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

Niv Dayan

Projects

56 groups -
mostly of threes

157 / 169 students
registered

Feedback week
next week - will
announce when.

Midterm

Oct 14 in class

Open book

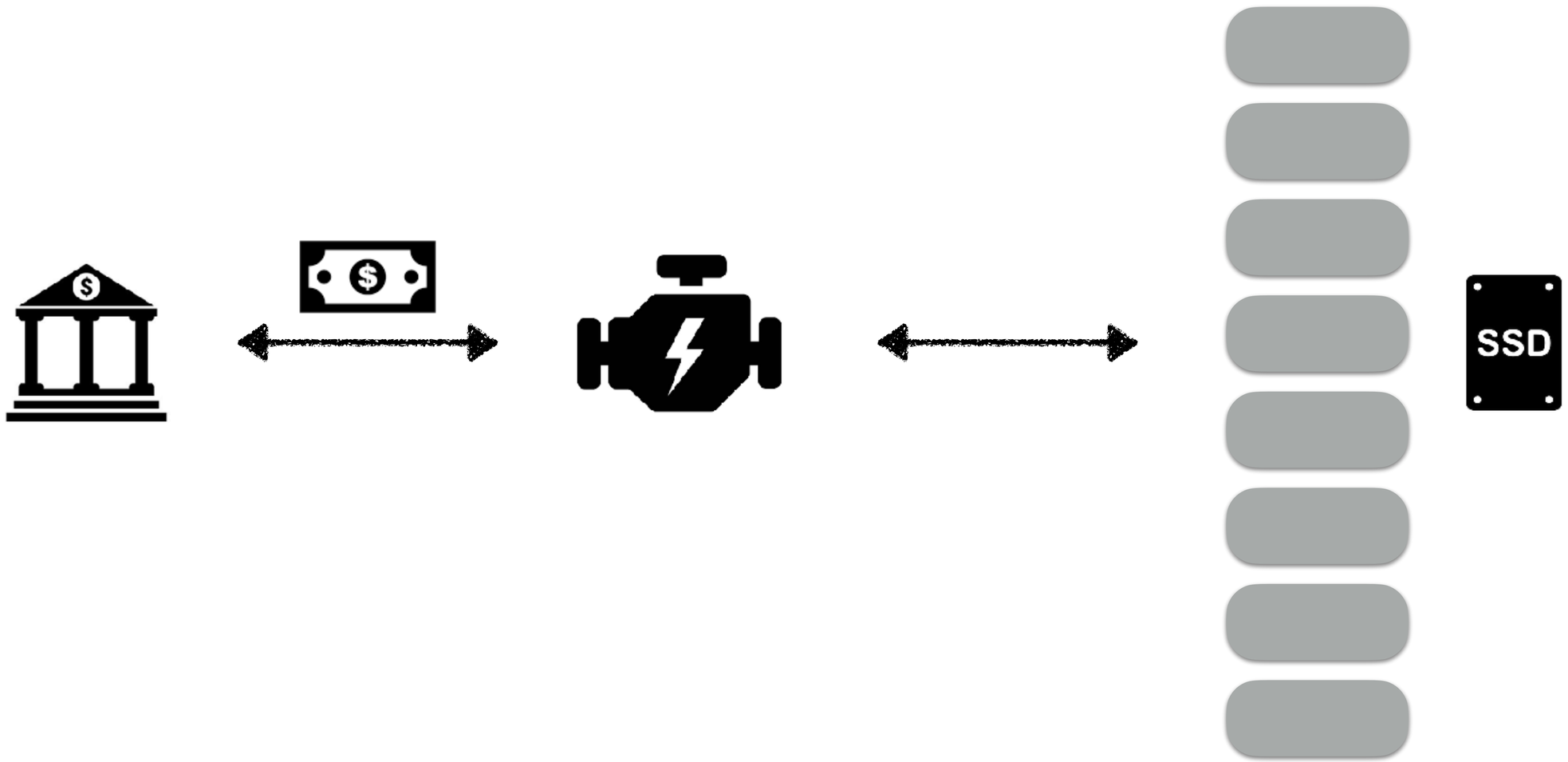
Today

Recap on LSM

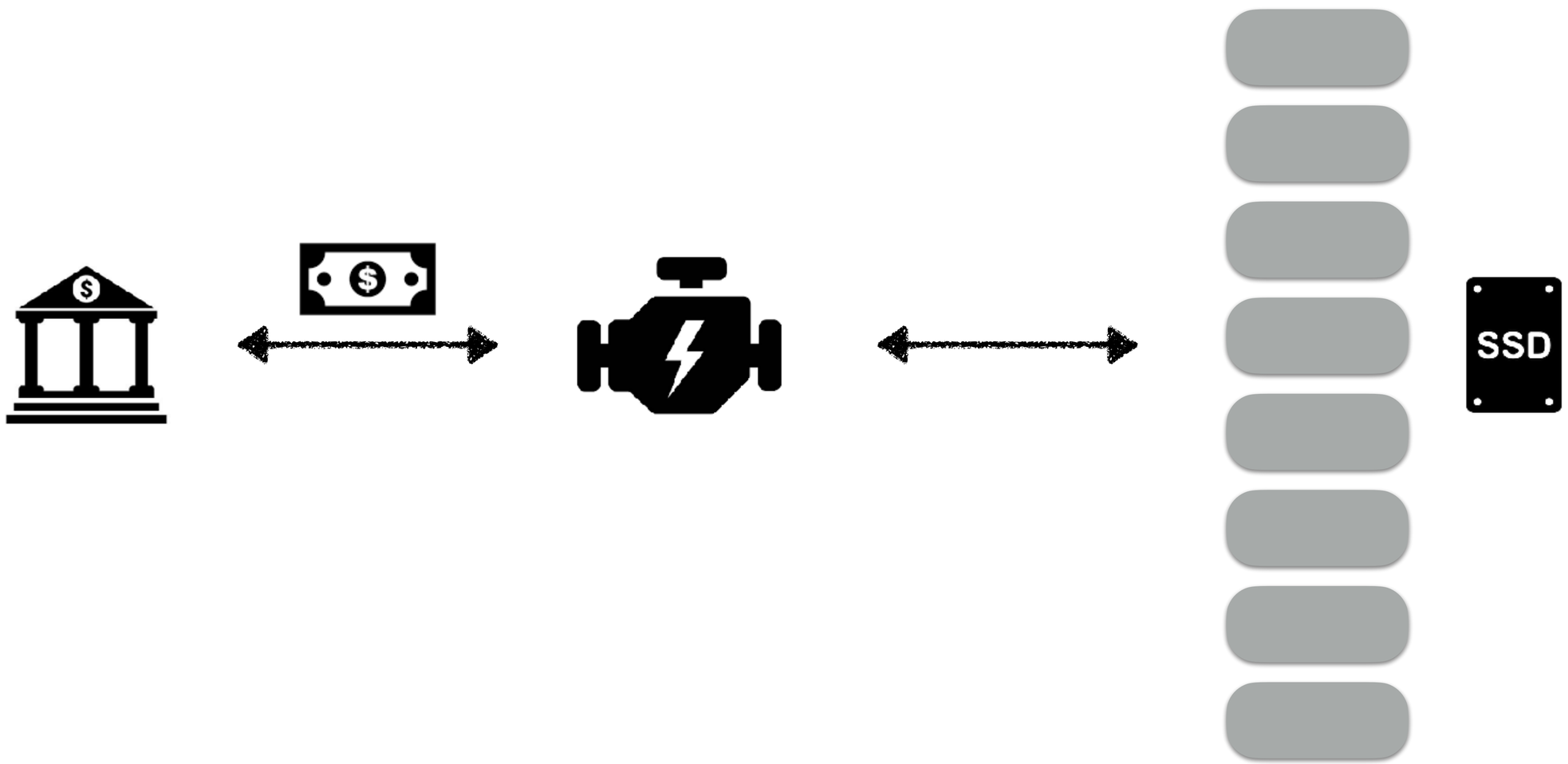
Bloom Filters

Research Lecture

Many DB operations are:

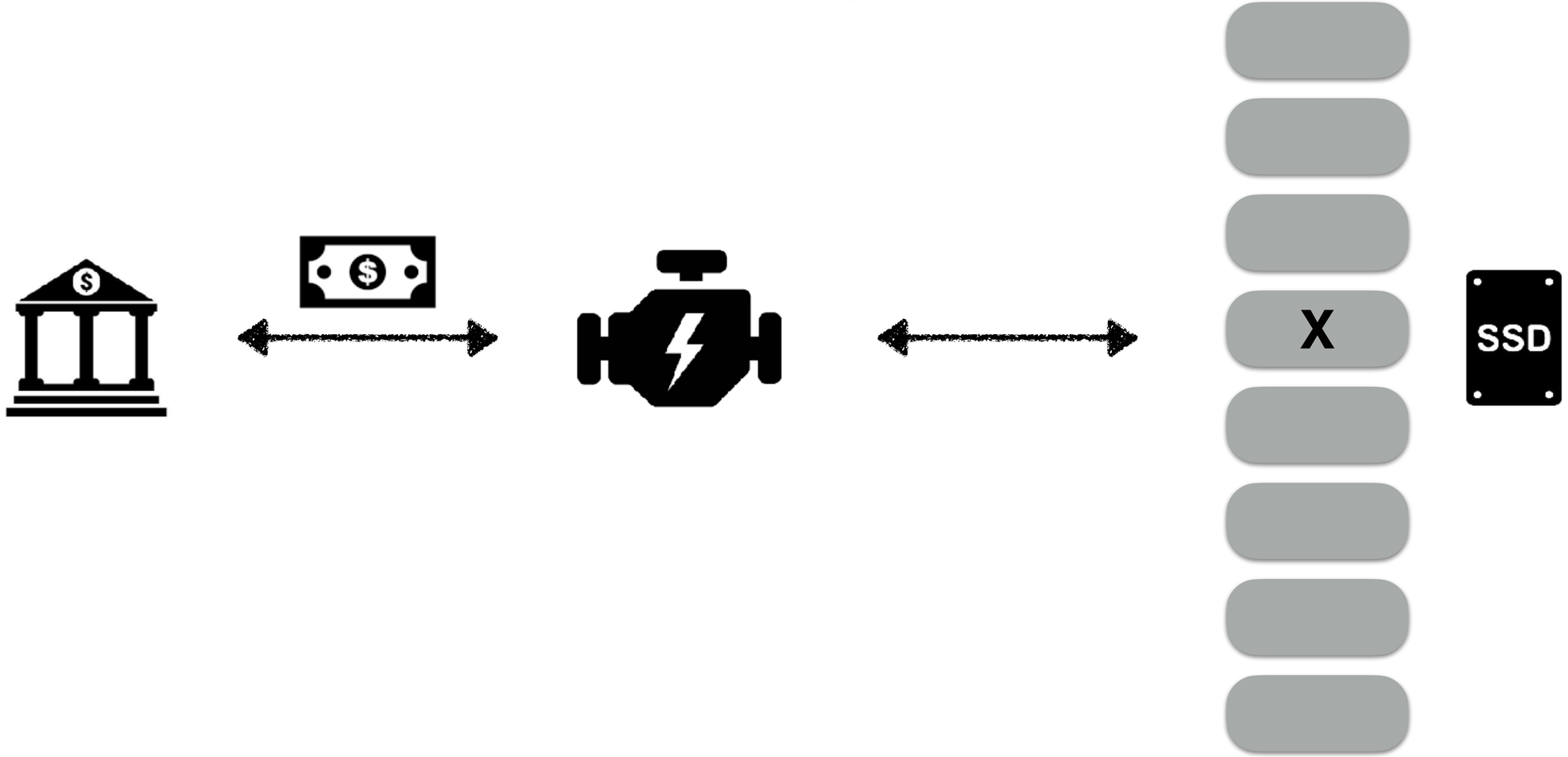


Many DB operations are: **selective**



Many DB operations are: **selective**

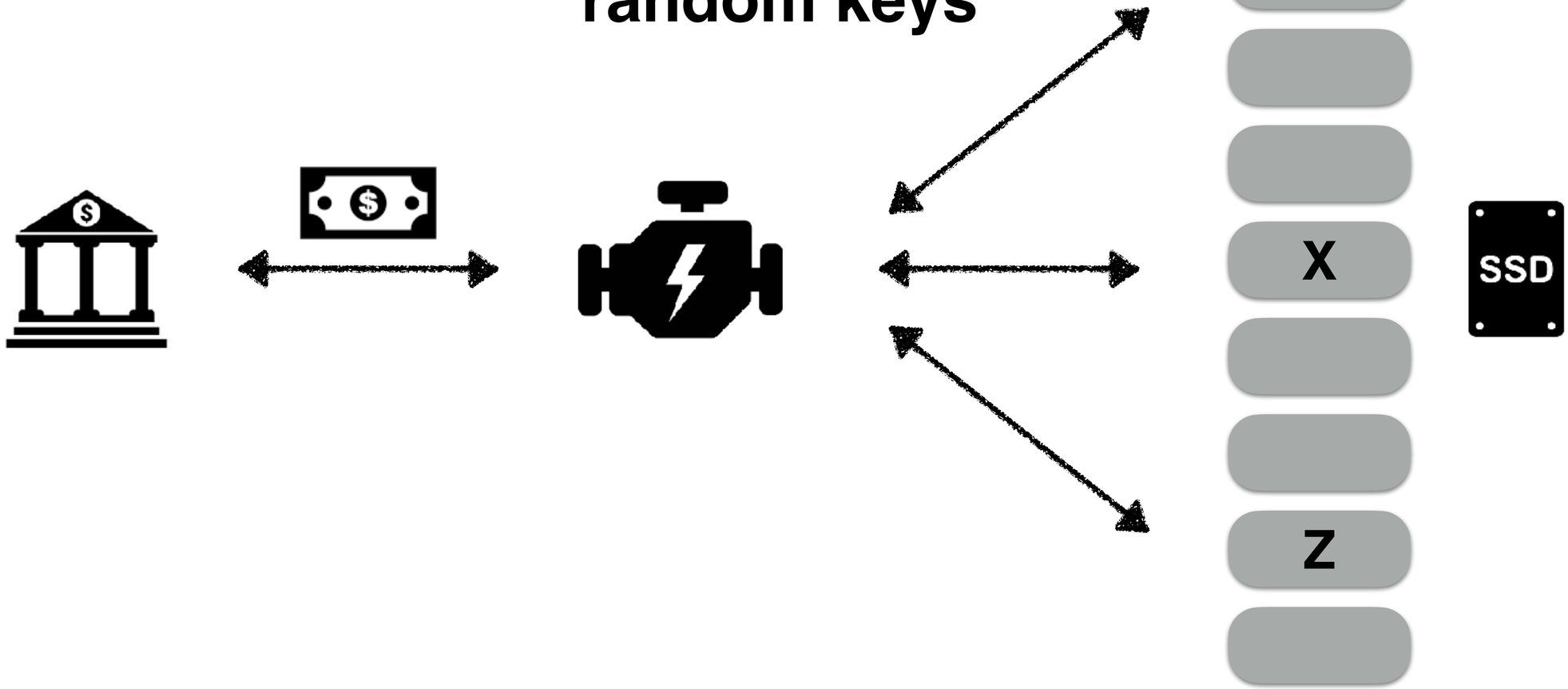
small data items



Many DB operations are: **selective**

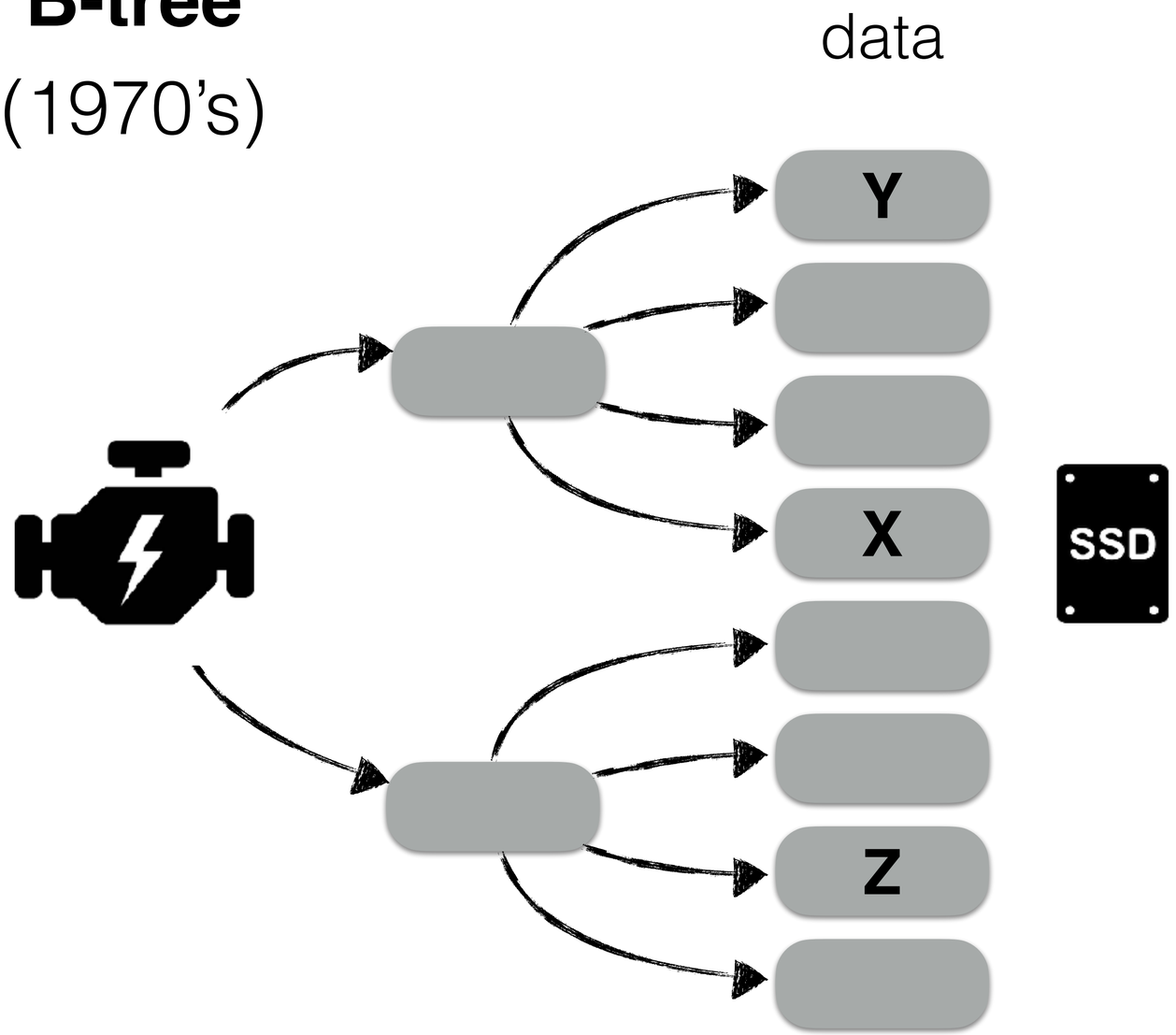
small data items

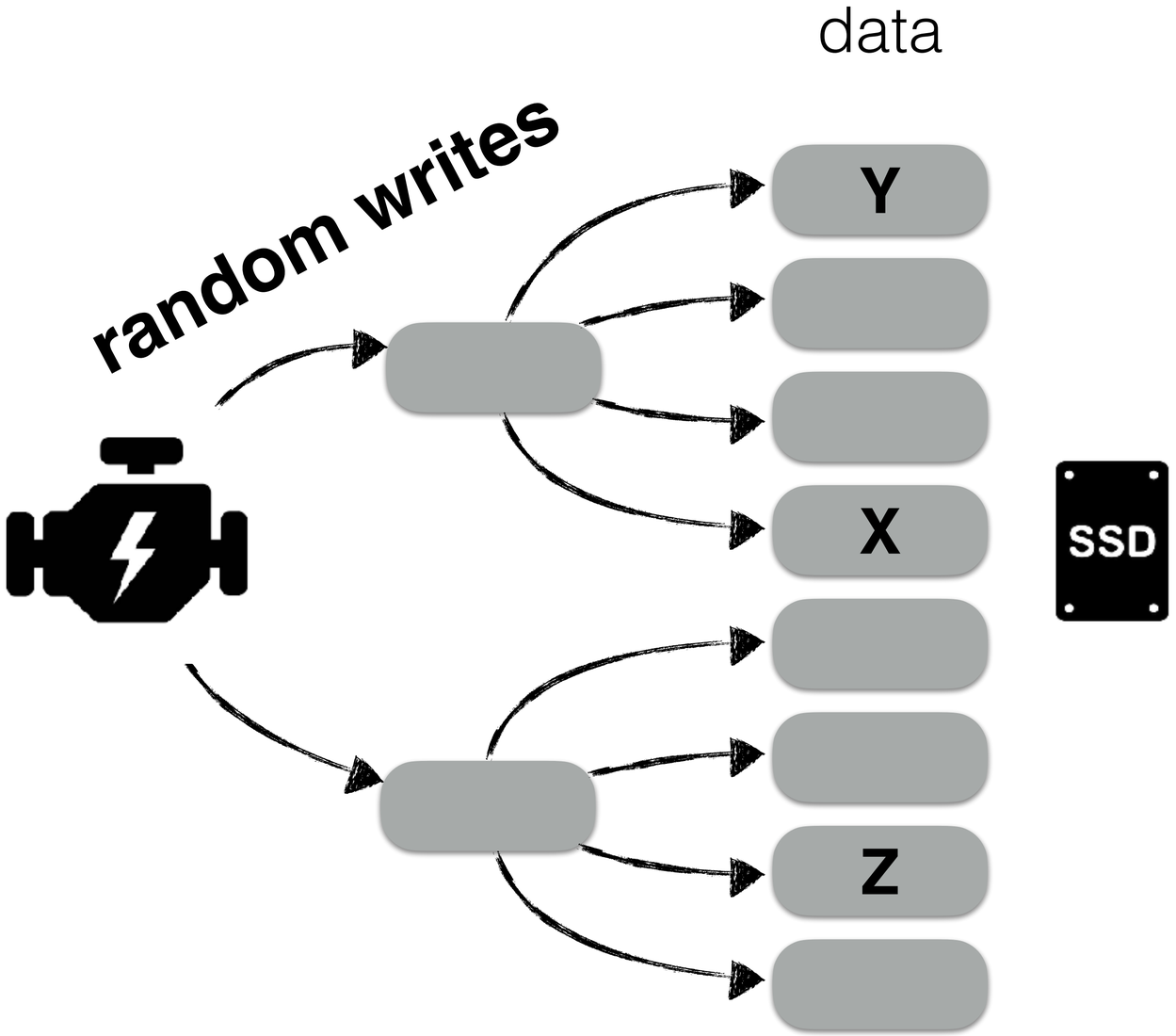
random keys



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

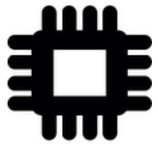
4KB access
garbage-collection

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

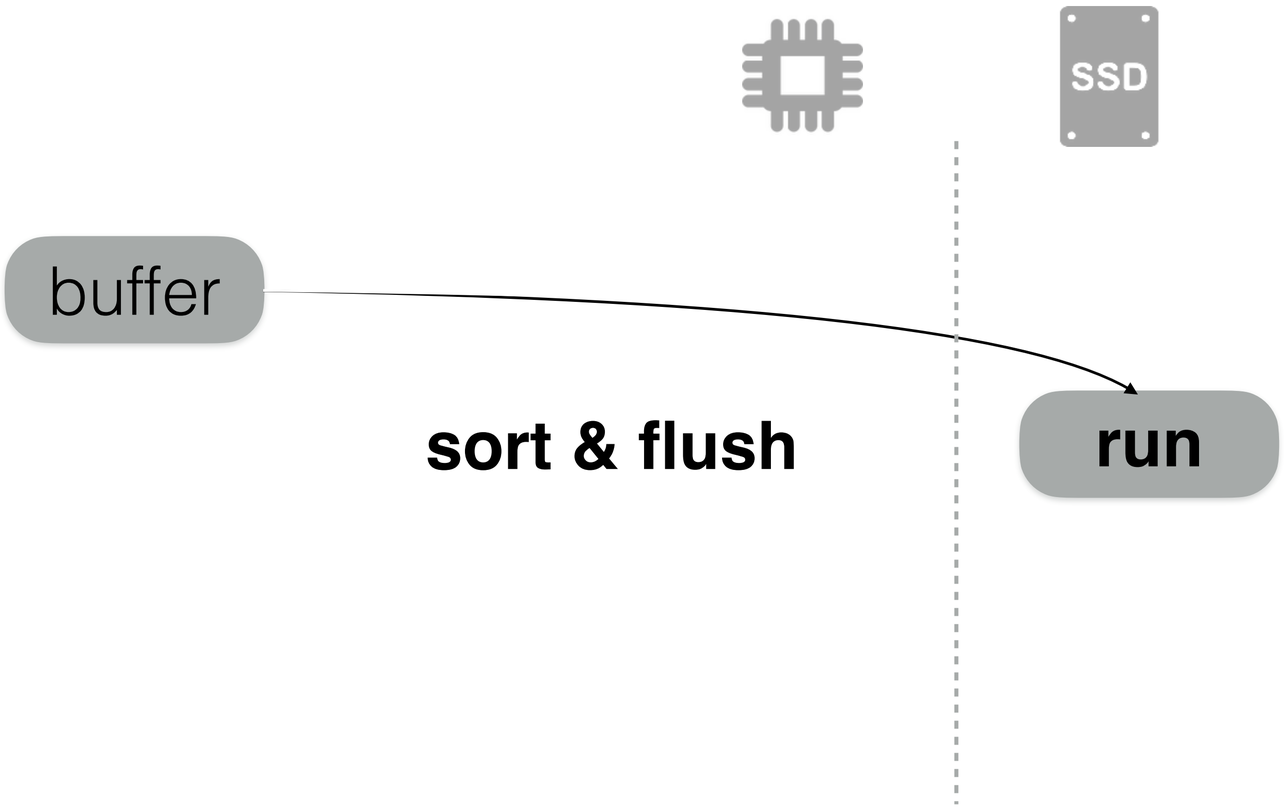
2 4 5

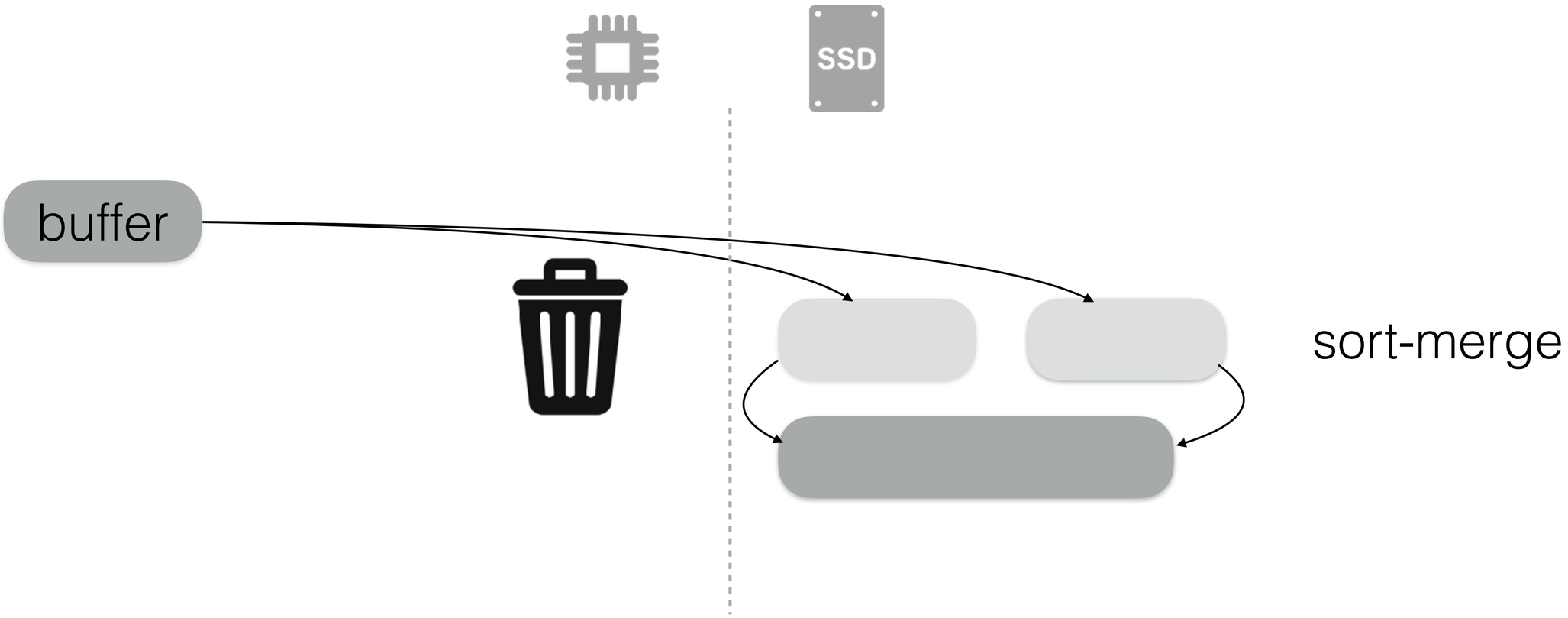
1 2 3 4 5 6

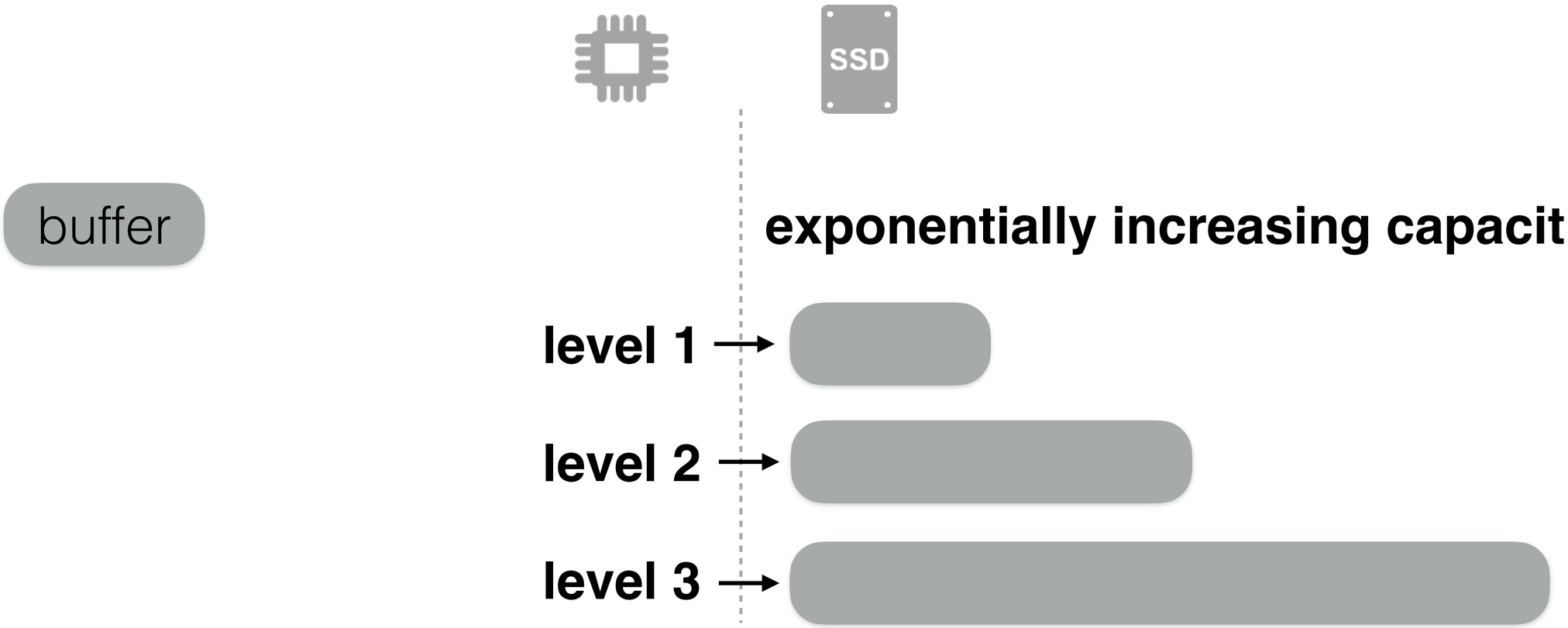
Inserts/updates/deletes
of key-value pairs

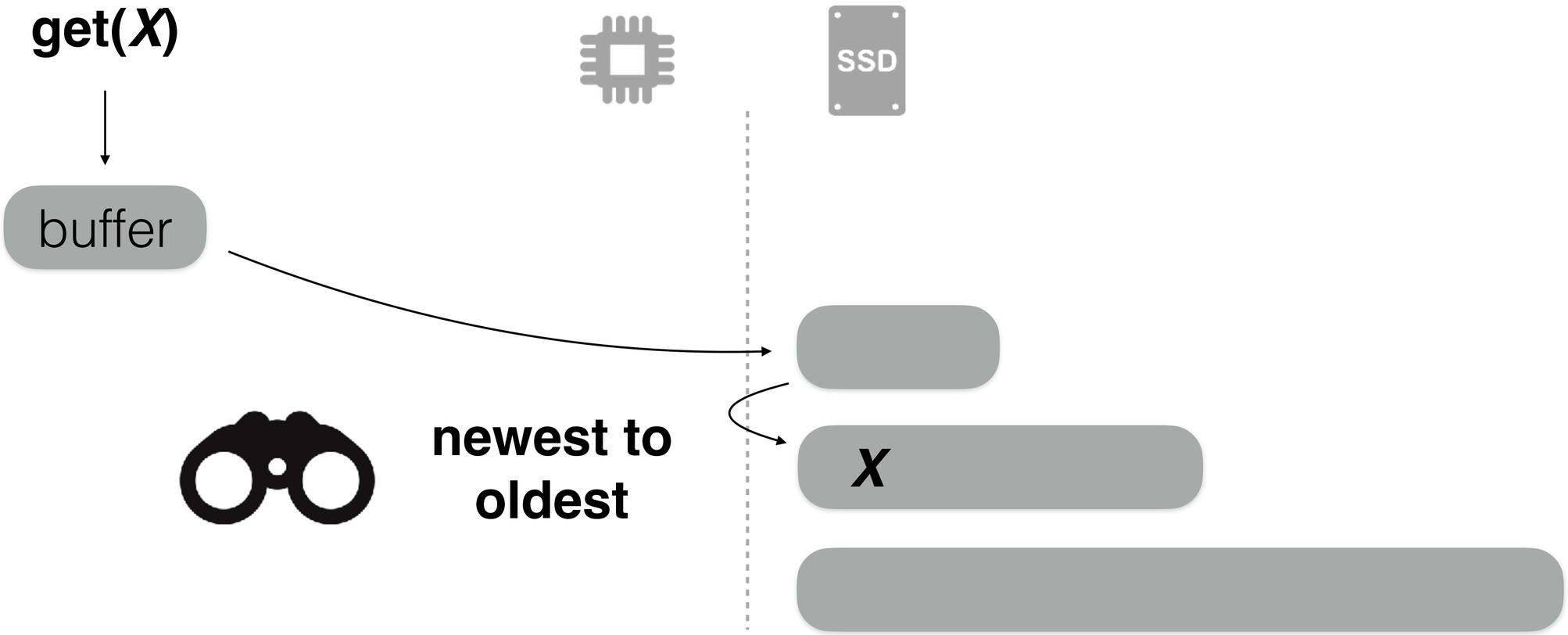
↓

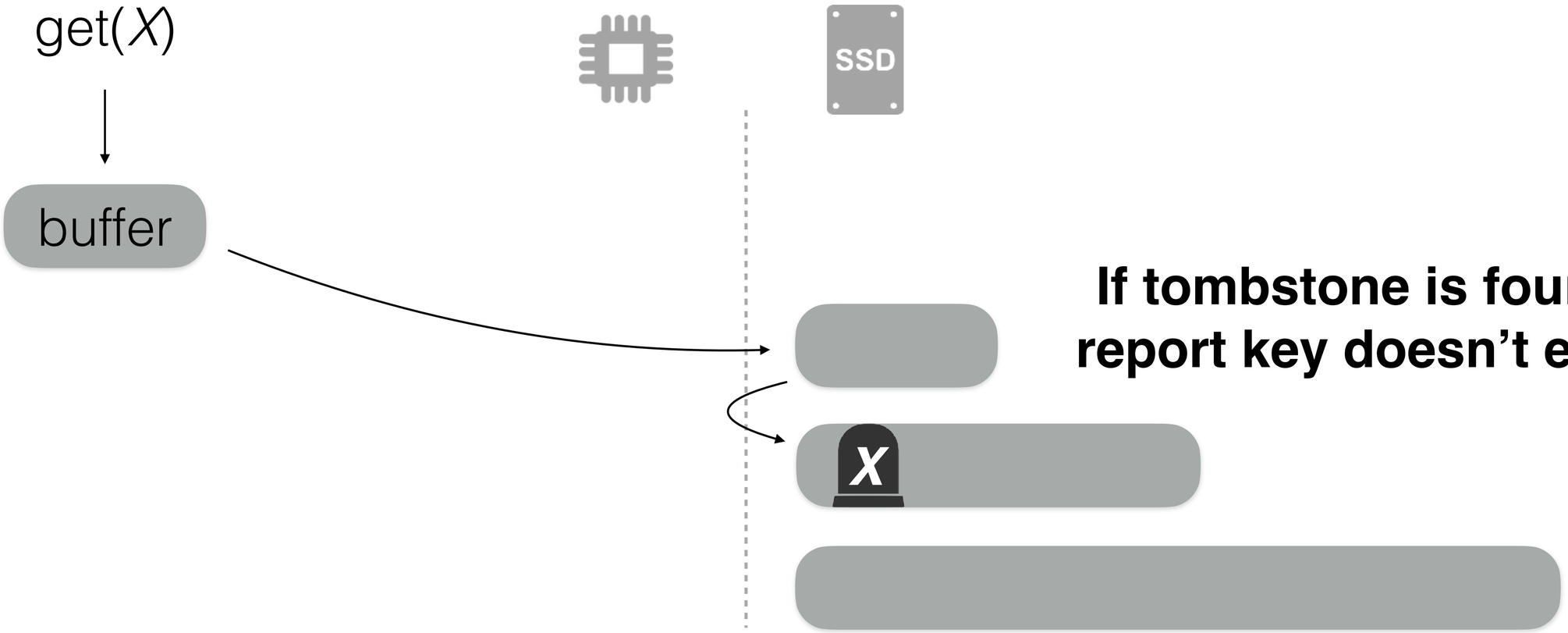
buffer


SSD
buffer
sort & flush
run


SSD
buffer
sort-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

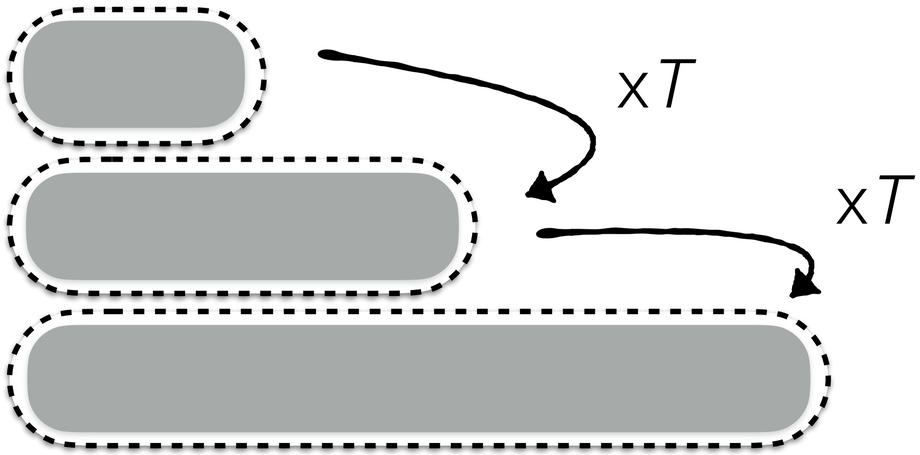
levels $L = \log_T(N/P)$

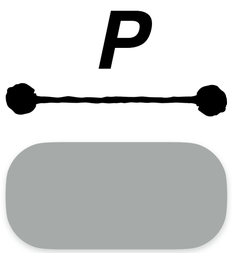
levels L = $\log_T(N/P)$

data size

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

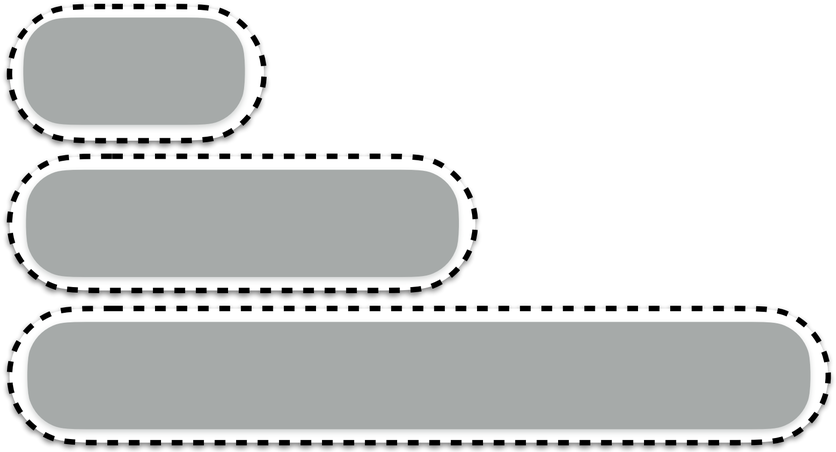
size ratio

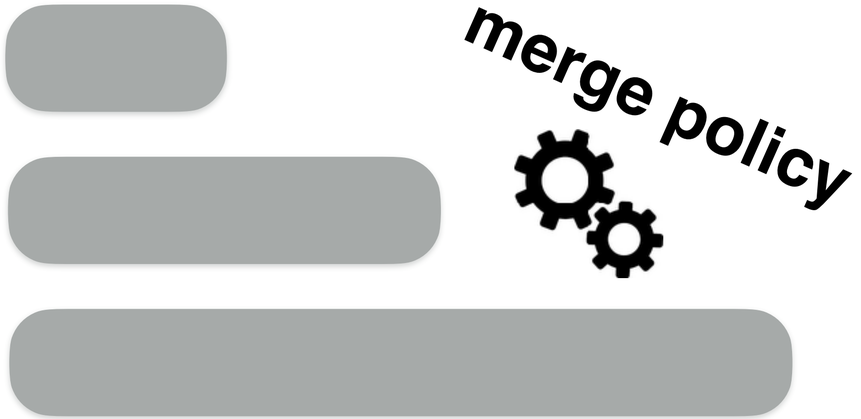




levels L = $\log_T(N/P)$

buffer size




merge policy

writes

merge policy

reads

writes

merge policy

reads

two merge policies

Tiering

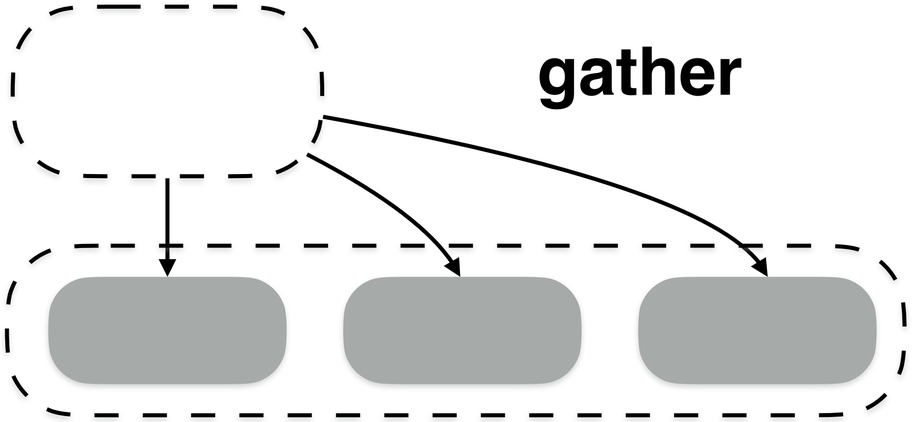
Leveling

Tiering

Leveling

Tiering

Leveling



Tiering

Leveling

gather

merge & flush

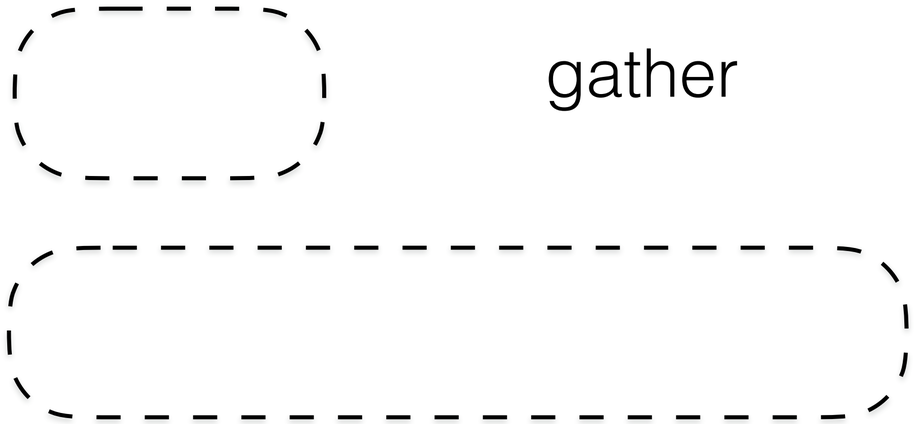
Tiering

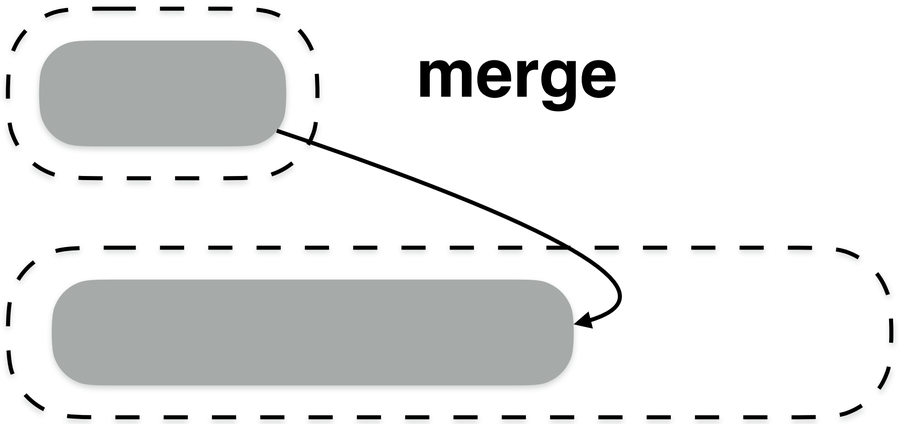
Leveling

gather

Tiering

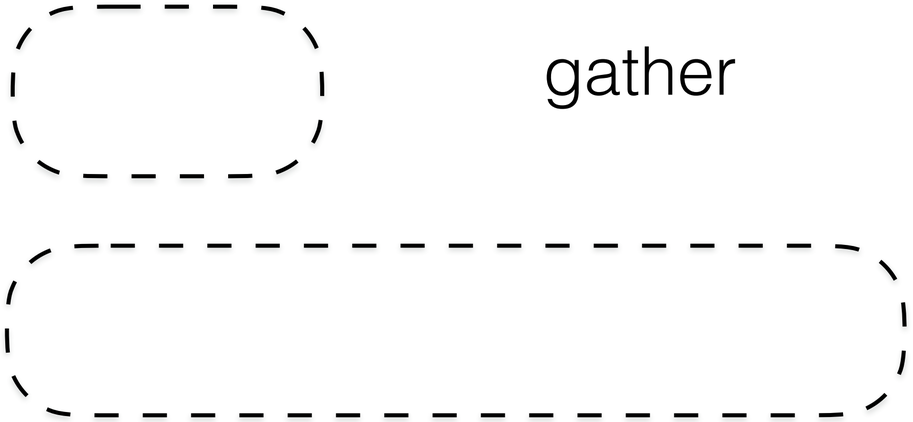
Leveling

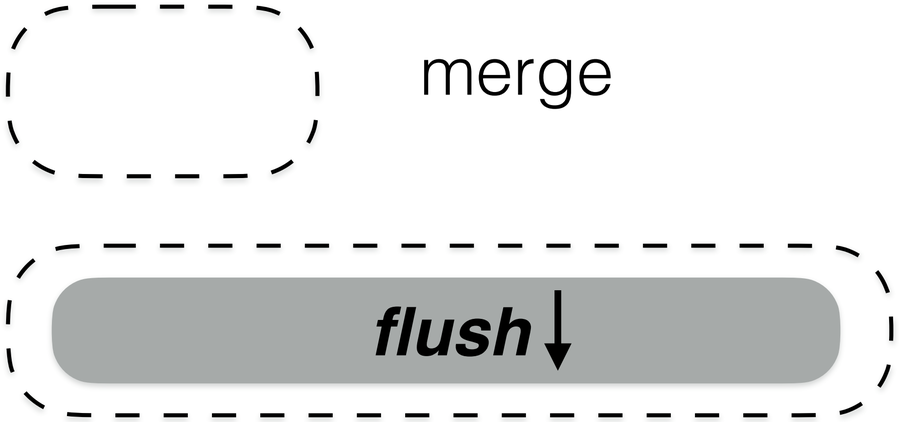




Tiering

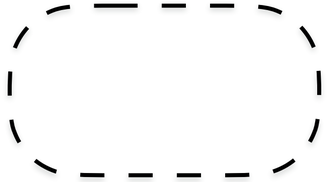
Leveling



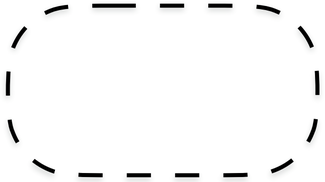


Tiering

Leveling



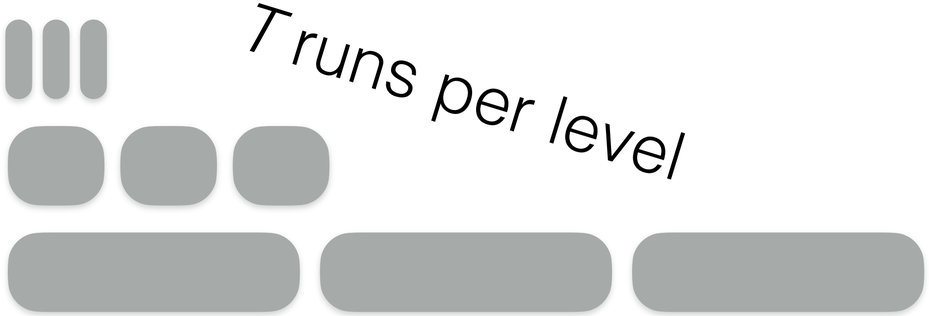
gather



merge

Tiering

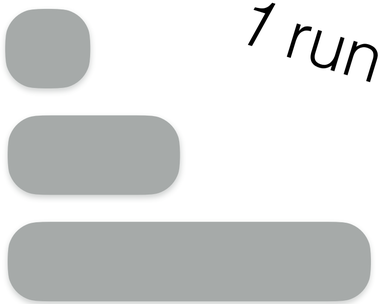
Leveling



size ratio *T*

Tiering

Leveling



1 run per level

1 run per level

size ratio $T = 2$

Tiering

Leveling

size ratio *T*

Tiering

Leveling

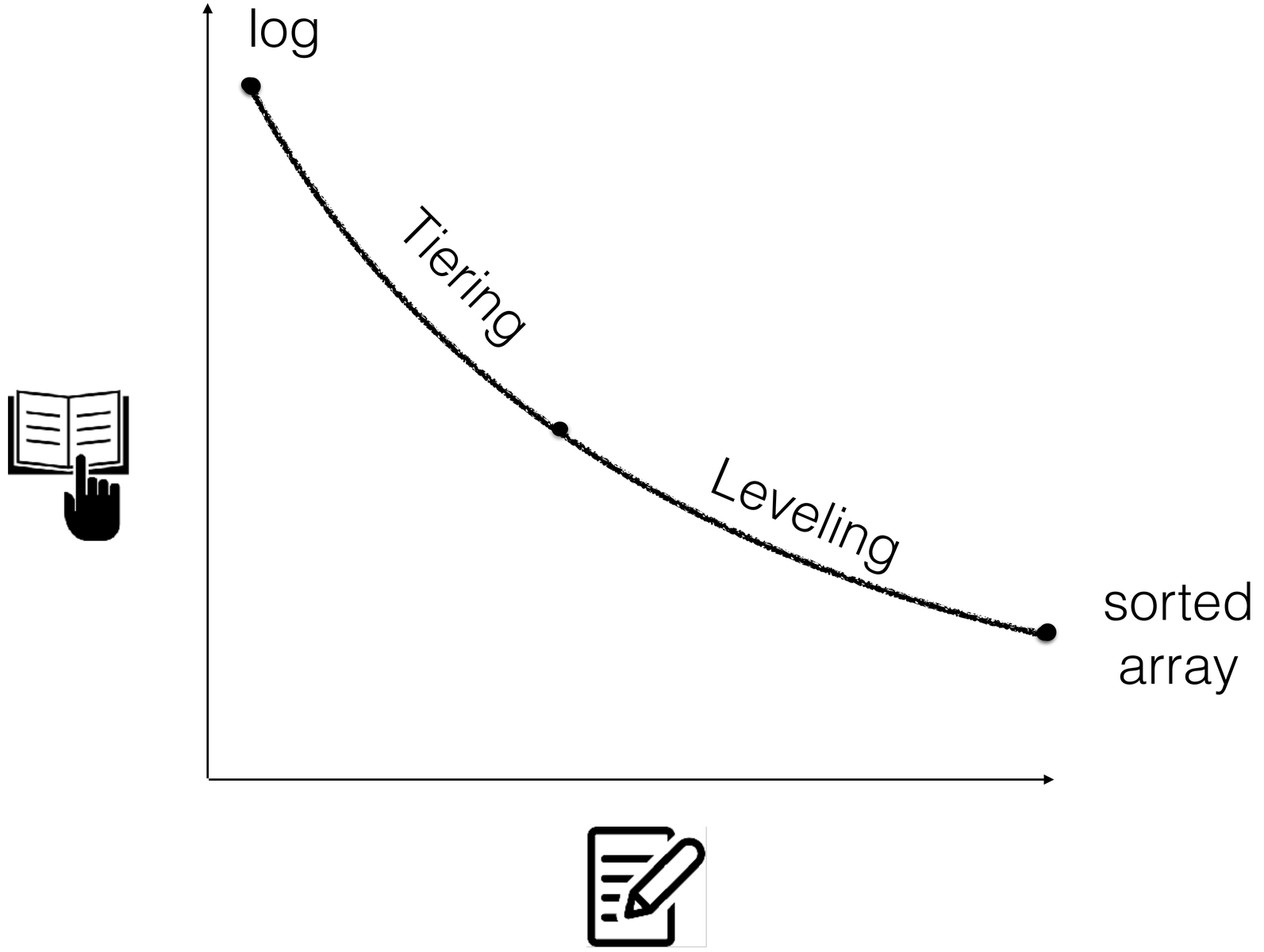
O(*N/P*) runs per level

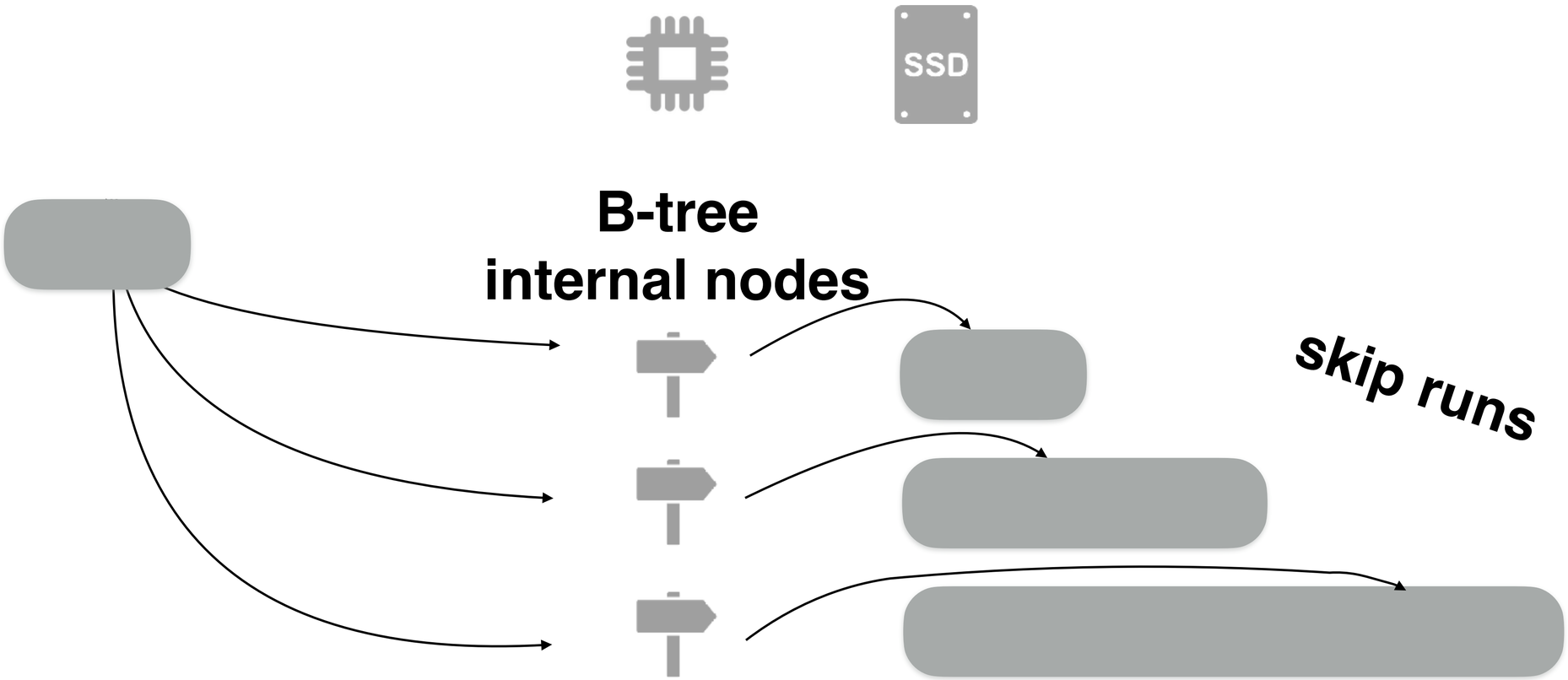
log

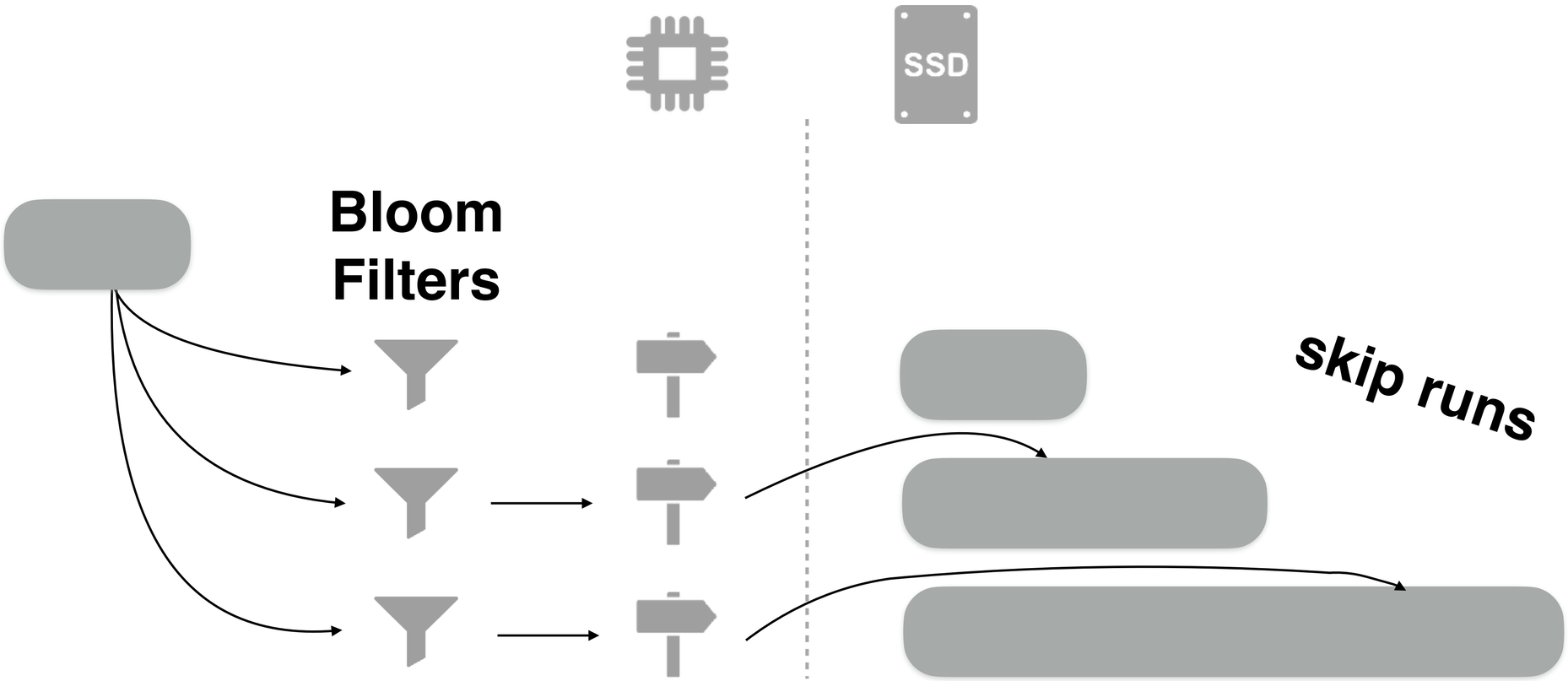
1 run per level

sorted array

size ratio *T* = *N/P*


log
Tiering
Leveling
sorted
array


SSD
B-tree
internal nodes
skip runs


SSD
Bloom
Filters
skip runs

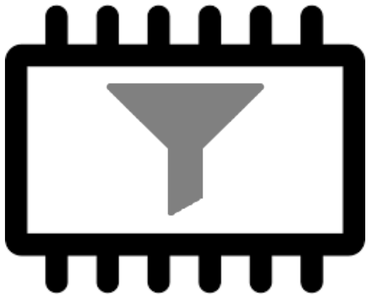
Why use a Filter?

Does key X exist

Data

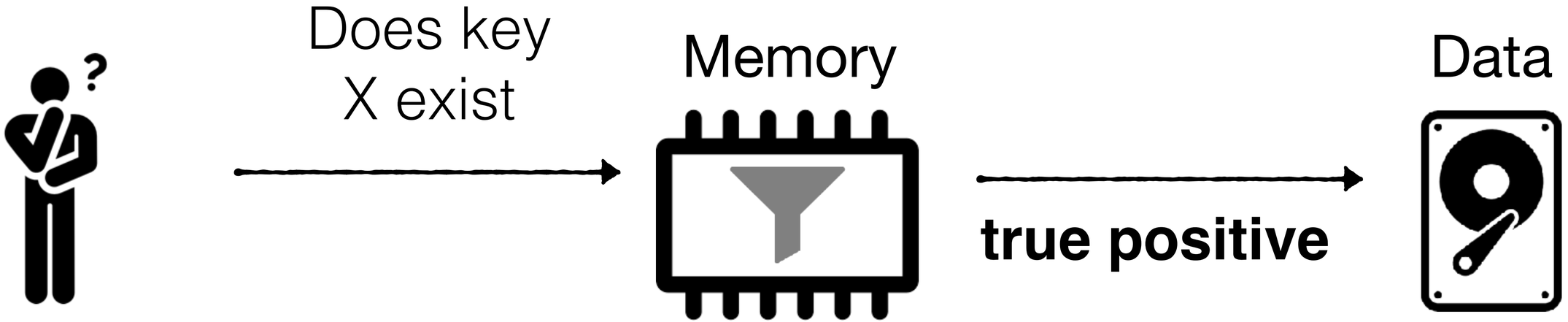
Does key X exist

Memory

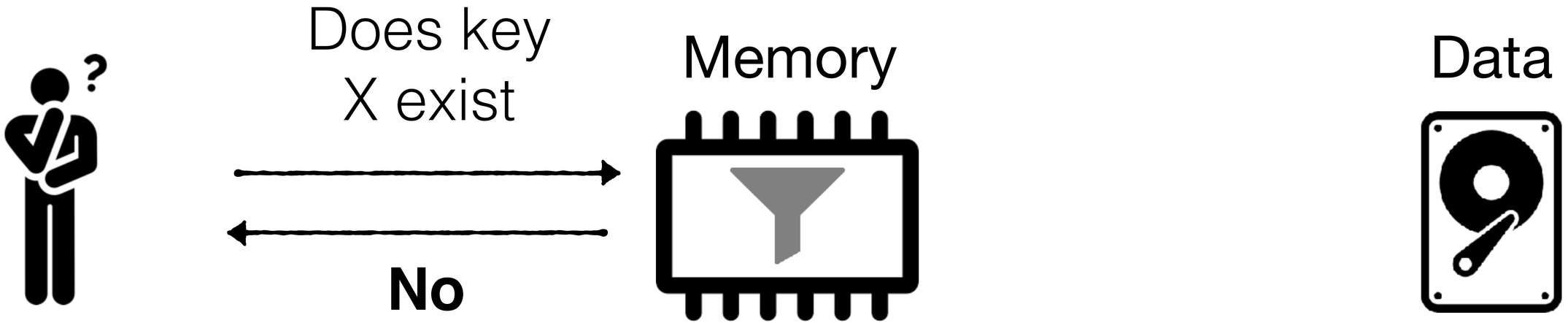


Data

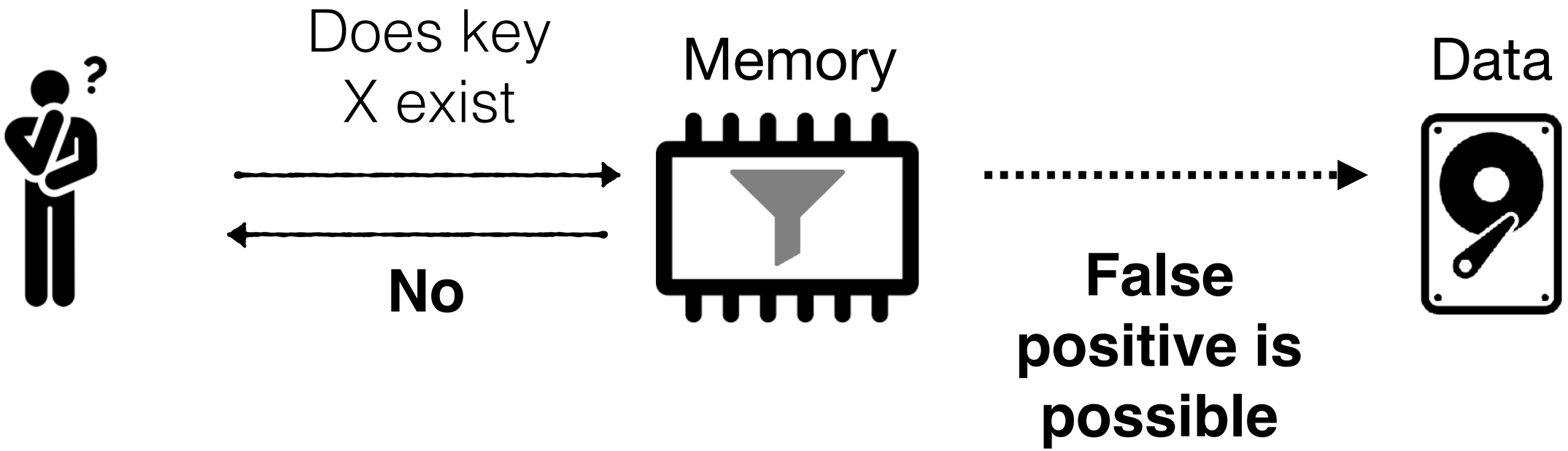
If key X exists



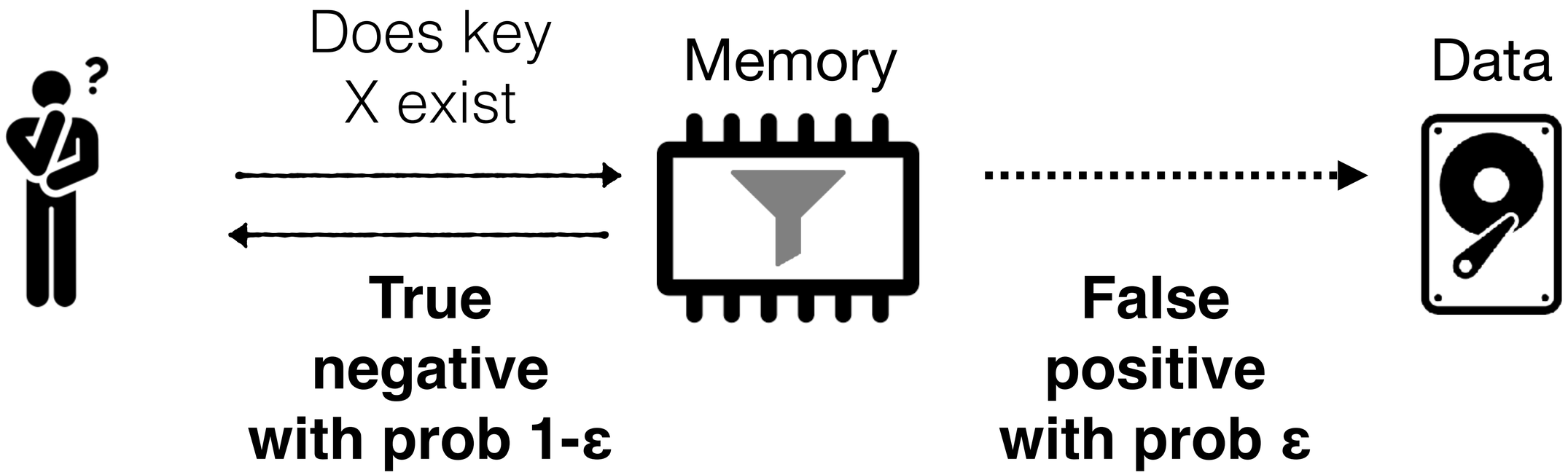
If key X does not exist



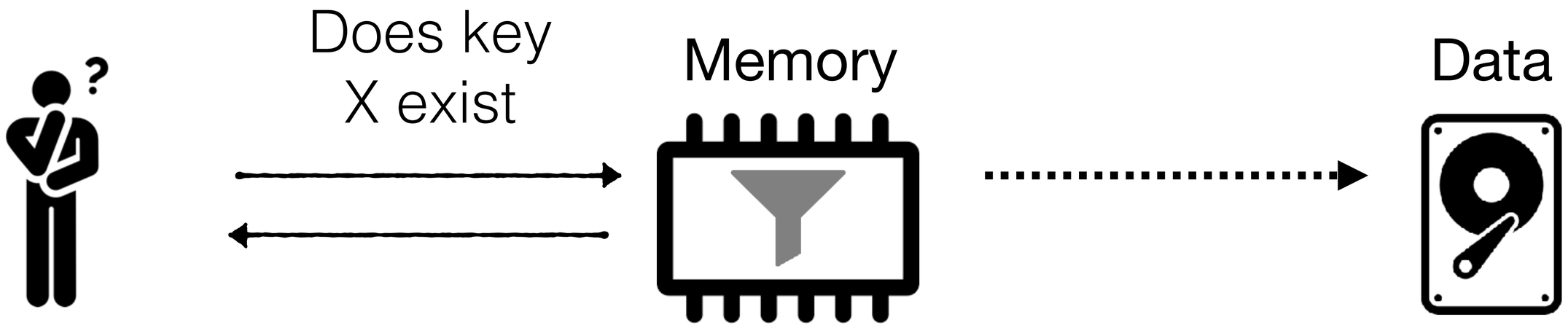
If key X does not exist



If key X does not exist



If key X does not exist



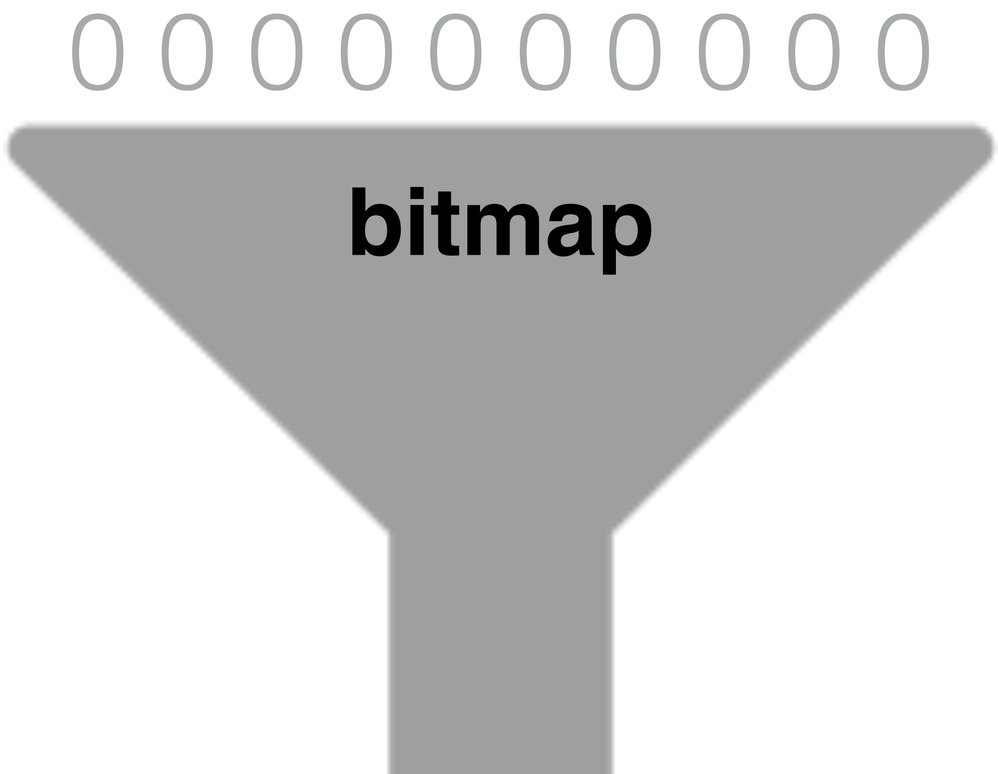
ε - false positive rate - FPR

Bloom Filters

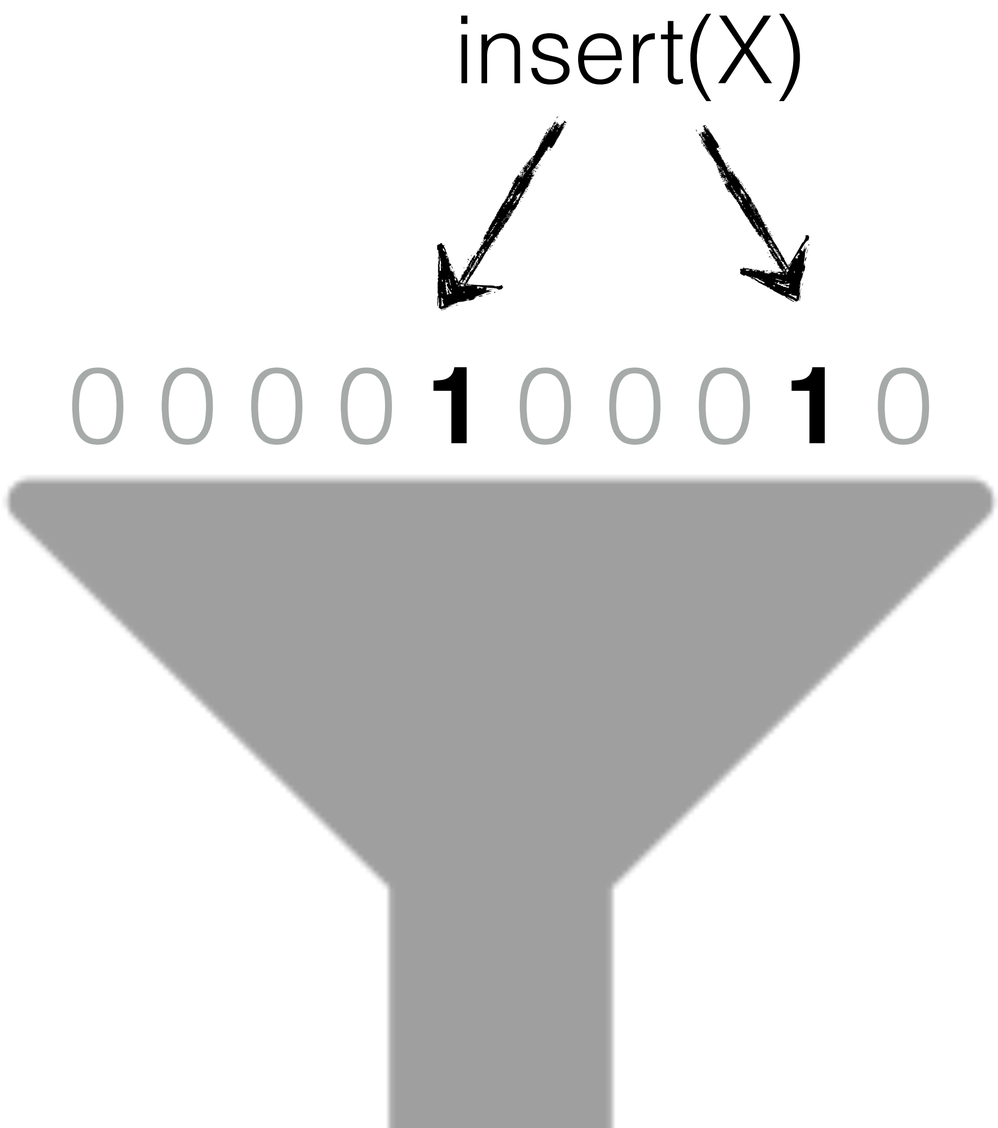
Bloom Filters

Space/time Trade-Offs in Hash Coding with Allowable Errors
Burton Howard Bloom. Communications of the ACM, 1970.

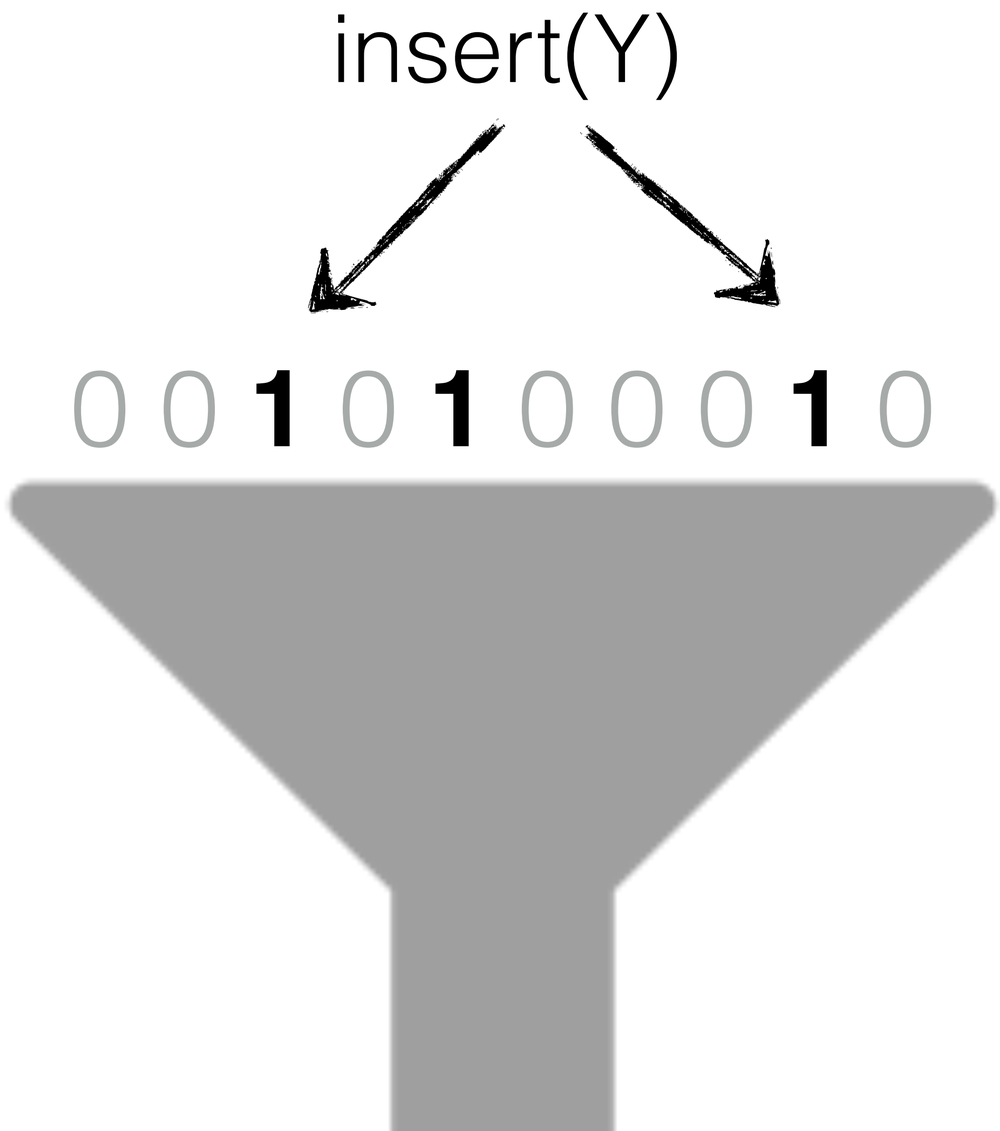
k hash functions



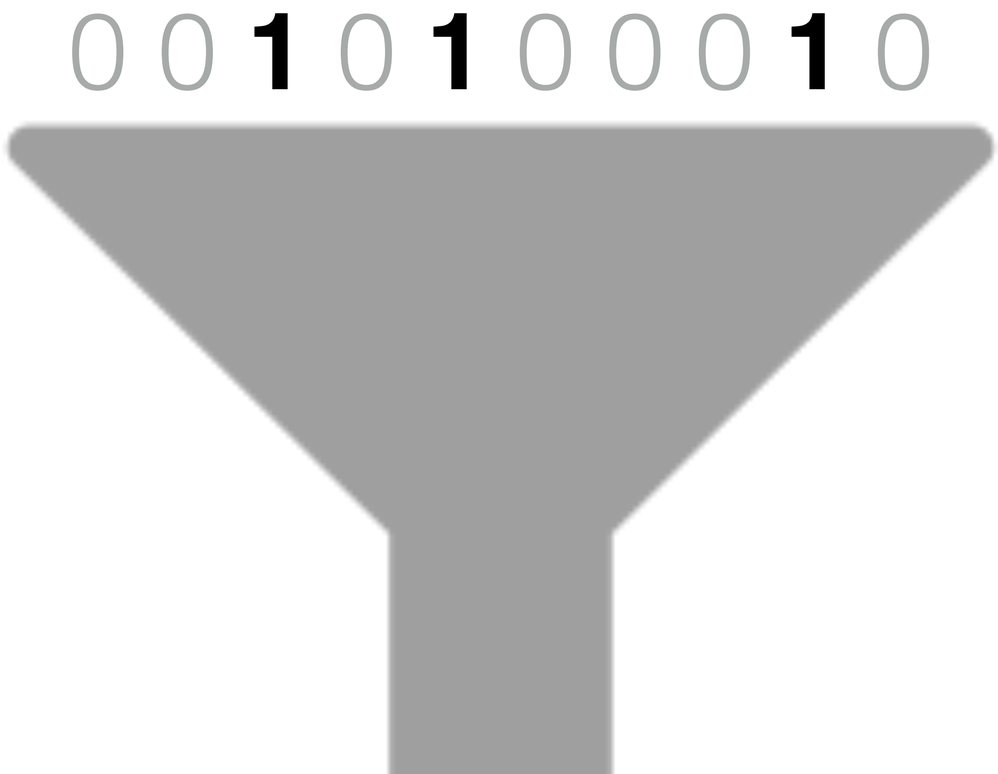
insert: Set from 0 to 1 or keep 1



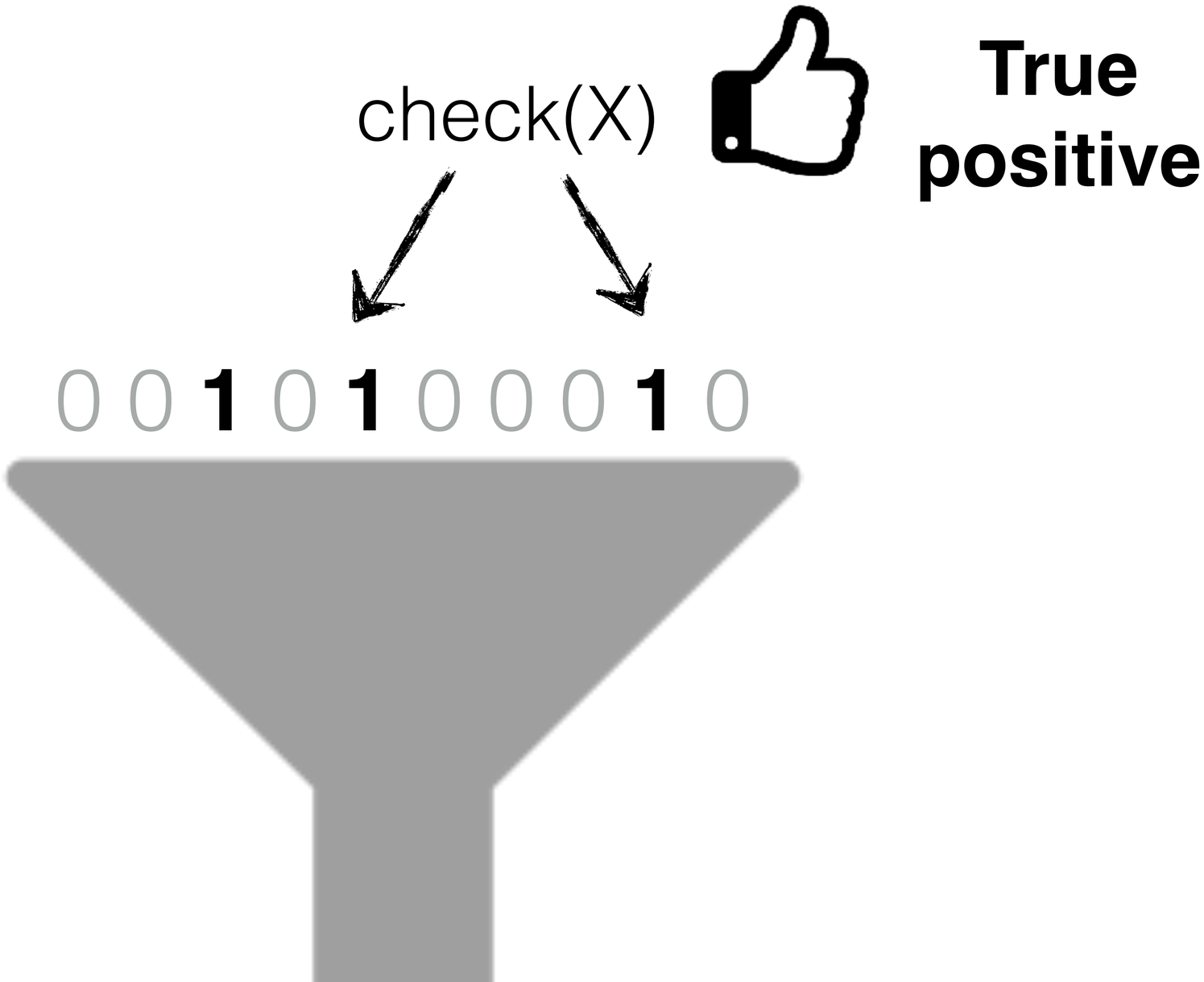
insert: Set from 0 to 1 or keep 1



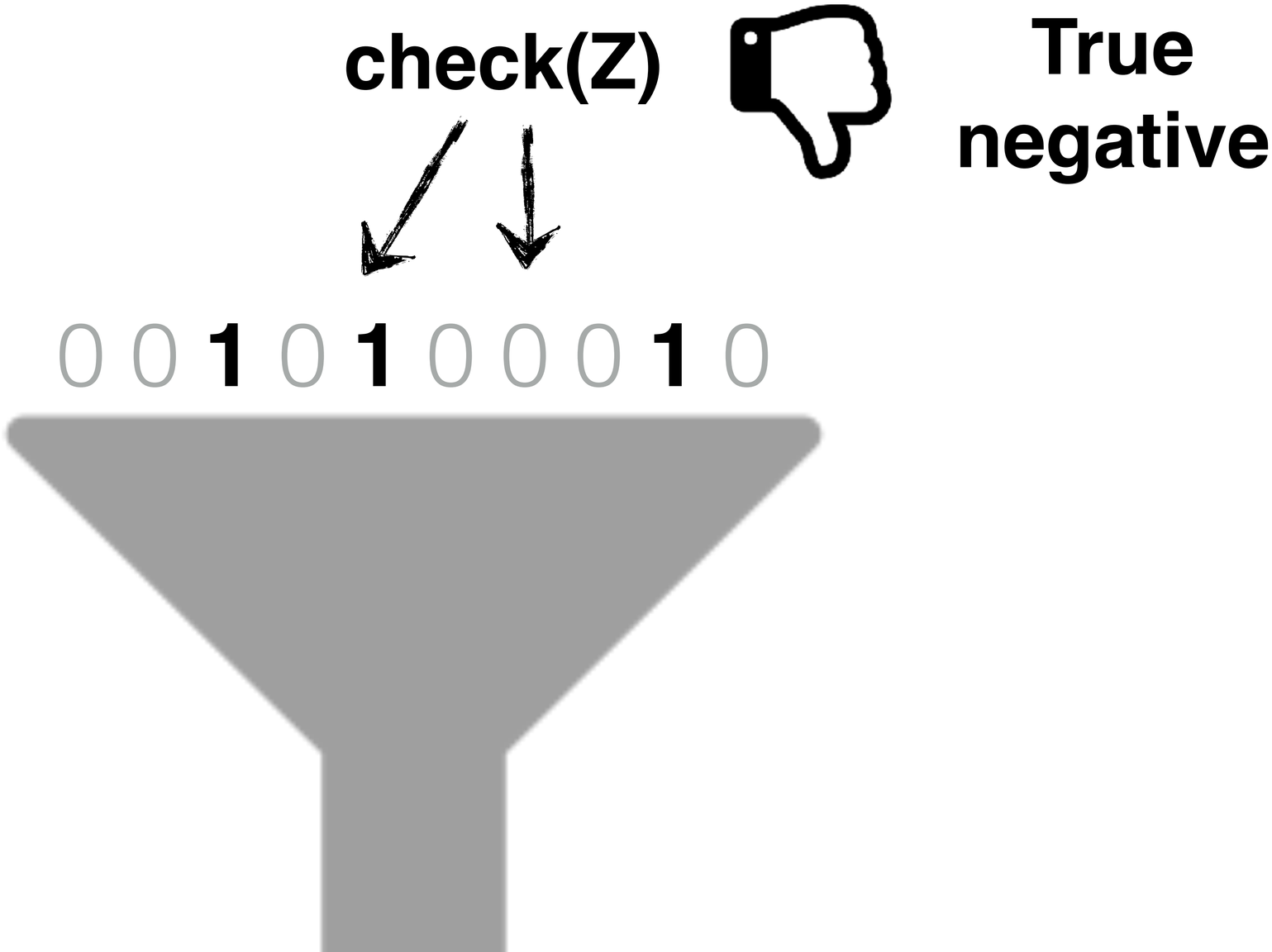
Queries: return positive if all hashed bits are 1s



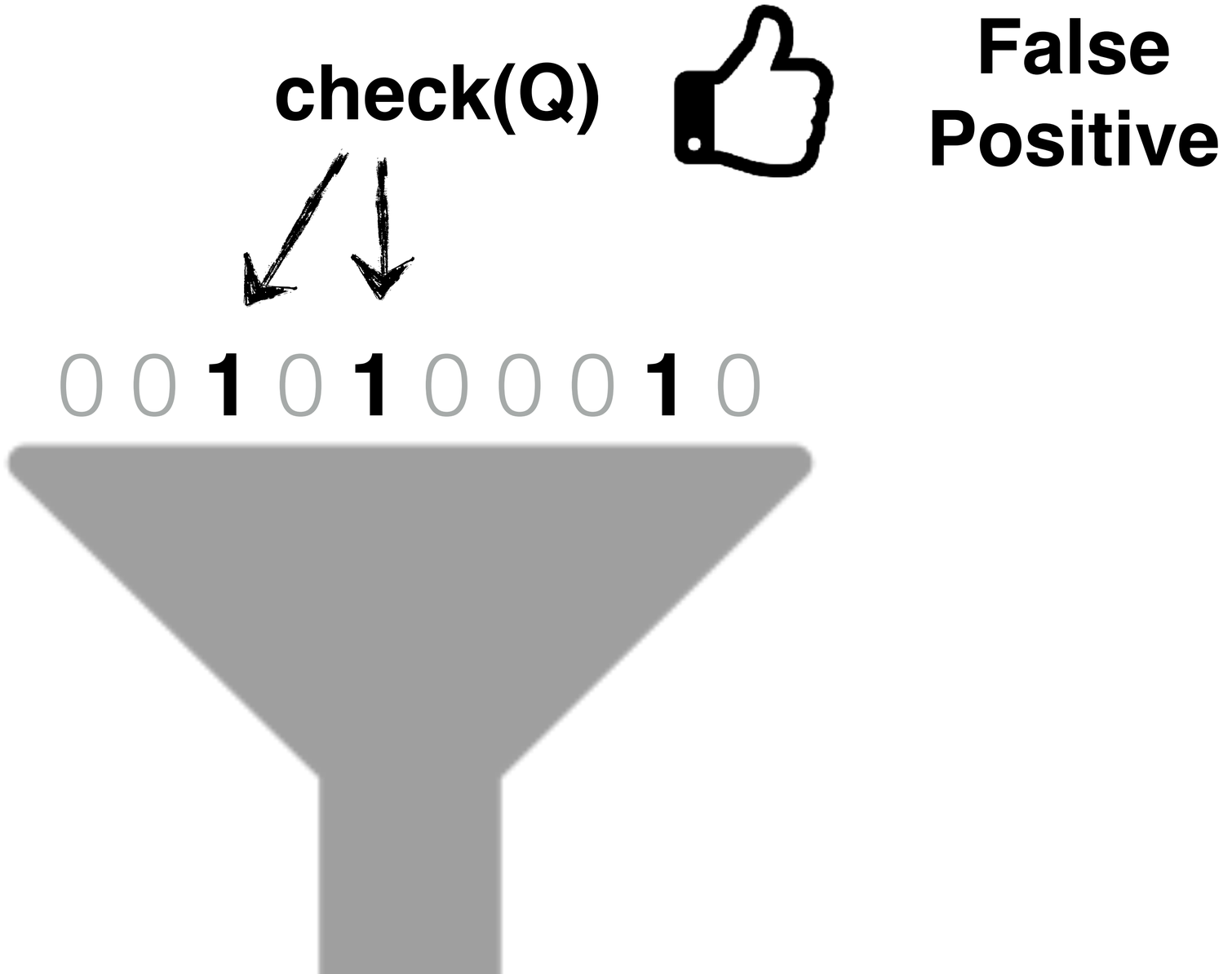
Queries: return positive if all hashed bits are 1s

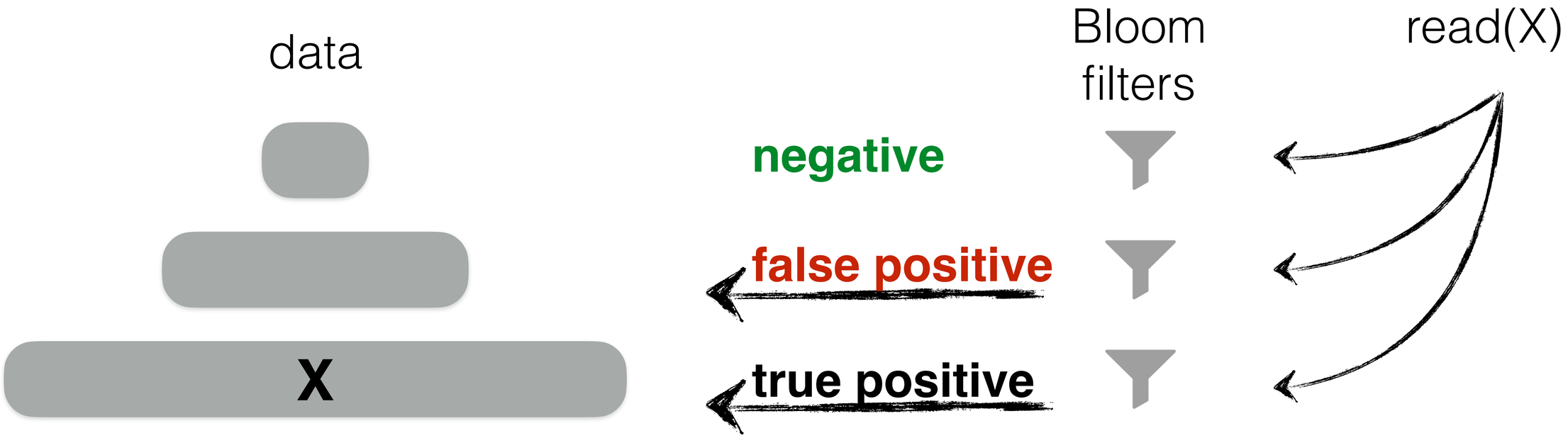


Queries: return positive if all hashed bits are 1s

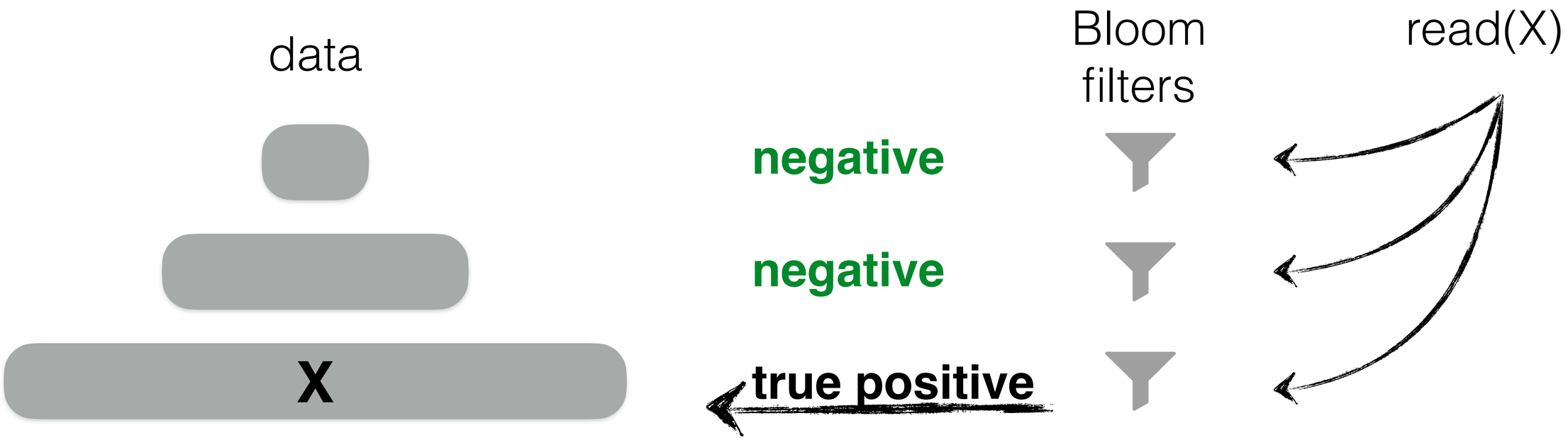


Queries: return positive if all hashed bits are 1s

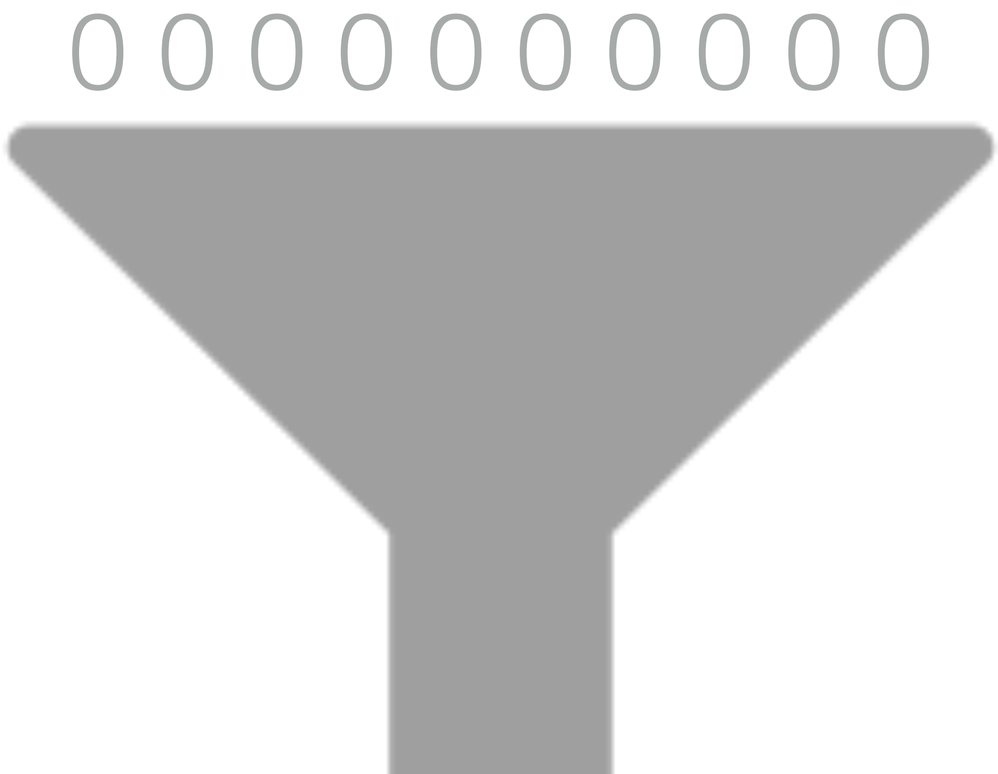



data
X
Bloom
filters
read(X)
negative
false positive
true positive

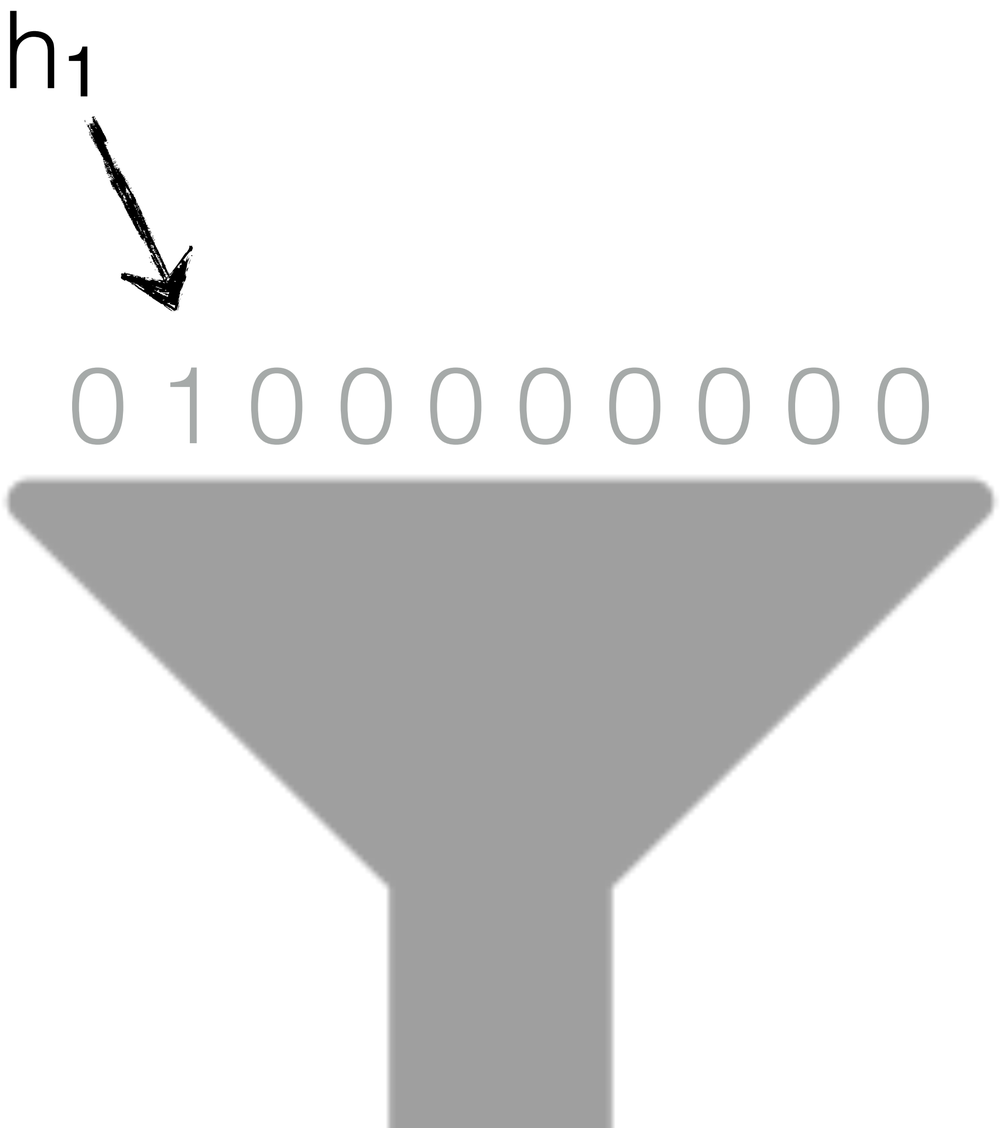
more memory ⟶ fewer false positives



How many hash functions should we use?

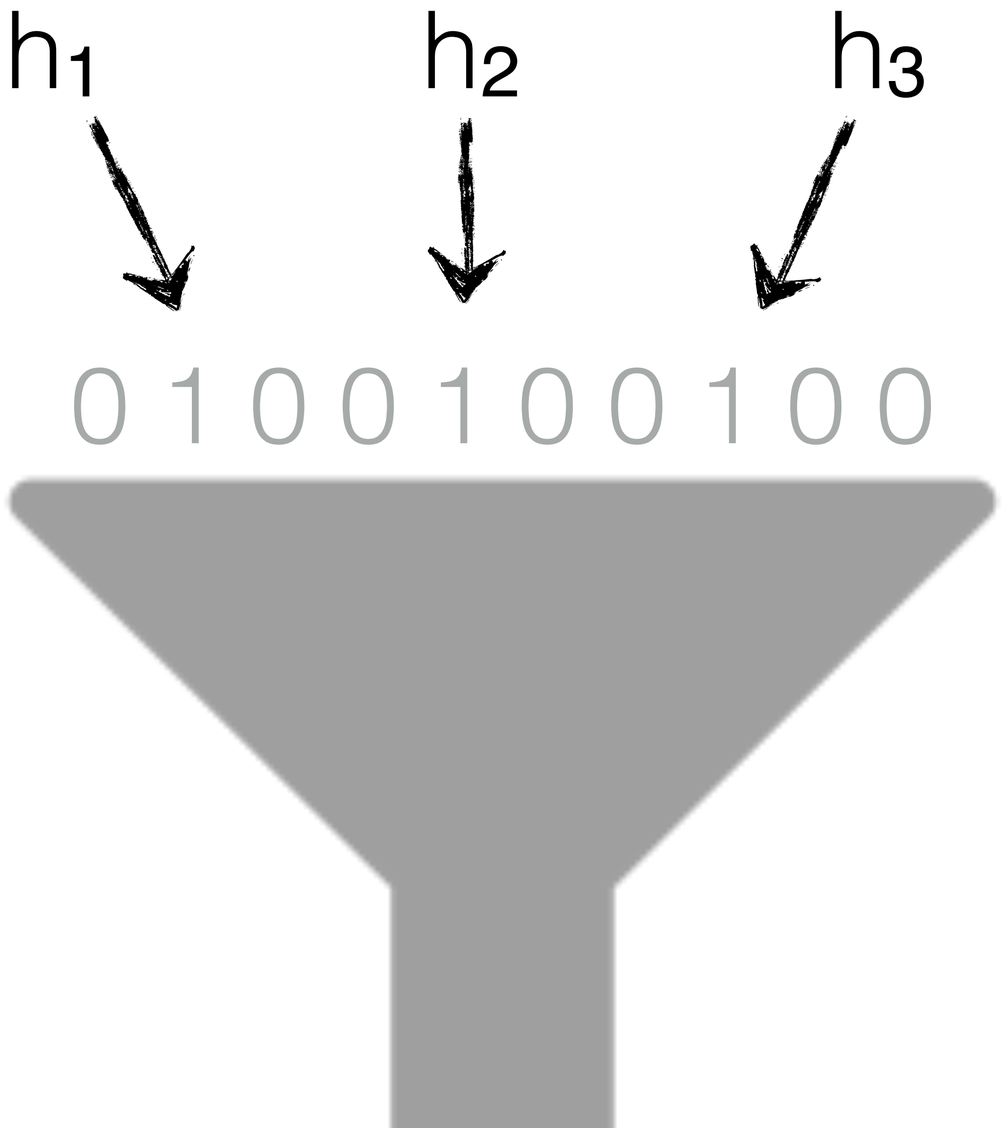


How many hash functions should we use?



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

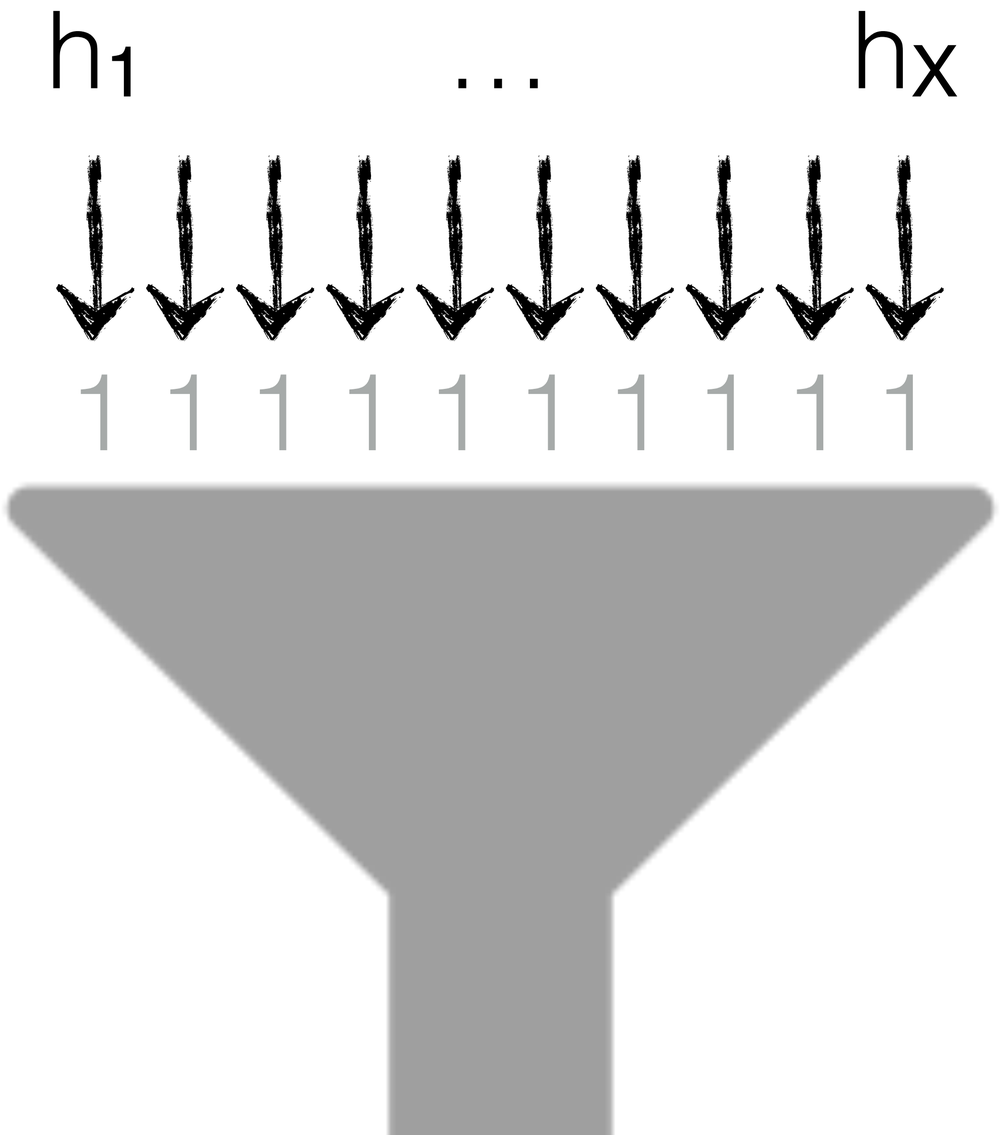
How many hash functions should we use?



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?

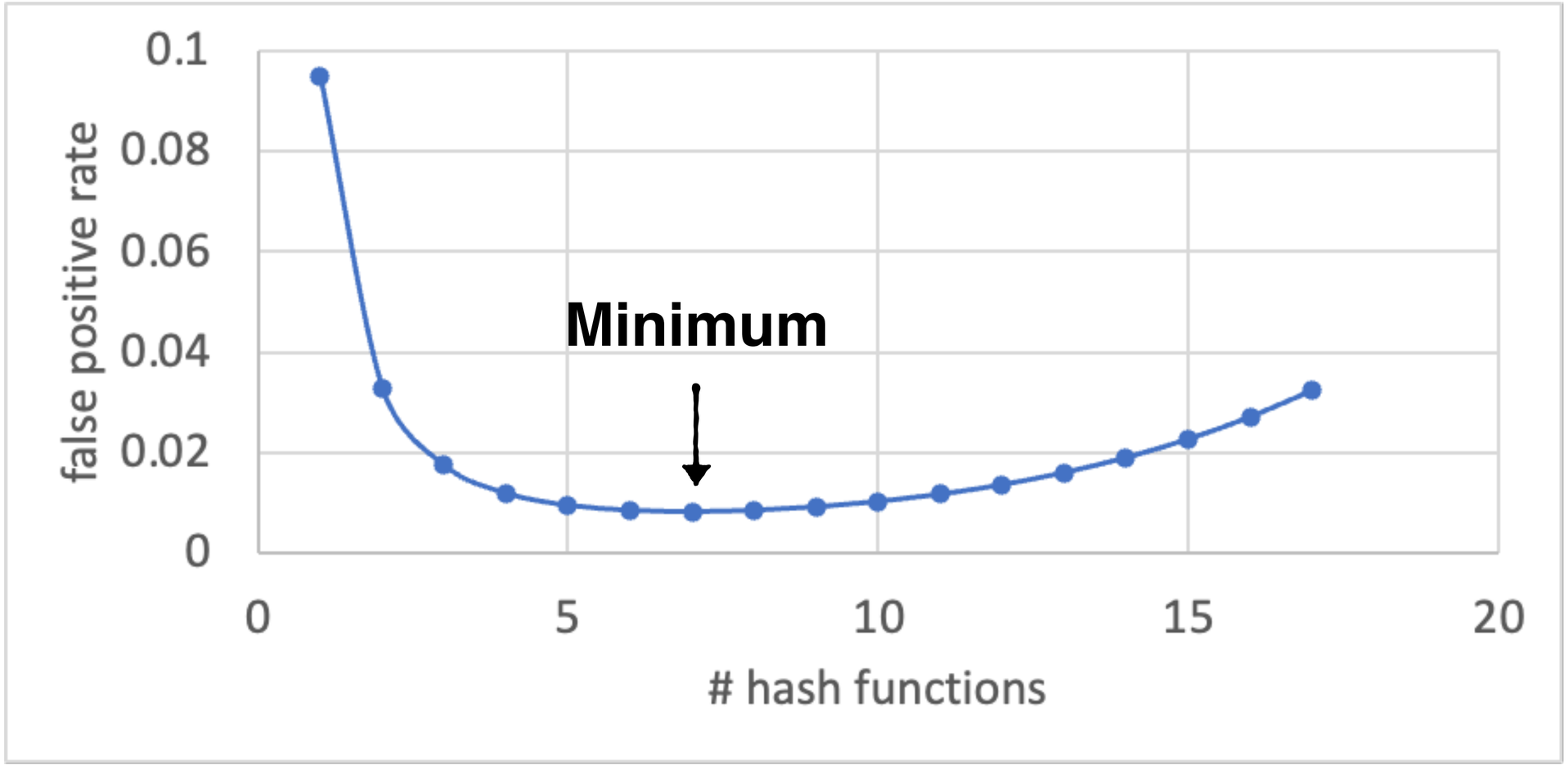


One is too few: false positive occurs whenever we hit a 1

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

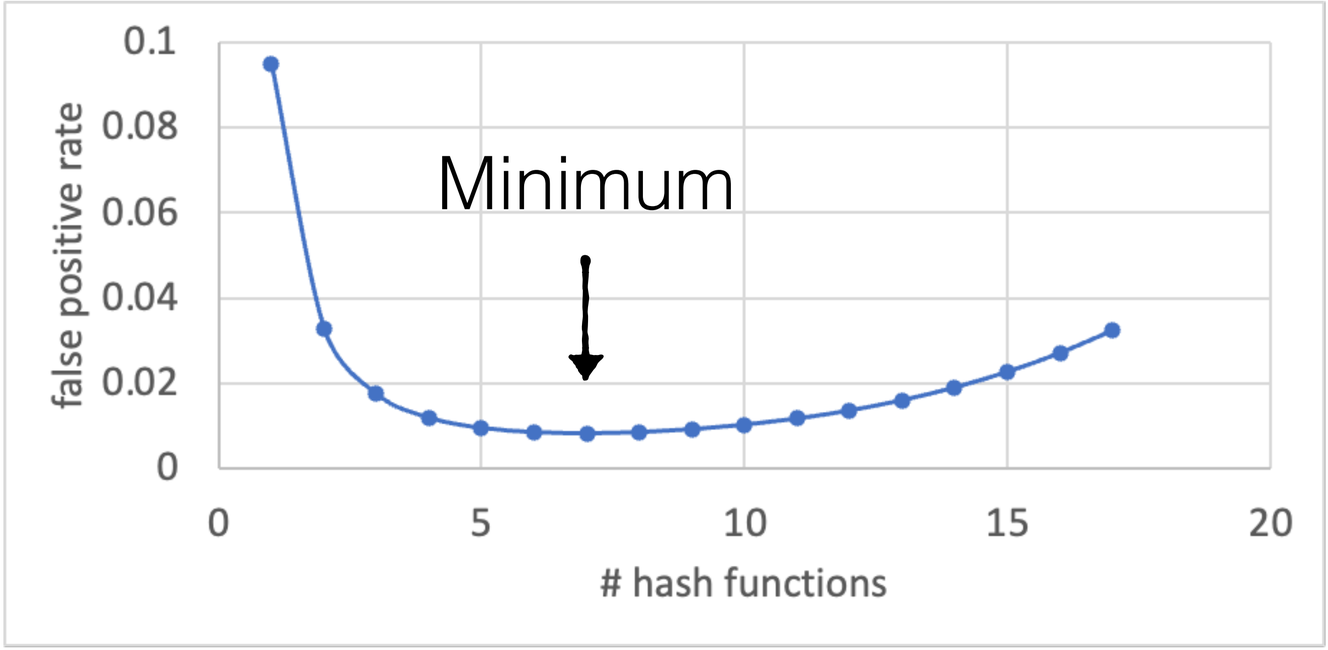
But too many hash functions wind up increasing the FPR.

How many hash functions should we use?



(Drawn for a filter using 10 bits per entry)

How many hash functions should we use?

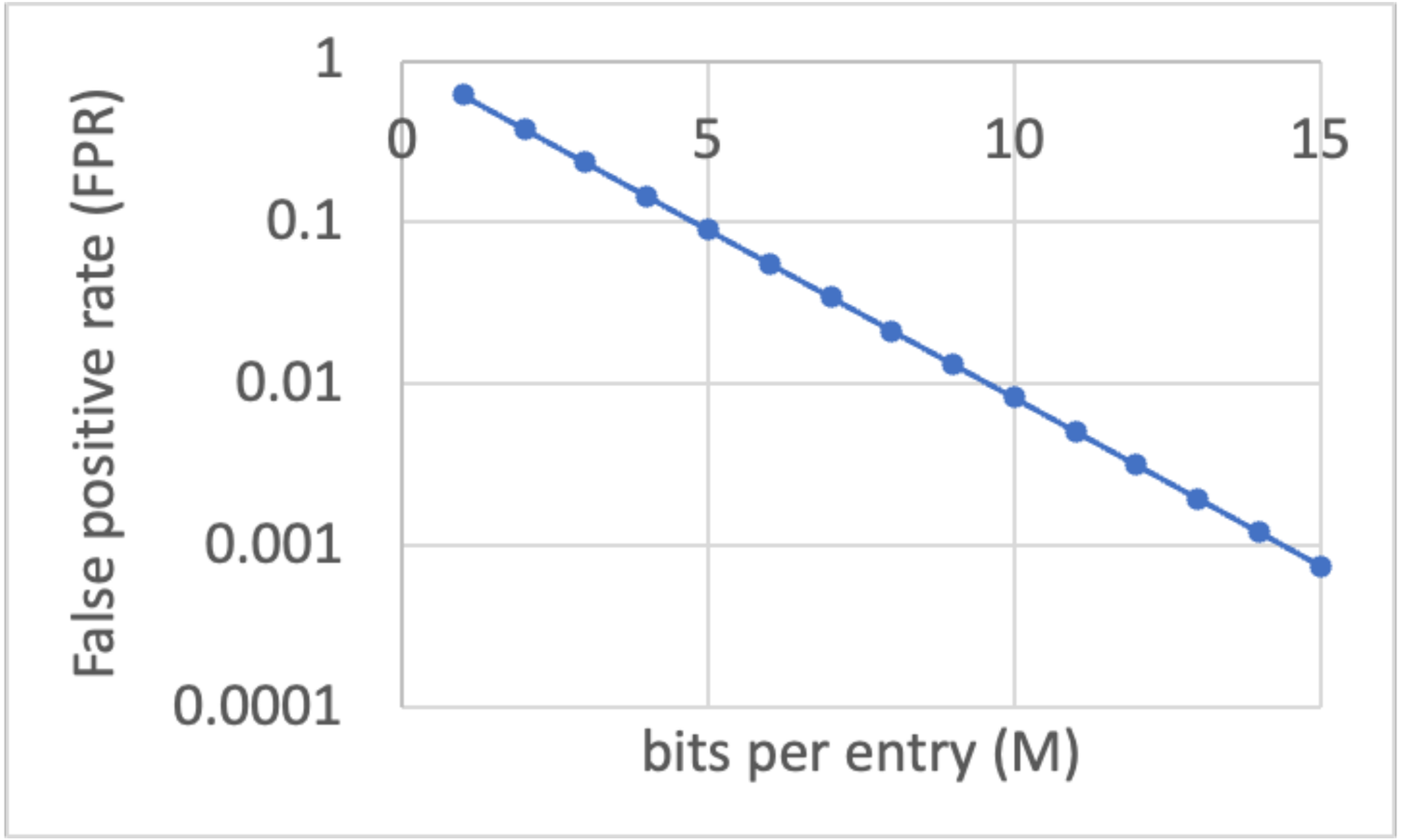


Optimal # hash functions = ln(2) · *M*

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

assuming the optimal # hash functions,

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



Operation Costs (in memory accesses)

Insertion =

Positive Query =

Avg. Negative Query =

Operation Costs (in memory accesses)

Insertion = $M \cdot \ln(2)$ (# hash functions)

Positive Query =

Avg. Negative Query =

Operation Costs (in memory accesses)

Insertion = $M \cdot \ln(2)$

Positive Query = $M \cdot \ln(2)$ (# hash functions)

Avg. Negative Query =

Operation Costs (in memory accesses)

Insertion = $M \cdot \ln(2)$

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

Avg. Negative Query =

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

Operation Costs (in memory accesses)

Insertion = $M \cdot \ln(2)$

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

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

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

Operation Costs (in memory accesses)

Insertion = $M \cdot \ln(2)$

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

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

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

Operation Costs (in memory accesses)

Insertion = $M \cdot \ln(2)$

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

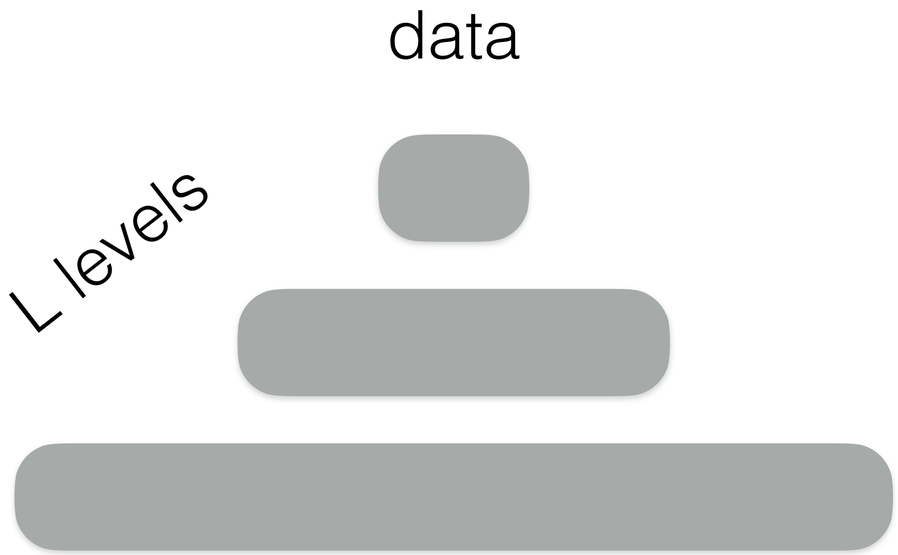
Avg. Negative Query = 2

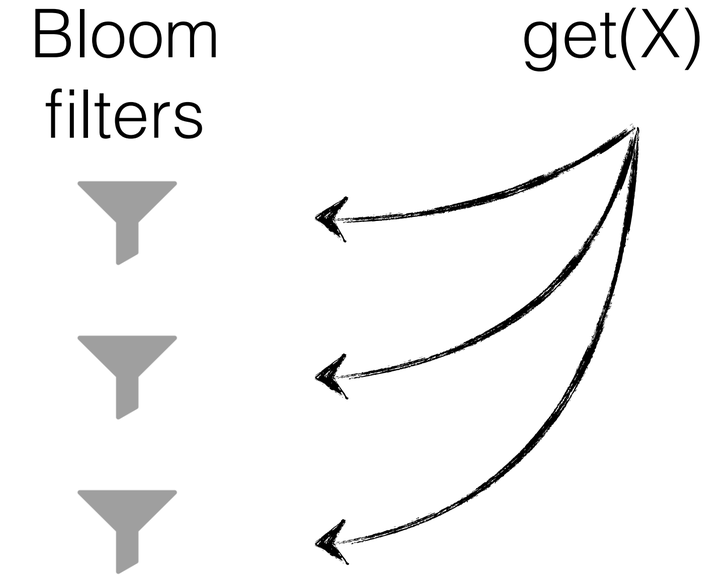
false positive rate = $2^{-M \cdot \ln(2)}$

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

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

Avg. Negative Query = 2





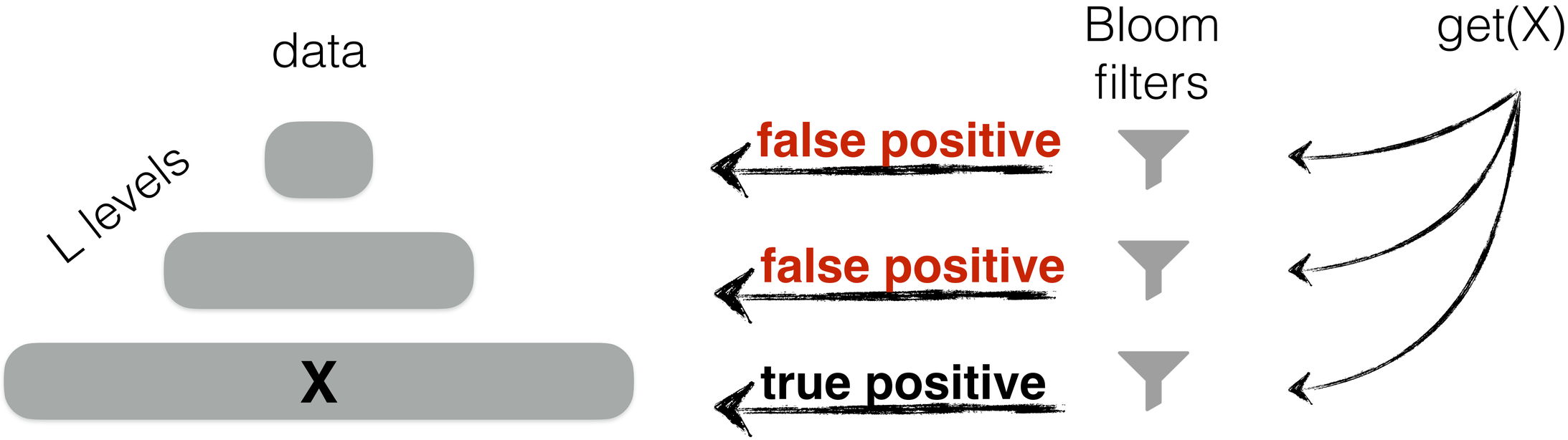
Worst-case:

Avg. worst-case:

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

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

Avg. Negative Query = 2



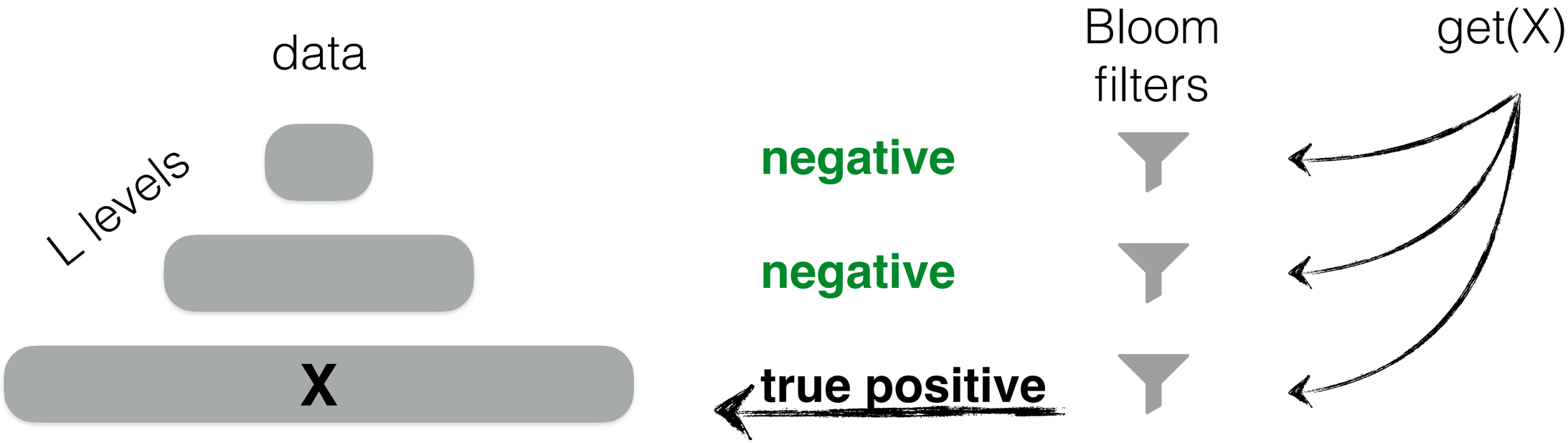
Worst-case: O(M·L)

Avg. worst-case:

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

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

Avg. Negative Query = 2



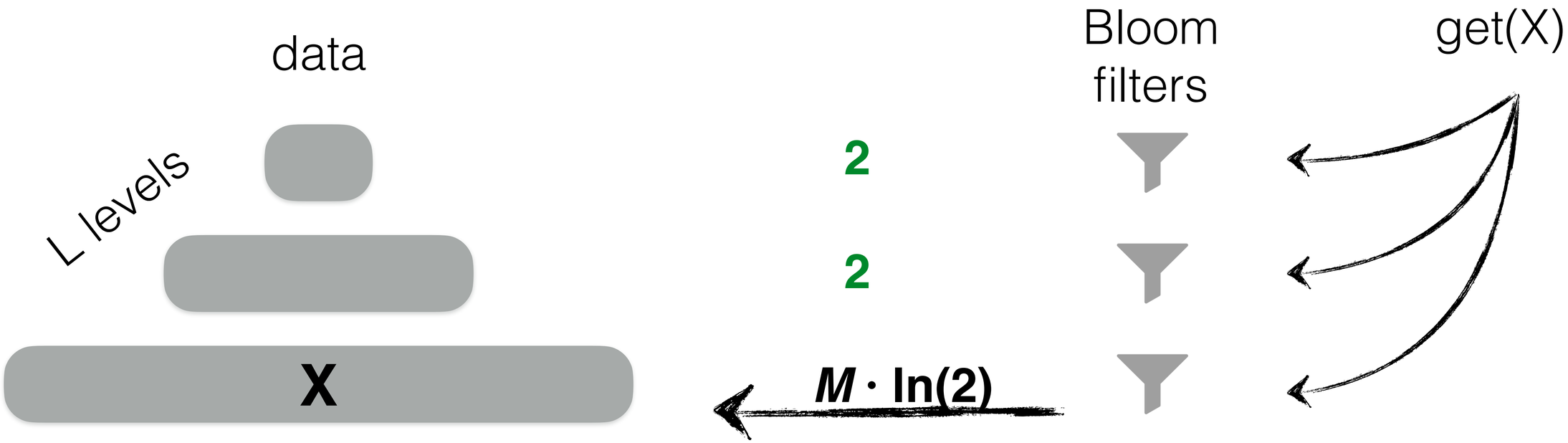
Worst-case: O(M·L)

Avg. worst-case:

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

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

Avg. Negative Query = 2



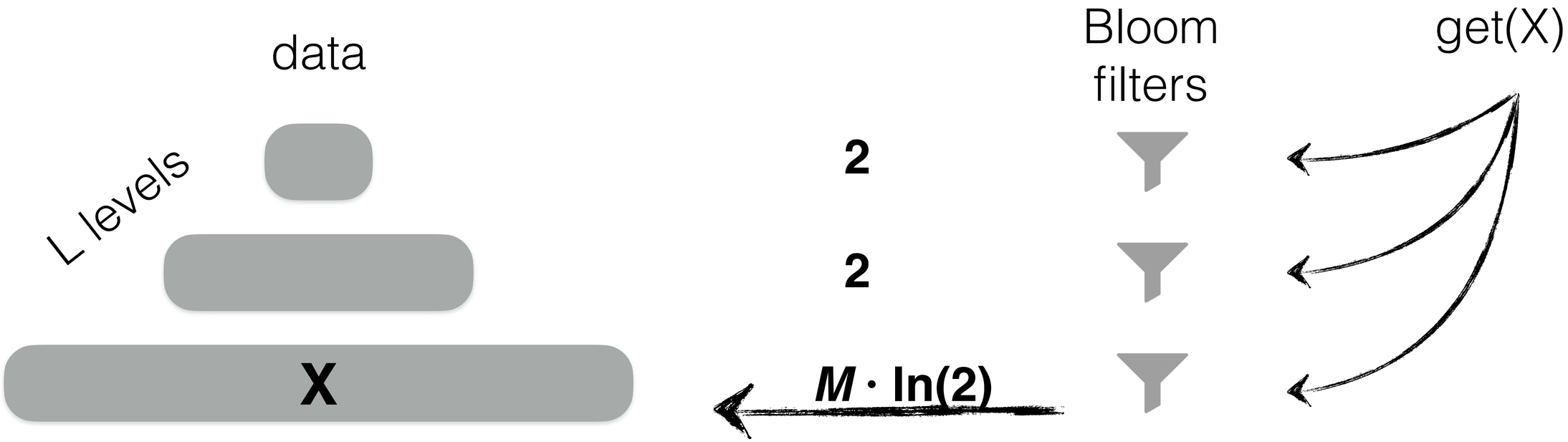
Worst-case: O(M·L)

Avg. worst-case:

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

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

Avg. Negative Query = 2



Worst-case: O(M·L)

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

Construction contract

Construction contract

Know specs in advance:

N - # entries to insert

ε - desired FPR

Construction contract

Know specs in advance:

N - # entries to insert

ε - desired FPR

Allocate filter with: $N \cdot \ln(2) \cdot \log_2(1/\varepsilon)$ bits

Construction contract

Know specs in advance:

N - # entries to insert

ε - desired FPR

Allocate filter with: $N \cdot \ln(2) \cdot \log_2(1/\varepsilon)$ bits

Insert N elements using $-\ln(\varepsilon)/\ln(2)$ hash functions

Construction contract

Know specs in advance:

N - # entries to insert

ε - desired FPR

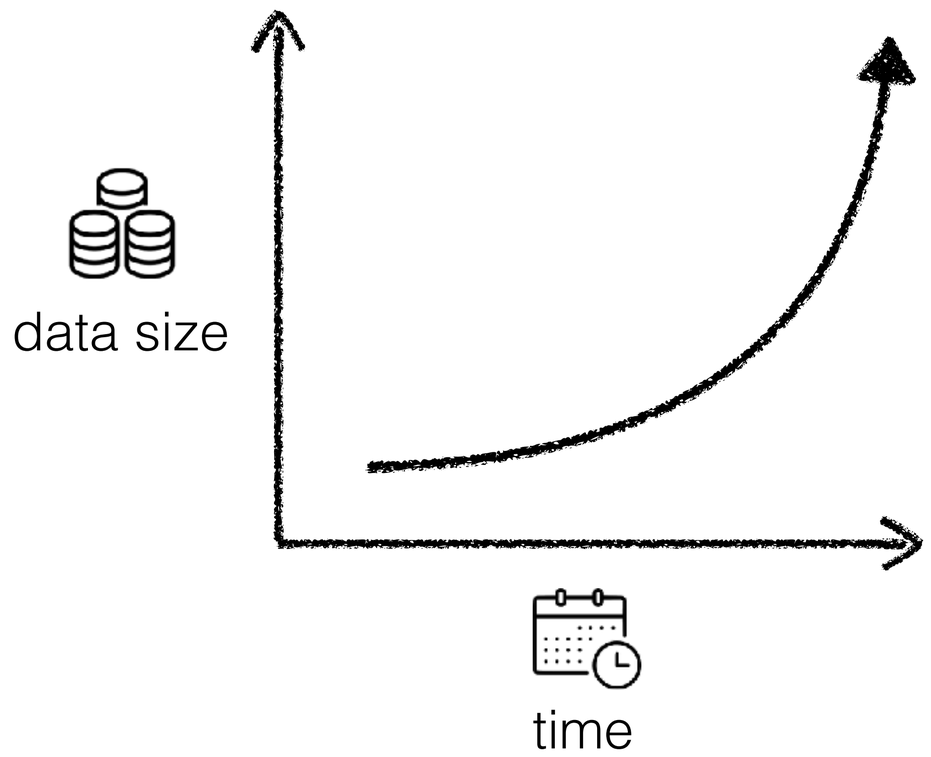
Allocate filter with: $N \cdot \ln(2) \cdot \log_2(1/\varepsilon)$ bits

Insert N elements using $-\ln(\varepsilon)/\ln(2)$ hash functions

Guarantee FPR of ε

Research Question

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 do better?

get I/O cost

$O(2^{-M} \cdot L)$

$O(?)$

insert I/O cost

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

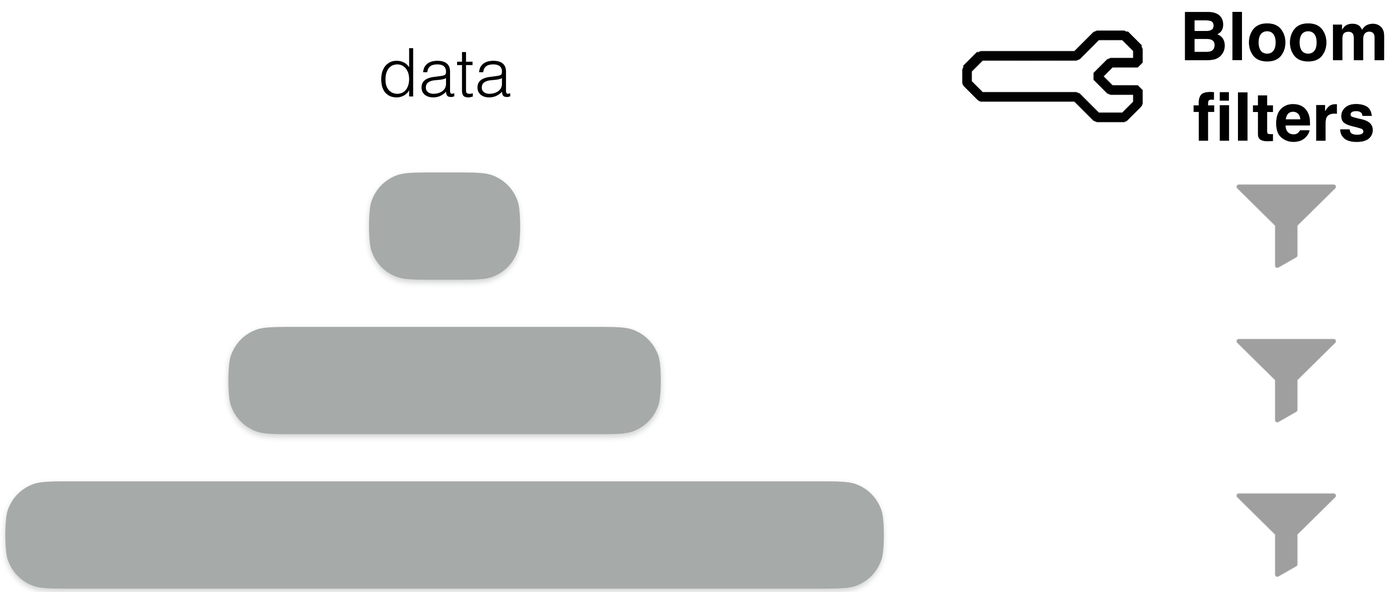
$O(\ ?\ )$

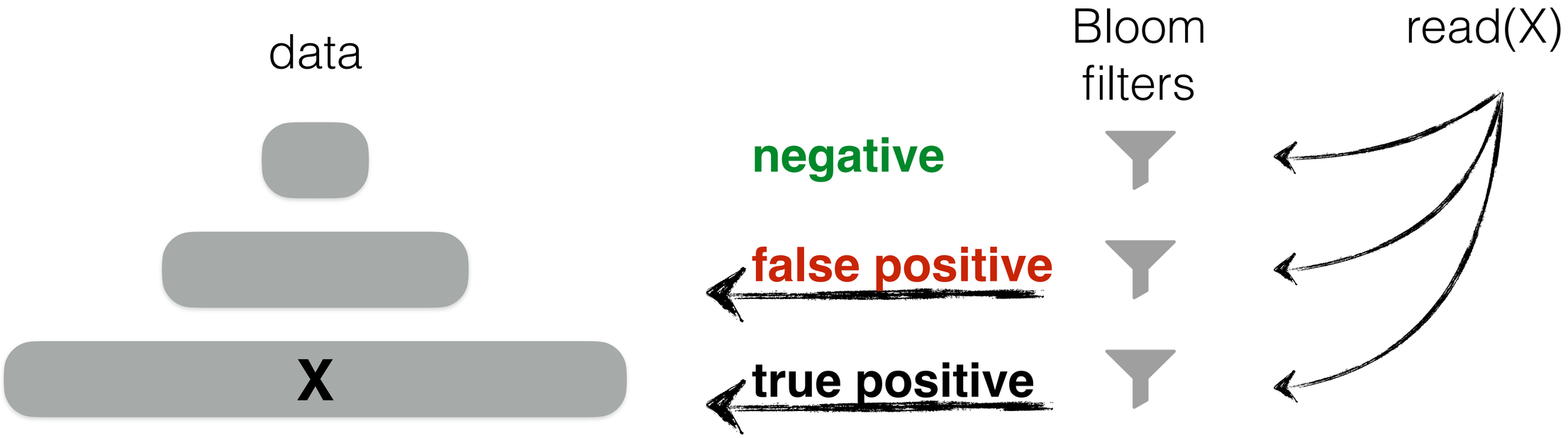
$L = \log_T N/P$

(Costs assuming leveling)

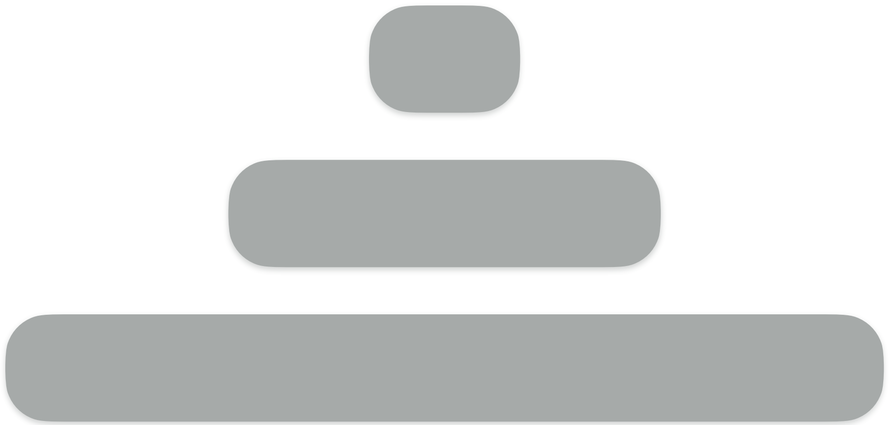
Monkey: **O**ptimal **N**avigable **Key**-Value Store SIGMOD17

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




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

data



Bloom filters

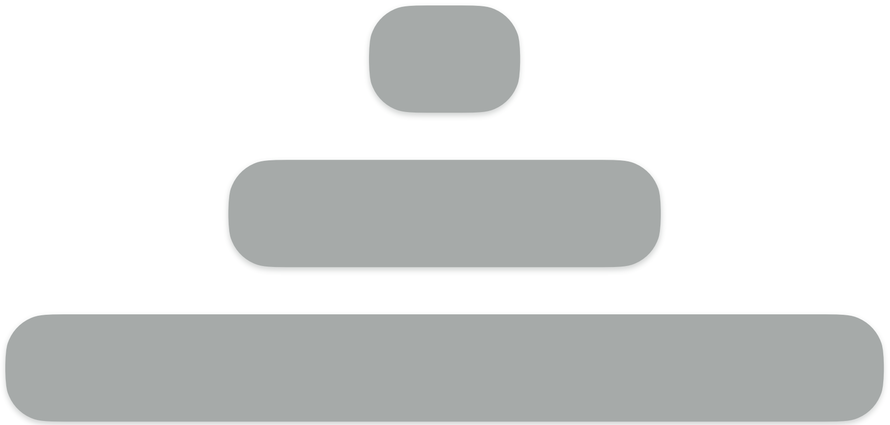
bits/entry

M

M

M

data



Bloom filters

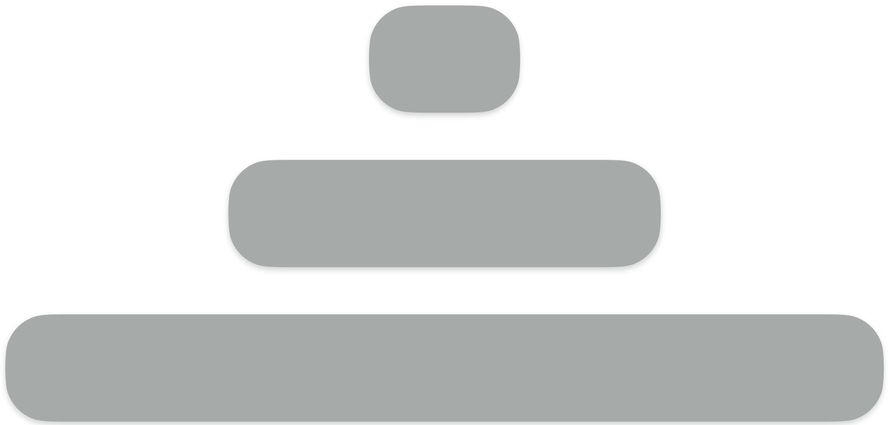
bits/entry

M

M

M

data



Bloom filters

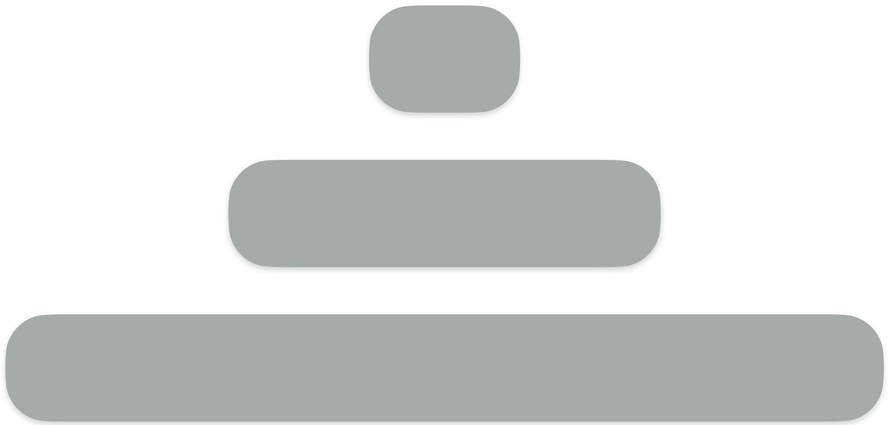
false positive rate

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

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

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

data



Bloom filters

false positive rate

2^{-M}

2^{-M}

2^{-M}

Bloom filters

false positive rate

2^{-M}

2^{-M}

2^{-M}

$= O(2^{-M} \cdot \log_T N/P)$

Bloom filters

false positive rate

2^{-M}

2^{-M}

2^{-M}

$= O(2^{-M} \cdot \log_T N/P)$

Bloom filters

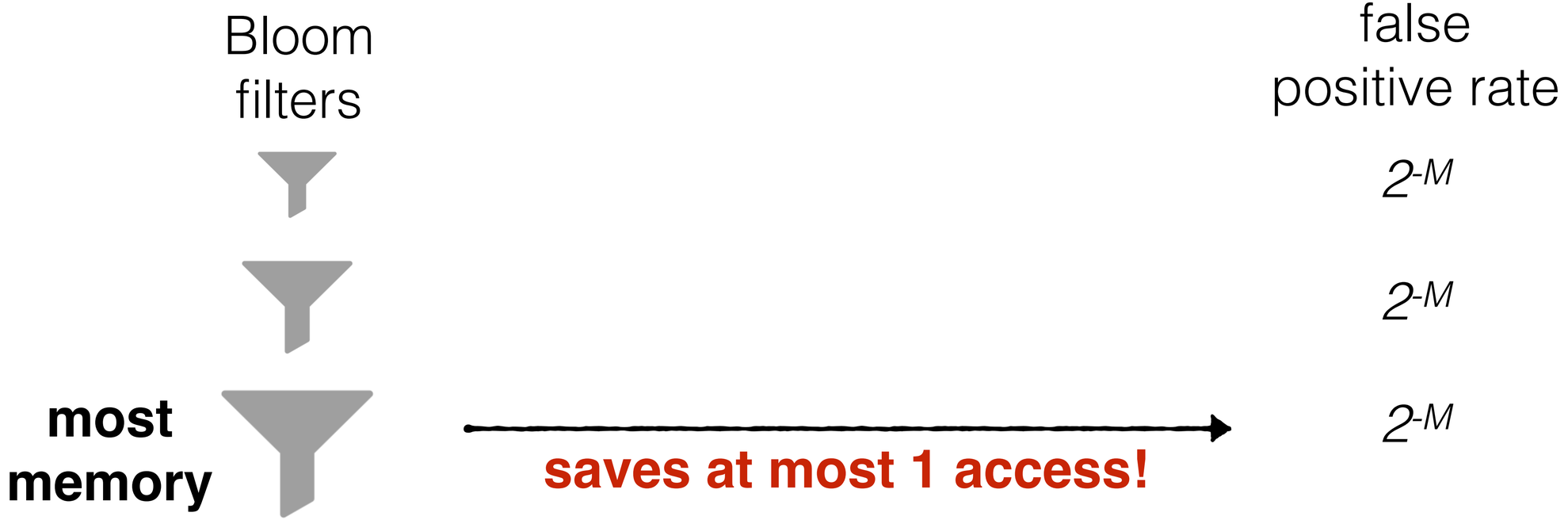
false positive rate

2^{-M}

2^{-M}

most memory

2^{-M}


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

bits / entry

M ***+ 2***

M ***+ 1***

M ***- 1***

reallocate

false
positive rates

$2^{-(M+2)}$ ↓

$2^{-(M+1)}$ ↓

$2^{-(M-1)}$ ↑

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

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

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

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

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

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

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

geometric progression

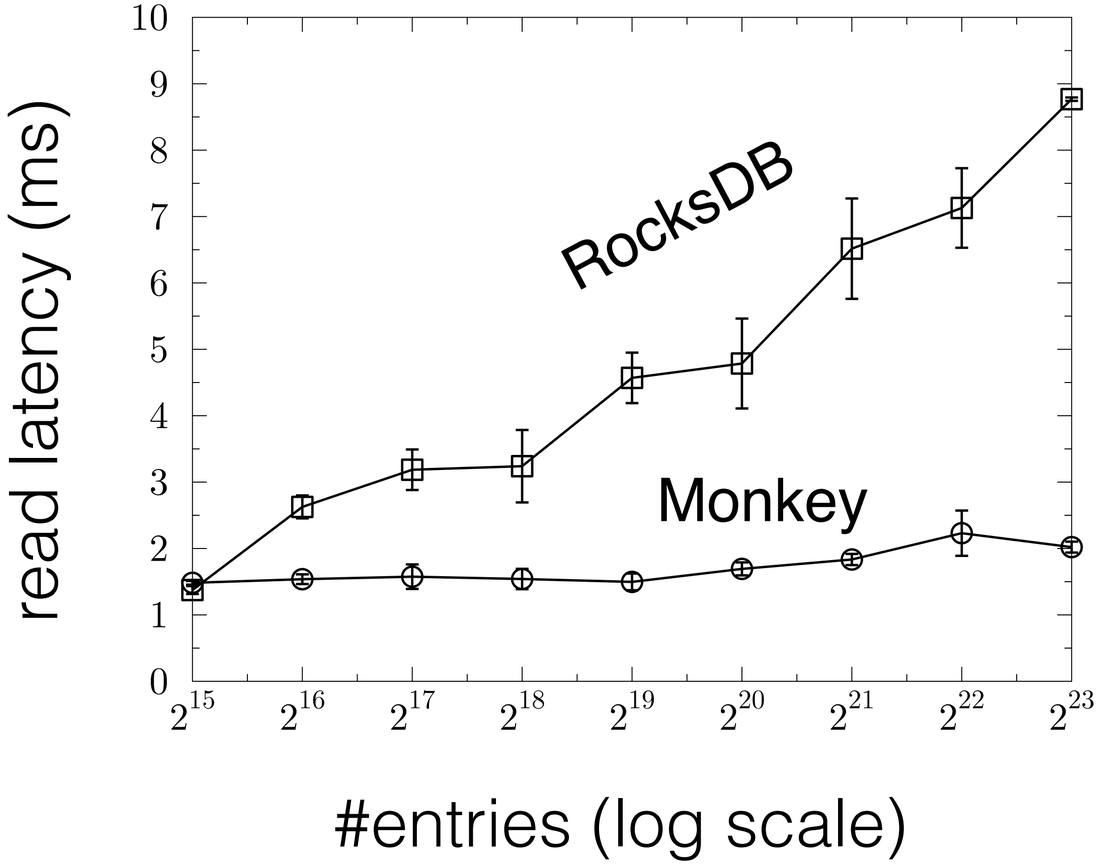
$= \quad O(2^{-M})$

Faster worst case

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

Configuration

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



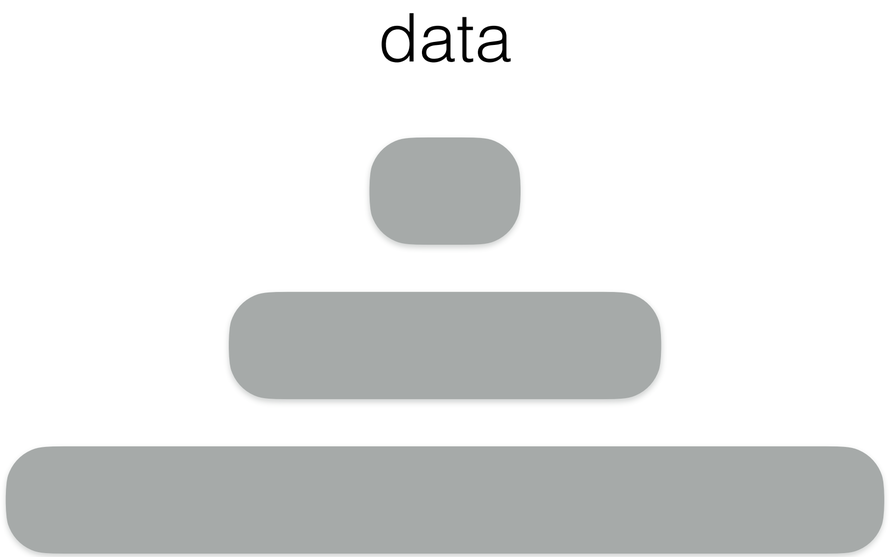
$O(2^{-M} \log_T N/P)$

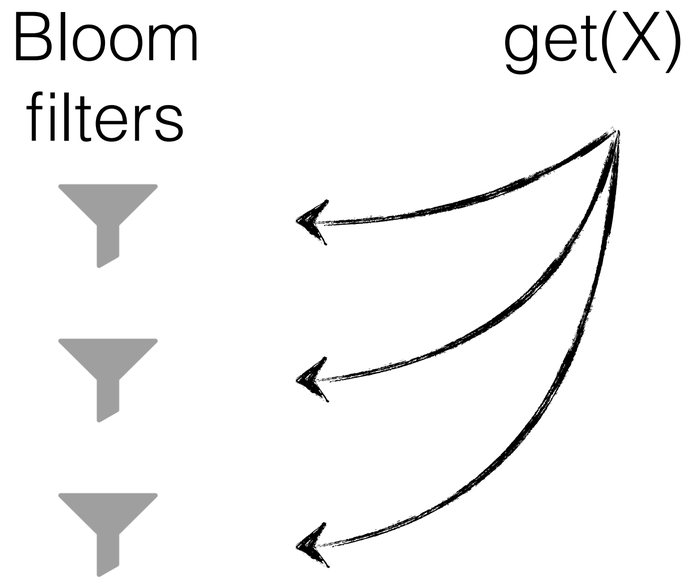
$O(2^{-M})$

Analyze filter access with Monkey

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

Avg. Negative Query = 2





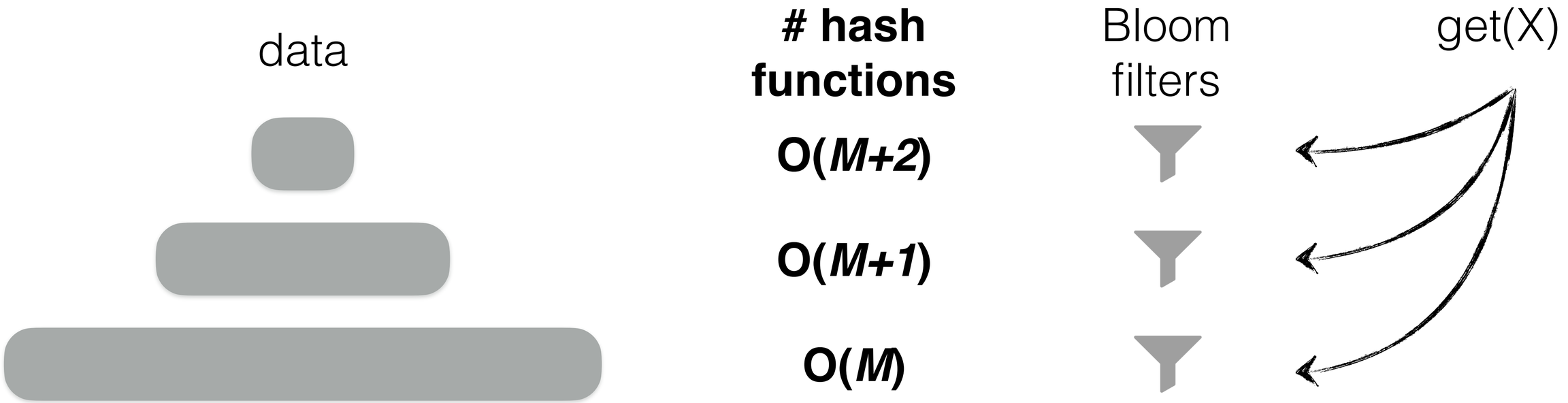
Worst-case:

Avg. worst-case:

Analyze filter access with Monkey

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

Avg. Negative Query = 2



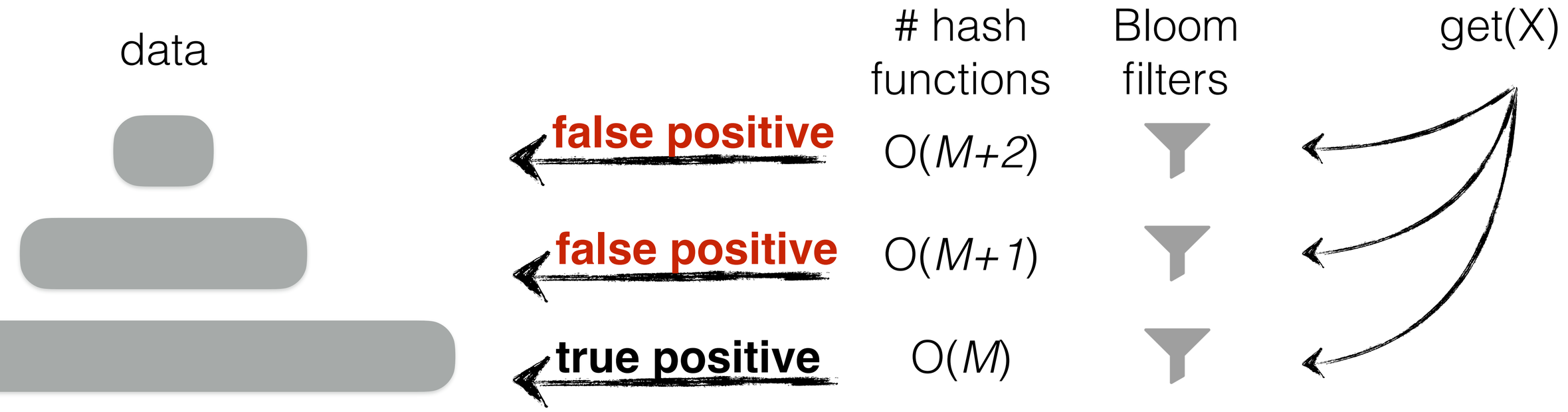
Worst-case:

Avg. worst-case:

Analyze filter access with Monkey

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

Avg. Negative Query = 2



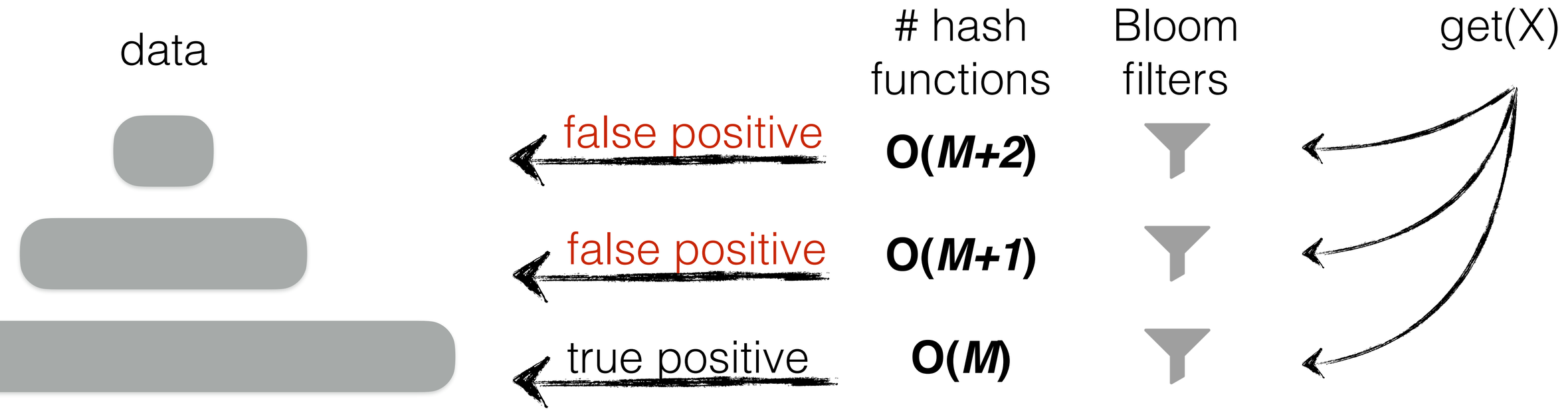
Worst-case:

Avg. worst-case:

Analyze filter access with Monkey

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

Avg. Negative Query = 2



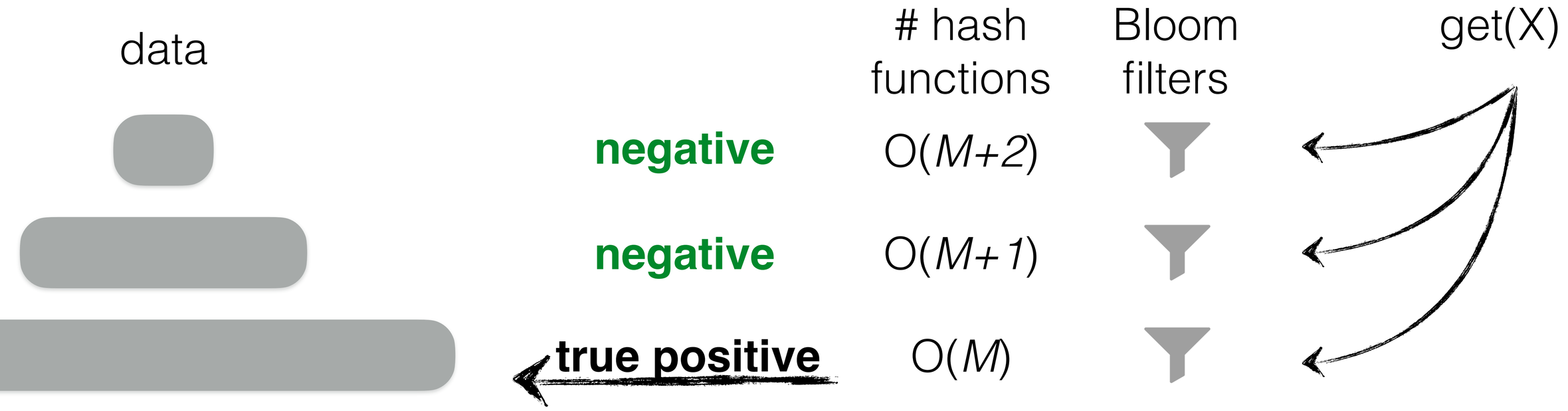
Worst-case: $O(M \cdot L + L^2)$

Avg. worst-case:

Analyze filter access with Monkey

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

Avg. Negative Query = 2



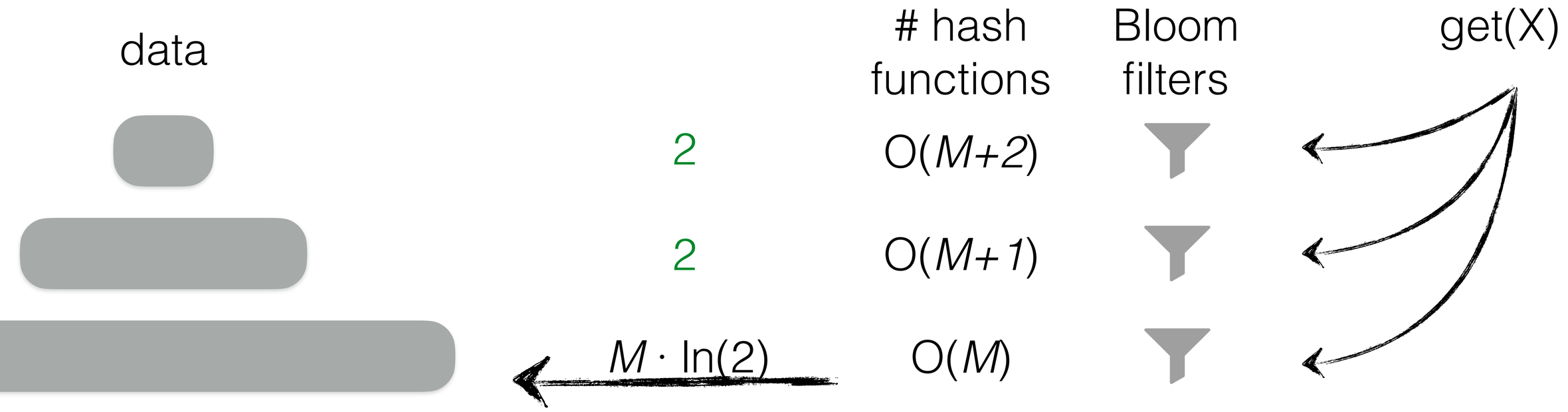
Worst-case: $O(M \cdot L + L^2)$

Avg. worst-case:

Analyze filter access with Monkey

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

Avg. Negative Query = 2



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

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

get I/O cost	$O(2^{-M} \cdot L)$		$O(2^{-M})$
insert I/O cost	$O((T \cdot L)/B)$		$O(?)$
transient space-amp	2x		$O(?)$

$L = \log_T N/P$

(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

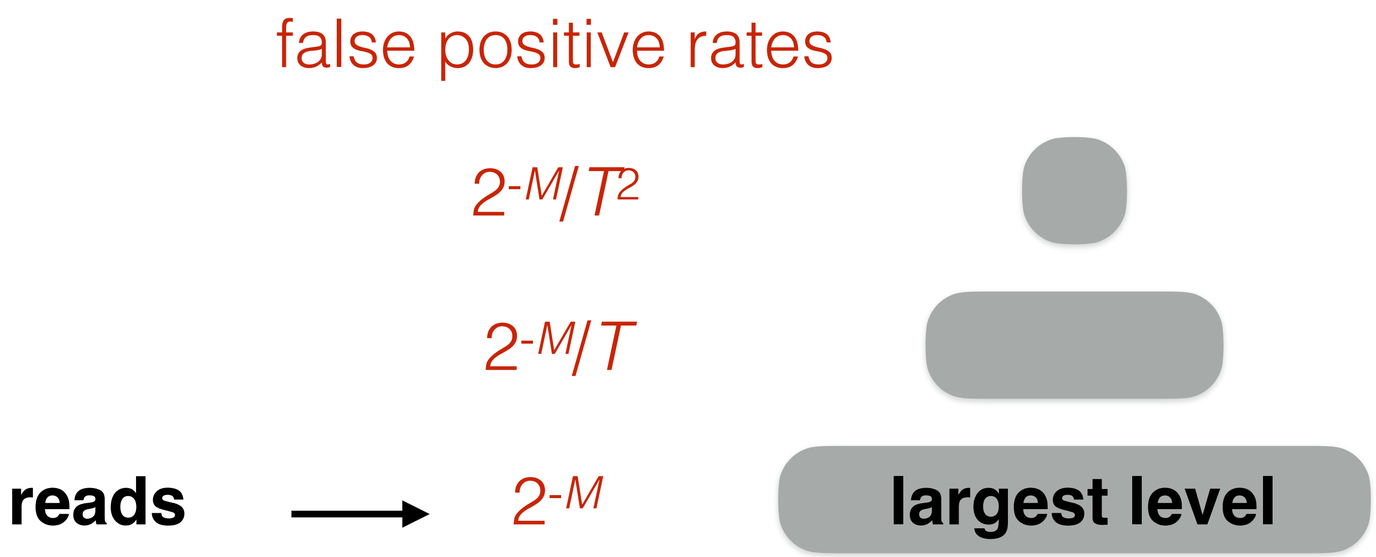
false positive rates

exponentially decreasing

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

$2^{-M/T}$

2^{-M}

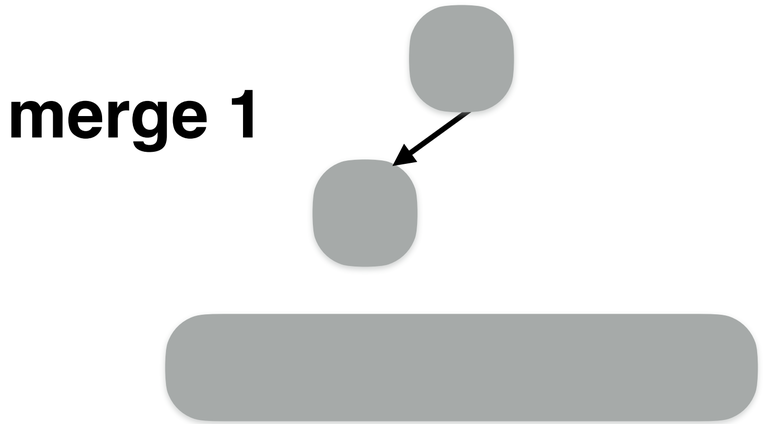

false positive rates
$2^{-M}/T^2$
$2^{-M}/T$
reads
2^{-M}
largest level

reads

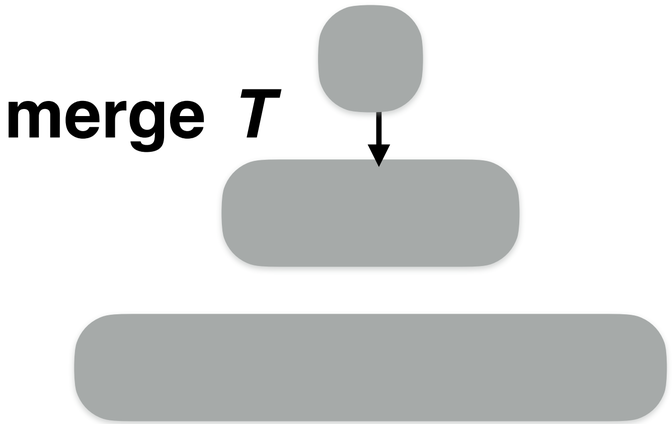
writes

$O(2^{-M})$

writes



writes



writes

O(***T/B***)

O(***T/B***)

O(***T/B***)

O((***T*** ·L)***/B***)

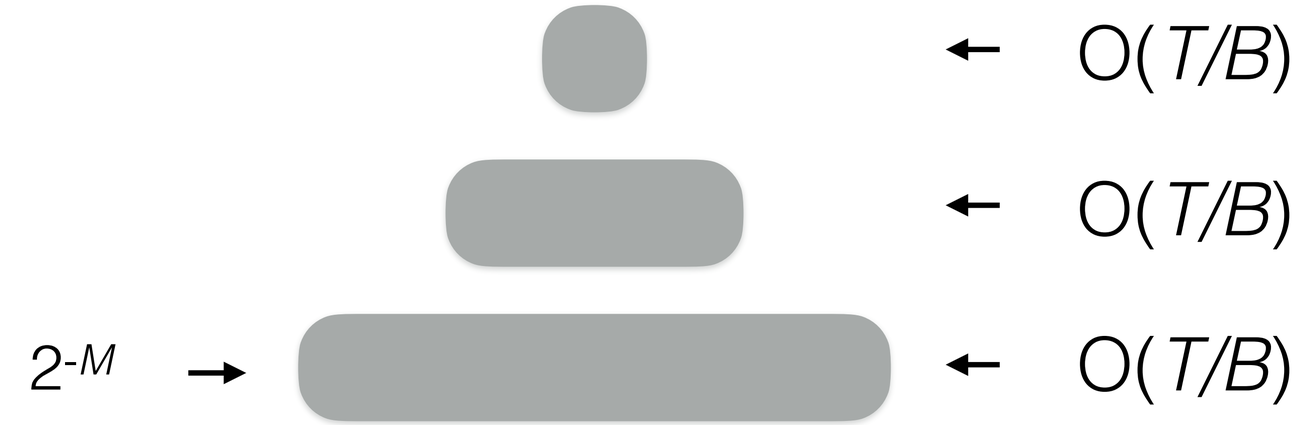
reads	writes
$O(2^{-M})$	$O((T \cdot L)/B)$
=	=
$2^{-M}/T^2$	$O(T/B)$
+	+
$2^{-M}/T$	$O(T/B)$
+	+
2^{-M}	$O(T/B)$

reads

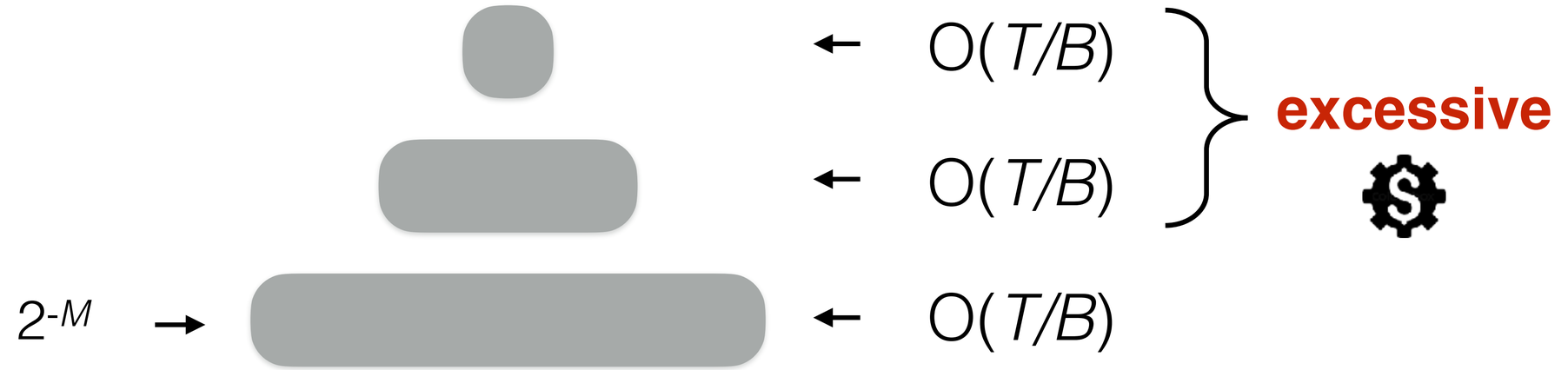
largest level

writes

all levels

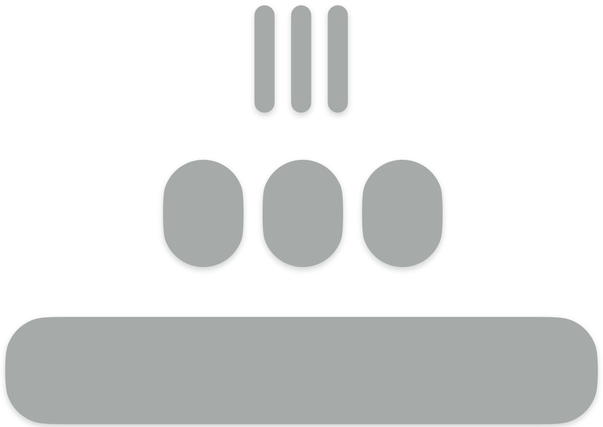


reads **writes**



reads **writes**

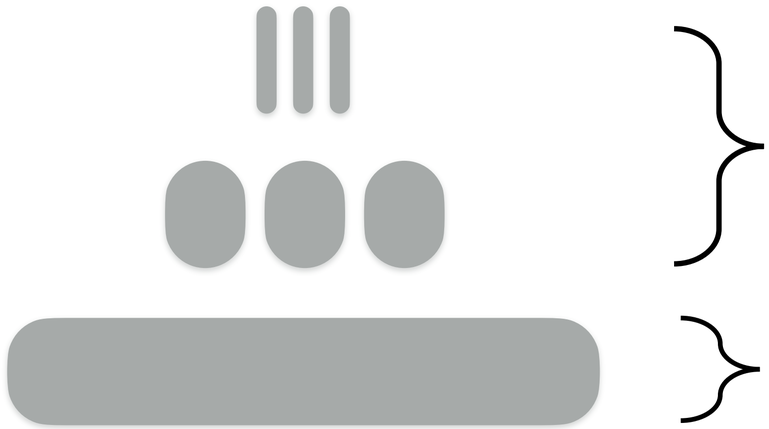
Dostoevsky



lazy

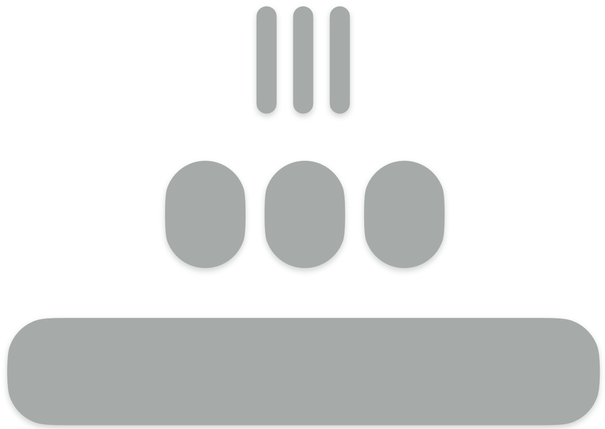
greedy

Dostoevsky



merge when
level fills up

Dostoevsky



merge when

new run comes in

reads **writes**

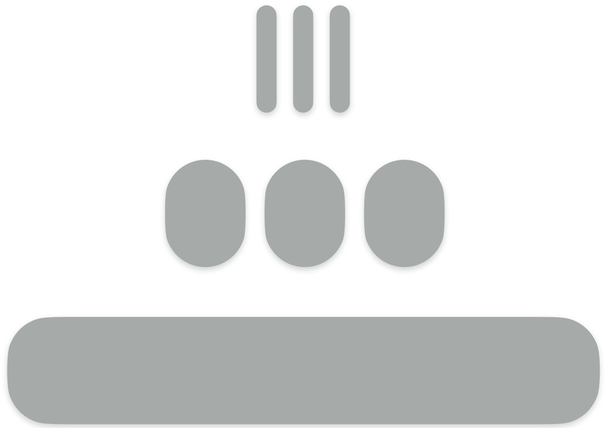
reads **writes**

false positive rates

$2^{-M/T^3}$

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

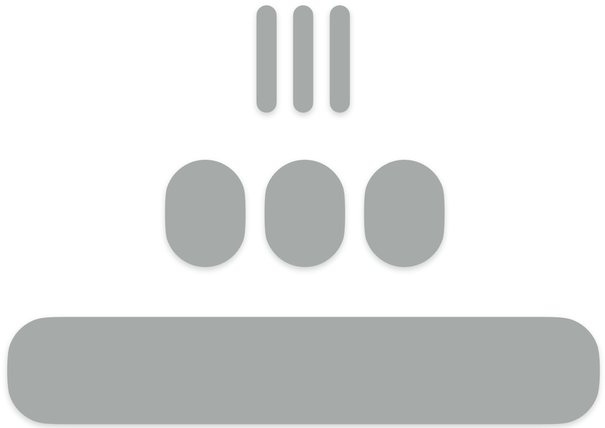
2^{-M}



reads

$O(2^{-M})$

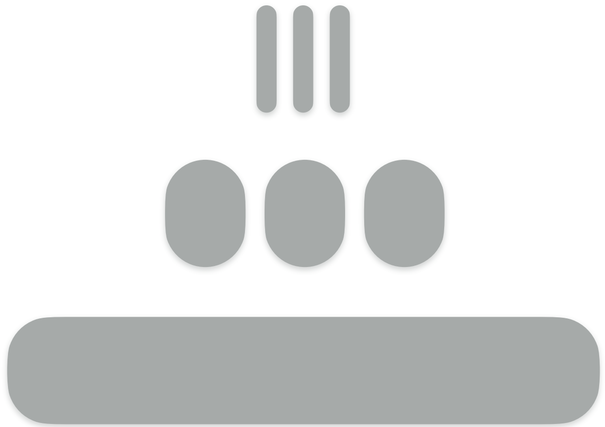
writes



reads

$O(2^{-M})$

writes



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

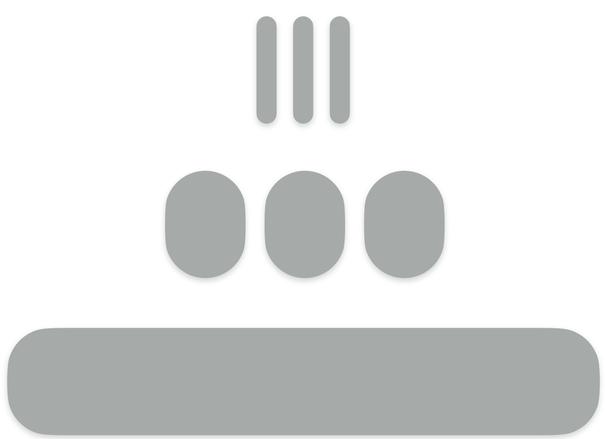
$O(\mathbf{1/B})$

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

reads

$O(2^{-M})$

writes



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

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

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

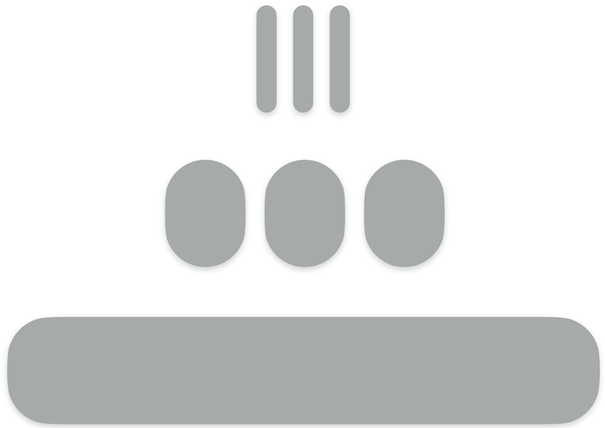
$O((\boldsymbol{T} + \mathbf{L})/\mathbf{B})$

reads

$O(2^{-M})$

writes

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



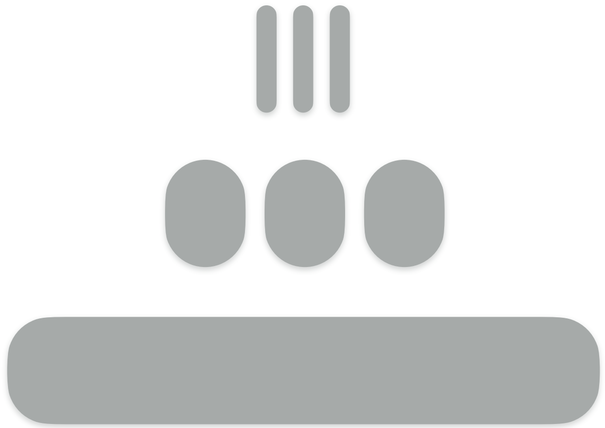
What would our read cost have been if we employed uniform FPRs at all levels?

reads

$O(2^{-M})$

writes

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



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

uniform

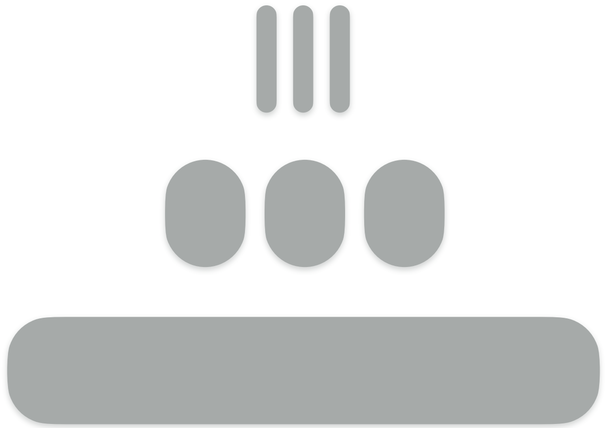
reads

$O(2^{-M} \cdot T \cdot L)$

2^{-M}

2^{-M}

2^{-M}



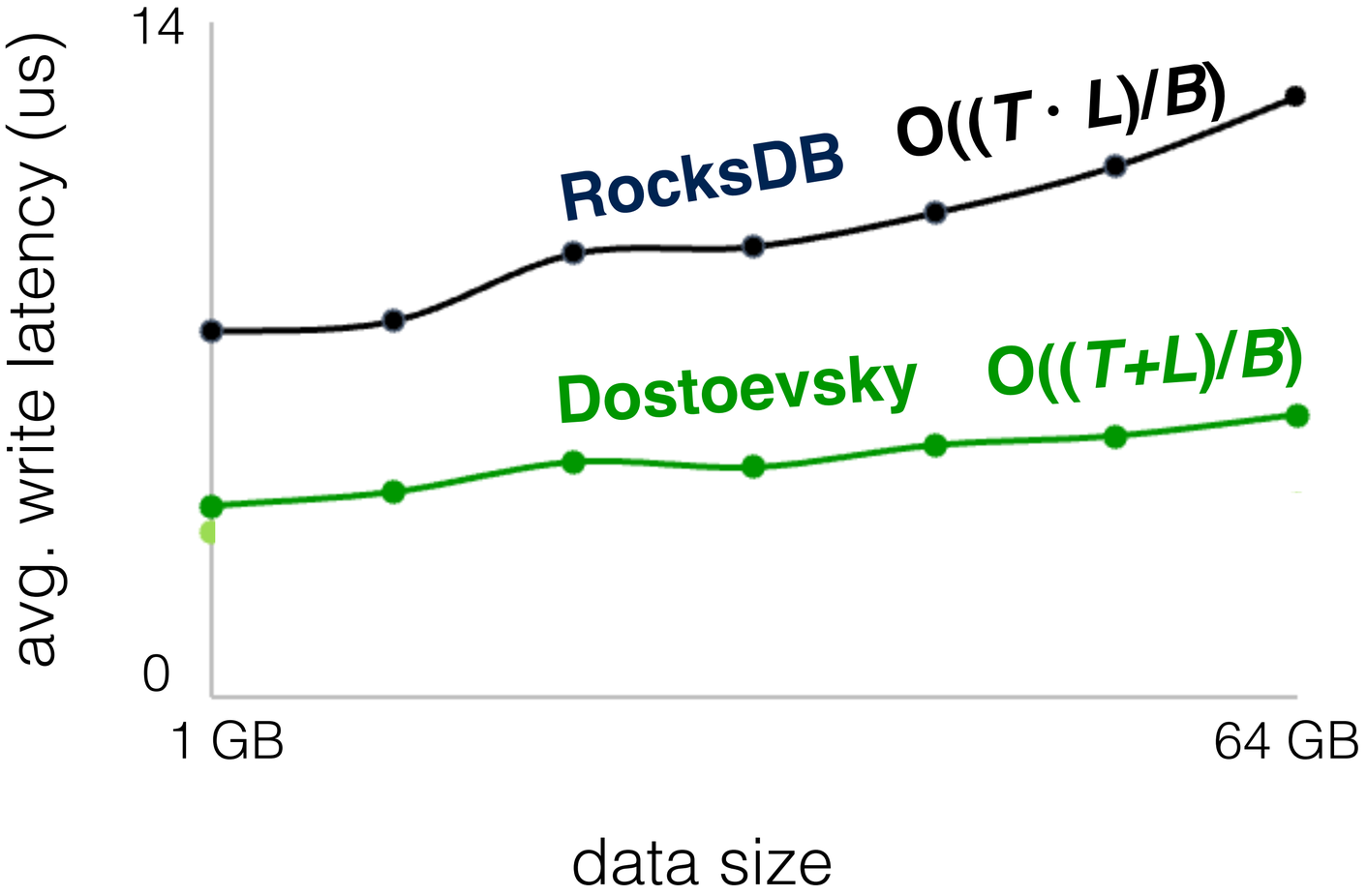
Configuration

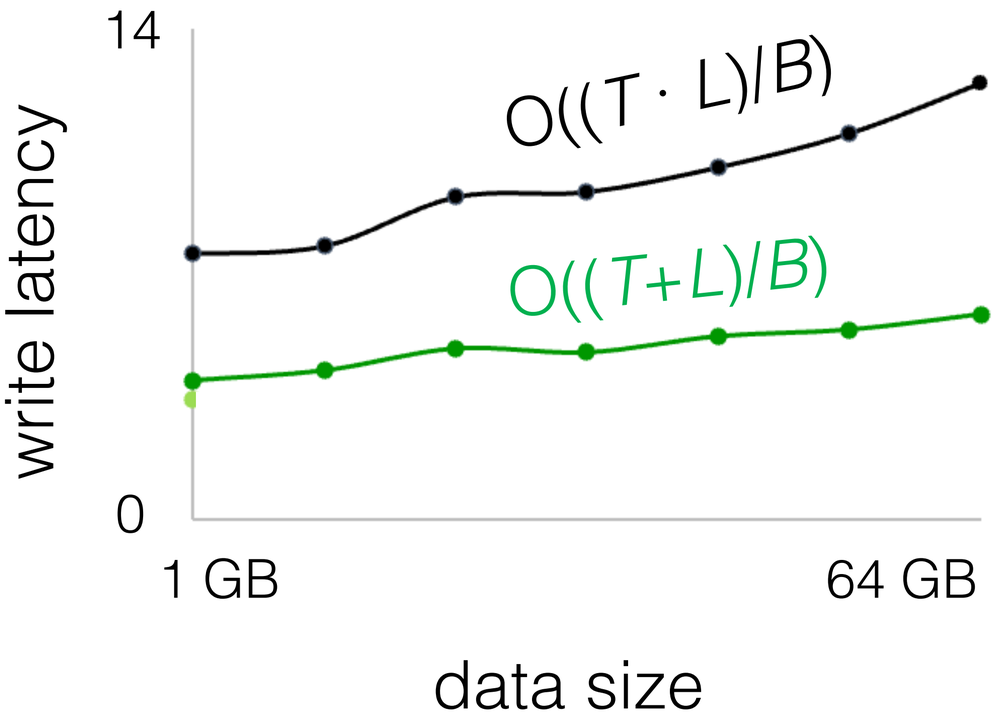
buffer 2MB

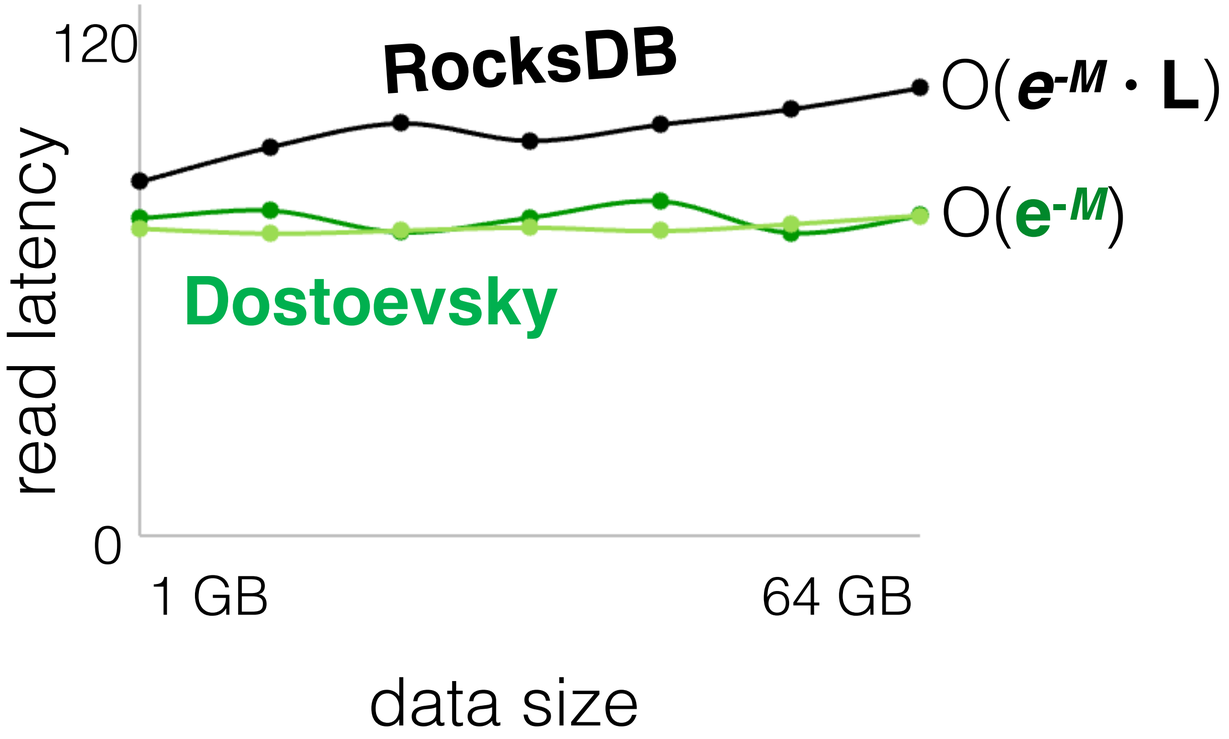
size ratio: 5

1KB entries

SSD storage




14
0
write latency
O((T · L)/B)
O((T+L)/B)
1 GB
64 GB
data size


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

get I/O cost

$O(2^{-M} \cdot L)$

$O(2^{-M})$

insert I/O cost

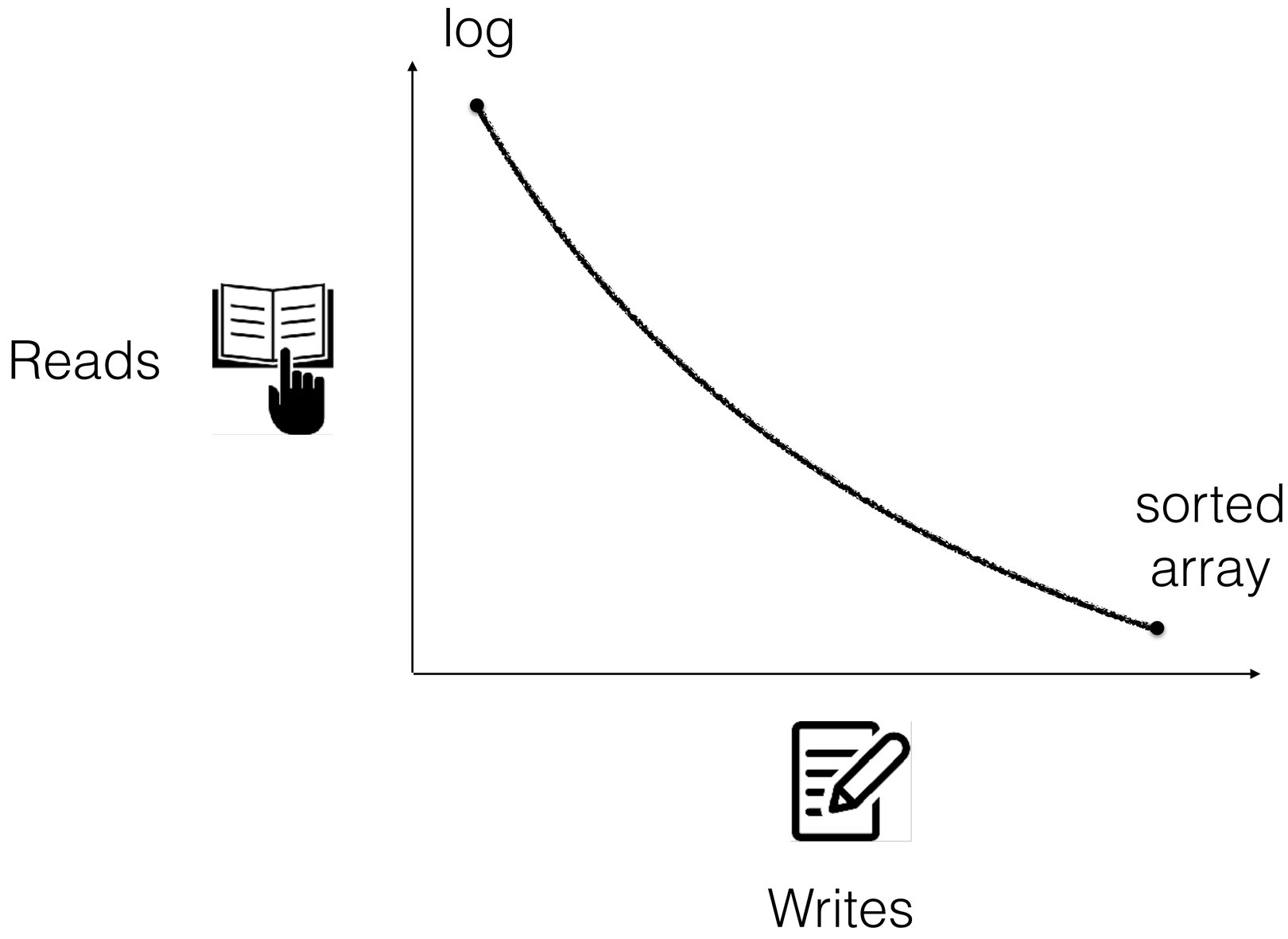
$O((T \cdot L)/B)$

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

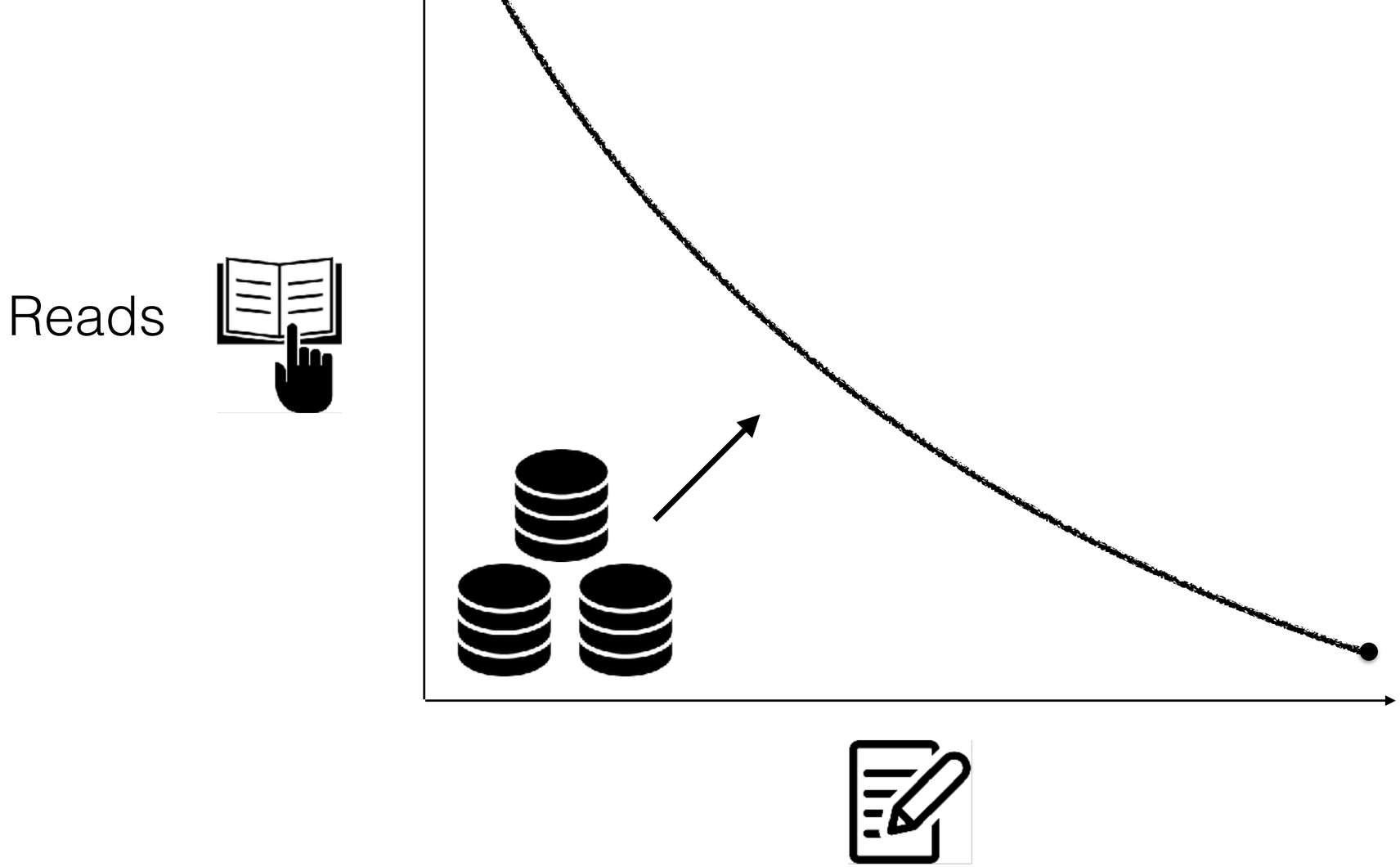
$L = \log_T N/P$

(Costs assuming leveling)

Better scalability with data growth

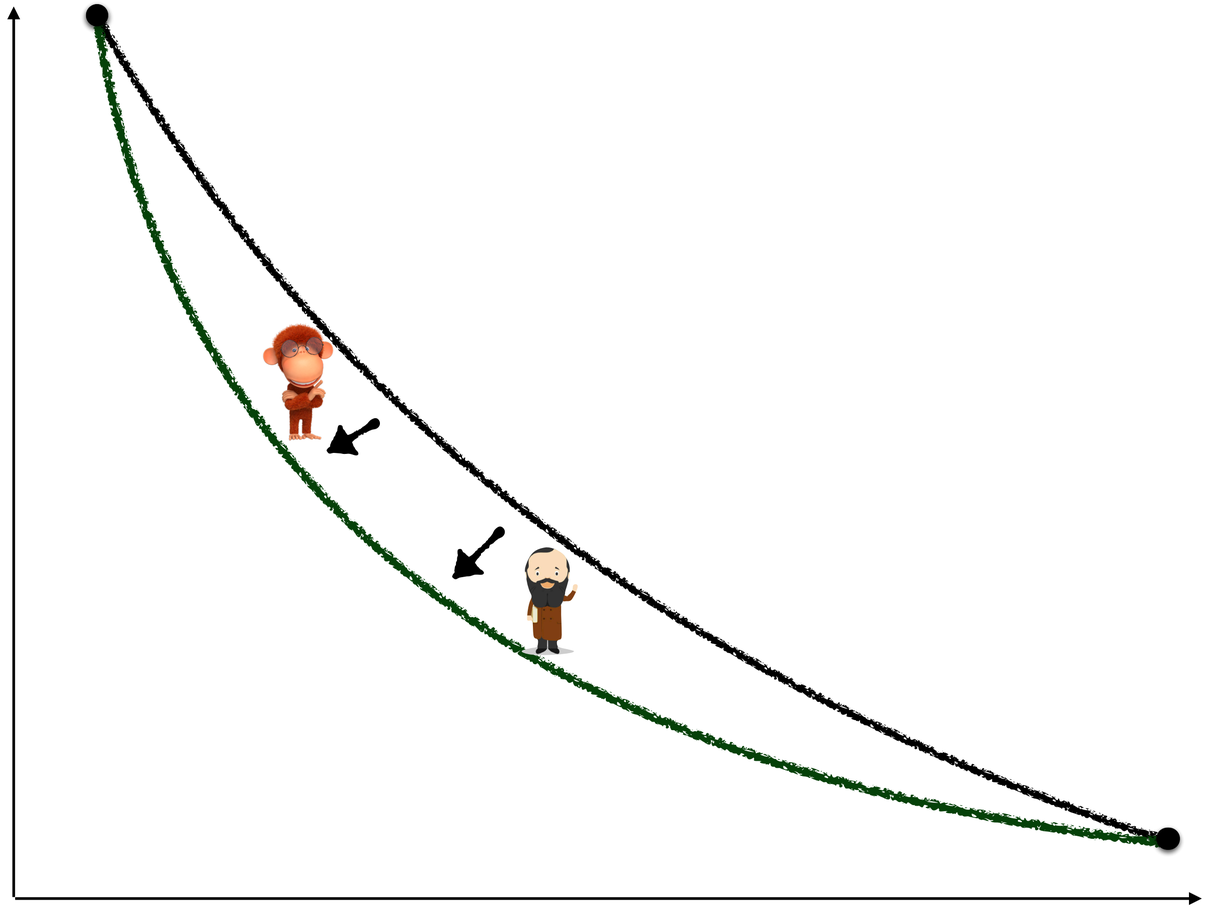

log
Reads
sorted
array
Writes

Reads



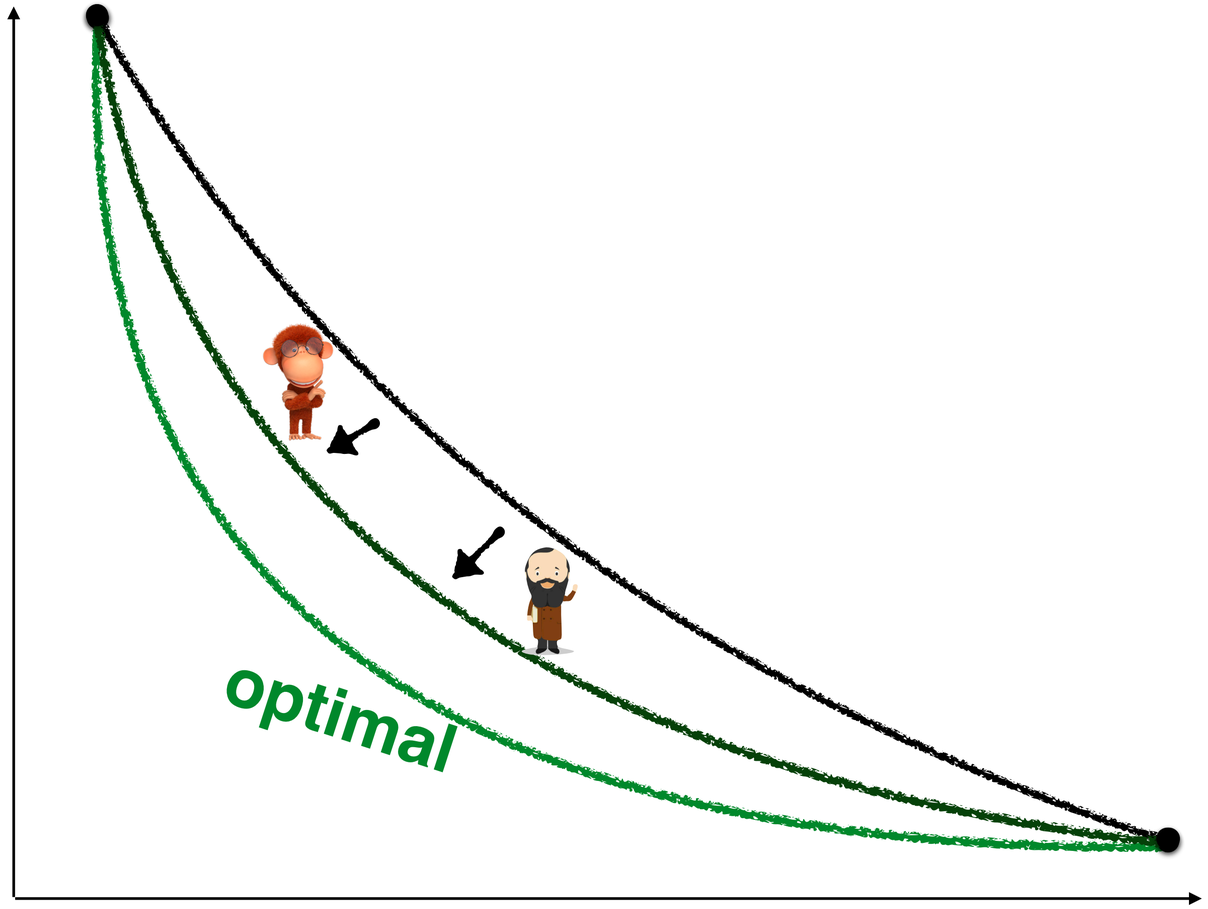
Writes

Reads

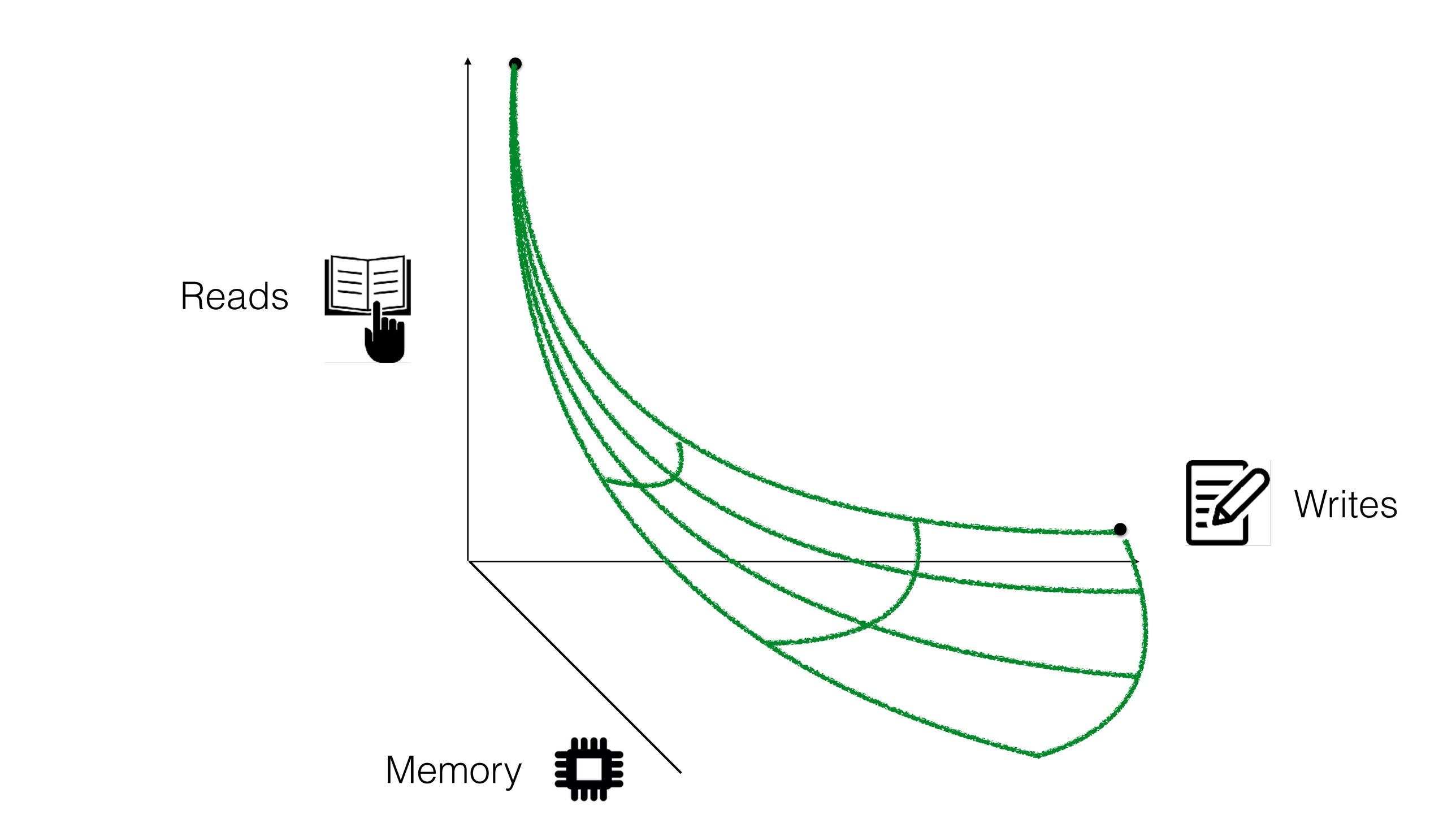


Writes

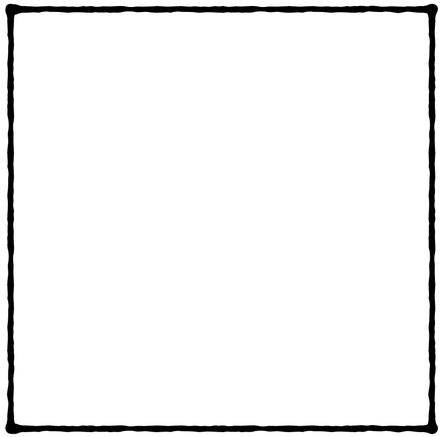
Reads



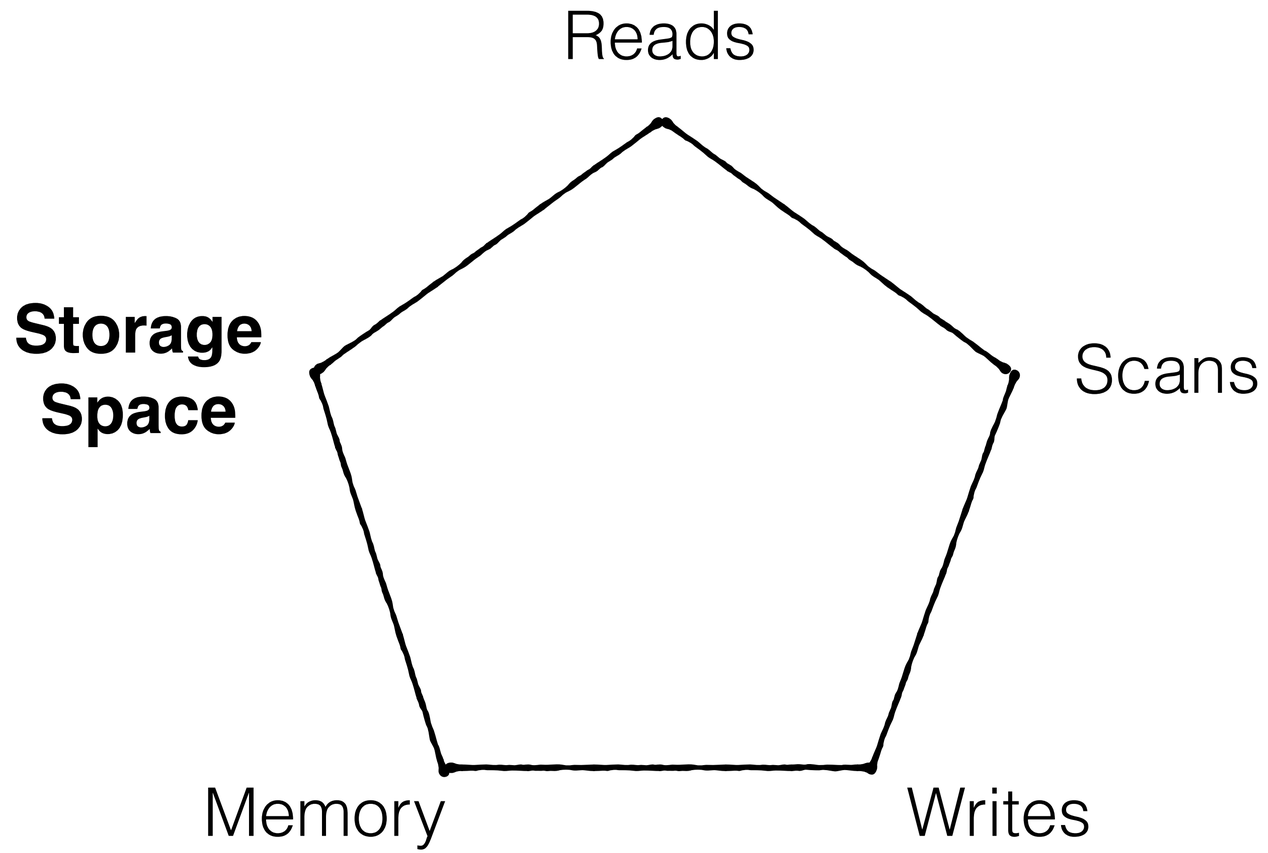
Writes

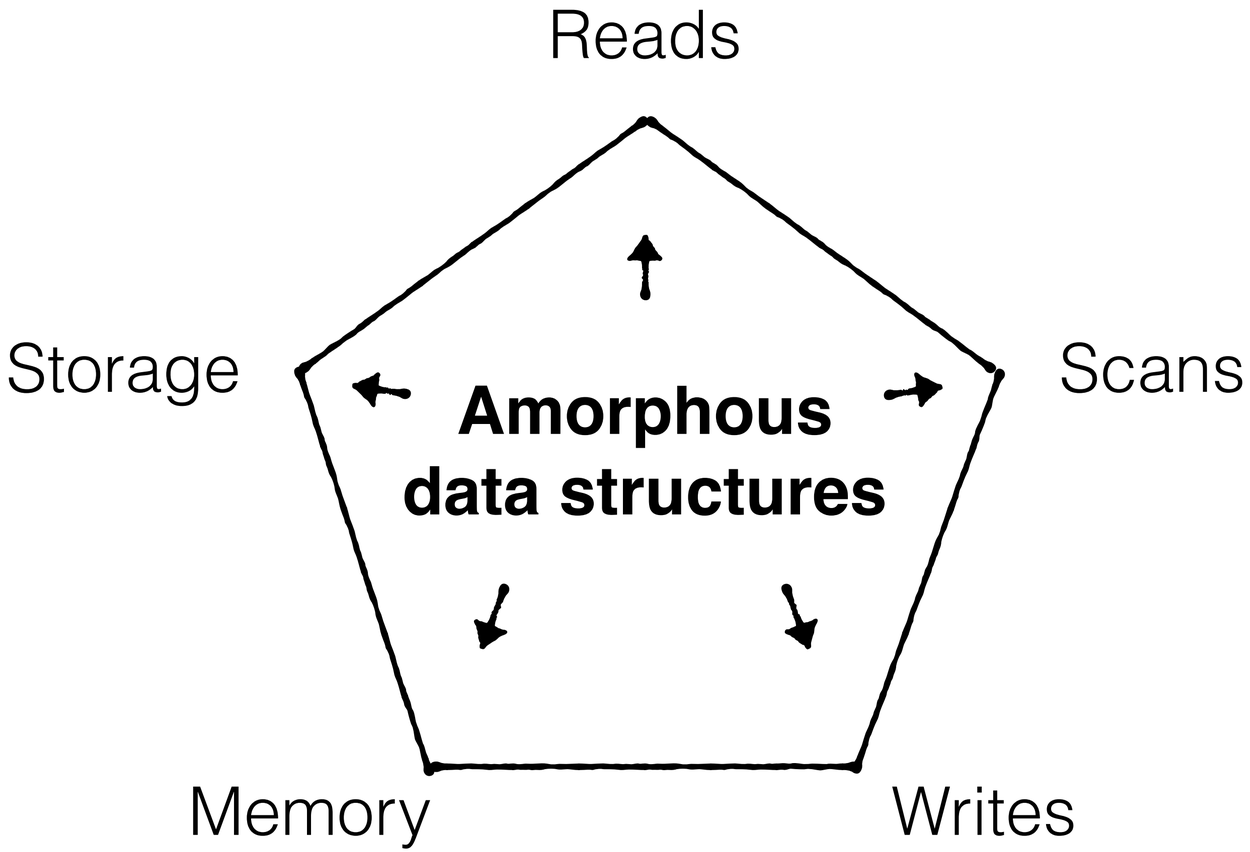

Reads
Writes
Memory

Reads **Range Scans**



Memory Writes


Reads
Storage Space
Scans
Memory
Writes


Reads
Storage
Scans
Amorphous
data structures
Memory
Writes

Thanks