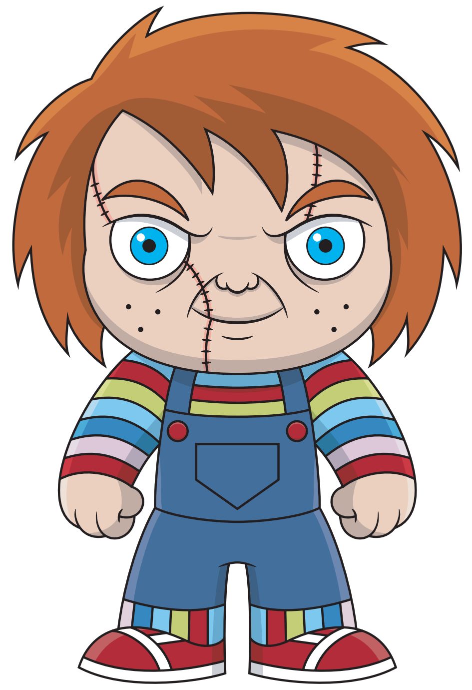
001

000

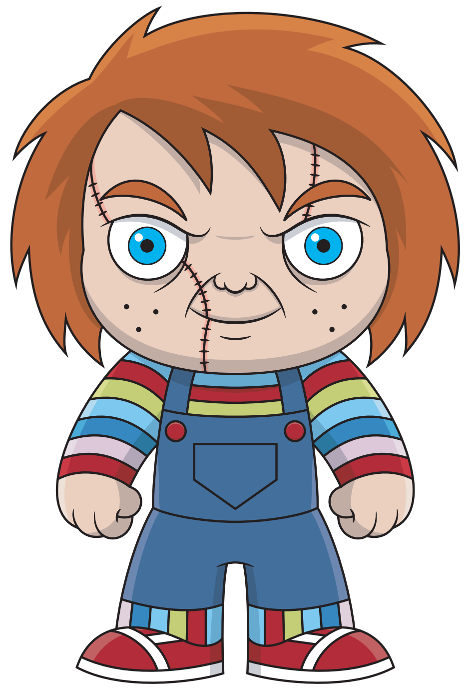
$\left.\right\}$ **$O(\log_2 \log_T N)$**

e.g., unary encoding

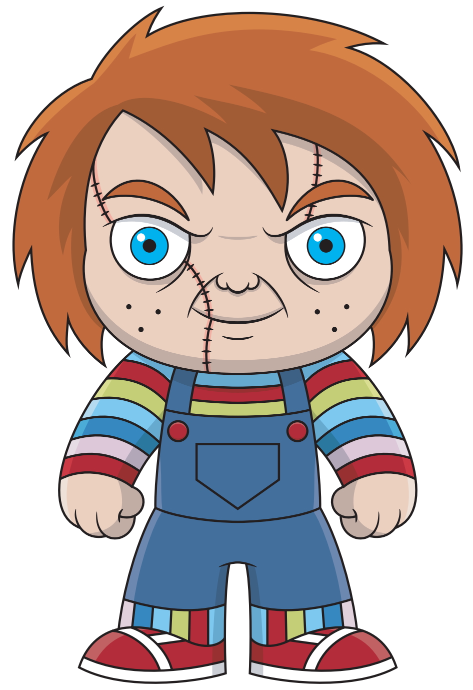
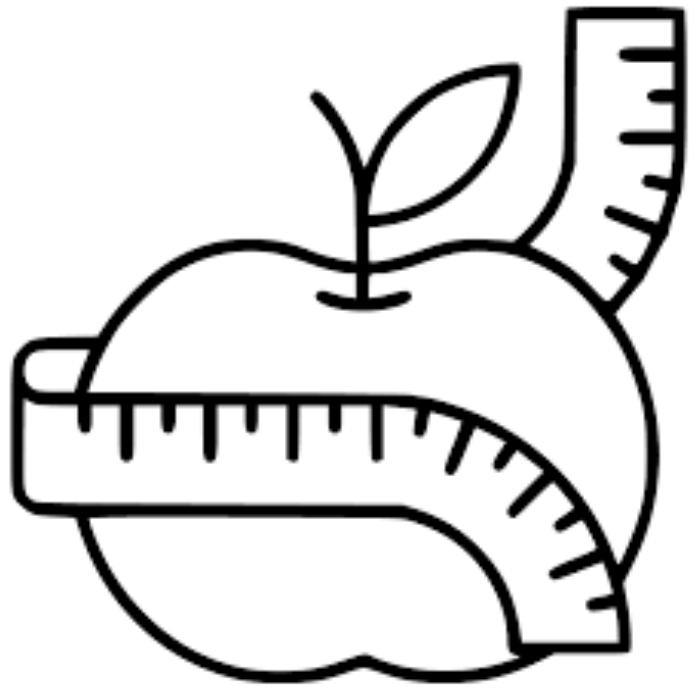




	Monkey w. Bloom	**Chucky w. Cuckoo**
Get I/O	$O(1+2^{-M \cdot \ln(2)})$	$O(1+2^{-M+3})$
Get CPU	$\mathbf{O(\log_T N)}$	$\mathbf{O(1)}$



	Monkey w. Bloom	**Chucky w. Cuckoo**
Get I/O	$O(1+2^{-M \cdot ln(2)})$	$O(1+2^{-M+3})$
Get CPU	$O(\log_T N)$	$O(1)$
Insert CPU	$O(\mathbf{T \cdot \log_T N})$	$O(\mathbf{\log_T N})$

	Chucky	**SlimDB**
Get I/O	$O(1+2^{-M+3})$	$\mathbf{O(1)}$
Memory	M	$\mathbf{M + 2^{-M} \cdot \log_2(N)}$
		Keeps full key in memory whenever fingerprints collide

5 fronts

Holistic Tuning

CPU

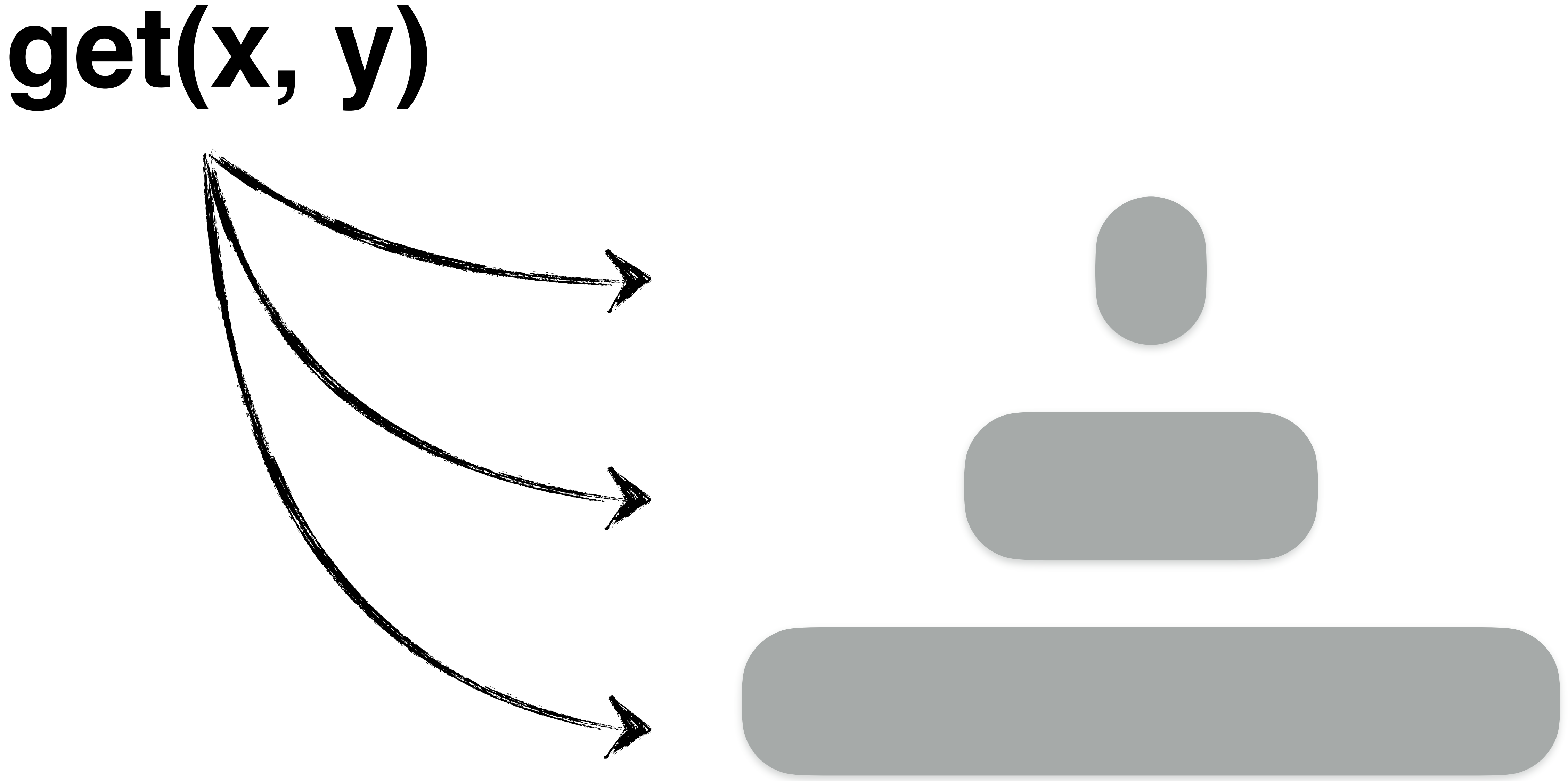
Improving Constants

Unification

Range

Range Filtering

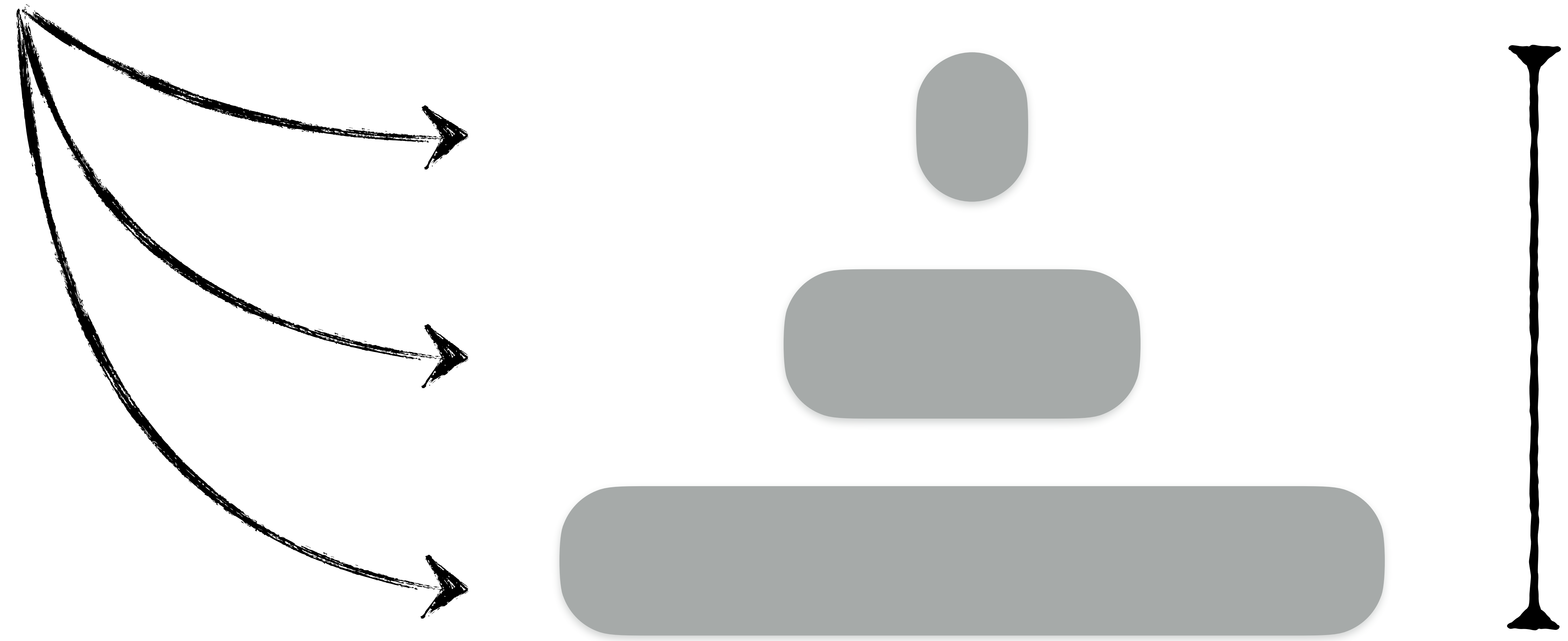
Traditional filters do not support ranges



Range Filtering

Traditional filters do not support ranges

get(x, y)



cost: $O(\log_T N)$

Range Filters

Prefix Filter

RocksDB20

Surf

ZhangSIGMOD18

Rosetta

LouSIGMOD21

Prefix Filter

RocksDB20

Users define prefix extraction method

Prefix Filter

Users define prefix extraction method

Country code

↓

USA1234

CAN9876

Prefix Filter

Users define prefix extraction method

USA

CAN

Insert prefixes

Prefix Filter

Users define prefix extraction method

USA

CAN

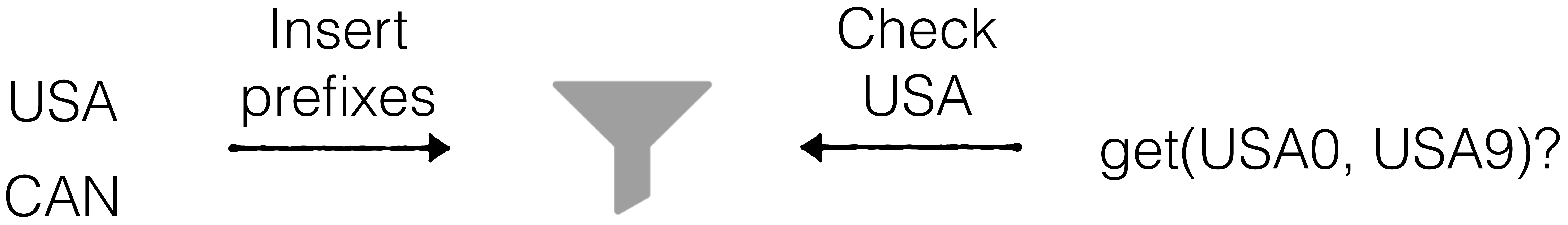
Insert prefixes

Check **USA**

get(**USA**0**, USA**9)?

Prefix Filter

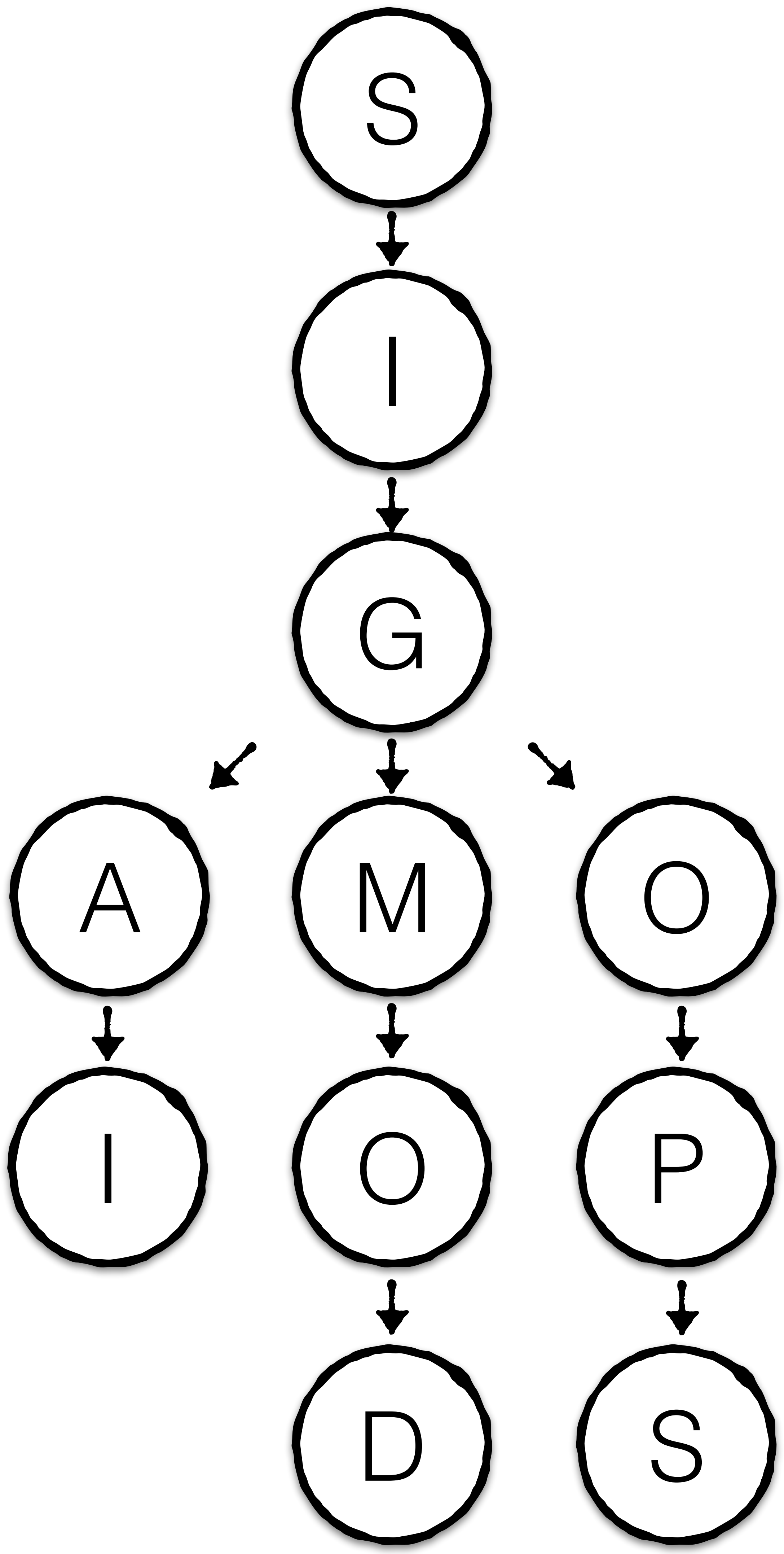
Users define prefix extraction method



Non-generic and requires API extension

Surf

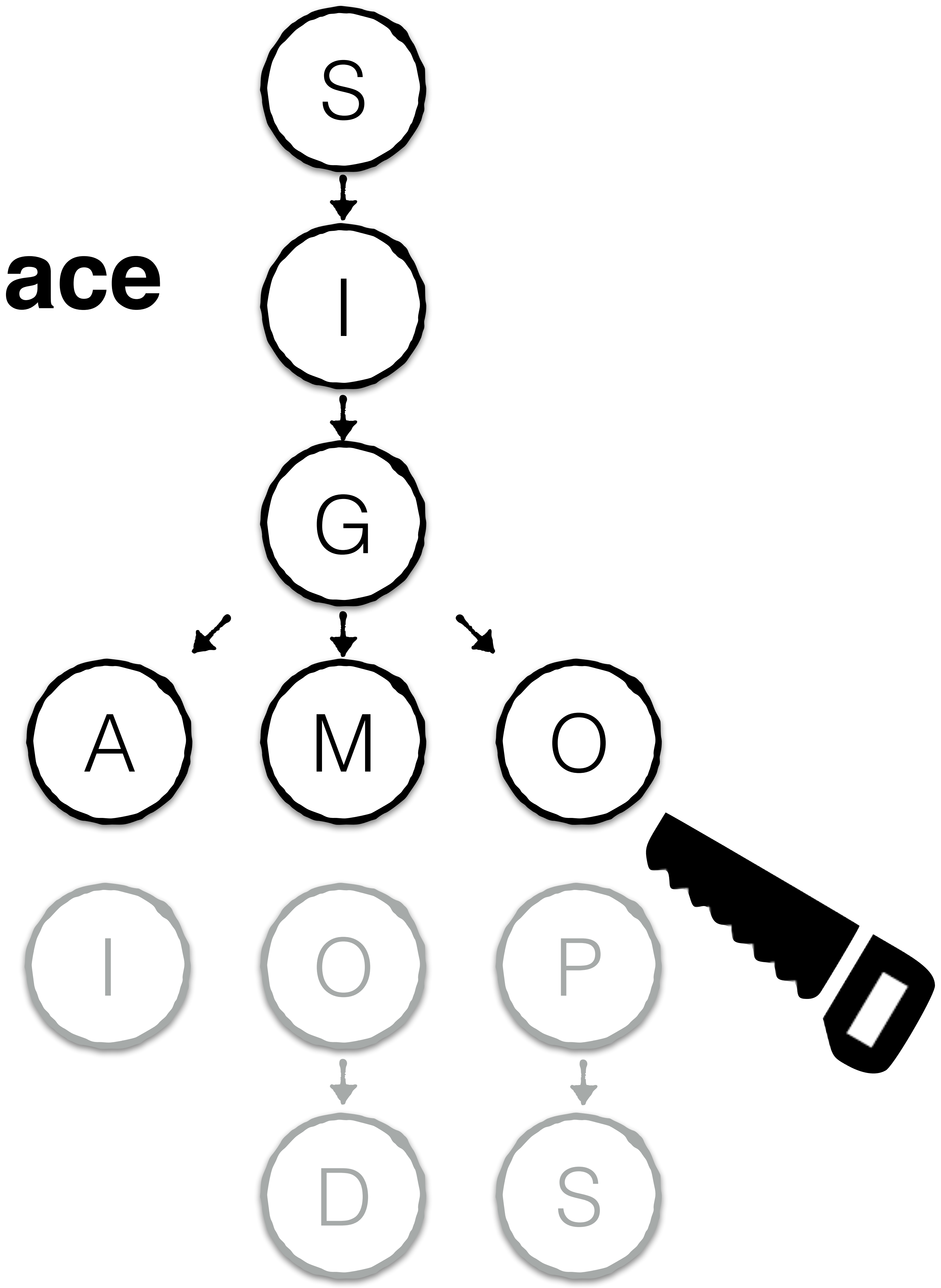
A trie of all keys



Surf

A trie of all keys

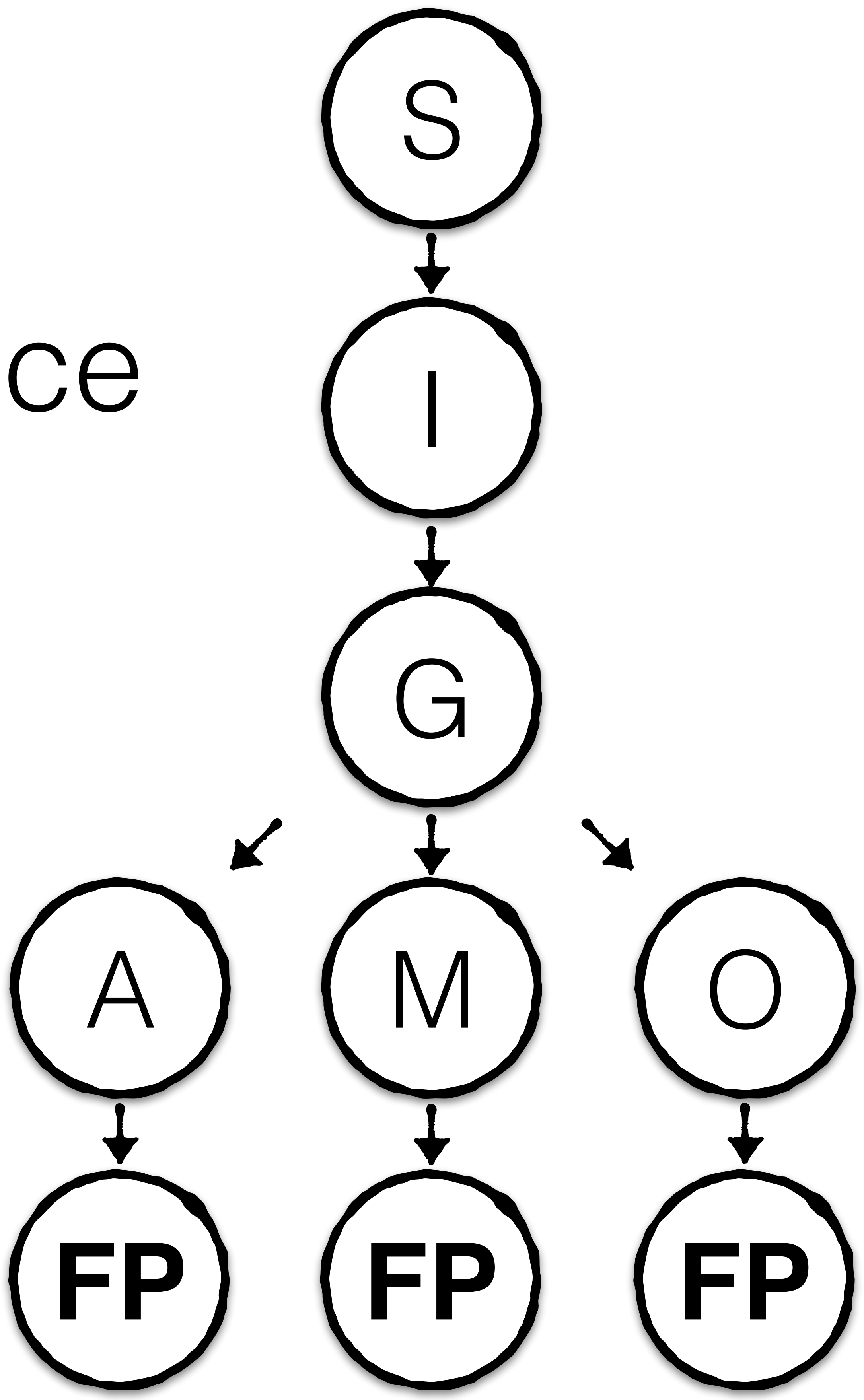
Truncated to reduce space



Surf

A trie of all keys

Truncated to reduce space

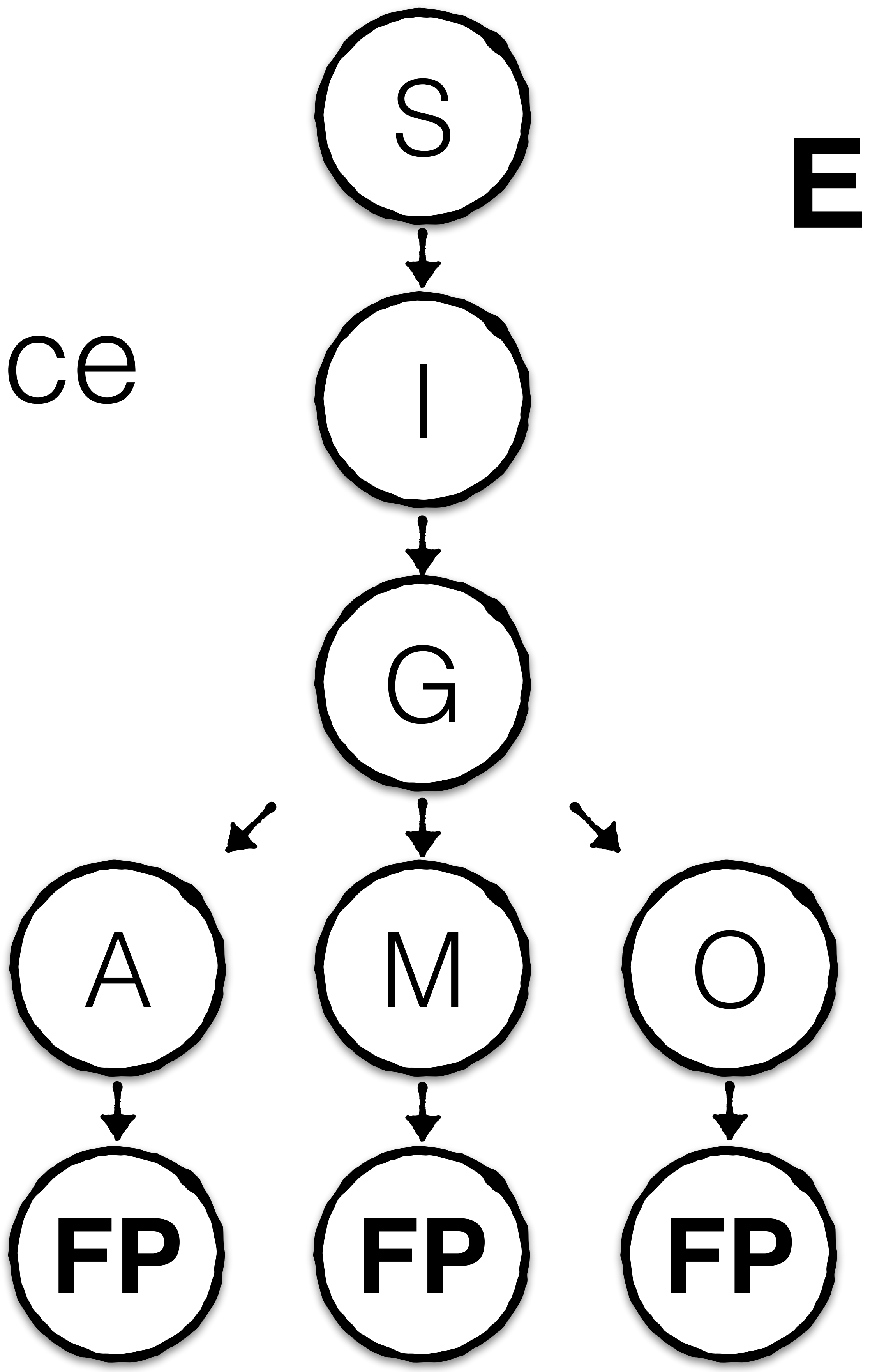


Add fingerprint for point reads

Surf

A trie of all keys

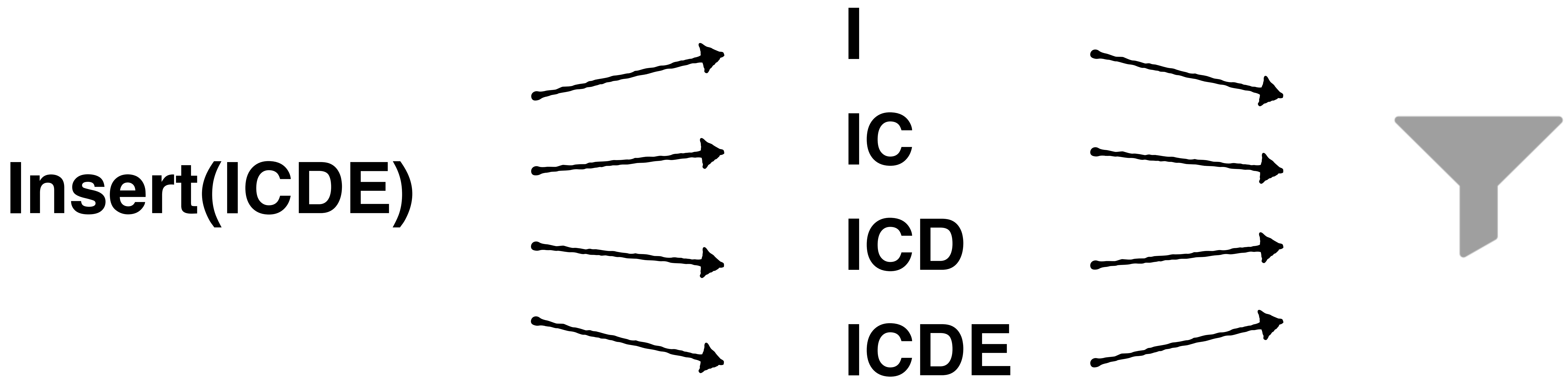
Truncated to reduce space



Encoded as succinct trie with rank & select

Add fingerprint for point reads

Rosetta



Add all prefixes of all keys to a Bloom filter

Rosetta

get(**ICD**E, **ICD**F) **ICD** 

Check largest common prefixes

Rosetta

get(ICD**E**, ICD**F**)

ICD

ICDE

ICDF

Check largest common prefixes

Add more fine-grained checks to reduce false positive rate

Surf

Rosetta

Better long range **Better short range**

Range Filters

Prefix Filters

RocksDB20

Surf

ZhangSIGMOD18

Rosetta

LouSIGMOD21

Remix

ZhongFAST21

Snarf

VaidyaVLDB22

Proteus

KnorrSIGMOD22

BloomRF

MößnerEDBT23

REncoder

WangICDE23

Outline

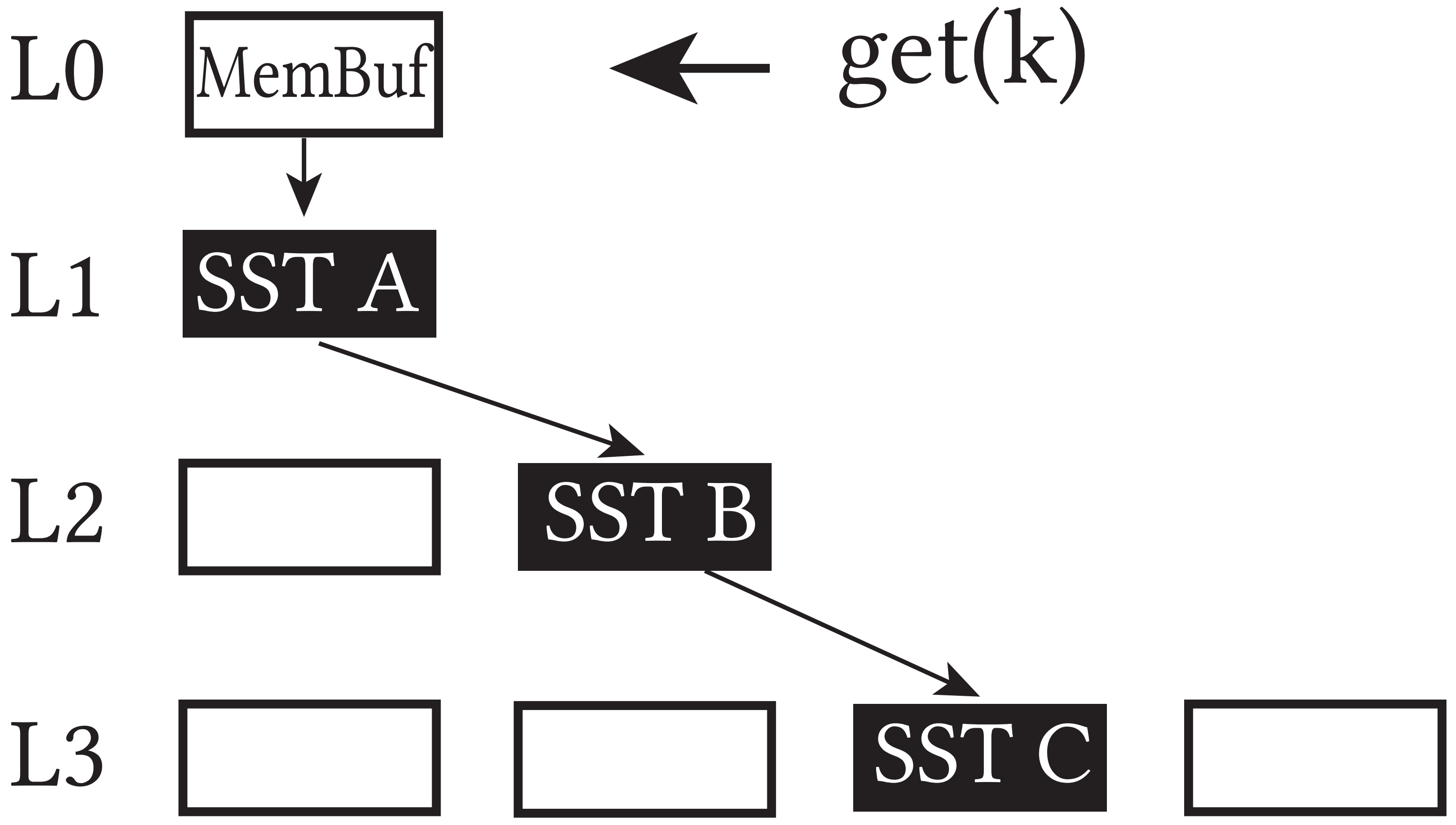
Part 1: **LSM Basics**

Part 2B: **Read Optimizations in LSMs**

Part 3: **Navigating the LSM Design Space**

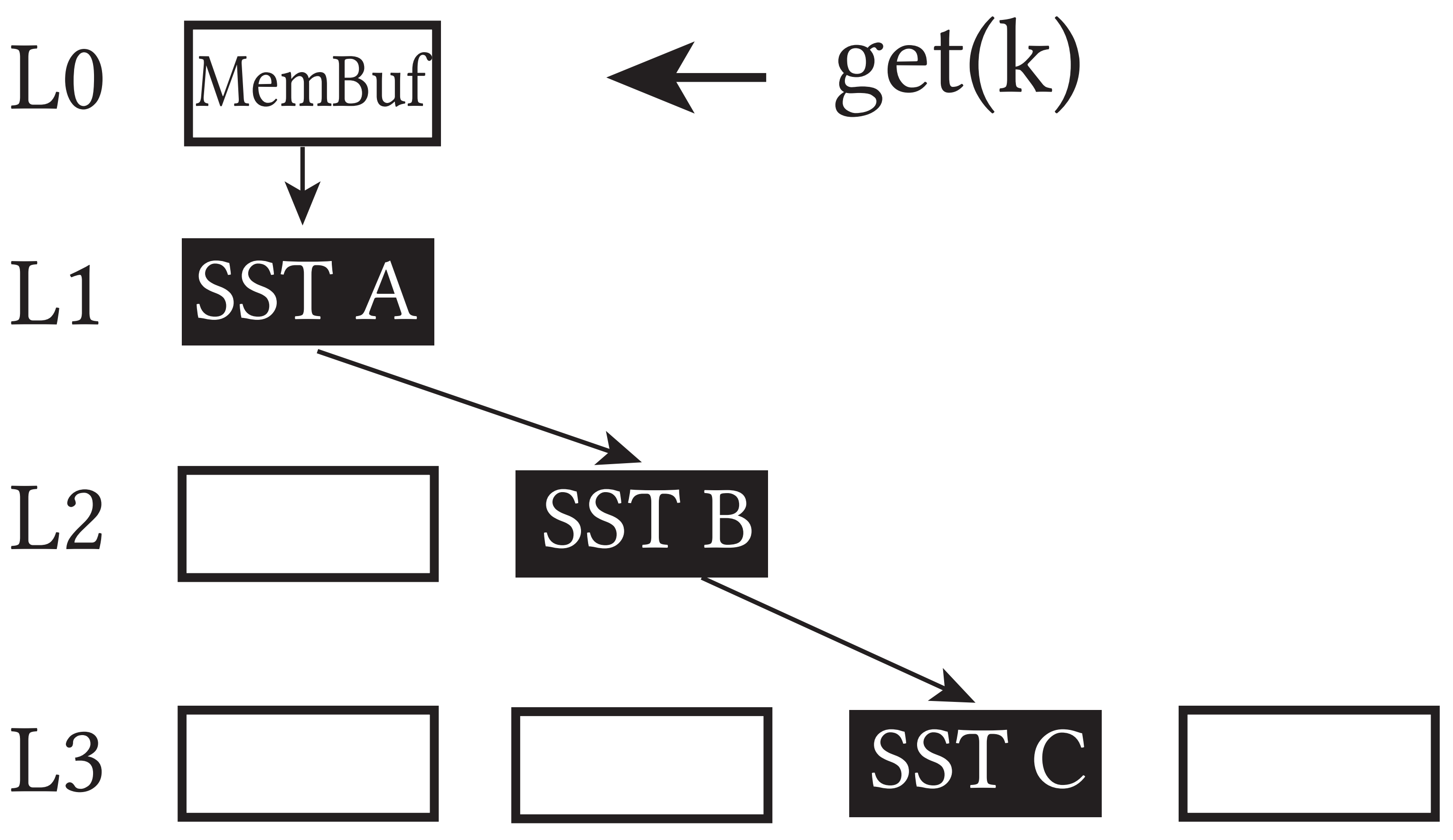
Reducing CPU Overheads in LSMs

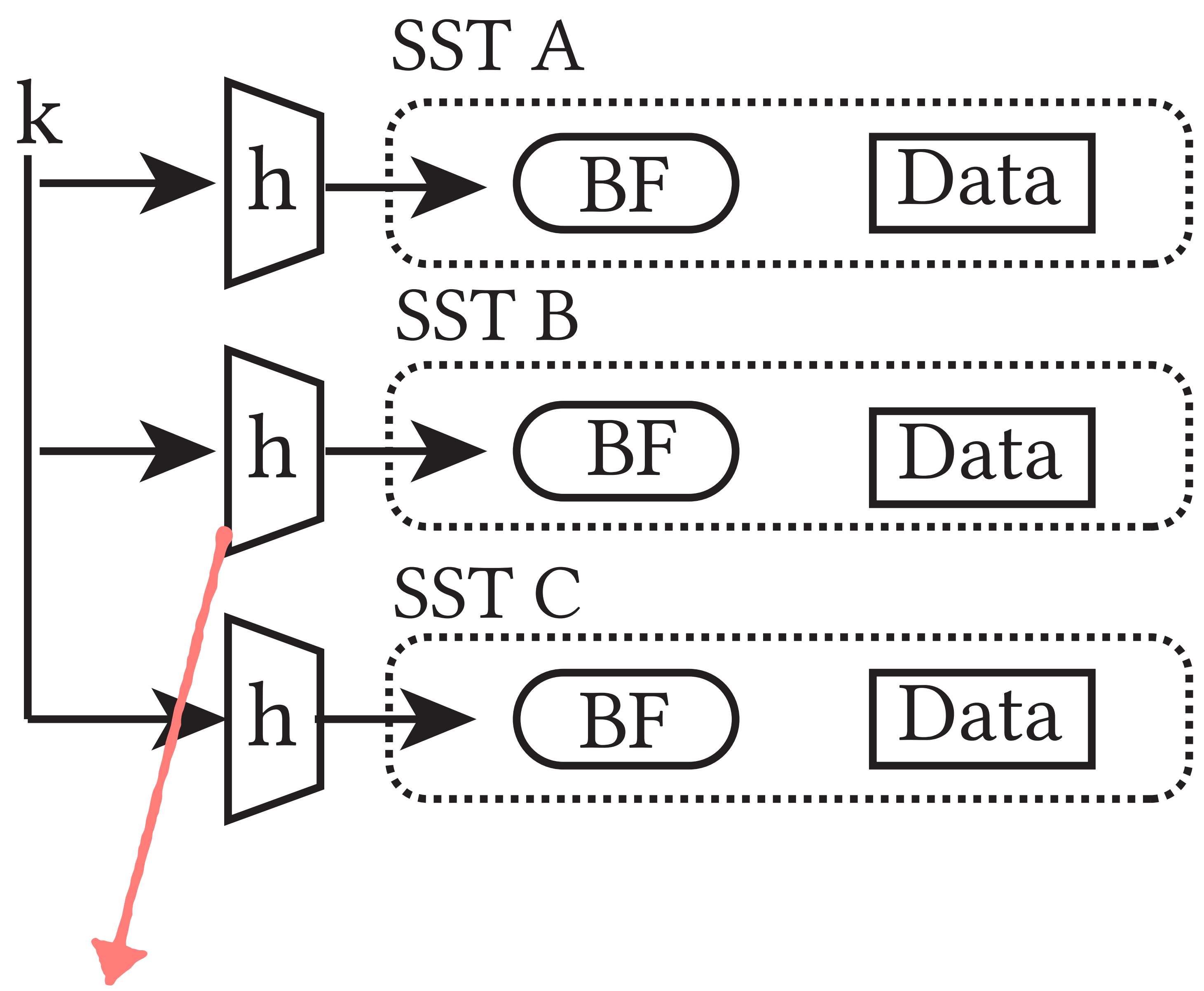
For every query ...



Reducing CPU Overheads in LSMs

For every query ...

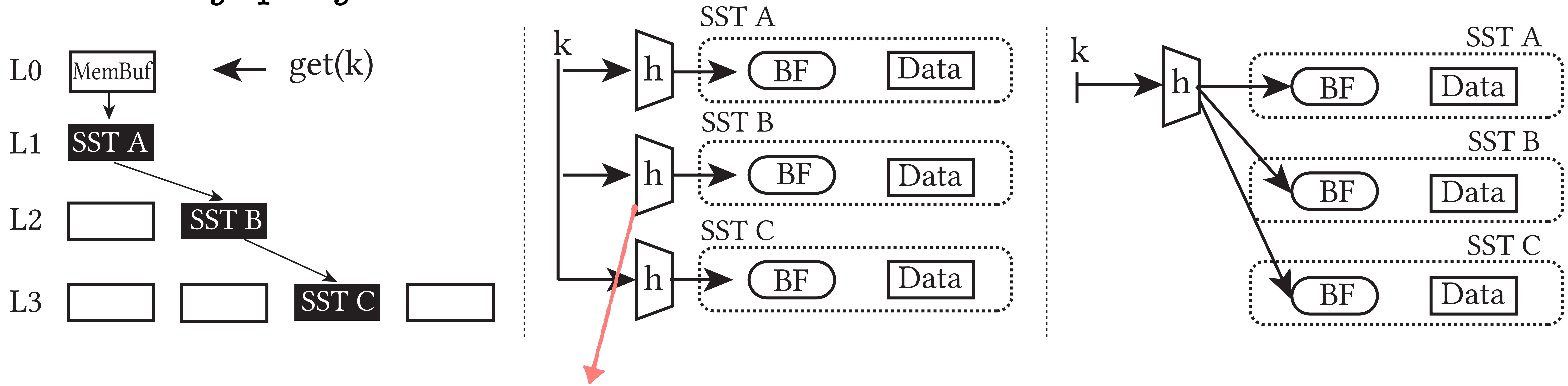




The same hash function is calculated **O(L)** times

Reducing CPU Overheads in LSMs

For every query ...

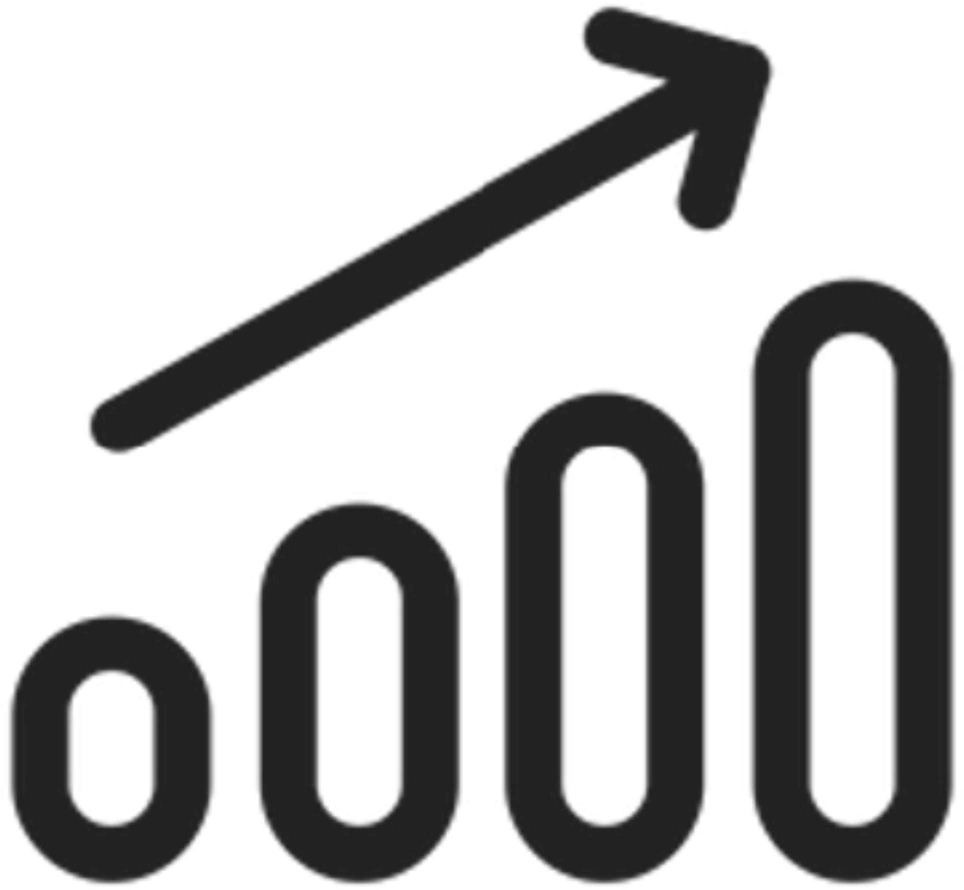


The same hash function is calculated **O(L)** times

Each key is hashed **O(1)** times

ZhuDaMoN21

Filters under **Memory Pressure**



data size ↑

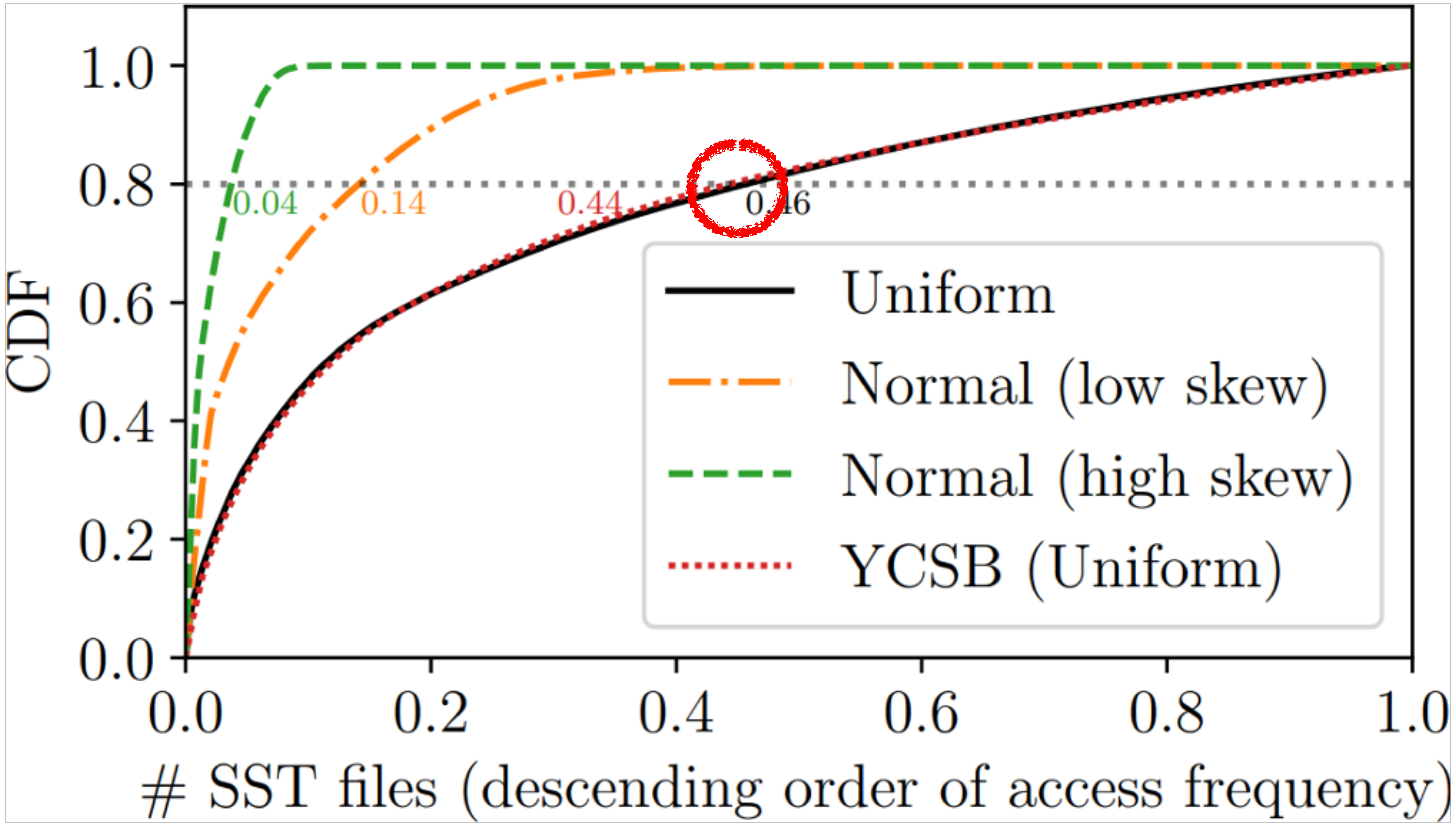
for 1TB data,
1.3GB filter &17.2GB index

1KB entry, 64B key, BPK=10

*price drop from **2010** to **today***
SSD: 60x DRAM: 10x

Filters under **Memory Pressure**

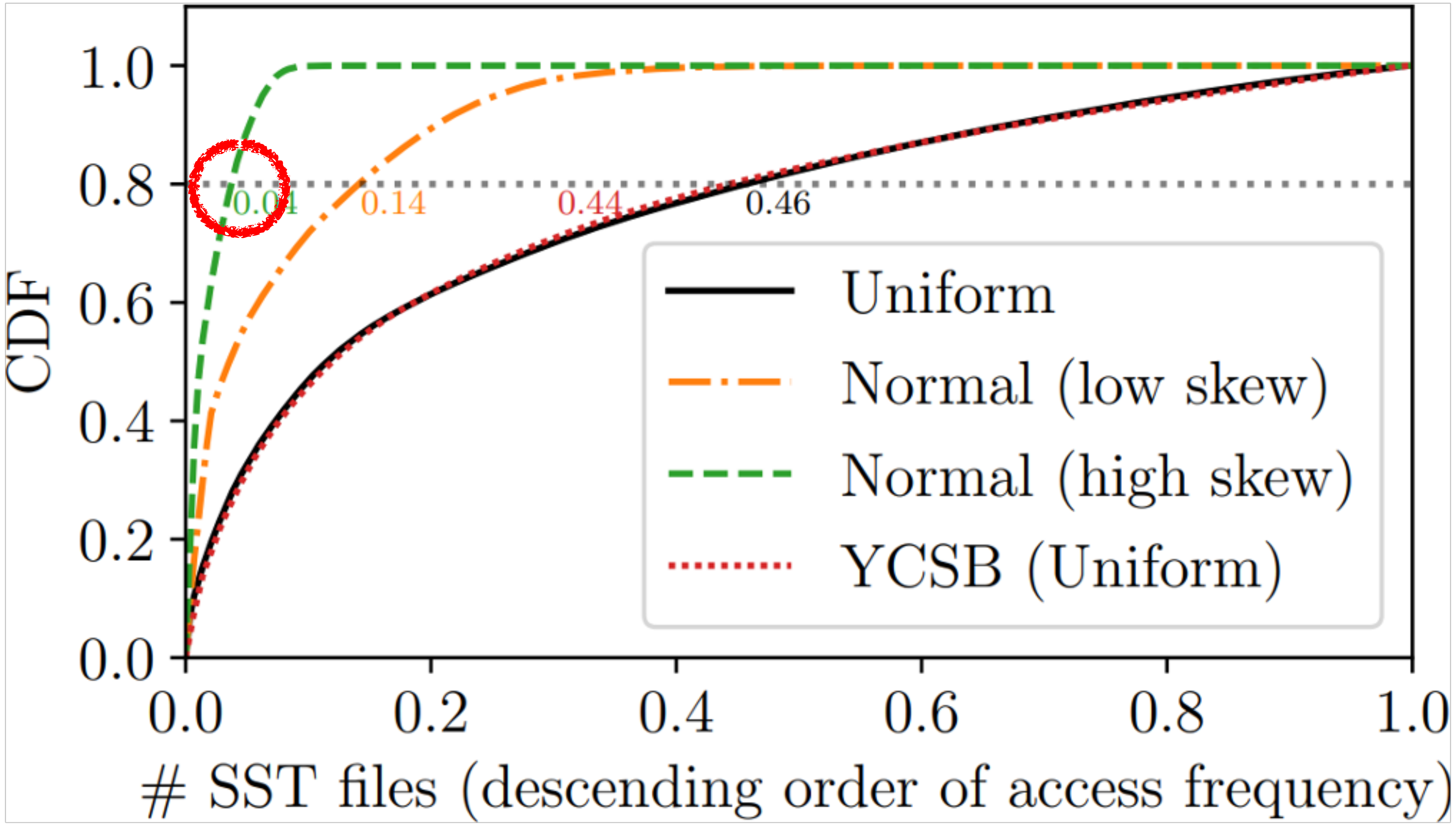
MunADMS22



Even in a **perfectly uniform** workload,
80% of the queries access **45% of the files**

Filters under **Memory Pressure**

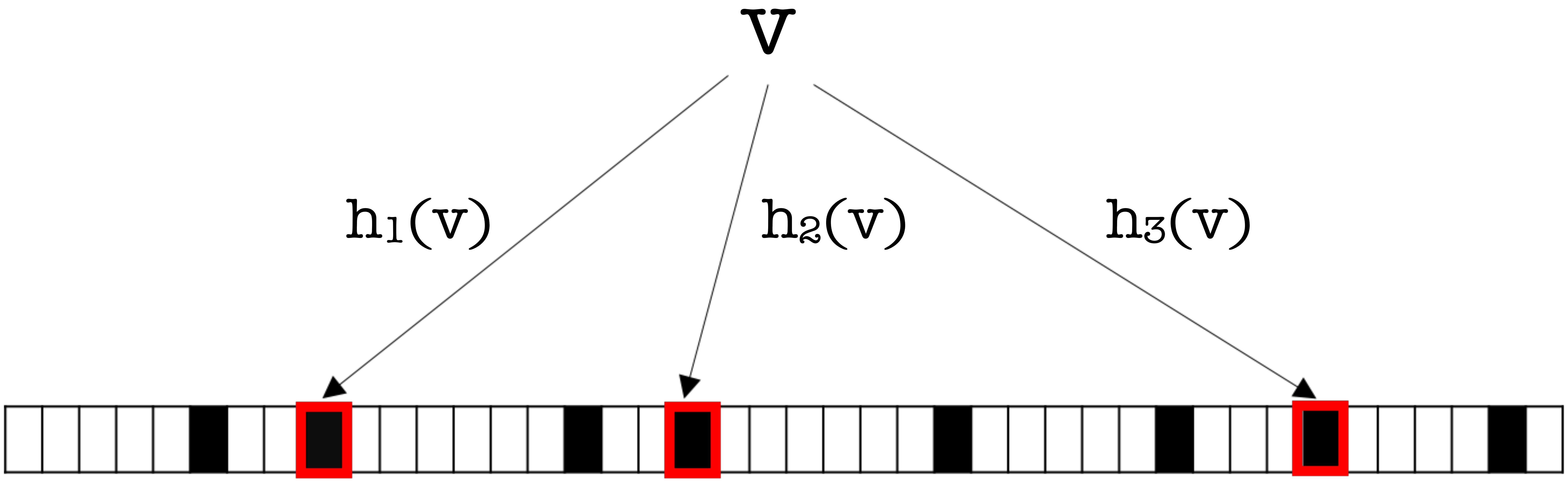
MunADMS22



For a **skewed** workload,
80% of the queries access less than **5% of the files**

Filters under **Memory Pressure**

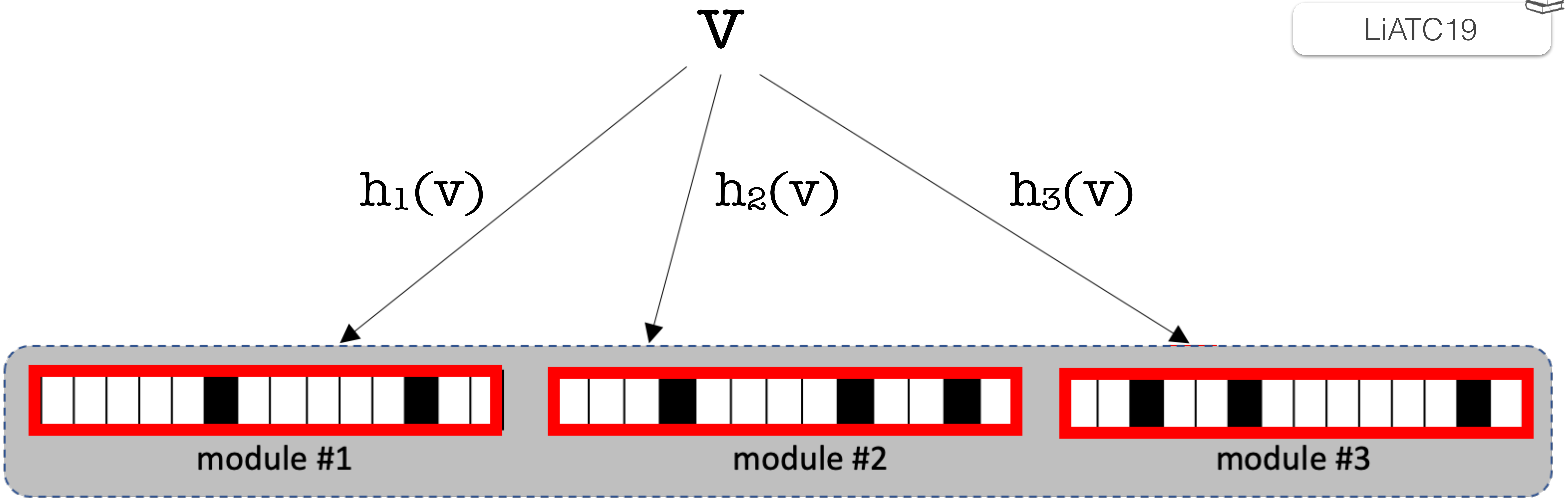
MunADMS22



Filters under **Memory Pressure**

MunADMS22

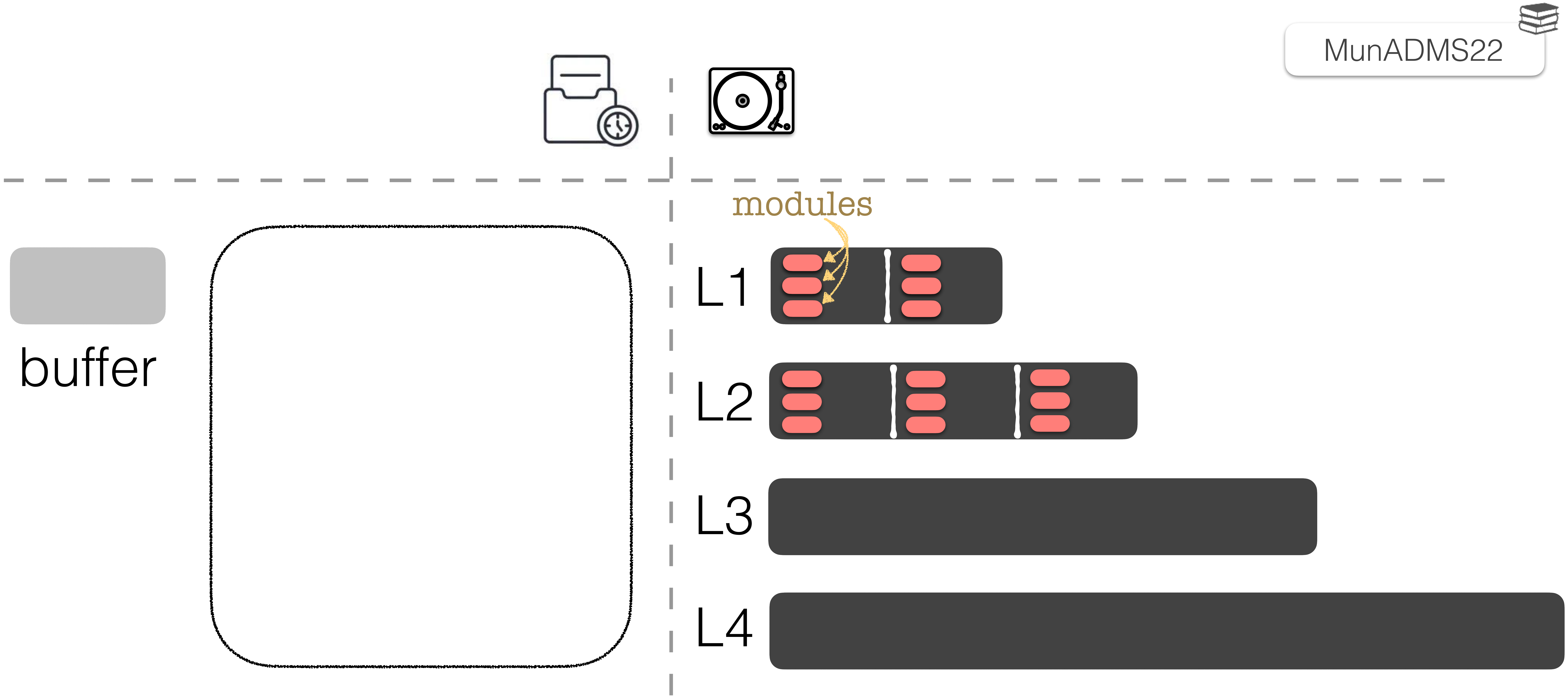
LiATC19



Modular Bloom filter is a collection of smaller Bloom filters

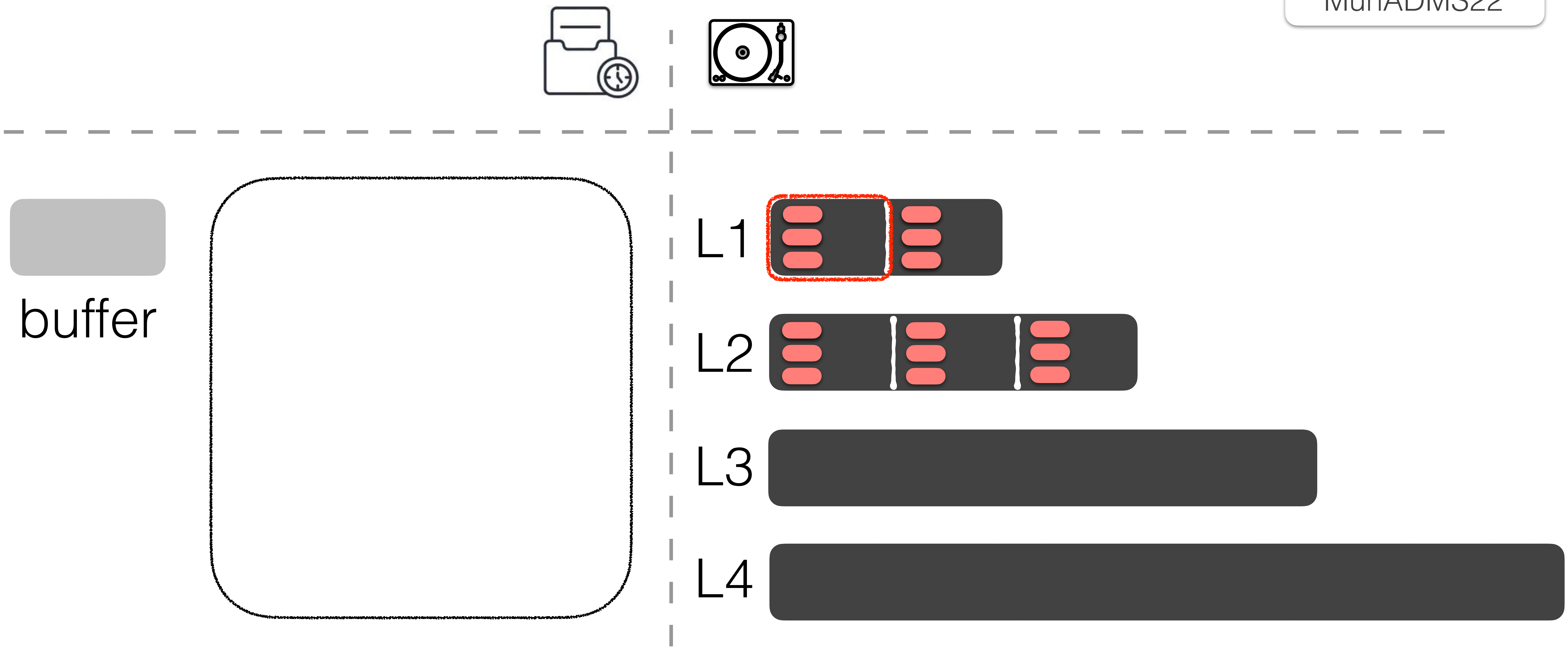
Elastic Bloom filter also works based on the same principle

Filters under **Memory Pressure**



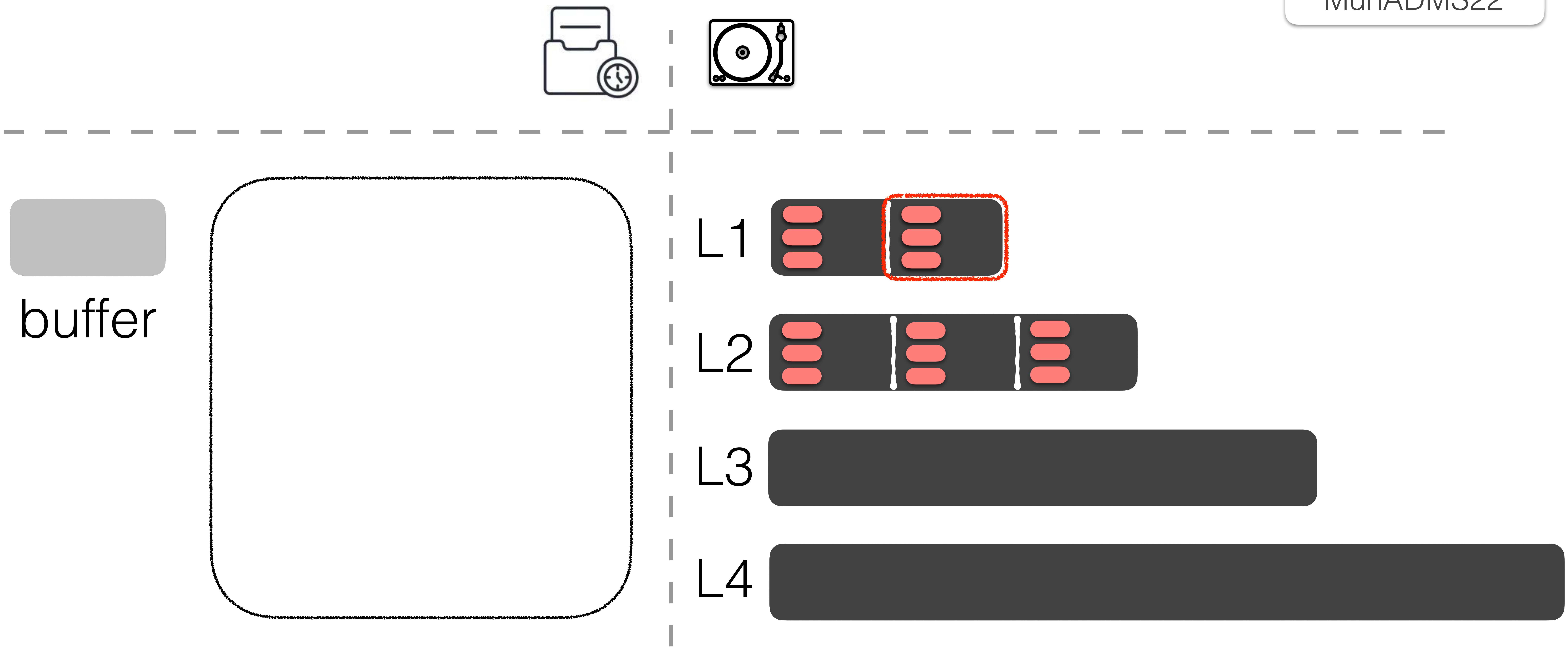
Filters under **Memory Pressure**

MunADMS22



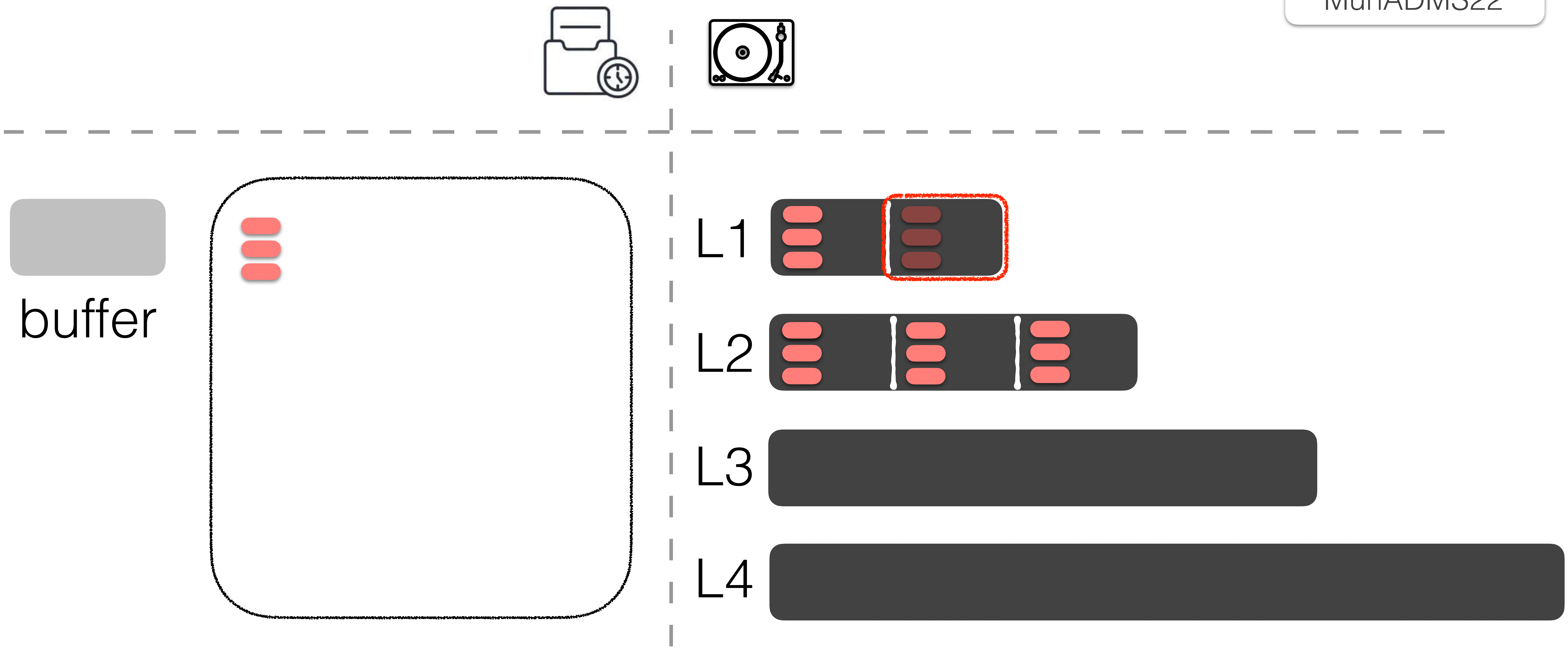
Filters under **Memory Pressure**

MunADMS22



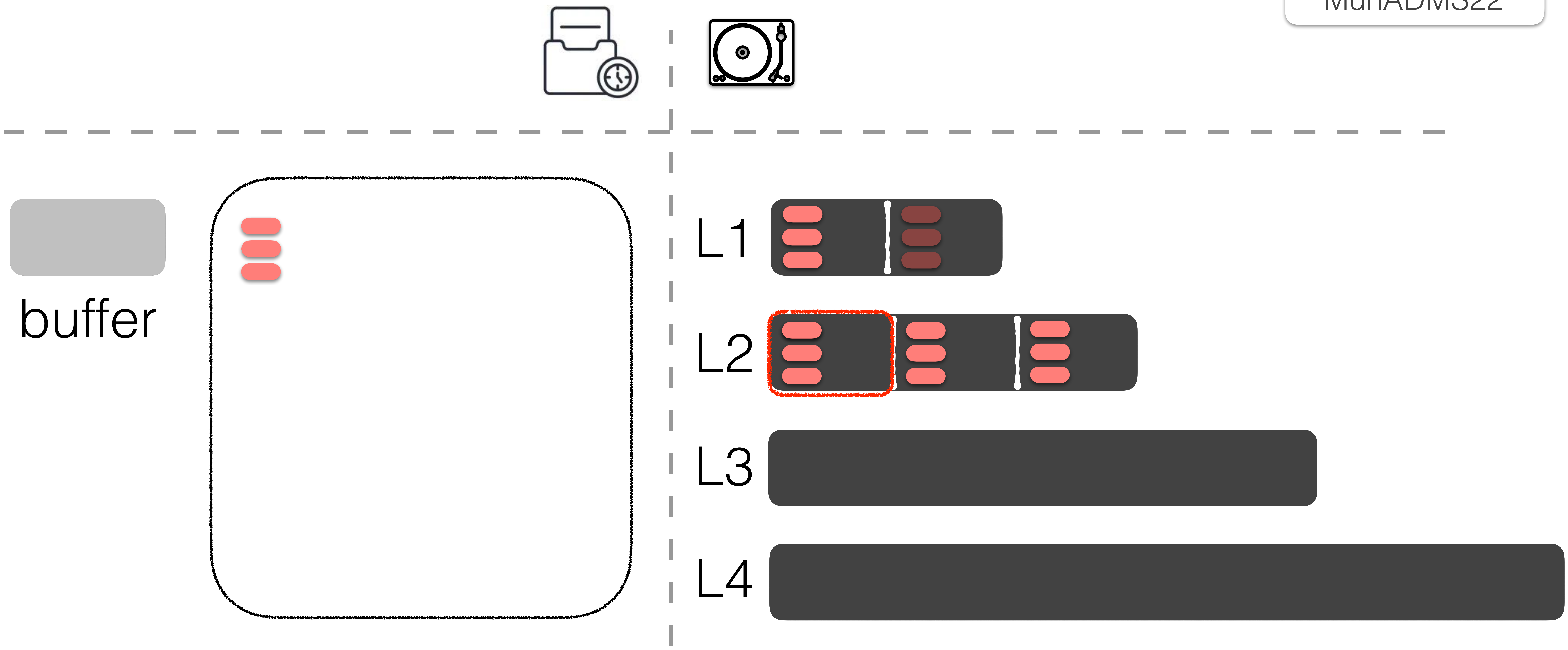
Filters under **Memory Pressure**

MunADMS22

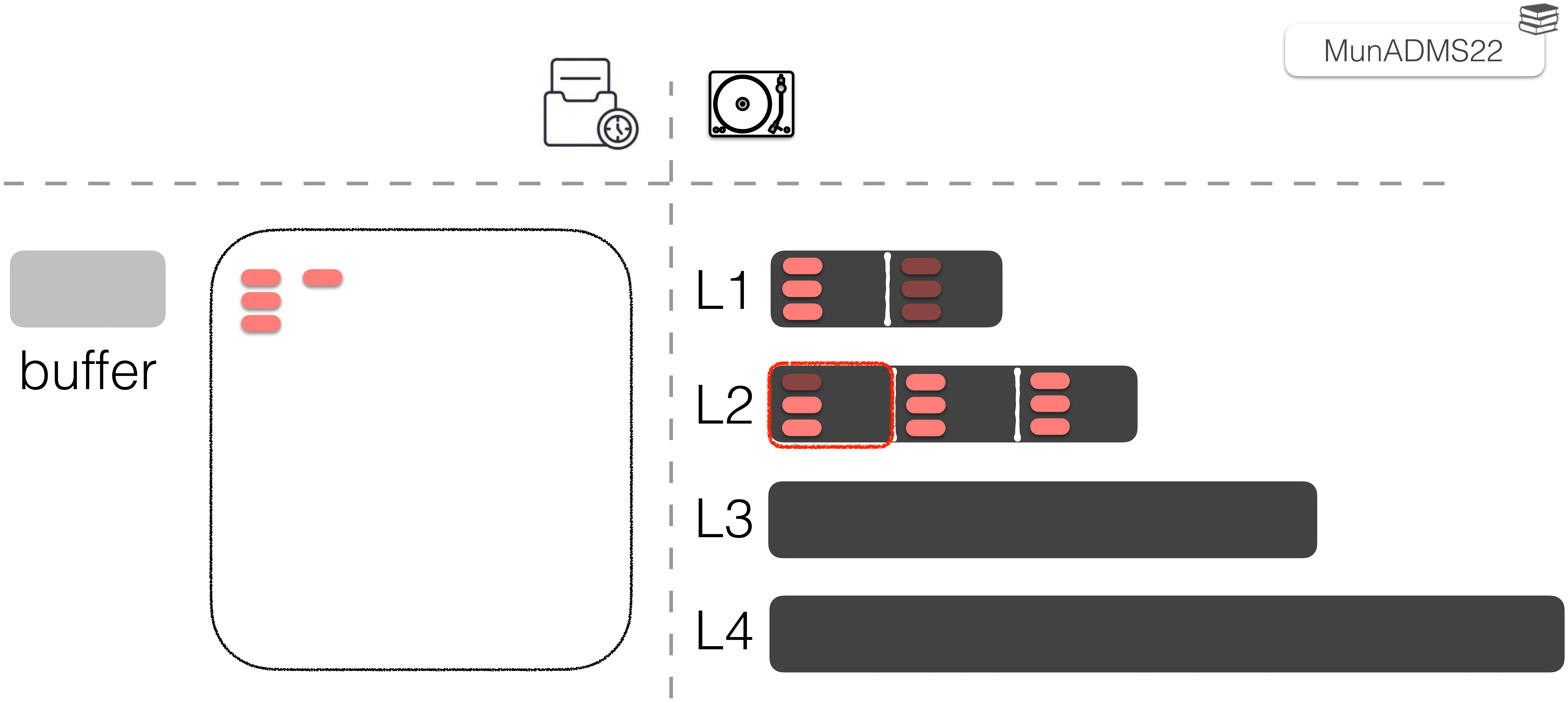


Filters under **Memory Pressure**

MunADMS22

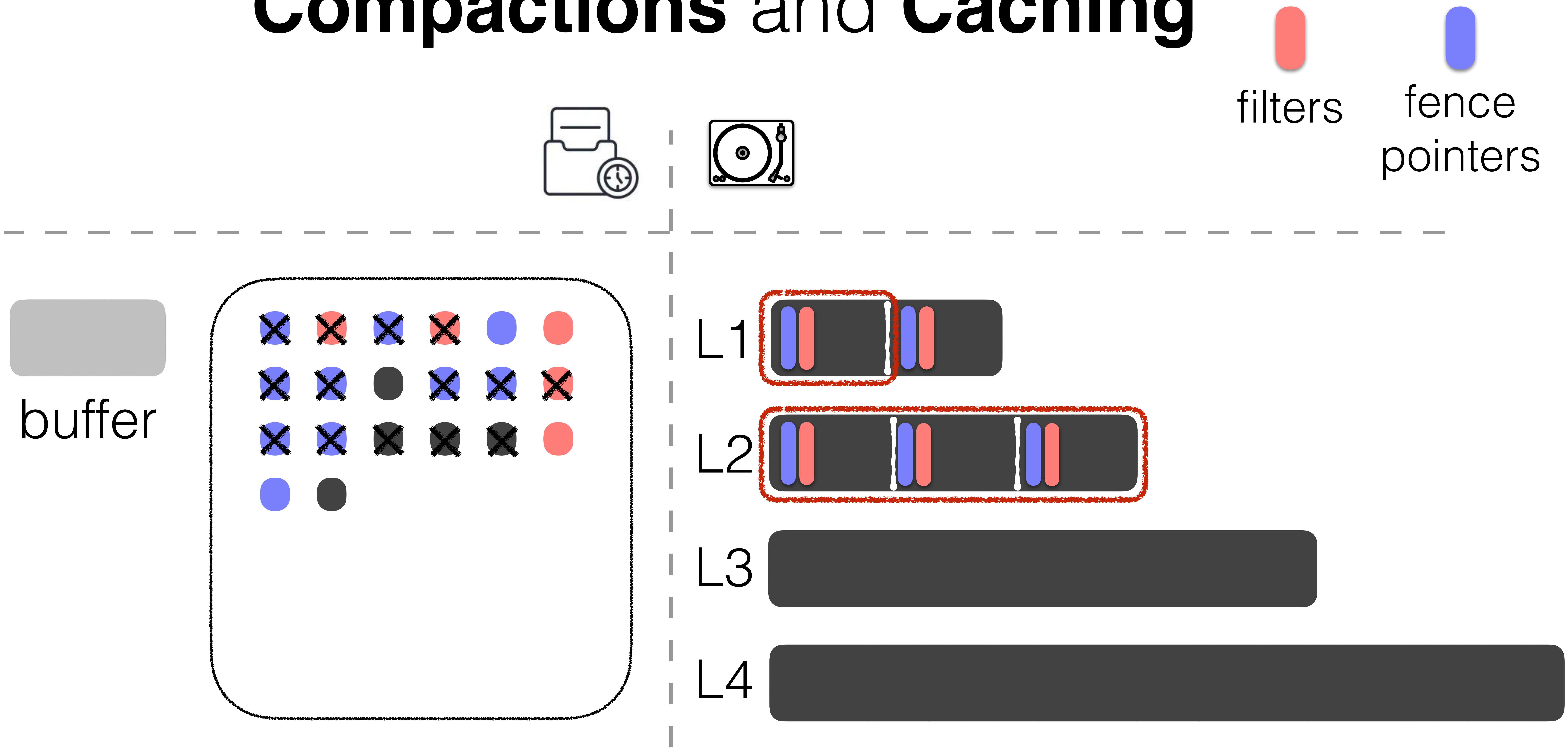


Filters under **Memory Pressure**

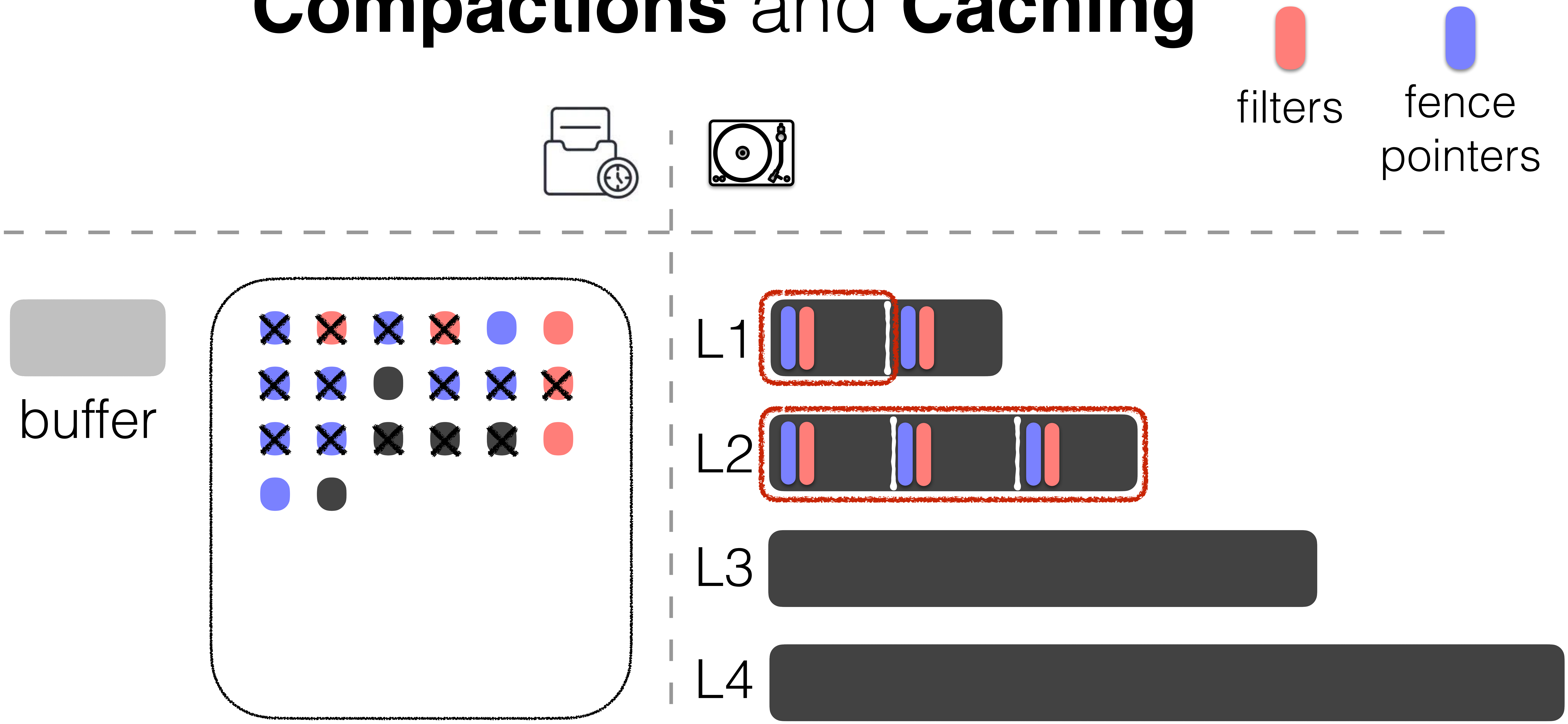


Overall, better performance with smaller memory budget

Compactions and **Caching**



Compactions and **Caching**



Leaper : A learned pre-fetcher that improves reads

YangVLDB20

Outline

Part 1: **LSM Basics**

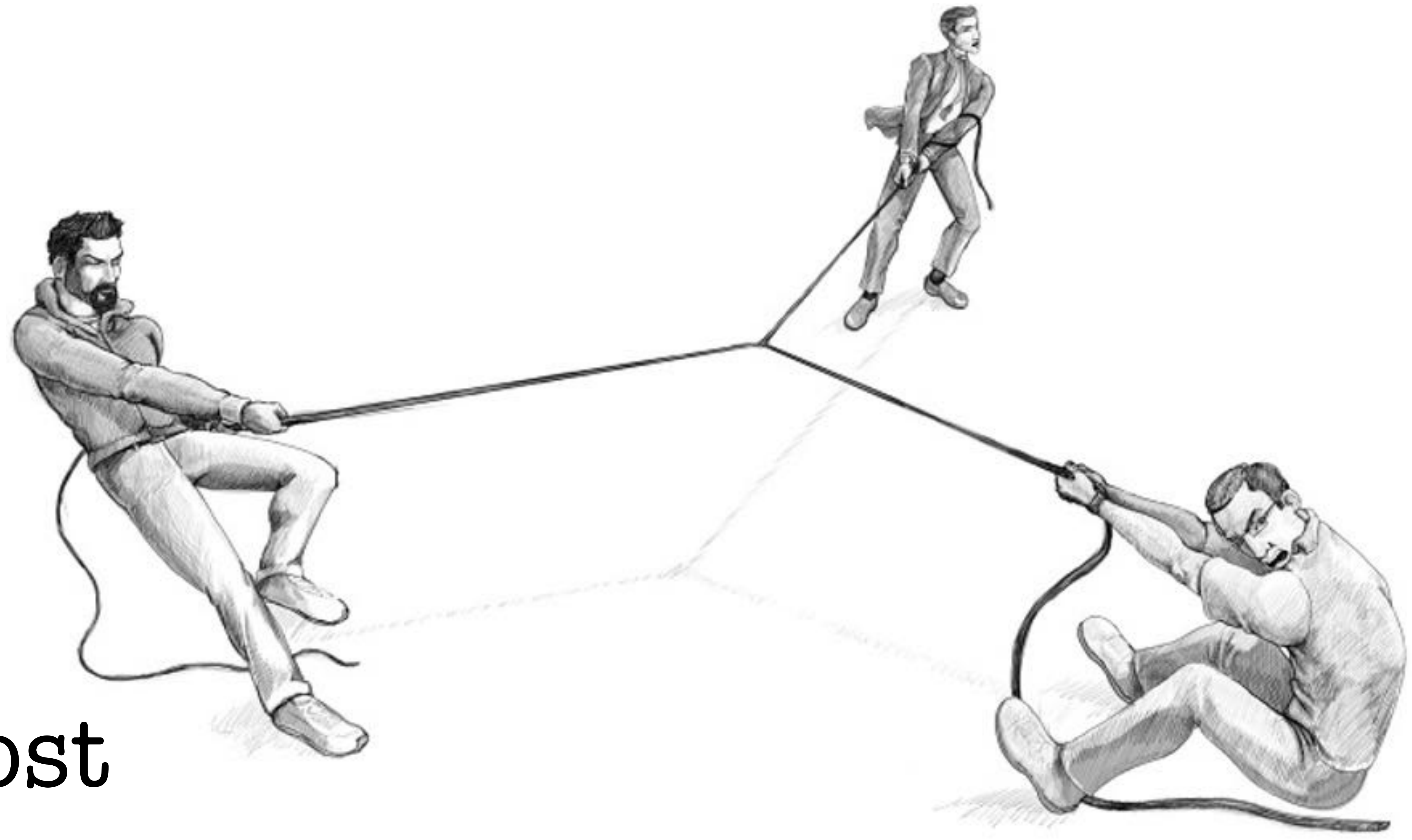
Part 2: **Read Optimizations in LSMs**

Part 3: **Navigating the LSM Design Space**

LSM **Design Space**

LSM **Design Space**

Update cost

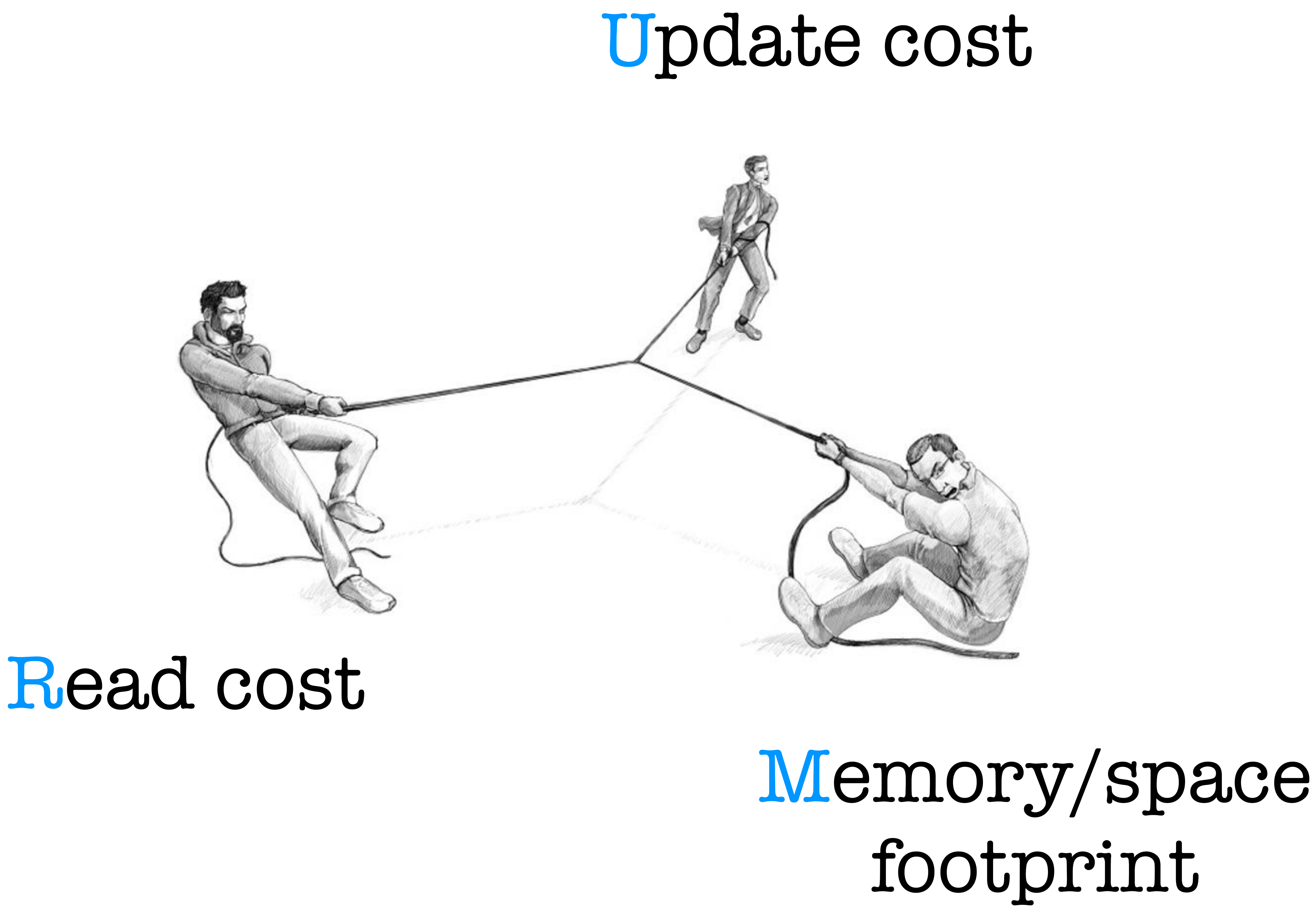


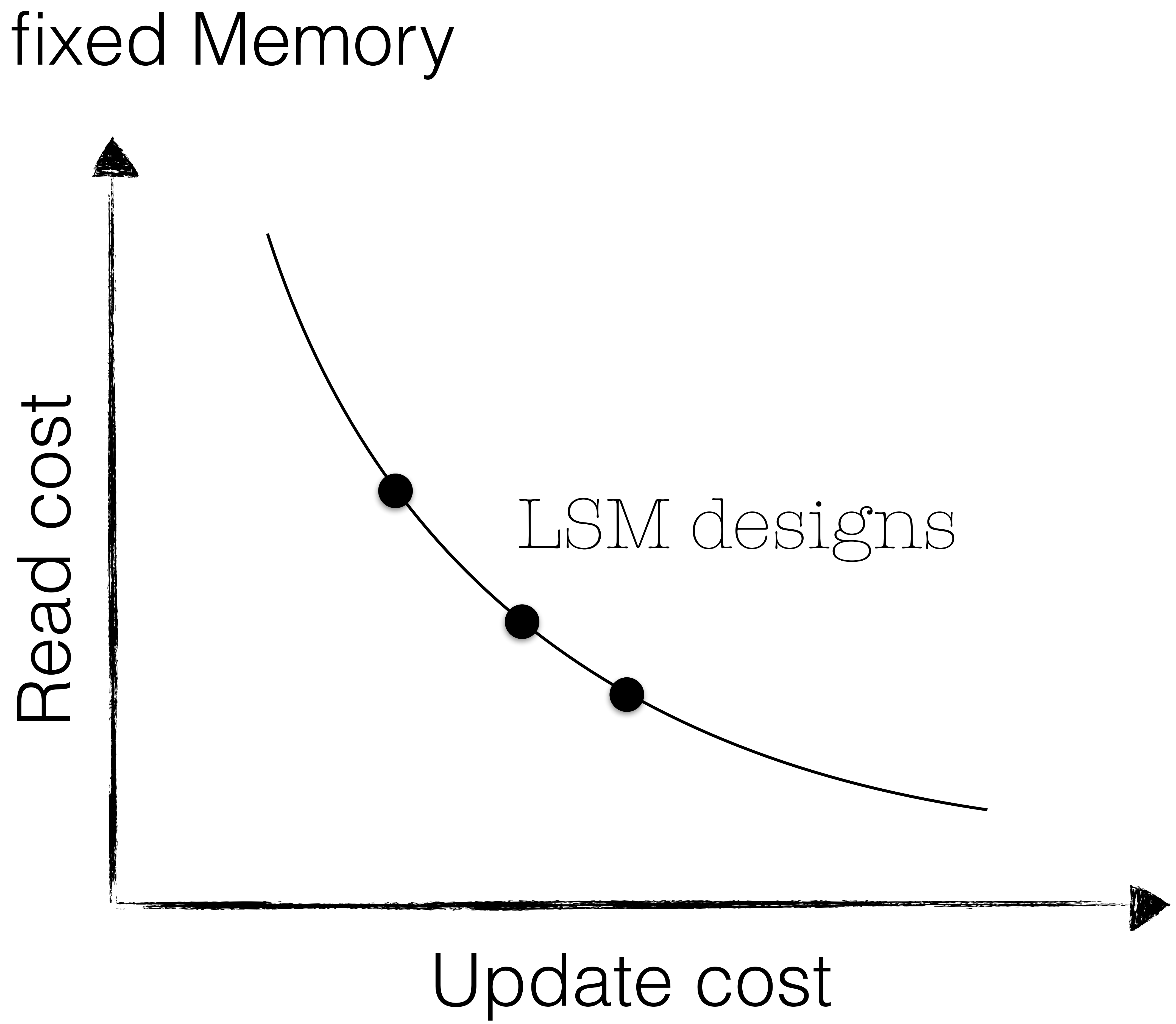
Read cost

Memory/space footprint

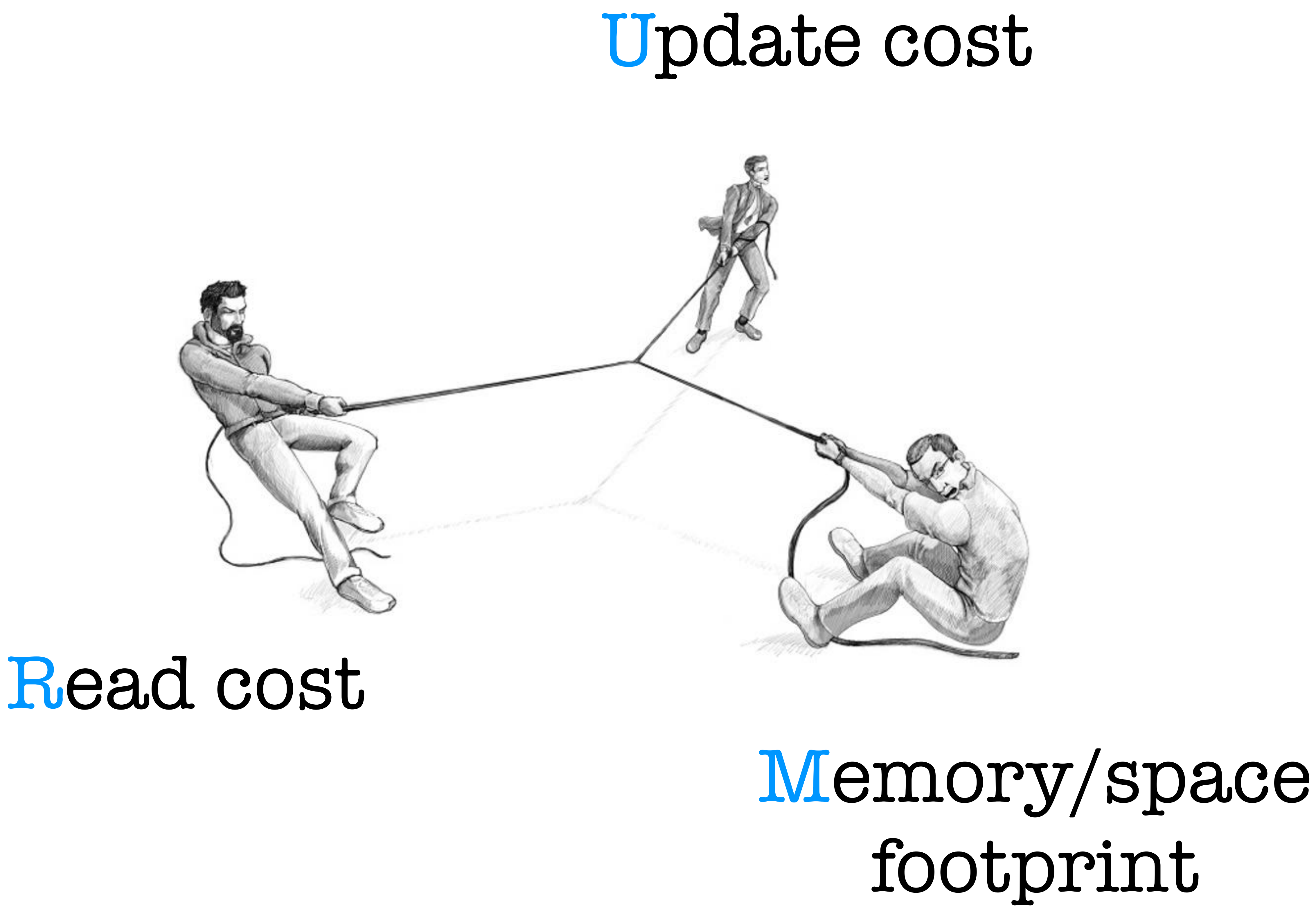
AthanassoulisEDBT16

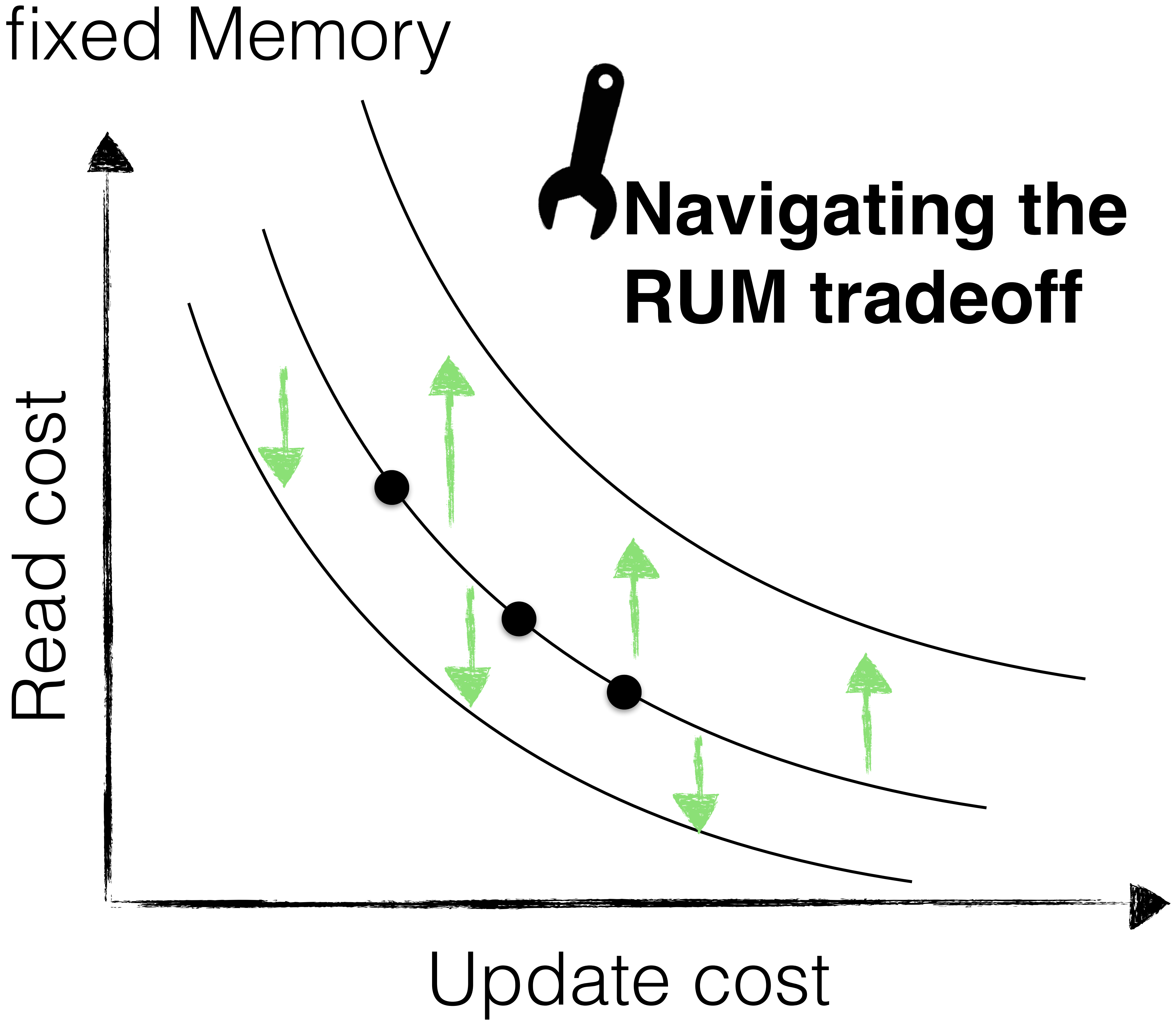
LSM **Design Space**



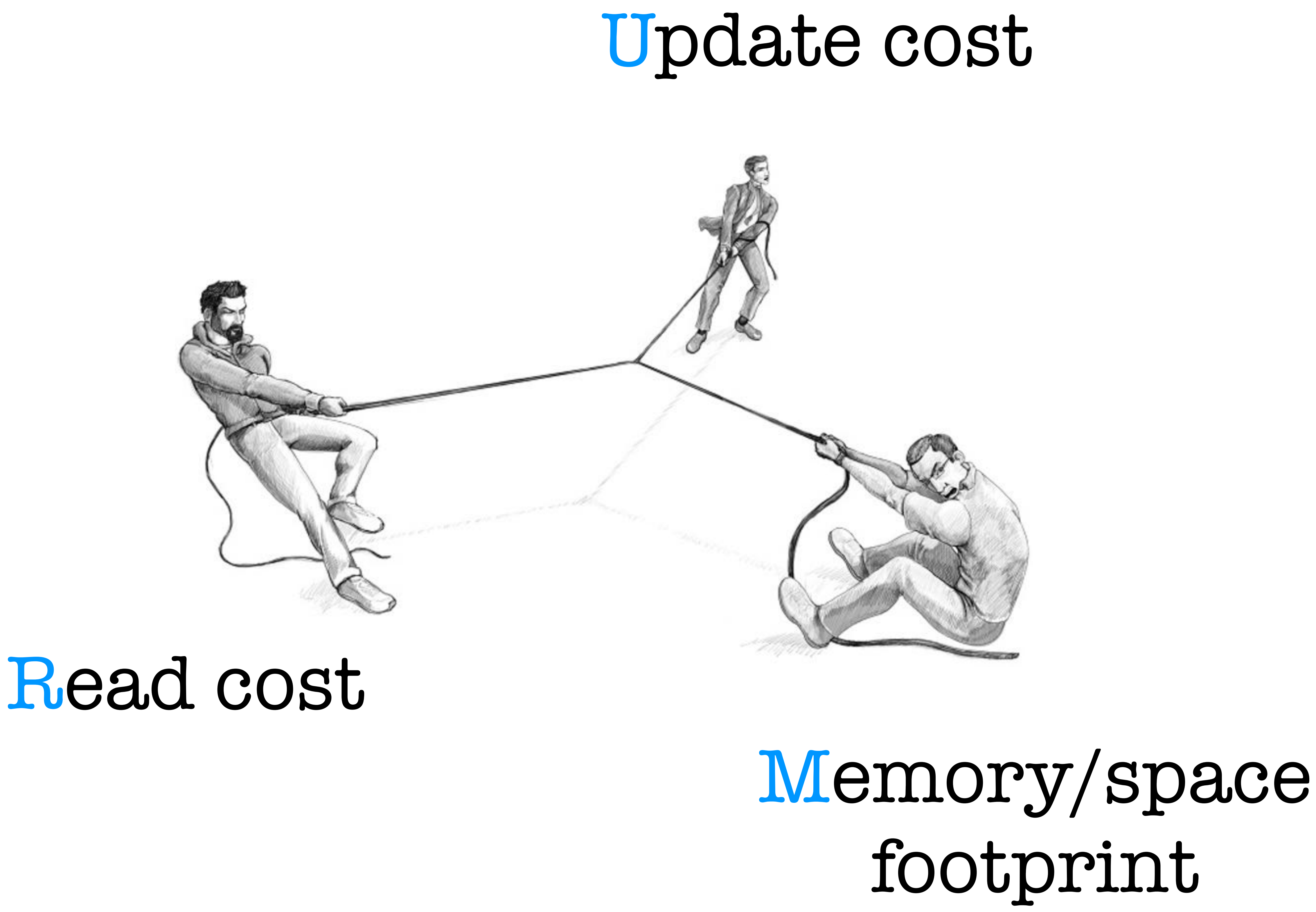


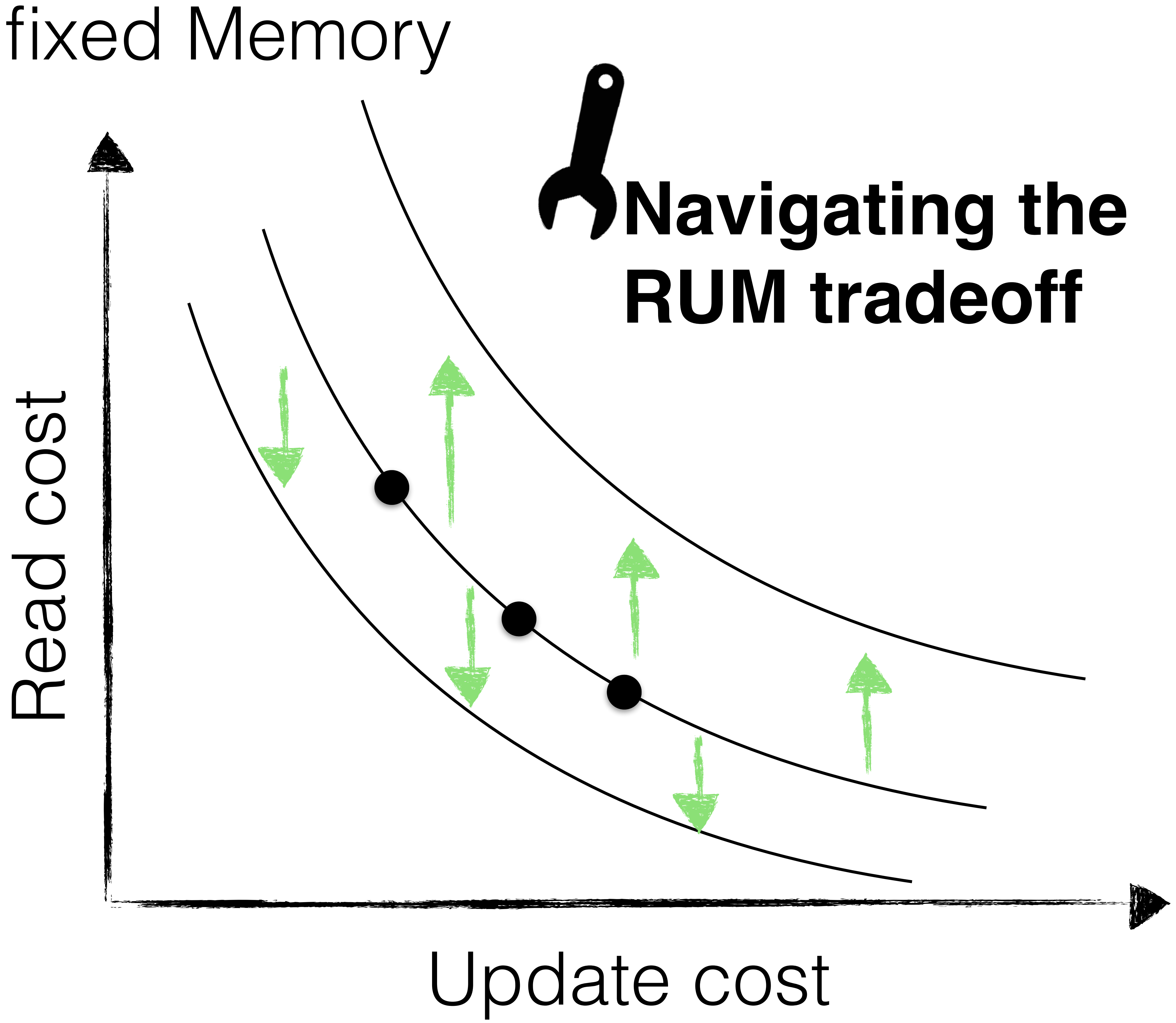
LSM **Design Space**



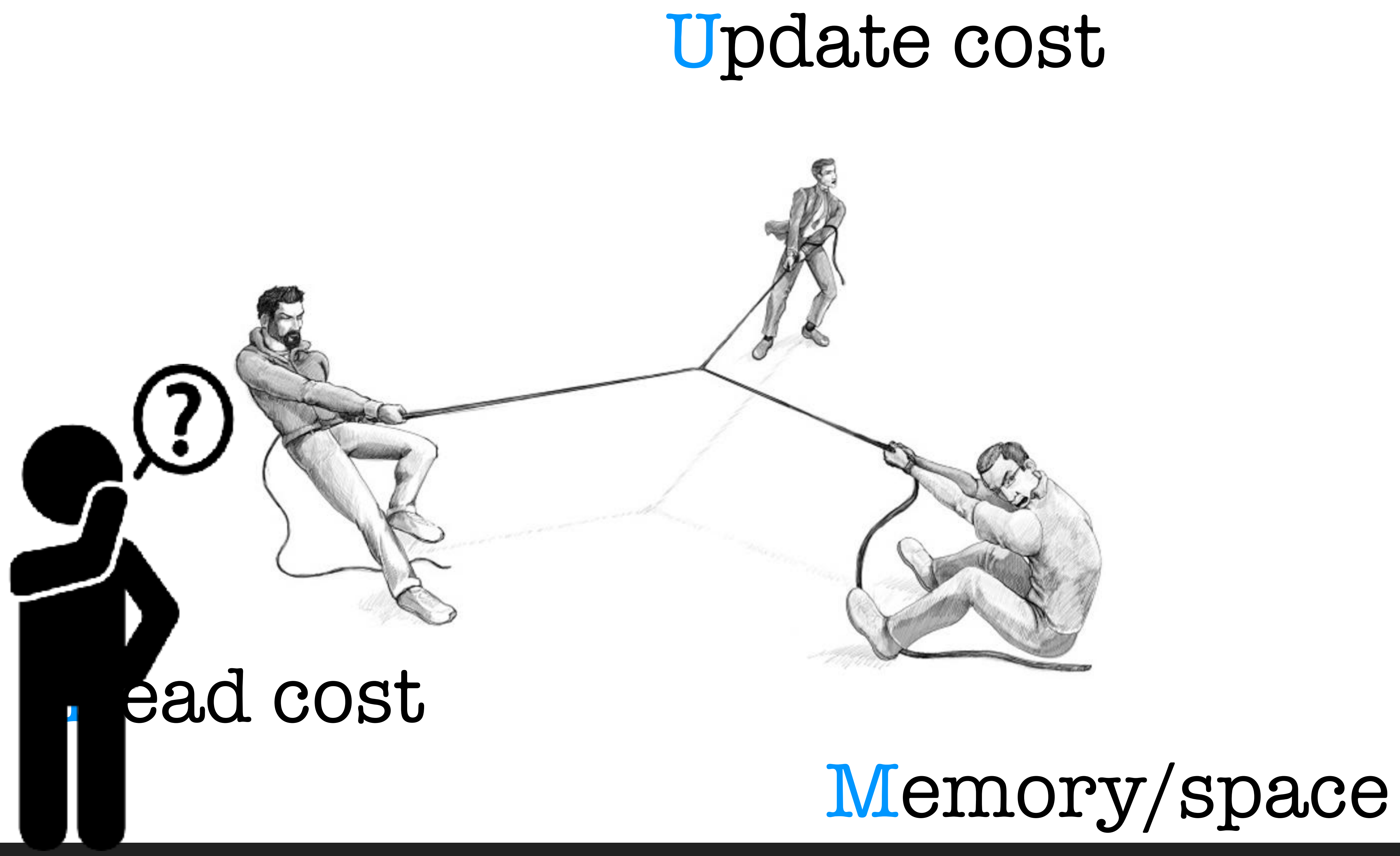


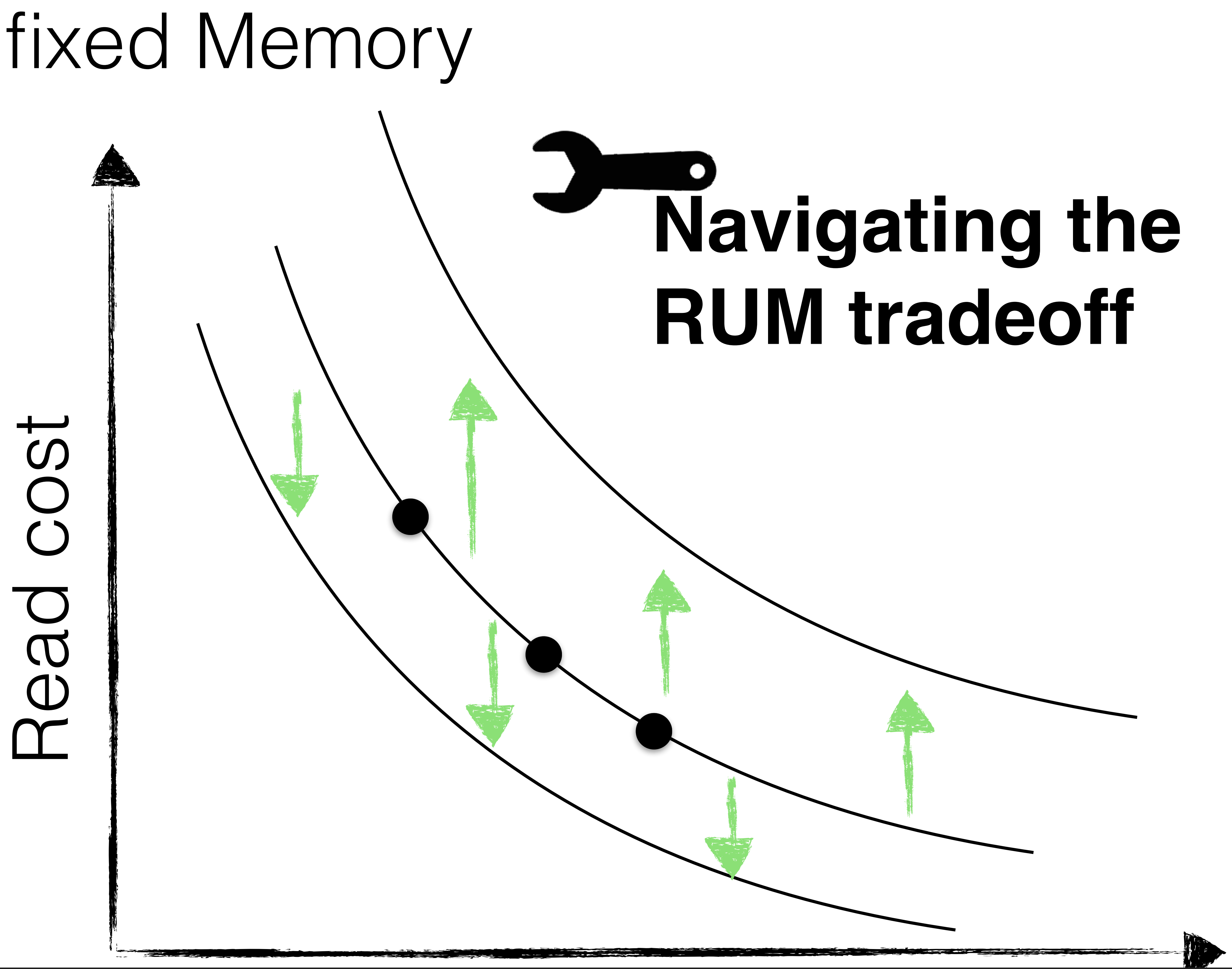
LSM **Design Space**



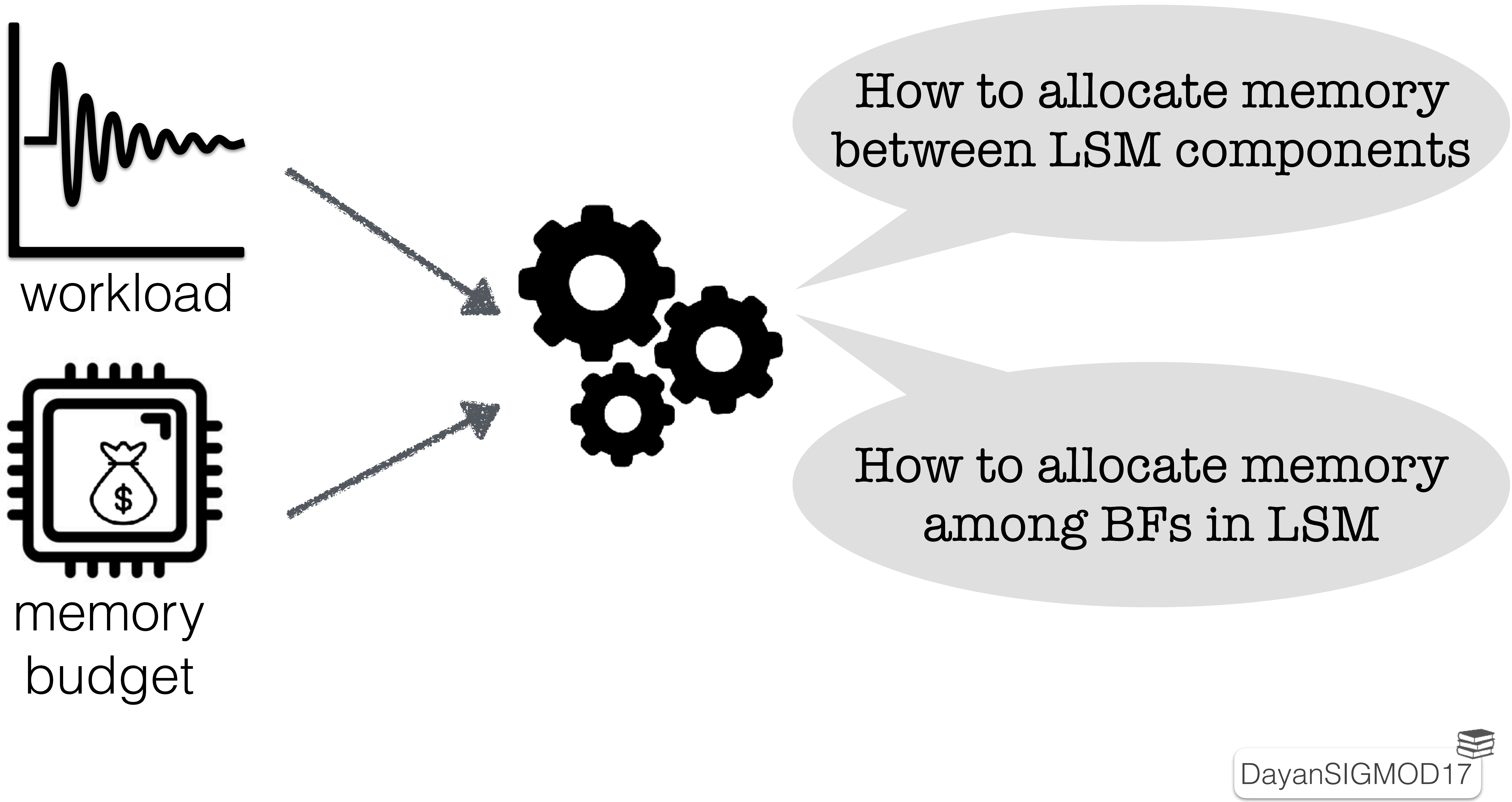


LSM **Design Space**





How to optimally allocate the available memory?



workload
memory
budget
How to allocate memory
between LSM components
How to allocate memory
among BFs in LSM
DayanSIGMOD17

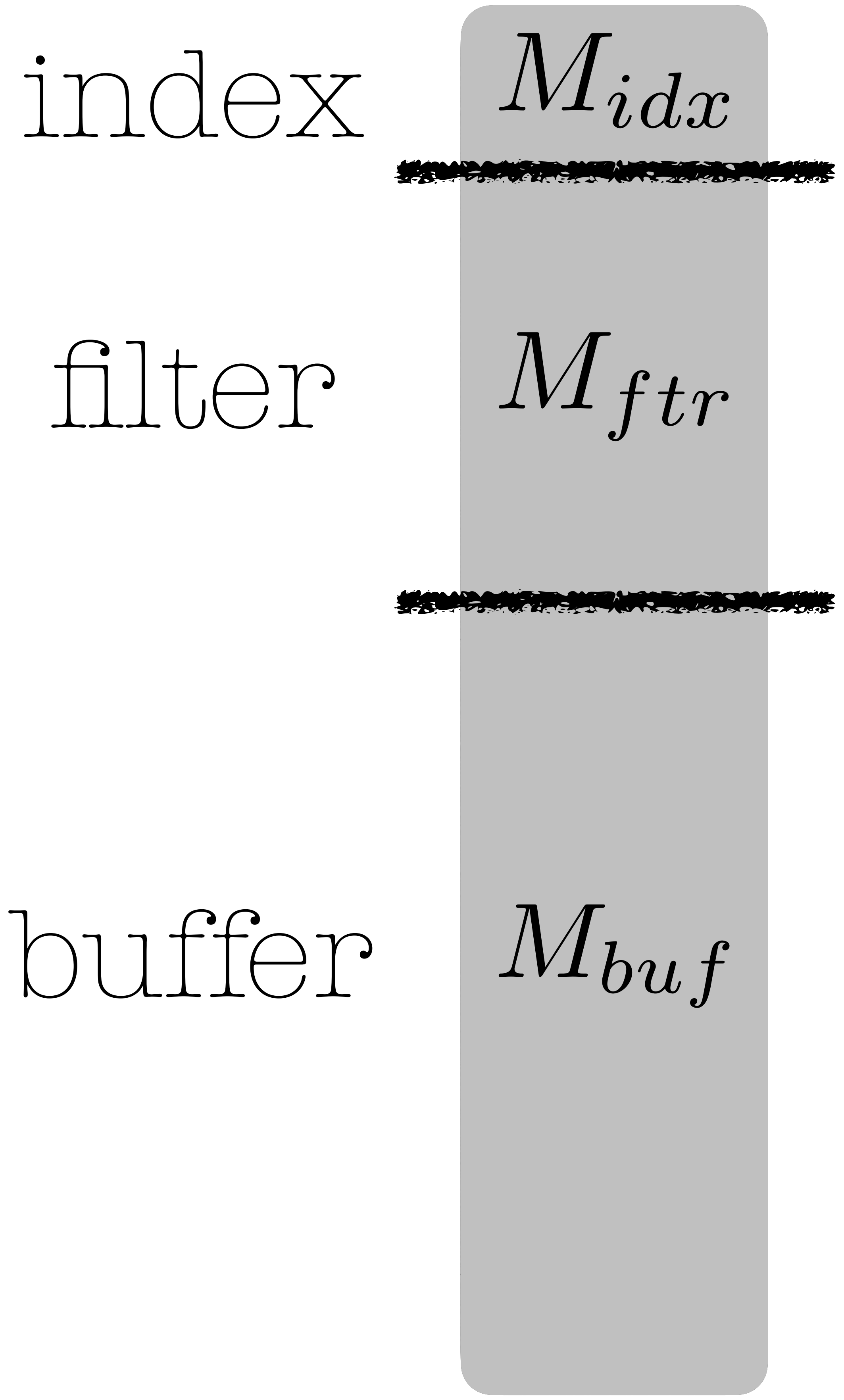
M: total memory
M_{idx}: index memory
M_{ftr}: filter memory
M_{buf}: buffer memory

The **Optimal** Memory Allocation

workload

$reads(R)$
vs.
$writes(W)$

available memory



M=M_{idx}+M_{ftr}+M_{buf}

$read\_cost(M_{idx}, M_{ftr}, M_{buf})$

$write\_cost(M_{buf})$

$cost = R \cdot read\_cost + W \cdot write\_cost$

DayanSIGMOD17

M: total memory
M_{idx}: index memory
M_{ftr}: filter memory
M_{buf}: buffer memory
M_{cache}: block cache memory

The **Optimal** Memory Allocation

available memory

$M = M_{cache} + M_{buf}$

workload

$reads(R)$

vs.

$writes(W)$

block cache M_{cache}

buffer M_{buf}

$read\_cost(M_{cache})$

$write\_cost(M_{buf})$

$$cost = R \cdot read\_cost + W \cdot write\_cost$$

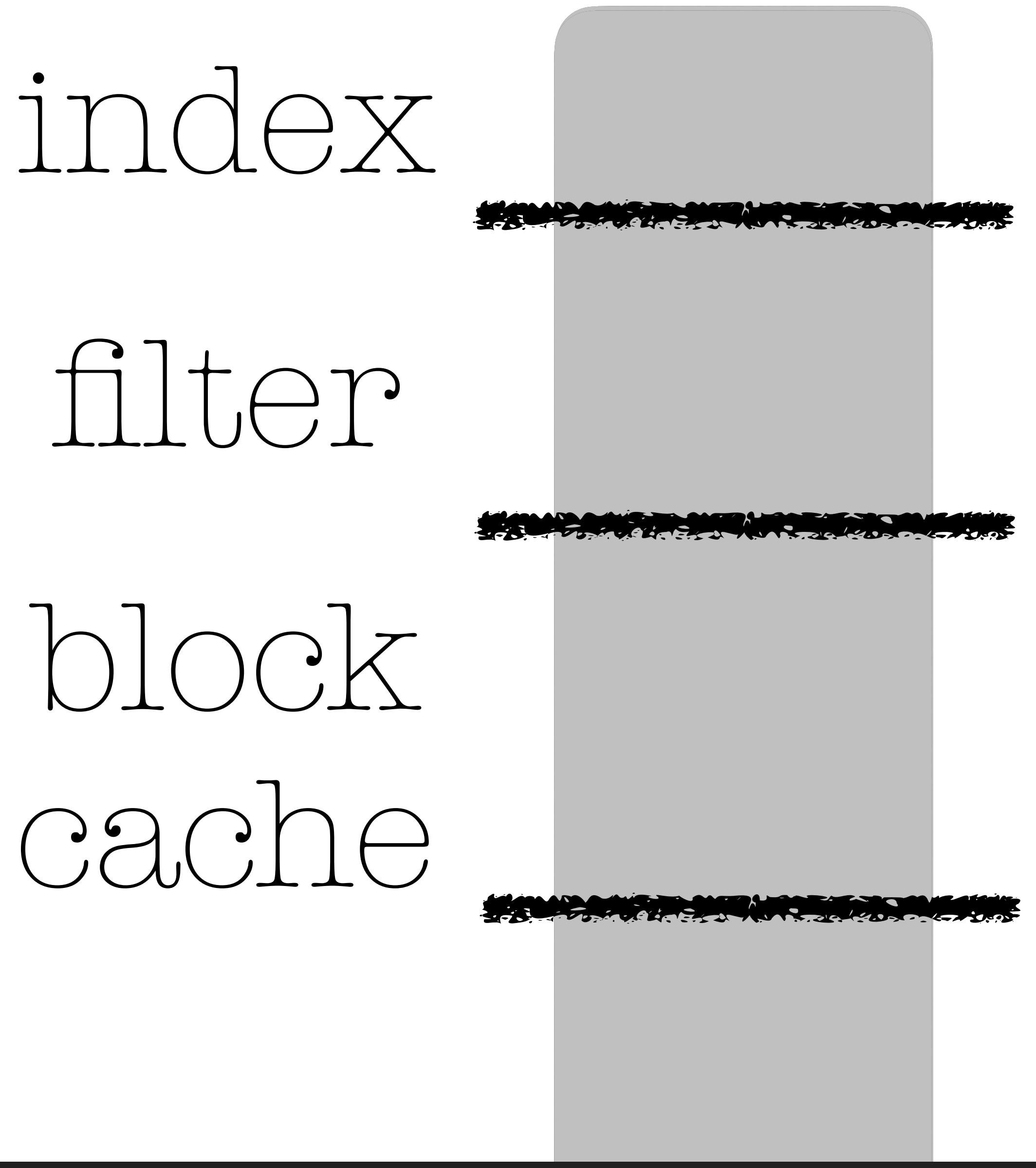
LuoSIGMOD20

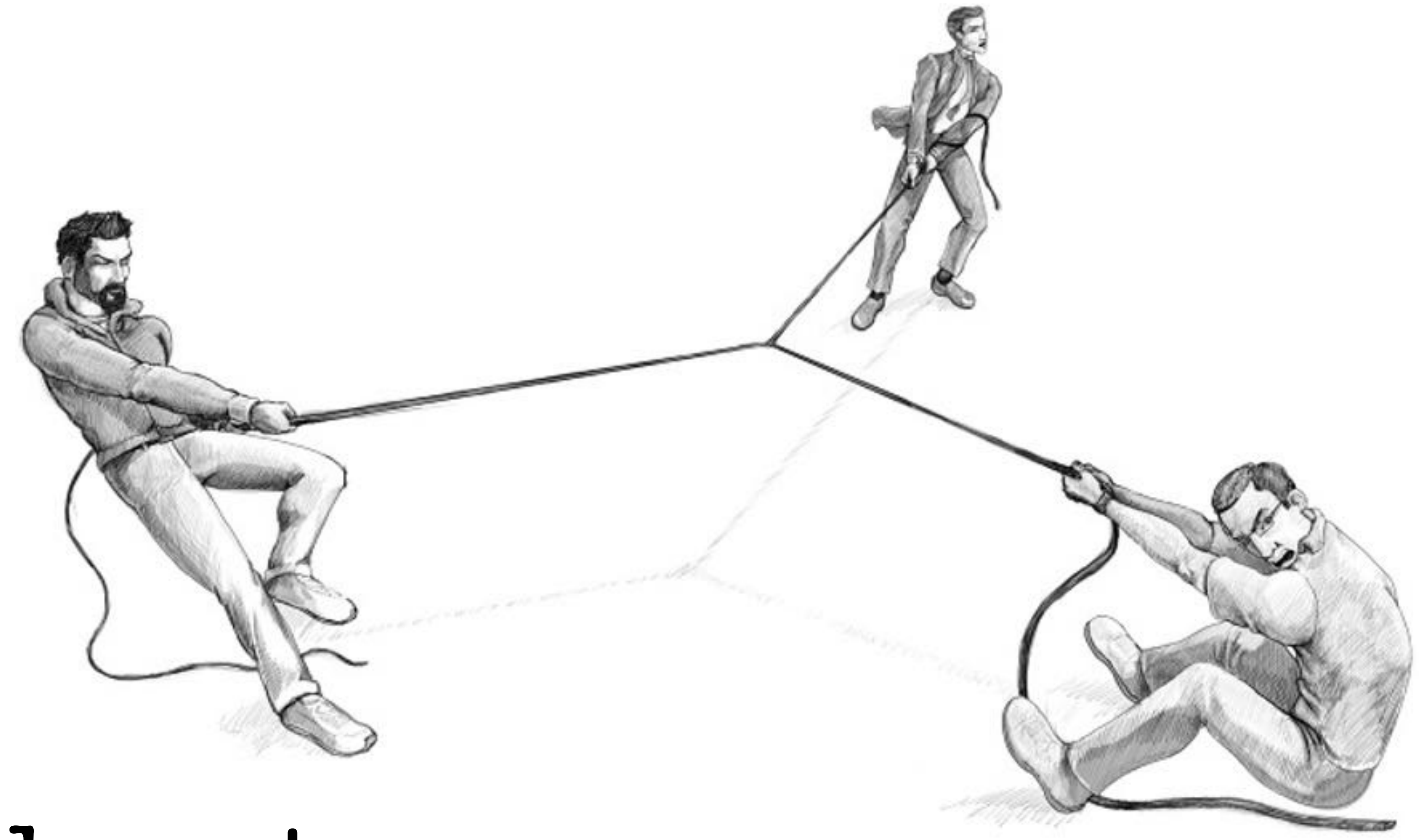
LuoVLDB20

[Open Problem]

The **Optimal** Memory Allocation

available memory

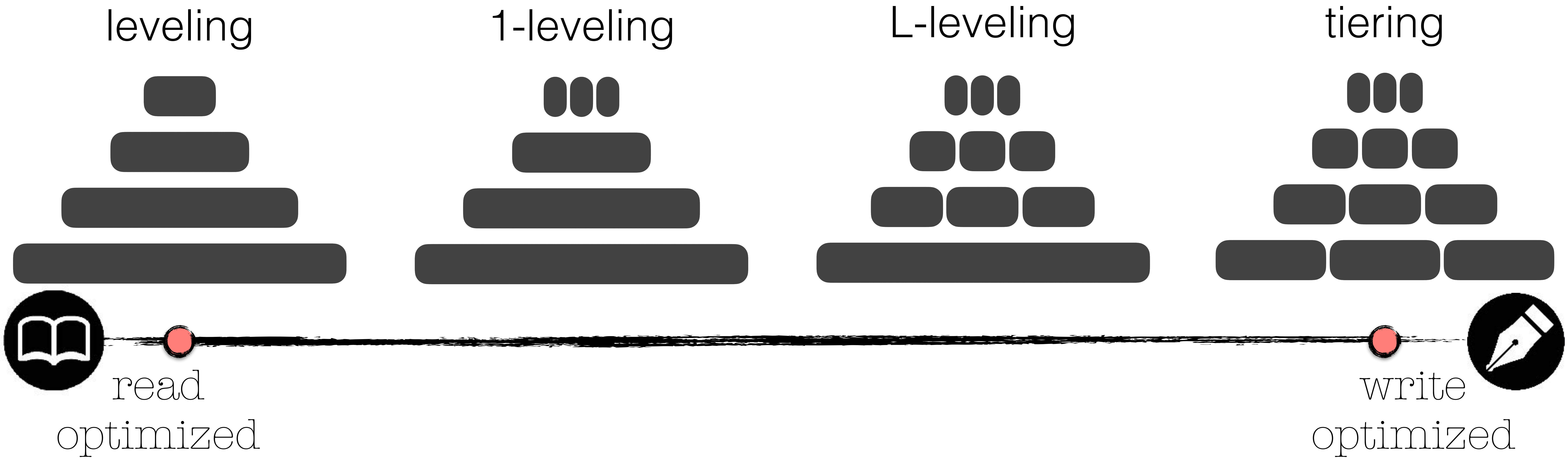




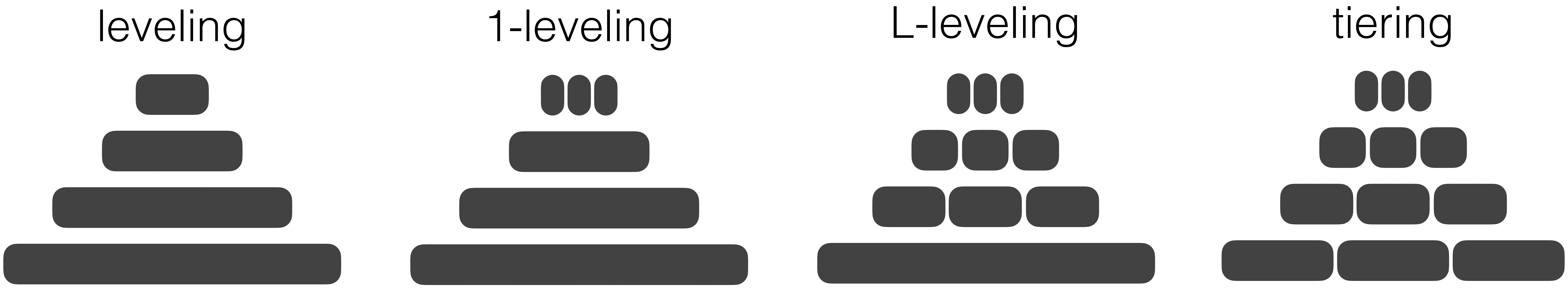
Navigating read vs. writes: **data layouts**

Data **Layout**

Data **Layout**


leveling
1-leveling
L-leveling
tiering
read
optimized
write
optimized

Storage Layer **Design Continuum**

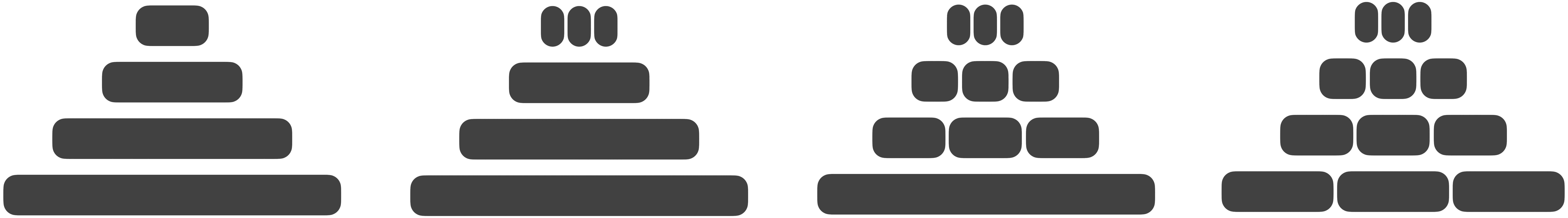


Any design can be defined by the tuple-set: (T, i)

SarkarVLDB21

Storage Layer **Design Continuum**

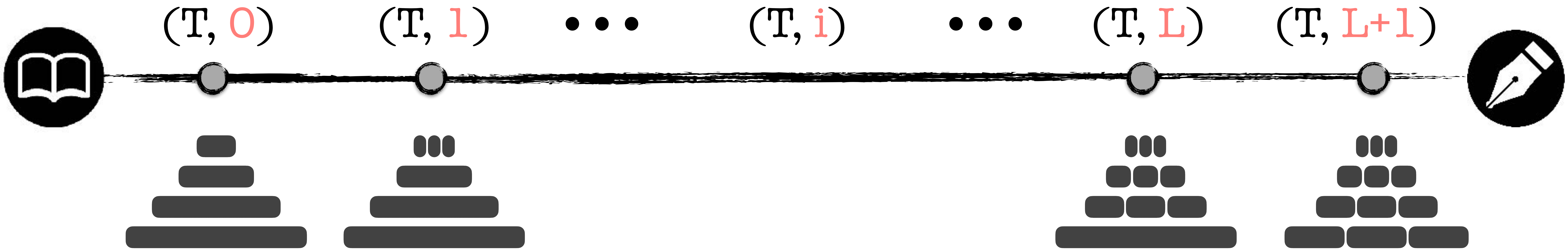
leveling	1-leveling	L-leveling	tiering
(T, 0)	(T, 1)	(T, L)	(T, L+1)



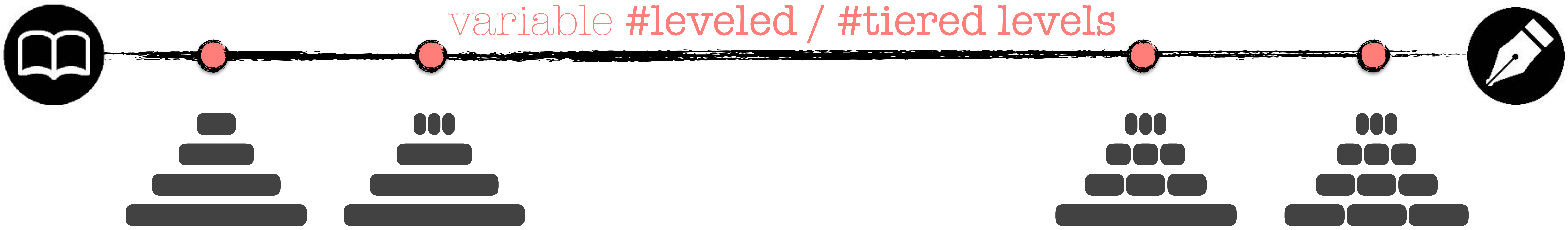
Any design can be defined by the tuple-set: (T, i)

SarkarVLDB21

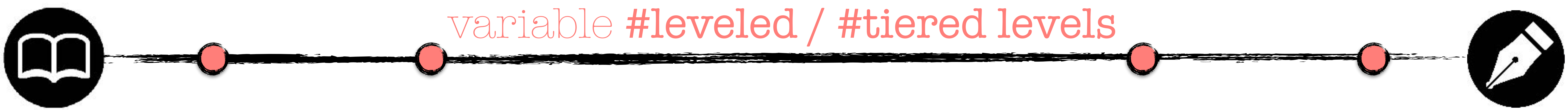
Storage Layer **Design Continuum**

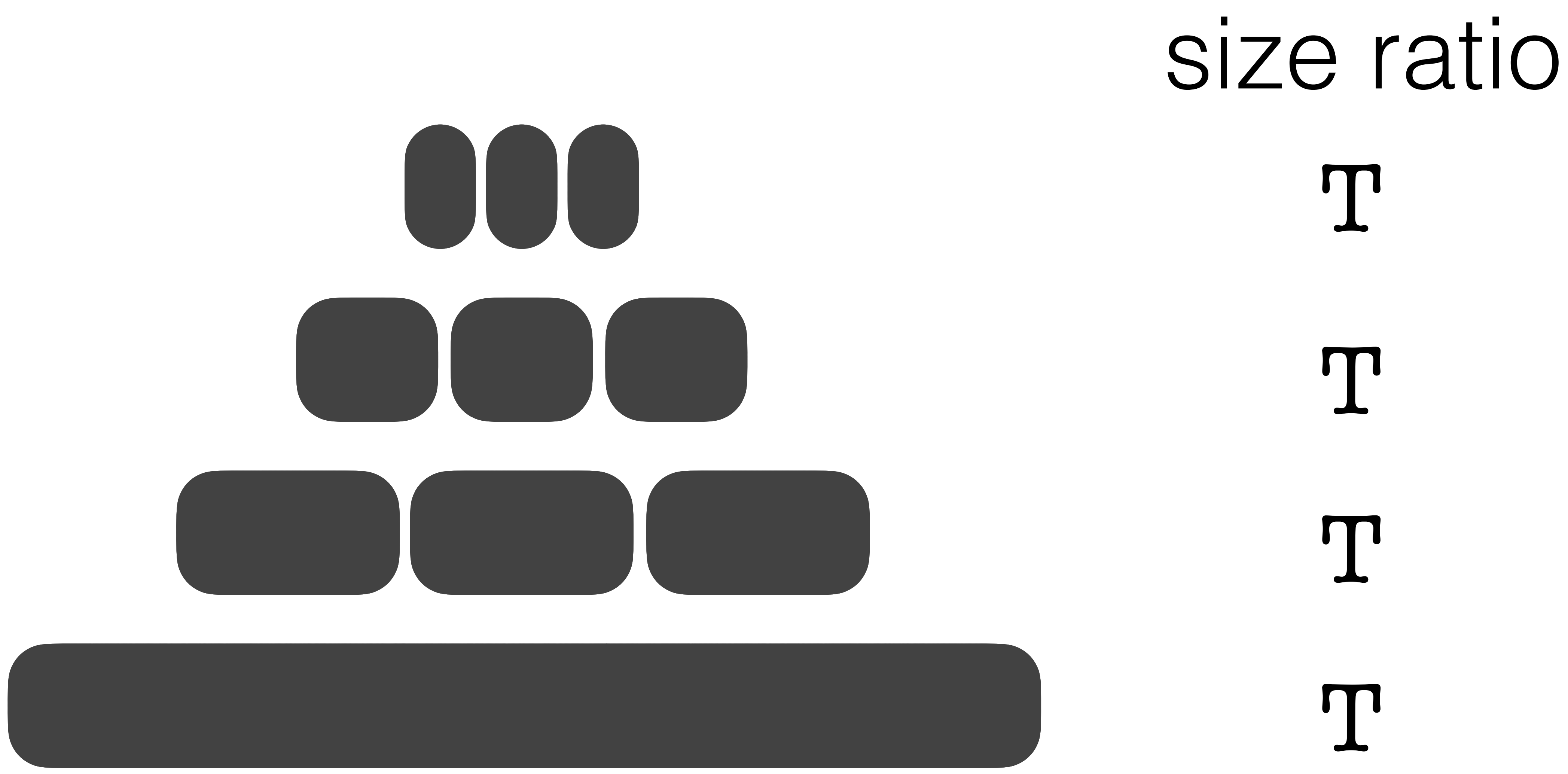


Storage Layer **Design Continuum**

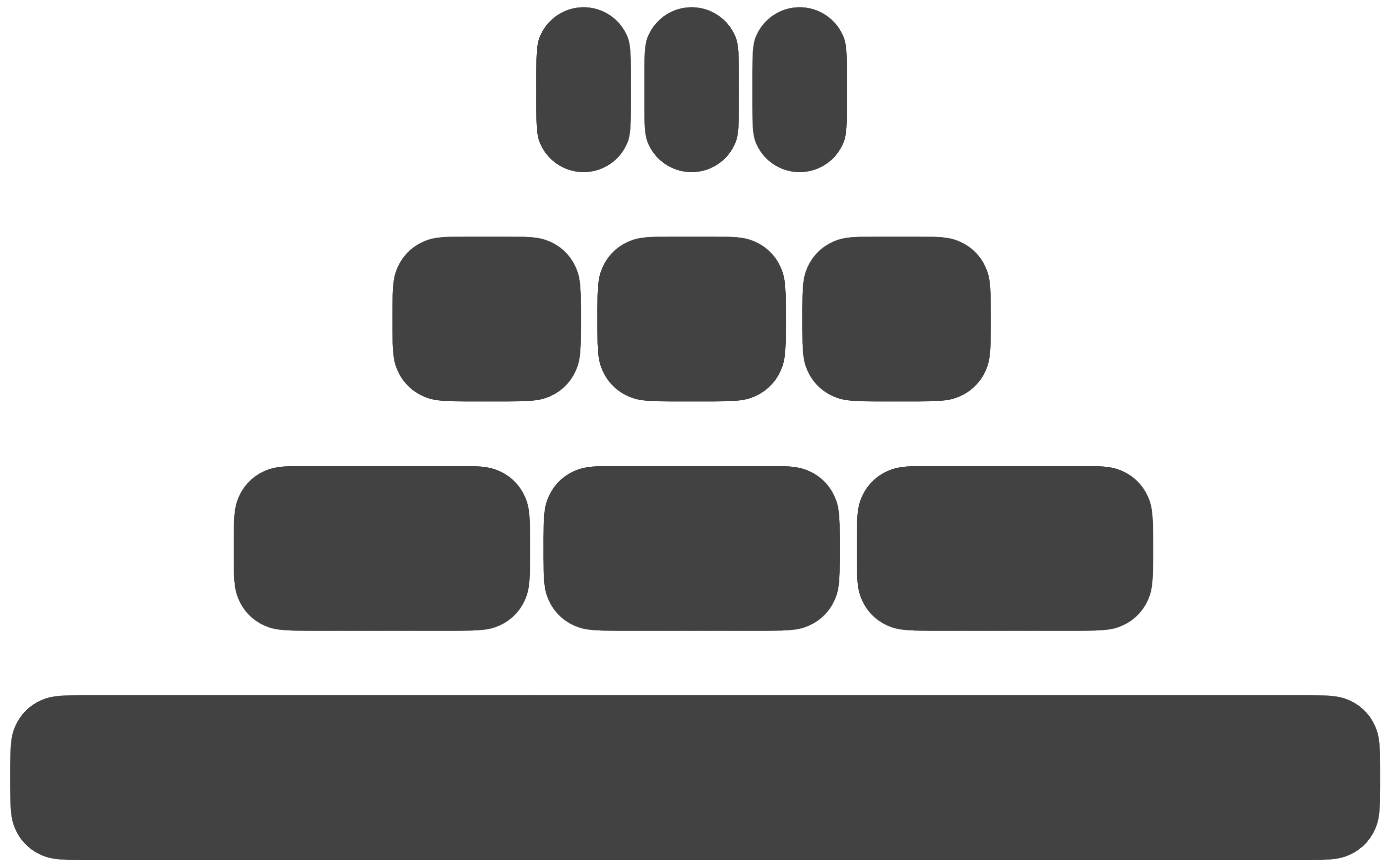


Storage Layer **Design Continuum**





Storage Layer **Design Continuum**



size ratio

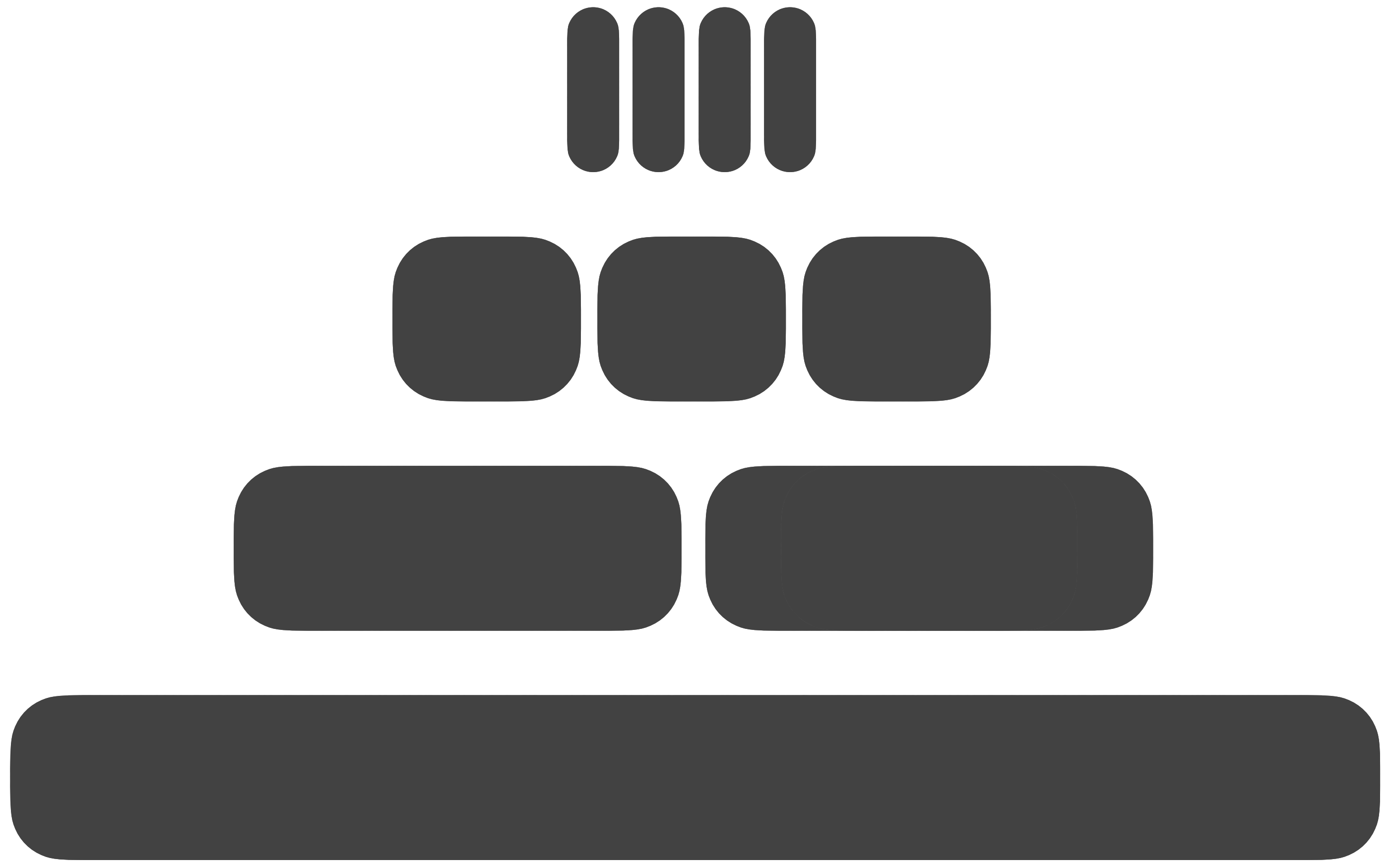
T

T

T

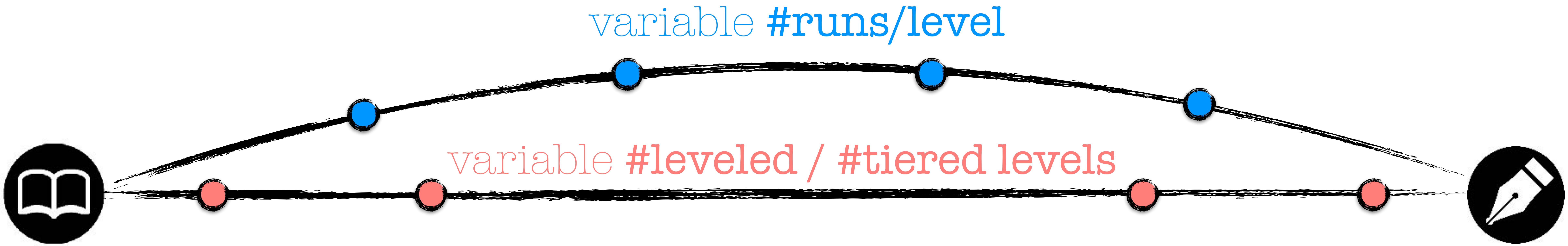
T

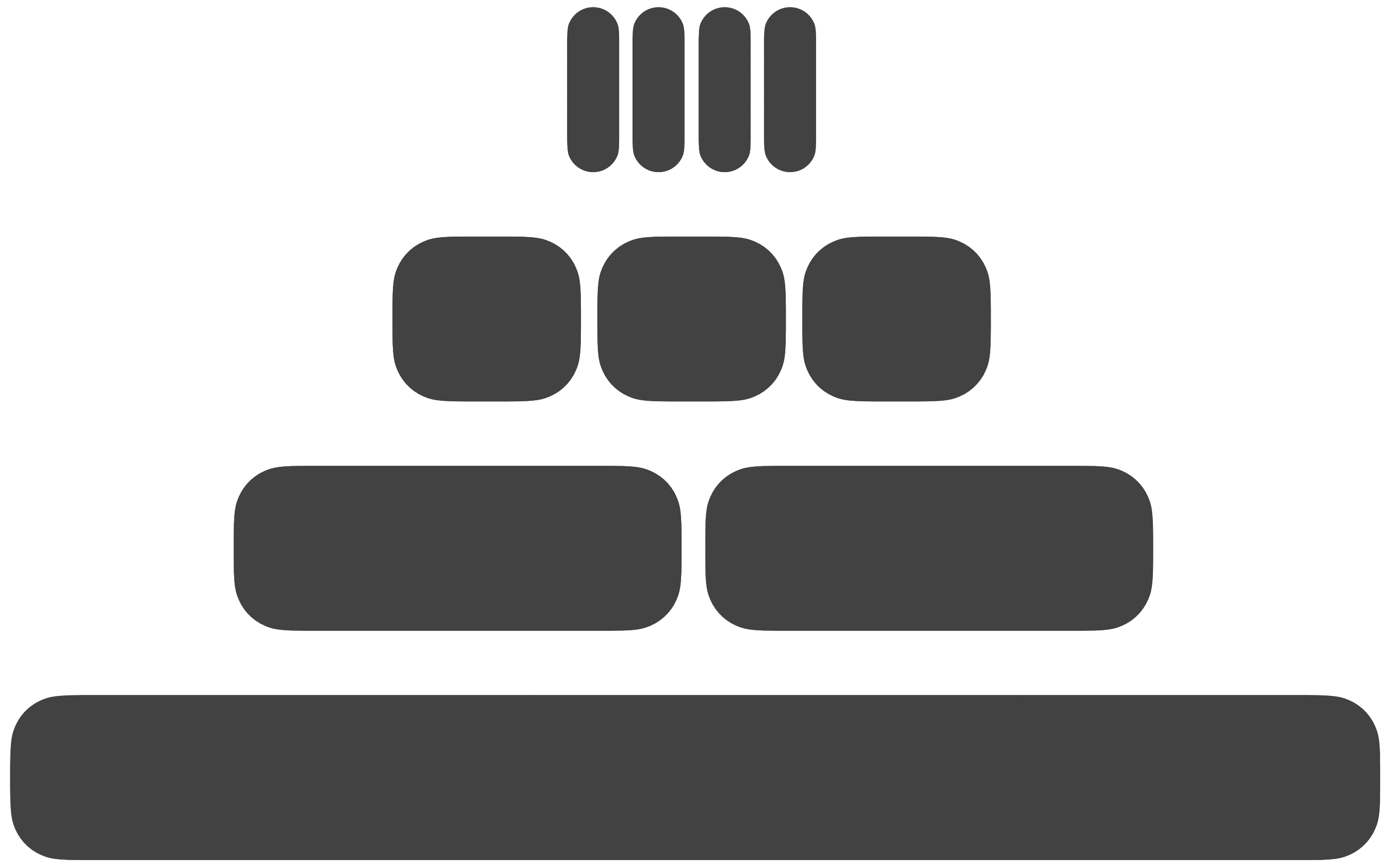
Storage Layer **Design Continuum**



size ratio	#runs
T	4
T	3
T	2
T	1

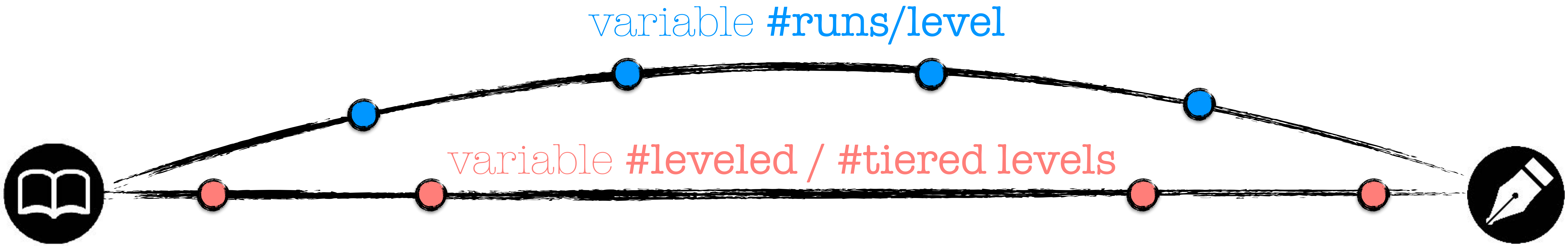
Storage Layer **Design Continuum**

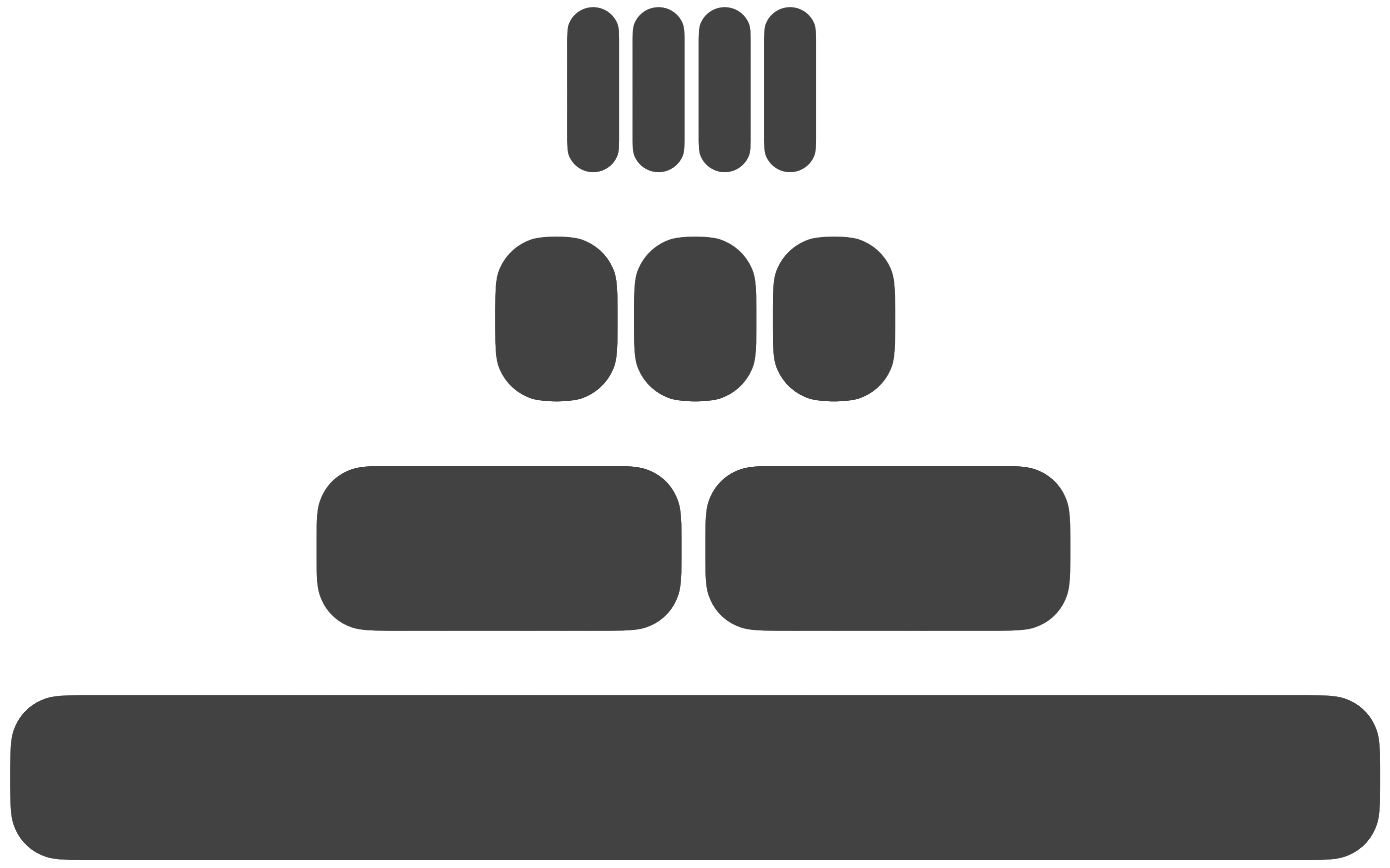




size ratio	#runs
T	4
T	3
T	2
T	1

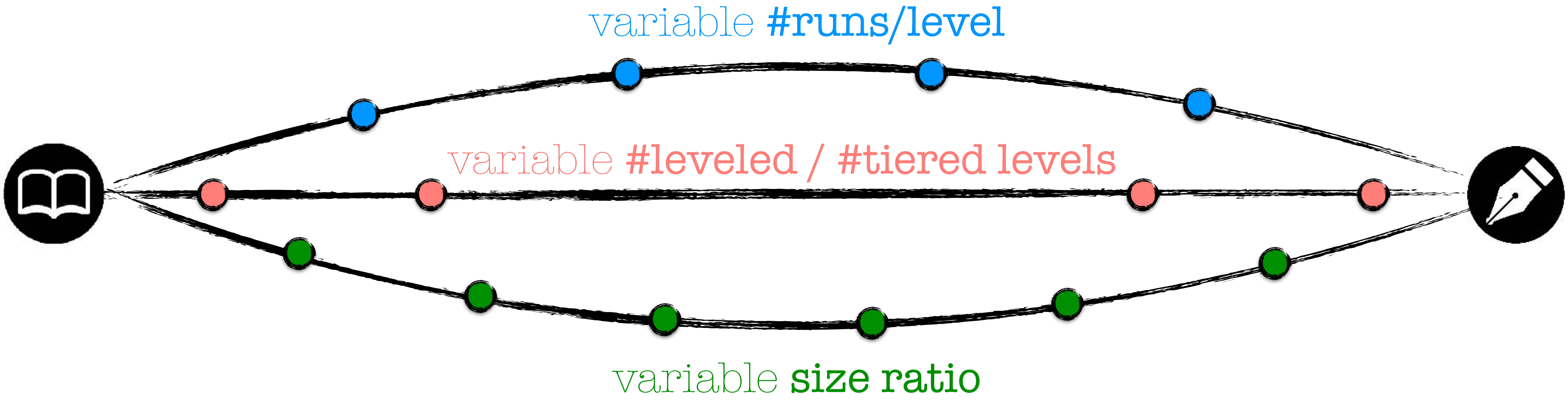
Storage Layer **Design Continuum**



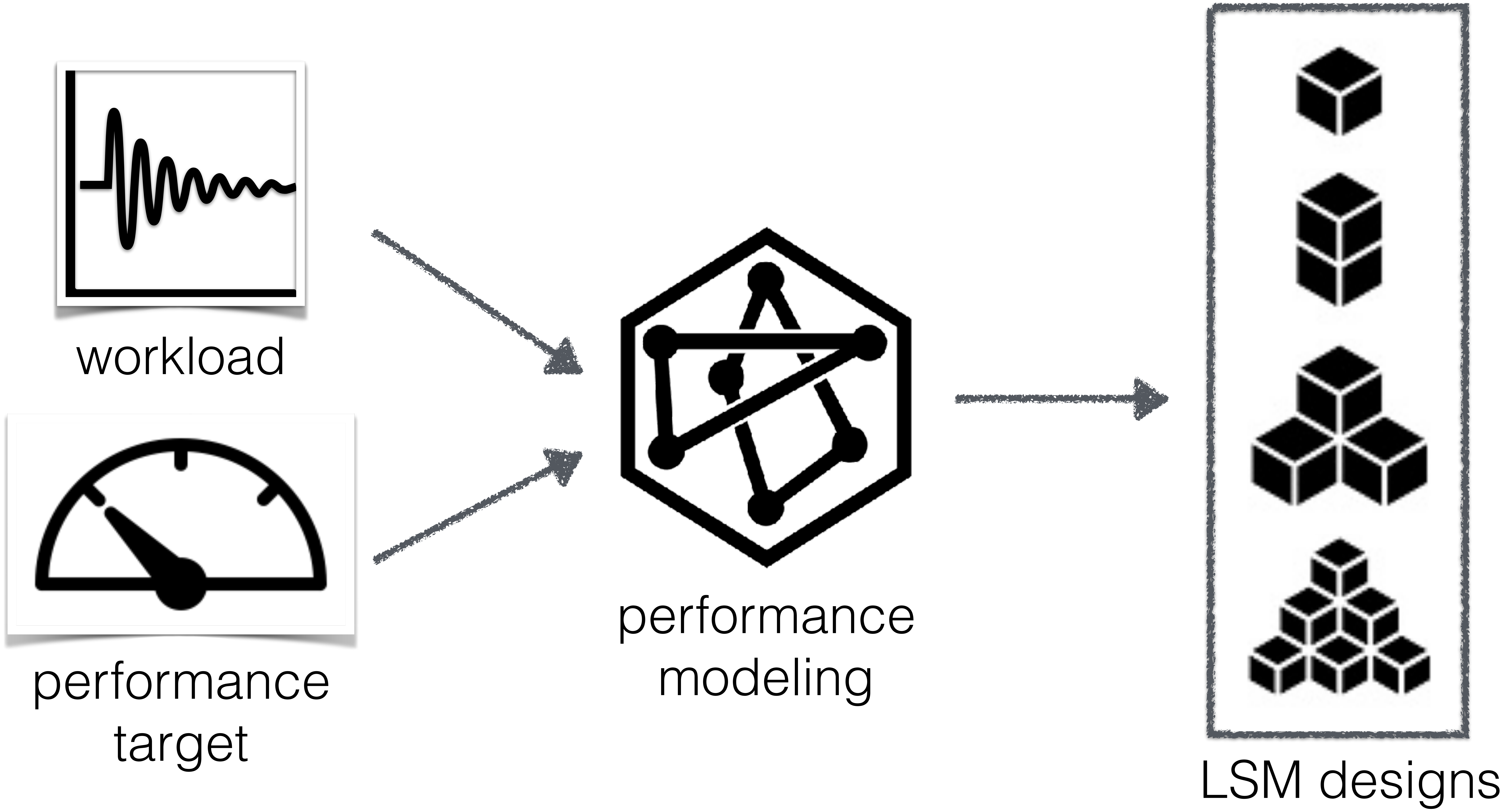


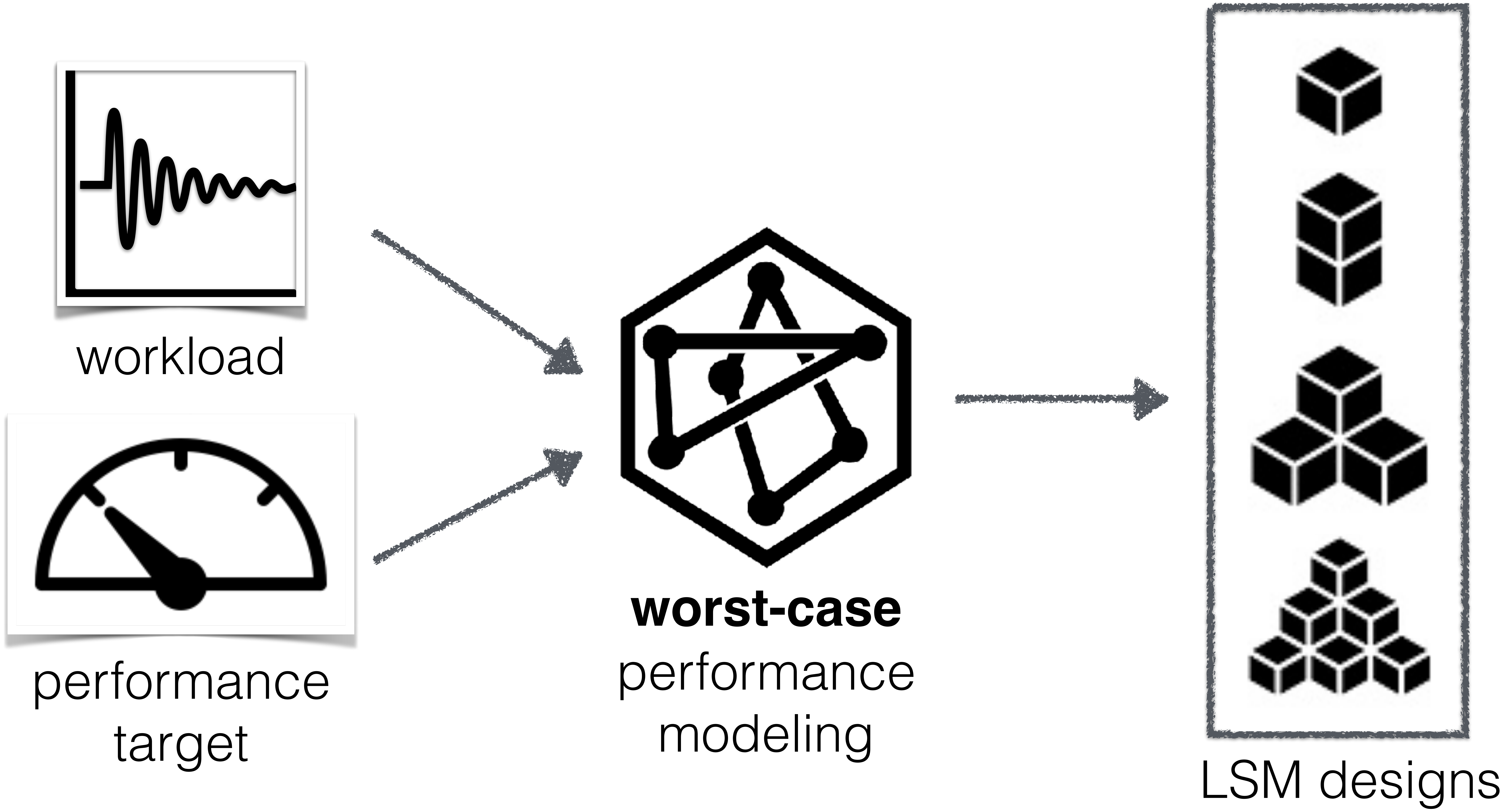
size ratio	#runs
2	4
2.5	3
3	2
4	1

Storage Layer **Design Continuum**



The LSM storage layer design continuum


workload
performance target
performance modeling
LSM designs

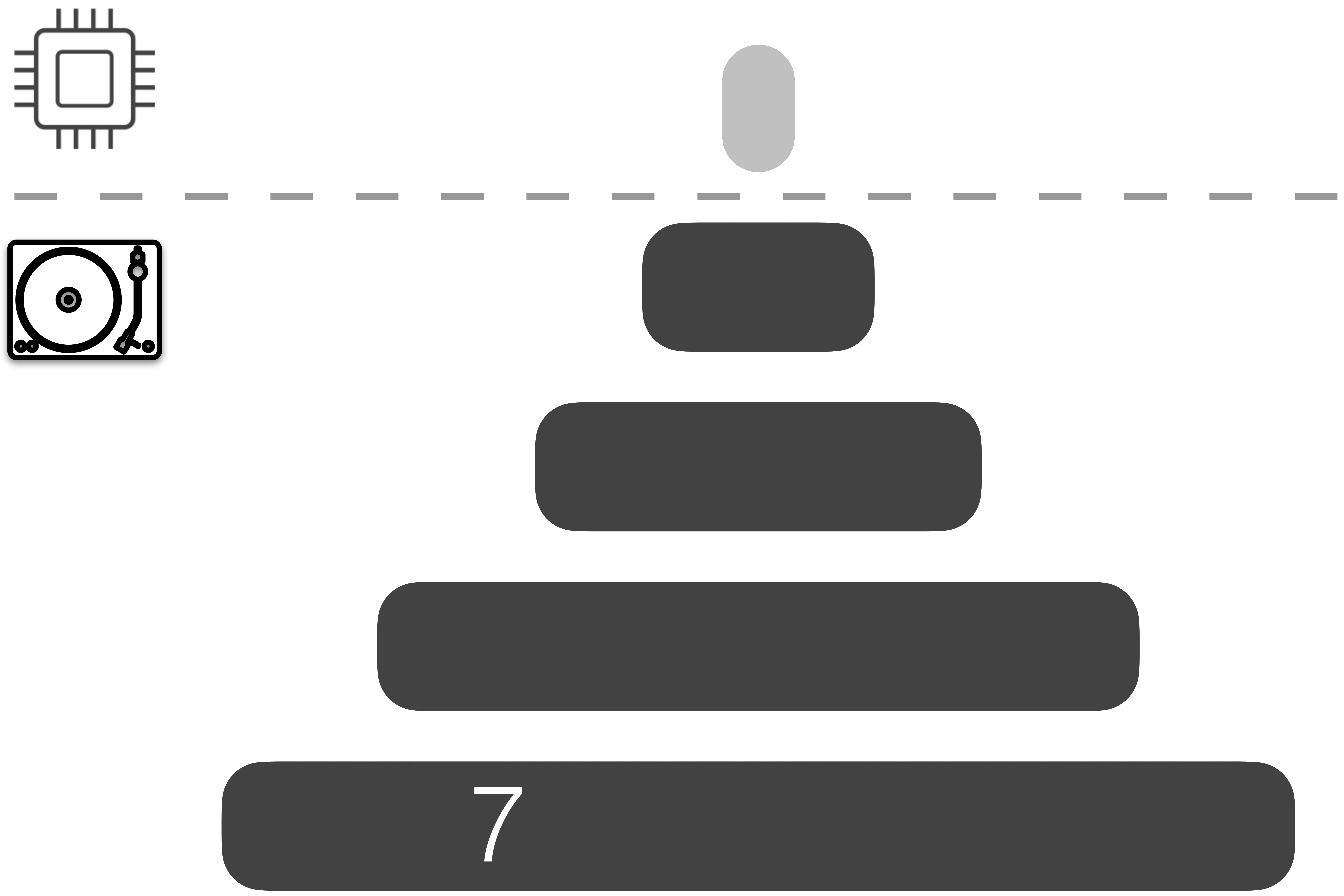

workload
performance
target
worst-case
performance
modeling
LSM designs

worst-case
performance
modeling

L: #levels

ϕ : FPR of BF

worst-case
performance
modeling

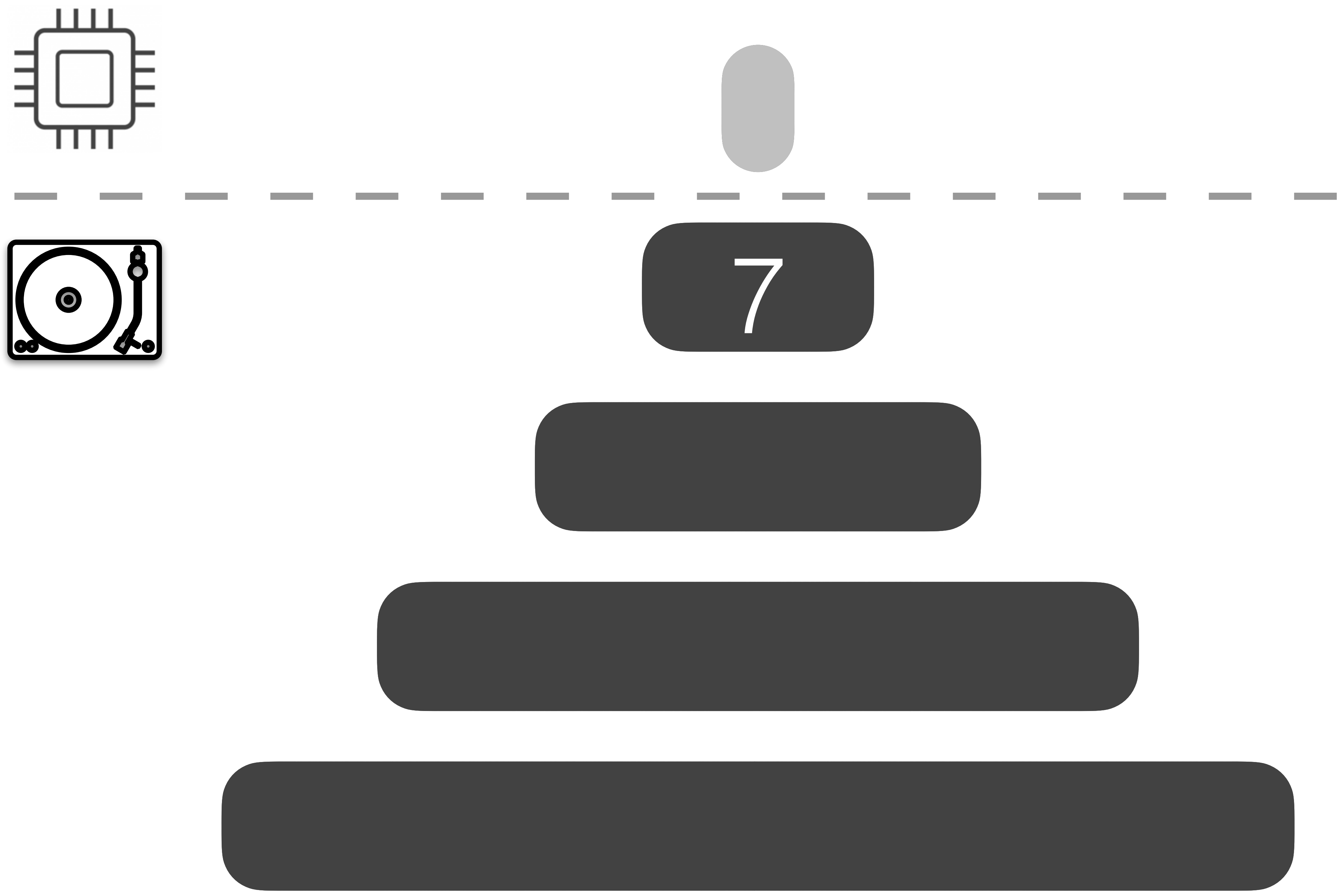


worst-case read cost: $1 + \sum_{i=1}^{L-1} \phi_i$

L : #levels

ϕ : FPR of BF

worst-case performance modeling



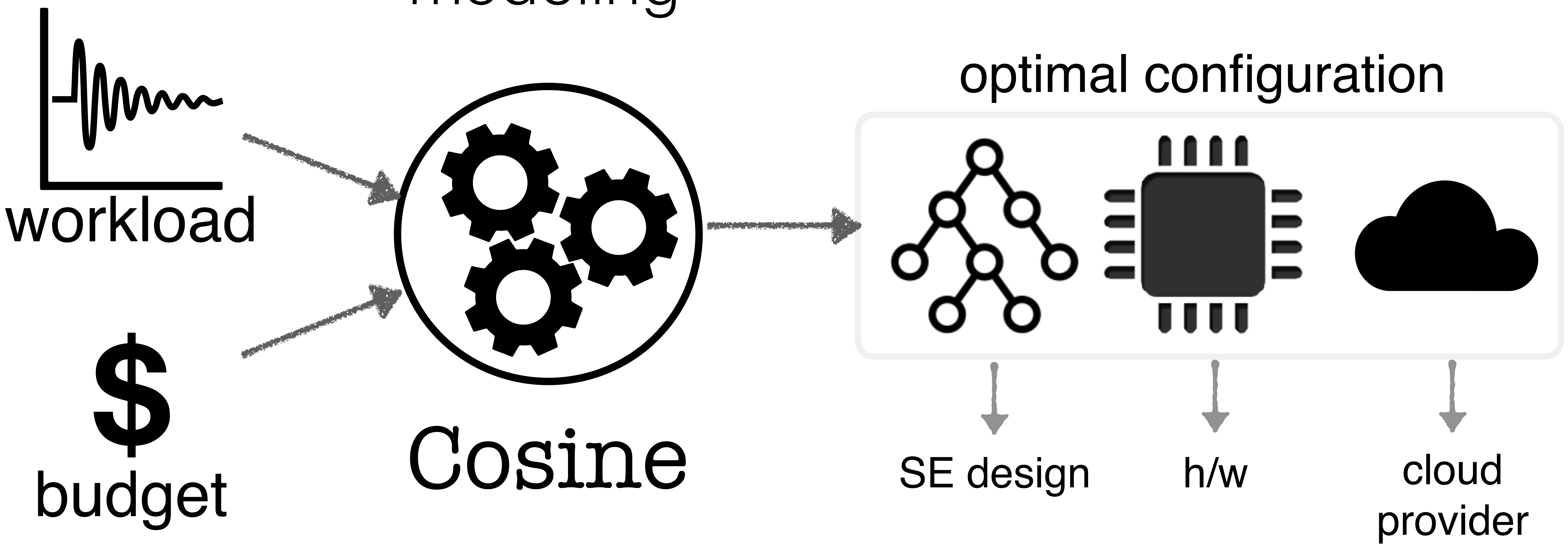
worst-case read cost: $1 + \sum_{i=1}^{L-1} \phi_i$

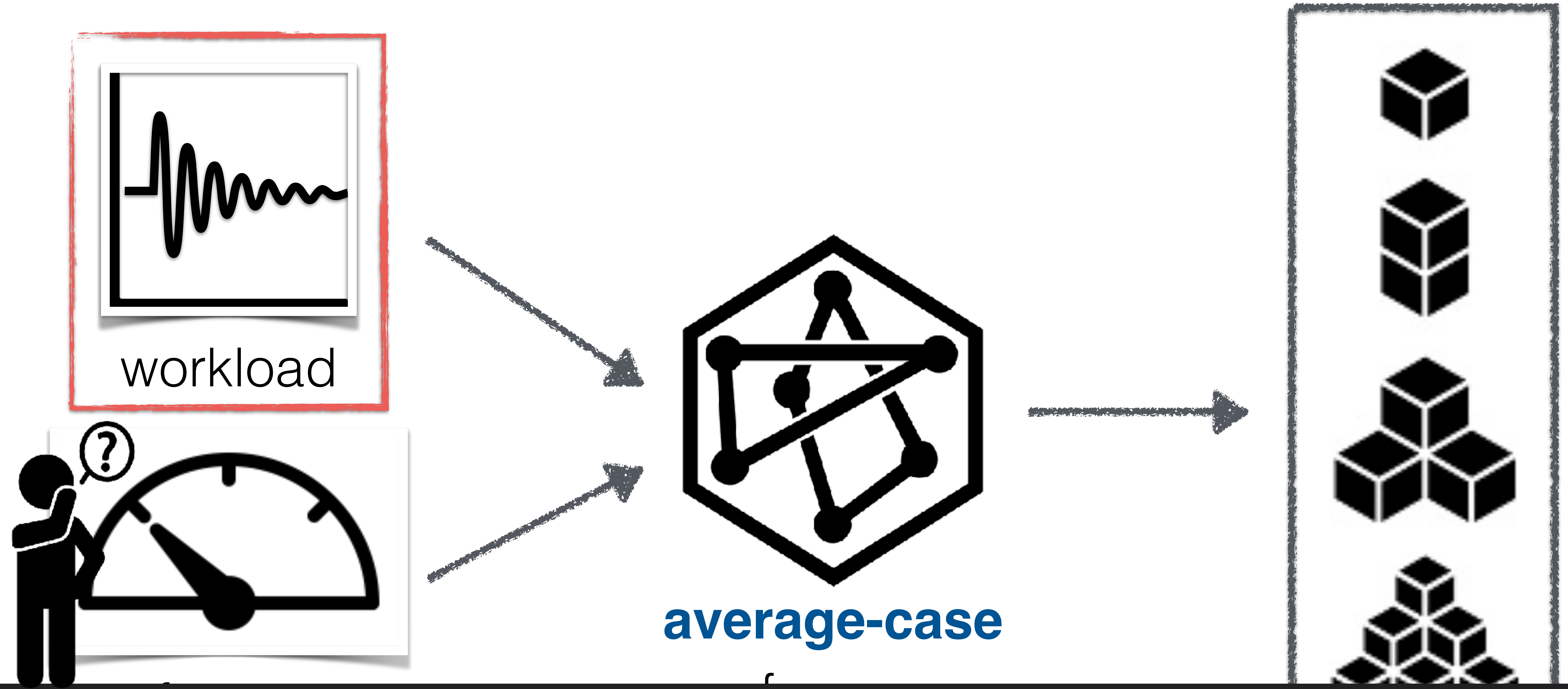
average-case performance modeling

$$\sum_{i=1}^{L} (\mathbb{P}[\text{query in } L_i] \cdot (1 + \sum_{j=1}^{i-1} \phi_i))$$

HuynhVLDB22 ChatterjeeVLDB22

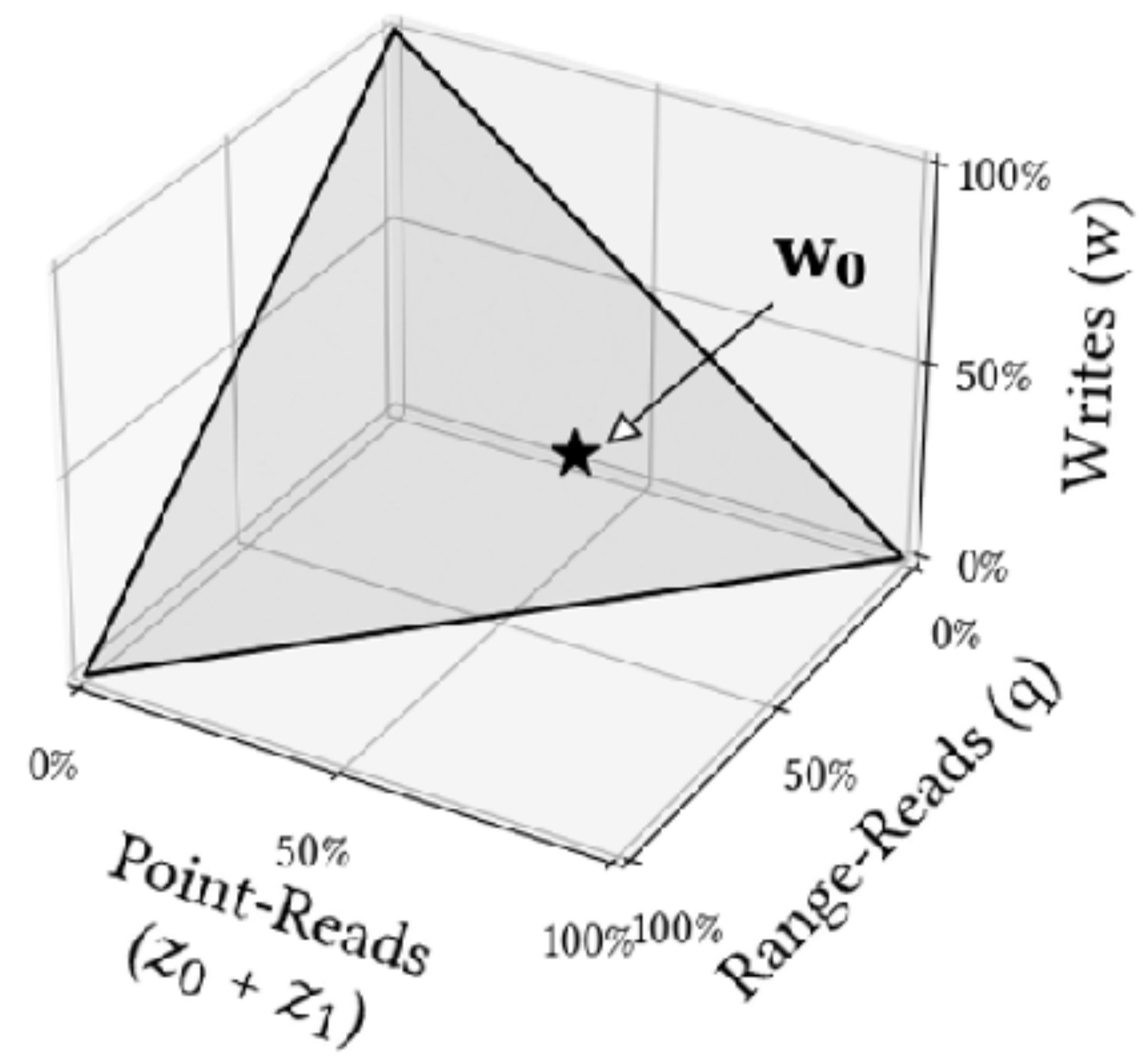
average-case performance modeling





What if the workload comes with **unpredictability**?

Workload-based Tuning



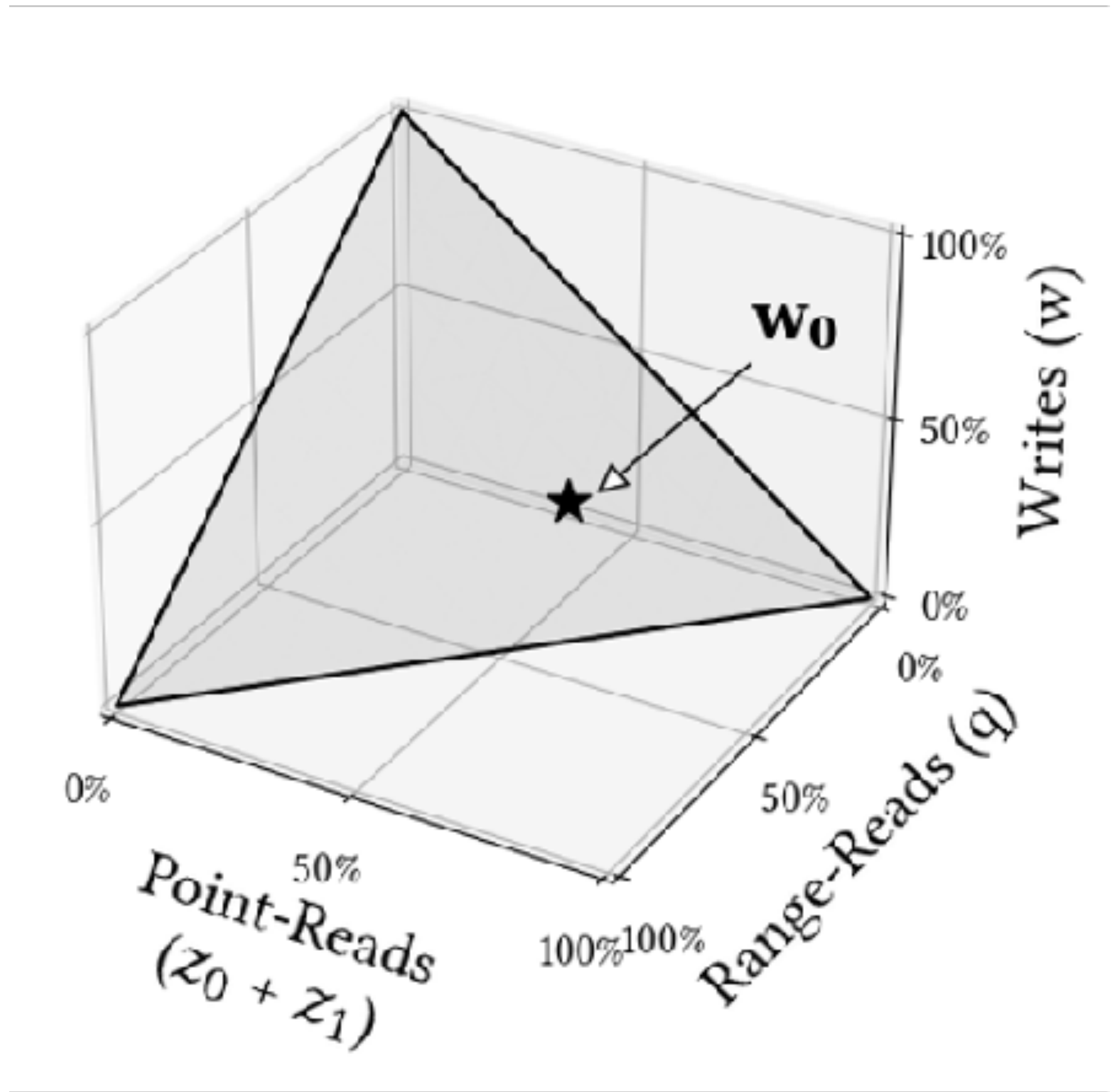
$$\pi_{w_0} = argmin_{\pi}(cost(w_0, \pi))$$

optimal configuration for w_0

$$cost(w_0, \pi_{w_0})$$

HuynhVLDB22

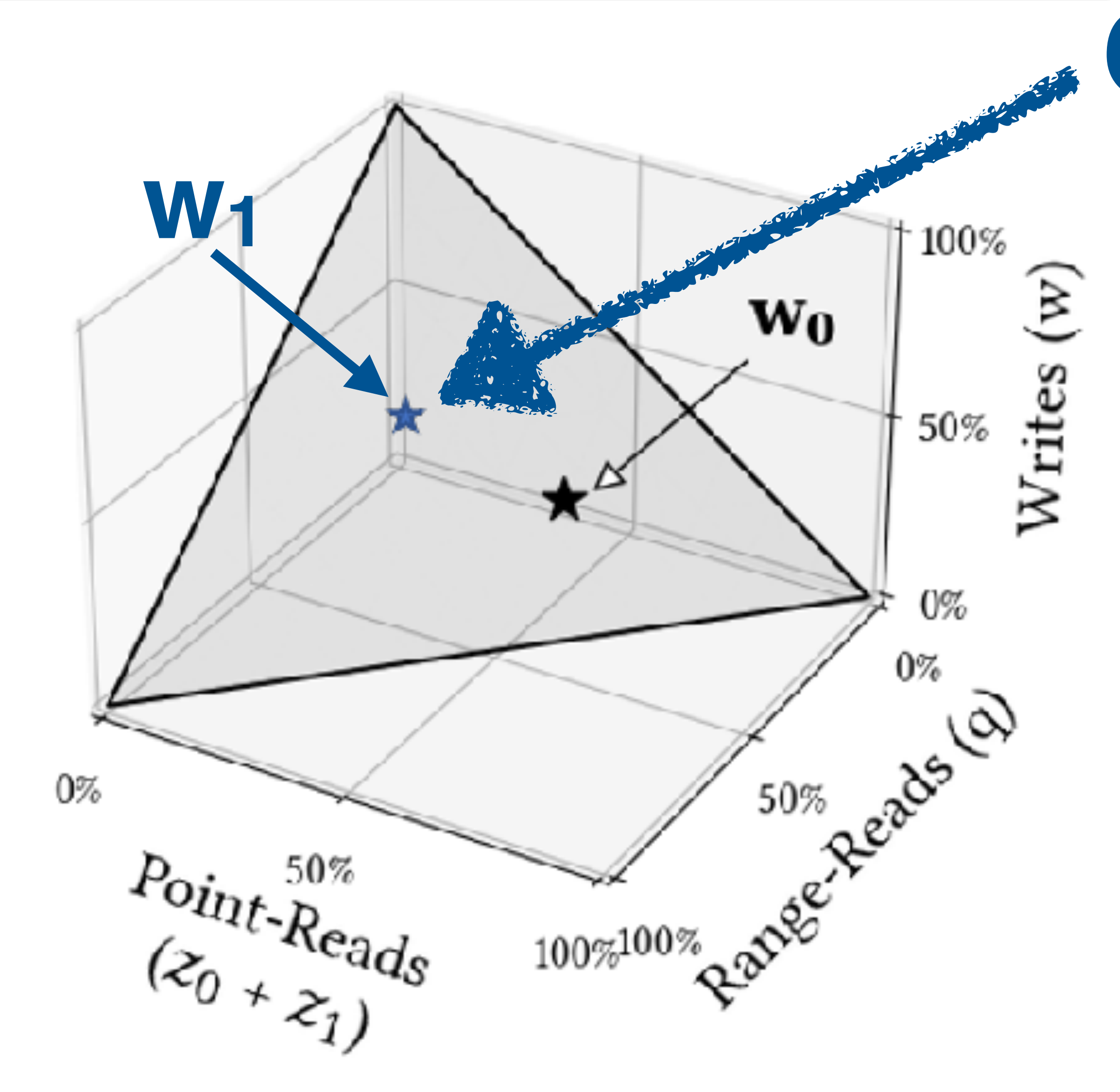
Nominal Tuning



$$\pi_{w_0} = argmin_{\pi}(cost(w_0, \pi)) \qquad cost(w_0, \pi_{w_0})$$

optimal configuration for w_0

same configuration



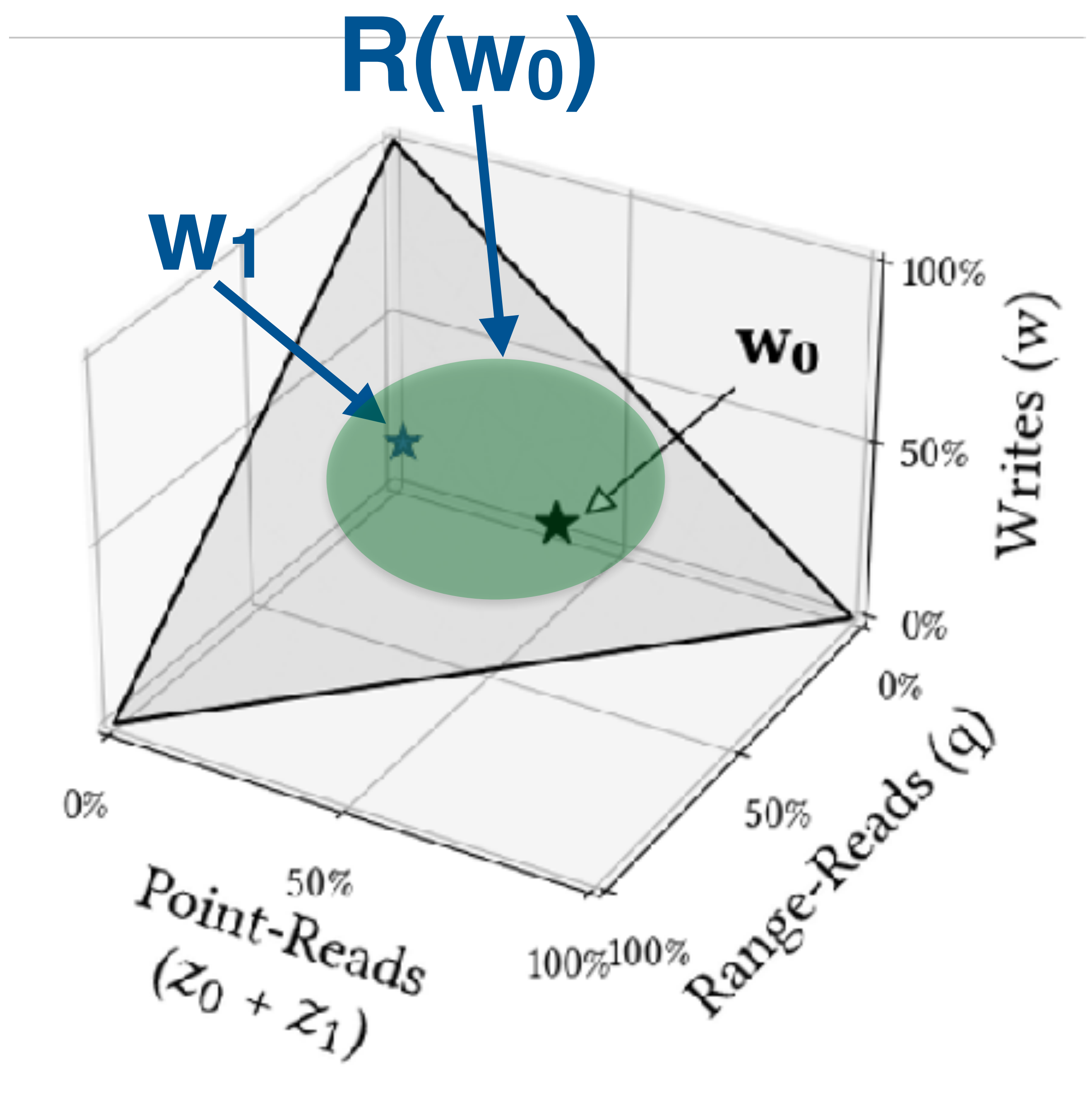
due to unpredictability

$$\pi_{w_0} \qquad cost(w_1, \pi_{w_0})$$

… but not optimal!

HuynhVLDB22

Robust Tuning



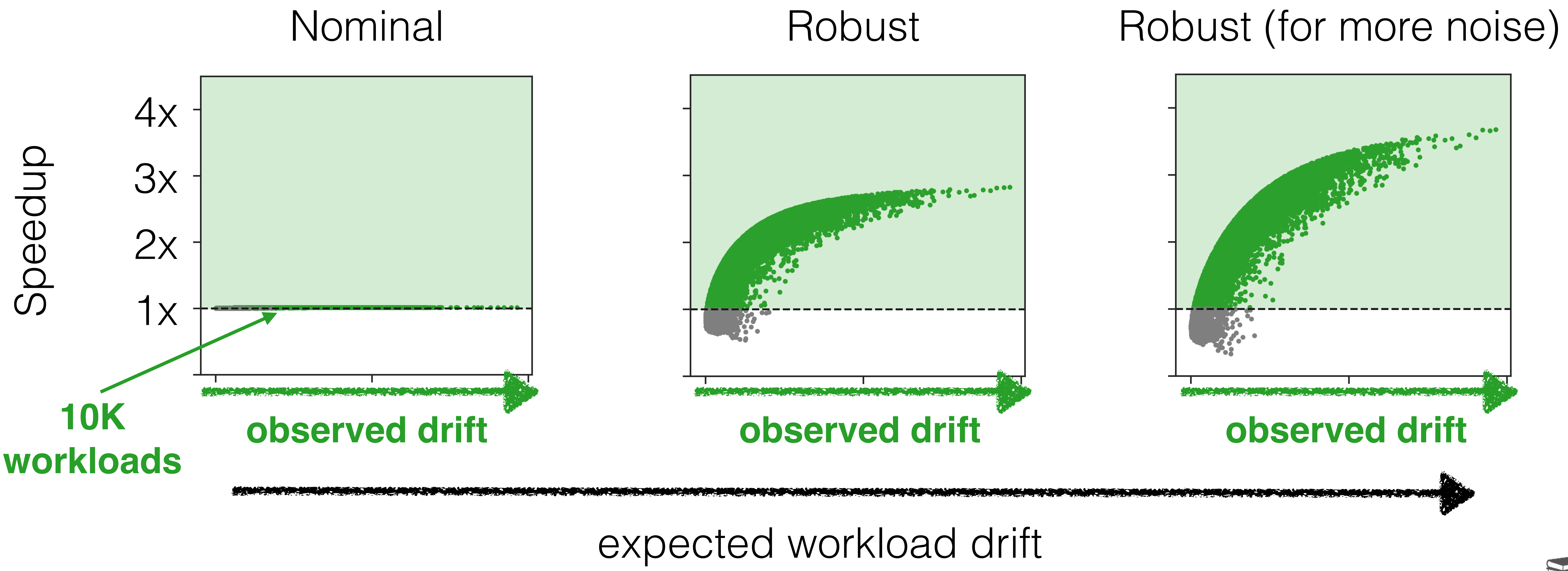
$$\pi^{robust}_{w_0} = argmin_{\pi} max_{w' \in R(w_0)} (cost(w_0, \pi^{robust}_{w_0}))$$

$$cost(w_1, \pi^{robust}_{w_0})$$

… close-to-optimal!

HuynhVLDB22

Robust Tuning



HuynhVLDB22

The **Key Takeaways**

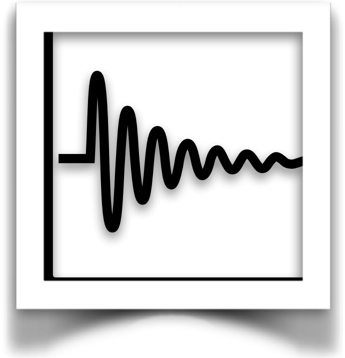
The LSM design space is **vast and complex**.

Read optimizations are crucial to make LSMs better.

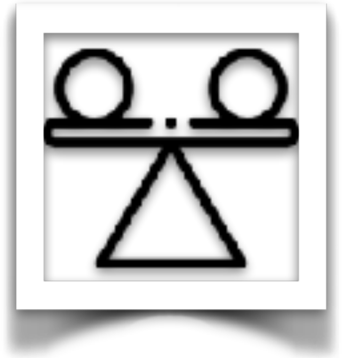
A **tuned LSM** engine can offer superior performance.

Open Research **Challenges**

Reduce write amplification



Workload-aware compactions & layout transformation



Performance Stability & Holistic Tuning

Automatic Tuning & Adaptive Behavior

Privacy-aware LSM designs

Please see our manuscript for all references!

REFERENCES

[1] H. Abu-Libdeh, D. Altınbüken, A. Beutel, E. H. Chi, L. Doshi, T. Kraska, Xiaozhou, Li, A. Ly, and C. Olston. Learned Indexes for a Google-scale Disk-based Database. In *Proceedings of the Workshop on ML for Systems at NeurIPS*, 2020.

[2] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. R. Borkar, Y. Bu, M. J. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y.-S. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V
AsterixDB: A S
Endowment, 7(

[3] Apache. Accun

[4] Apache. HBase

[5] Apache. Cassar

[6] M. Athanassou
Methods. In *Pro*
on Managemen

[7] M. Athanassou
A. Ailamaki, a
RUM Conjectu
Extending Data

[8] O. Balmau, F.
and D. Didona
Merge Key-Val
Trans. Comput.

[9] O. Balmau, R.
Unlocking Mer
the ACM Europ
80–94, 2017.

[10] M. A. Bender,
maul, D. Medje
Don't Thrash: I
VLDB Endowm

[22] N. Dayan, Y. Rochman, I. Naiss, S. Dashevsky, N. Rabinovich, E. Bortnikov, I. Maly, O. Frishman, I. B. Zion, Avraham, M. Twitto, U. Beitler, E. Ginzburg, and M. Mokryn. The End of Moore's Law and the Rise of The Data Processor. *Proceedings of the VLDB Endowment*, 14(12):2932–2944, 2021.

[23] N. Dayan and M. Twitto. Chucky: A Succinct Cuckoo Filter for LSM-Tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 365–378, 2021.

[24] N. Dayan, T. Weiss, S. Dashevsky, M. Pan, E. Bortnikov, and M. Twitto.

[44] J. Kim, S. Lee, and J. S. Vetter. PapyrusKV: a high-performance parallel key-value store for distributed NVM architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 57:1—-57:14, 2017.

[45] Y.-S. Kim, T. Kim, M. J. Carey, and C. Li. A Comparative Study of Log-Structured Merge-Tree-Based Spatial Indexes for Big Data. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 147–150, 2017.

[46] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. RadixSpline: a single-pass learned index. In *Proceedings of the International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM@SIGMOD)*, pages 5:1—-5:5, 2020.

[47] H. Kondylakis, N. Dayan, K. Zoumpatianos, and T. Palpanas. Coconut Palm: Static and Streaming Data Series Exploration Now in your Palm. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2019.

[48] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The Case for Learned Index Structures. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 489–504, 2018.

[49] Y. Li, Z. Liu, P. P. C. Lee, J. Wu, Y. Xu, Y. Wu, L. Tang, Q. Liu, and Q. Cui. Differentiated Key-Value Storage Management for Balanced I/O Performance. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 673–687, 2021.

[67] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 497–514, 2017.

[68] K. Ren, Q. Zheng, J. Arulraj, and G. Gibson. SlimDB: A Space-Efficient Key-Value Storage Engine For Semi-Sorted Data. *Proceedings of the VLDB Endowment*, 10(13):2037–2048, 2017.

[69] RocksDB. Leveled Compaction. *https://github.com/facebook/rocksdb/wiki/Leveled-Compaction*, 2020.

[70] RocksDB. Prefix Bloom Filter. *https://github.com/facebook/rocksdb/wiki/Prefix-Seek#configure-prefix-bloom-filter*, 2020.

[71] RocksDB. Block Cache. *https://github.com/facebook/rocksdb/wiki/Block-Cache*, 2021.

[72] S. Sarkar and M. Athanassoulis. Dissecting, Designing, and Optimizing LSM-based Data Stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2022.

[73] S. Sarkar, J.-P. Banâtre, L. Rilling, and C. Morin. Towards Enforcement of the EU GDPR: Enabling Data Erasure. In *Proceedings of the IEEE International Conference of Internet of Things (iThings)*, pages 1–8, 2018.

[74] S. Sarkar, K. Chen, Z. Zhu, and M. Athanassoulis. Compactionary: A Dictionary for LSM Compactions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2022.

[75] S. Sarkar, T. I. Papon, D. Staratzis, and M. Athanassoulis. Lethe:

LSM-Trees &
its **Read Optimizations**

Subhadeep Sarkar *Niv Dayan* *Manos Athanassoulis*

Thank You!

Questions?

BOSTON UNIVERSITY

UNIVERSITY OF TORONTO