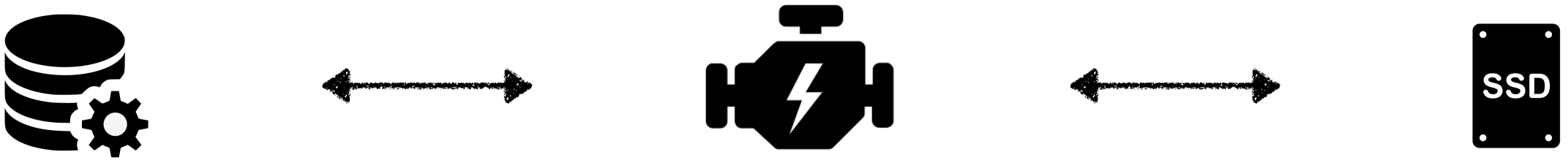
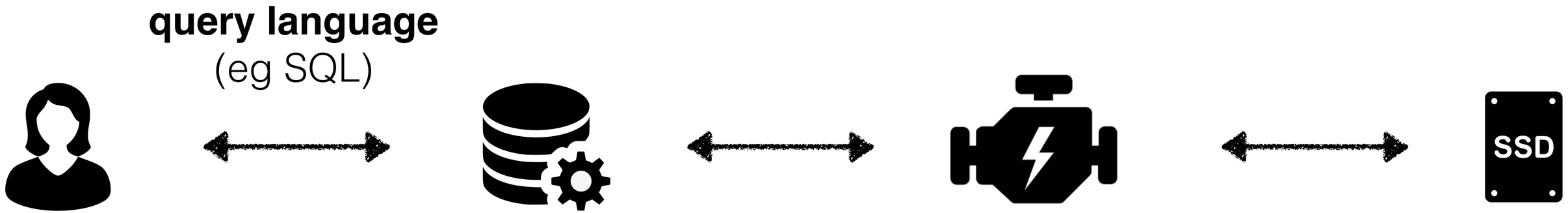
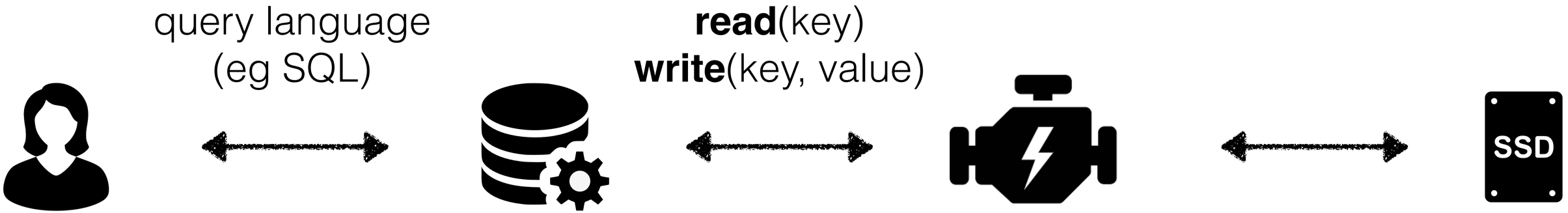
Scaling Storage Engines for 100x Big Data

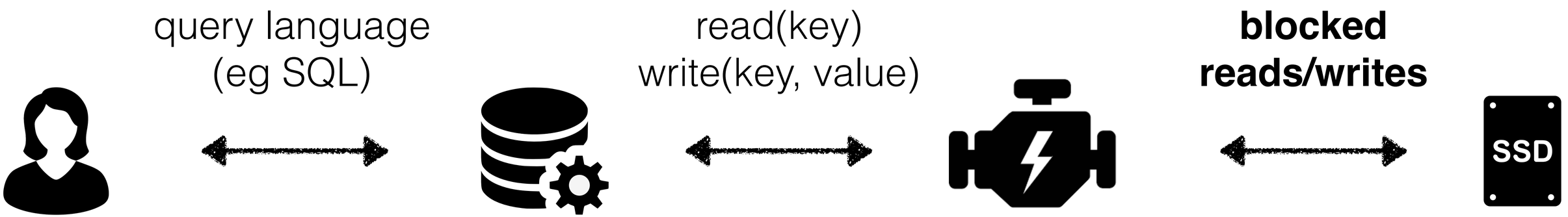
Niv Dayan
University of Toronto



Storage Engines

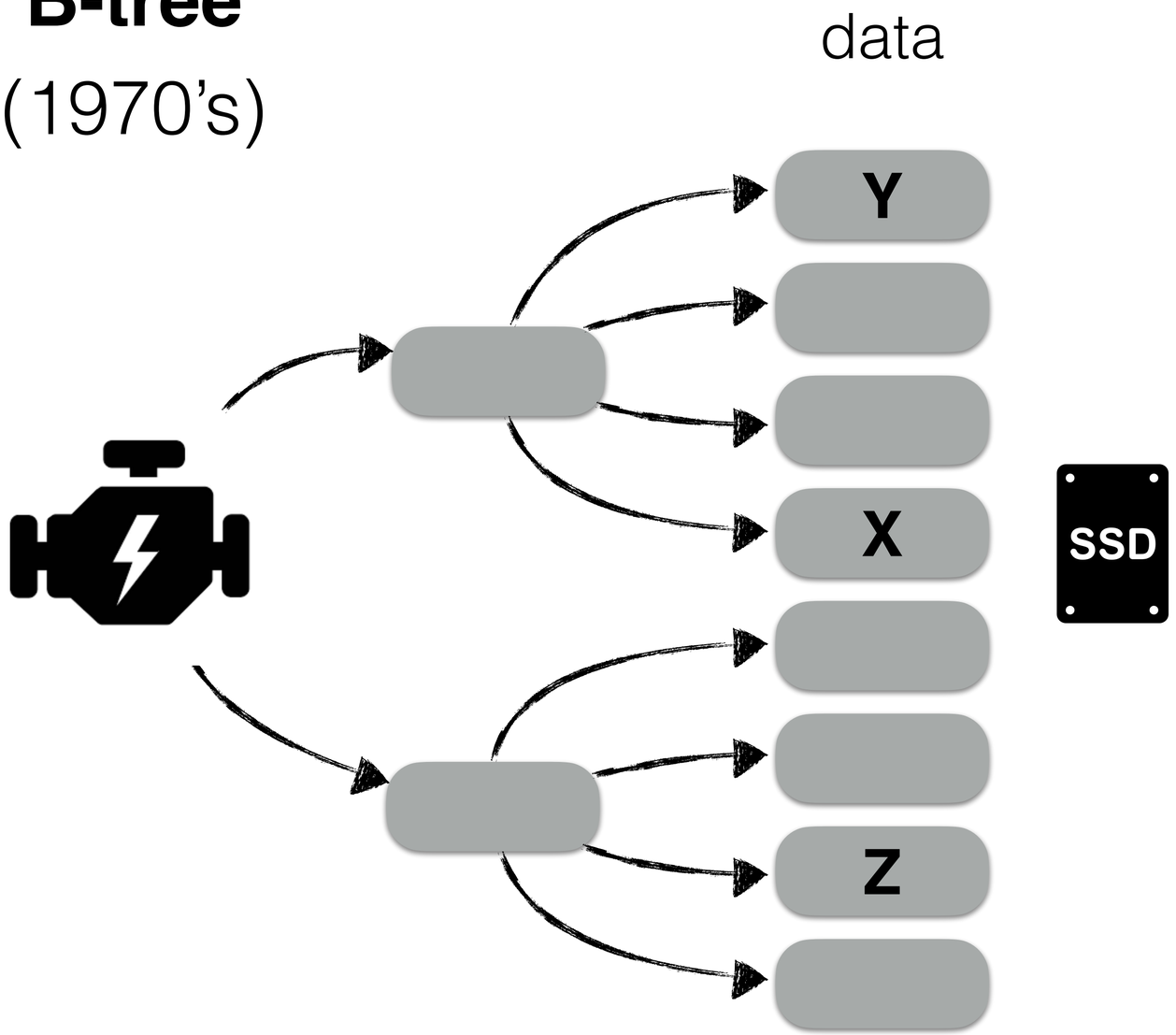

query language
(eg SQL)
SSD

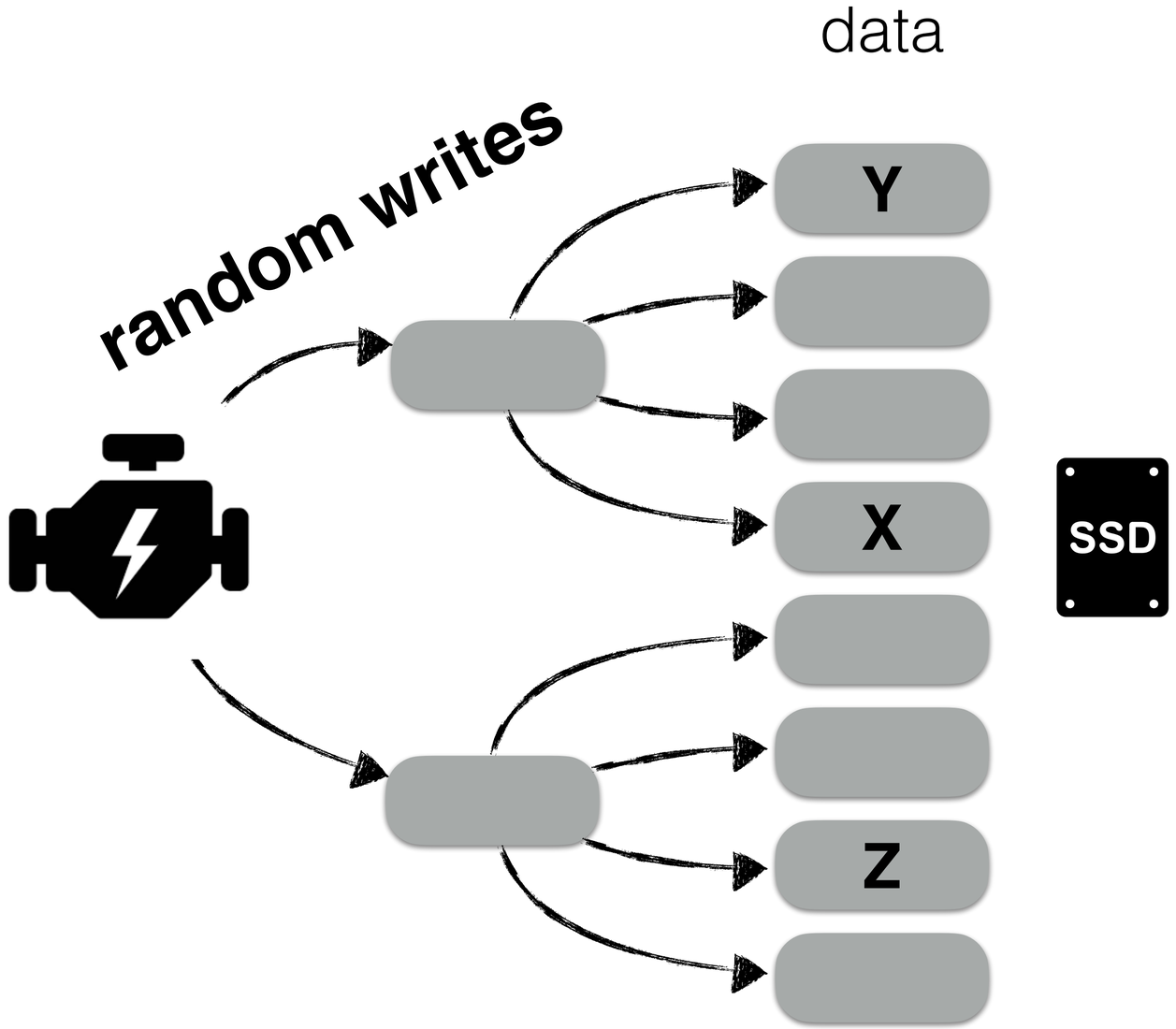

query language
(eg SQL)
read(key)
write(key, value)
SSD


query language
(eg SQL)
read(key)
write(key, value)
blocked
reads/writes
SSD

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

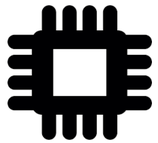
Can random writes be eliminated?

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

only sequential writes

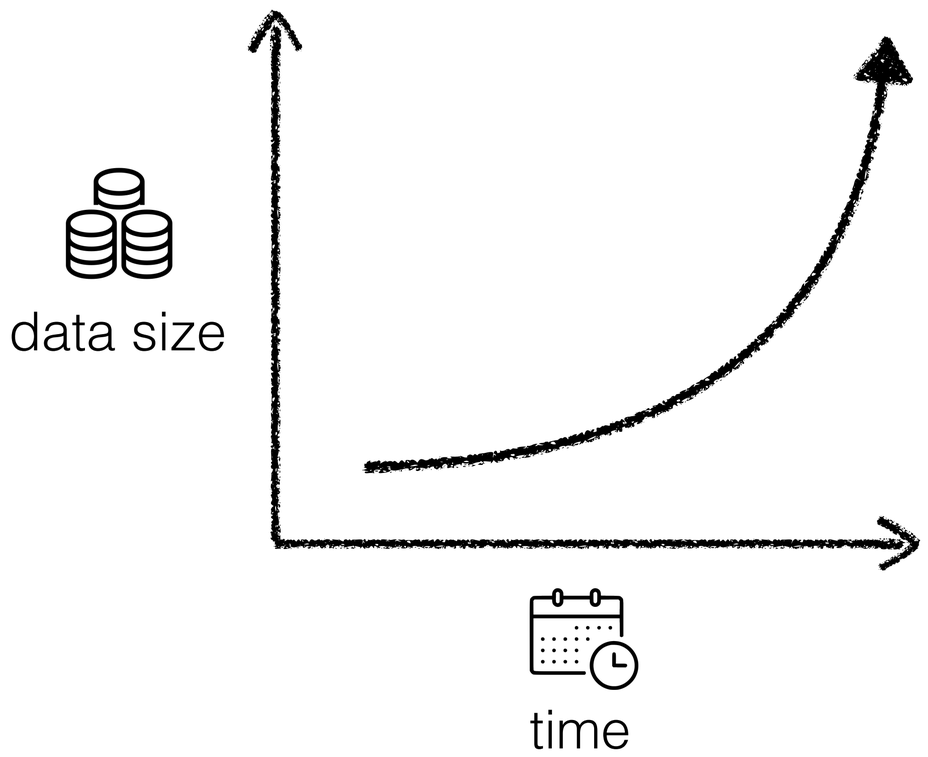
SSD

1 3 6

2 4 5

1 2 3 4 5 6

How well can LSM-tree handle exponential data growth?



logarithmic scaling

O(log N)

logarithmic scaling

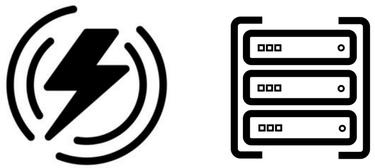
$O(\log N)$

exponential growth

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

linear scaling

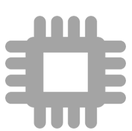
O(**time**)

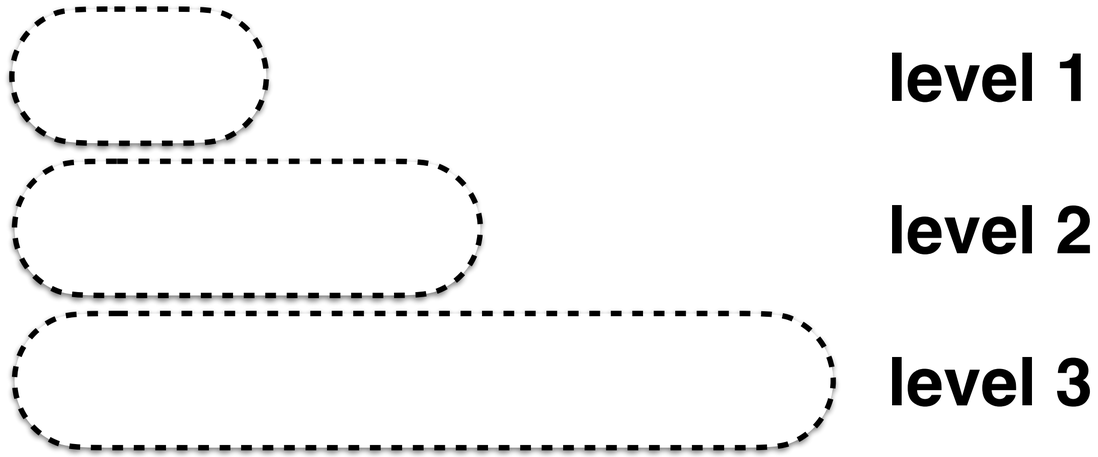


Can we scale better?

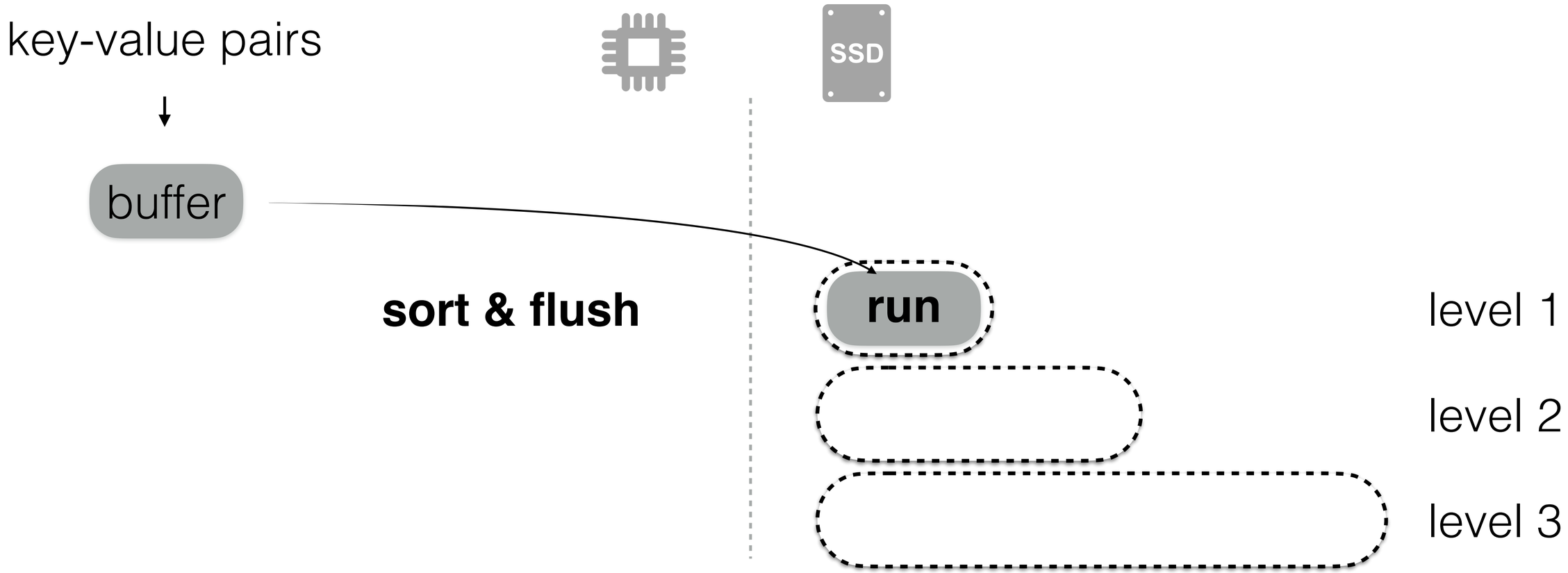
LSM-Tree

key-value pairs

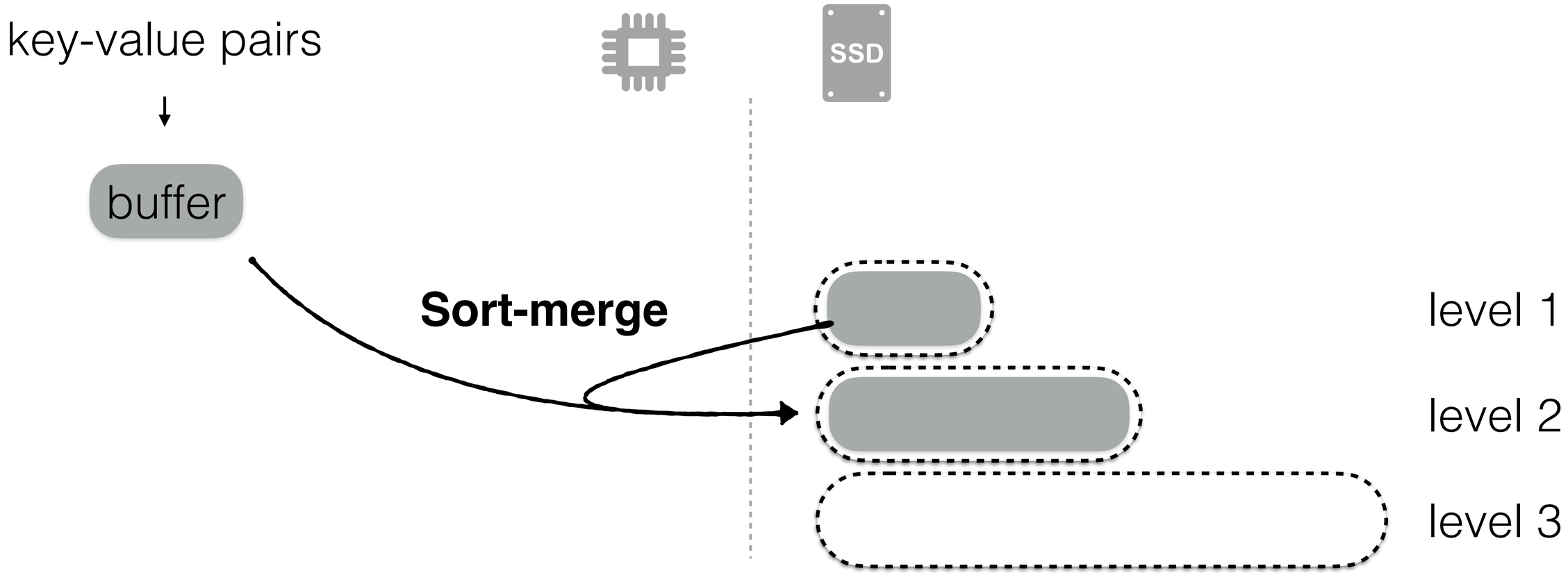


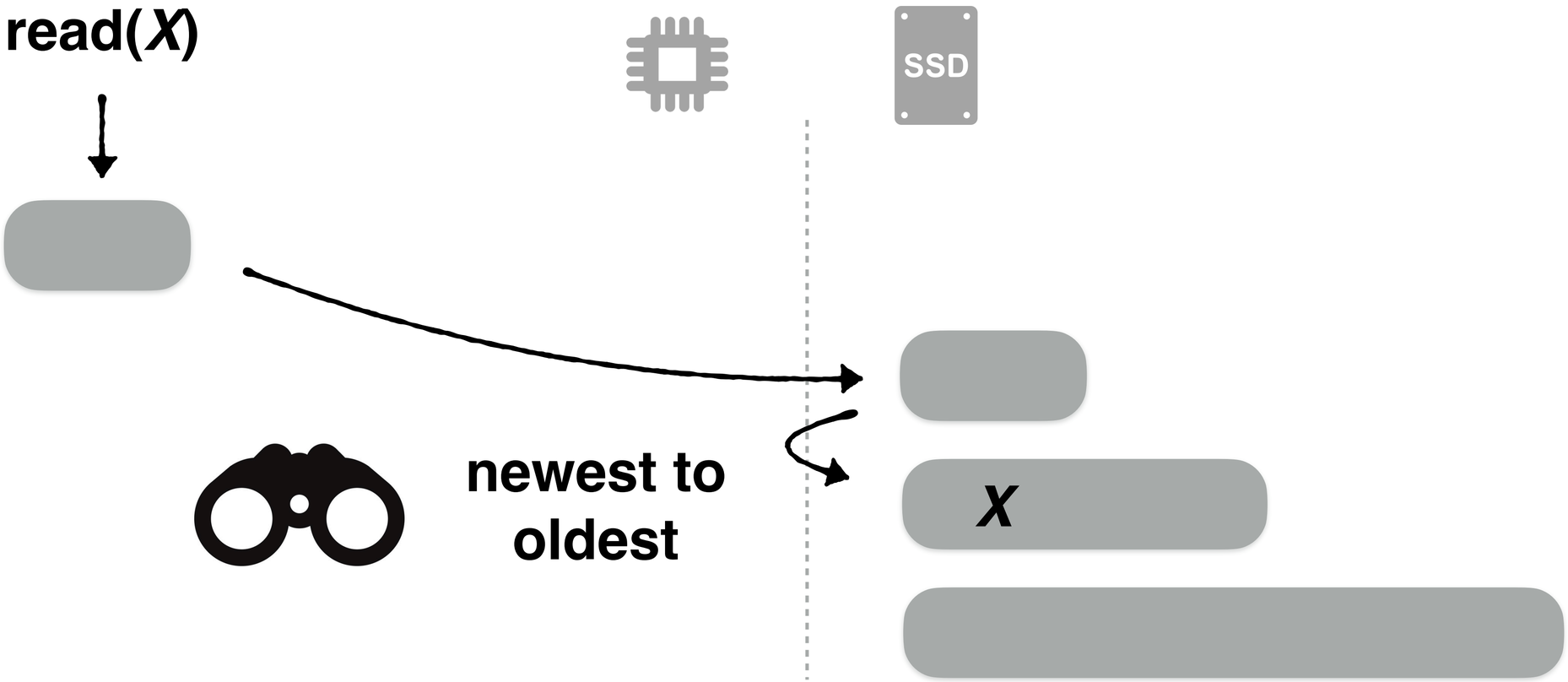


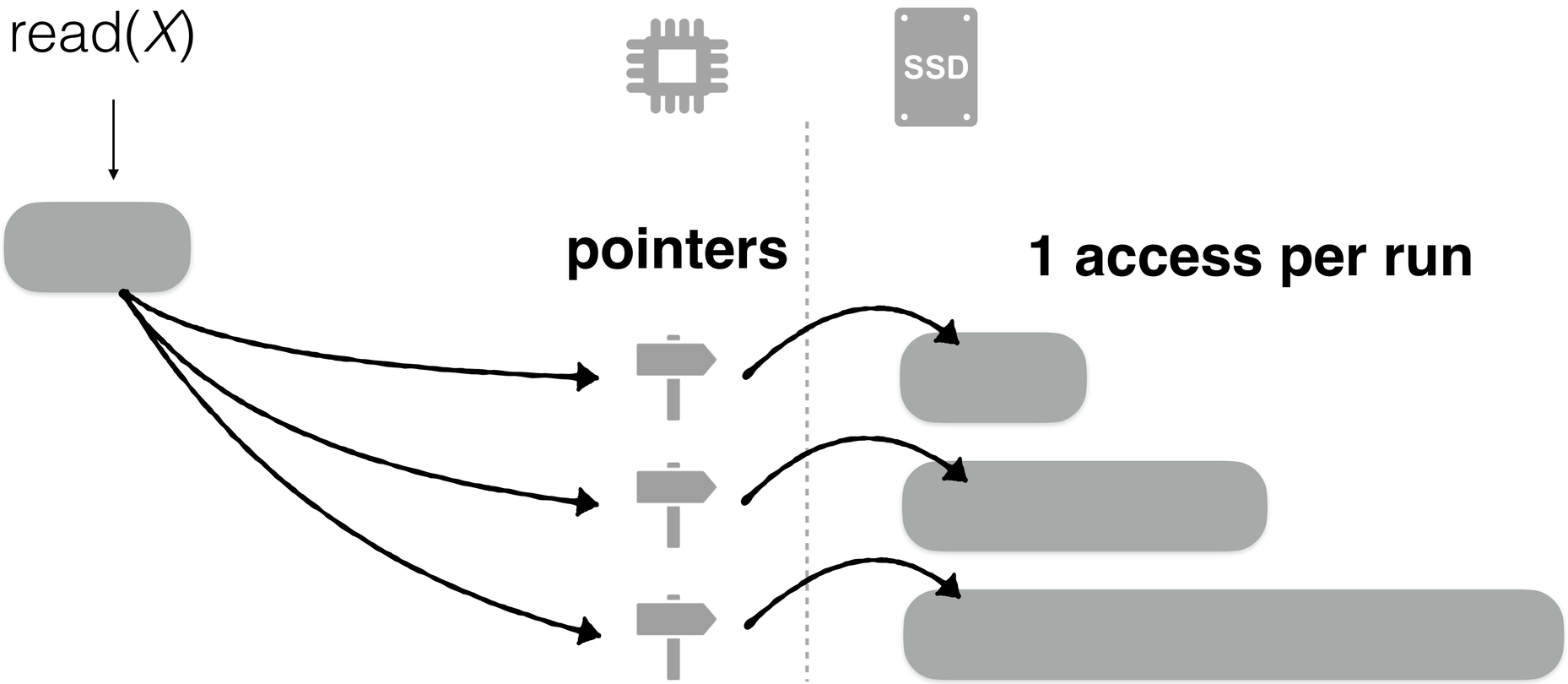
LSM-Tree

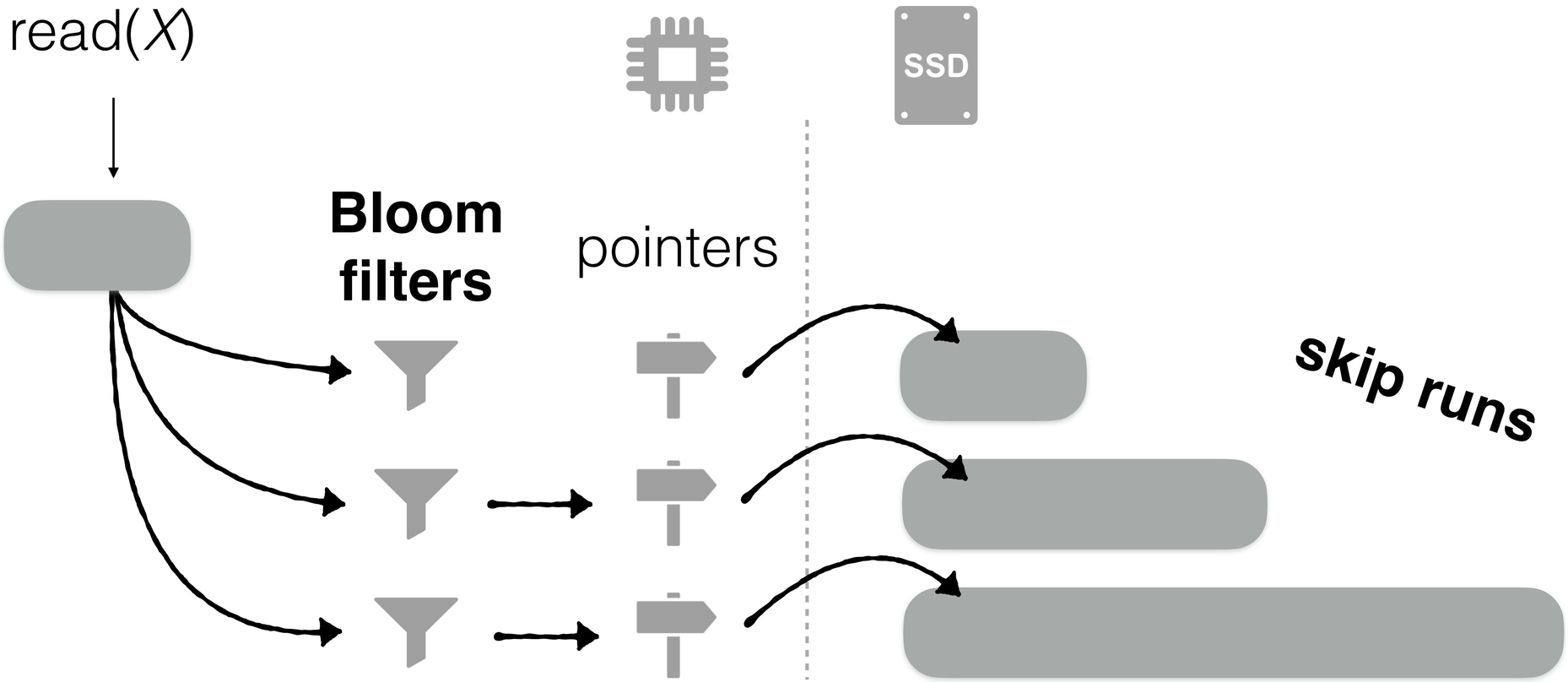

key-value pairs
buffer
SSD
sort & flush
run
level 1
level 2
level 3

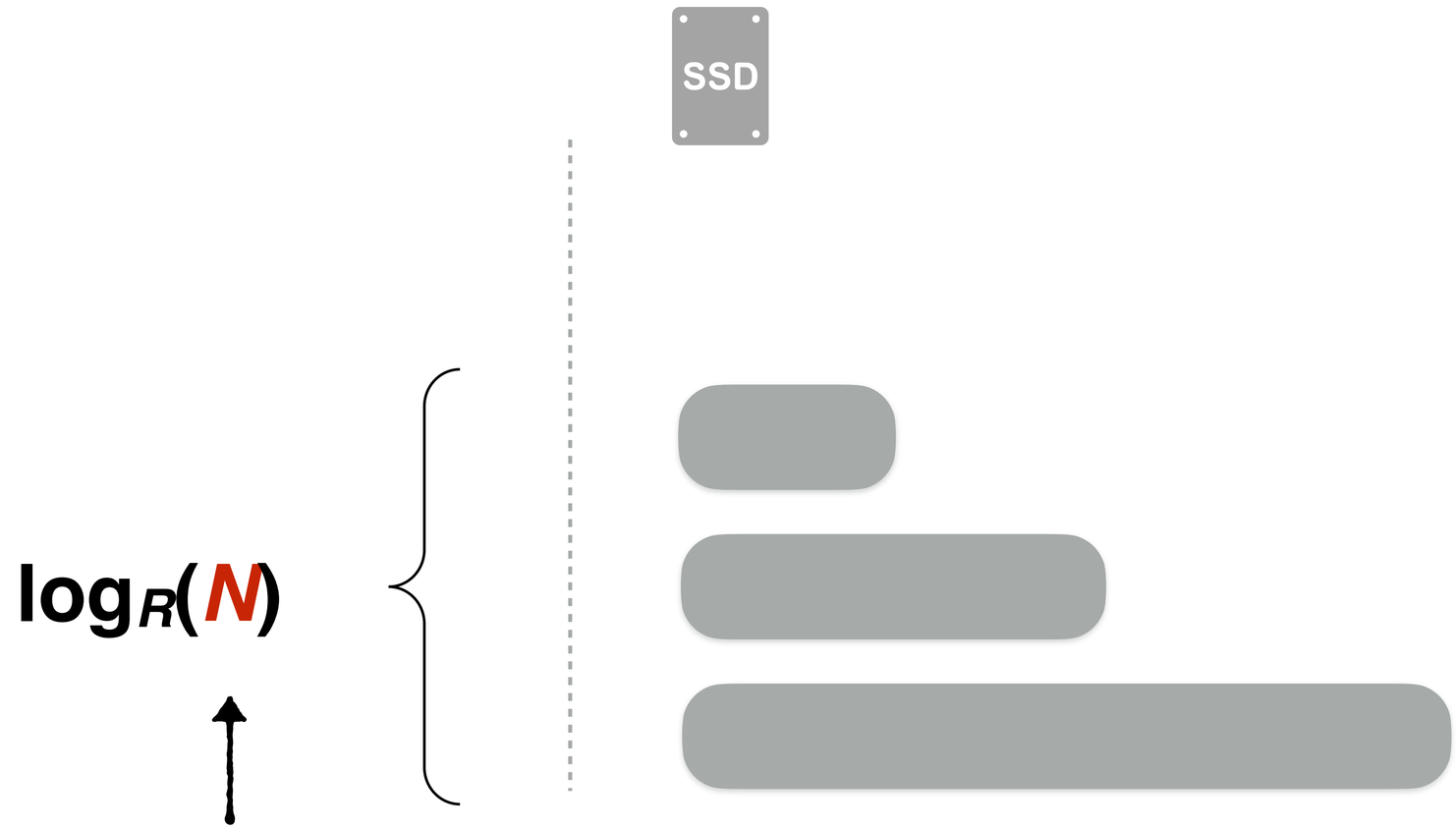
LSM-Tree

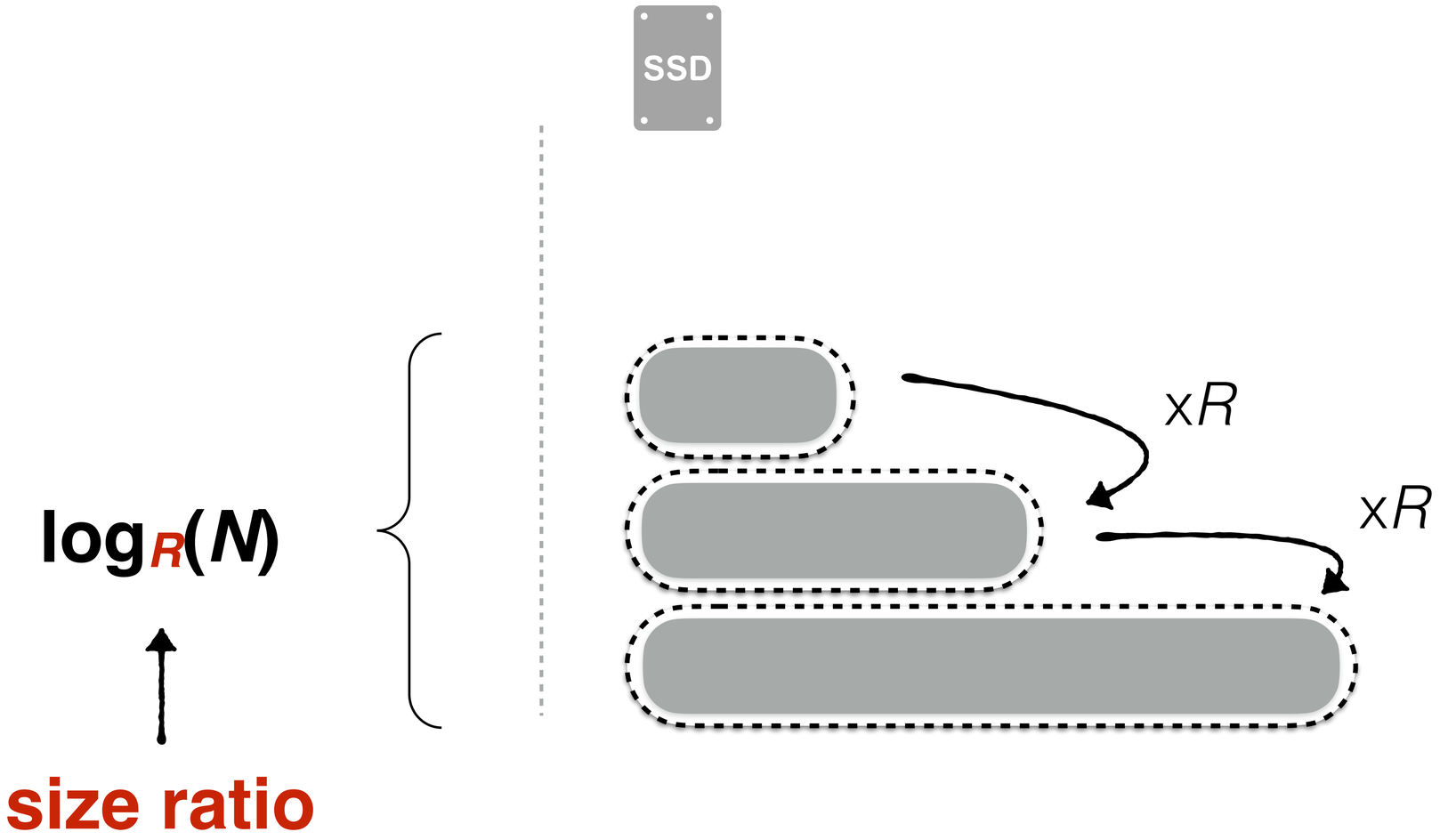



read(X)
SSD
newest to
oldest
X


read(X)
SSD
pointers
1 access per run


read(X)
SSD
Bloom filters
pointers
skip runs


SSD
$\log_R(N)$
data size


SSD
xR
xR
log_R(N)
size ratio

reads & **writes** $\in$ O(**log** ***N***)

storage reads

O(log *N*)

O(*?*)

storage writes

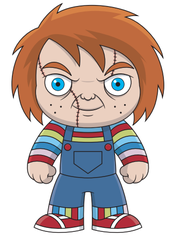
O(log *N*)

O(*?*)

O(*?*)

CPU reads

O(log *N*)

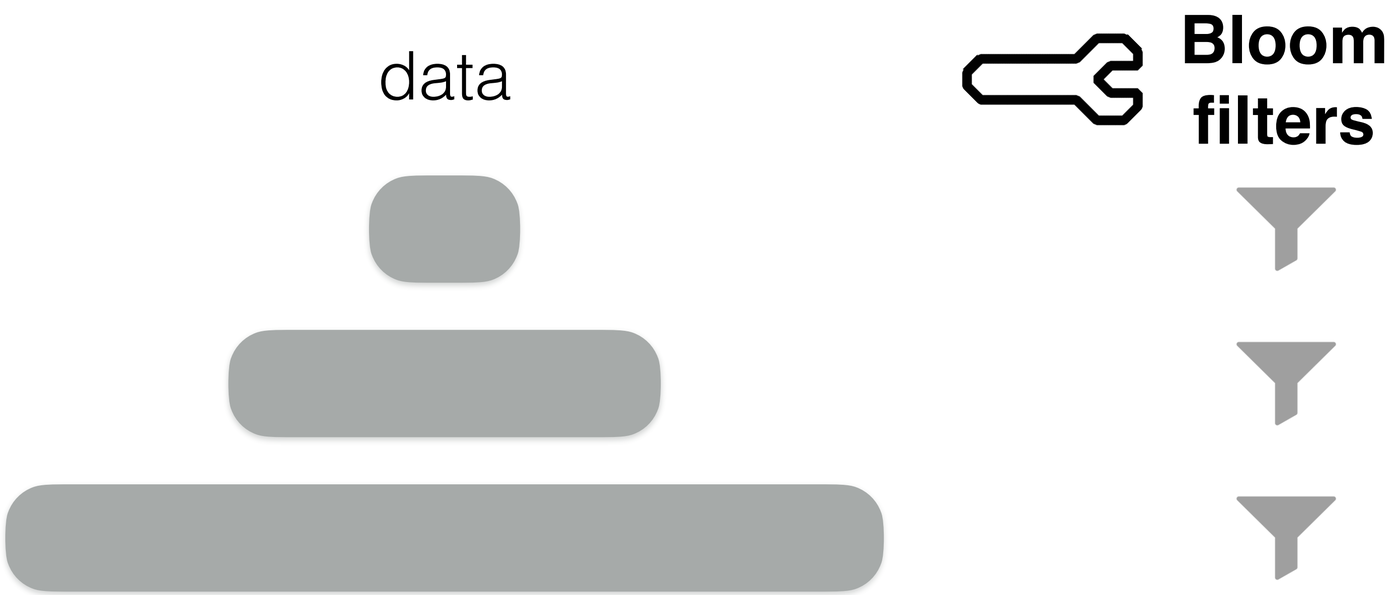


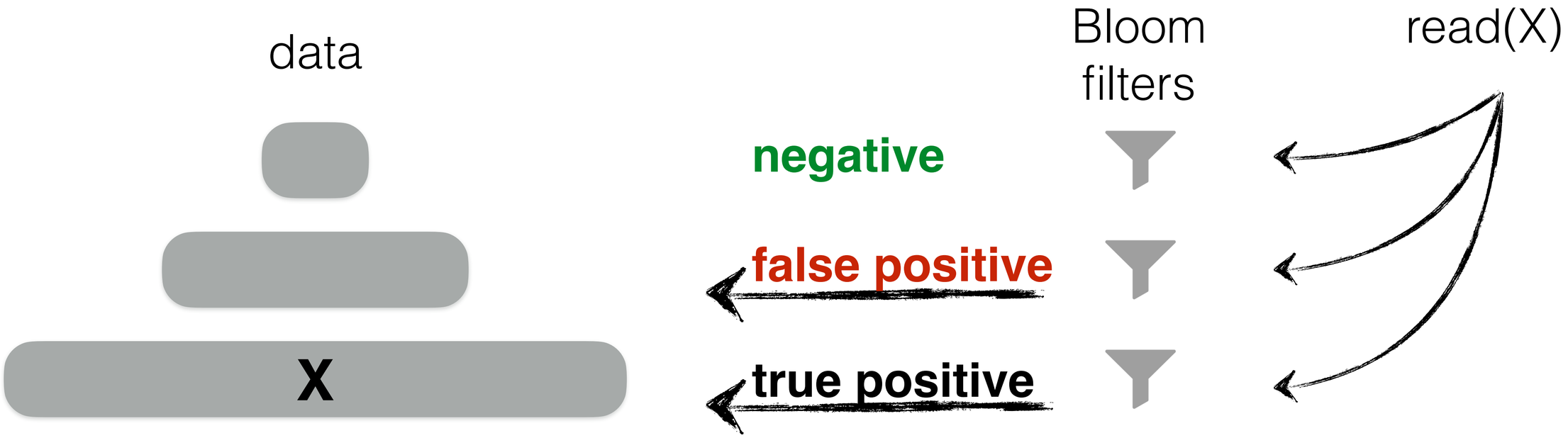
O(*?*)

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

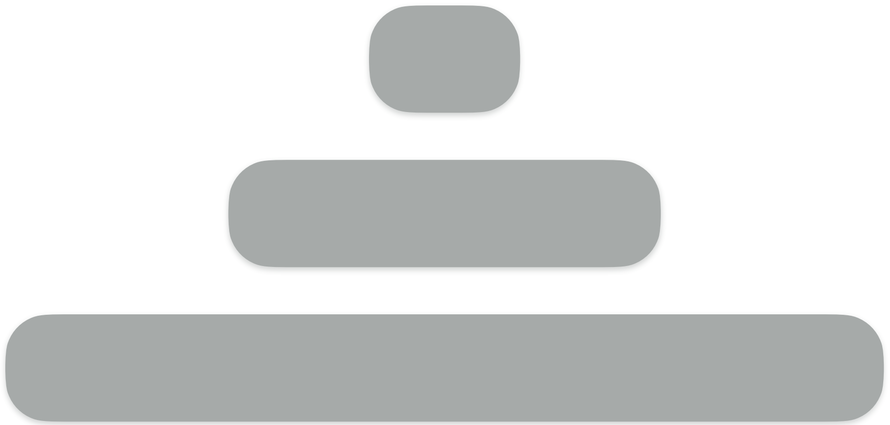
SIGMOD17

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




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

data



Bloom filters

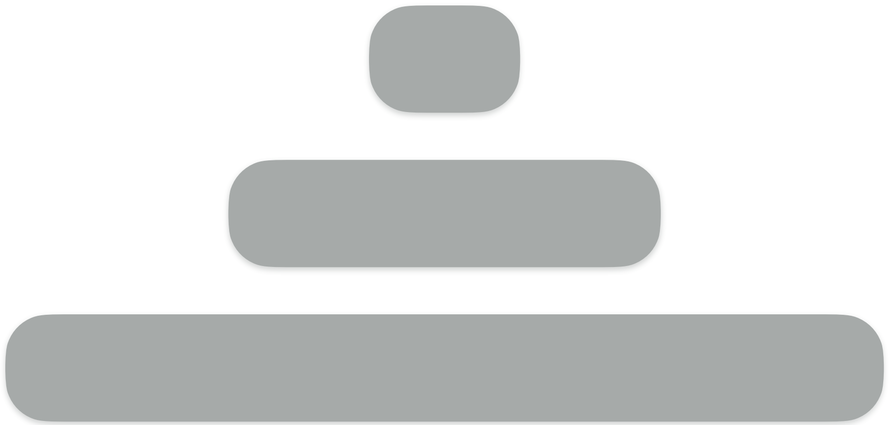
bits/entry

M

M

M

data



Bloom filters

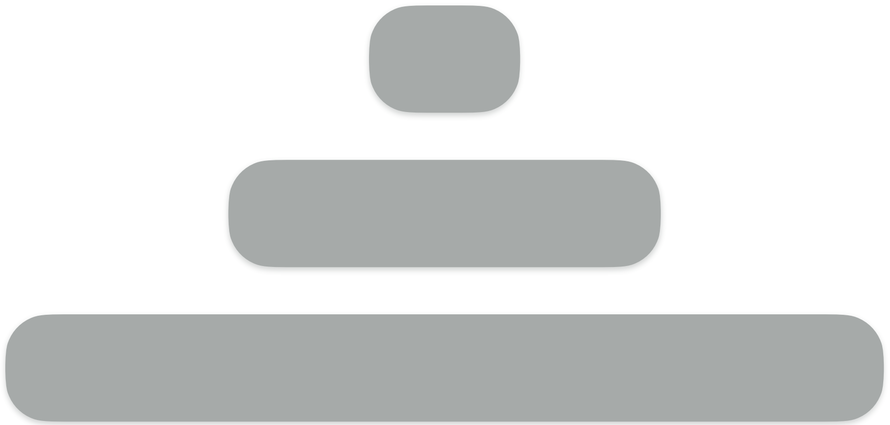
bits/entry

M

M

M

data



Bloom filters

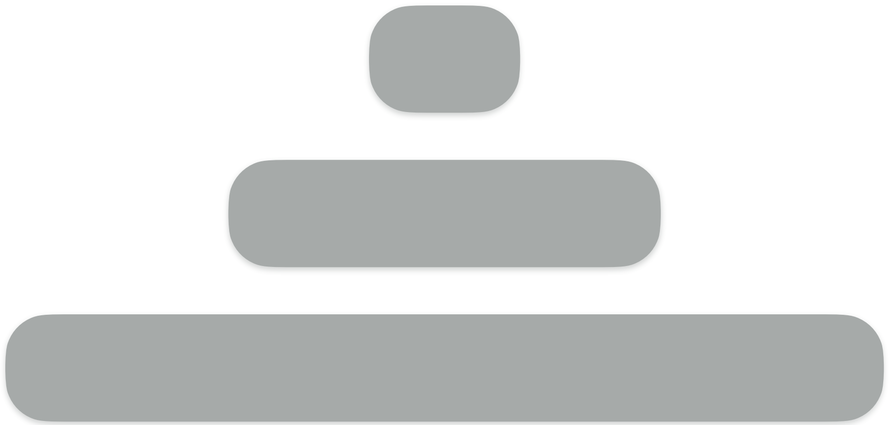
false positive rate

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

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

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

data



Bloom filters

false positive rate

2^{-M}

2^{-M}

2^{-M}

Bloom filters

false positive rate

2^{-M}

2^{-M}

2^{-M}

$= O(2^{-M} \cdot \log_R N)$

Bloom filters

false positive rate

2^{-M}

2^{-M}

2^{-M}

$= O(\mathbf{2^{-M} \cdot \log_R N})$

Bloom filters

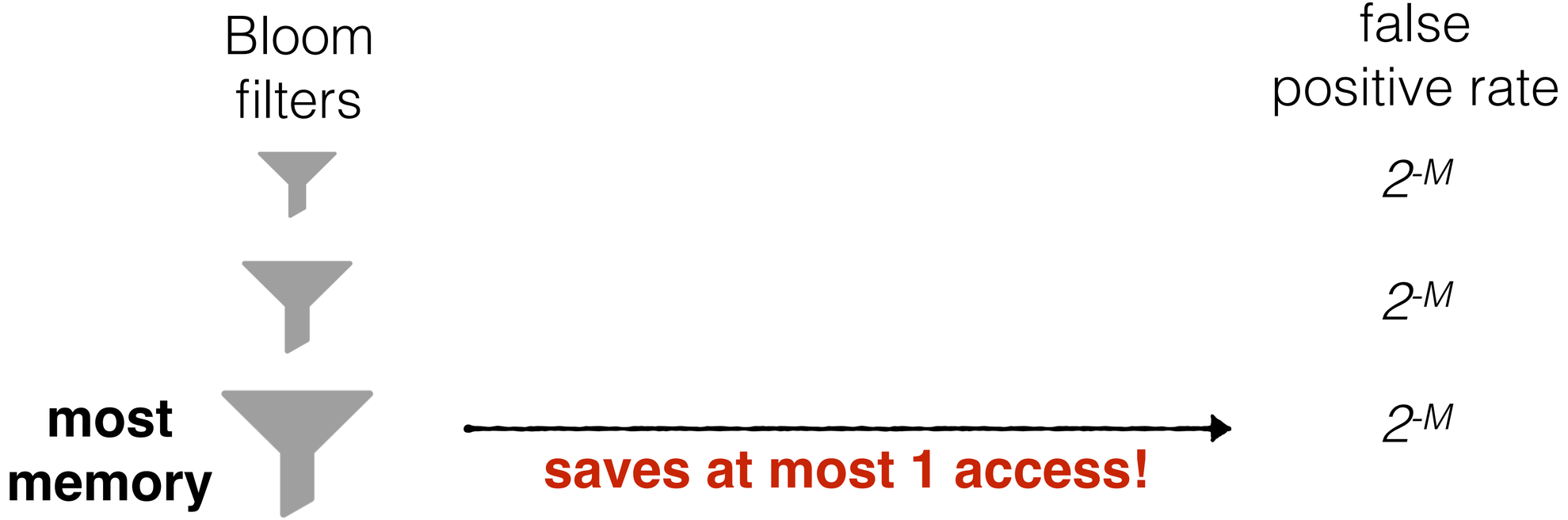
false positive rate

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

$2^{-(M+2)}$ ↓

$2^{-(M+1)}$ ↓

$2^{-(M-1)}$ ↑

Design Principle: The optimal false positive rate for a run's filter is proportional to the run's size.


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


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


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

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

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

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

geometric progression

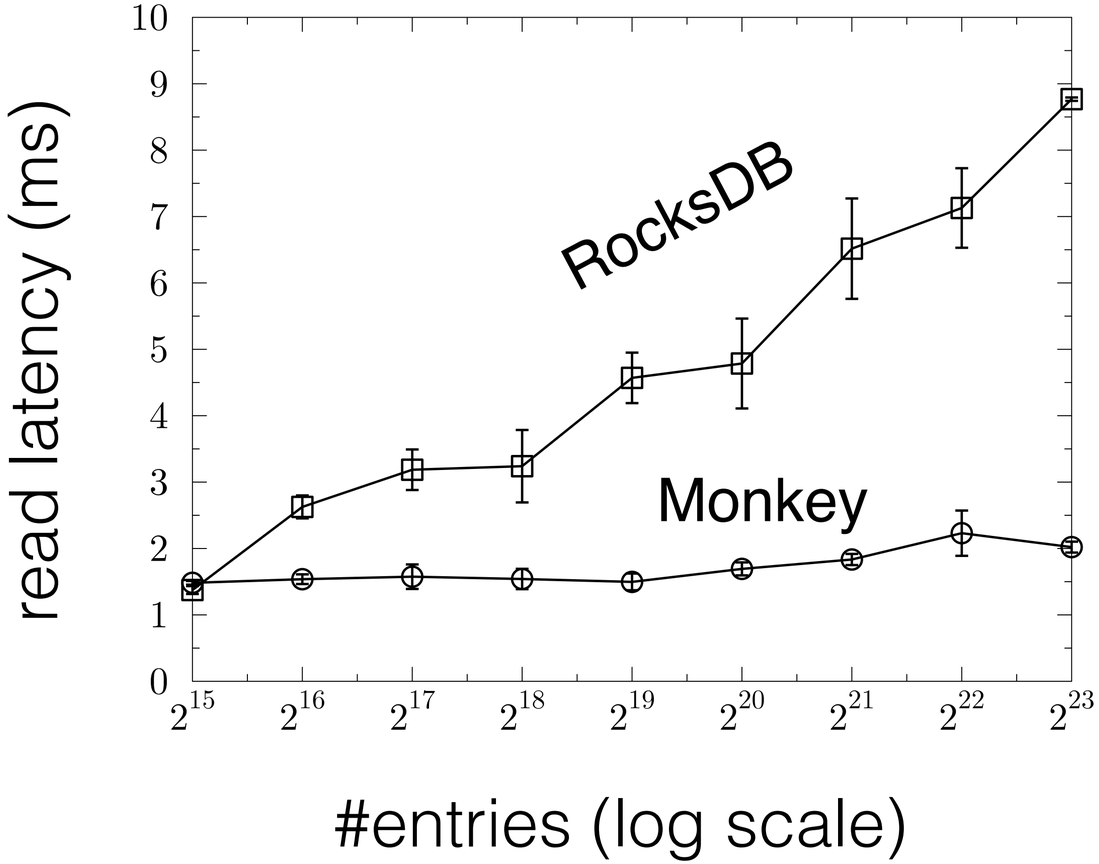
$= \quad O(2^{-M})$

Faster worst case

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

Configuration

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



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

$O(2^{-M})$

storage reads

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

$O(2^{-M})$

storage writes

$O(\log N)$

$O(\ ?\ )$

$O(\ ?\ )$

CPU reads

$O(\log N)$

$O(\ ?\ )$

Dostoevsky

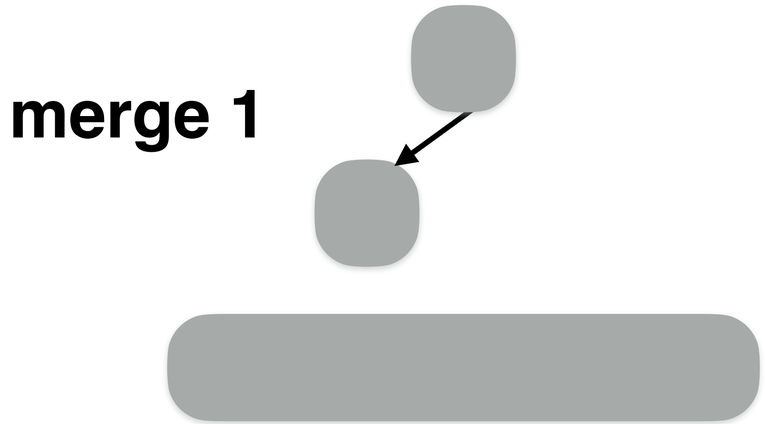
SIGMOD18

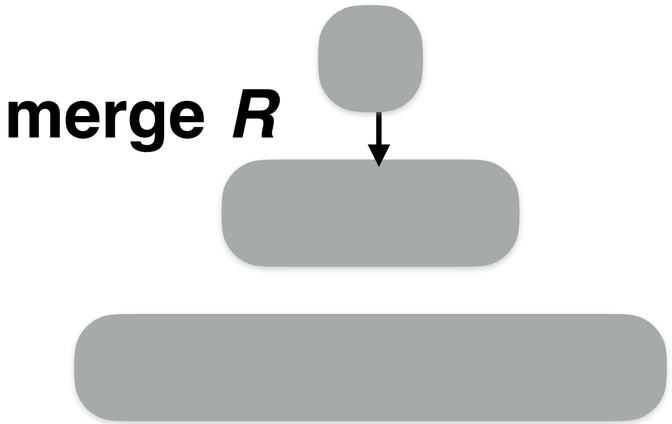
Dostoevsky: **S**pace-**T**ime **O**ptimized **Ev**olvable **S**calable **K**e**y**-Value Store

write optimized

write cost breakdown

write cost breakdown




merge *R*

writes

O(***R***)

O(***R***)

O(***R***)

O(*R* · log$_R$ *N*)

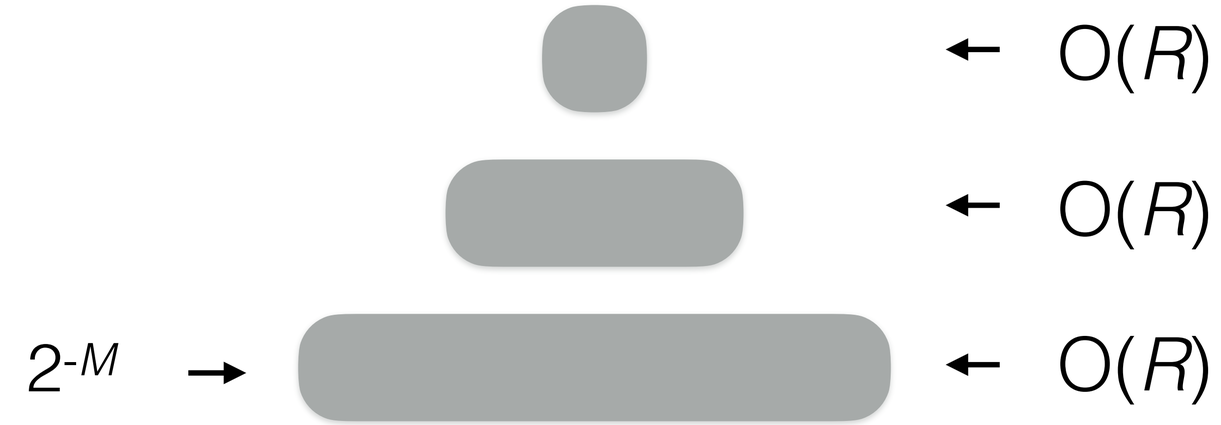
reads		writes
$O(2^{-M})$		$O(R \cdot \log_R N)$
=		=
$2^{-M}/R^2$		$O(R)$
+		+
$2^{-M}/R$		$O(R)$
+		+
2^{-M}		$O(R)$

reads

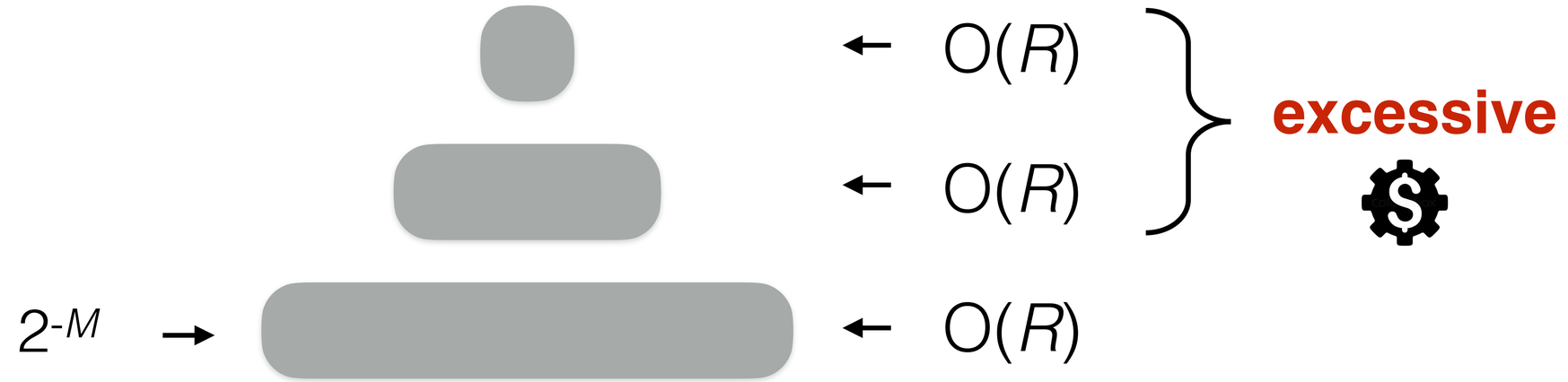
largest level

writes

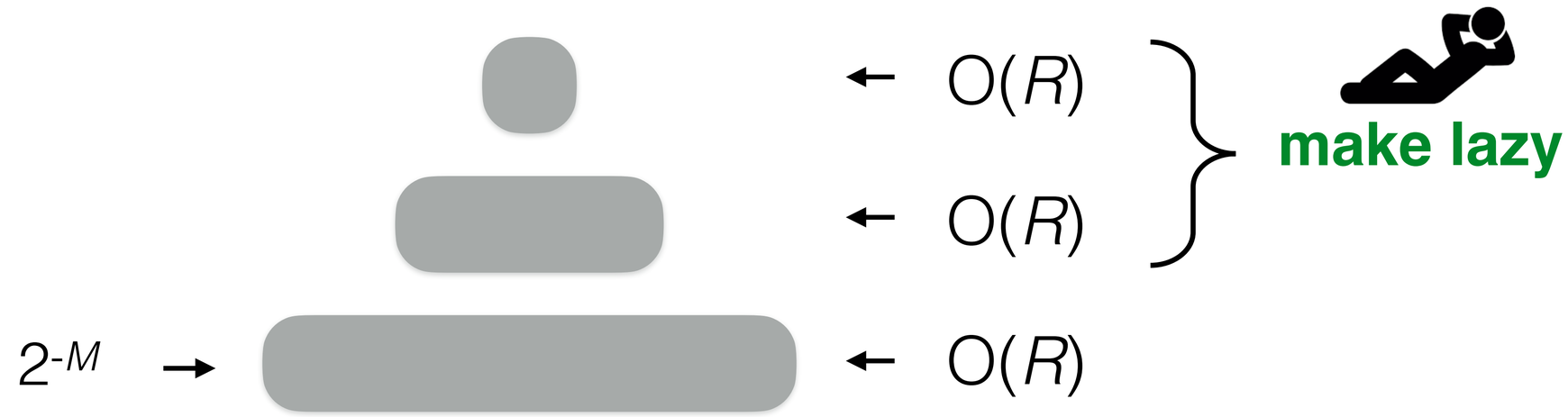
all levels



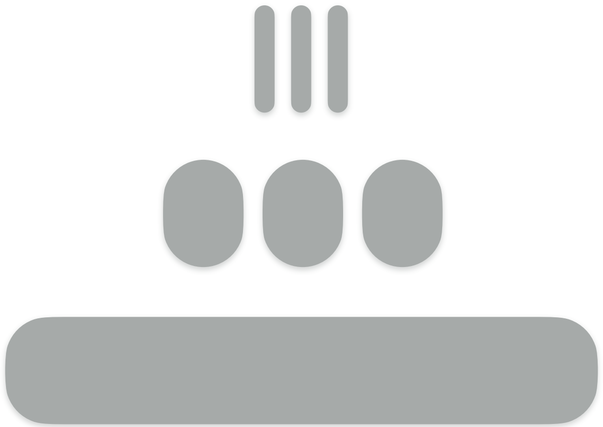
reads **writes**



reads **writes**



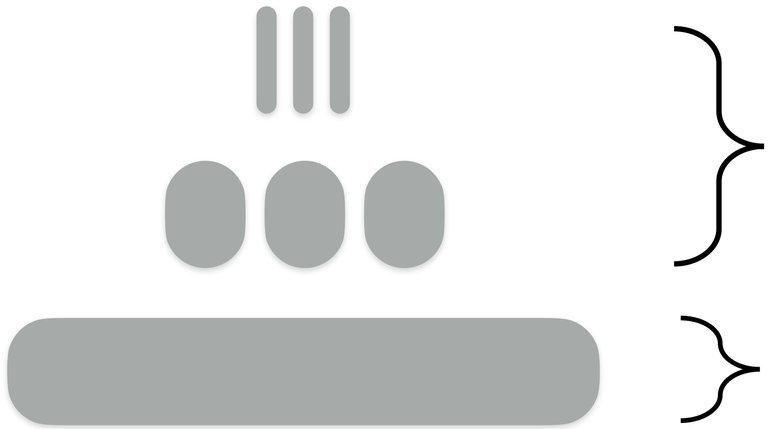
Dostoevsky



lazy

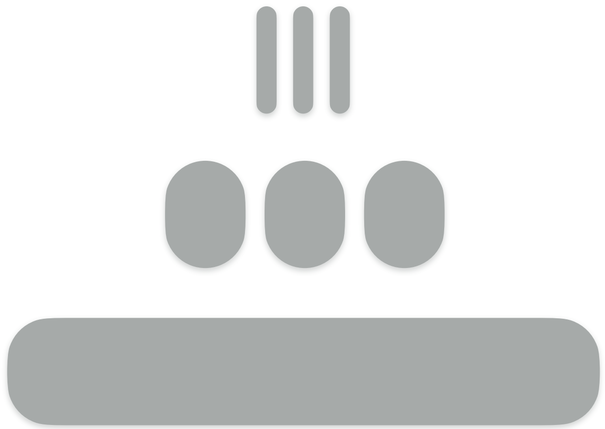
greedy

Dostoevsky



merge when
level fills up

Dostoevsky



merge when

new run comes in

reads **writes**

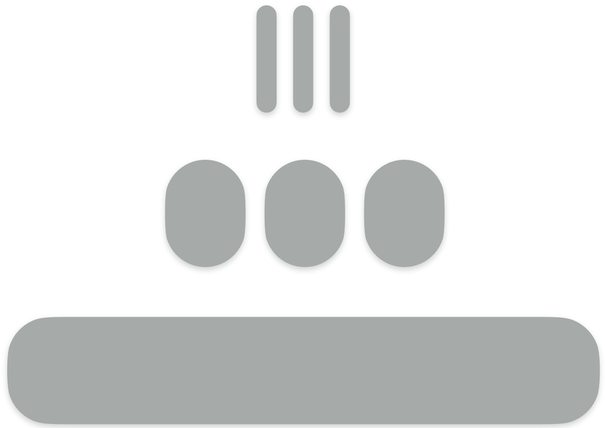
reads **writes**

false positive rates

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

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

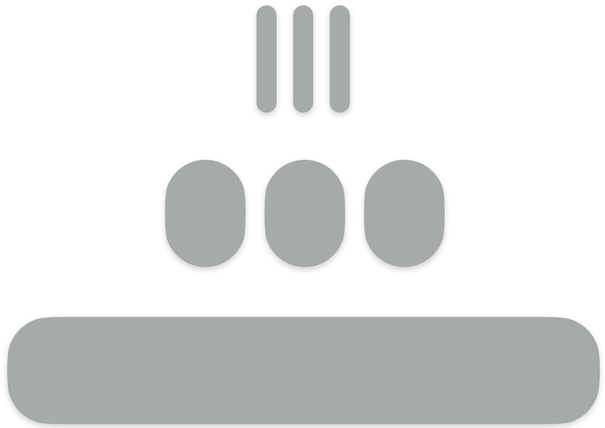
2^{-M}



reads

$O(2^{-M})$

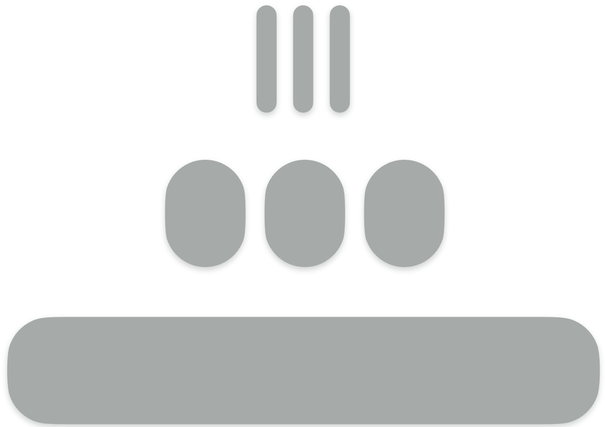
writes



reads

$O(2^{-M})$

writes



$O(\mathbf{1})$

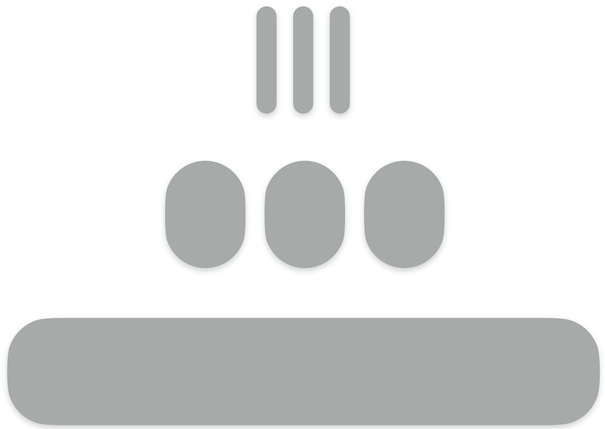
$O(\mathbf{1})$

$O(\boldsymbol{R})$

reads

$O(2^{-M})$

writes



$O(1)$

$O(1)$

$O(R)$

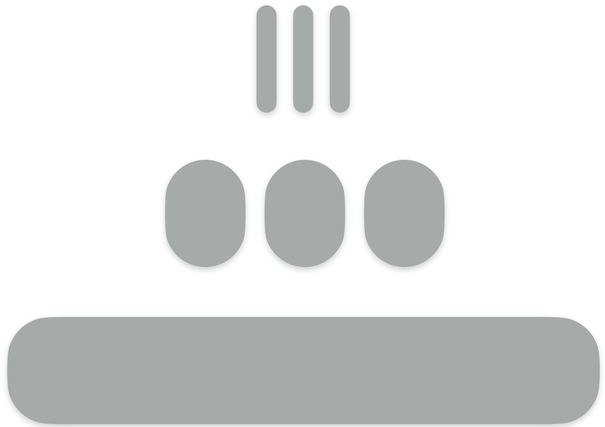
$O(\boldsymbol{R + \log_R N})$

reads

$O(e^{-M})$

writes

$O(R + \log_R N)$ **<** $O(R \cdot \log_R N)$



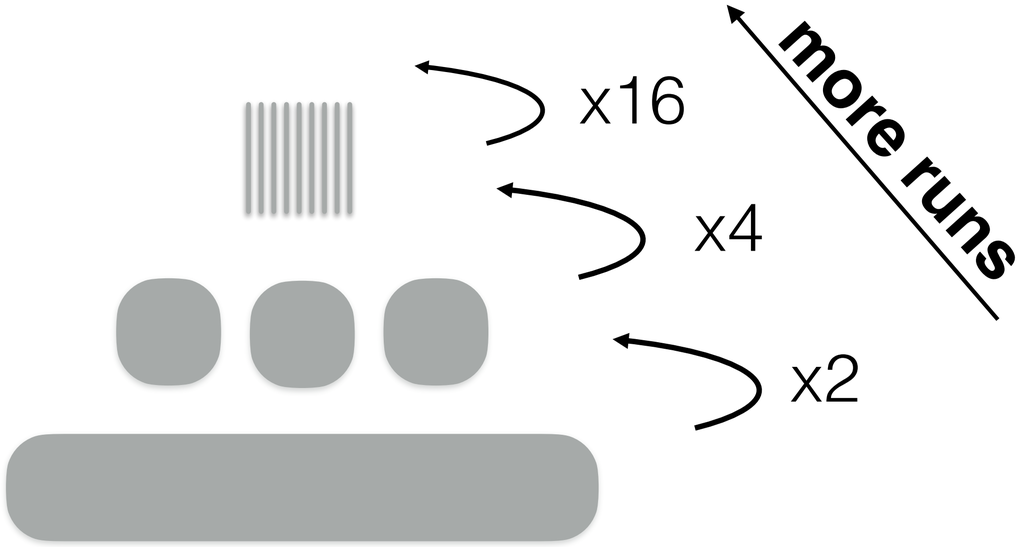
Dostoevsky

$O(R + \log_R N)$

SIGMOD 18

LSM-Bush

$O(\log_2 \log_2 N)$



SIGMOD 19

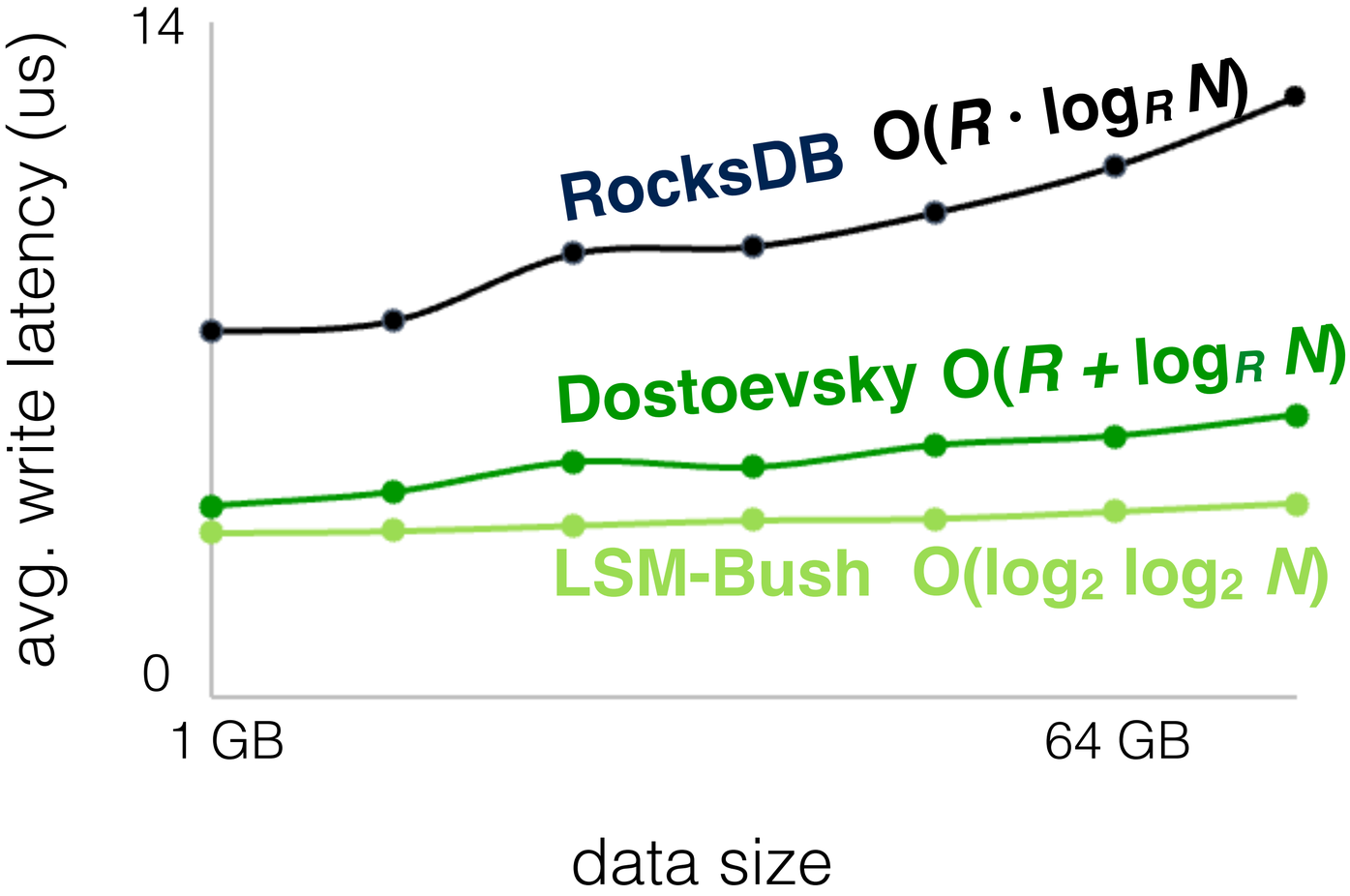
Configuration

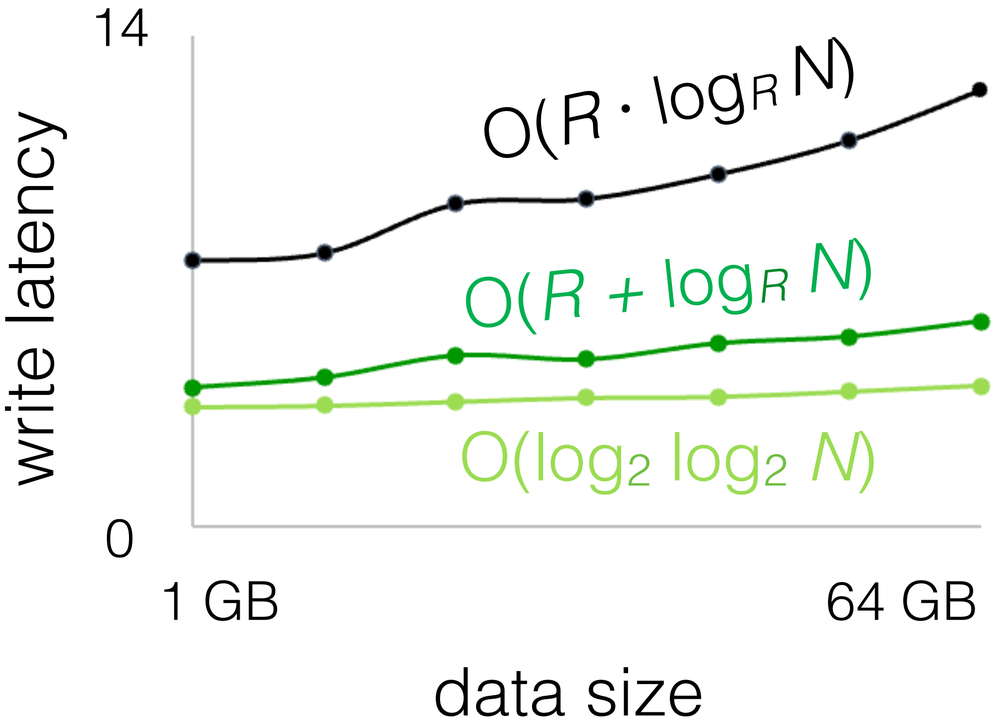
buffer 2MB

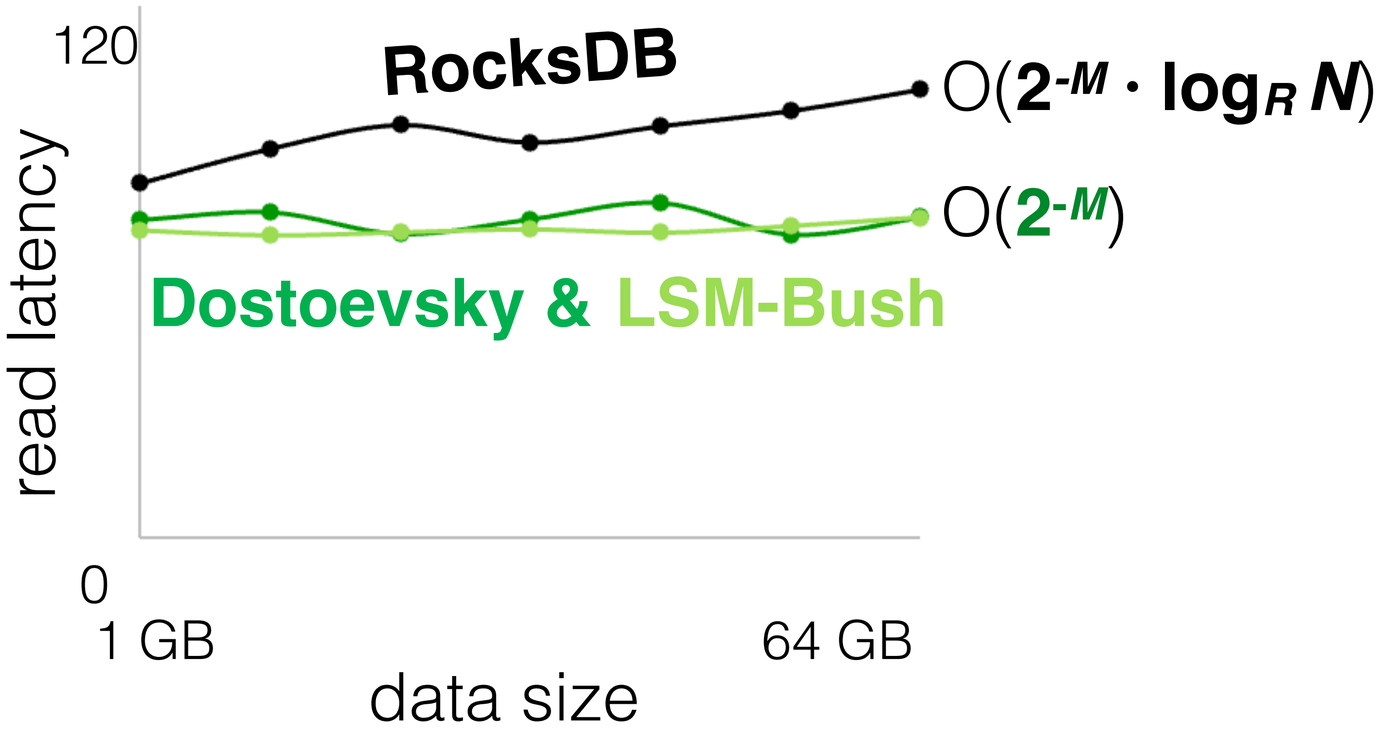
size ratio: 5

1KB entries

SSD storage




14
0
write latency
O(R · log_R N)
O(R + log_R N)
O(log_2 log_2 N)
1 GB
64 GB
data size


120
0
read latency
RocksDB
O(2^-M · log_R N)
O(2^-M)
Dostoevsky & LSM-Bush
1 GB
64 GB
data size

storage reads

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

$O(e^{-M})$

storage writes

$O(R \cdot \log_R N)$

$O(R + \log_R N)$

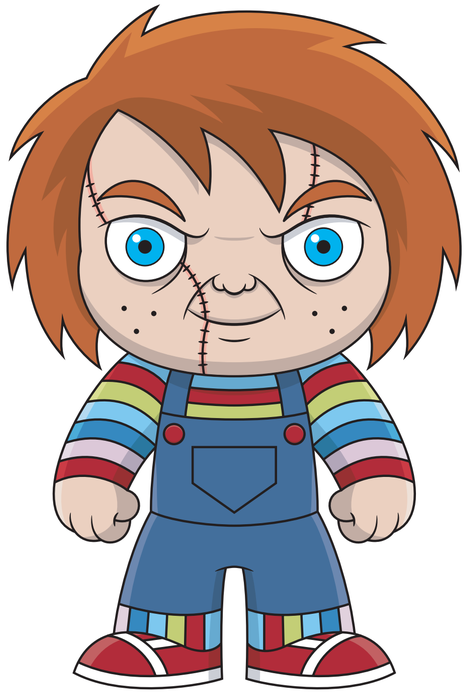
$O(\log_2 \log_2 N)$

CPU reads

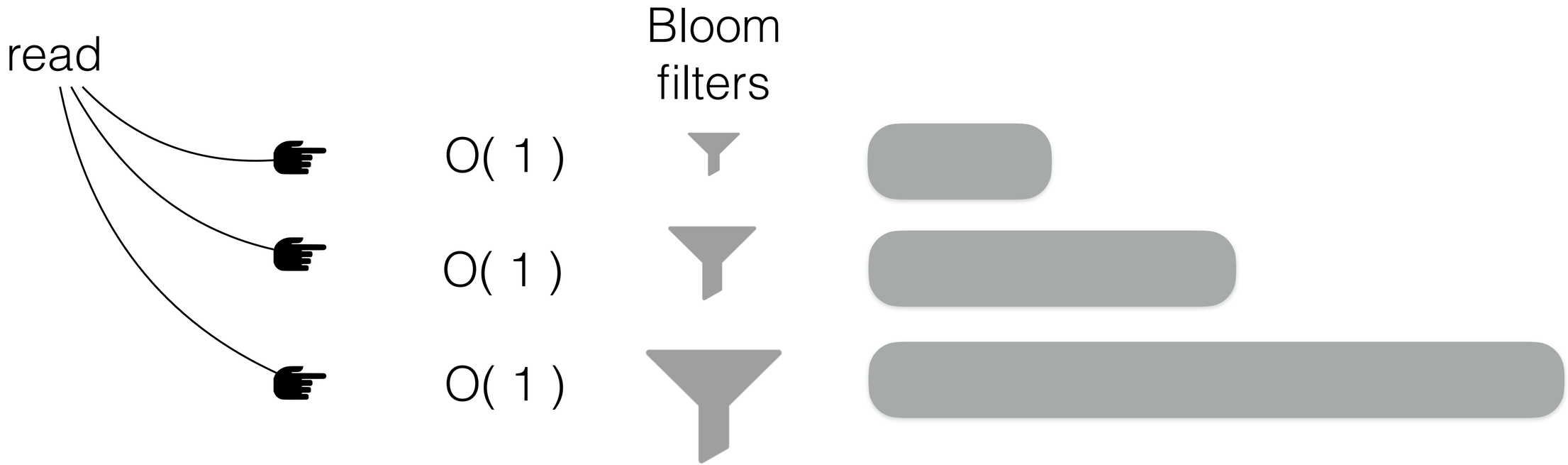
$O(\log_R N)$

$O(?)$

Chucky: Succinct Cuckoo Filter for LSM-Tree



SIGMOD 2021

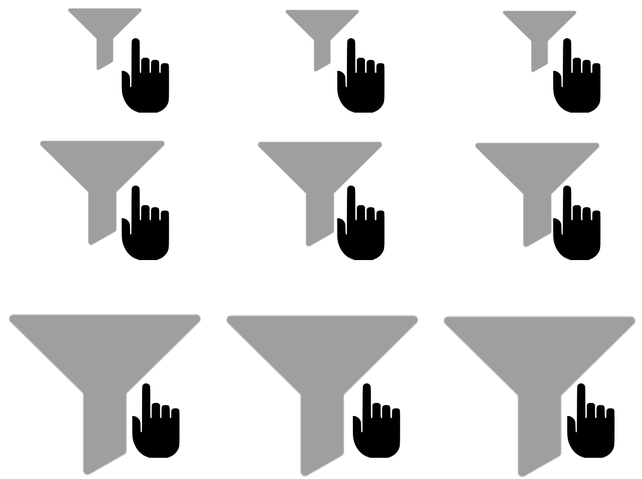


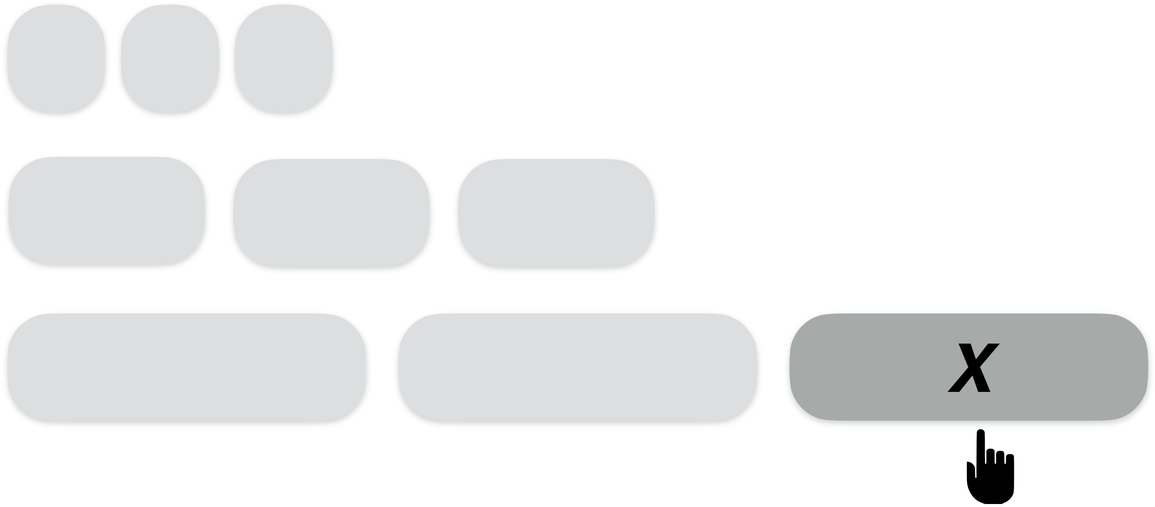
= O($\log_R N$)

O(*R*)

O(*R*)

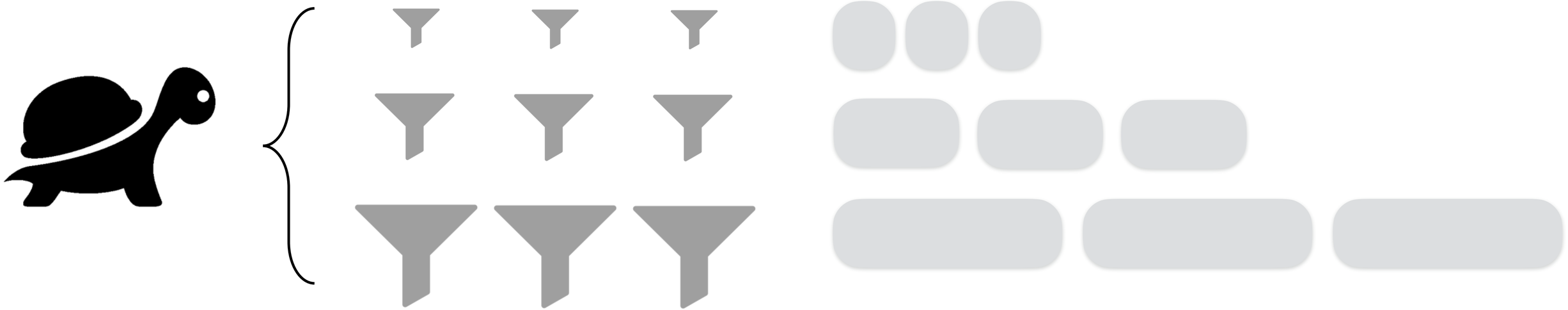
O(*R*)

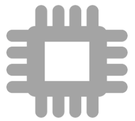




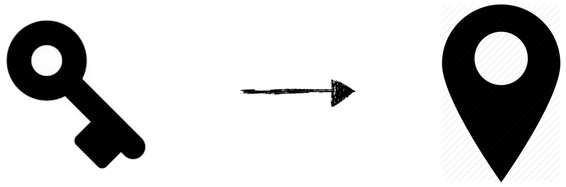
= O(*R* · $\log_R$ *N*)

Problem: Reduce filter accesses



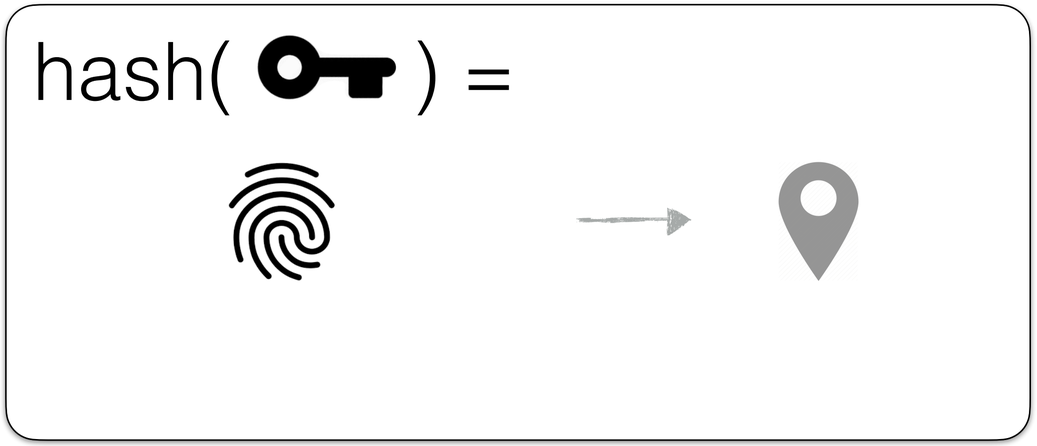


hash table



key **Level ID**

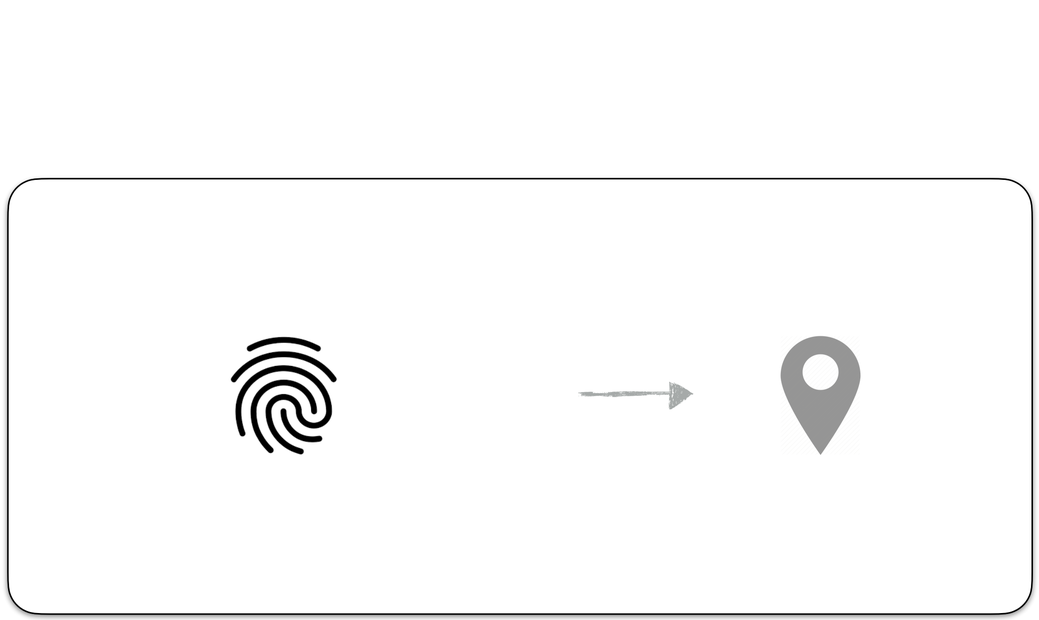
hash() =

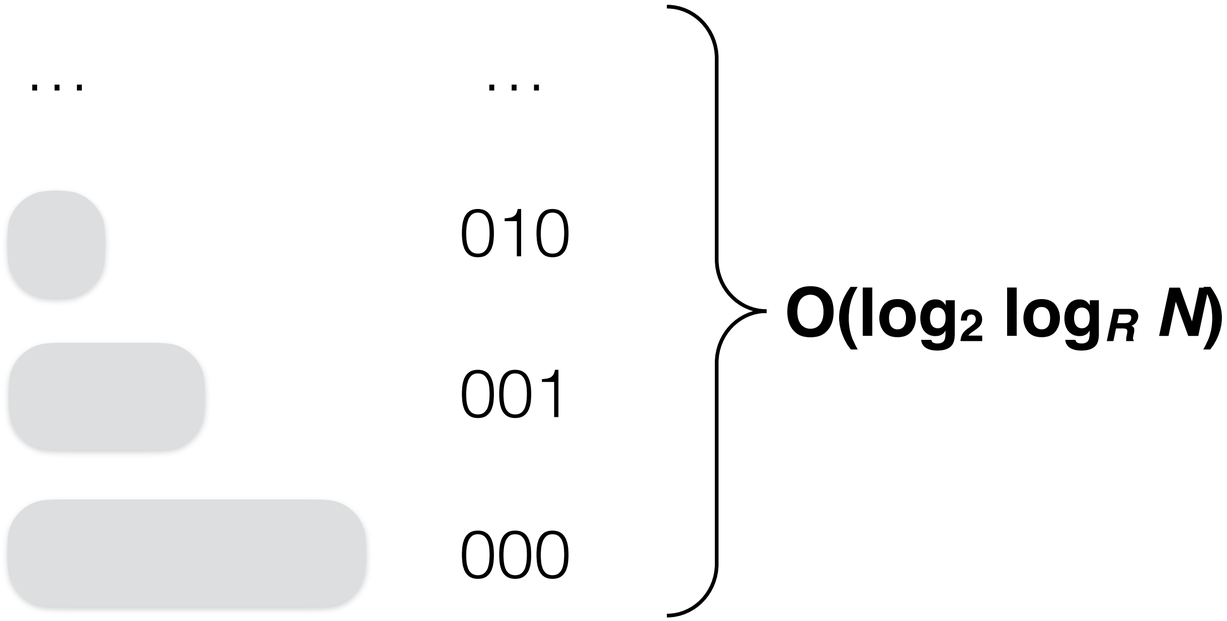


cuckoo filter

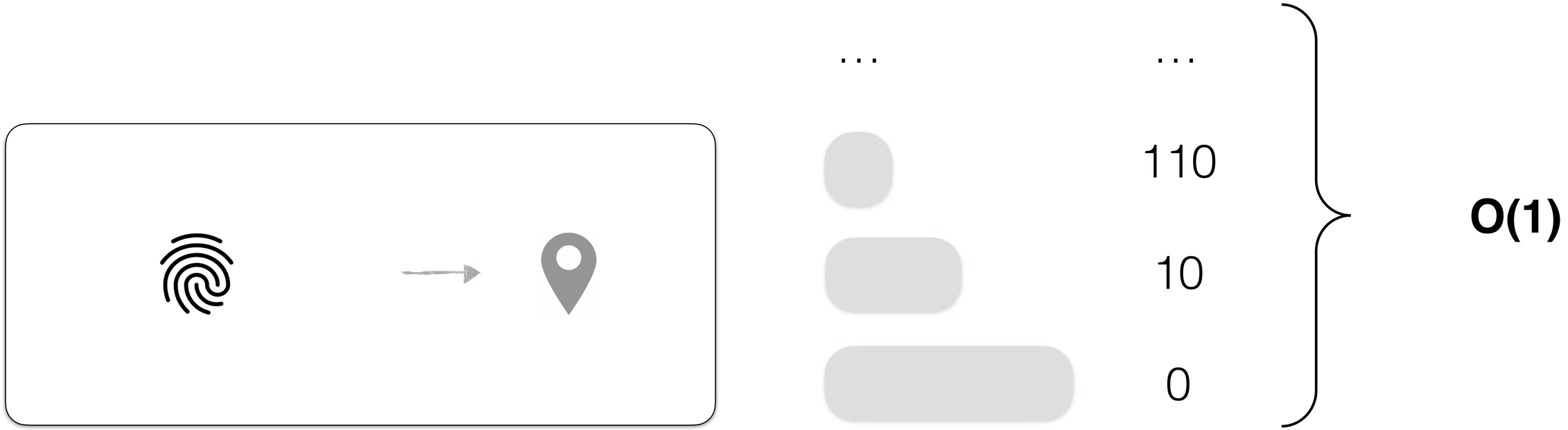
$O(\log_2 \log_R M)$

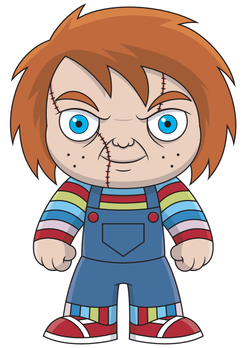
Binary encoding





e.g., unary encoding





	Monkey		Chucky
Bits / entry	**O(1)**	**≈**	**O(1)**
memory accesses	**$O(\log_R N)$**	**>**	**O(1)**

storage reads

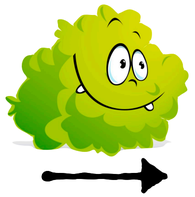
$O(2^{-M} \cdot \log_R N)$

$O(2^{-M})$

storage writes

$O(R \cdot \log_R N)$

$O(R + \log_R N)$



$O(\log_2 \log_2 N)$

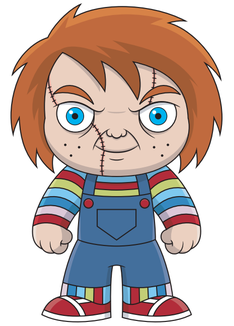
CPU reads

$O(\mathbf{\log_R N})$

$O(\mathbf{1})$

Our storage engines
can better handle exponential data growth



Our storage engines
can better handle exponential data growth

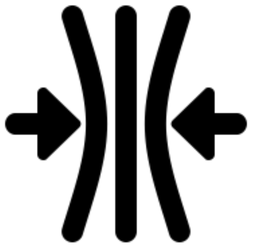



Many open problems

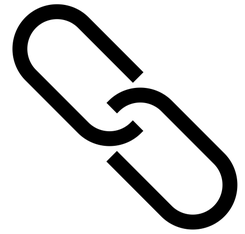
open problems

range filtering

compression



key-value separation



Thanks!