Research Lecture: **LSM-trees & Filters**

Niv Dayan - Feb 7, 2023

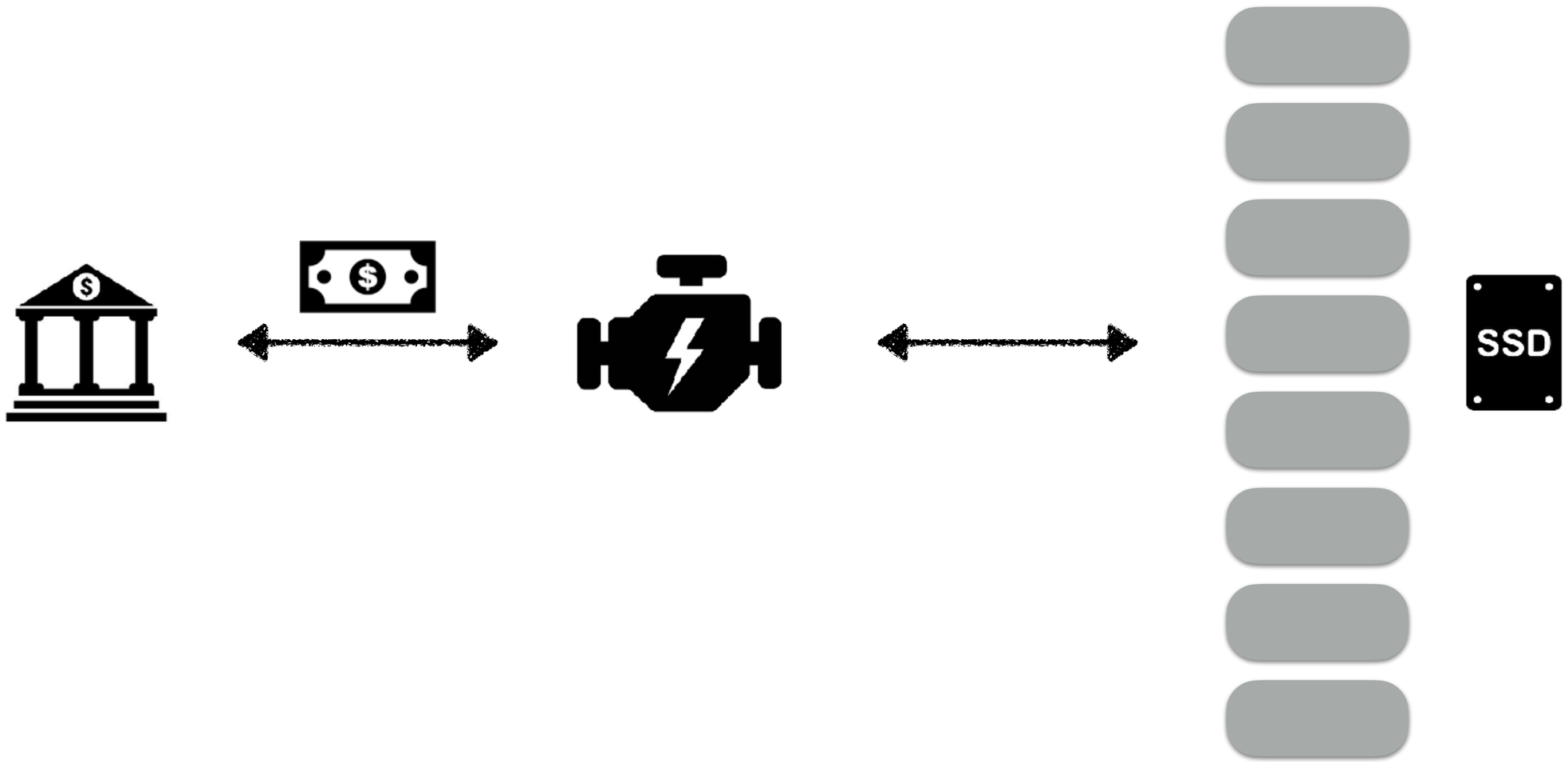
We will start at 2:10 pm

Instr **Results Sharing** 2/7/23

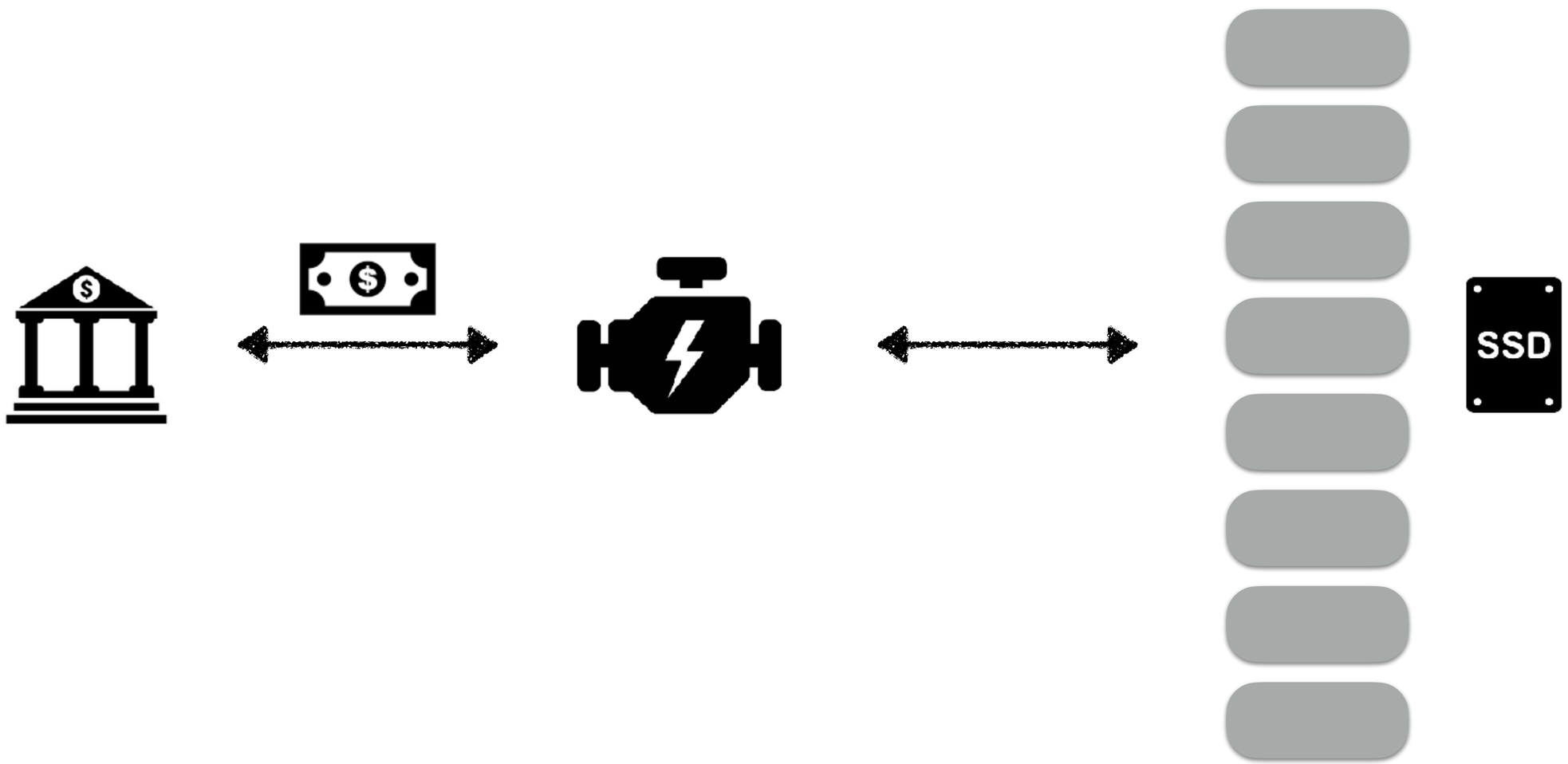
Hi Class, This post will be a forum to allow groups to share information on the experimental results they're gettin

- **Added to reading list**

Many DB operations are:

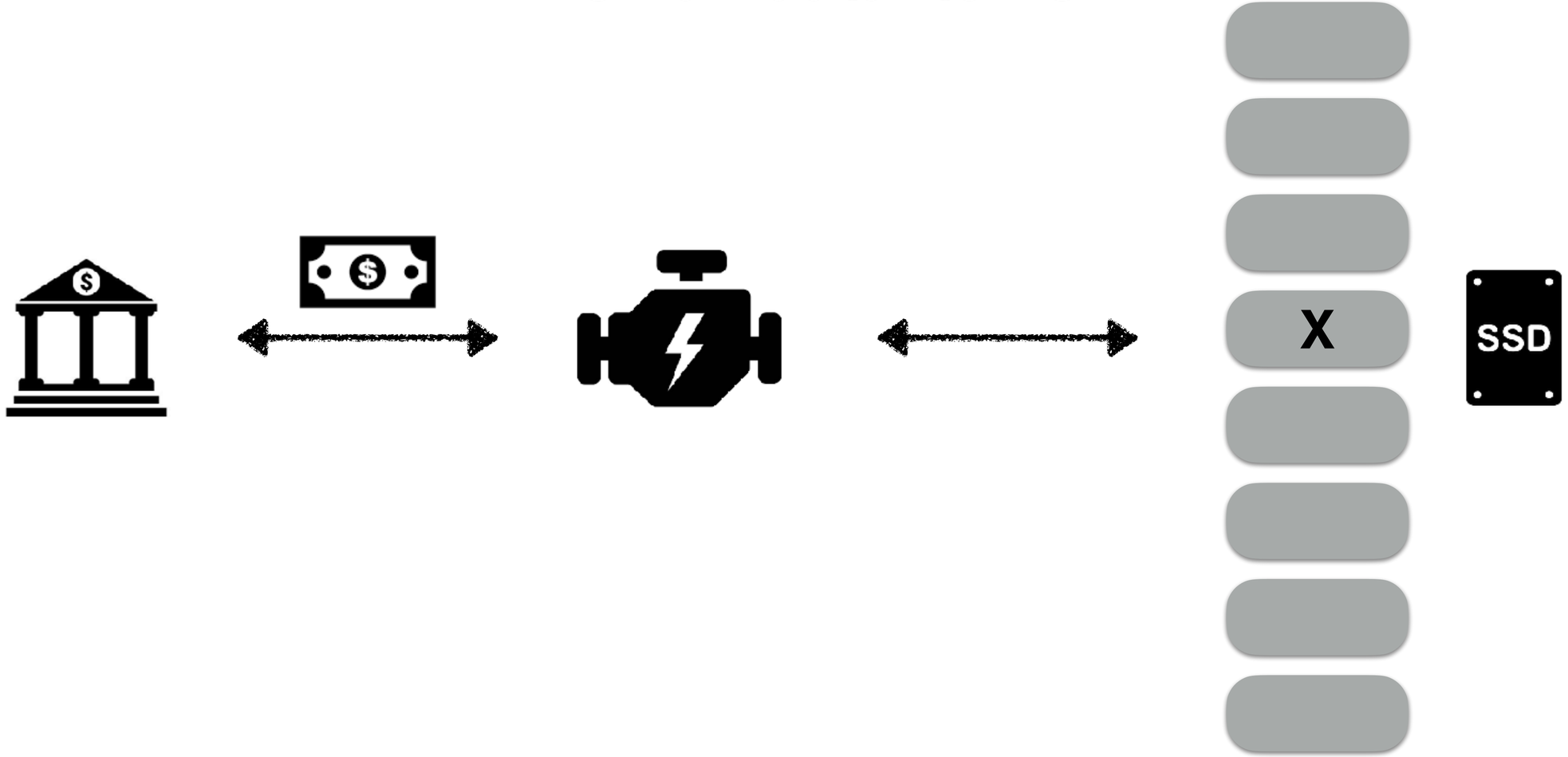


Many DB operations are: **selective**



Many DB operations are: **selective**

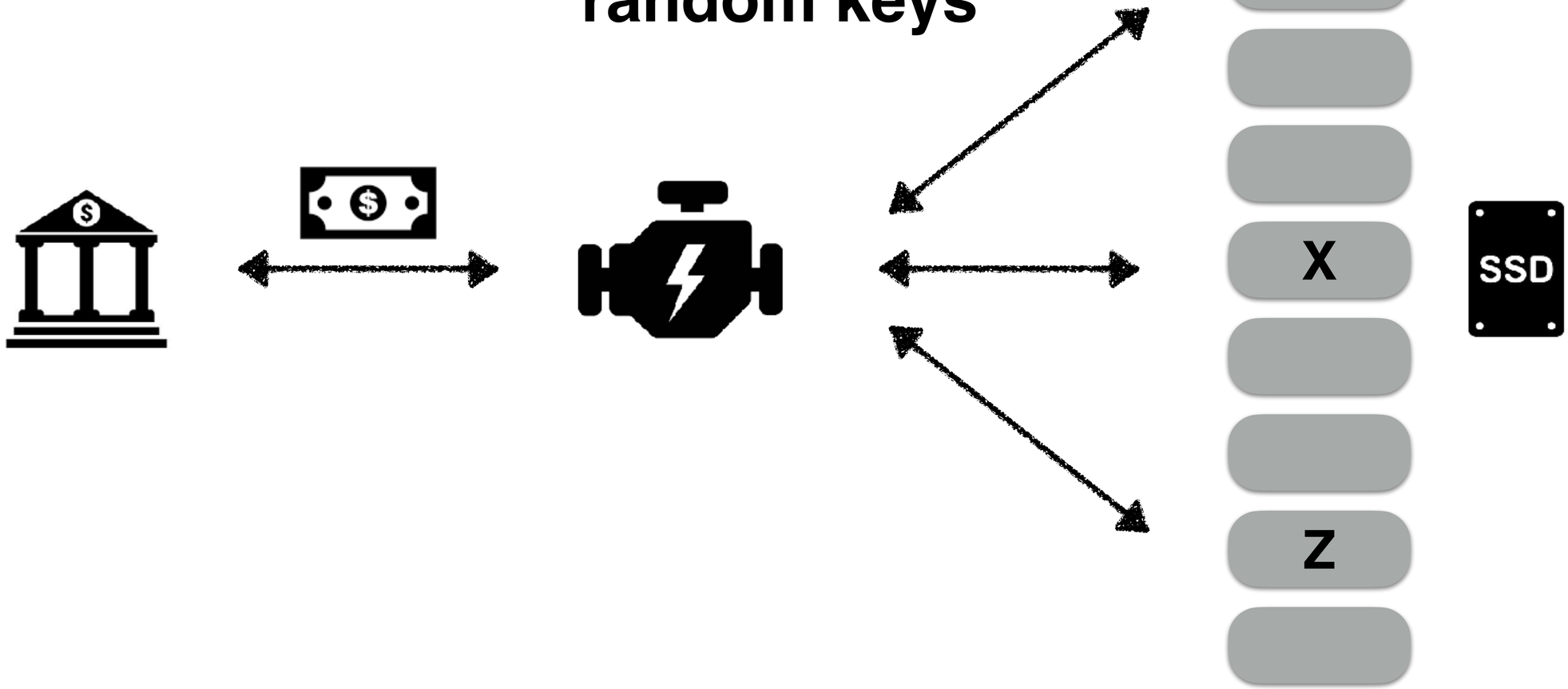
small data items



Many DB operations are: **selective**

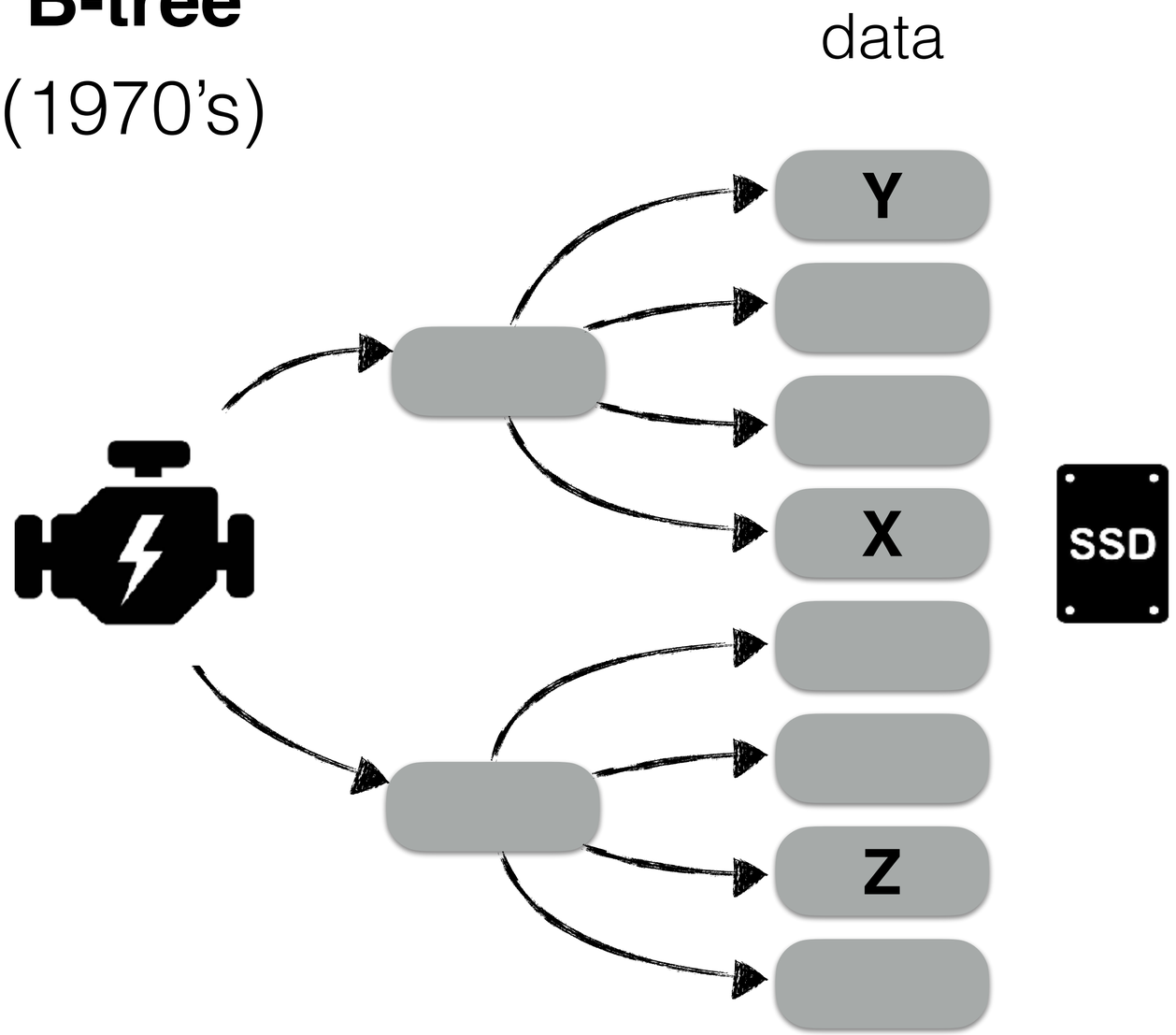
small data items

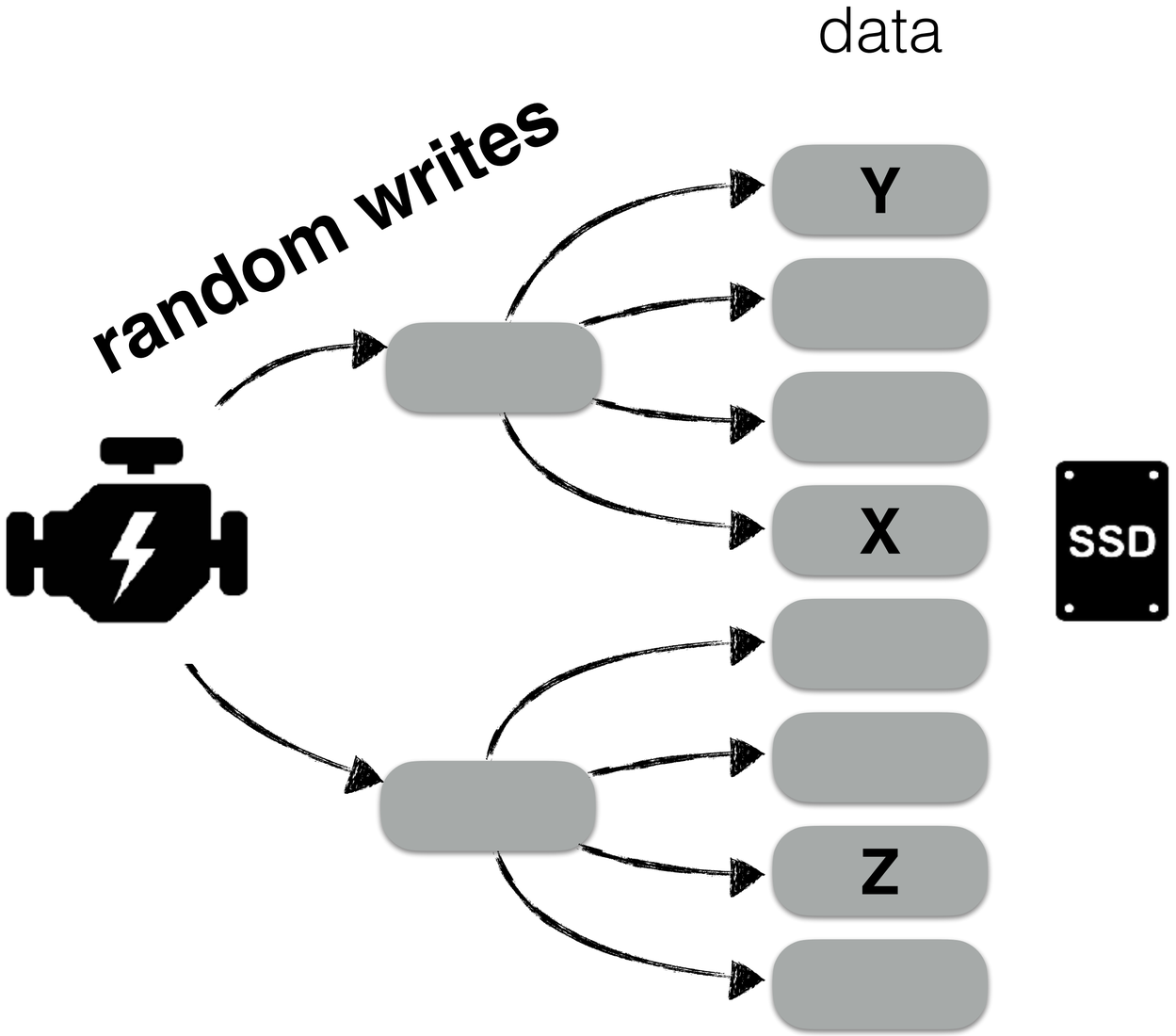
random keys



B-tree

(1970's)




data
random writes
Y
X
Z
SSD

random writes to small entries: a bad idea

random writes to small entries: a bad idea

mechanical latency

random writes to small entries: a bad idea

mechanical
latency

4KB access

random writes to small entries: a bad idea

mechanical
latency

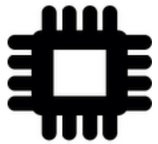
4KB access
garbage-collection

The Log-Structured Merge-Tree

1996 - Patrick O’Neil

LSM-Tree


Google's **BigTable**
Amazon's **DynamoDB**
Facebook's **RocksDB**
...
Apache **Cassandra**



buffer

merge-sort

1 3 6

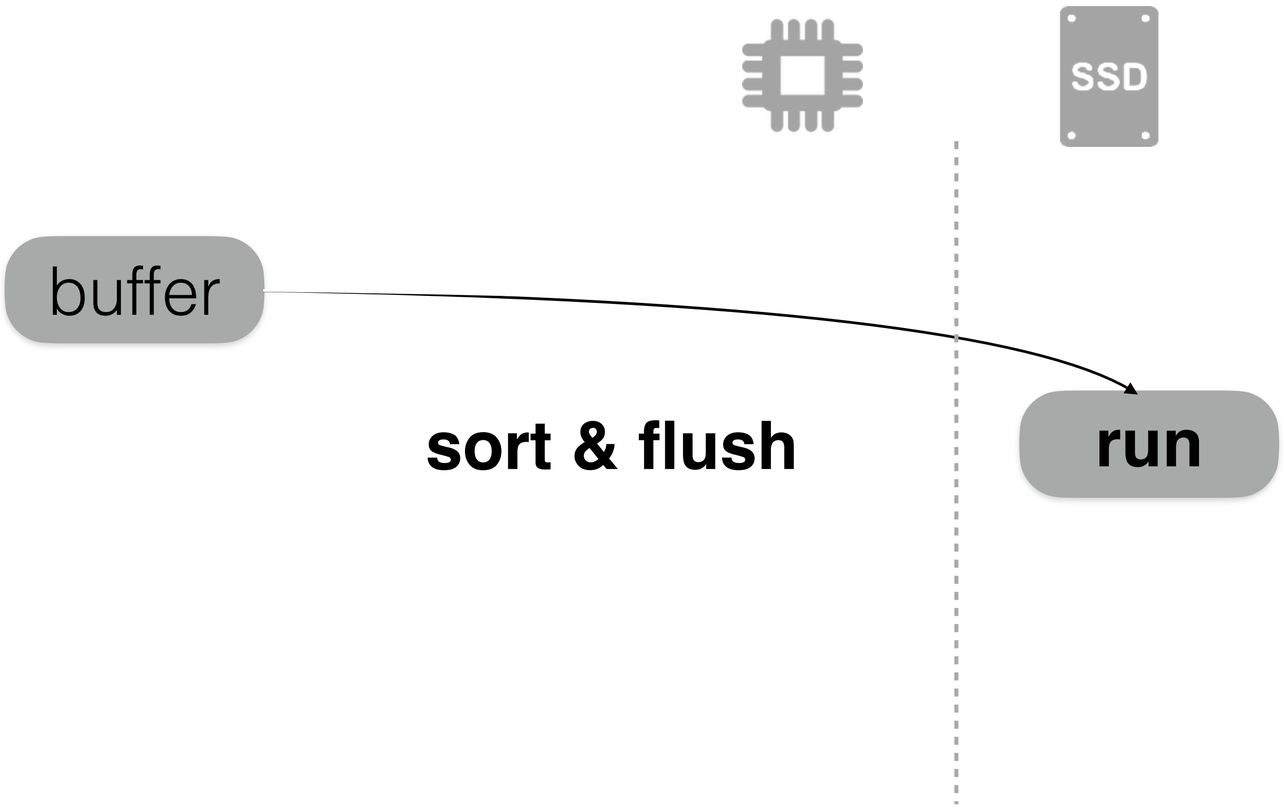
2 4 5

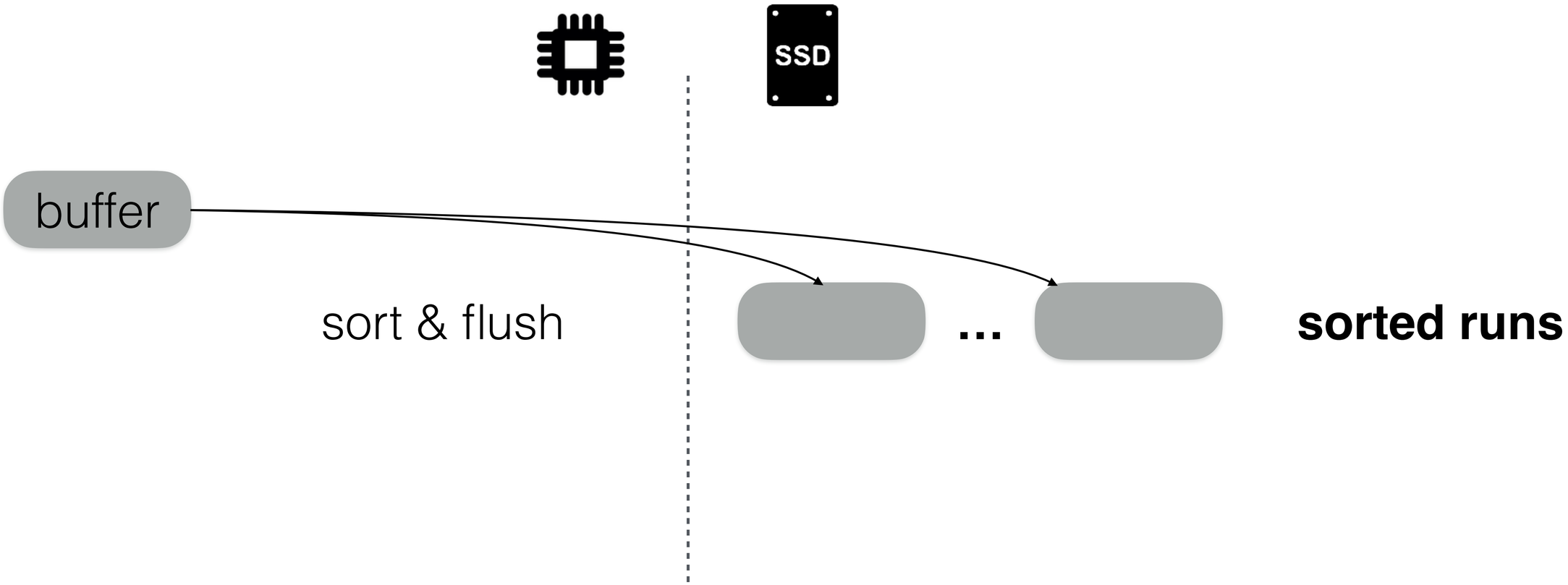
1 2 3 4 5 6

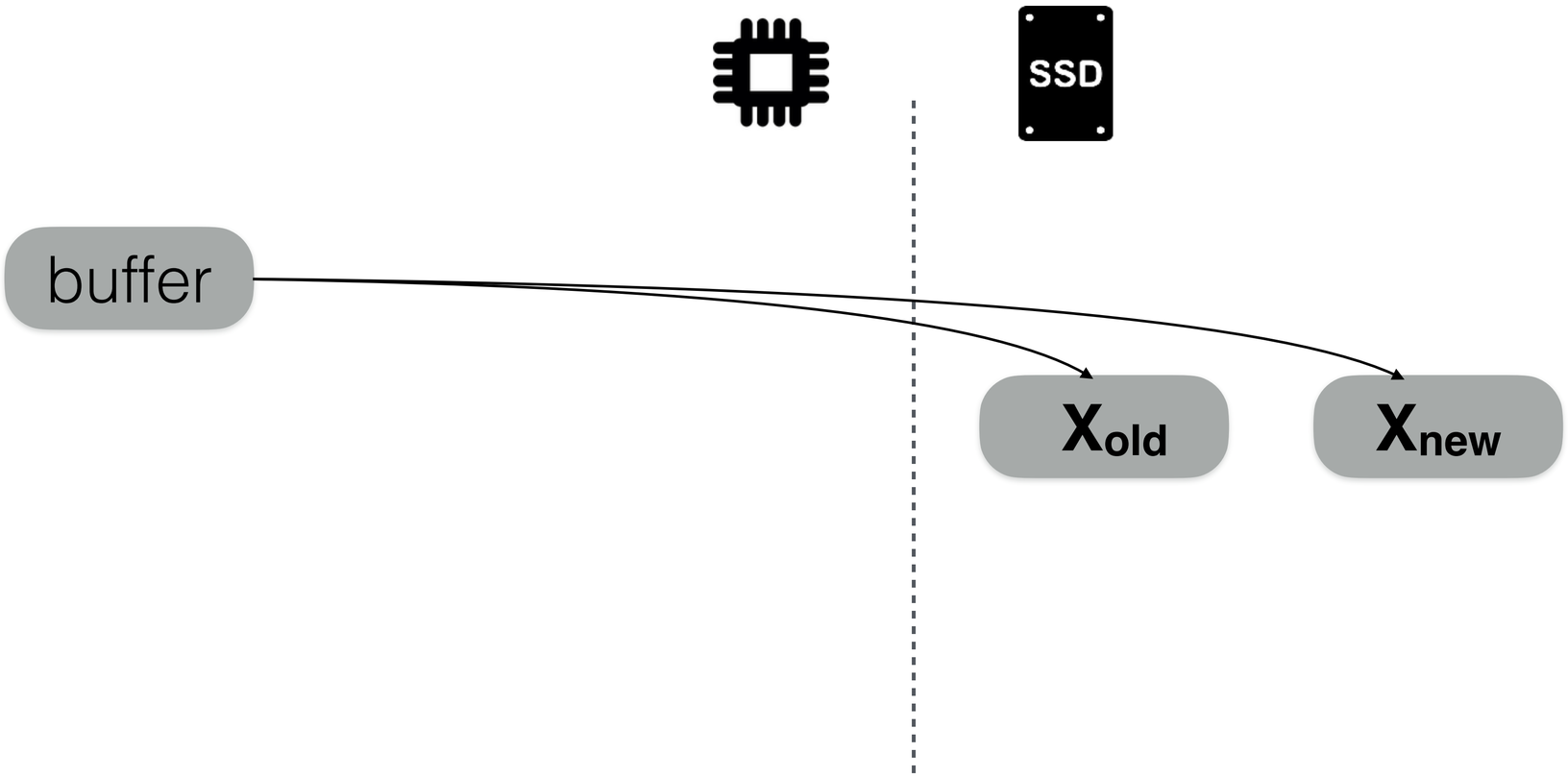
Inserts/updates/deletes
of key-value pairs

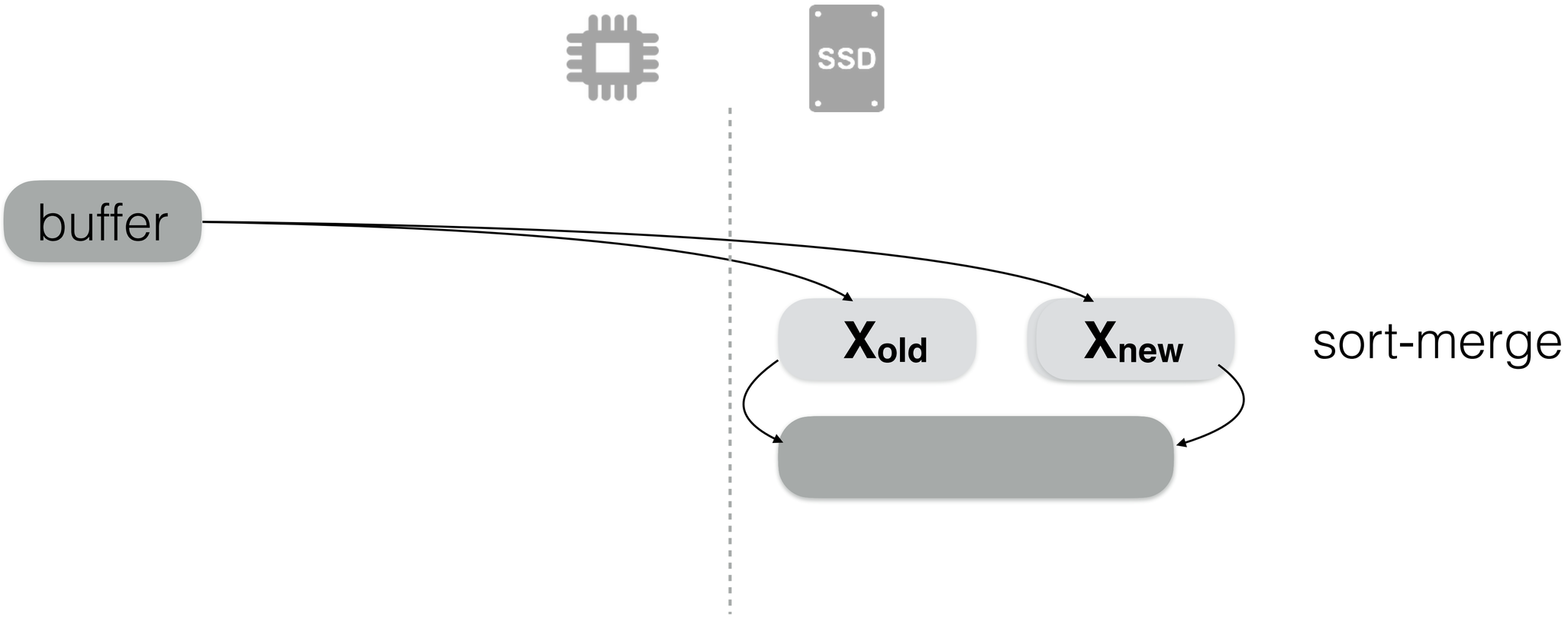
↓

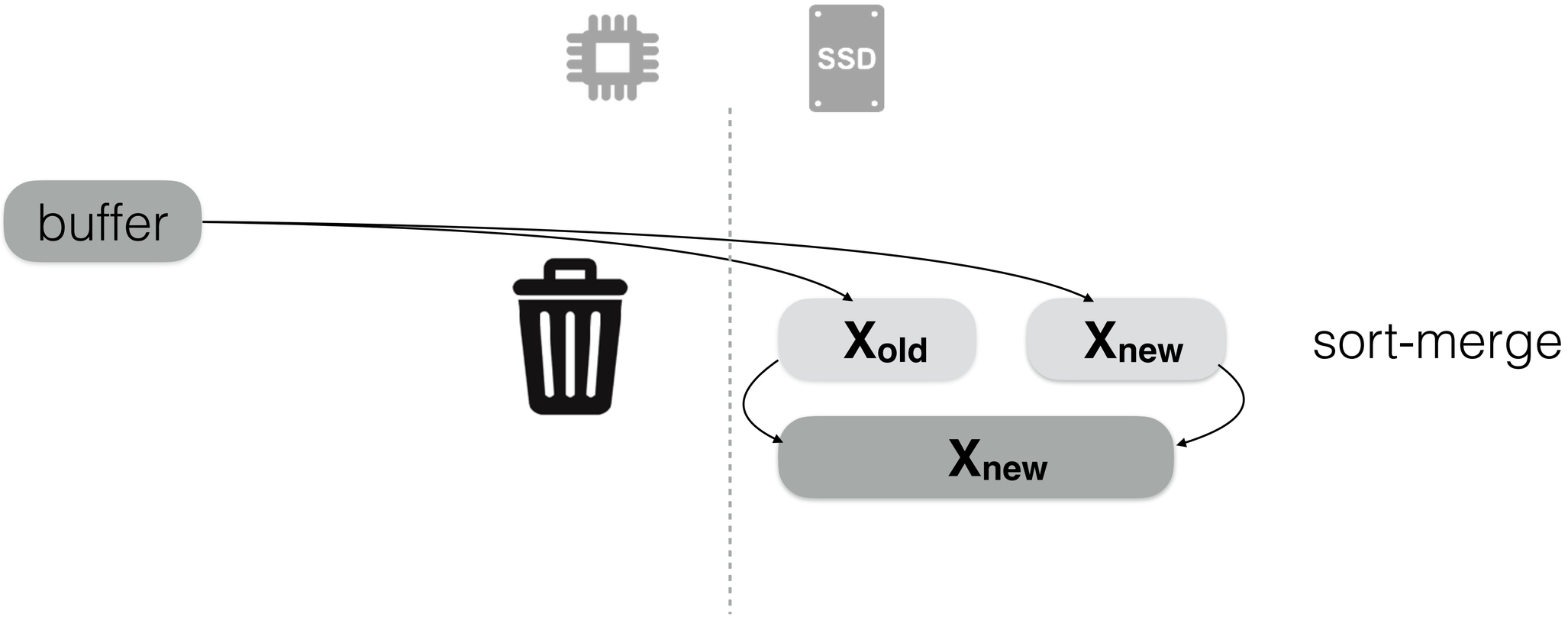
buffer

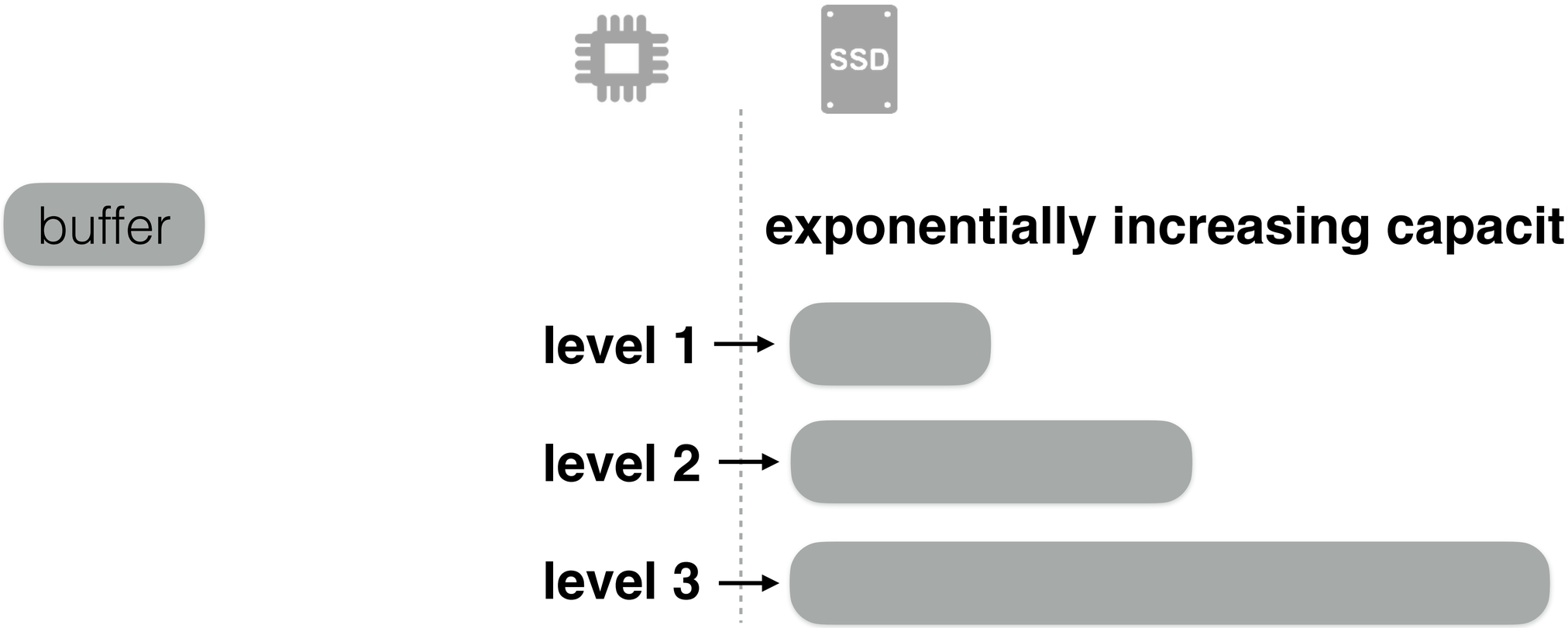

SSD
buffer
sort & flush
run

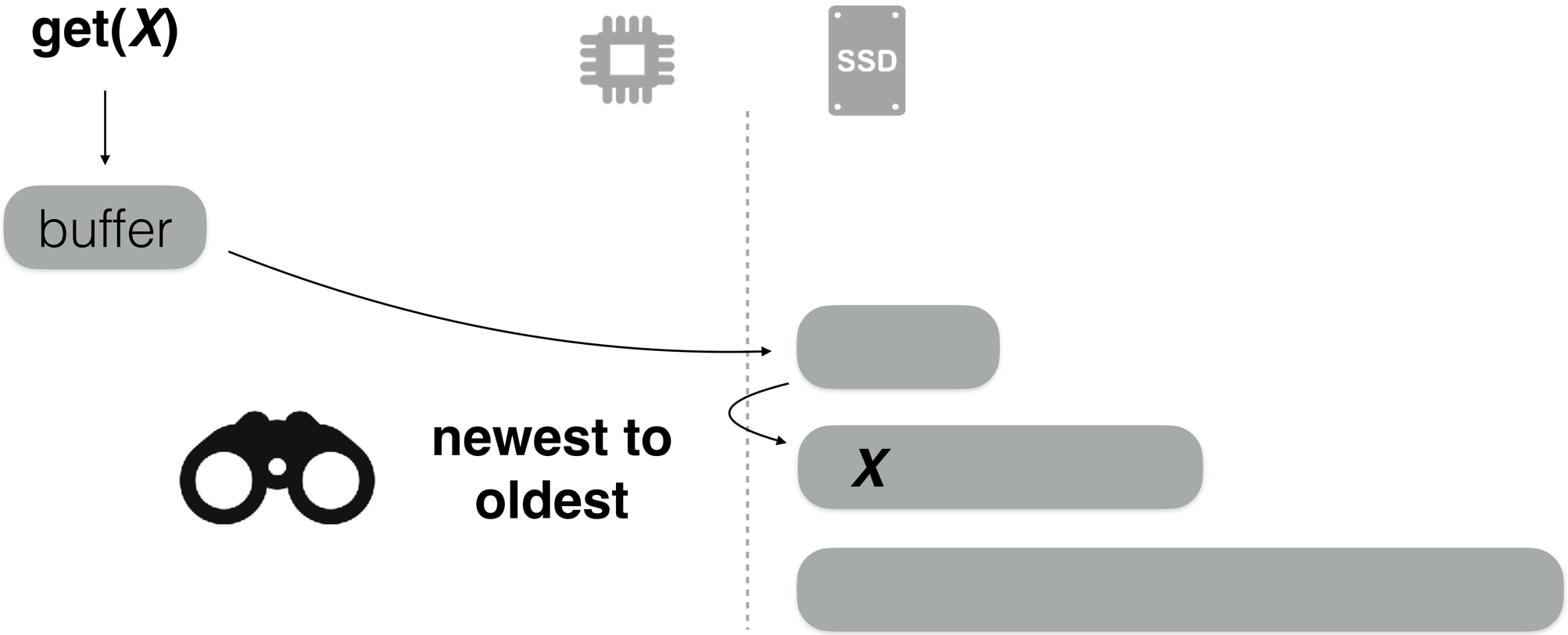

SSD
buffer
sort & flush
...
sorted runs

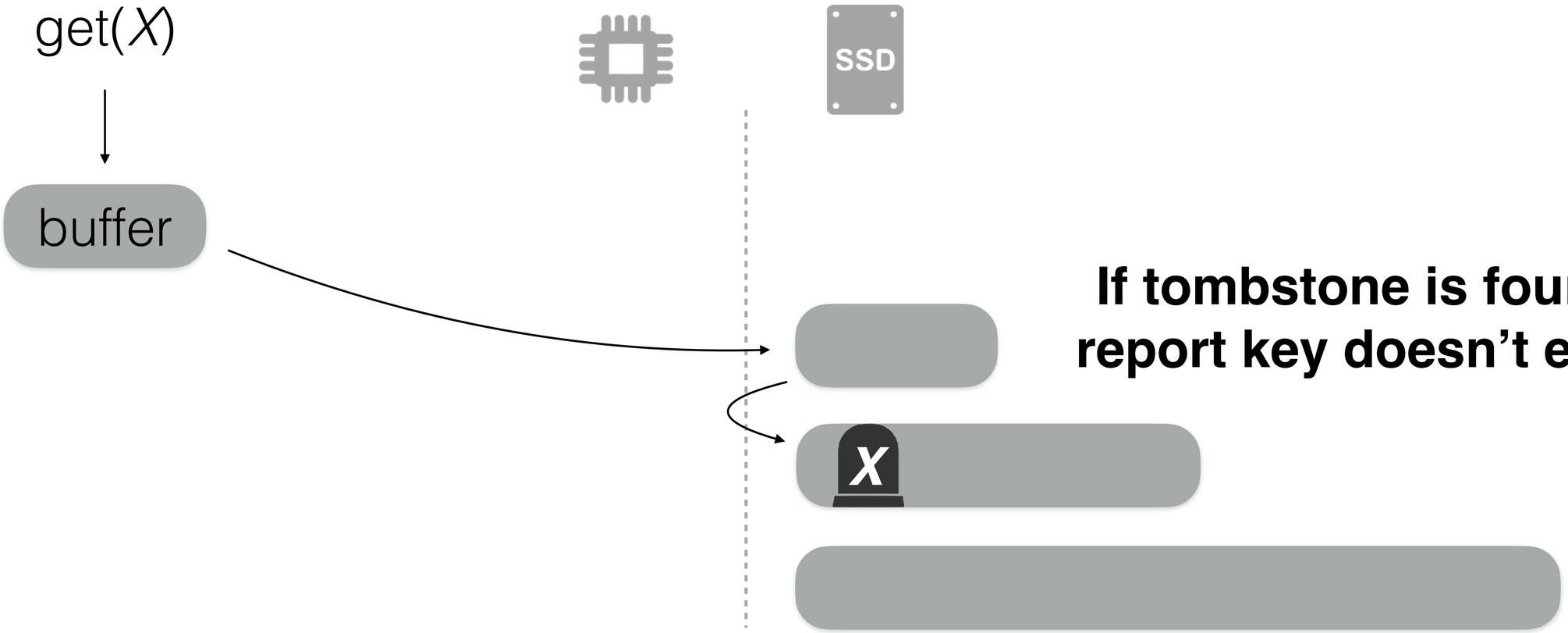

SSD
buffer
Xold
Xnew


SSD
buffer
X_{old}
X_{new}
sort-merge

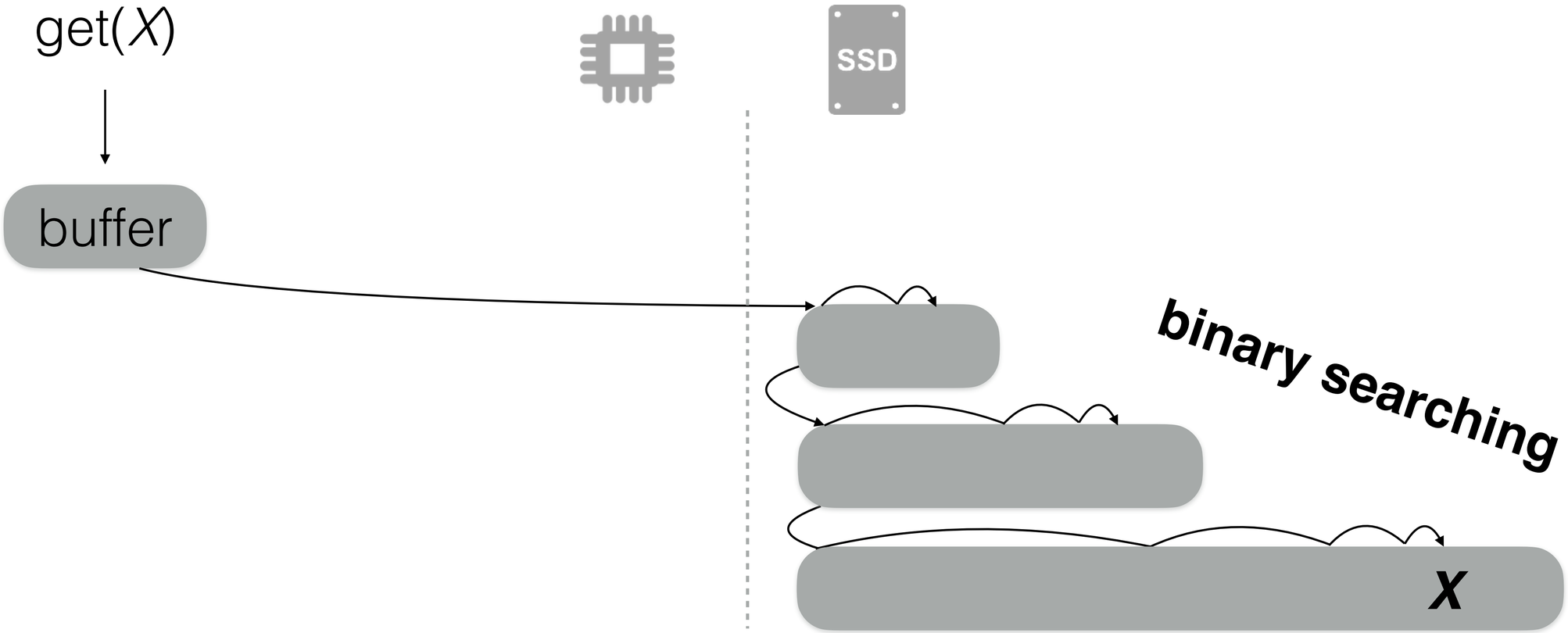

SSD
buffer
X_{old}
X_{new}
X_{new}
sort-merge

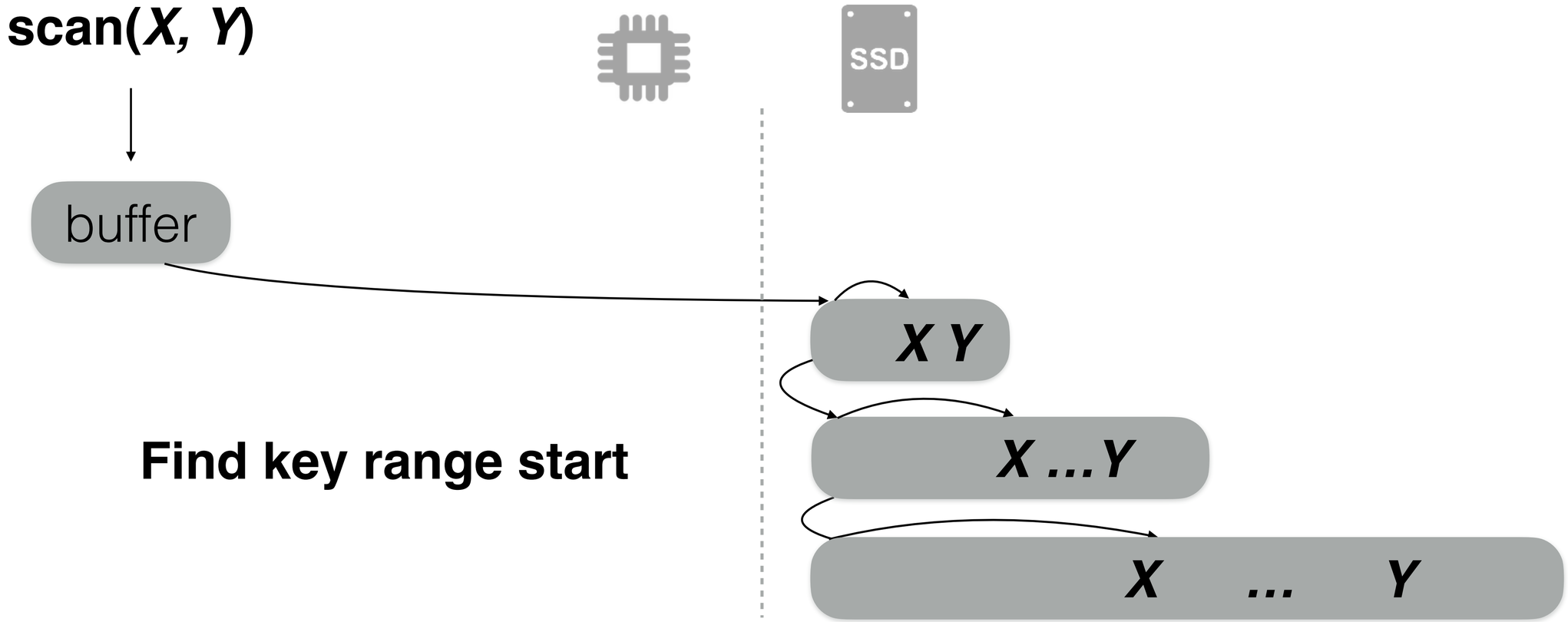

SSD
buffer
exponentially increasing capacities
level 1
level 2
level 3


get(X)
SSD
buffer
newest to
oldest
X



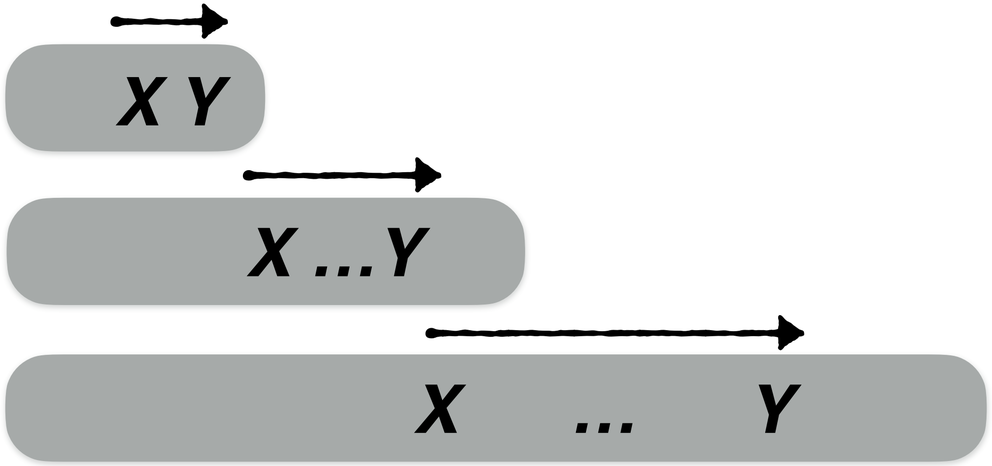
If tombstone is found, report key doesn’t exist

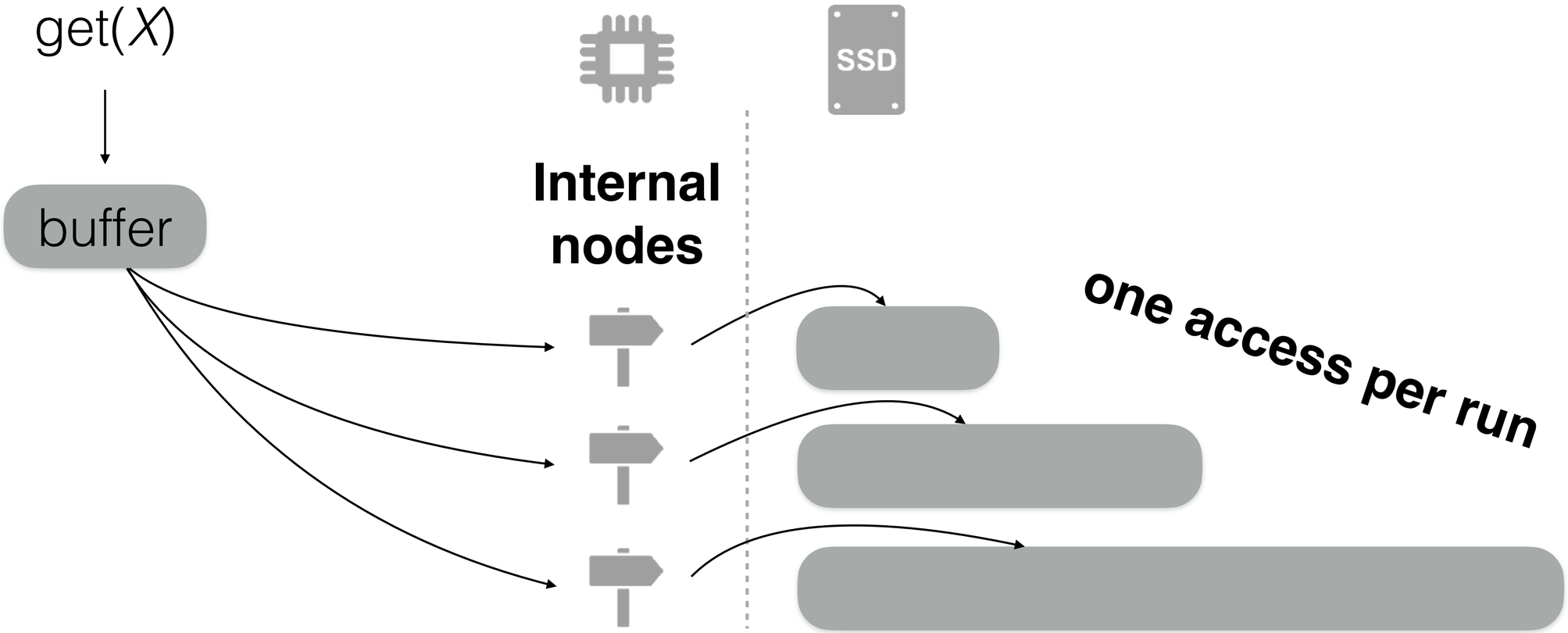

get(X)
SSD
buffer
binary searching
X

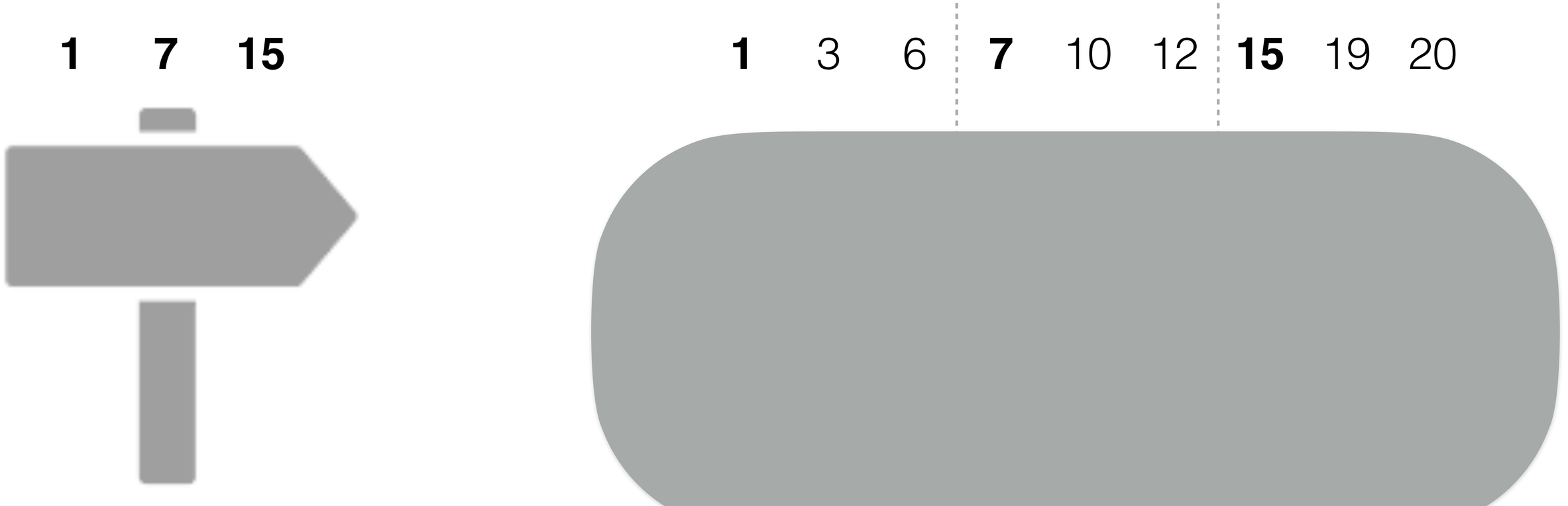

scan(X, Y)
SSD
buffer
X Y
X ... Y
X ... Y
Find key range start

scan(*X, Y*)

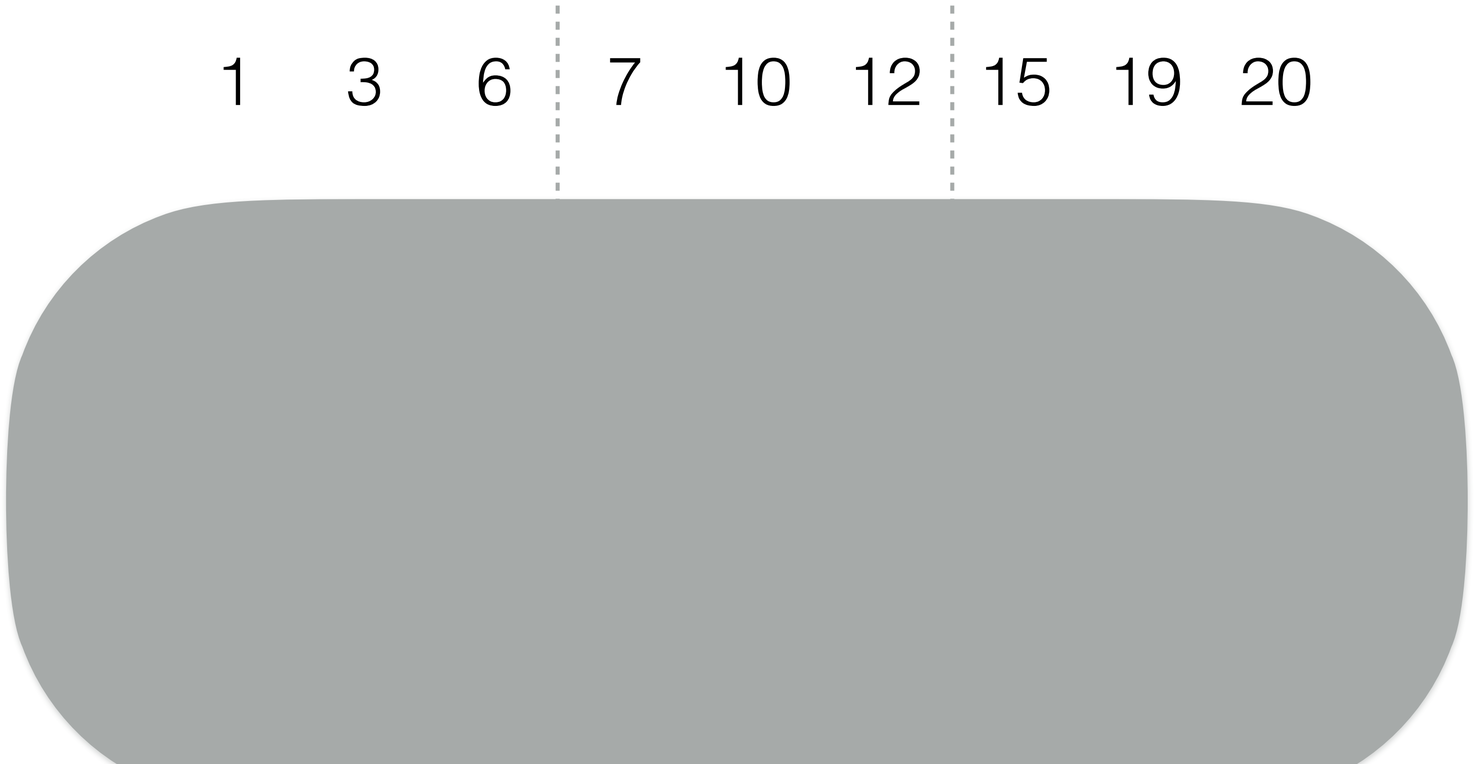
Scan & sort merge

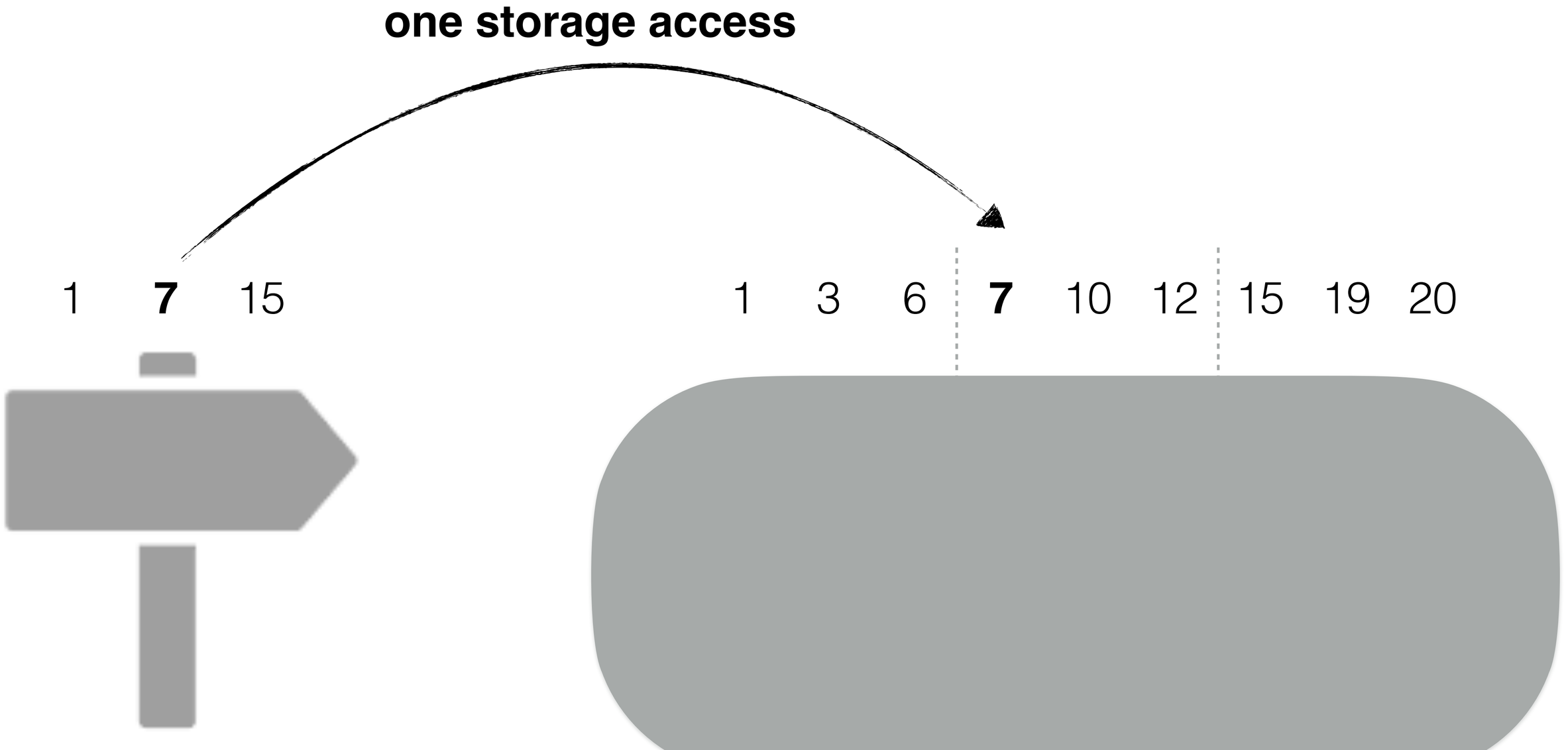



get(X)
buffer
SSD
Internal nodes
one access per run

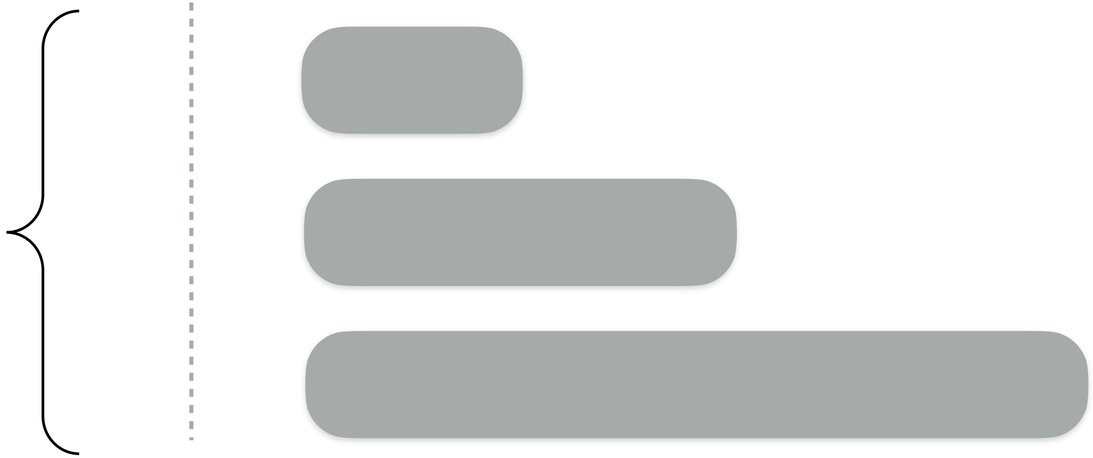

1 7 15
1 3 6 7 10 12 15 19 20

binary search




one storage access
1 7 15
1 3 6 7 10 12 15 19 20

levels $L = \log_R(N/P)$



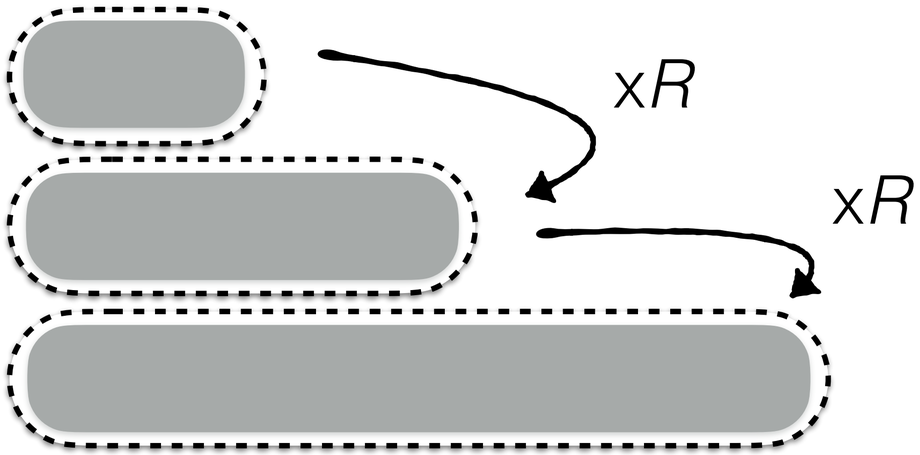
levels L = $\log_R(N/P)$

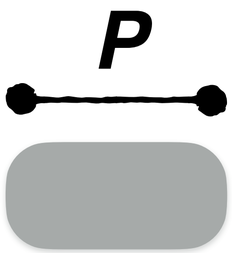
data size

levels L = $\log_R(N/P)$

size ratio

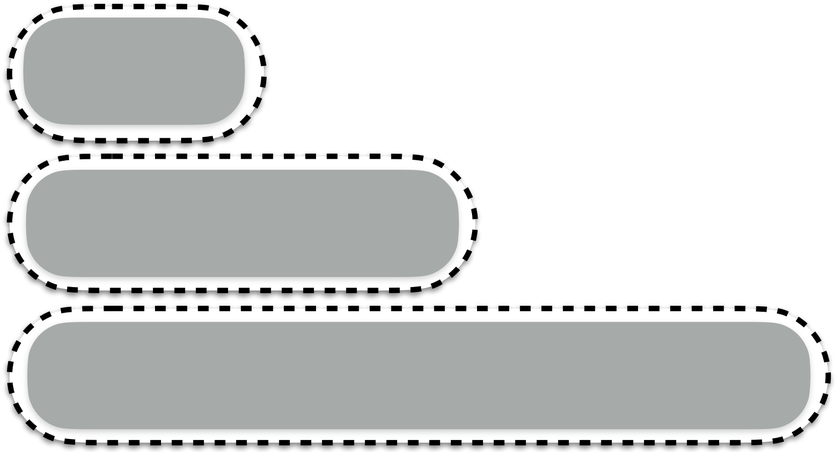
SSD

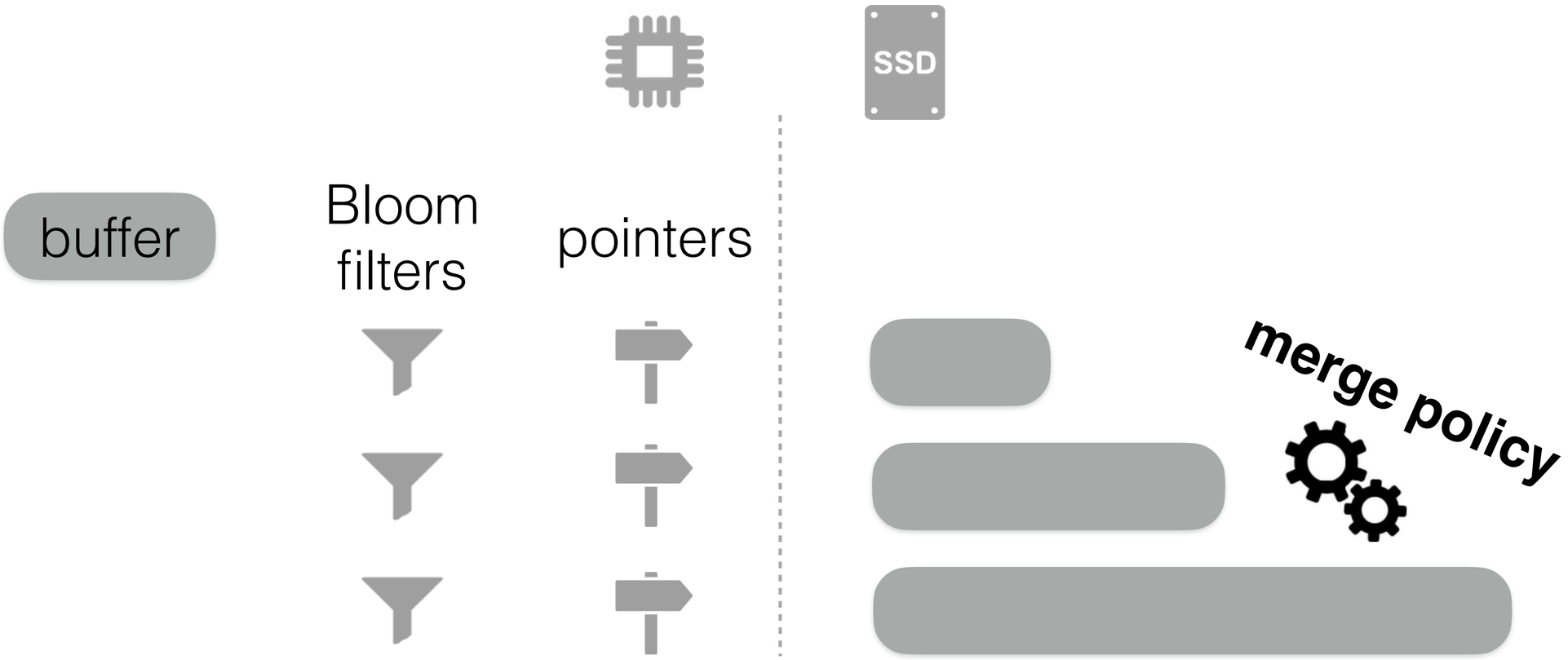




levels L = $\log_R(N/P)$

buffer size




SSD
buffer
Bloom filters
pointers
merge policy

writes

merge policy

reads

writes

merge policy

reads

two merge policies

Tiering

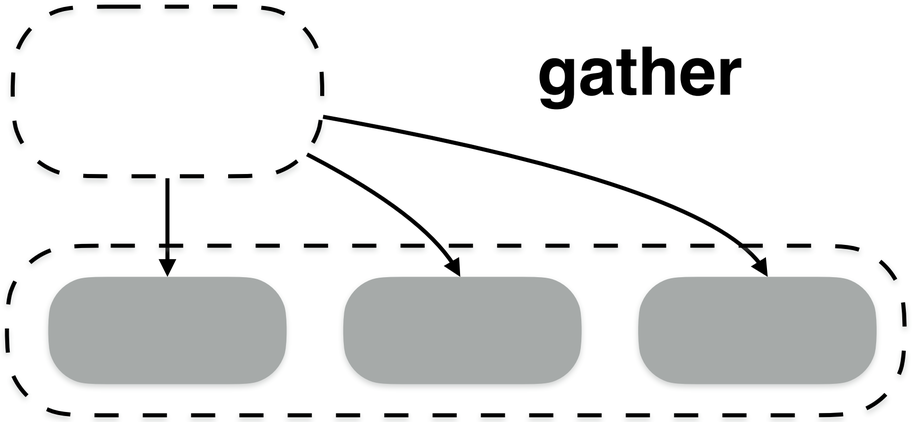
Leveling

Tiering

Leveling

Tiering

Leveling



Tiering

Leveling

gather

merge & flush

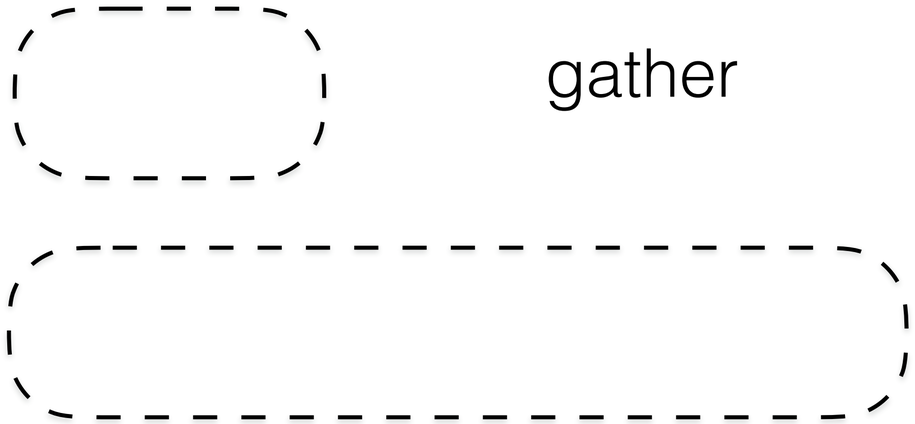
Tiering

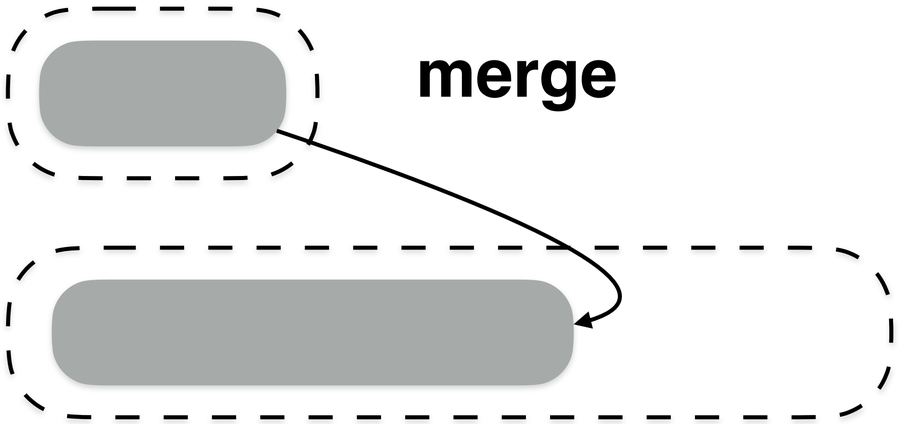
Leveling

gather

Tiering

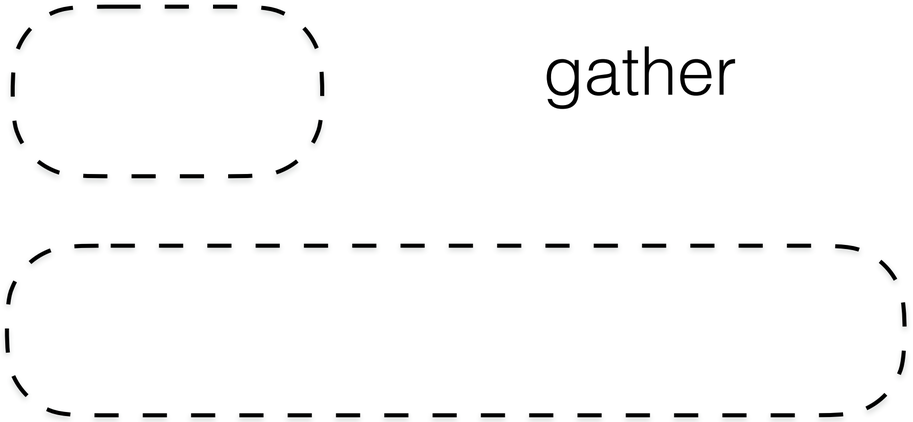
Leveling

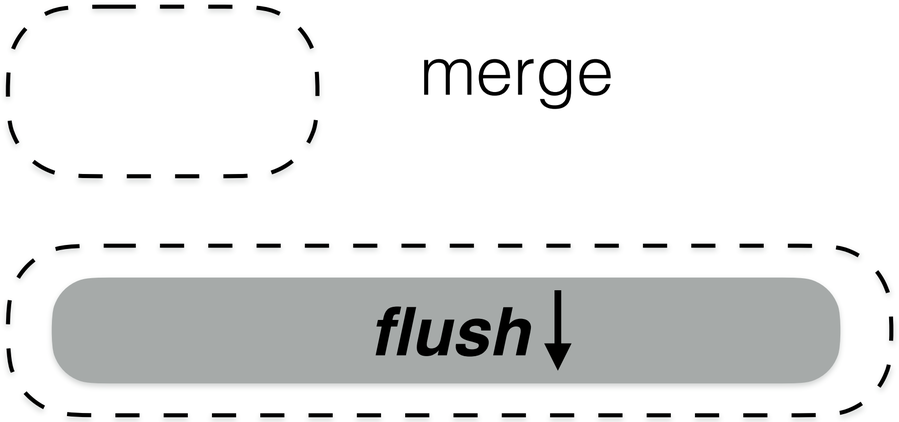




Tiering

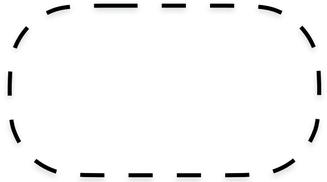
Leveling



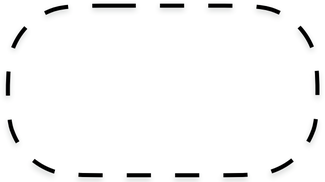


Tiering

Leveling



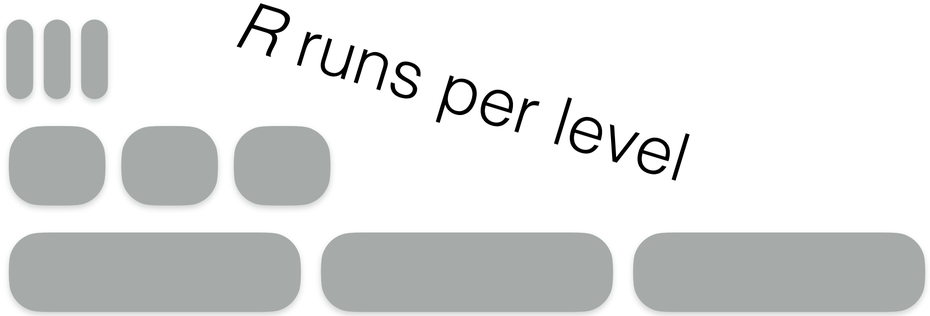
gather



merge

Tiering

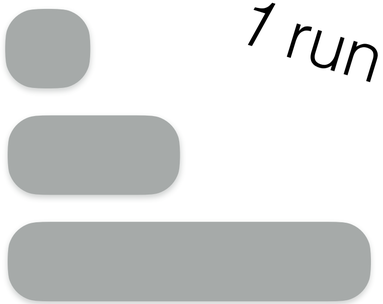
Leveling



size ratio *R*

Tiering

Leveling



1 run per level

1 run per level

size ratio *R* = 2

Tiering

Leveling

size ratio *R*

Tiering

Leveling

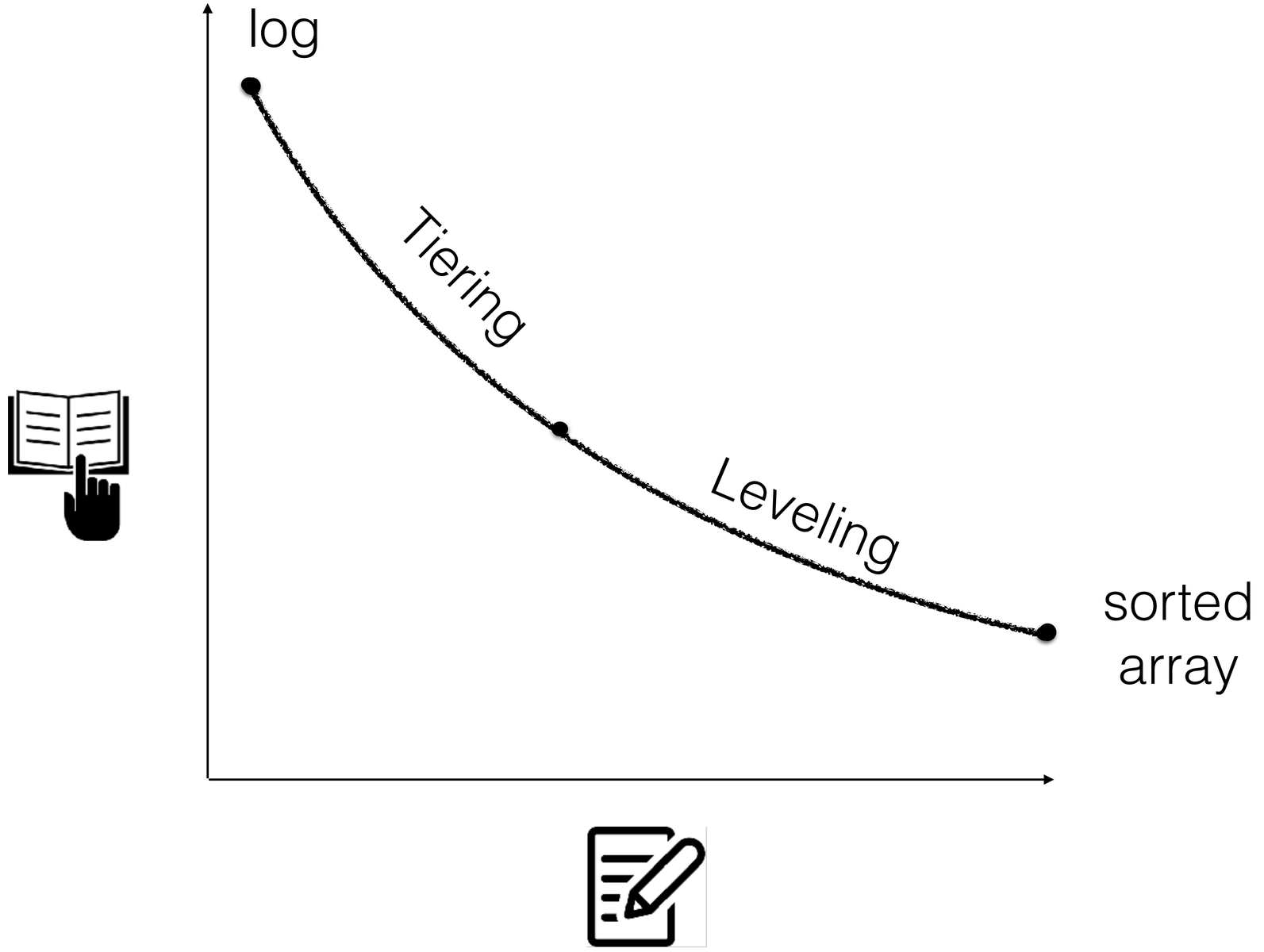
O(*N/P*) runs per level

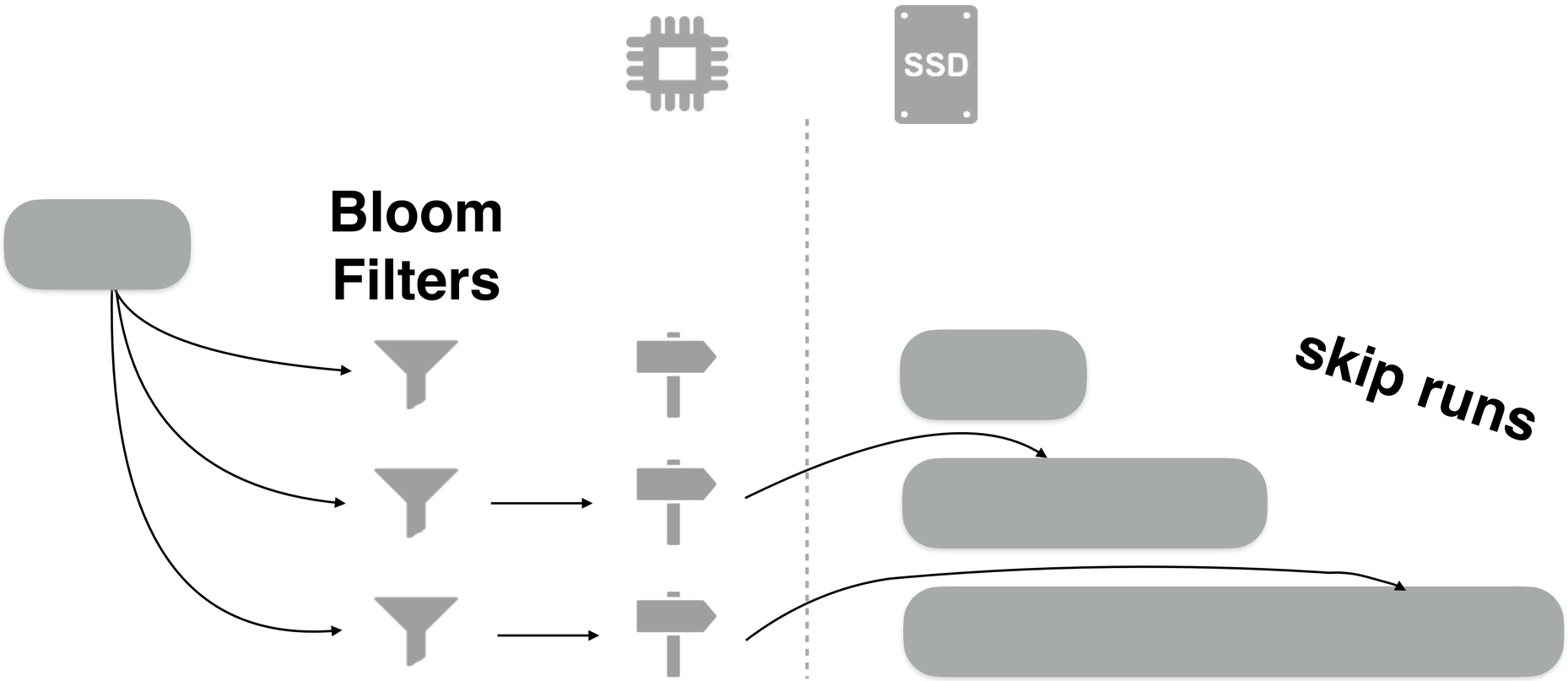
log

1 run per level

sorted array

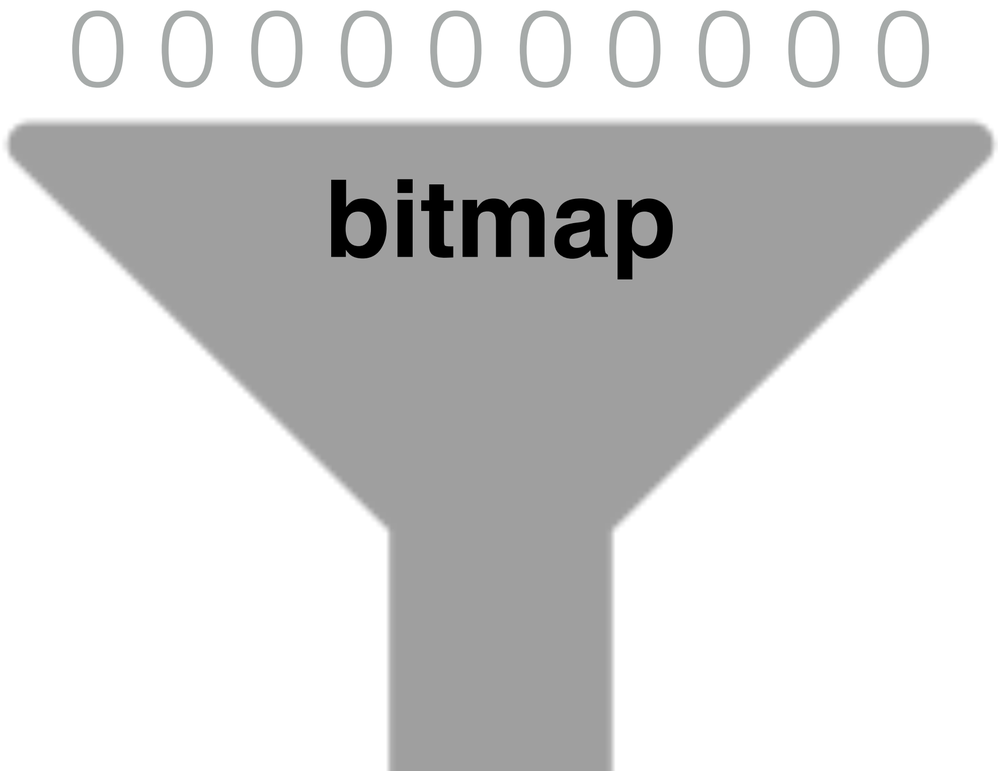
size ratio *R* = *N/P*


log
Tiering
Leveling
sorted
array

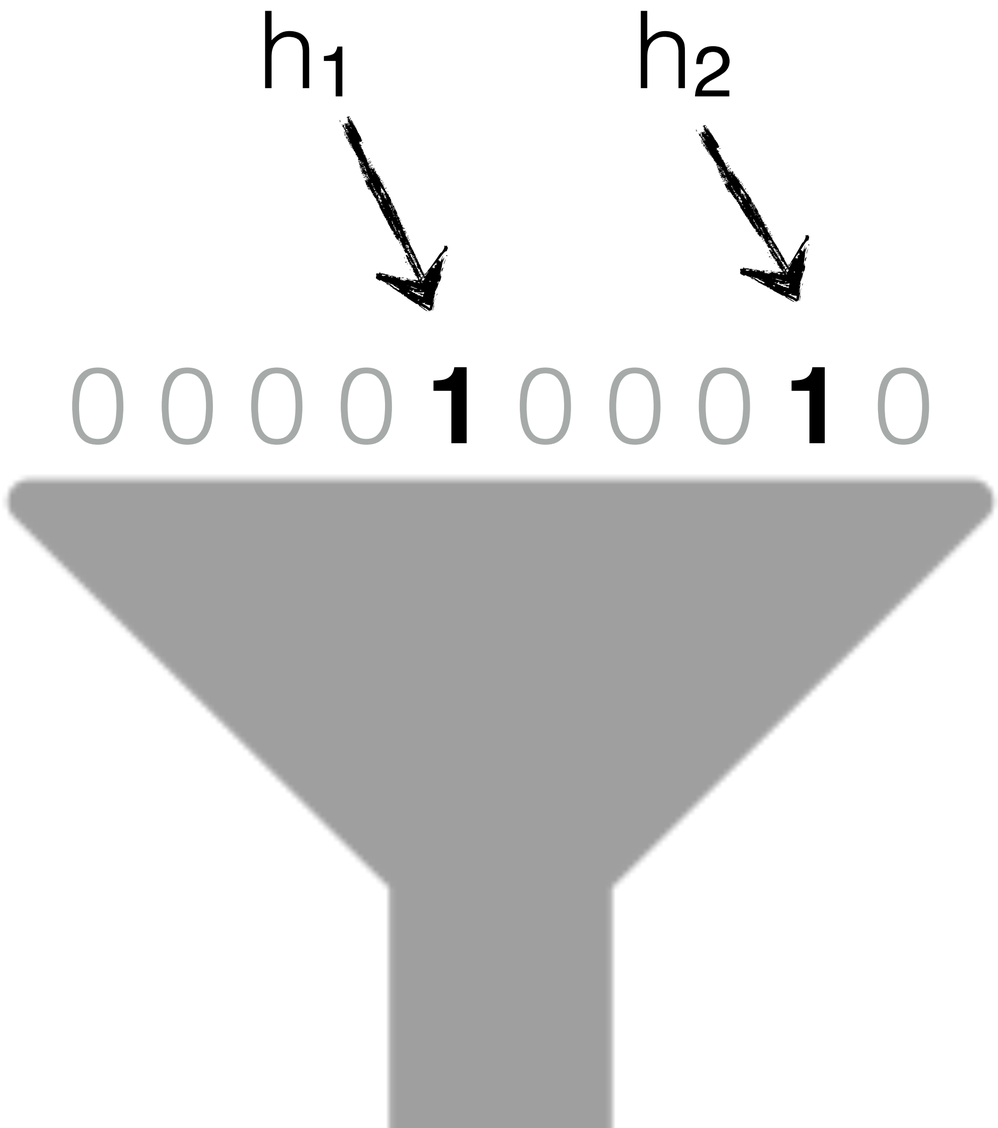

SSD
Bloom
Filters
skip runs

Bloom Filters: Deep Look

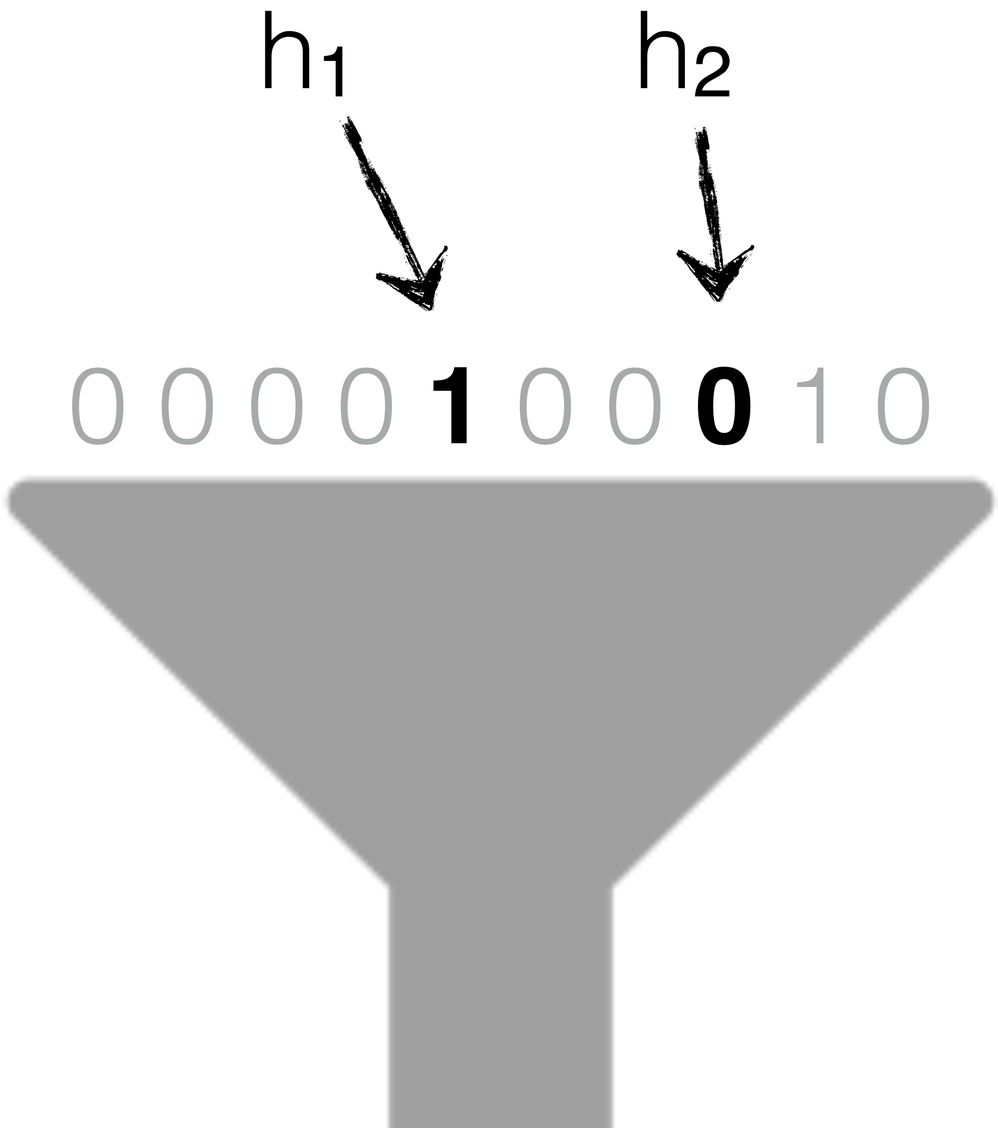
k hash functions



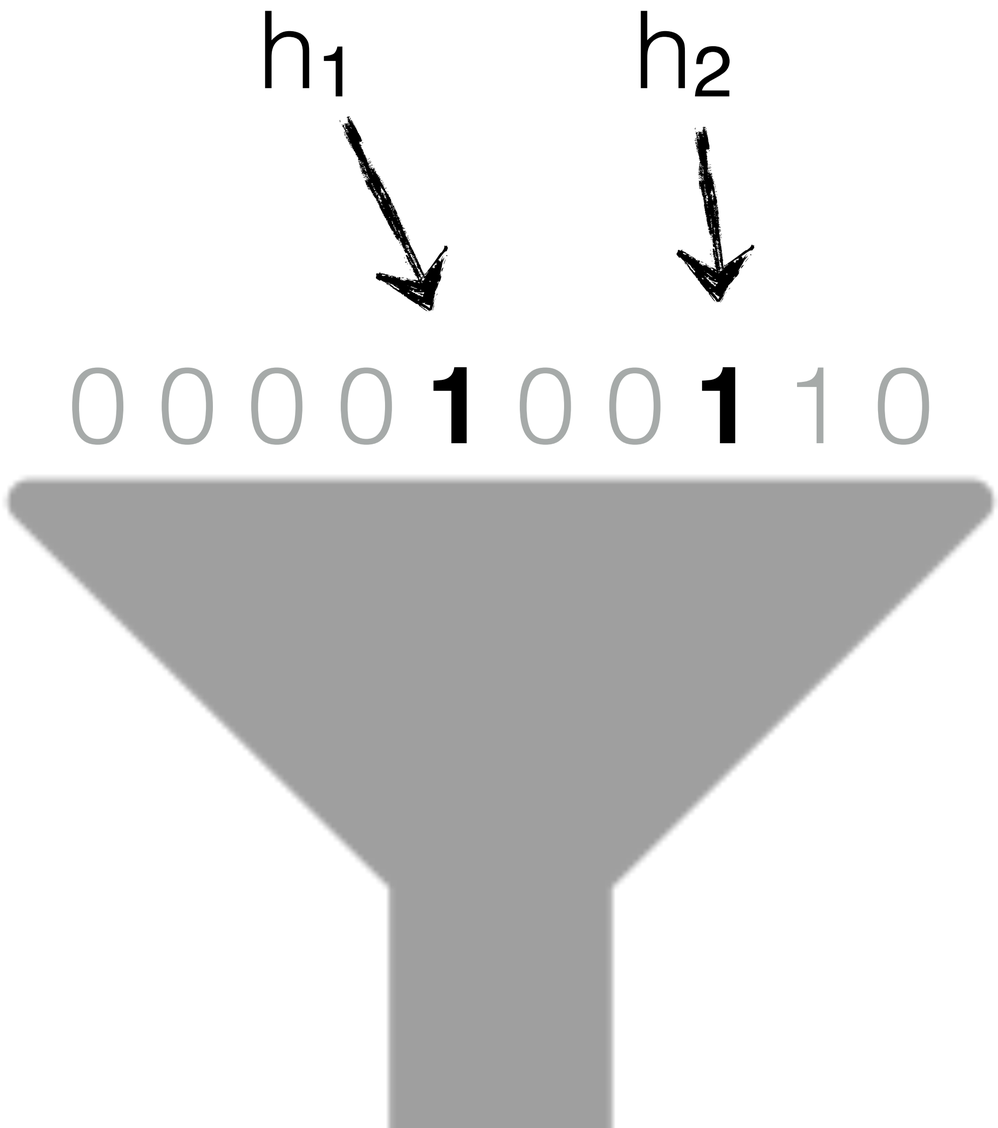
insert: Set from 0 to 1 or keep 1

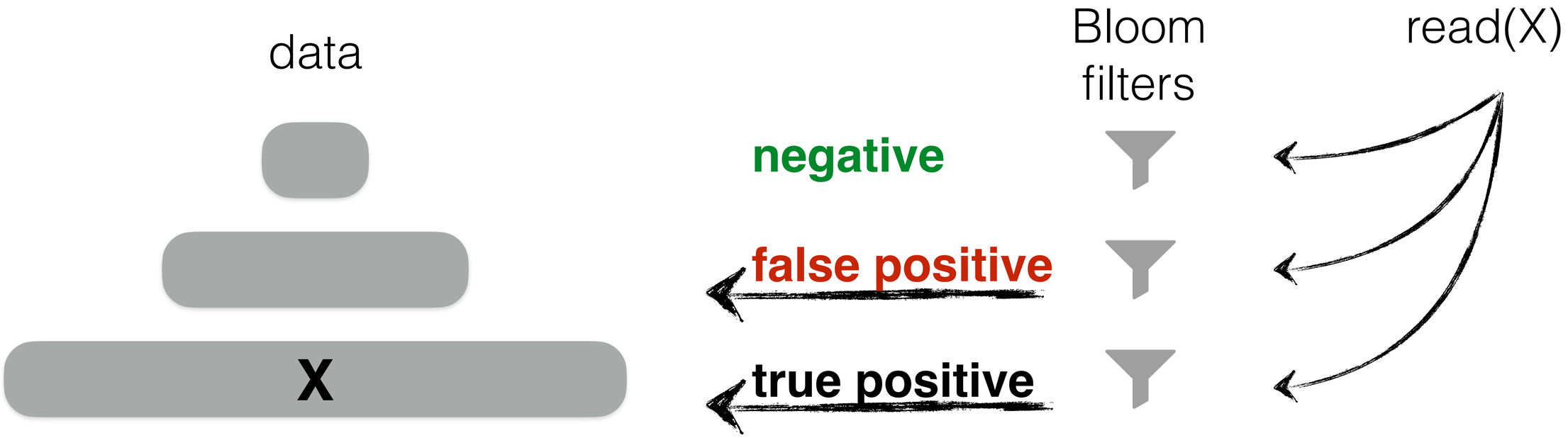


negative lookup: at least one bit is zero

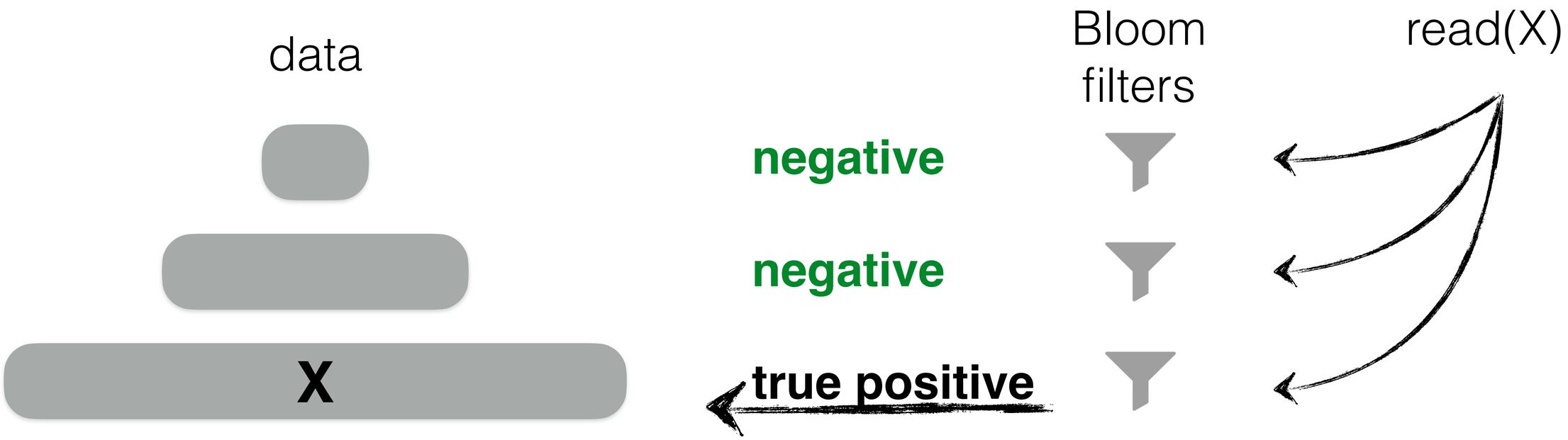


true or false positive lookup

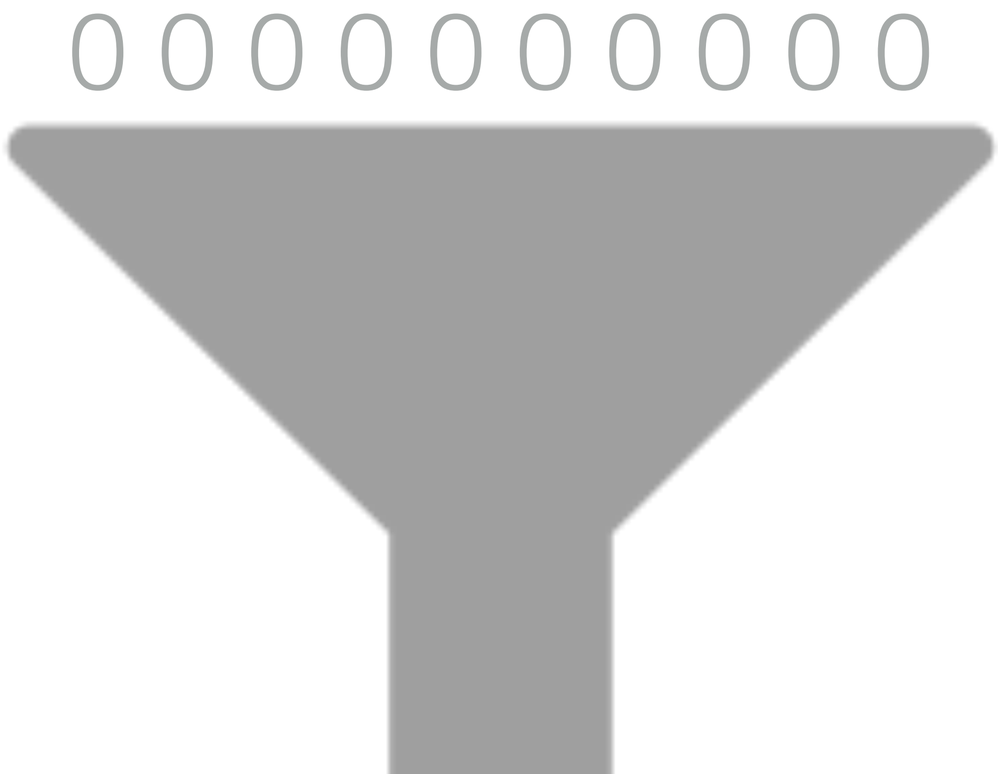



data
X
Bloom
filters
read(X)
negative
false positive
true positive

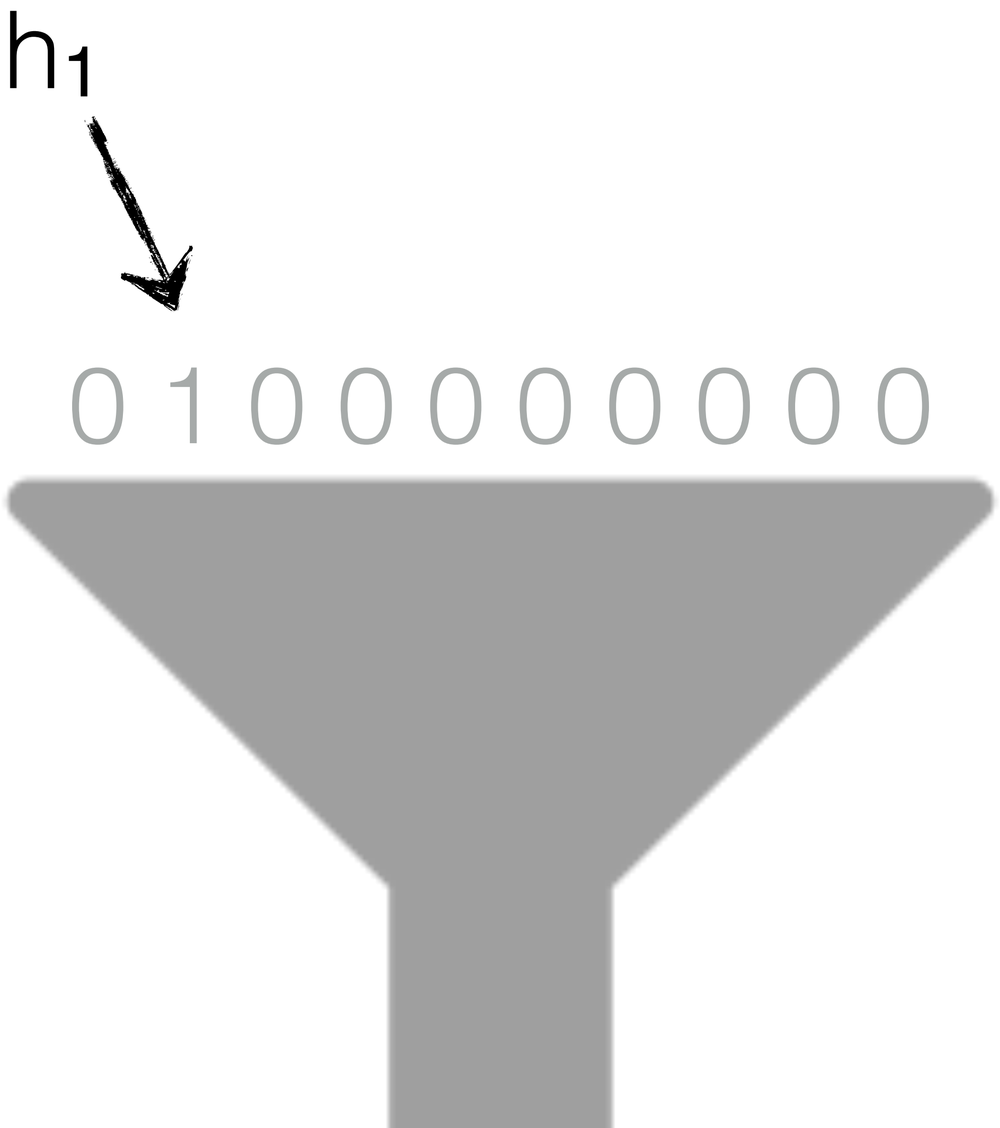
more memory ⟶ fewer false positives



How many hash functions should we use?

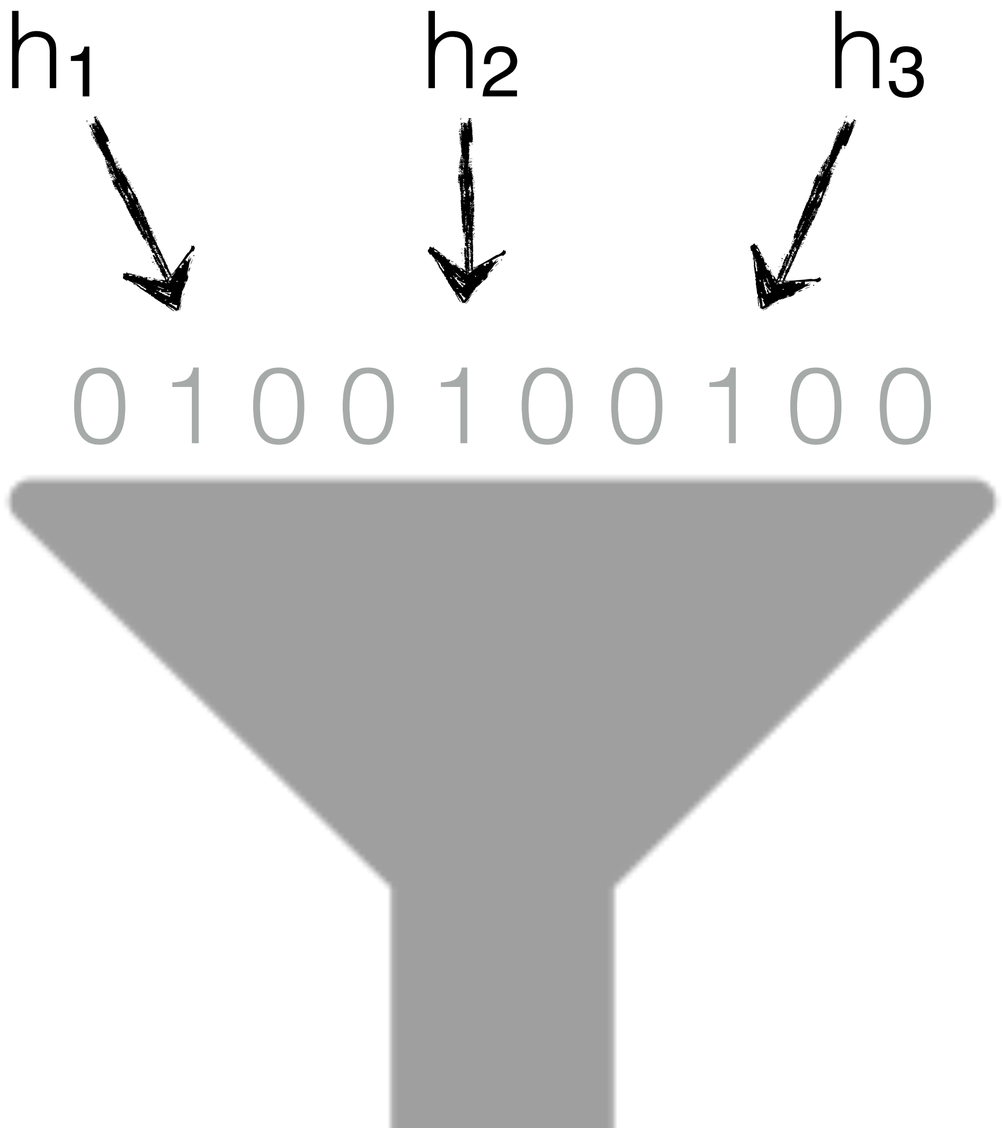


How many hash functions should we use?



One is too few: false positive occurs whenever we hit a 1

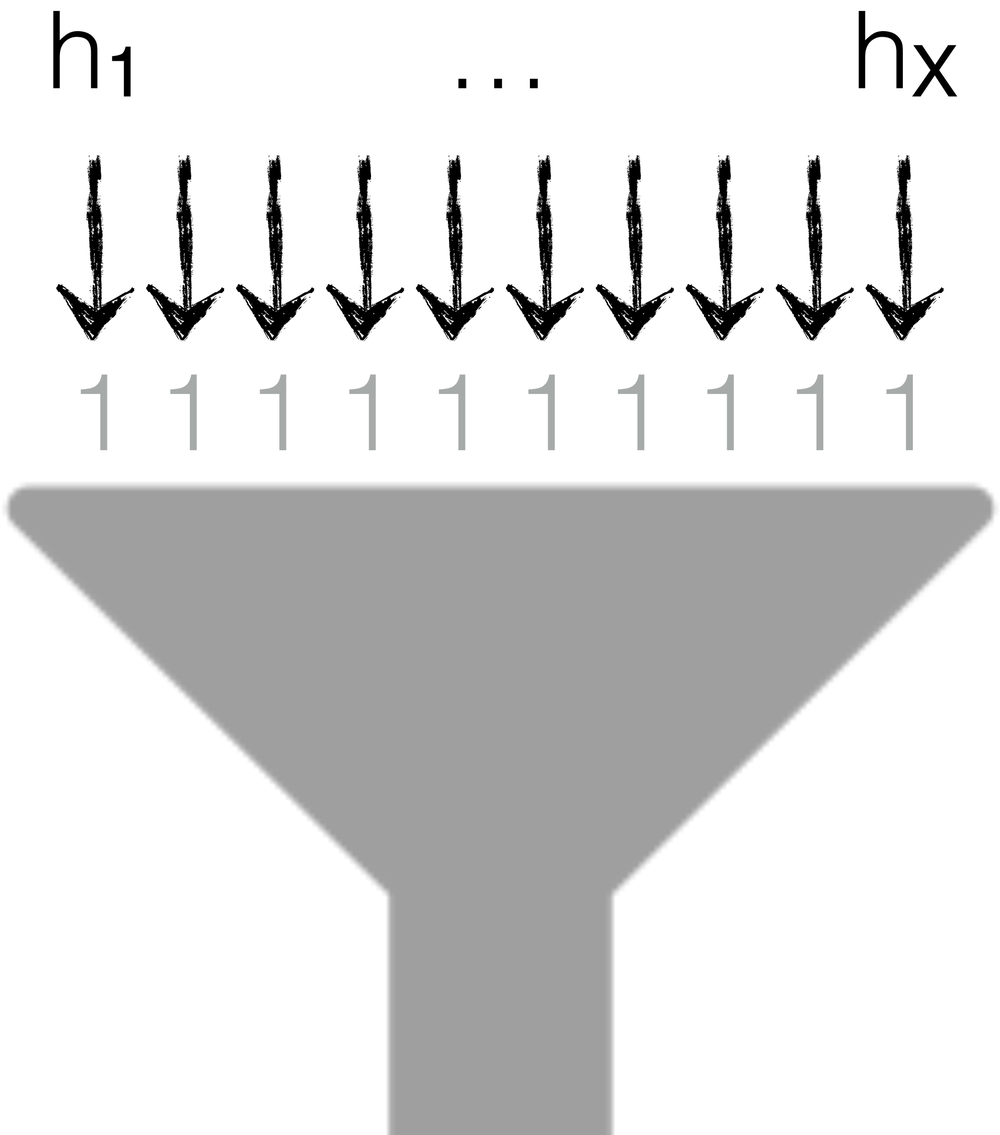
How many hash functions should we use?



One is too few: false positive occurs whenever we hit a 1

By adding hash functions, we initially decrease the false positive rate (FPR).

How many hash functions should we use?

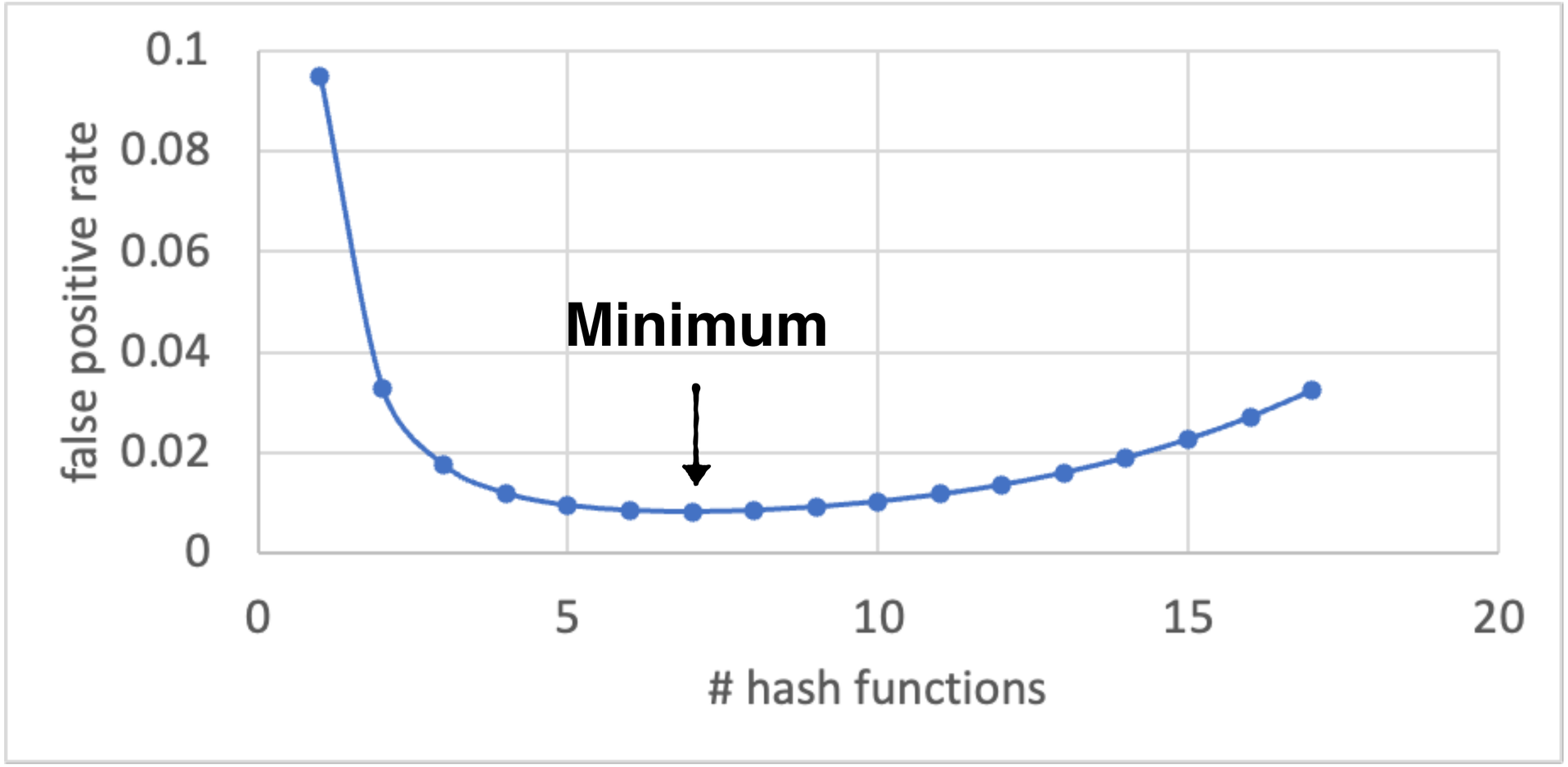


One is too few: false positive occurs whenever we hit a 1

By adding hash functions, we initially decrease the false positive rate (FPR).

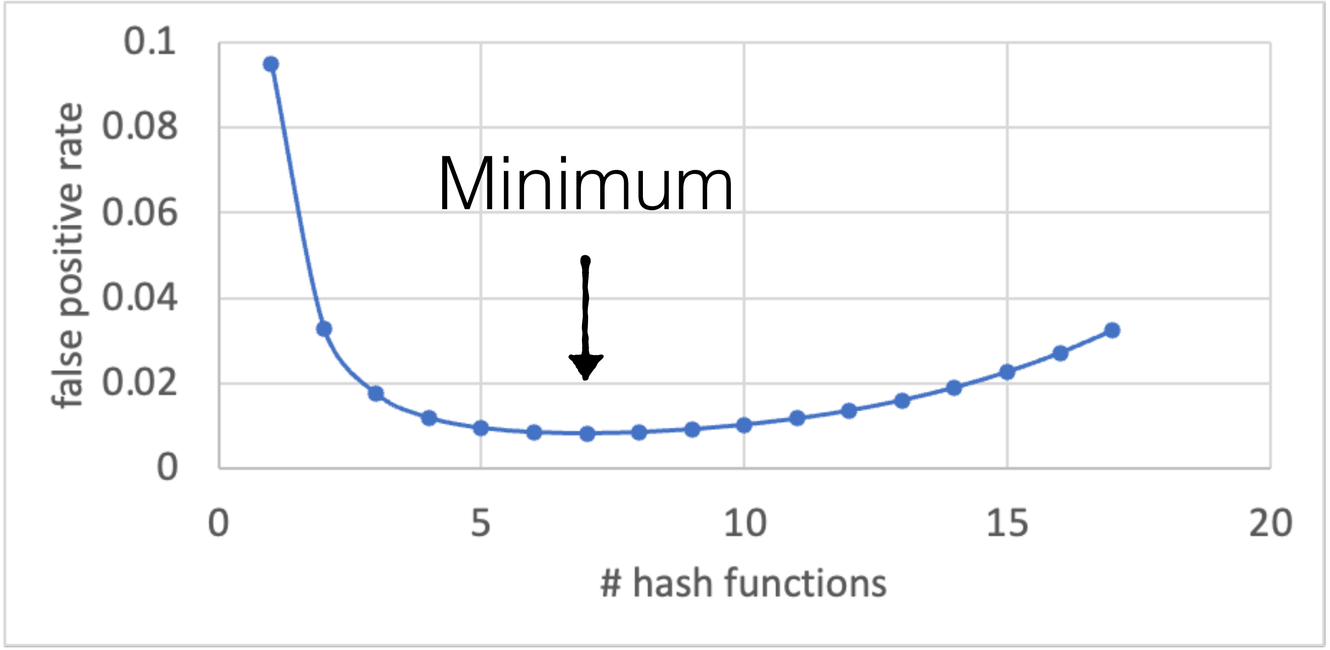
But too many hash functions wind up increasing the FPR.

How many hash functions should we use?



(Drawn for a filter using 10 bits per entry)

How many hash functions should we use?

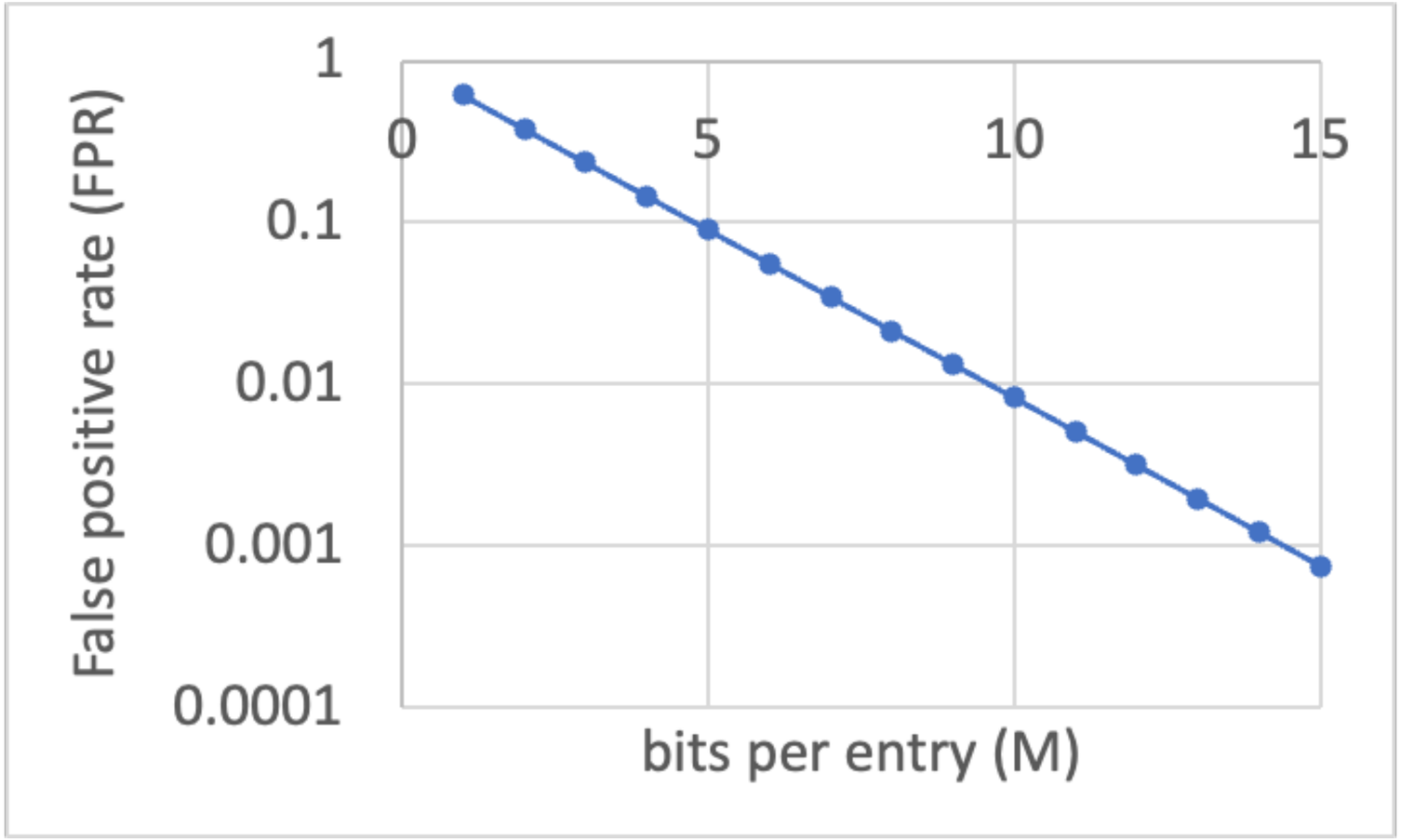


Optimal # hash functions = ln(2) * *M*

(*M* is the number of bits per entry)

assuming the optimal # hash functions,

false positive rate = $e^{-M \cdot \ln(2)^2}$



Operation Costs (in memory accesses)

Insertion =

Positive Query =

Avg. Negative Query =

Operation Costs (in memory accesses)

Insertion = M * ln(2) (# hash functions)

Positive Query =

Avg. Negative Query =

Operation Costs (in memory accesses)

Insertion = M * ln(2)

Positive Query = M * ln(2) (# hash functions)

Avg. Negative Query =

Operation Costs (in memory accesses)

Insertion = $M * \ln(2)$

Positive Query = $M * \ln(2)$

Avg. Negative Query =

(fraction of ones in filter is 0.5 with optimal number of hash functions)

Operation Costs (in memory accesses)

Insertion = $M * \ln(2)$

Positive Query = $M * \ln(2)$

Avg. Negative Query = $1 + 1/2\,(1 + 1/2 * (\ldots))$

(fraction of ones in filter is 0.5 with optimal number of hash functions)

Operation Costs (in memory accesses)

Insertion = $M * \ln(2)$

Positive Query = $M * \ln(2)$

Avg. Negative Query = $1 + 1/2 + 1/4 + \ldots$ **= 2**

(fraction of ones in filter is 0.5 with optimal number of hash functions)

More memory *M*, lower FPR, worst performance

Insertion = $M * \ln(2)$

Positive Query = $M * \ln(2)$

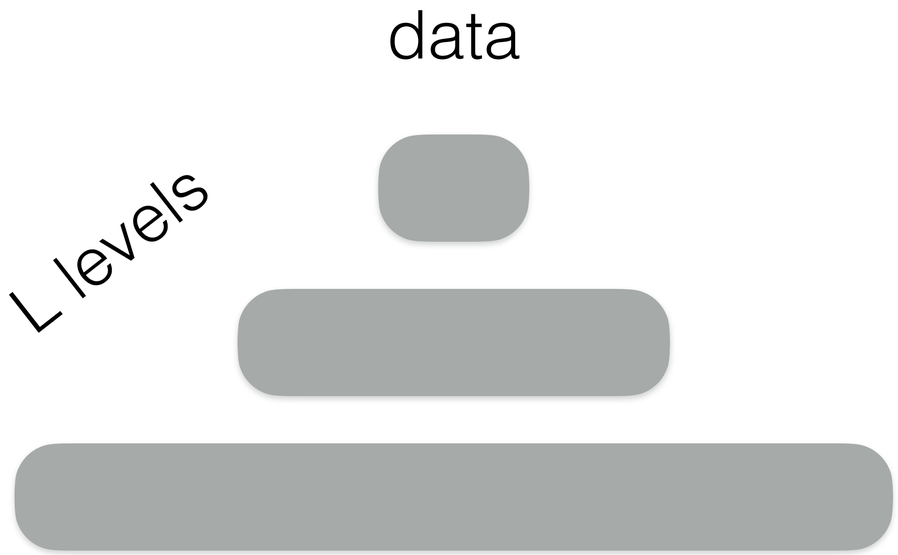
Avg. Negative Query = 2

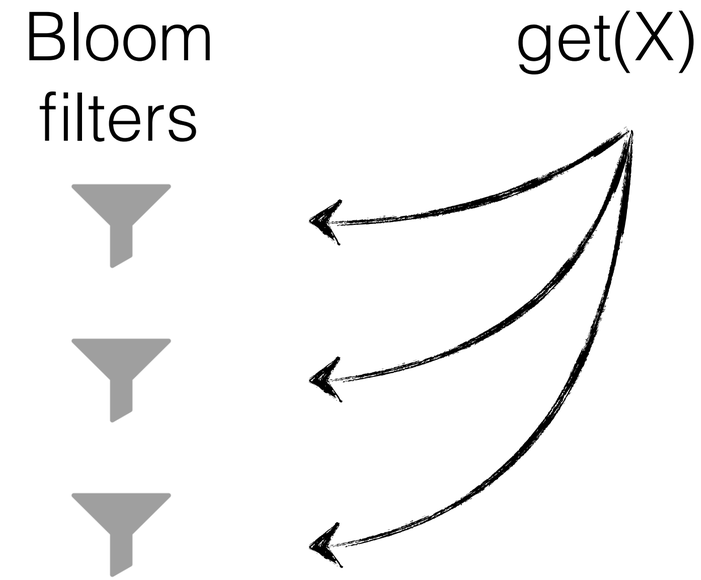
false positive rate = $e^{-M \cdot \ln(2)^2}$

Let's analyze overall filter access cost for basic LSM-tree

Positive Query = $M * \ln(2)$

Avg. Negative Query = 2





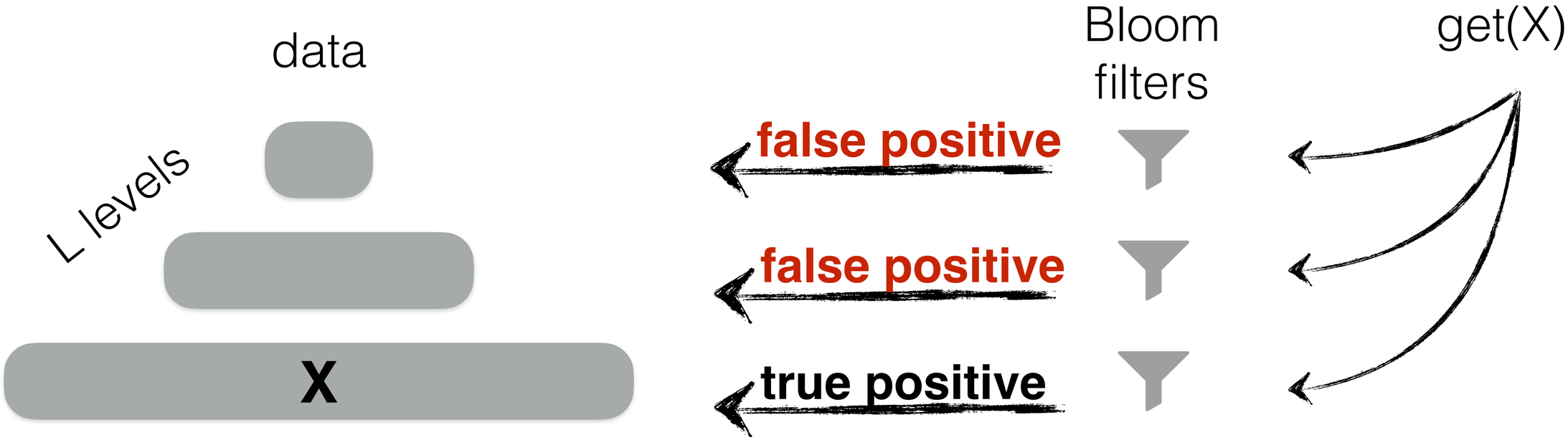
Worst-case:

Avg. worst-case:

Let's analyze overall filter access cost for basic LSM-tree

Positive Query = $M * \ln(2)$

Avg. Negative Query = 2



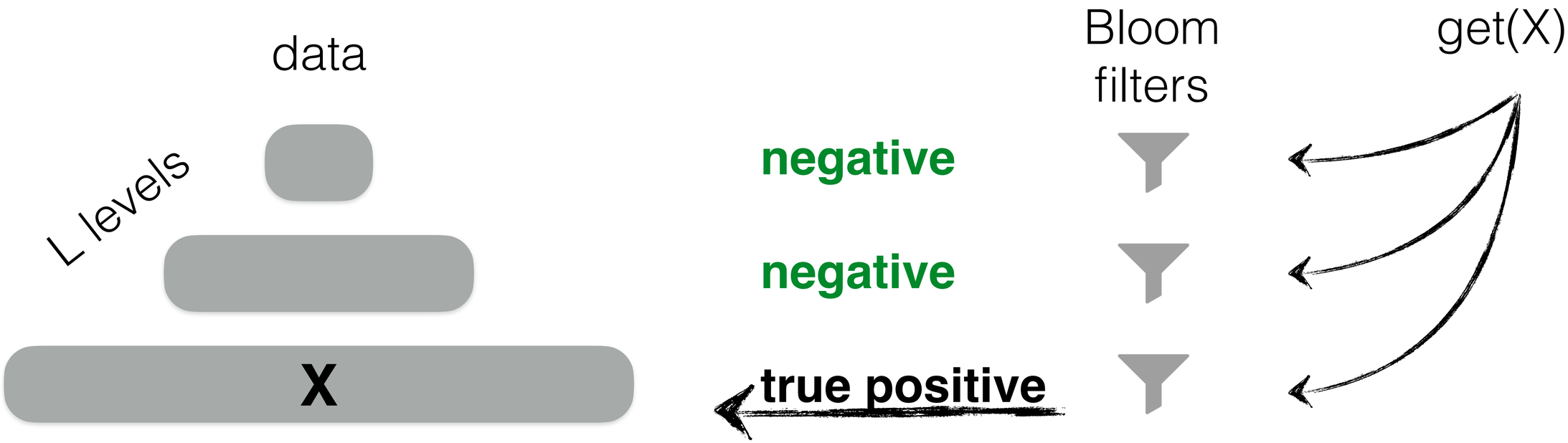
Worst-case: O(M·L)

Avg. worst-case:

Let's analyze overall filter access cost for basic LSM-tree

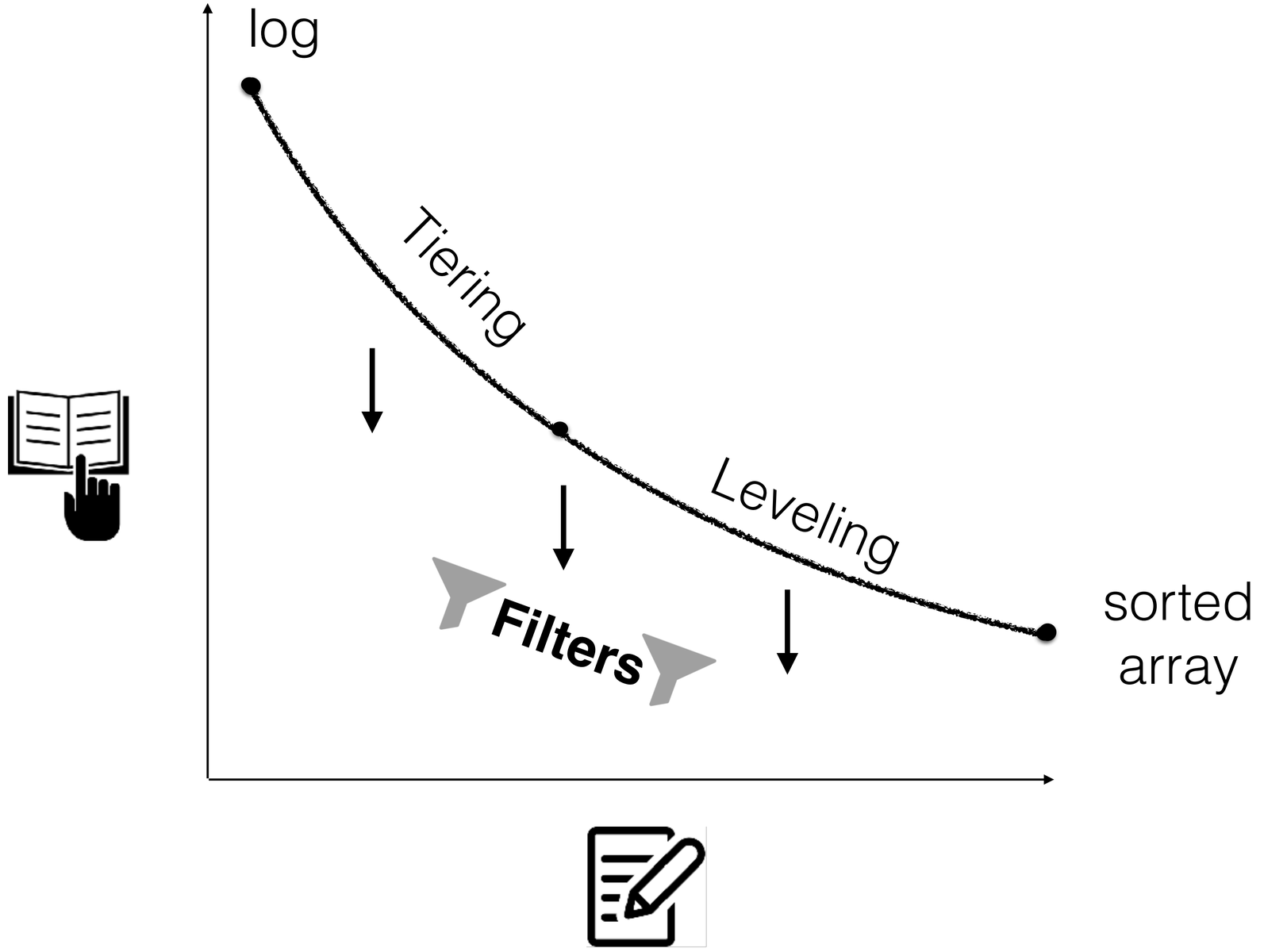
Positive Query = $M * \ln(2)$

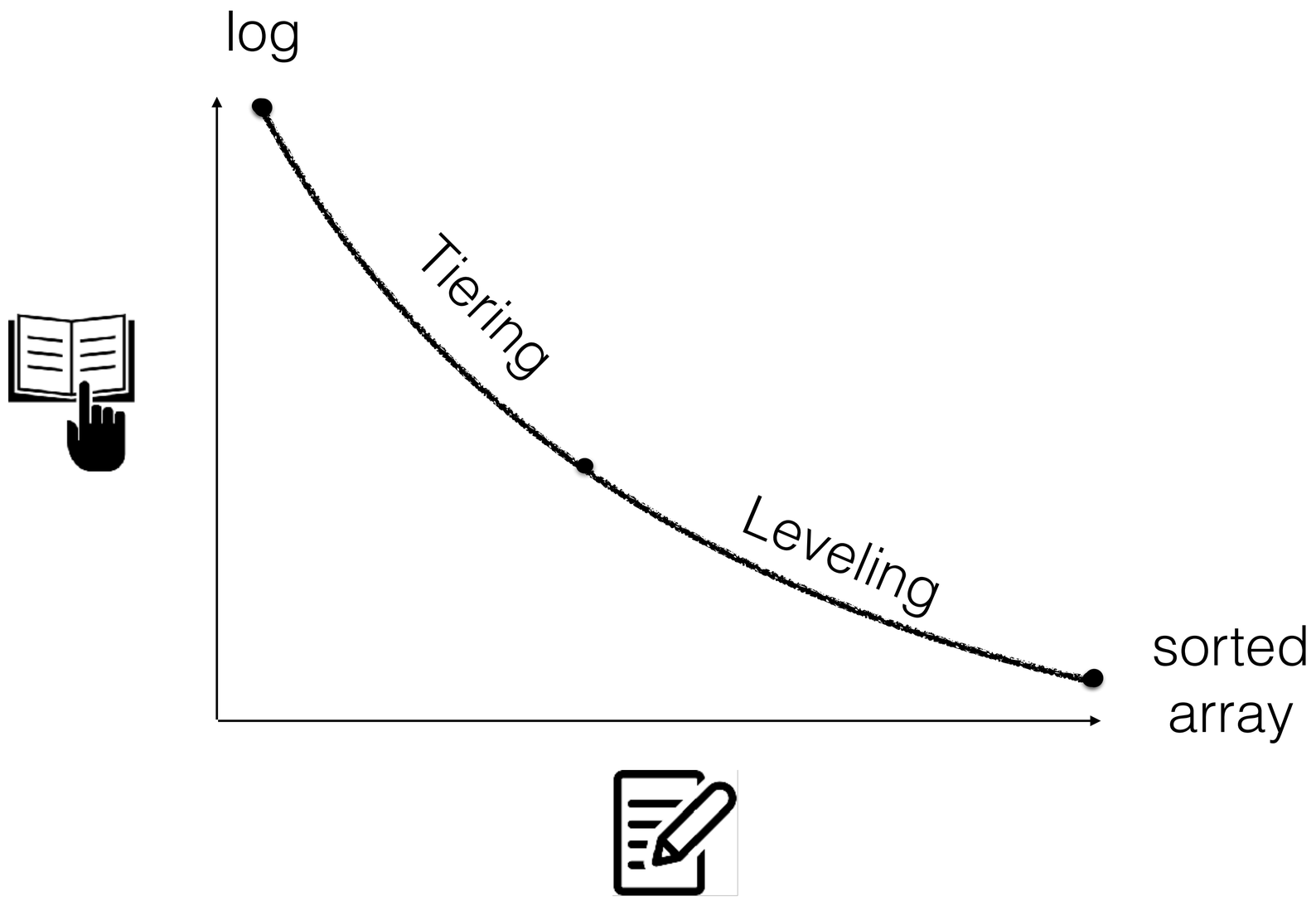
Avg. Negative Query = 2

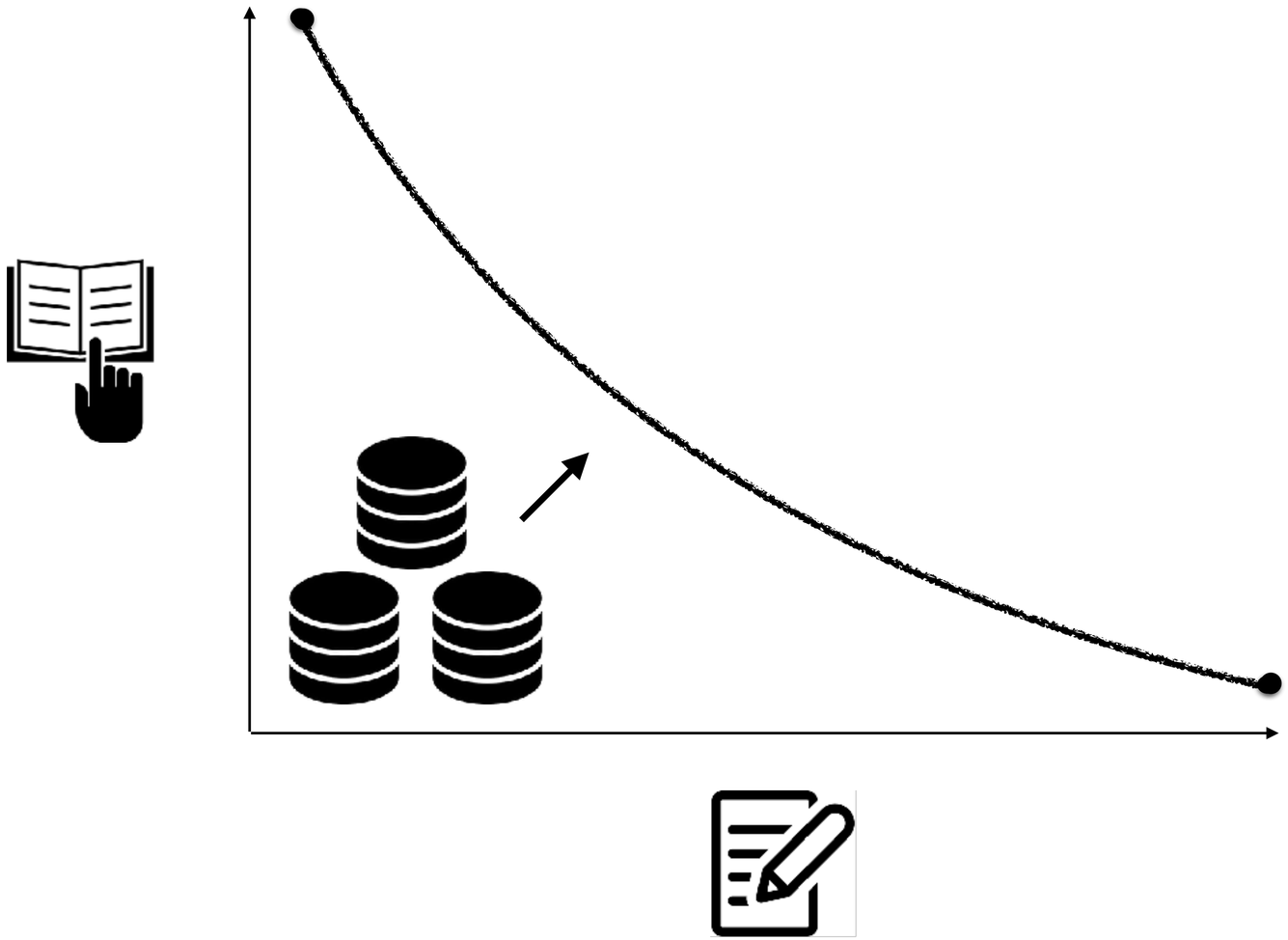


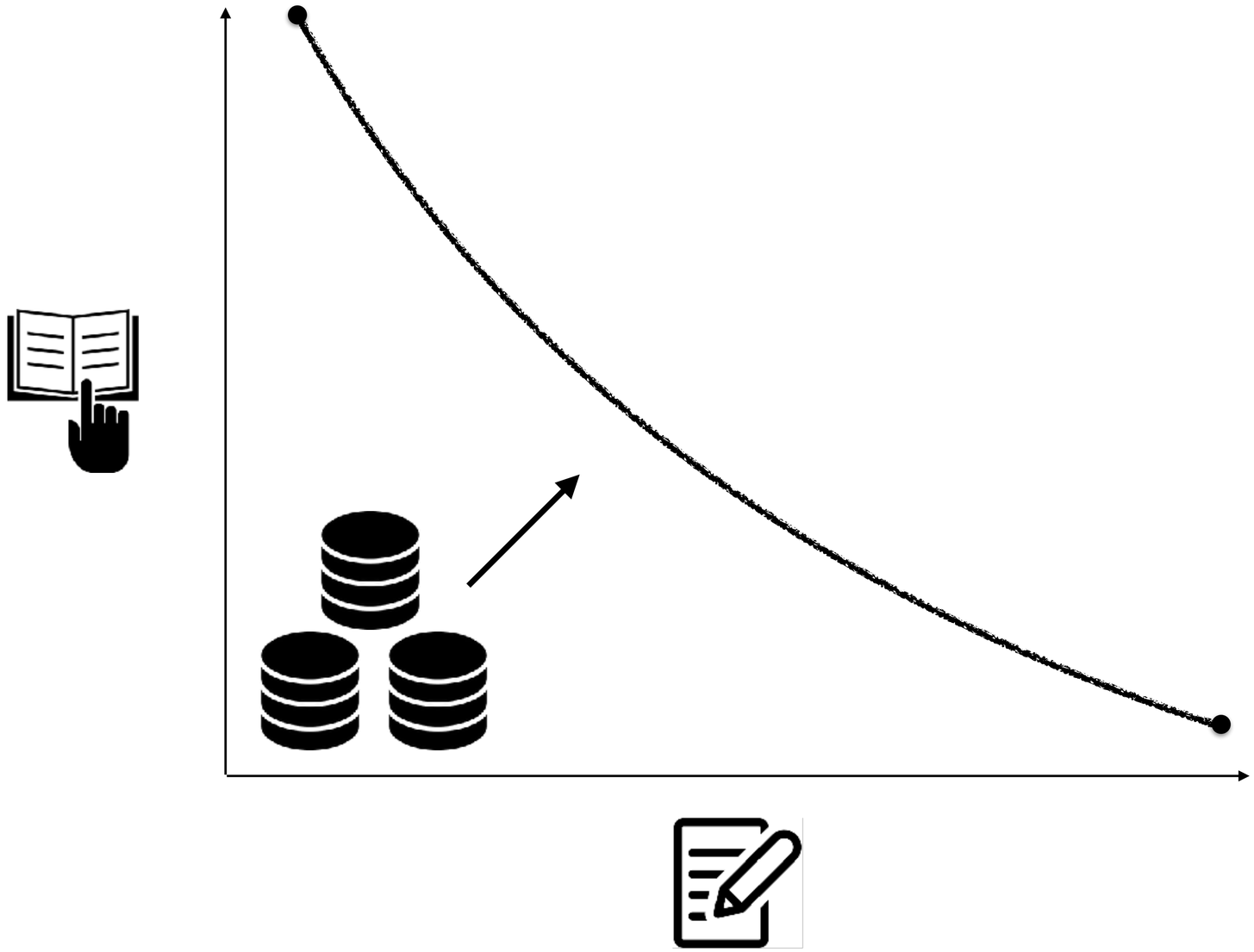
Worst-case: O(M·L)

Avg. worst-case: O(M+L)

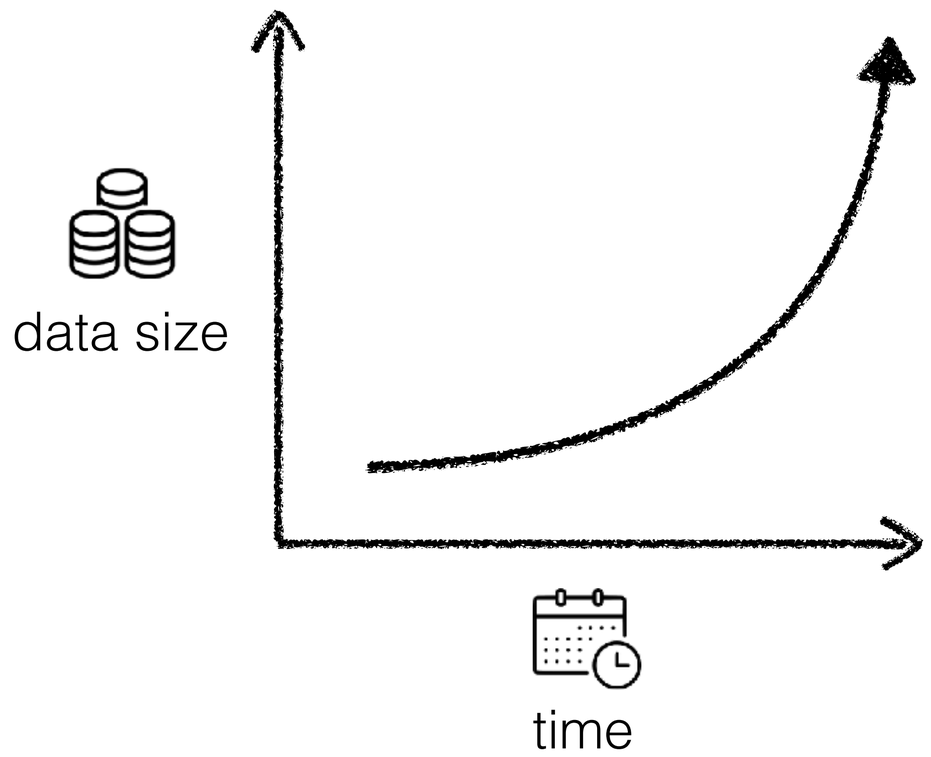

log
Tiering
Leveling
Filters
sorted
array


log
Tiering
Leveling
sorted
array





Can LSM-tree handle exponential data growth?



logarithmic scaling

$$L = O(\log N)$$

logarithmic scaling

$L = O(\log N)$

exponential growth

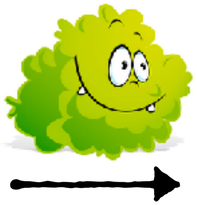
$N \in O(\mathbf{2}^{\mathbf{time}})$

linear scaling

O(**time**)

Can we obtain sub-logarithmic scaling?

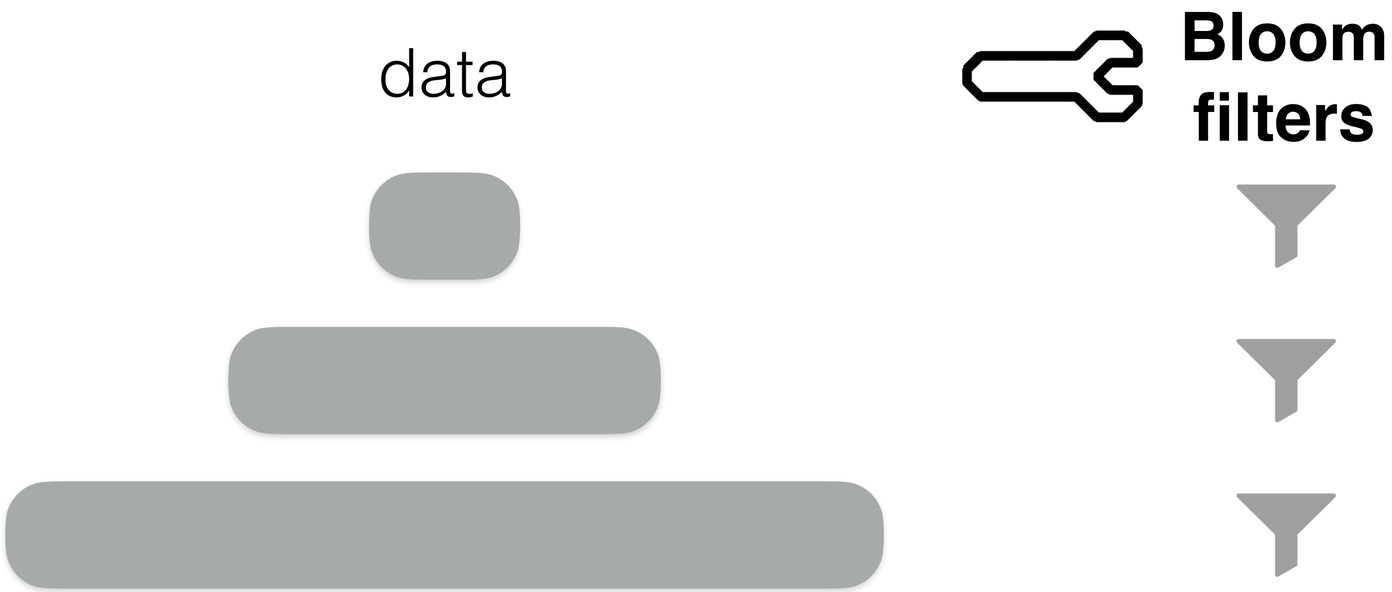
O(log N) → O(log log N) → O(1)

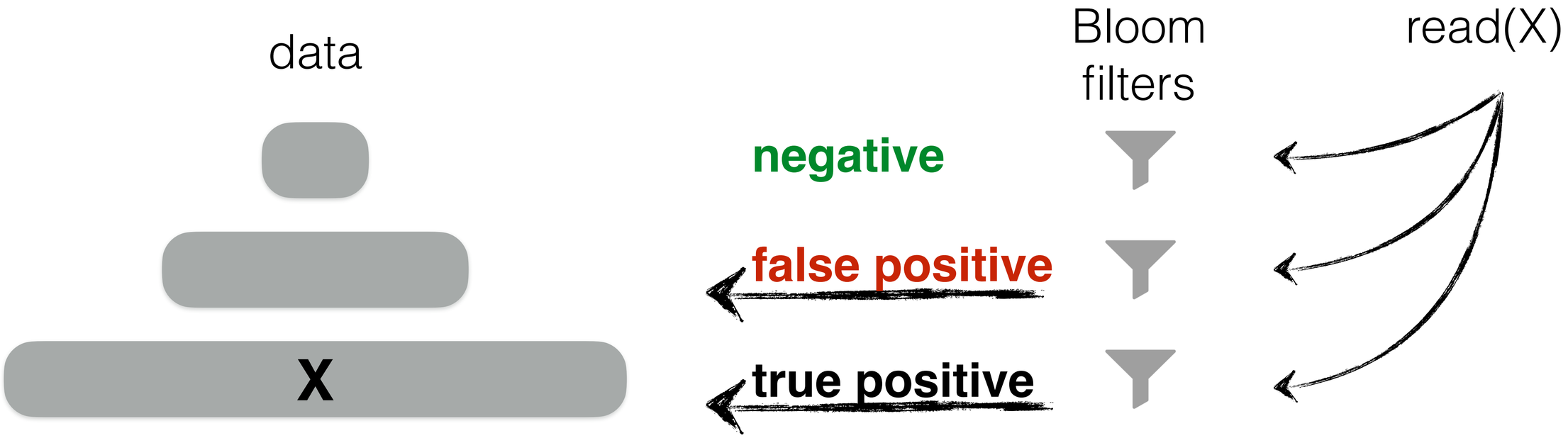
get I/O cost	$O(e^{-M} \cdot L)$		$O(?)$		
insert I/O cost	$O((R \cdot L)/B)$		$O(?)$		$O(?)$
transient space-amp	2x		$O(?)$		
filter mem. reads	$O(M + L)$		$O(?)$		

(Costs assuming leveling)

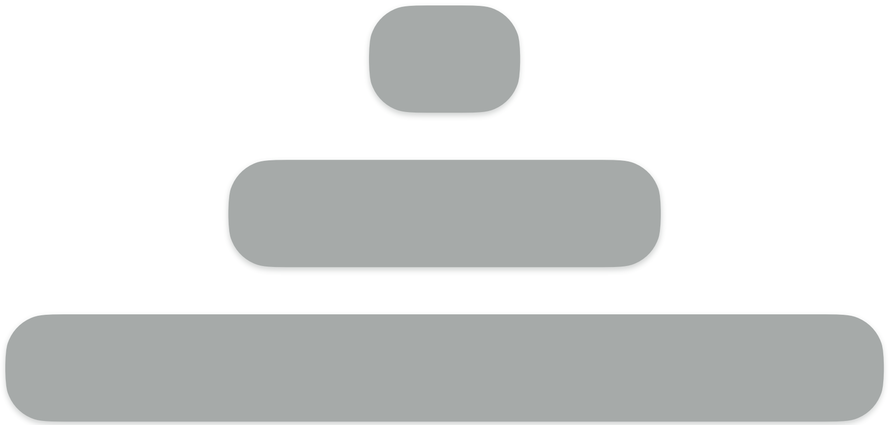
Monkey: **O**ptimal **N**avigable **Key**-Value Store SIGMOD17

Monkey: **O**ptimal **N**avigable **Key**-Value Store SIGMOD17




data
X
Bloom
filters
read(X)
negative
false positive
true positive

data



Bloom filters

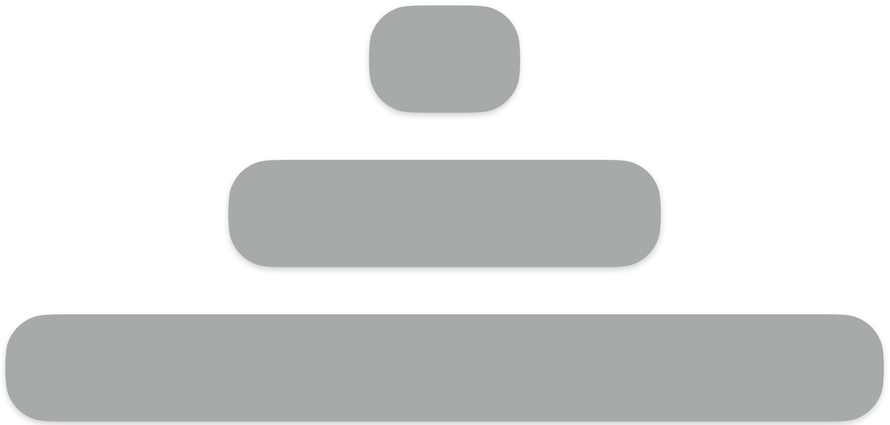
bits/entry

M

M

M

data



Bloom filters

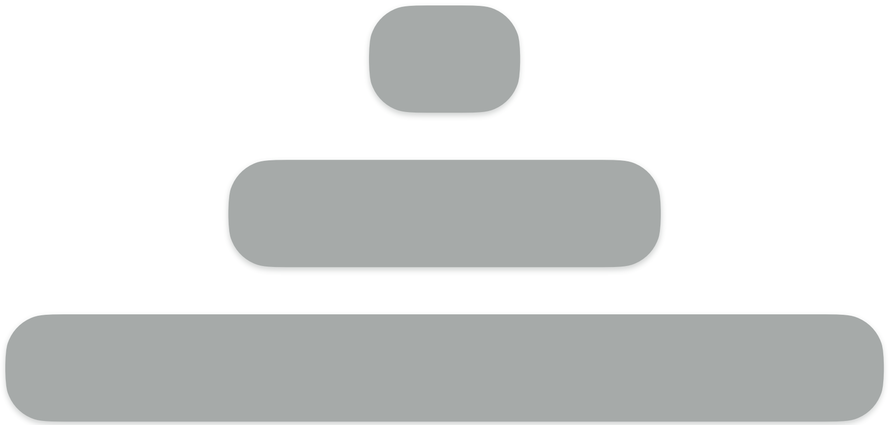
bits/entry

M

M

M

data



Bloom filters

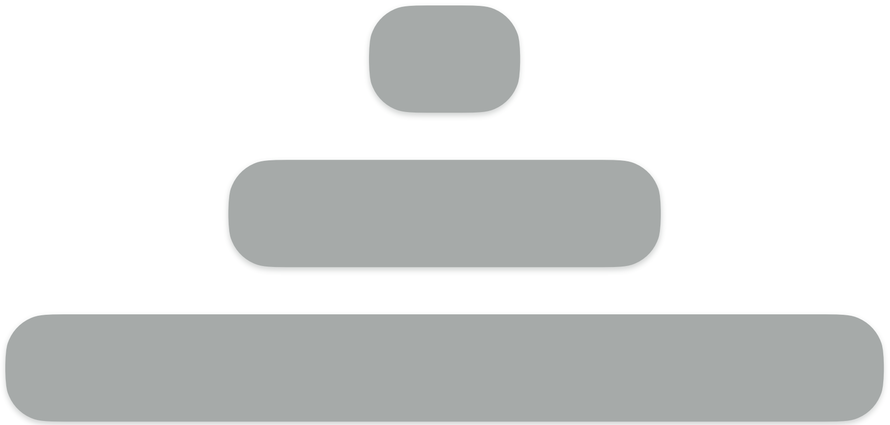
false positive rate

$e^{-M \cdot ln(2)^2}$

$e^{-M \cdot ln(2)^2}$

$e^{-M \cdot ln(2)^2}$

data



Bloom filters

false positive rate

e^{-M}

e^{-M}

e^{-M}

Bloom filters

false positive rate

e^{-M}

e^{-M}

e^{-M}

$= \quad O(\boldsymbol{e^{-M} \cdot \log_R N/P})$

Bloom filters

false positive rate

e^{-M}

e^{-M}

e^{-M}

$= O(e^{-M} \cdot \log_R N/P)$

Bloom filters

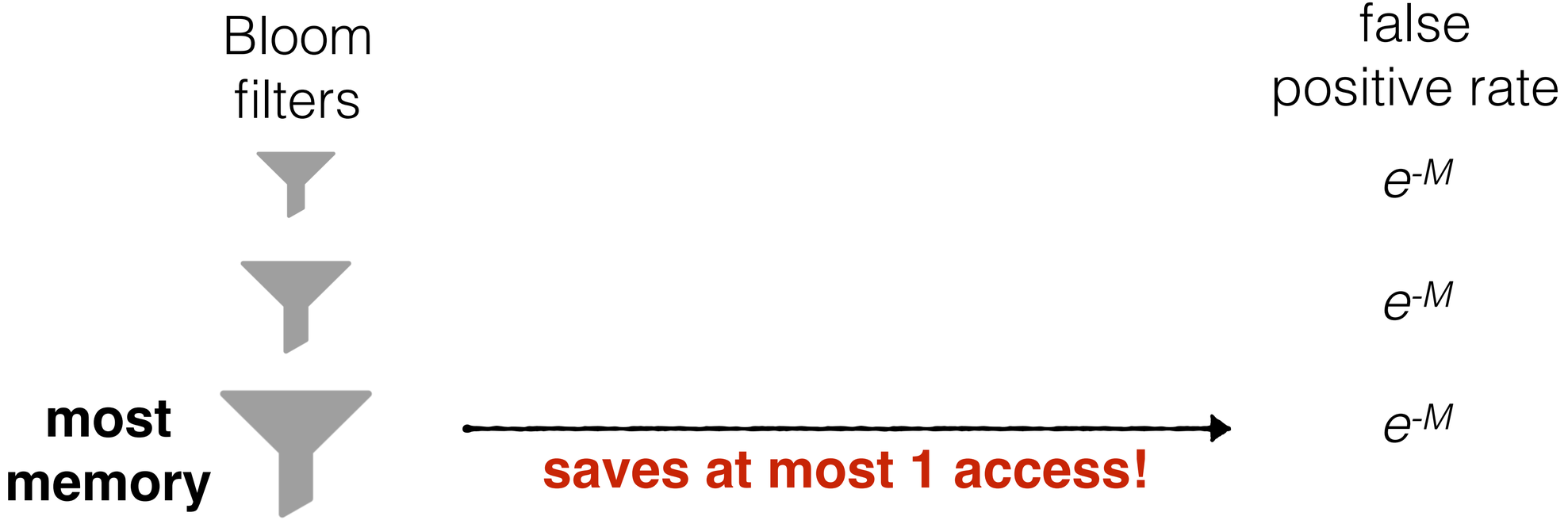
most memory

false positive rate

e^{-M}

e^{-M}

e^{-M}


Bloom
filters
false
positive rate
e^{-M}
e^{-M}
e^{-M}
most
memory
saves at most 1 access!

bits / entry

M ***+ 2***

M ***+ 1***

M ***- 1***

reallocate

false
positive rates

$e^{-(M + 2)}$ ↓

$e^{-(M + 1)}$ ↓

$e^{-(M - 1)}$ ↑

relax

false positive rates

$0 < p_0 < 1$

$0 < p_1 < 1$

$0 < p_2 < 1$

relax

false positive rates

$0 < p_0 < 1$

$0 < p_1 < 1$

$0 < p_2 < 1$

model

read cost $= \sum_{1}^{L} p_i$

memory footprint $= -\sum_{i}^{L} \frac{N}{T^{L-i}} \cdot \frac{\ln(p_i)}{\ln(2)^2}$

DASlab @ Harvard SEAS

relax

model

optimize

false positive rates

$0 < p_0 < 1$

$0 < p_1 < 1$

$0 < p_2 < 1$

read cost $= \sum_{1}^{L} p_i$

memory footprint $= -\sum_{i}^{L} \frac{N}{T^{L-i}} \cdot \frac{\ln(p_i)}{\ln(2)^2}$

in terms of p_0, p_1 …

DASlab @ Harvard SEAS

$\propto \quad e^{-M} / R^{2}$

$\propto \quad e^{-M} / R^{1}$

$\propto \quad e^{-M} / R^{0}$

e^{-M} / R^{2}

e^{-M} / R^{1}

e^{-M} / R^{0}

geometric progression

$= \quad O(e^{-M})$

Faster worst case

$$O(e^{-M}) \quad < \quad O(e^{-M} \log_R N/P)$$

Configuration

buffer 2MB

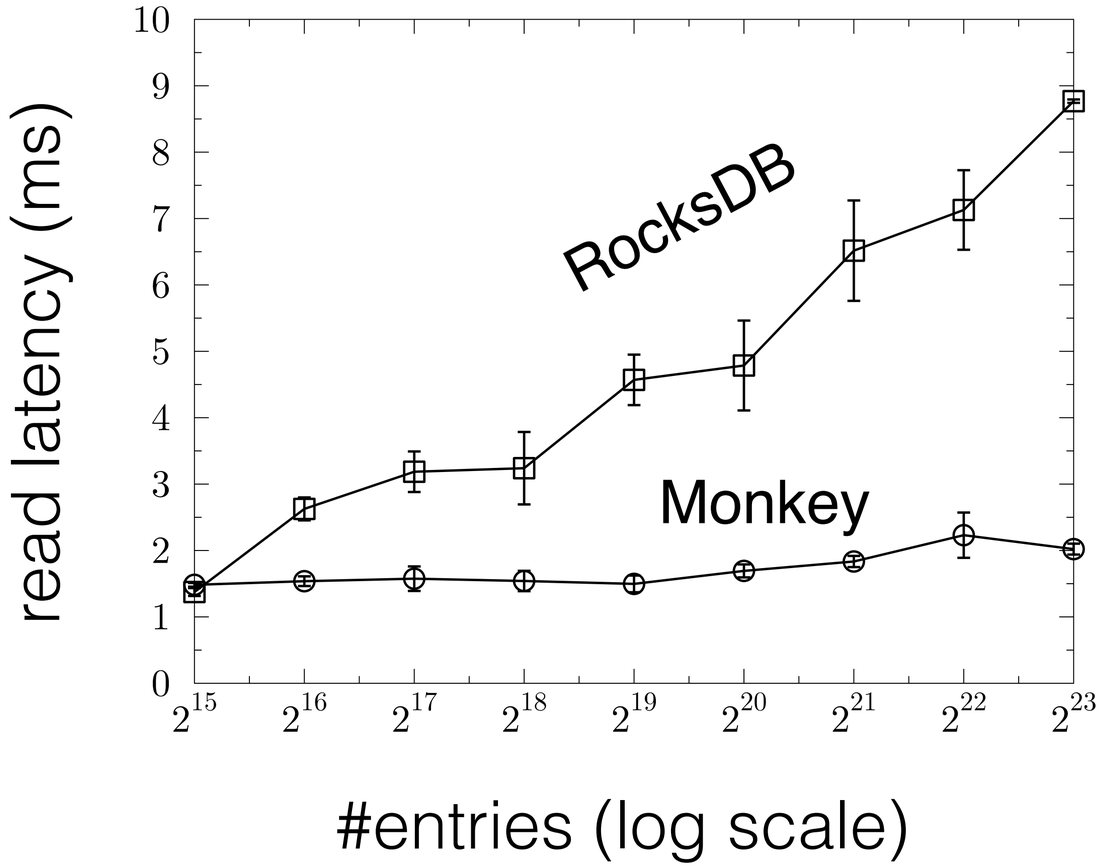
bits/entry: 5

size ratio: 2

1KB entries

queries to missing keys

hard disk storage



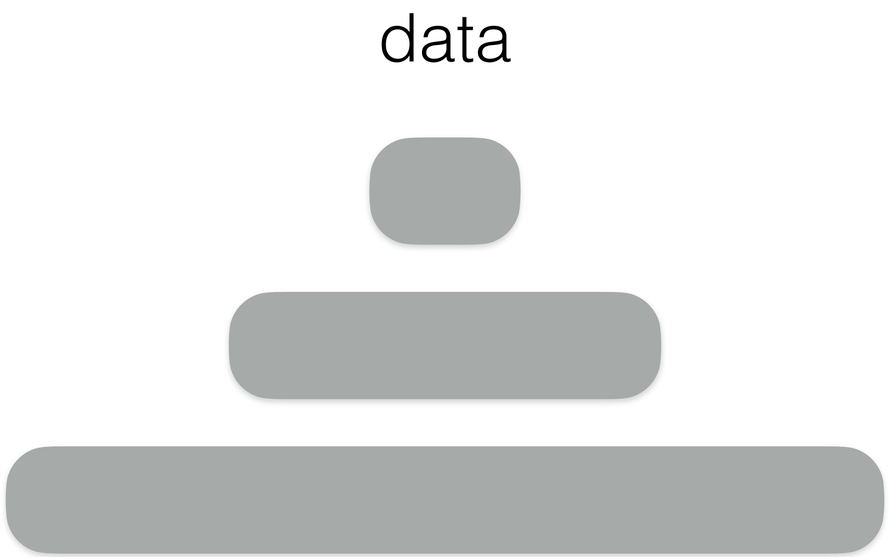
$O(e^{-M} \log_R N/P)$

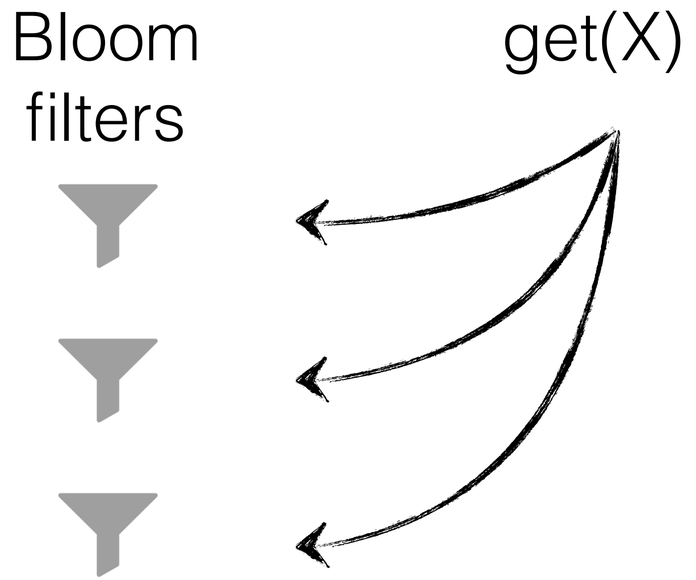
$O(e^{-M})$

Analyze filter access with Monkey

Positive Query = $M * \ln(2)$

Avg. Negative Query = 2





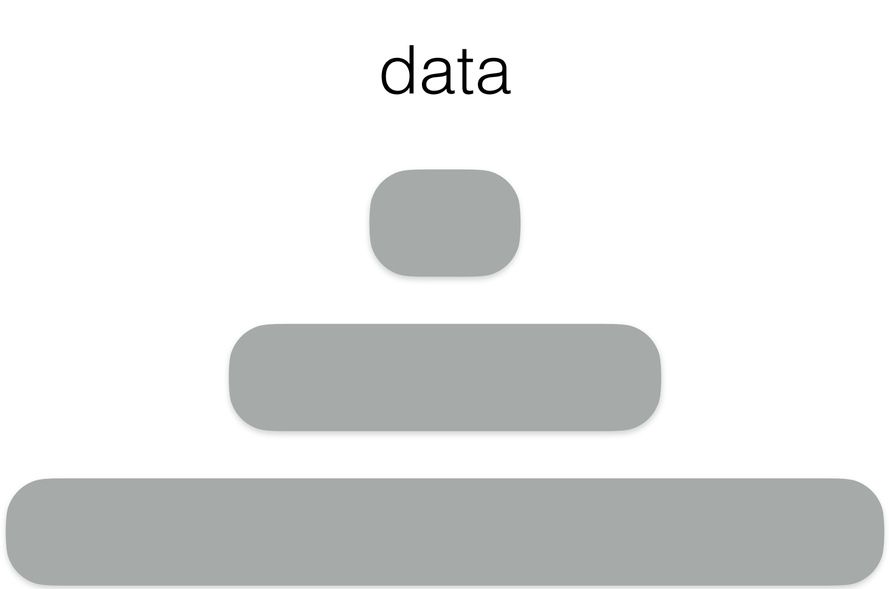
Worst-case:

Avg. worst-case:

Analyze filter access with Monkey

Positive Query = $M * \ln(2)$

Avg. Negative Query = 2

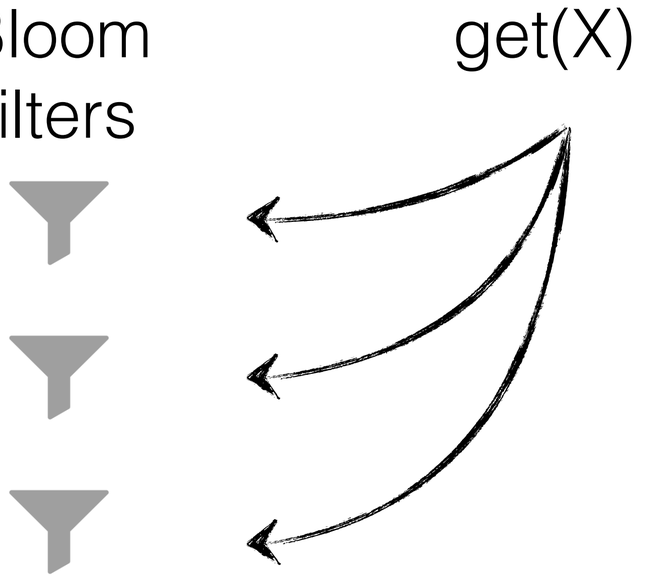


 # hash functions

O(*M+2*)

O(*M+1*)

O(*M*)



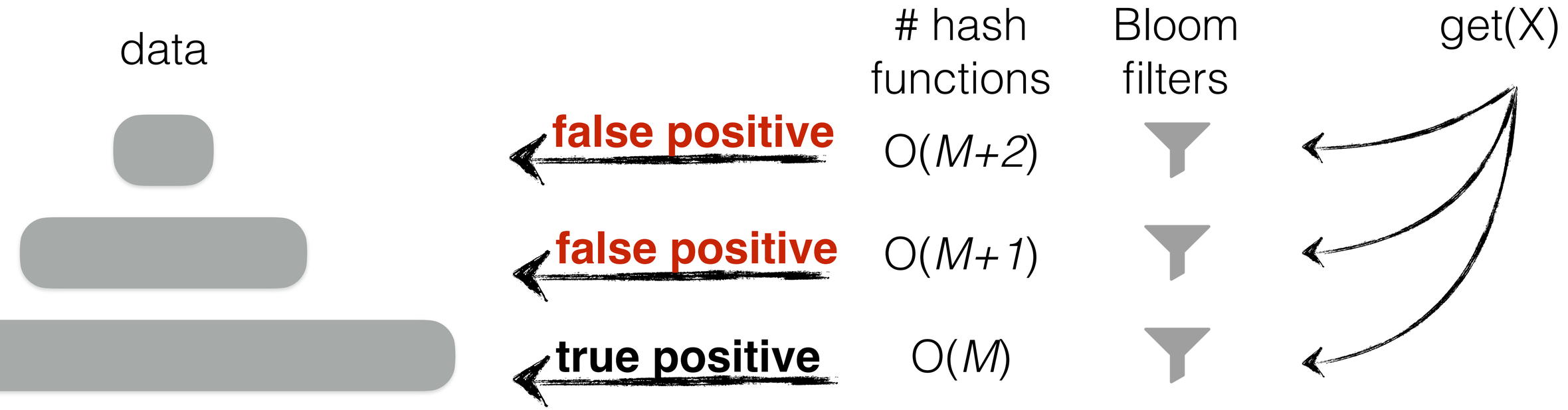
Worst-case:

Avg. worst-case:

Analyze filter access with Monkey

Positive Query = $M * \ln(2)$

Avg. Negative Query = 2



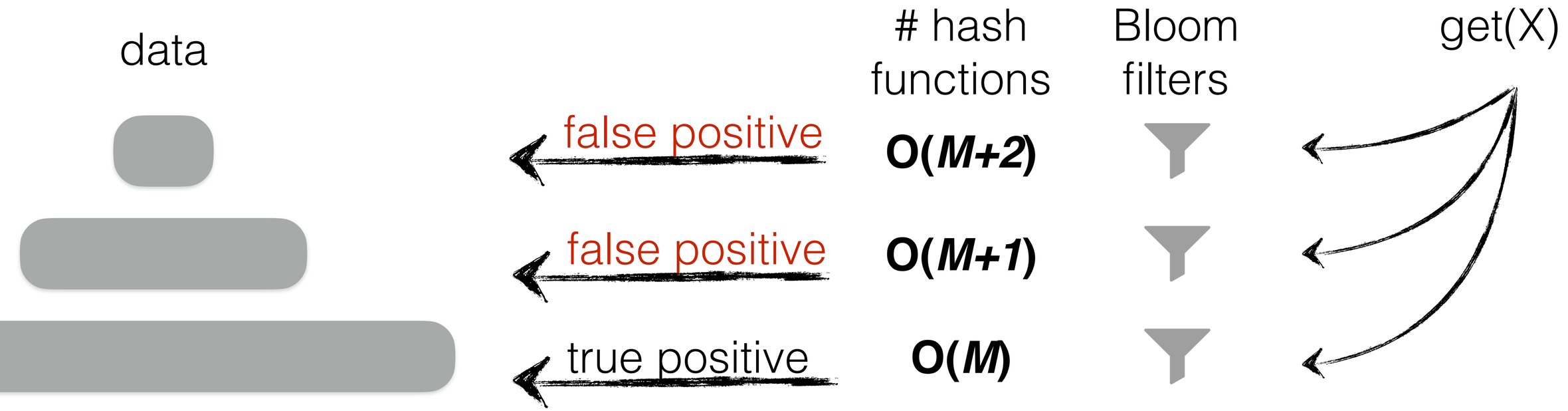
Worst-case:

Avg. worst-case:

Analyze filter access with Monkey

Positive Query = $M * \ln(2)$

Avg. Negative Query = 2



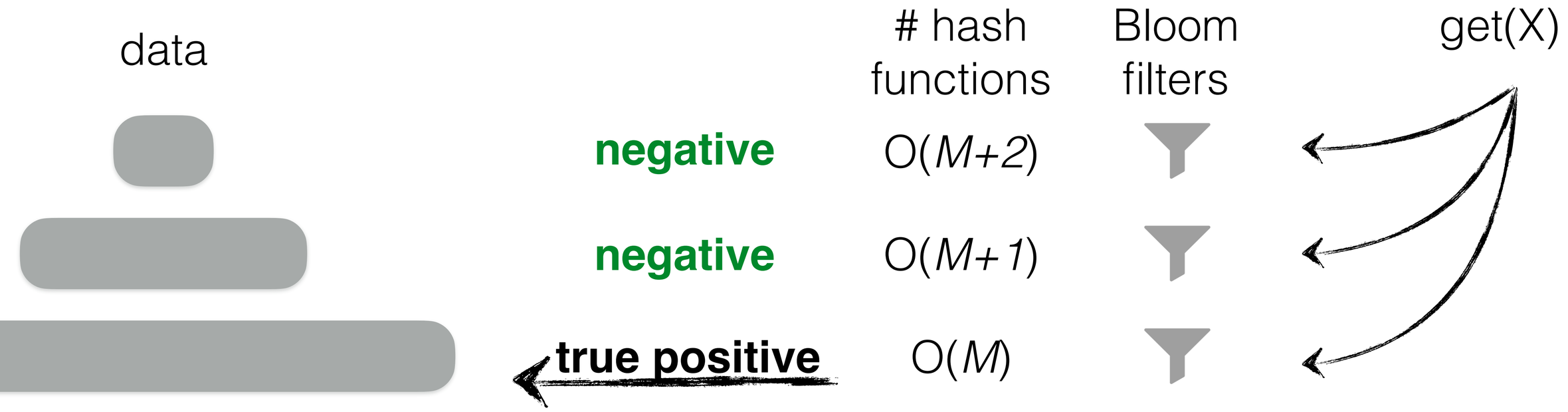
Worst-case: $O(M \cdot L + L^2)$

Avg. worst-case:

Analyze filter access with Monkey

Positive Query = $M * \ln(2)$

Avg. Negative Query = 2



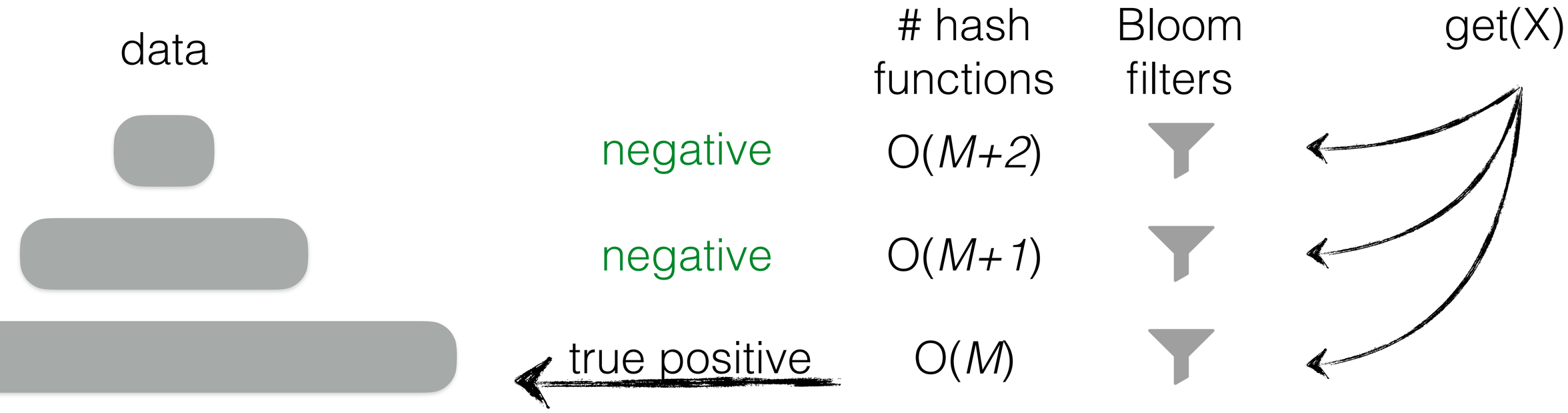
Worst-case: $O(M \cdot L + L^2)$

Avg. worst-case:

Analyze filter access with Monkey

Positive Query = $M * \ln(2)$

Avg. Negative Query = 2



Worst-case: $O(M \cdot L + L^2)$

Avg. worst-case: O(M+L)

get I/O cost	$O(e^{-M} \cdot L)$		$O(e^{-M})$		
insert I/O cost	$O((R \cdot L)/B)$		$O(?)$		$O(?)$
transient space-amp	2x		$O(?)$		
filter mem. reads	$O(M + L)$		$O(?)$		

$L = \log_R N/P$

(Costs assuming leveling)

Break :)

get I/O cost	$O(e^{-M} \cdot L)$		$O(e^{-M})$		
insert I/O cost	$O((R \cdot L)/B)$		$O(?)$		$O(?)$
transient space-amp	2x		$O(?)$		
filter mem. reads	$O(M + L)$		$O(?)$		

$L = \log_R N/P$

(Costs assuming leveling)

Dostoevsky

SIGMOD18

Dostoevsky: **S**pace-**T**ime **O**ptimized **Ev**olvable **S**calable **K**e**y**-Value Store

reads & writes cost breakdown

reads

false positive rates

exponentially decreasing

e^{-M/R^2}

$e^{-M/R}$

e^{-M}

false positive rates

e^{-M/R^2}

$e^{-M/R}$

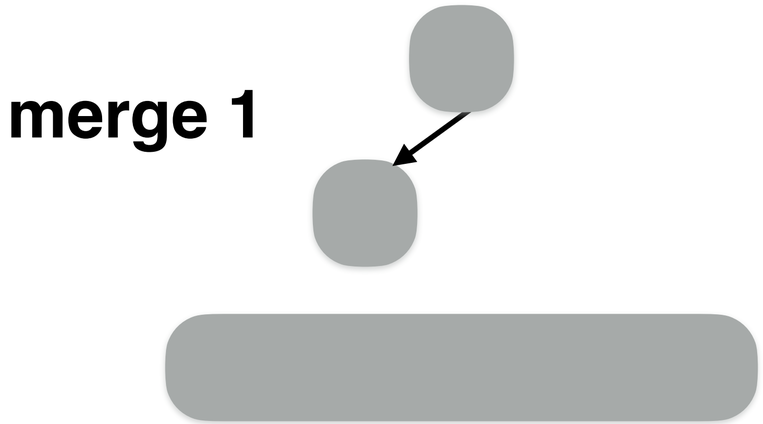
reads ⟶ e^{-M}

reads

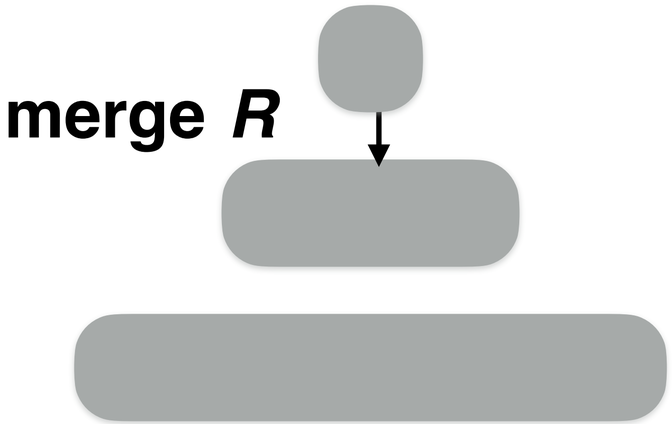
writes

$O(e^{-M})$

writes



writes



writes

O(***R/B***)

O(***R/B***)

O(***R/B***)

O((***R*** ·L***)/B***)

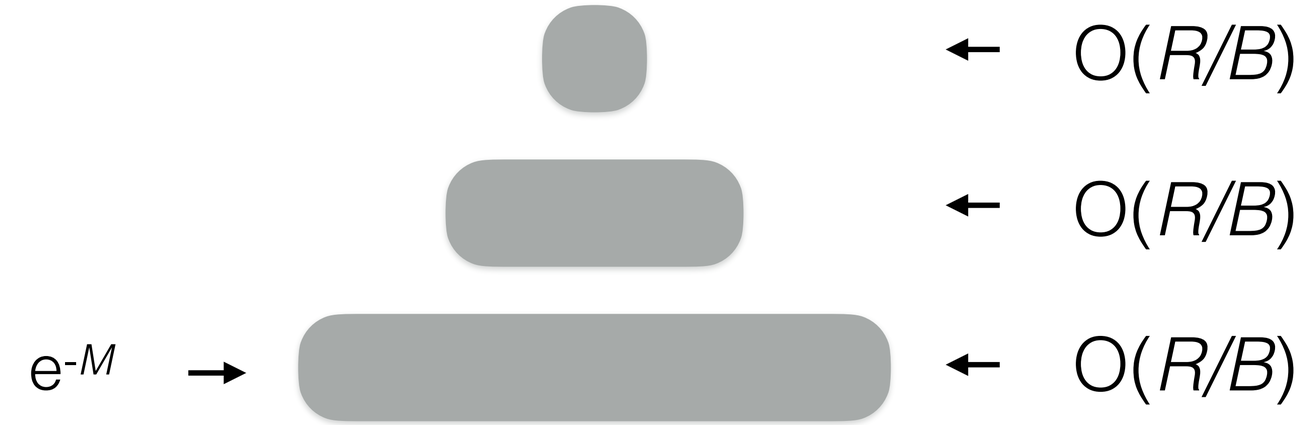
reads		writes
$O(e^{-M})$		$O((R \cdot L)/B)$
=		=
e^{-M}/R^2		$O(R/B)$
+		+
e^{-M}/R		$O(R/B)$
+		+
e^{-M}		$O(R/B)$

reads

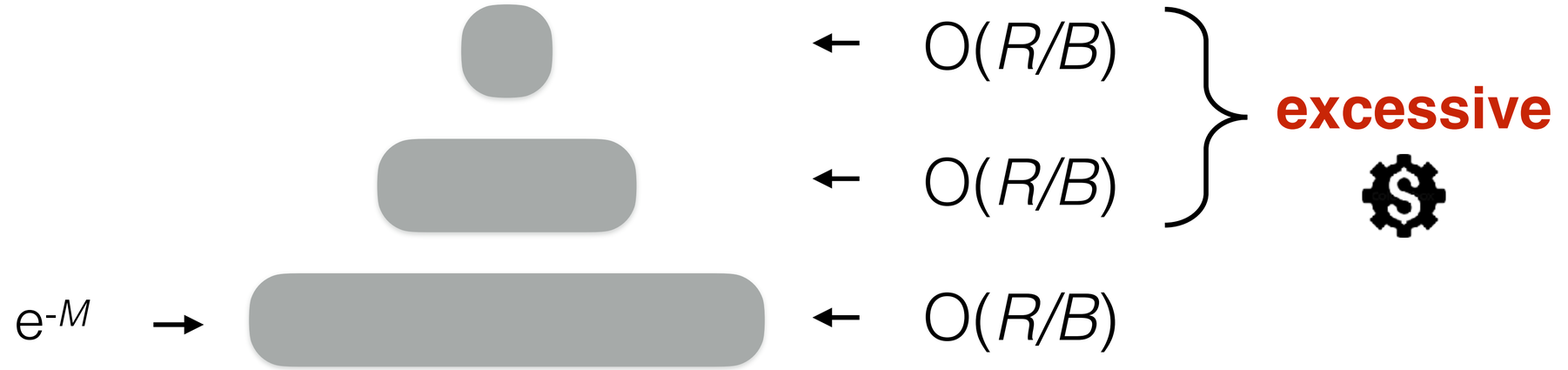
largest level

writes

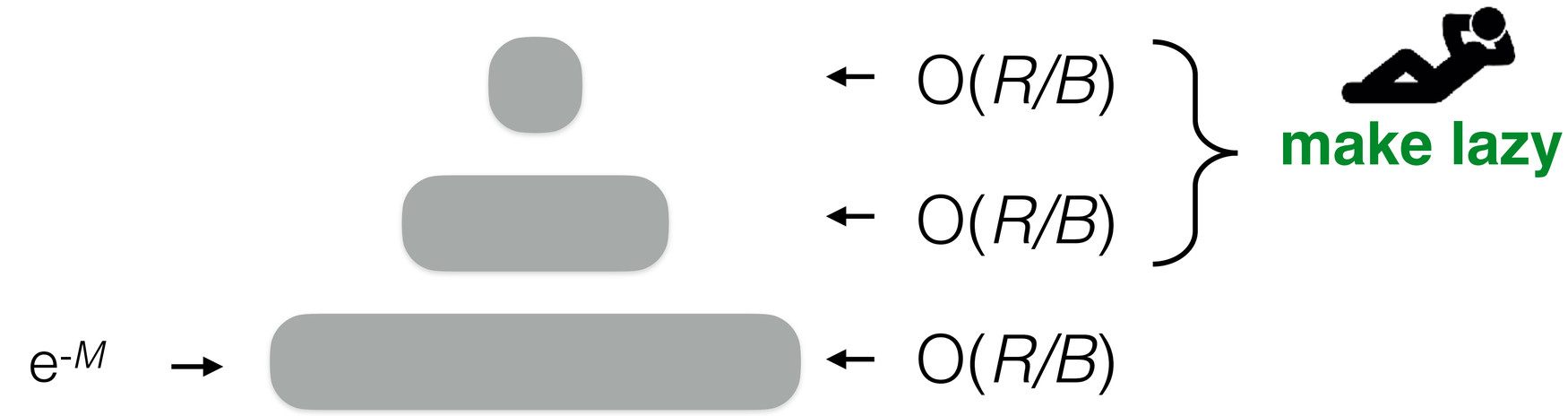
all levels



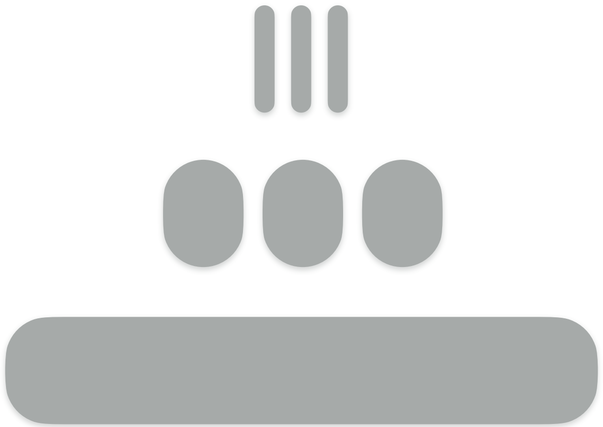
reads **writes**



reads **writes**



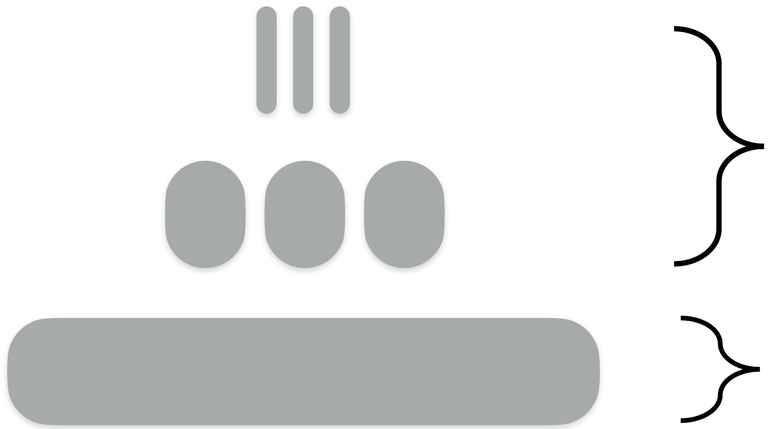
Dostoevsky



lazy

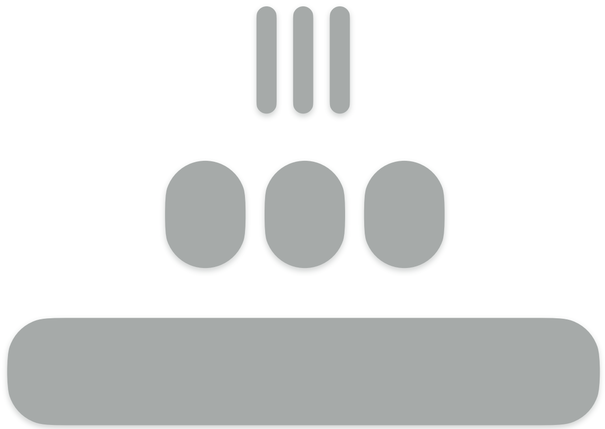
greedy

Dostoevsky



merge when
level fills up

Dostoevsky



merge when

new run comes in

reads **writes**

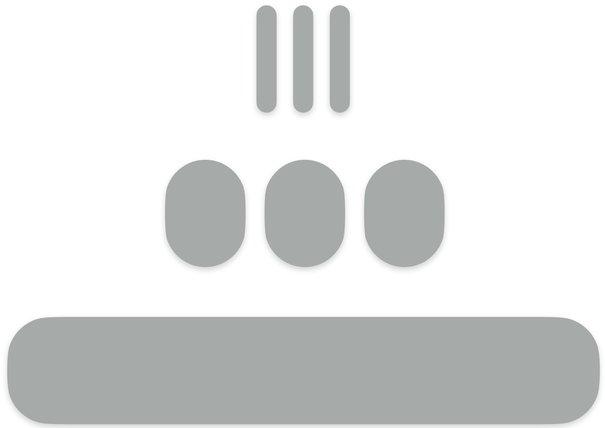
reads **writes**

false positive rates

e^{-M/R^3}

e^{-M/R^2}

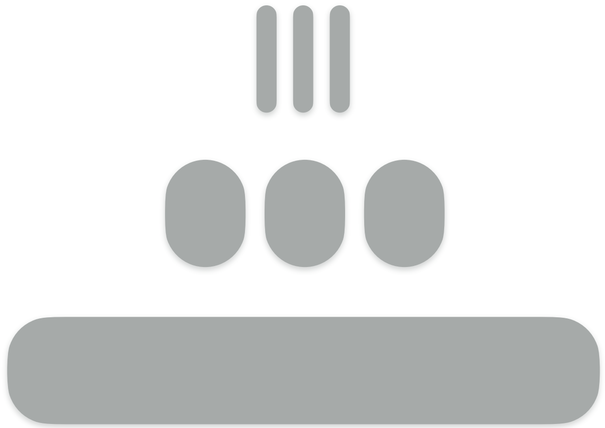
e^{-M}



reads

$O(e^{-M})$

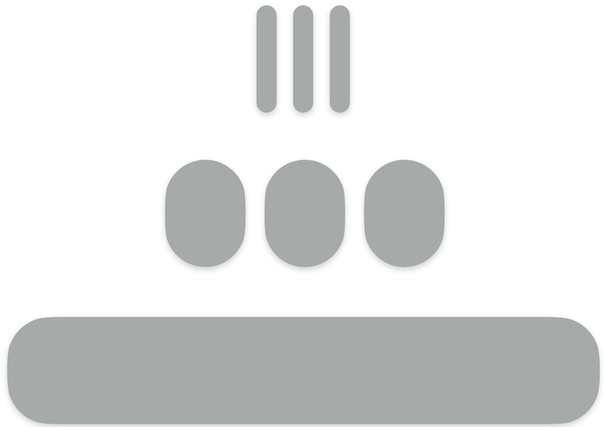
writes



reads

$O(e^{-M})$

writes



O(**1/B**)

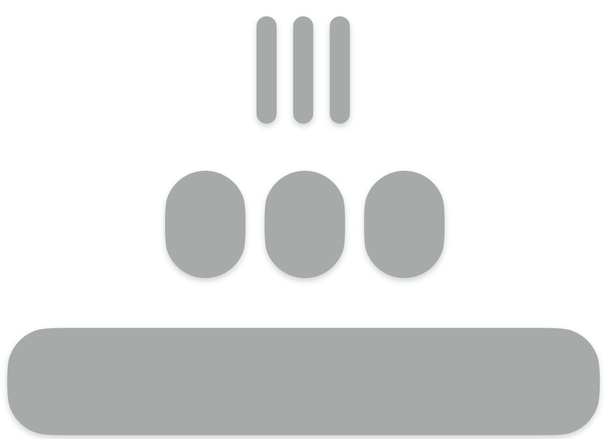
O(**1/B**)

O(***R/B***)

reads

$O(e^{-M})$

writes



$O(\mathbf{1/B})$

$O(\mathbf{1/B})$

$O(\boldsymbol{R/B})$

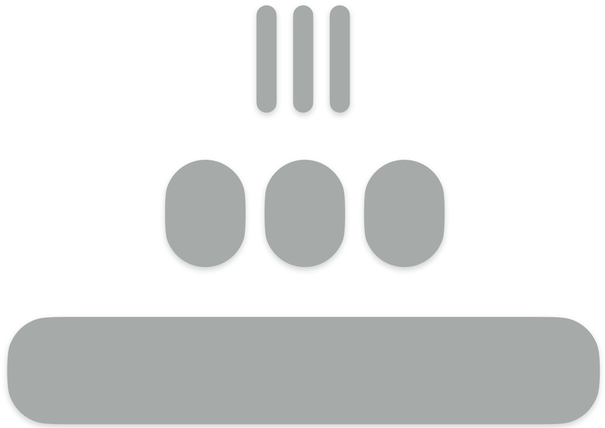
$O((\boldsymbol{R} + \mathbf{L})/\mathbf{B})$

reads

$O(e^{-M})$

writes

$O((R + L)/B)$



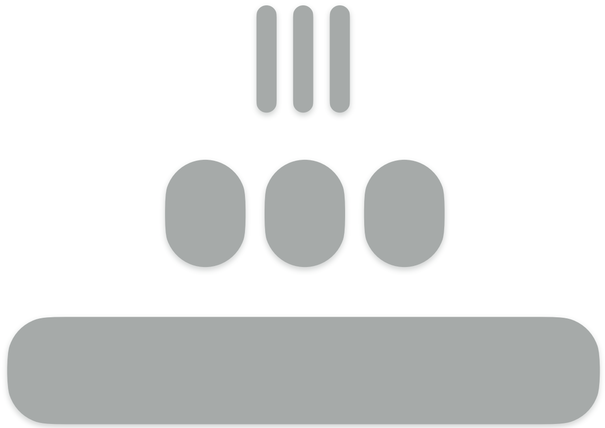
What would our read cost have been if we employed uniform FPRs at all levels?

reads

$O(e^{-M})$

writes

$O((R + L)/B)$



What would our read cost have been if we employed uniform FPRs at all levels?

uniform

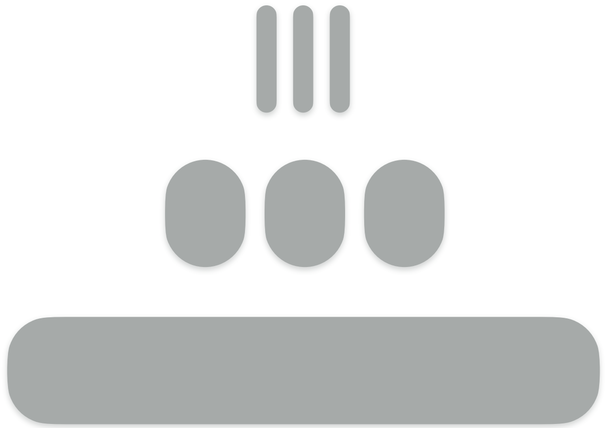
reads

$O(e^{-M} \cdot R \cdot \log_R N)$

e^{-M}

e^{-M}

e^{-M}

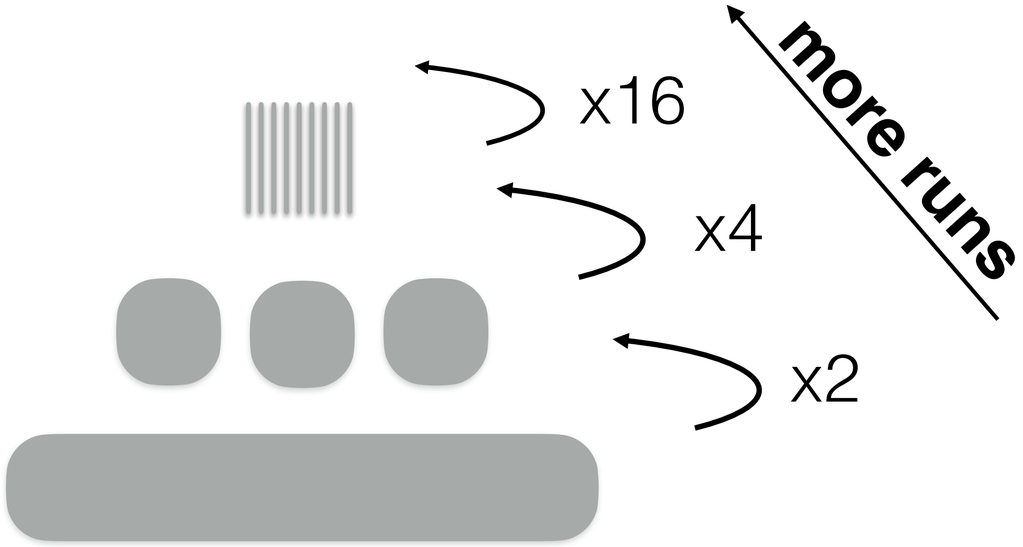


Dostoevsky

$O((R + L)/B)$

LSM-Bush

$O(1/B \log_2 \log_2 N/P)$



SIGMOD 18

SIGMOD 19

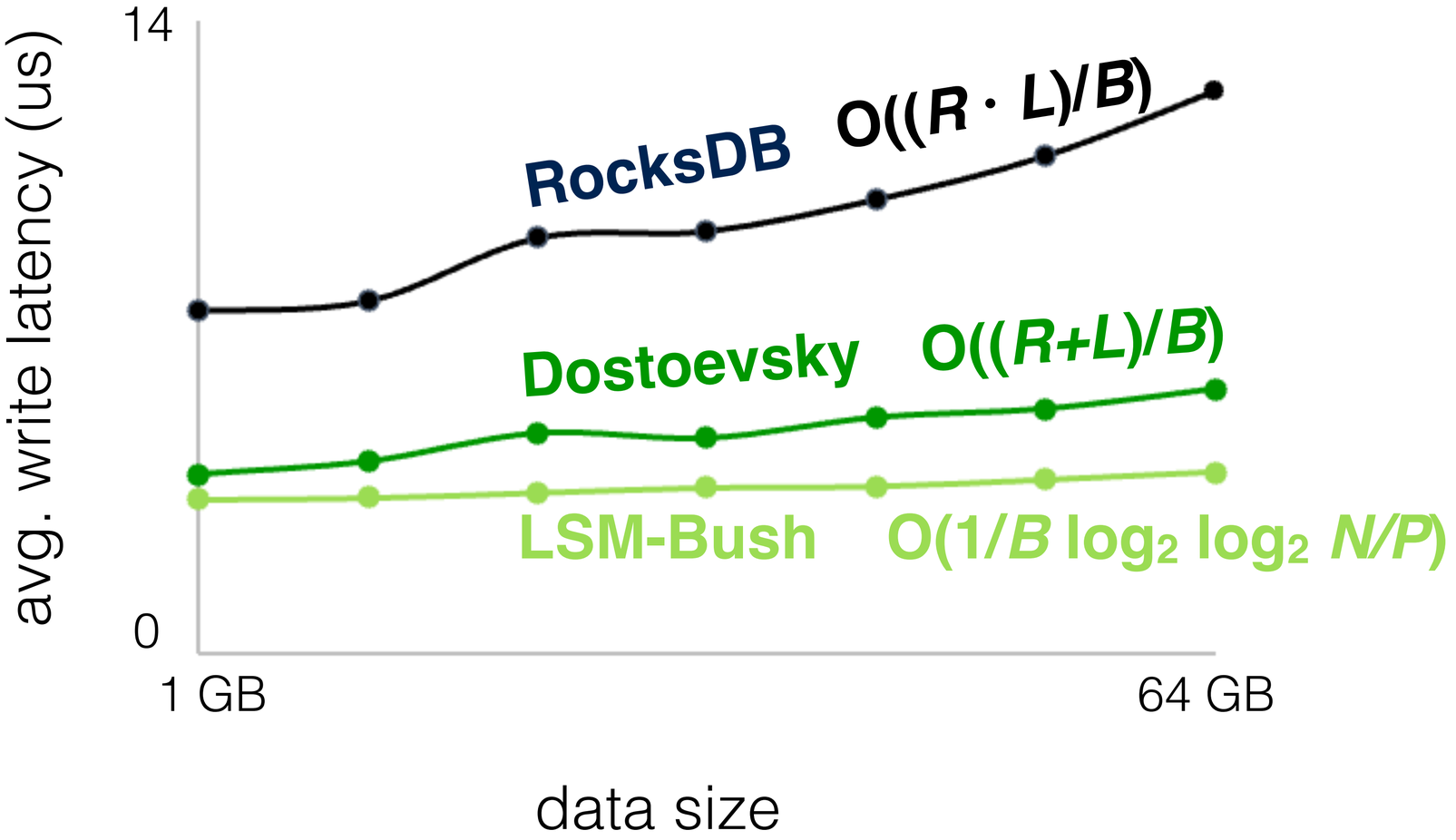
Configuration

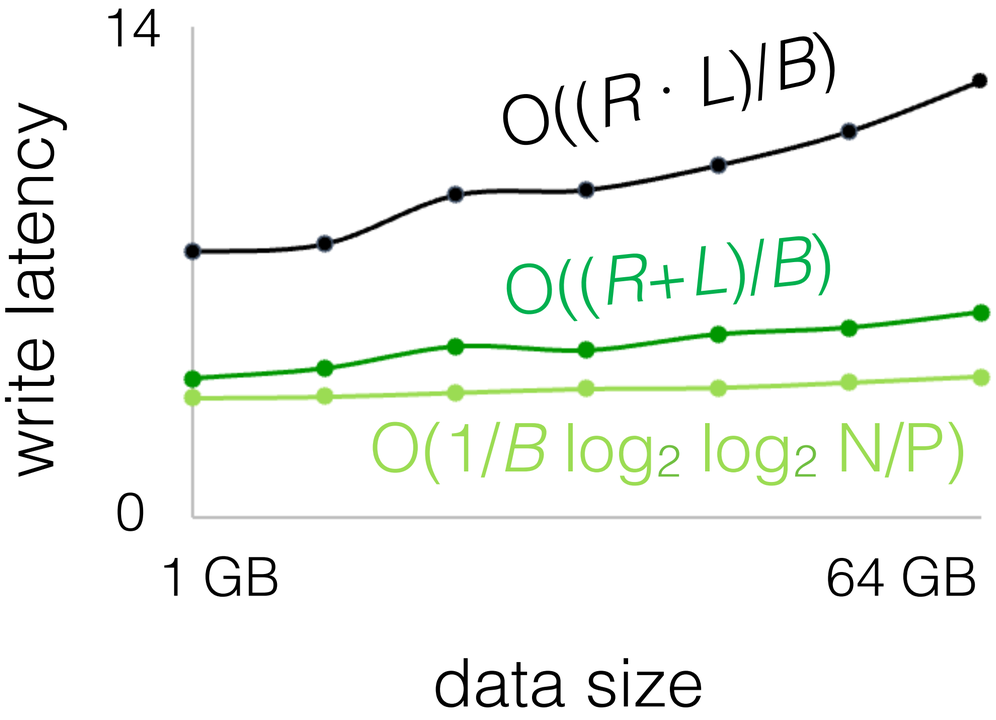
buffer 2MB

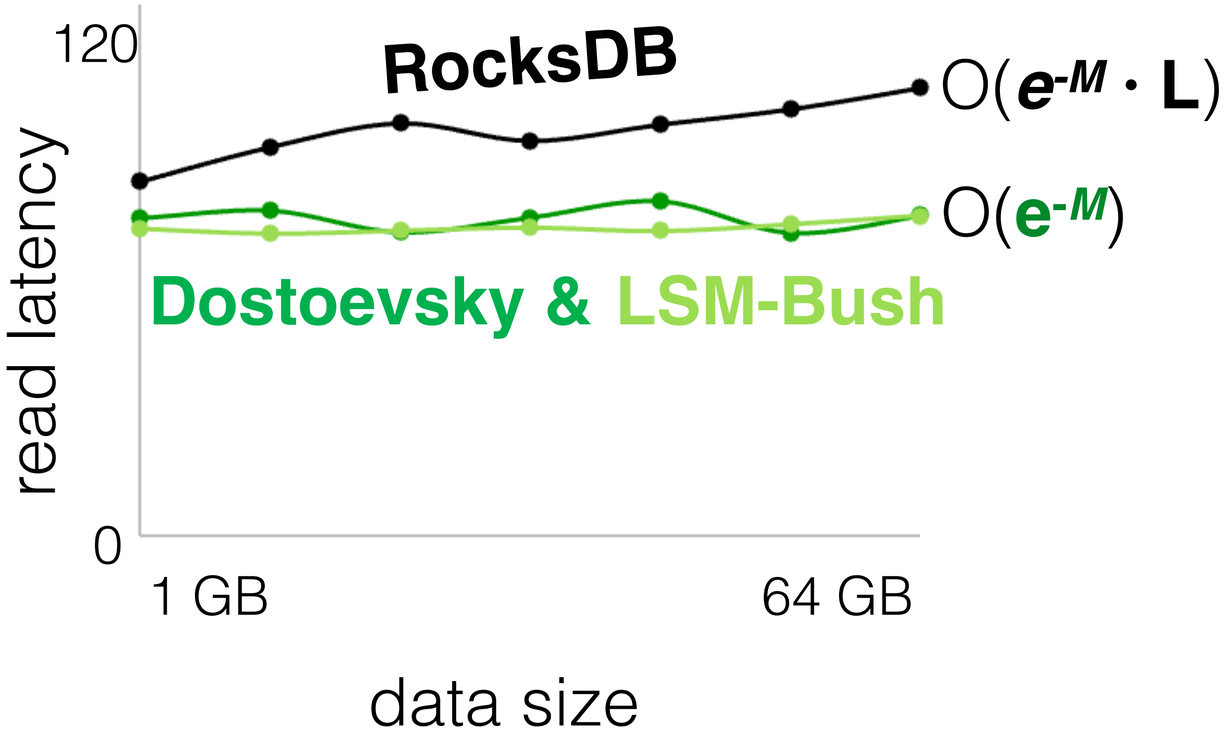
size ratio: 5

1KB entries

SSD storage




14
0
write latency
O((R · L)/B)
O((R+L)/B)
O(1/B log2 log2 N/P)
1 GB
64 GB
data size


120
0
read latency
RocksDB
O(e-M · L)
O(e-M)
Dostoevsky & LSM-Bush
1 GB
64 GB
data size

get I/O cost	$O(e^{-M} \cdot L)$		$O(e^{-M})$		
insert I/O cost	$O((R \cdot L)/B)$		**$O((R+L)/B)$**		**$O(1/B \log_2 \log_2 N/P)$**
transient space-amp	2x		$O(\ ?\ )$		
filter mem. reads	$O(\ M + L\ )$		$O(\ ?\ )$		

$L = \log_R N/P$

(Costs assuming leveling)

Spooky

VLDB 2022

Spooky: **p**artitioned c**o**mpaction f**o**r **k**e**y**-value stores

VLDB 2022

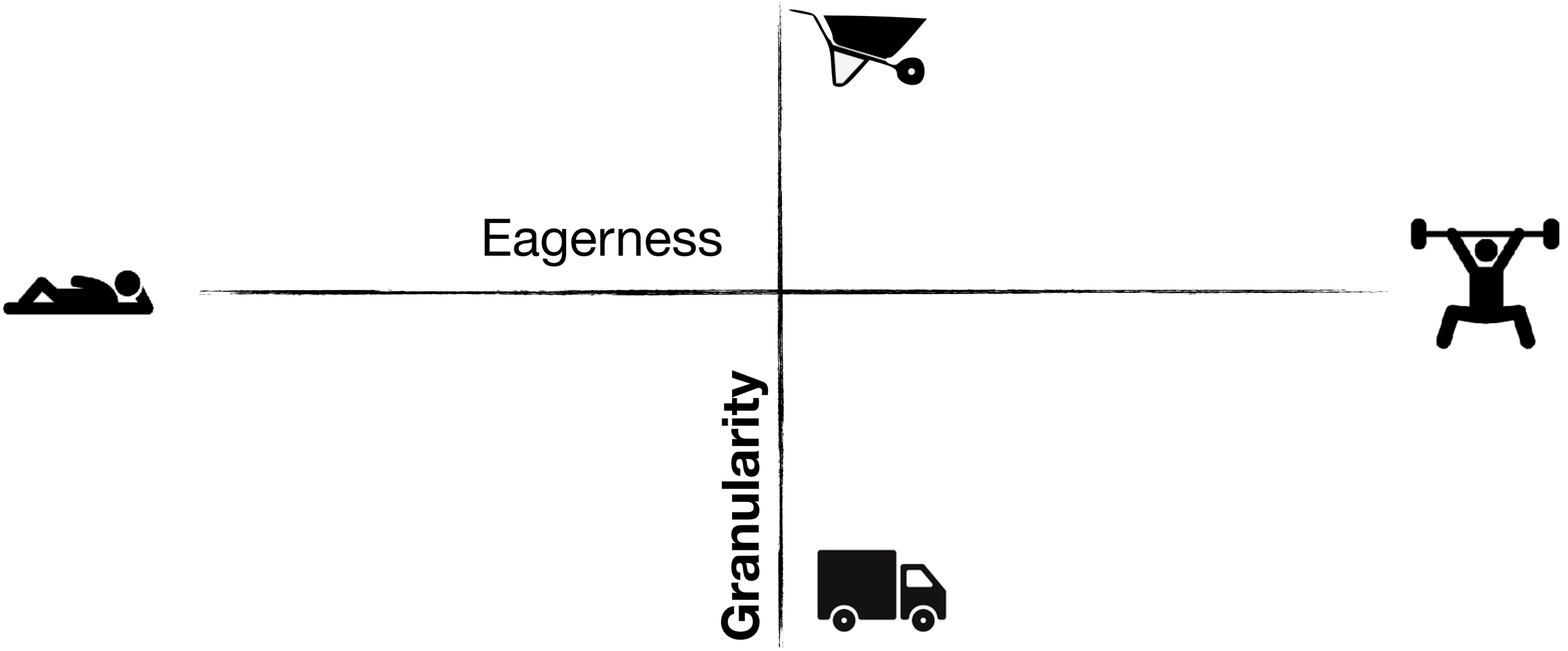
Compaction policy

Write-optimized

Eagerness

Read-optimized

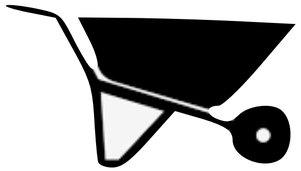
Compaction policy



Compaction Granularity

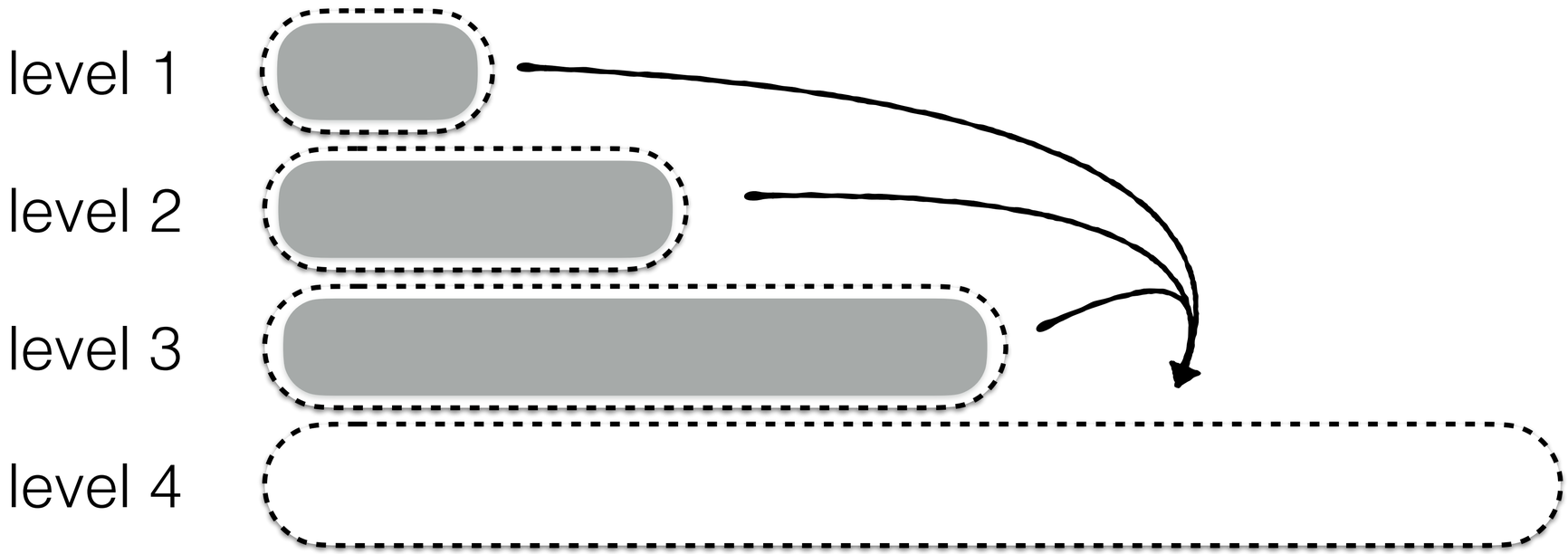
Full Merge

Partial Merge



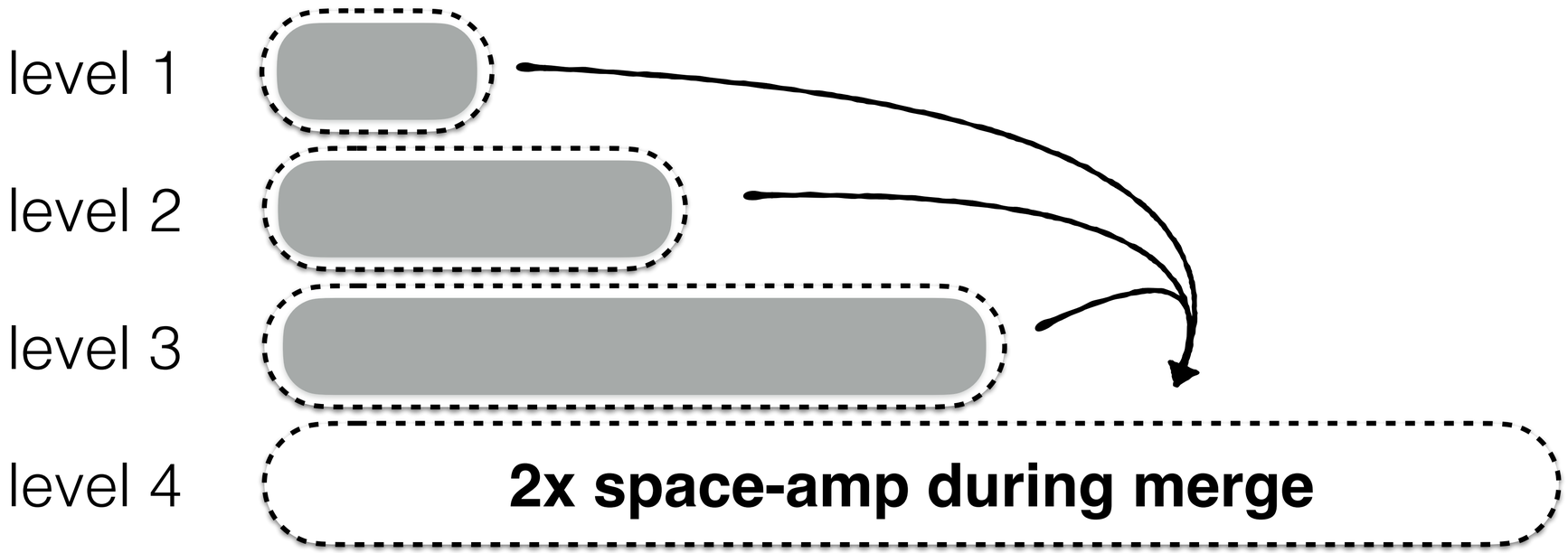
Full Merge

Merge consecutive full levels into first non-full level



Full Merge

Merge consecutive full levels into first non-full level

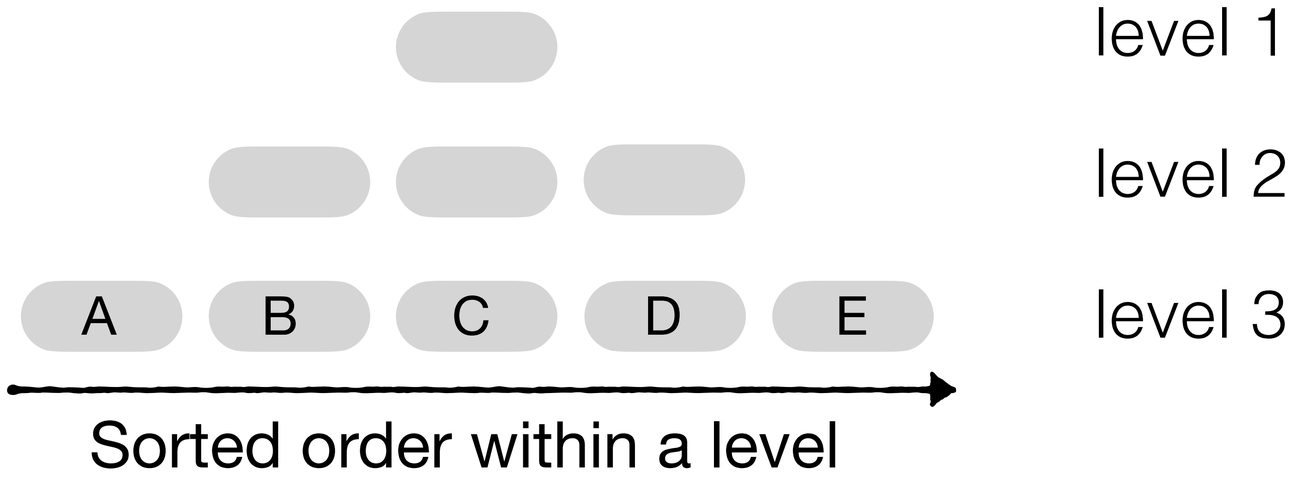


Full Merge

Partial Merge

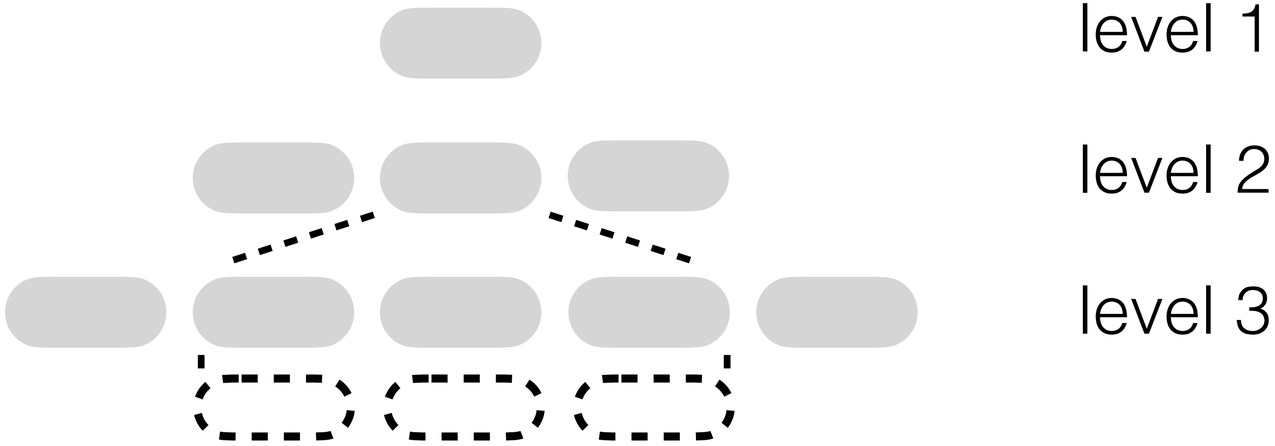
Partial Merge

1. split runs into many files (SSTs) in each level



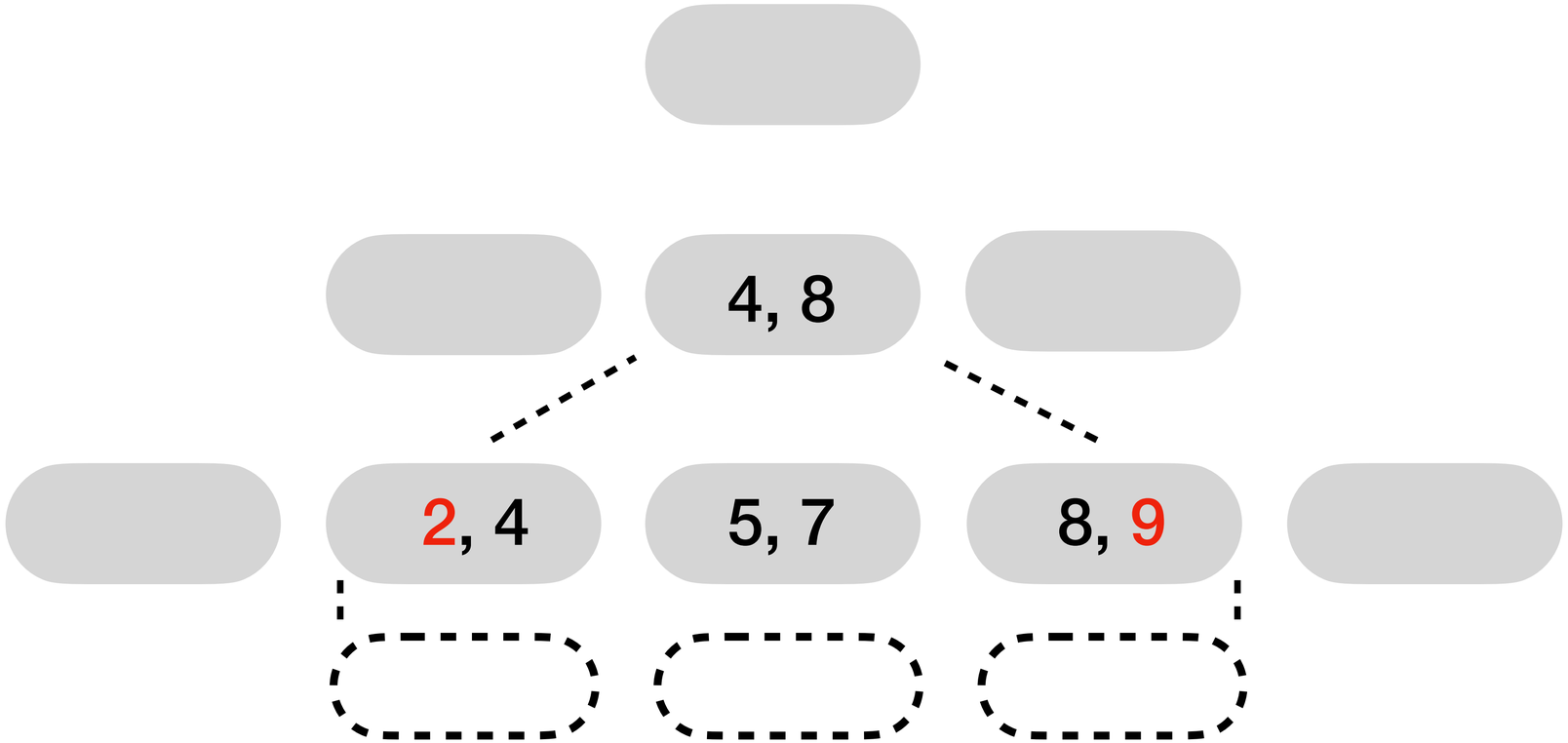
Partial Merge

1. split runs into many files (SSTs) in each level
2. When a level is full, pick SST with smallest intersection into next level



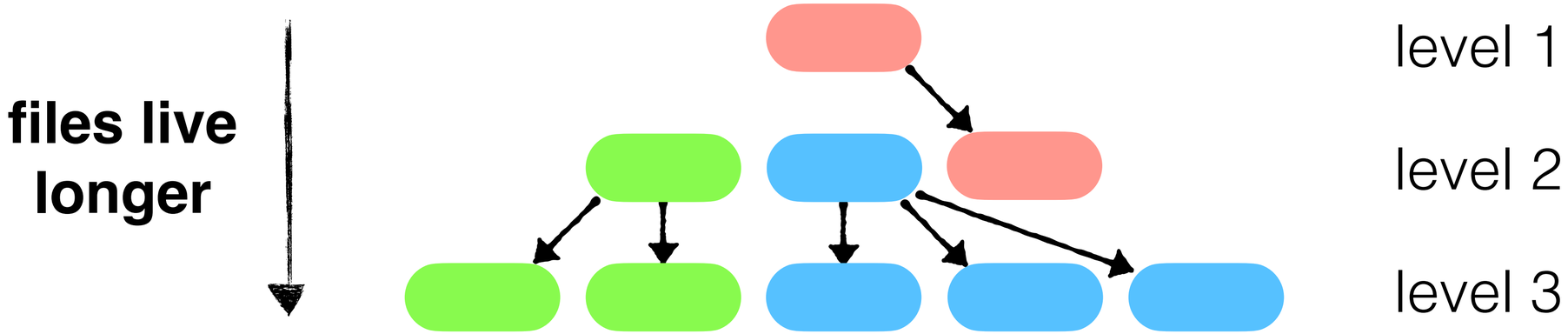
Partial Merge

Problem 1: non-intersecting entries increase write-amplification



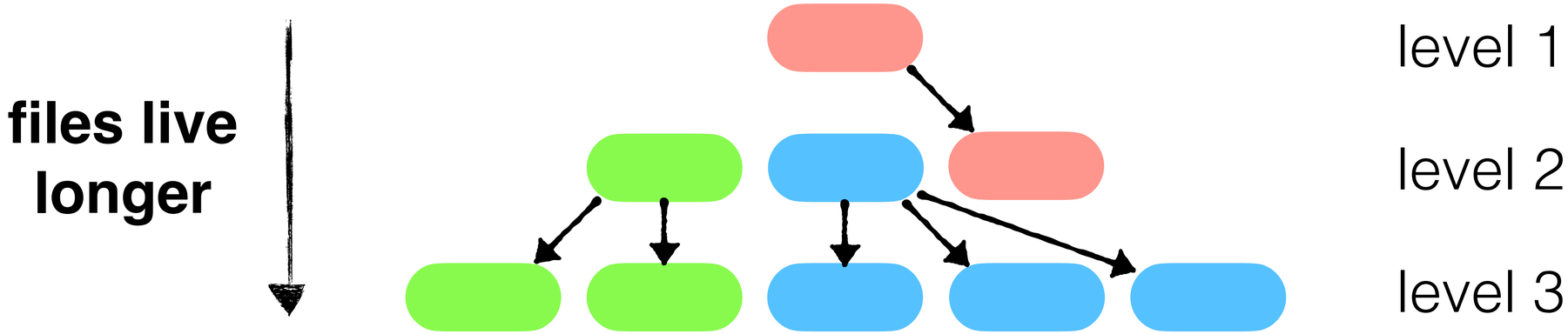
Partial Merge

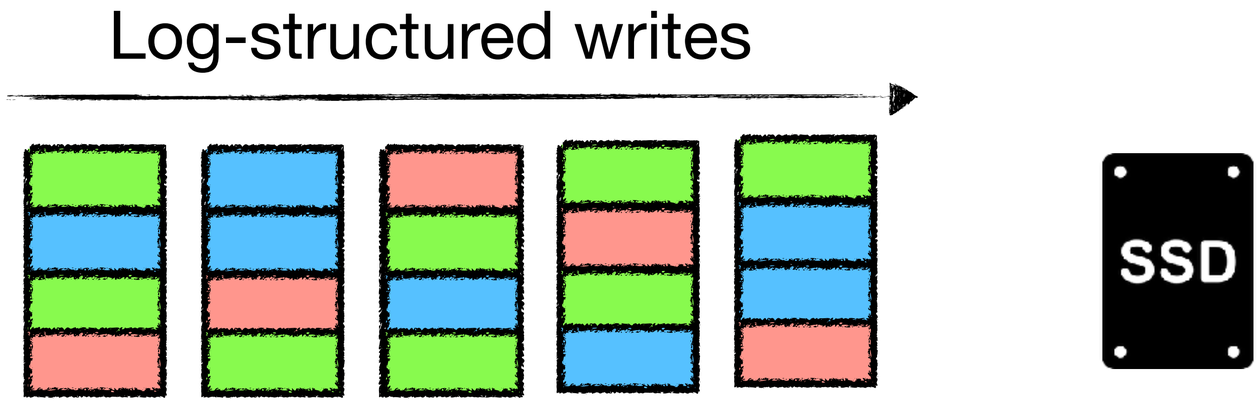
Problem 2: many small simultaneous compactions



Partial Merge

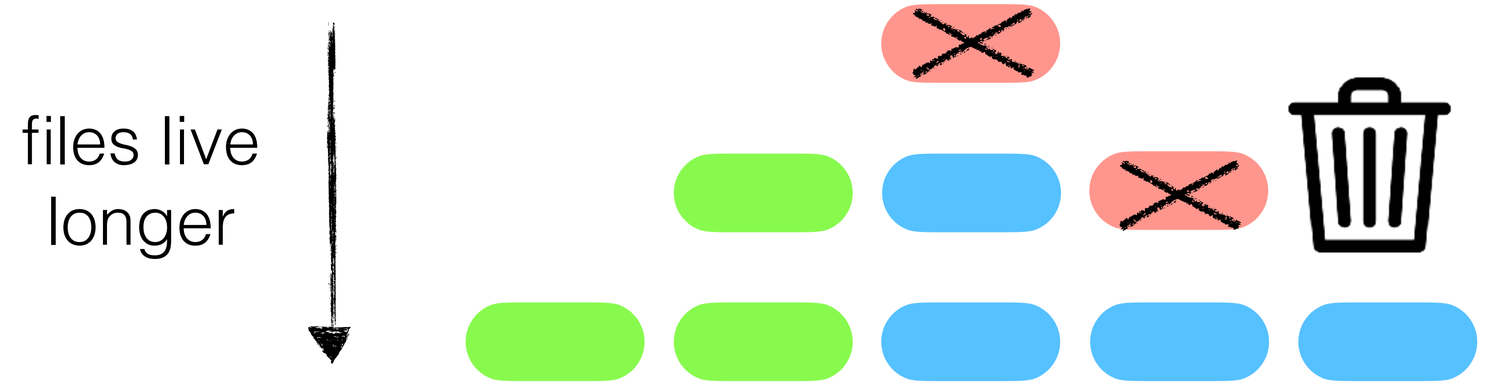
Problem 2: many small simultaneous compactions

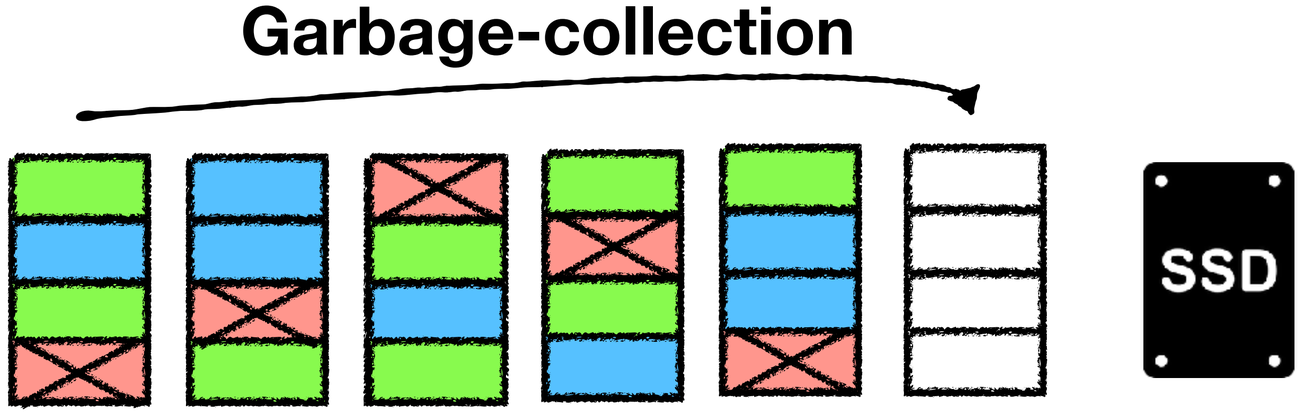




Partial Merge

Problem 2: many small simultaneous compactions

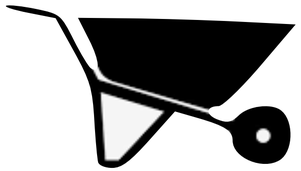




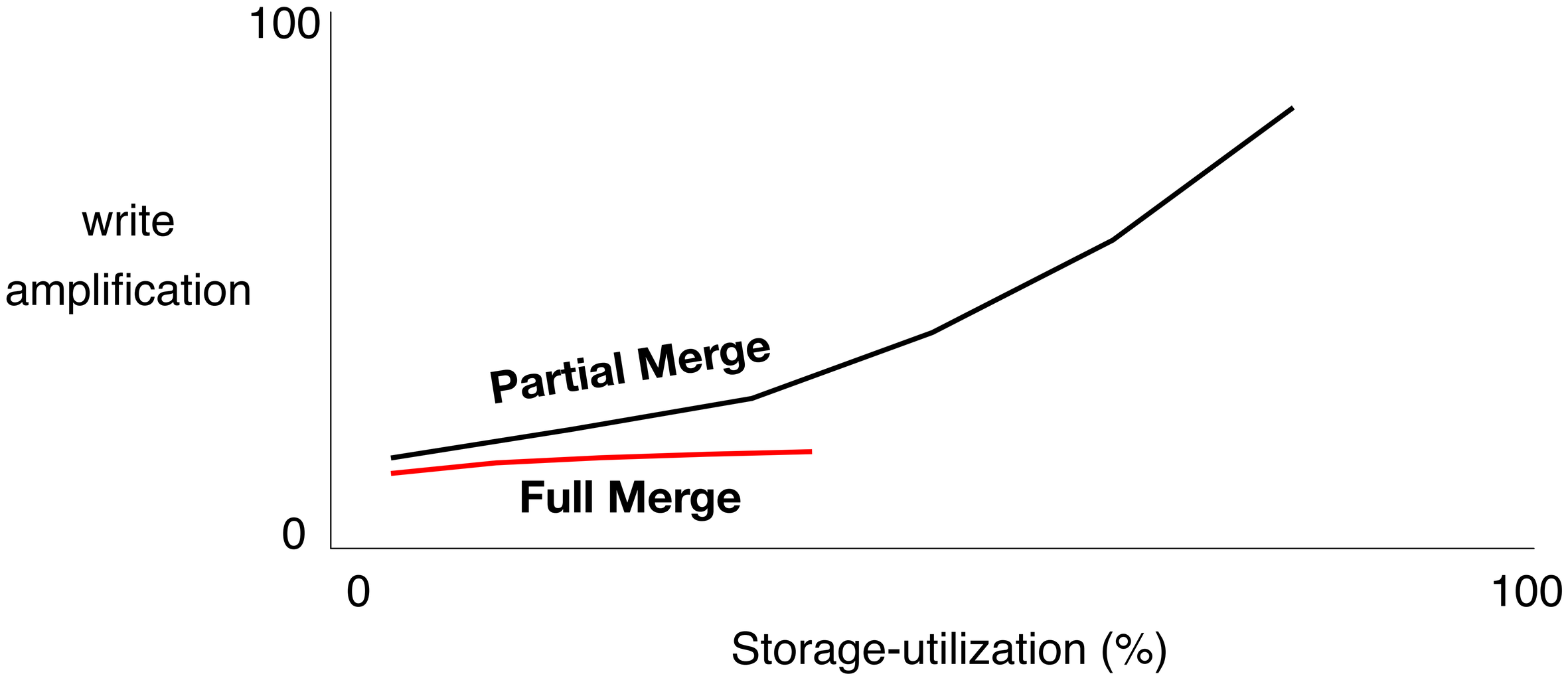
Full Merge

space amplification

Partial Merge



write amplification

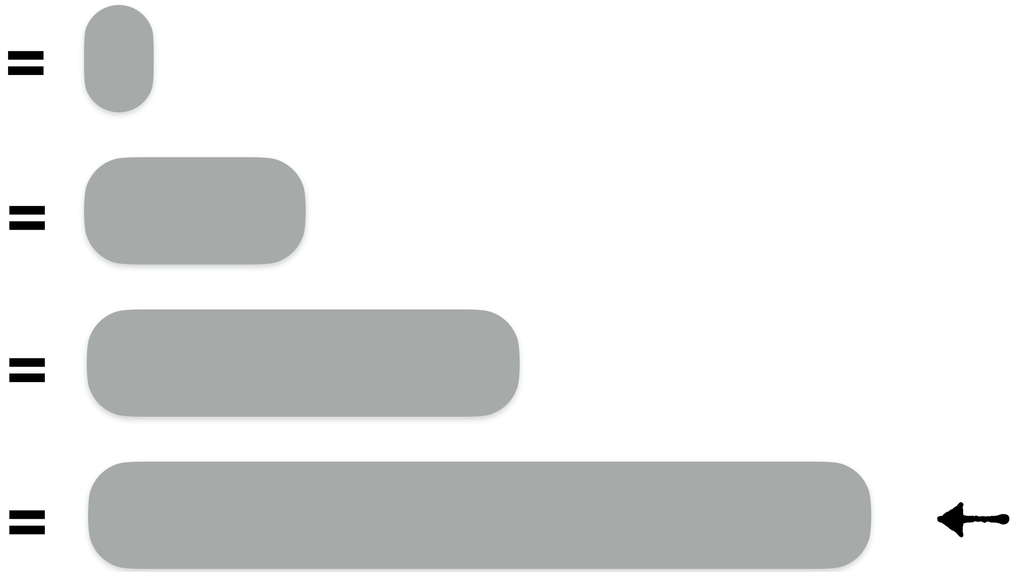

100
write
amplification
Partial Merge
Full Merge
0
0
100
Storage-utilization (%)

Spooky's intuition

transient
space amplification

Spooky's intuition

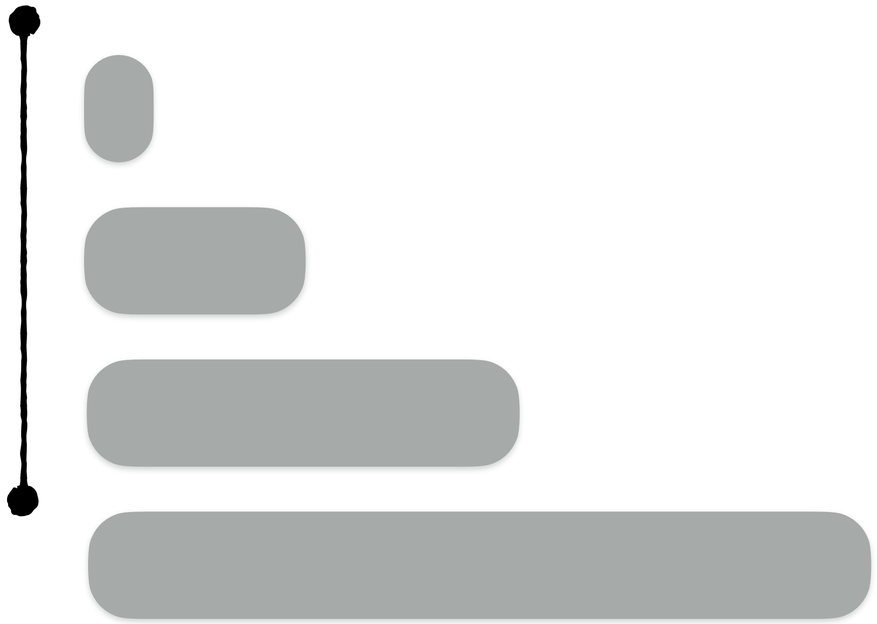
write amplification



transient
space amplification

Spooky's intuition

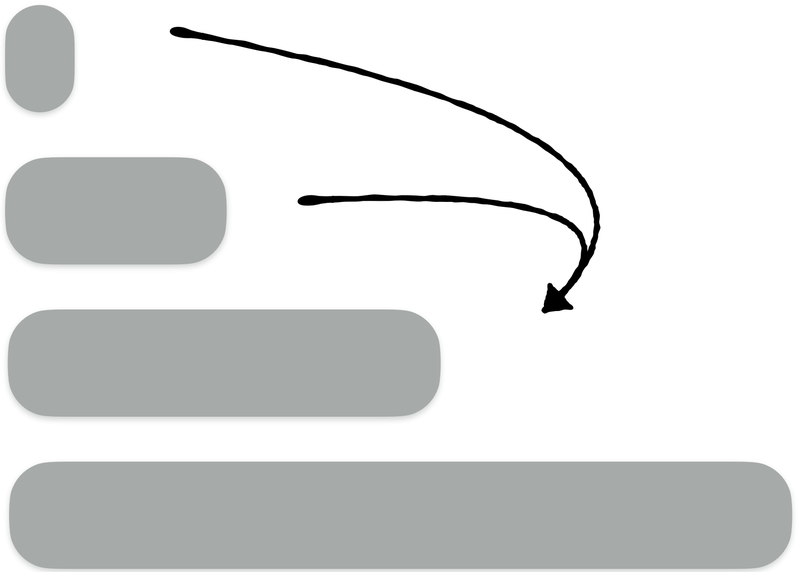
Full merge
(write optimized)



Partial Merge Variant
(space optimized)

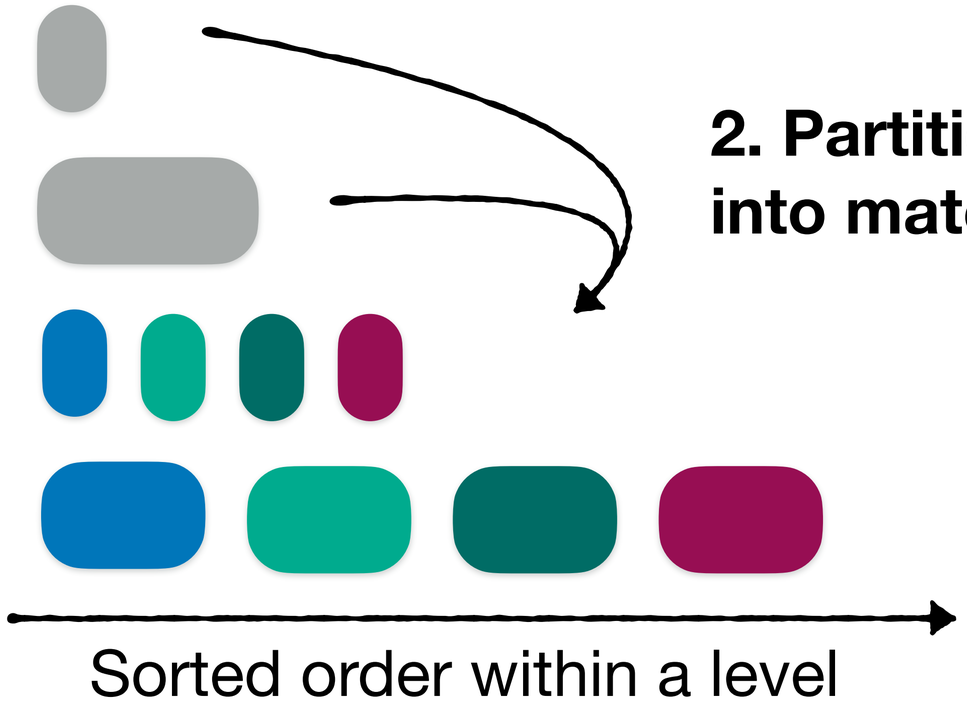
Spooky

1. Full merge at up to second largest level



Spooky

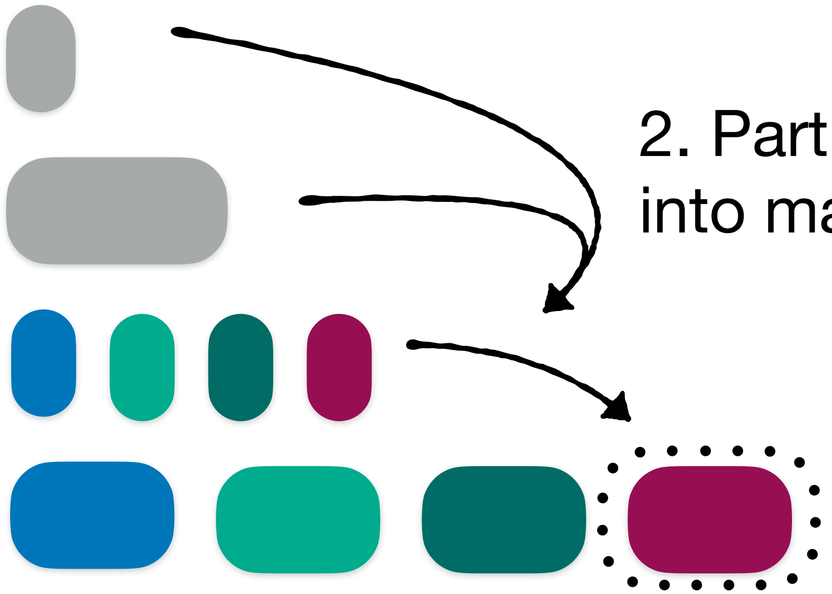
1. Full merge at up to second largest level



2. Partition largest two level into matching pairs of files

Spooky

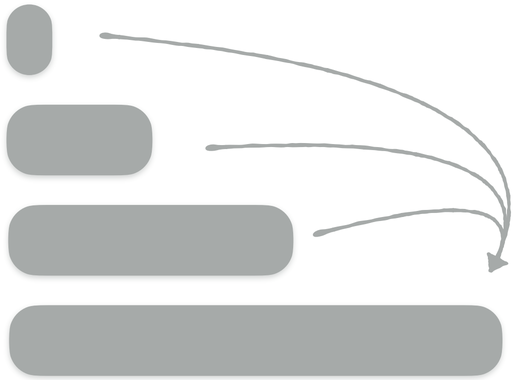
1. Full merge at up to second largest level



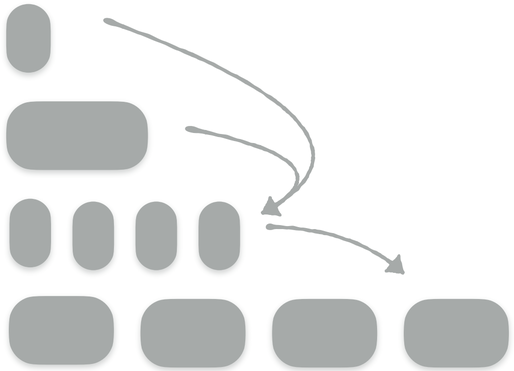
2. Partition largest two level into matching pairs of files

3. Merge one partition at a time

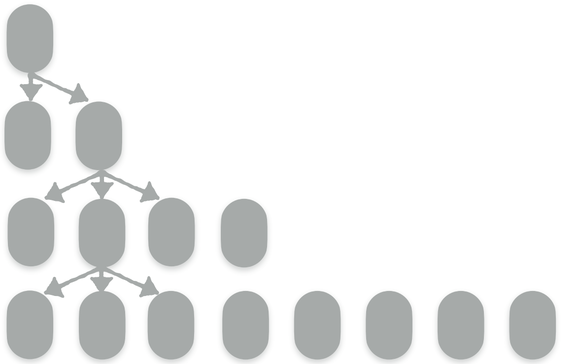
Full

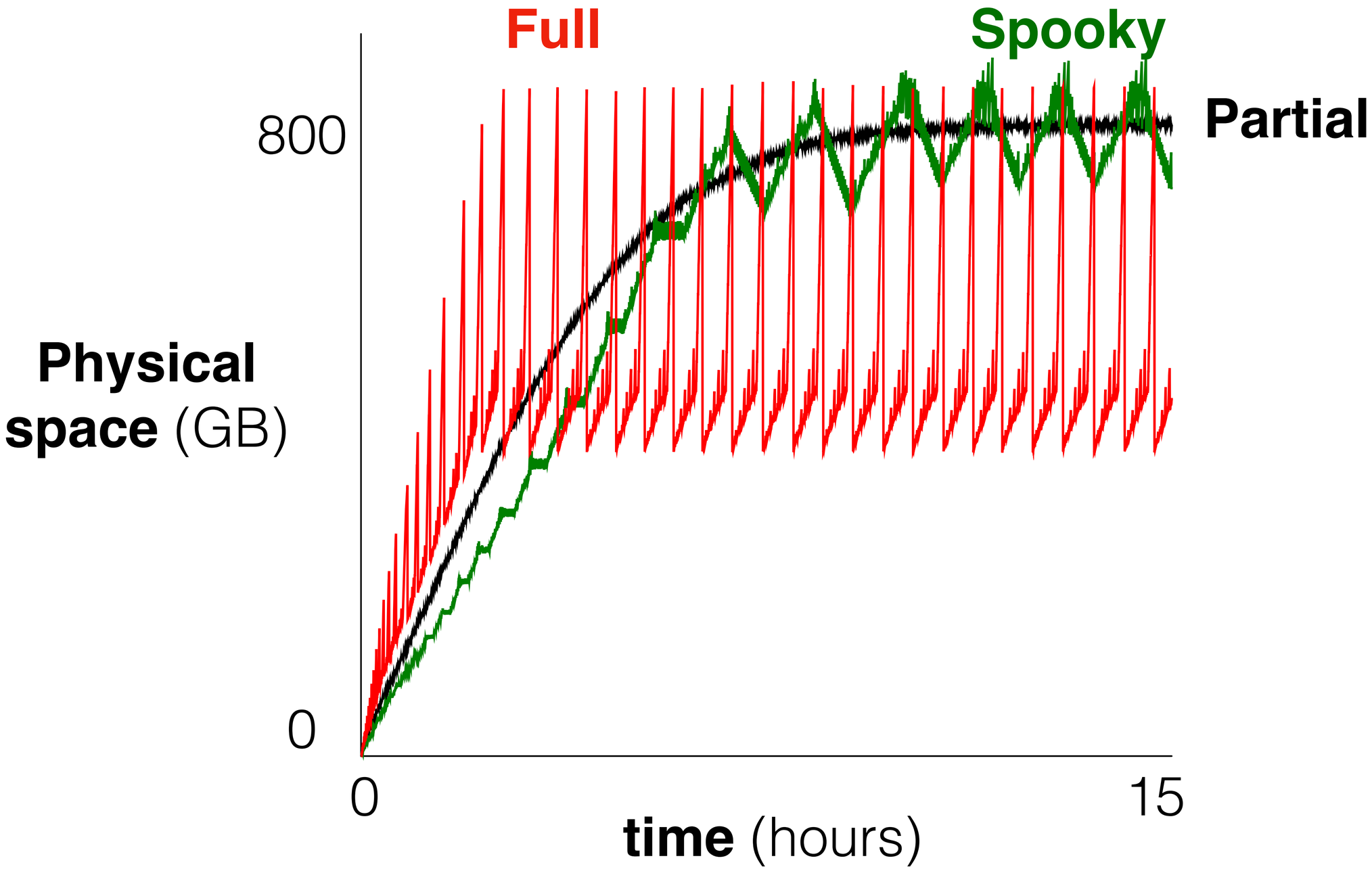


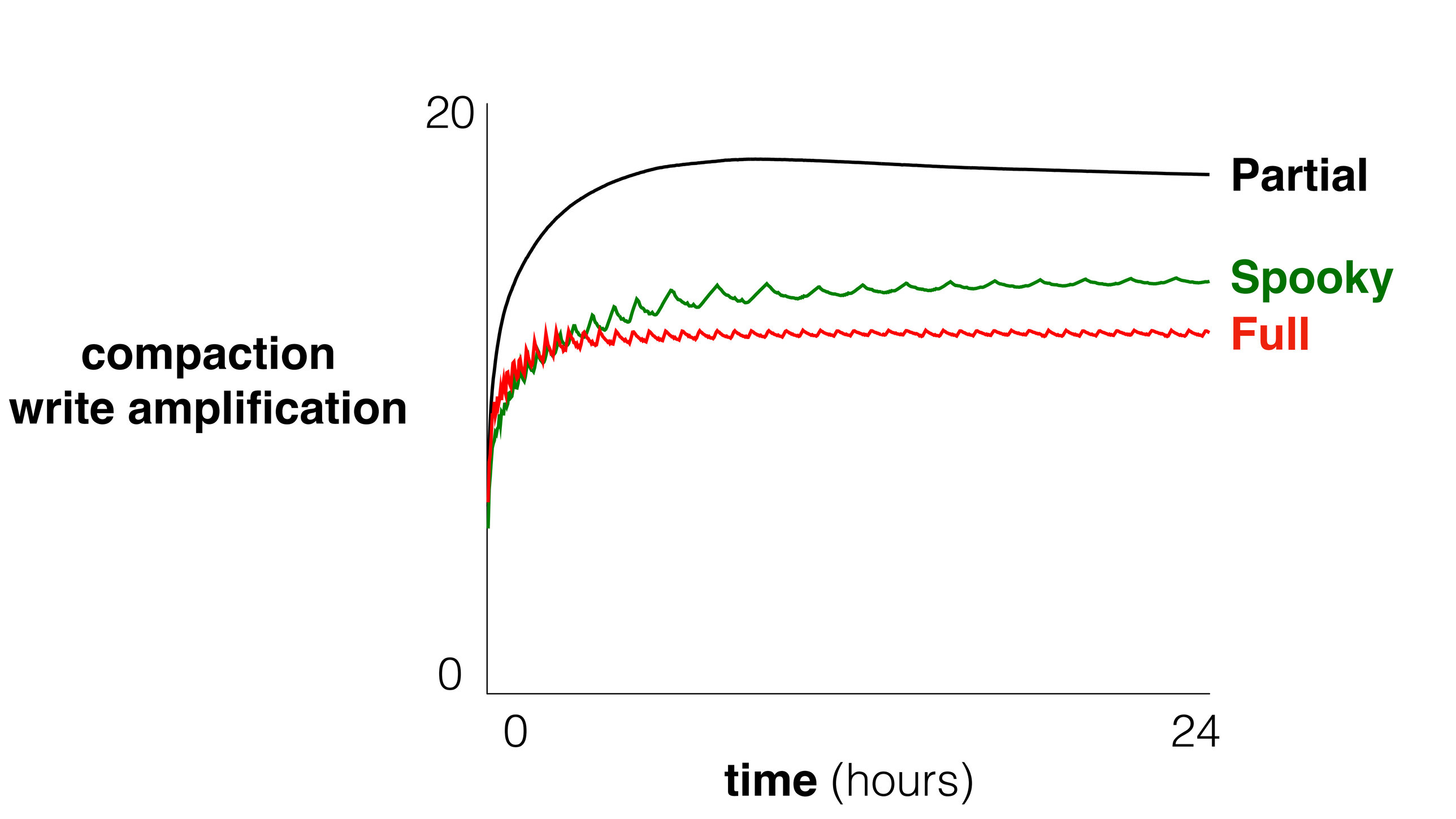
Spooky



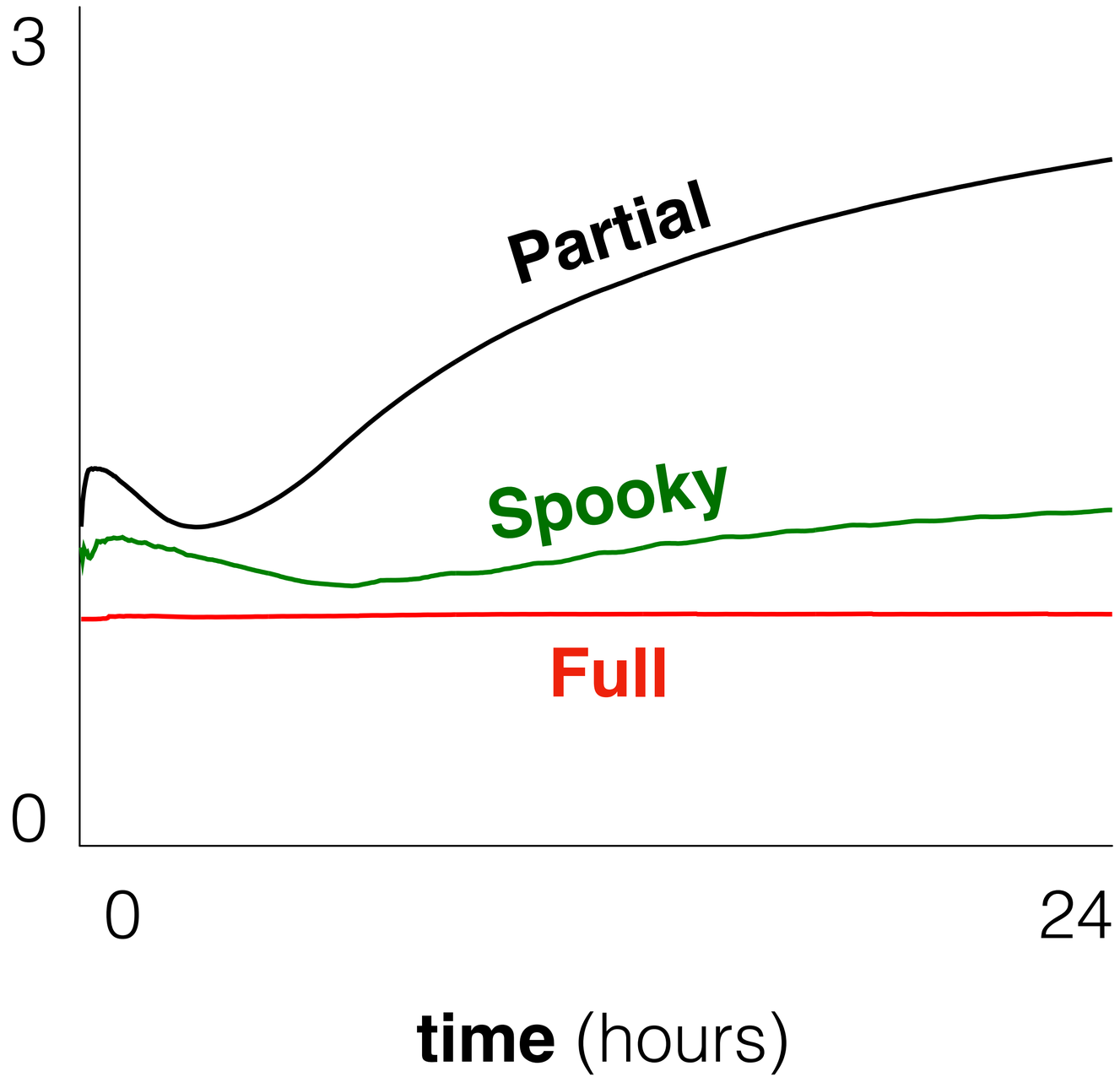
Partial




Full
Spooky
Partial
800
0
Physical space (GB)
0
15
time (hours)


compaction
write amplification
20
0
0
24
time (hours)
Partial
Spooky
Full

Garbage Collection
write amplification



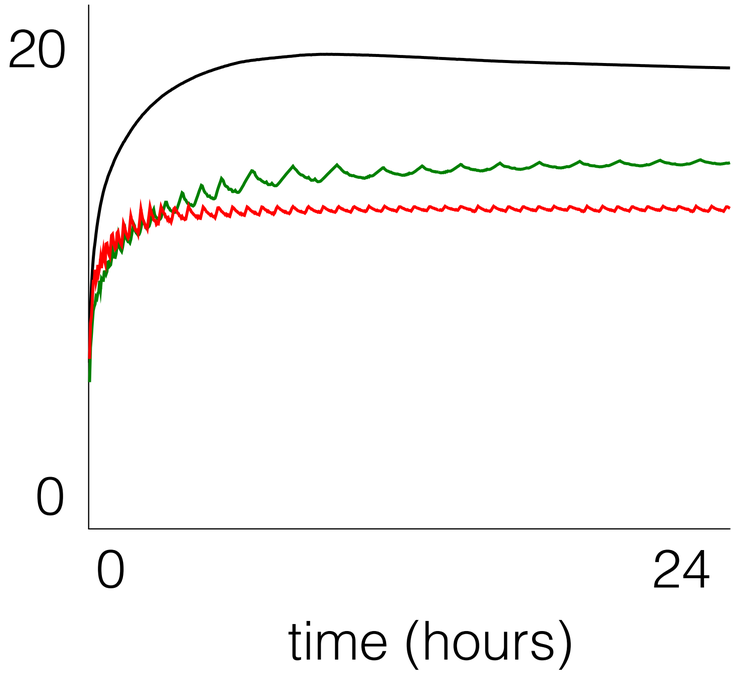
Compaction
write amplification

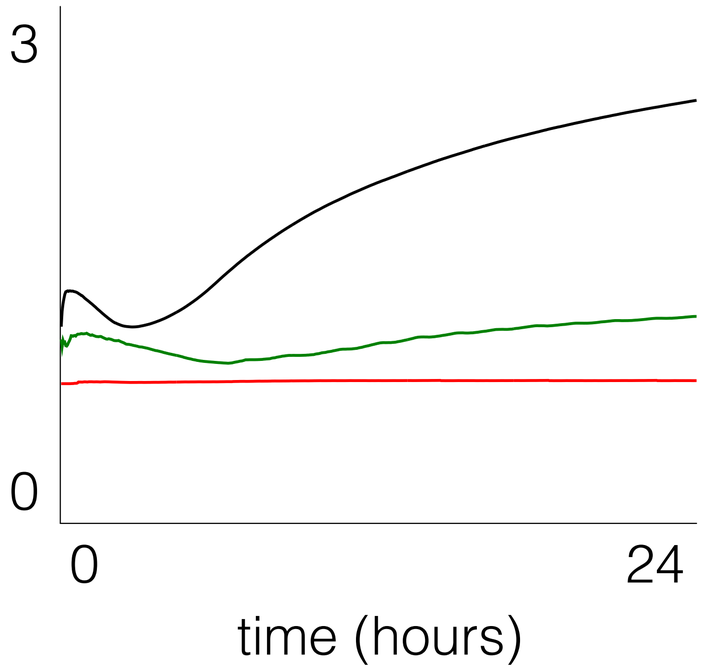
X

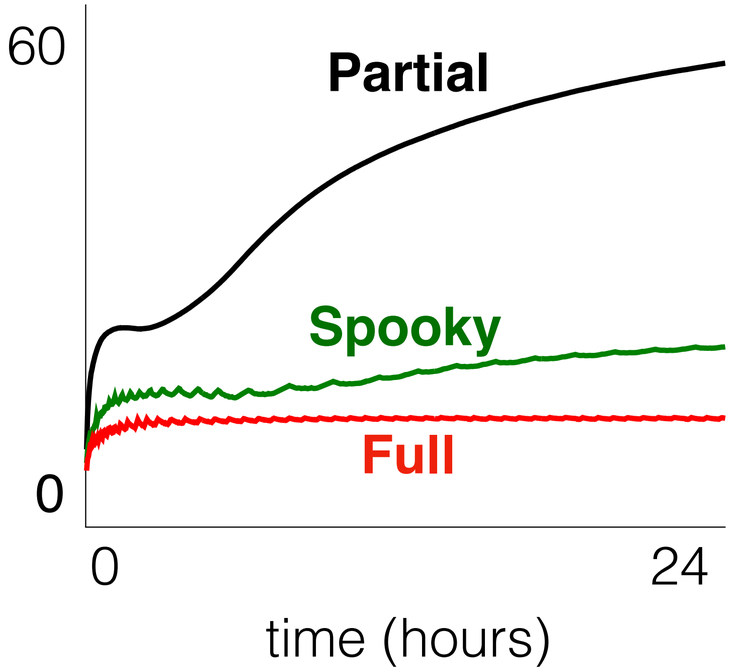
Garbage Collection
write amplification

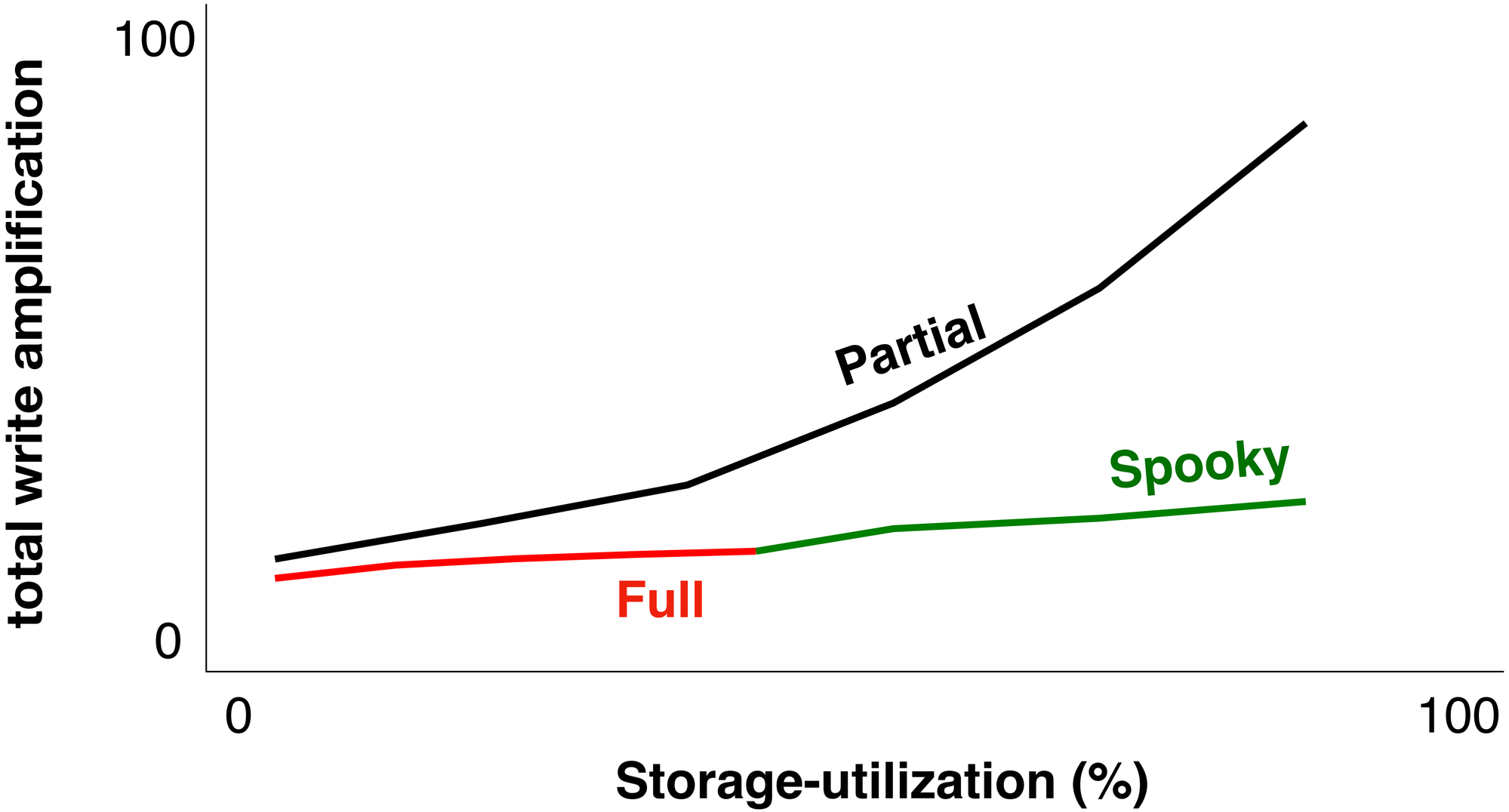
=

Total
write amplification








100
0
total write amplification
Partial
Spooky
Full
0
100
Storage-utilization (%)

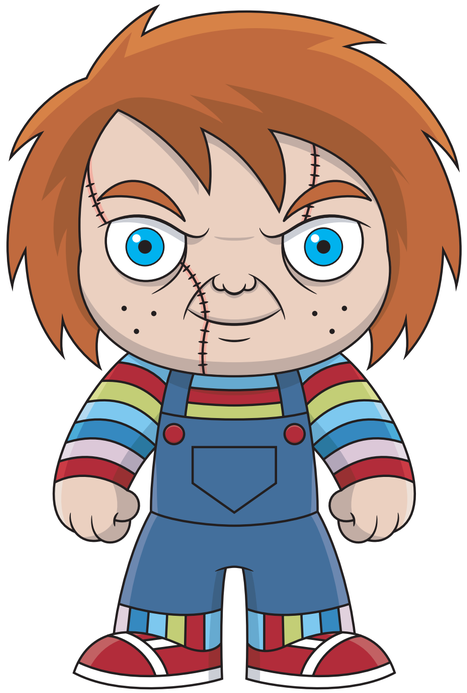
get I/O cost	$O(e^{-M} \cdot L)$		$O(e^{-M})$
insert I/O cost	$O((R \cdot L)/B)$		$O((R+L)/B)$
transient space-amp	**≈2**		**≈1**
filter mem. reads	$O(M + L)$		$O(?)$

$O((\log_2 \log_2 N)/B)$

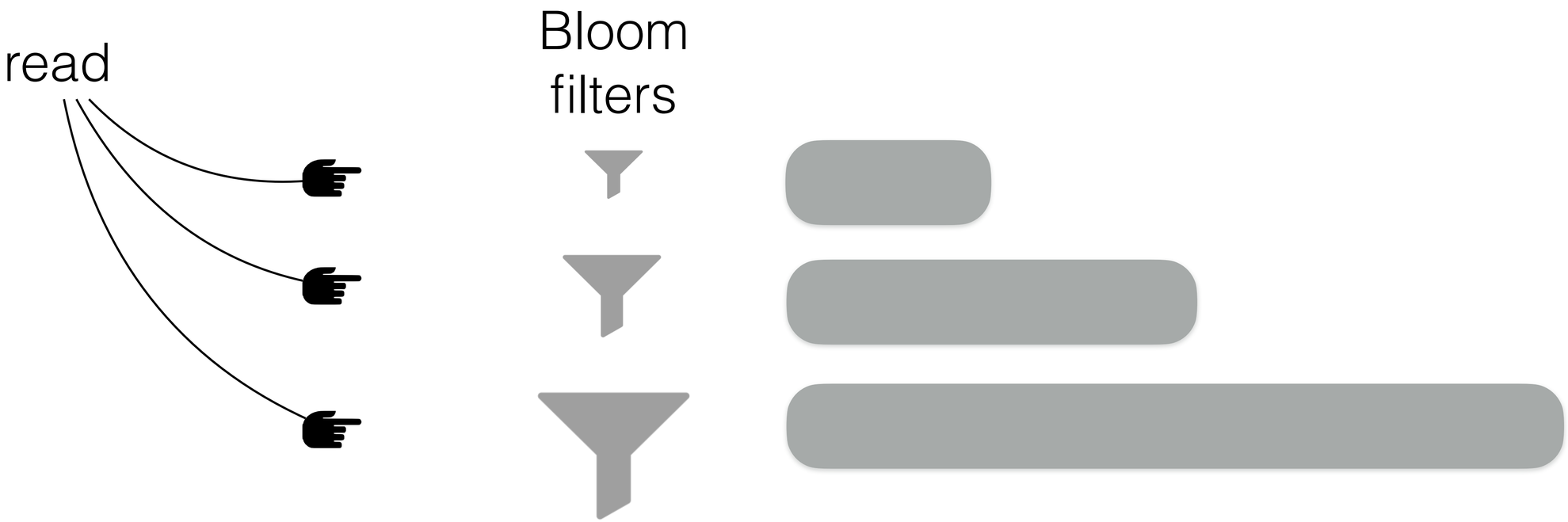
$L = \log_R N/P$

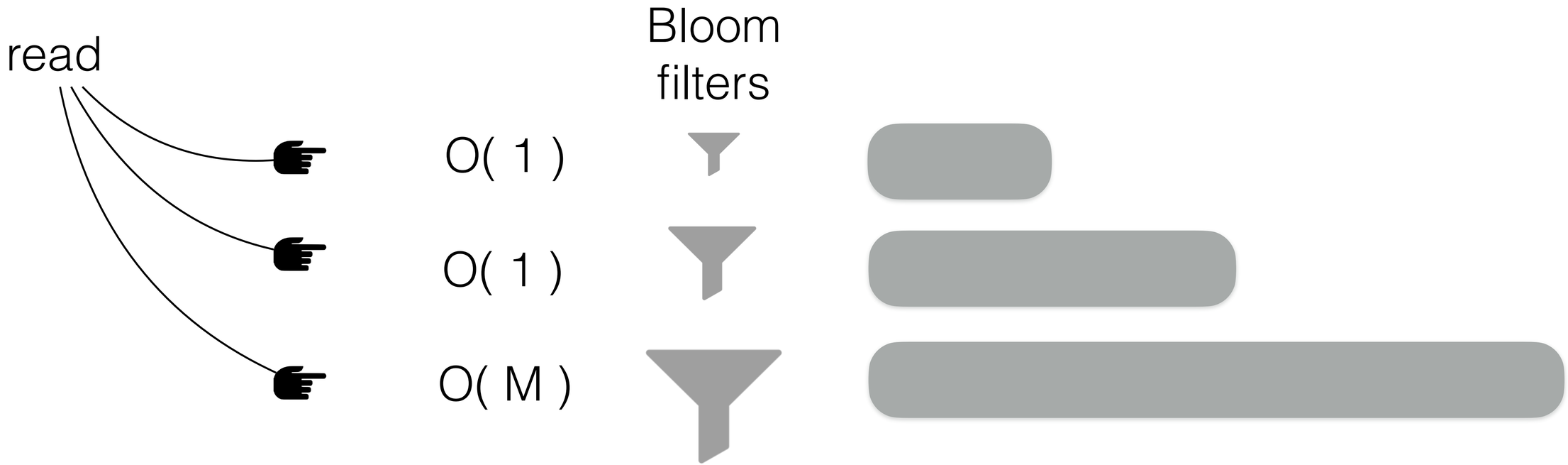
(Costs assuming leveling)

Chucky: Succinct Cuckoo Filter for LSM-Tree



SIGMOD 2021


read
Bloom
filters

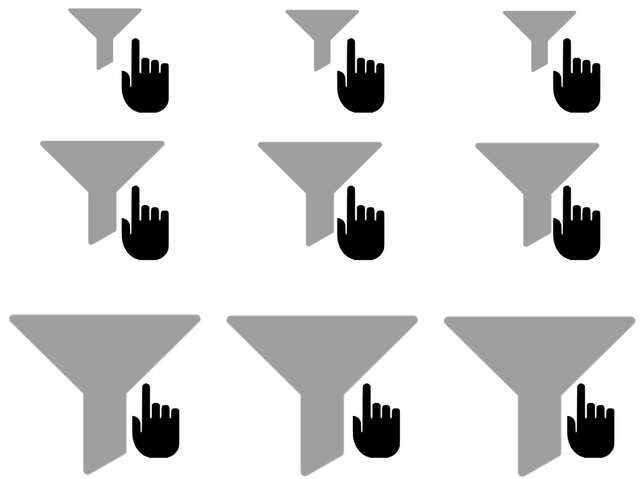


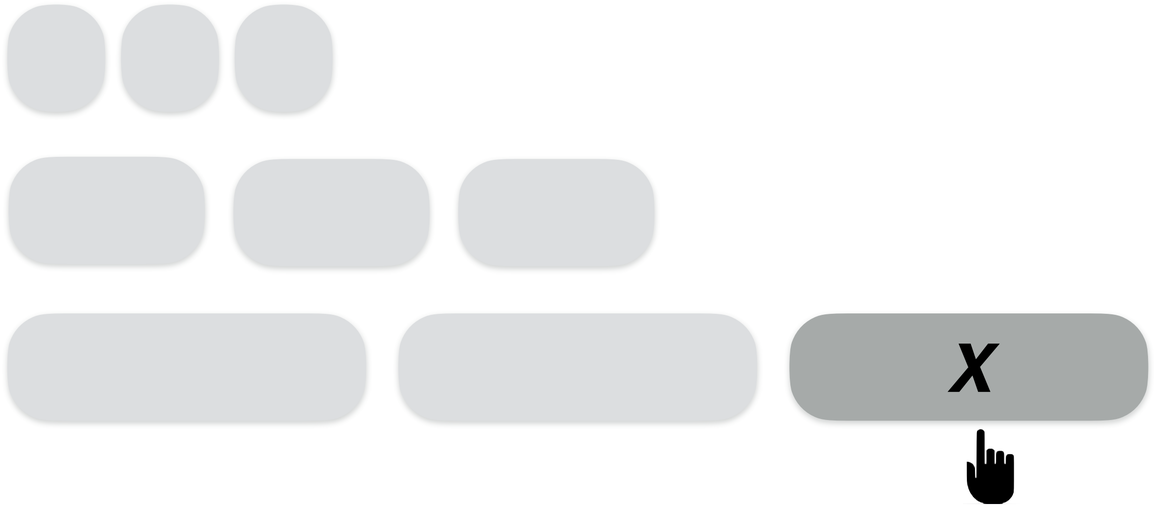
= O(M + L)

O(*R*)

O(*R*)

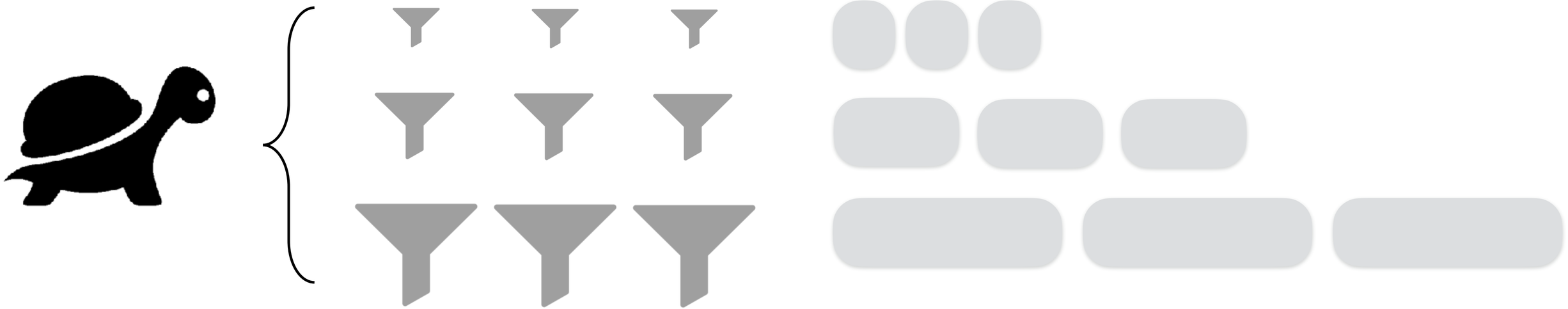
O(*R* + M)





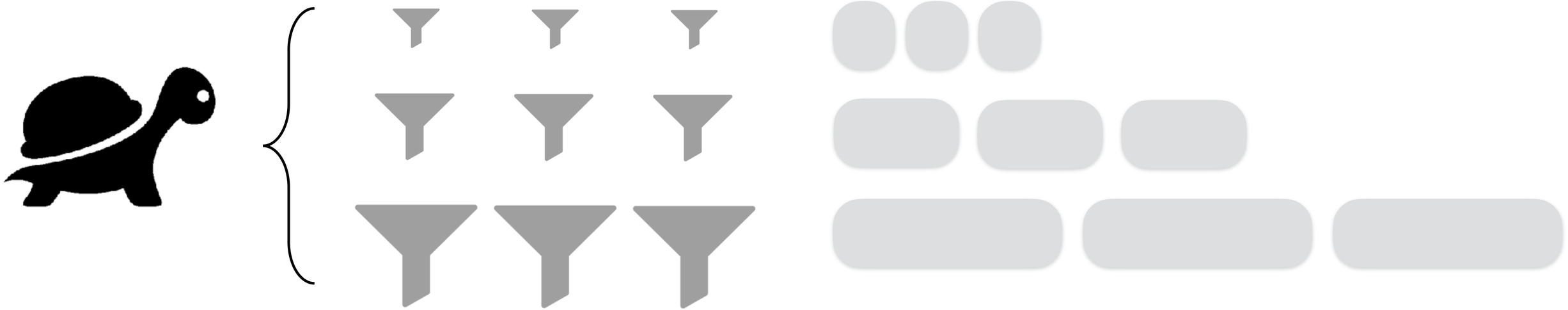
= O(*M* + *L* · R)

Problem: Reduce filter accesses

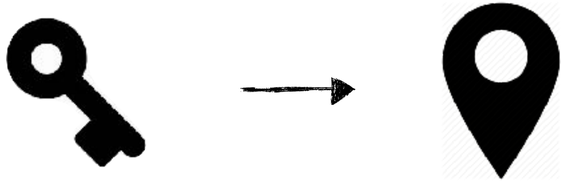


Problem: Reduce filter accesses

Constraint: Keep false positives & memory the same



hash table

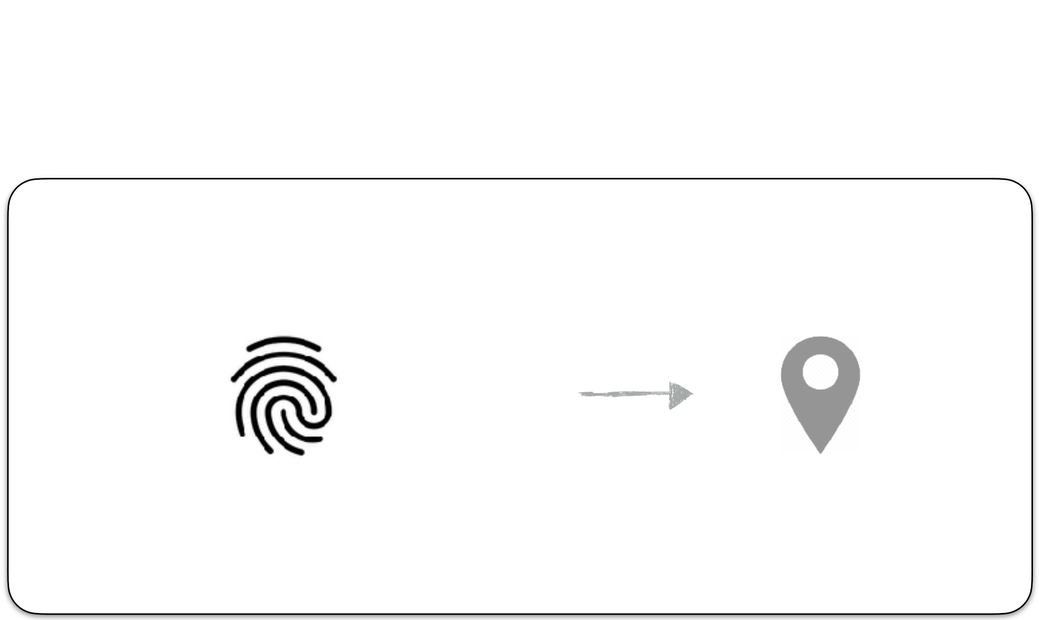


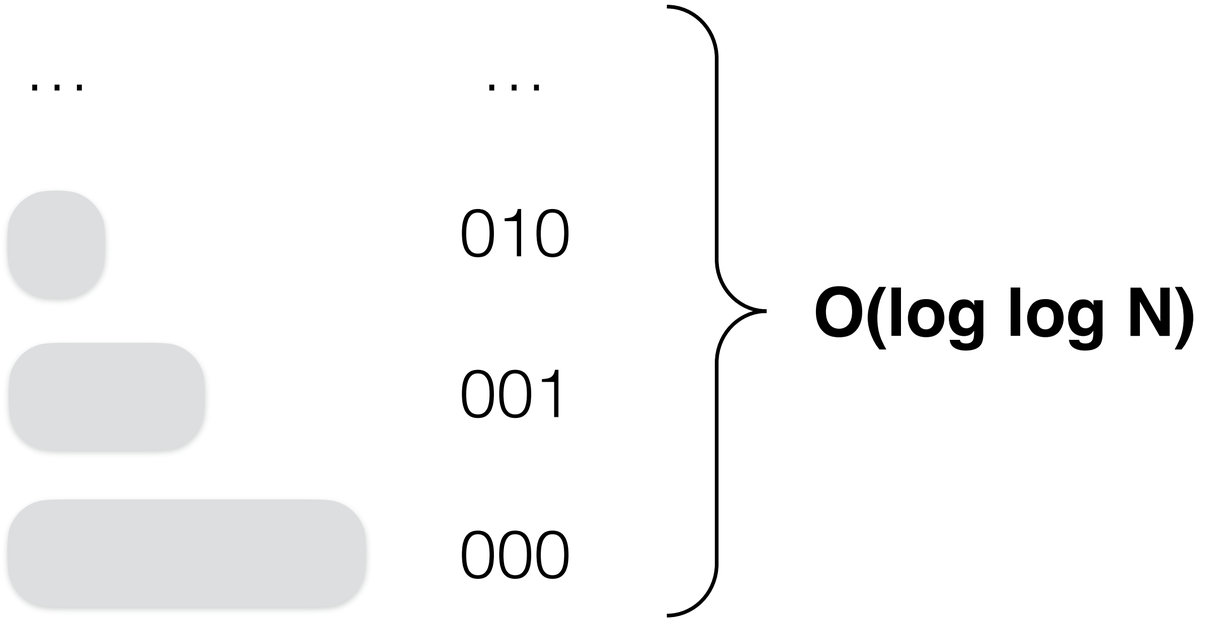
key **Level ID**

hash() =

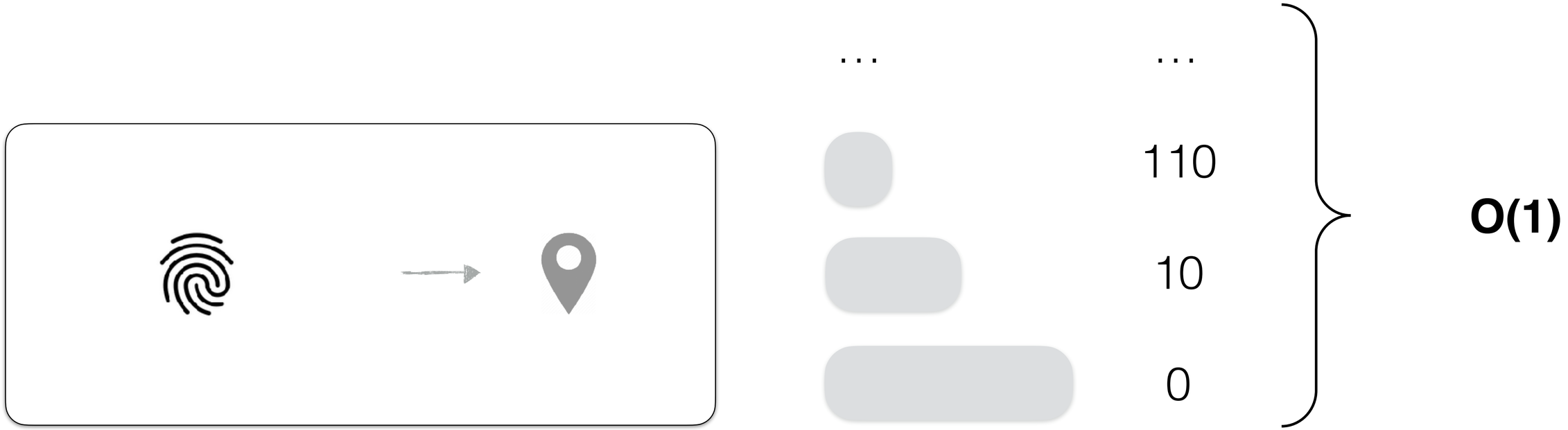
O(log log N)

Binary encoding

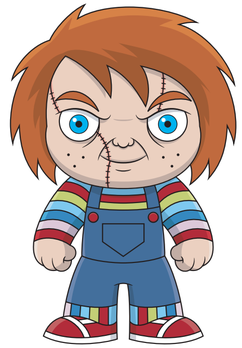




e.g., unary encoding

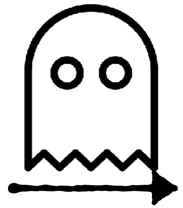


Monkey



Chucky

Bits / entry	**O(*M*)**	**≈**	**O(*M*)**
memory accesses	**O(*L* + *M*)**	**>**	**O(1)**

get I/O cost	$O(e^{-M} \cdot L)$		$O(e^{-M})$		
insert I/O cost	$O((R \cdot L)/B)$		$O((R+L)/B)$		$O((\log_2 \log_2 N/P)/B)$
transient space-amp	**≈2**		**≈1**		
filter mem. reads	$O(M + L)$		$O(1)$		

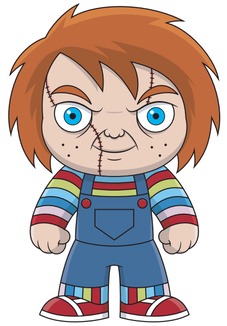
$L = \log_R N/P$

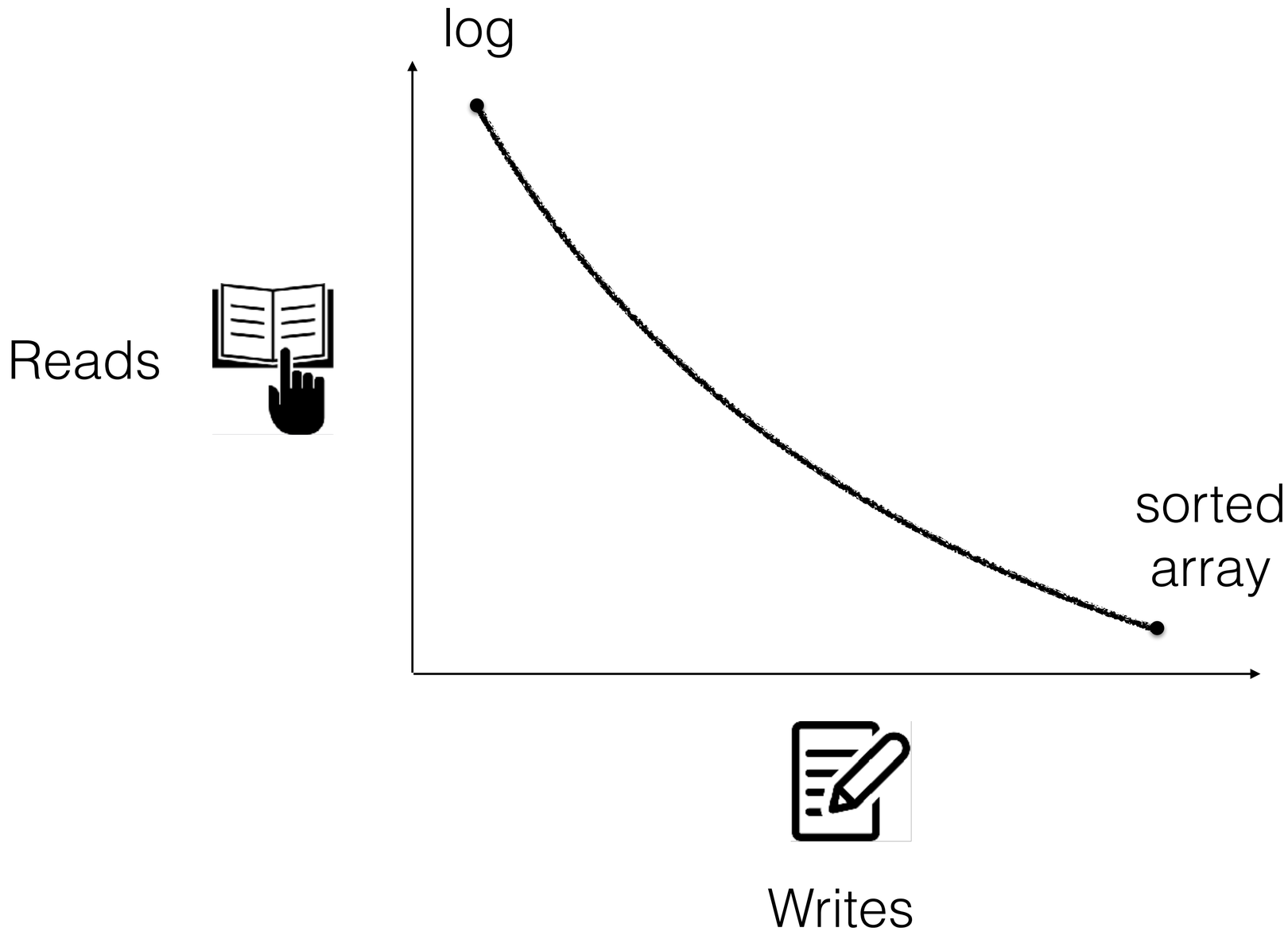
(Costs assuming leveling)

Our transactional storage engines
can now far better accommodate exponential data growth

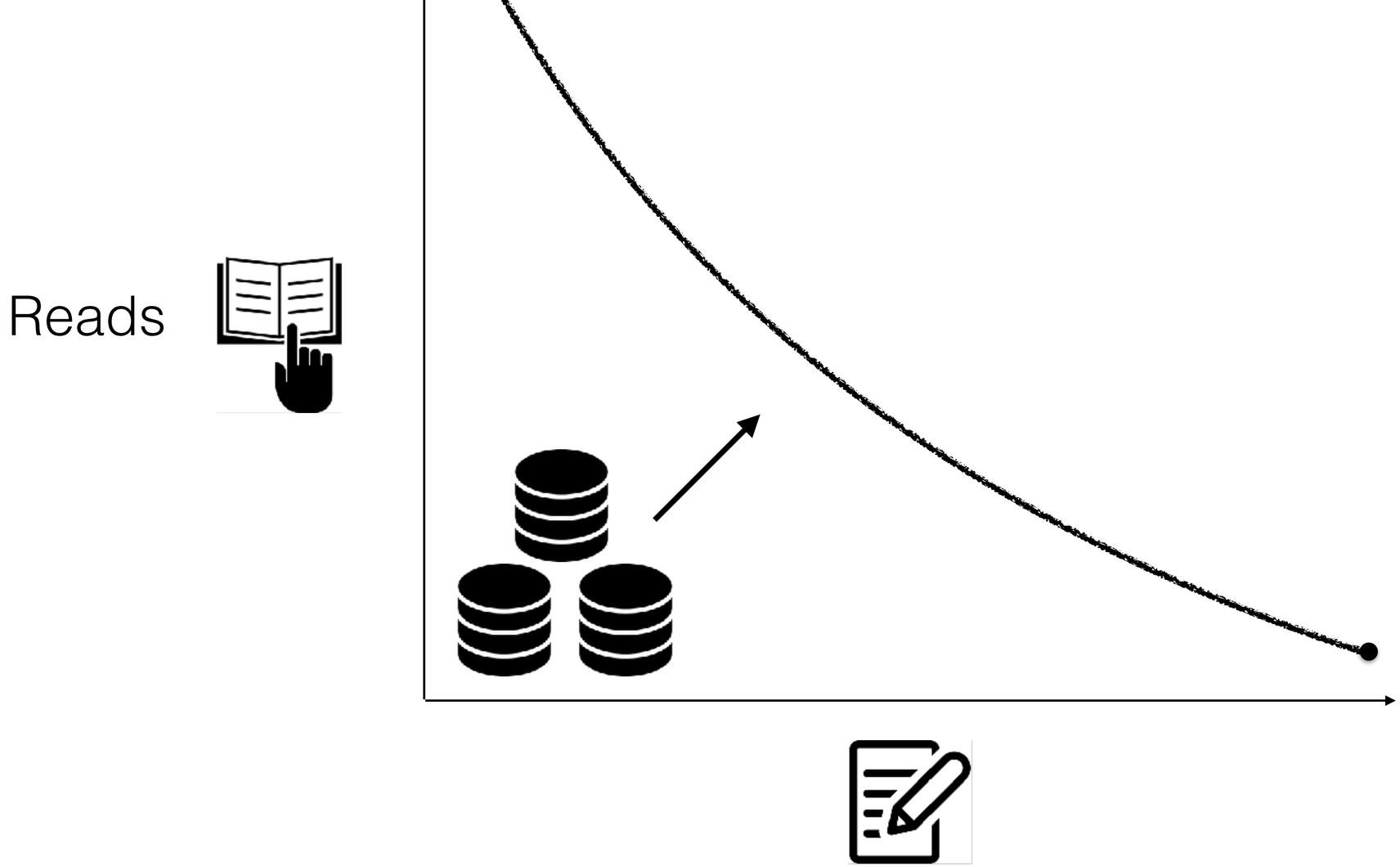






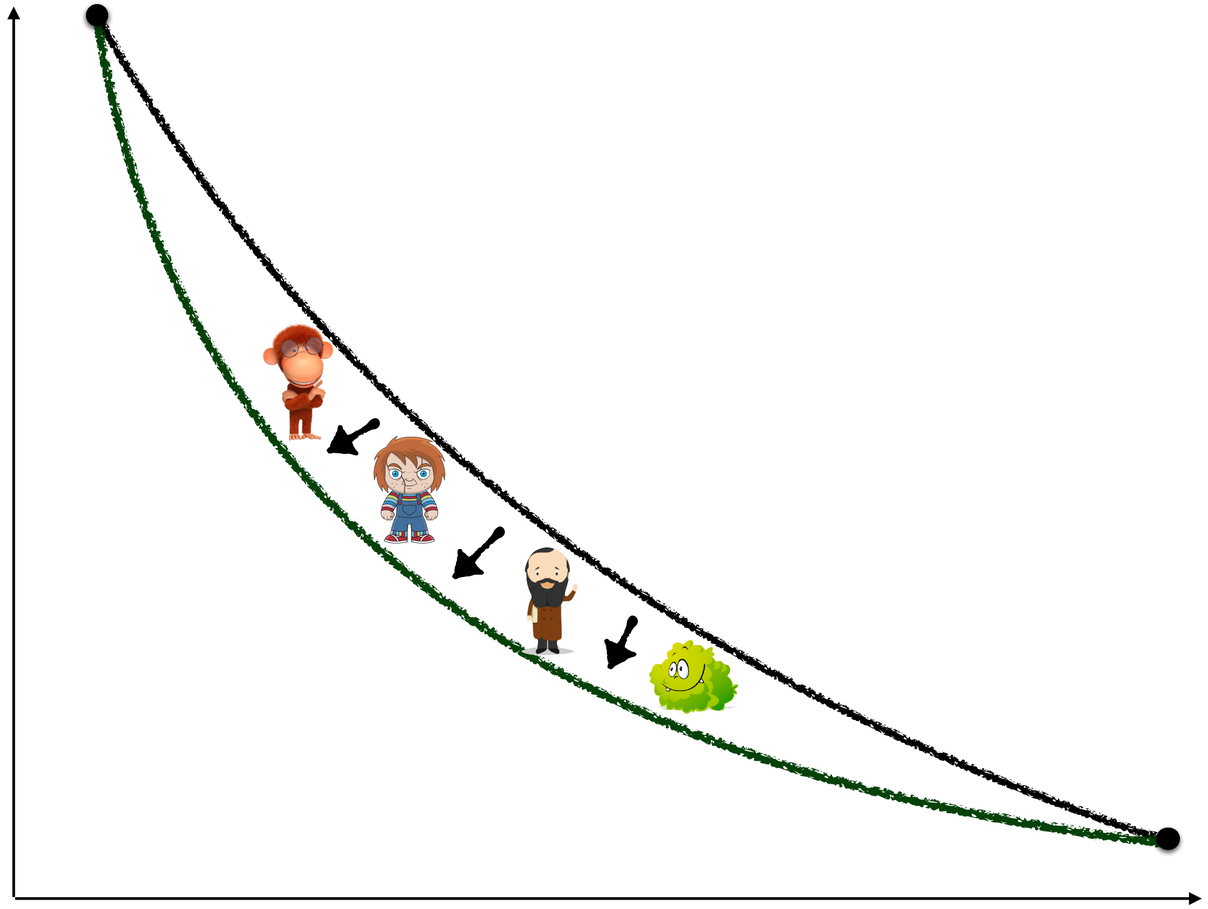

log
Reads
sorted
array
Writes

Reads



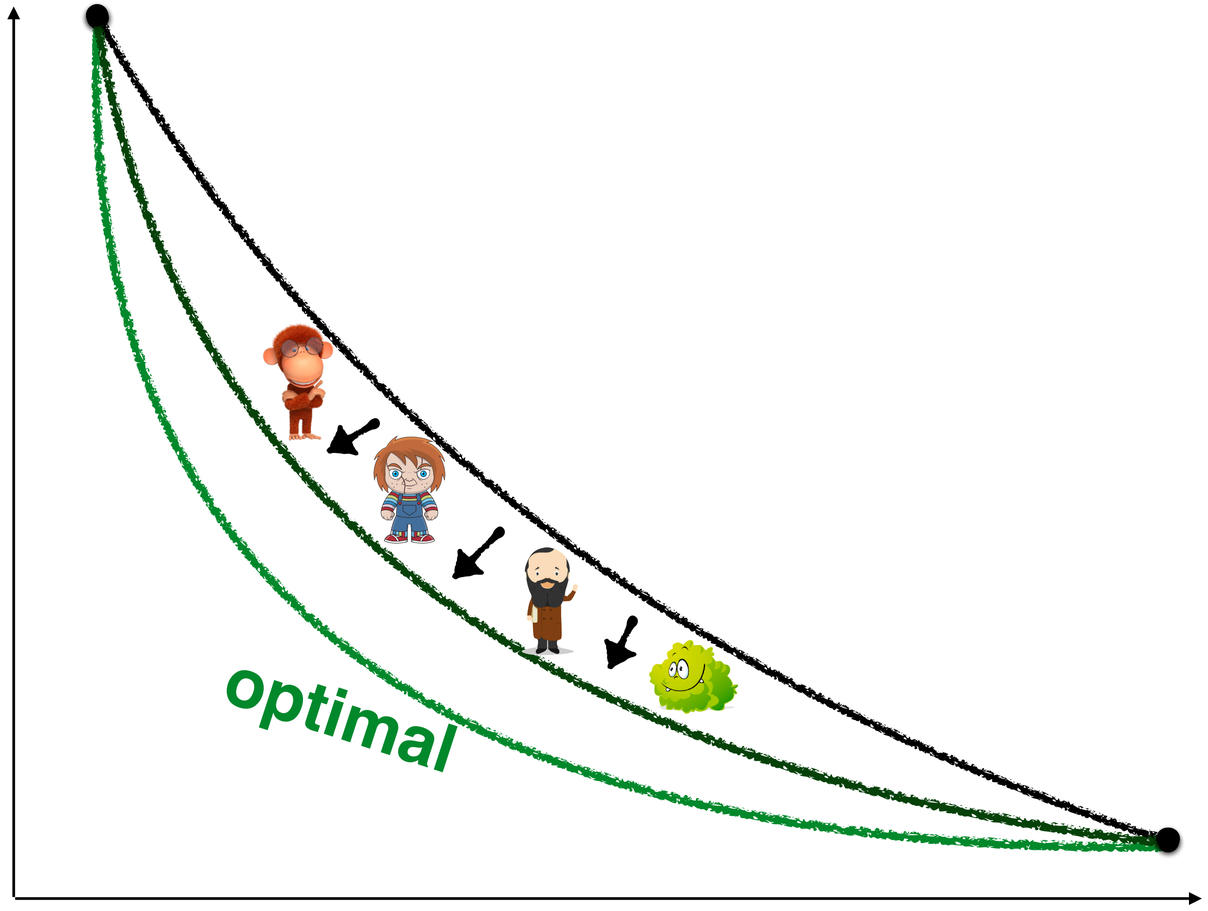
Writes

Reads

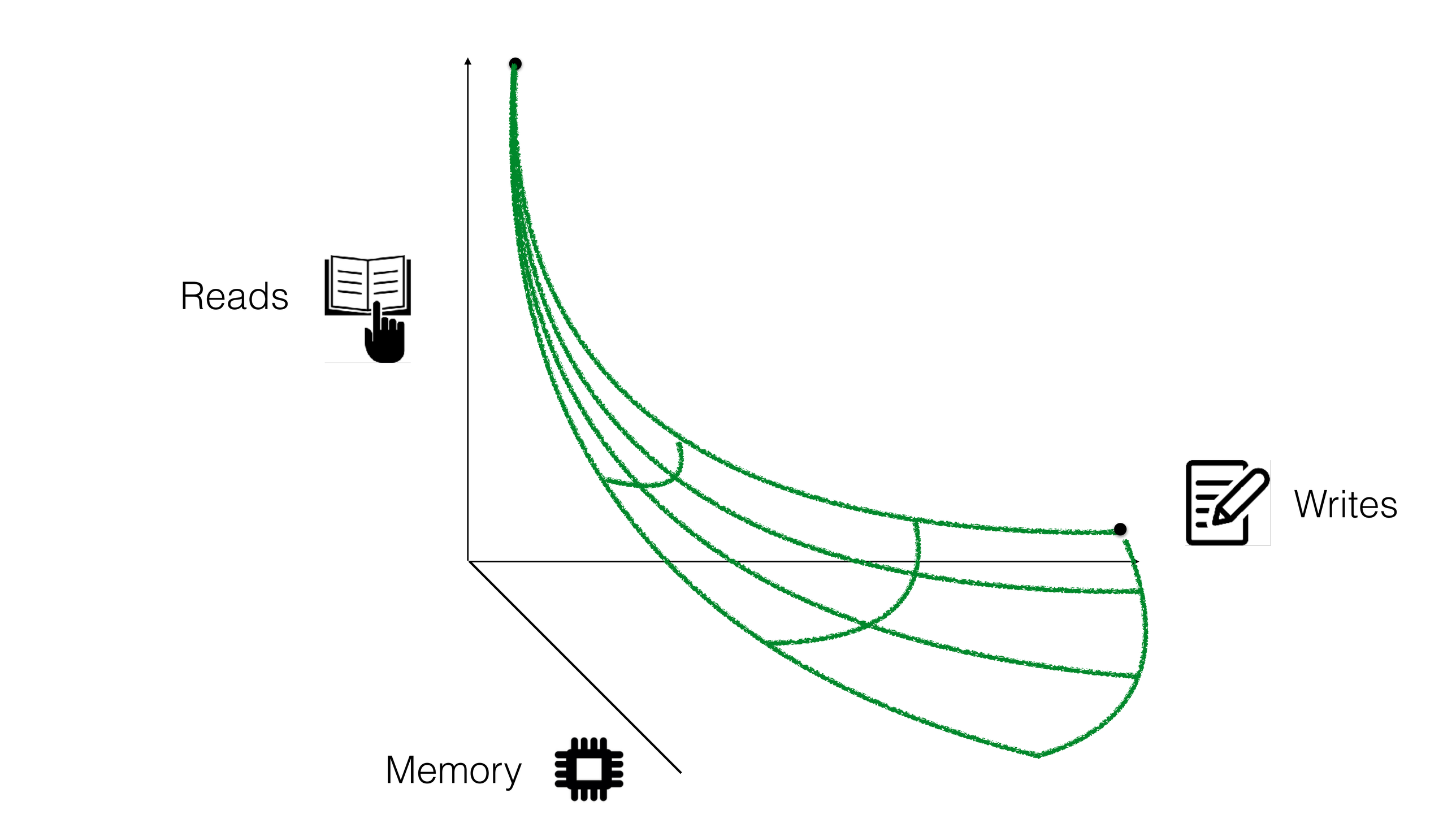


Writes

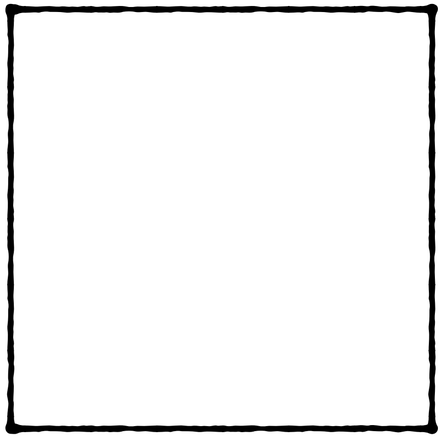
Reads



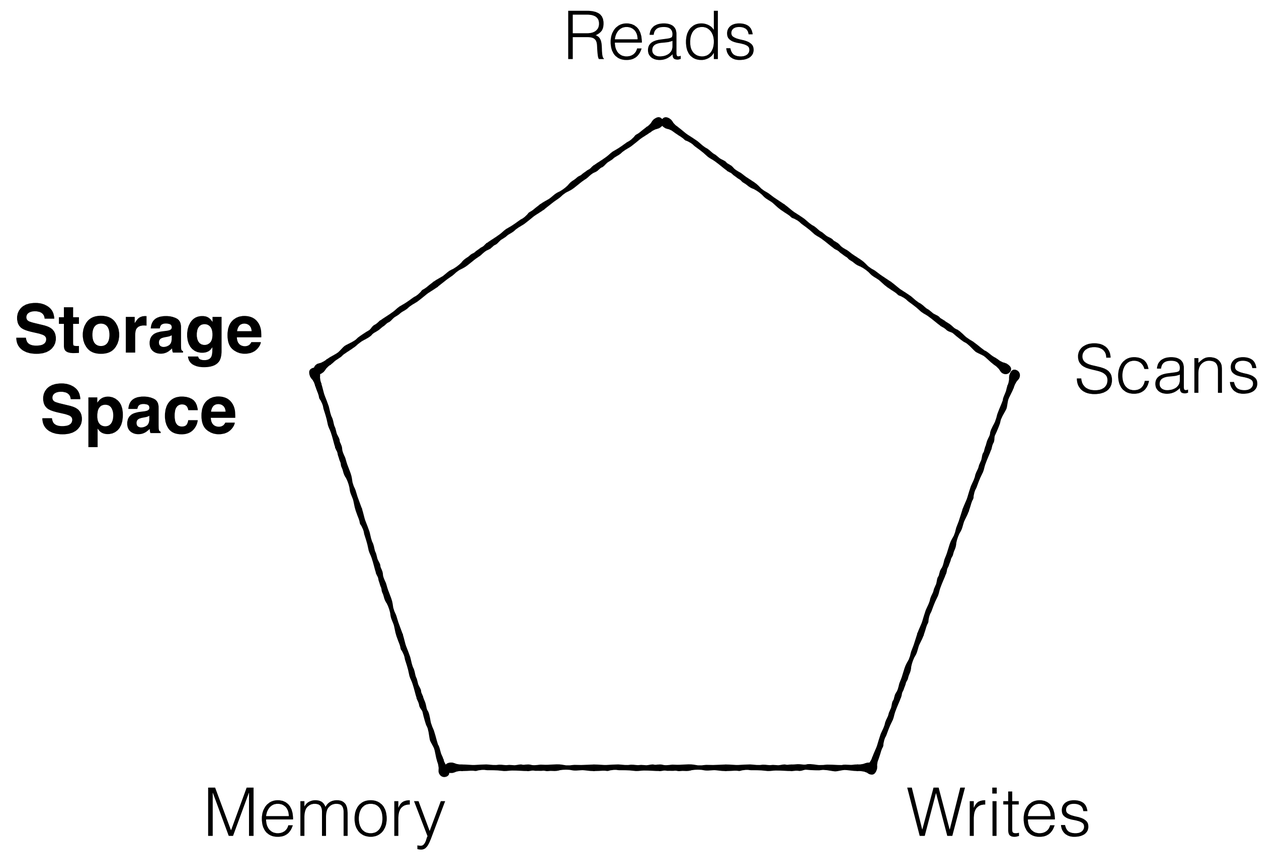
Writes

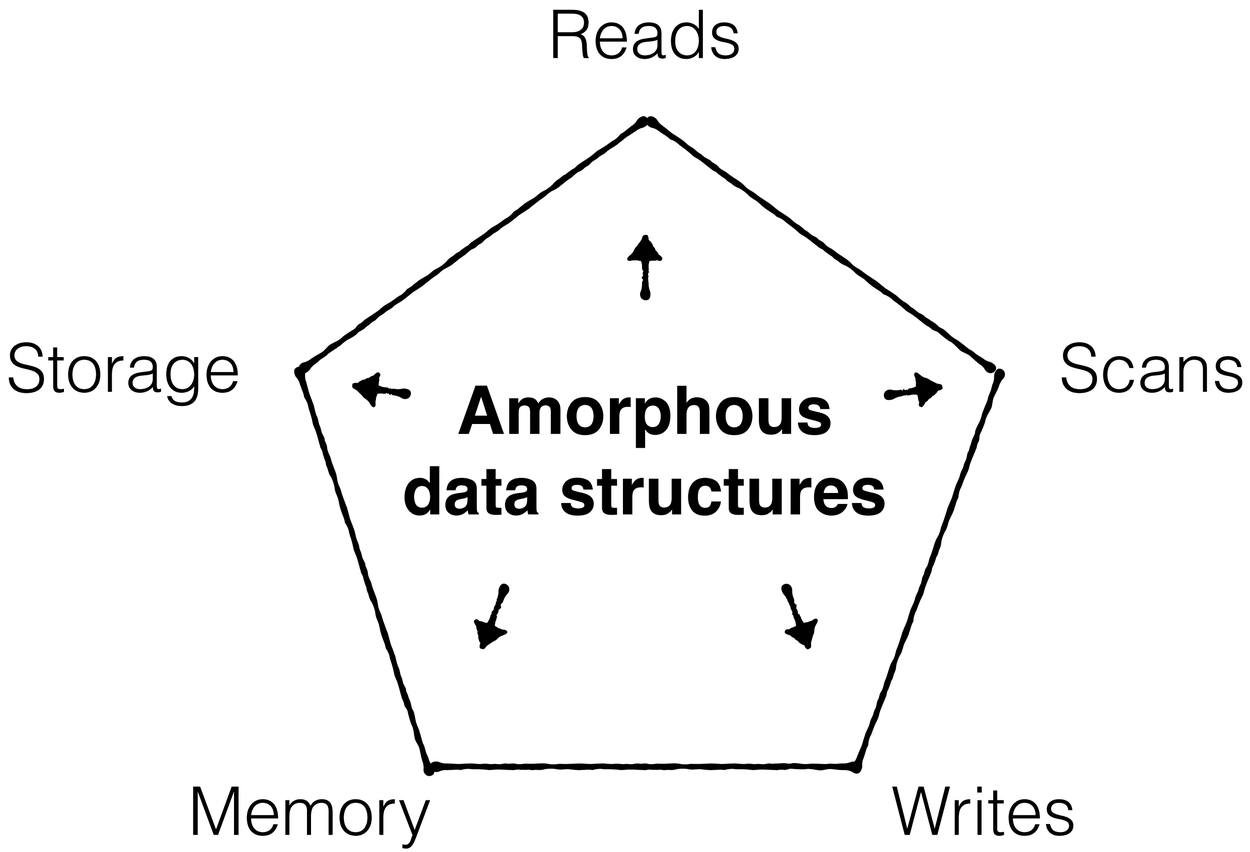

Reads
Writes
Memory

Reads **Range Scans**



Memory Writes


Reads
Storage Space
Scans
Memory
Writes


Reads
Storage
Scans
Amorphous
data structures
Memory
Writes

Thanks