

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

Niv Dayan - Feb 7, 2023

We will start at 2:10 pm





Instr **Results Sharing**

2/7/23

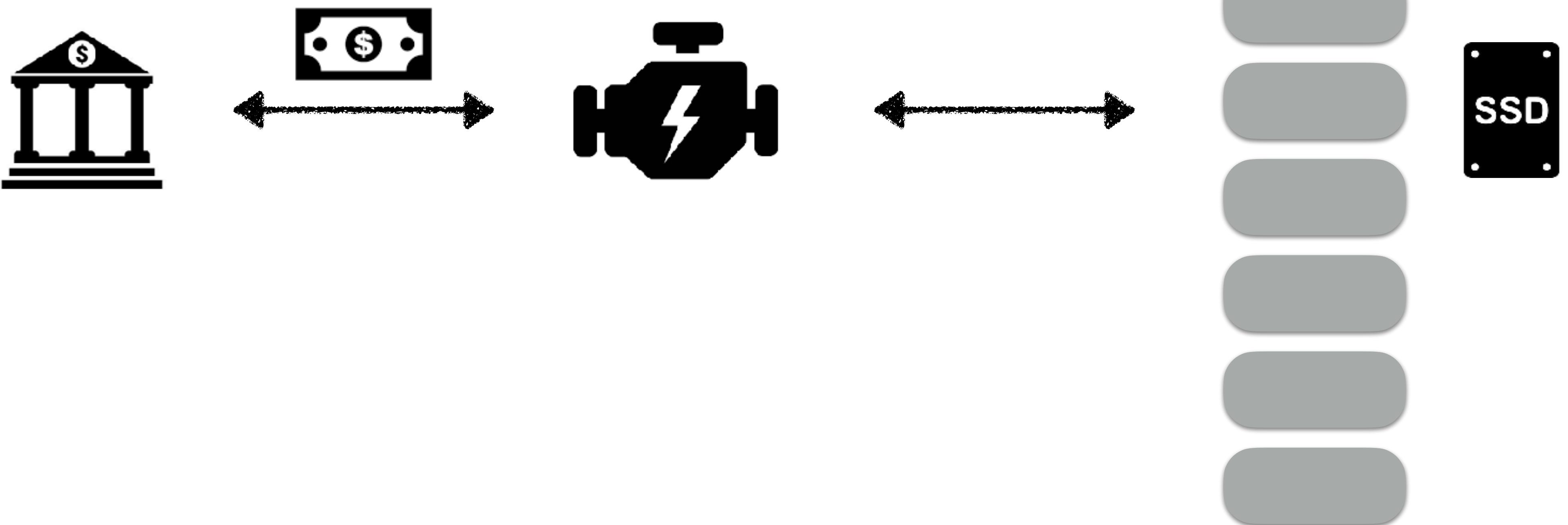
Hi Class, This post will be a forum to allow groups to share information on the experimental results they're gettin



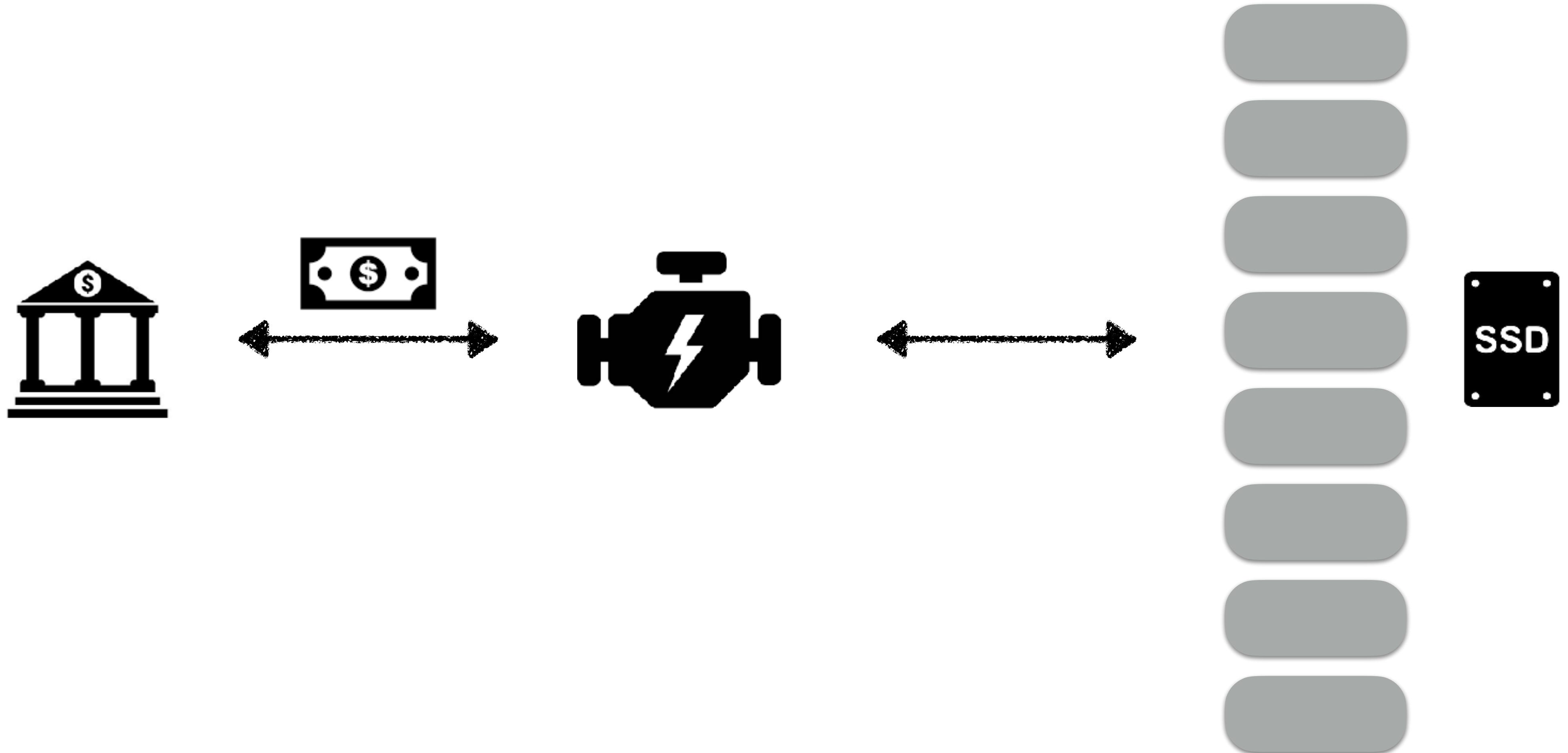
- **Added to reading list**

Many DB operations are:

data



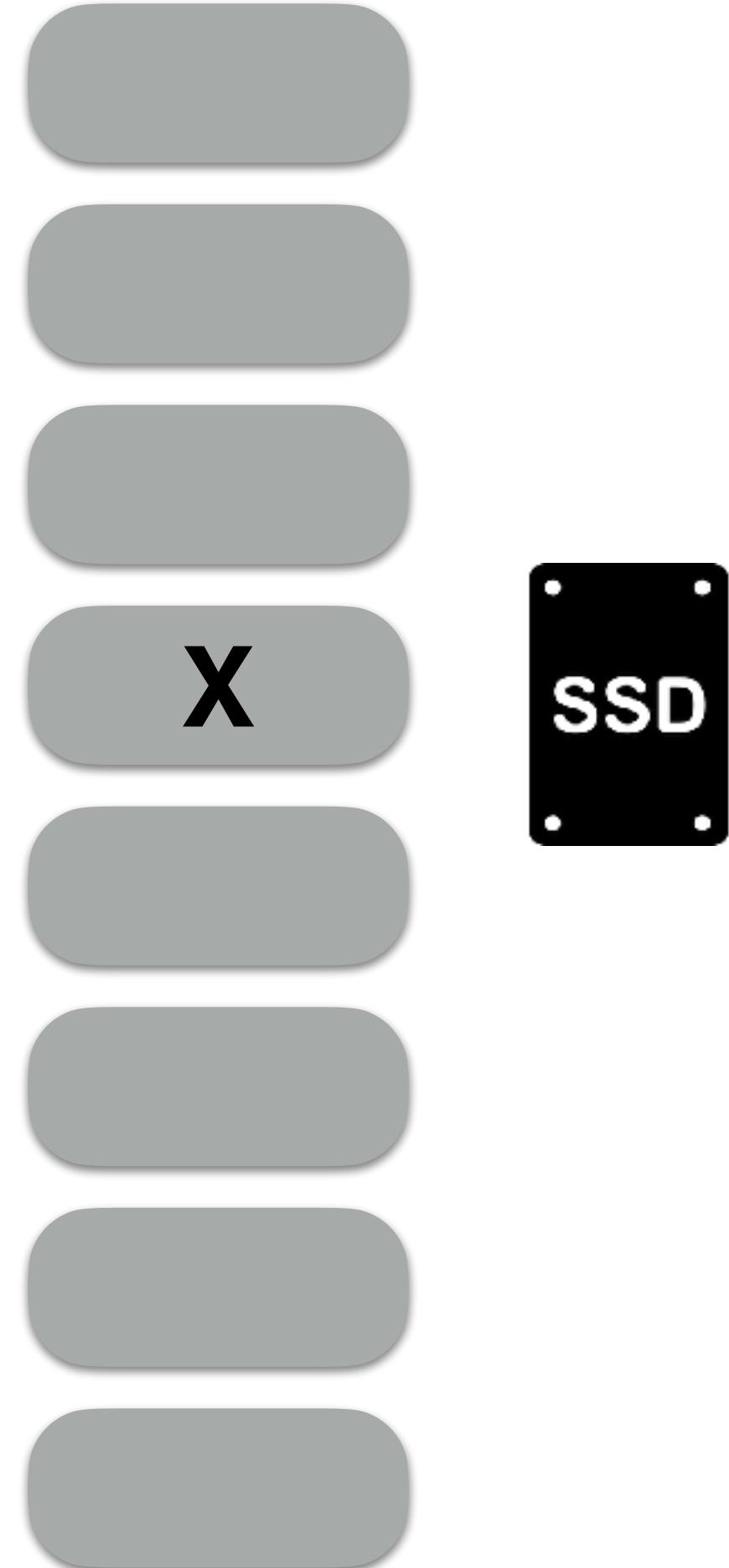
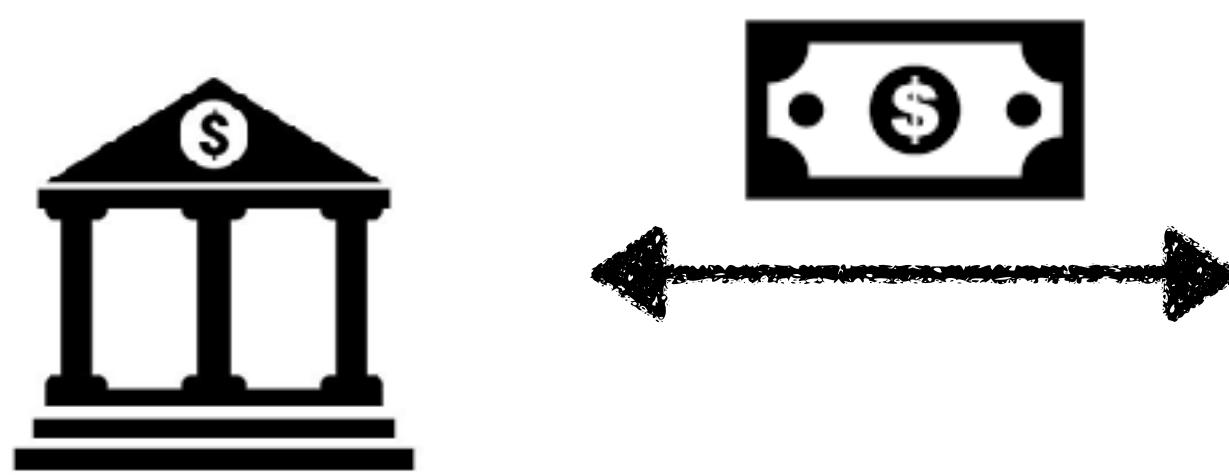
Many DB operations are: **selective** data



Many DB operations are:

**selective
small data items**

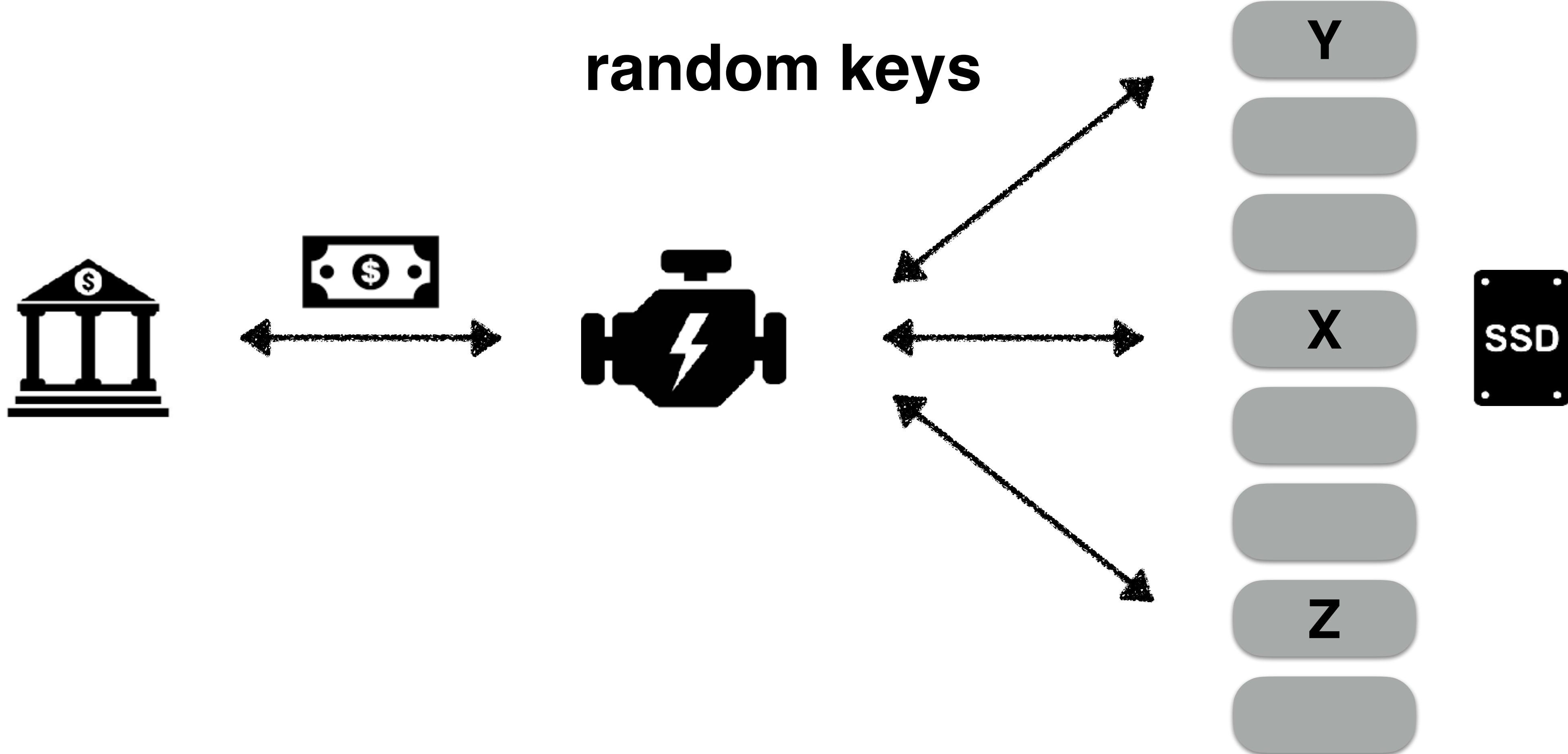
data



Many DB operations are:

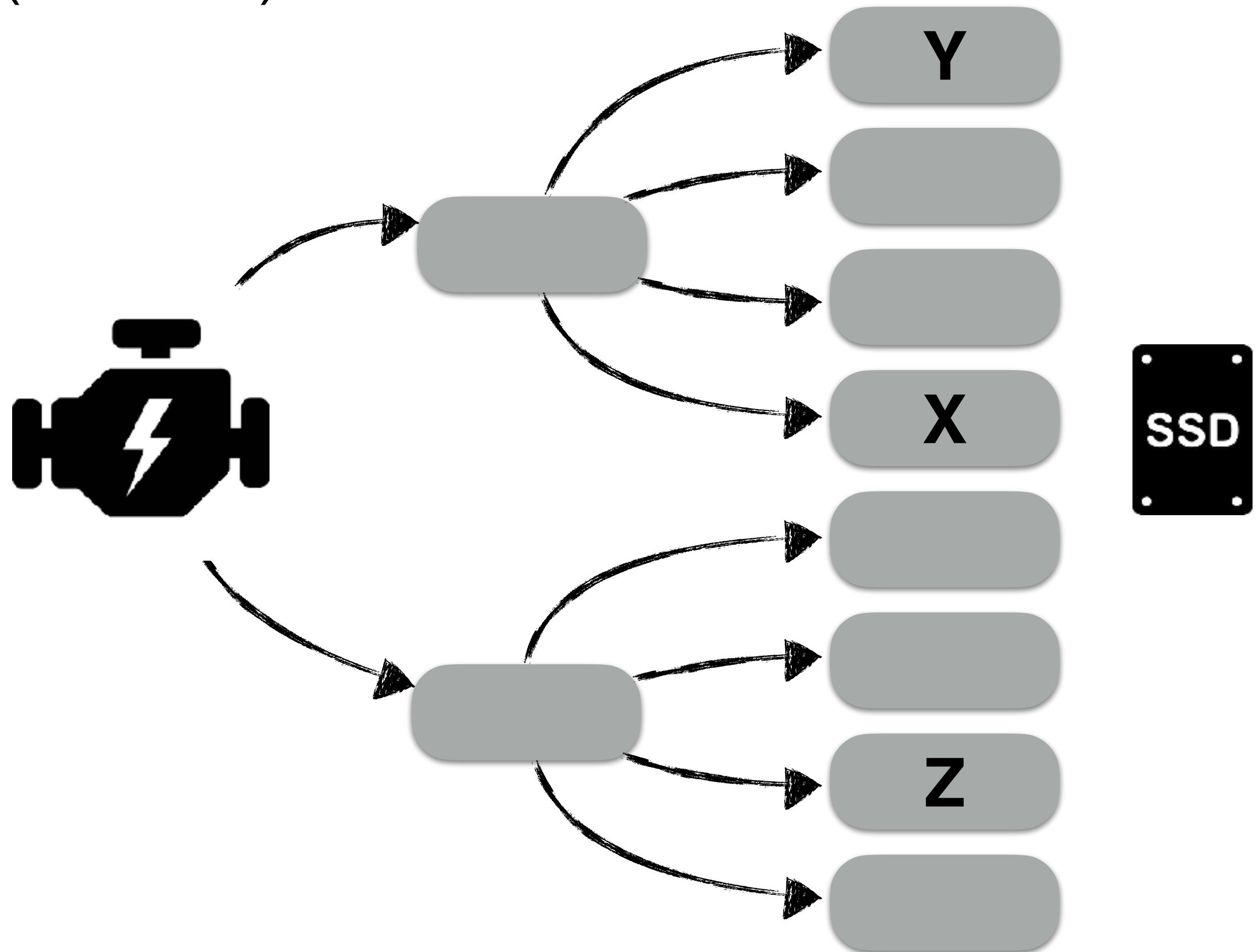
**selective
small data items
random keys**

data

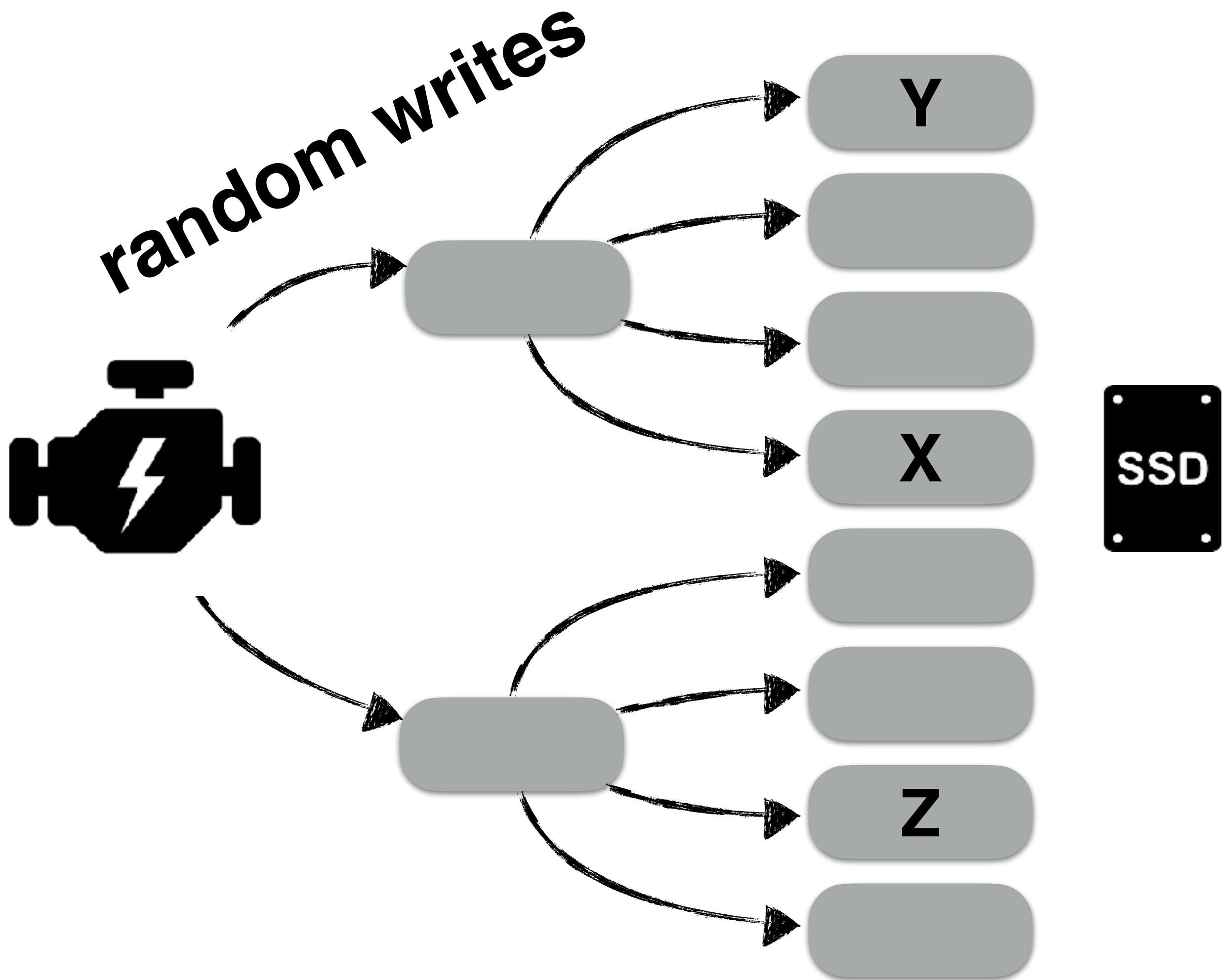


B-tree (1970's)

data



data



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

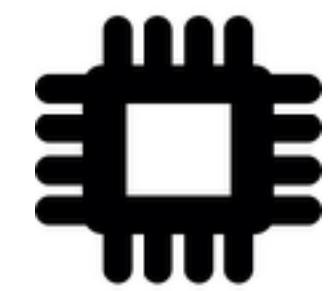
The Log-Structured Merge-Tree

1996 - Patrick O'Neil



LSM-Tree





buffer

1 3 6

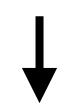


merge-sort

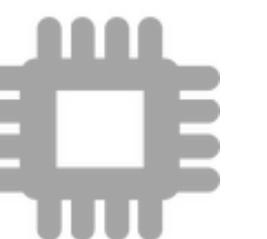
2 4 5

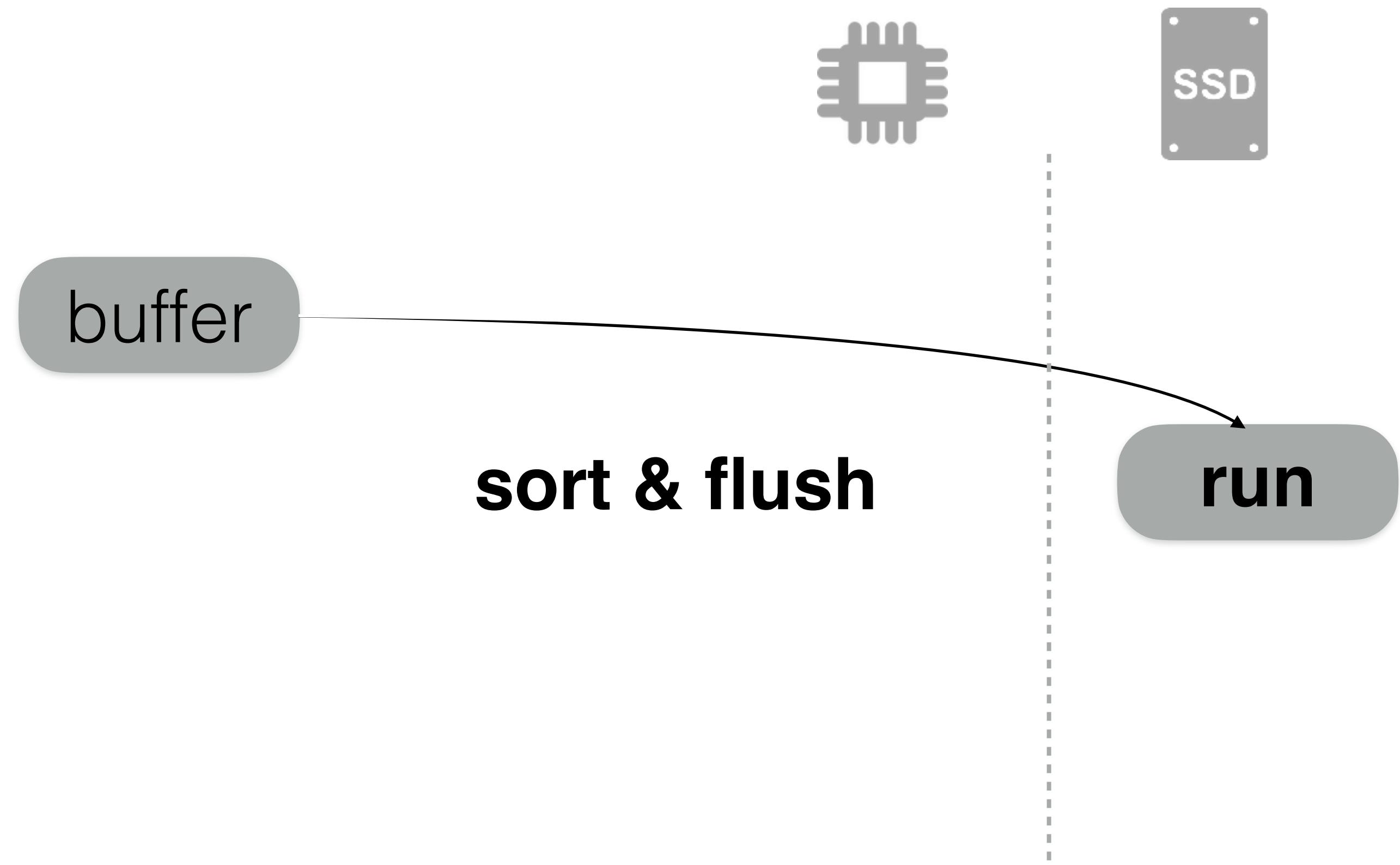
1 2 3 4 5 6

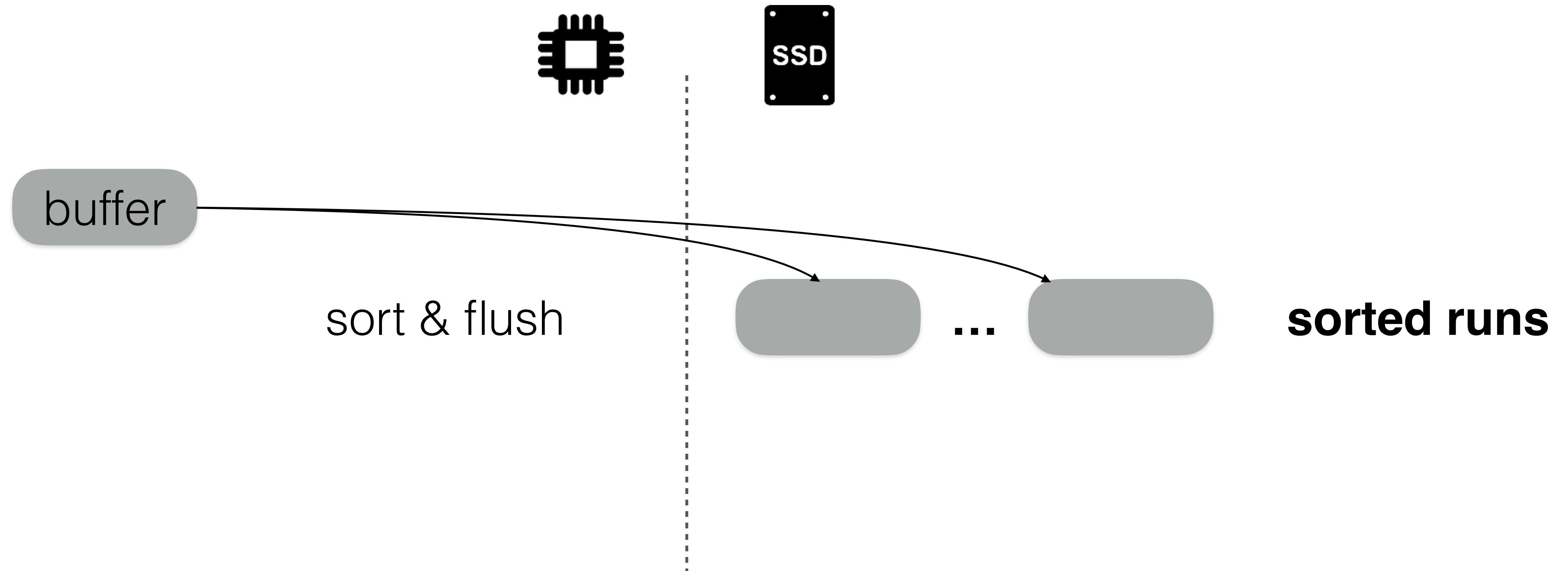
Inserts/updates/deletes
of key-value pairs

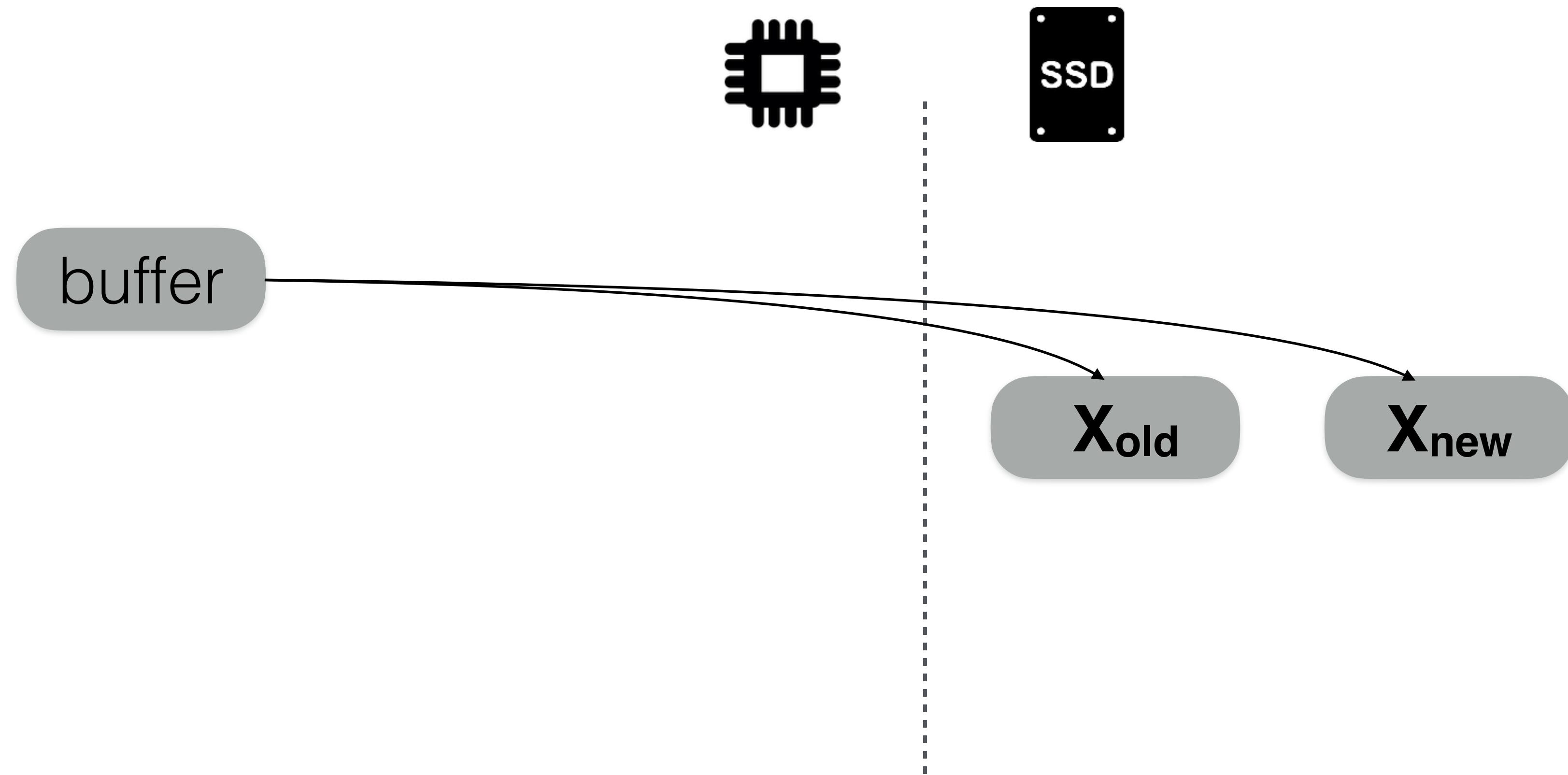


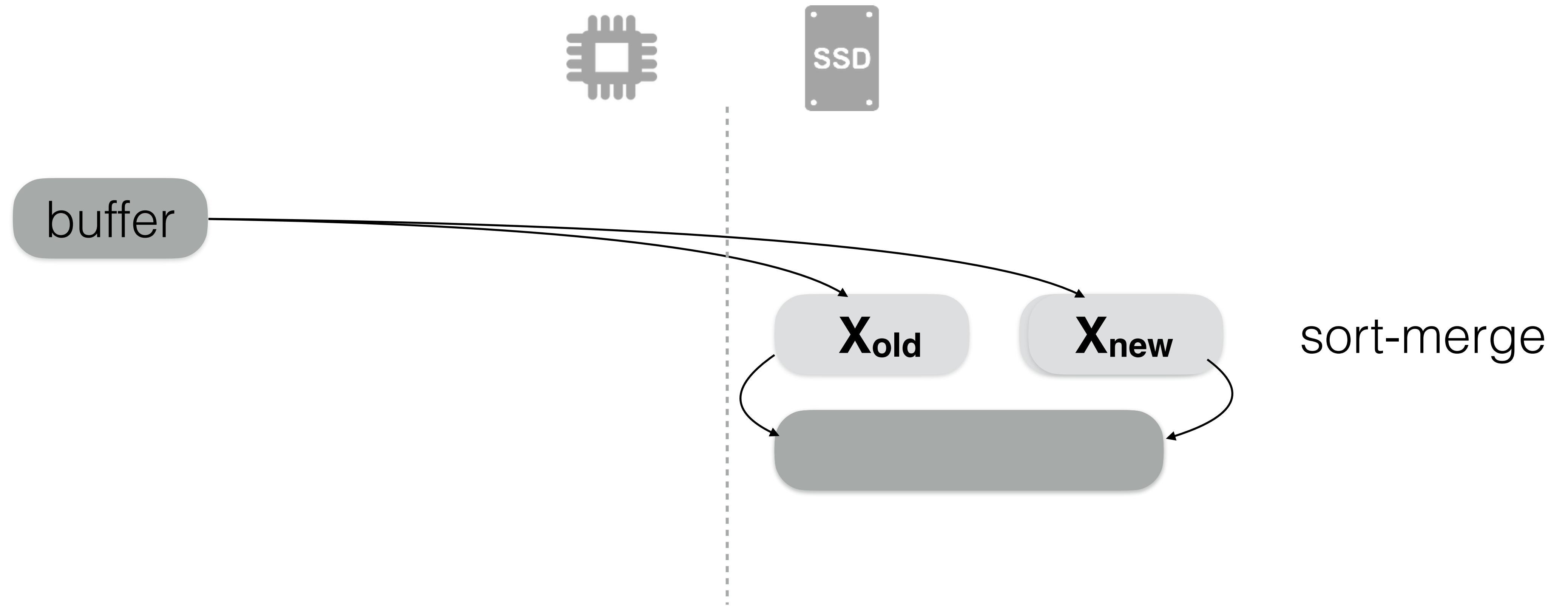
buffer

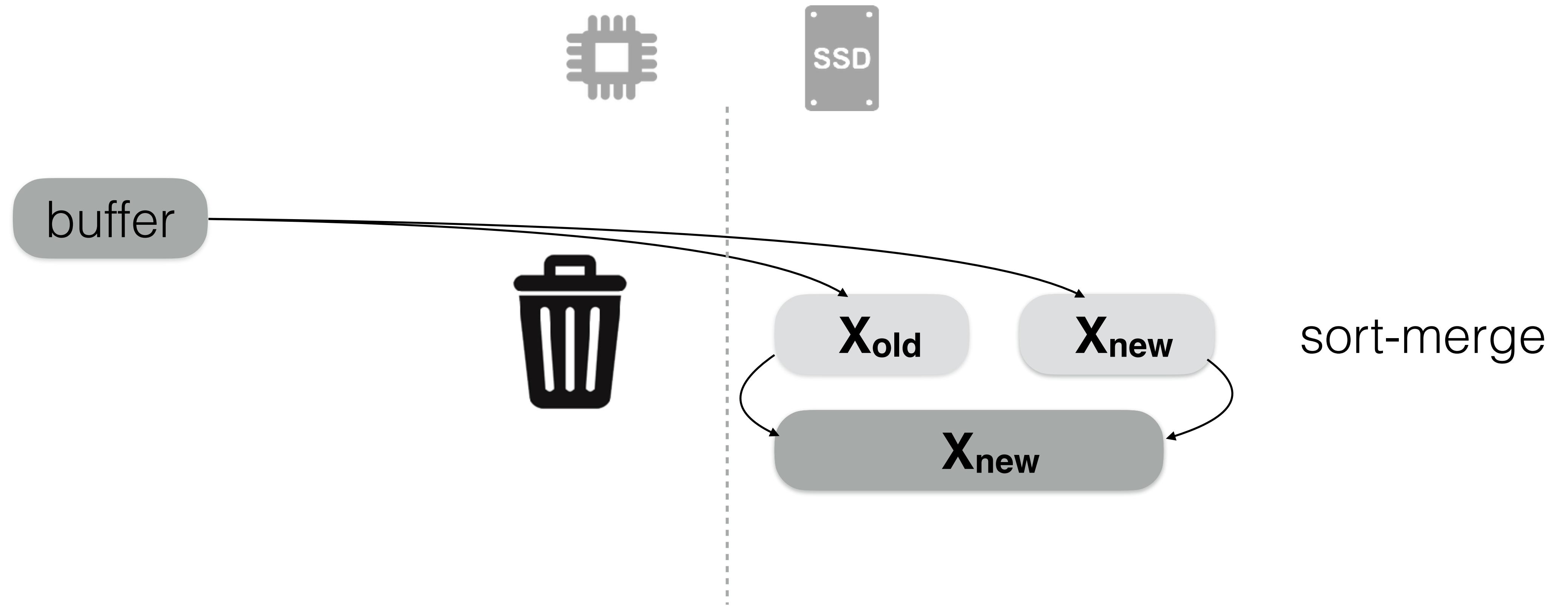




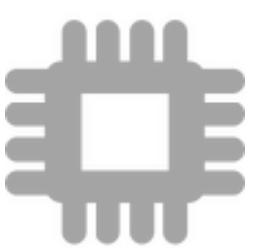








buffer



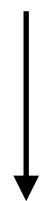
level 1 →

level 2 →

level 3 →

exponentially increasing capacities

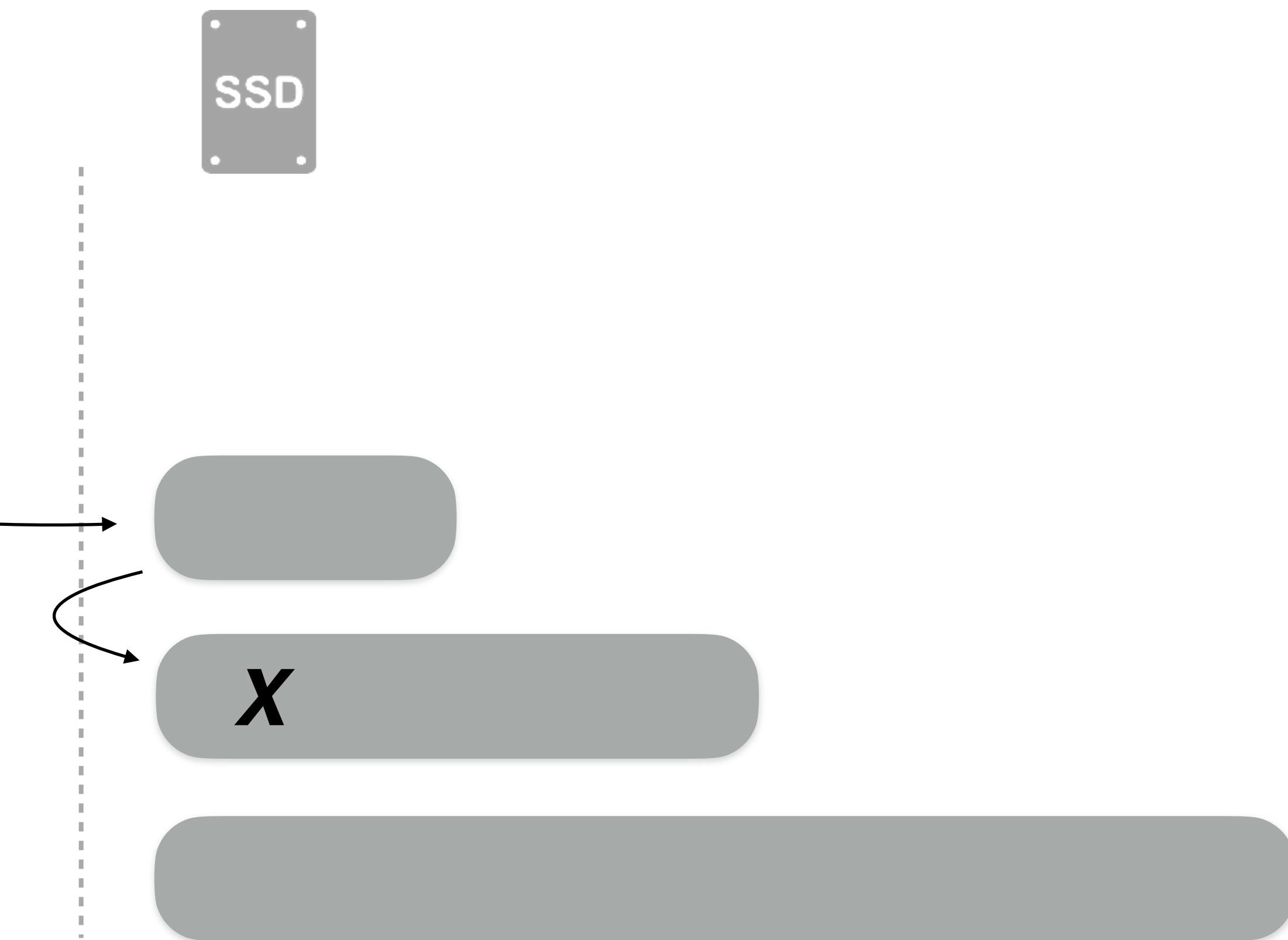
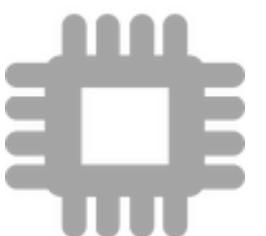
get(X)



buffer



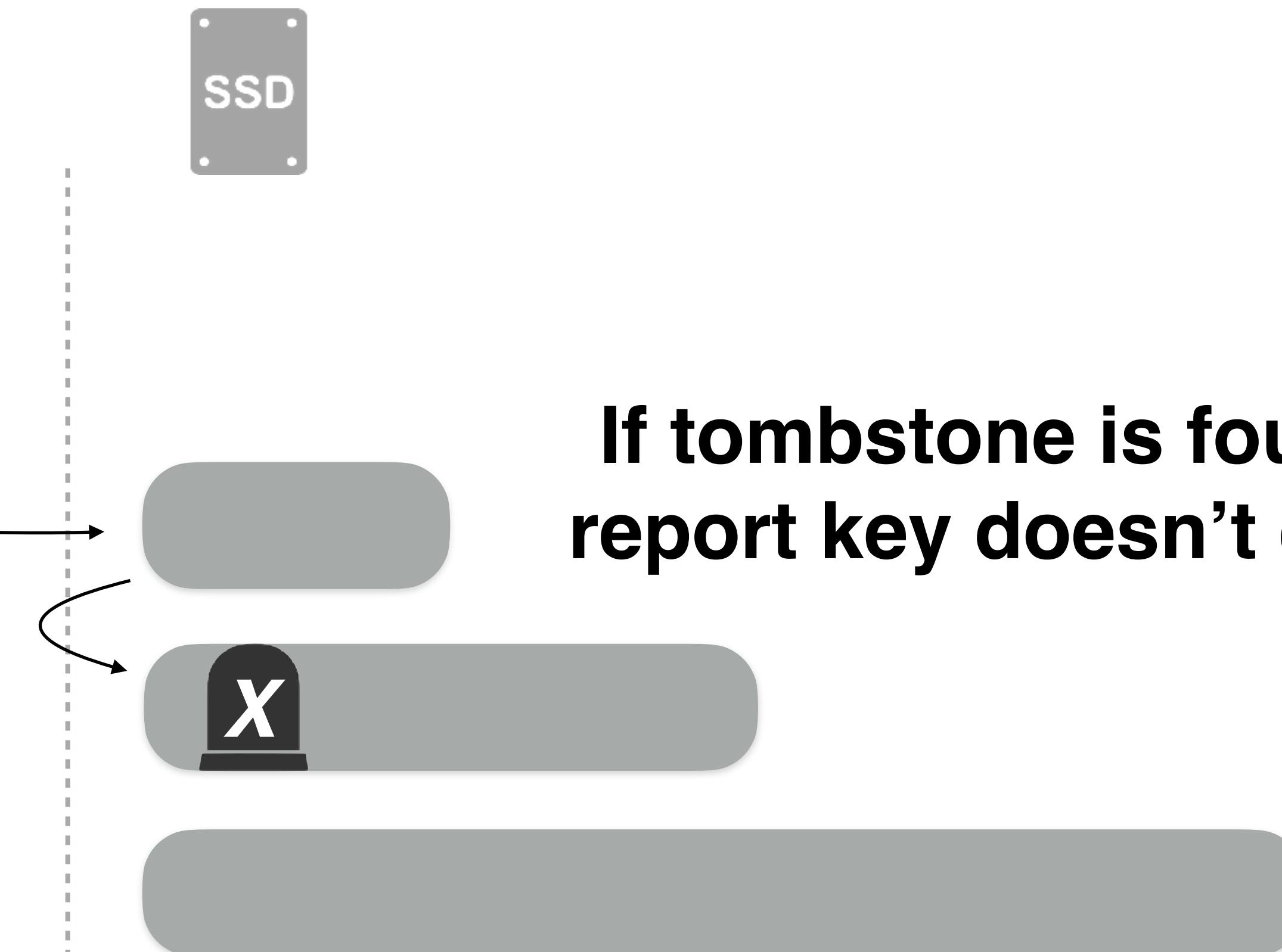
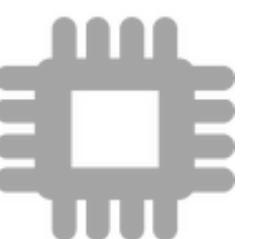
**newest to
oldest**



get(X)

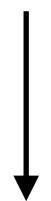


buffer

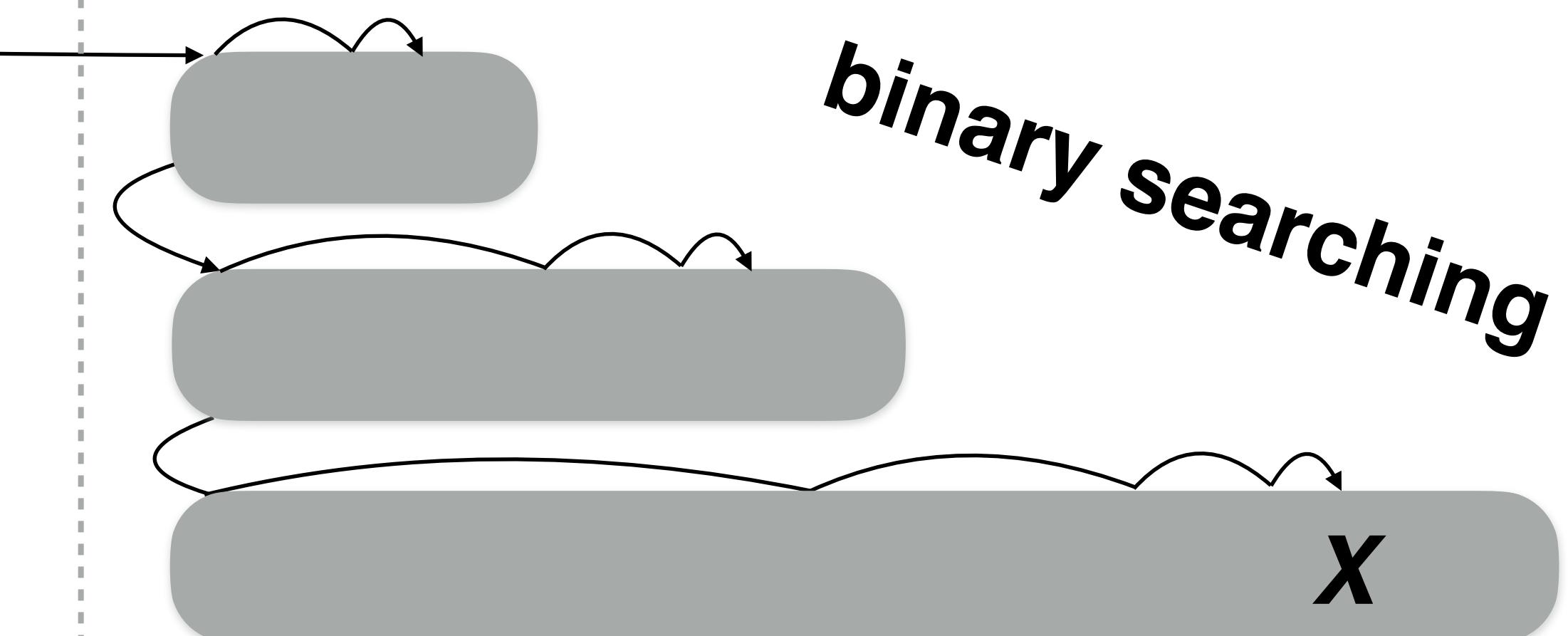
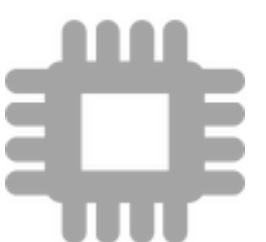


**If tombstone is found,
report key doesn't exist**

$\text{get}(X)$



buffer

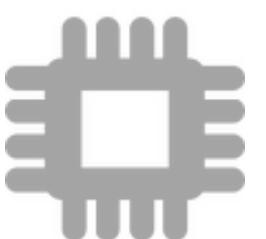


scan(X, Y)



buffer

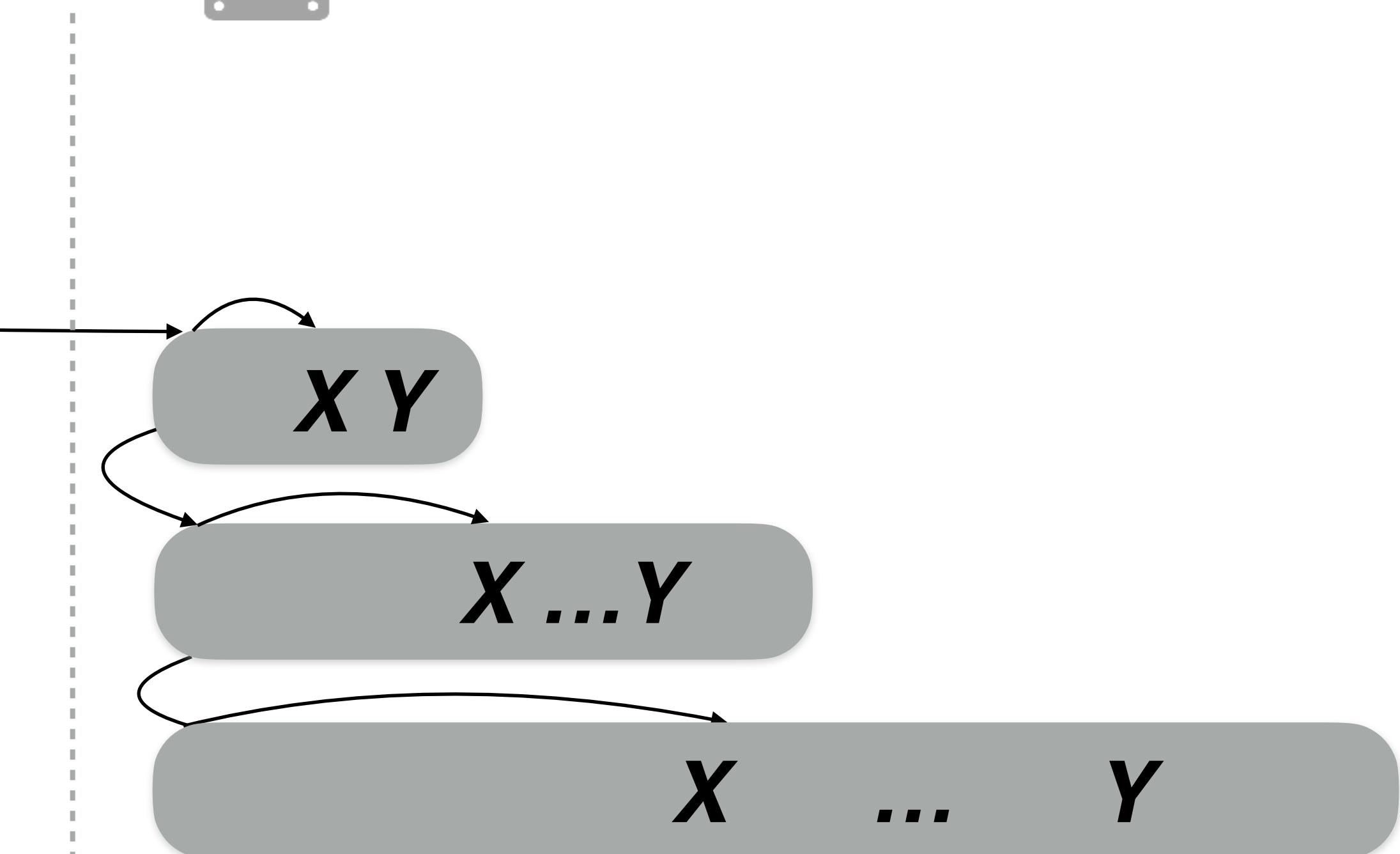
Find key range start



$X Y$

$X \dots Y$

$X \dots Y$

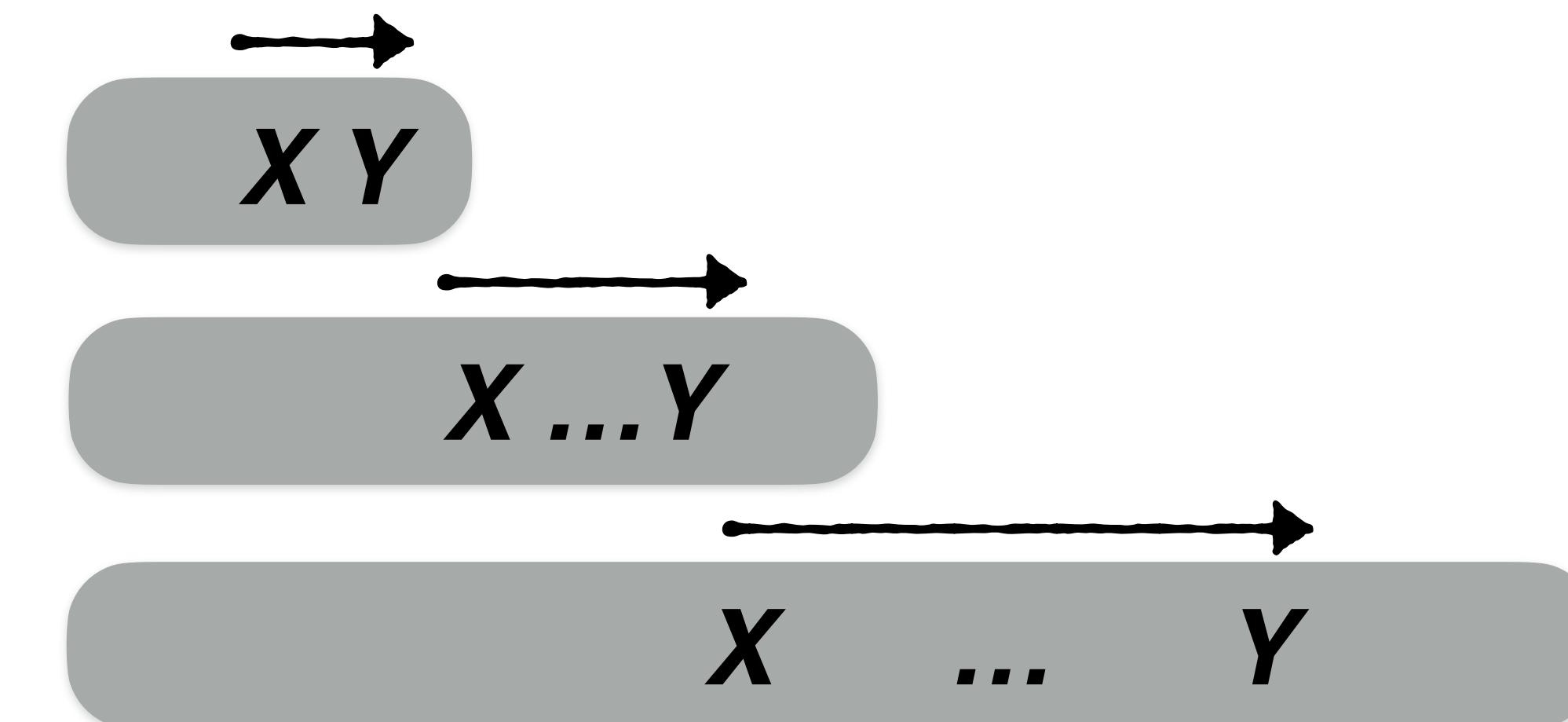
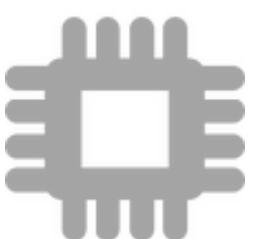


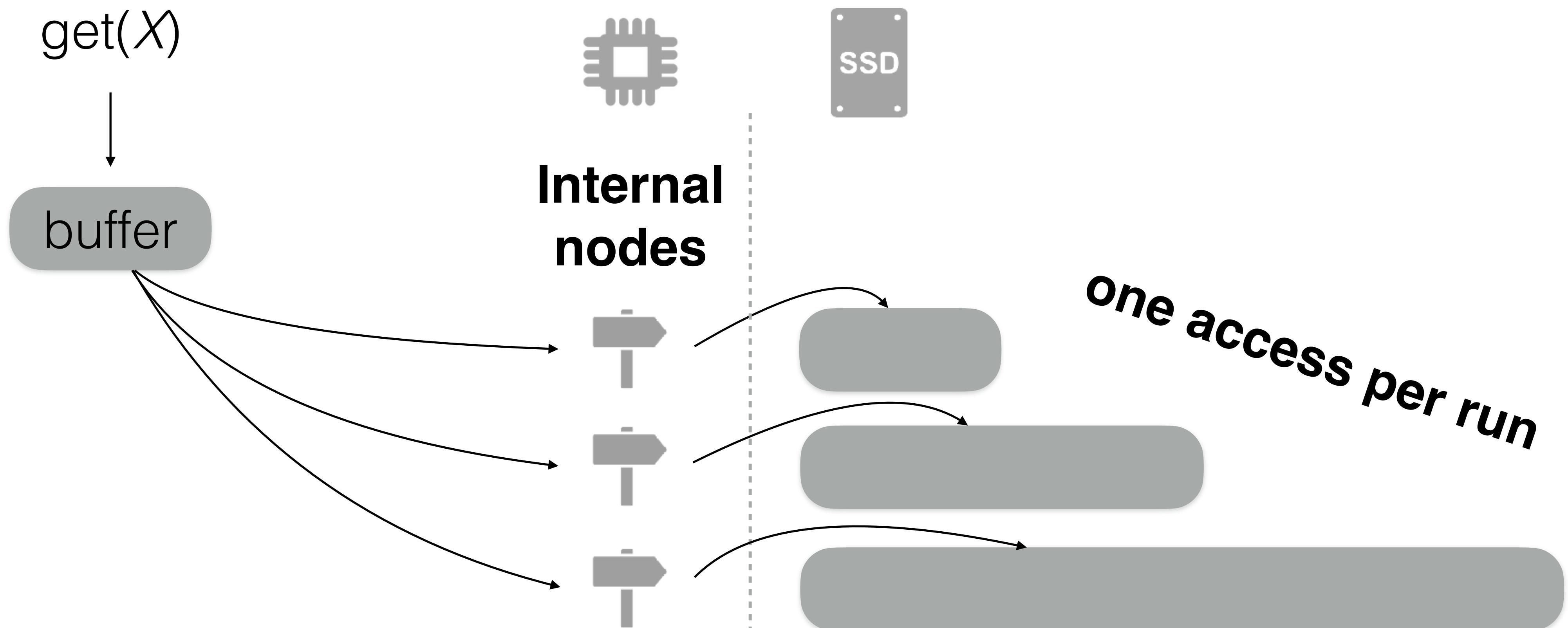
scan(X, Y)

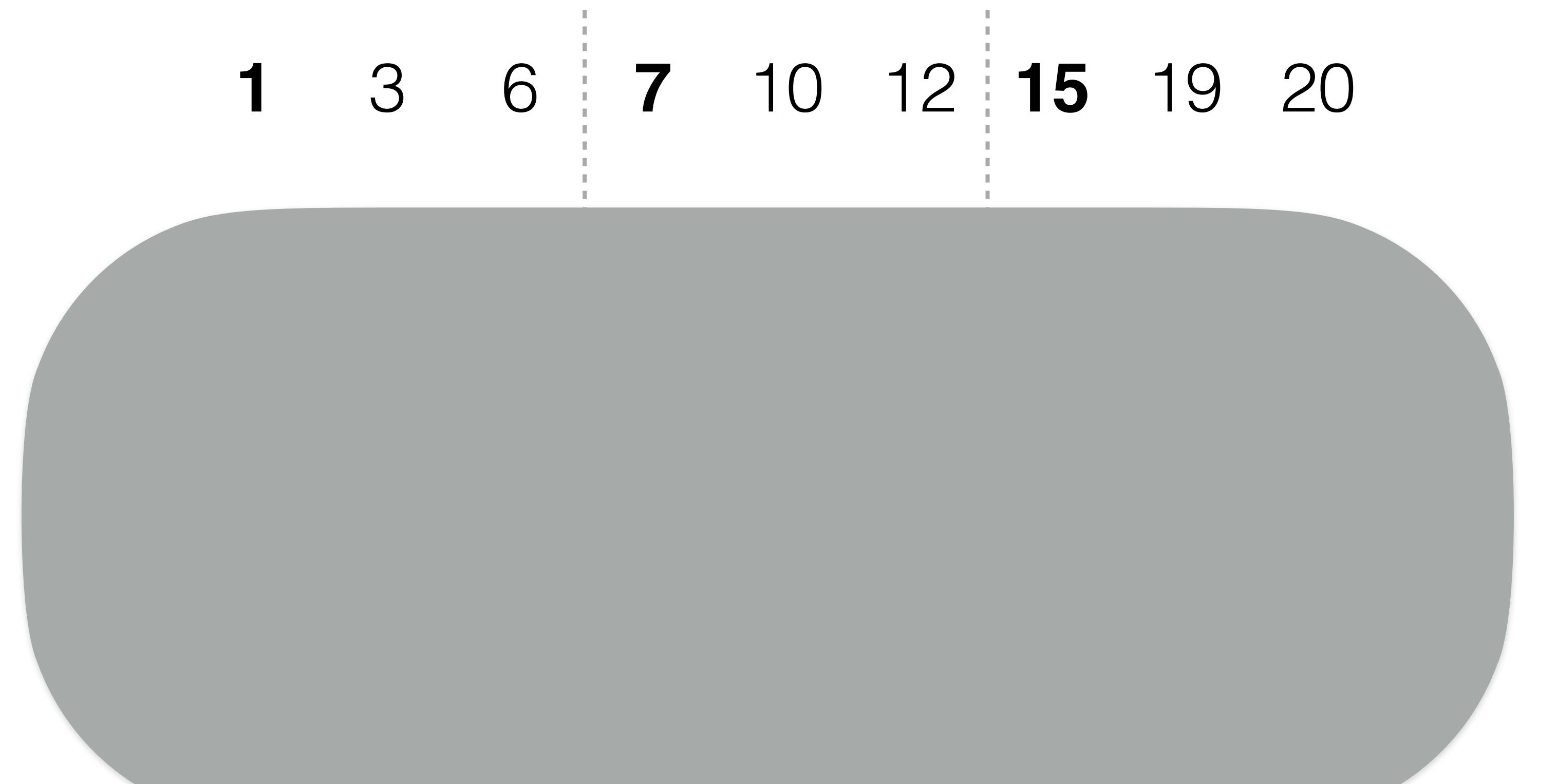


buffer

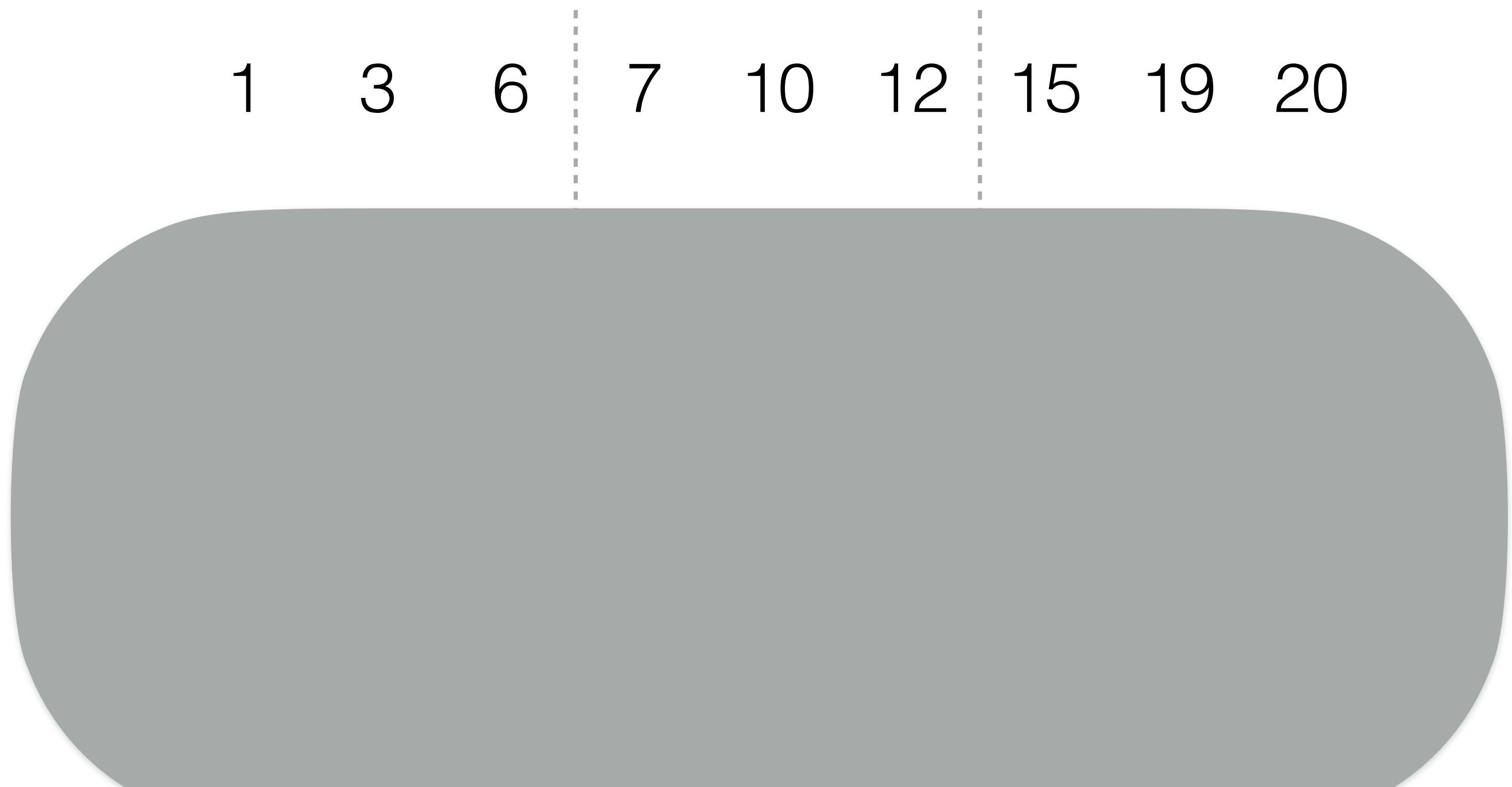
Scan & sort merge







binary search



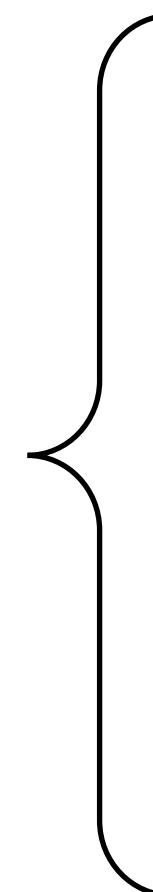
one storage access

1 7 15

1 3 6 7 10 12 15 19 20



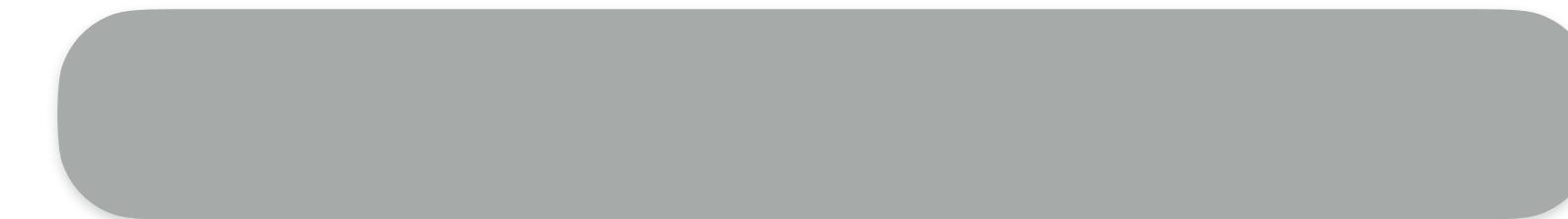
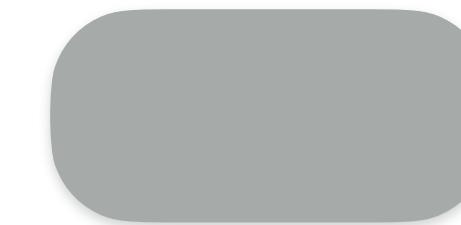
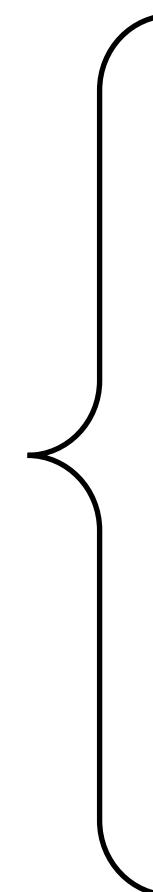
levels $L = \log_R(N/P)$



$$\# \text{ levels } L = \log_R(N/P)$$



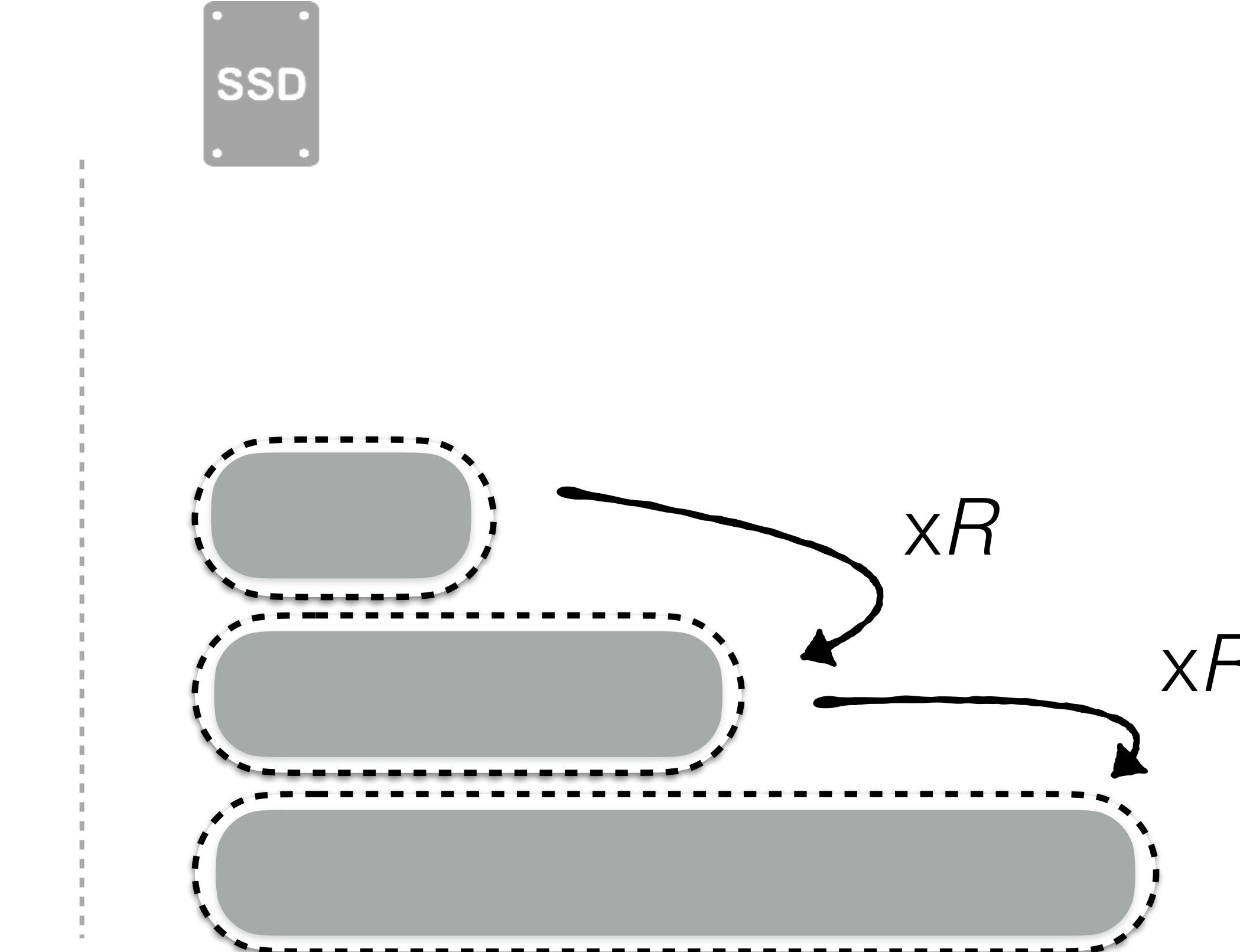
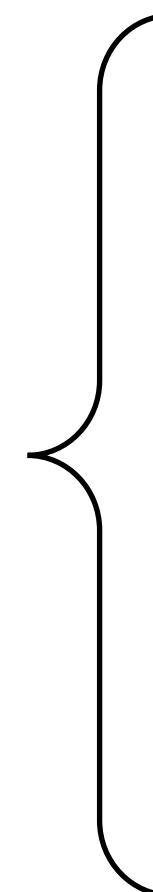
data size

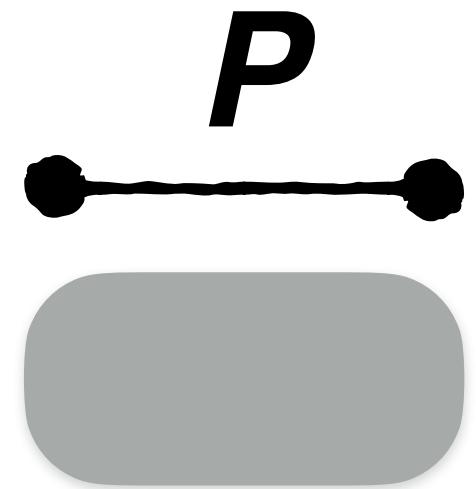


$$\# \text{ levels } L = \log_{\textcolor{red}{R}}(N/P)$$



size ratio

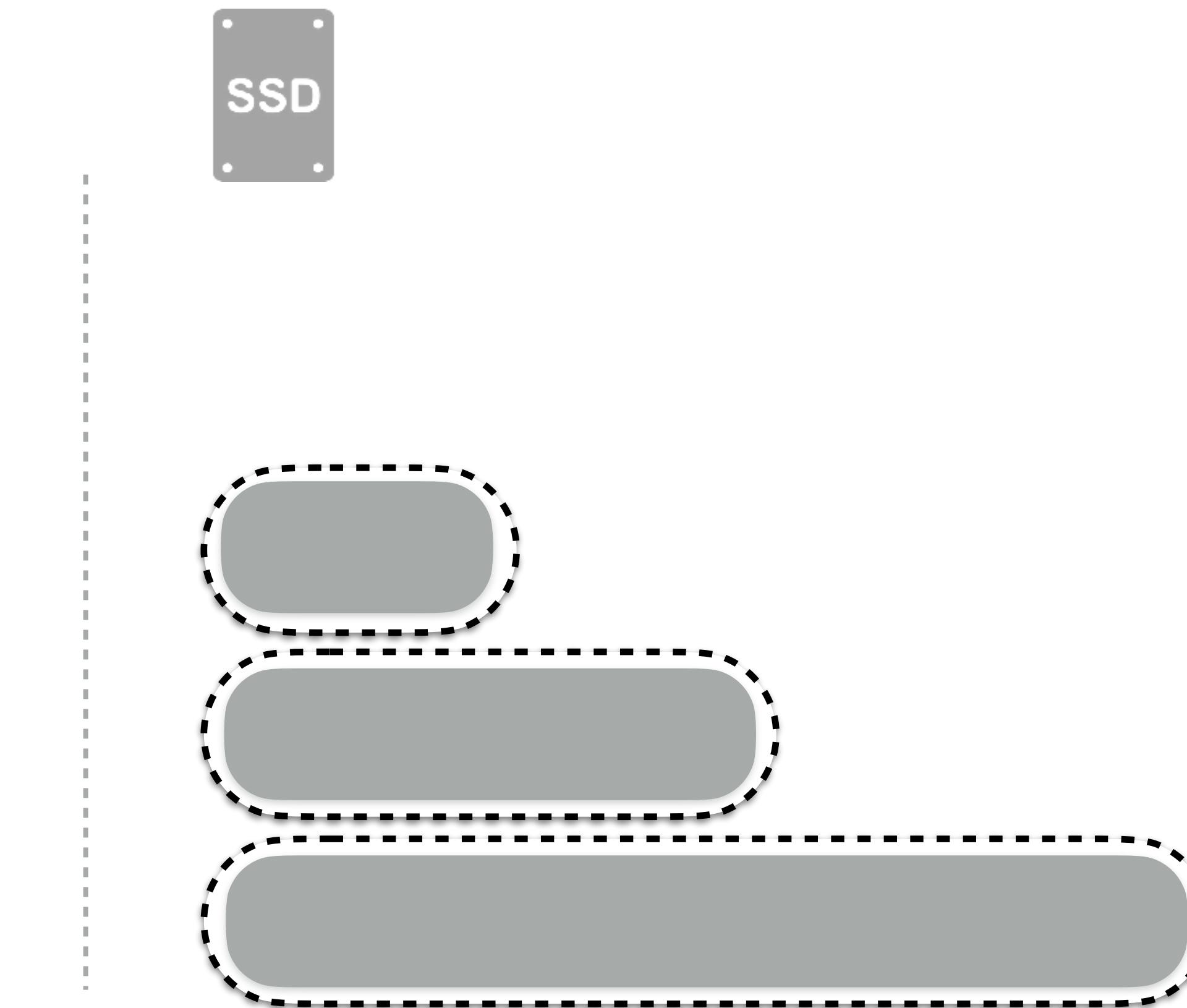




$$\# \text{ levels } L = \log_R(N/P)$$

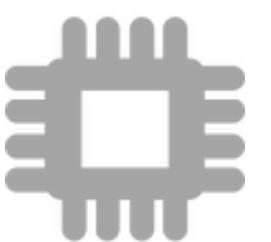
buffer size

The equation $\# \text{ levels } L = \log_R(N/P)$ is displayed. To its right is a vertical curly brace that spans from the bottom text "buffer size" up to the term N/P . Below the brace is a thin black arrow pointing upwards.



buffer

Bloom
filters



pointers

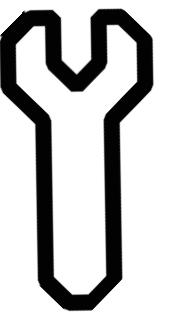


merge policy



writes

merge policy



reads



writes



merge policy



reads

two merge policies



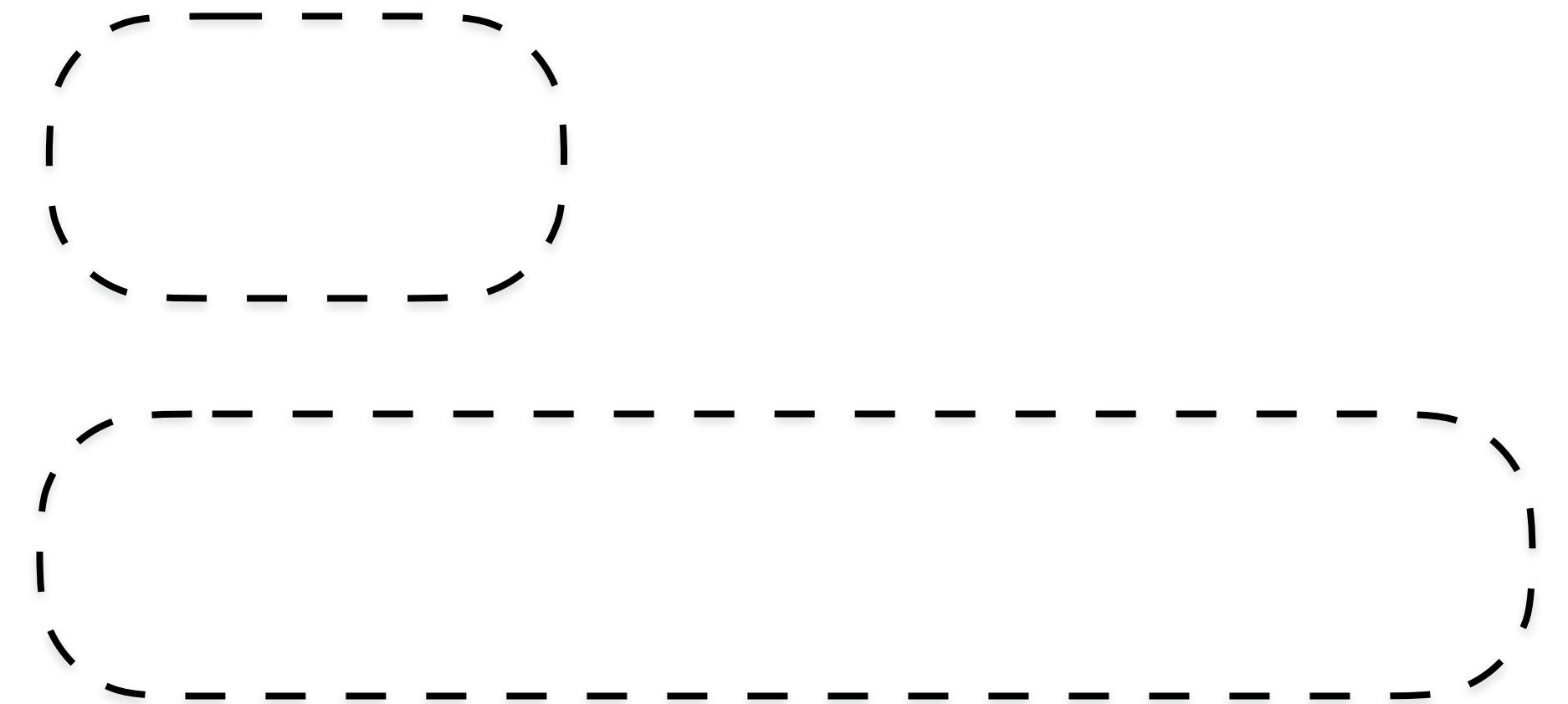
Tiering

Leveling





Tiering

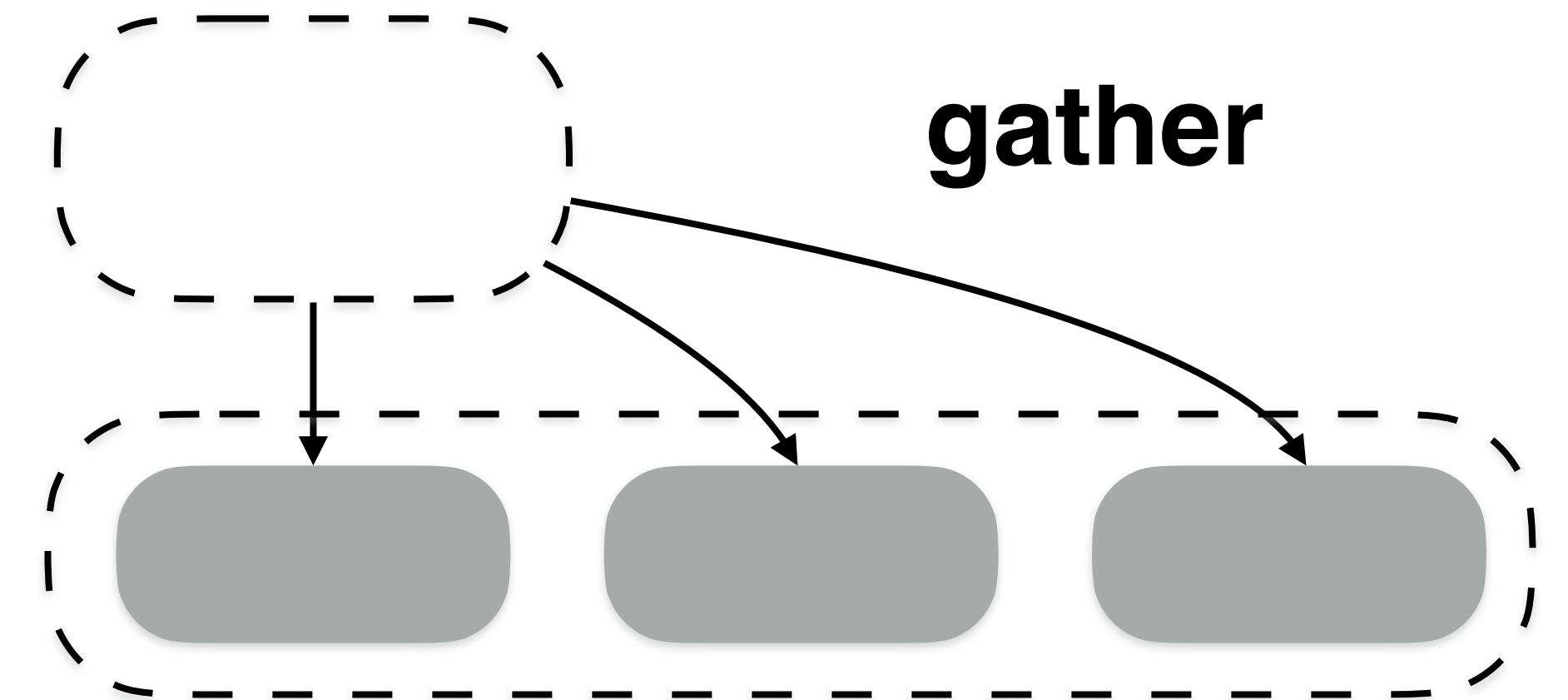


Leveling





Tiering



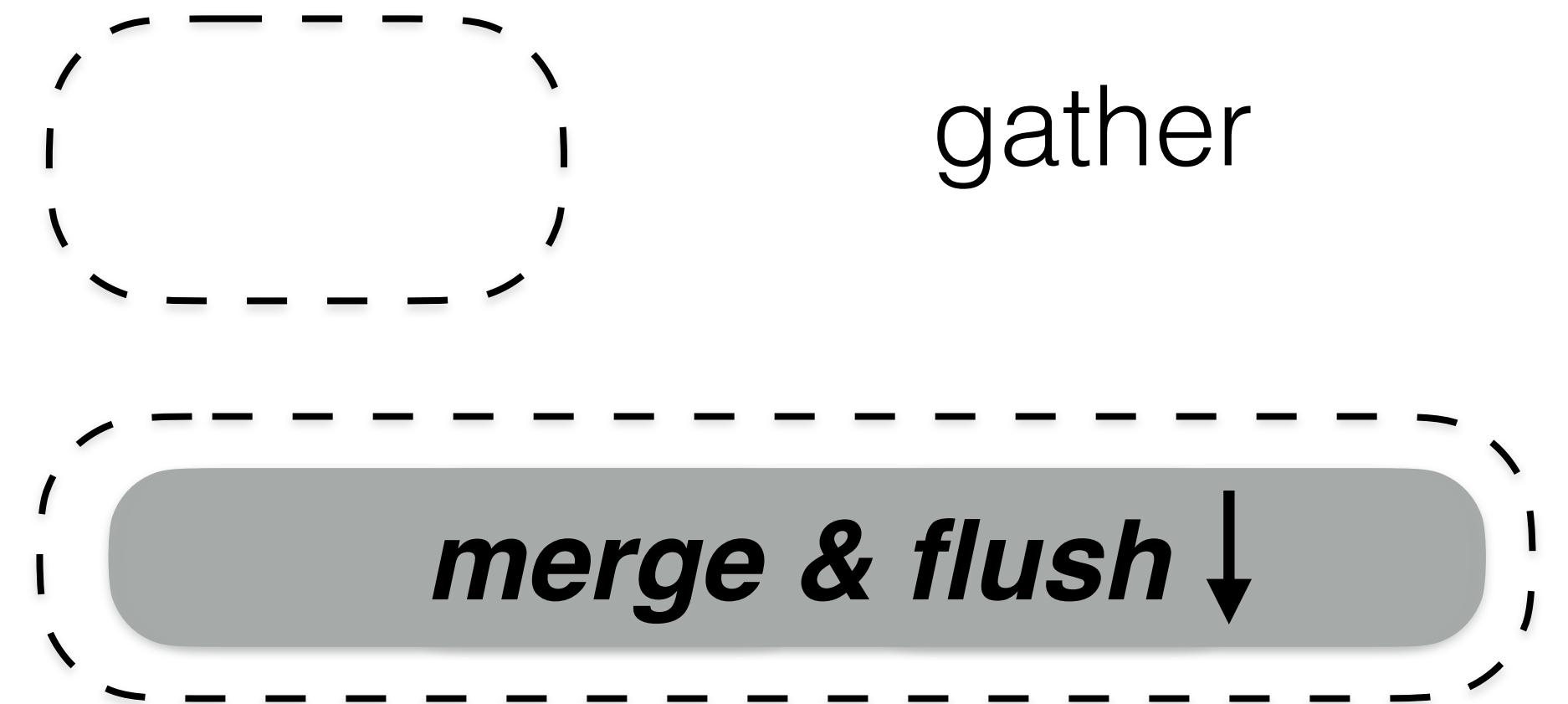
gather

Leveling





Tiering



Leveling

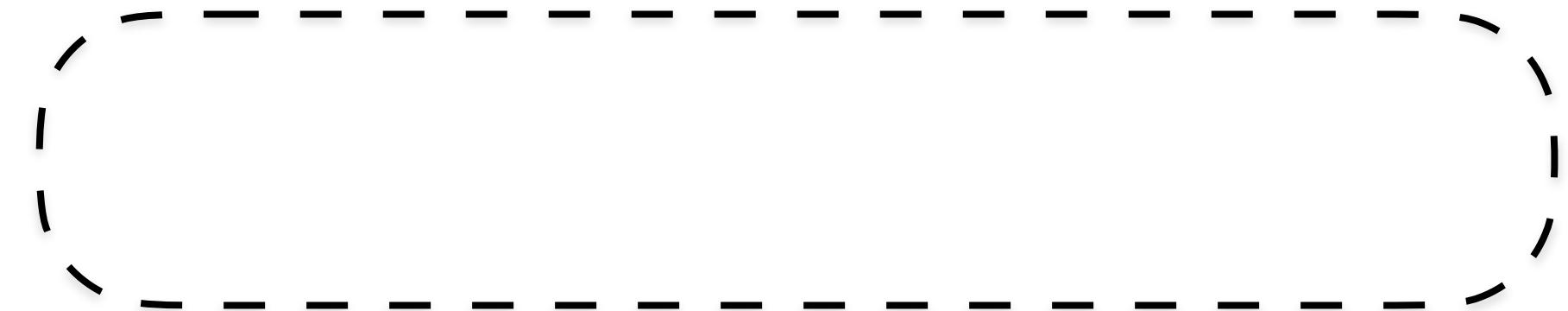
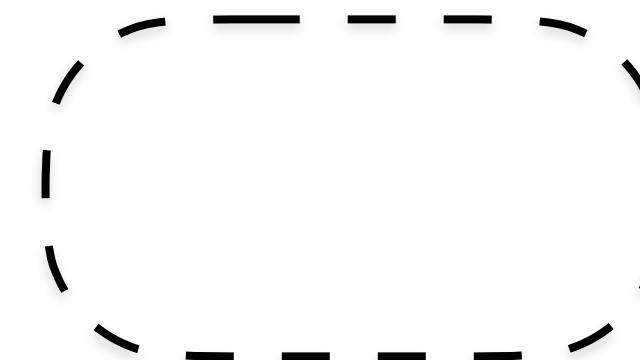
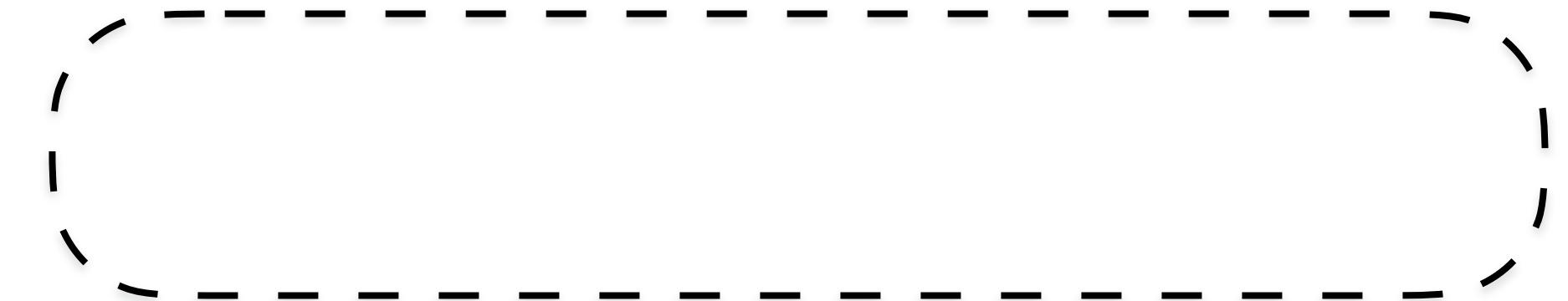




Tiering



gather

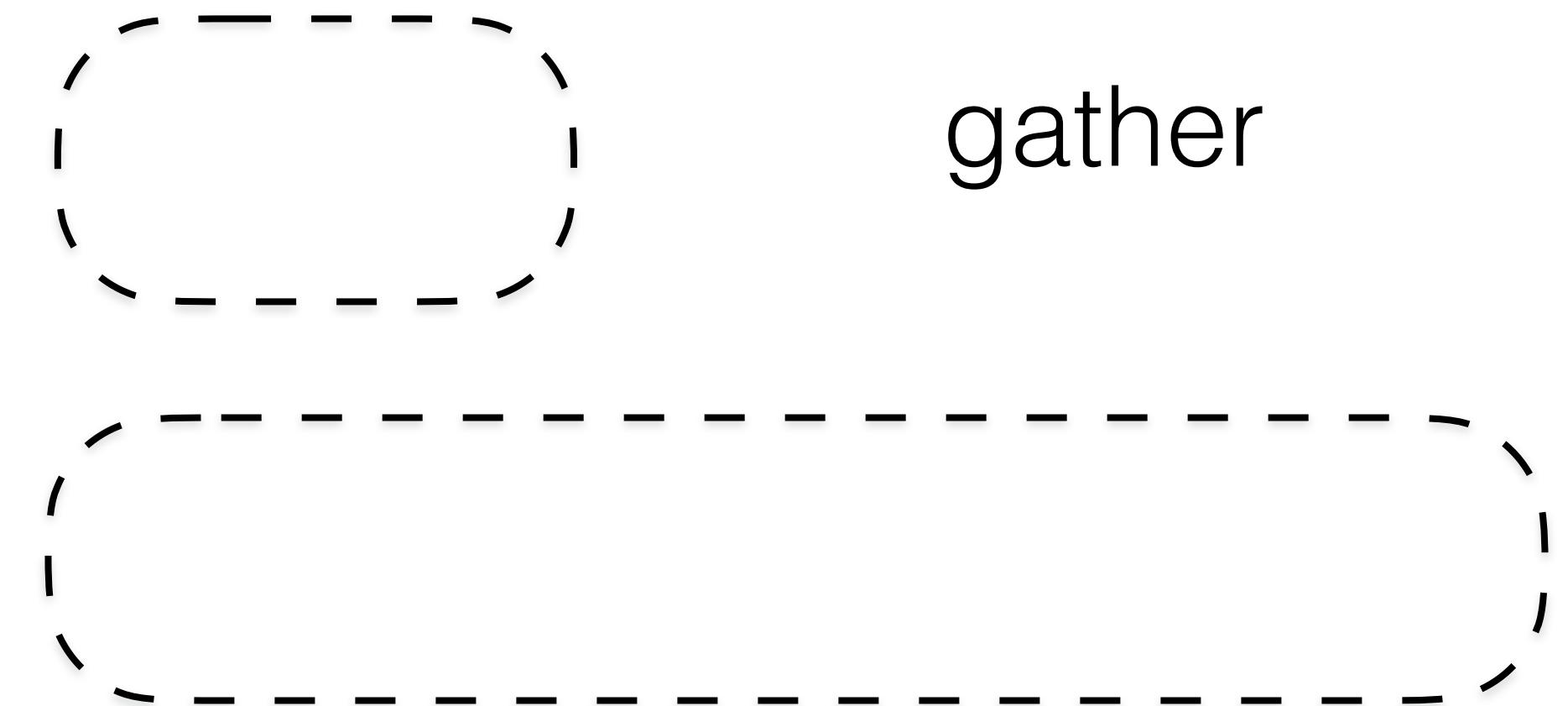


Leveling

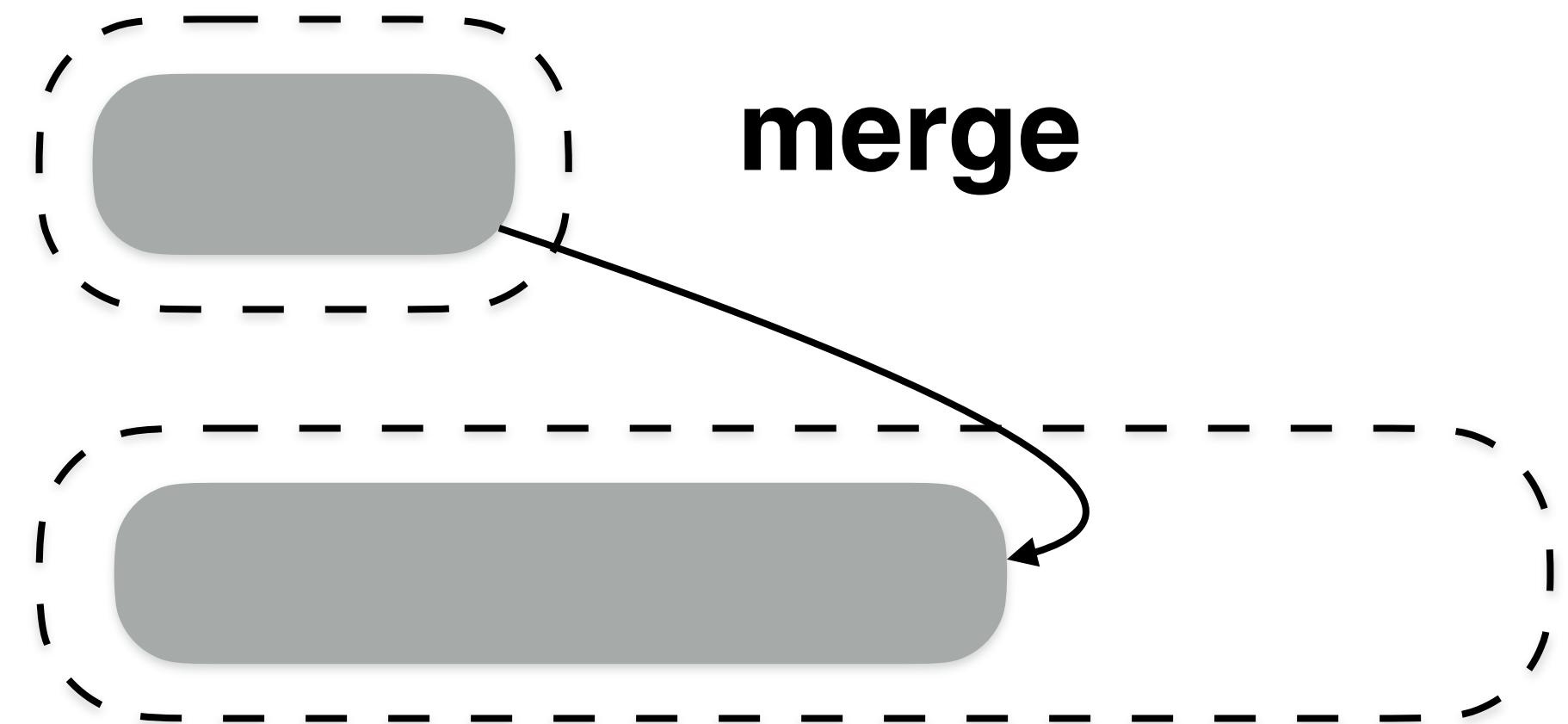




Tiering

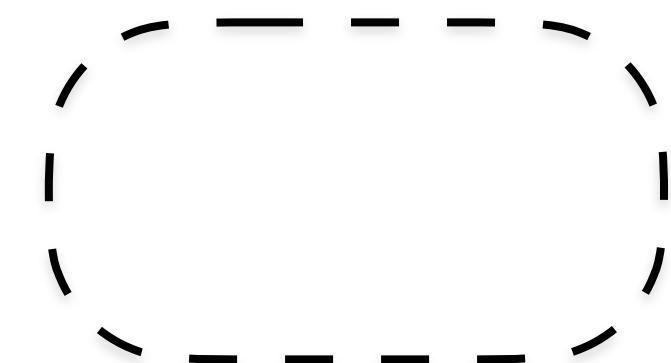


Leveling





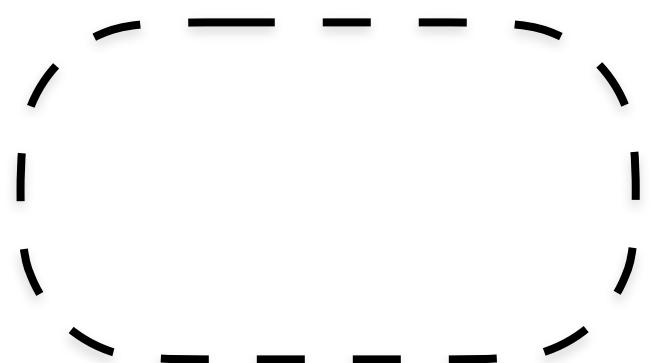
Tiering



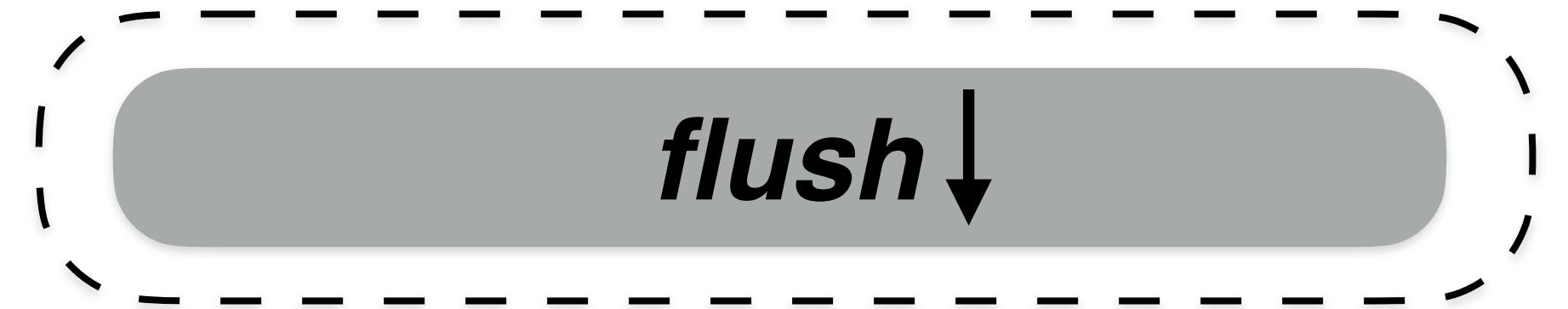
gather



Leveling



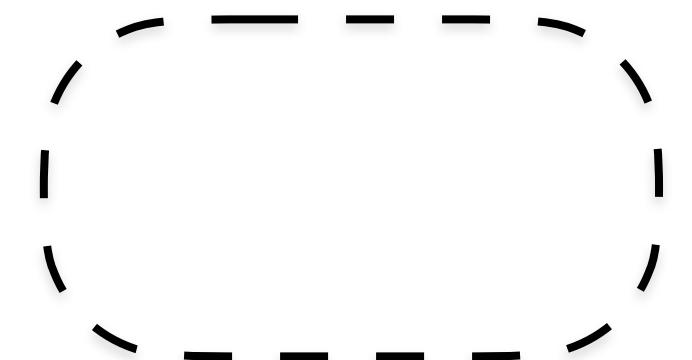
merge



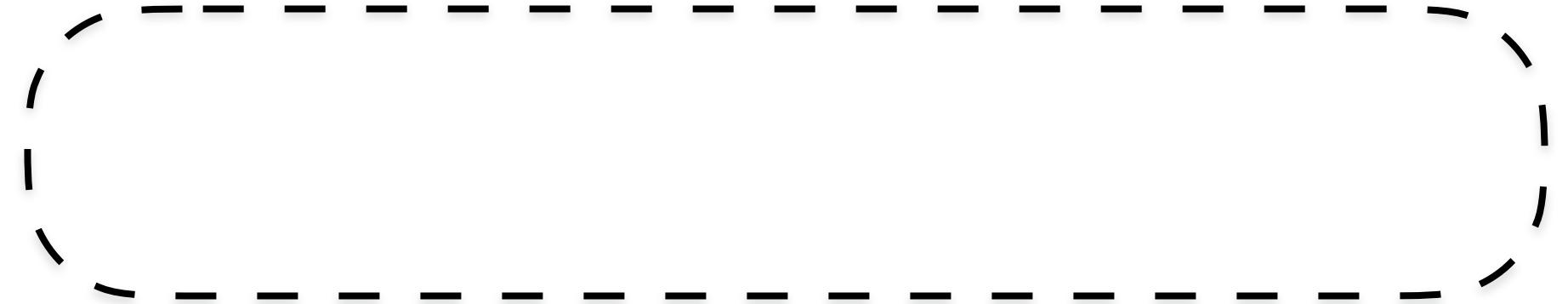
flush ↓



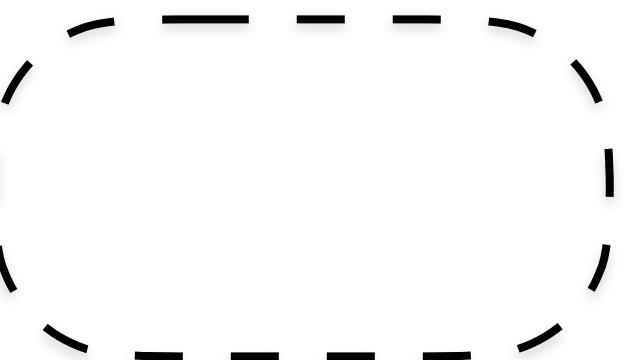
Tiering



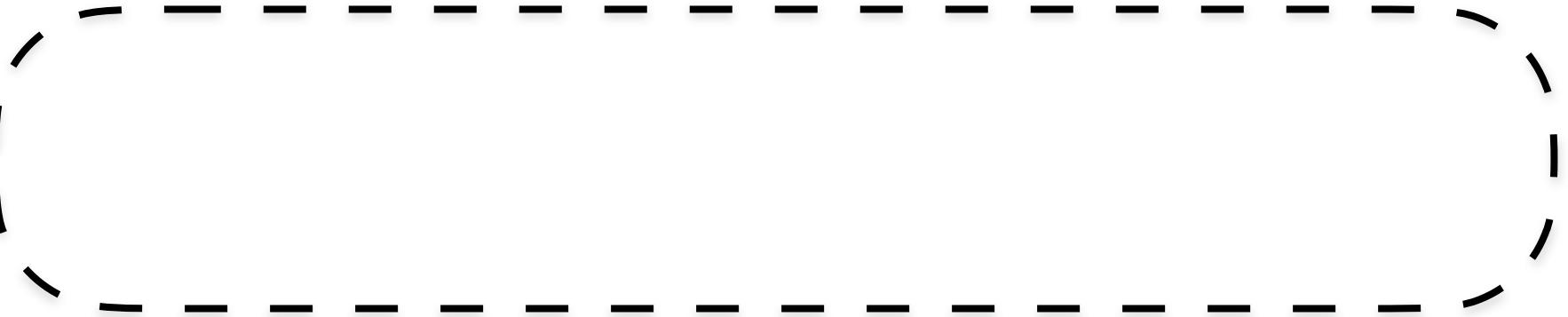
gather



Leveling

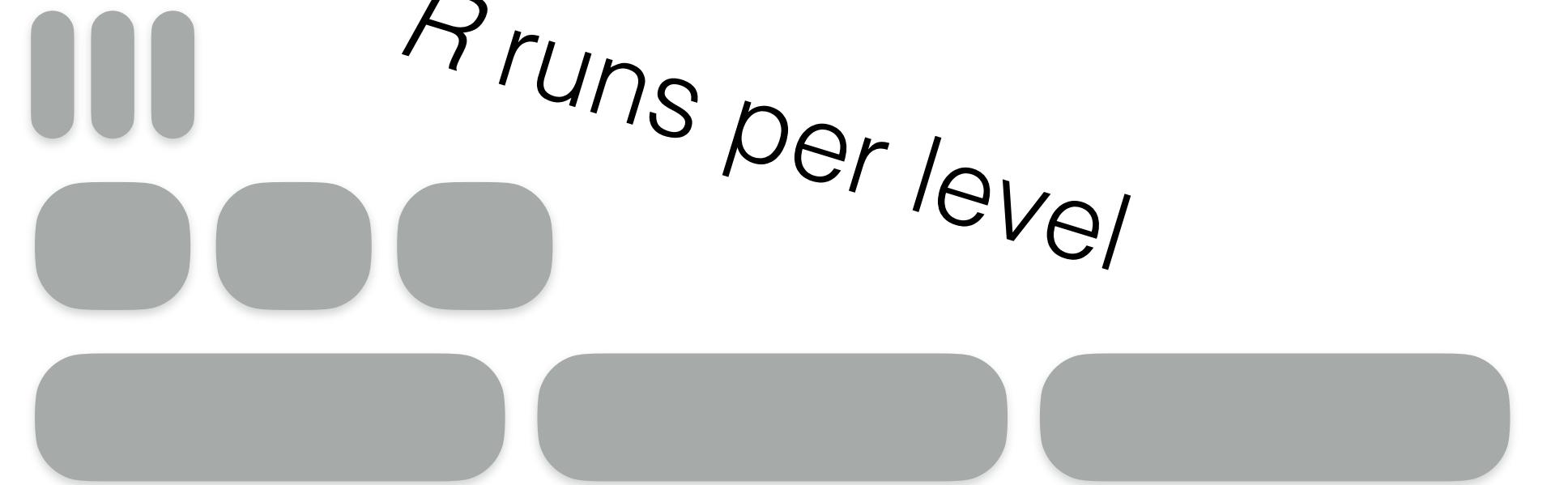


merge





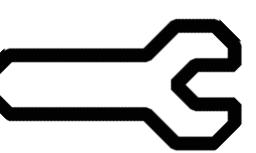
Tiering



R runs per level



1 run per level



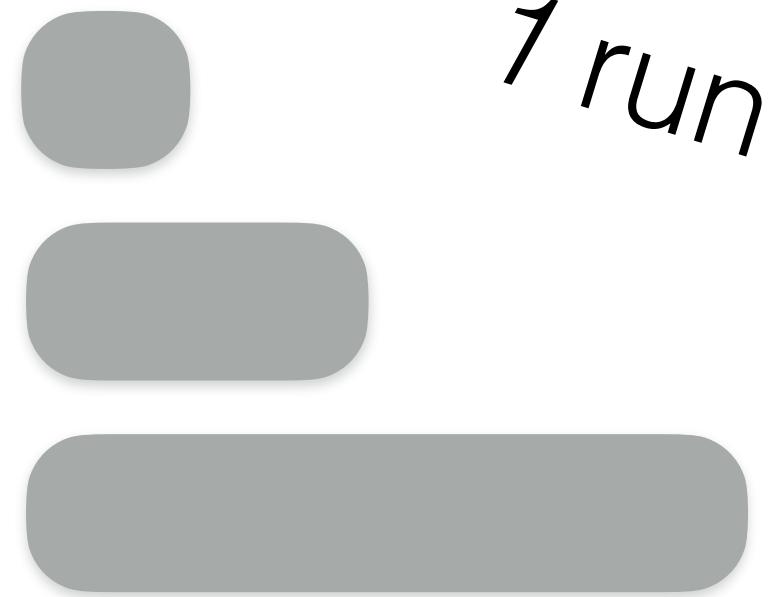
size ratio R





Tiering

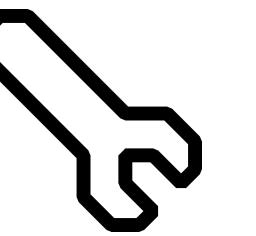
Leveling



1 run per level



1 run per level



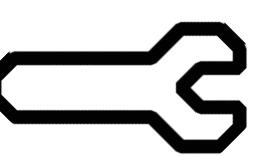
size ratio $R = 2$



Tiering



Leveling



size ratio R

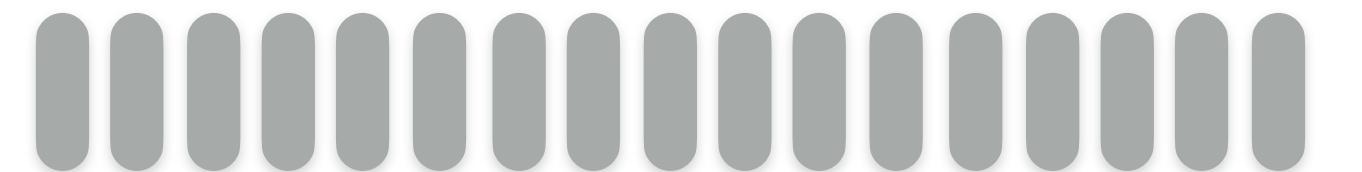


Tiering

Leveling



$O(N/P)$ runs per level



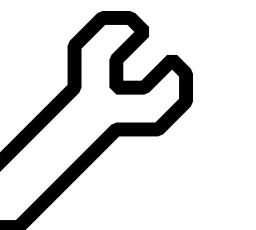
log

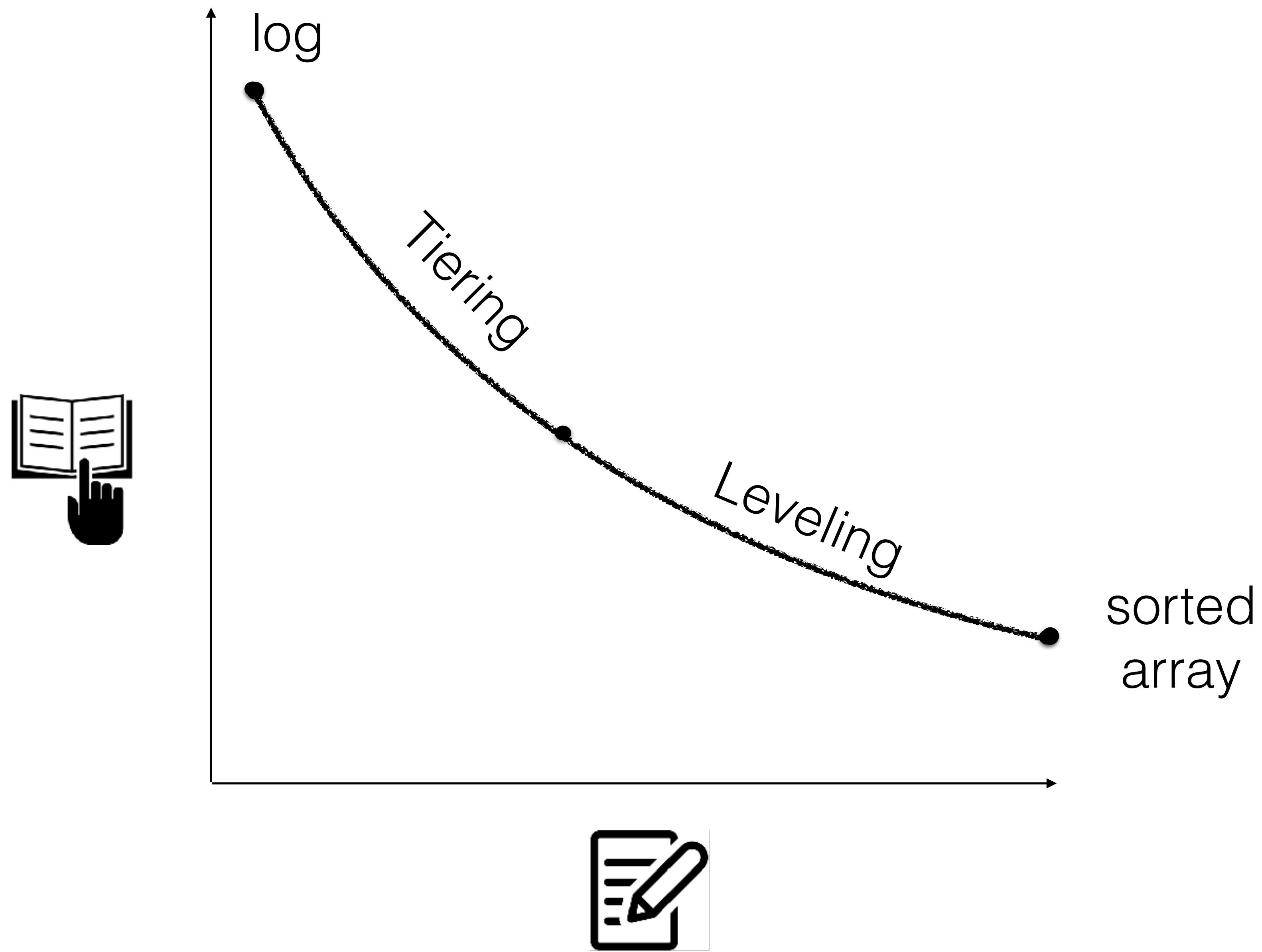
1 run per level

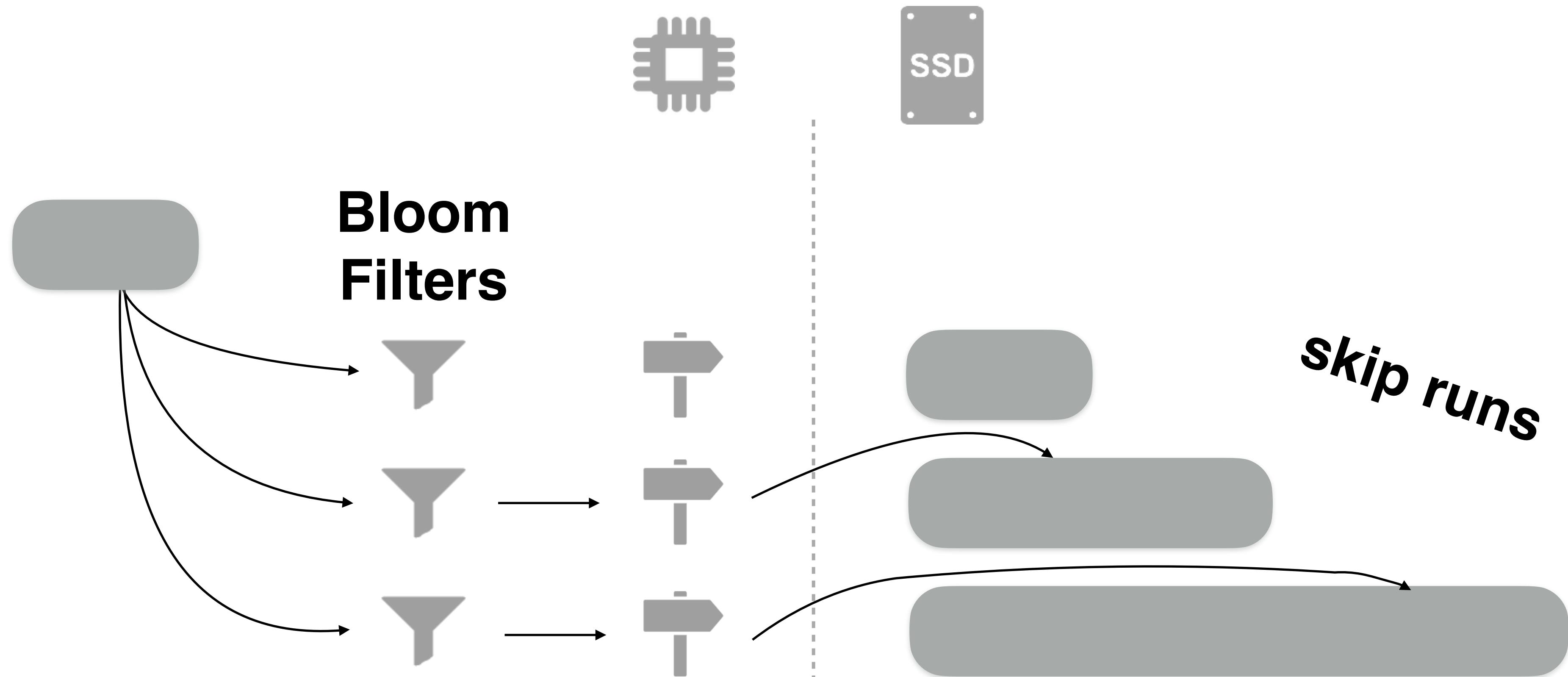


**sorted
array**

size ratio $R = N/P$





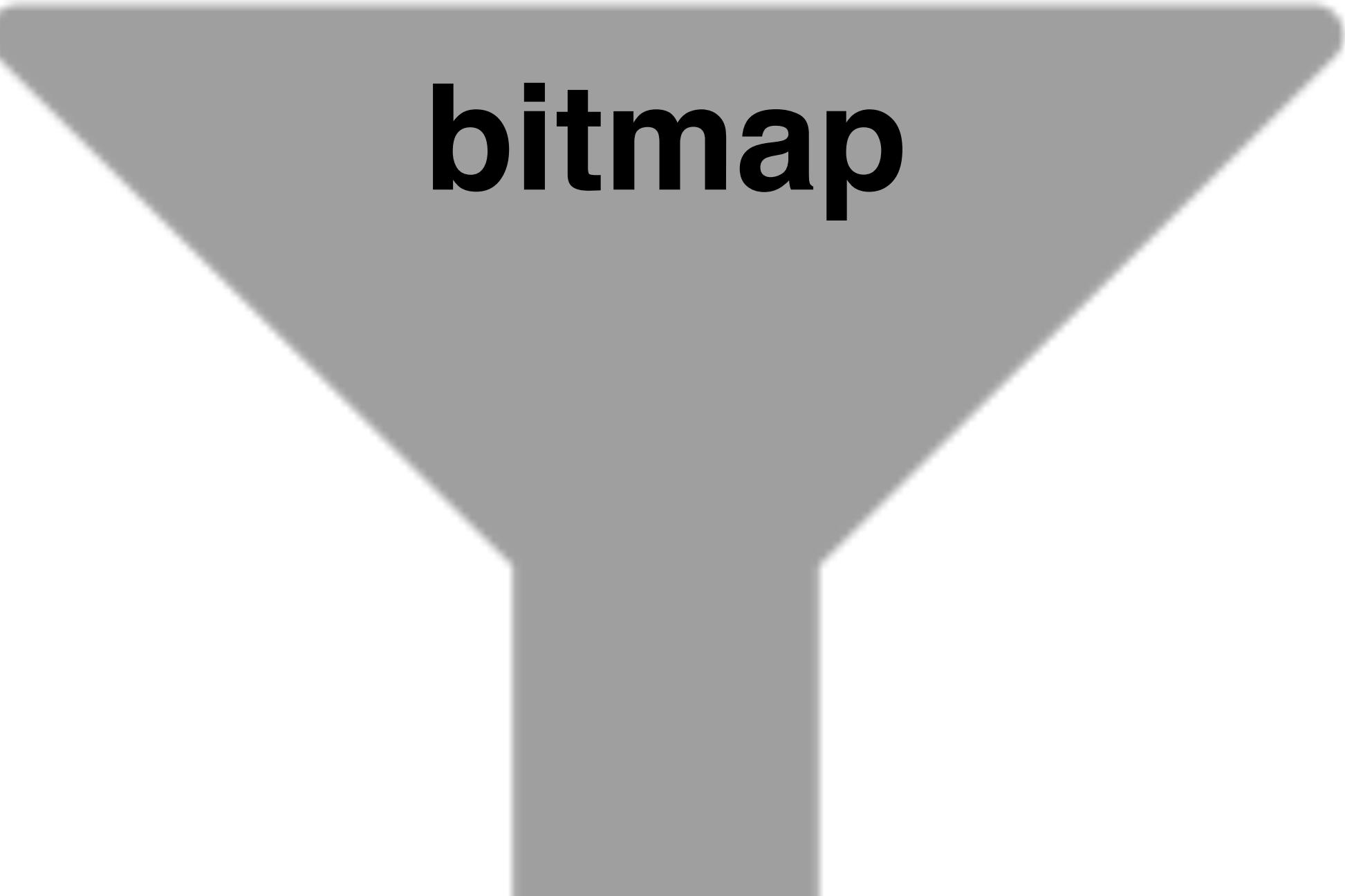


Bloom Filters: Deep Look



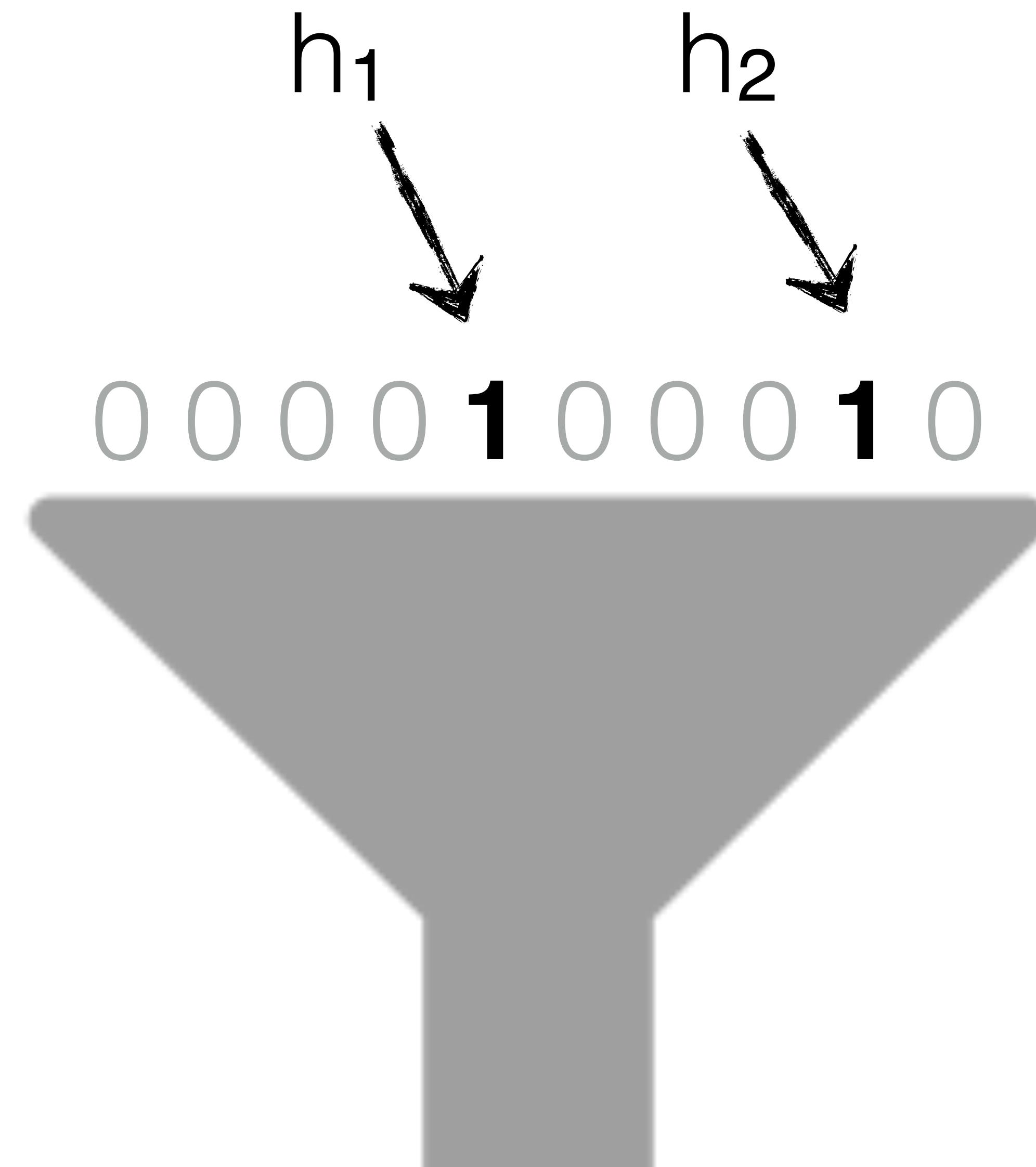
k hash functions

0 0 0 0 0 0 0 0 0

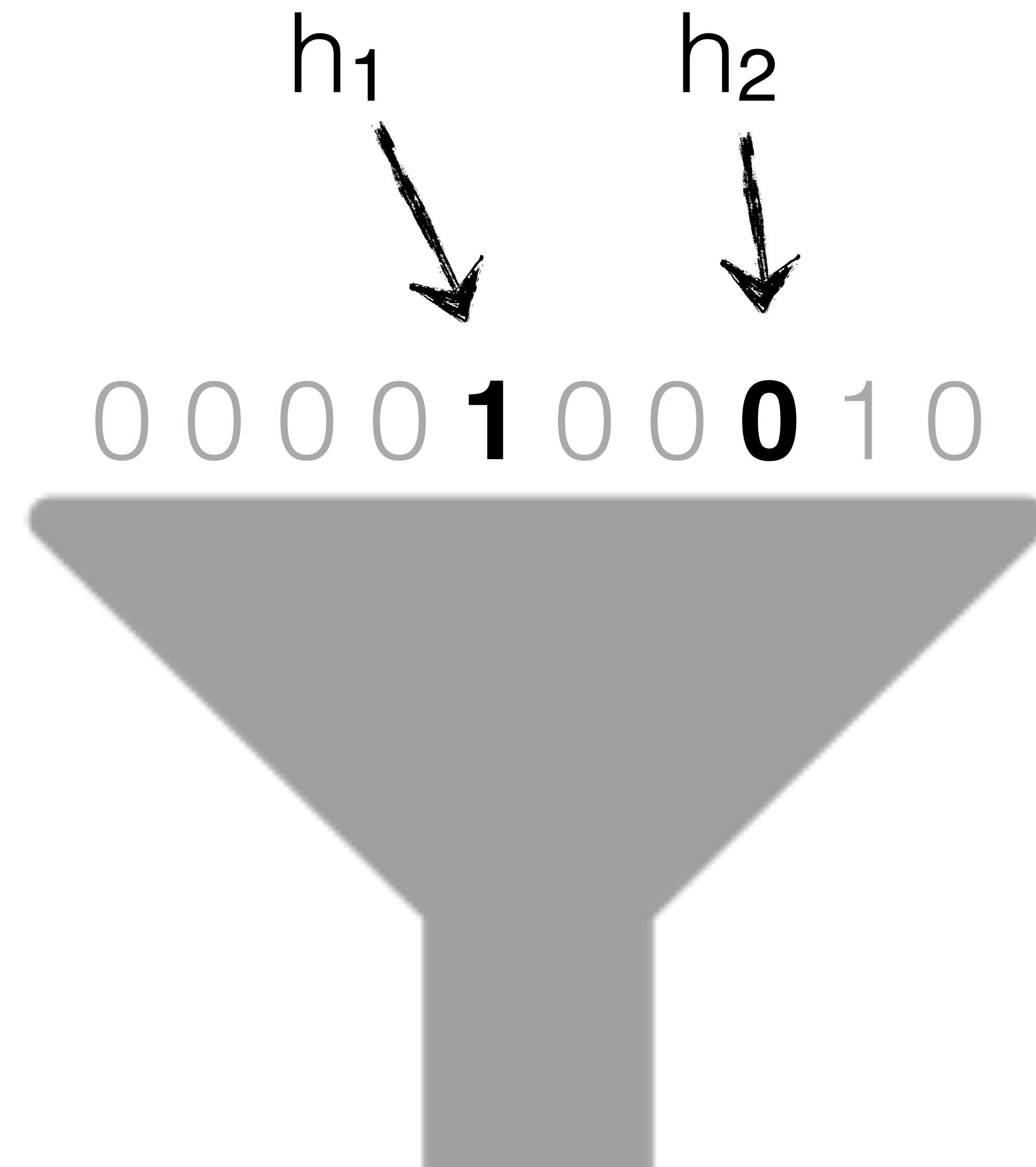


bitmap

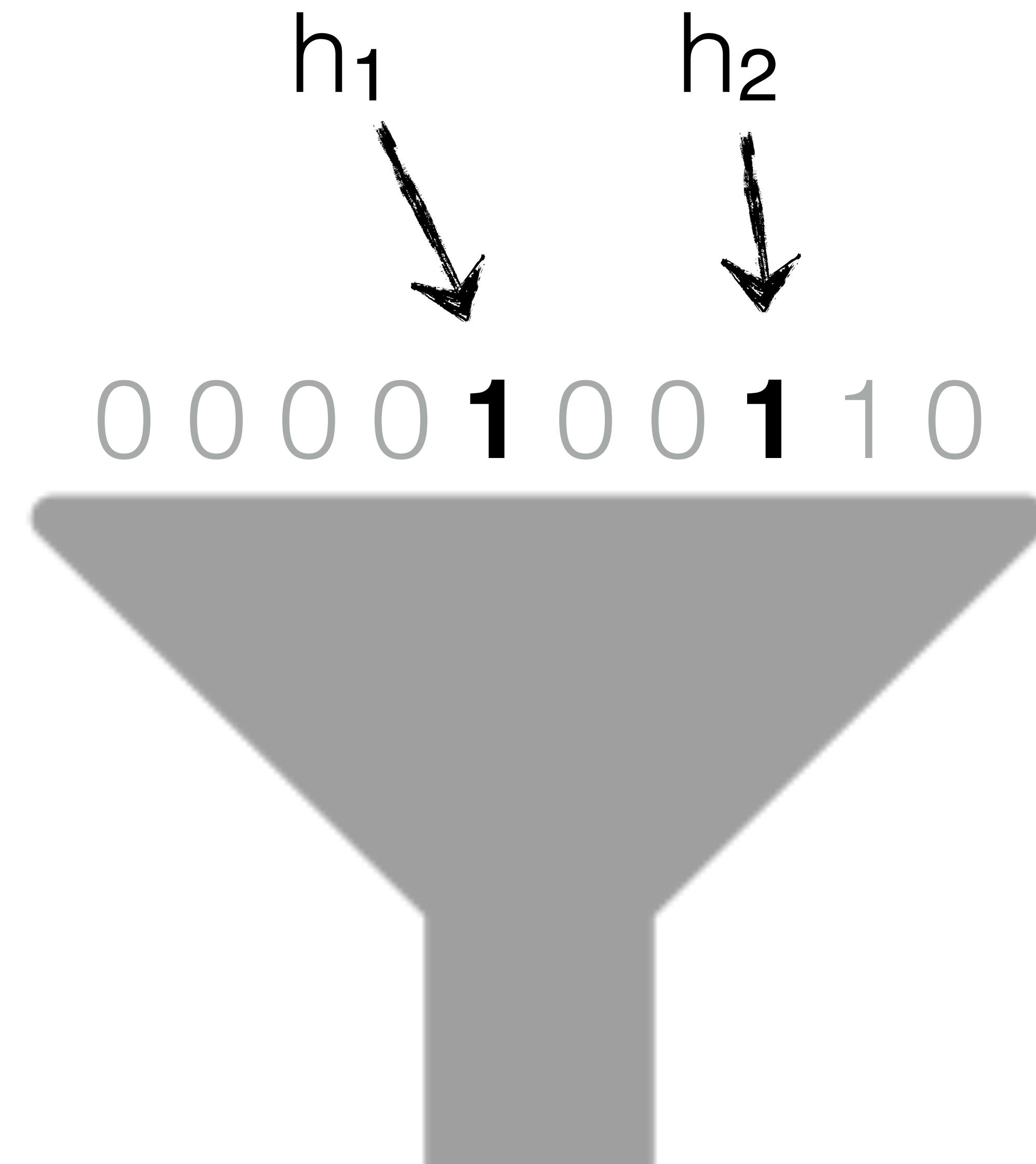
insert: Set from 0 to 1 or keep 1

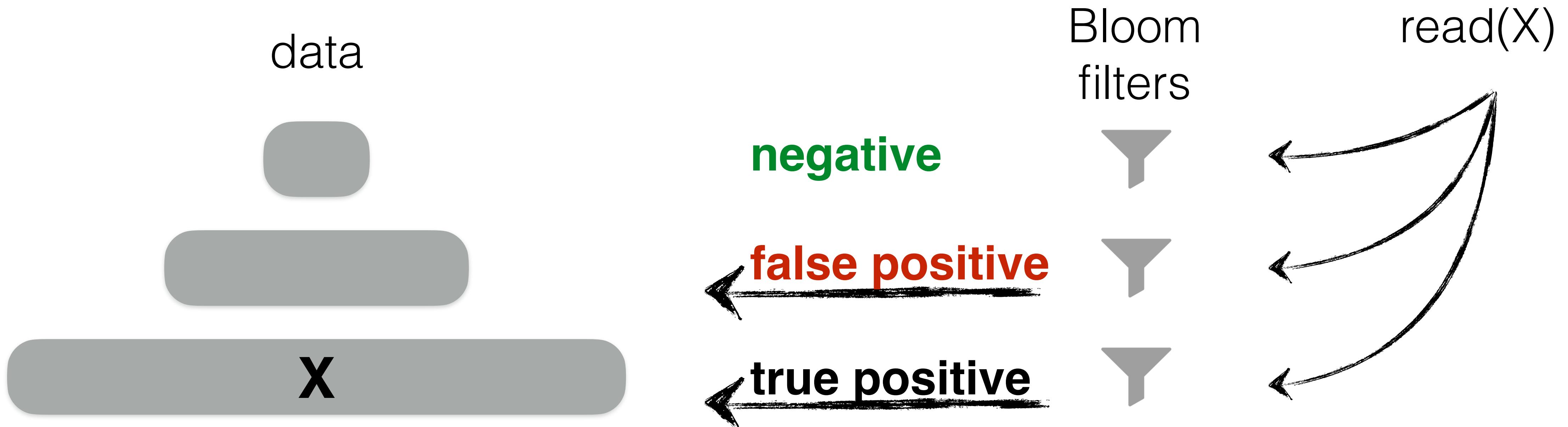


negative lookup: at least one bit is zero

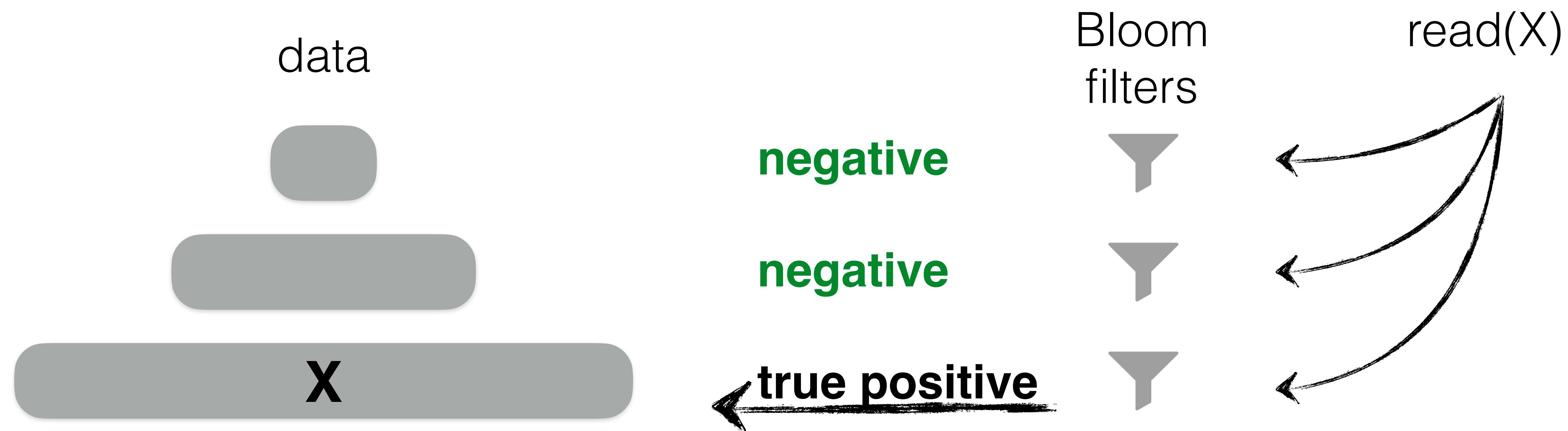


true or false positive lookup





more memory → fewer false positives

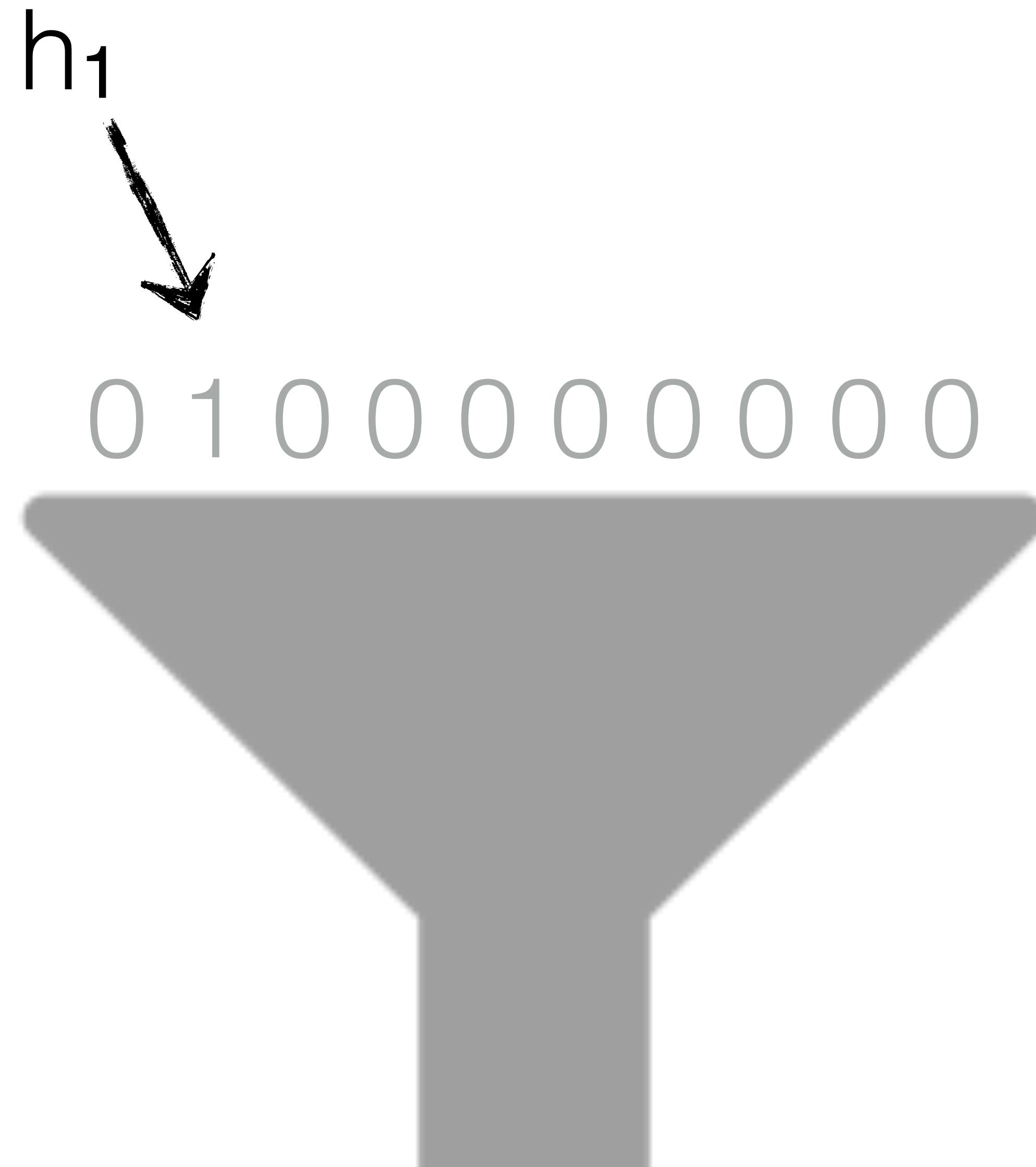


How many hash functions should we use?

0 0 0 0 0 0 0 0

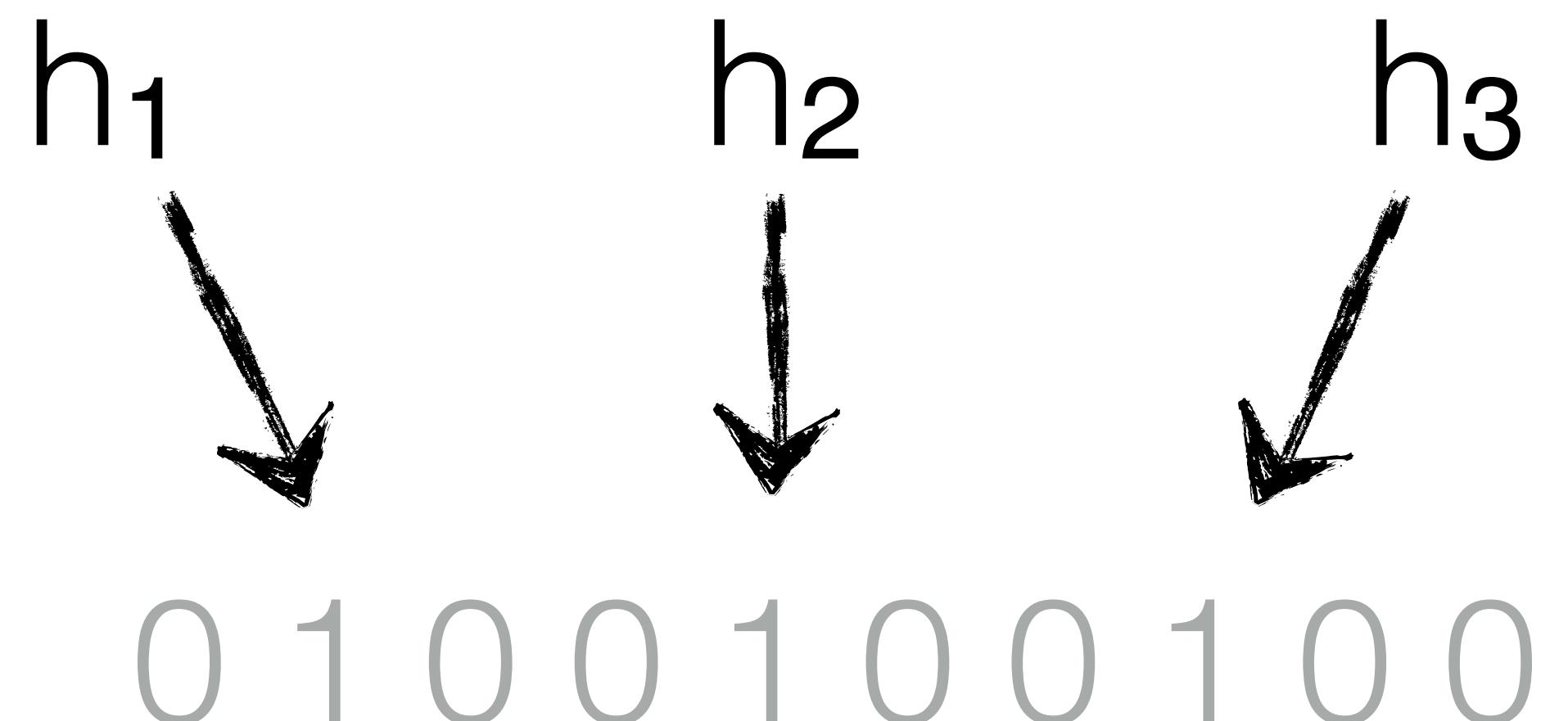


How many hash functions should we use?



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

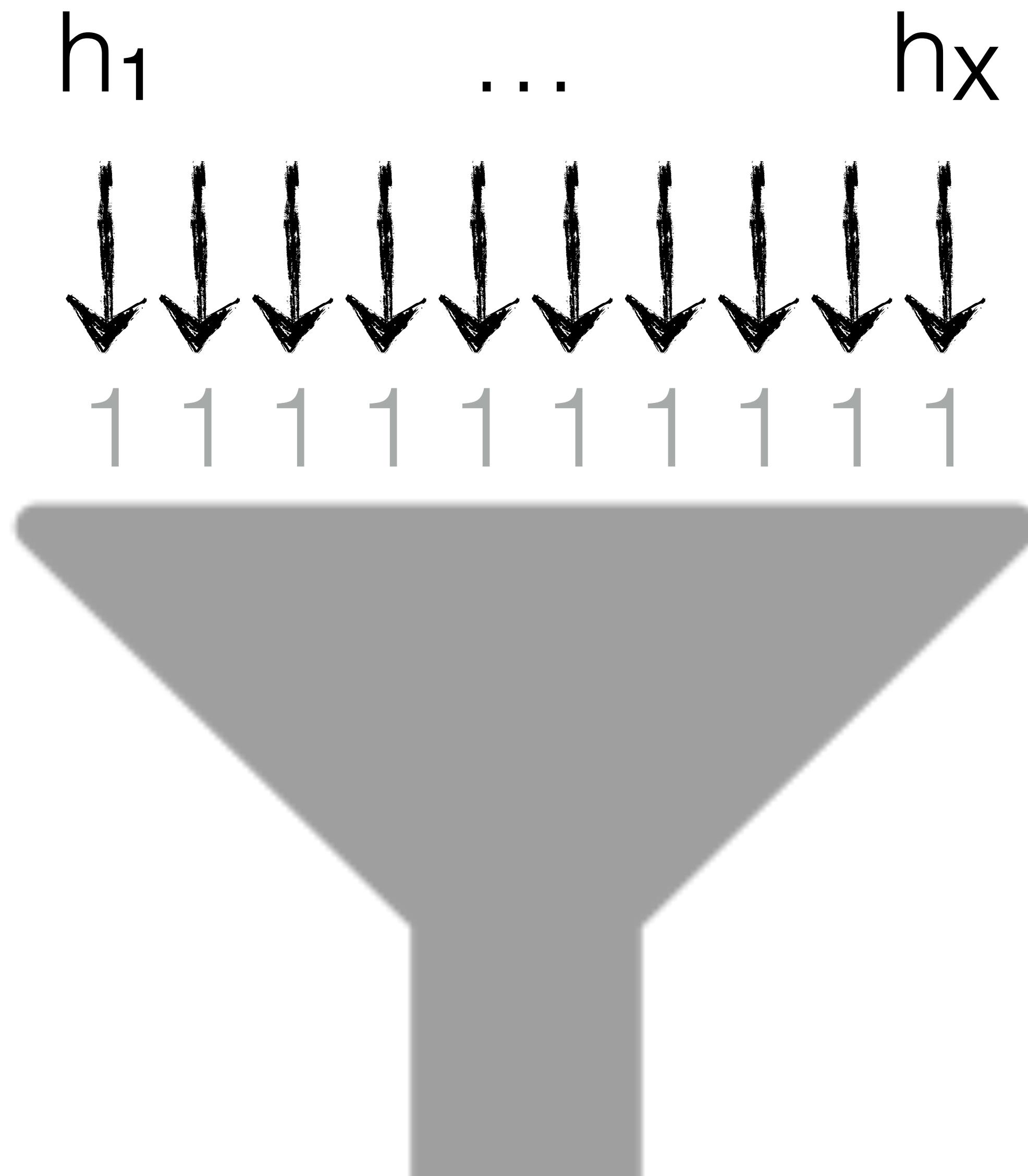
How many hash functions should we use?



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?

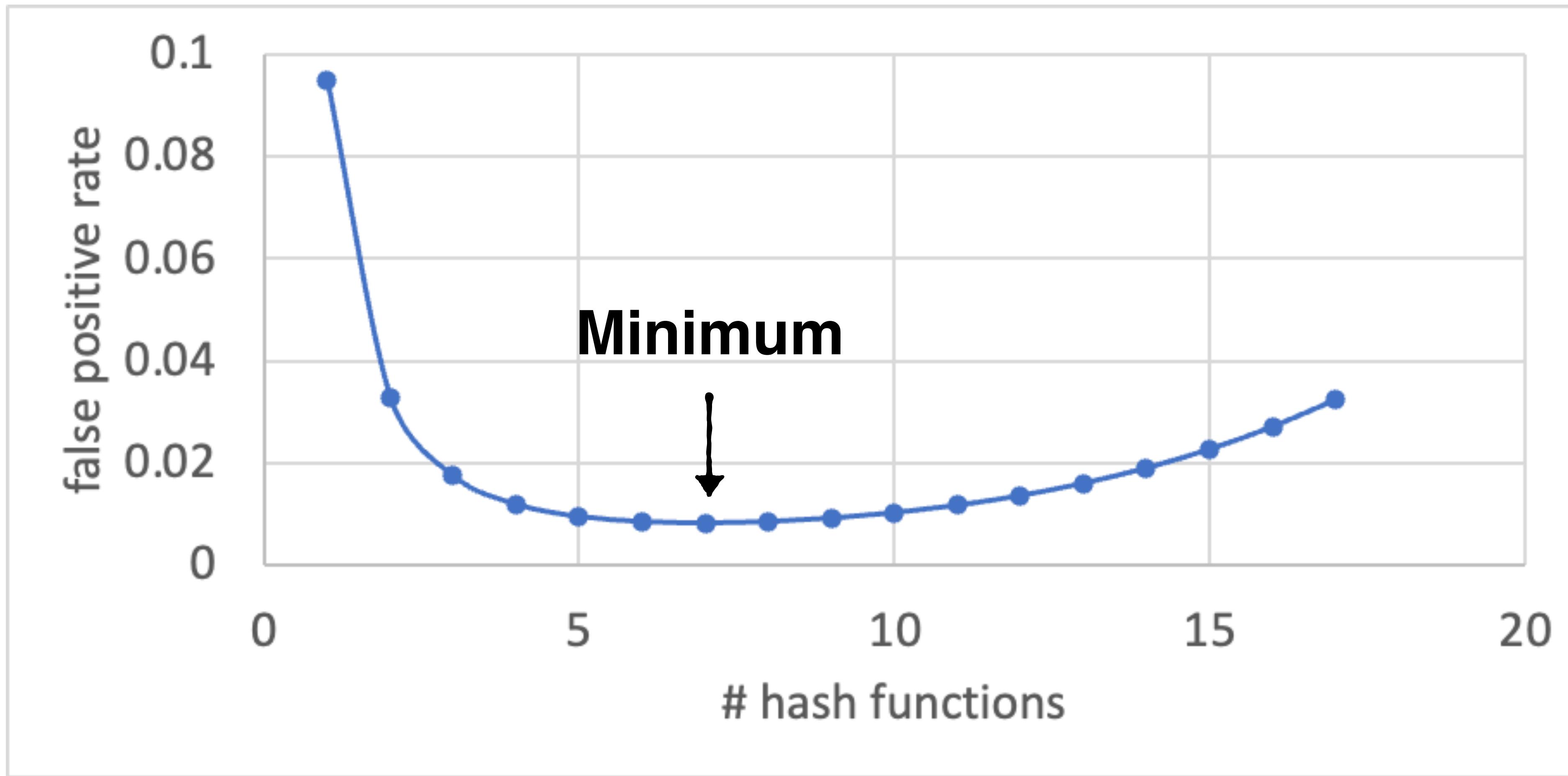


One is too few: false positive occurs whenever we hit a 1

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

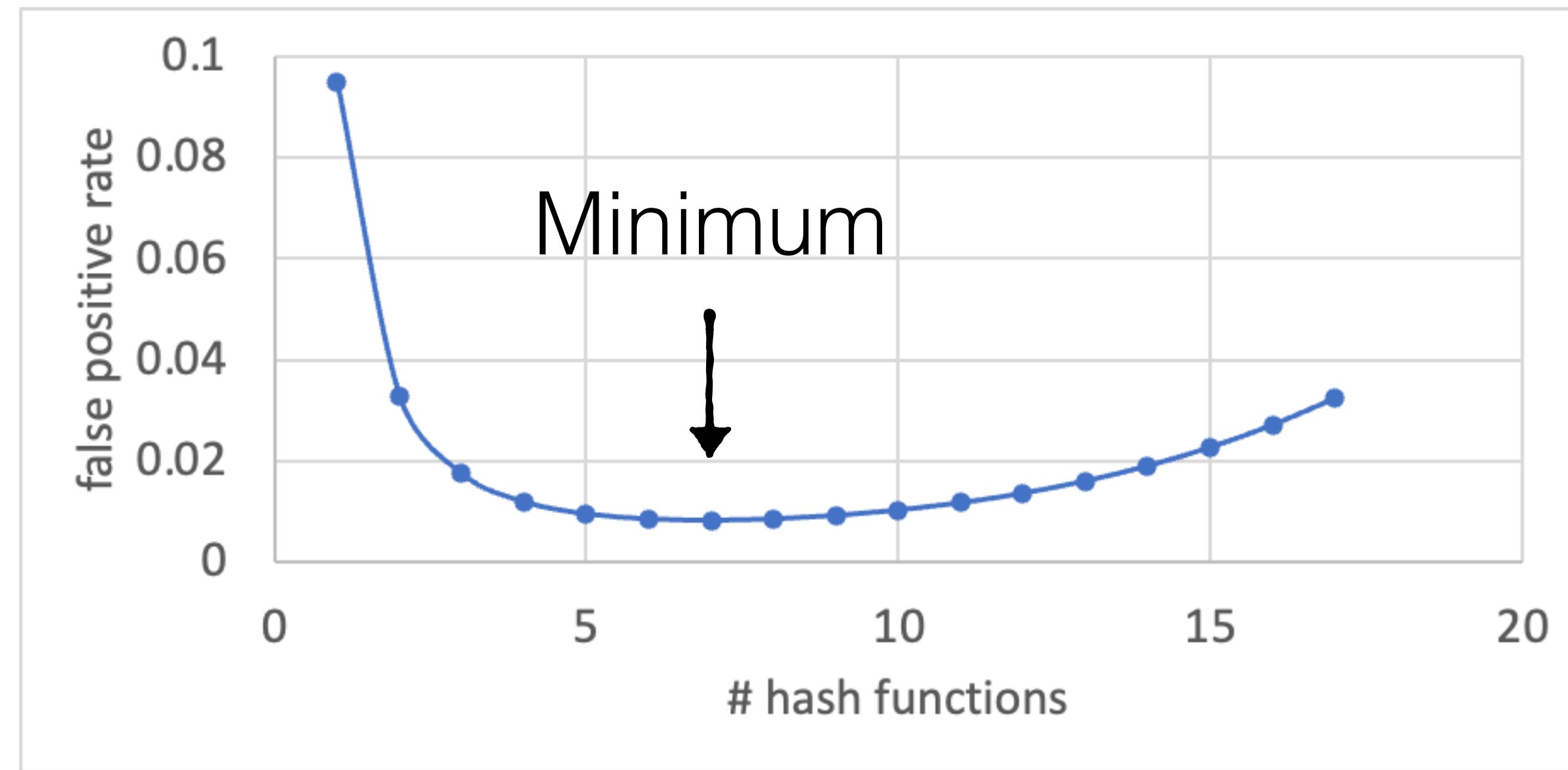
But too many hash functions wind up increasing the FPR.

How many hash functions should we use?



(Drawn for a filter using 10 bits per entry)

How many hash functions should we use?

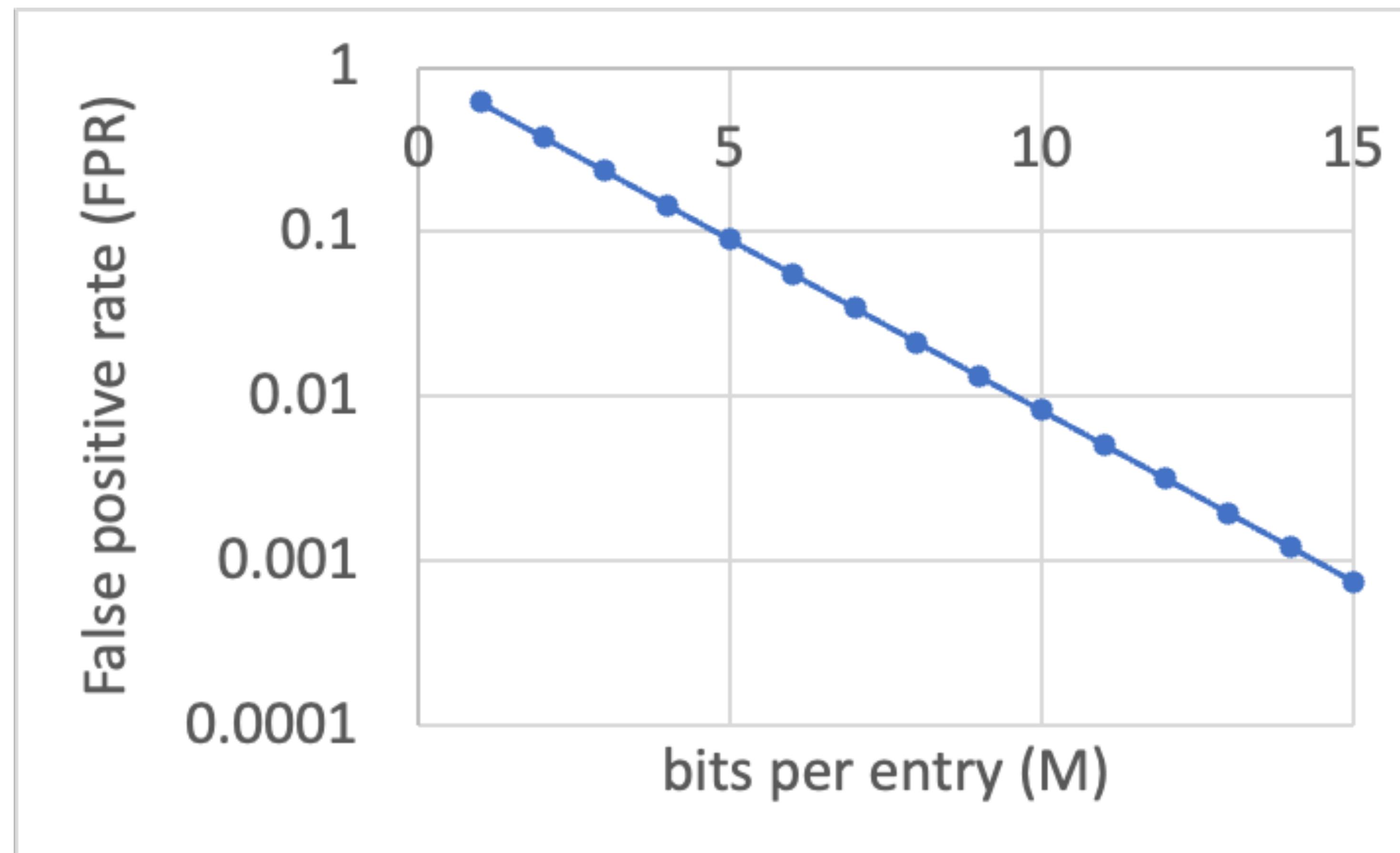


Optimal # hash functions = $\ln(2) * M$

(M is the number of bits per entry)

assuming the optimal # hash functions,

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



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 = $1 + 1/2 (1 + 1/2 * (\dots))$

(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 = $1 + 1/2 + 1/4 + \dots = 2$

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

More memory M , lower FPR, worst performance

$$\text{Insertion} = M * \ln(2)$$

$$\text{Positive Query} = M * \ln(2)$$

$$\text{Avg. Negative Query} = 2$$

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

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

Positive Query = $M * \ln(2)$
Avg. Negative Query = 2

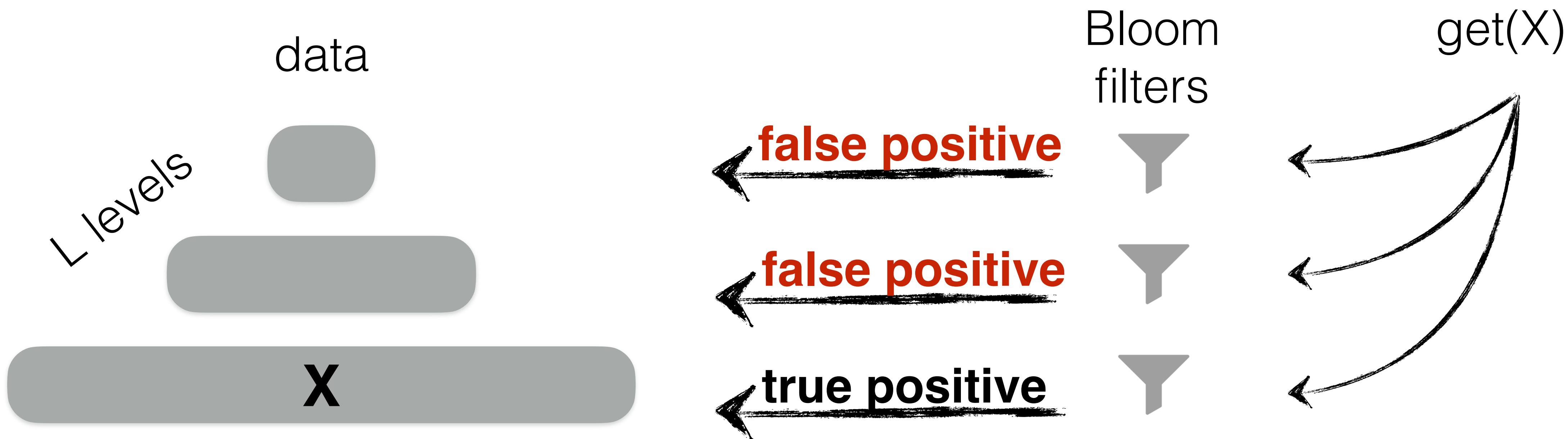


Worst-case:

Avg. worst-case:

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

Positive Query = $M * \ln(2)$
Avg. Negative Query = 2

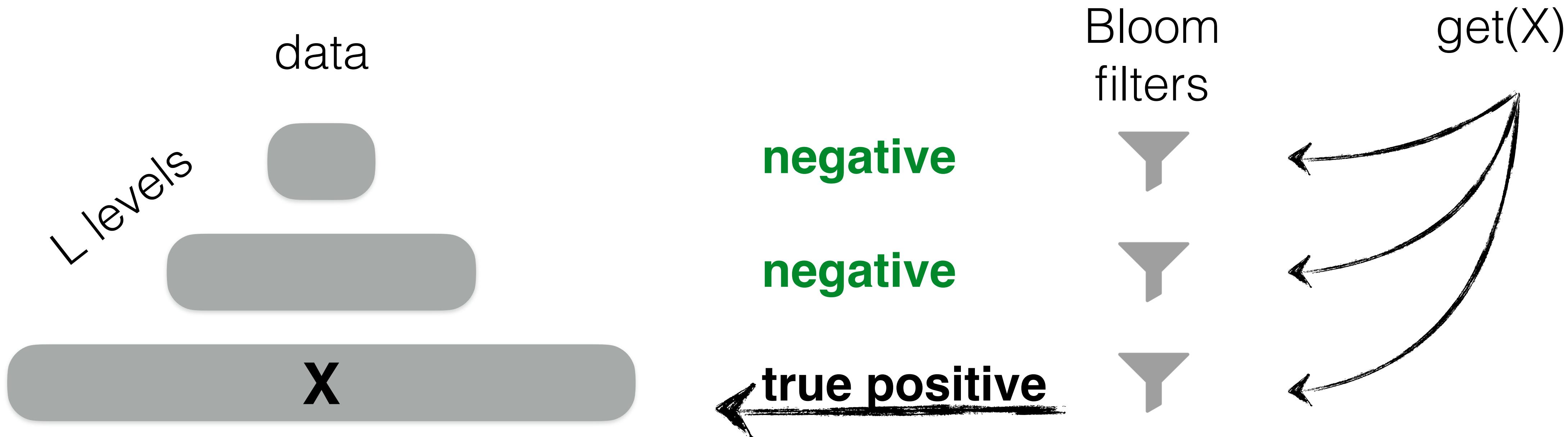


Worst-case: $O(M \cdot L)$

Avg. worst-case:

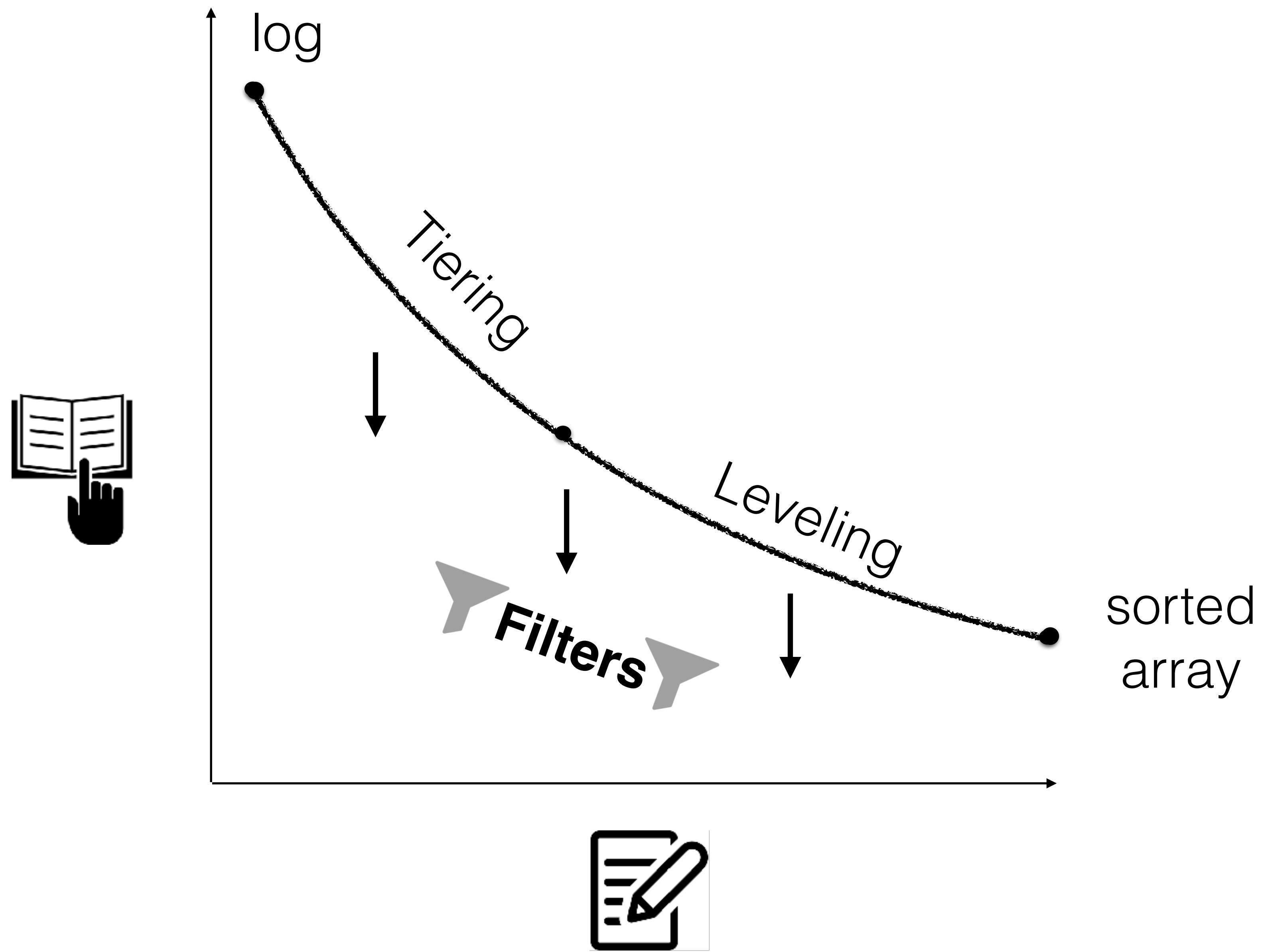
Let's analyze overall filter access cost for basic LSM-tree

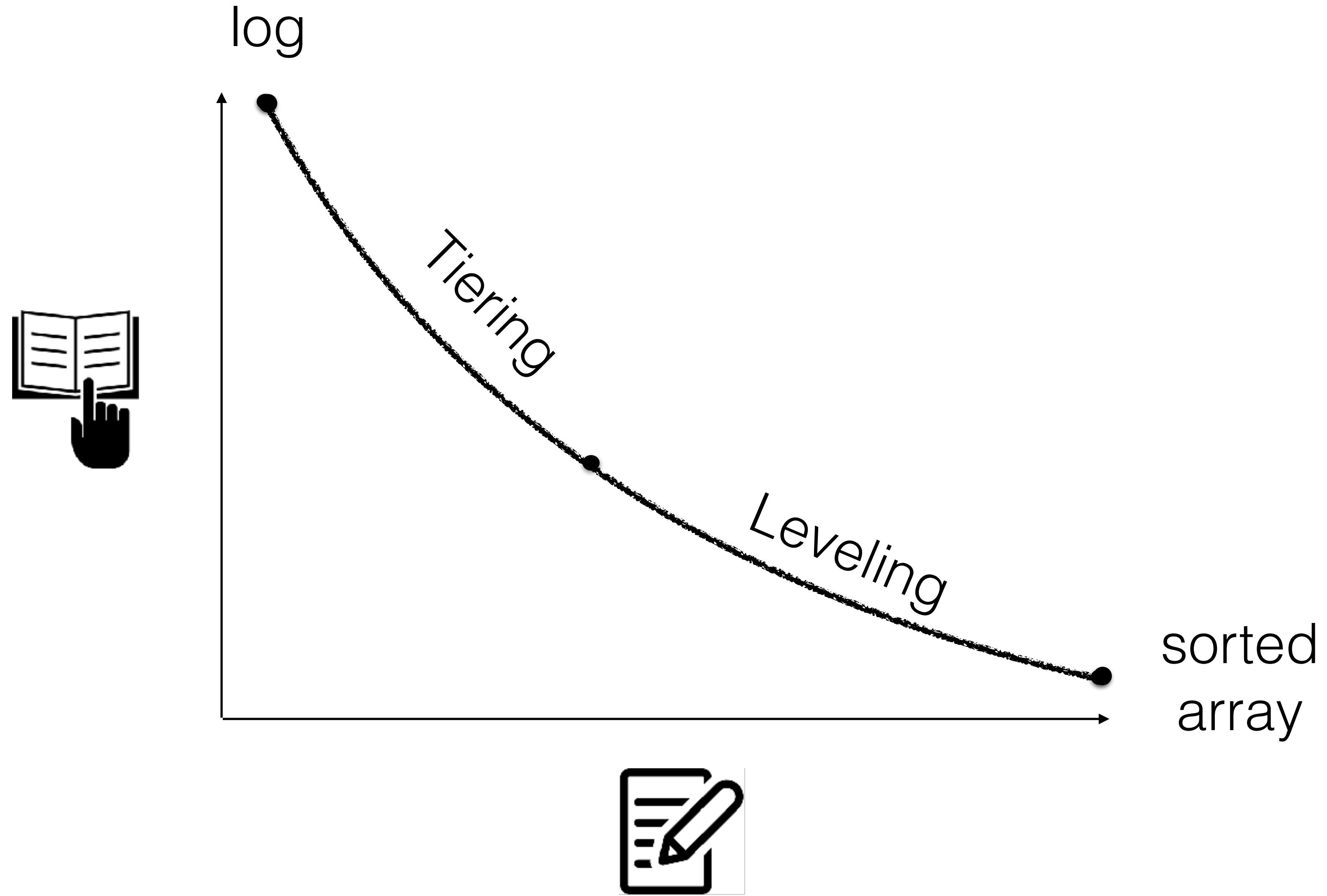
Positive Query = $M * \ln(2)$
Avg. Negative Query = 2

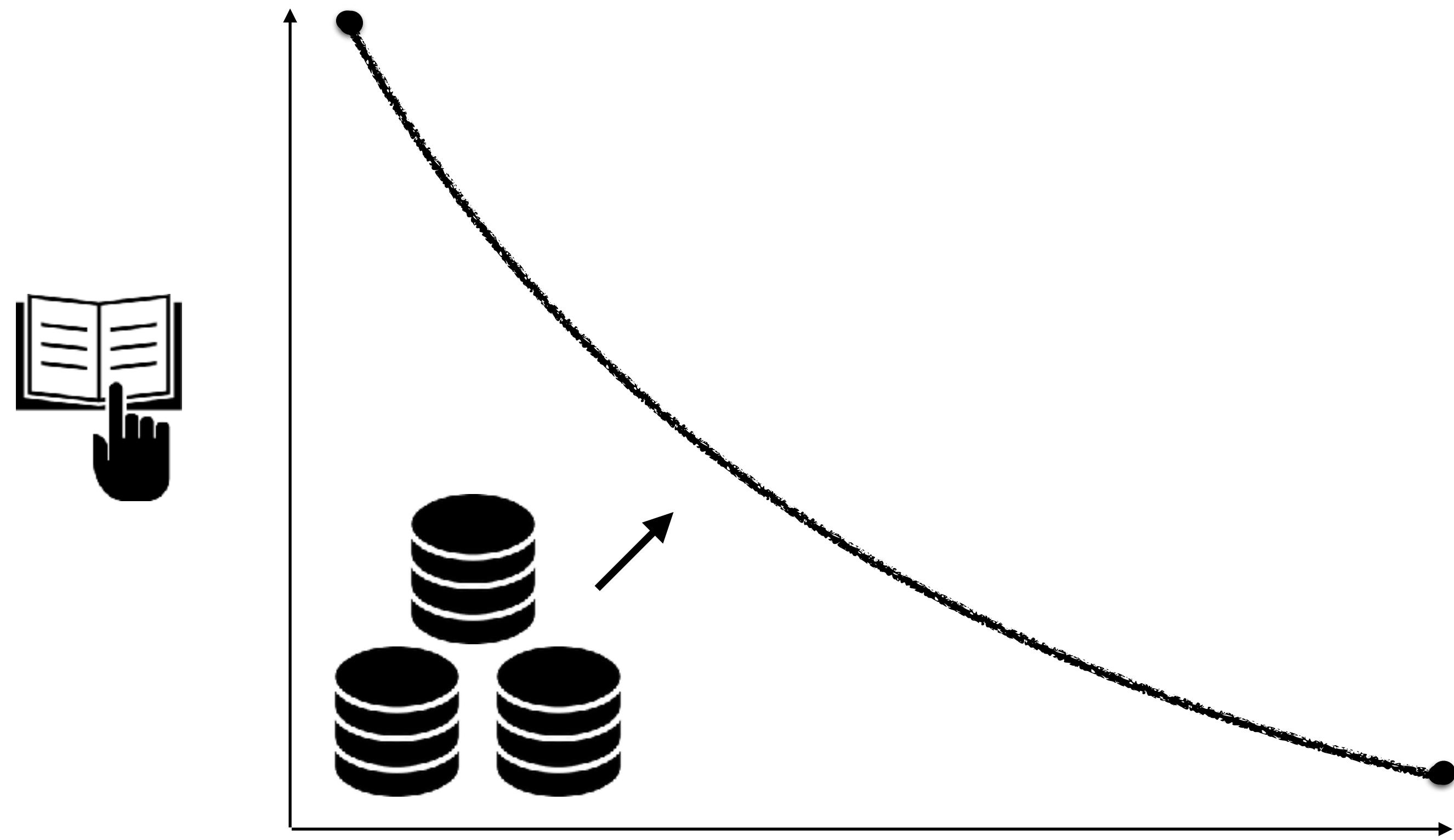


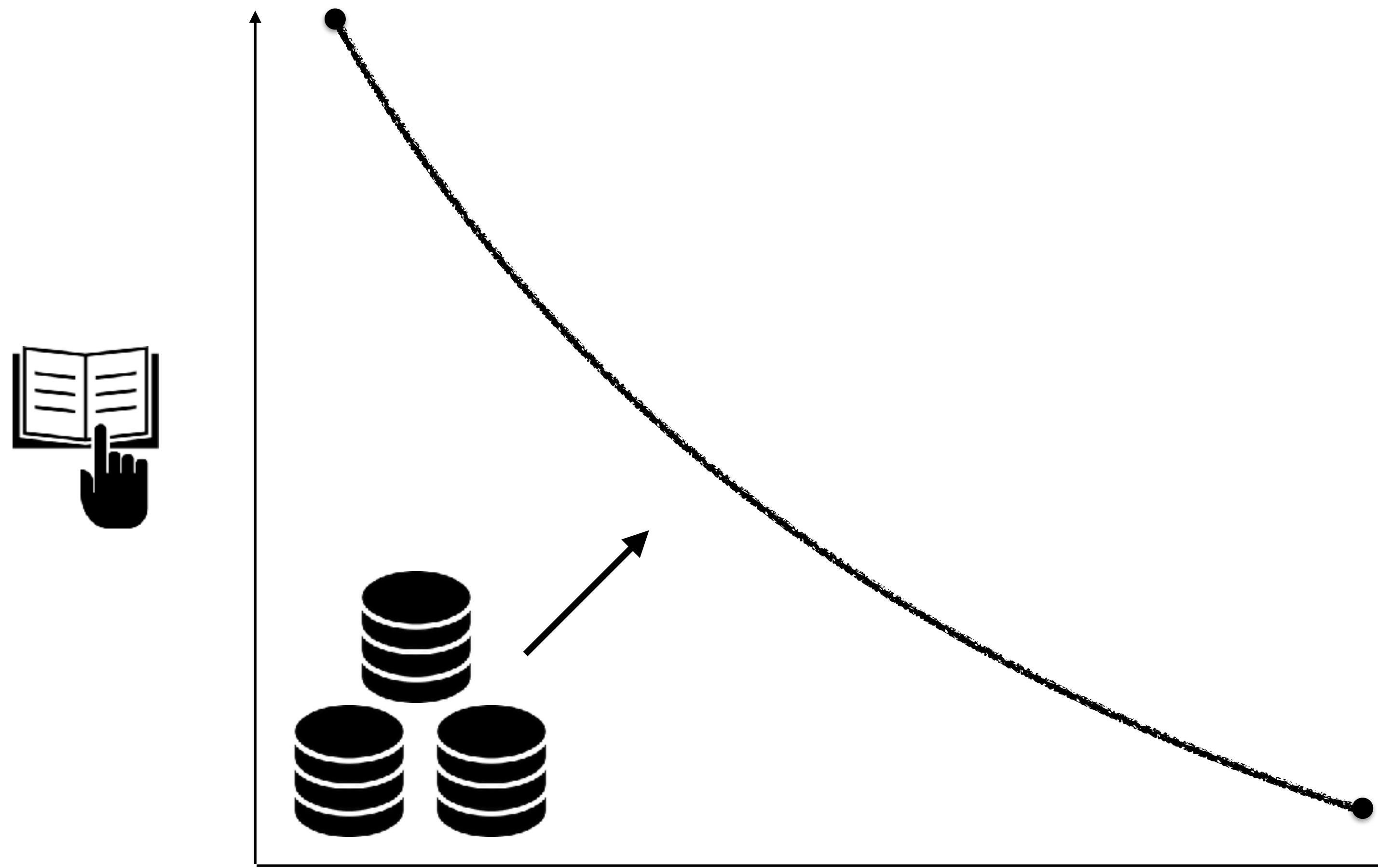
Worst-case: $O(M \cdot L)$

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

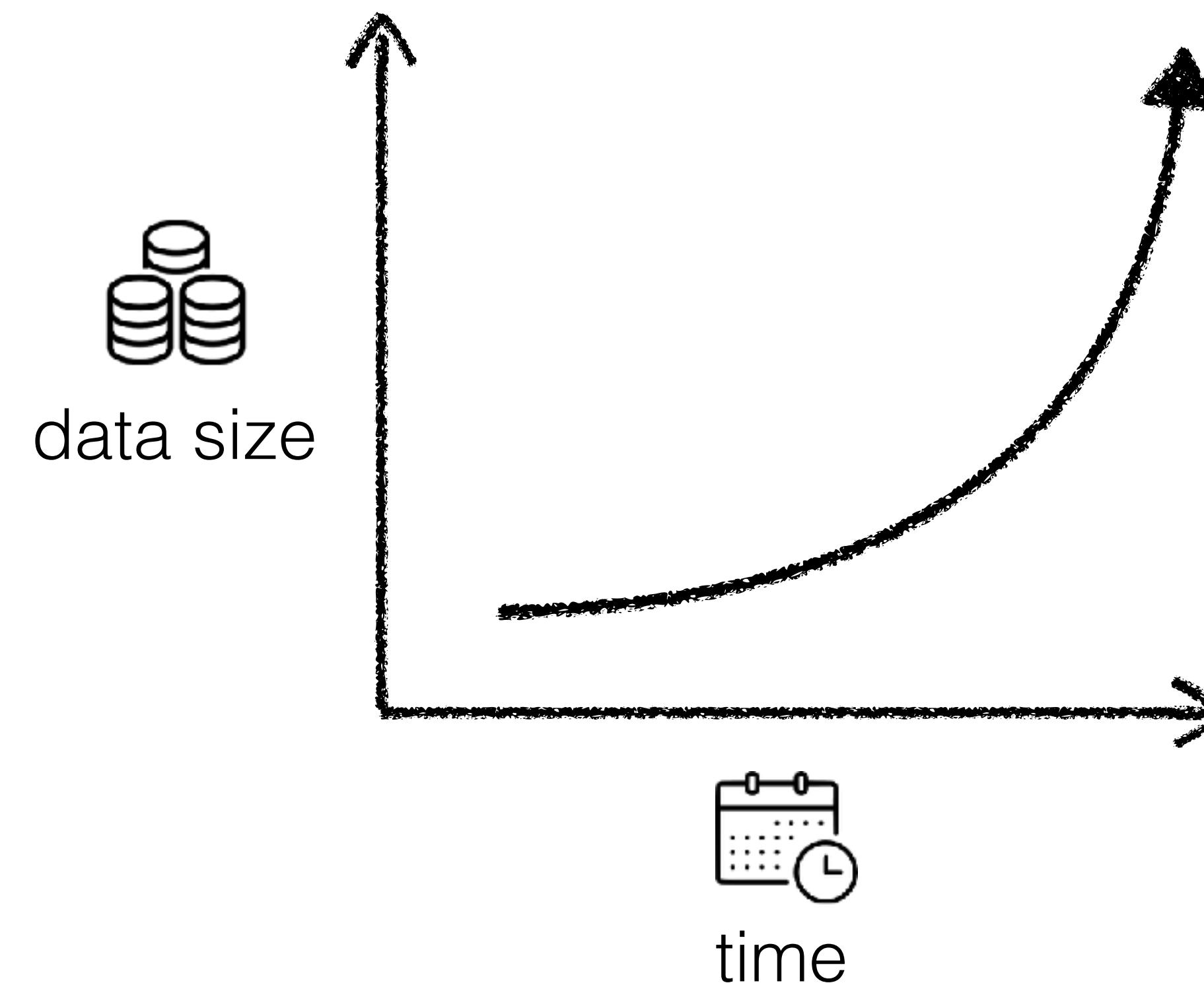








Can LSM-tree handle exponential data growth?





logarithmic scaling

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



logarithmic scaling

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

exponential growth

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



linear scaling

$O(\text{ time })$





Can we obtain sub-logarithmic scaling?

$O(\log N) \rightarrow O(\log \log N) \rightarrow O(1)$

get I/O
cost

$$O(e^{-M} \cdot L)$$



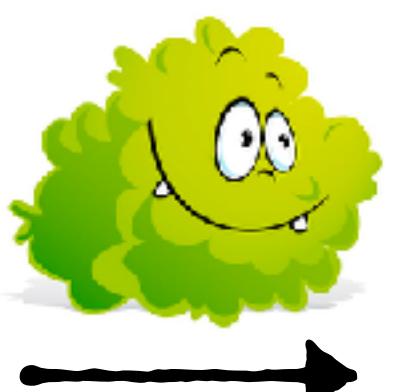
$$O(?)$$

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)

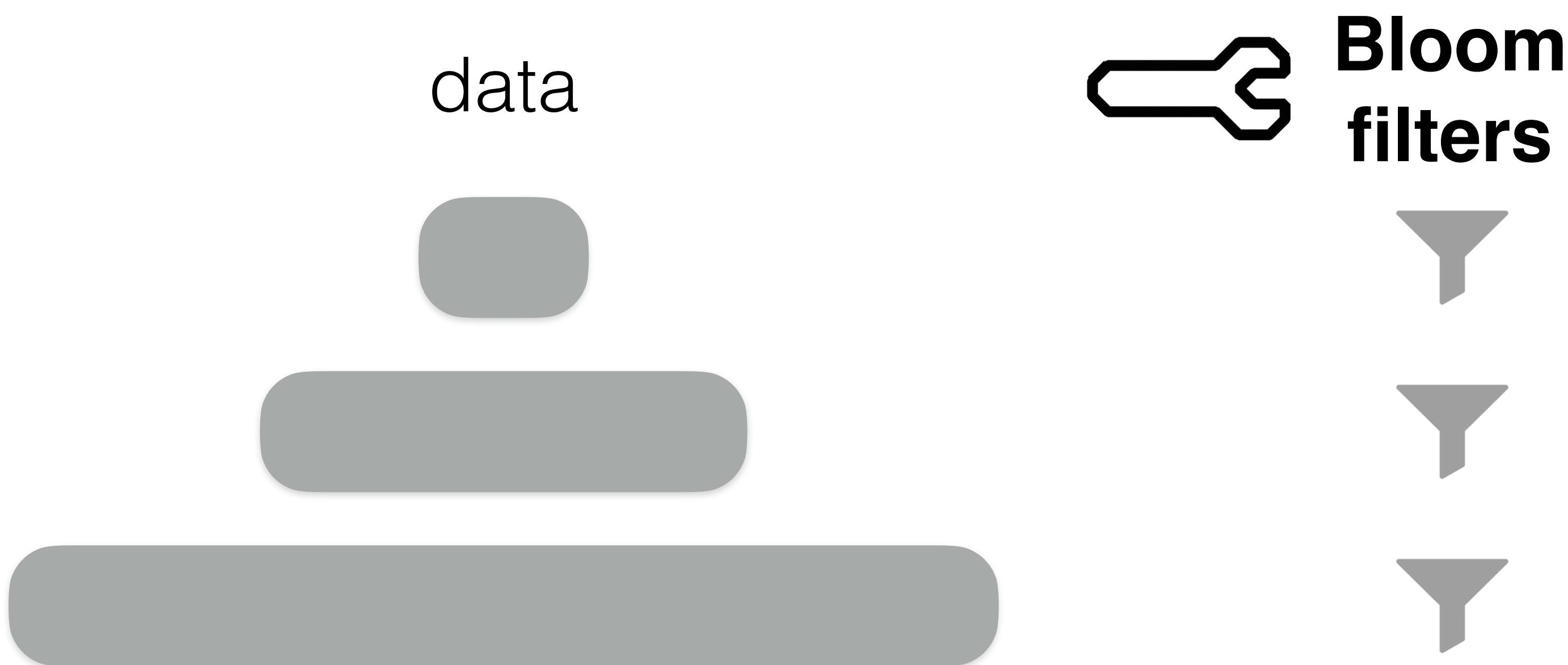
Monkey: Optimal Navigable **Key**-Value Store

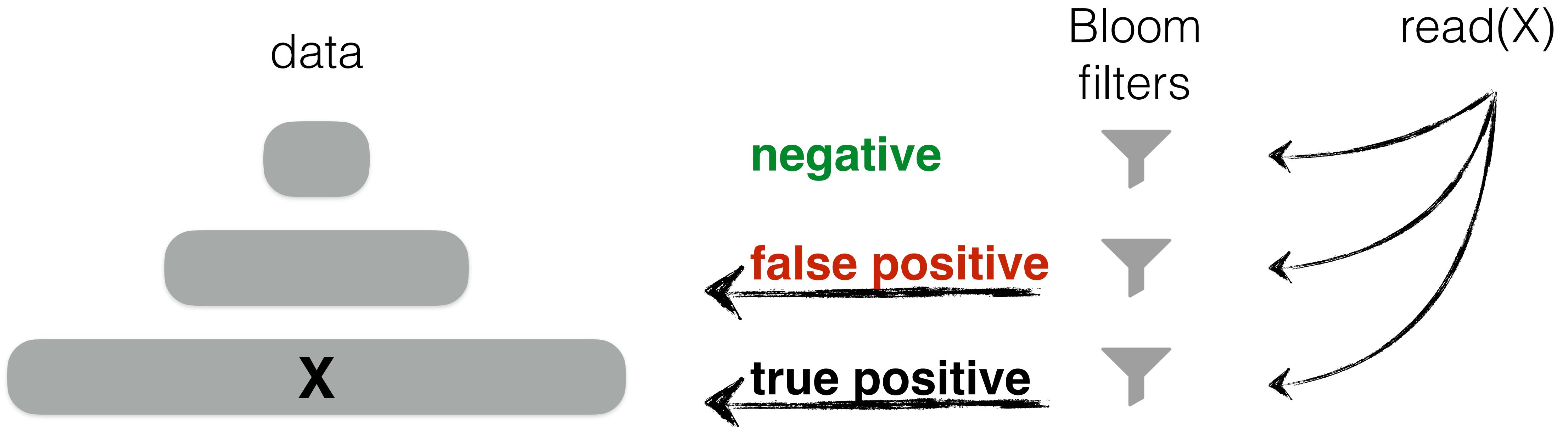
SIGMOD17



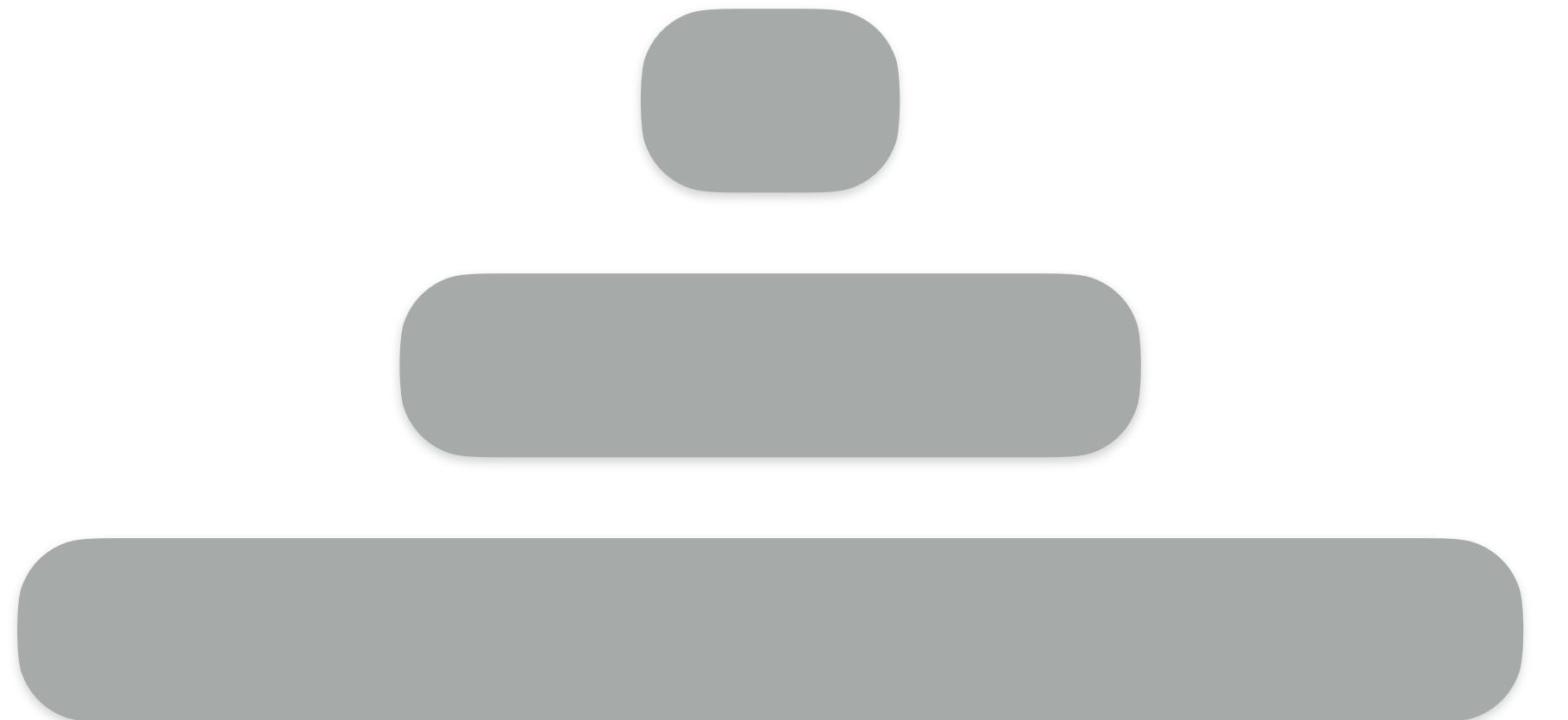
Monkey: Optimal Navigable **Key**-Value Store

SIGMOD17





data



Bloom
filters



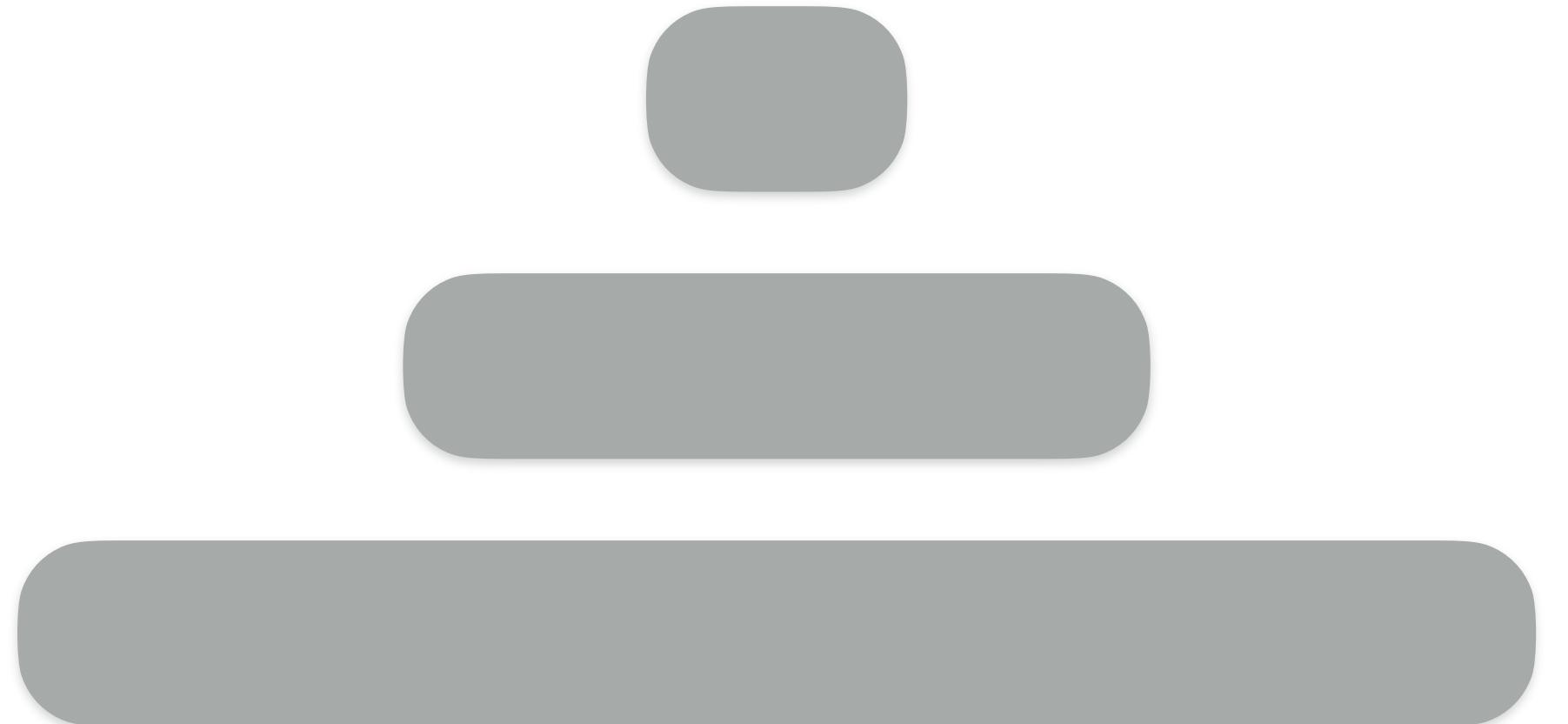
bits/entry

M

M

M

data



Bloom
filters



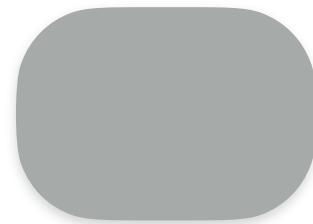
bits/entry

M

M

M

data



Bloom
filters



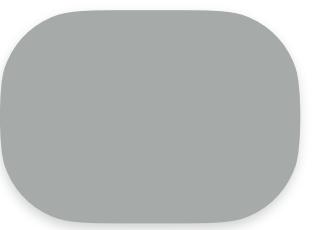
**false
positive rate**

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

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

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

data



Bloom
filters



**false
positive rate**

$$e^{-M}$$

$$e^{-M}$$

$$e^{-M}$$

Bloom
filters



false
positive rate

$$e^{-M}$$

$$e^{-M}$$

$$e^{-M}$$

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

Bloom
filters



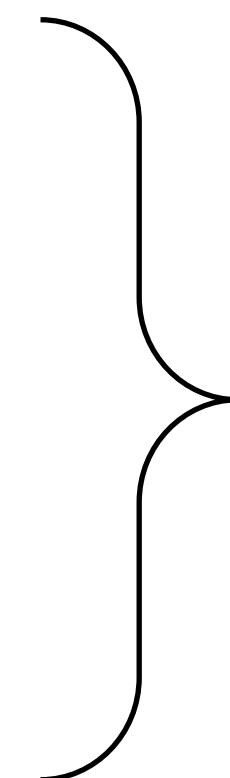
false
positive rate

$$e^{-M}$$

$$e^{-M}$$

$$e^{-M}$$

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



Bloom
filters



false
positive rate

$$e^{-M}$$

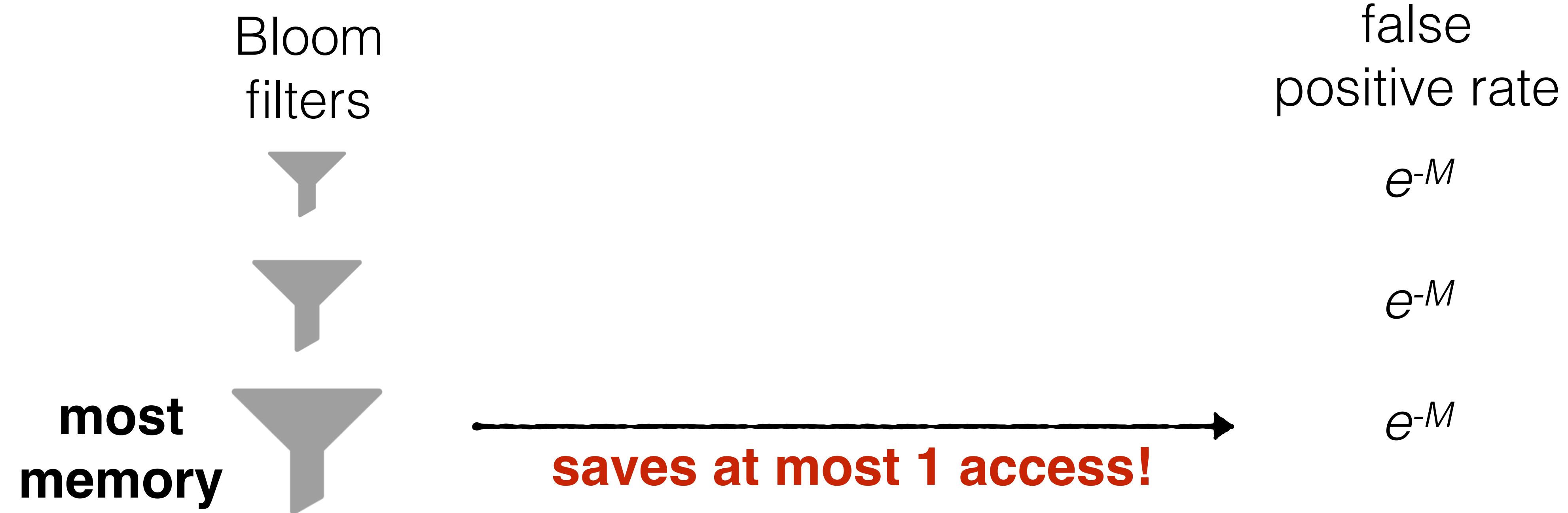


$$e^{-M}$$

**most
memory**



$$e^{-M}$$



bits / entry



$M + 2$

$M + 1$

$M - 1$

reallocates



false positive rates



$$e^{-(M + 2)} \quad \downarrow$$

$$e^{-(M + 1)} \quad \downarrow$$

$$e^{-(M - 1)} \quad \uparrow$$



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

$$\text{read cost} = \sum_1^L p_i$$

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

relax
false positive rates

$$0 < p_0 < 1$$

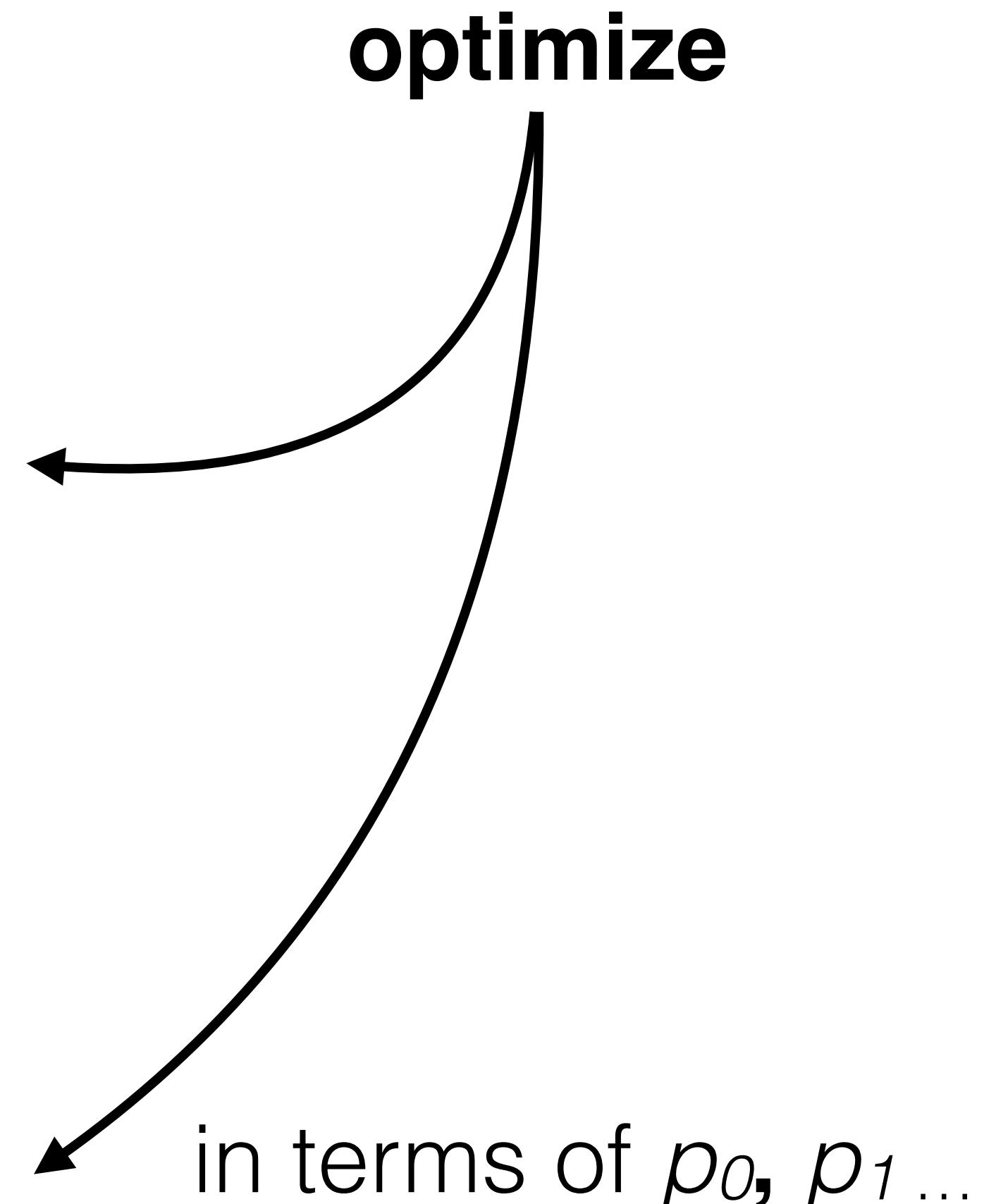
$$0 < p_1 < 1$$

$$0 < p_2 < 1$$

model

read cost = $\sum_1^L p_i$

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





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



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



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



$$\left. \begin{array}{l} e^{-M} / R^2 \\ e^{-M} / R^1 \\ e^{-M} / R^0 \end{array} \right\}$$

**geometric
progression**
= $O(e^{-M})$

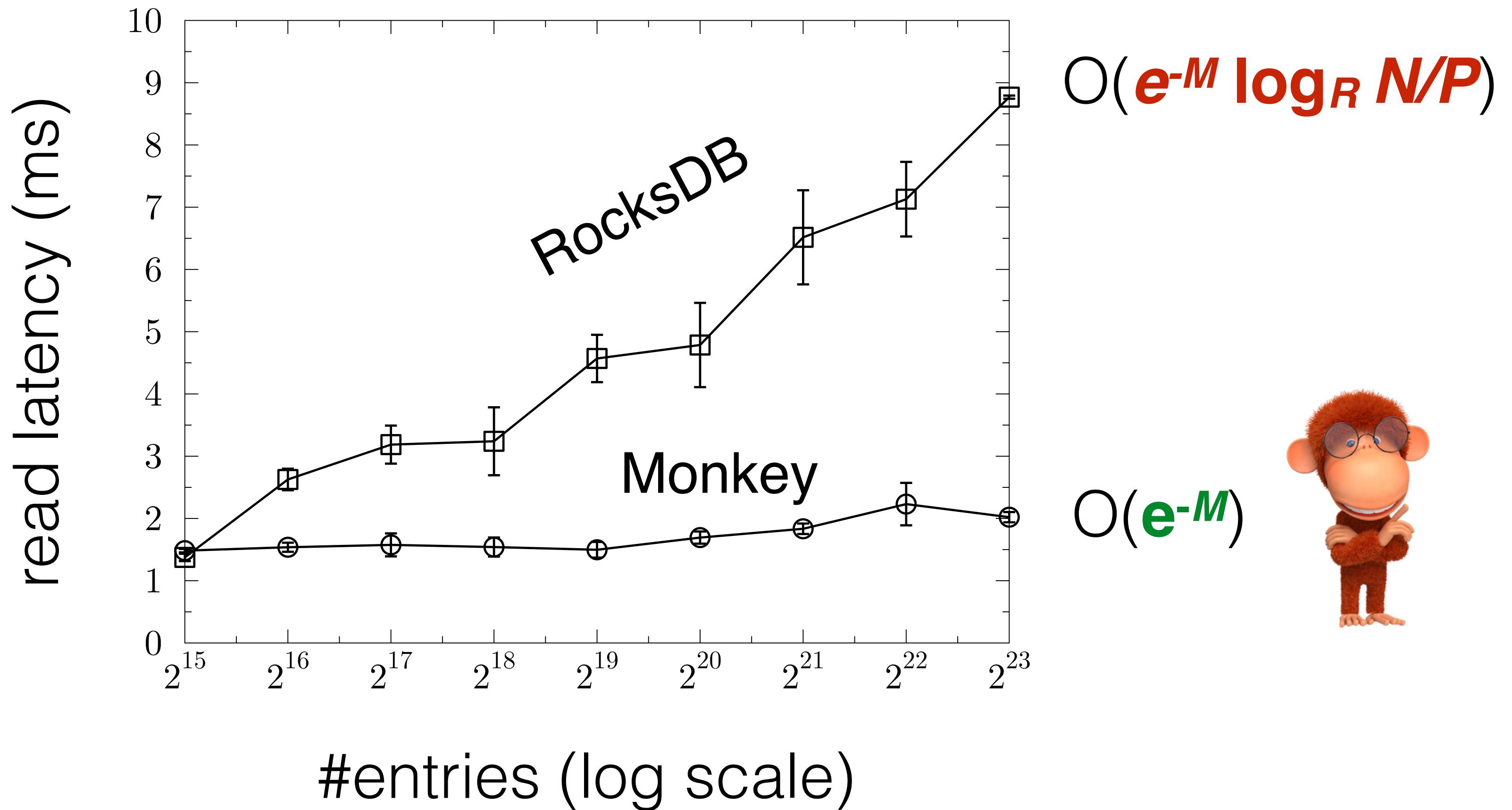


Faster worst case

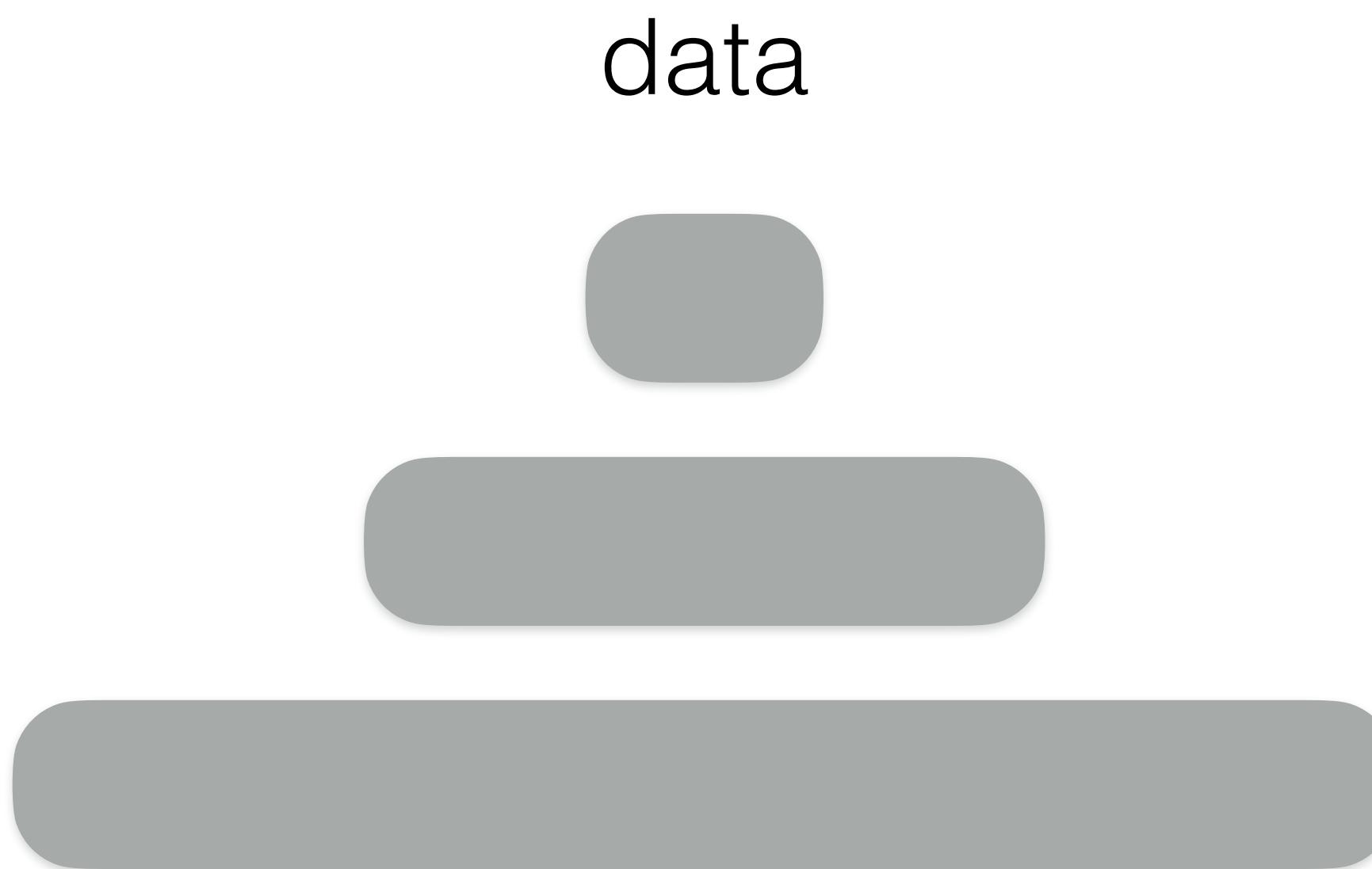
$$O(\textcolor{green}{e^{-M}}) < O(\textcolor{red}{e^{-M} \log_R N/P})$$

Configuration

buffer 2MB
bits/entry: 5
size ratio: 2
1KB entries
queries to missing keys
hard disk storage

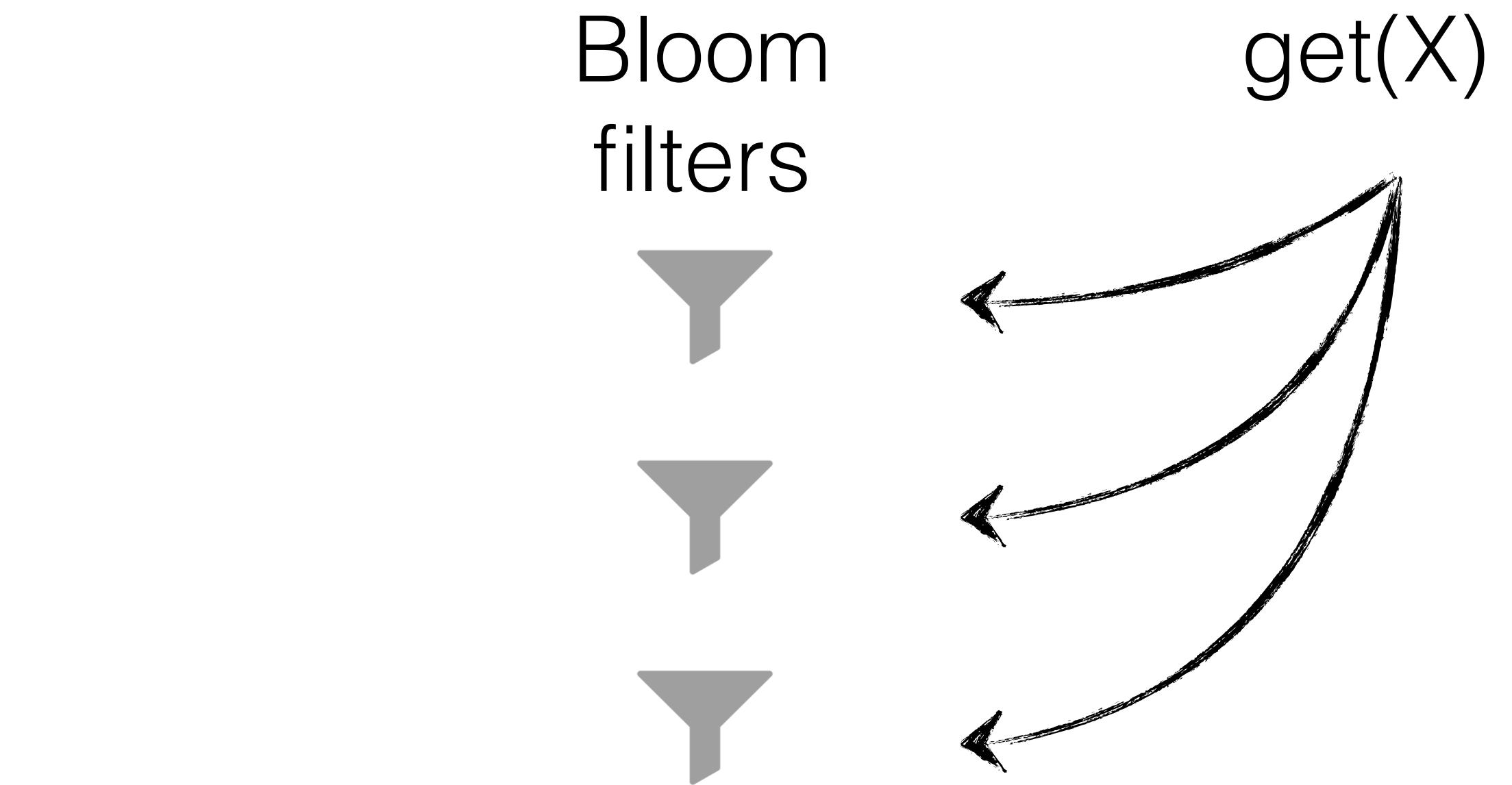


Analyze filter access with Monkey



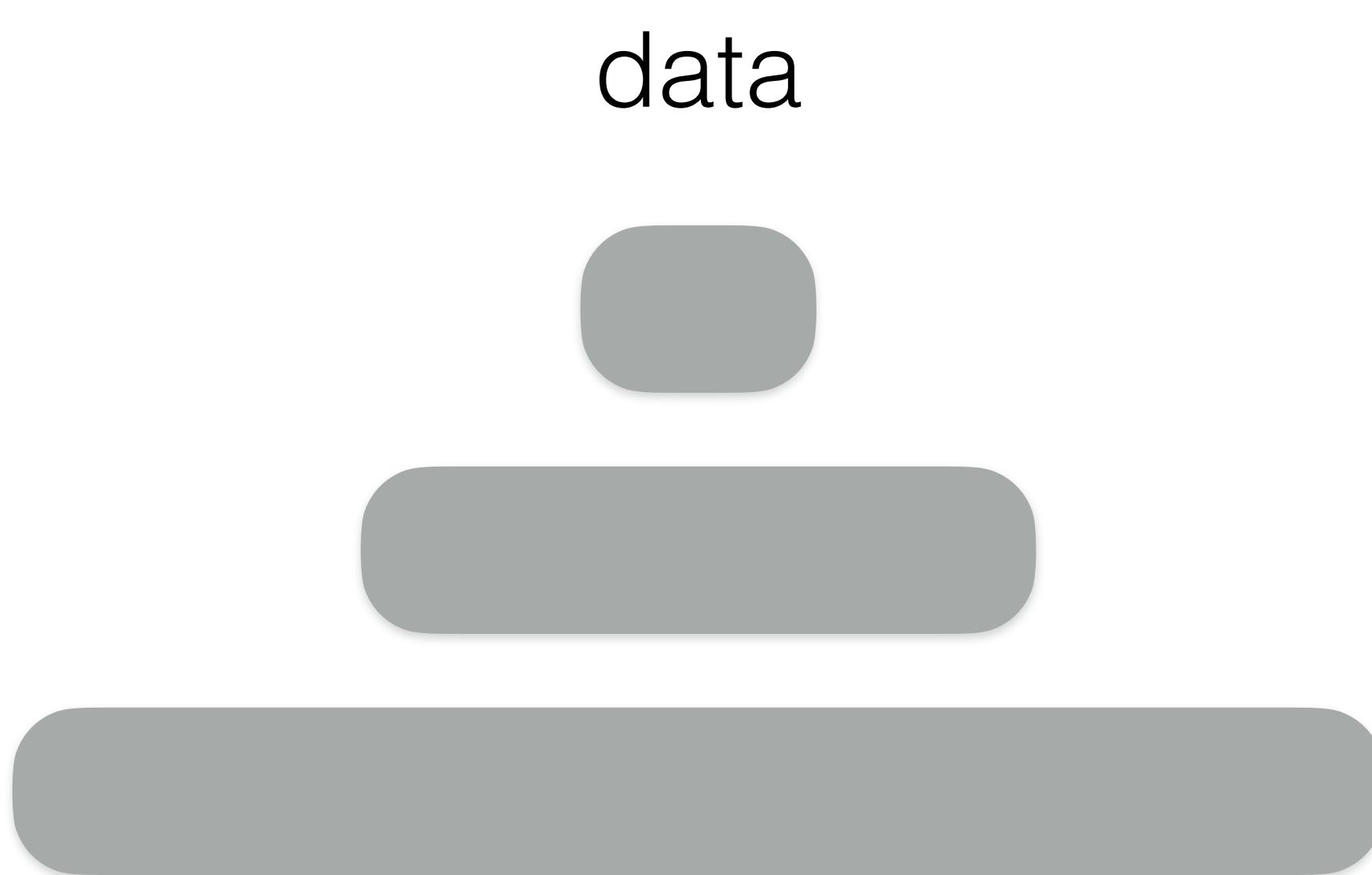
Worst-case:

$$\text{Positive Query} = M * \ln(2)$$
$$\text{Avg. Negative Query} = 2$$



Avg. worst-case:

Analyze filter access with Monkey



**# hash
functions**

$O(M+2)$

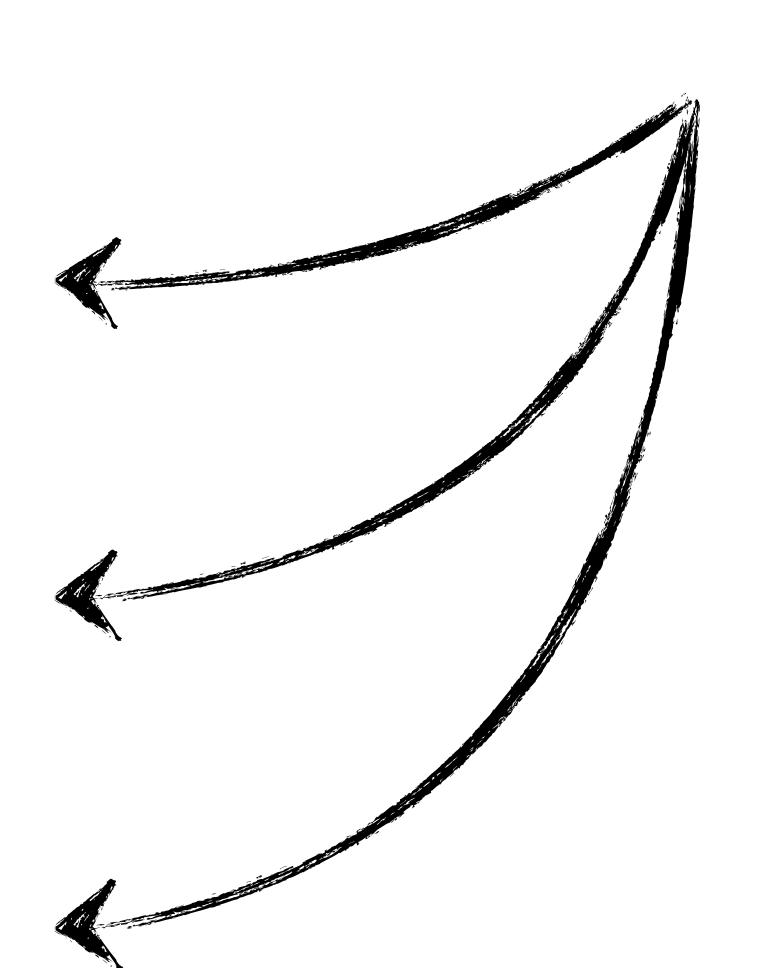
$O(M+1)$

$O(M)$

Bloom
filters



get(X)



Worst-case:

Avg. worst-case:

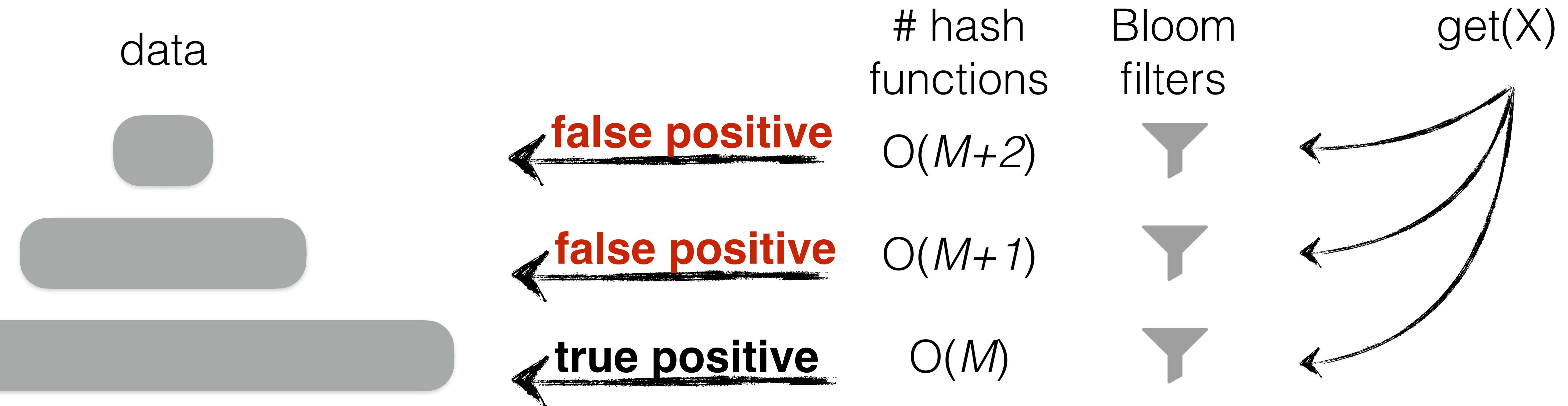
Positive Query = $M * \ln(2)$

Avg. Negative Query = 2

Analyze filter access with Monkey

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

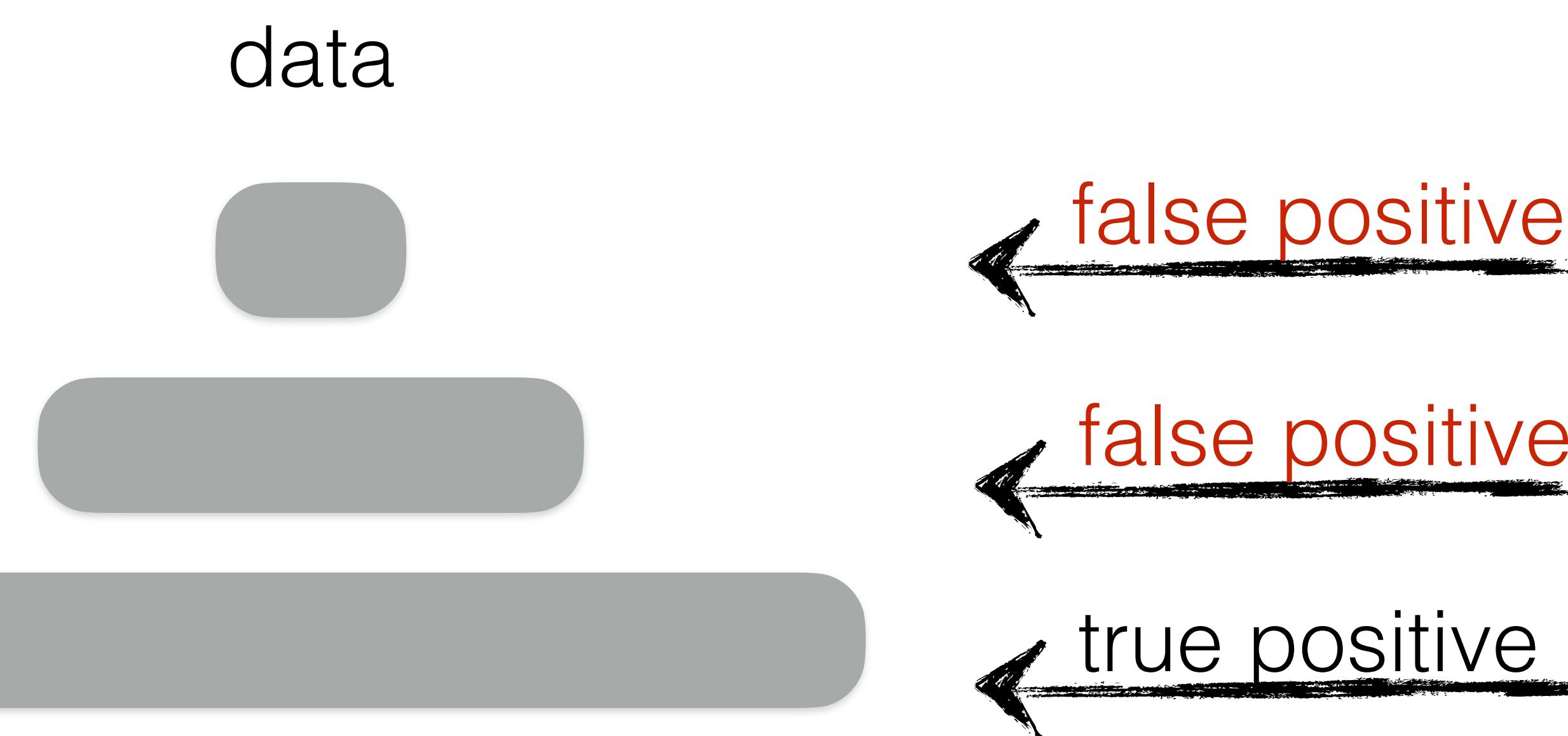
Avg. Negative Query = 2



Worst-case:

Avg. worst-case:

Analyze filter access with Monkey



hash
functions

$O(M+2)$

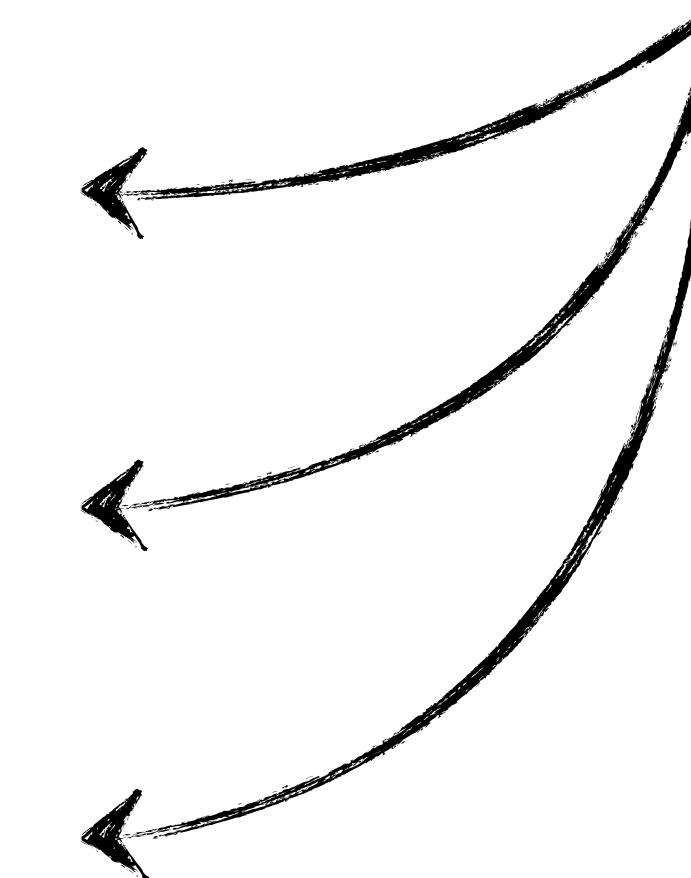
$O(M+1)$

$O(M)$

Bloom
filters



get(X)



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

Avg. worst-case:

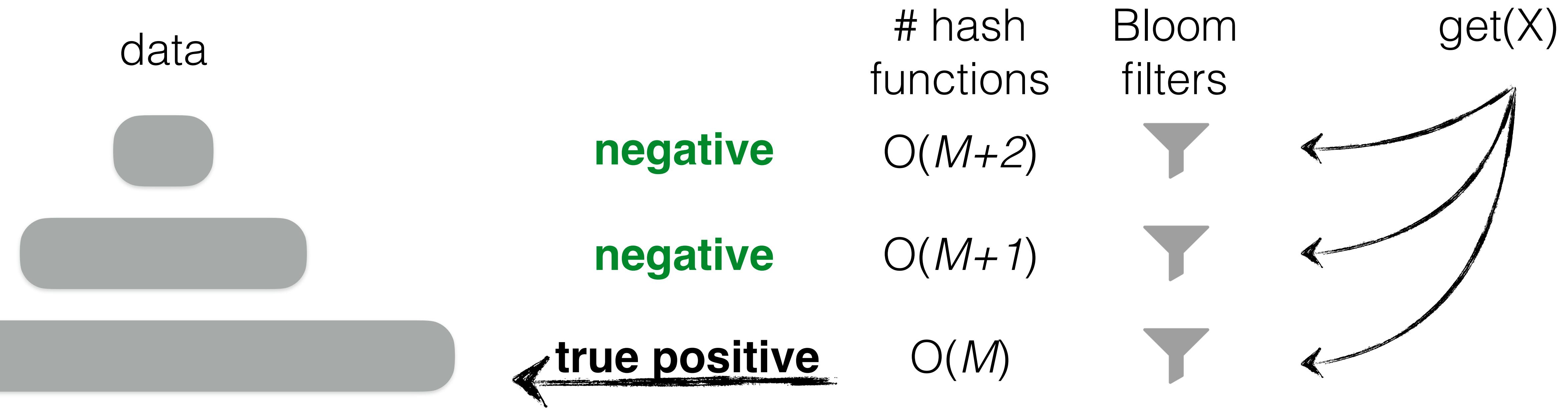
$$\text{Positive Query} = M * \ln(2)$$

$$\text{Avg. Negative Query} = 2$$

Analyze filter access with Monkey

$$\text{Positive Query} = M^* \ln(2)$$

$$\text{Avg. Negative Query} = 2$$



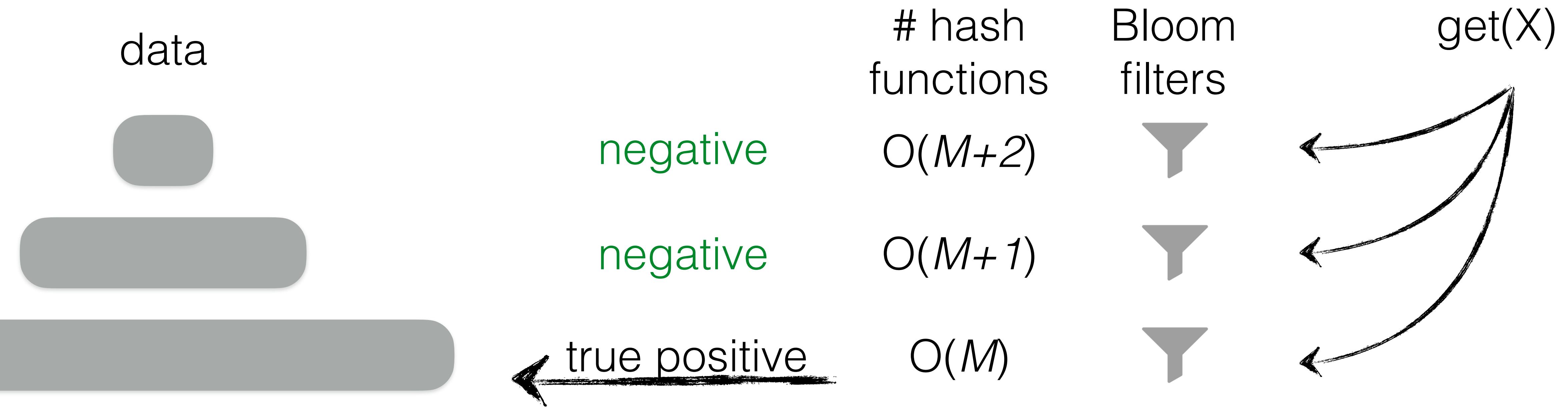
Worst-case: $O(M \cdot L + L^2)$

Avg. worst-case:

Analyze filter access with Monkey

$$\text{Positive Query} = M^* \ln(2)$$

$$\text{Avg. Negative Query} = 2$$



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

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

get I/O
cost

$$O(e^{-M} \cdot L)$$



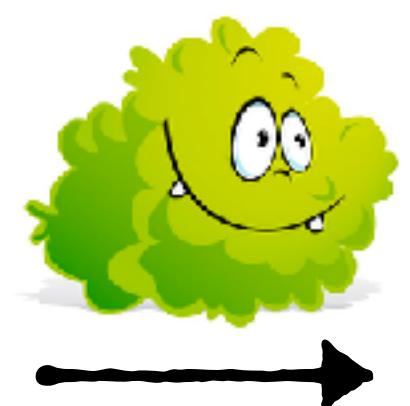
$$O(e^{-M})$$

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

$$O(e^{-M} \cdot L)$$



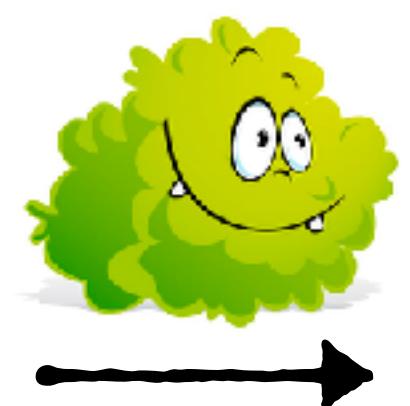
$$O(e^{-M})$$

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 **E**volvable **S**calable **K**ey-Value Store

reads & writes cost breakdown



reads

false positive rates

**exponentially
decreasing**



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

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

$$e^{-M}$$



false positive rates

reads



e^{-M}/R^2

e^{-M}/R

largest level

reads

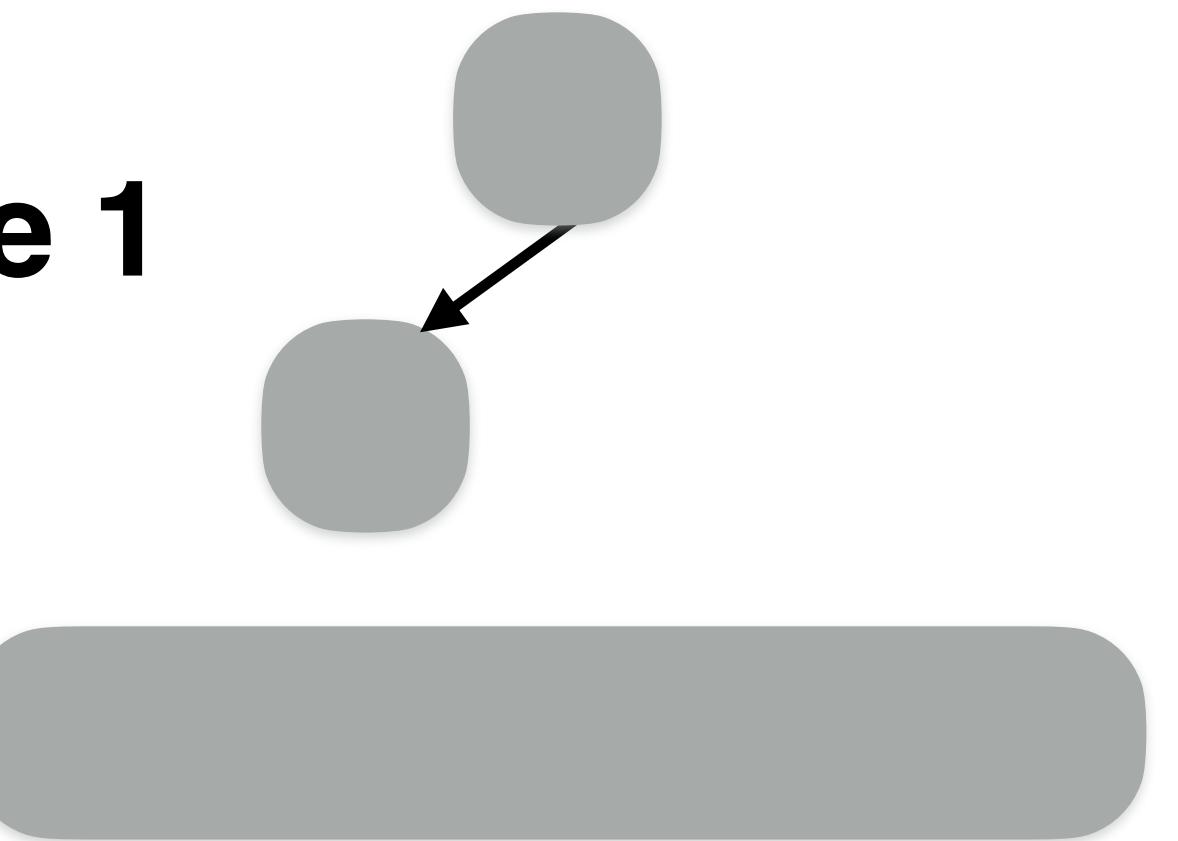
$O(e^{-M})$

writes

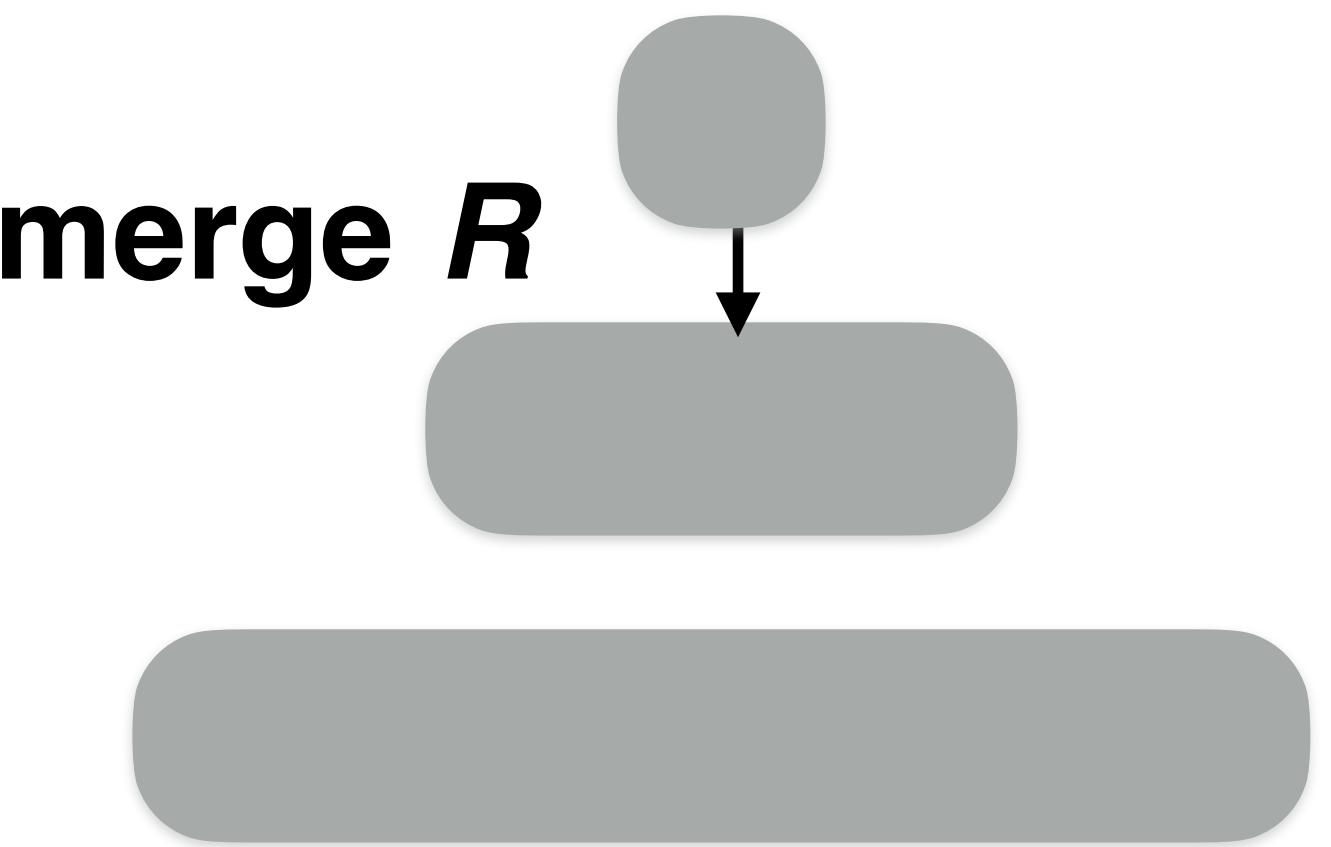


writes

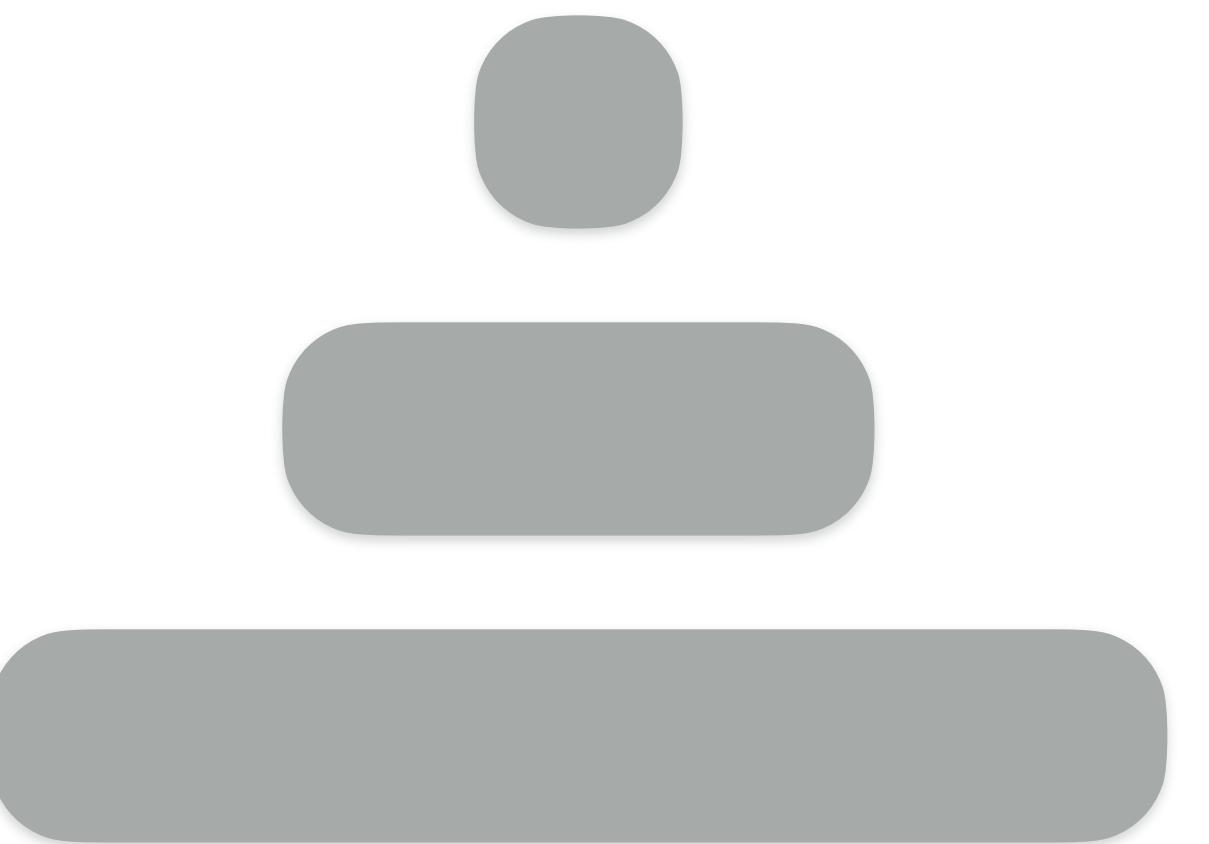
merge 1



writes



writes



$O(\textcolor{red}{R/B})$

$O(\textcolor{red}{R/B})$

$O(\textcolor{red}{R/B})$

$\left. \right\}$

$O((R \cdot L)/B)$

reads

$$O(e^{-M})$$

=

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

+

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

+

$$e^{-M}$$

writes

$$O((R \cdot L)/B)$$

=

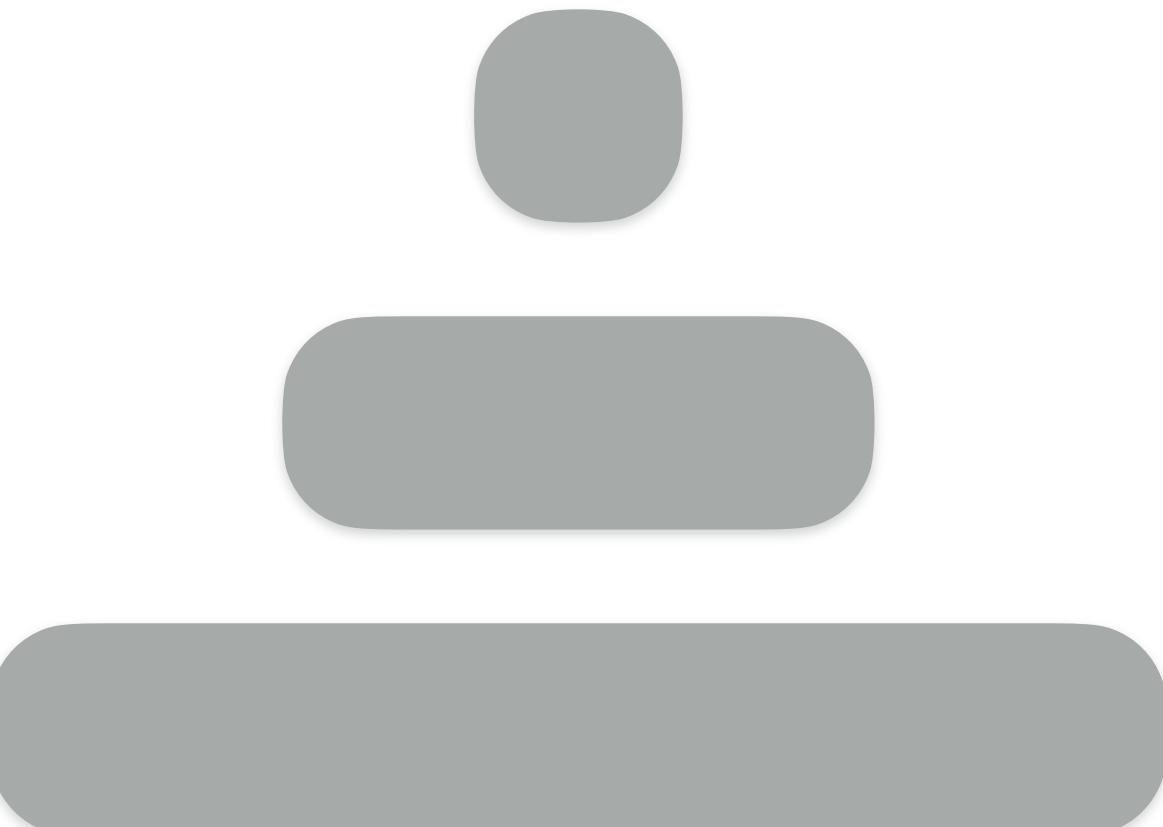
$$O(R/B)$$

+

$$O(R/B)$$

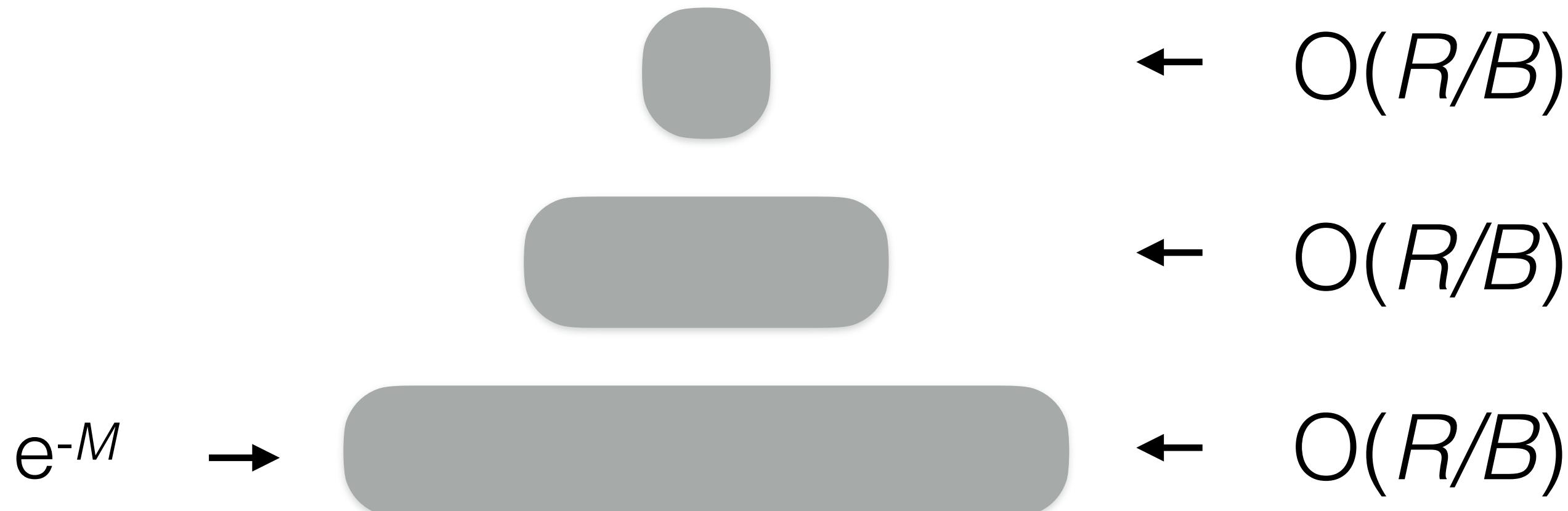
+

$$O(R/B)$$



reads
largest level

writes
all levels



reads

writes



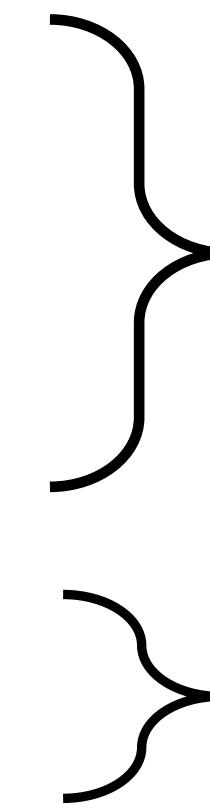
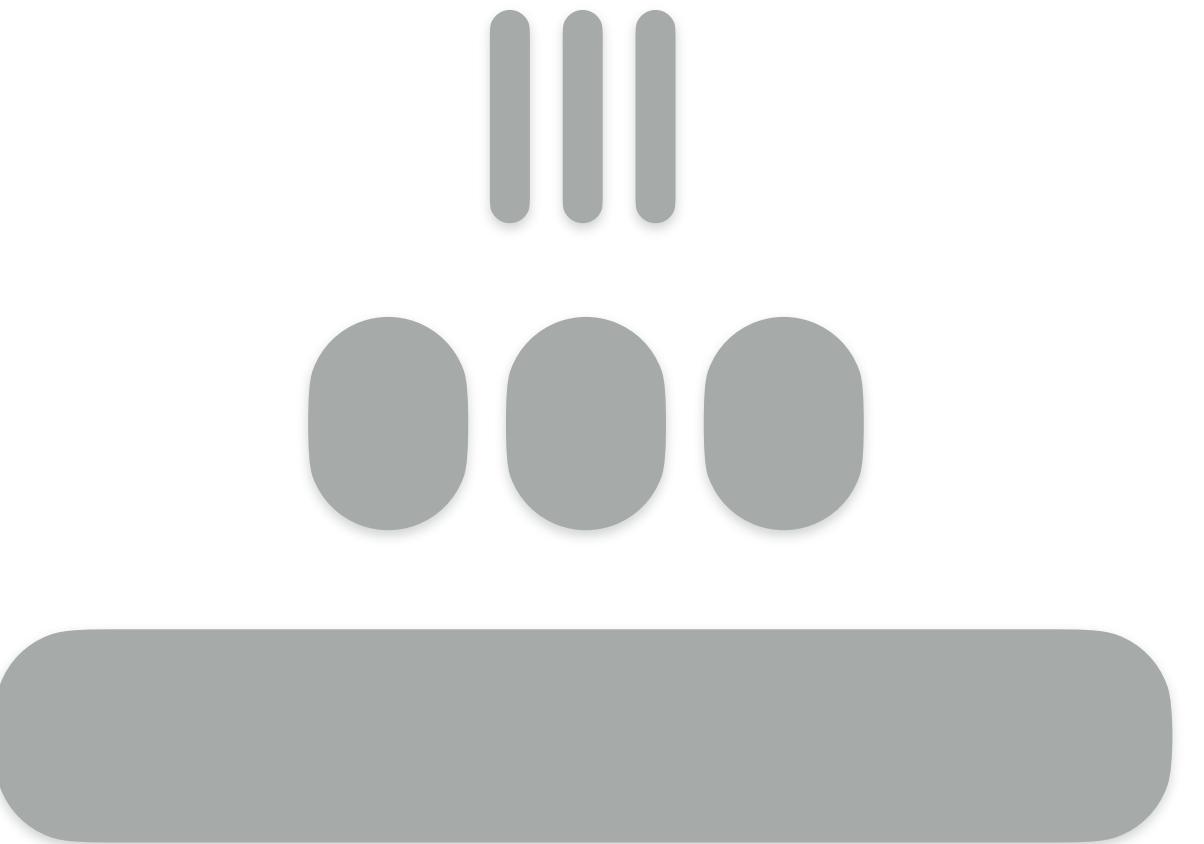
reads

writes



make lazy

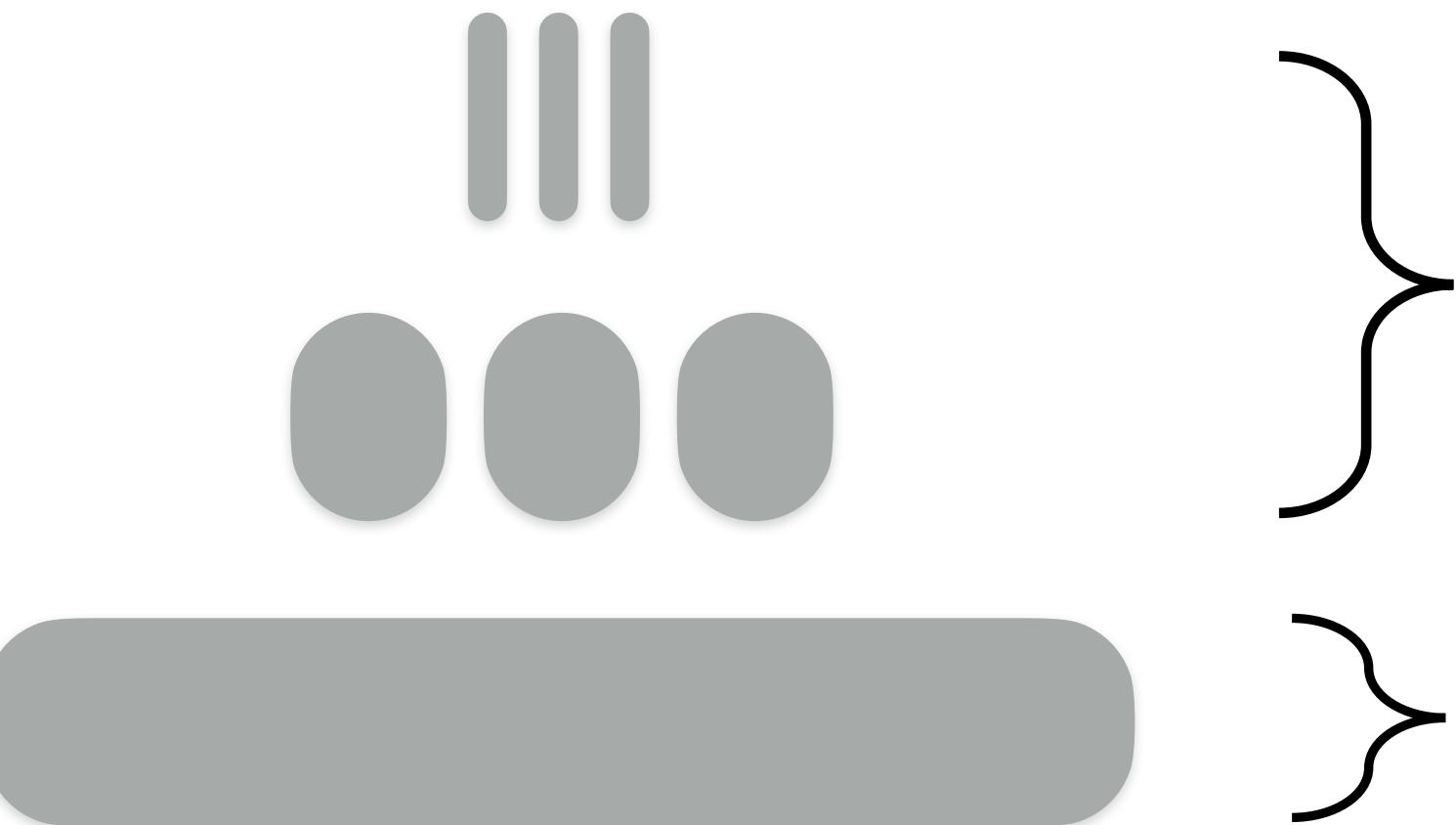
Dostoevsky



lazy

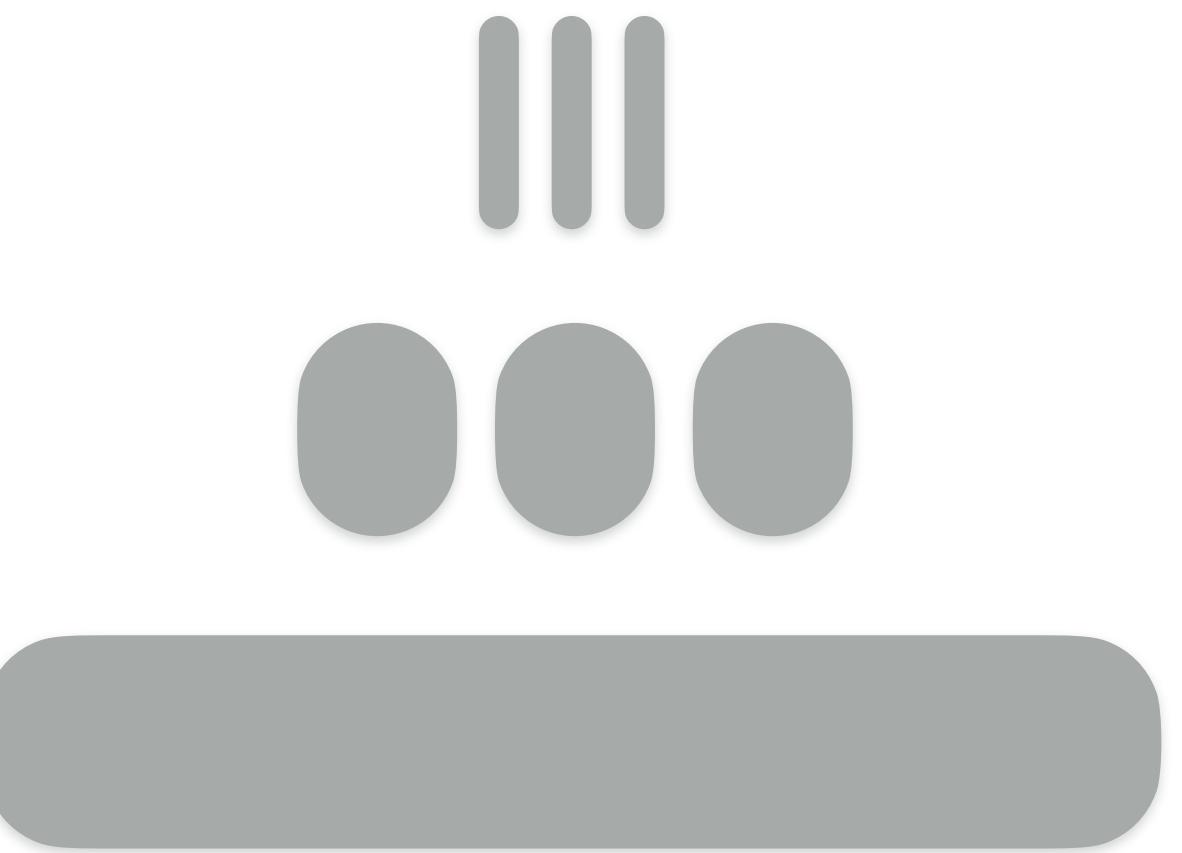
greedy

Dostoevsky



merge when
level fills up

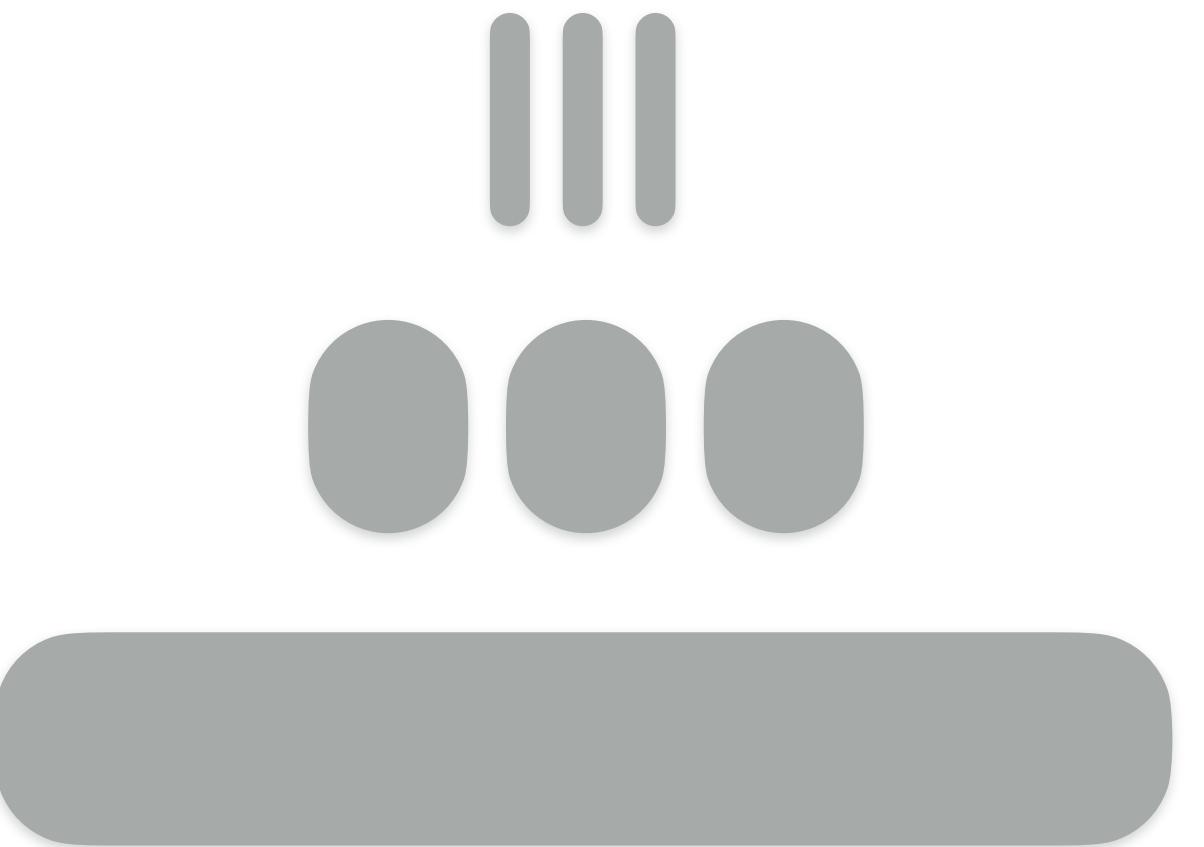
Dostoevsky



merge when
new run comes in

reads

writes

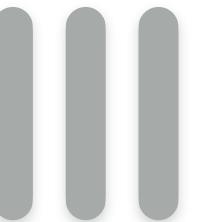


reads

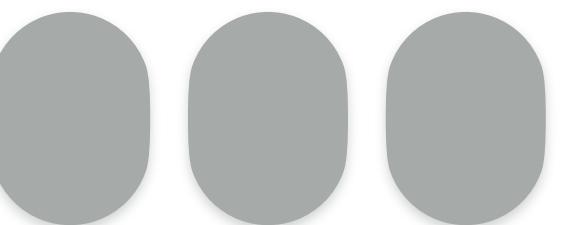
writes

false positive rates

e^{-M}/R^3



e^{-M}/R^2



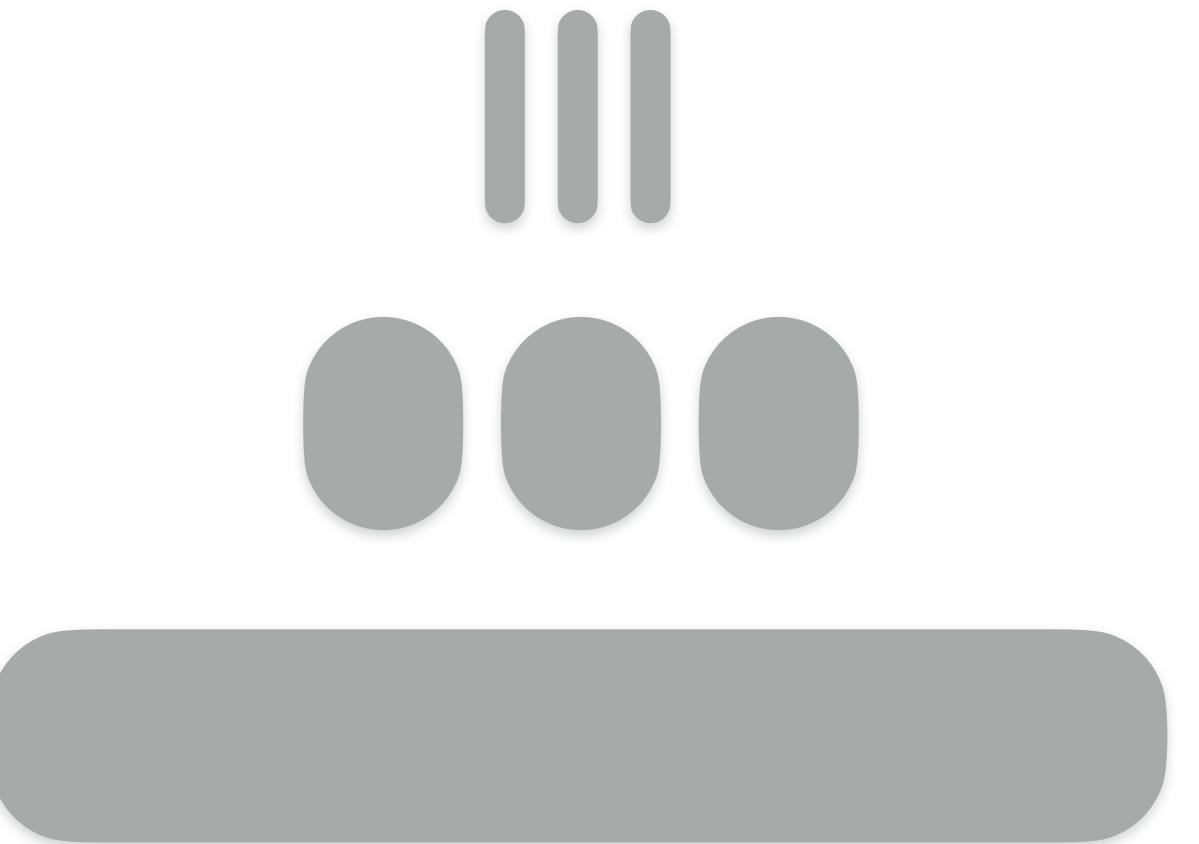
e^{-M}



reads

$O(e^{-M})$

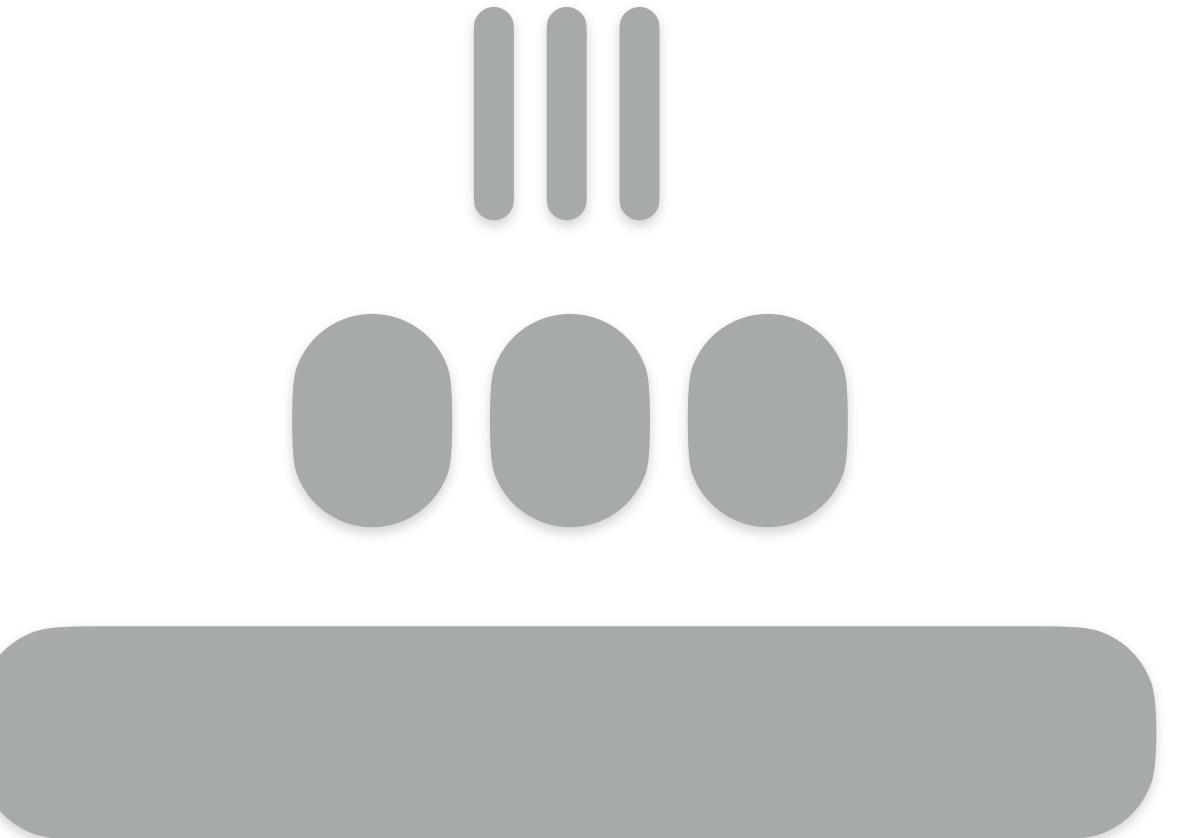
writes



reads

$O(e^{-M})$

writes



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

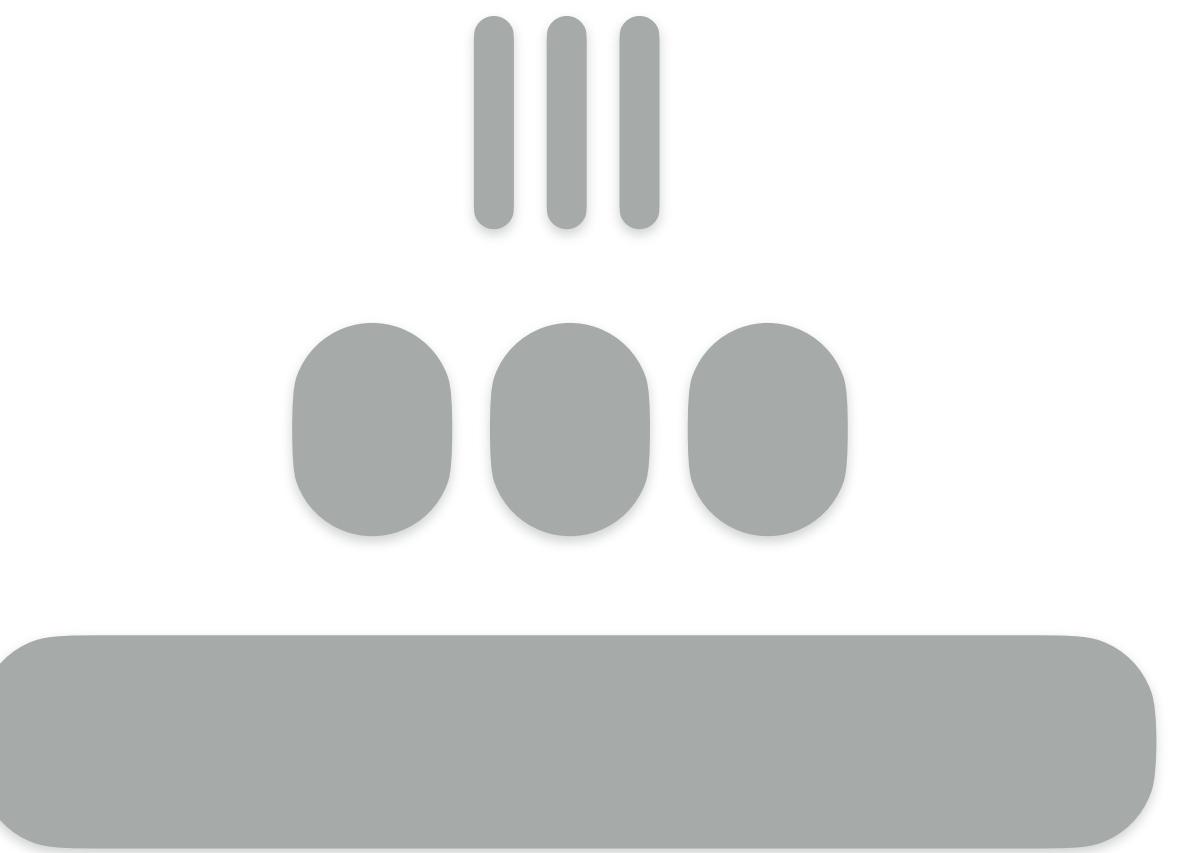
$O(\mathbf{1/B})$

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

reads

$$O(e^{-M})$$

writes



$$O(1/B)$$

$$O(1/B)$$

$$O(R/B)$$

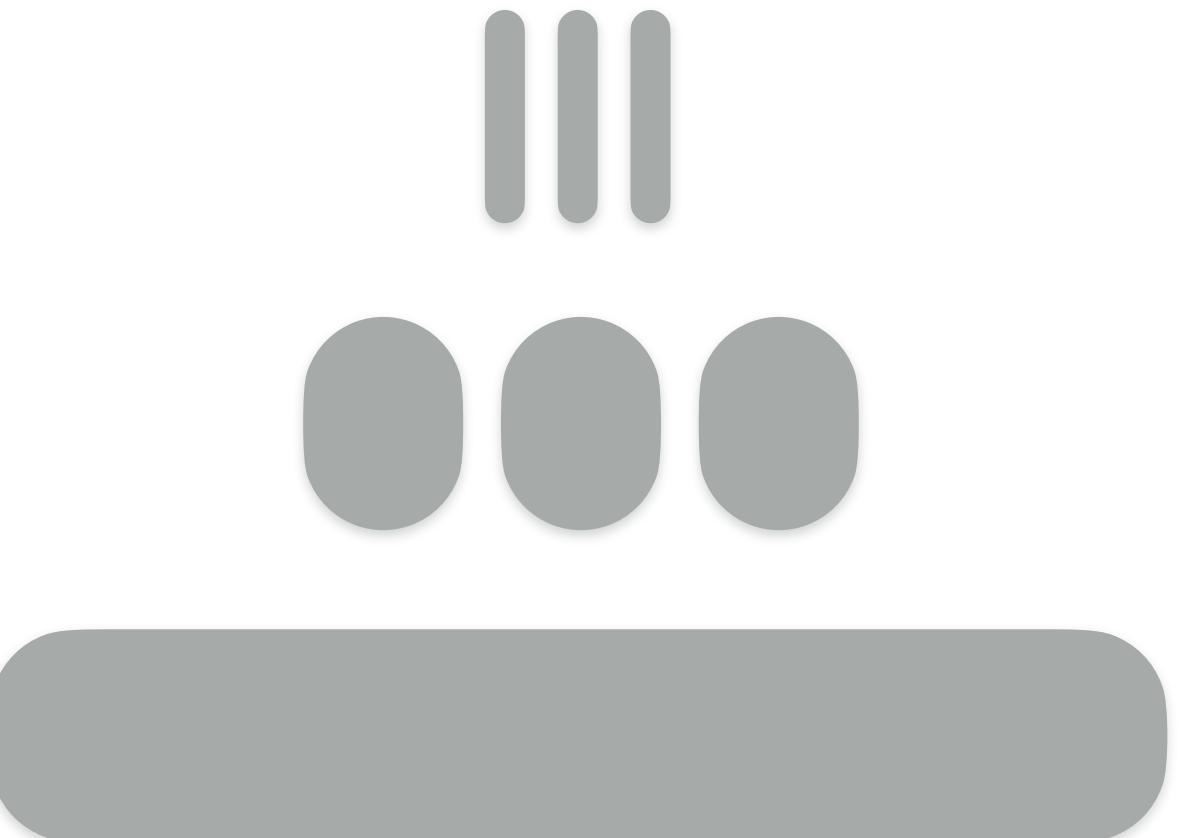
$$O((R + L)/B)$$

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

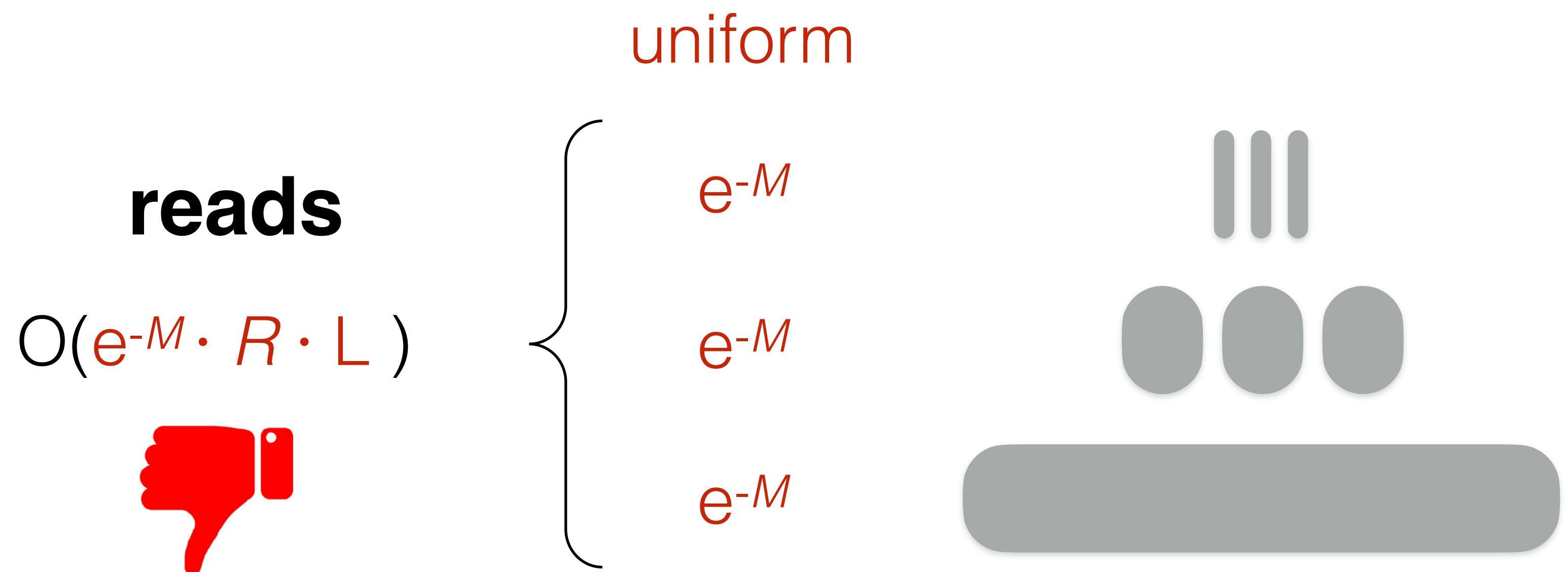
$$O(e^{-M})$$

writes

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



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



Dostoevsky

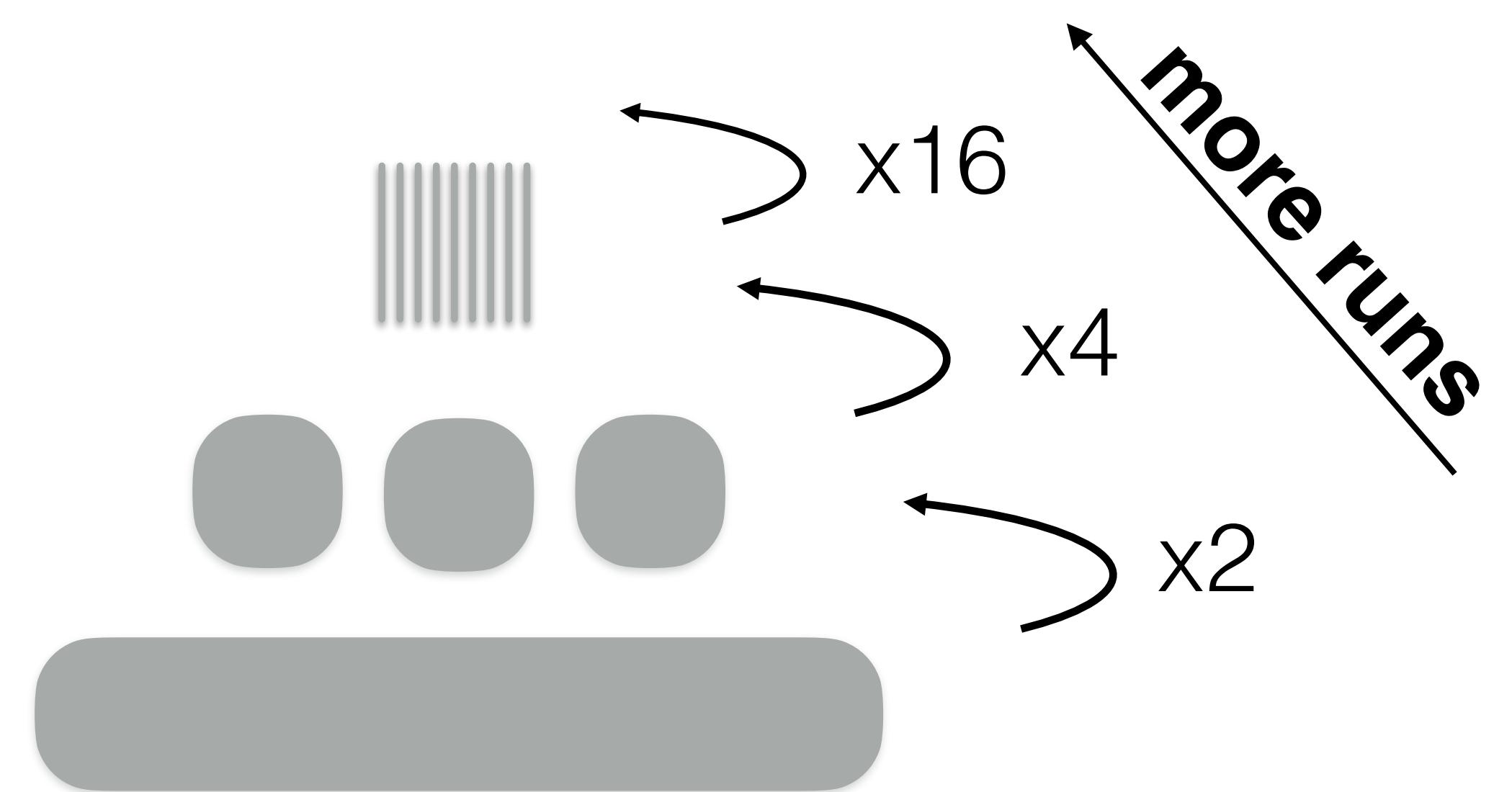
$O((R + L)/B)$



SIGMOD 18

LSM-Bush

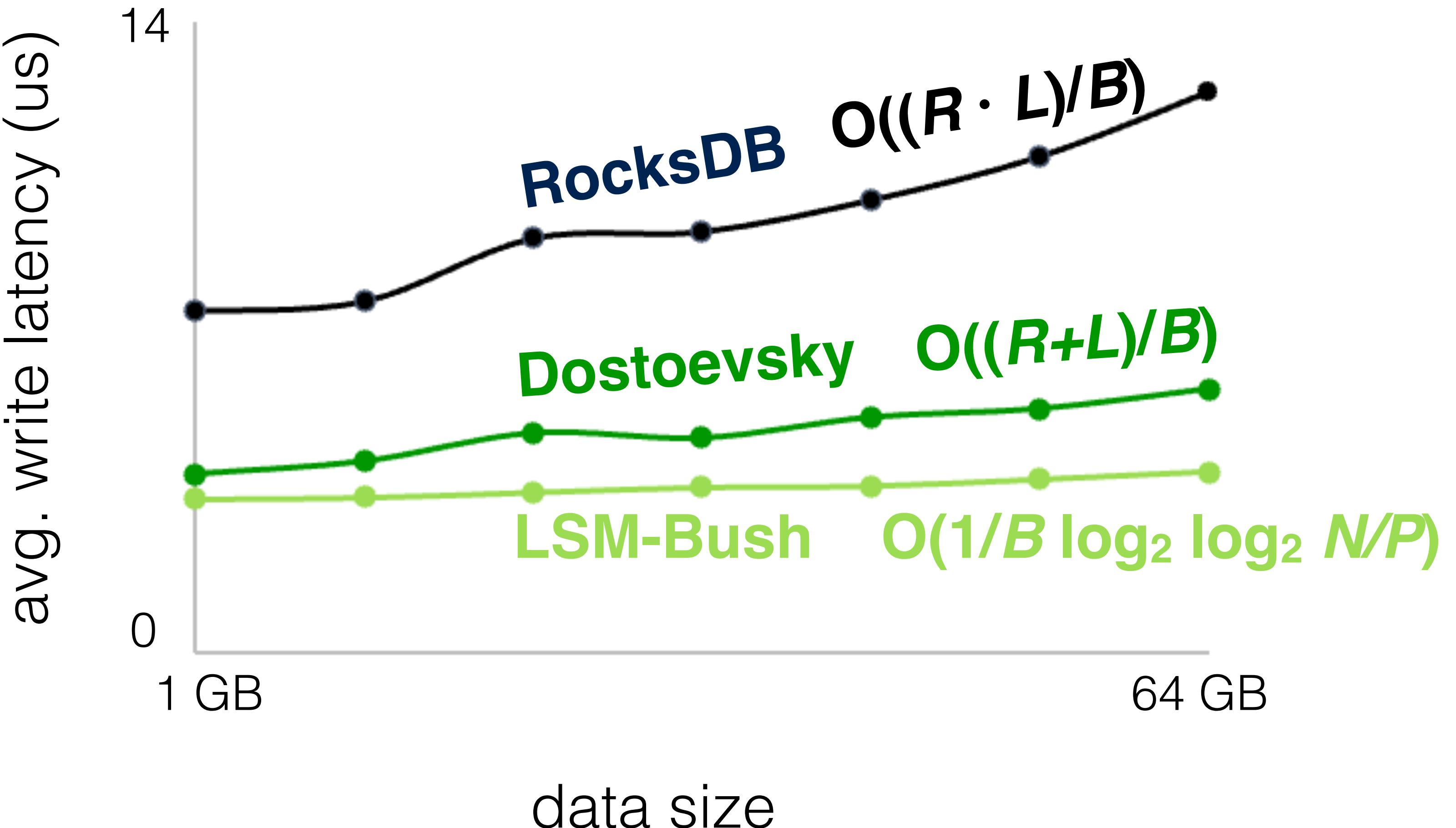
$O(1/B \log_2 \log_2 N/P)$

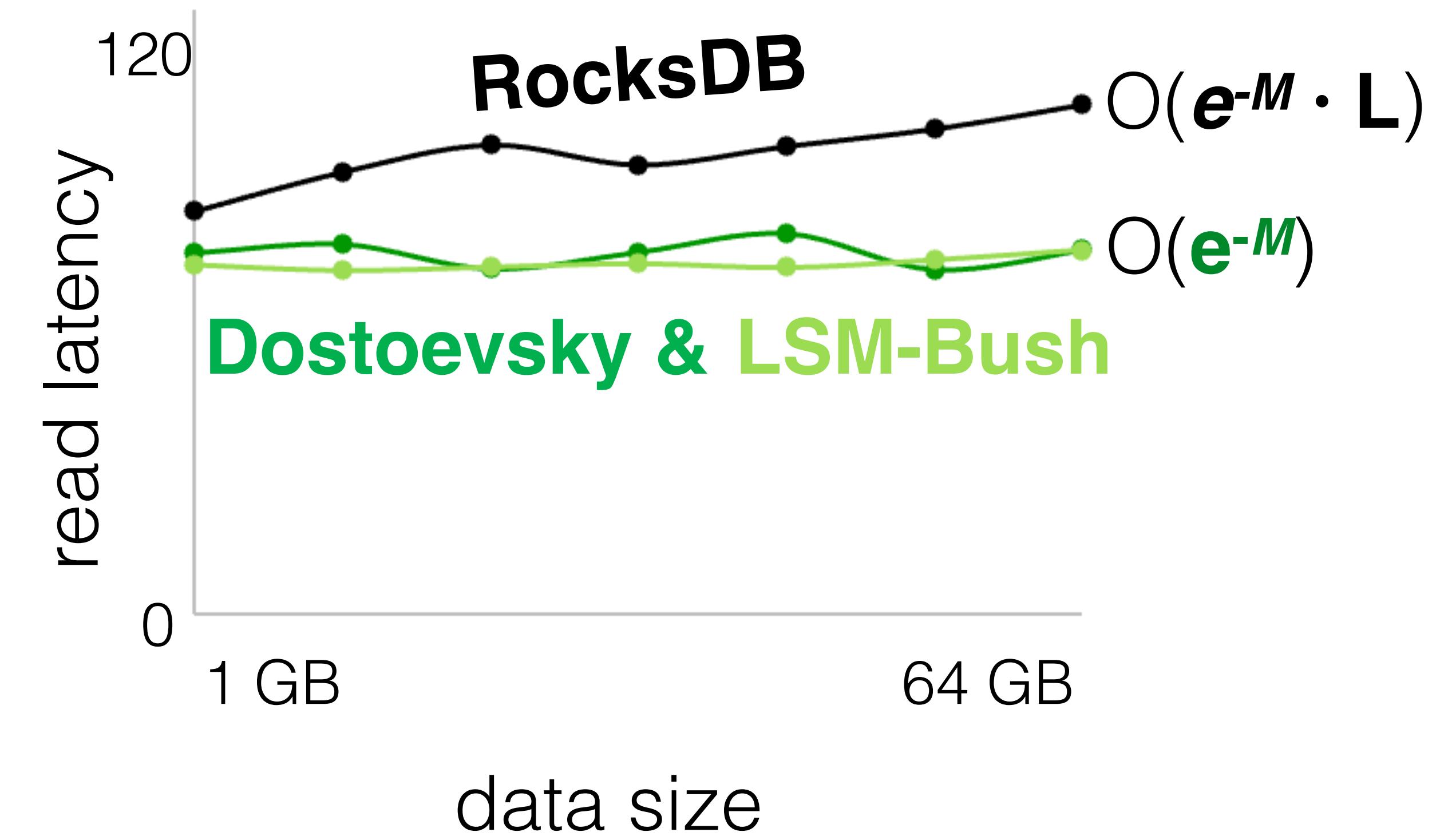
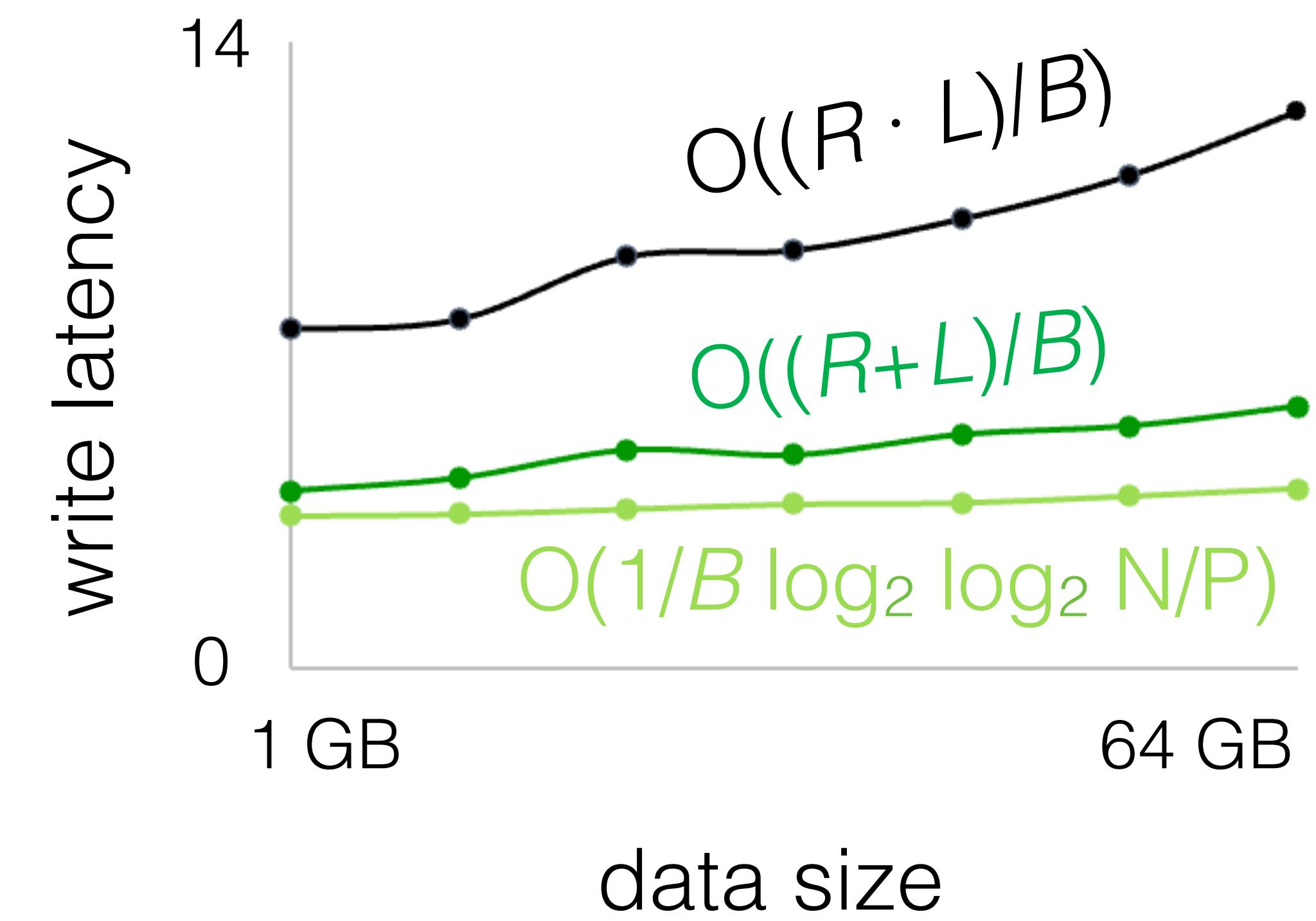


SIGMOD 19

Configuration

- buffer 2MB
- size ratio: 5
- 1KB entries
- SSD storage





get I/O
cost

$$O(e^{-M} \cdot L)$$



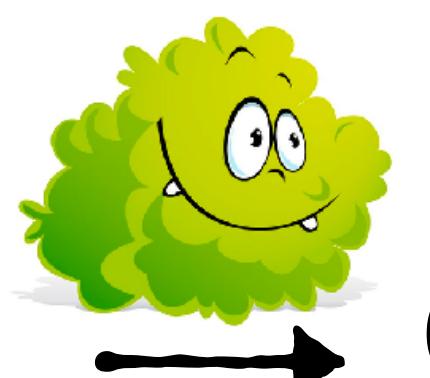
$$O(e^{-M})$$

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: partitioned compaction for key-value stores



VLDB 2022

Compaction policy



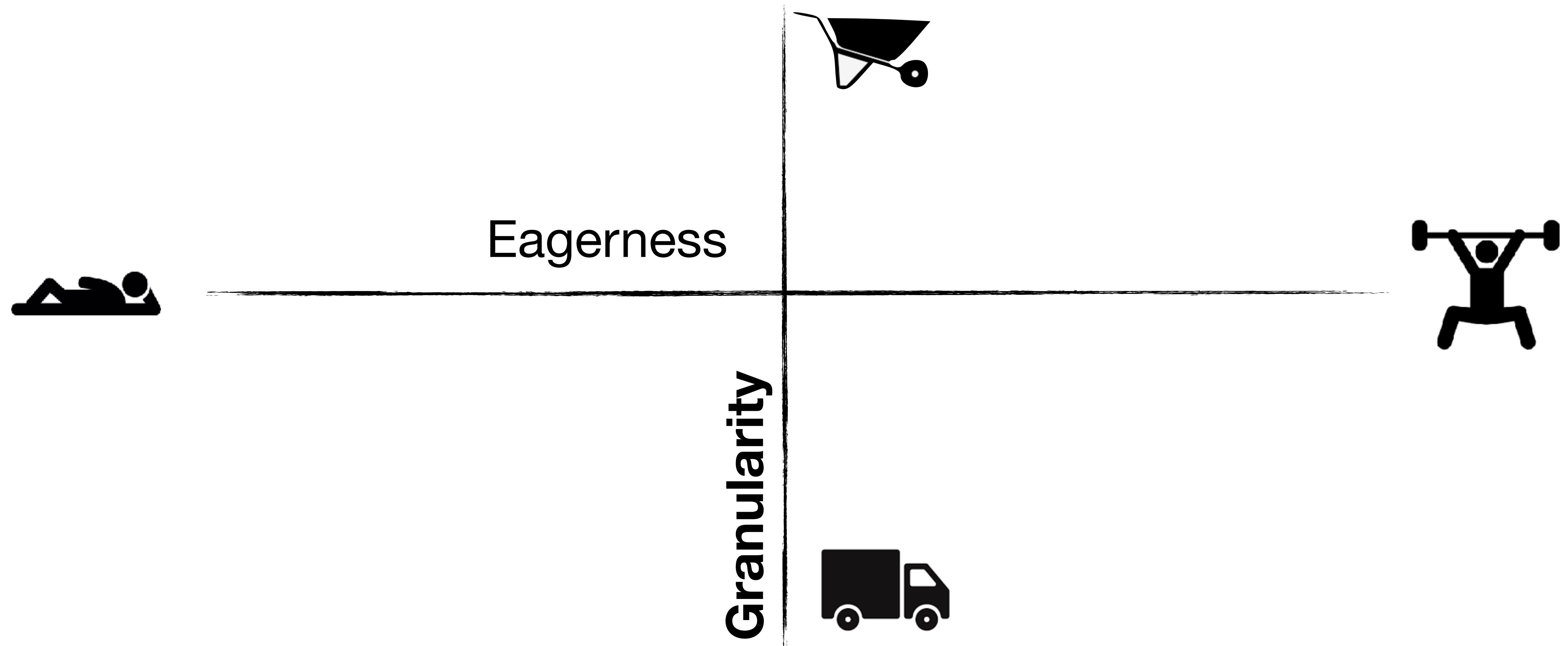
Write-
optimized

Eagerness

Read-
optimized



Compaction policy

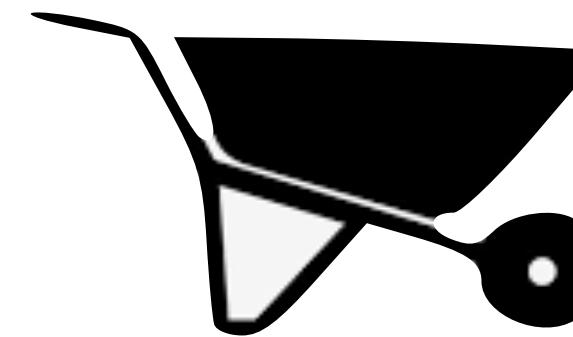


Compaction Granularity

Full Merge

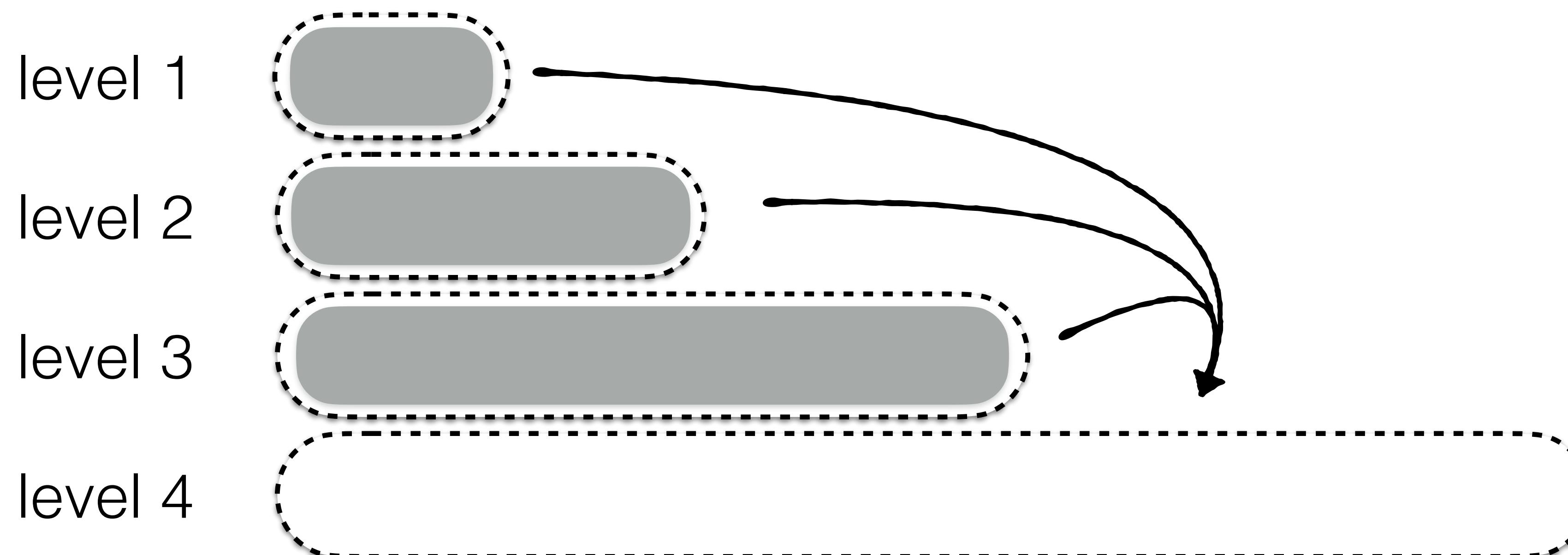


Partial Merge



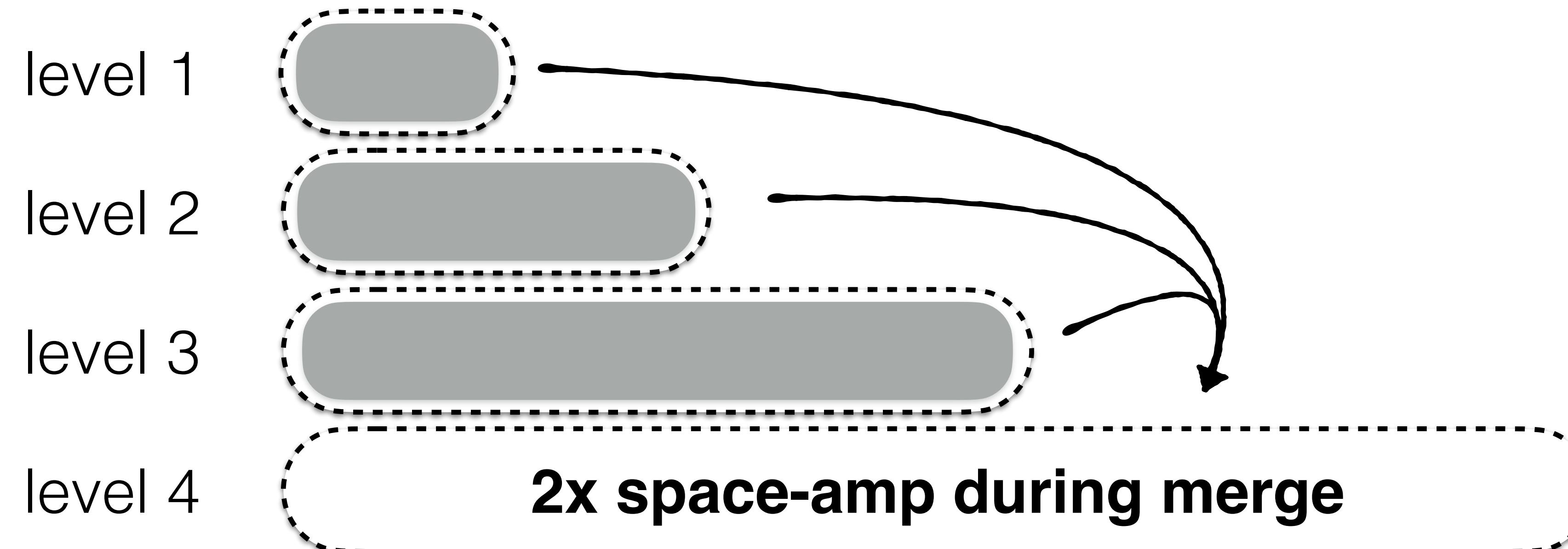
Full Merge

Merge consecutive full levels into first non-full level



Full Merge

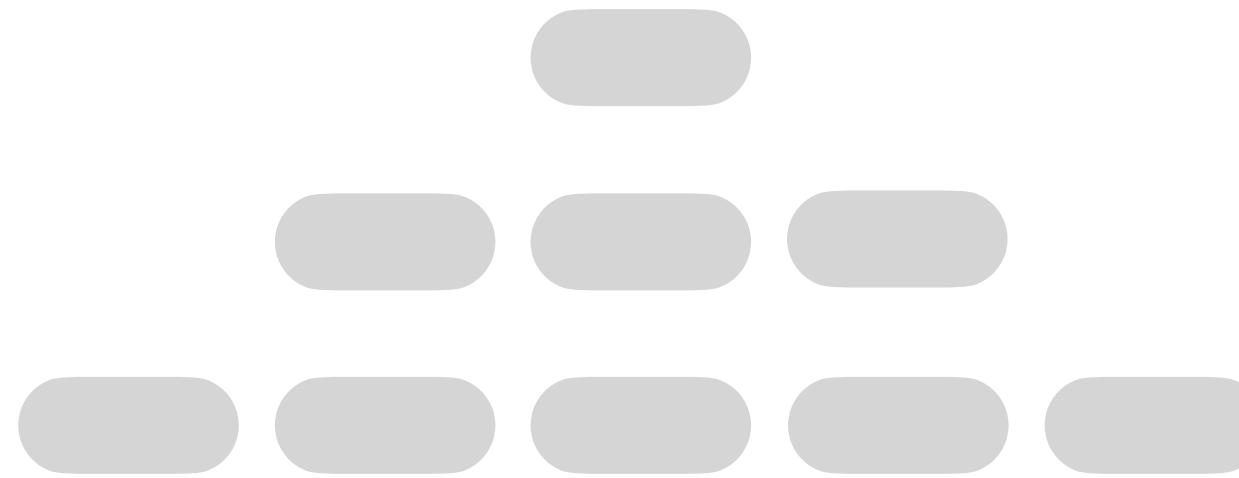
Merge consecutive full levels into first non-full level



Full Merge

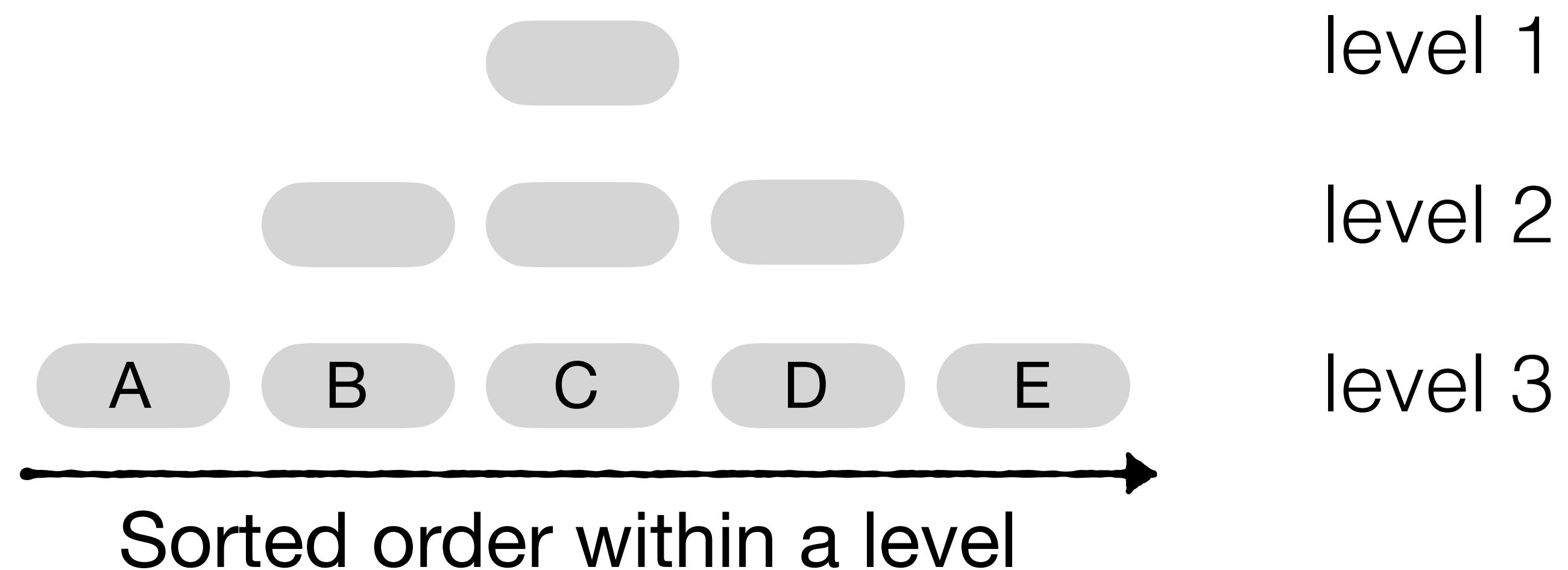


Partial Merge



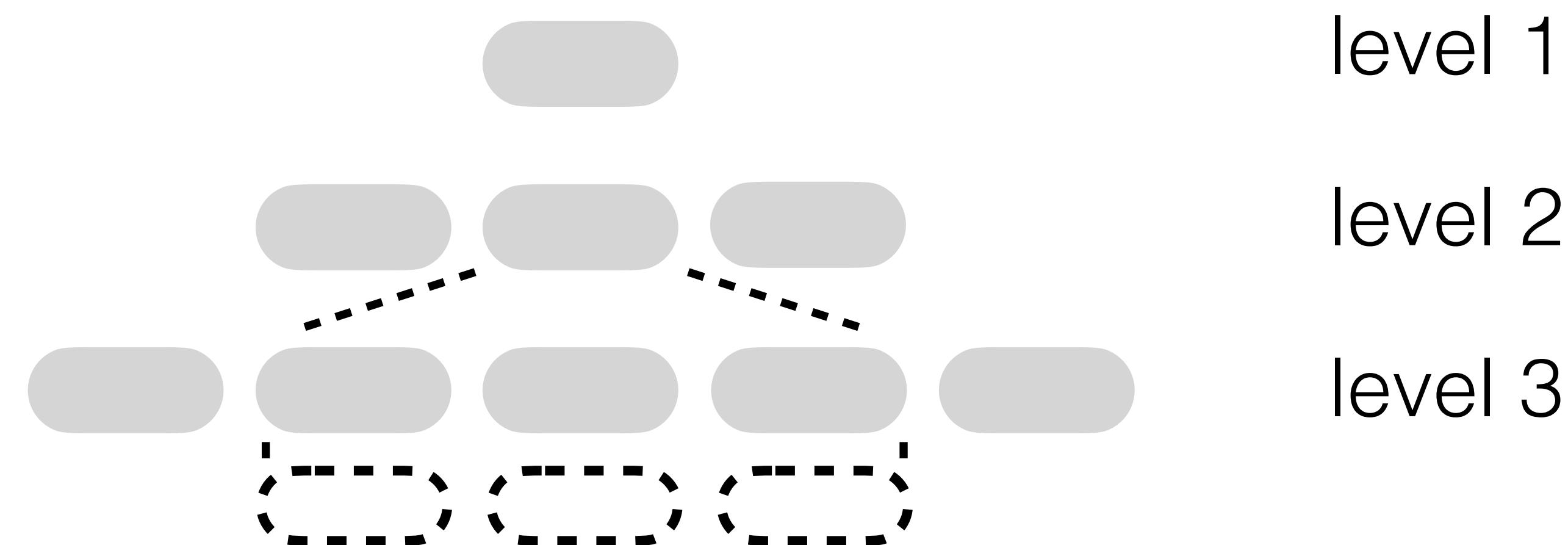
Partial Merge

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



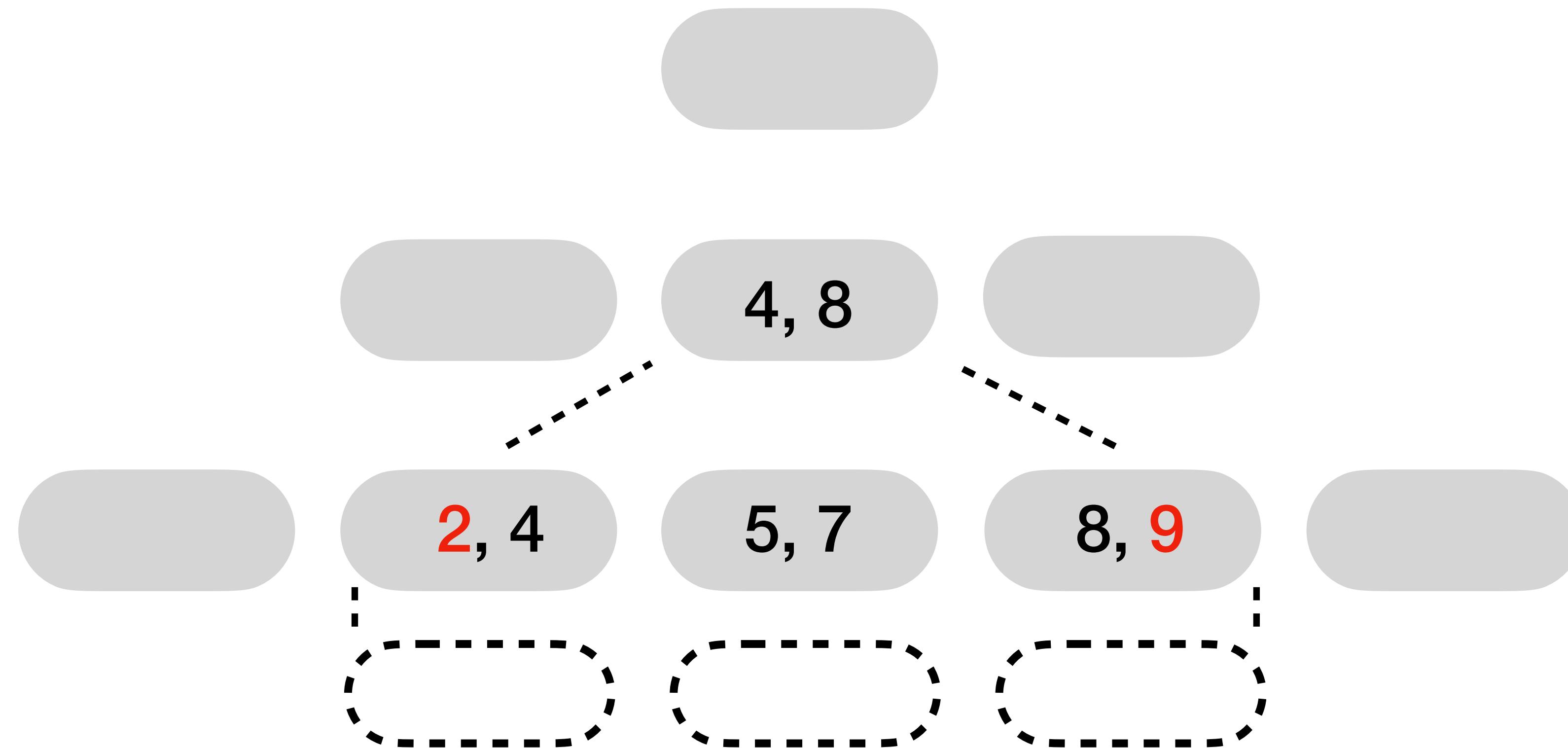
Partial Merge

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



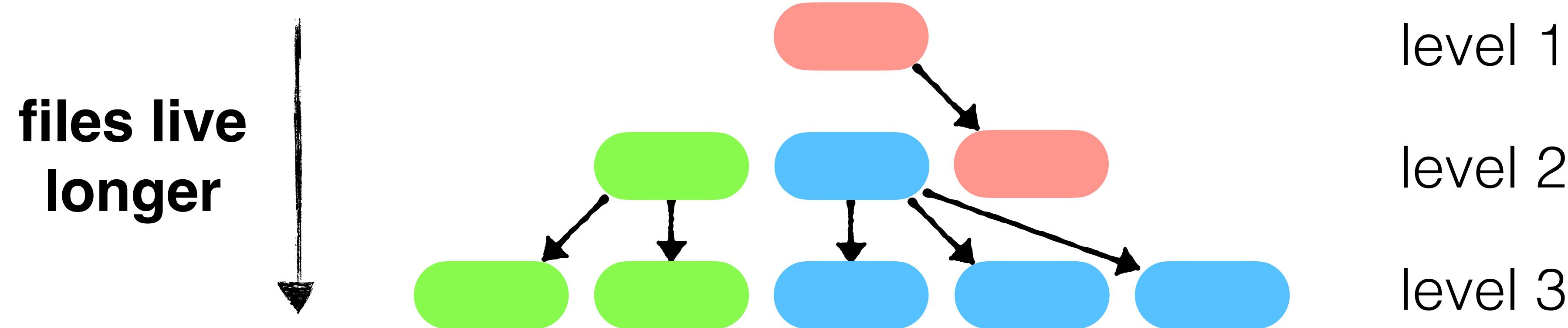
Partial Merge

Problem 1: non-intersecting entries increase write-amplification



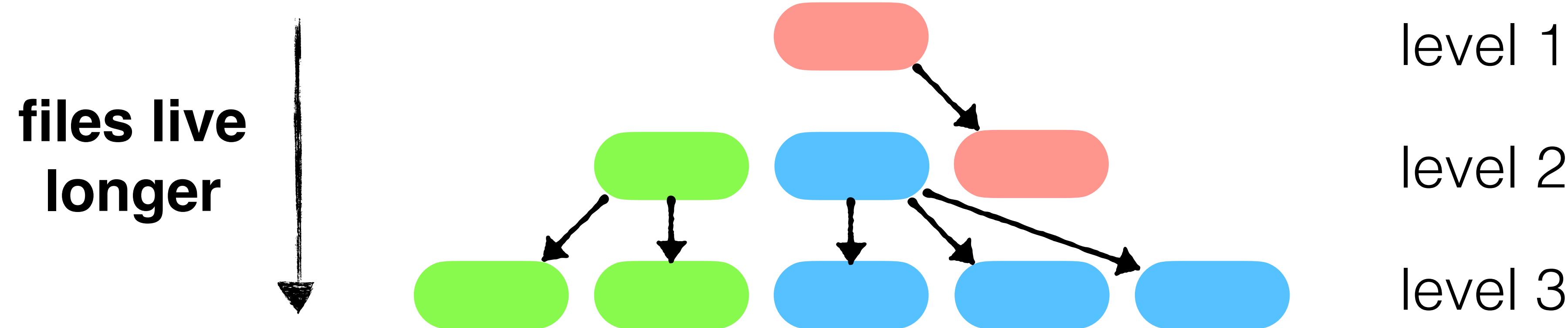
Partial Merge

Problem 2: many small simultaneous compactions

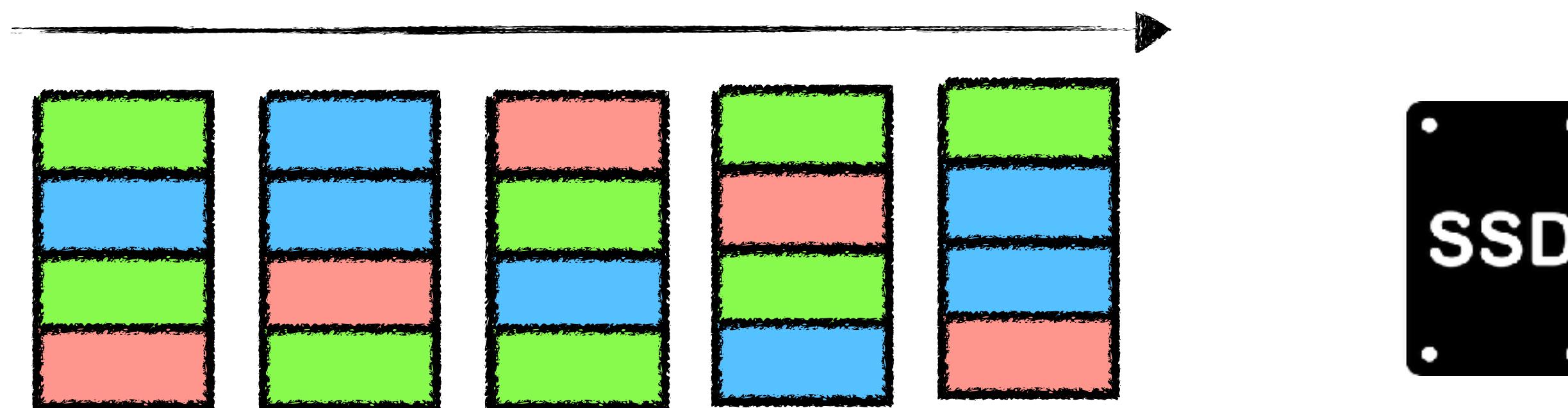


Partial Merge

Problem 2: many small simultaneous compactions

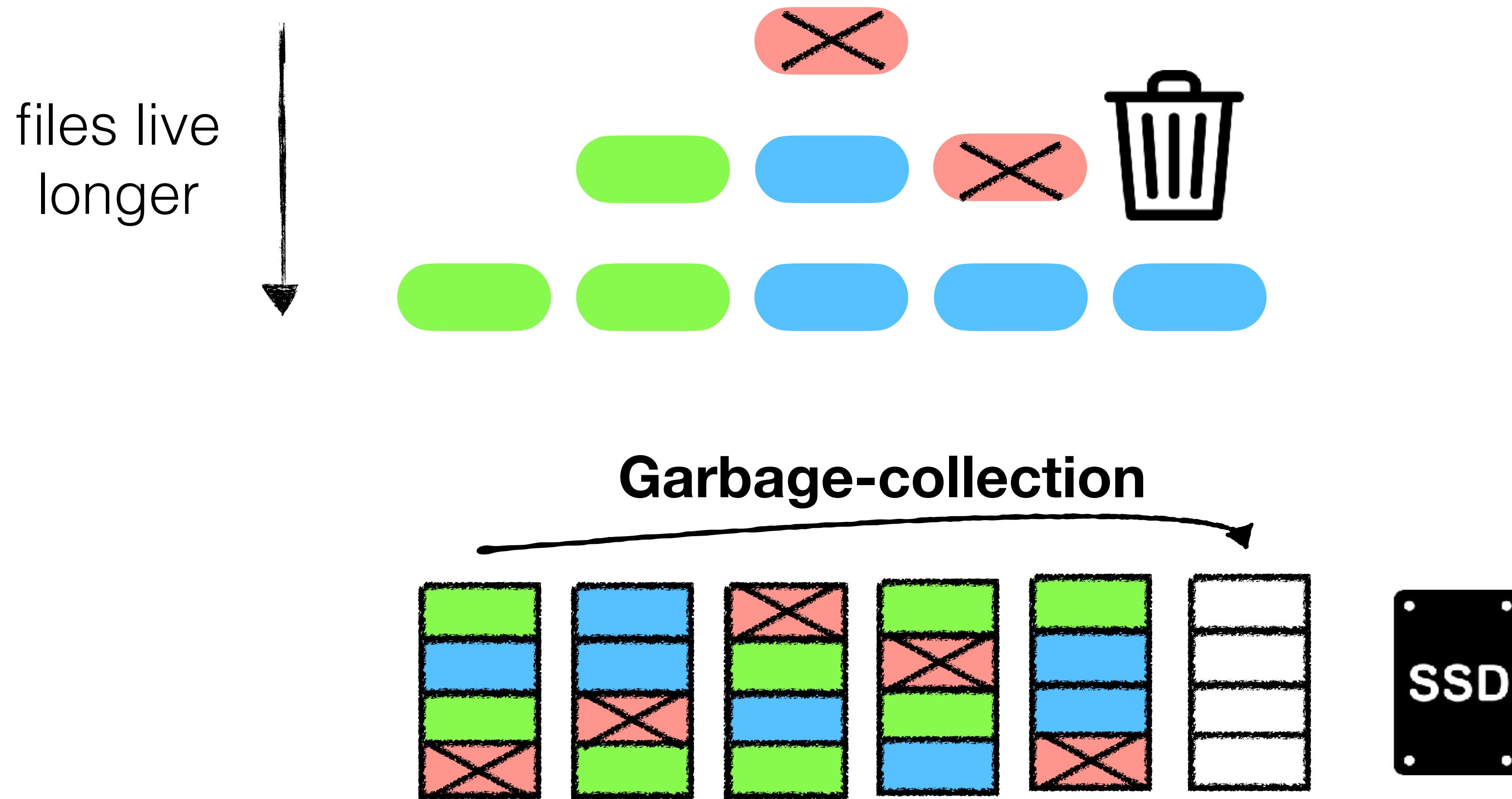


Log-structured writes



Partial Merge

Problem 2: many small simultaneous compactions



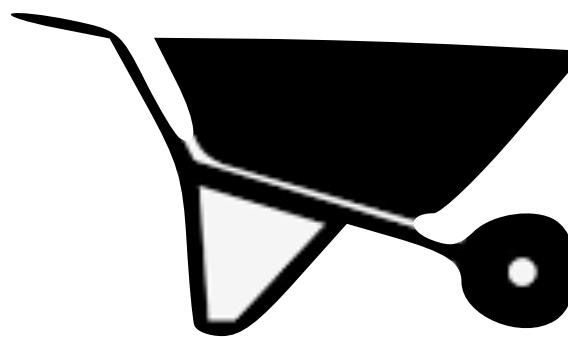
Full Merge



space amplification

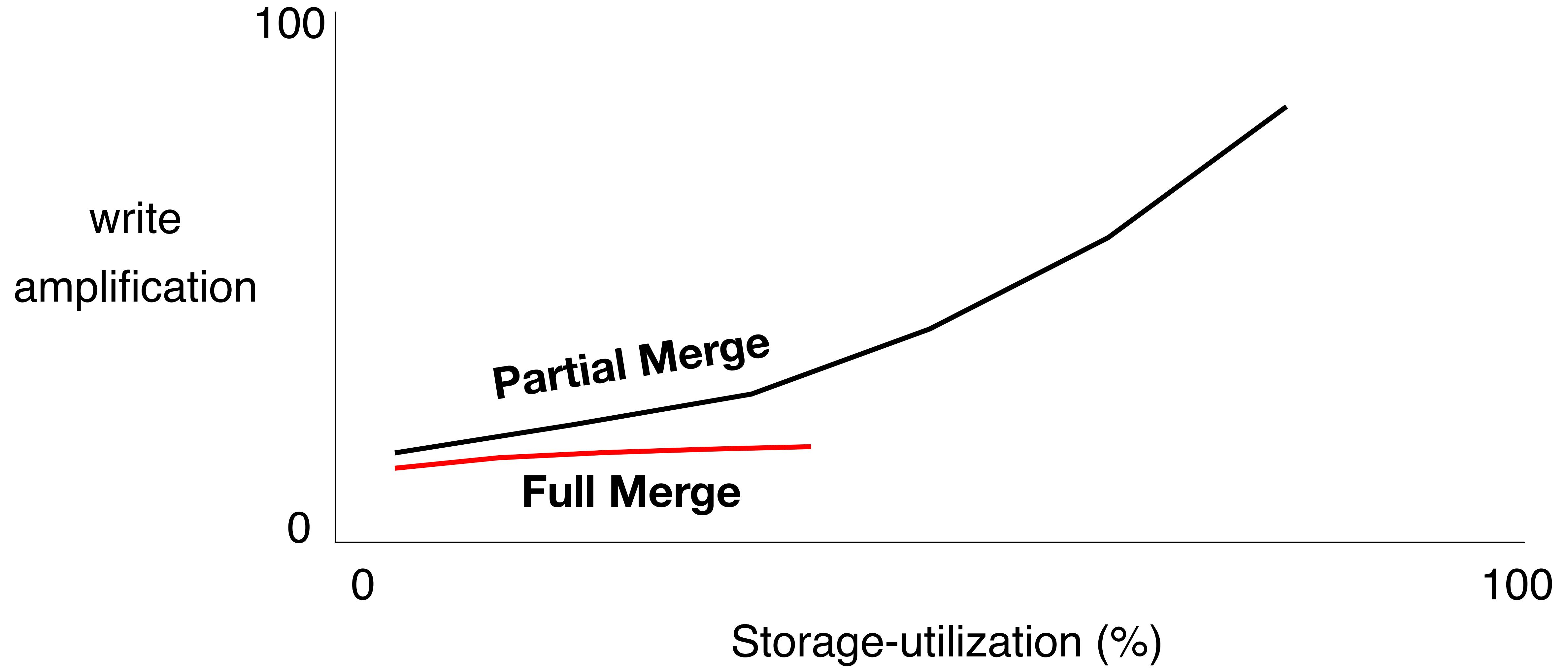


Partial Merge

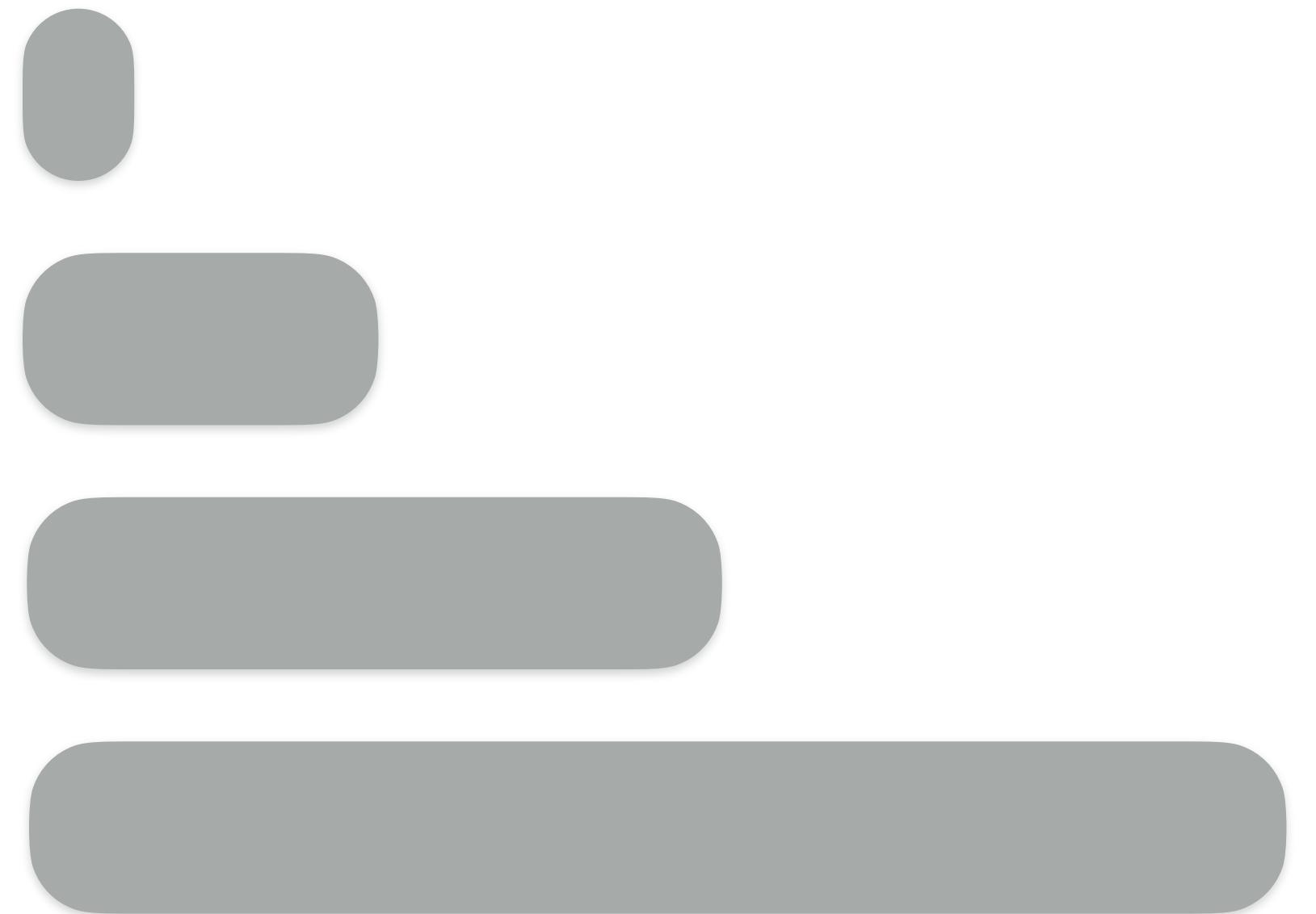


write amplification





Spooky's intuition



**transient
space amplification**

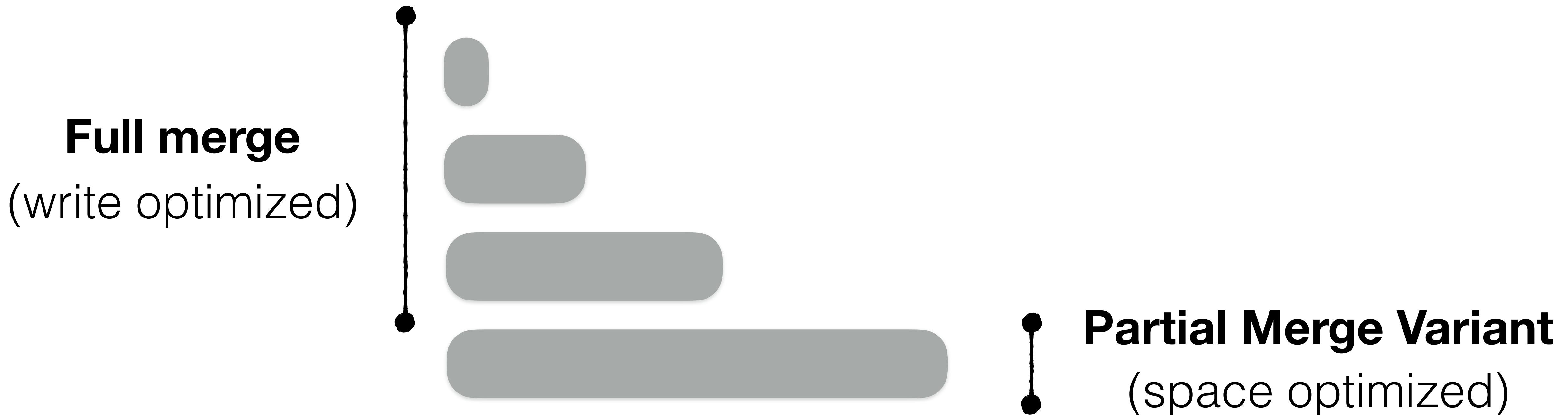
Spooky's intuition

**write
amplification**



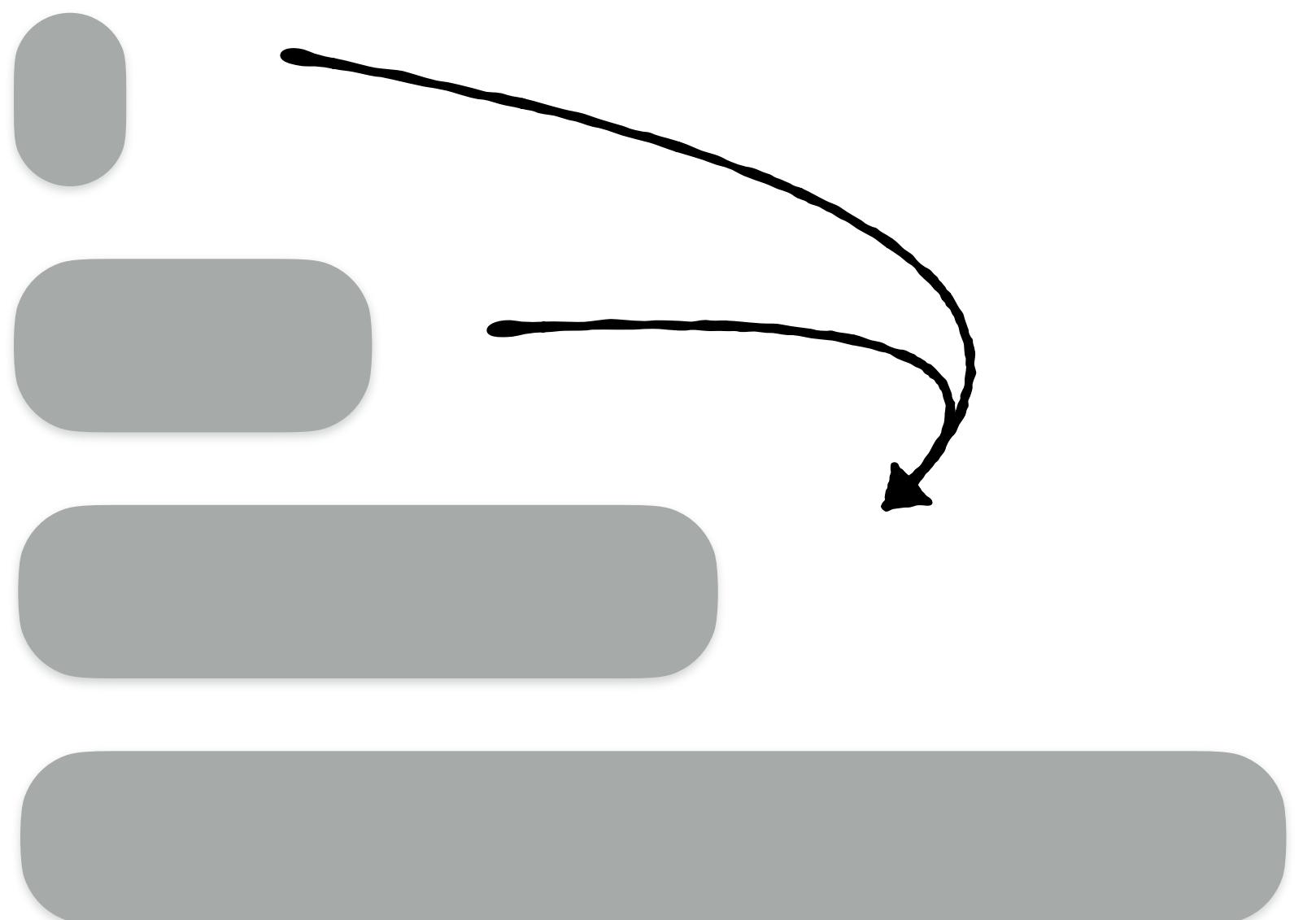
← transient
space amplification

Spooky's intuition



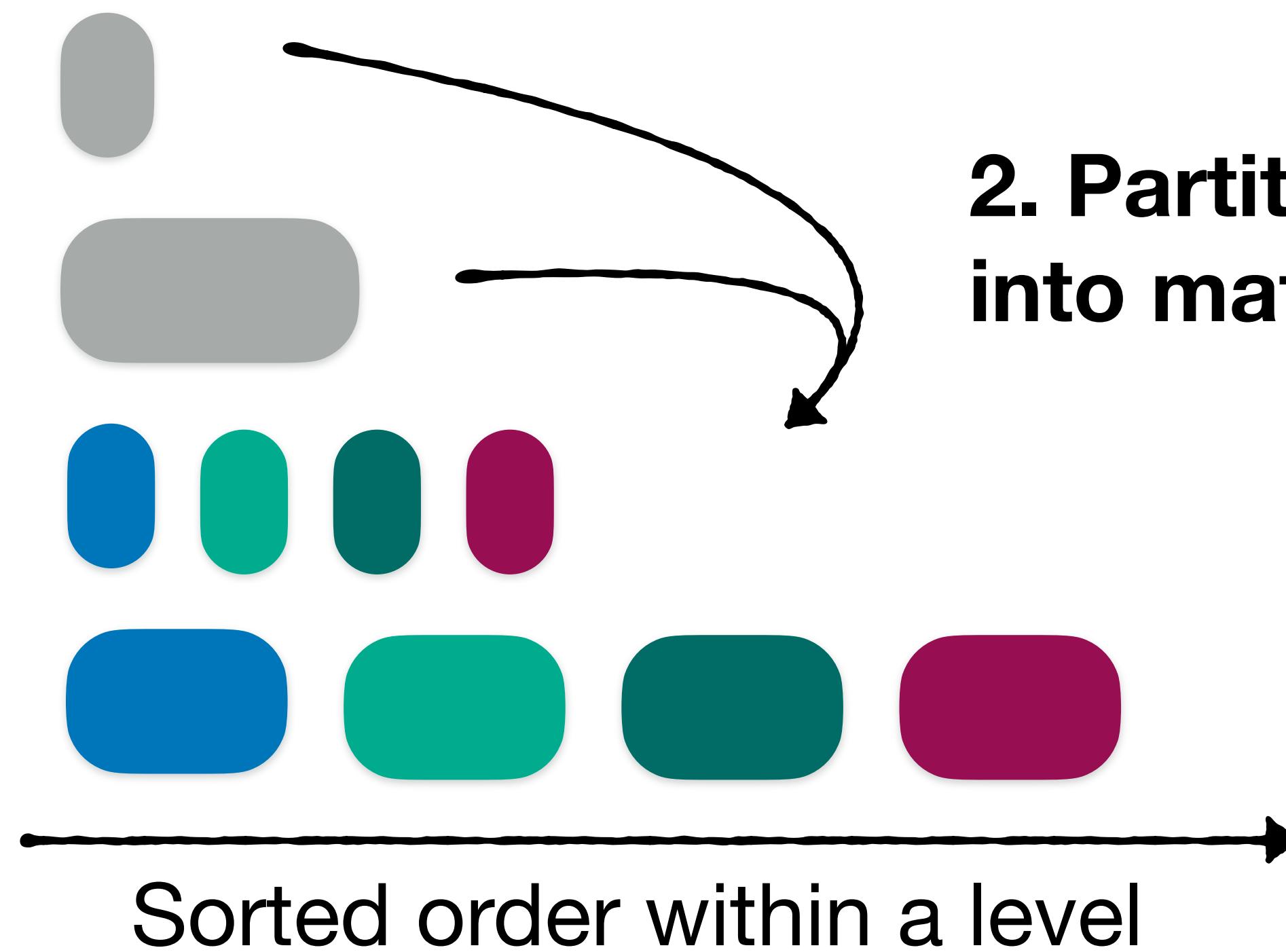
Spooky

1. Full merge at up to second largest level



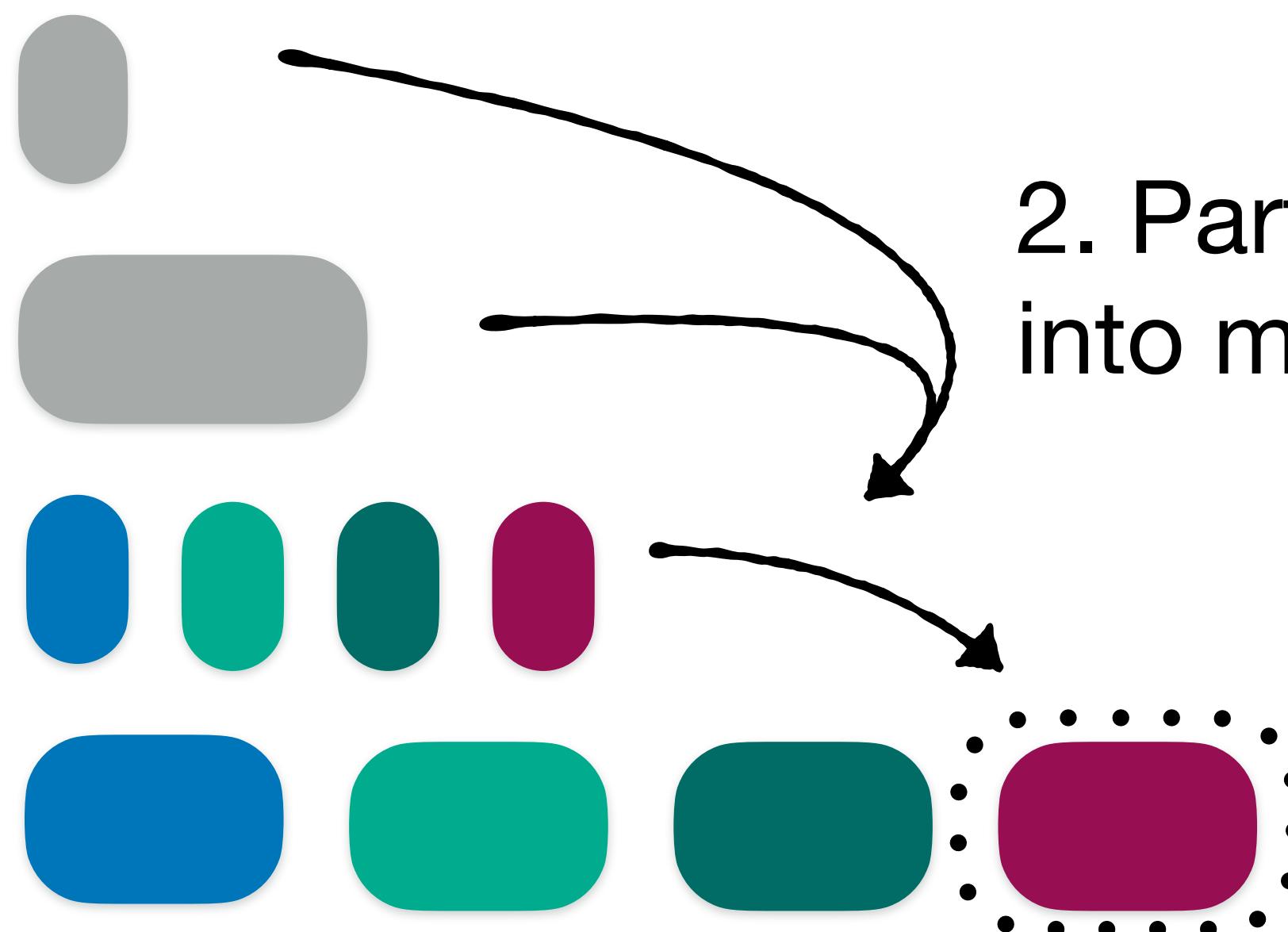
Spooky

1. Full merge at up to second largest level



Spooky

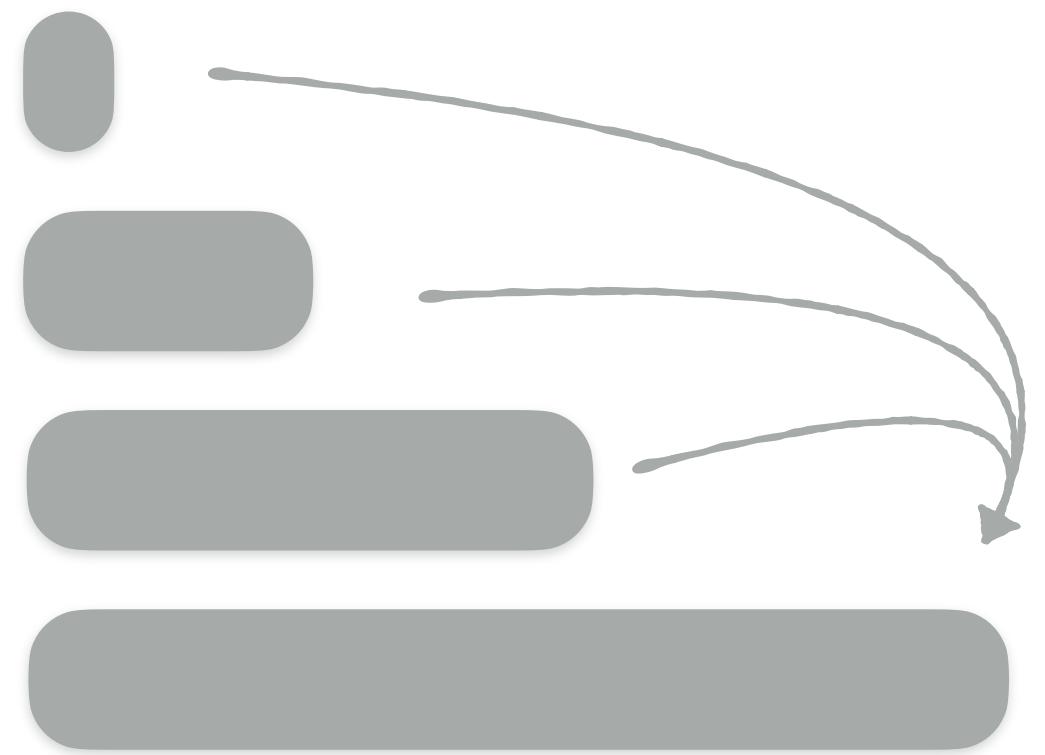
1. Full merge at up to second largest level



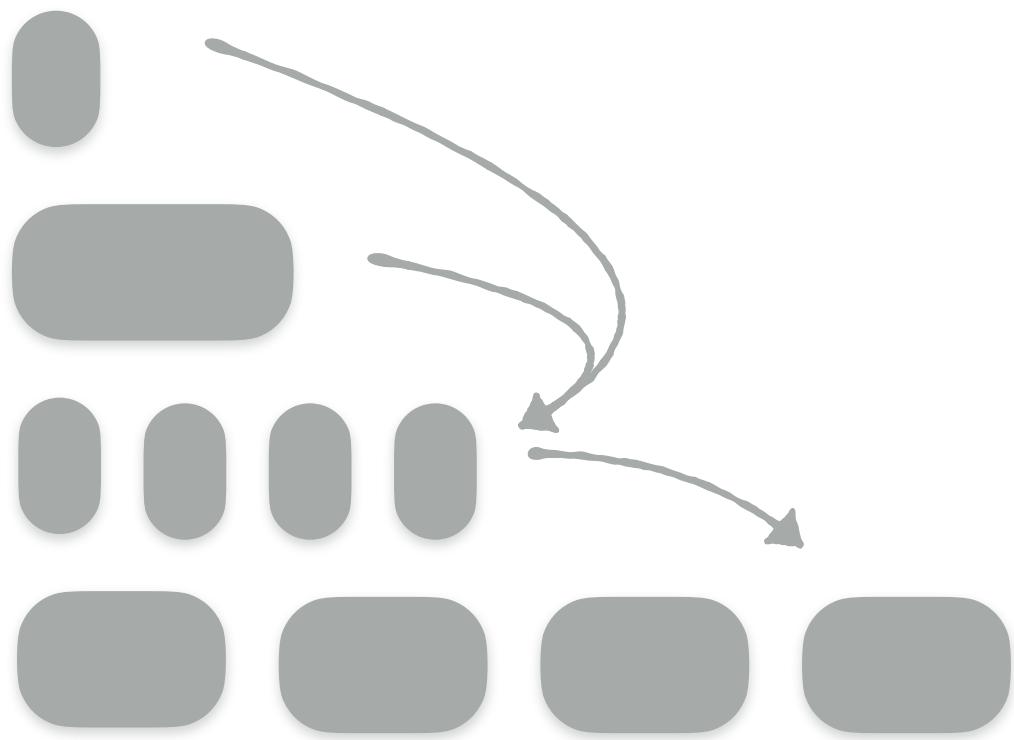
2. Partition largest two level
into matching pairs of files

3. Merge one
partition at a time

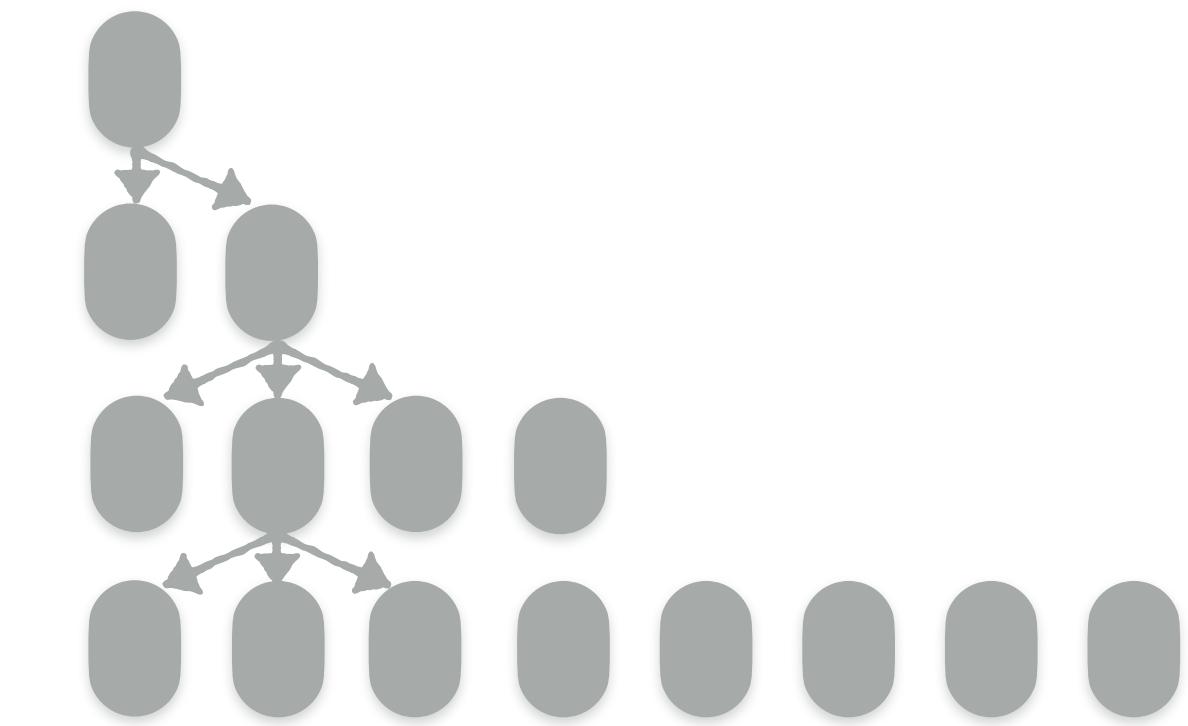
Full

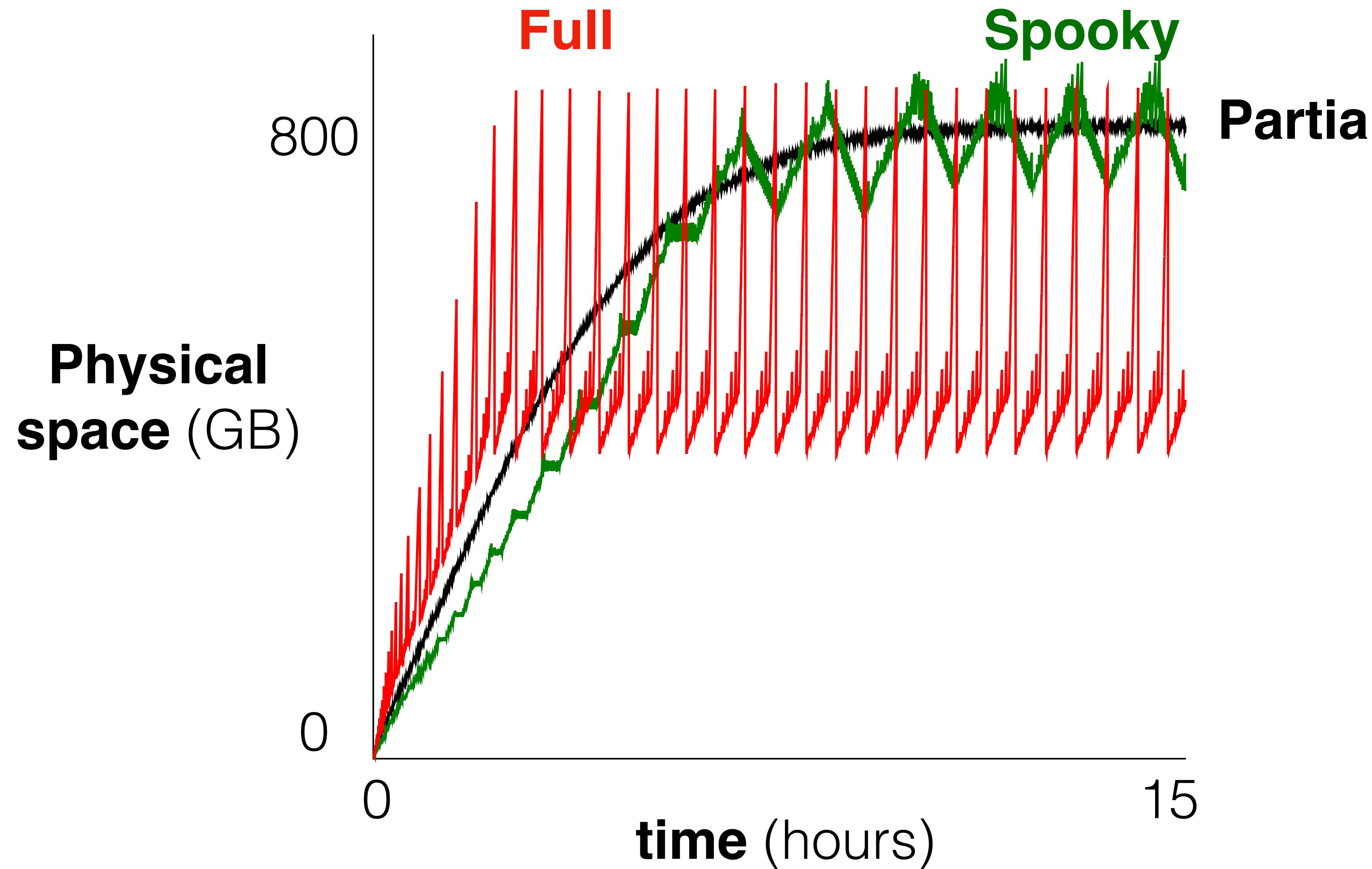


Spooky

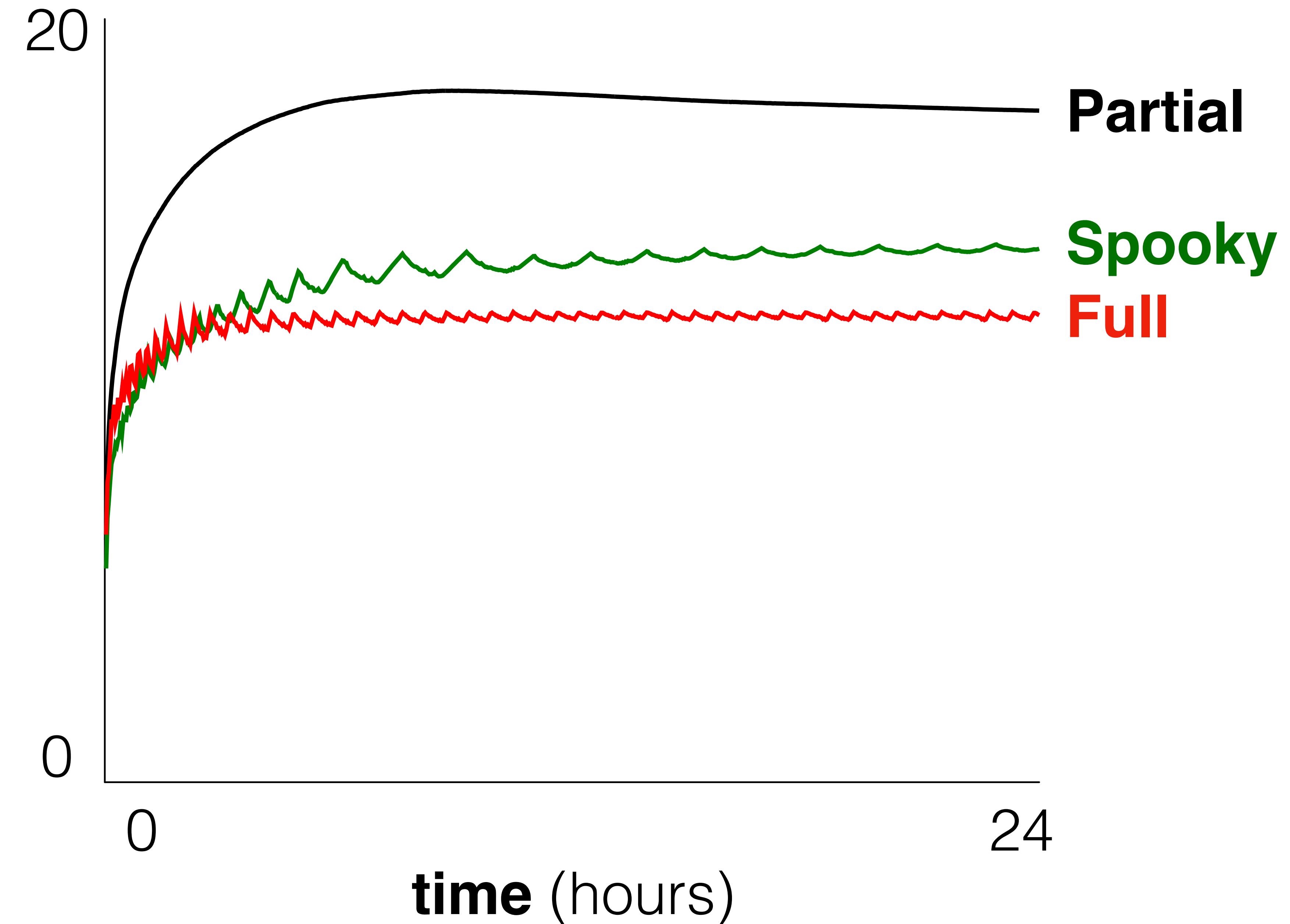


Partial



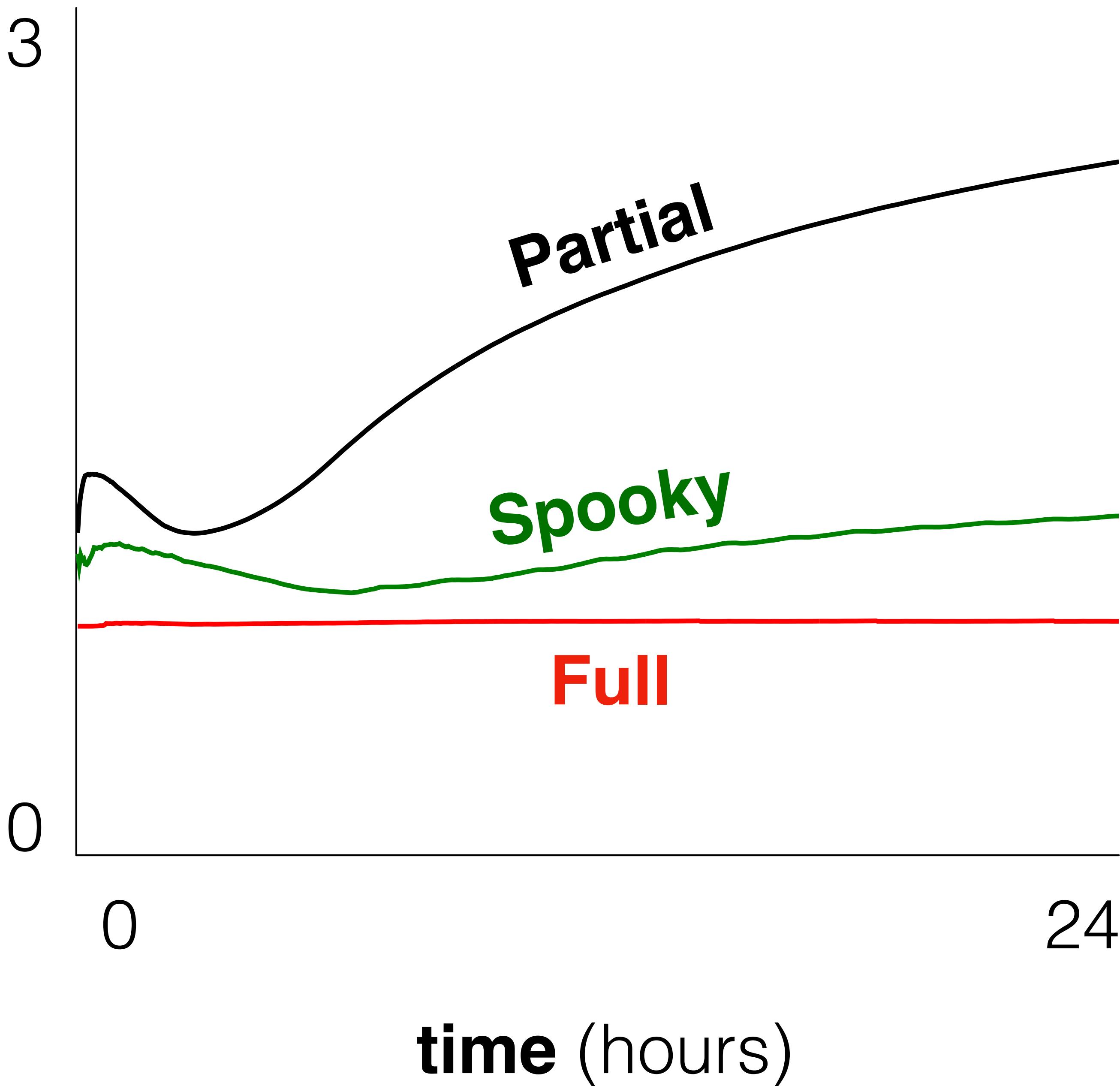


compaction
write amplification



Garbage Collection

write amplification



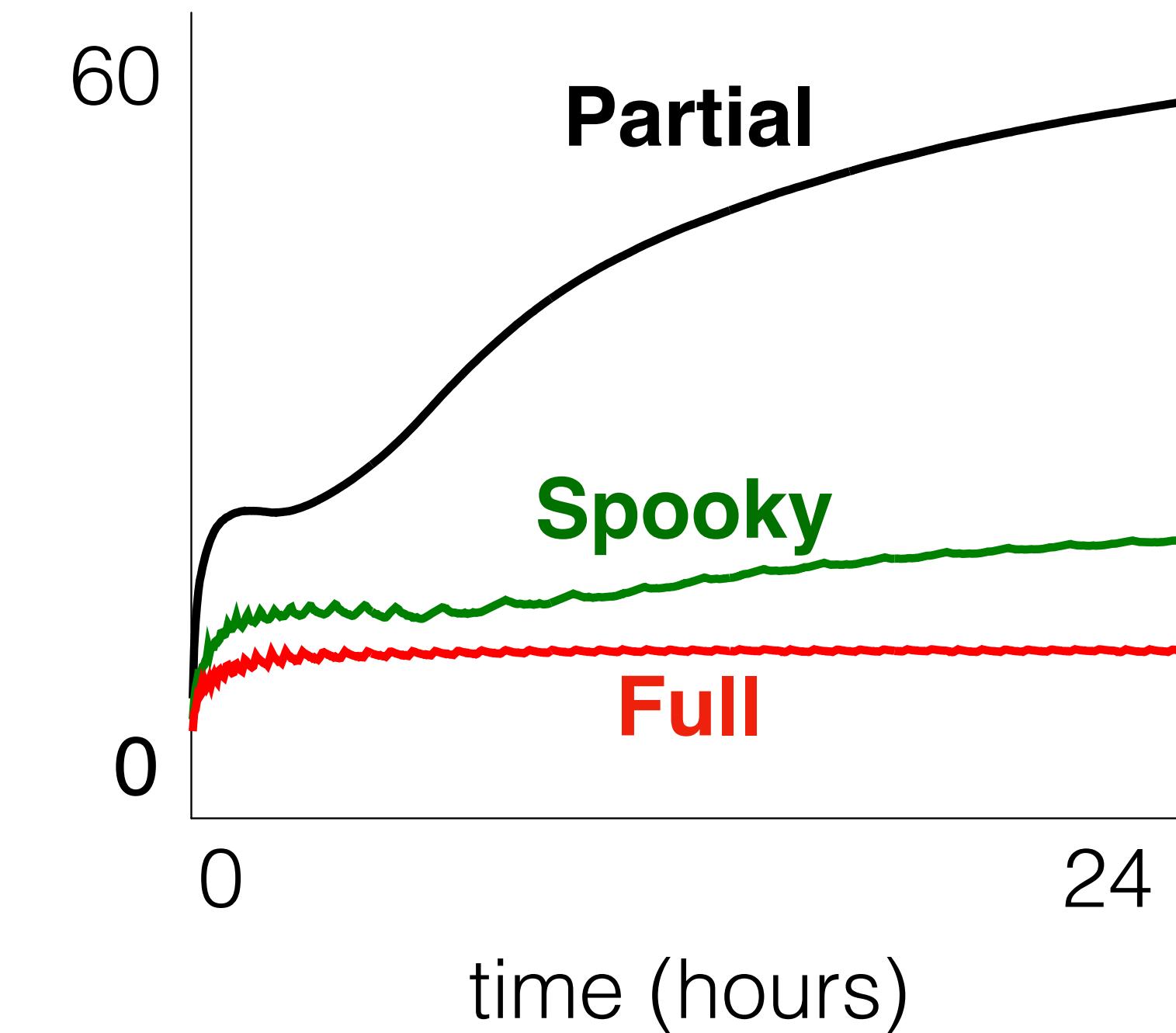
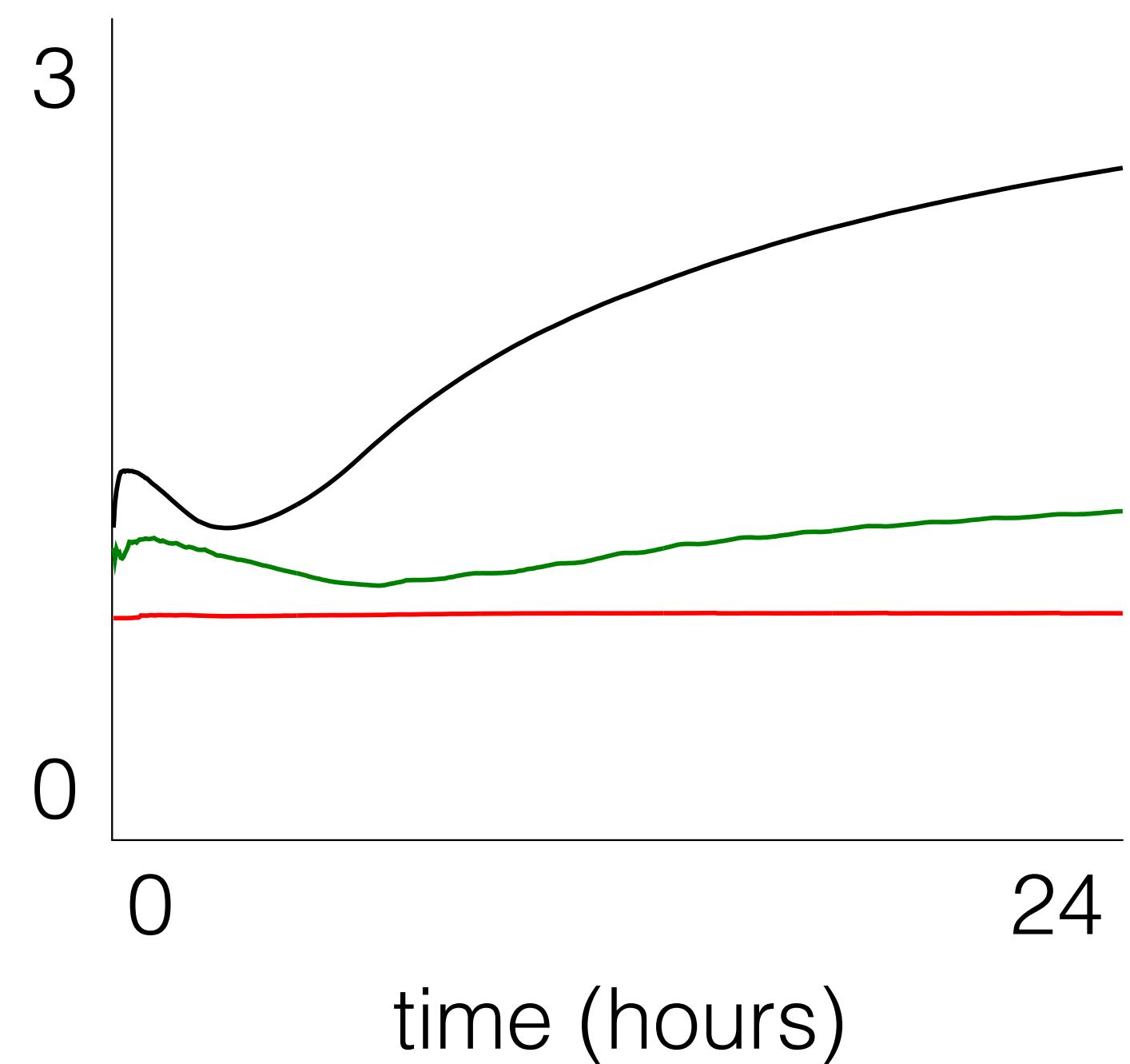
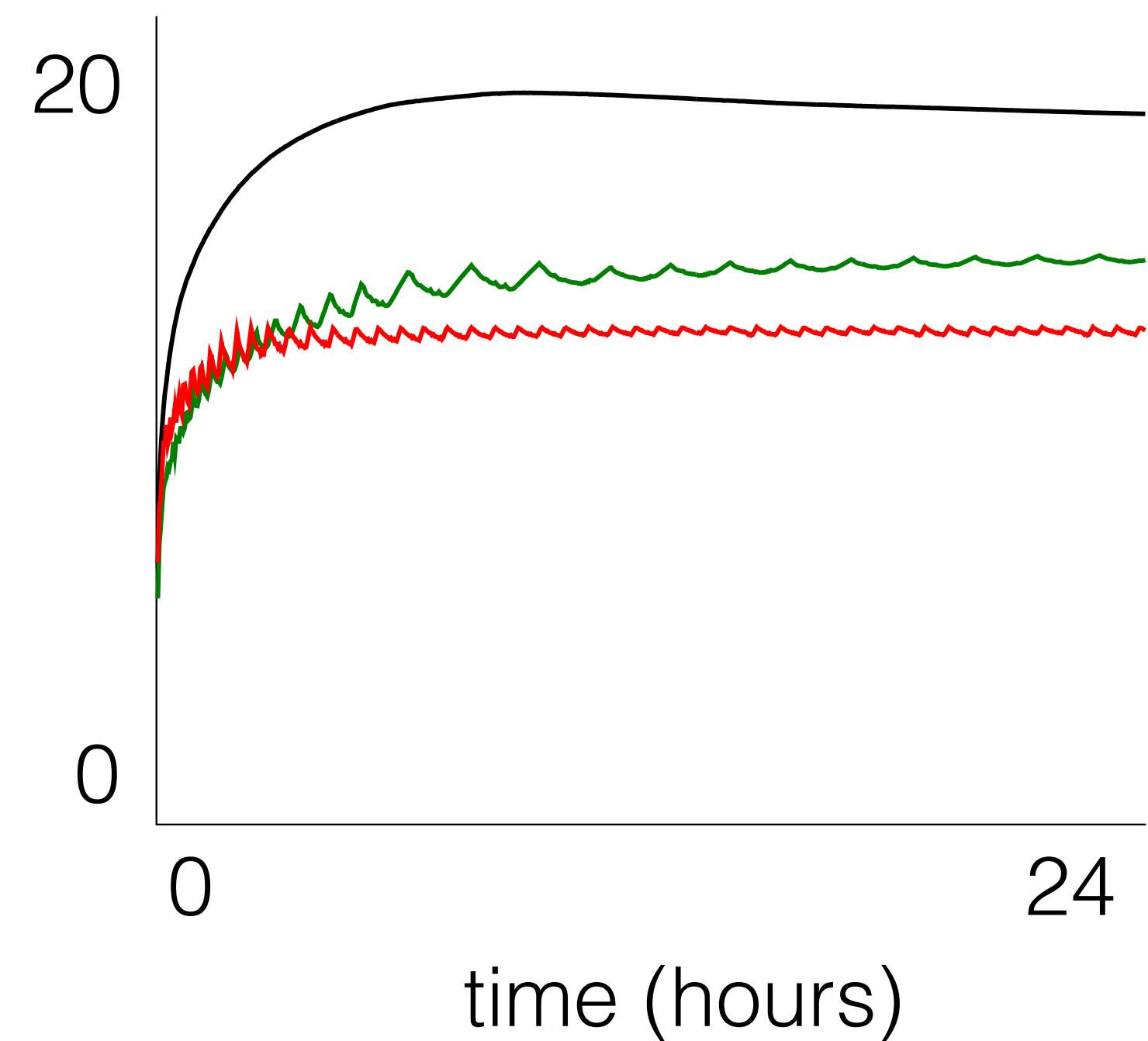
Compaction
write amplification

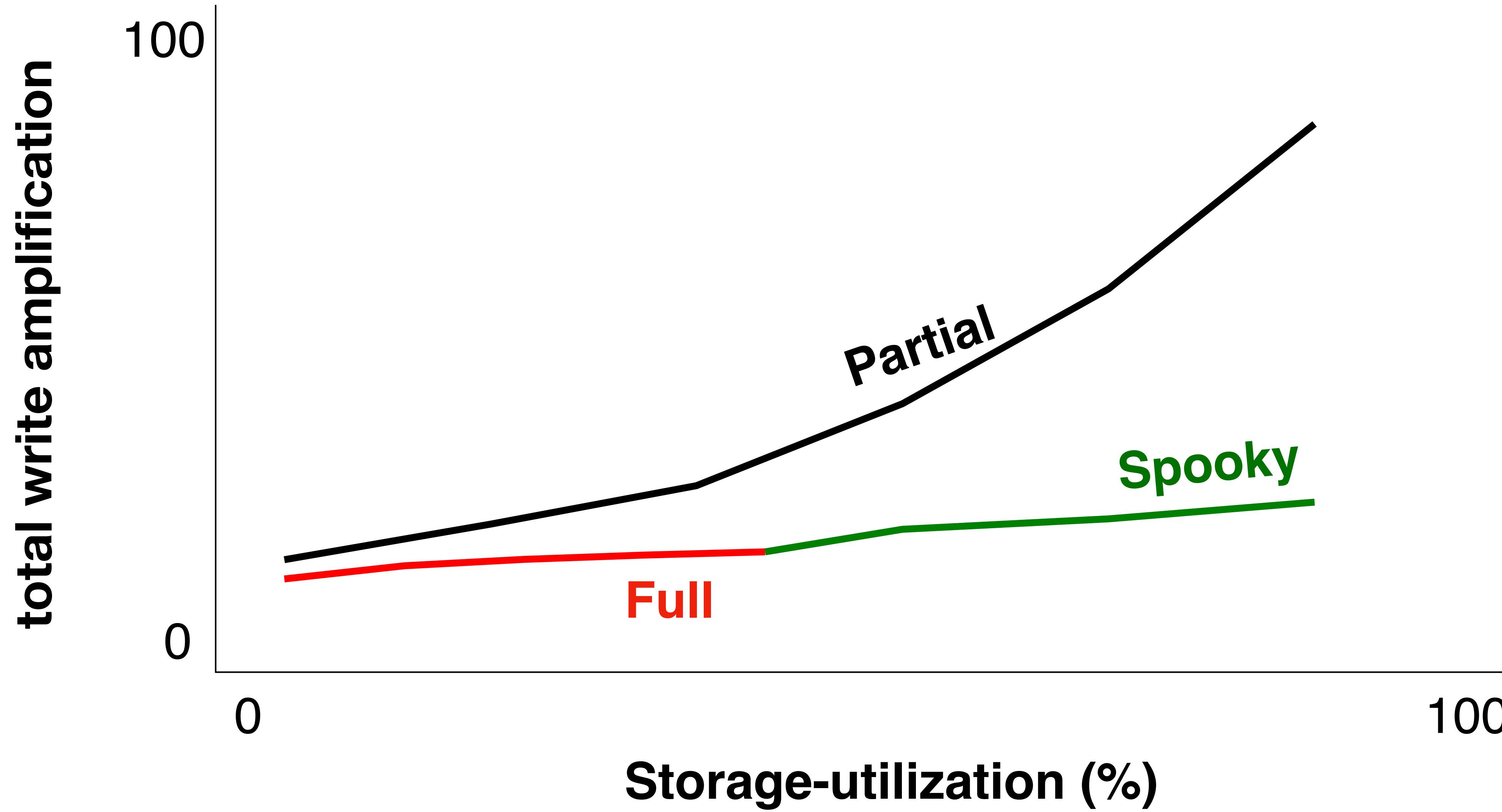
X

Garbage Collection
write amplification

=

**Total
write amplification**





get I/O
cost

$$O(e^{-M} \cdot L)$$



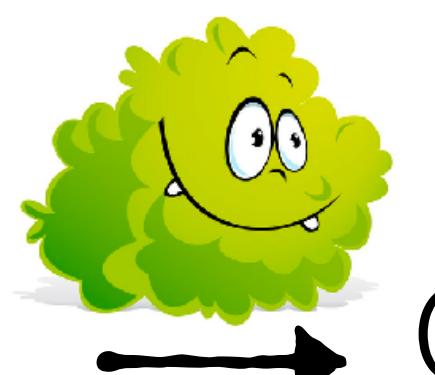
$$O(e^{-M})$$

insert I/O
cost

$$O((R \cdot L)/B)$$



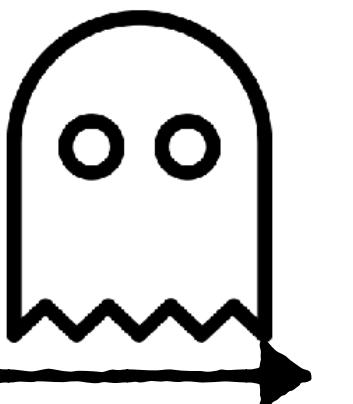
$$O((R+L)/B)$$



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

**transient
space-amp**

≈ 2



≈ 1

filter mem.
reads

$$O(M + L)$$

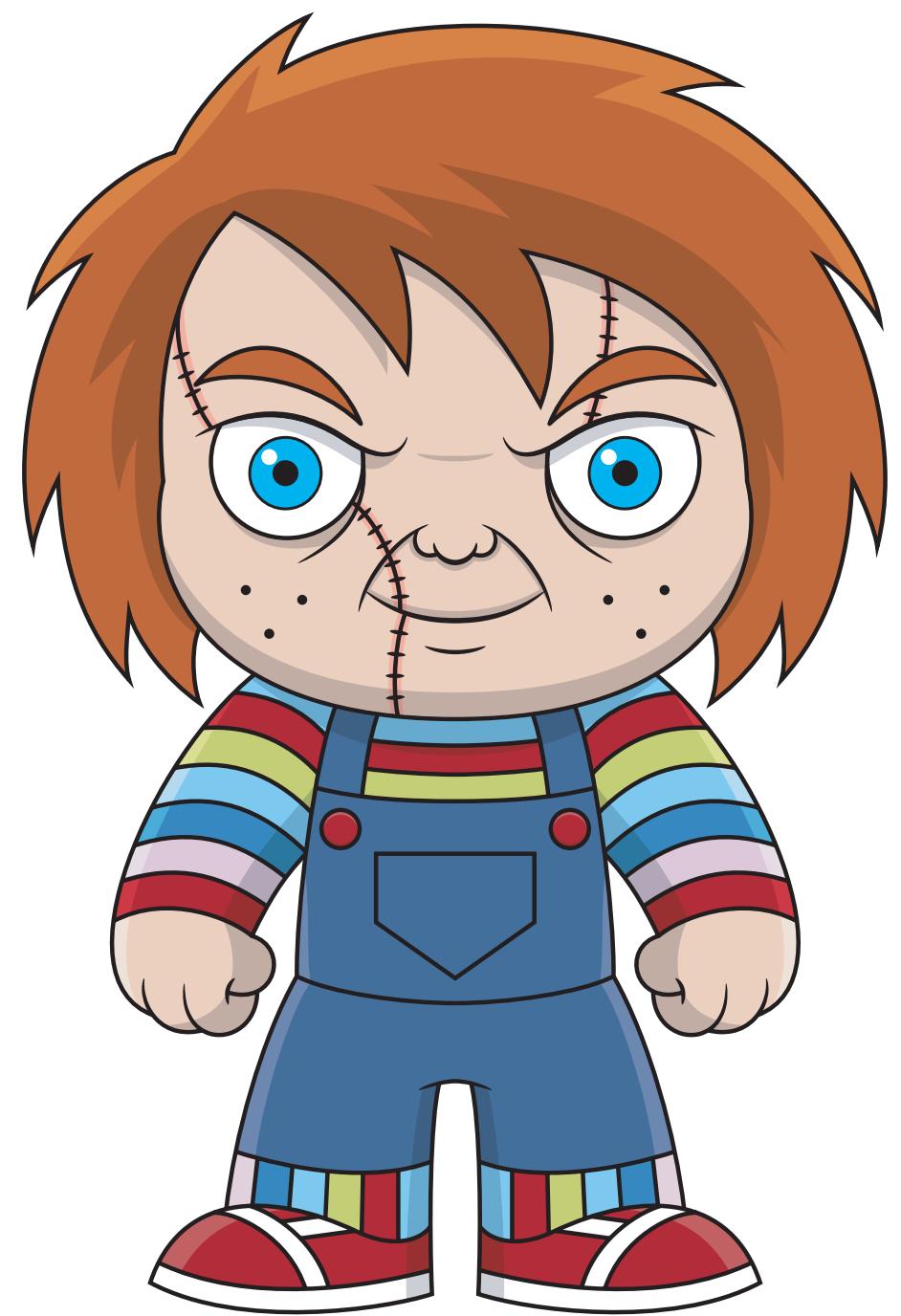


$$O(?)$$

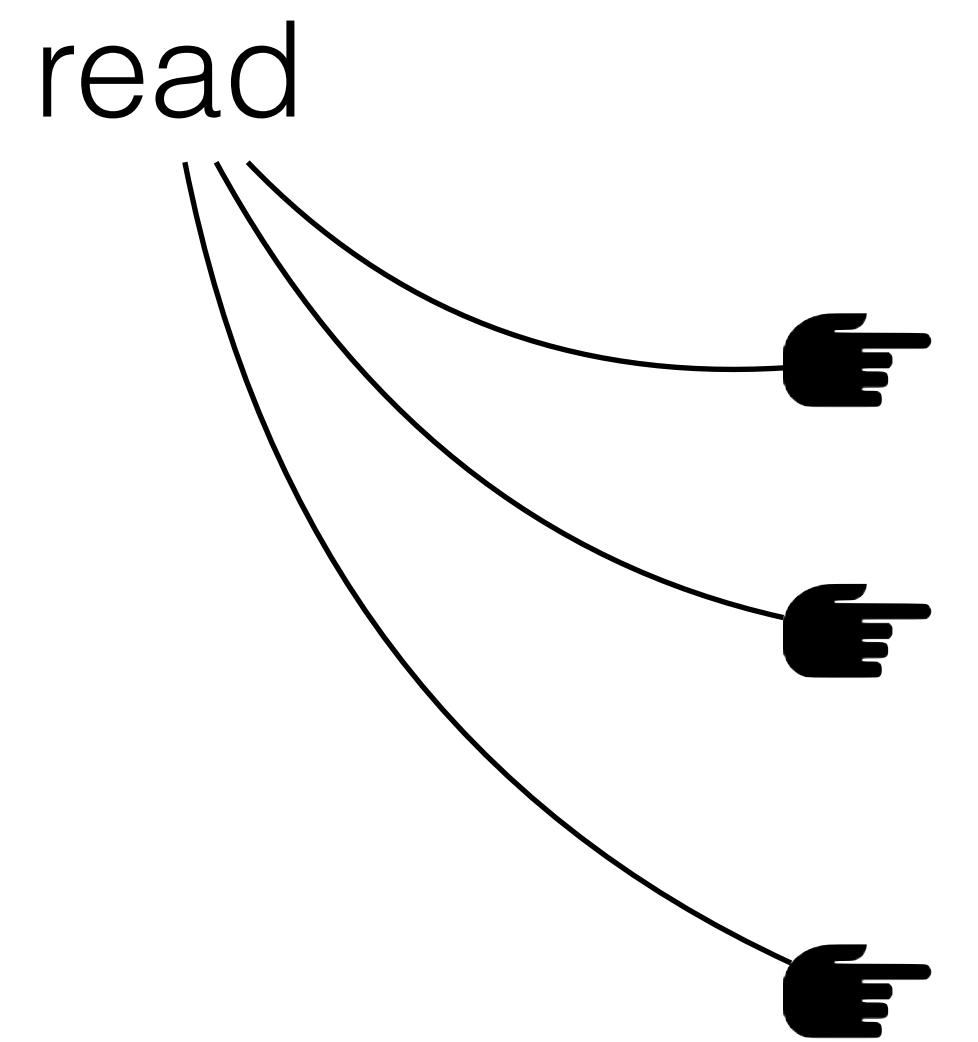
$$L = \log_R N/P$$

(Costs assuming leveling)

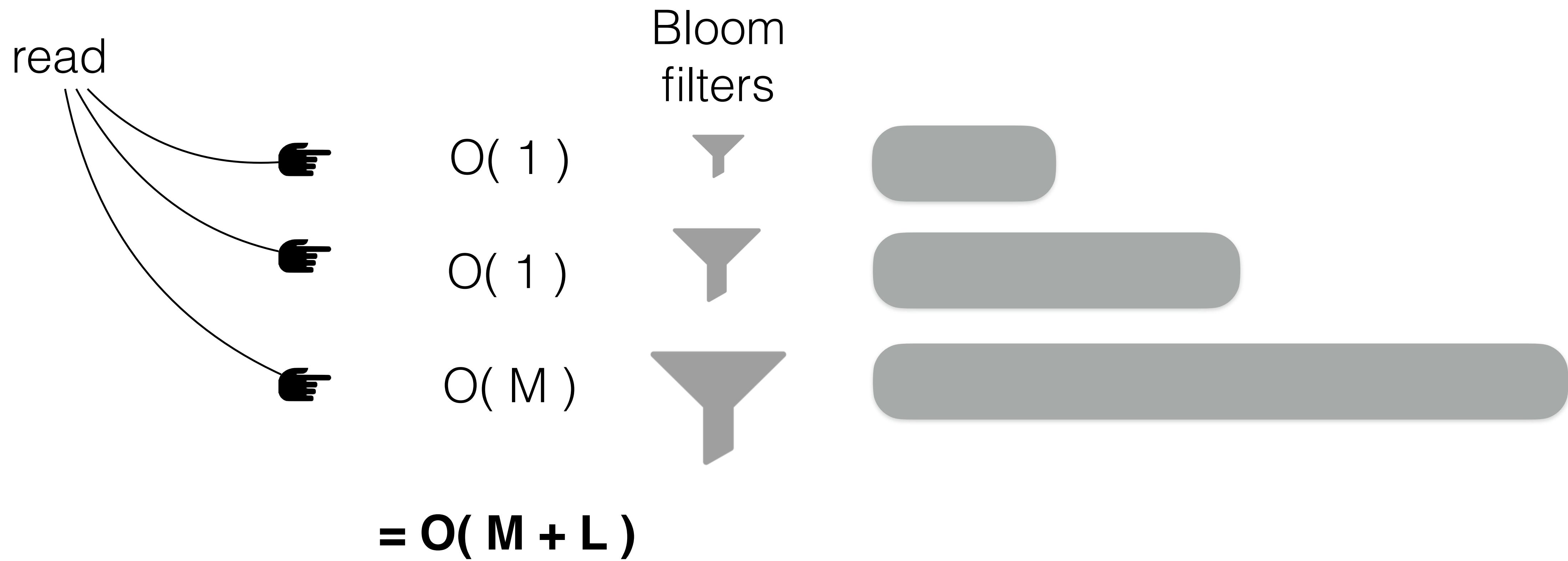
Chucky: Succinct Cuckoo Filter for LSM-Tree

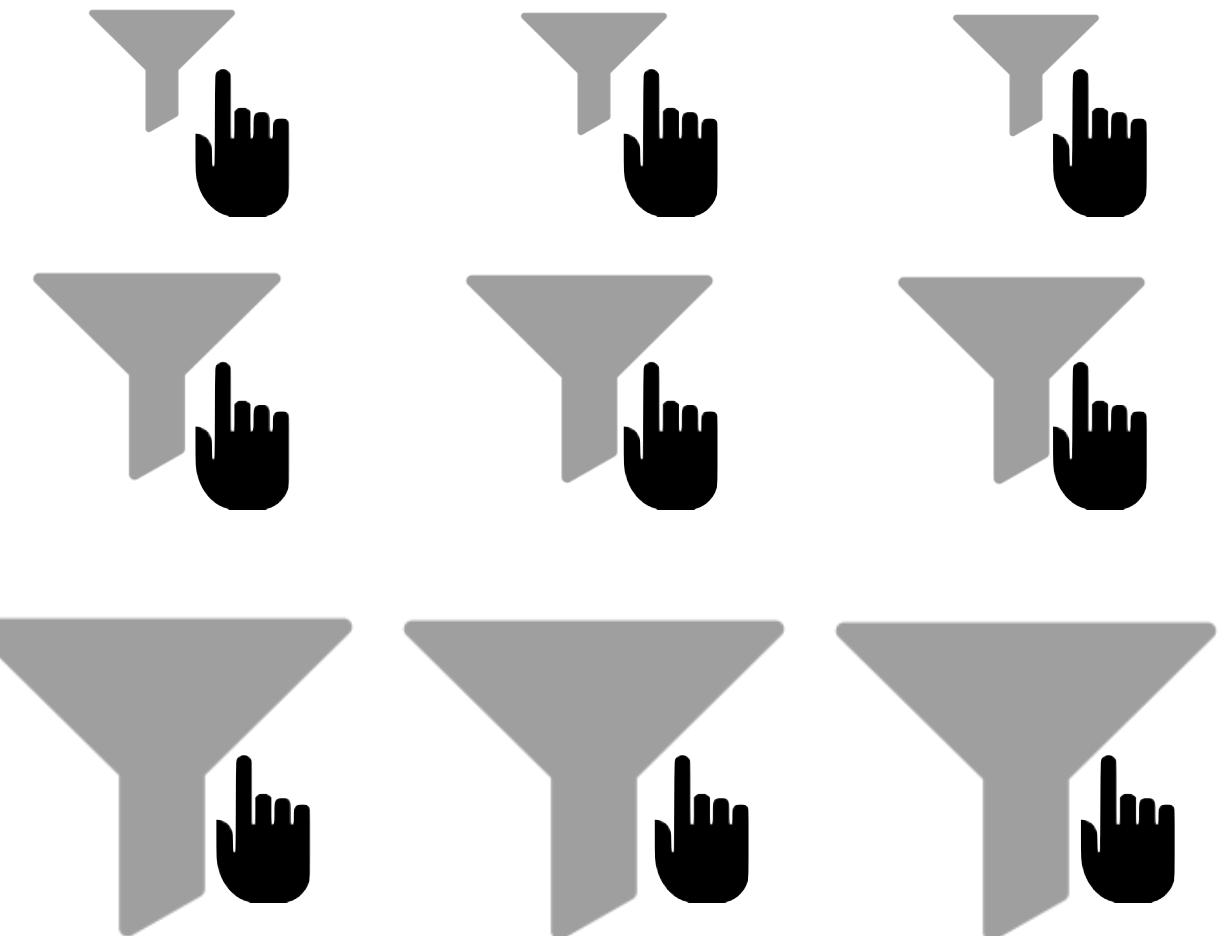
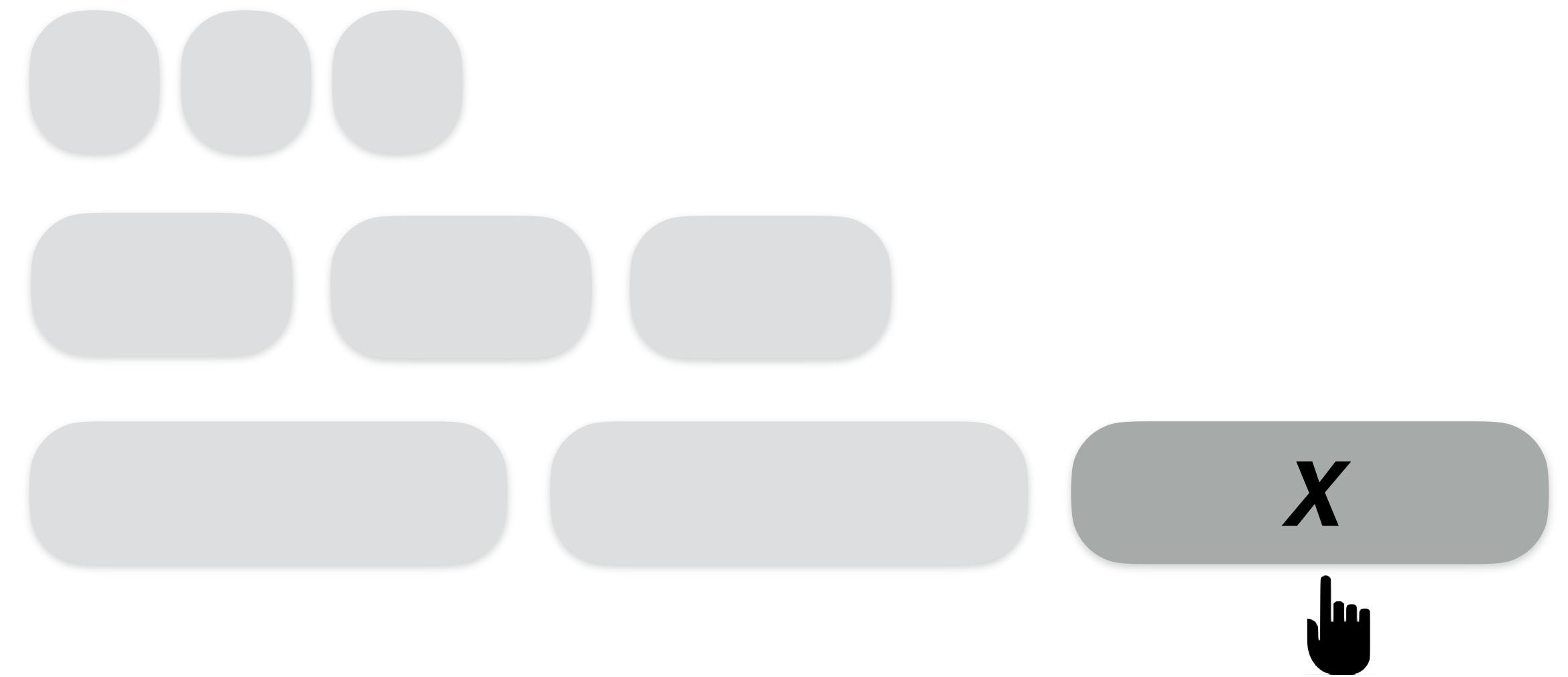


SIGMOD 2021

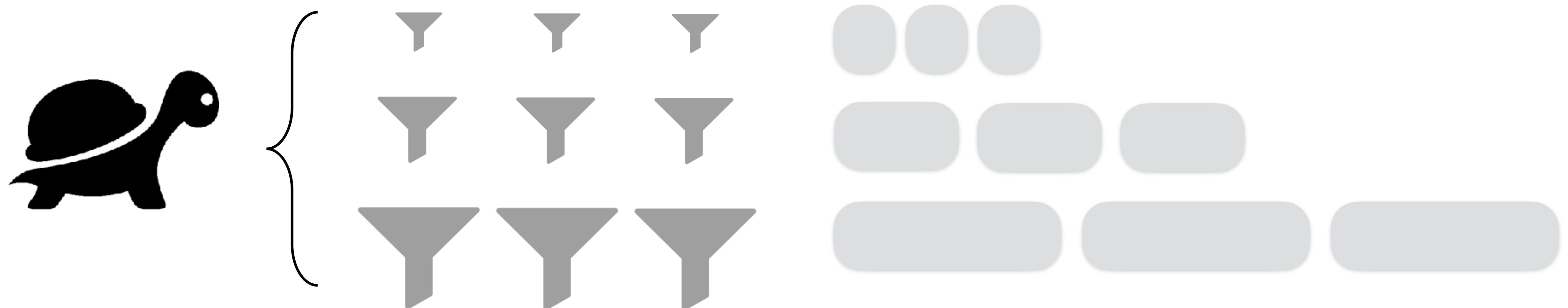


Bloom
filters



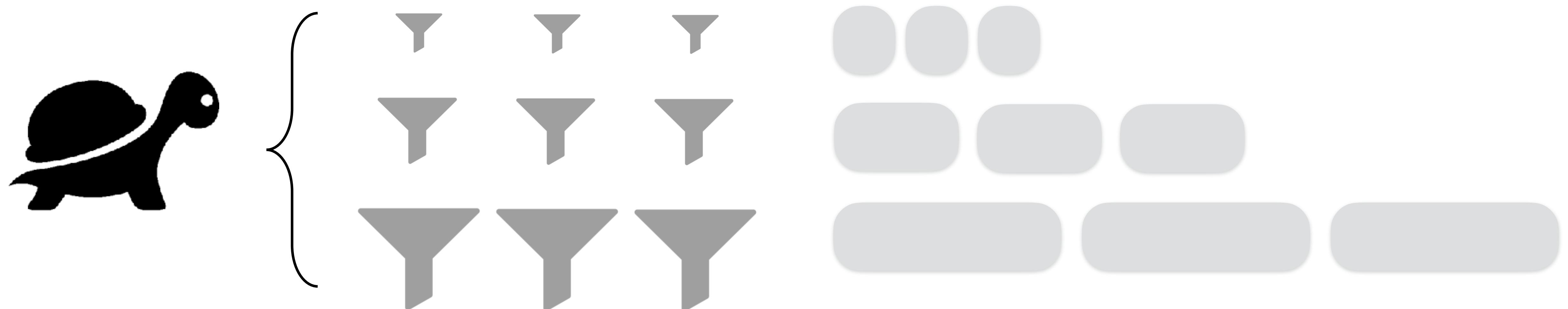
$O(R)$  $O(R)$ $O(R + M)$ $= O(M + L \cdot R)$ 

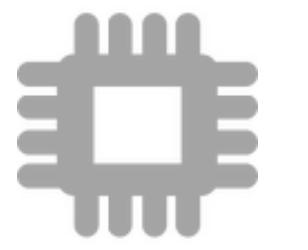
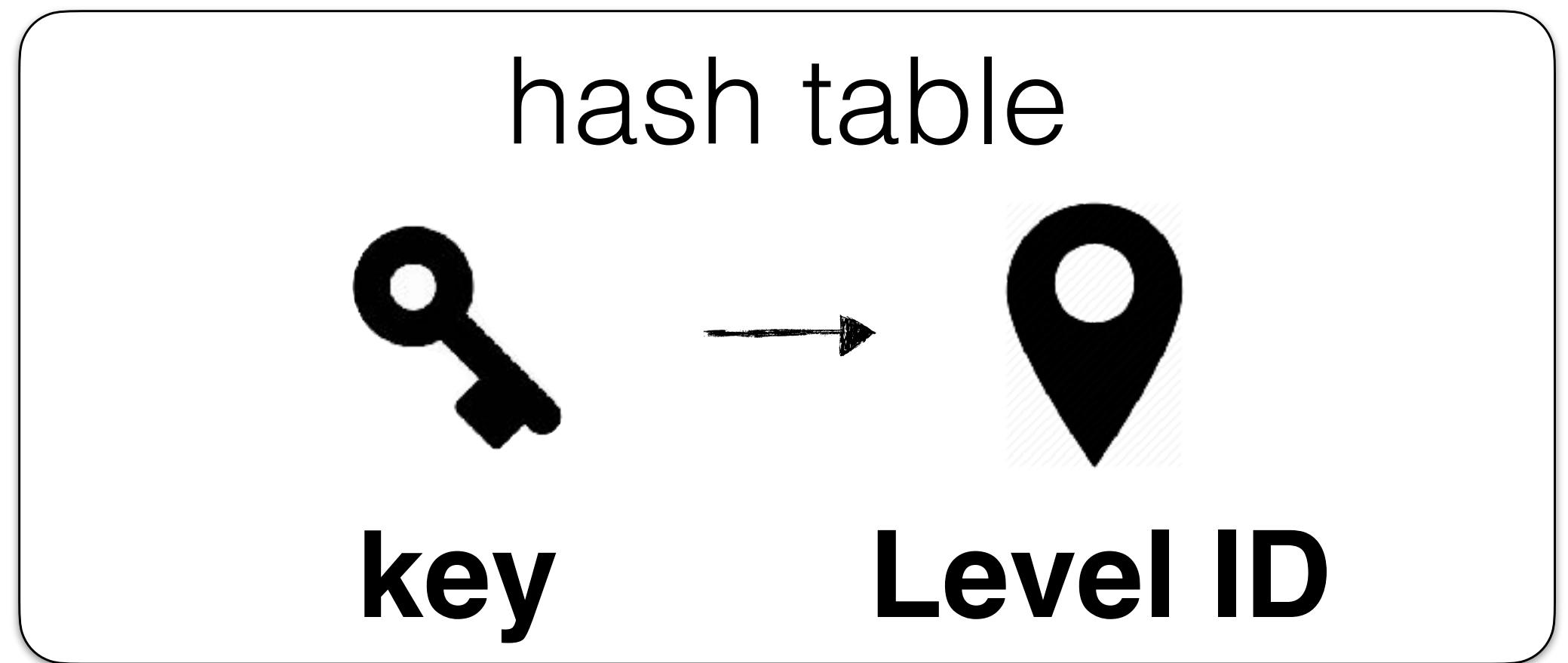
Problem: Reduce filter accesses



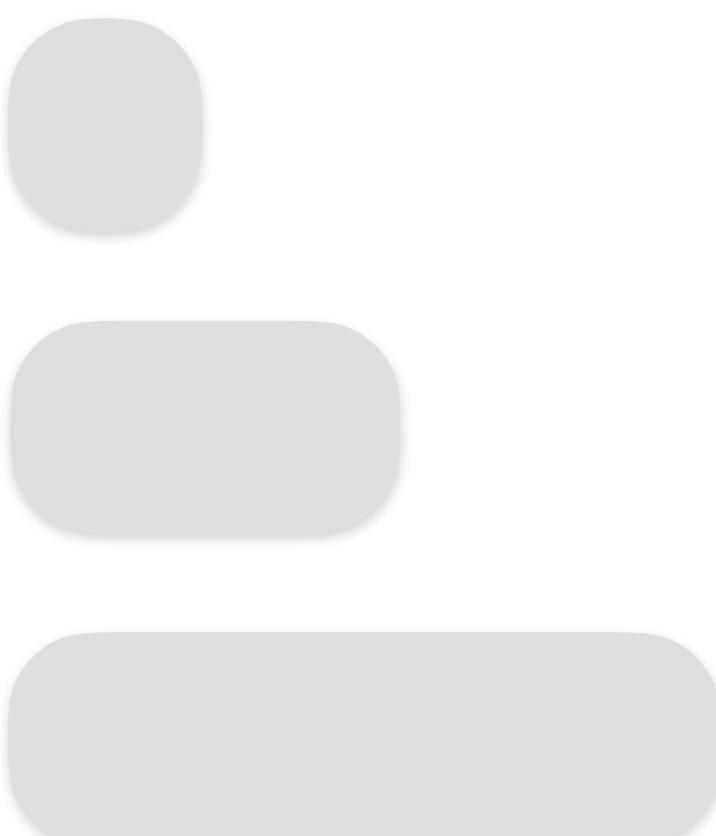
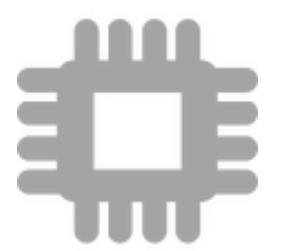
Problem: Reduce filter accesses

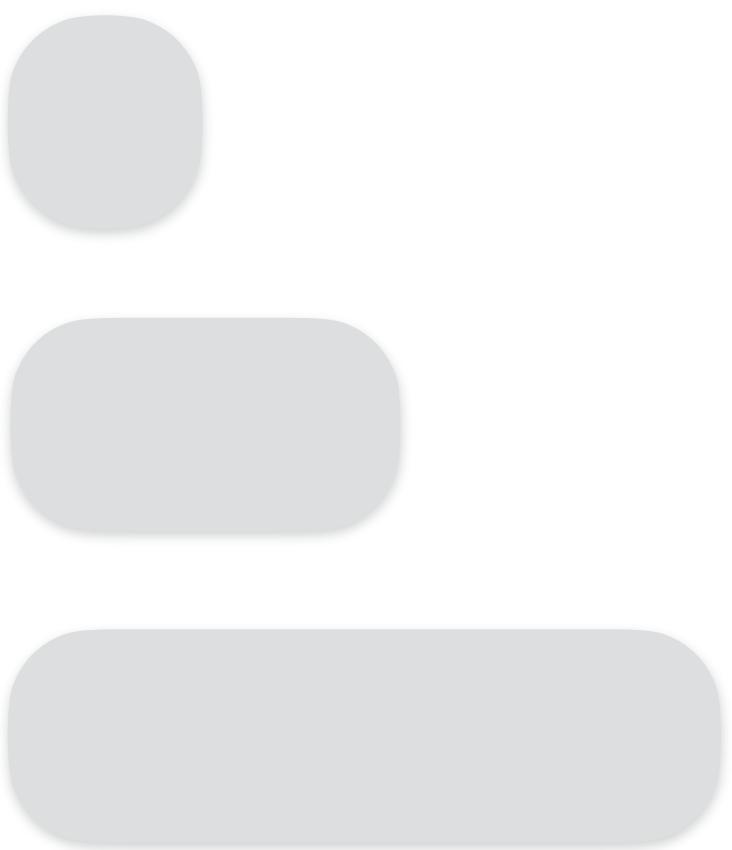
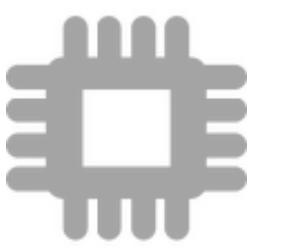
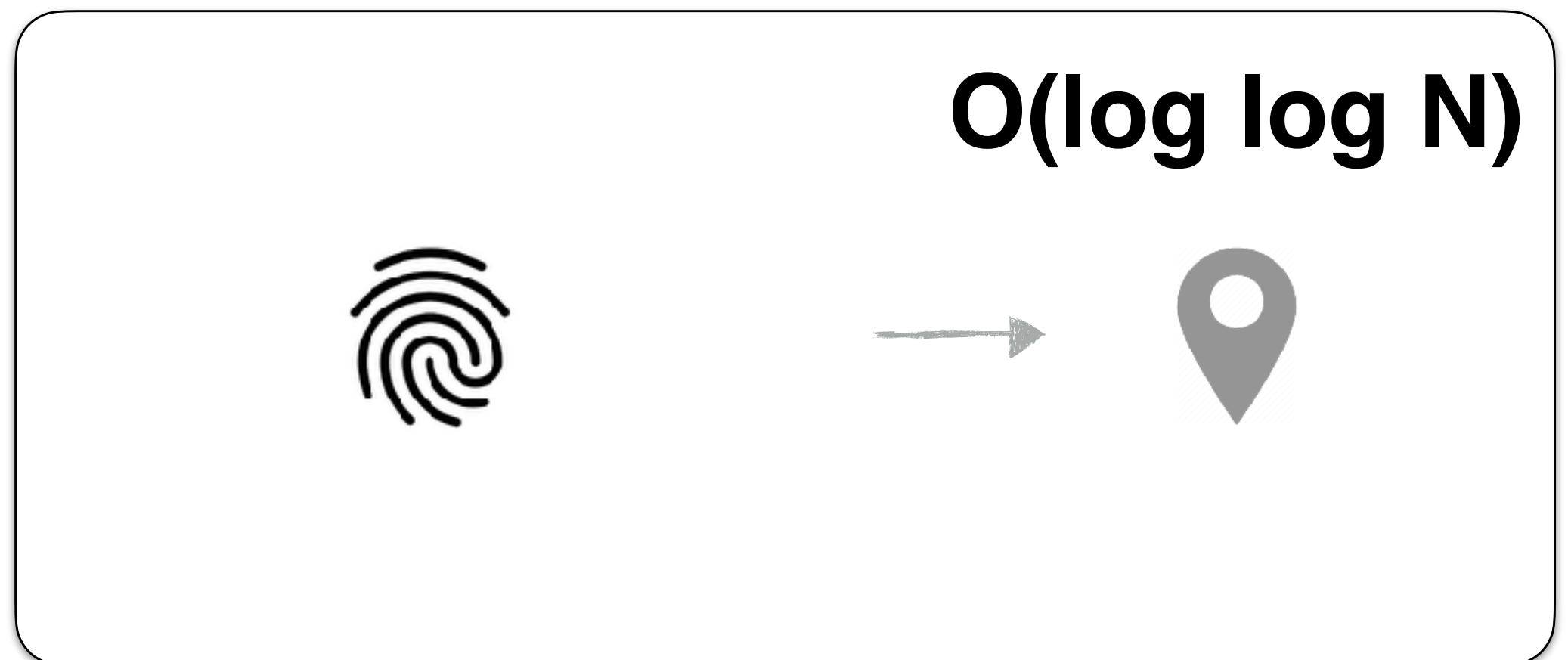
Constraint: Keep false positives & memory the same



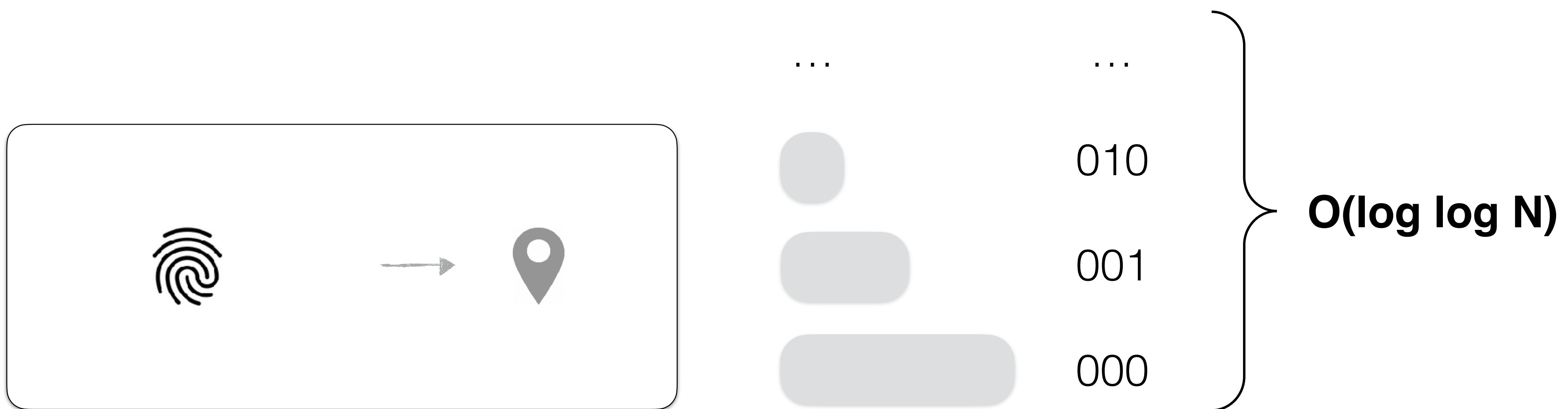


hash() =

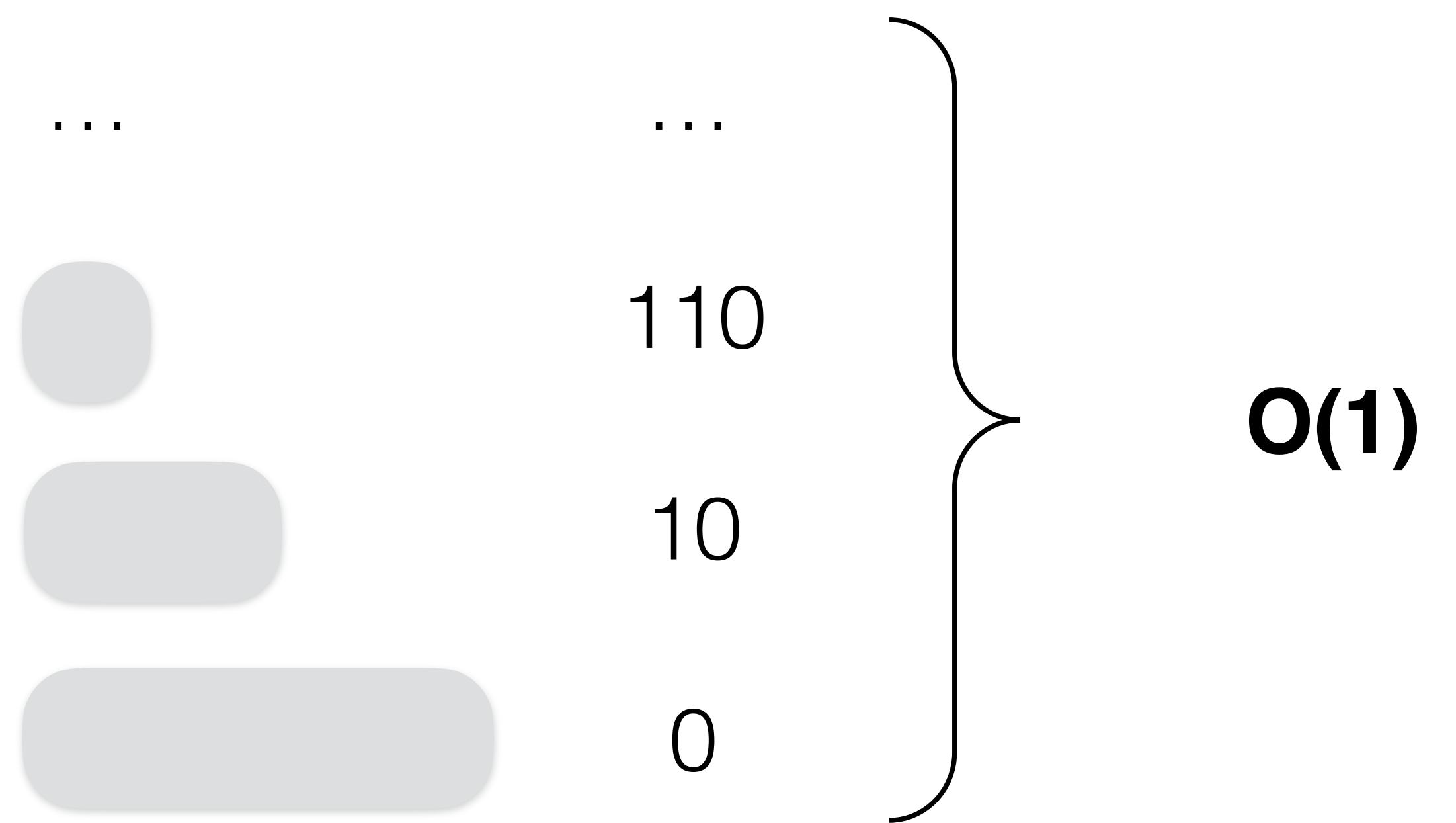
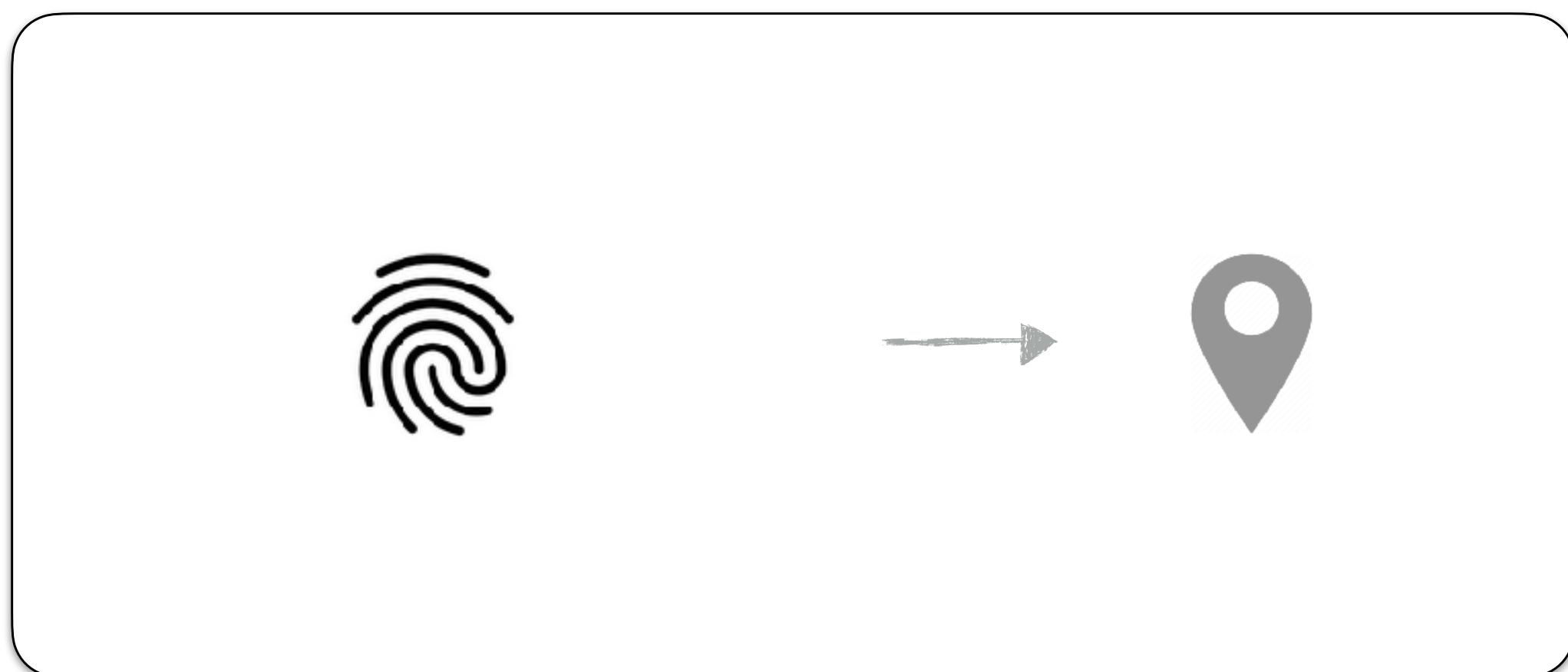




Binary encoding



e.g., unary encoding





Monkey



Chucky

Bits / entry

$O(M)$

\approx

$O(M)$

**memory
accesses**

$O(L + M)$

$>$

$O(1)$

get I/O
cost

$$O(e^{-M} \cdot L)$$



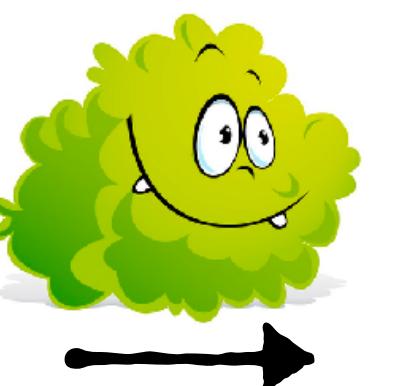
$$O(e^{-M})$$

insert I/O
cost

$$O((R \cdot L)/B)$$



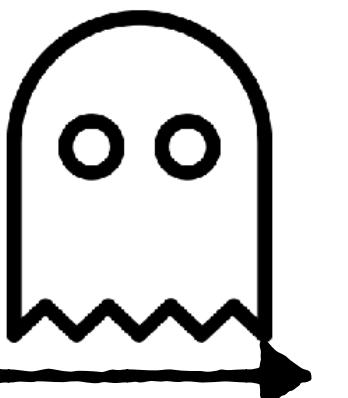
$$O((R+L)/B)$$



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

**transient
space-amp**

≈ 2



≈ 1

filter mem.
reads

$$O(M + L)$$



$$O(1)$$

$$L = \log_R N/P$$

(Costs assuming leveling)

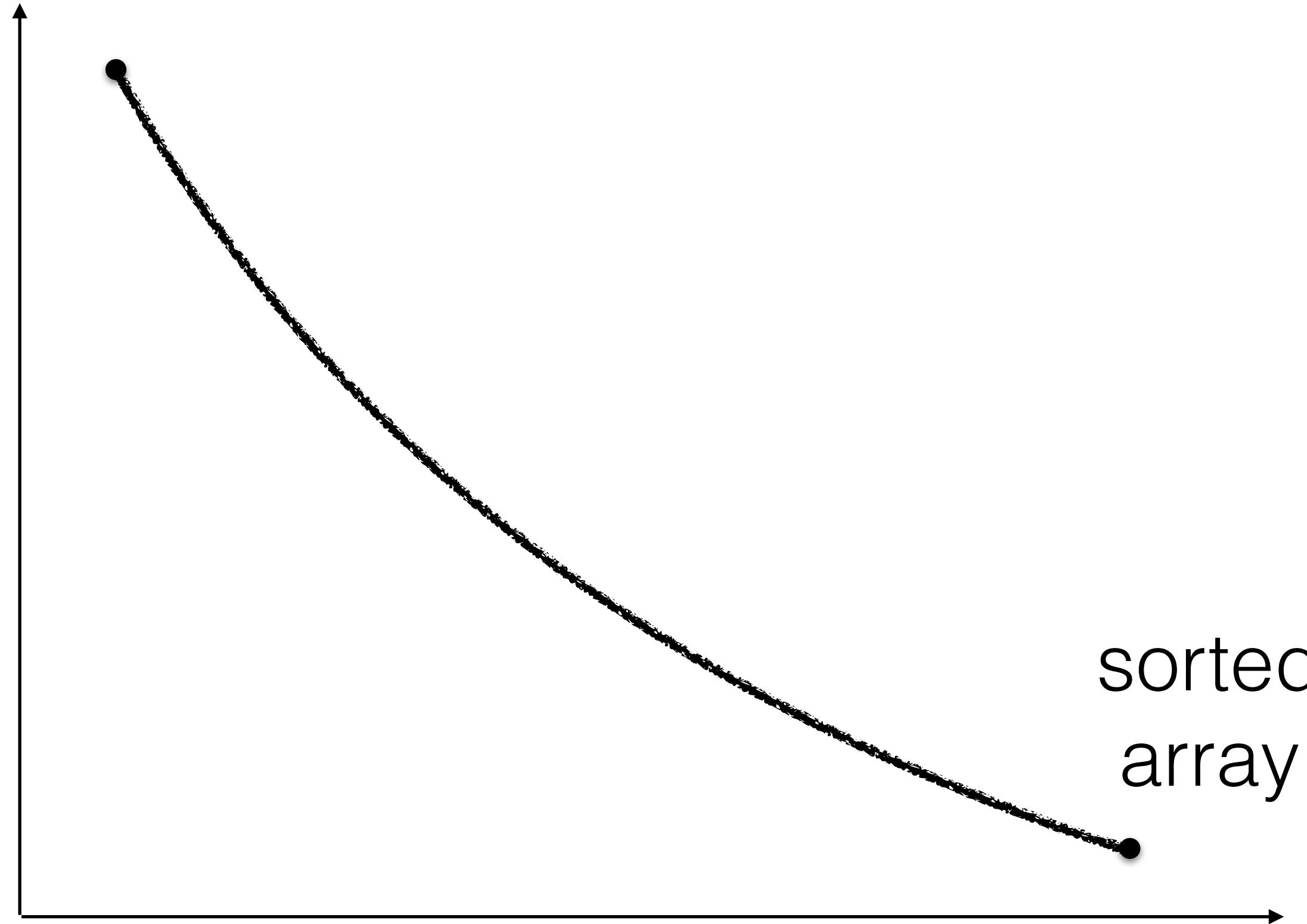
**Our transactional storage engines
can now far better accommodate exponential data growth**



Reads

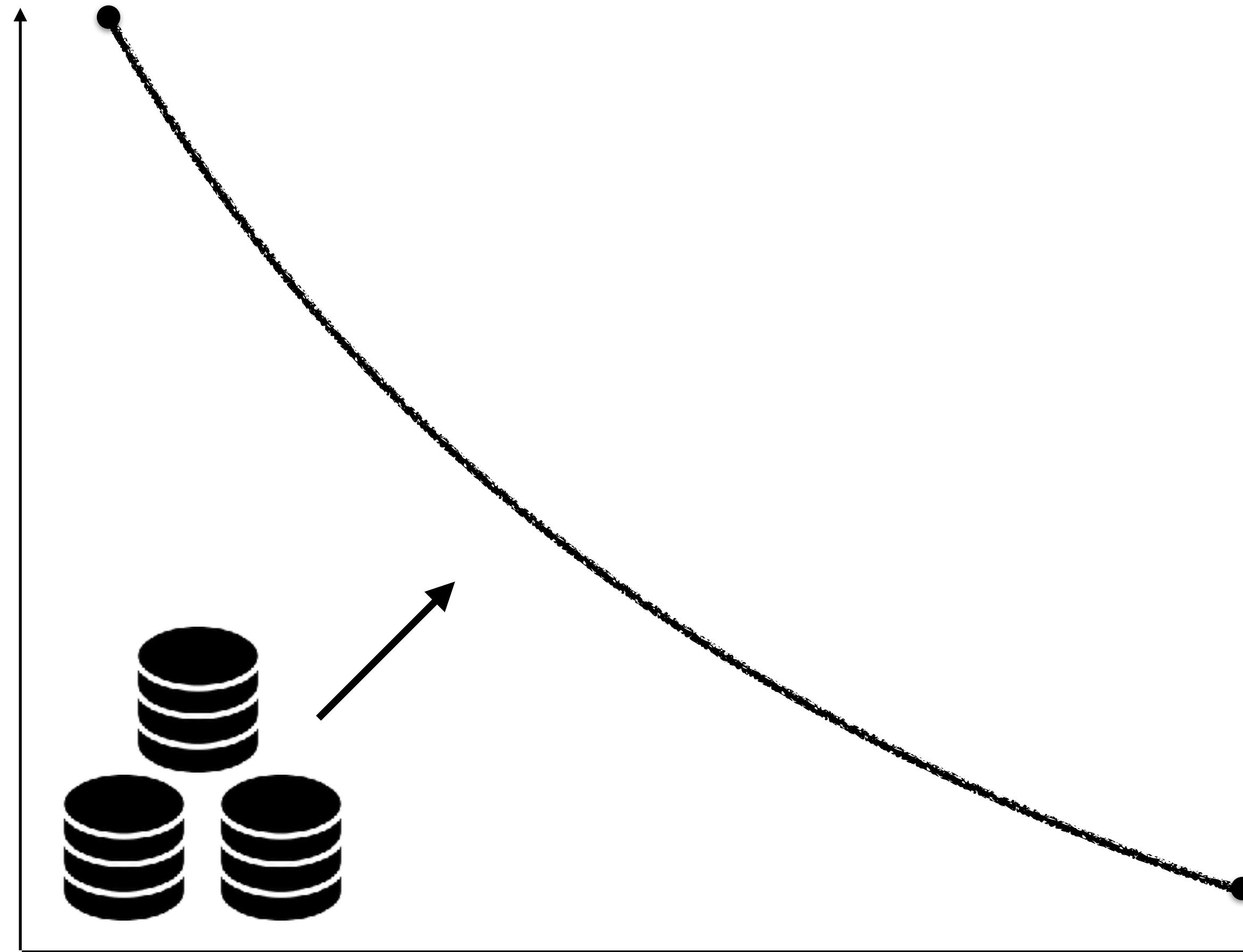


log



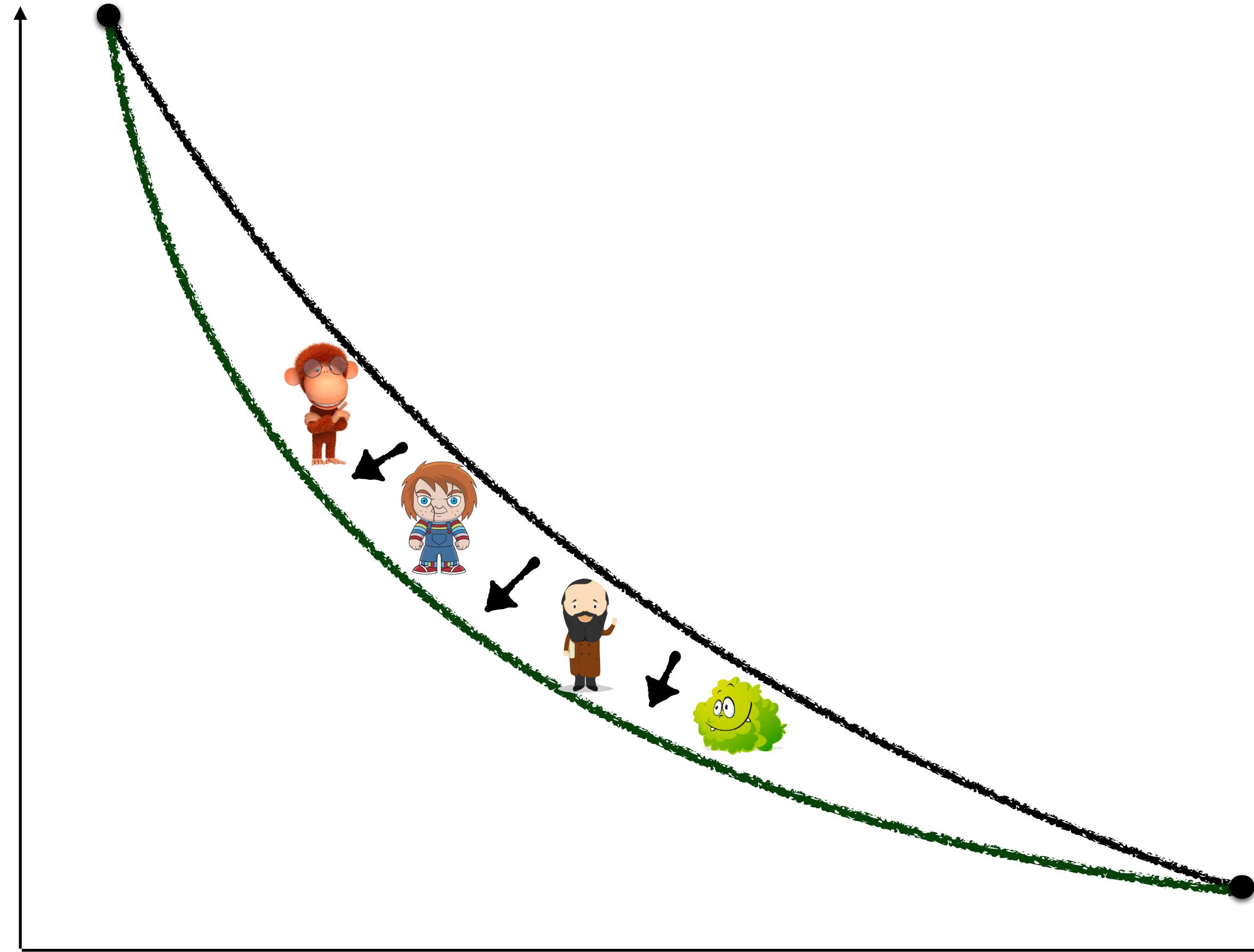
Writes

Reads



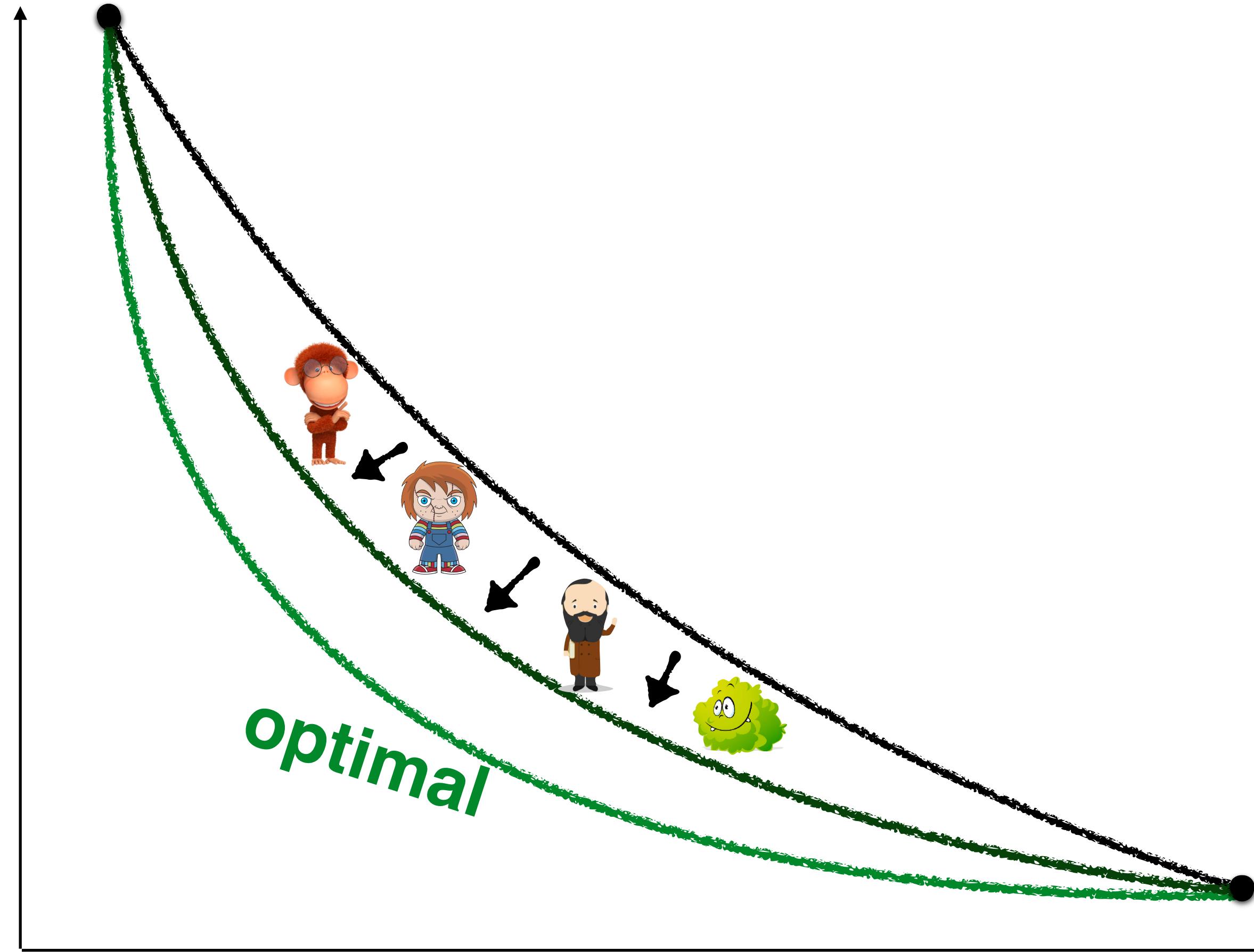
Writes

Reads



Writes

Reads

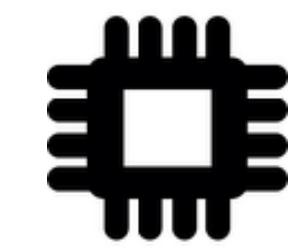


Writes

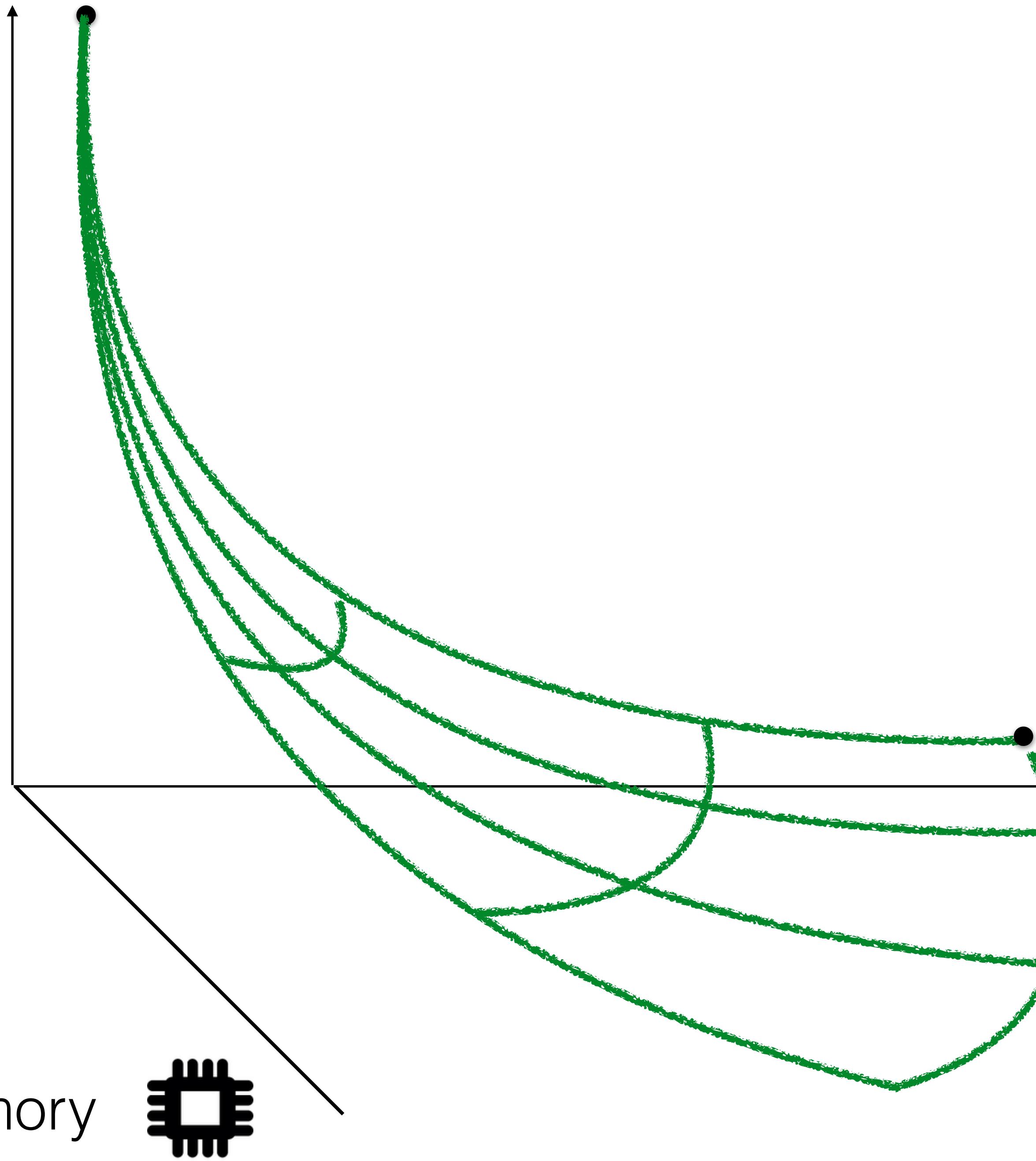
Reads



Memory

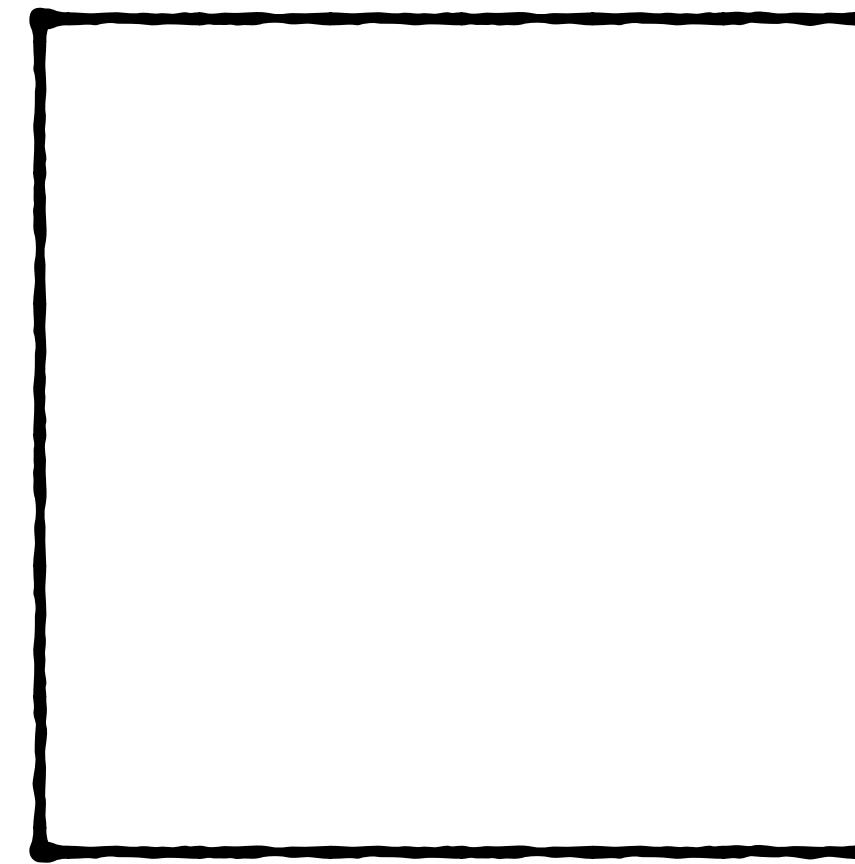


Writes



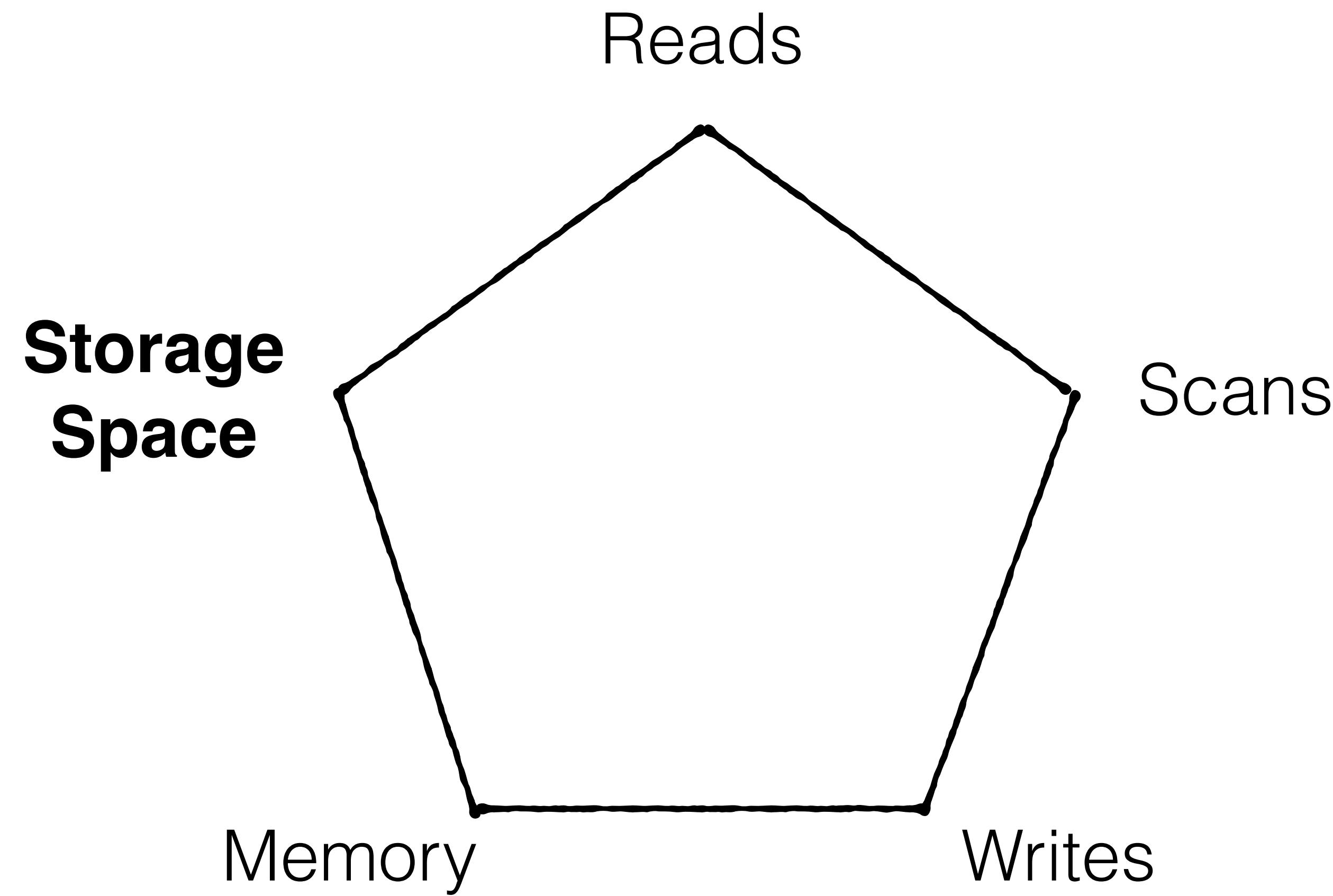
Reads

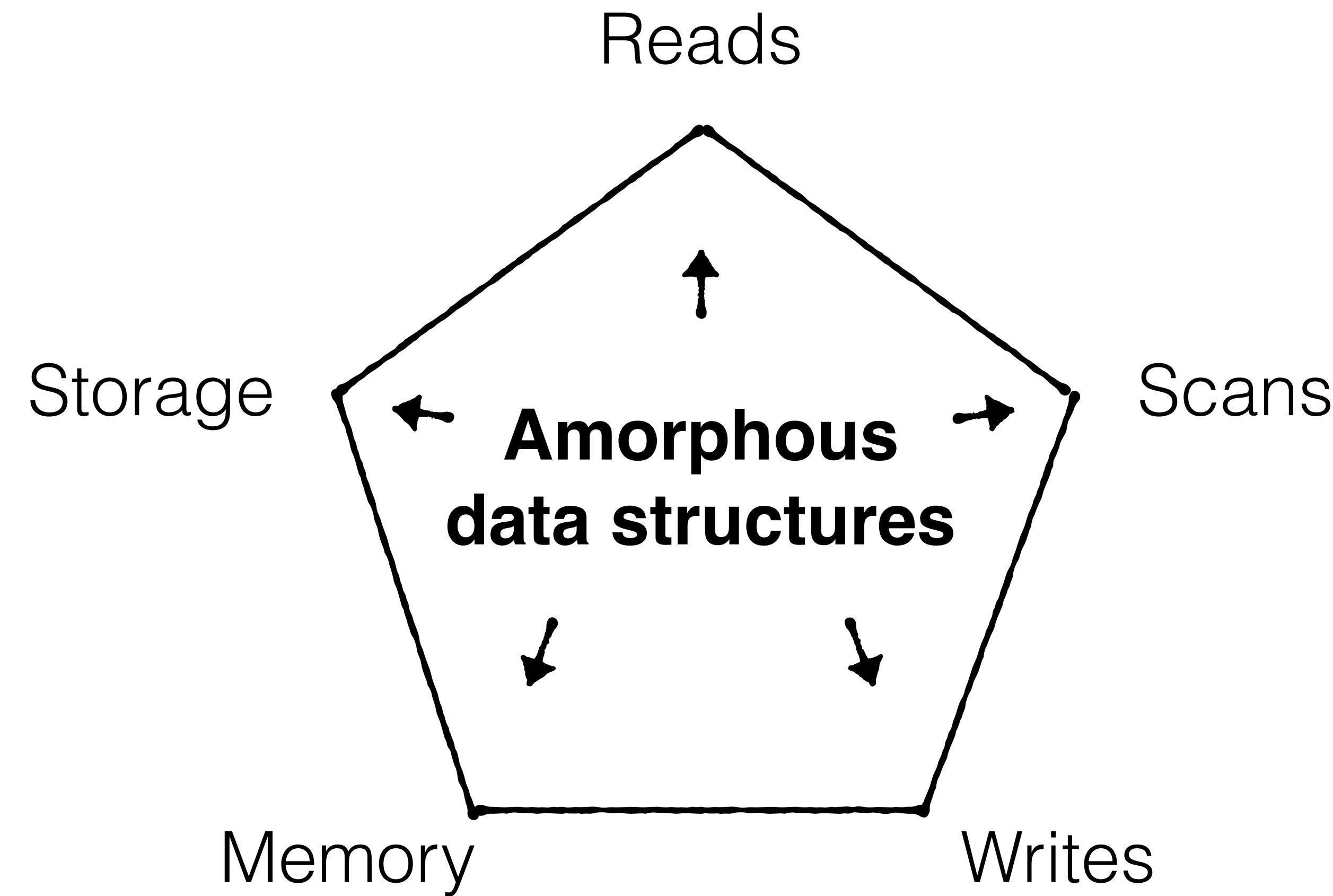
Range Scans



Memory

Writes





Thanks



DASlab
@ Harvard SEAS

PLIOPS >