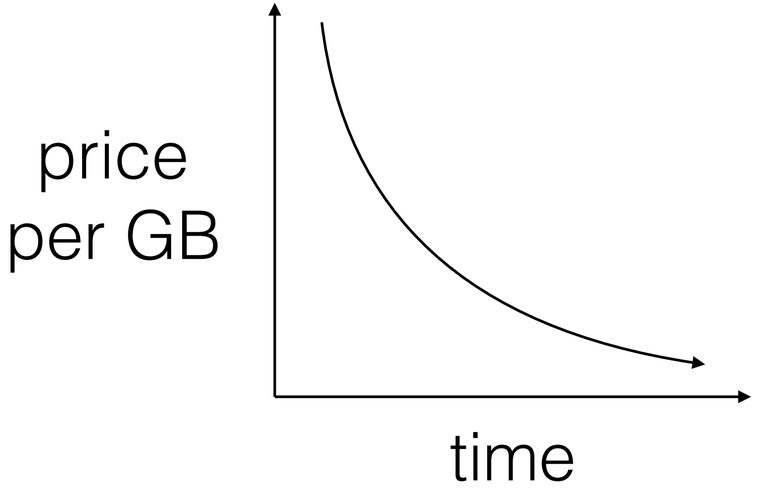
A Self-Designing Key-Value Store

Niv Dayan, Wilson Qin, Manos Athanassoulis, Stratos Idreos

http://daslab.seas.harvard.edu/crimsondb/

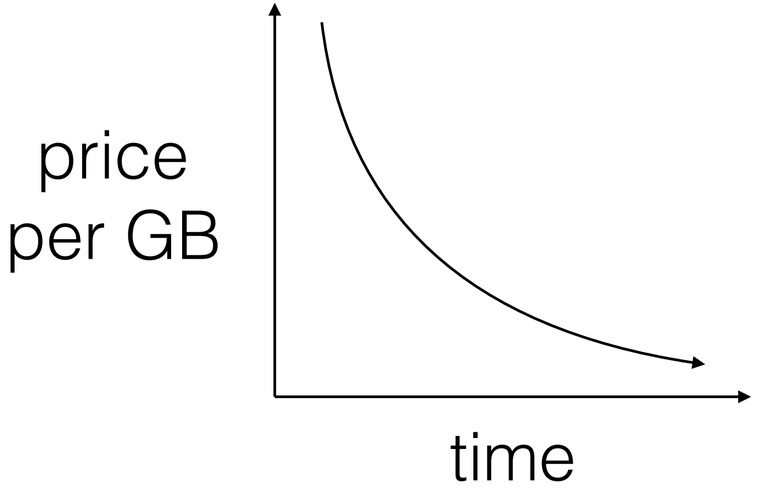
DASlab @ Harvard SEAS

storage is cheaper



$$\frac{\text{inserts \& updates}}{\text{workload}} \uparrow$$

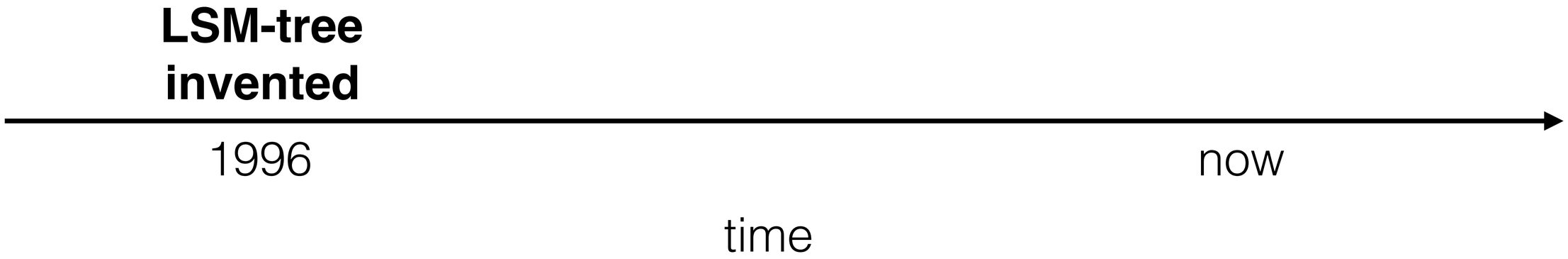
storage is cheaper



$$\frac{\text{inserts \& updates}}{\text{workload}} \uparrow$$

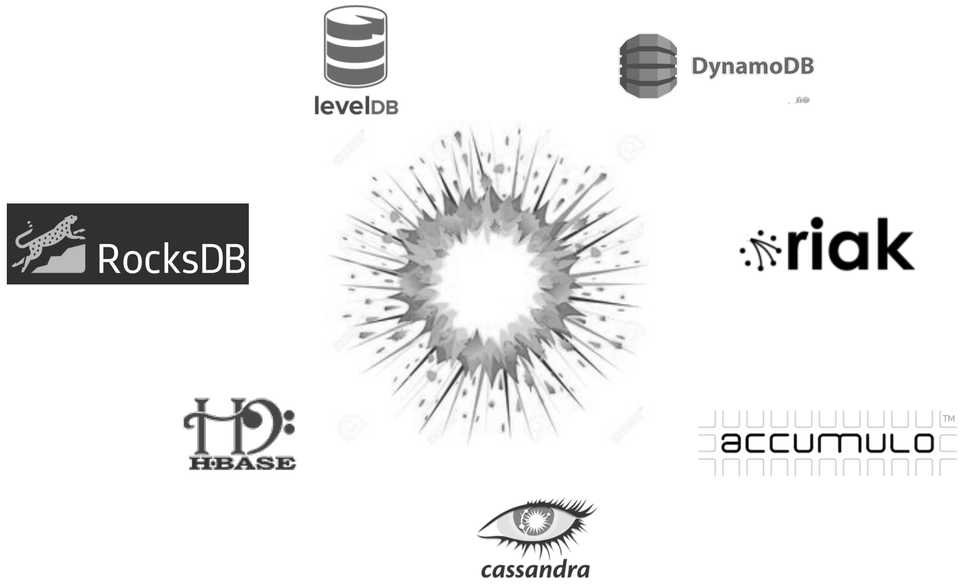
need for write-optimized database structures

need for write-optimized database structures



need for write-optimized database structures

Key-Value Stores

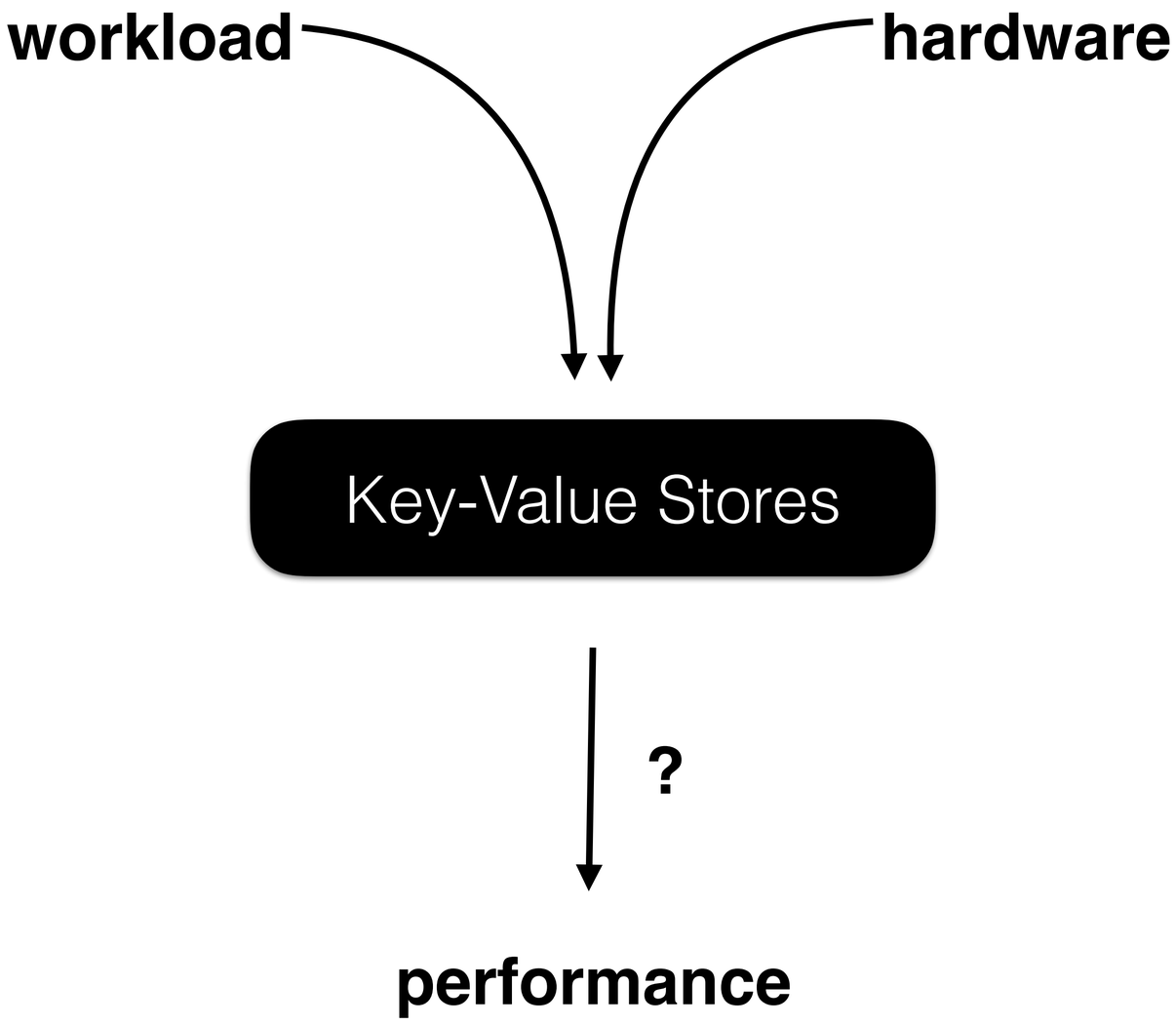


LSM-tree invented

1996 → now

time

Key-Value Stores


workload
hardware
Key-Value Stores
?
performance

workload

hardware

Key-Value Stores

?

performance

workload

hardware

Suboptimal Design

?

performance

CrimsonDB

Map Trade-Offs

Navigate

What-if?

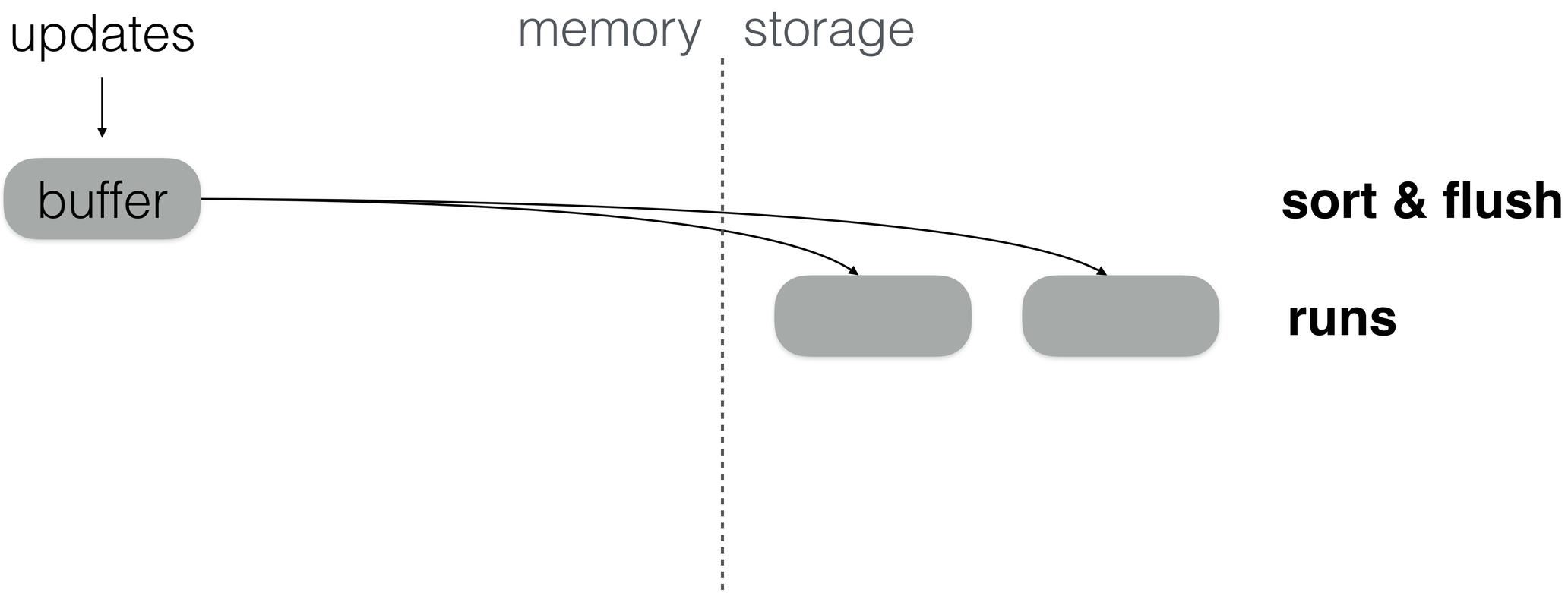
http://daslab.seas.harvard.edu/crimsondb/

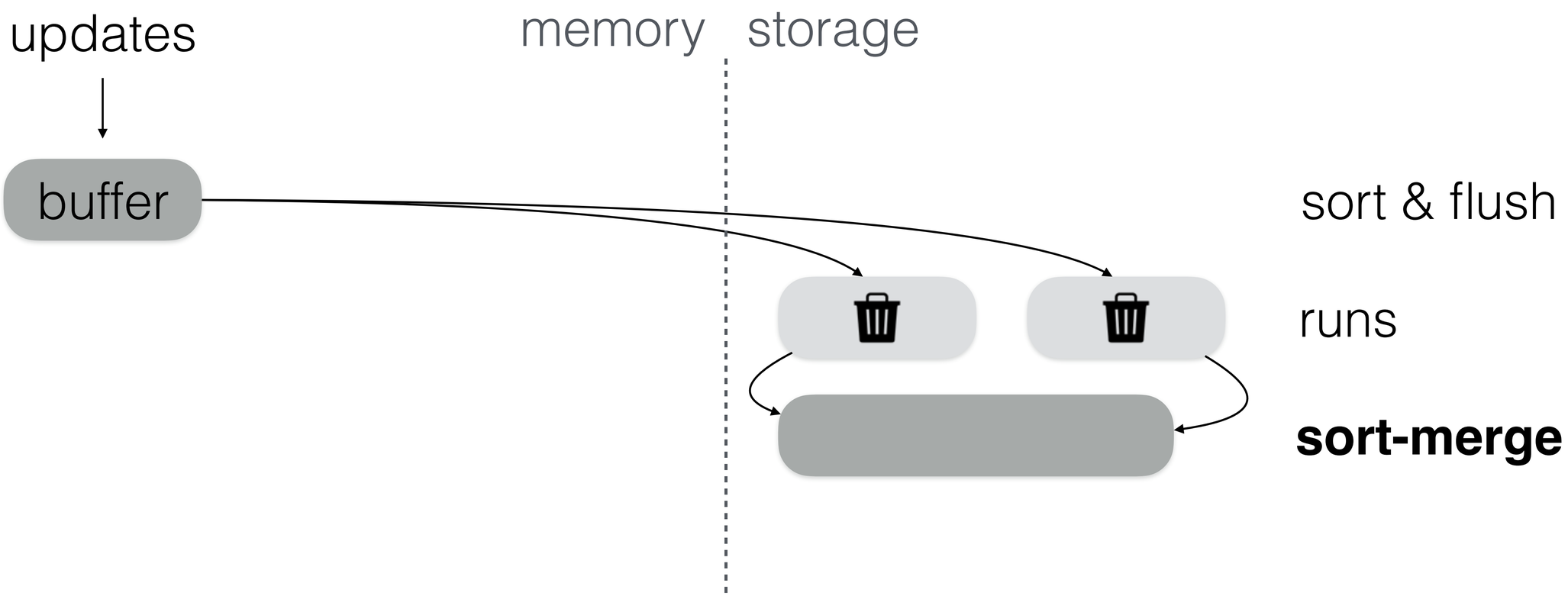
DASlab @ Harvard SEAS

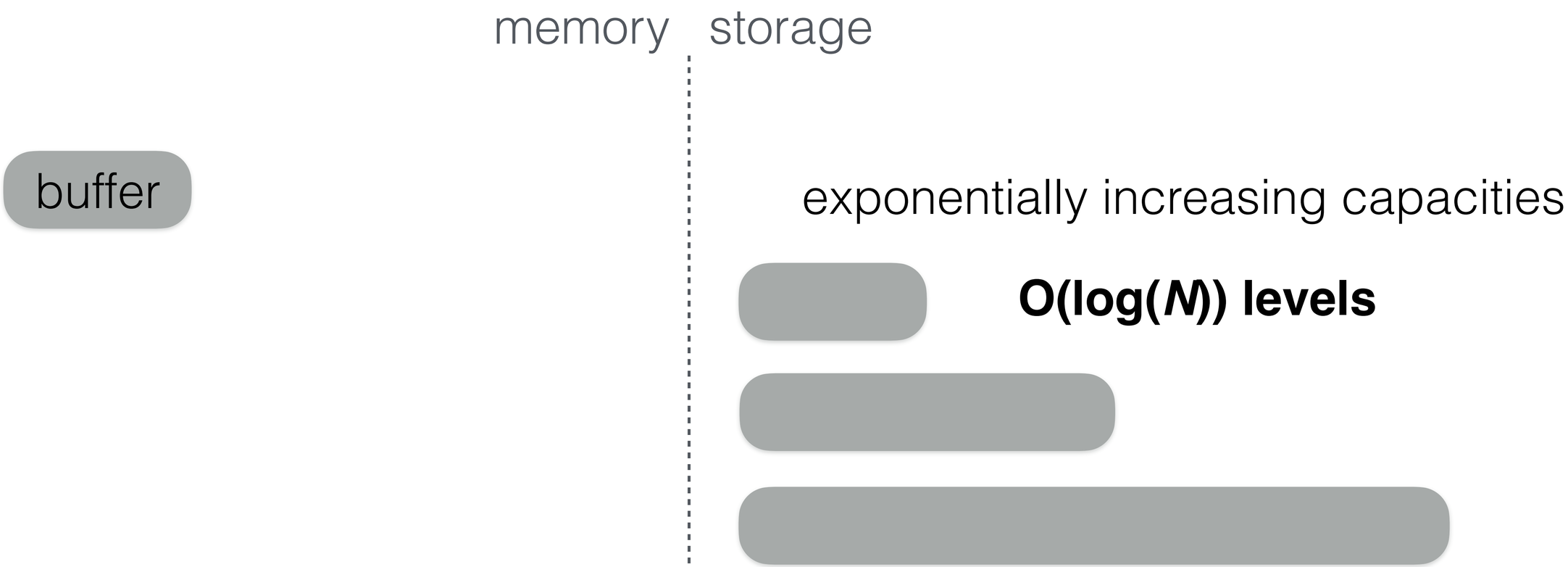
LSM-tree
Key-Value Stores

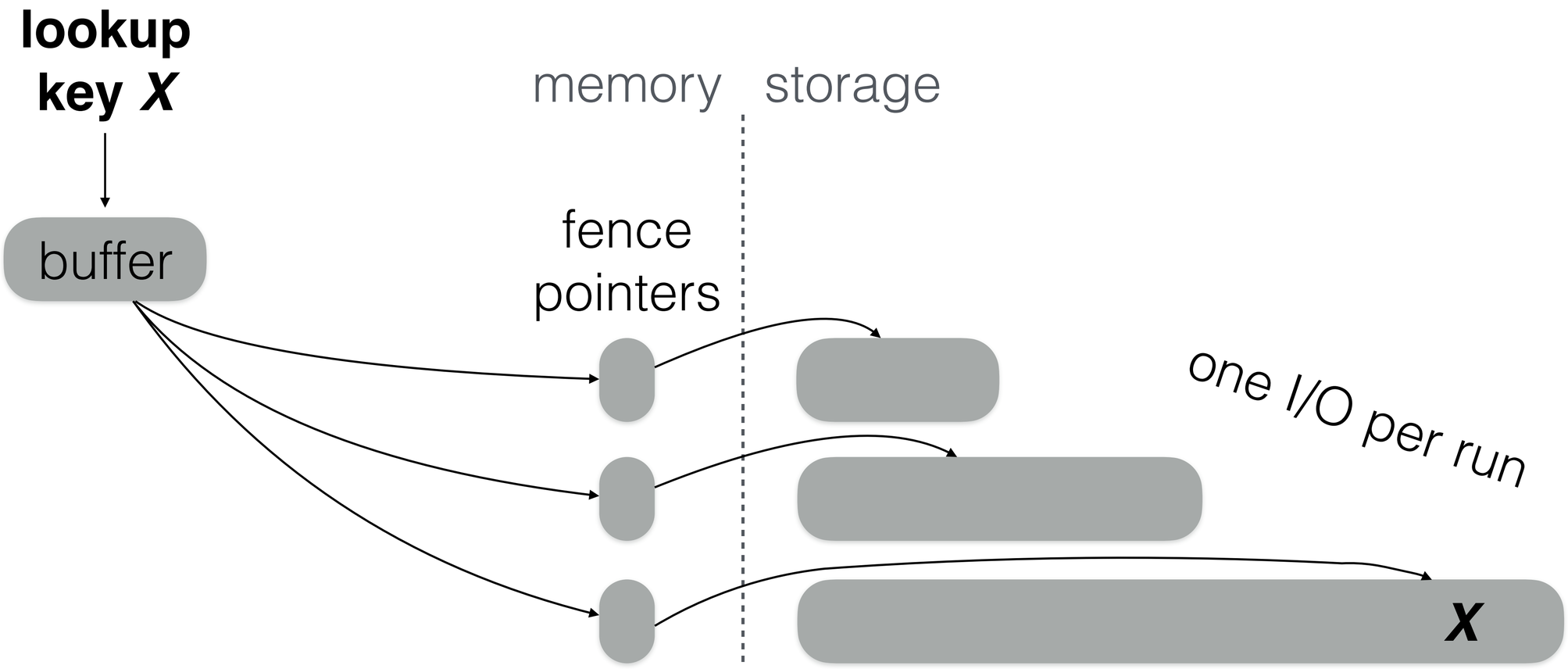
What are they really?

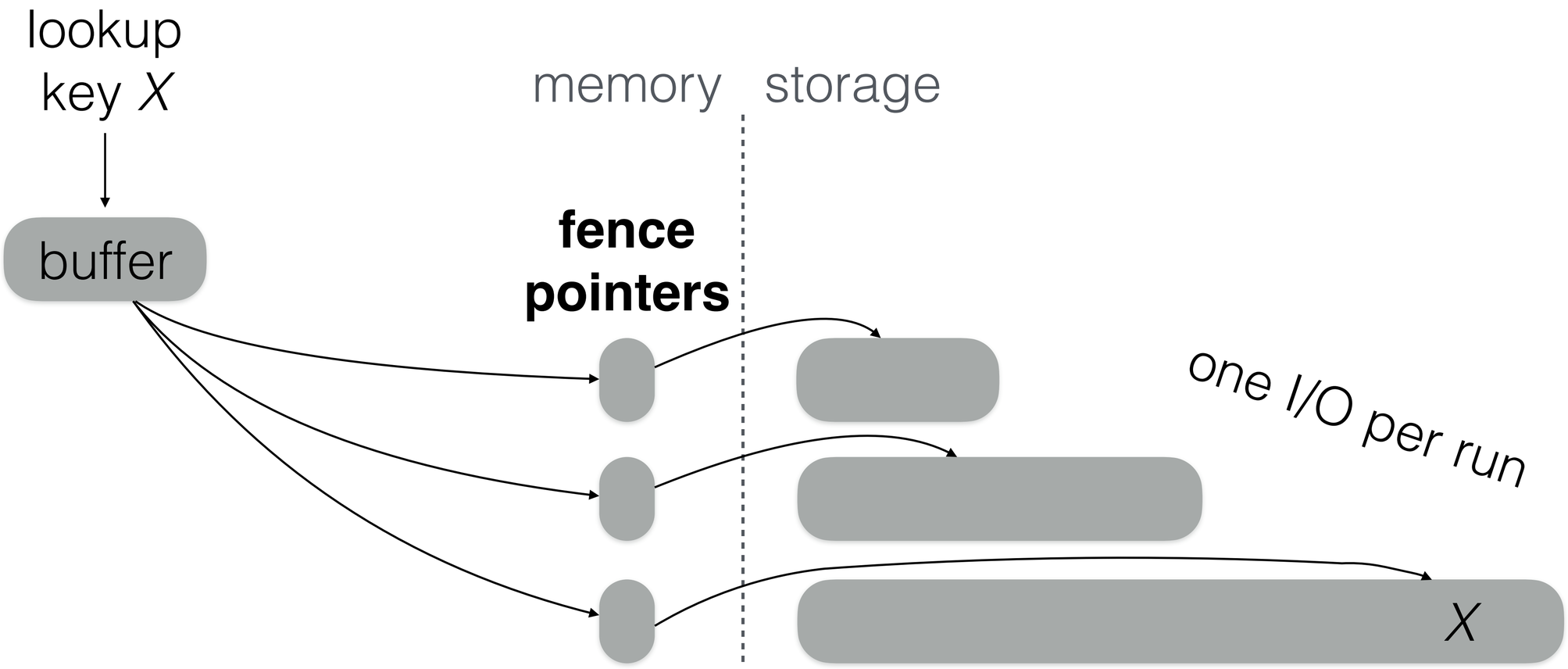

updates
buffer
memory
storage

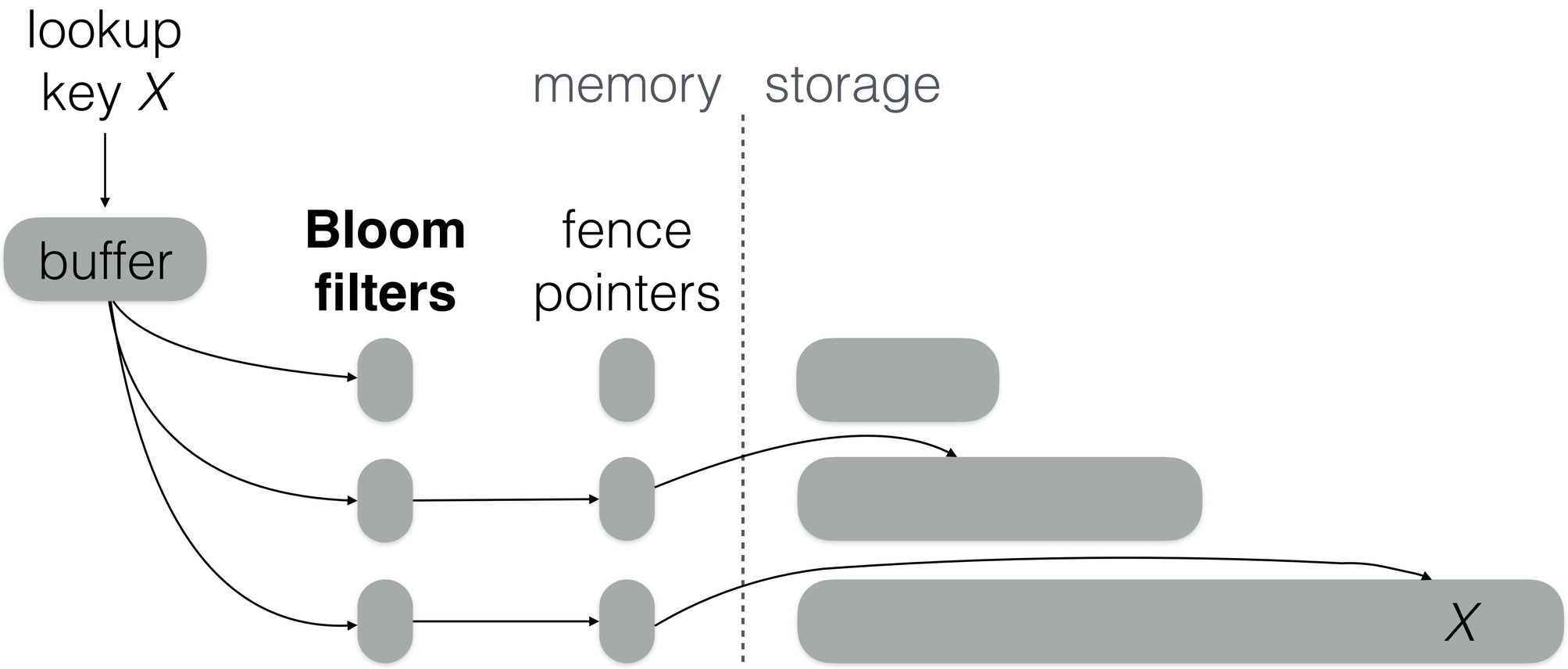

updates
memory
storage
buffer
sort & flush
runs

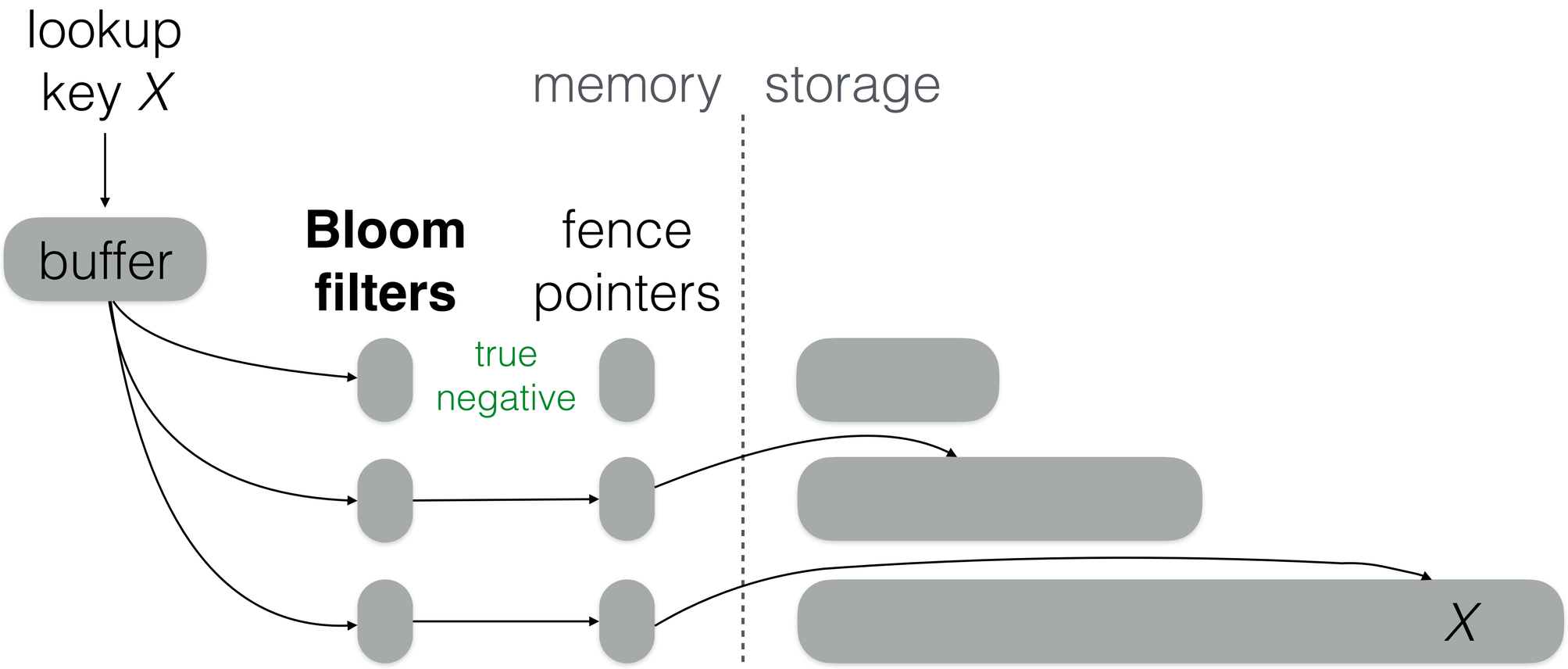

updates
memory
storage
buffer
sort & flush
runs
sort-merge

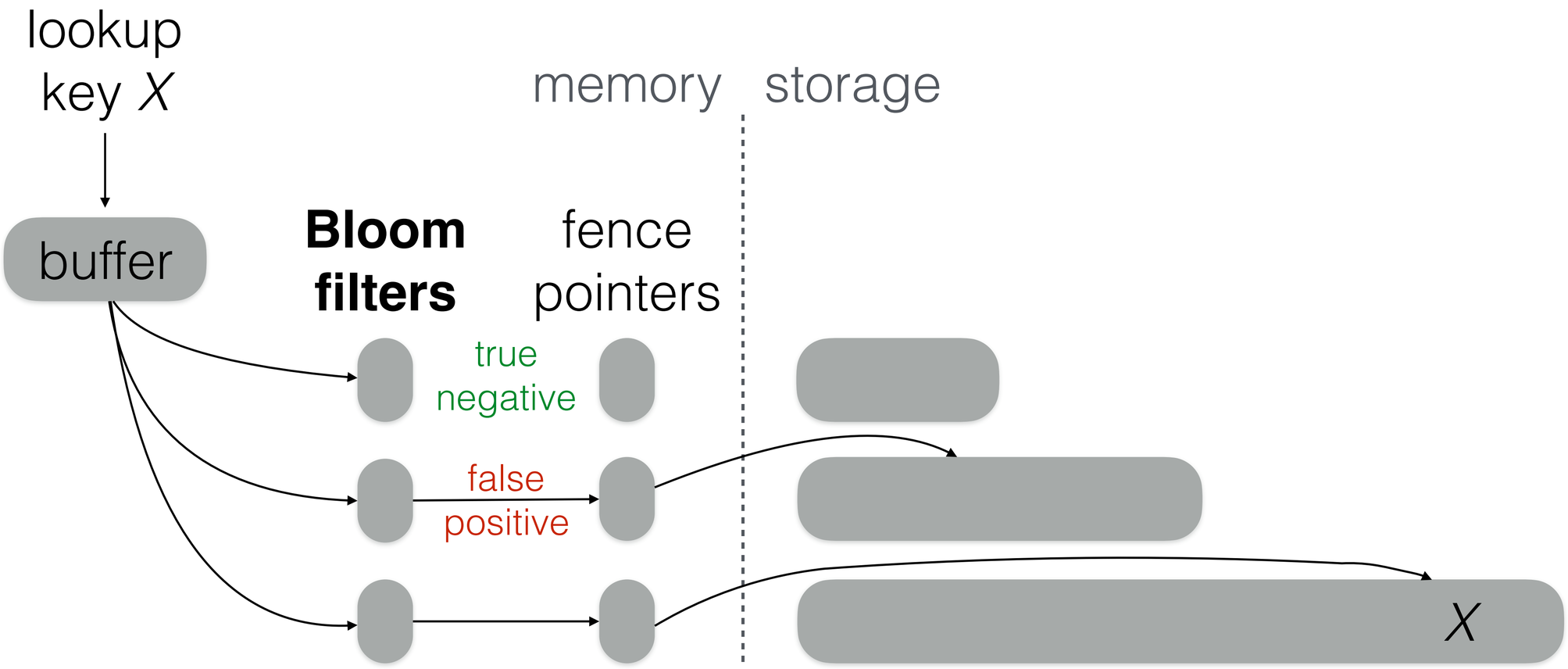

memory
storage
buffer
exponentially increasing capacities
O(log(M)) levels

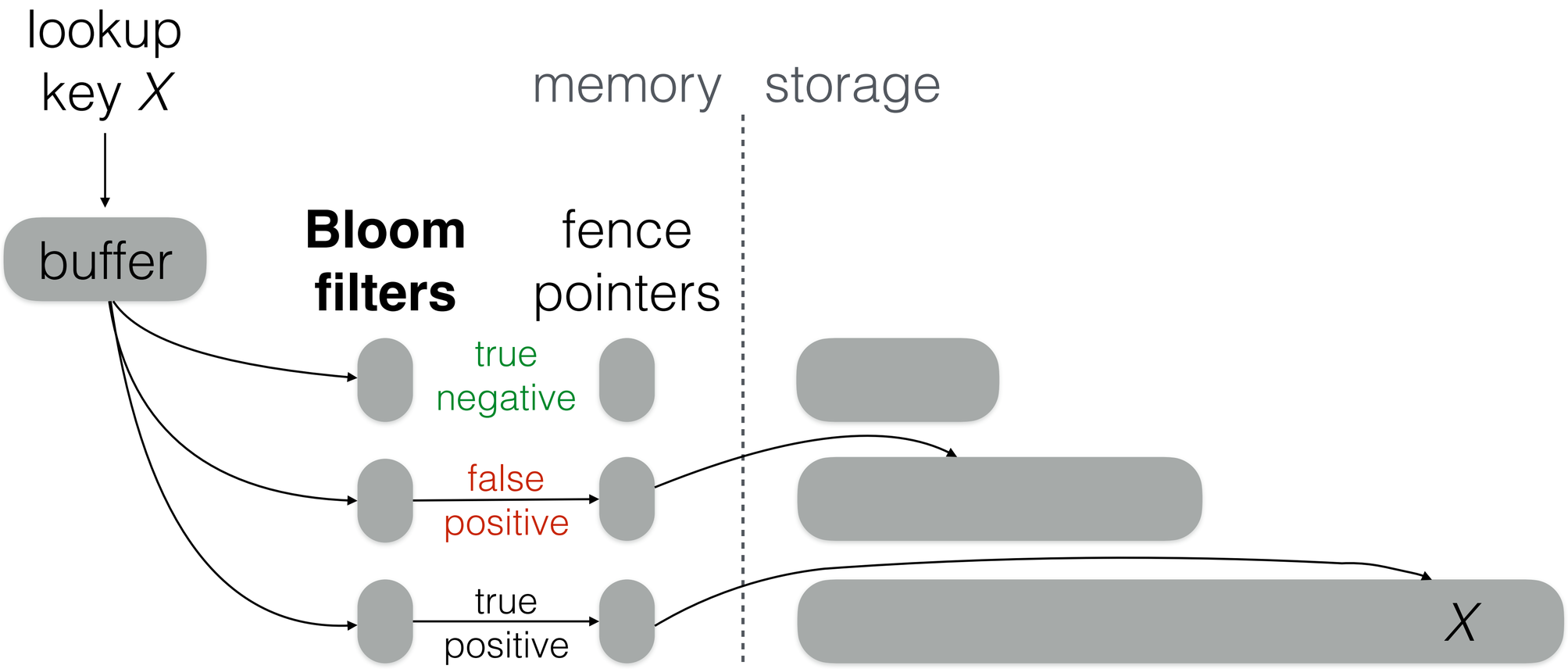

lookup
key X
buffer
memory
storage
fence
pointers
one I/O per run
X


lookup
key X
buffer
memory
storage
fence
pointers
one I/O per run
X

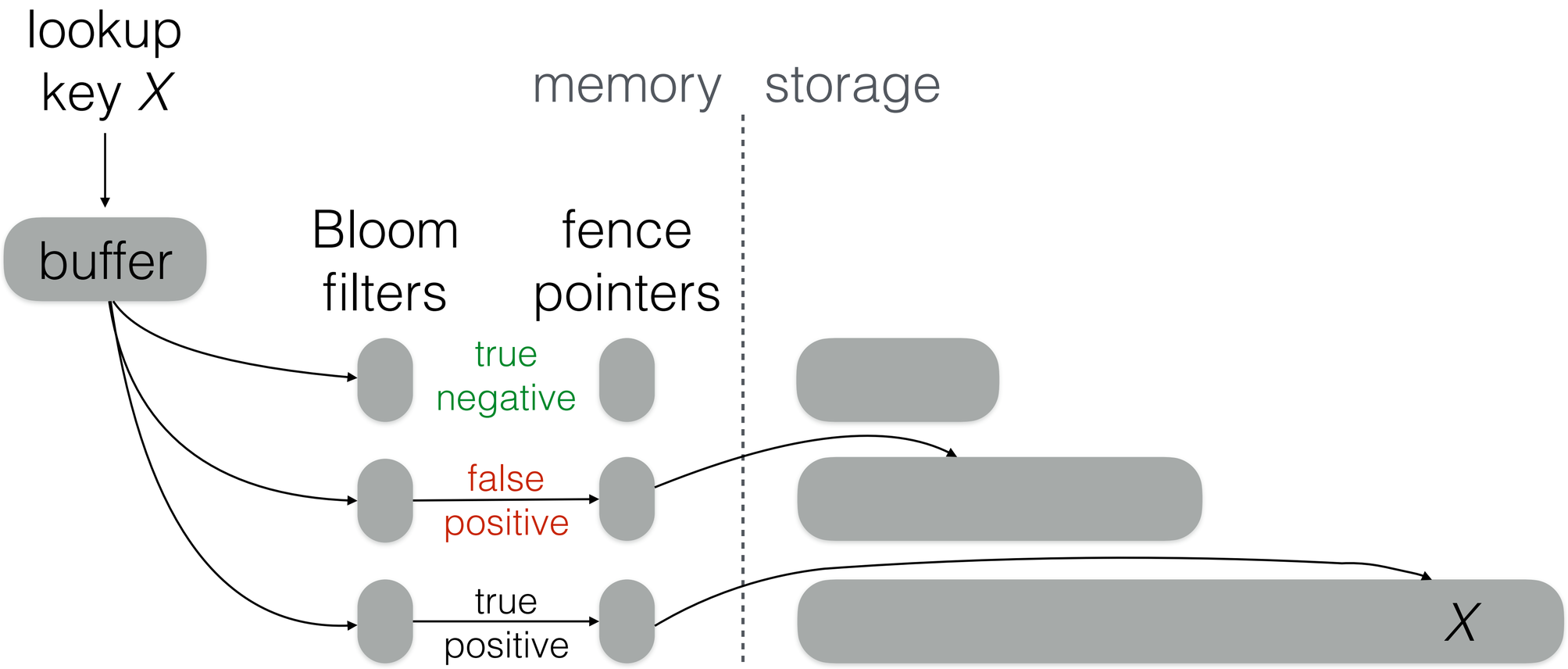

lookup
key X
memory
storage
buffer
Bloom
filters
fence
pointers
X


lookup
key X
memory
storage
buffer
Bloom
filters
fence
pointers
true
negative
X

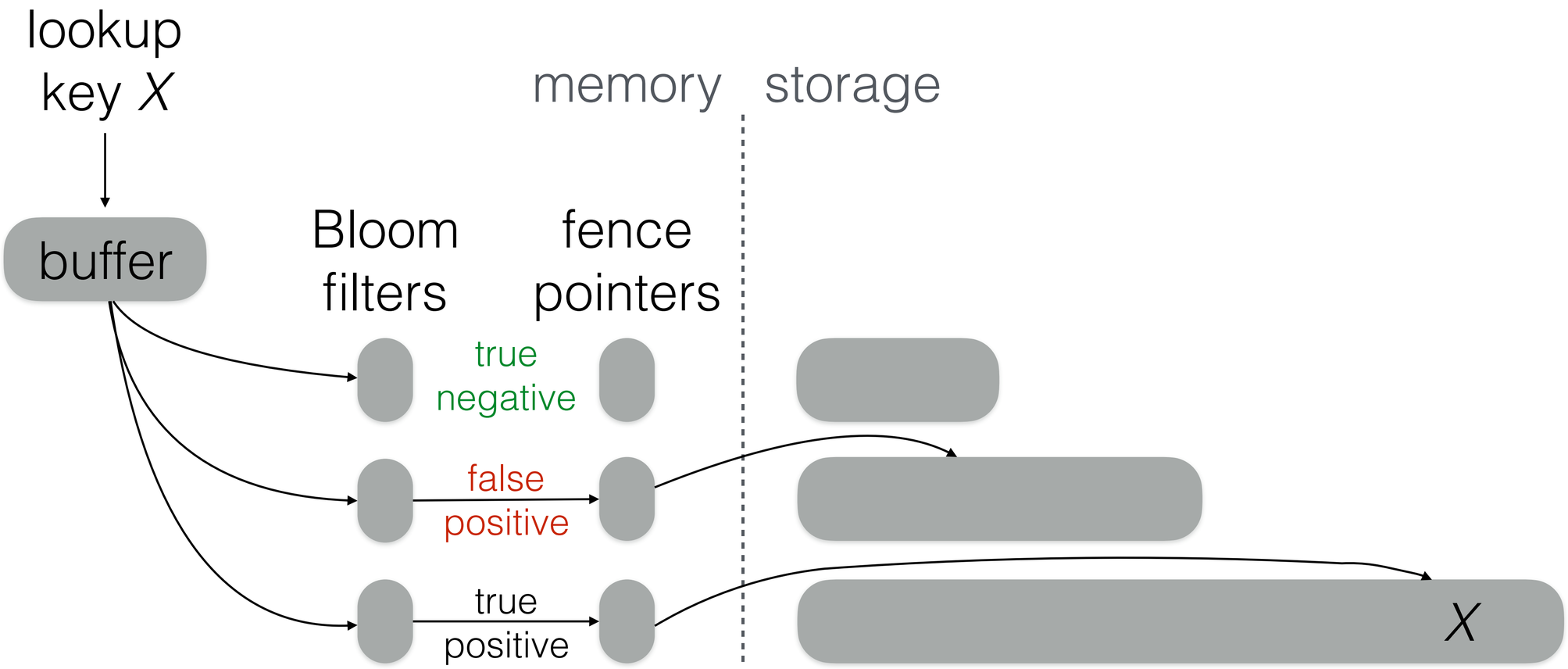

lookup
key X
memory
storage
buffer
Bloom
filters
fence
pointers
true
negative
false
positive
X


lookup
key X
memory
storage
buffer
Bloom
filters
fence
pointers
true
negative
false
positive
true
positive
X

Performance & Cost Tradeoffs

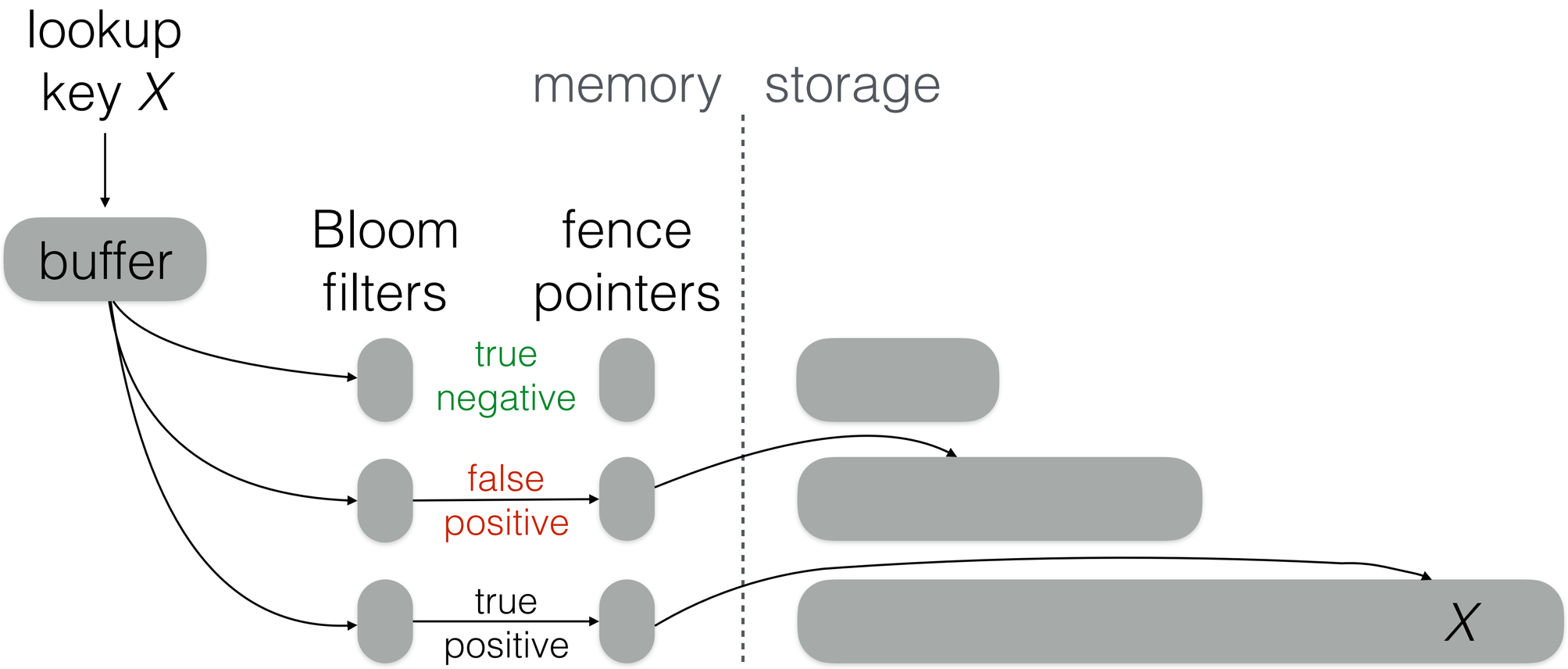


Performance & Cost Tradeoffs



bigger filters → fewer false positives

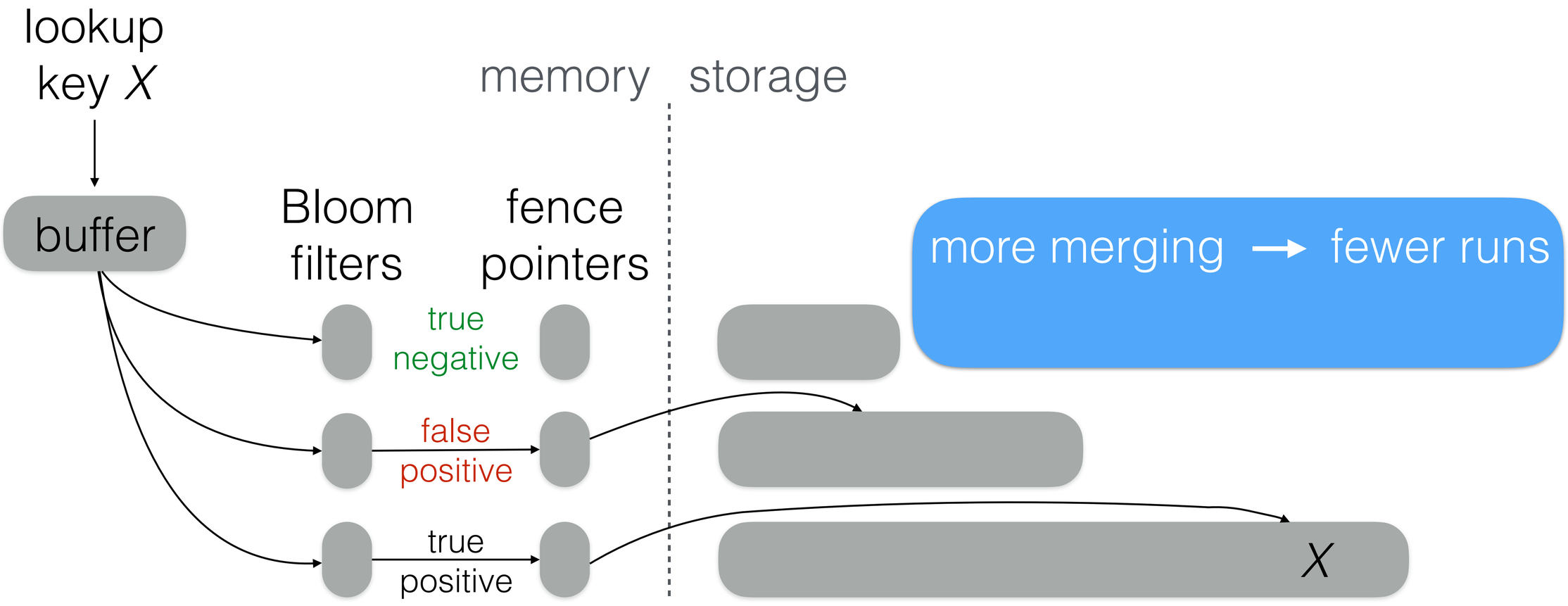
Performance & Cost Tradeoffs



bigger filters → fewer false positives

memory vs. lookups

Performance & Cost Tradeoffs

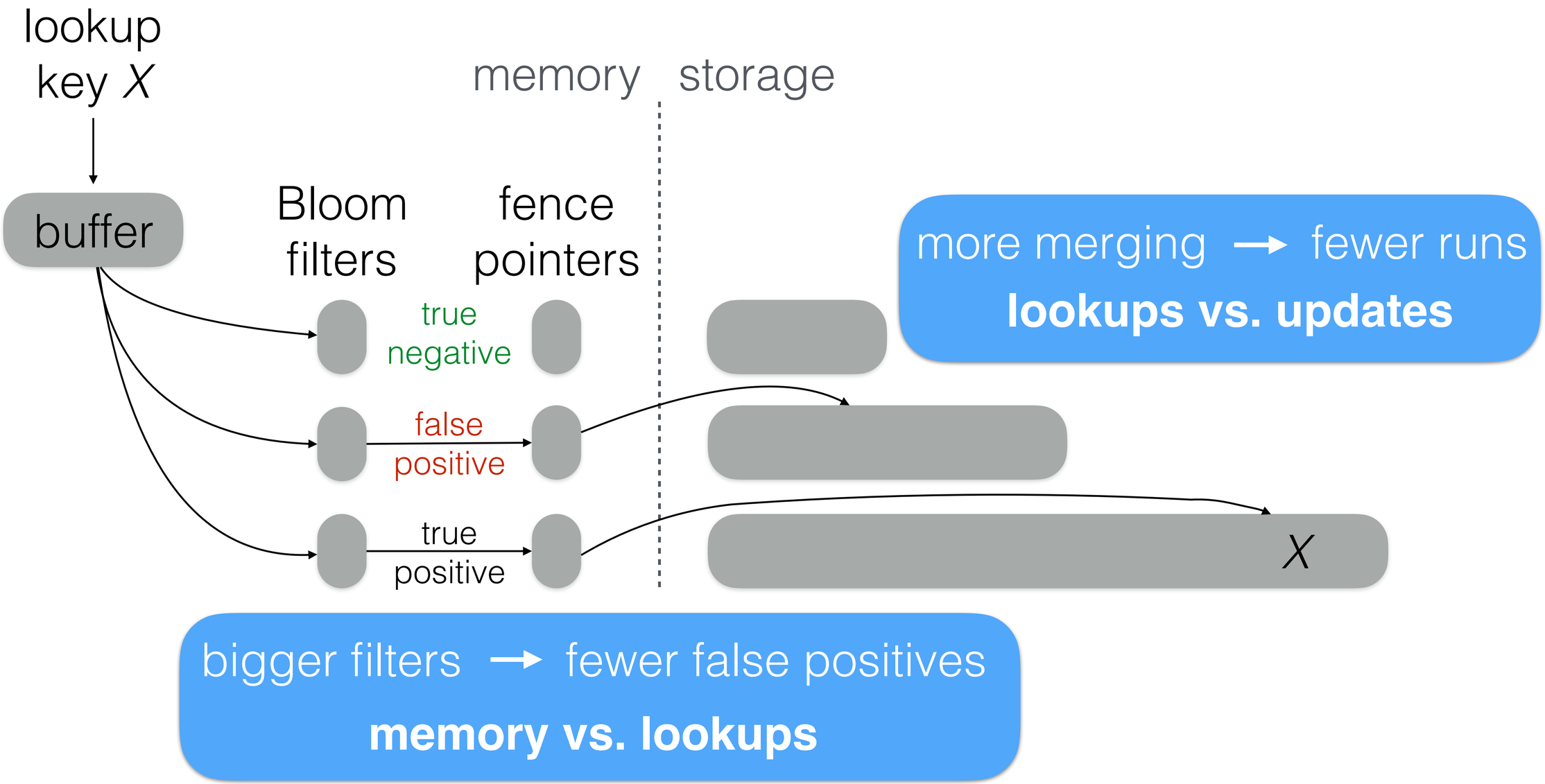


more merging → fewer runs

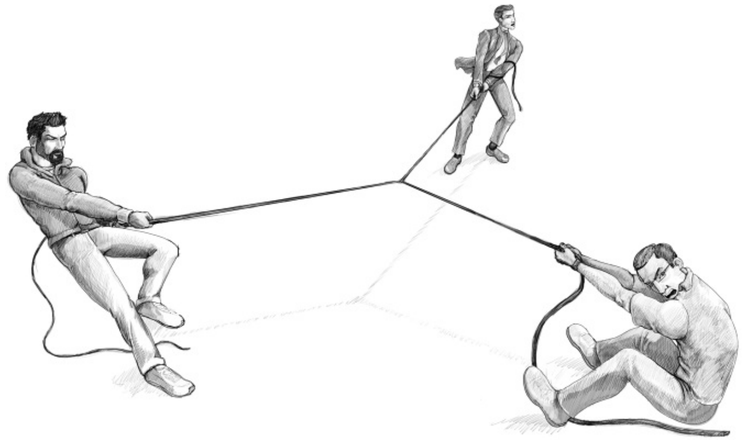
bigger filters → fewer false positives

memory vs. lookups

Performance & Cost Tradeoffs

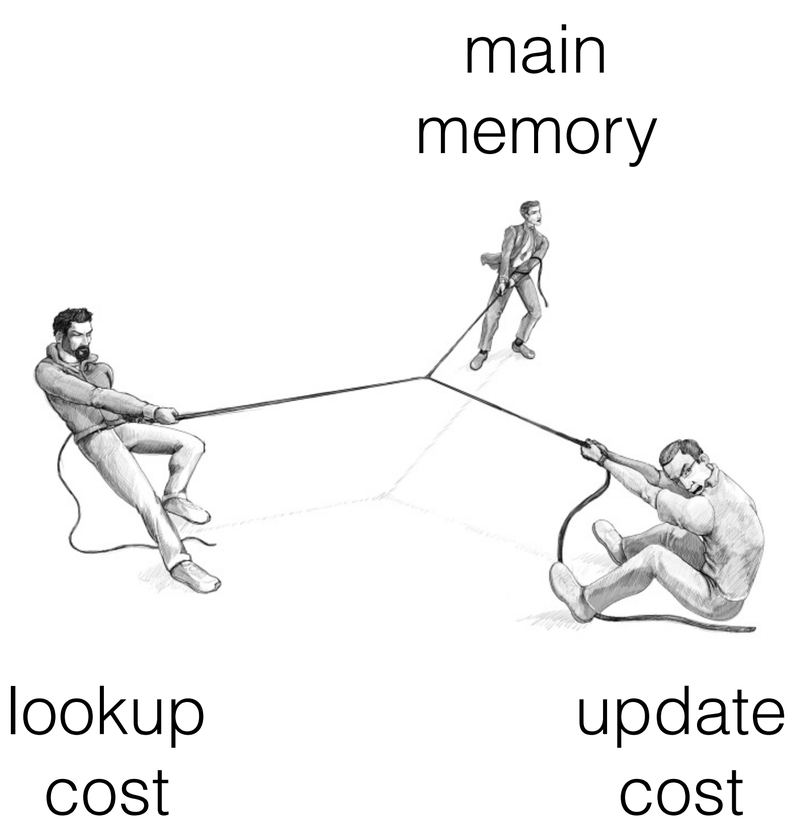


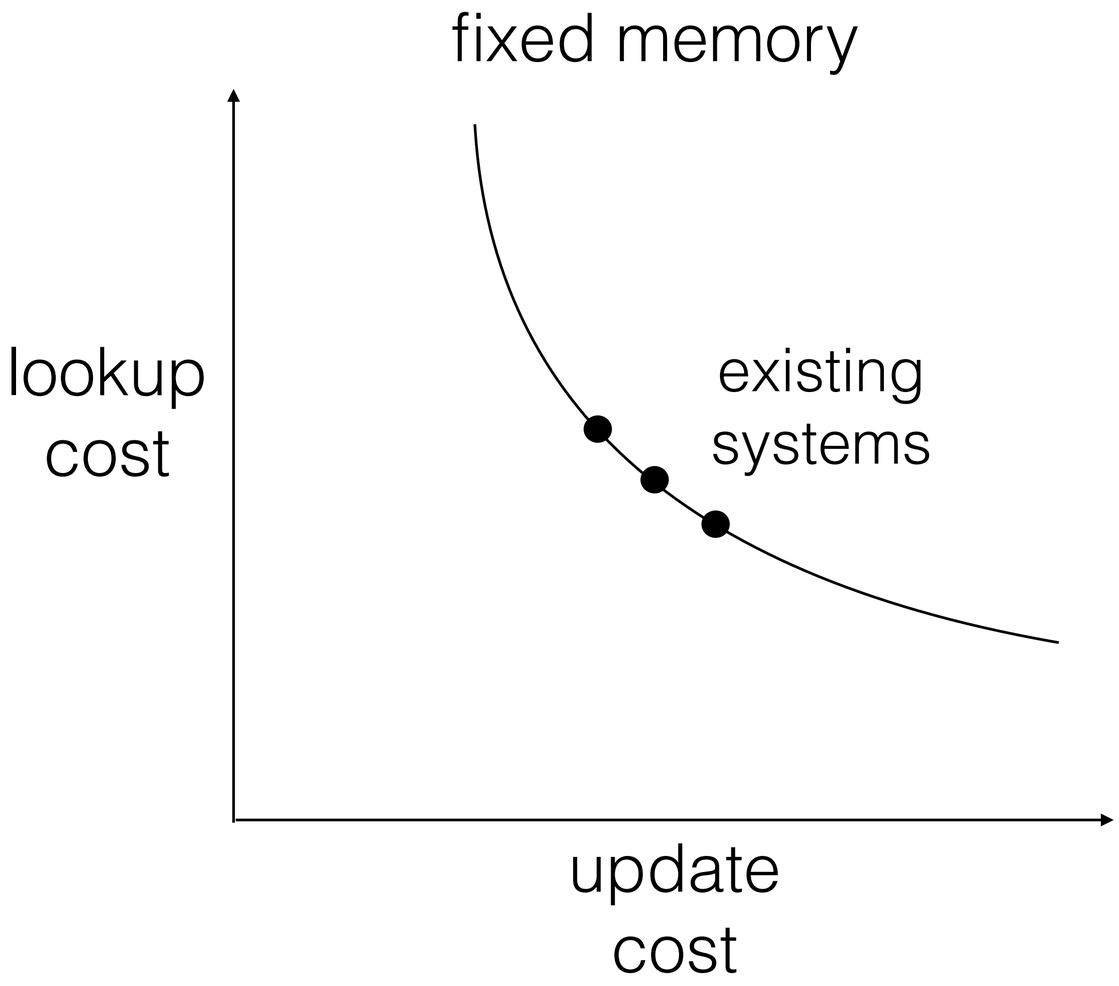
main
memory

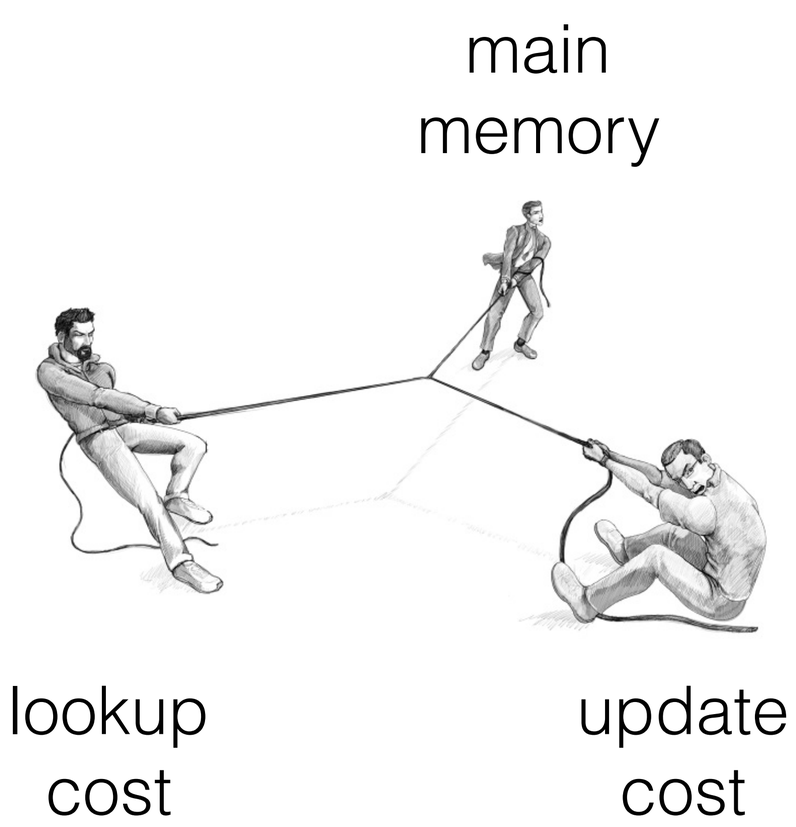


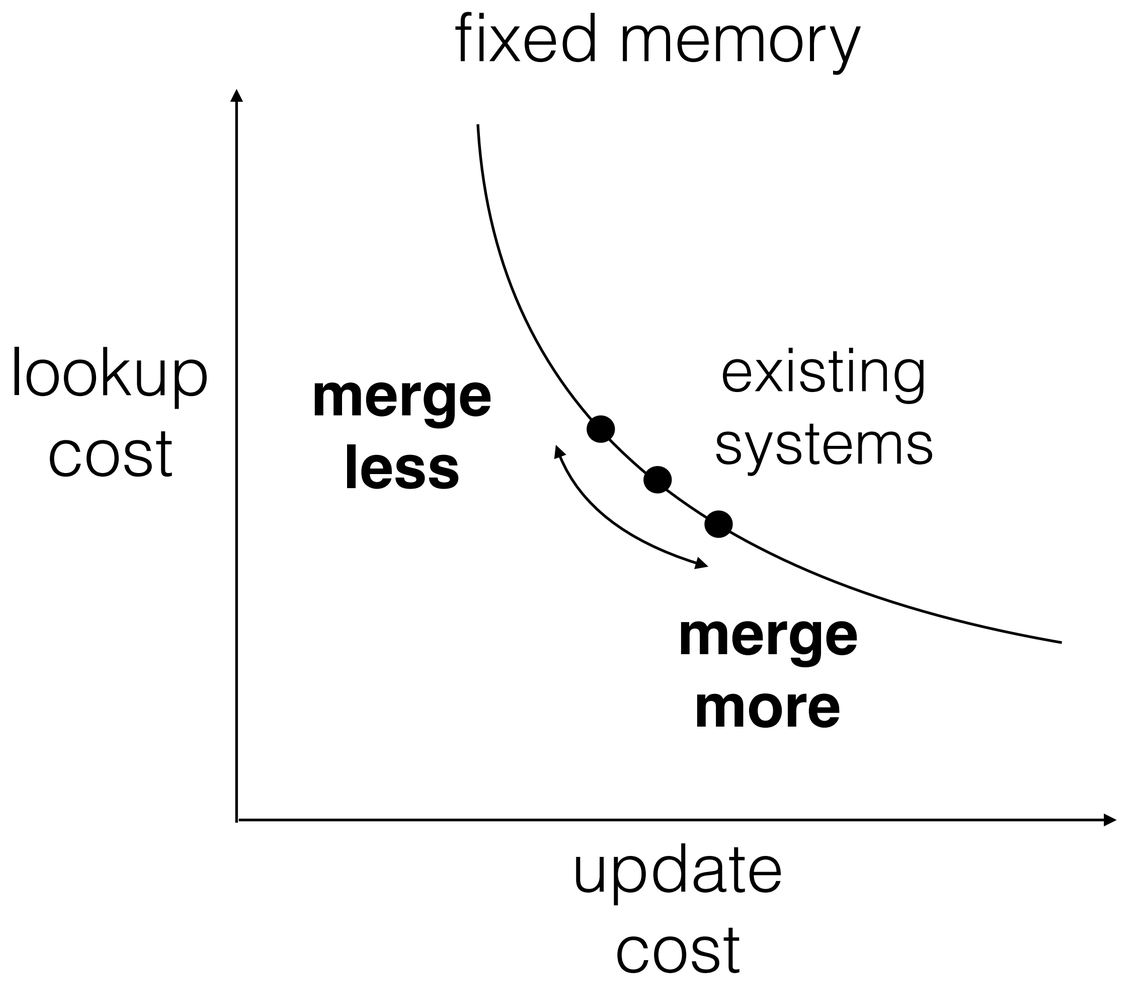
lookup
cost

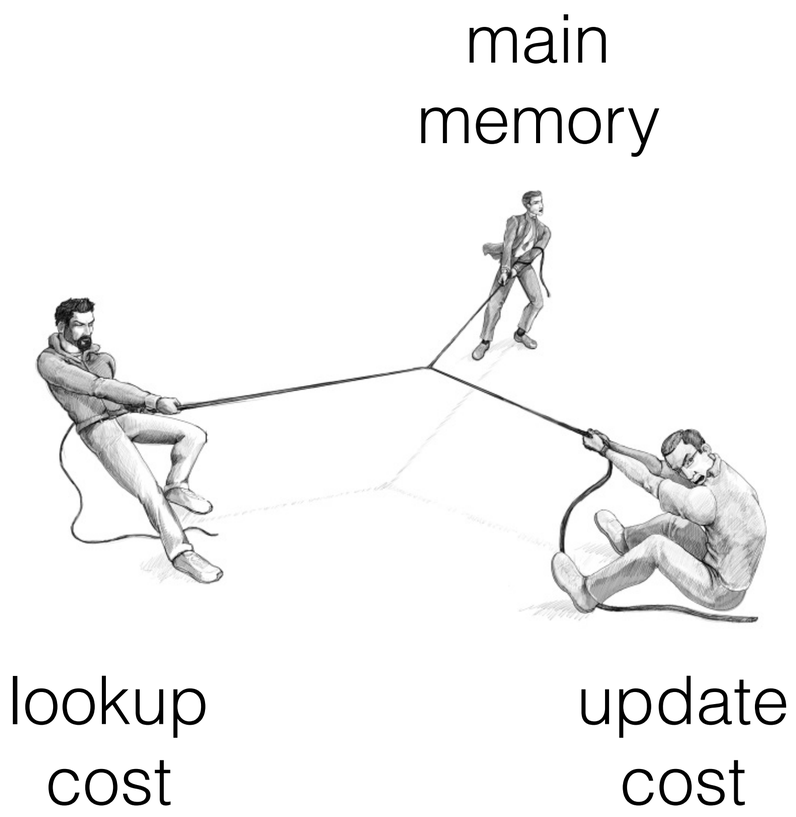
update
cost

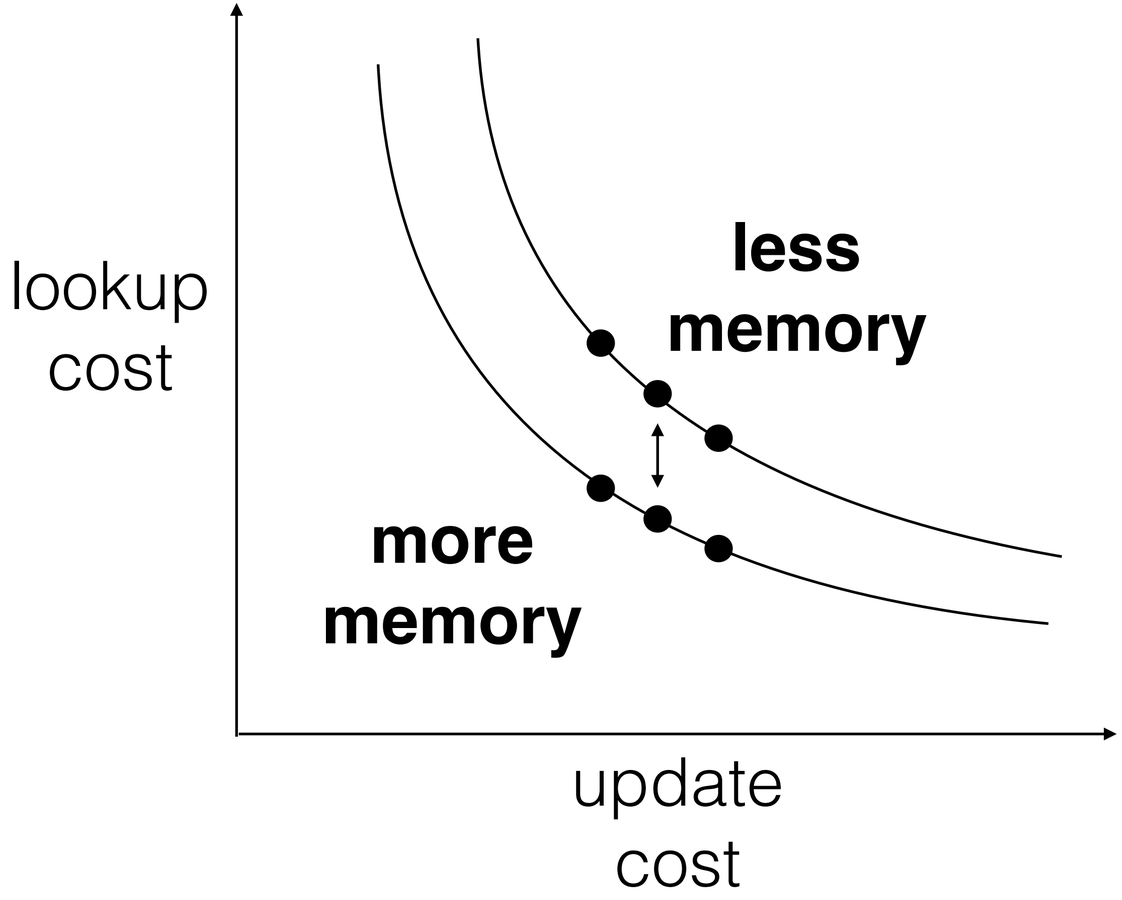

main
memory
lookup
cost
update
cost


fixed memory
lookup
cost
existing
systems
update
cost


main
memory
lookup
cost
update
cost

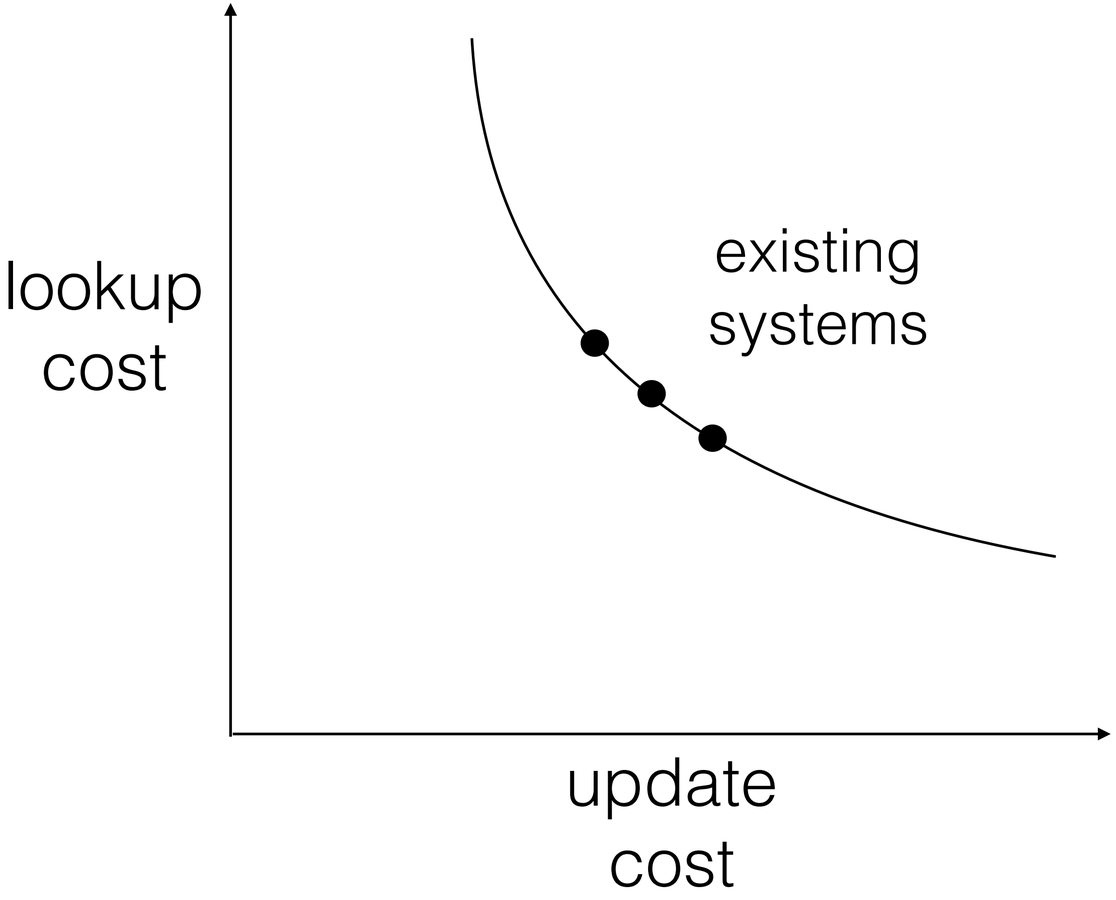

fixed memory
lookup
cost
merge
less
existing
systems
merge
more
update
cost


main
memory
lookup
cost
update
cost


lookup
cost
less
memory
more
memory
update
cost

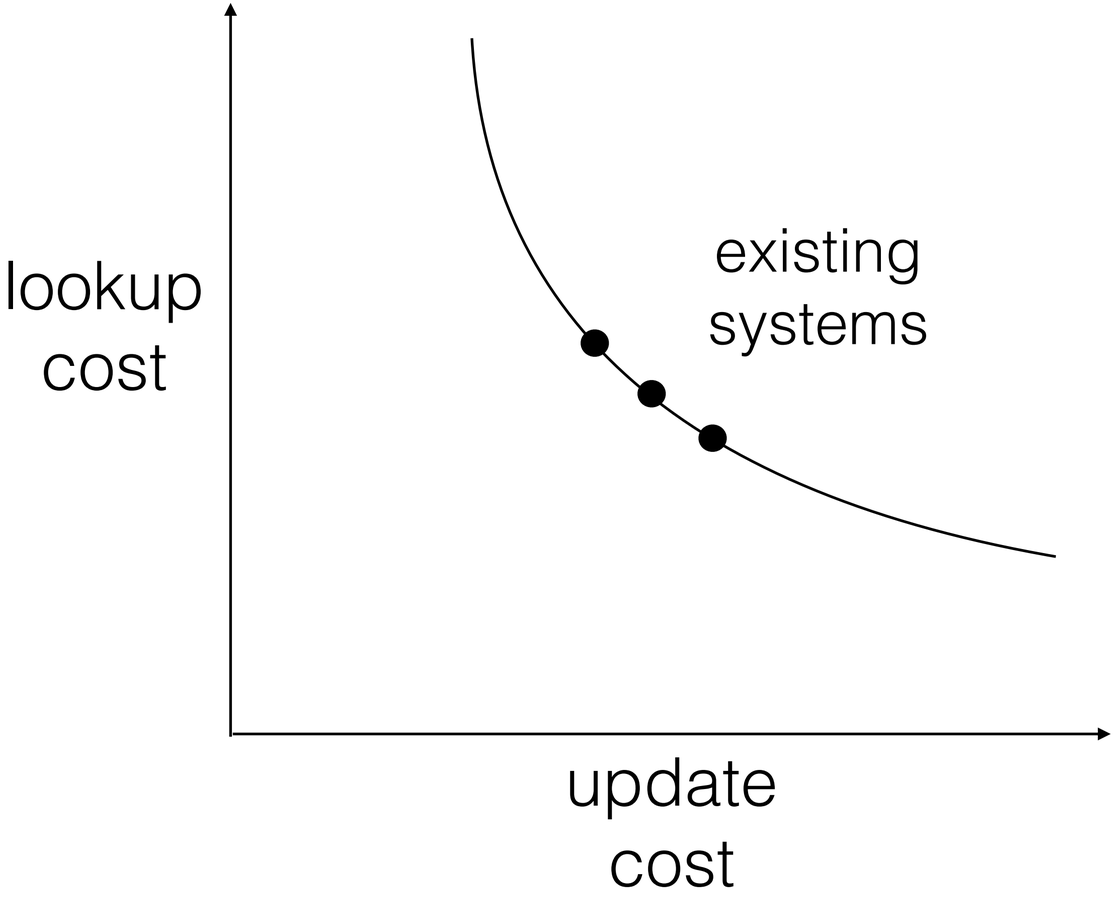
Problem 1:

Problem 2:



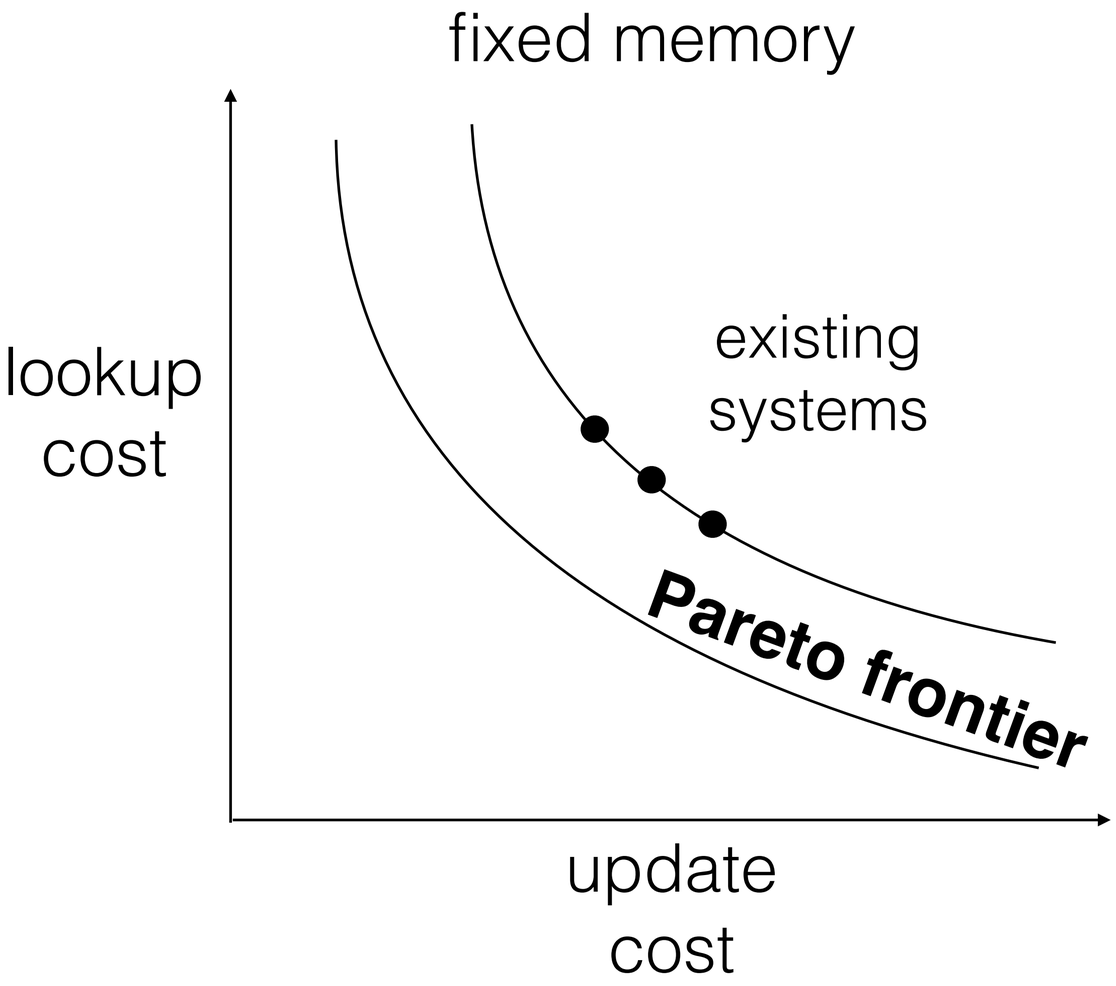
Problem 1: **suboptimal design**

Problem 2:



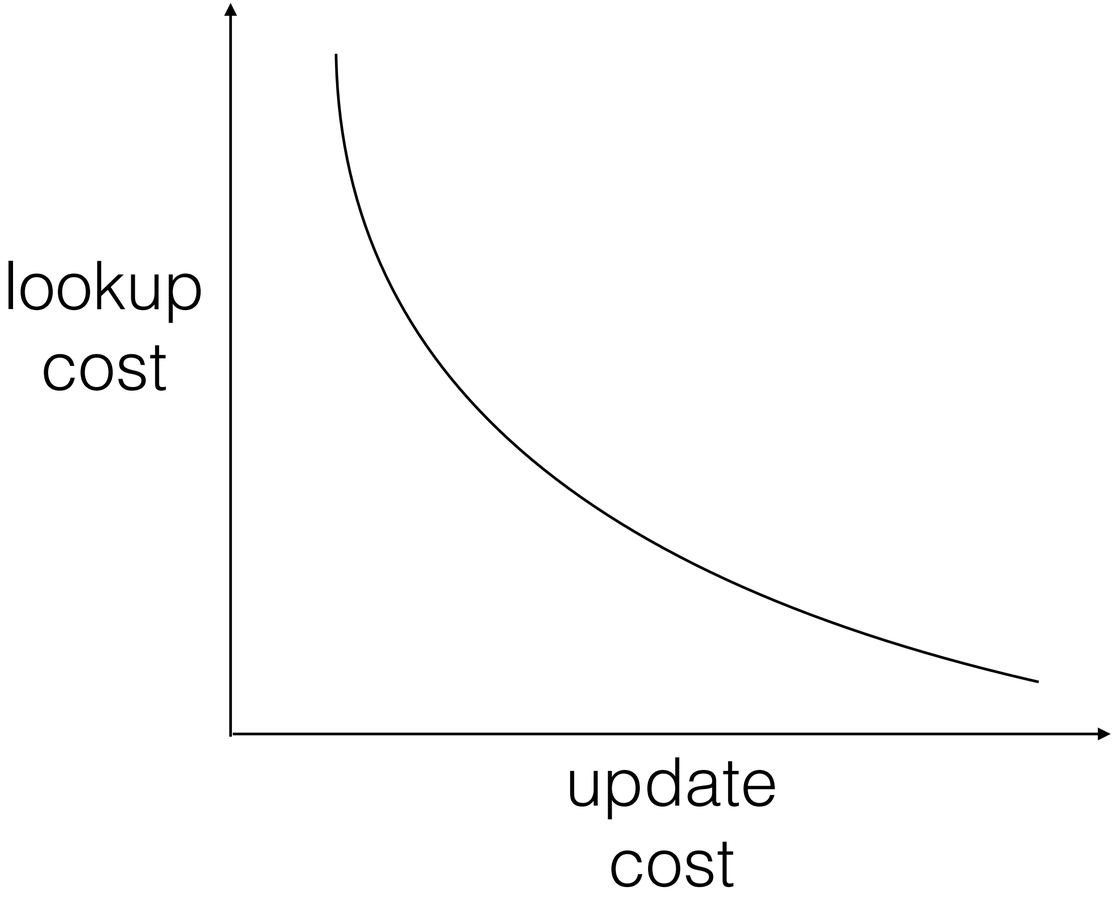
Problem 1: **suboptimal design**

Problem 2:



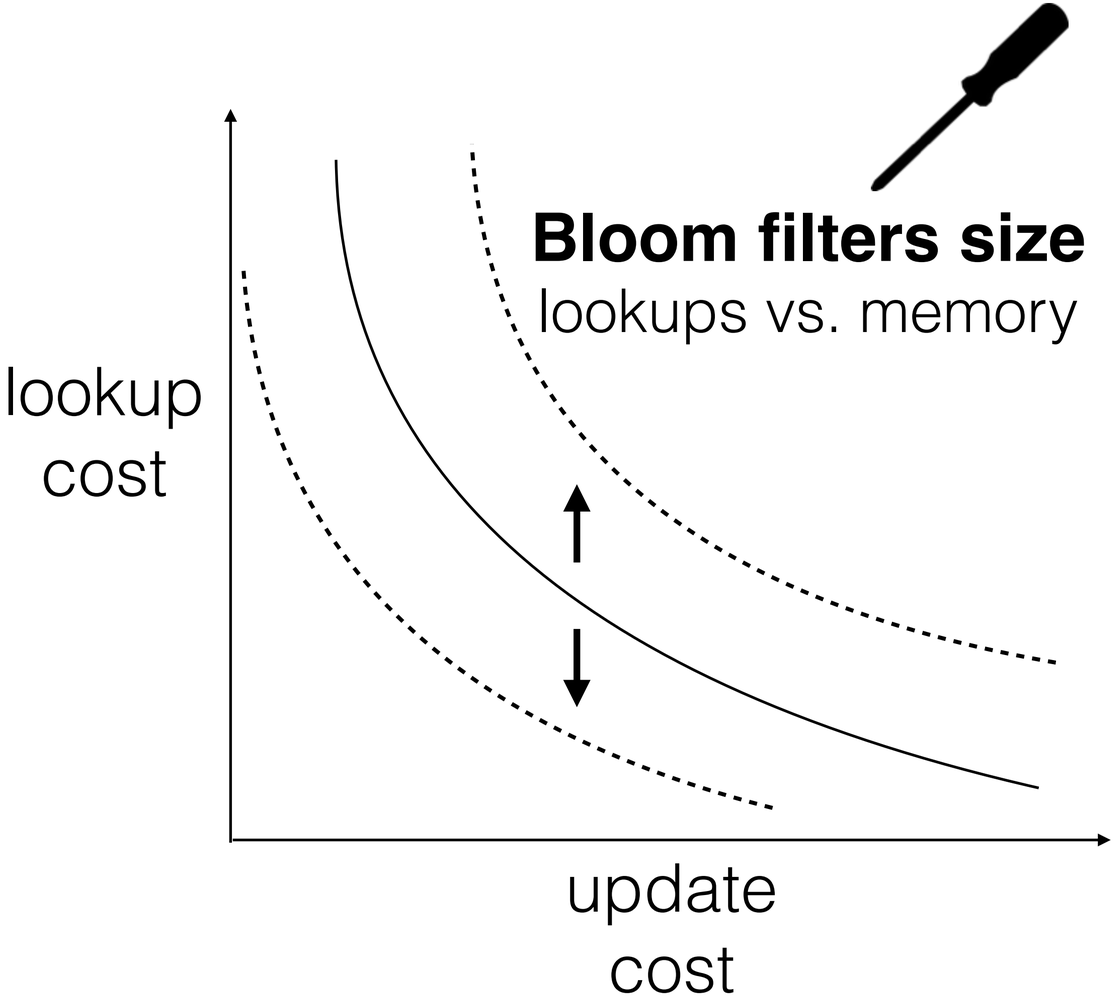
Problem 1: suboptimal design

Problem 2: **hard to tune**



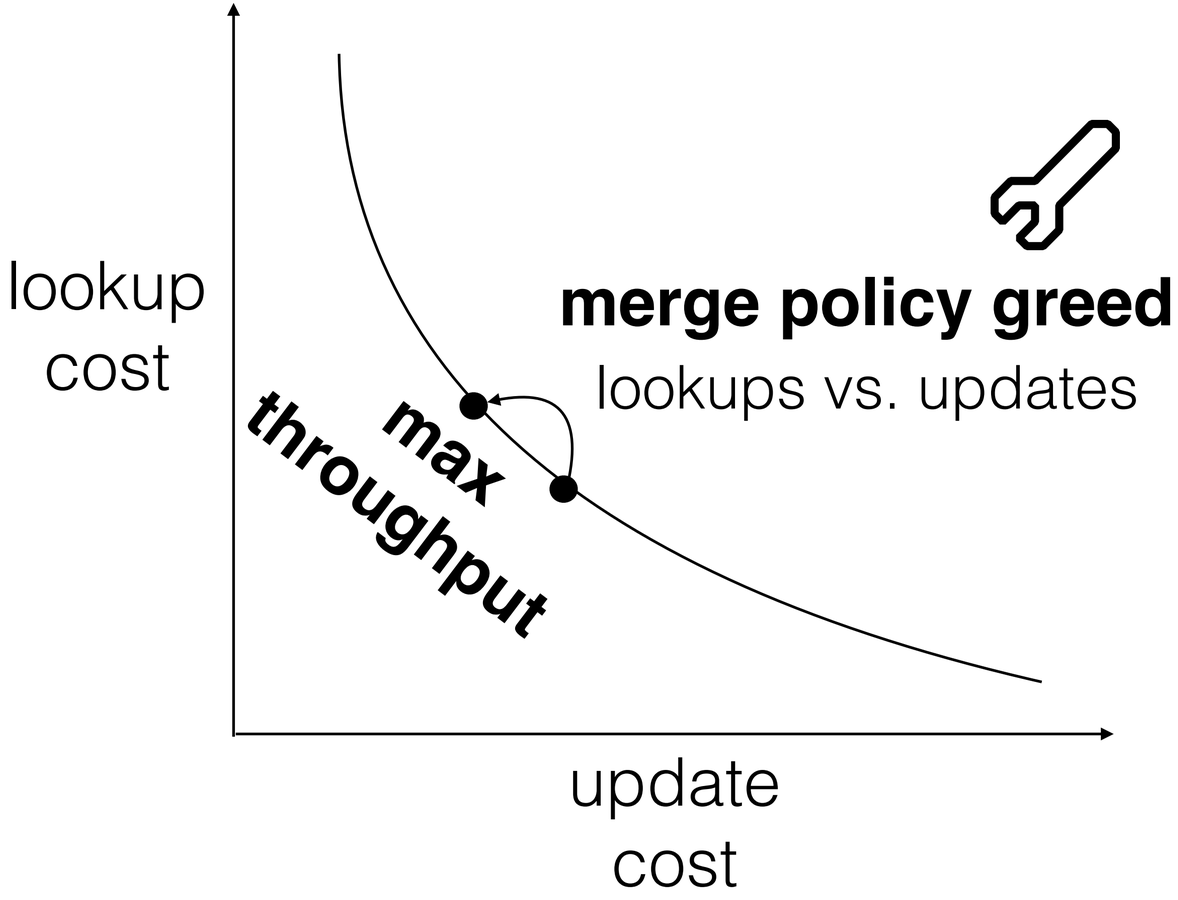
Problem 1: suboptimal design

Problem 2: **hard to tune**



Problem 1: suboptimal design

Problem 2: **hard to tune**



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

Niv Dayan, Manos Athanasollis, Stratos Idreos

SIGMOD 2017

“Best of SIGMOD”

DASlab @ Harvard SEAS

Monkey:	**O**ptimal	**N**avigable	**Key**-Value Store
observations:			
insights:			
steps:			

DASlab @ Harvard SEAS

Monkey:	**O**ptimal	**N**avigable	**Key**-Value Store
observations:	filters fixed false positive rates		
insights:	lookup cost = $\sum p_i$ suboptimal		
steps:	optimize allocation asymptotically better memory vs. lookups		

DASlab @ Harvard SEAS

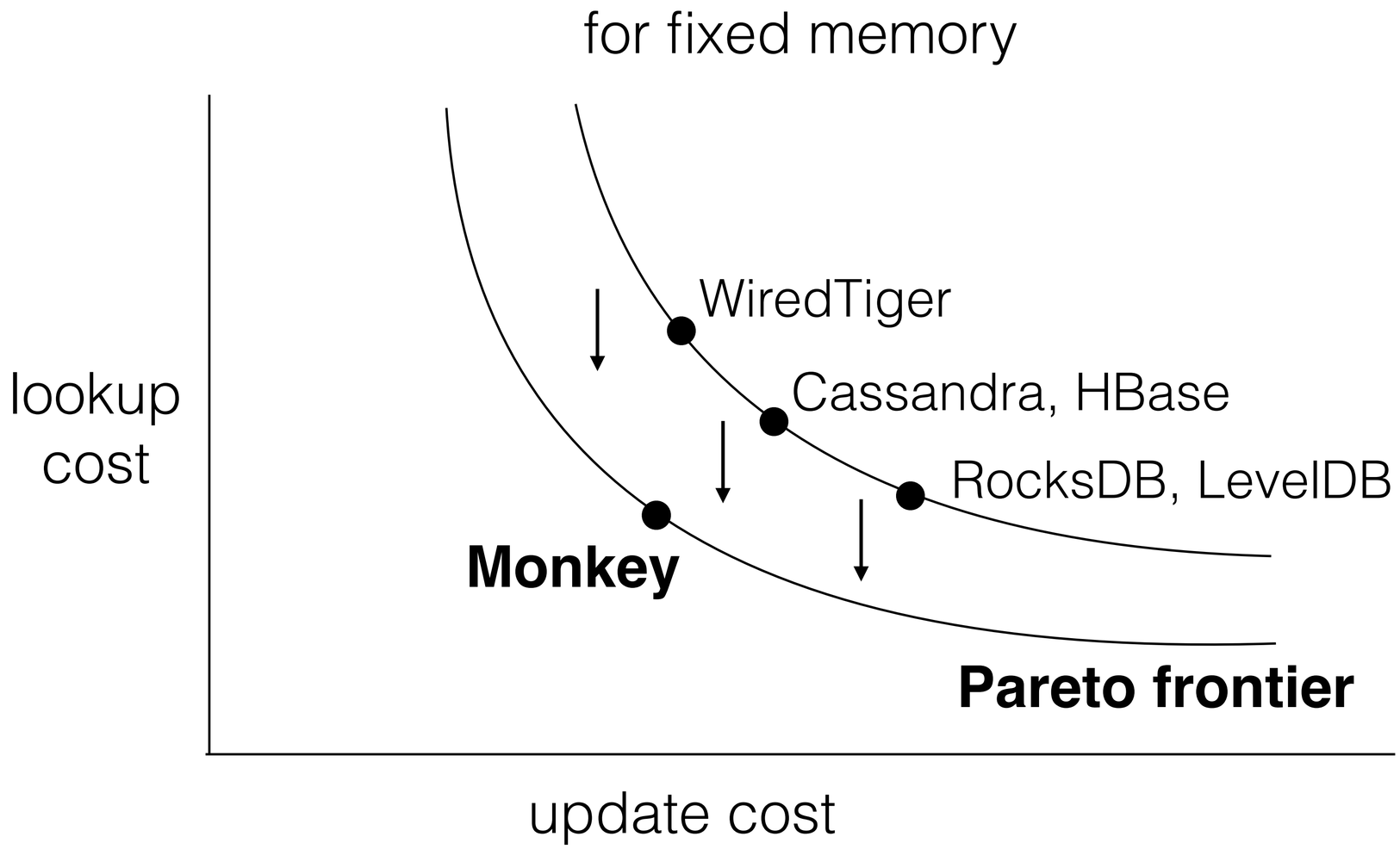
Monkey:	**O**ptimal	**N**avigable	**Key**-Value Store
observations:	filters fixed false positive rates	LSM-tree merge policy ↓ ? performance	
insights:	lookup cost = $\sum p_i$ suboptimal	log sorted array	
steps:	optimize allocation asymptotically better memory vs. lookups	updates vs. lookups navigate	

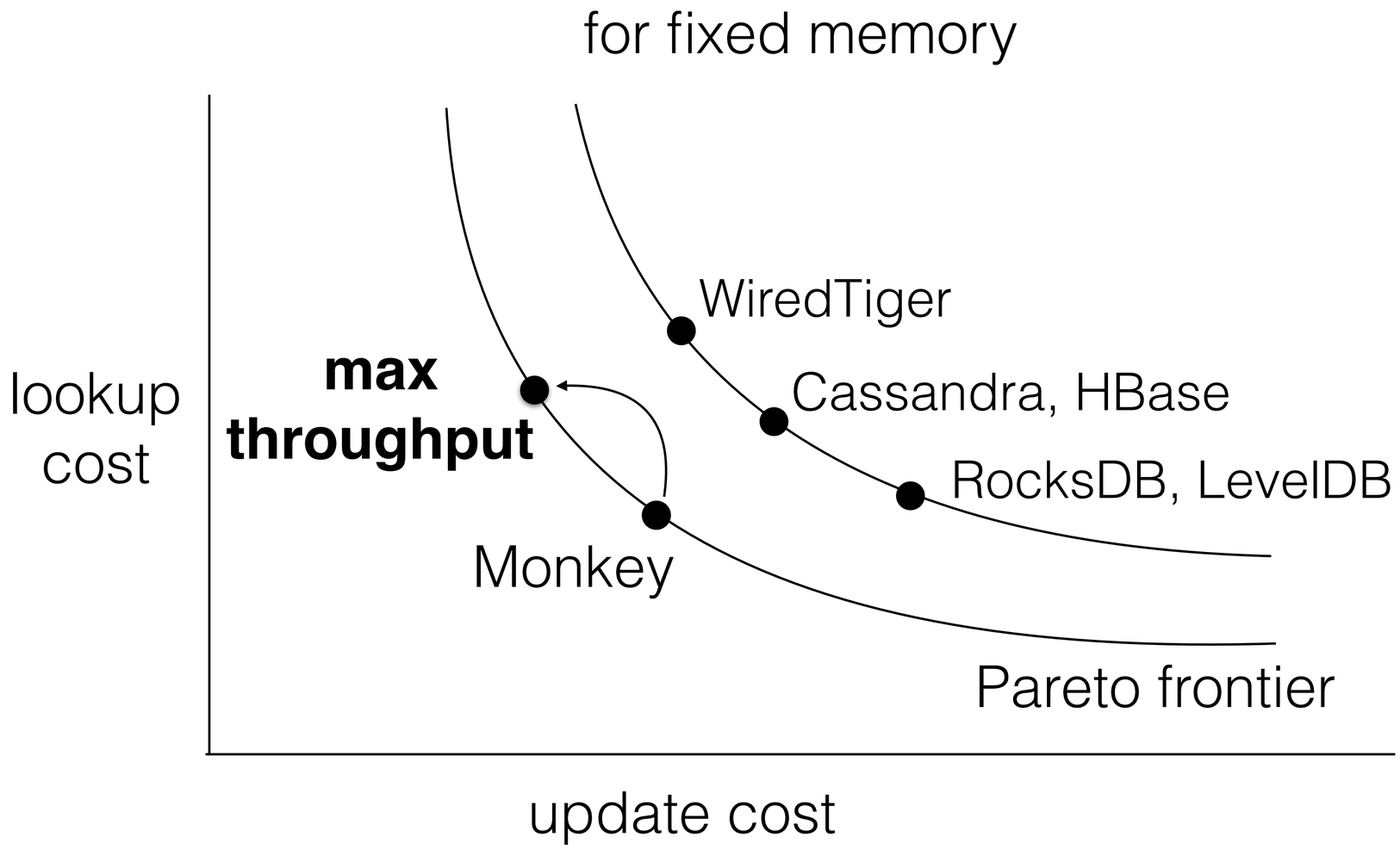
DASlab @ Harvard SEAS

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

observations:	filters fixed false positive rates	LSM-tree merge policy ↓ ? performance	ad-hoc trade-offs memory lookups updates
insights:	lookup cost = $\sum p_i$ suboptimal	log sorted array	lookup cost existing Monkey update cost
steps:	optimize allocation asymptotically better memory vs. lookups	updates vs. lookups navigate	answer what-if design questions

DASlab @ Harvard SEAS


for fixed memory
lookup cost
WiredTiger
Cassandra, HBase
RocksDB, LevelDB
Monkey
Pareto frontier
update cost


for fixed memory
lookup cost
max throughput
WiredTiger
Cassandra, HBase
RocksDB, LevelDB
Monkey
Pareto frontier
update cost

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

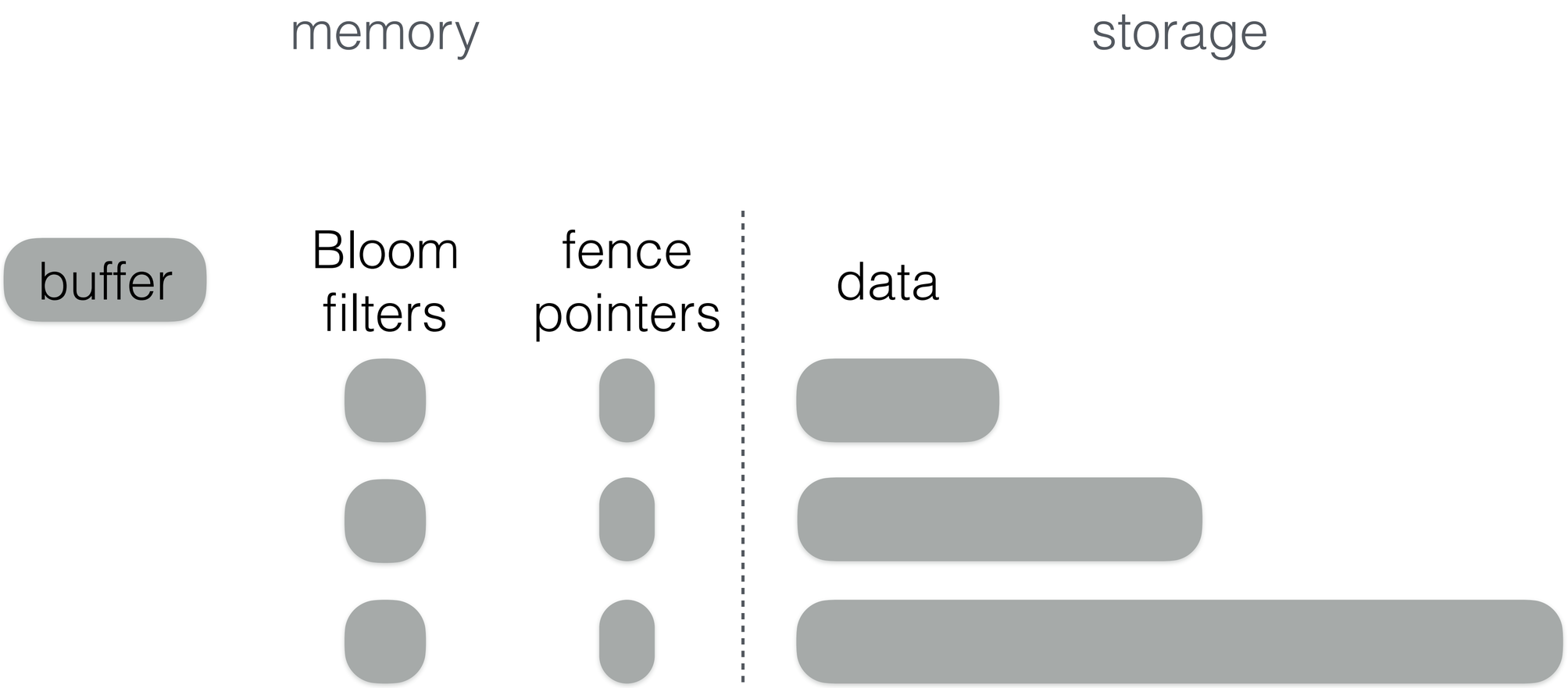
observations:	filters fixed false positive rates	LSM-tree merge policy ↓ ? performance	ad-hoc trade-offs memory lookups updates
insights:	lookup cost = $\sum p_i$ suboptimal	log sorted array	lookup cost existing Monkey update cost
steps:	optimize allocation asymptotically better memory vs. lookups	updates vs. lookups navigate	answer what-if design questions

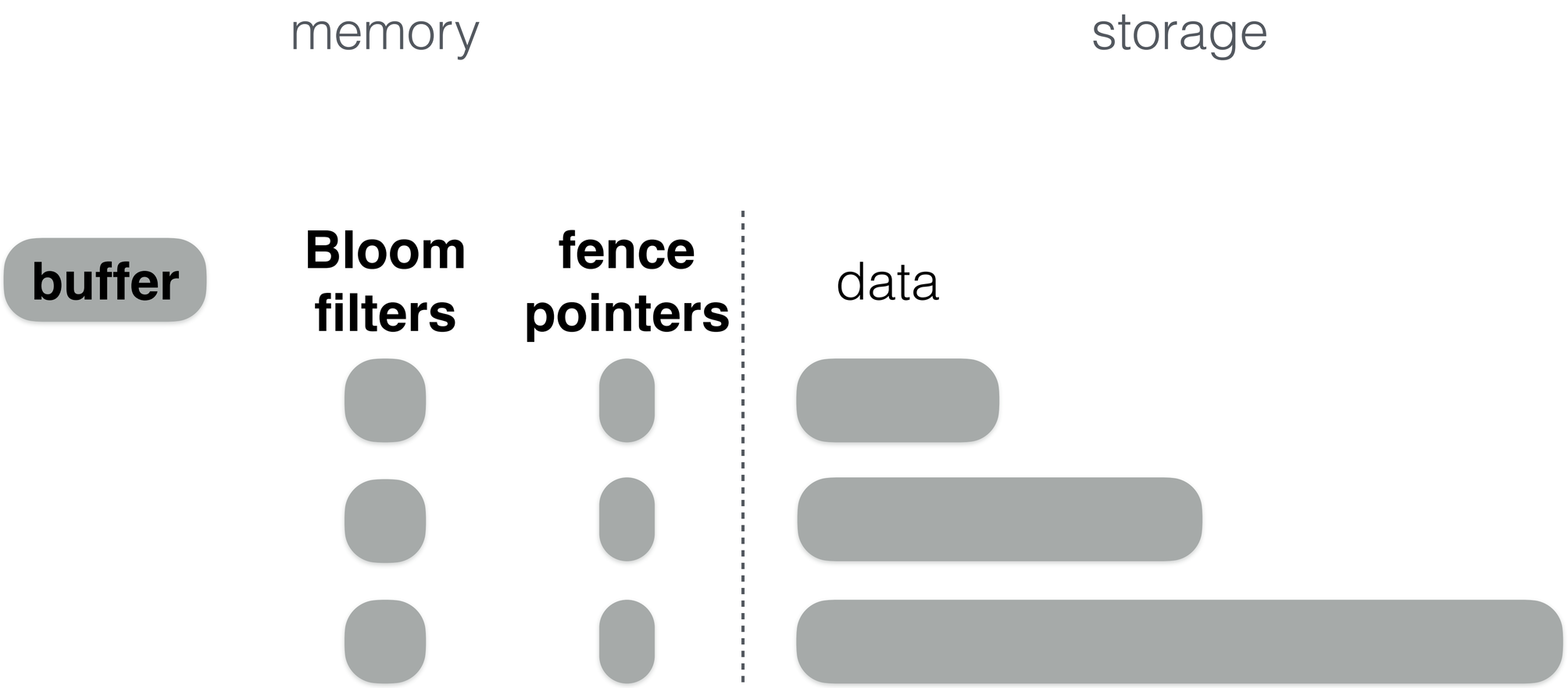
DASlab @ Harvard SEAS

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

observations:	filters fixed false positive rates	LSM-tree merge policy ↓ ? performance	ad-hoc trade-offs memory lookups updates
insights:	lookup cost = $\sum p_i$ suboptimal	log sorted array	lookup cost existing Monkey update cost
steps:	optimize allocation asymptotically better memory vs. lookups	updates vs. lookups navigate	answer what-if design questions

DASlab @ Harvard SEAS


memory
storage
buffer
Bloom filters
fence pointers
data


memory
storage
buffer
Bloom filters
fence pointers
data

memory

storage

buffer

Bloom filters

fence pointers

data

memory

storage

buffer < **Bloom filters** > fence pointers

data

memory

storage

Bloom filters

data

memory

storage

***X* bits per entry**

Bloom filters

data

memory

storage

***X* bits per entry**

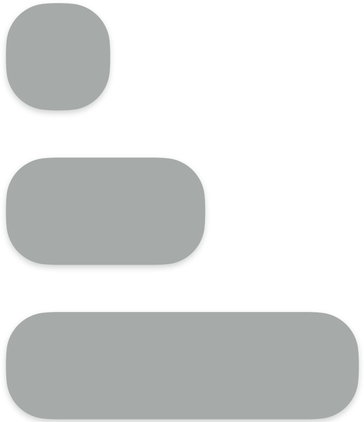
Bloom filters

data

memory

***X* bits per entry**

Bloom filters



$$\textbf{false positive rate } p = e^{-\frac{\textbf{bits } M}{\textbf{entries } N} \ln(2)^2}$$

DASlab @ Harvard SEAS

memory

X bits per entry

Bloom filters

p

p

p

$$\text{false positive rate } \boldsymbol{p} = e^{-\frac{\text{bits } M}{\text{entries } N} \ln(2)^2}$$

memory

X bits per entry

Bloom filters

p

p

p

worst-case I/O overhead:

$$\text{false positive rate } p = e^{-\frac{\text{bits } M}{\text{entries } N} \ln(2)^2}$$

memory

X bits per entry

Bloom filters

p

p

p

worst-case I/O overhead:

$$O(\sum \boldsymbol{p})$$

$$\text{false positive rate } p = e^{-\frac{\text{bits } M}{\text{entries } N} \ln(2)^2}$$

memory

X bits per entry

Bloom filters

p

p

p

worst-case I/O overhead:

$$O(\sum \boldsymbol{p})$$

$$\textit{false positive rate } \boldsymbol{p} = \mathbf{e}^{-\frac{\textbf{bits } M}{\textbf{entries } N} \ln(2)^2}$$

memory

X bits per entry

Bloom filters

p

p

p

worst-case I/O overhead:

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

$$\text{false positive rate } \boldsymbol{p} = \mathbf{e}^{-\frac{\textbf{bits } M}{\textbf{entries } N}\ln(2)^2}$$

memory

X bits
per entry

Bloom
filters

p

p

p

O(log(*N*))

worst-case I/O overhead:

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

memory

X bits
per entry

Bloom
filters

p

p

p

O(log(*N*))

worst-case I/O overhead:

$$O(\ \mathbf{log(\mathit{N})} \cdot \mathbf{e^{-\mathit{M/N}}}\ )$$

memory

worst-case I/O overhead:

$$O(\log(N) \cdot e^{-M/N})$$

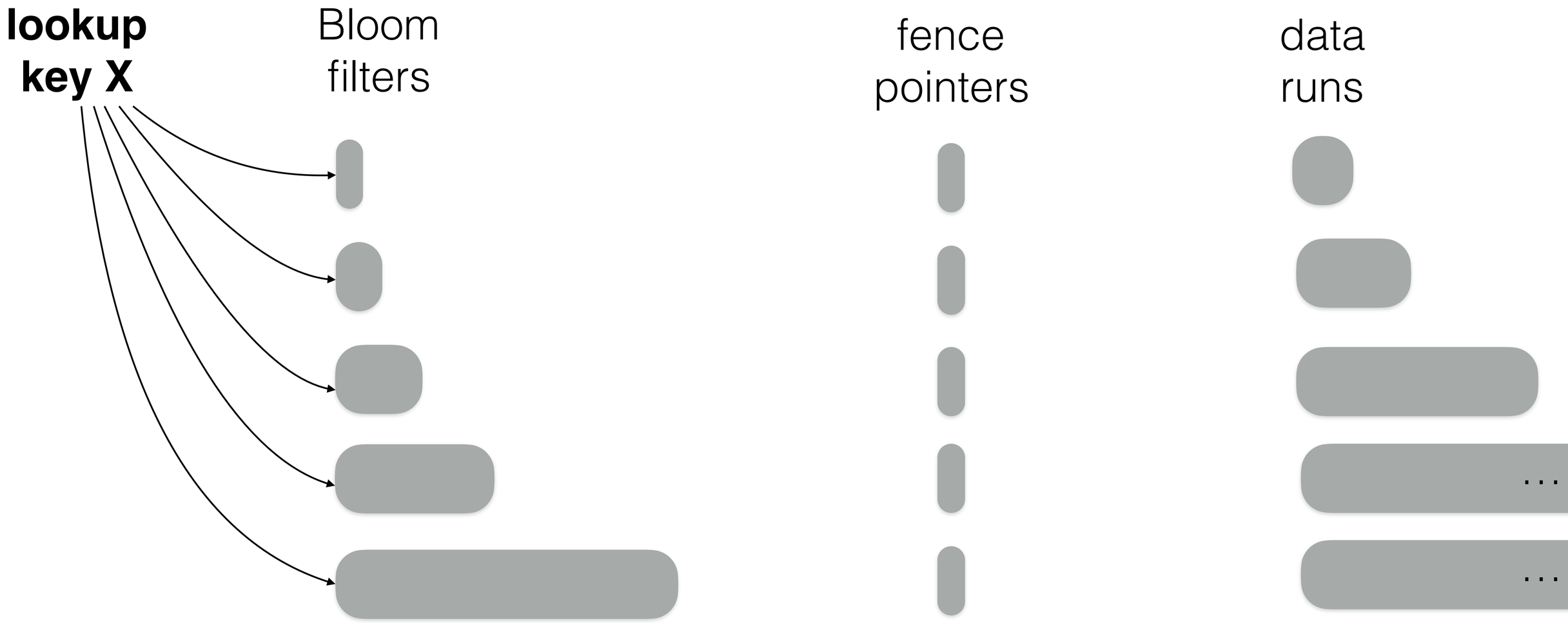
Can we do better?

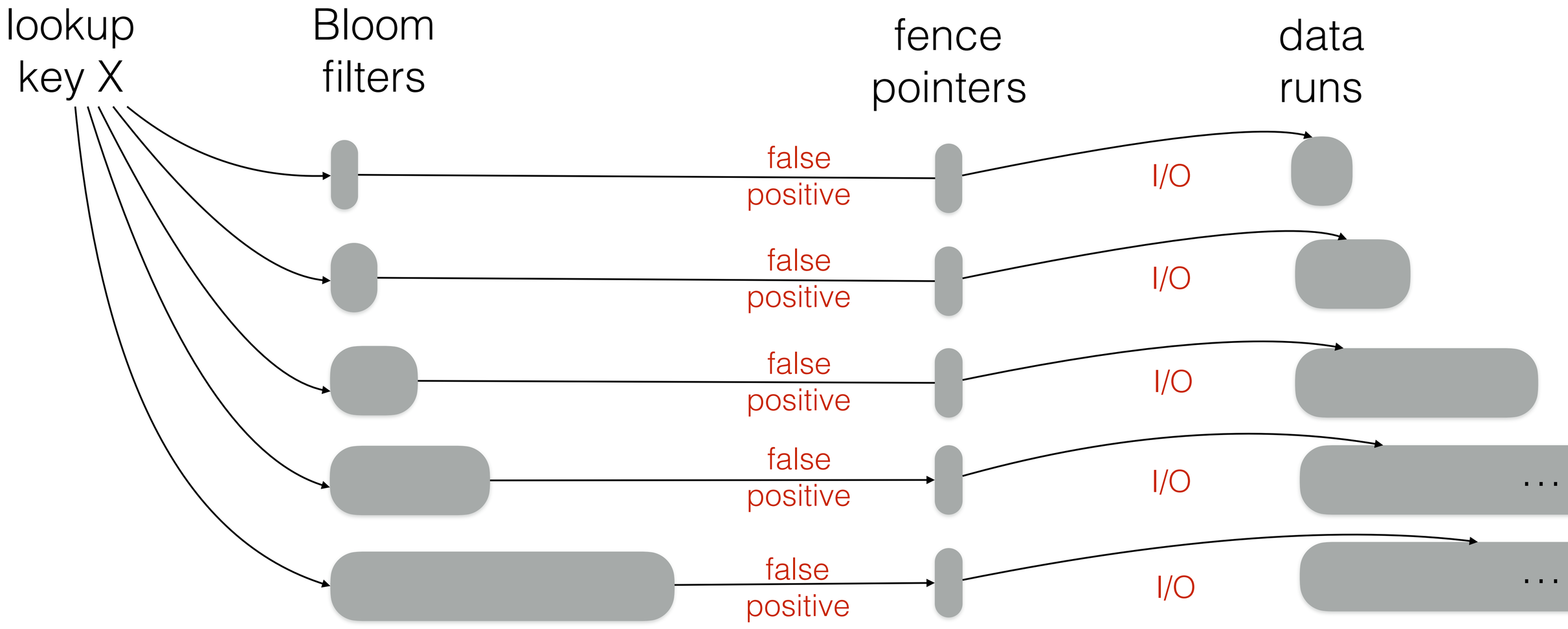
filters

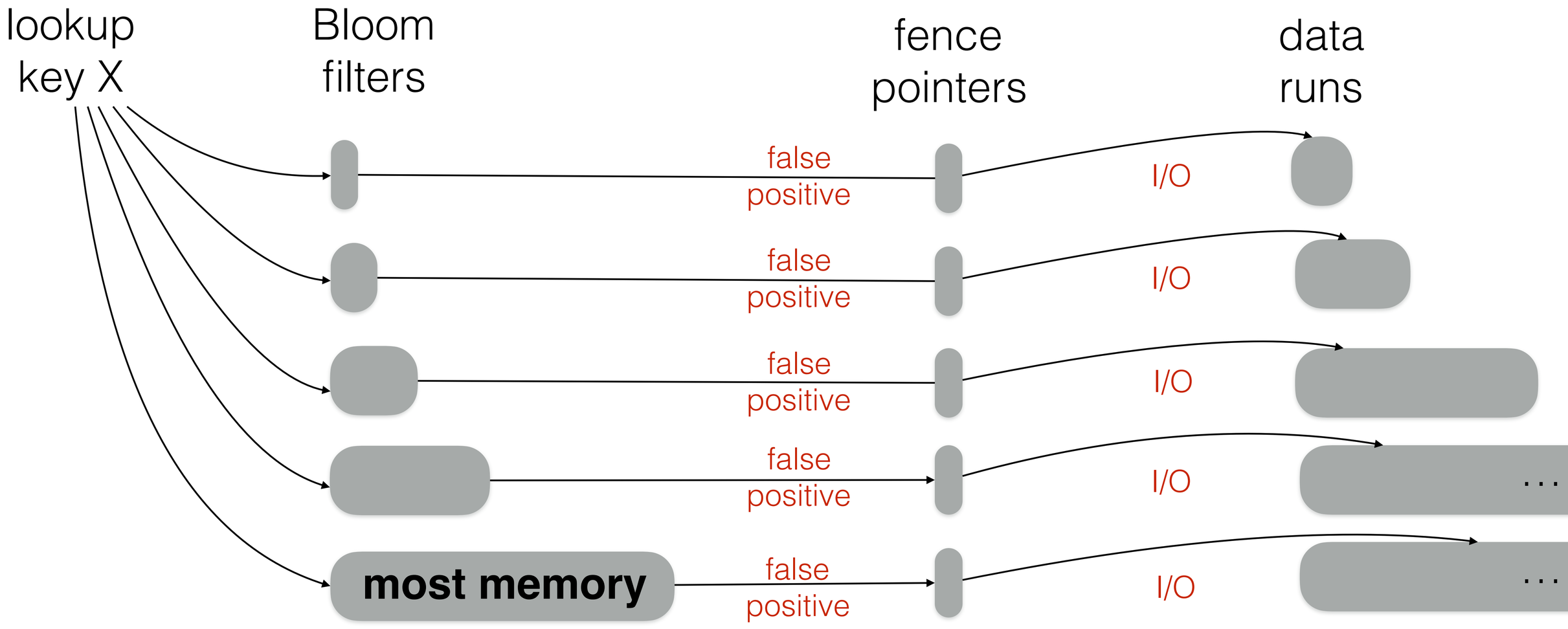
p

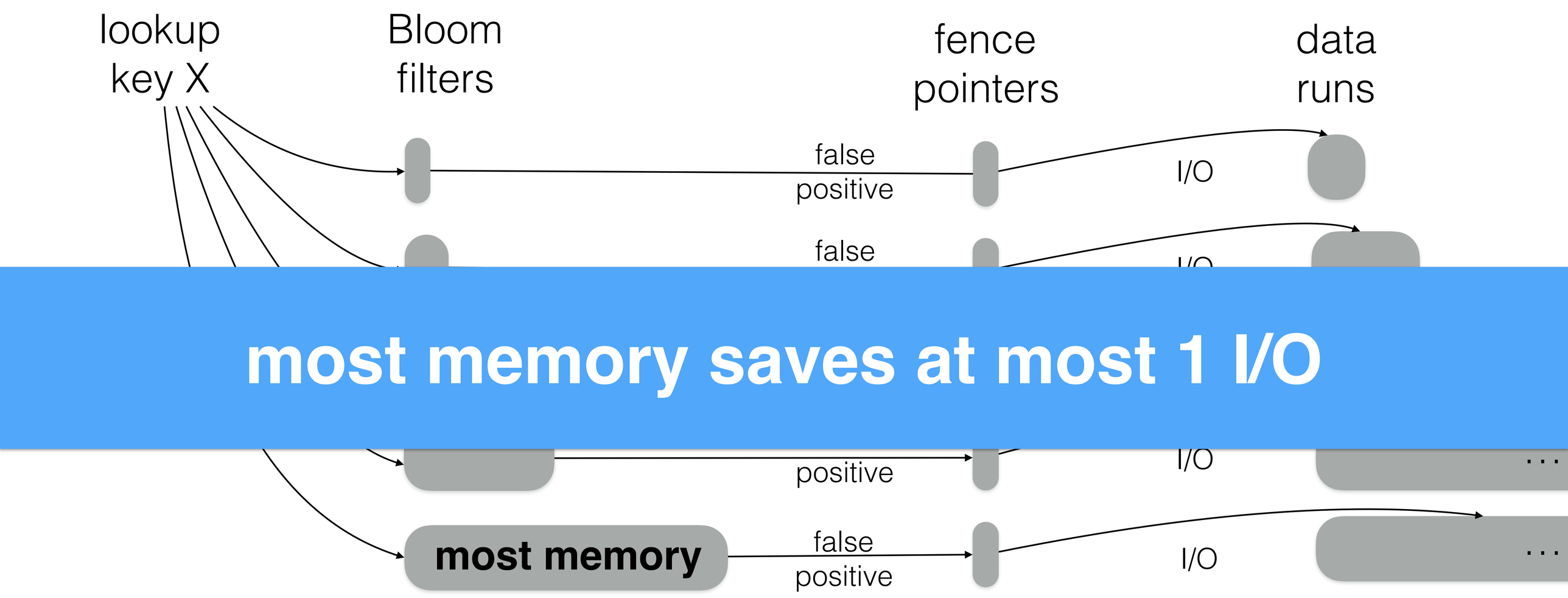
p

p

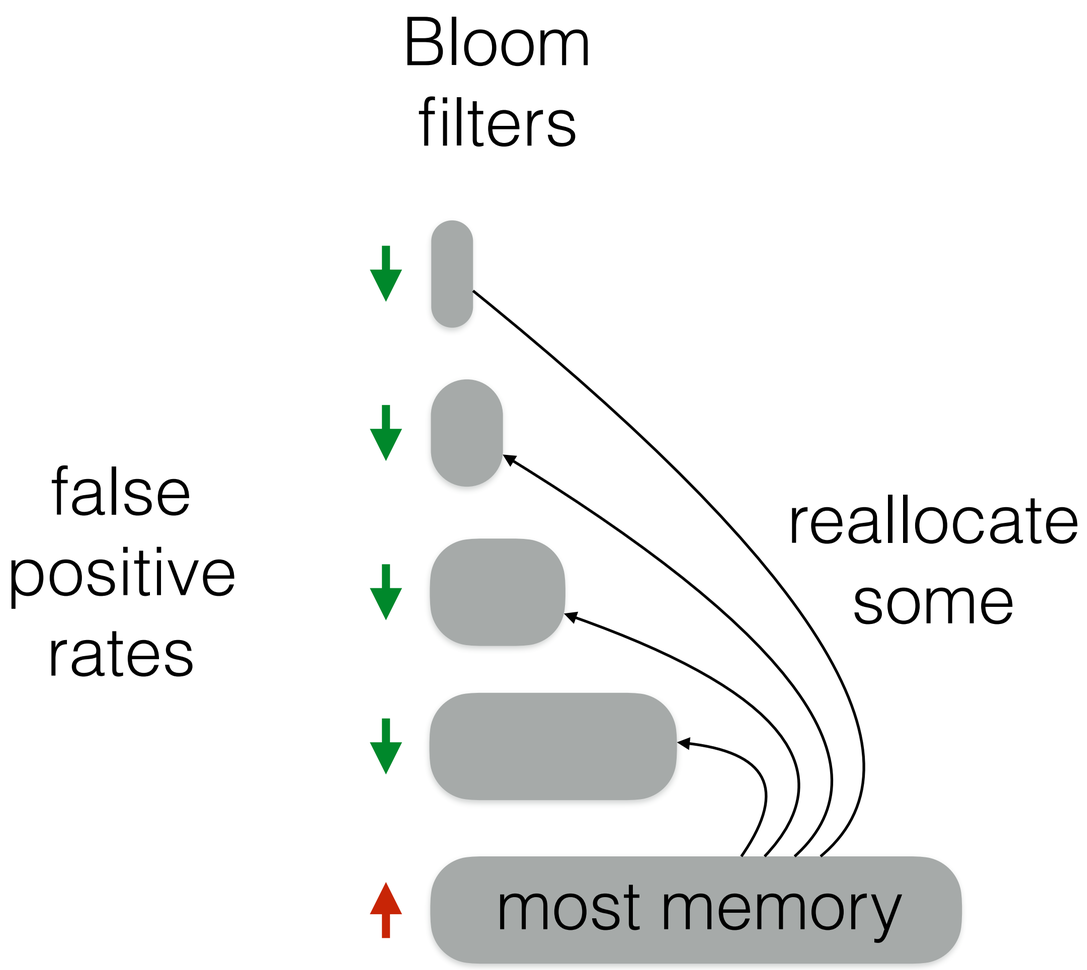

lookup
key X
Bloom
filters
fence
pointers
data
runs
...
...


lookup
key X
Bloom
filters
fence
pointers
data
runs
false
positive
I/O
false
positive
I/O
false
positive
I/O
false
positive
I/O
...
false
positive
I/O
...


lookup
key X
Bloom
filters
fence
pointers
data
runs
false
positive
I/O
false
positive
I/O
false
positive
I/O
false
positive
I/O
...
most memory
false
positive
I/O
...

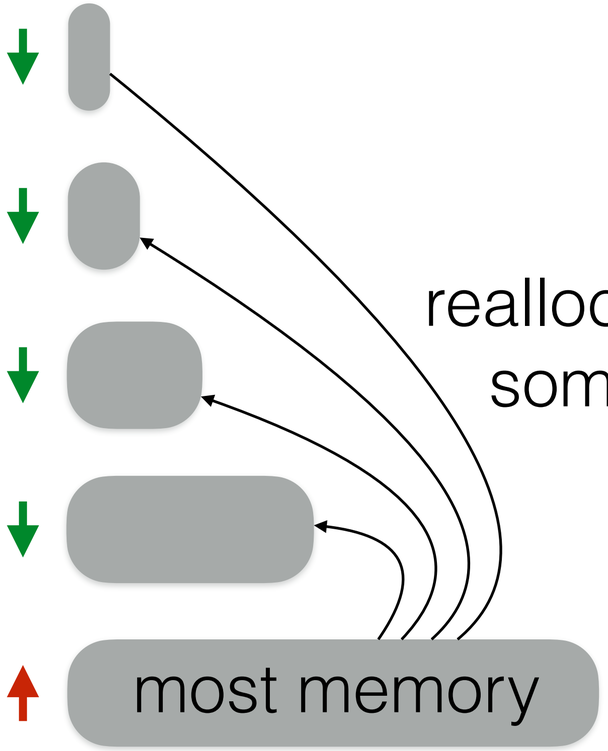


most memory saves at most 1 I/O


Bloom filters
false positive rates
reallocate some
most memory

same memory, fewer lookup I/Os

false positive rates



relax

false positive rates

p_2

p_1

p_0

relax

false positive rates

$0 < p_2 < 1$

$0 < p_1 < 1$

$0 < p_0 < 1$

relax

false positive rates

$0 < p_2 < 1$

$0 < p_1 < 1$

$0 < p_0 < 1$

model

lookup cost $= f(\mathbf{p_0}, \mathbf{p_1} \ldots)$

memory footprint $= f(\mathbf{p_0}, \mathbf{p_1} \ldots)$

relax

false positive rates

$0 < p_2 < 1$

$0 < p_1 < 1$

$0 < p_0 < 1$

model

lookup cost $= f(p_0, p_1 \ldots)$

memory footprint $= f(p_0, p_1 \ldots)$

optimize

in terms of **p_0, p_1**

Bloom filters

false positive rates

$\vdots$

p_2

p_1

p_0

lookup cost $= \boldsymbol{\Sigma\ p_i}$

Bloom filters

⋮

false positive rates

p_2

p_1

p_0

lookup cost $= \sum p_i$

memory footprint

$$\textbf{\textit{false positive rate}} = e^{-\frac{\textbf{\textit{bits}}}{\textbf{\textit{entries}}} \ln(2)^2}$$

Bloom filters

⋮

false positive rates

p_2

p_1

p_0

lookup cost $= \sum p_i$

memory footprint

$$\textbf{\textit{bits}} = -\frac{\ln(\textbf{\textit{false positive rate}})}{\ln(2)^2}\ \textbf{\textit{entries}}$$

Bloom filters	memory footprint
false positive rates	
⋮	⋮
p_2	$bits(p_2, \mathbf{N/T^2})$
p_1	$bits(p_1, \mathbf{N/T})$
p_0	$bits(p_0, \mathbf{N})$

lookup cost $= \sum p_i$

Bloom filters	memory footprint
⋮	⋮
p_2	$bits(p_2, N/T^2)$
p_1	$bits(p_1, N/T)$
p_0	$bits(p_0, N)$

false positive rates

lookup cost $= \sum p_i$

$$\textbf{memory} = -\ c \cdot \boldsymbol{N} \cdot \sum \frac{\ln(\boldsymbol{p_i})}{\boldsymbol{T^i}}$$

false positive rates

size ratio

constant

entries

Bloom filters

false positive rates

$\vdots$

p_2

p_1

p_0

optimize

lookup cost $= \sum p_i$

memory $= - \; c \cdot N \cdot \sum \frac{\ln(p_i)}{T^i}$

Monkey

Bloom filters

false positive rates

⋮

p_0/T^2

p_0/T

p_0

exponential decrease

Monkey
Bloom filters

false positive rates

⋮

p_0/T^2

p_0/T

p_0

exponential decrease

State-of-the-Art
Bloom filters

p

p

p

same

Monkey
Bloom filters

State-of-the-Art
Bloom filters

false positive rates

⋮	…	<	…
	p_0/T^2	<	p
	p_0/T	<	p
	p_0	>	p

lookup cost $= \sum p_i$ < $= \sum p$

Monkey
Bloom filters

State-of-the-Art
Bloom filters

⋮

false positive rates

…	<	…
p_0/T^2	<	p
p_0/T	<	p
p_0	>	p

lookup cost $= \sum p_i$ < $= \sum p$

$= O(\mathbf{e^{-M/N}})$ $= O(\mathbf{\log(N) \cdot e^{-M/N}})$

N | number of entries
M | overall memory for Bloom filters

DASlab @ Harvard SEAS

Monkey
Bloom filters

State-of-the-Art
Bloom filters

⋮

false positive rates

…	<	…
p_0/T^2	<	p
p_0/T	<	p
p_0	>	p

lookup cost $= \sum p_i$ < $= \sum p$

$= O(\mathbf{e^{-M/N}})$ $= O(\mathbf{\log(N) \cdot e^{-M/N}})$

asymptotic win

lookup cost increases at slower rate as data grows

Monkey

Bloom filters

false positive rates

⋮

p_0/T^2

p_0/T

p_0

convergent geometric series

Monkey

Bloom filters

false positive rates

$\vdots$

p_0/T^2

p_0/T

p_0

$$\textbf{memory} = c \cdot entries \cdot \sum \frac{-\ln(p_i)}{T^i}$$

Monkey
Bloom filters

false positive rates

⋮

p_0/T^2

p_0/T

p_0

memory = c · *entries* · −ln(**lookup cost**)

Monkey
Bloom filters

false positive rates

⋮

p_0/T^2

p_0/T

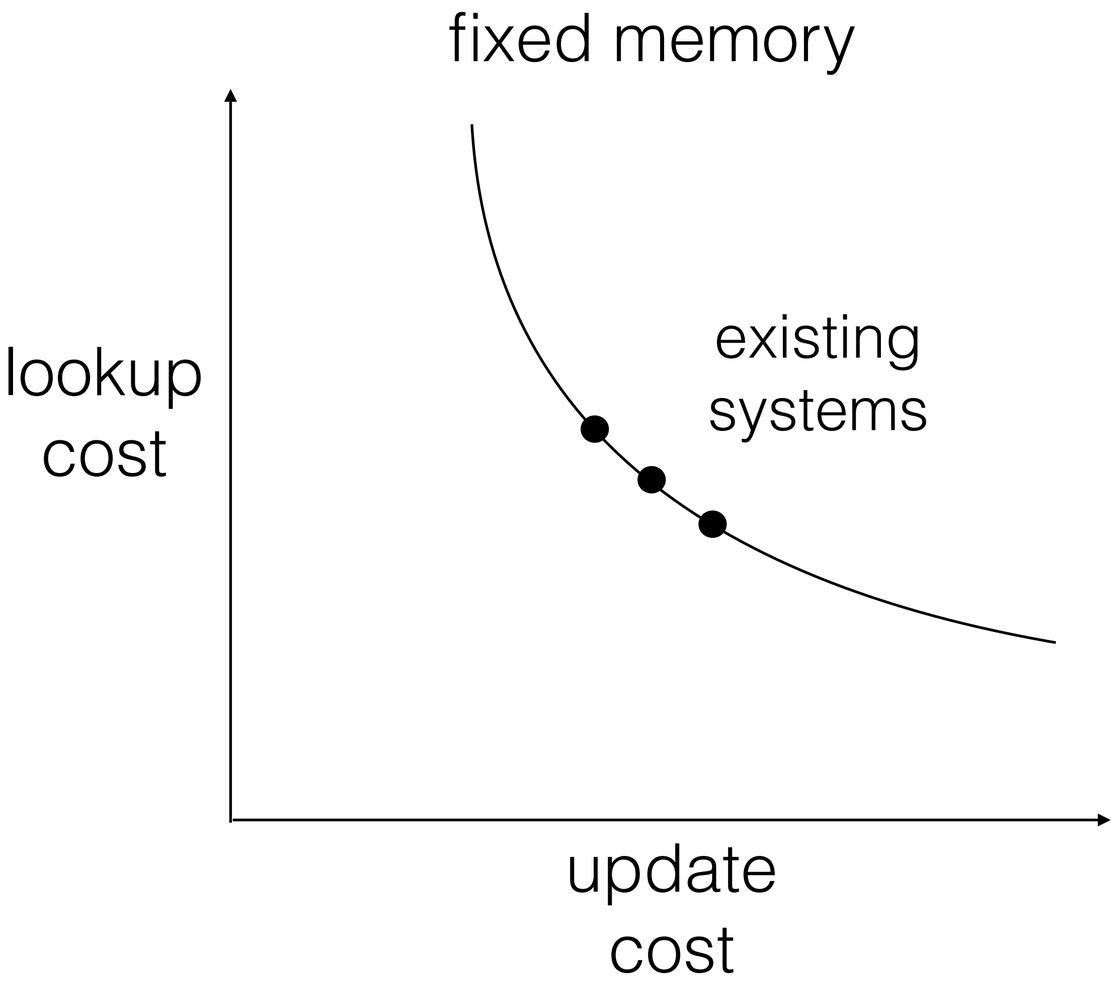
p_0

memory = c · *entries* · −ln(**lookup cost**)

model lookups vs. memory trade-off

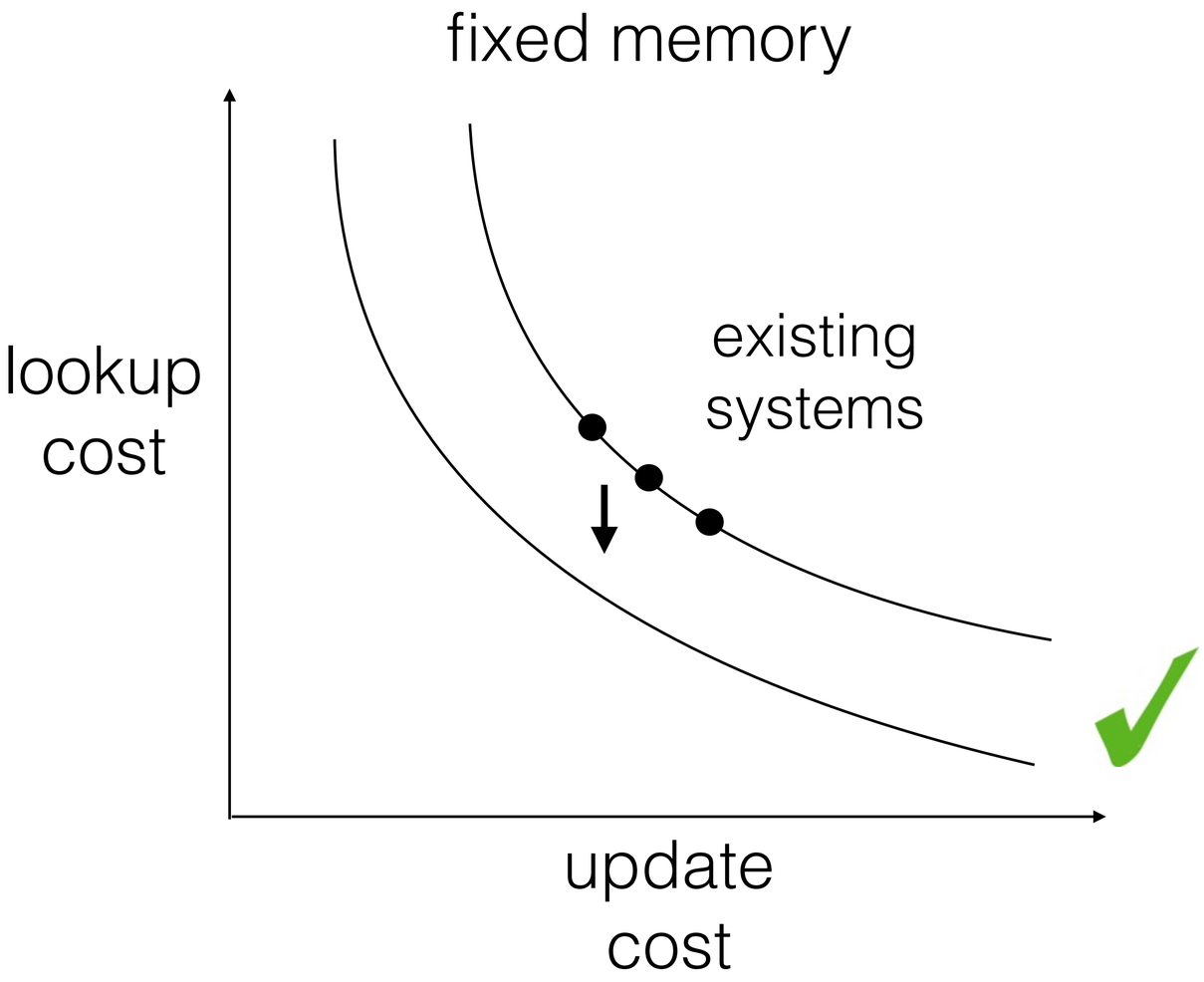
Problem 1: suboptimal filters allocation

Problem 2: hard to tune



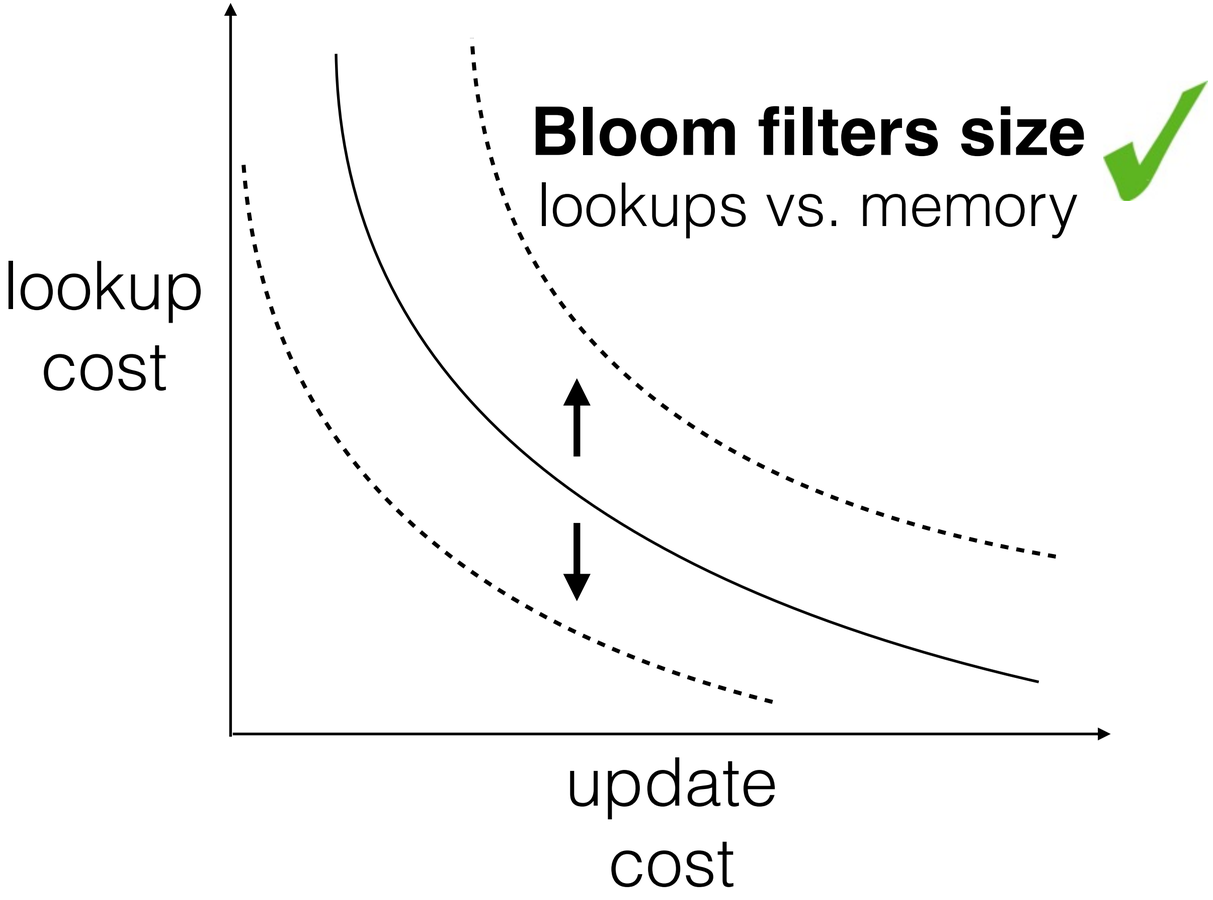
Problem 1: **suboptimal filters allocation**

Problem 2: hard to tune



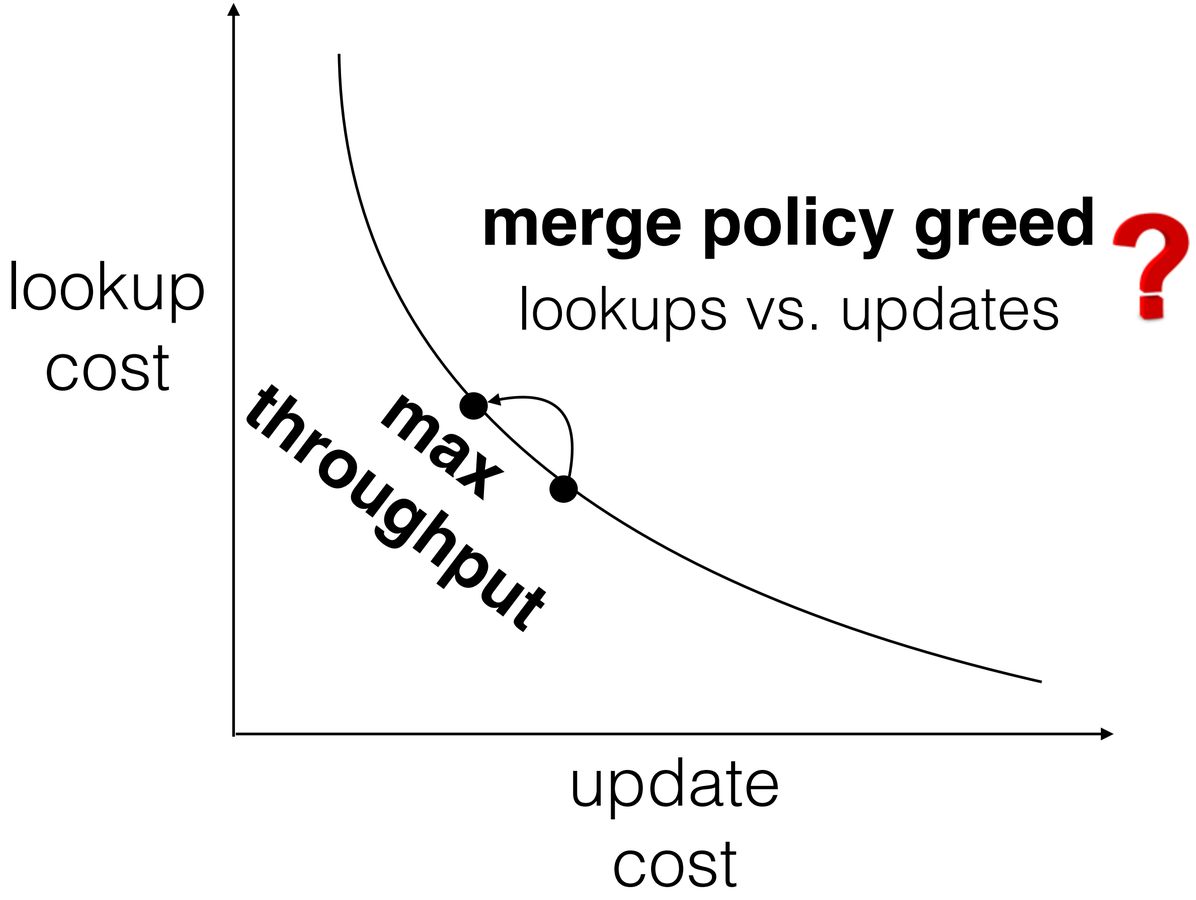
Problem 1: suboptimal filters allocation

Problem 2: **hard to tune**



Problem 1: suboptimal filters allocation

Problem 2: **hard to tune**



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

observations:	filters fixed false positive rates	LSM-tree merge policy ↓ ? performance	ad-hoc trade-offs memory lookups updates
insights:	lookup cost = $\sum p_i$ suboptimal	log sorted array	lookup cost existing Monkey update cost
steps:	optimize allocation asymptotically better memory vs. lookups	updates vs. lookups navigate	answer what-if design questions

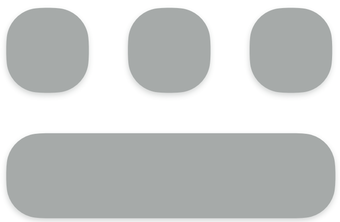
DASlab @ Harvard SEAS

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

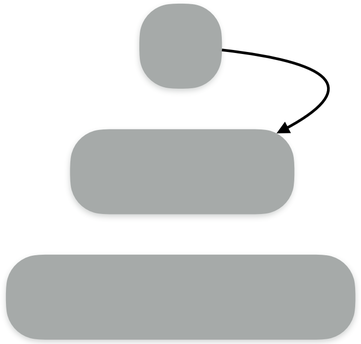
observations:	filters fixed false positive rates	LSM-tree merge policy ↓ ? performance	ad-hoc trade-offs memory lookups updates
insights:	lookup cost = $\sum p_i$ suboptimal	log sorted array	lookup cost existing Monkey update cost
steps:	optimize allocation asymptotically better memory vs. lookups	updates vs. lookups navigate	answer what-if design questions

DASlab @ Harvard SEAS

Identify

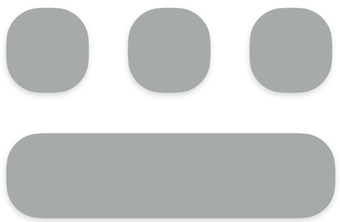


merge policy

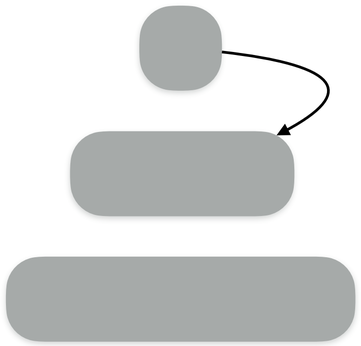


size ratio

Identify

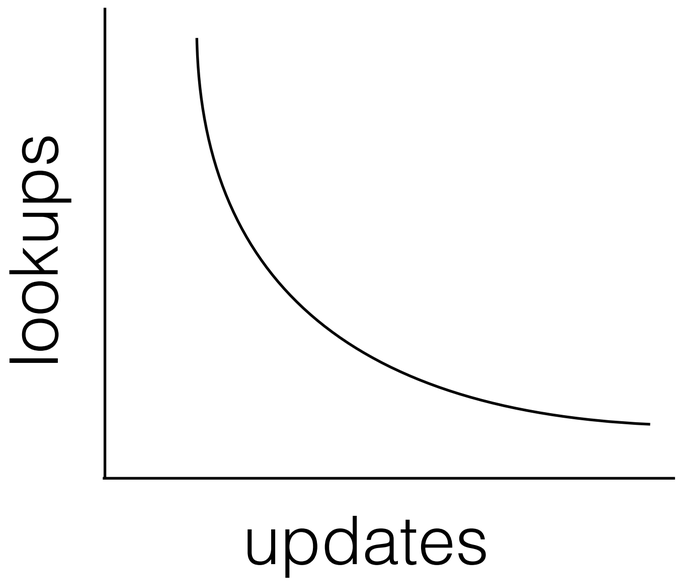


merge policy



size ratio

Map



Identify

Map

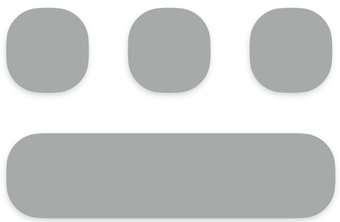
merge policy

log ⟷ LSM-tree ⟷ sorted array

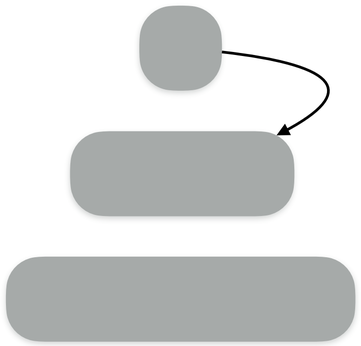
updates

DASlab @ Harvard SEAS

Identify

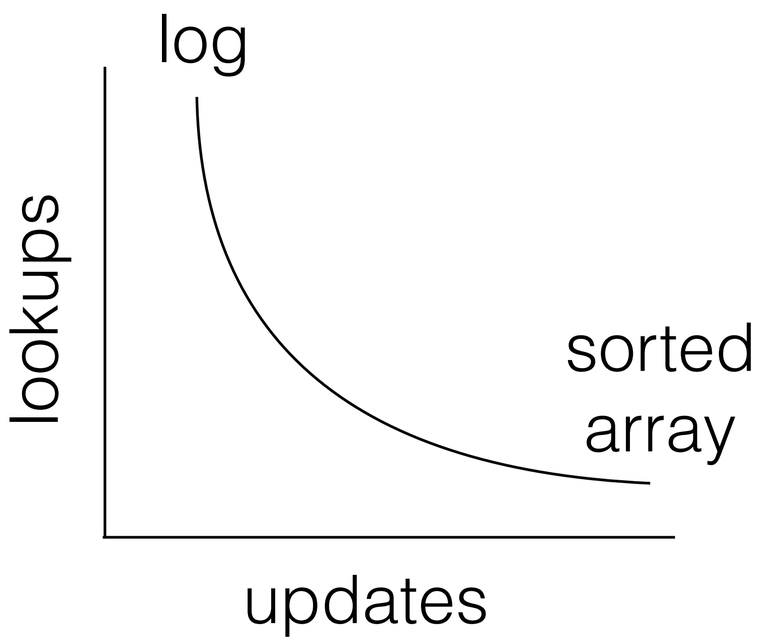


merge policy

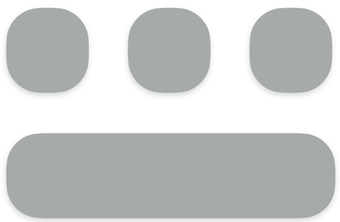


size ratio

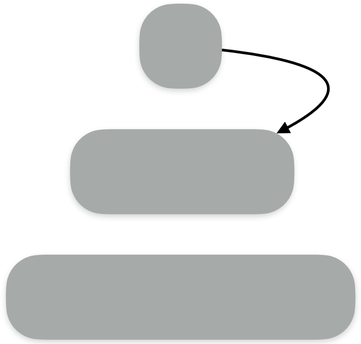
Map



Identify

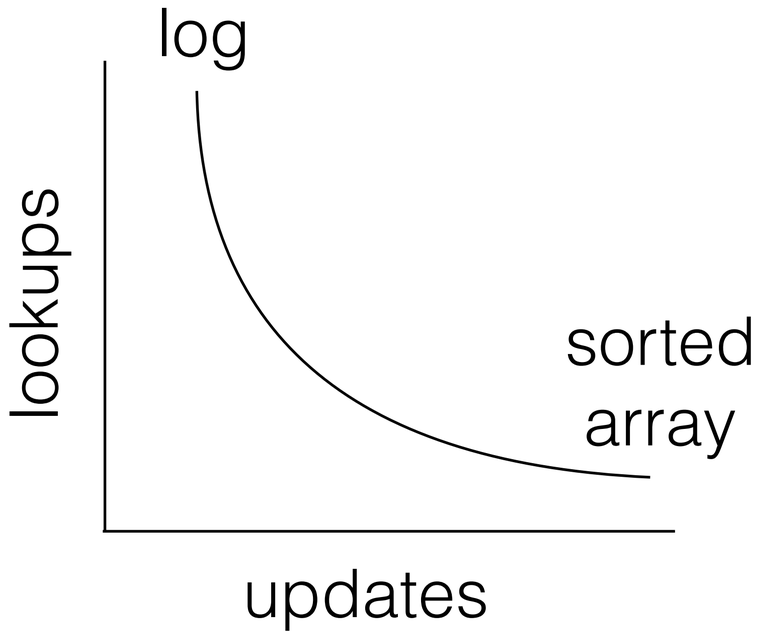


merge policy

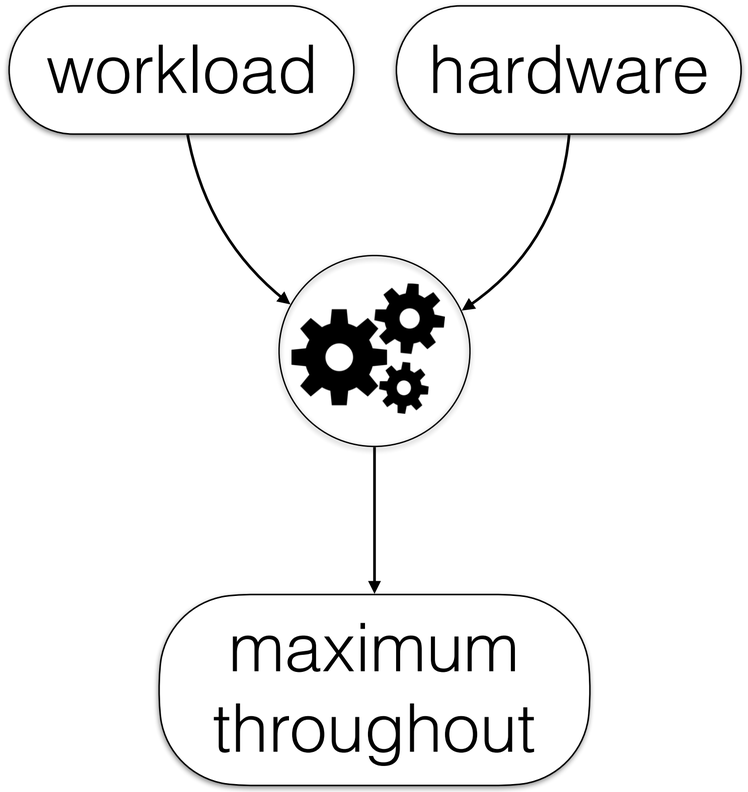


size ratio

Map



Navigate



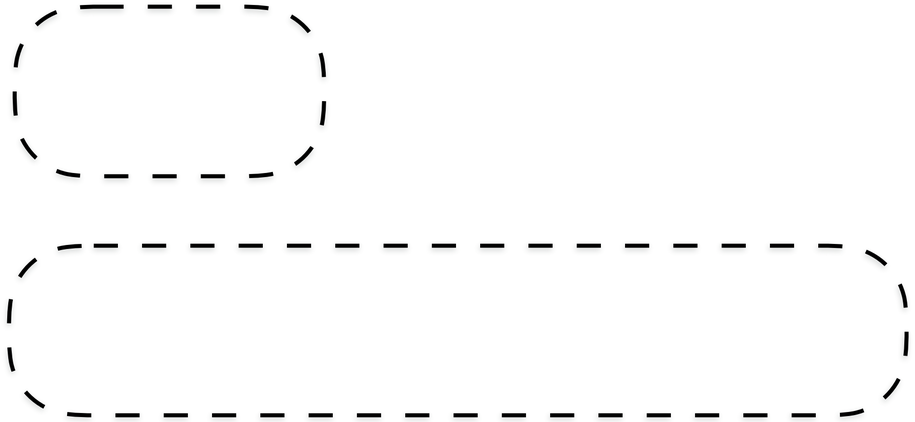
Merge Policies

Tiering
write-optimized

Leveling
read-optimized

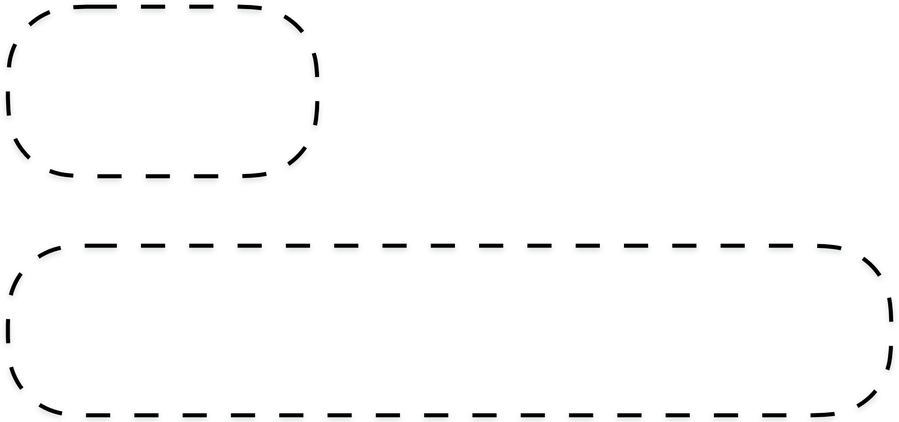
Tiering

write-optimized



Leveling

read-optimized

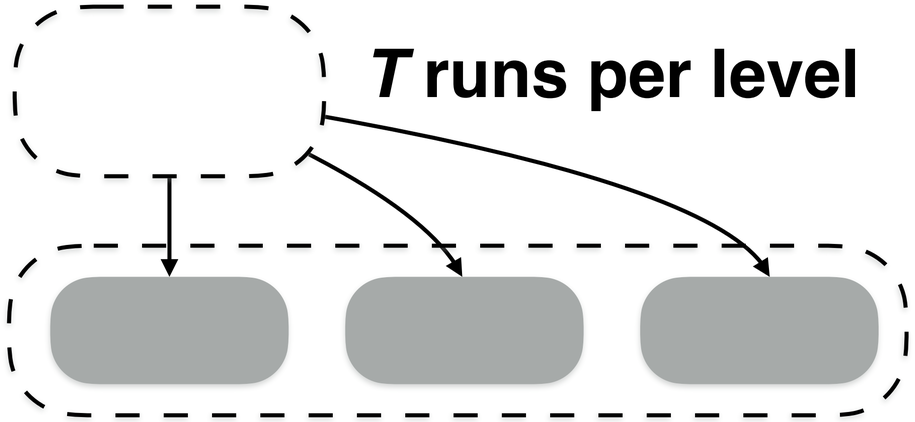


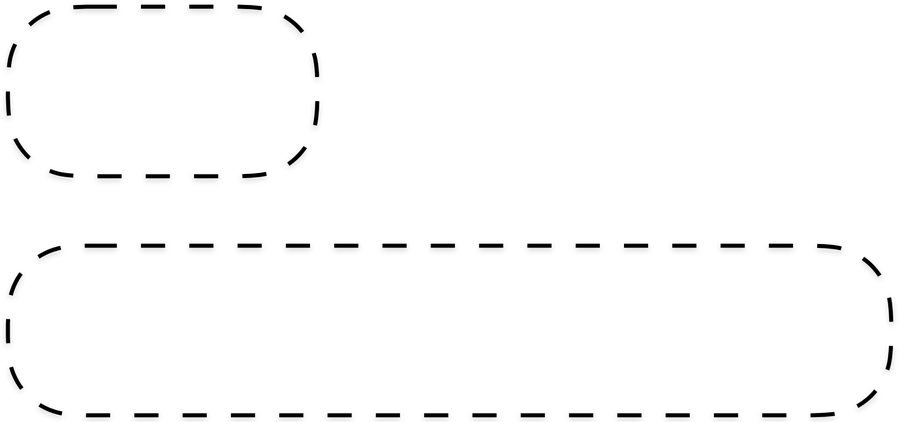
Tiering

write-optimized

Leveling

read-optimized



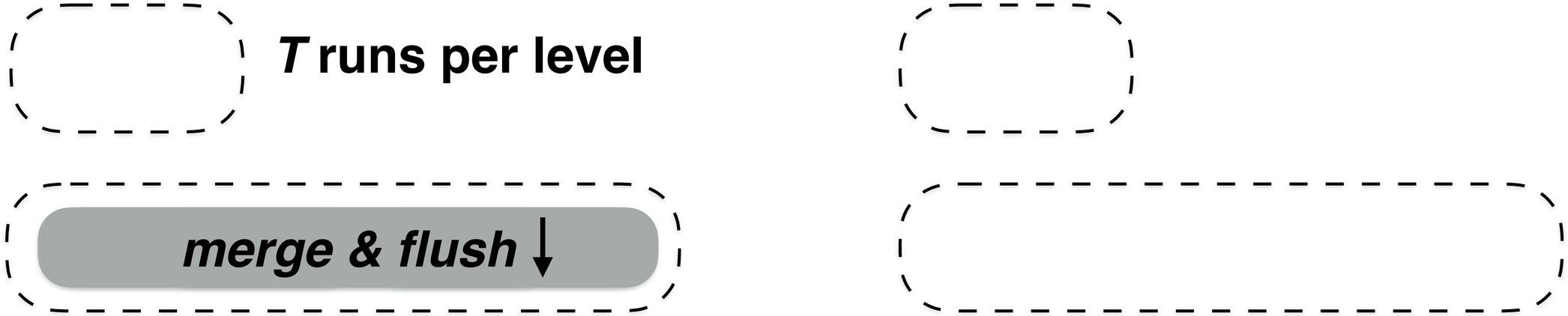


Tiering

write-optimized

Leveling

read-optimized

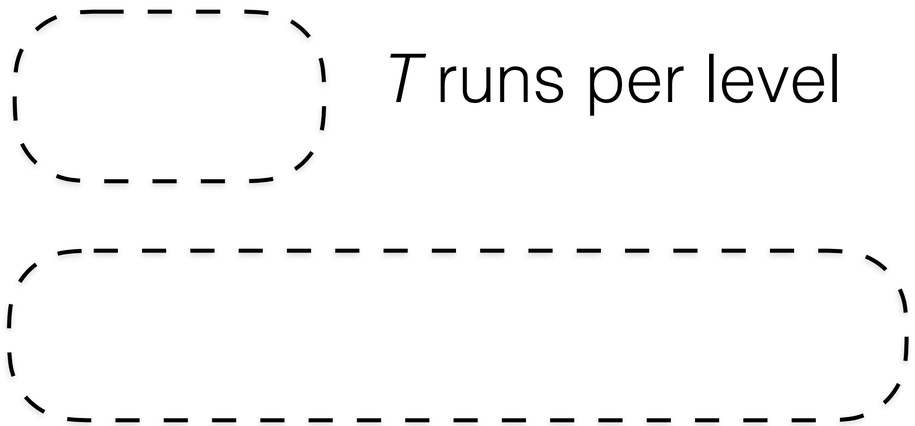


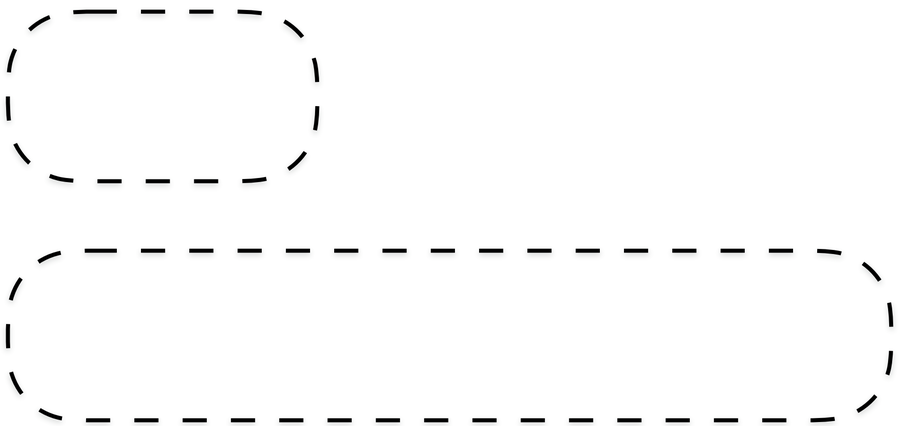
Tiering

write-optimized

Leveling

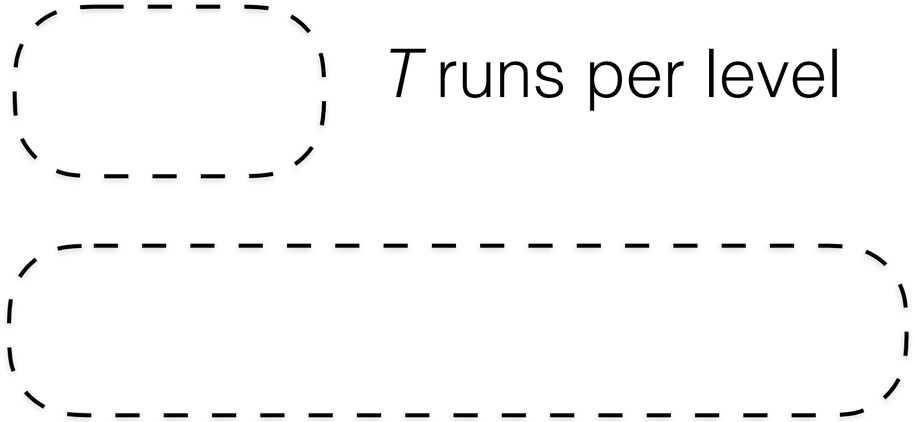
read-optimized





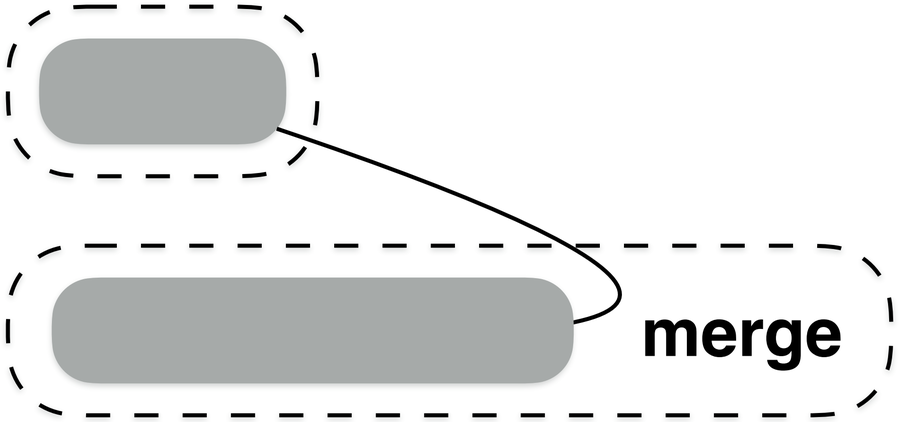
Tiering

write-optimized



Leveling

read-optimized

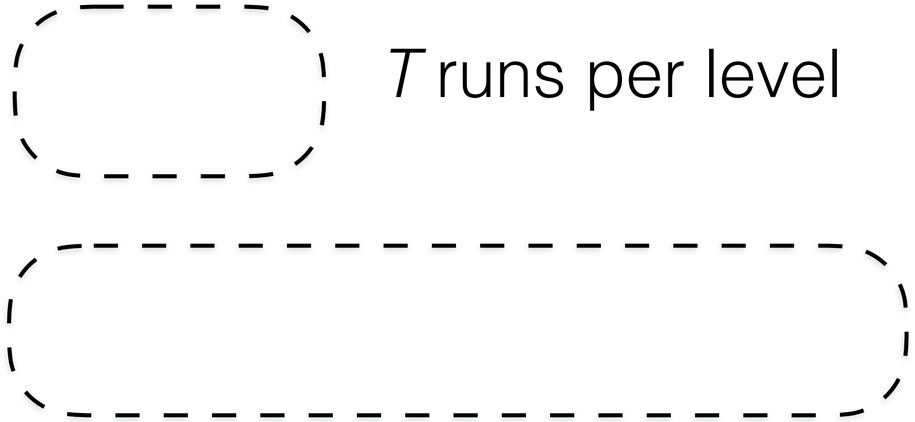


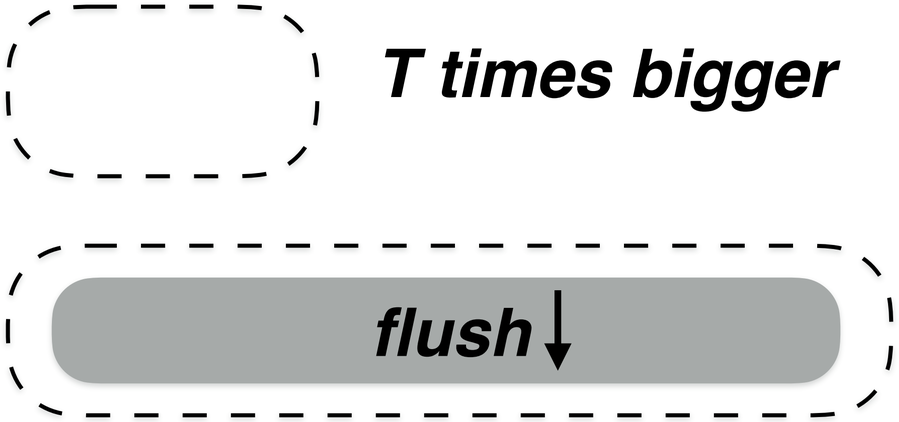
Tiering

write-optimized

Leveling

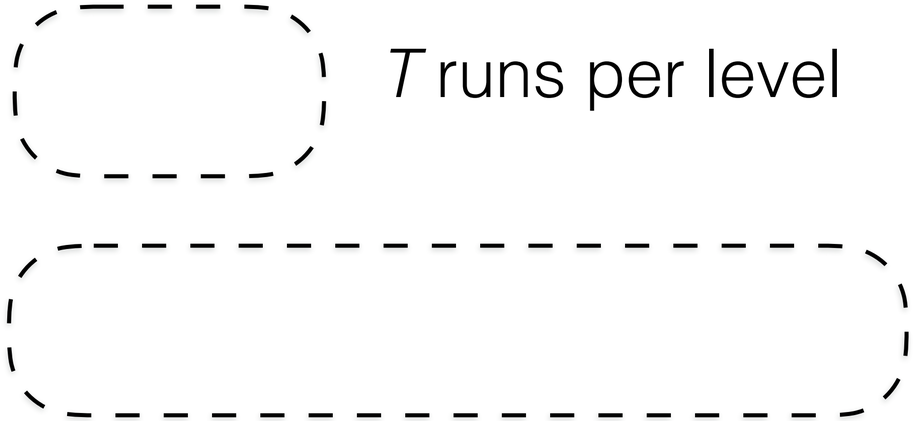
read-optimized





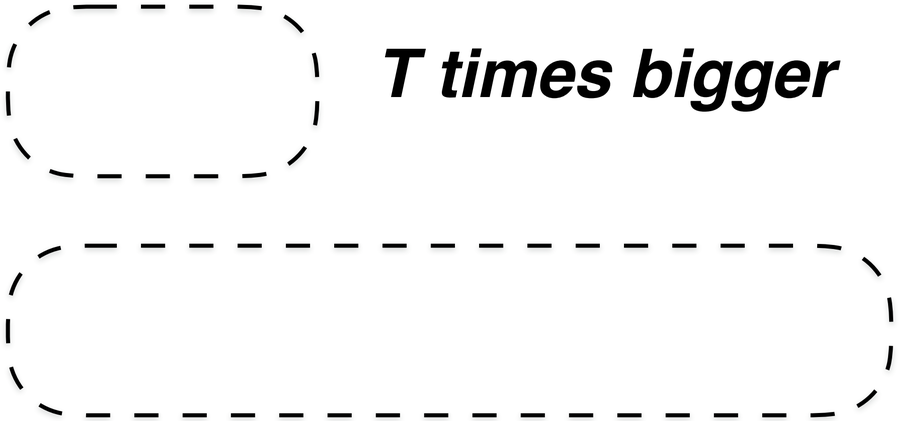
Tiering

write-optimized



Leveling

read-optimized



Tiering

write-optimized

Leveling

read-optimized

Tiering

write-optimized

Leveling

read-optimized

lookup cost:

$$O(\boldsymbol{T} \cdot \log_{\boldsymbol{T}}(\boldsymbol{N}) \cdot e^{-\boldsymbol{M/N}})$$

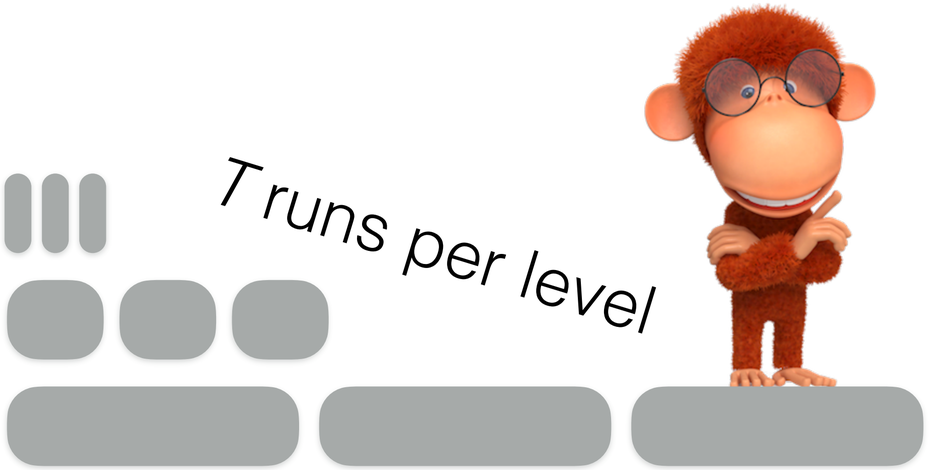
runs per level — *levels* — *false positive rate*

$$O(\log_{\boldsymbol{T}}(\boldsymbol{N}) \cdot e^{-\boldsymbol{M/N}})$$

levels — *false positive rate*

Tiering

write-optimized



Leveling

read-optimized

lookup cost:

$$O(\boldsymbol{T} \cdot \log_{T}(N) \cdot e^{-M/N})$$

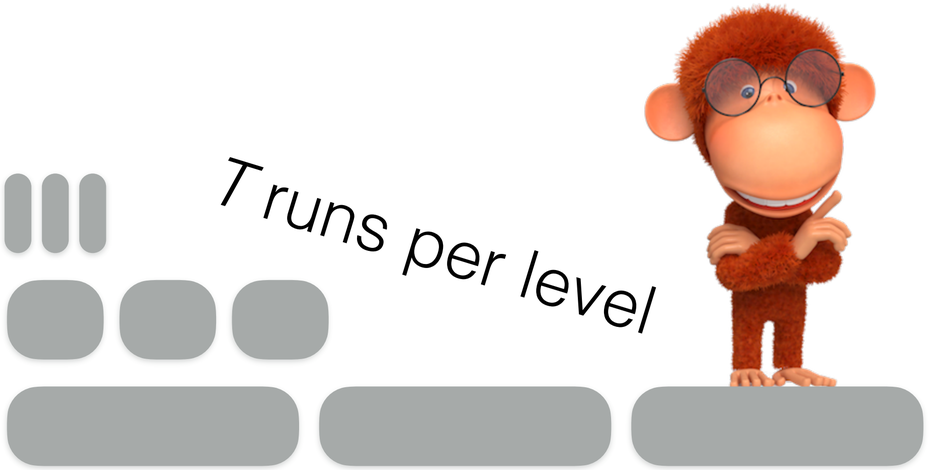
runs per level — ***levels*** — *false positive rate*

$$O(\log_{T}(N) \cdot e^{-M/N})$$

levels — *false positive rate*

Tiering

write-optimized



Leveling

read-optimized

lookup cost:

$O(\boldsymbol{T} \cdot \mathbf{e}^{-M/N})$

runs per level

false positive rate

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

false positive rate

	Tiering write-optimized	Leveling read-optimized
		
lookup cost:	$O(T \cdot e^{-M/N})$	$O(e^{-M/N})$
update cost:	$O(\mathbf{\log_T(N)})$ ↑ *levels*	$O(\mathbf{T \cdot \log_T(N)})$ ↑ *merges per level* ↑ *levels*

DASlab @ Harvard SEAS

	Tiering write-optimized	Leveling read-optimized
		
lookup cost:	$O(T \cdot e^{-M/N})$	$O(e^{-M/N})$
update cost:	$O(\log_T(N))$	$O(T \cdot \log_T(N))$

size ratio *T* ⌄

Tiering

write-optimized

1 run per level

Leveling

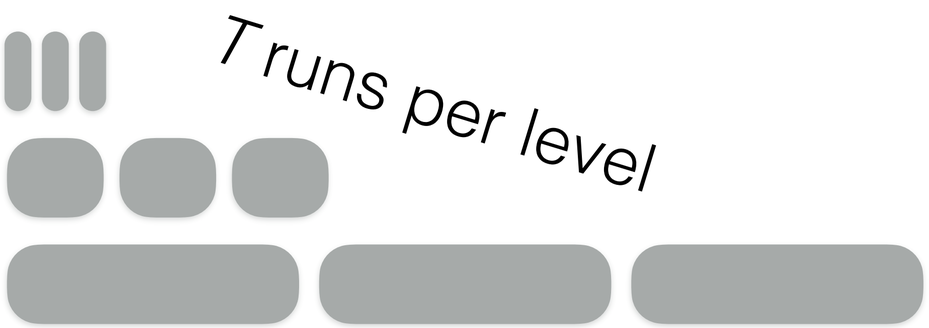
read-optimized

1 run per level

lookup cost: $O(e^{-M/N}) = O(e^{-M/N})$

update cost: $O(\log(N)) = O(\log(N))$

size ratio *T* ⋙

	Tiering write-optimized	Leveling read-optimized
		
lookup cost:	$O(T \cdot e^{-M/N})$	$O(e^{-M/N})$
update cost:	$O(\log_T(N))$	$O(T \cdot \log_T(N))$

size ratio *T* ⌃

	Tiering write-optimized	Leveling read-optimized
	O(N) runs per level	*1* run per level
lookup cost:	$O(N \cdot e^{-M/N})$	$O(e^{-M/N})$
update cost:	$O(1)$	$O(N)$

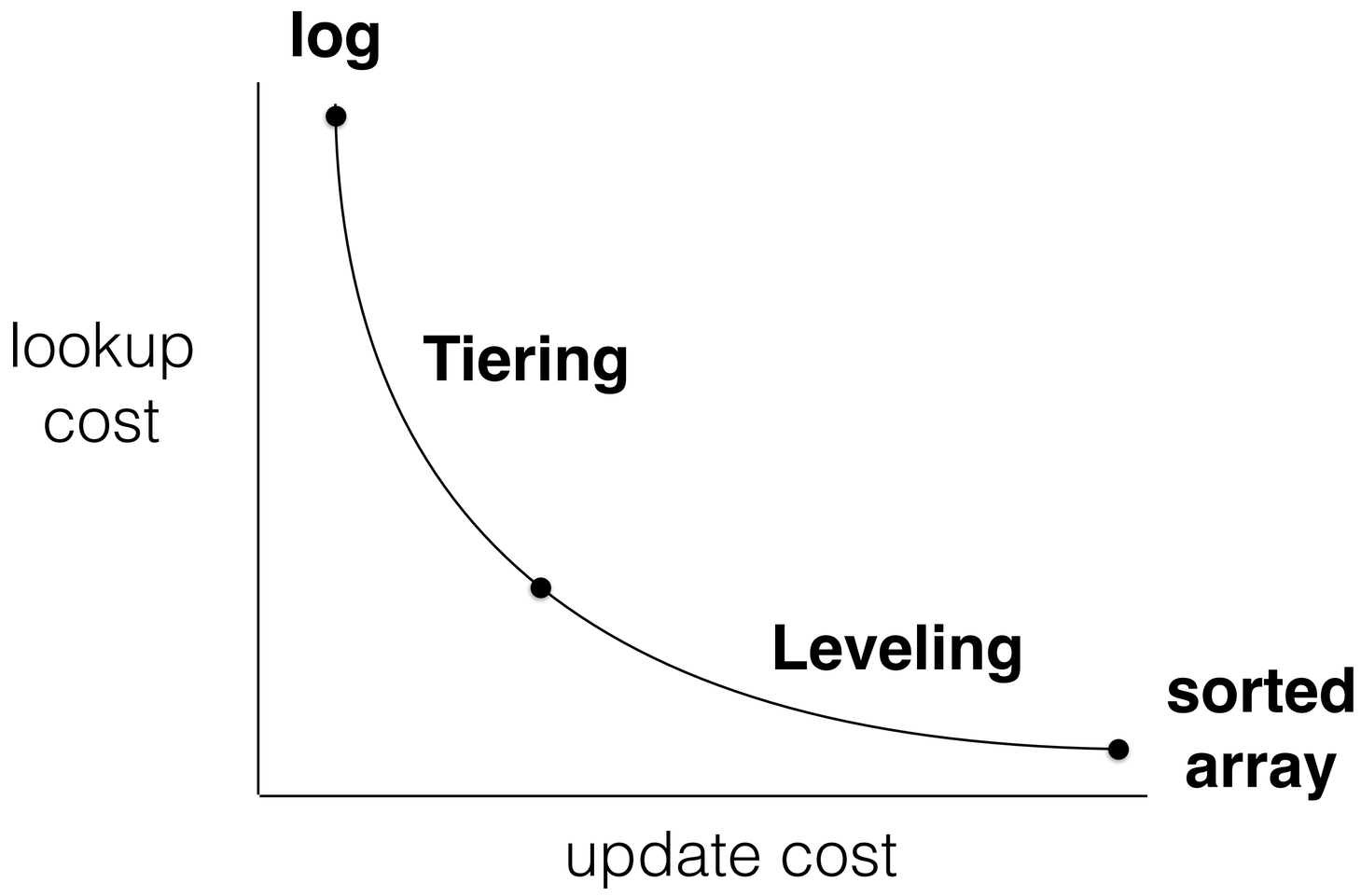
size ratio *T* ⌃

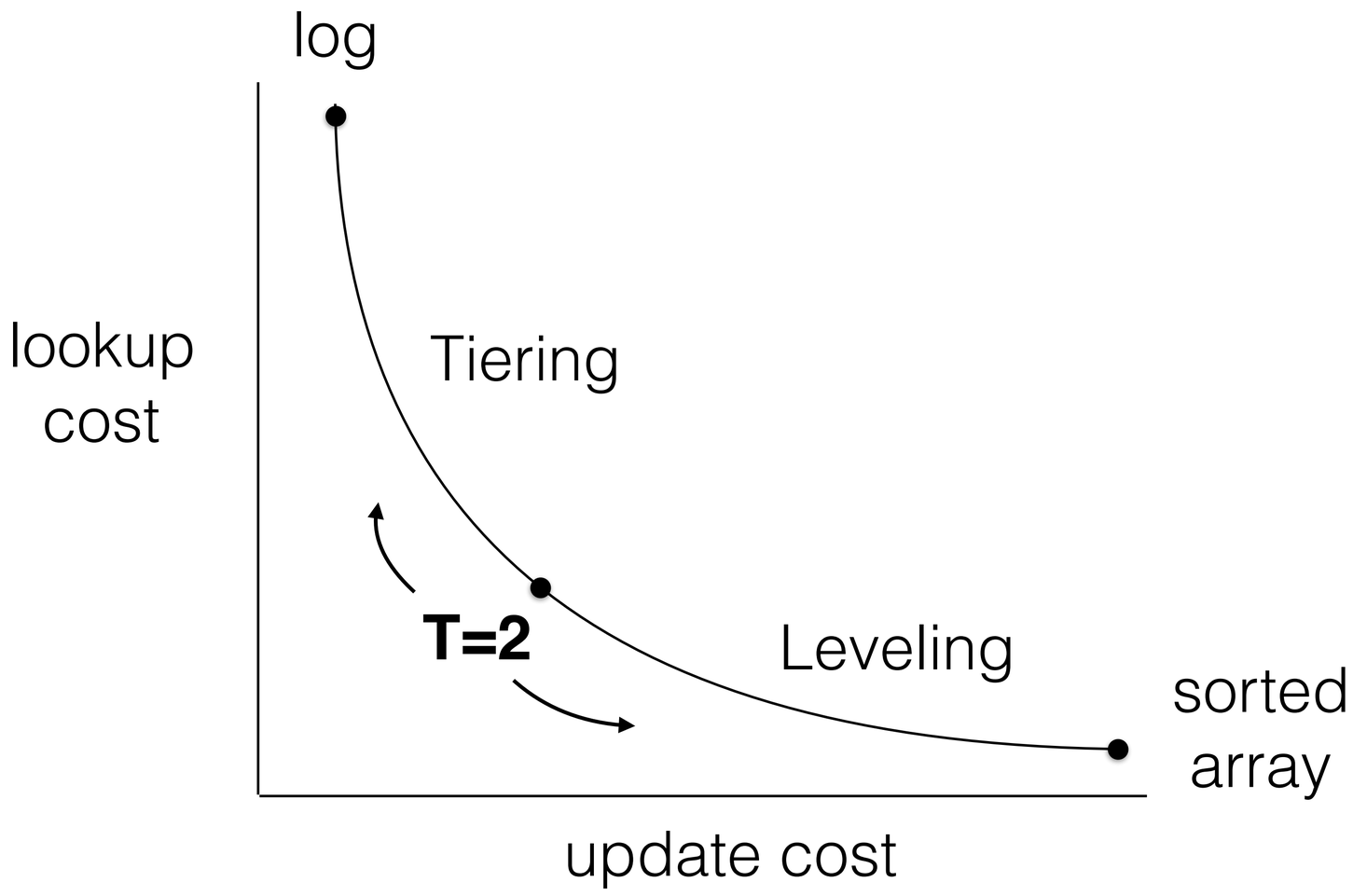
DASlab @ Harvard SEAS

	Tiering write-optimized	Leveling read-optimized
	O(*N*) runs per level	*1* run per level
	log	**sorted array**
lookup cost:	$O(N \cdot e^{-M/N})$	$O(e^{-M/N})$
update cost:	$O(1)$	$O(N)$

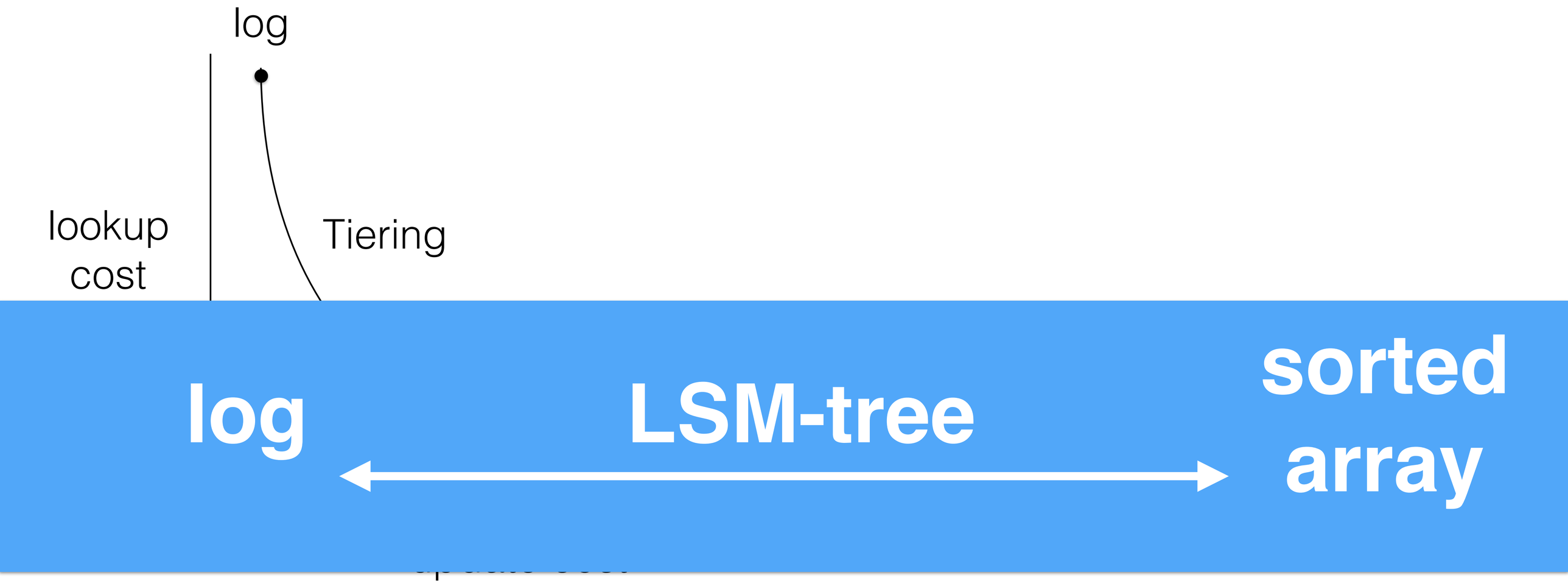
size ratio *T* ⌃

DASlab @ Harvard SEAS

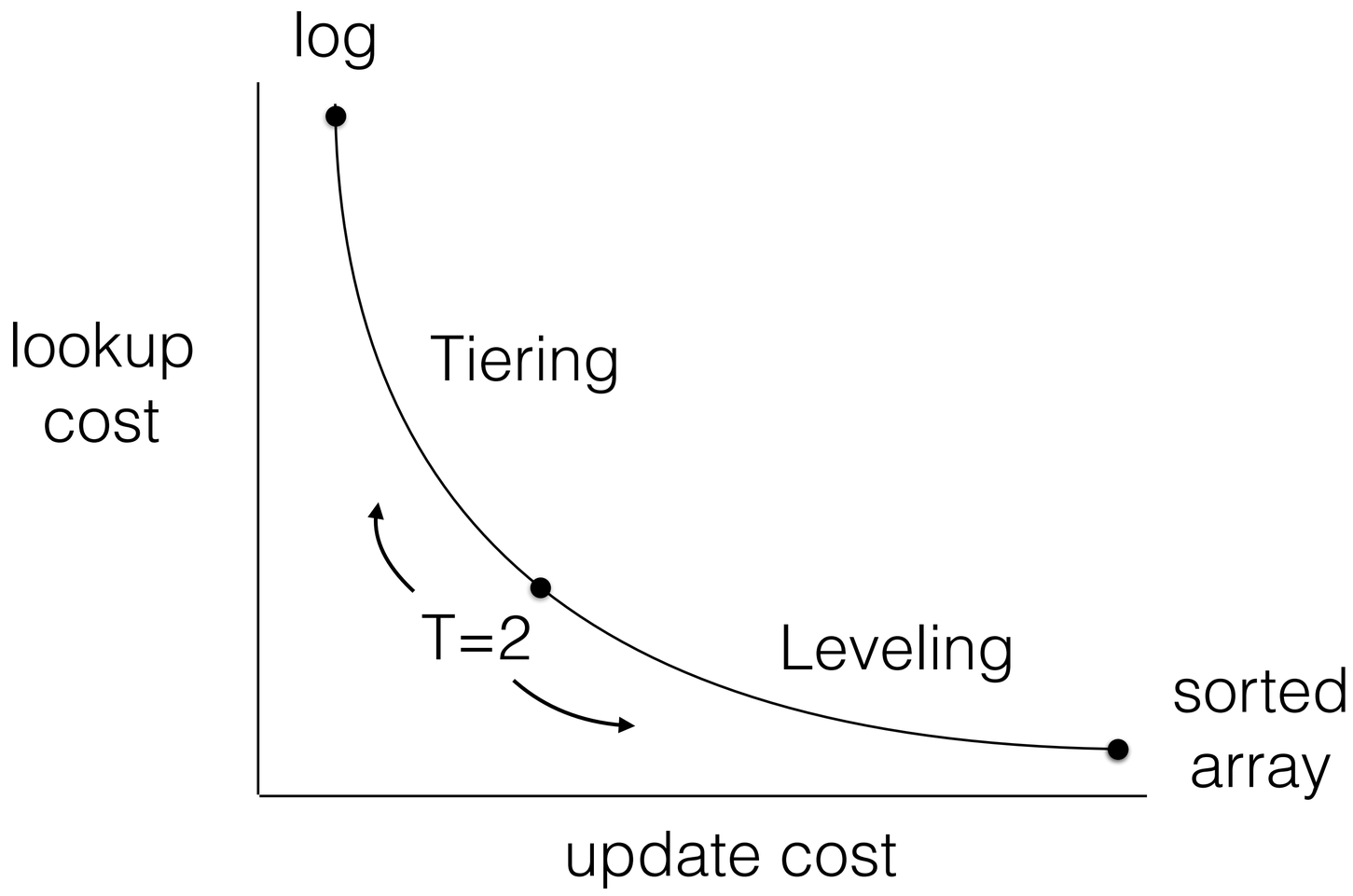

log
lookup cost
Tiering
Leveling
sorted array
update cost


log
lookup cost
Tiering
T=2
Leveling
sorted array
update cost

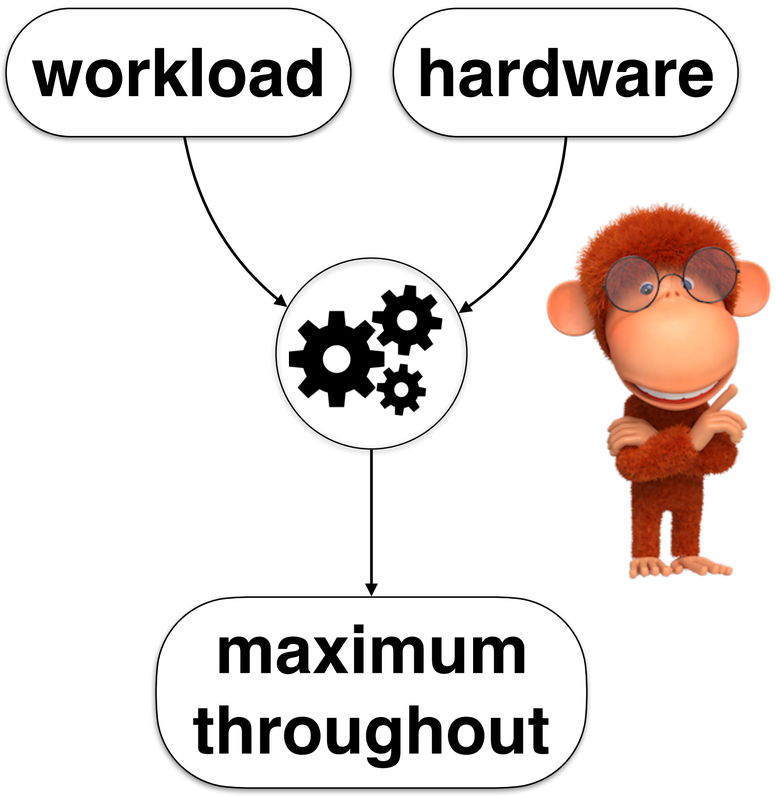
T | size ratio



T | size ratio

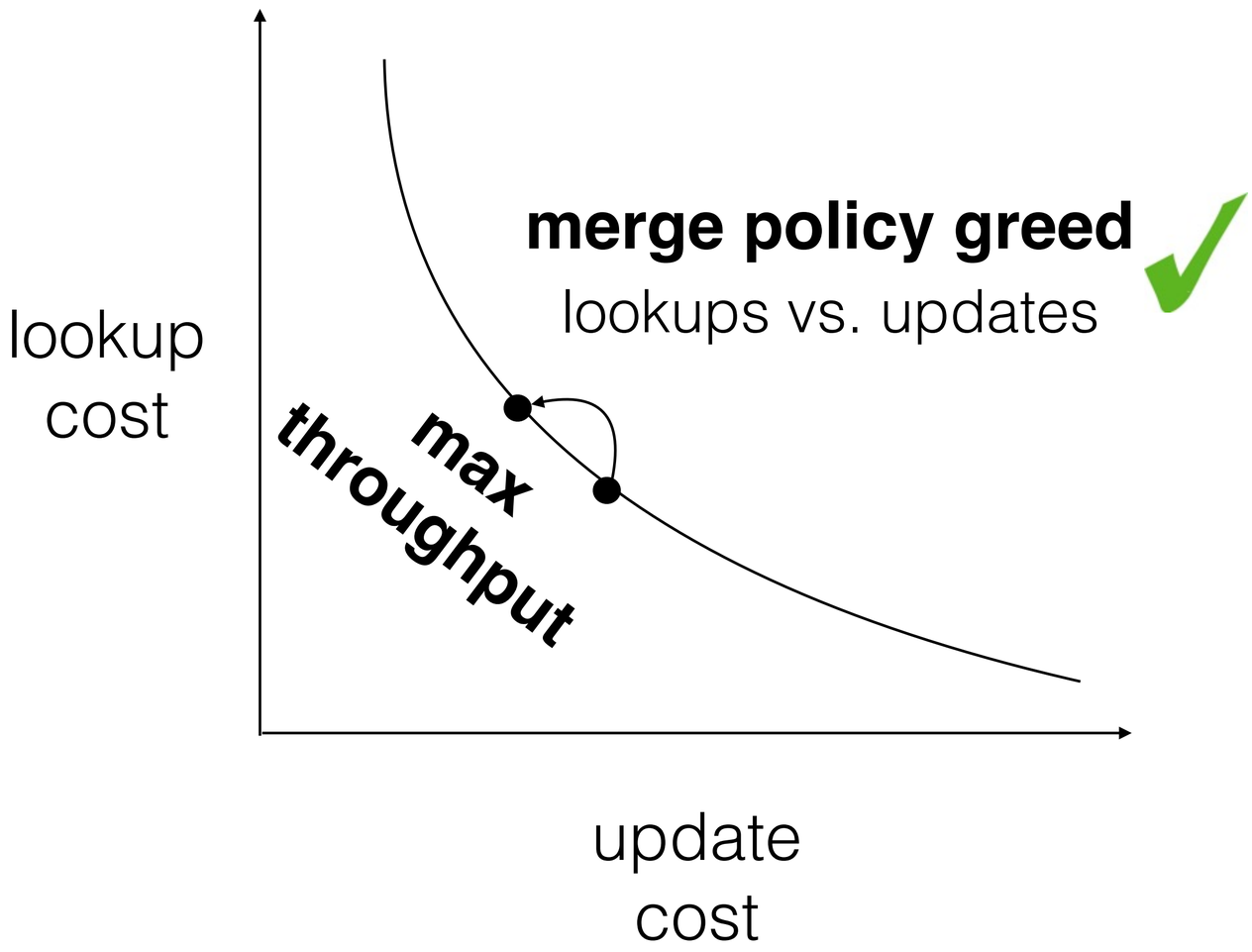

log
lookup cost
Tiering
T=2
Leveling
sorted array
update cost

T | size ratio


workload
hardware
maximum throughout

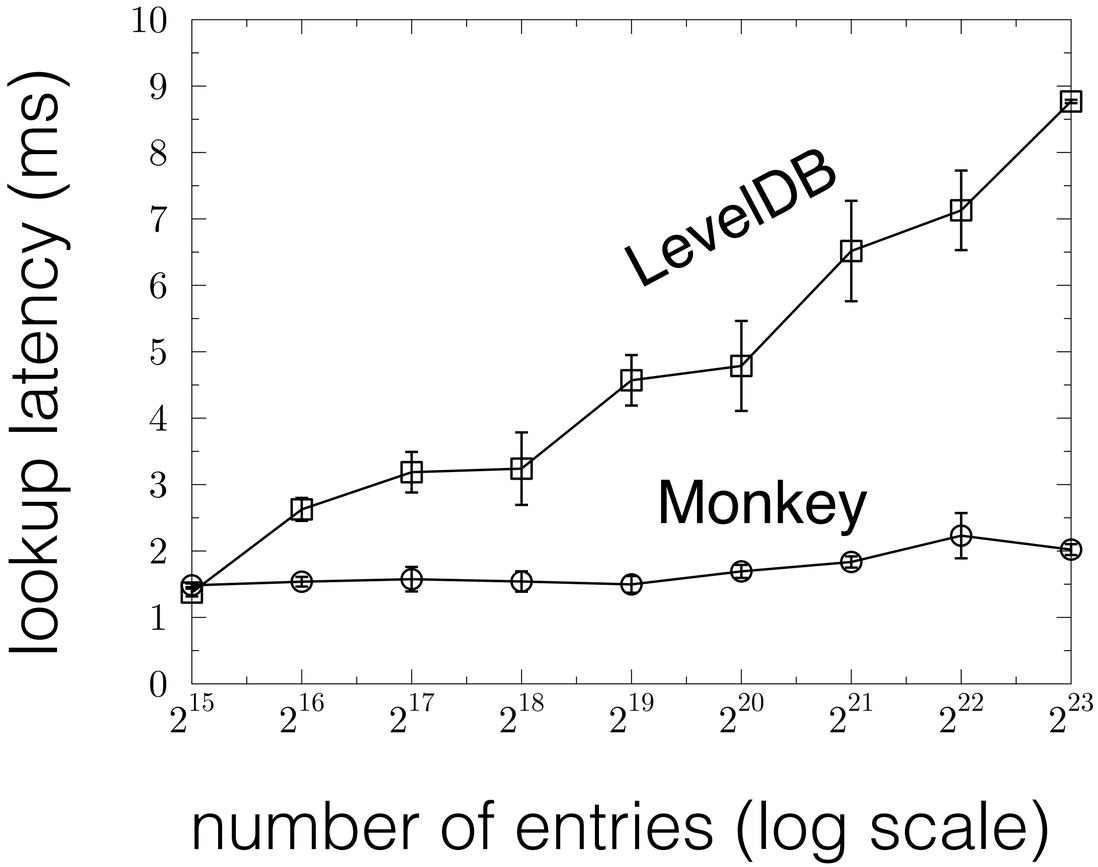
Problem 1: suboptimal filters allocation

Problem 2: **hard to tune**



levelDB

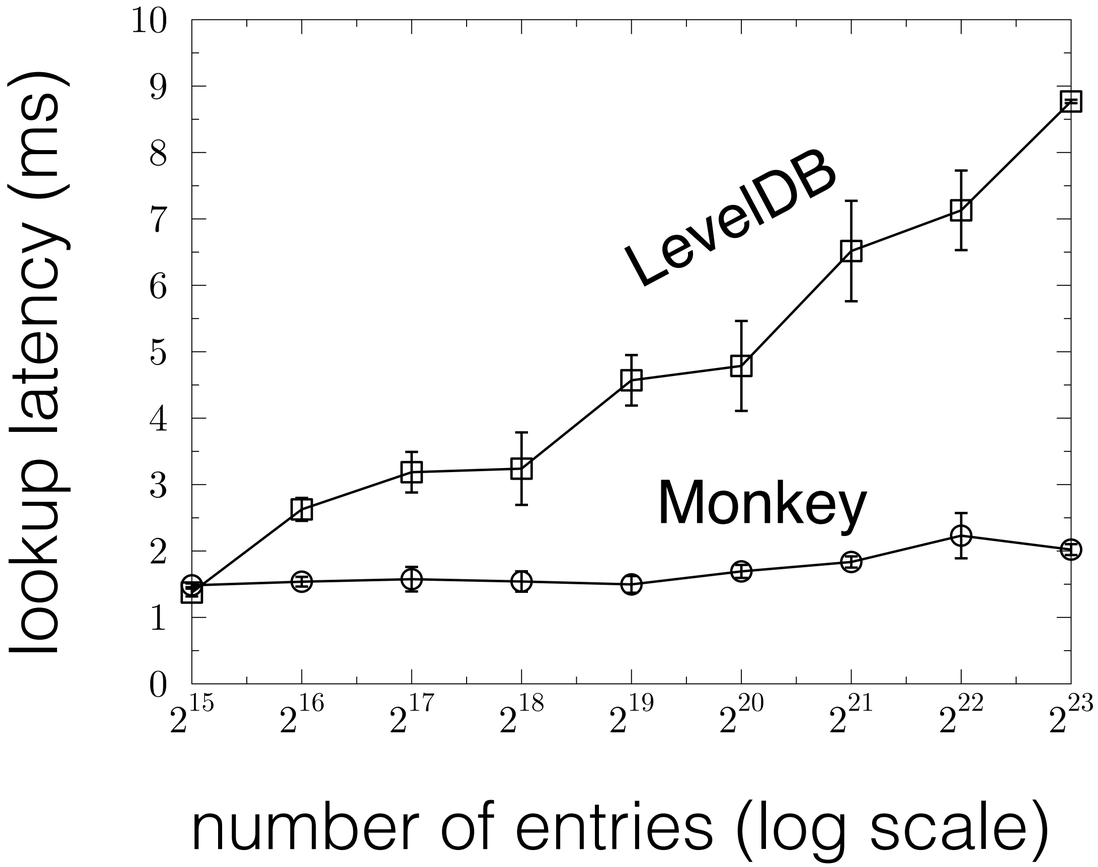
better asymptotic scalability



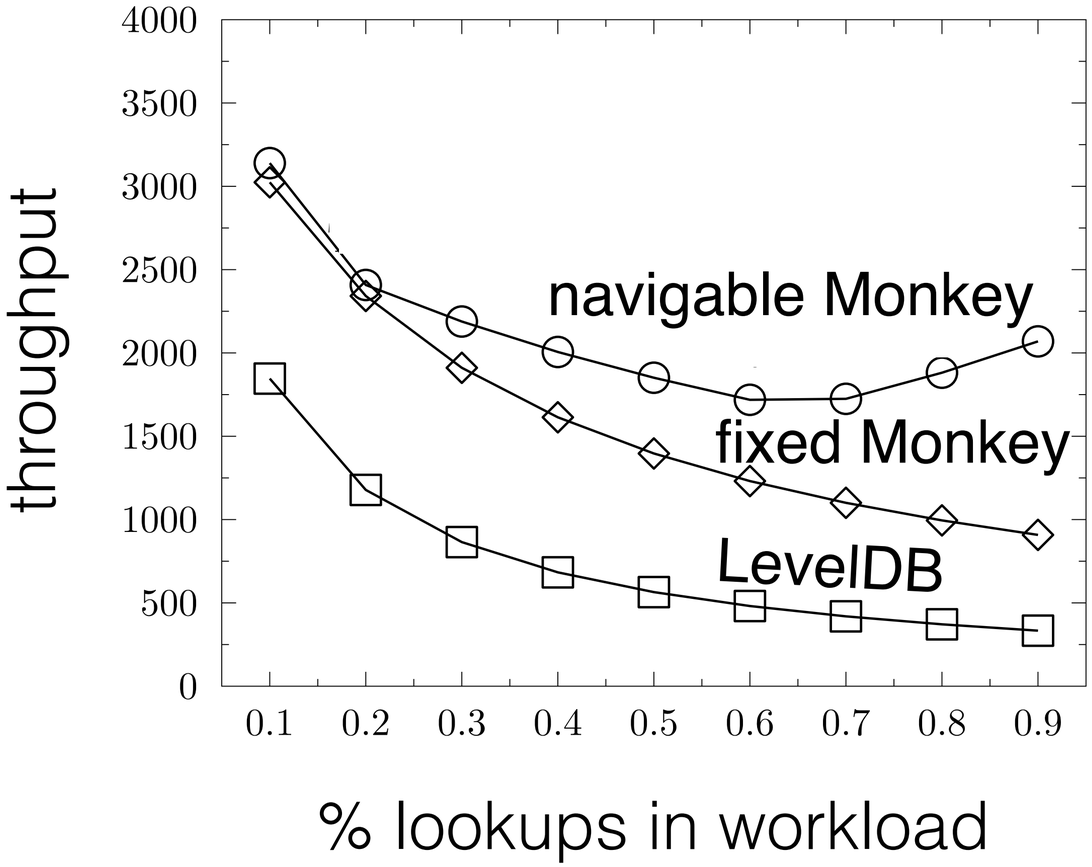
levelDB

DASlab @ Harvard SEAS

better asymptotic scalability



workload adaptability



DASlab @ Harvard SEAS

CrimsonDB

Map Trade-Offs

Navigate

What-if?

http://daslab.seas.harvard.edu/crimsondb/

DASlab @ Harvard SEAS

CrimsonDB

Map Trade-Offs

Navigate

What-if?

http://daslab.seas.harvard.edu/crimsondb/

Thanks!

DASlab @ Harvard SEAS