Static Filters

Research Topics in Database Management

Niv Dayan

What is a Filter?

What is a Filter?

Does X exist? →

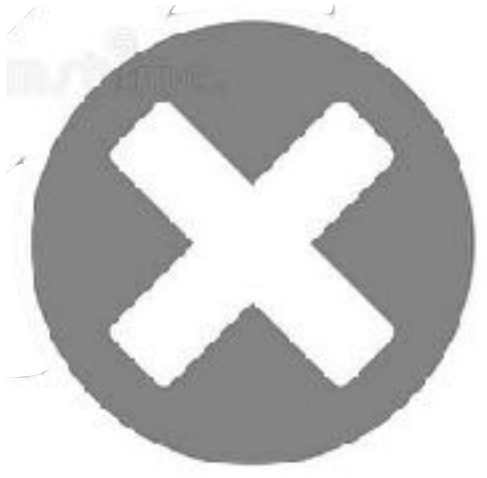
Set

X Y Z

What is a Filter?

Does X exist? →  → Set: X Y Z

No false negatives

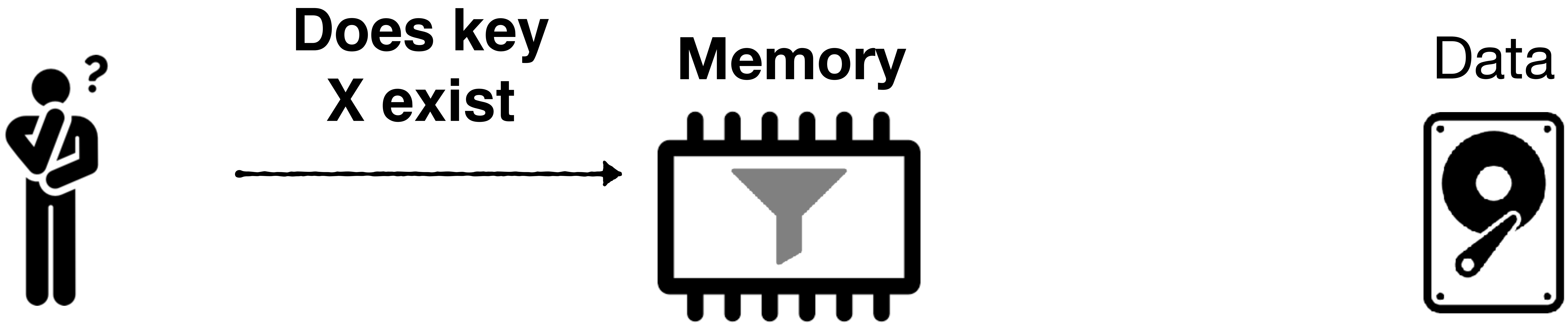


false positives with tunable probability

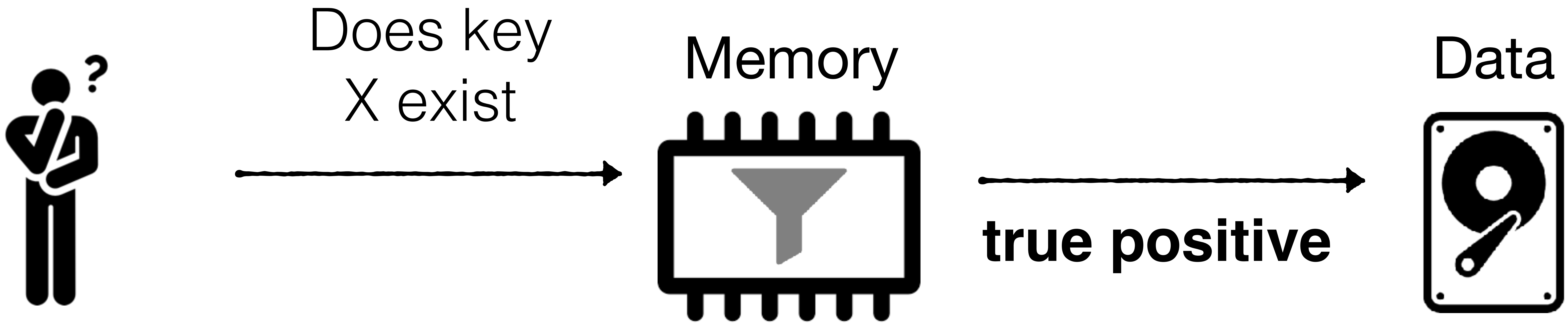
Why use a Filter?

Data

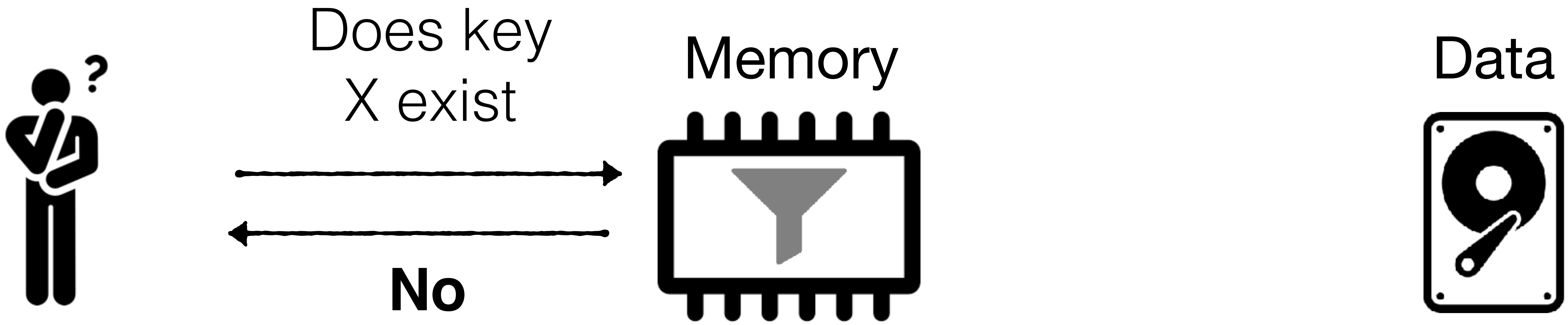

Data
Does key X exist


Does key
X exist
Memory
Data

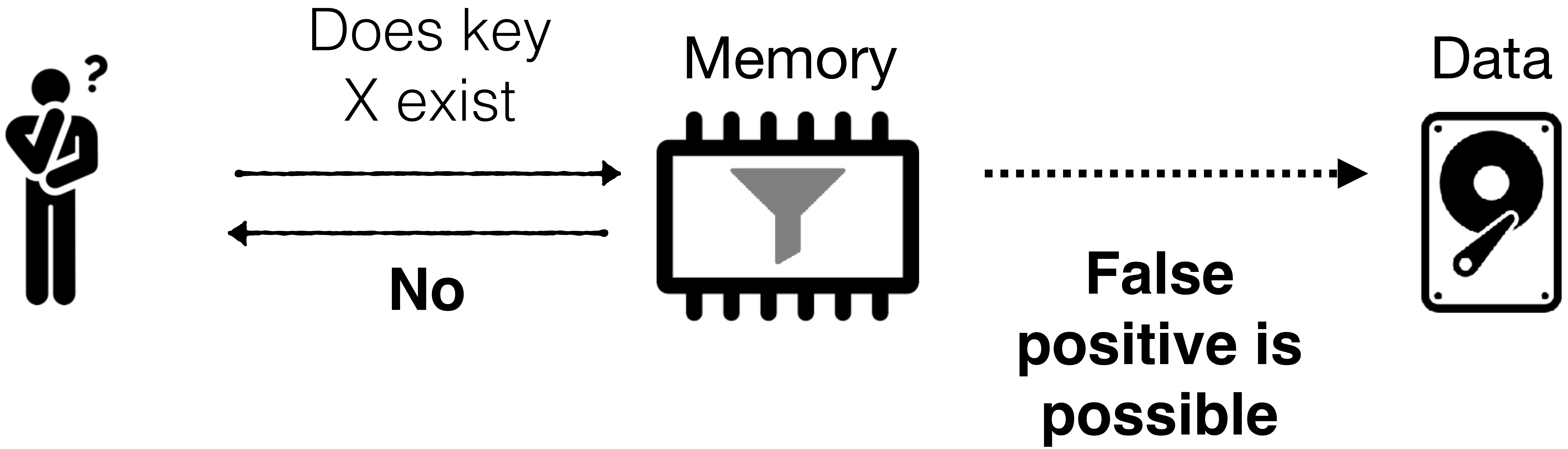
If key X exists



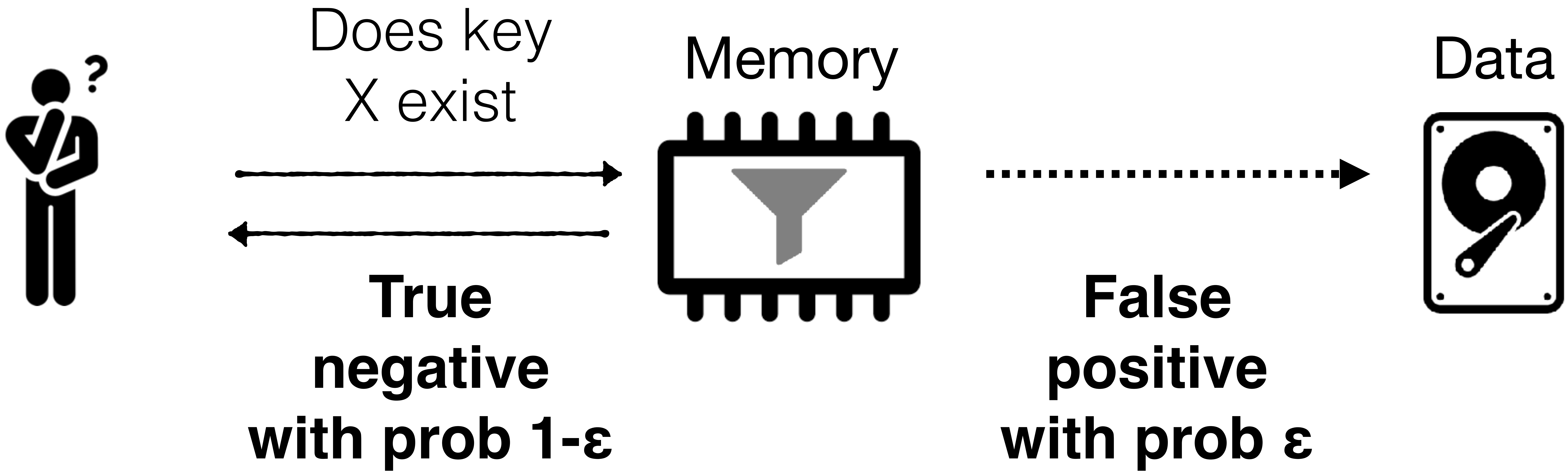
If key X does not exist



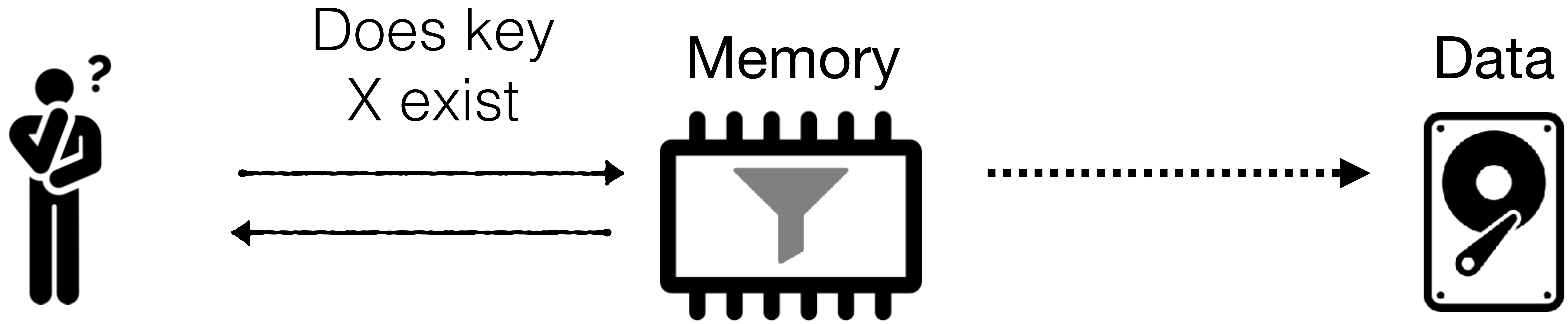
If key X does not exist



If key X does not exist



If key X does not exist

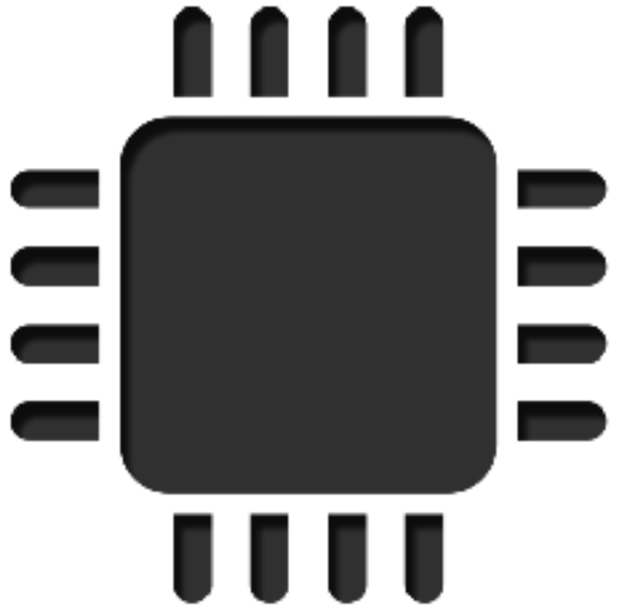


ε - false positive rate - FPR

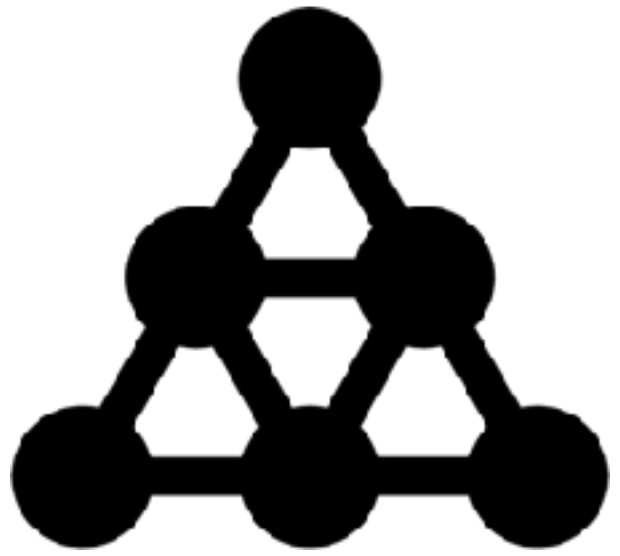
Why care about filters?

Widely used in systems

Hardware Optimizations



Algorithmic Reasoning/ Techniques

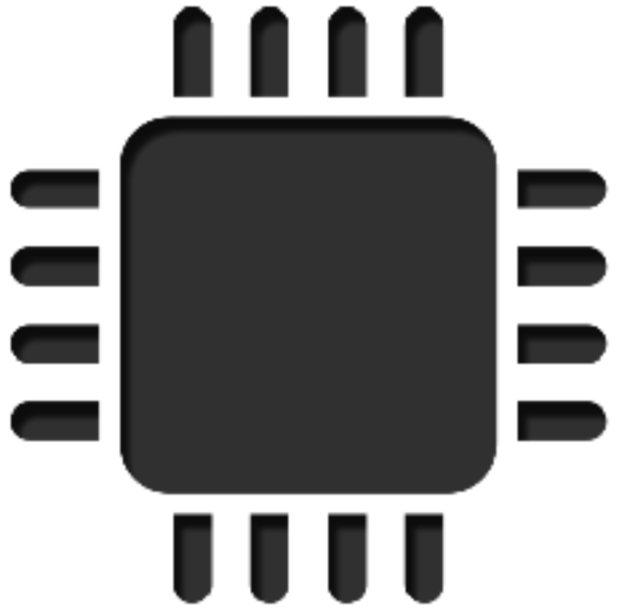


Why care about filters?

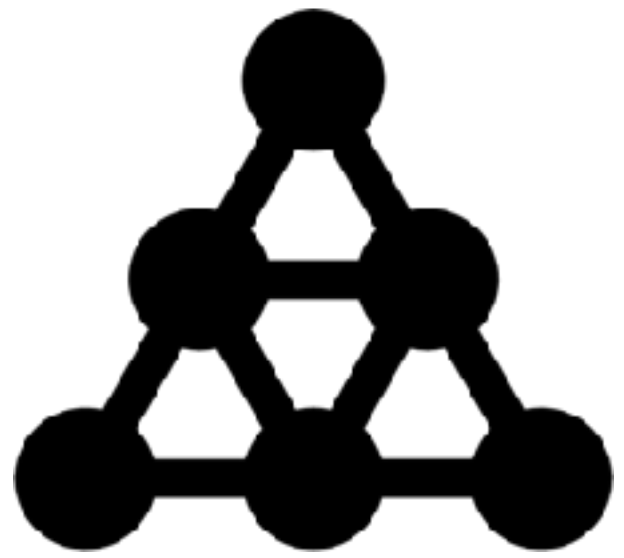
Means to learn

Widely used in systems

Hardware Optimizations

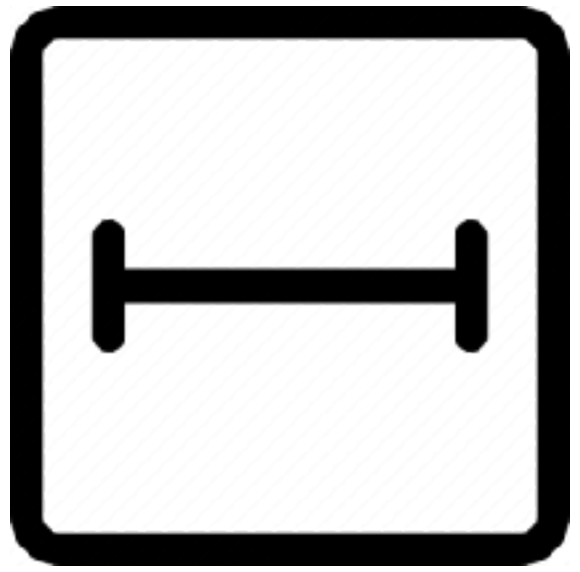


Algorithmic Reasoning/ Techniques



Static Filters

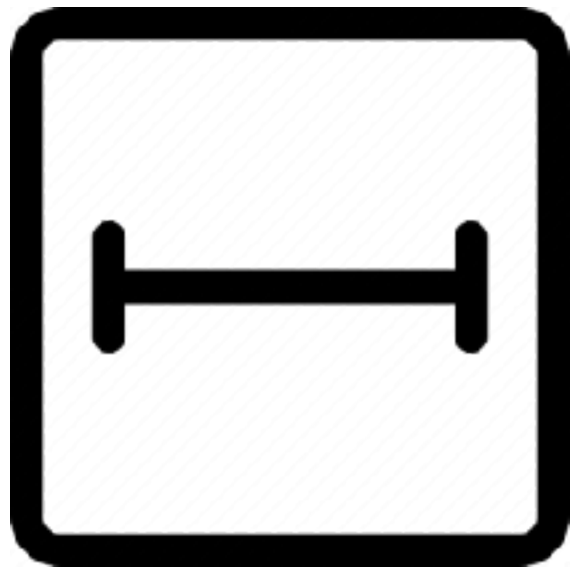
No deletes



No Resizing

Static Filters

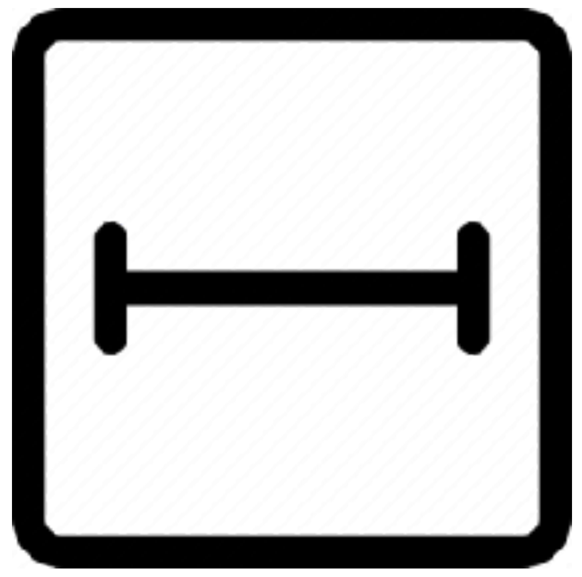
No deletes



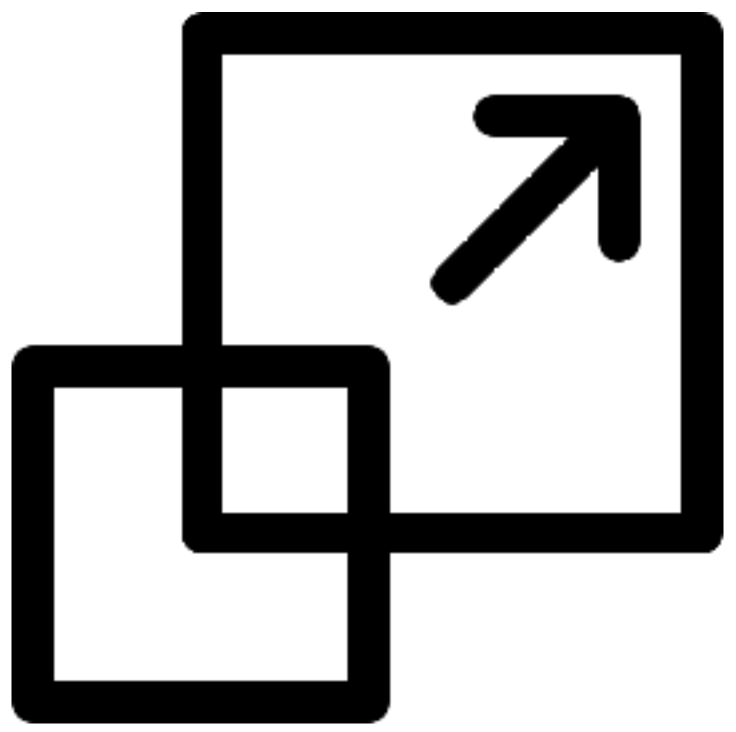
No Resizing

Modifications require rebuilding from scratch

Static Filters

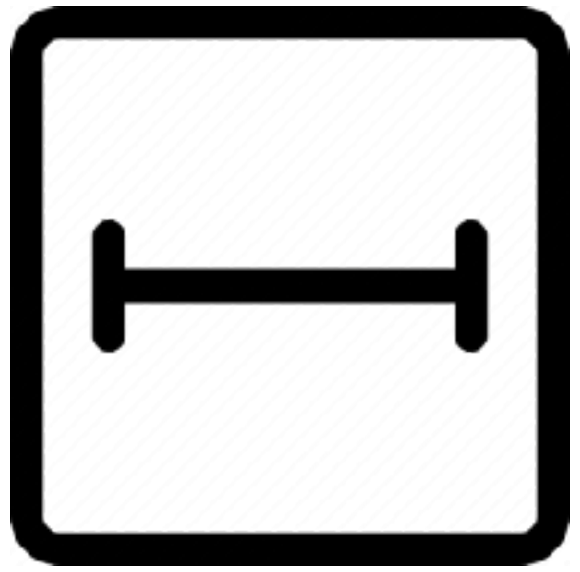


Dynamic Filters (next week)



Delete + Resize

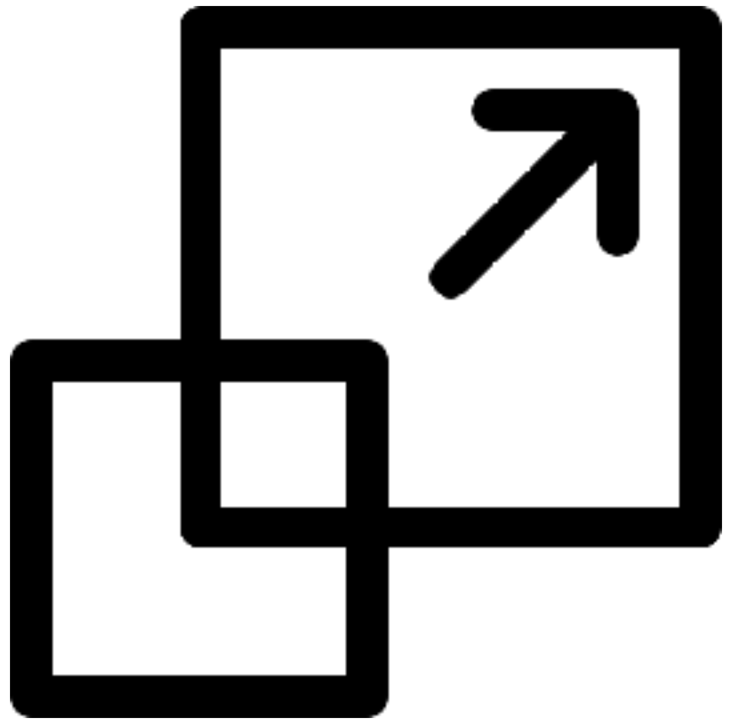
Static Filters



Fastest Queries

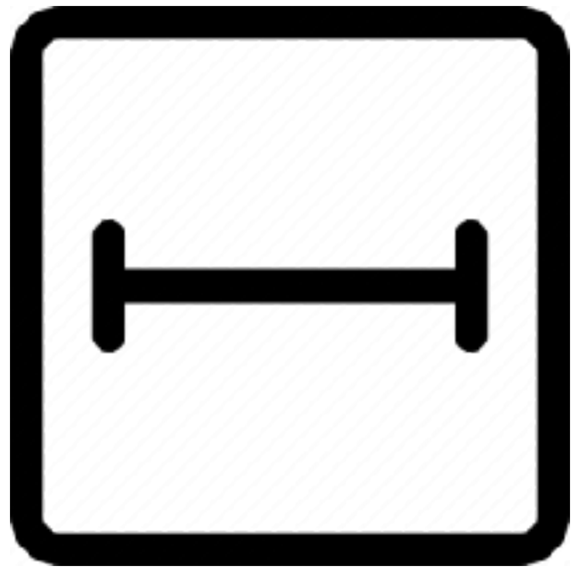
Lowest FPR

Dynamic Filters



Delete + Resize

Static Filters

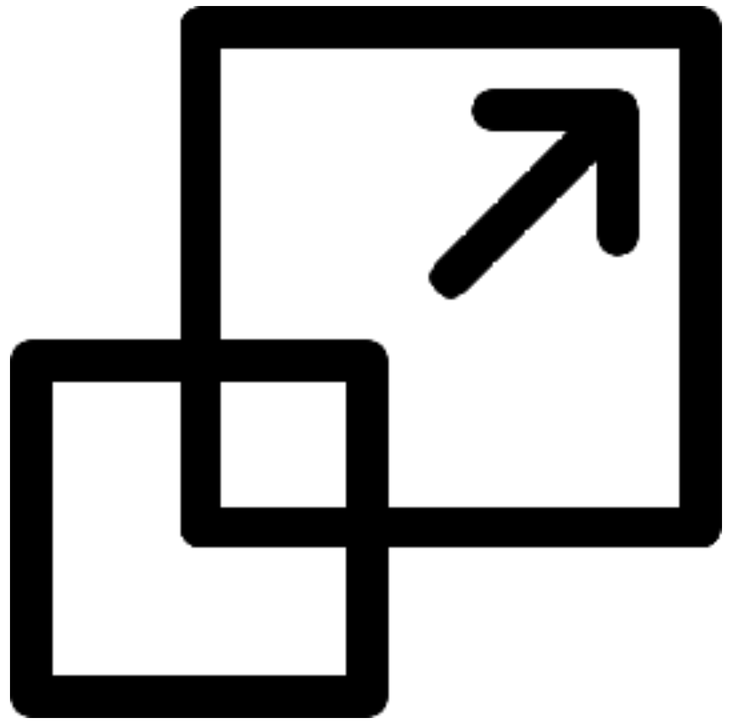


Fastest Queries

Lowest FPR

But not both at same time

Dynamic Filters



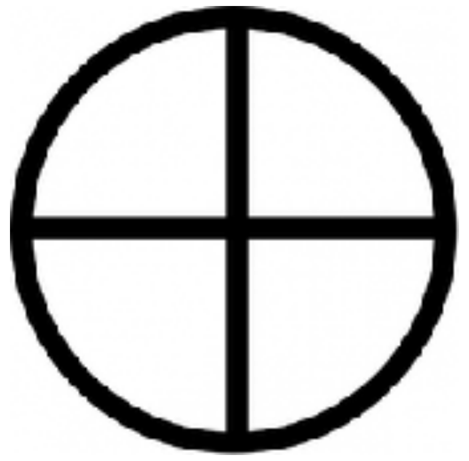
Delete + Resize

Static Filters

Bloom

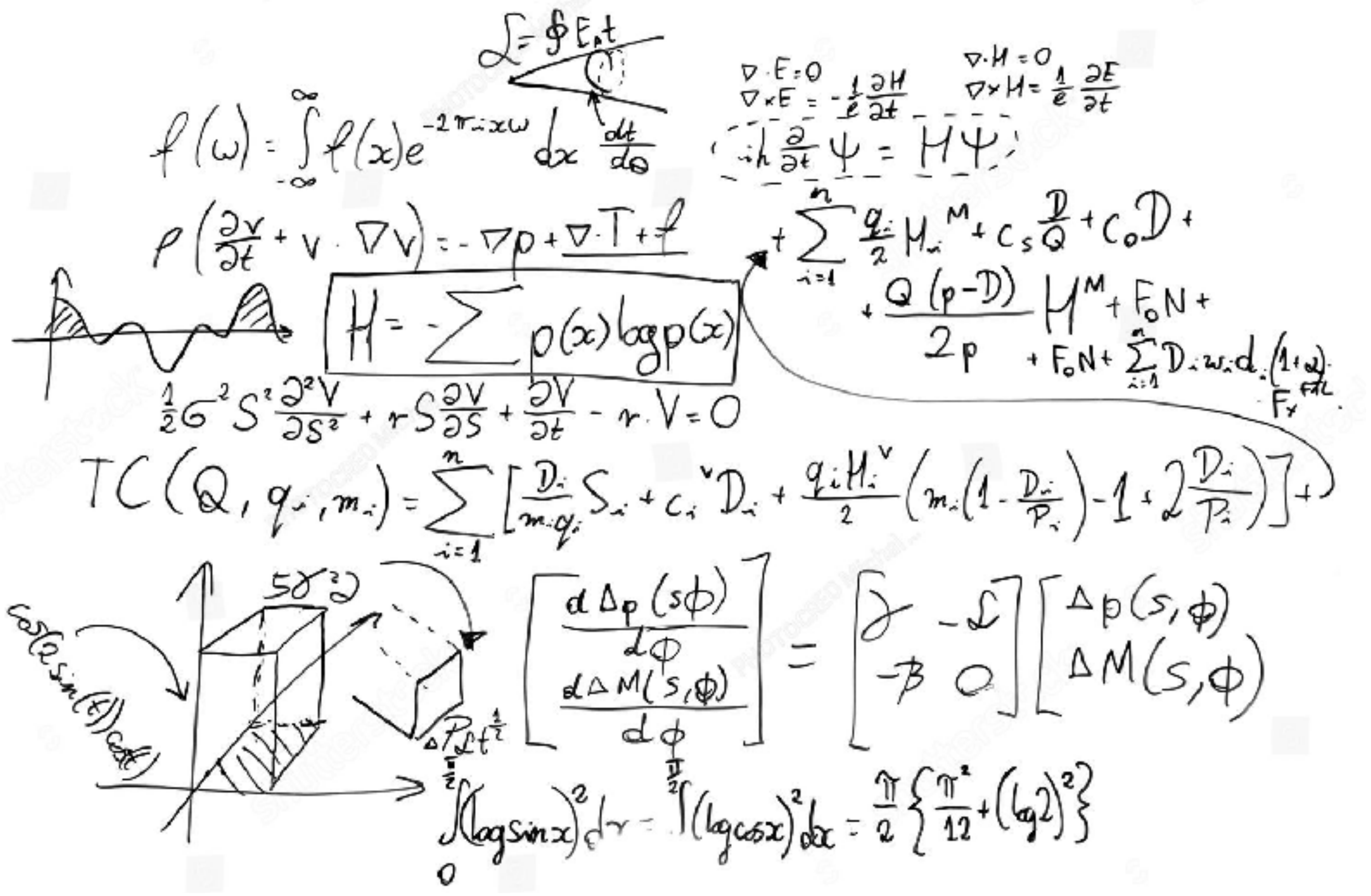
Fastest Queries

XOR



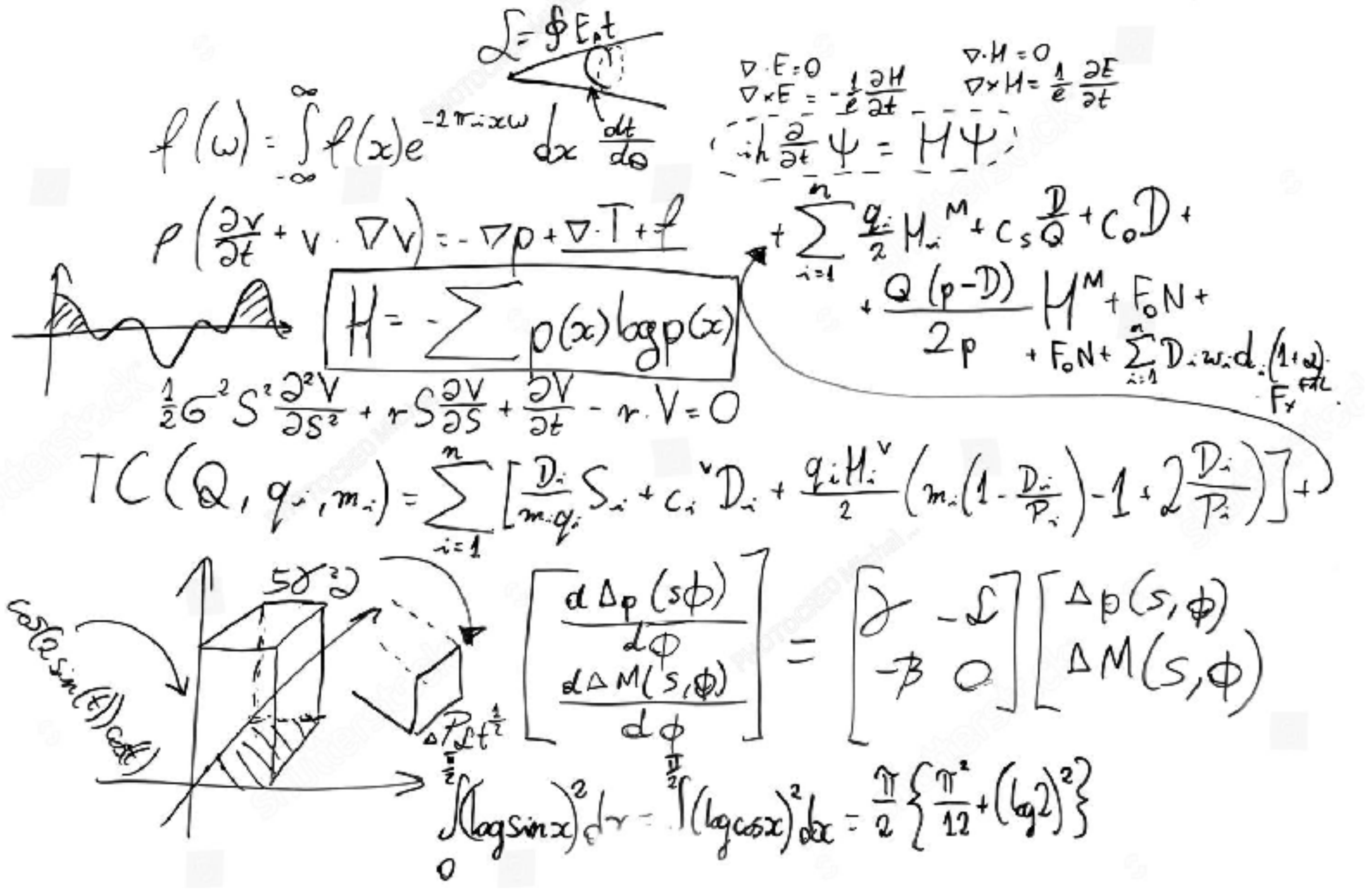
Lowest FPR

Note



Today is more mathematical than usual

Note



Today is more mathematical than usual

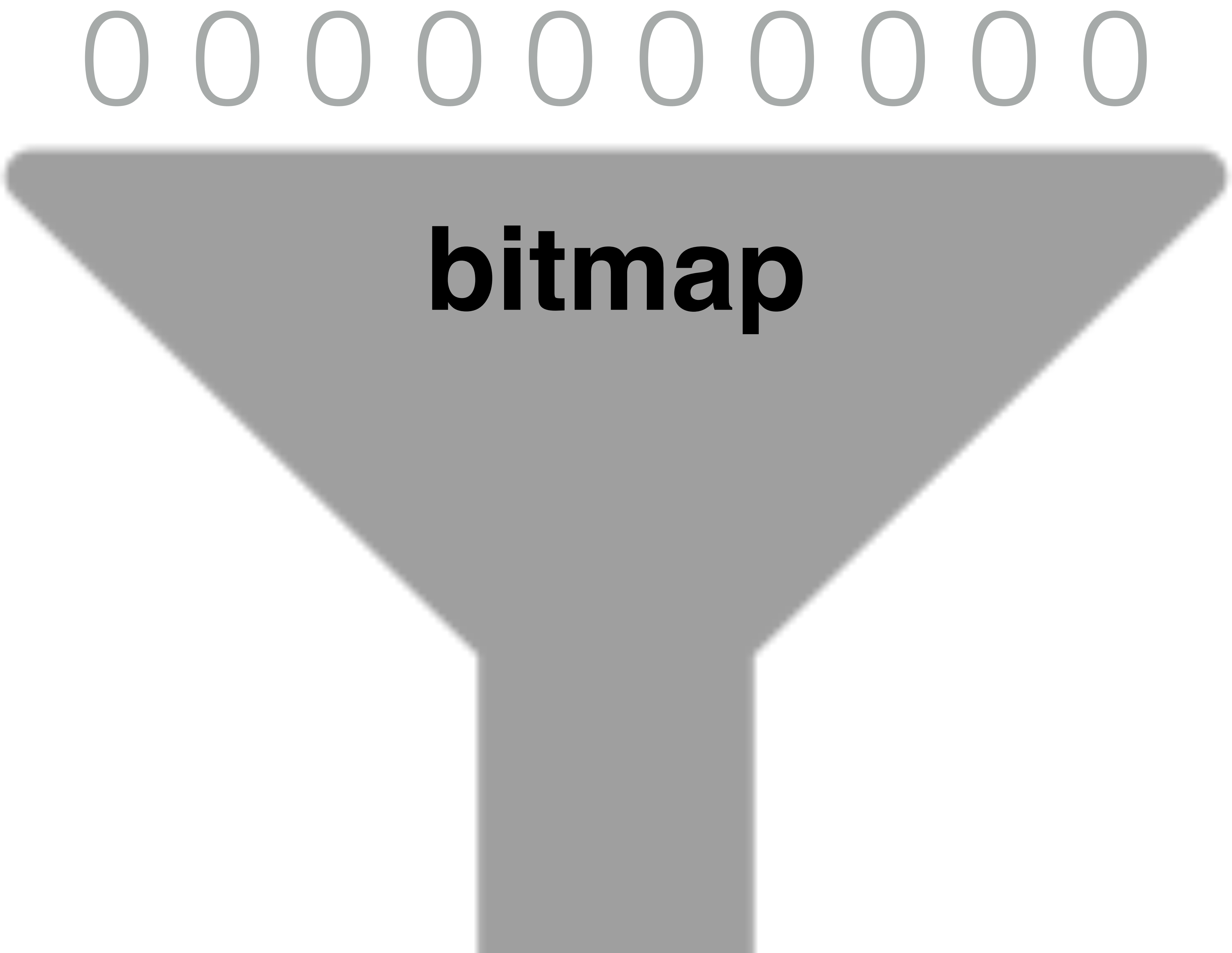
(Do not be intimidated)

Bloom Filters

Bloom Filters

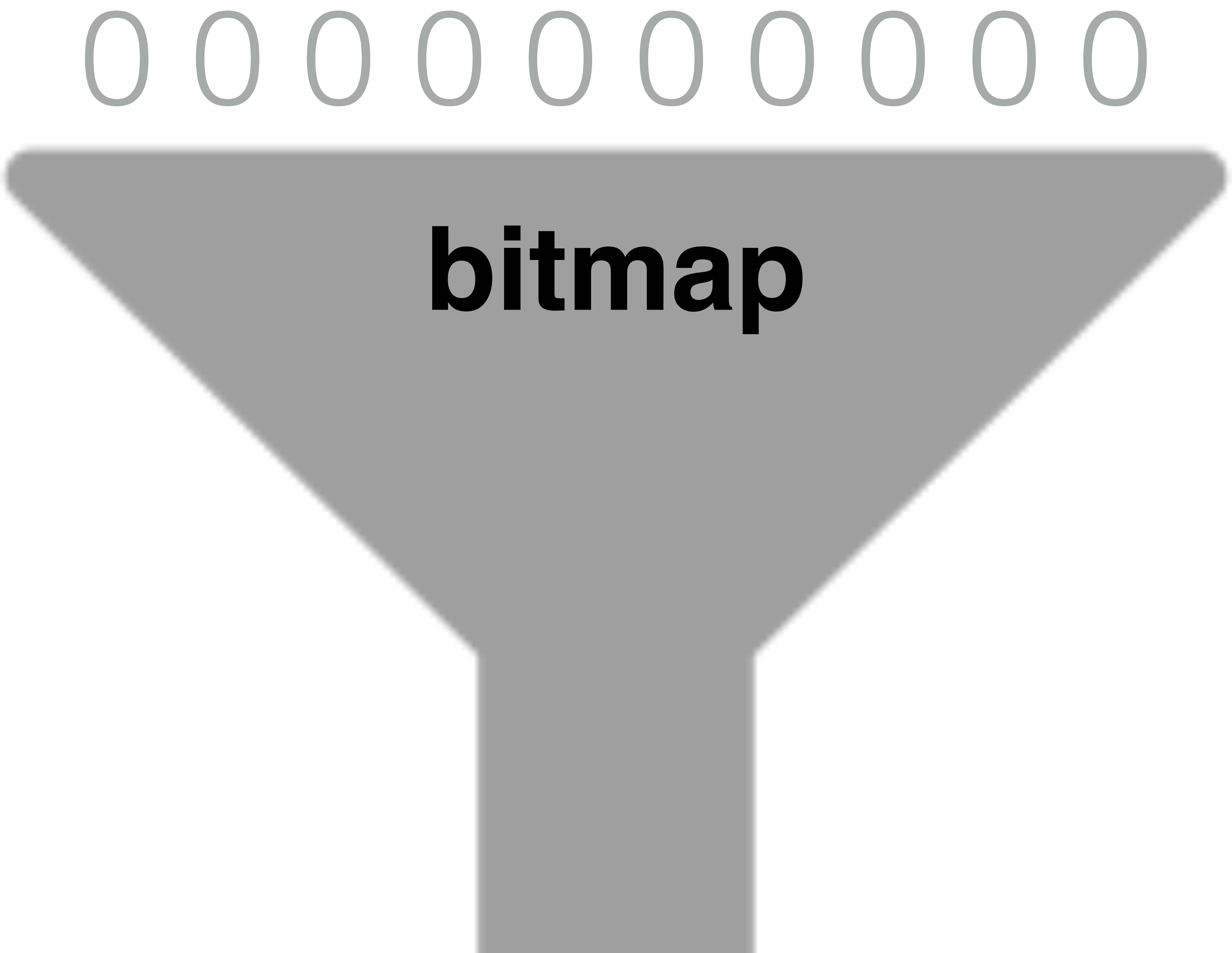
Space/time Trade-Offs in Hash Coding with Allowable Errors
Burton Howard Bloom. Communications of the ACM, 1970.

k hash functions

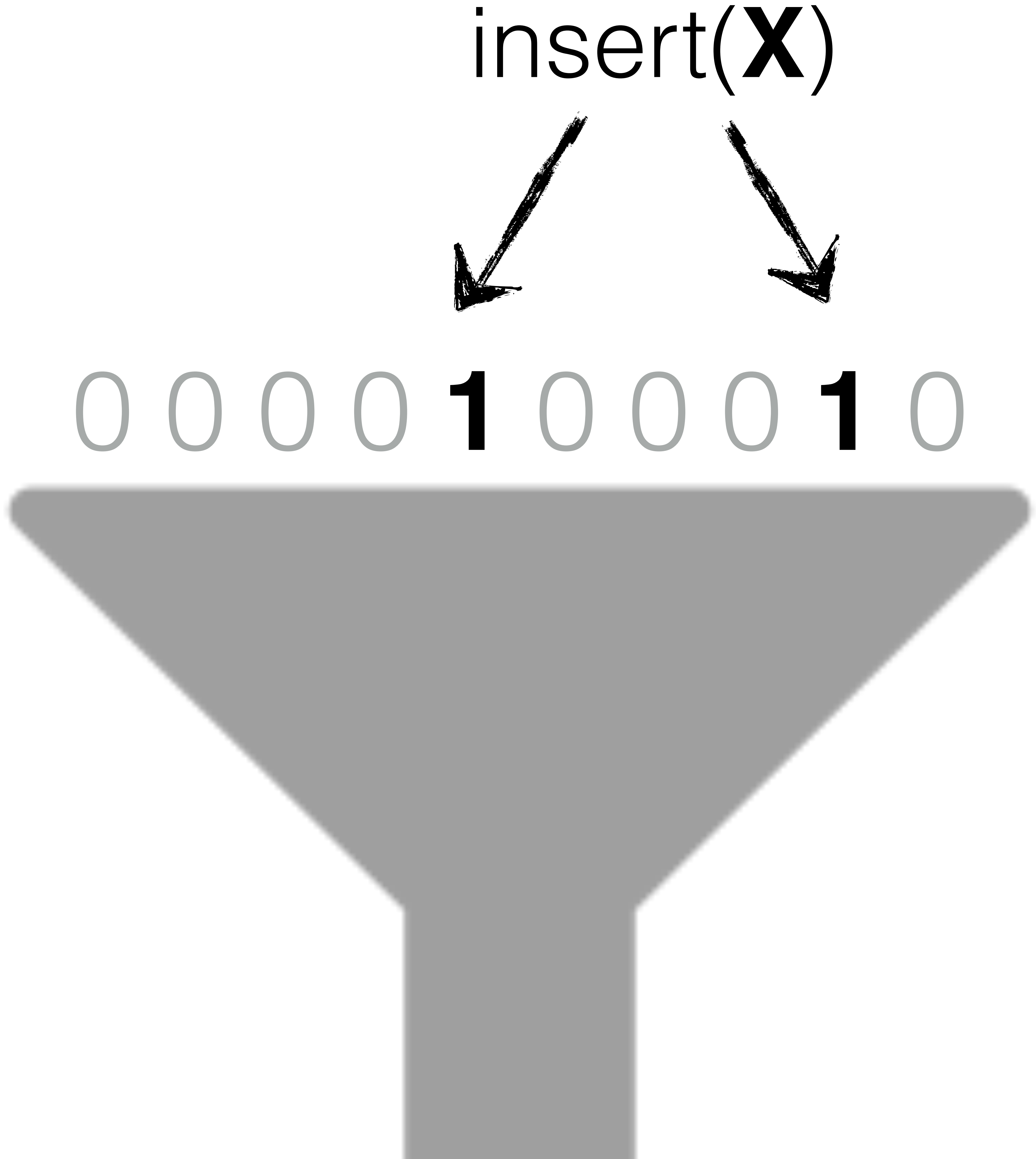


***k* hash functions**

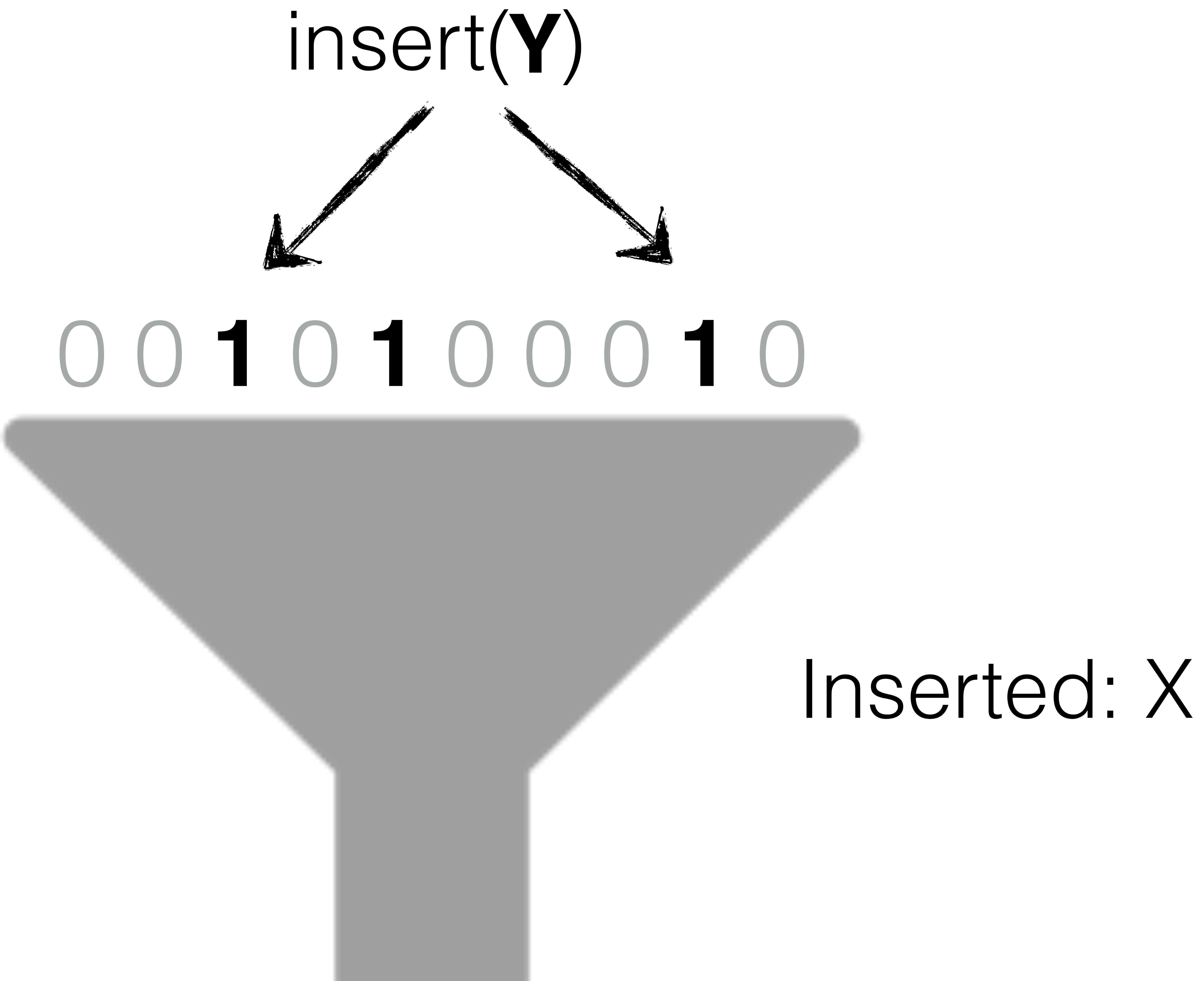
(k=2 in this example)



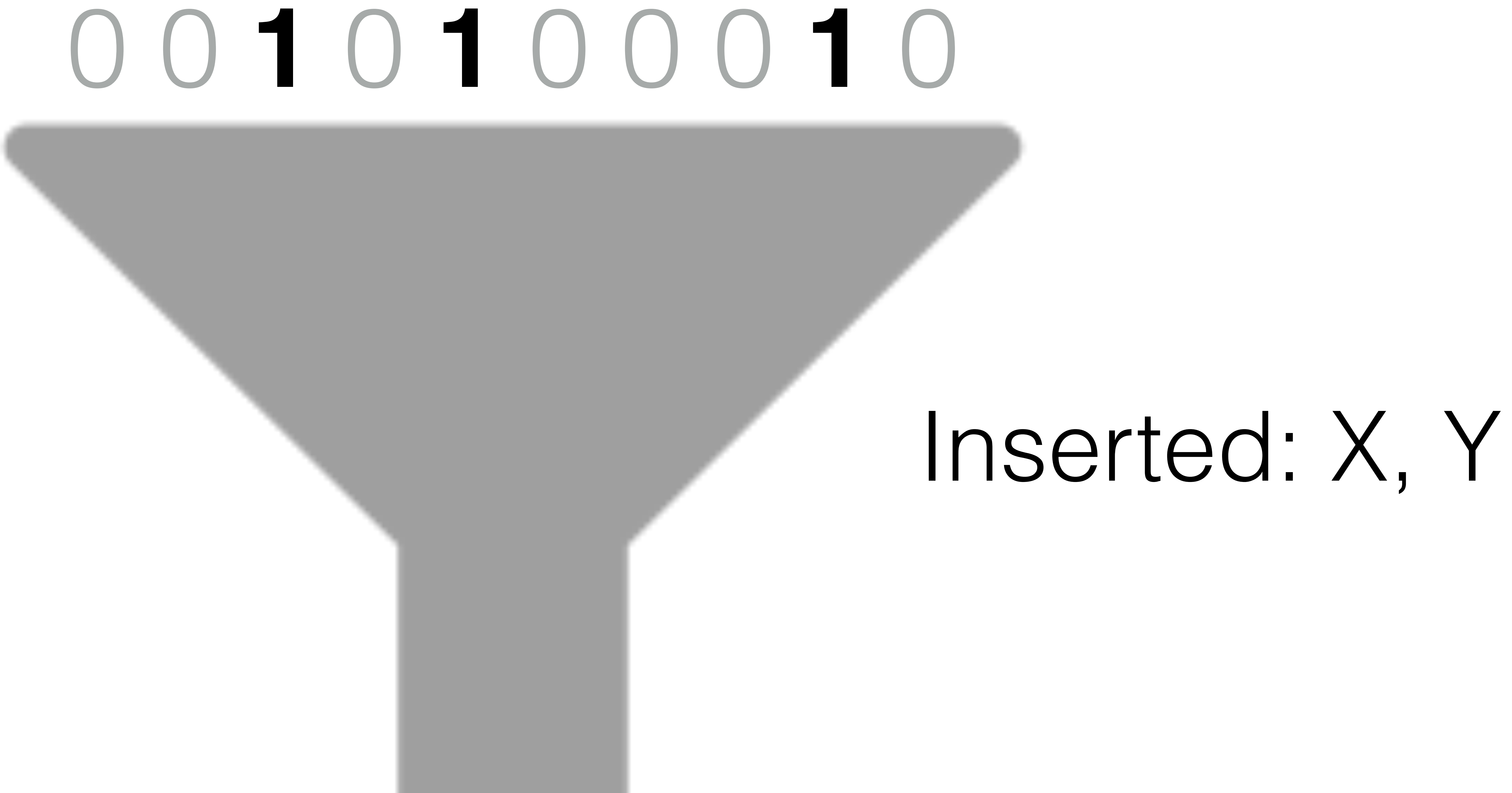
insert: Set from 0 to 1 or keep 1



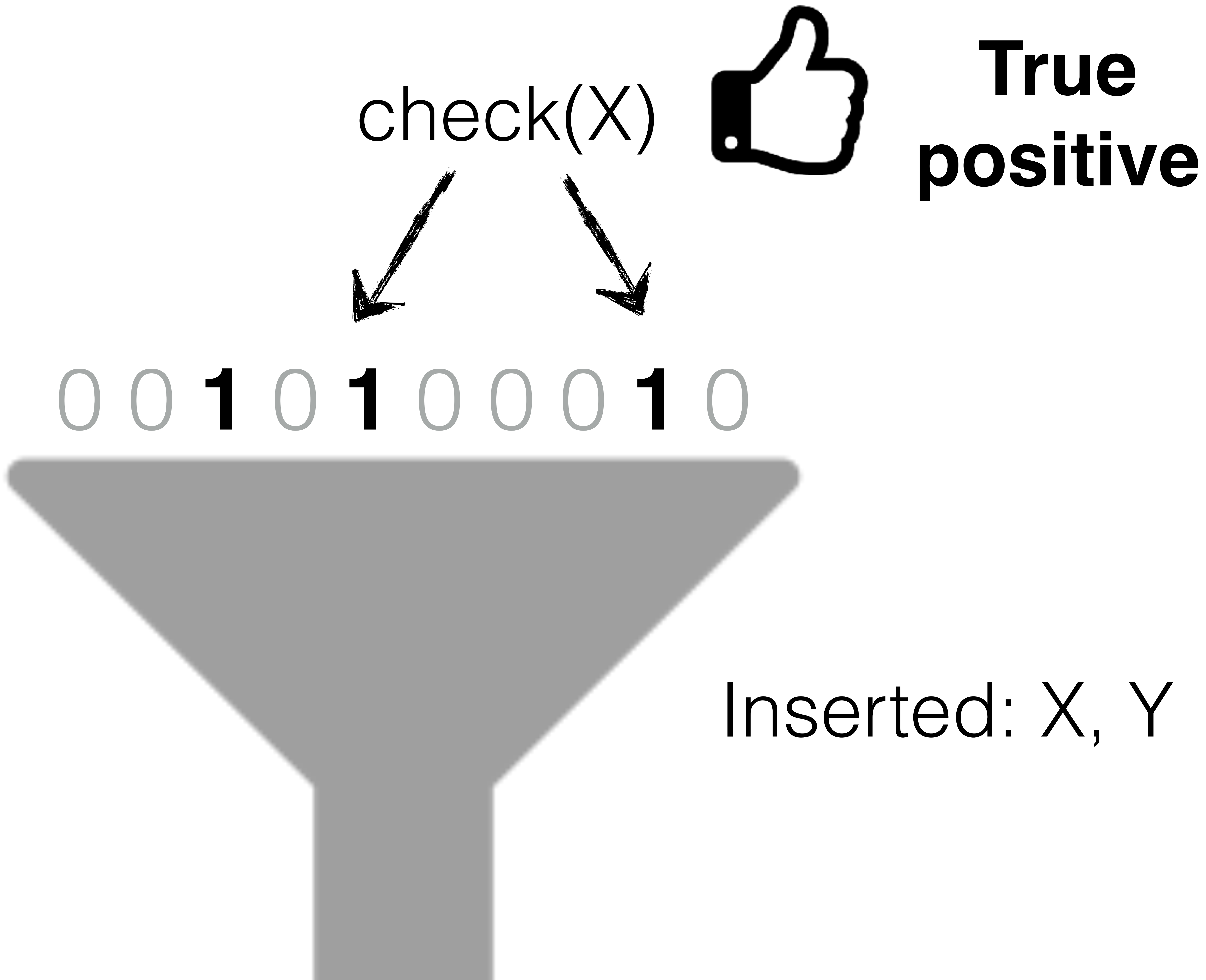
insert: Set from 0 to 1 or keep 1



Queries: return positive if all hashed bits are 1s



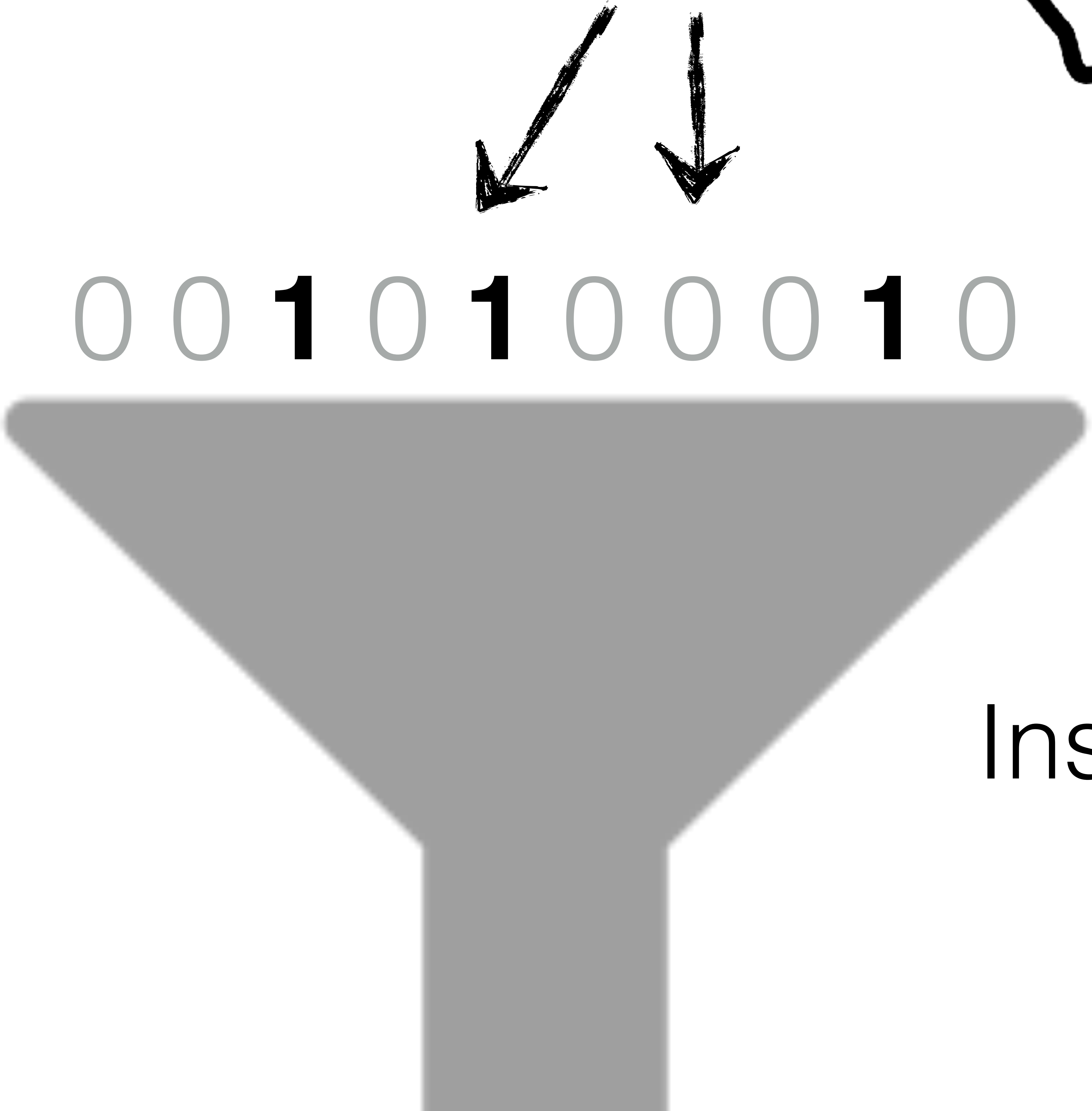
Queries: return positive if all hashed bits are 1s



Queries: return positive if all hashed bits are 1s

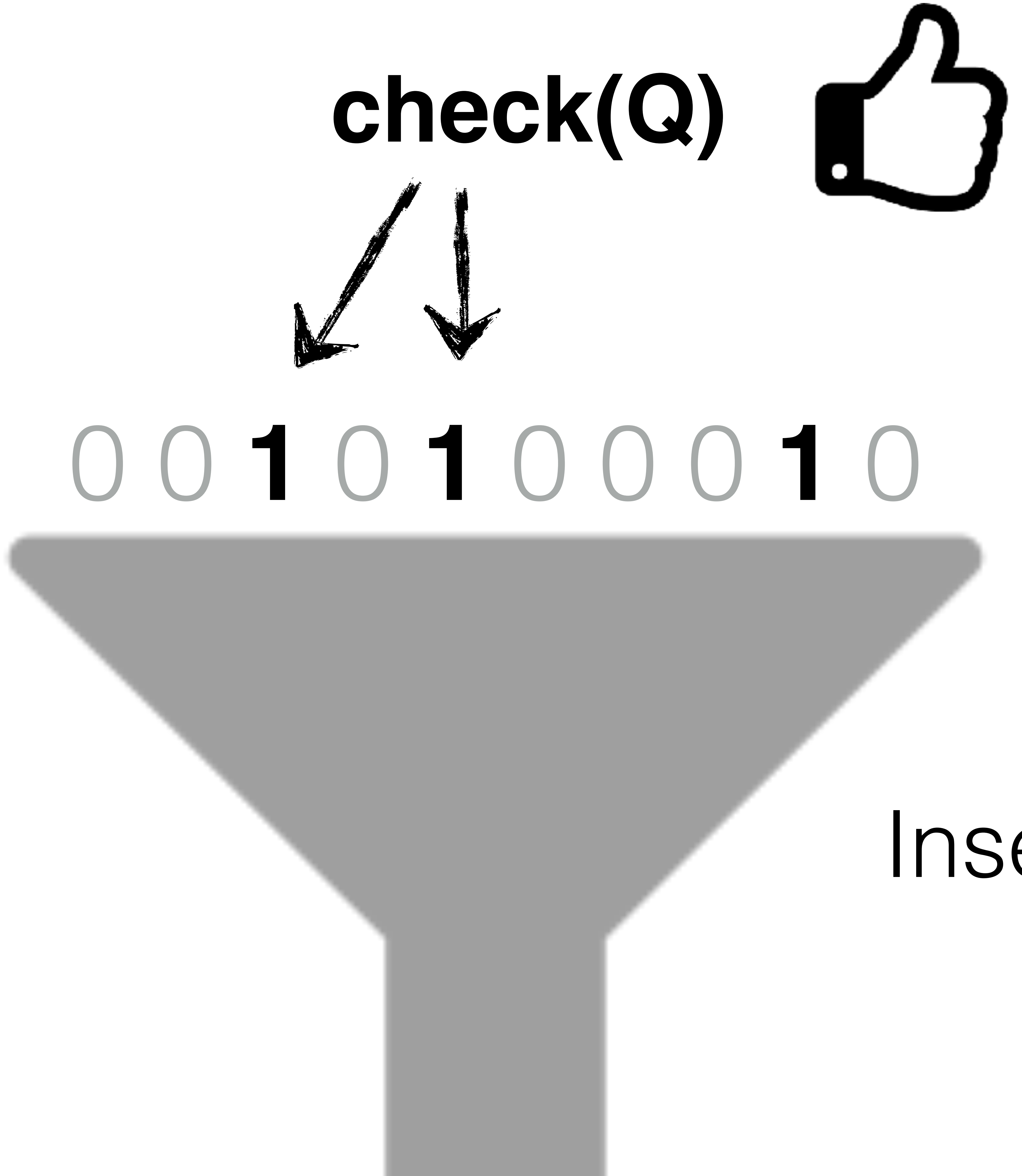
check(Z)

True negative



Inserted: X, Y

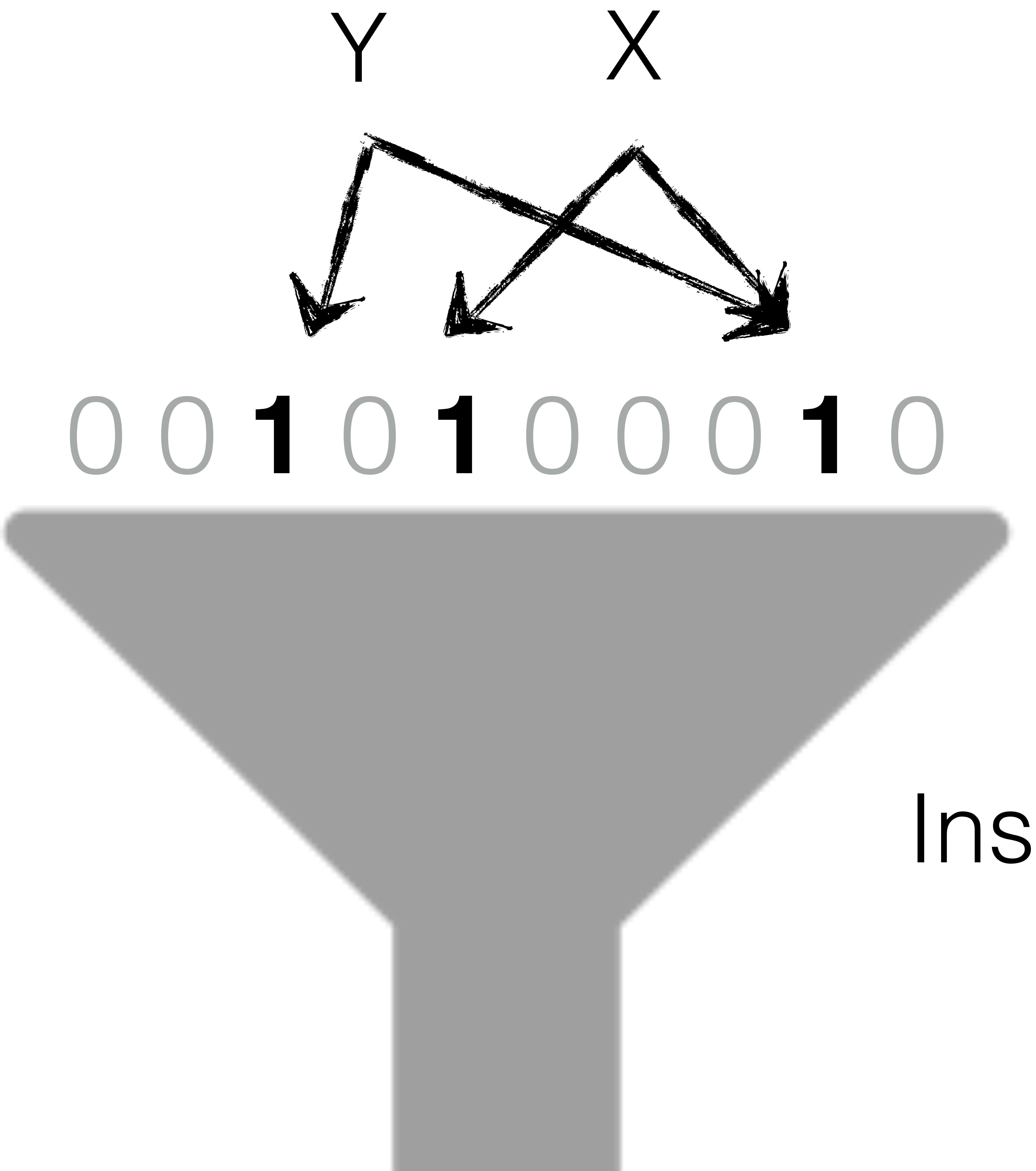
Queries: return positive if all hashed bits are 1s



False Positive

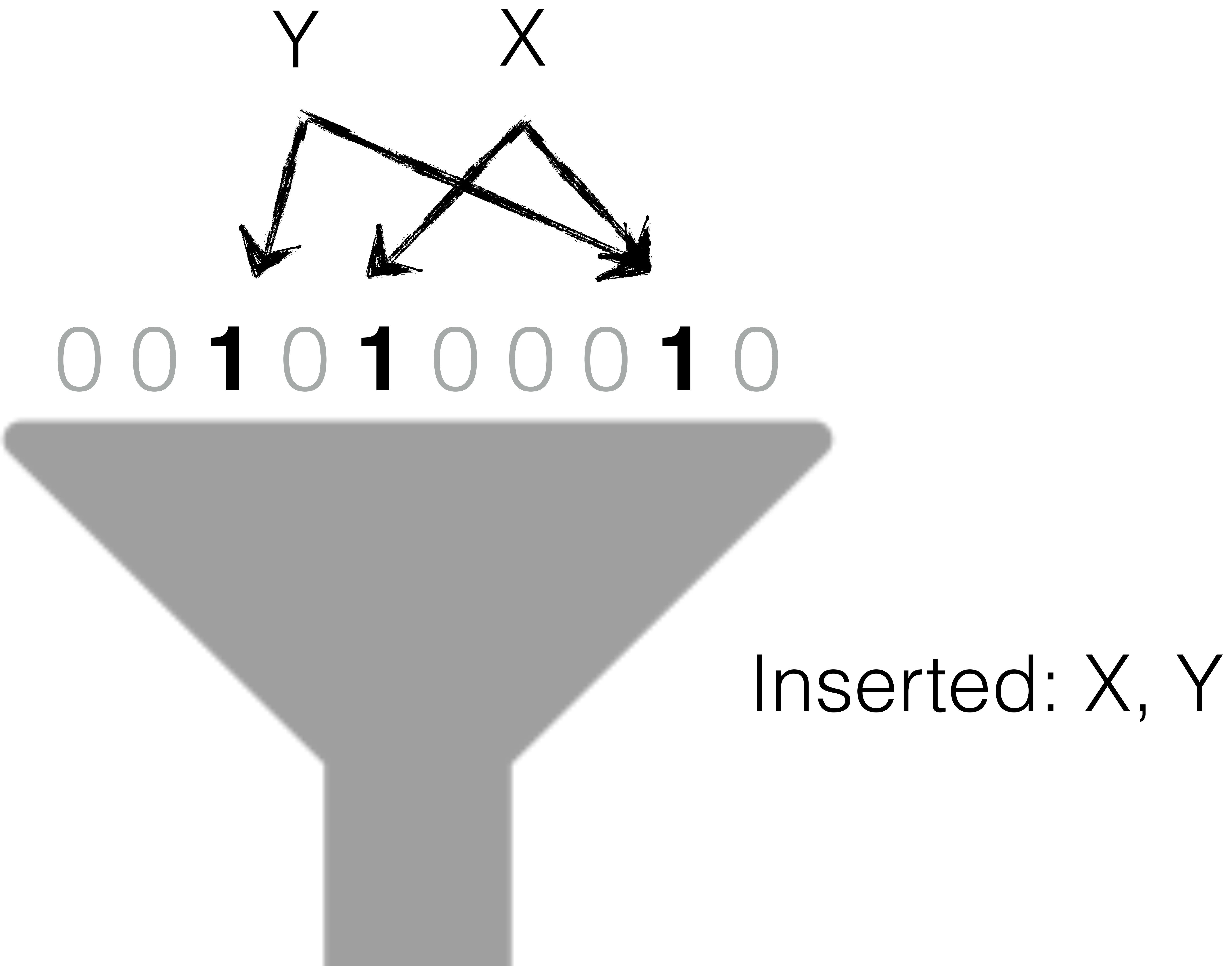
Inserted: X, Y

No deletes - can lead to false negatives



Inserted: X, Y

Thus, we consider it static



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

Construction contract

Know specs in advance:

N - # entries to insert

ε - desired FPR

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

Insert N elements

Construction contract

Know specs in advance:

N - # entries to insert

ε - desired FPR

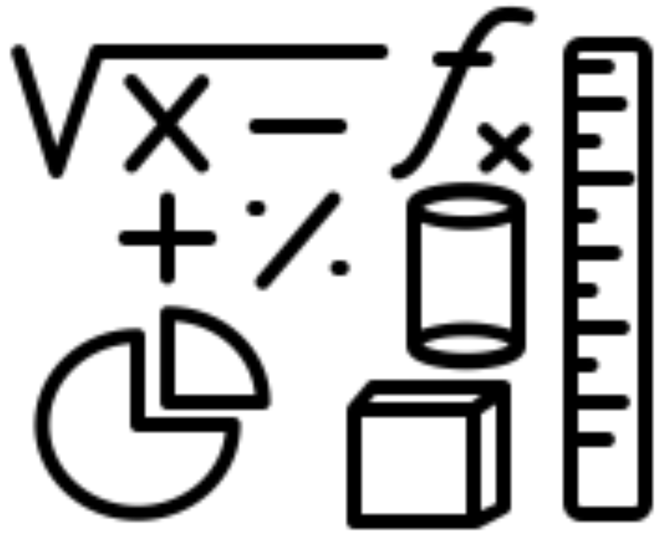
Allocate filter with: $N \cdot (\log_2(1/\varepsilon) / \ln(2))$ bits

Insert N elements

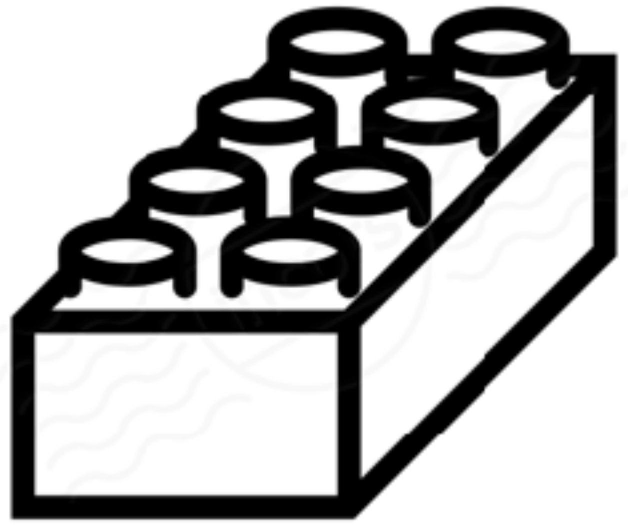
Guarantee FPR of ε

Bloom Filters

Analysis



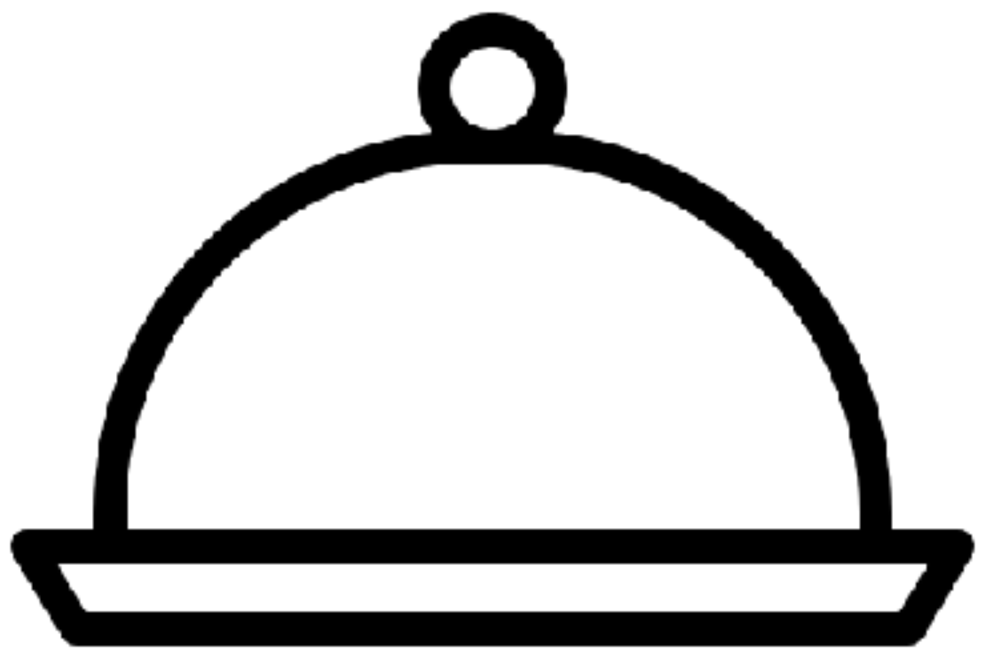
Blocking



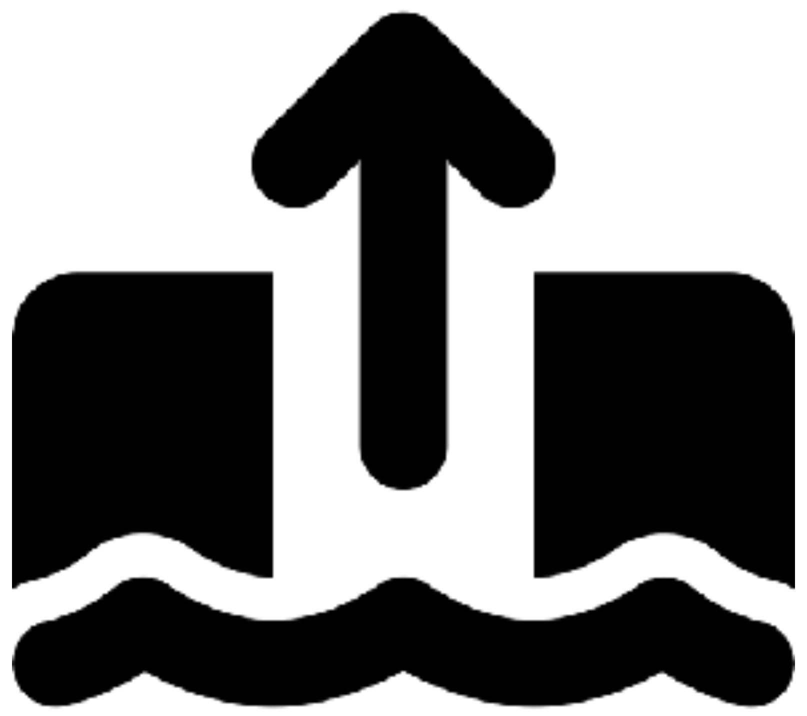
Sectorization

Analysis

In CSC443



Now: ground up



FPR Analysis

Network Applications of Bloom Filters: A Survey
Andrei Broder and Michael Mitzenmacher.
Allerton Conference, 2002

FPR Analysis

M:	Total number of bits
N:	Total number of keys
K:	# hash functions

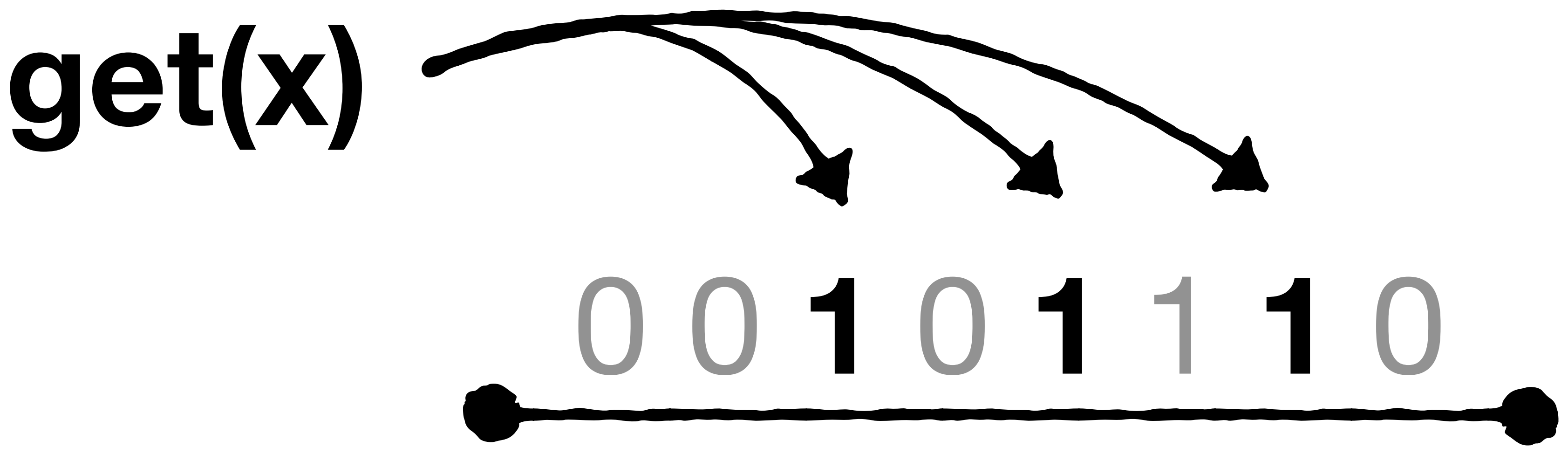
FPR Analysis

M: Total number of bits

N: Total number of keys

K: # hash functions

Probability that all k bits for a non-existing key are set?



FPR Analysis

M: Total number of bits

N: Total number of keys

K: # hash functions

Probability that all k bits for a non-existing key are set?

Probability that some random bit is still not set after N insertions?

0 0 1 **0** 1 1 1 0

FPR Analysis

M: Total number of bits

N: Total number of keys

K: # hash functions

Probability that all k bits for a non-existing key are set?

↓

Probability that some random bit is still not set after N insertions?

↓

Probability that some random bit is still not set after 1 insertion?

FPR Analysis

M: Total number of bits

N: Total number of keys

K: # hash functions

Probability that all k bits for a non-existing key are set?

$\downarrow$

Probability that some random bit is still not set after N insertions?

$\downarrow$

Probability that some random bit is still not set after 1 insertion?

$\downarrow$

Probability that some random bit is set after 1 hash function?

1/M

FPR Analysis

M: Total number of bits

N: Total number of keys

K: # hash functions

Probability that all k bits for a non-existing key are set?

↓

Probability that some random bit is still not set after N insertions?

↓

Probability that some random bit is still not set after 1 insertion?

↓

Probability that some random bit is **not** set after 1 hash function?

1-1/M

FPR Analysis

M: Total number of bits

N: Total number of keys

K: # hash functions

Probability that all k bits for a non-existing key are set?

$\downarrow$

Probability that some random bit is still not set after N insertions?

$\downarrow$

Probability that some random bit is still not set after 1 insertion?

$$(1-1/M)^{K}$$

FPR Analysis

M: Total number of bits

N: Total number of keys

K: # hash functions

Probability that all k bits for a non-existing key are set?

↓

Probability that some random bit is still not set after N insertions?

$$(1-1/M)^{KN}$$

FPR Analysis

M: Total number of bits

N: Total number of keys

K: # hash functions

Probability that all k bits for a non-existing key are set?

$\downarrow$

Probability that some random bit is still not set after N insertions?

$$(1-1/M)^{KN}$$

Known identity: $\mathbf{(1-1/M)^{M} = e^{-1}}$ For any M

FPR Analysis

M: Total number of bits

N: Total number of keys

K: # hash functions

Probability that all k bits for a non-existing key are set?

$\downarrow$

Probability that some random bit is still not set after N insertions?

$$((1-1/M)^M)^{KN/M}$$

Known identity: $(1-1/M)^M = e^{-1}$ For any M

FPR Analysis

M: Total number of bits

N: Total number of keys

K: # hash functions

Probability that all k bits for a non-existing key are set?

$\downarrow$

Probability that some random bit is still not set after N insertions?

$$(e^{-1})^{KN/M}$$

FPR Analysis

M: Total number of bits

N: Total number of keys

K: # hash functions

Probability that all k bits for a non-existing key are set?

Probability that some random bit is still not set after N insertions?

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

FPR Analysis

M: Total number of bits

N: Total number of keys

K: # hash functions

Probability that all k bits for a non-existing key are set?

↓

Probability that some random bit is ~~still not~~ set after N insertions?

$$\mathbf{1}-e^{-KN/M}$$

FPR Analysis

M: Total number of bits

N: Total number of keys

K: # hash functions

Probability that all k bits for a non-existing key are set?

$\downarrow$

Probability that some random bit is set after N insertions?

$$1-e^{-KN/M}$$

FPR Analysis

M: Total number of bits

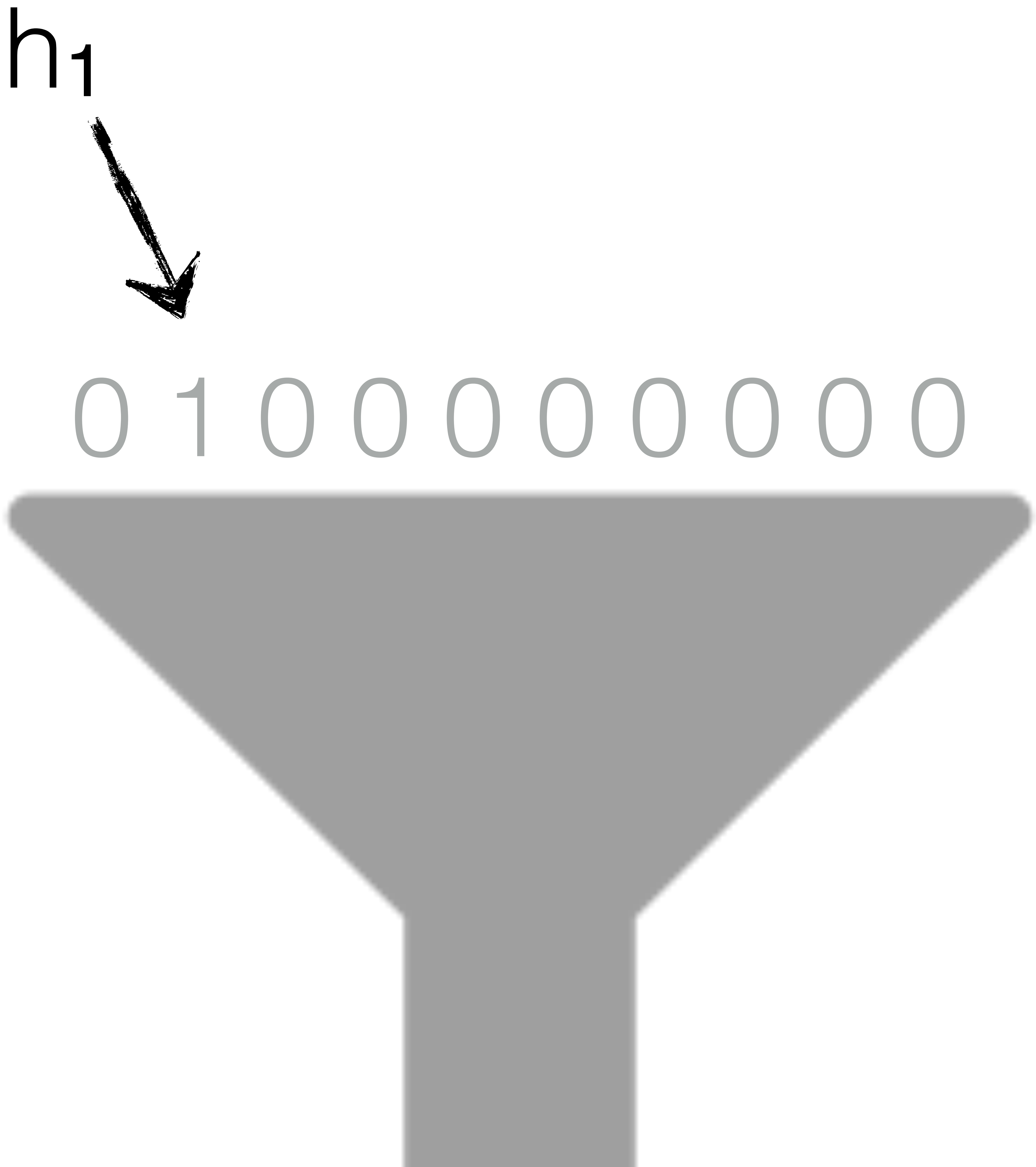
N: Total number of keys

K: # hash functions

Probability that all k bits for a non-existing key are set?

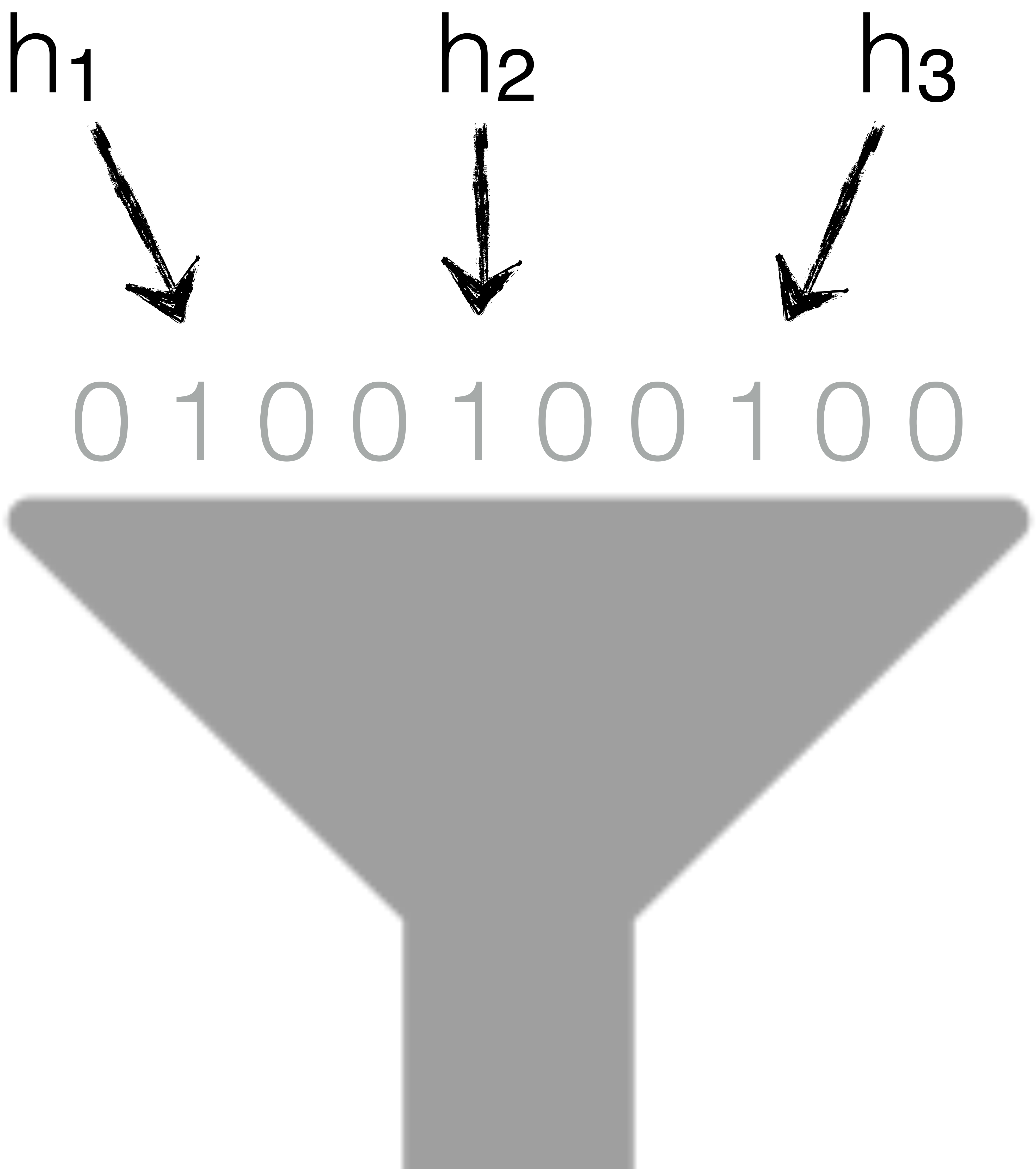
$$(1-e^{-KN/M})^{K}$$

How many hash functions should we use?



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

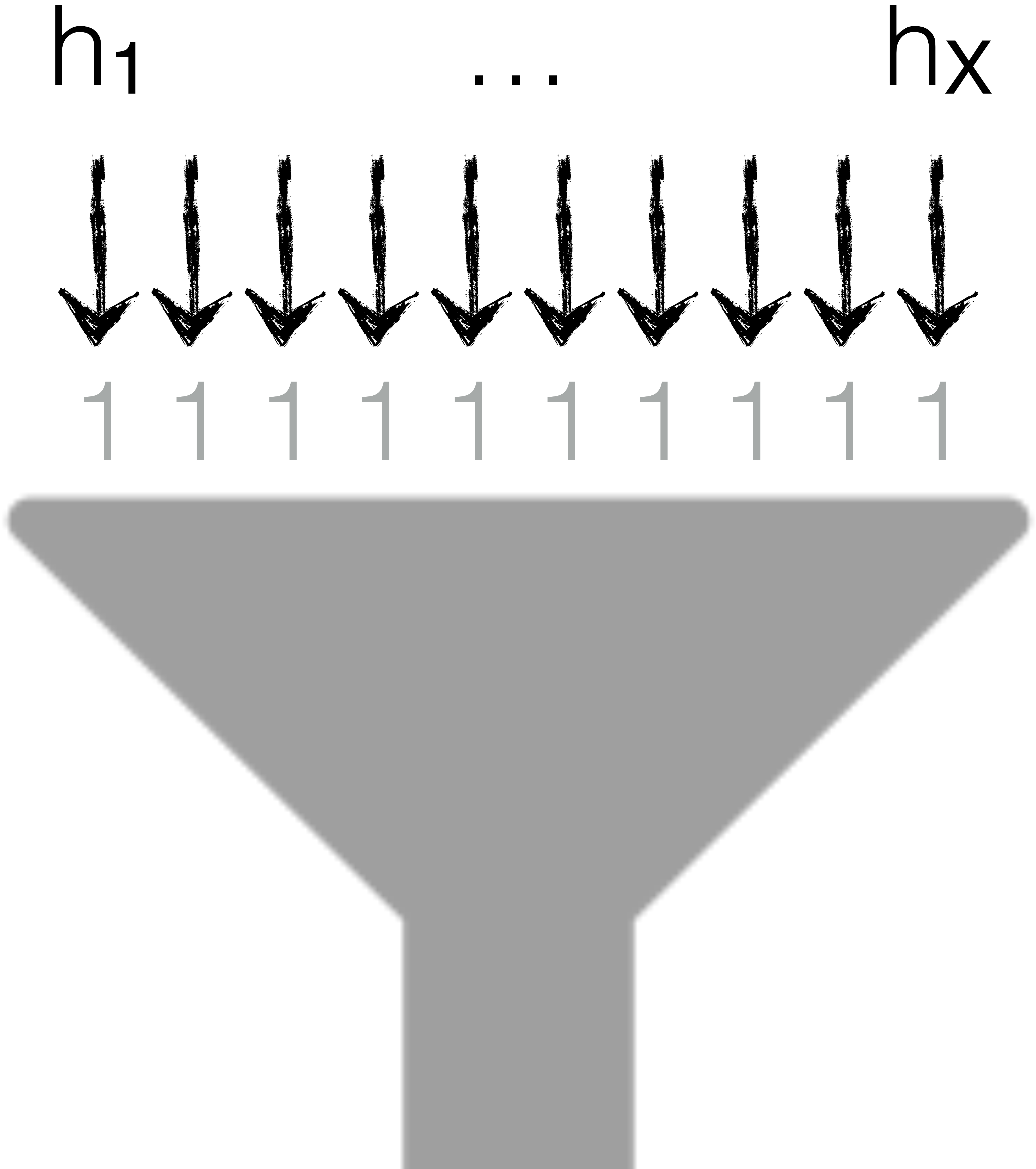
How many hash functions should we use?



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?

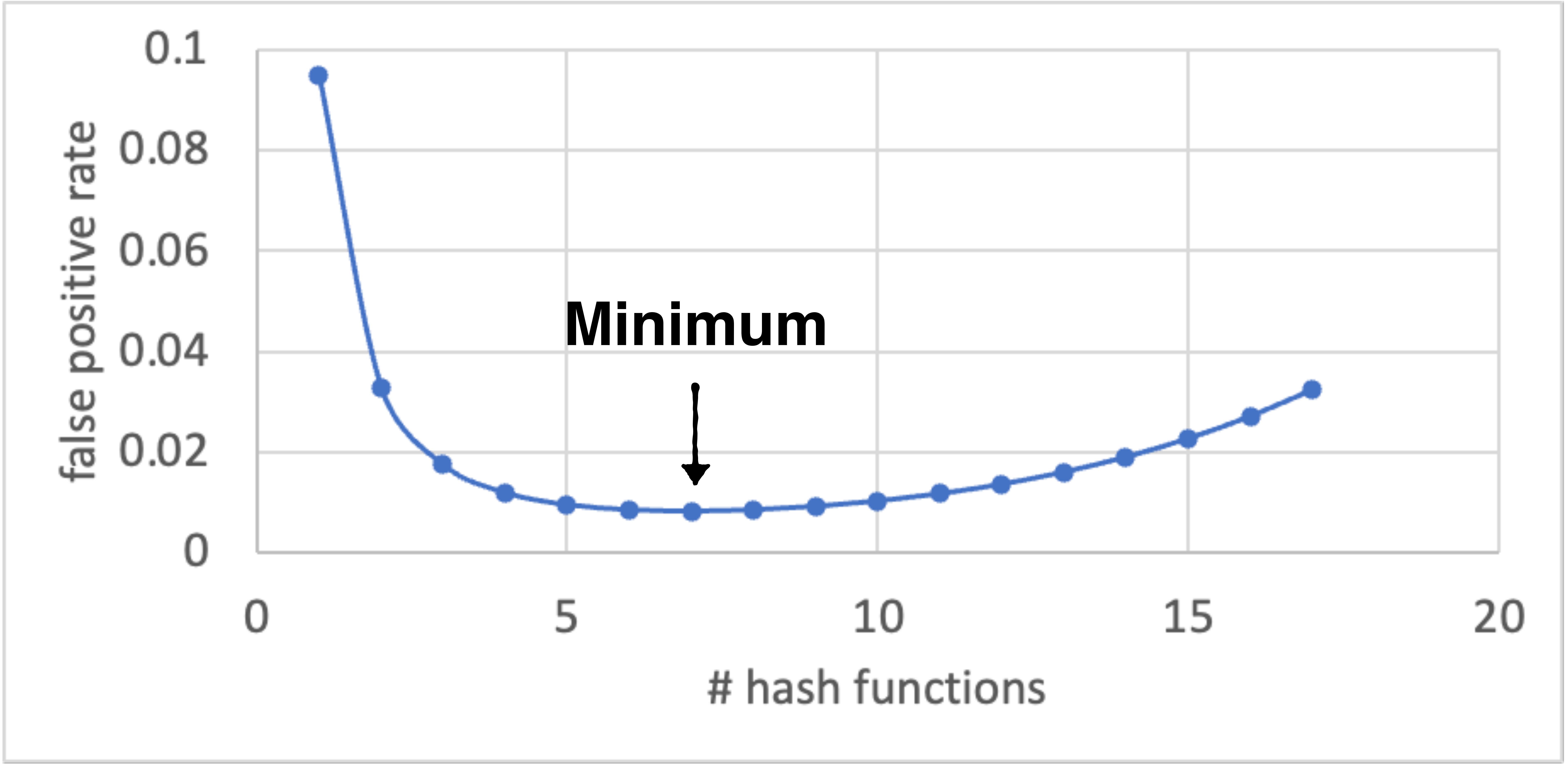


One is too few: false positive occurs whenever we hit a 1

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

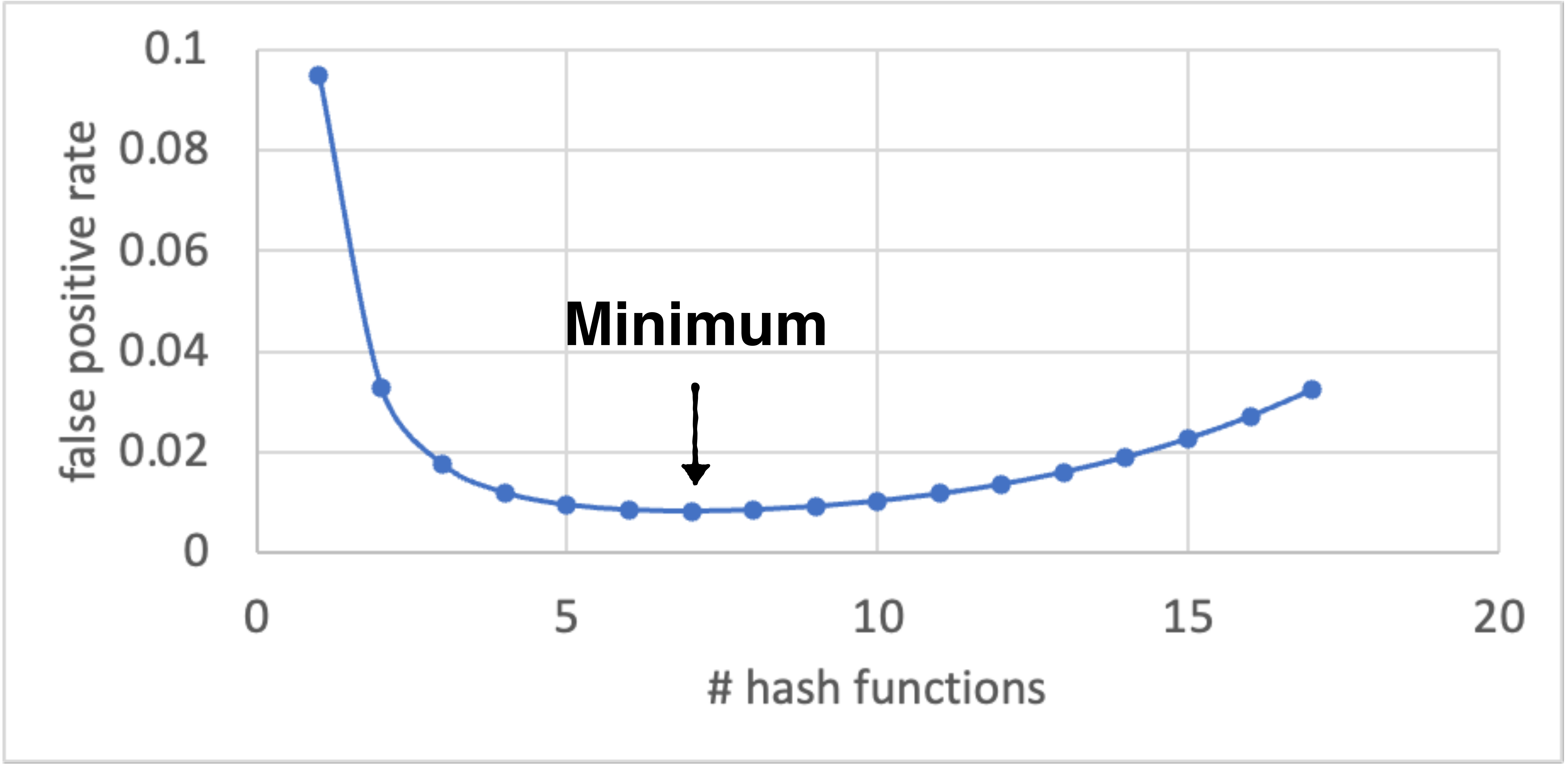
But too many hash functions wind up increasing the FPR.

How many hash functions should we use?



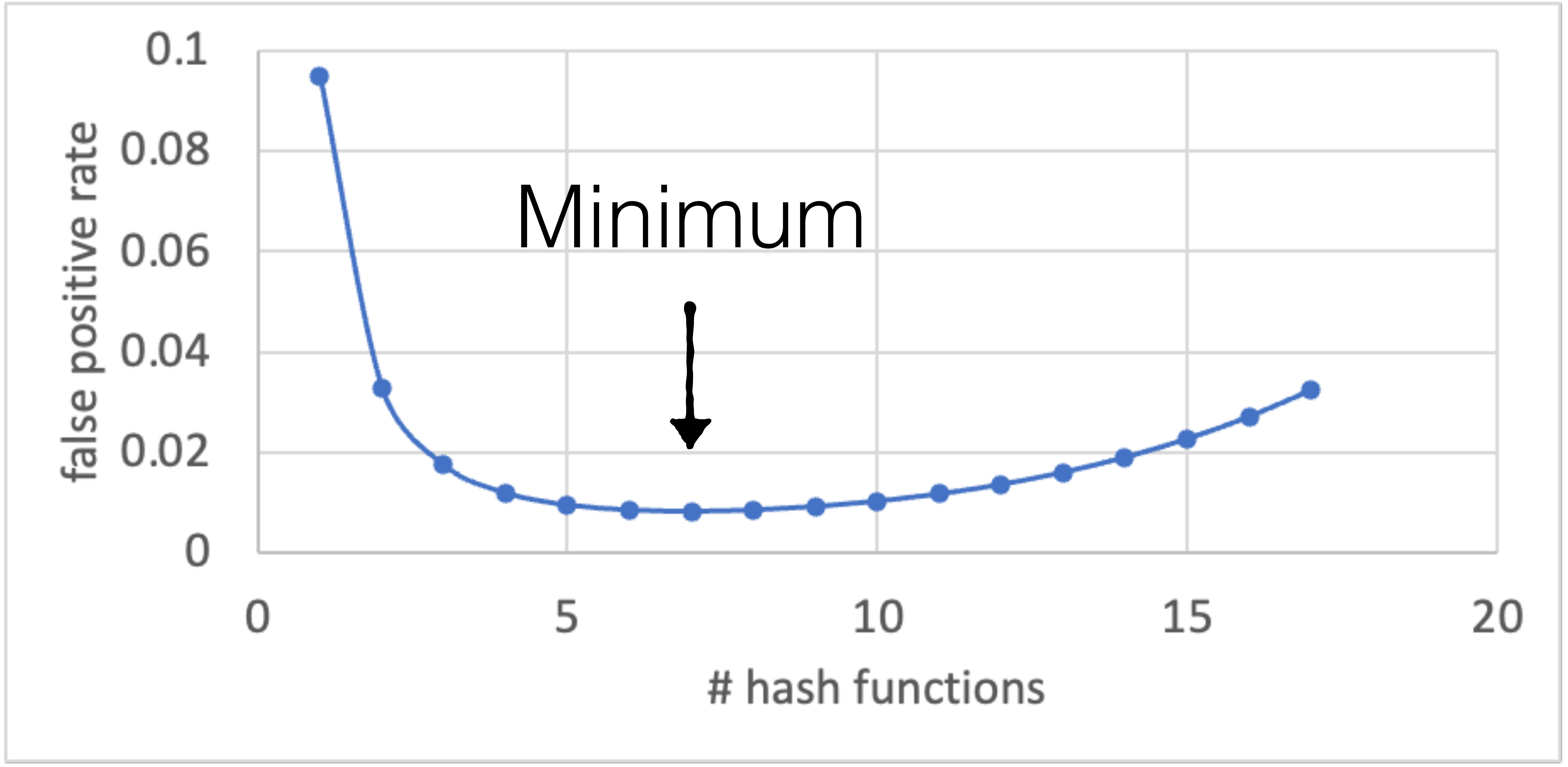
(Drawn for a filter using 10 bits per entry)

How many hash functions should we use?



Differentiate $(1-e^{-KN/M})^K$ with respect to K

How many hash functions should we use?

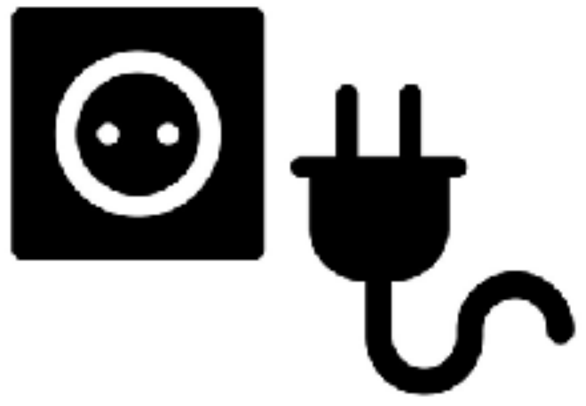


Differentiate $(1-e^{-KN/M})^K$ with respect to K

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

(e.g. with Wolfram Alpha)

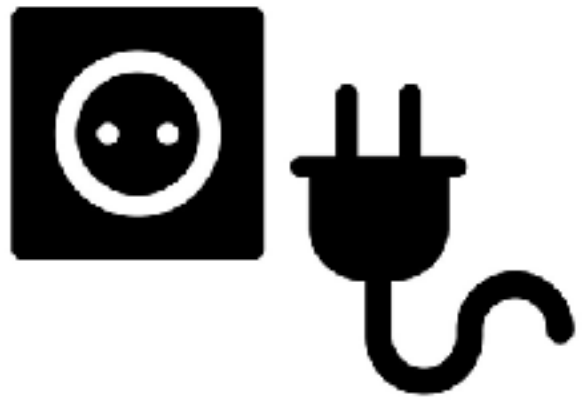
Optimal # hash functions $\mathbf{k = \ln(2) \cdot} \boldsymbol{M/N}$



False positives rate

$$(1-e^{-KN/M})^K$$

Optimal # hash functions $\mathbf{k = \ln(2) \cdot \mathit{M/N}}$



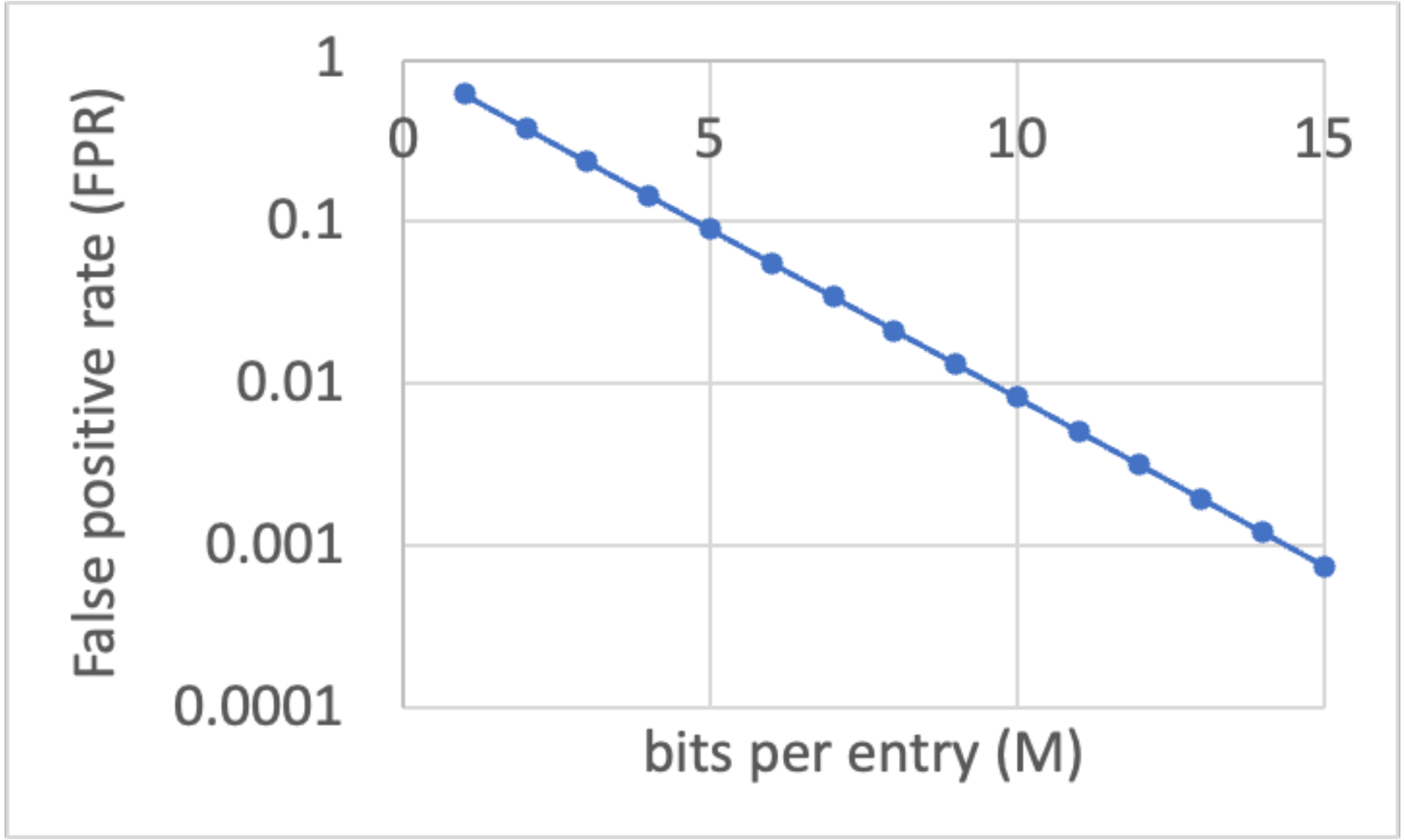
False positives rate

$$\mathbf{(1-e^{-KN/M})^K}$$

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

assuming the optimal # hash functions,

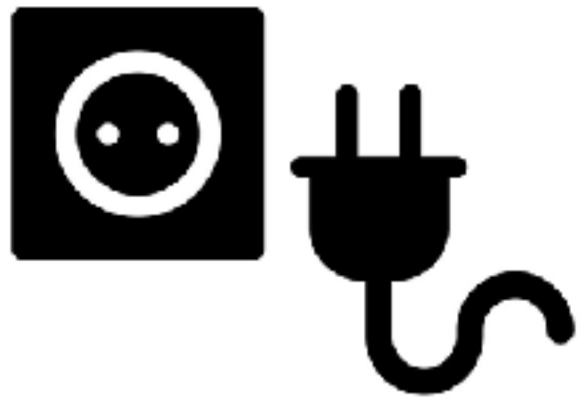
false positive rate = $2^{-M/N \cdot \ln(2)}$



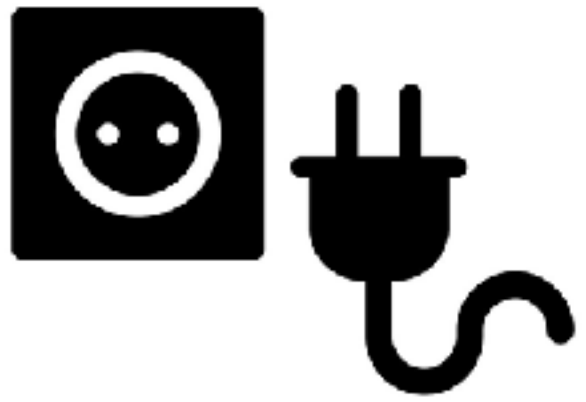
Optimal # hash functions $\mathbf{k = ln(2)} \cdot \boldsymbol{M/N}$

Some bit is not set

$e^{-KN/M}$



Optimal # hash functions $k = \ln(2) \cdot M/N$

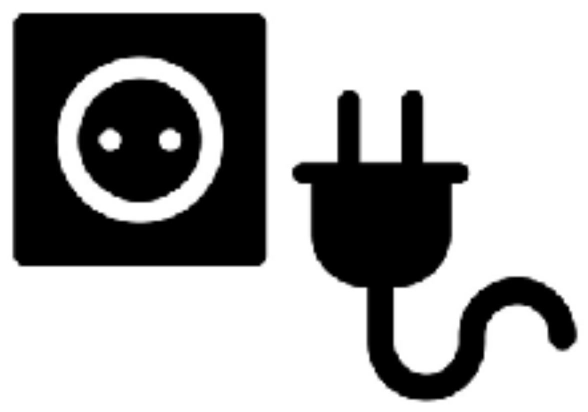


Some bit is
not set

$$\mathbf{e^{-KN/M}}$$

0.5

Optimal # hash functions $k = \ln(2) \cdot M/N$



Some bit is
not set

$e^{-KN/M}$

0.5

50% of all bits are zero once the filter is full

Operation Costs (in hash functions computed)

Insertion =

Positive Query =

Negative Query =

Operation Costs (in hash functions computed)

 Insertion = **$M/N \cdot \ln(2)$**

 Positive Query =

 Negative Query =

Operation Costs (in hash functions computed)

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

Positive Query = $\mathbf{M/N \cdot ln(2)}$

Negative Query =

Operation Costs (in hash functions computed)

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

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

Negative Query =

(50% of bits are zeros)

Operation Costs (in hash functions computed)

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

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

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

(50% of bits are zeros)

Operation Costs (in hash functions computed)

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

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

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

(50% of bits are zeros)

Operation Costs (in hash functions computed)

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

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

Avg. Negative Query = 2

Full analysis from ground up :)

Operation Costs (in hash functions computed)

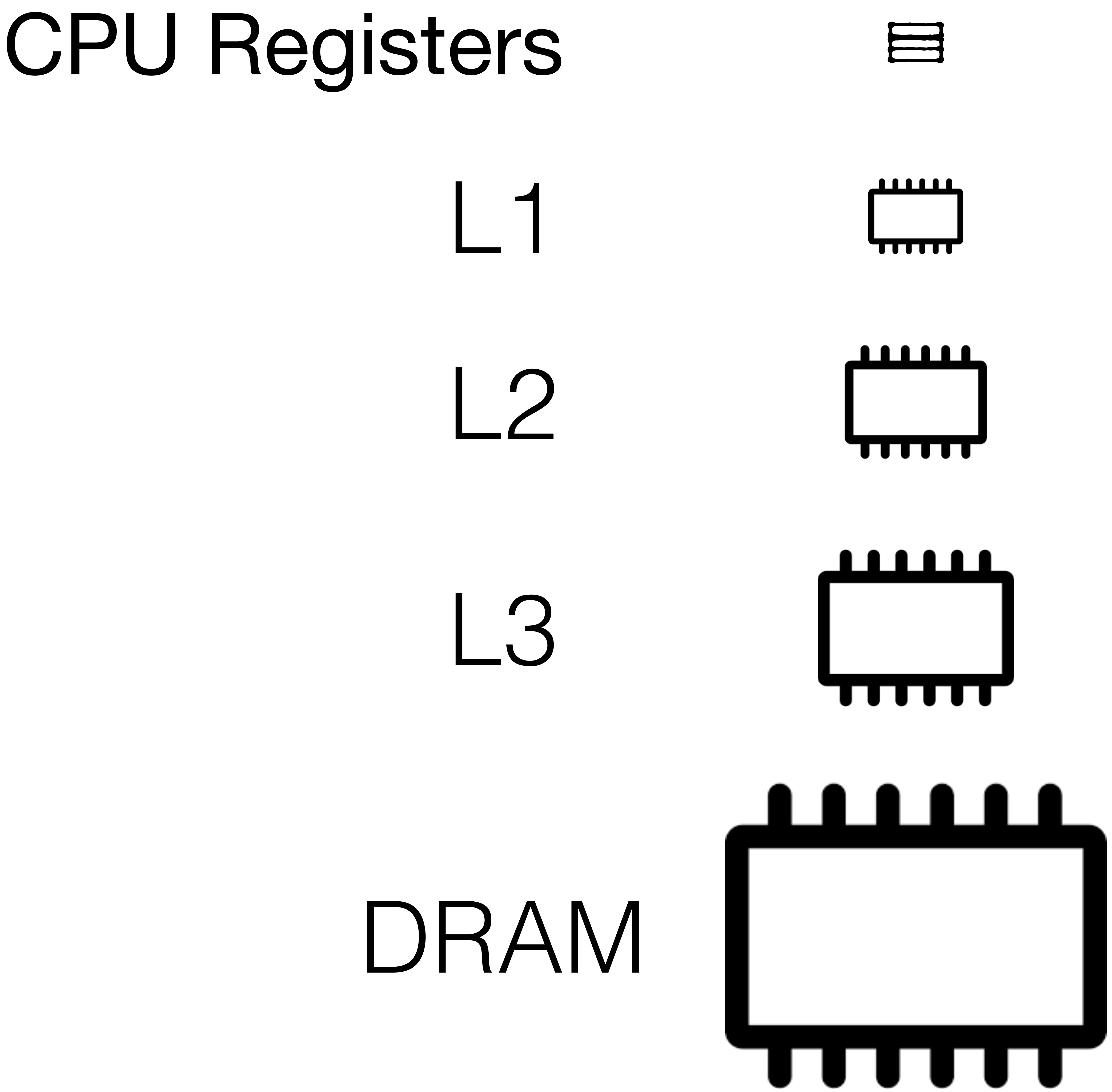
Insertion = $M/N \cdot \ln(2)$

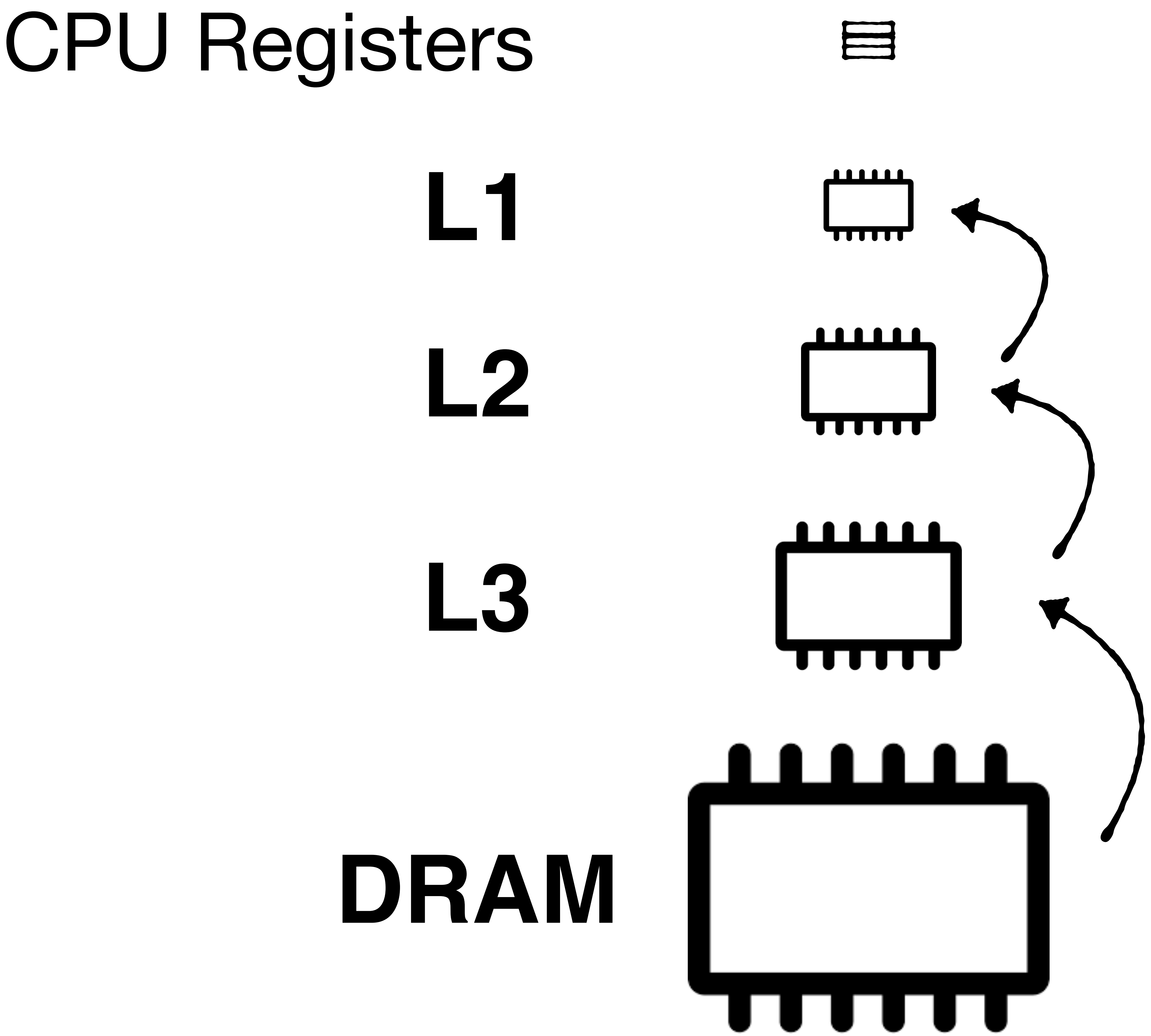
Positive Query = $M/N \cdot \ln(2)$

Avg. Negative Query = 2

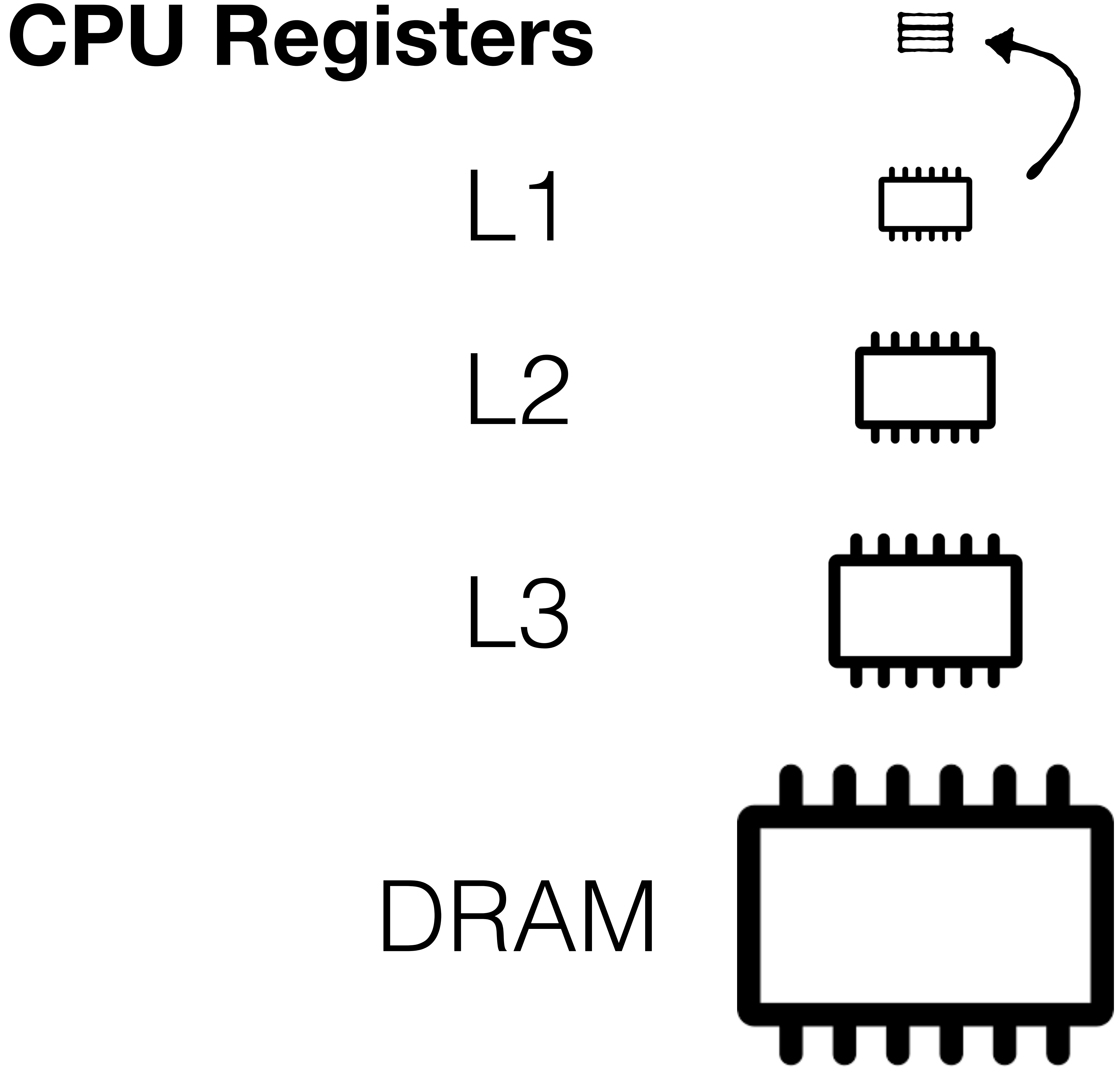
Is this ok for modern hardware?

Recall the memory hierarchy

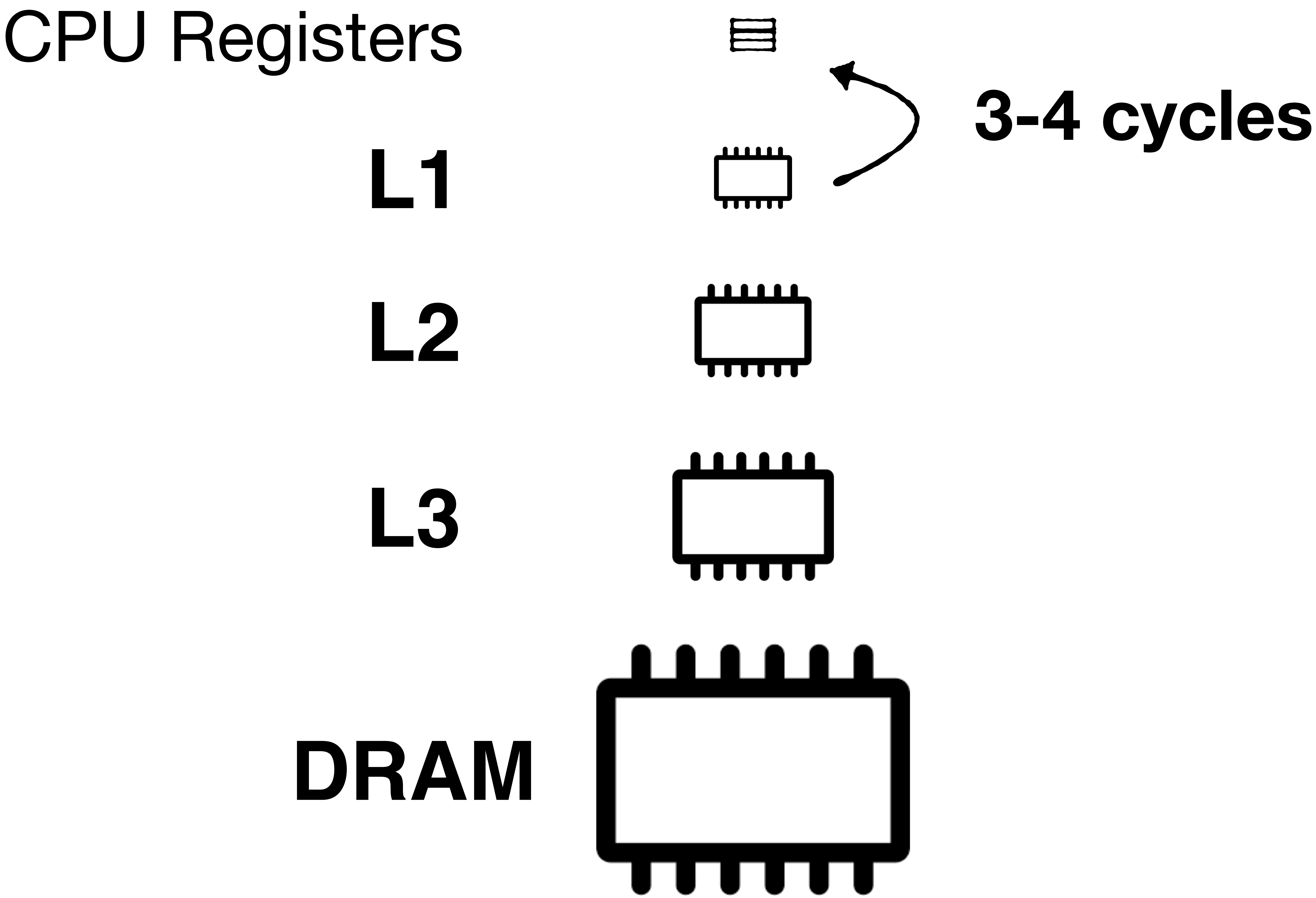


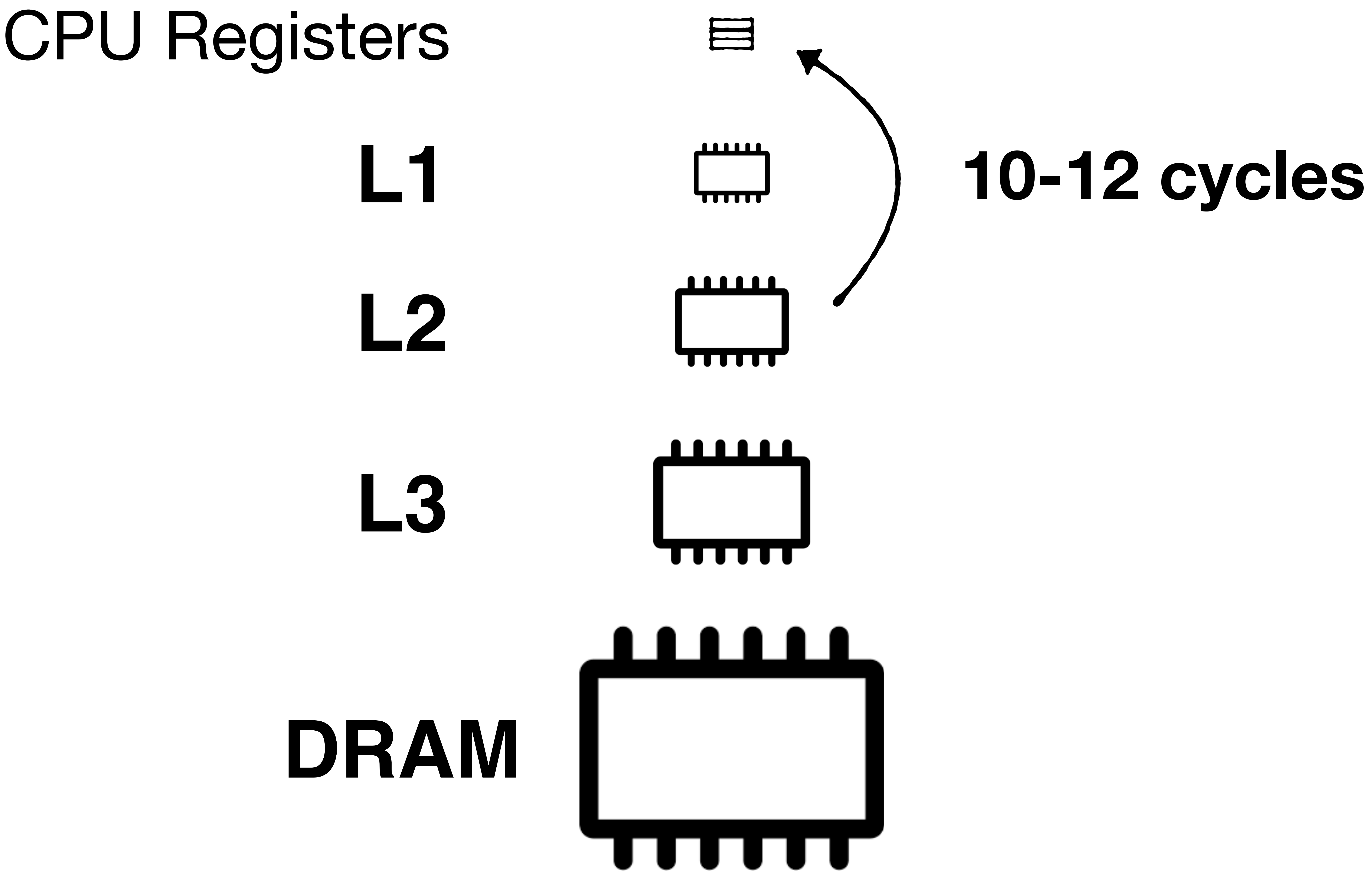


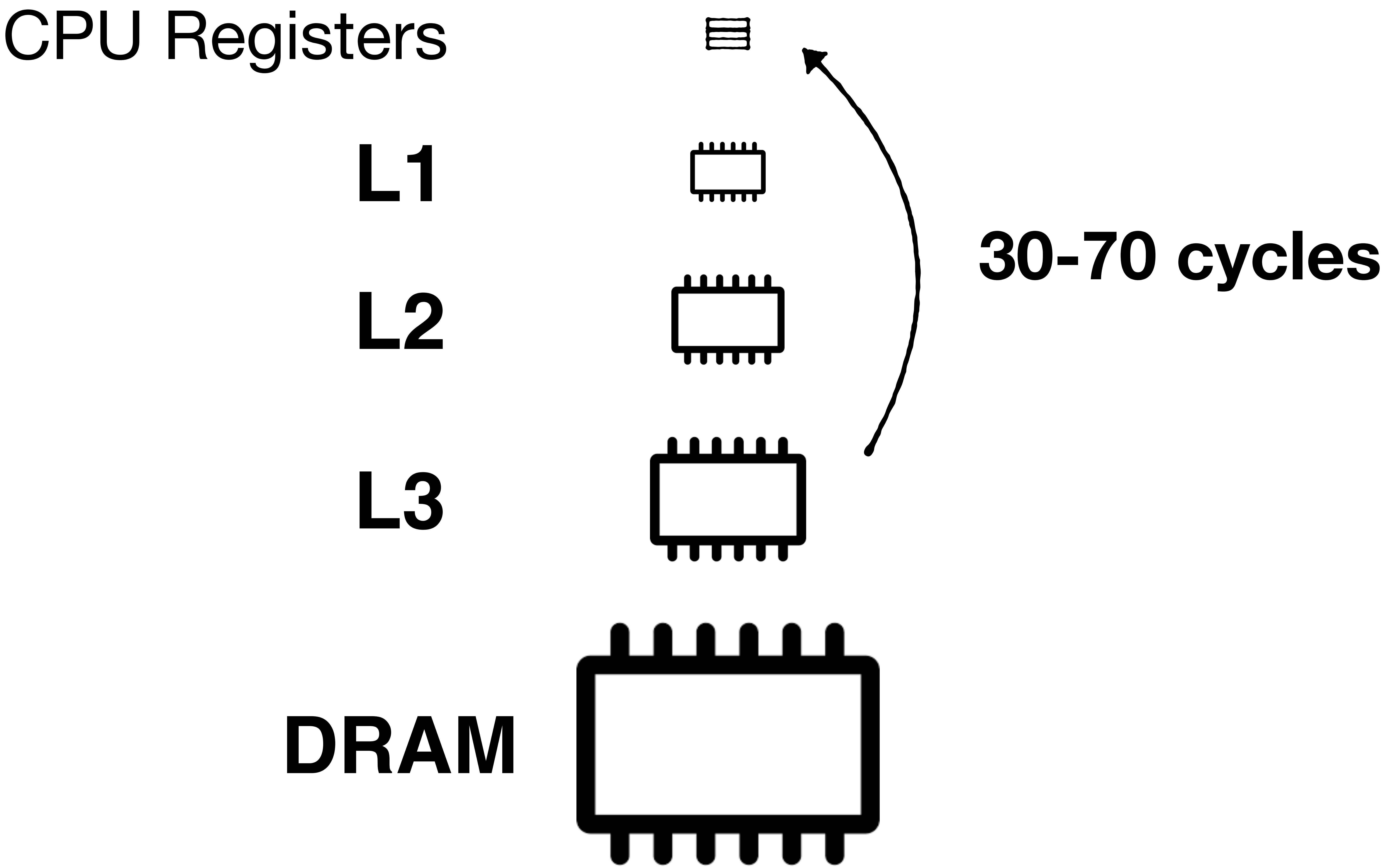
Move data at "cache line" granularity (e.g., 64B)

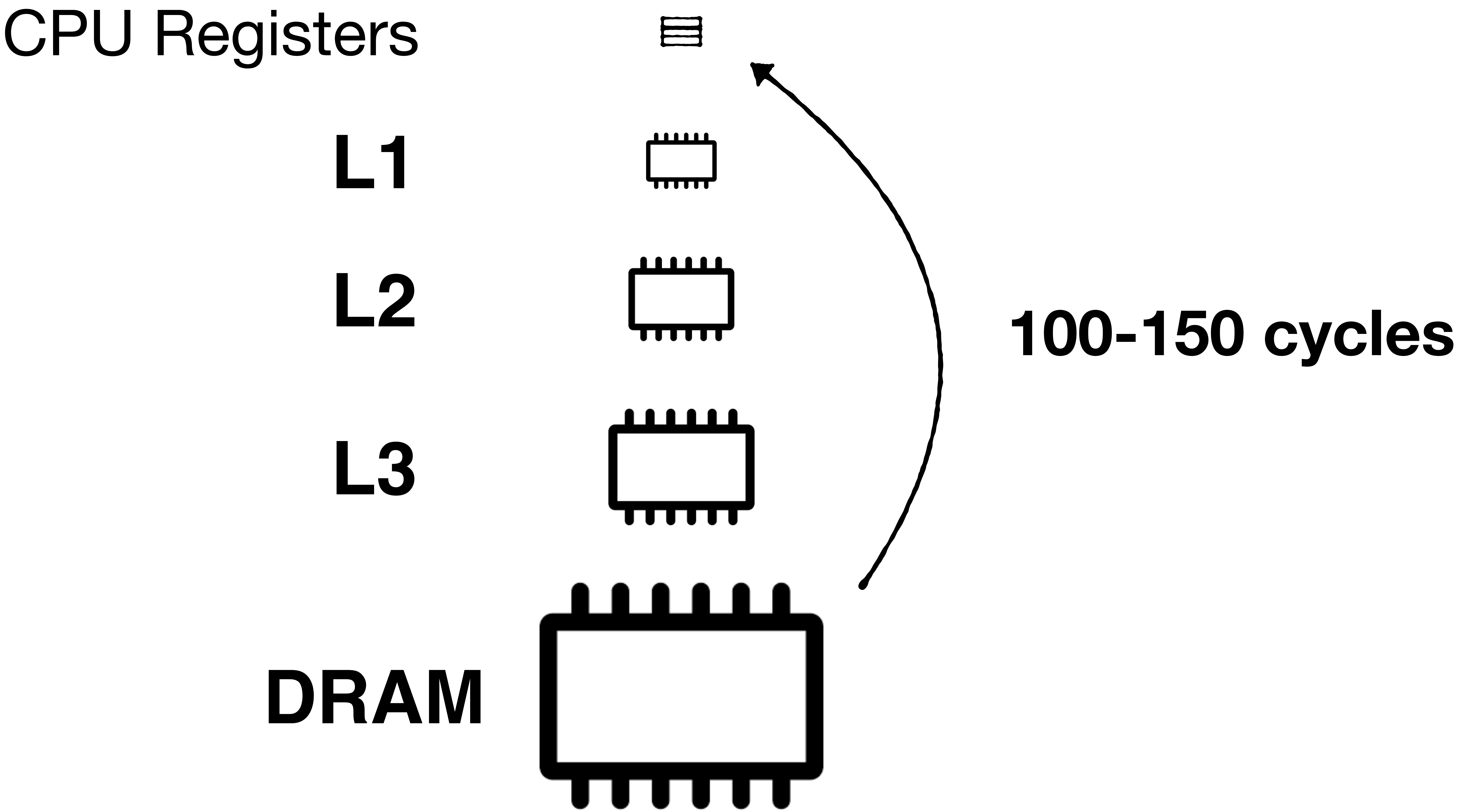


Move data at "word" granularity (e.g., 8B)


CPU Registers
3-4 cycles
L1
L2
L3
DRAM


CPU Registers
L1
L2
L3
DRAM
10-12 cycles


CPU Registers
L1
L2
L3
DRAM
30-70 cycles


CPU Registers
L1
L2
L3
DRAM
100-150 cycles

Source: http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/

(Numbers from **2016)**

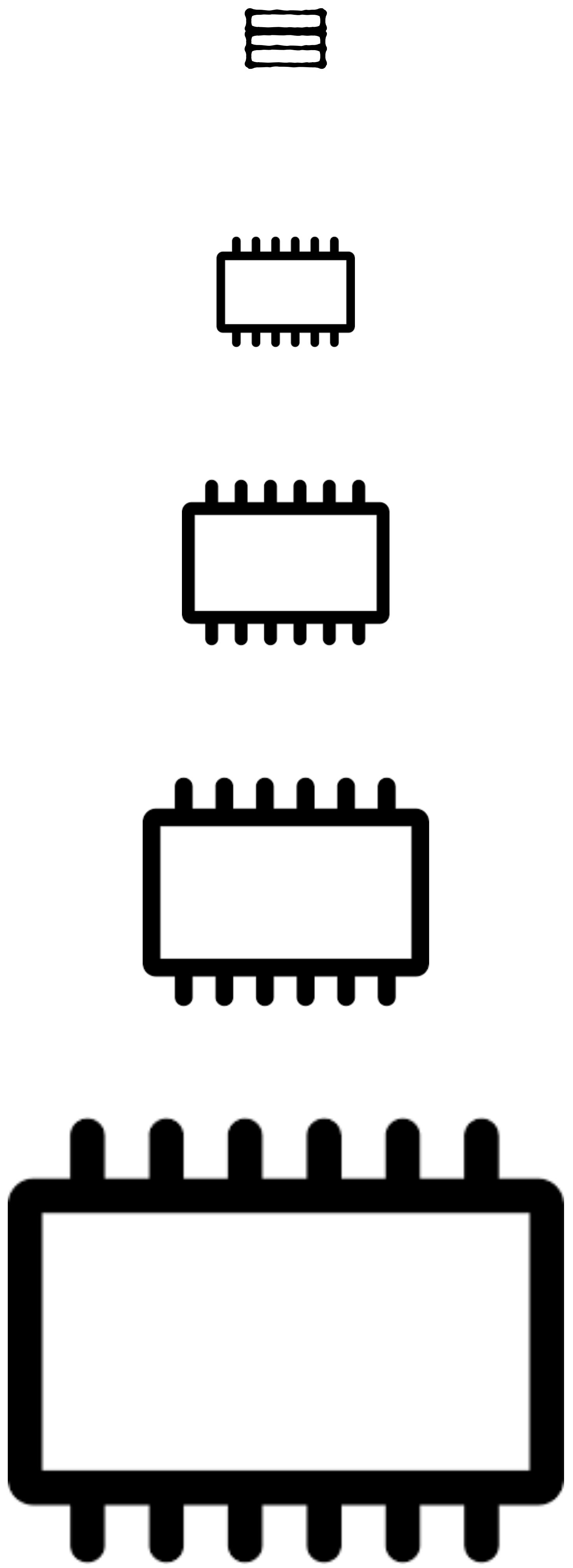
CPU Registers

L1

L2

L3

DRAM



Each hash function can lead to a cache miss

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

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

Avg. Negative Query = 2

Each hash function can lead to a cache miss

Insertion = $M/N \cdot \ln(2) \cdot 100$ **ns**

Positive Query = $M/N \cdot \ln(2) \cdot 100$ **ns**

Avg. Negative Query = $2 \cdot 100$ **ns**

Observation

Basic Bloom filter is slowest filter

Observation

Basic Bloom filter is slowest filter

With hardware optimizations, it is the fastest

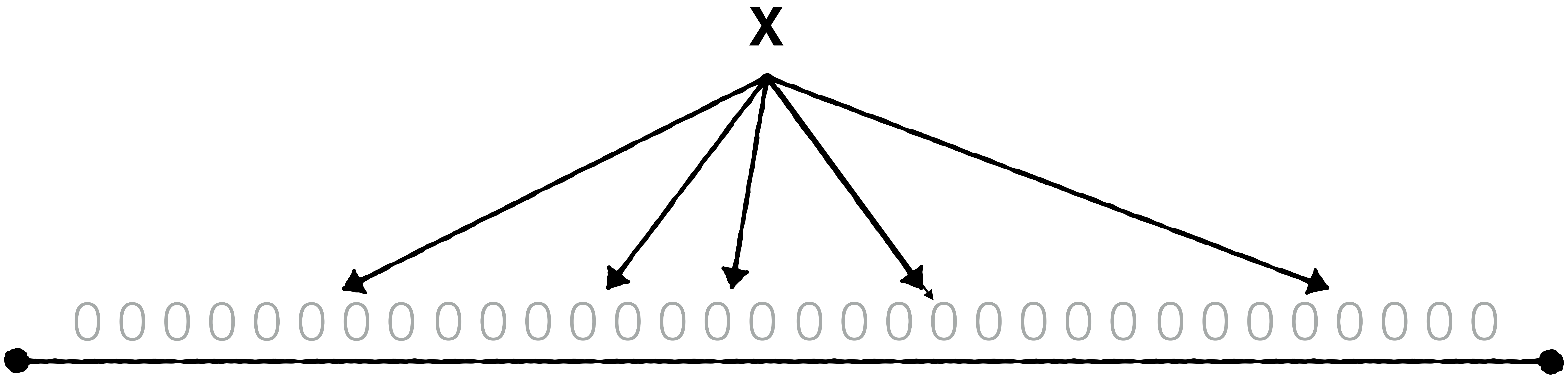
Observation

Basic Bloom filter is slowest filter

With hardware optimizations, it is the fastest

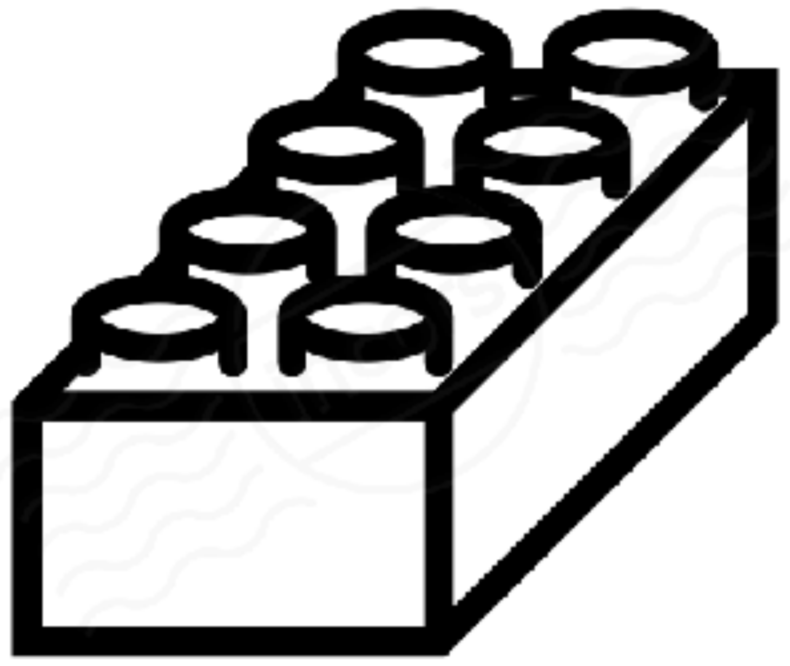
lends itself to hardware optimization

Alleviate random accesses?



Bloom filter

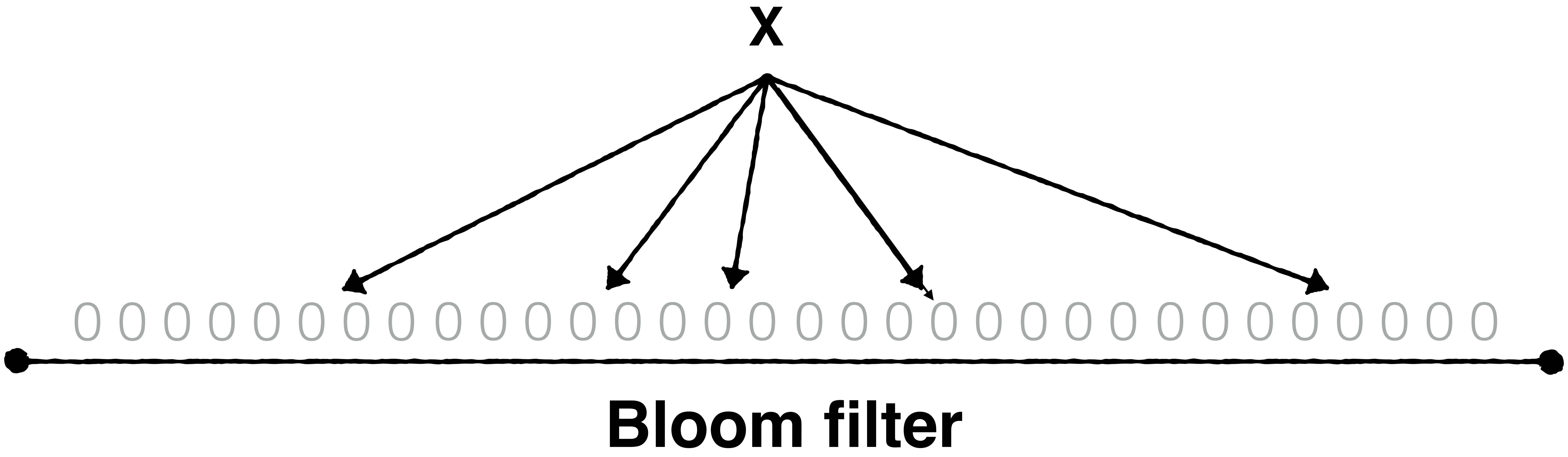
Blocked Bloom Filters



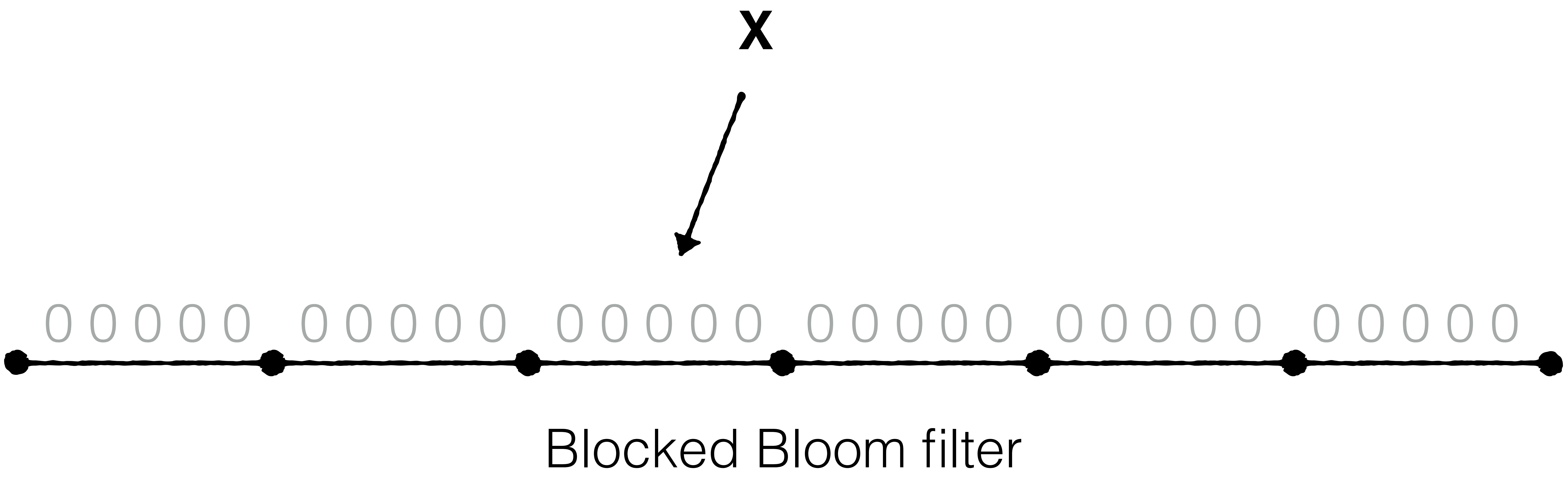
Cache-, Hash- and Space-Efficient Bloom Filters

Journal of Experimental Algorithms, 2010

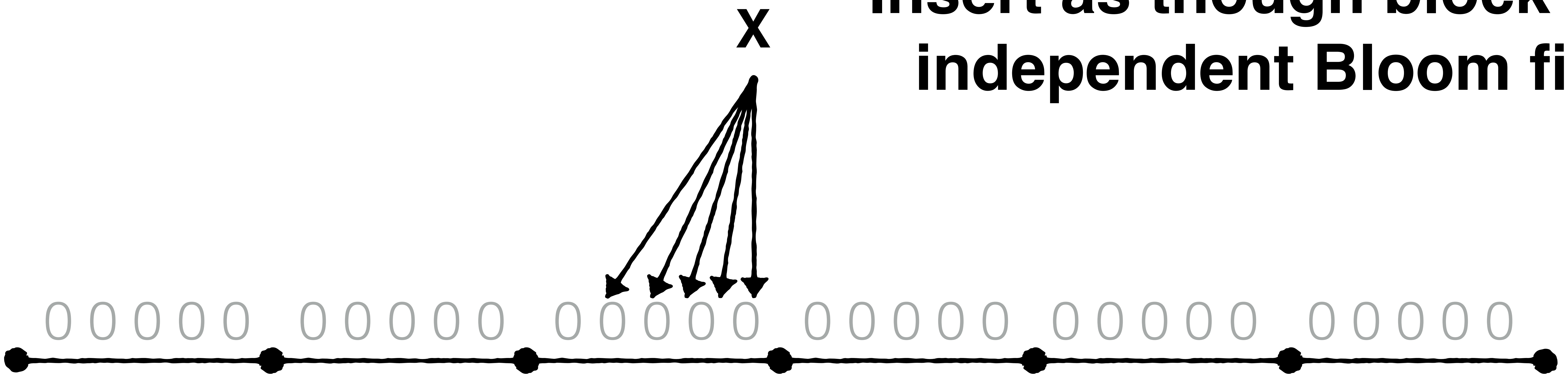
Felix Putze, Peter Sanders, Johannes Singler


x
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Bloom filter

Hash to one block, sized as a cache line

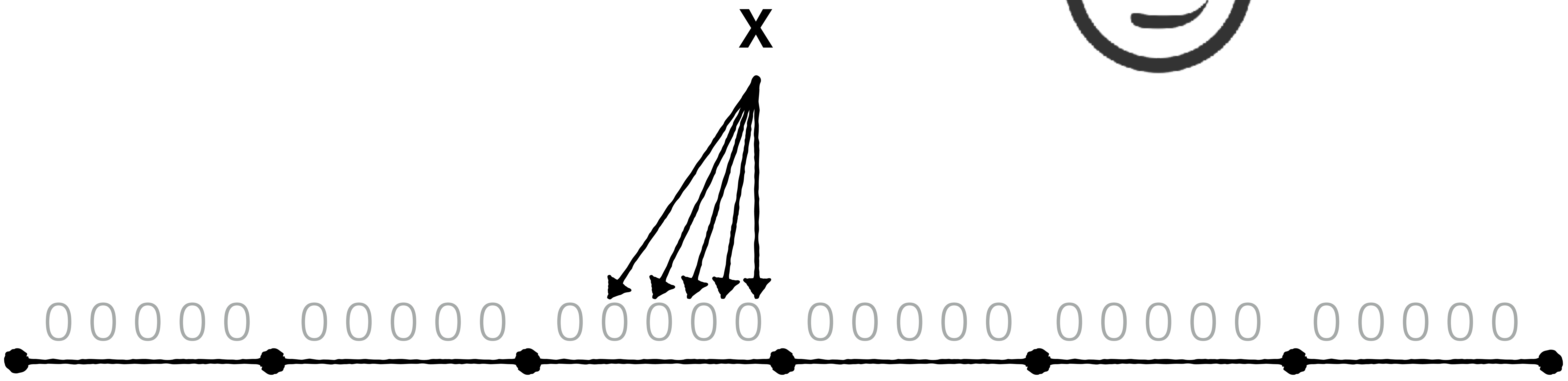


Insert as though block is an independent Bloom filter



Blocked Bloom filter

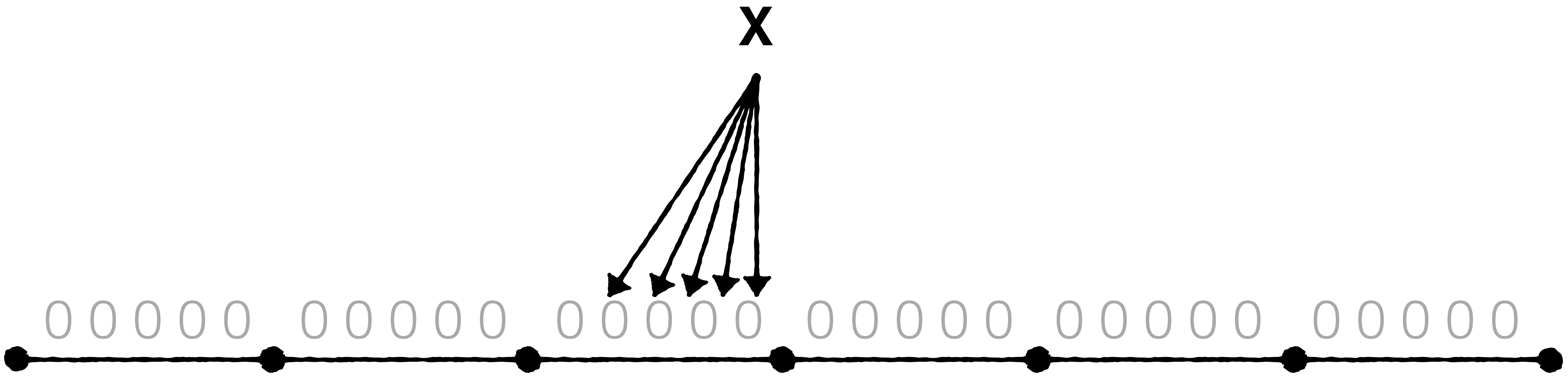
1 cache miss per query/insertion



Blocked Bloom filter

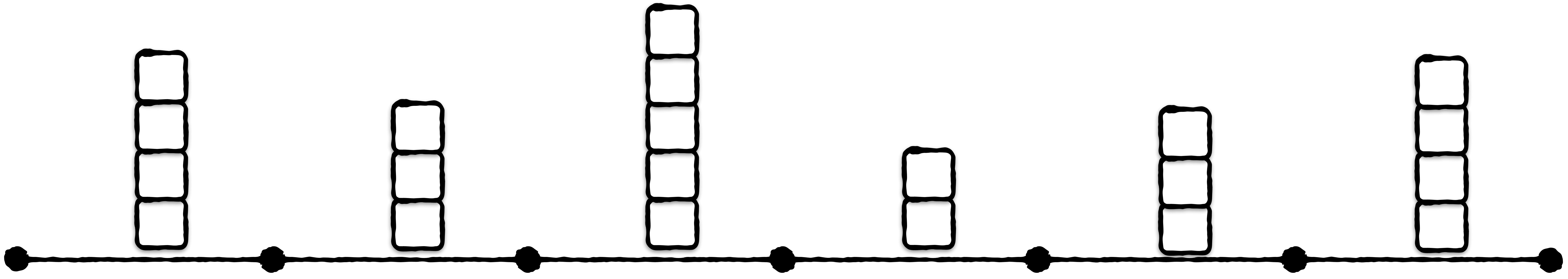
1 cache miss per query/insertion

Anything bad?



Blocked Bloom filter

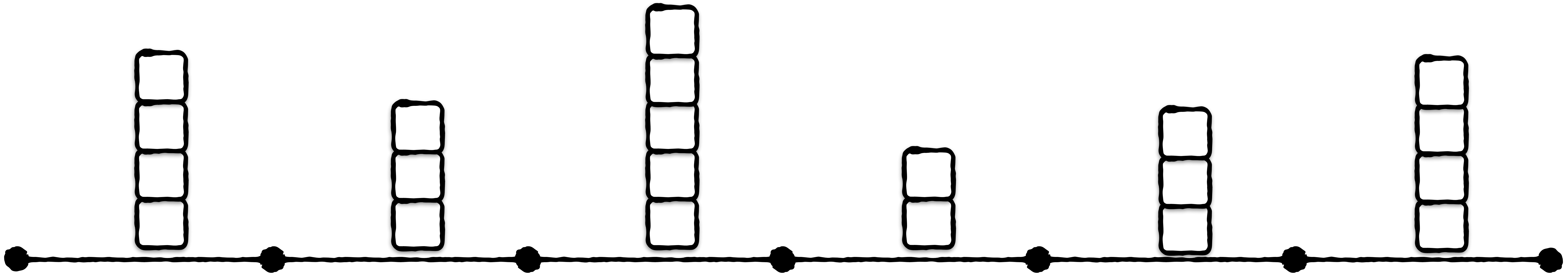
uneven distribution of entries across blocks



Blocked Bloom filter

uneven distribution of entries across blocks

impact on FPR?

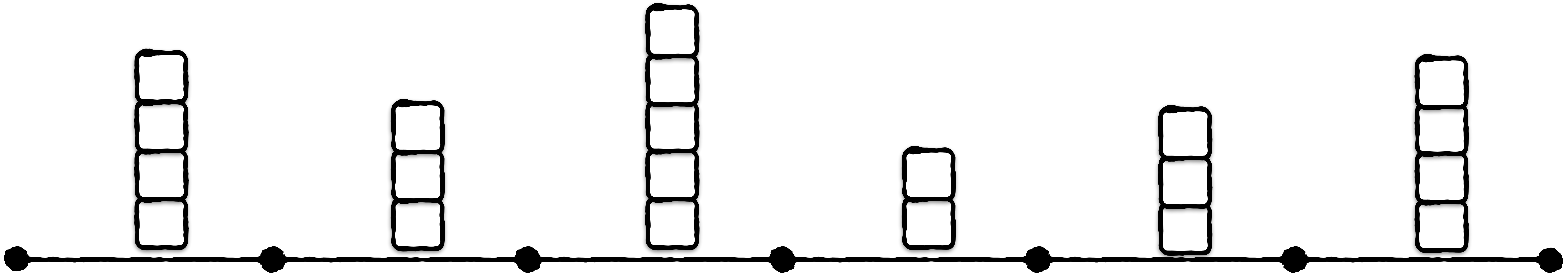


Blocked Bloom filter

uneven distribution of entries across blocks

impact on FPR?

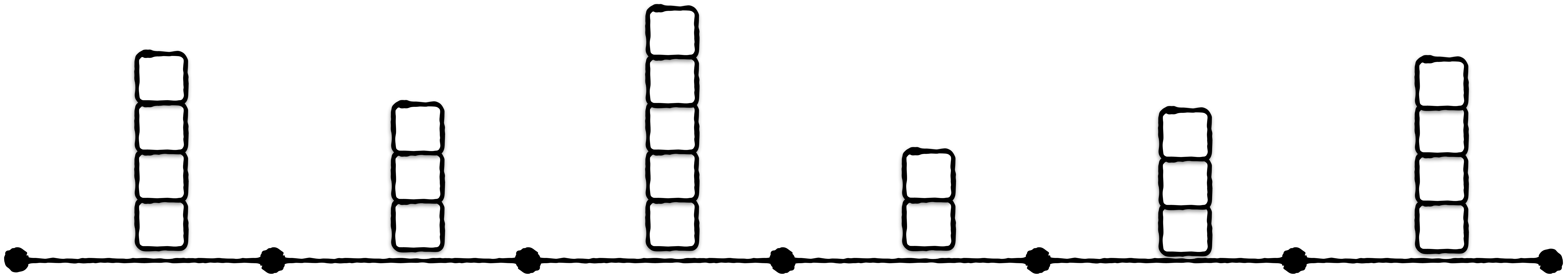
General analysis technique for hash tables



Blocked Bloom filter

entries per block follows binomial distribution

$$f(n,p,i) = \binom{n}{i} \cdot p^i \cdot (1-p)^{n-i}$$

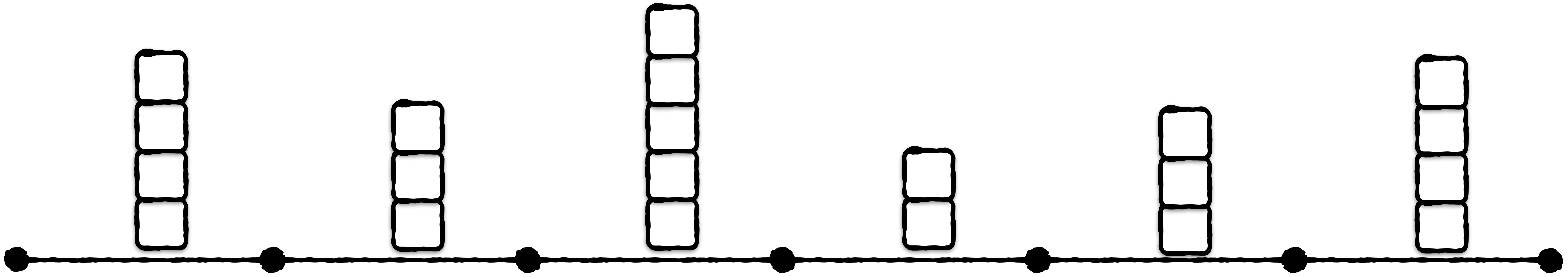


Blocked Bloom filter

entries per block follows binomial distribution

$$f(n, p, i) = \binom{n}{i} \cdot p^i \cdot (1-p)^{n-i}$$

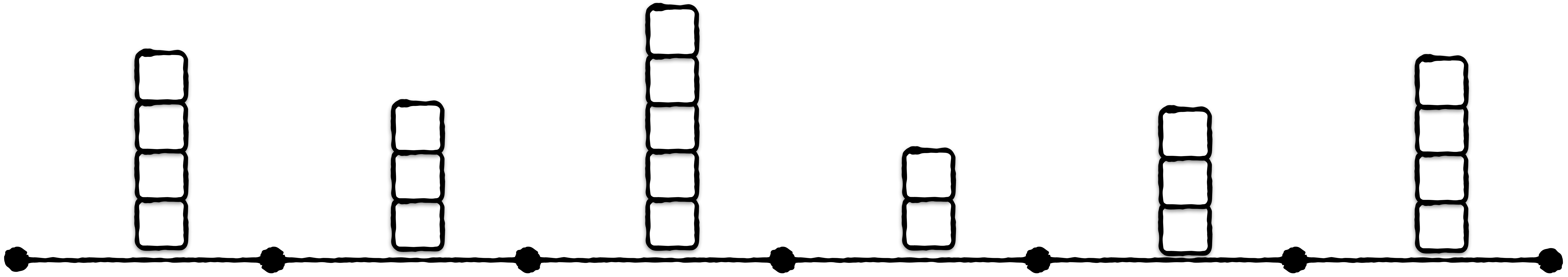
 # entries



entries per block follows binomial distribution

$$f(n, p, i) = \binom{n}{i} \cdot p^i \cdot (1 - p)^{n-i}$$

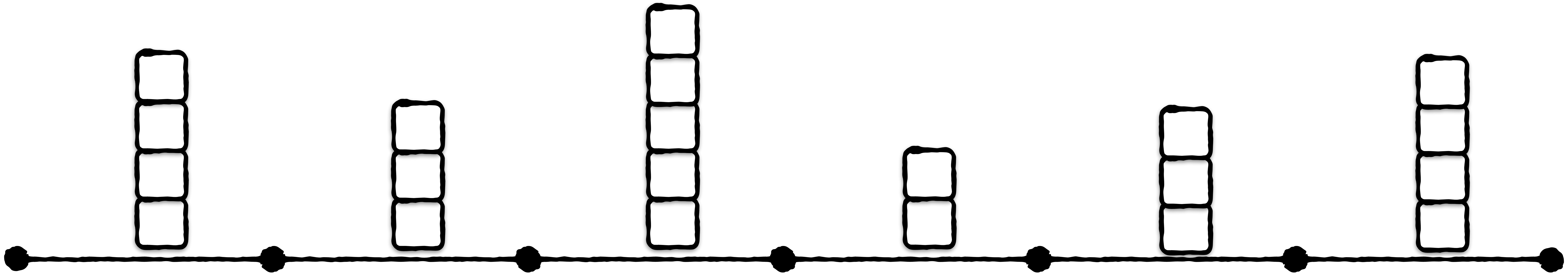
Prob of 1 entry falling into a given block, i.e., 1/n



entries per block follows binomial distribution

$$f(n, p, i) = \binom{n}{i} \cdot p^i \cdot (1 - p)^{n-i}$$

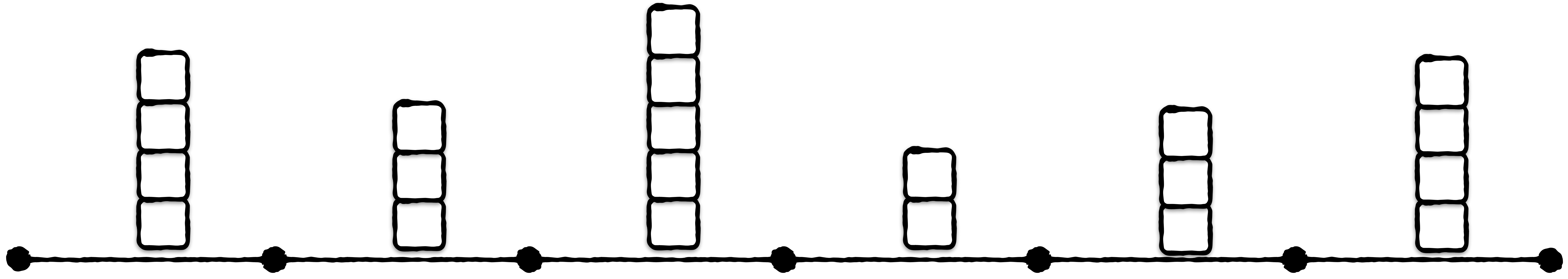
i entries falling into our bucket



entries per block follows binomial distribution

$$f(n, p, i) = \binom{n}{i} \cdot p^i \cdot (1 - p)^{n-i}$$

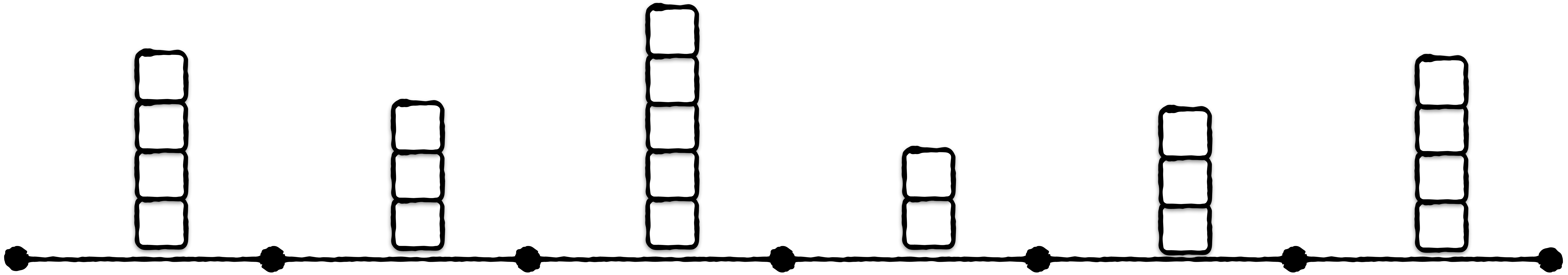
Ways of choosing i out of n entries



entries per block follows binomial distribution

$$f(n, p, i) = \binom{n}{i} \cdot p^i \cdot (1 - p)^{n-i}$$

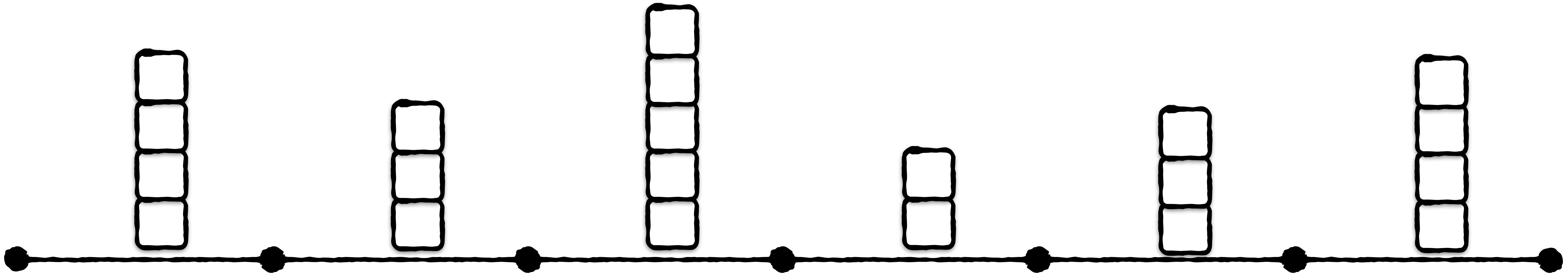
i entries falling into our block



entries per block follows binomial distribution

$$f(n,p,i) = \binom{n}{i} \cdot p^i \cdot (1-p)^{n-i}$$

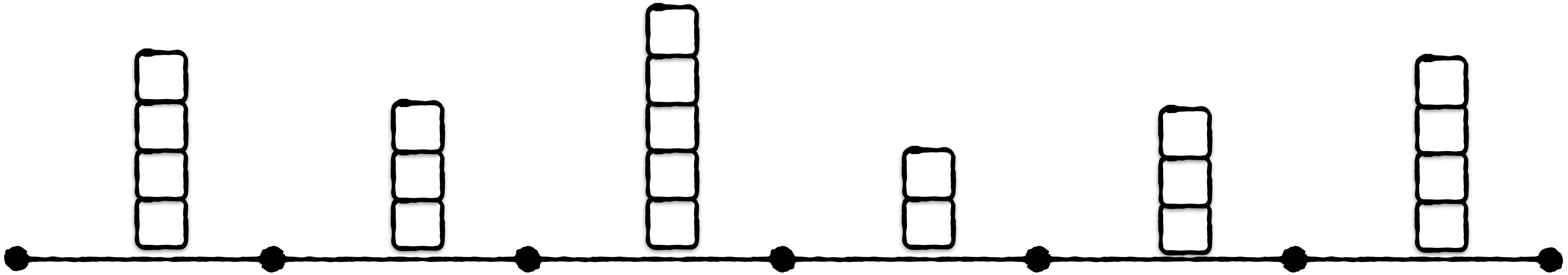
All other entries falling into other blocks



entries per block follows binomial distribution

$$f(n,p,i) = \binom{n}{i} \cdot p^i \cdot (1-p)^{n-i}$$

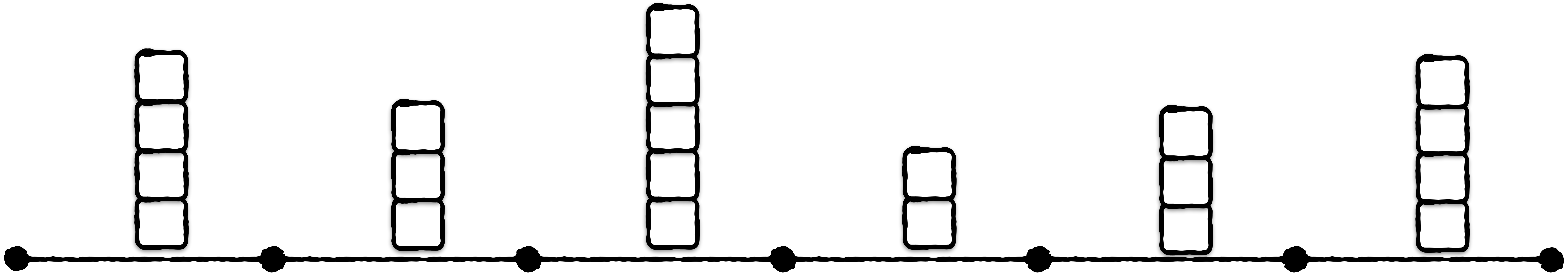
Cumbersome



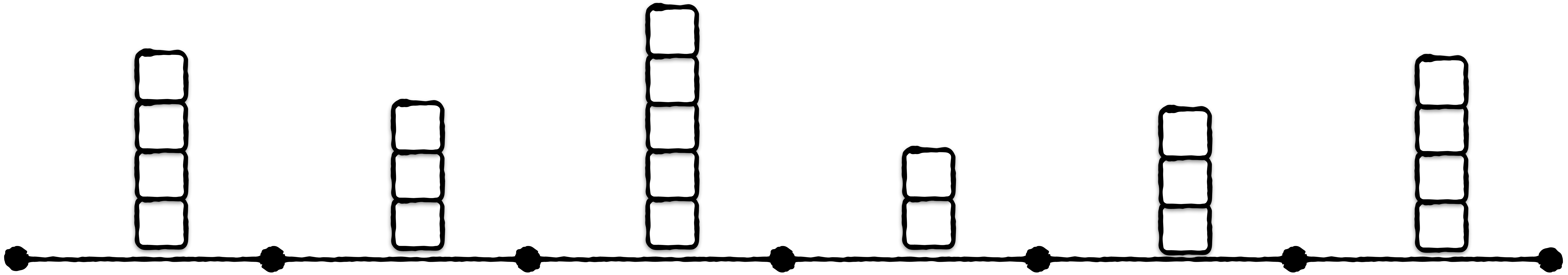
entries per block follows binomial distribution

$$f(n, p, i) = \binom{n}{i} \cdot p^i \cdot (1-p)^{n-i}$$

For $n \rightarrow \infty$ & $p \rightarrow 0$, binomial converges to Poisson

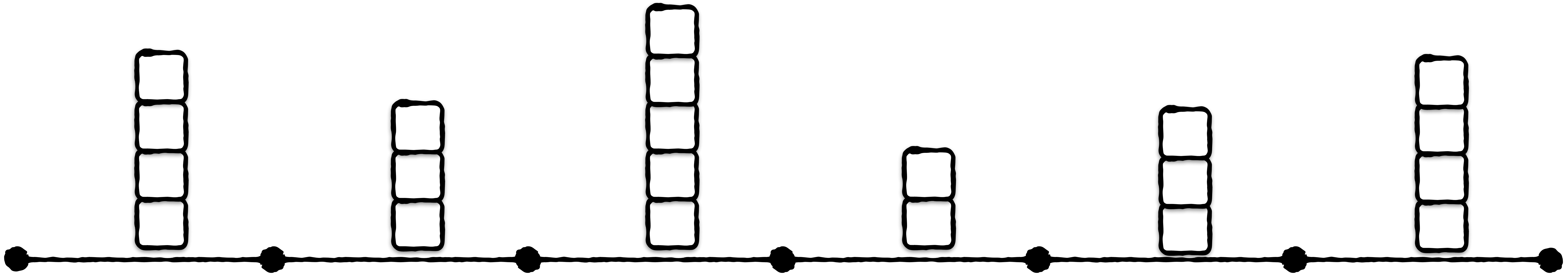


P[i entries fall into given block] ~ Poisson(i, λ) = $\dfrac{\lambda^i \cdot e^{-\lambda}}{i!}$



P[i entries fall into given block] ~ Poisson(i, λ) = $\dfrac{\lambda^i \cdot e^{-\lambda}}{i!}$

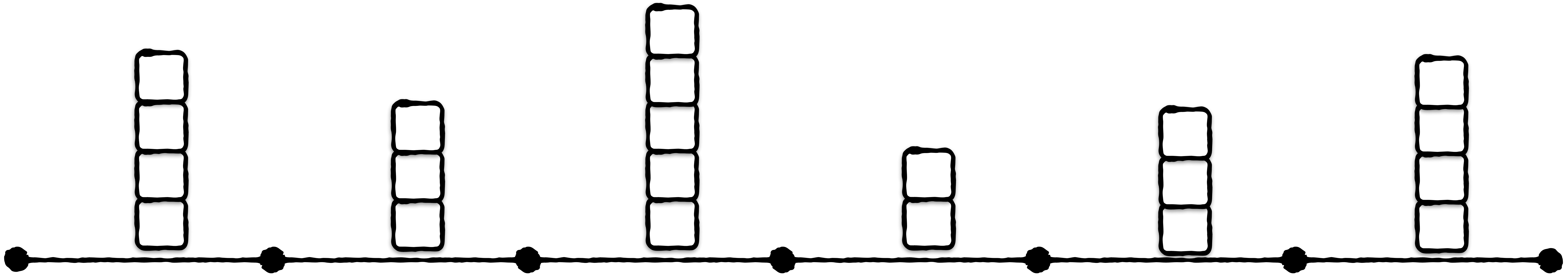
λ = avg. entries per block



P[i entries fall into given block] ~ Poisson(i, λ) = $\frac{\lambda^i \cdot e^{-\lambda}}{i!}$

λ = avg. entries per block **= B**/(M/N)

Bits per cache line (e.g., 256)

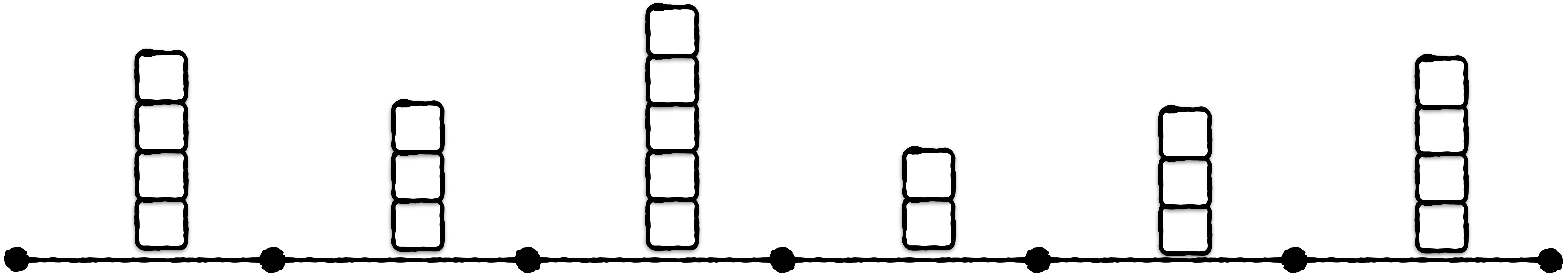


P[i entries fall into given block] ~ Poisson(i, λ) = $\dfrac{\lambda^i \cdot e^{-\lambda}}{i!}$

λ = avg. entries per block = B/**(M/N)**

Bits per cache line (e.g., 256)

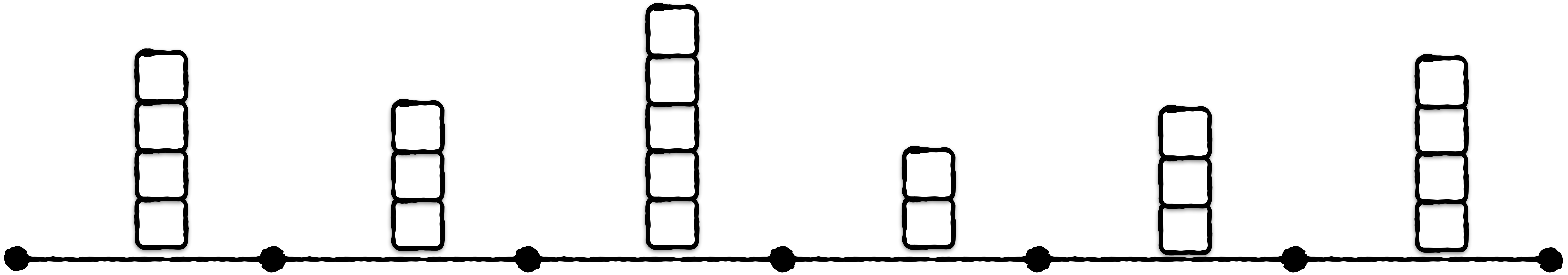
Bits per entry (e.g., 8)



$$P[\text{i entries fall into given block}] \sim \text{Poisson}(i, \lambda) = \frac{\lambda^i \cdot e^{-\lambda}}{i!}$$

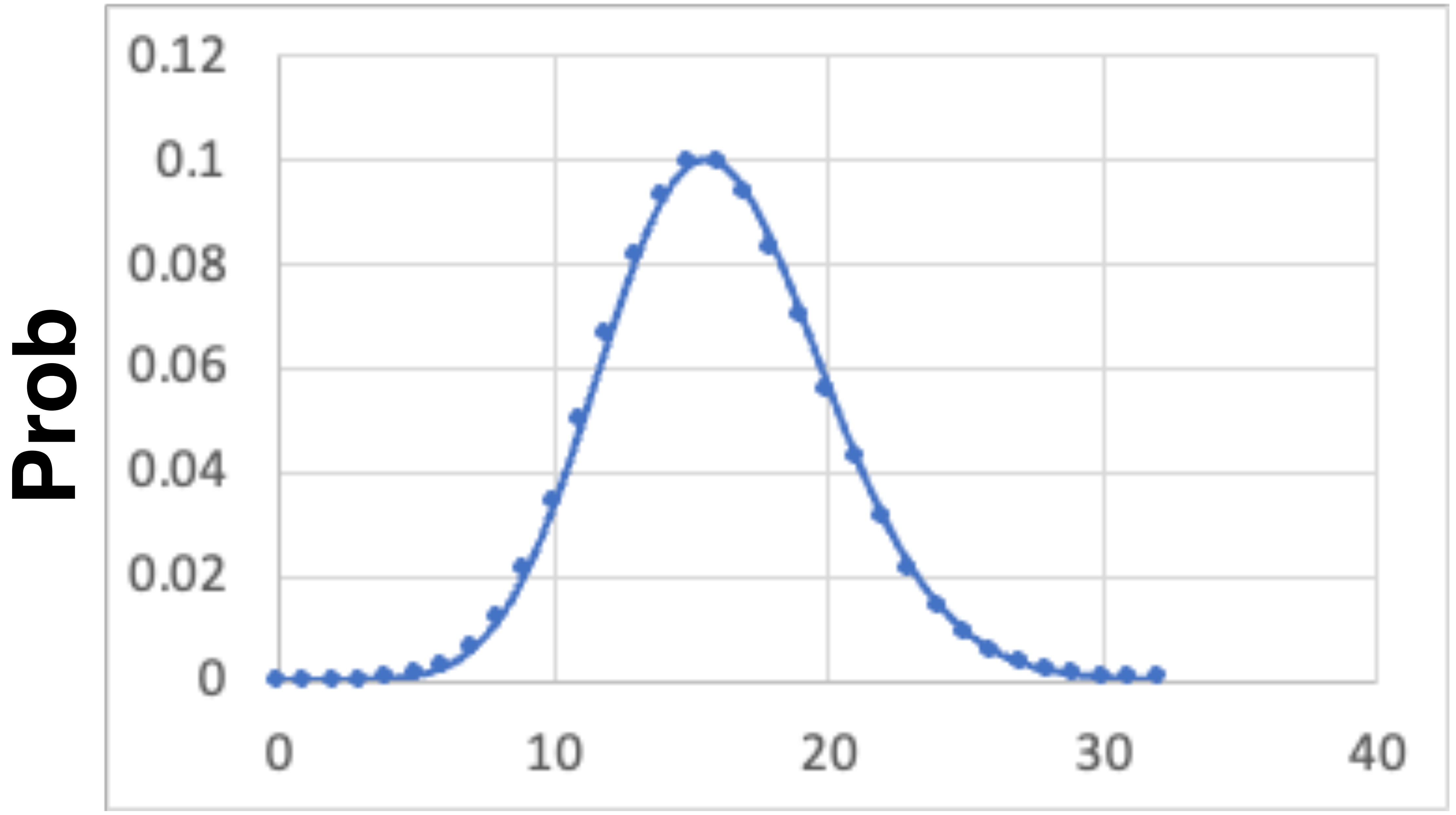
λ = avg. entries per block = B/(M/N)

256 / 8 = 32



P[i entries fall into given block] ~ Poisson(i, λ) = $\dfrac{\lambda^i \cdot e^{-\lambda}}{i!}$

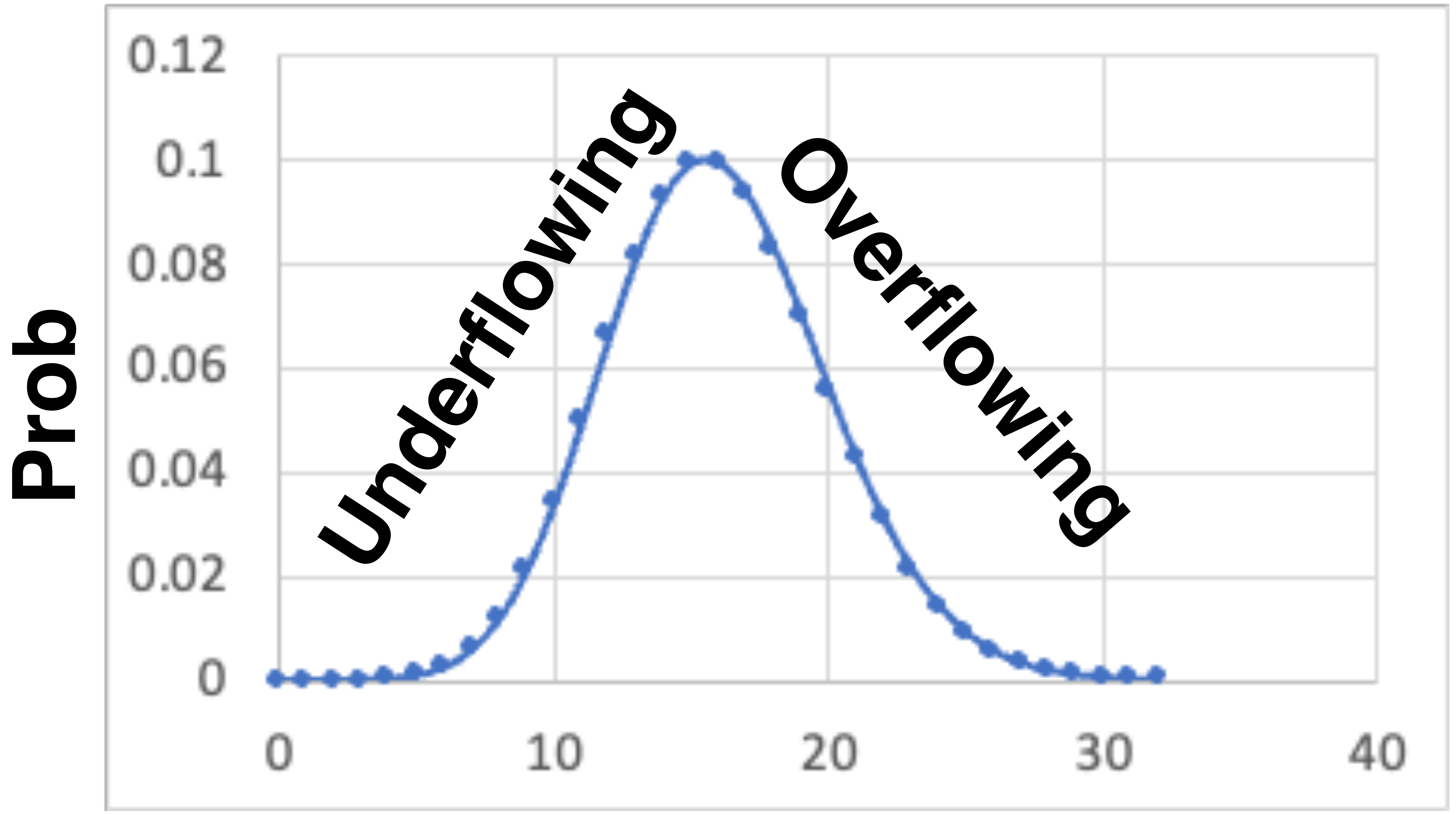
λ = avg. entries per block



Entries i per block with λ = 16

P[i entries fall into given block] ~ Poisson(i, λ) = $\dfrac{\lambda^i \cdot e^{-\lambda}}{i!}$

λ = avg. entries per block



Entries i per block with λ = 16

entries in a block ~ Poisson(i, B/(M/N))

entries in a block ~ Poisson(i, B/(M/N))

FPR for filter ~ $\mathrm{FPR}(N, M, K) = (1-e^{-KN/M})^K$

entries in a block ~ Poisson(i, B/(M/N))

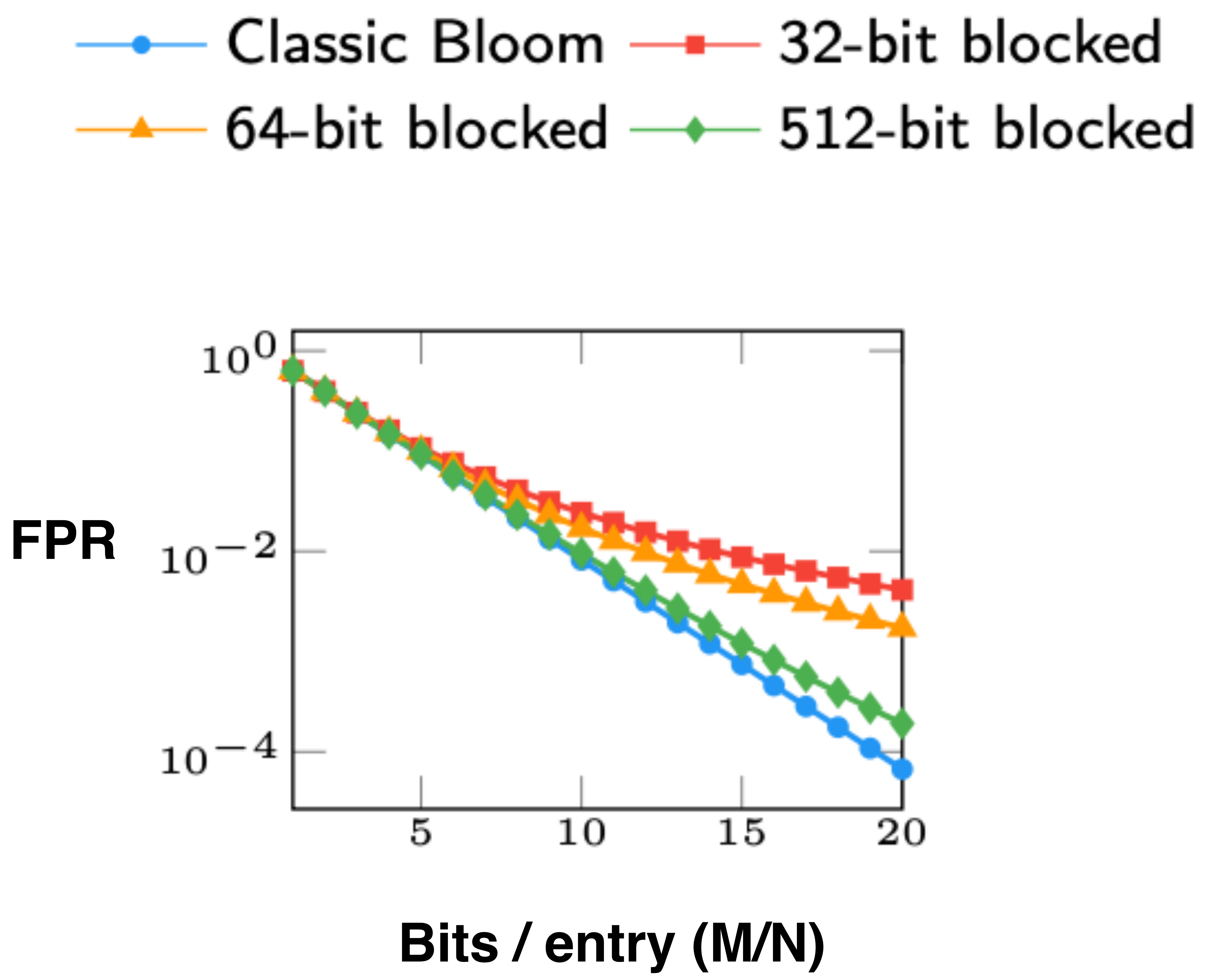
FPR for filter ~ FPR(**i**, **B, K**)

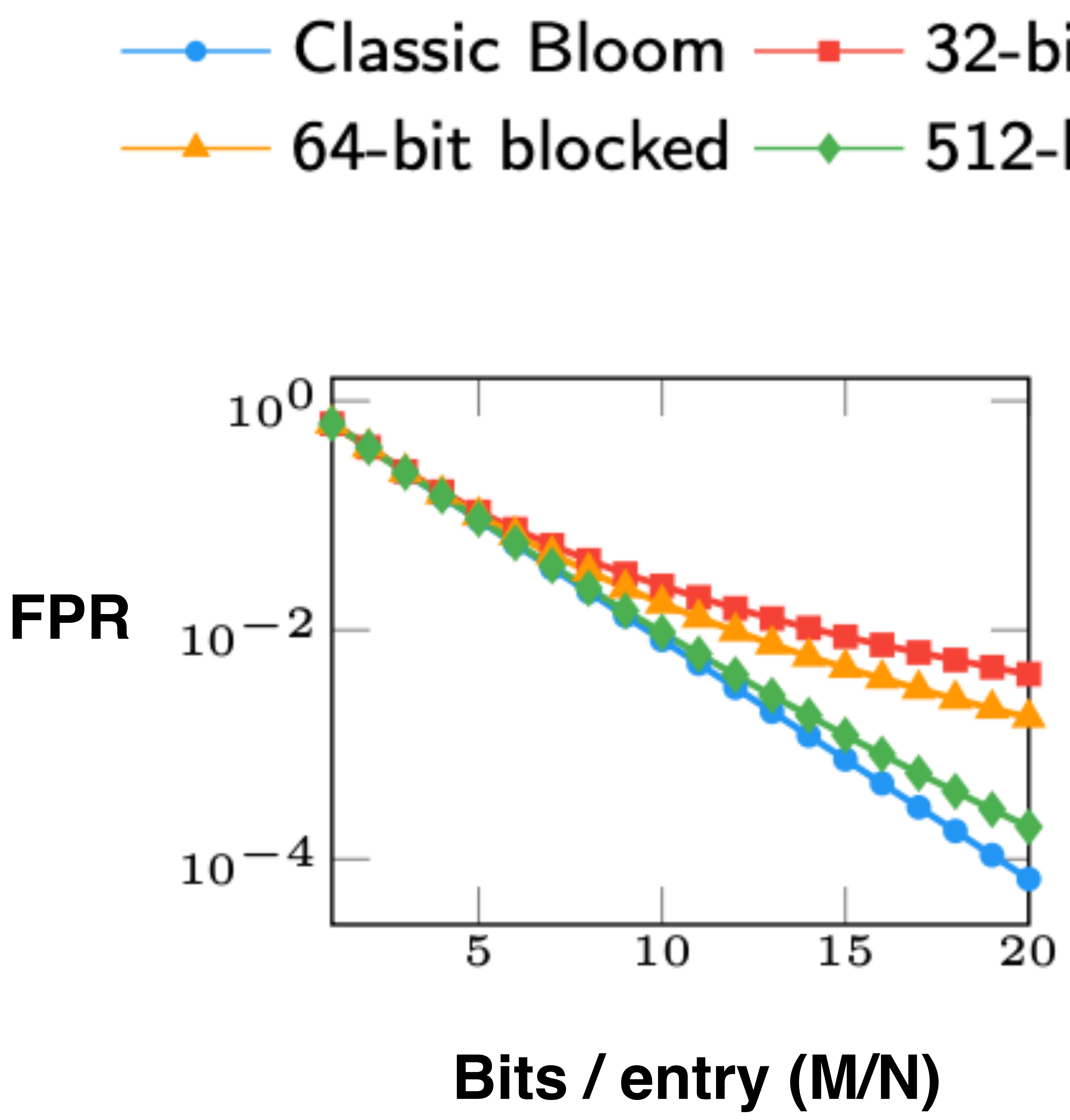
i entries in cache line

B bits in cache line

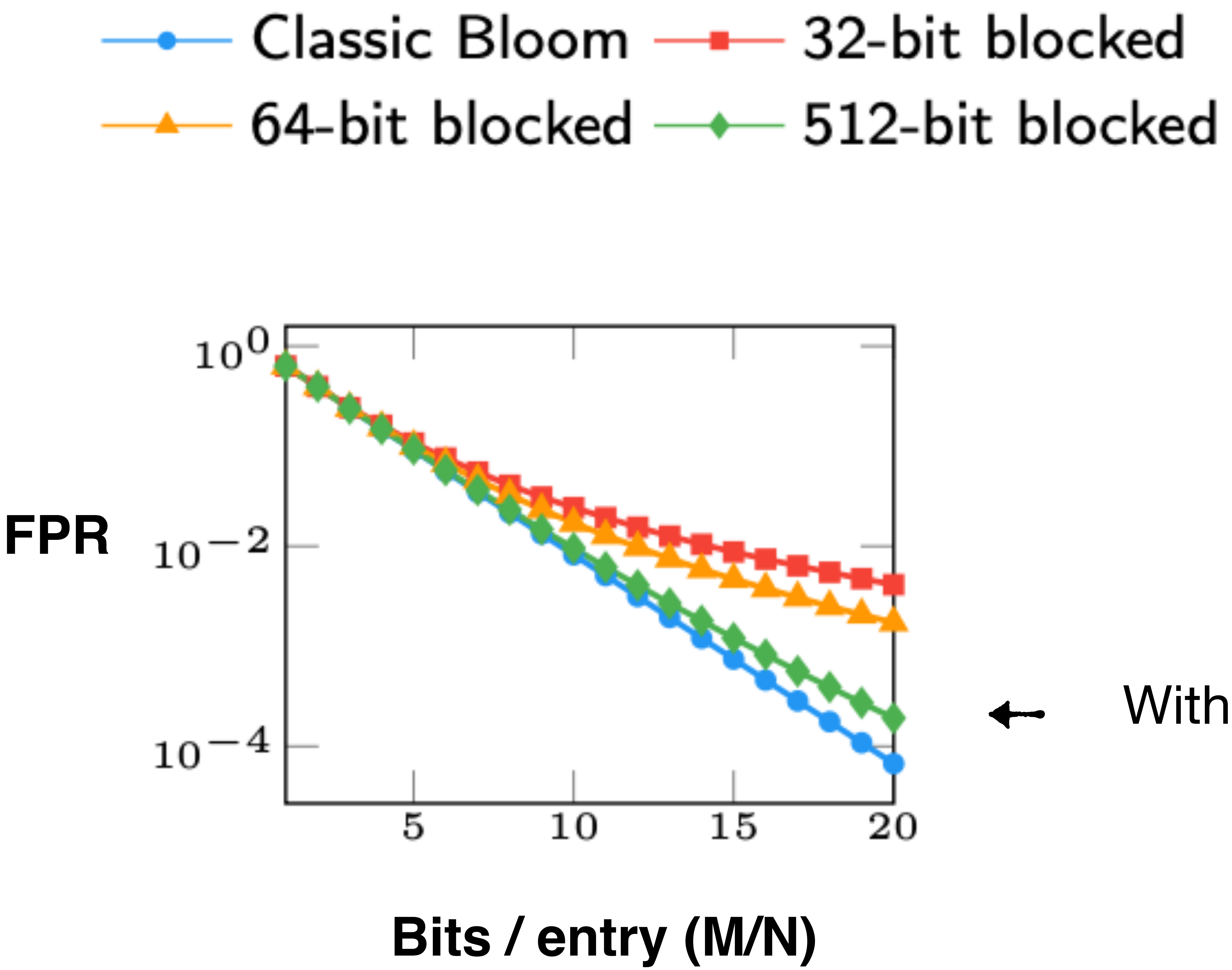
K hash functions

$$\textbf{Avg. FPR across all blocks} \quad \mathbf{=} \sum_{i=0}^{\infty} \text{Poisson}(i, B/(M/N)) \cdot \text{FPR}(i, B, K)$$


Classic Bloom
32-bit blocked
64-bit blocked
512-bit blocked
FPR
10^{0}
10^{-2}
10^{-4}
5
10
15
20
Bits / entry (M/N)

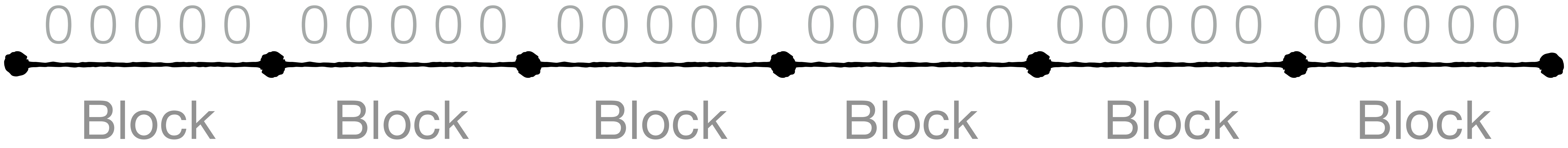


With smaller blocks, there is more variation in entries across blocks. Overflowing blocks blow up the FPR.

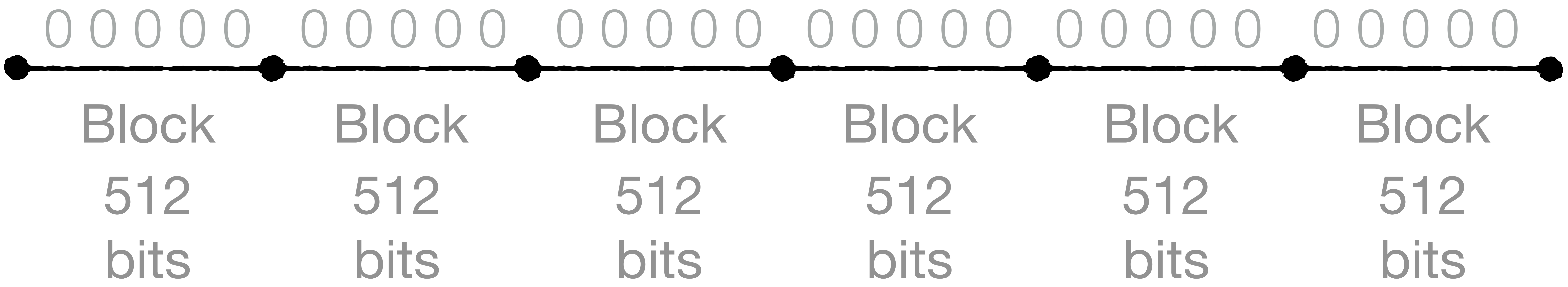


With large blocks, we don't lose much

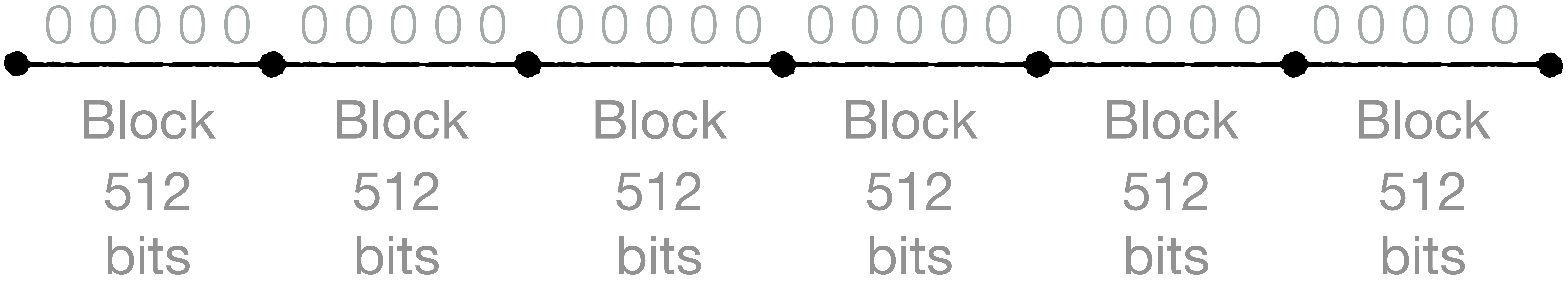
size blocks as cache lines



size blocks as cache lines

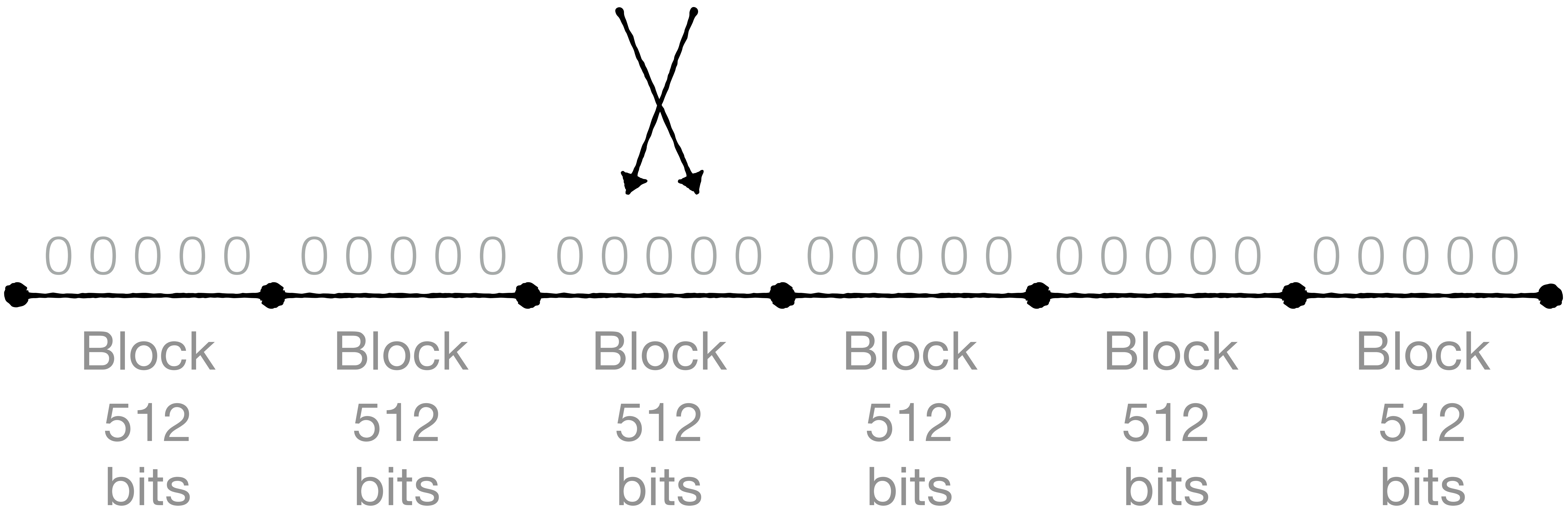


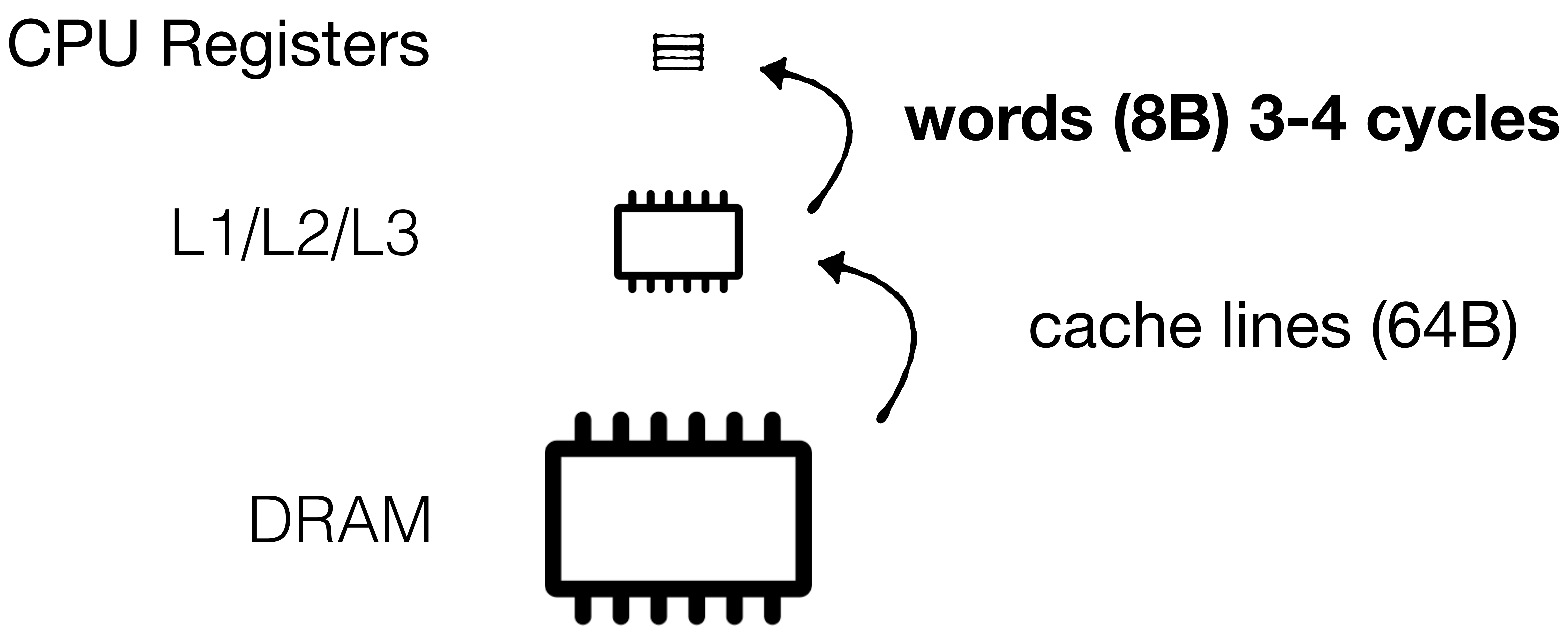
Any remaining issue?



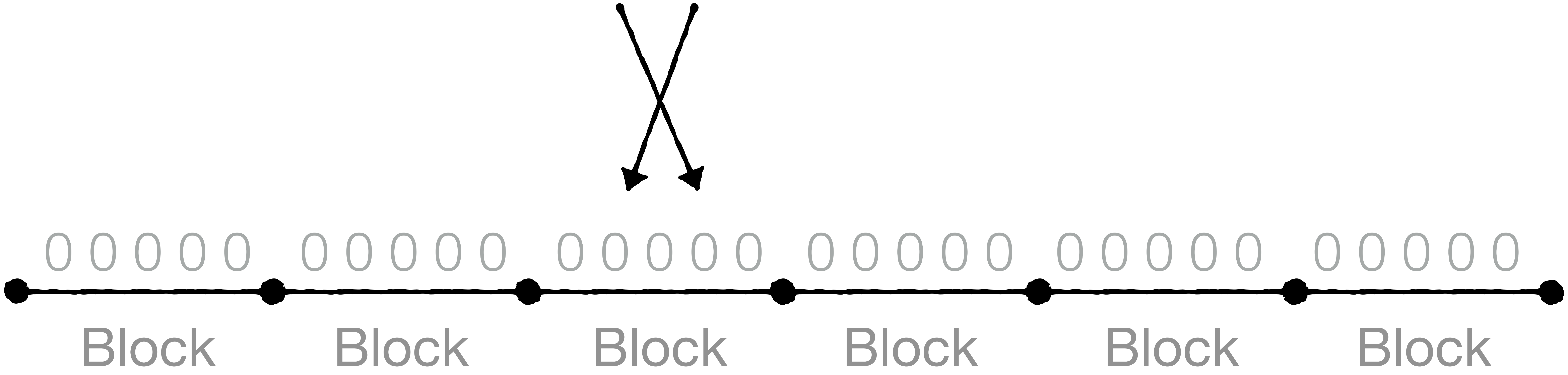
Any remaining issue?

Random access within a cache line

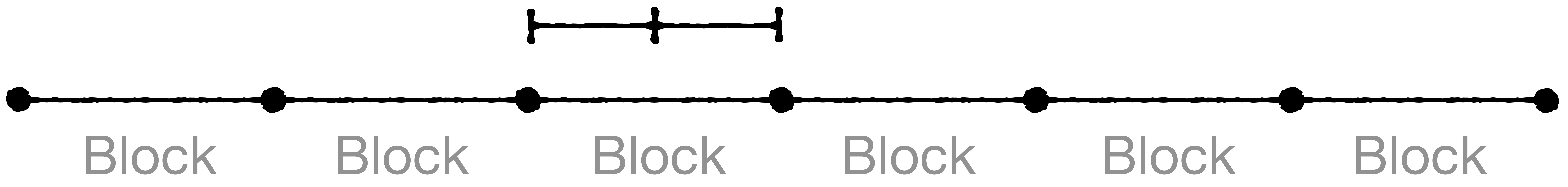



CPU Registers
L1/L2/L3
DRAM
words (8B) 3-4 cycles
cache lines (64B)

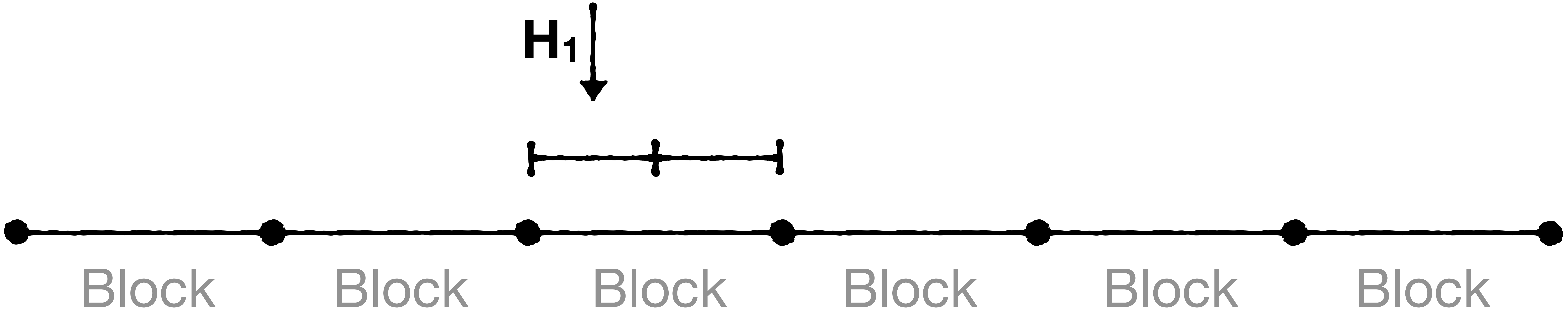
Each random access moves different word from L1 cache to register



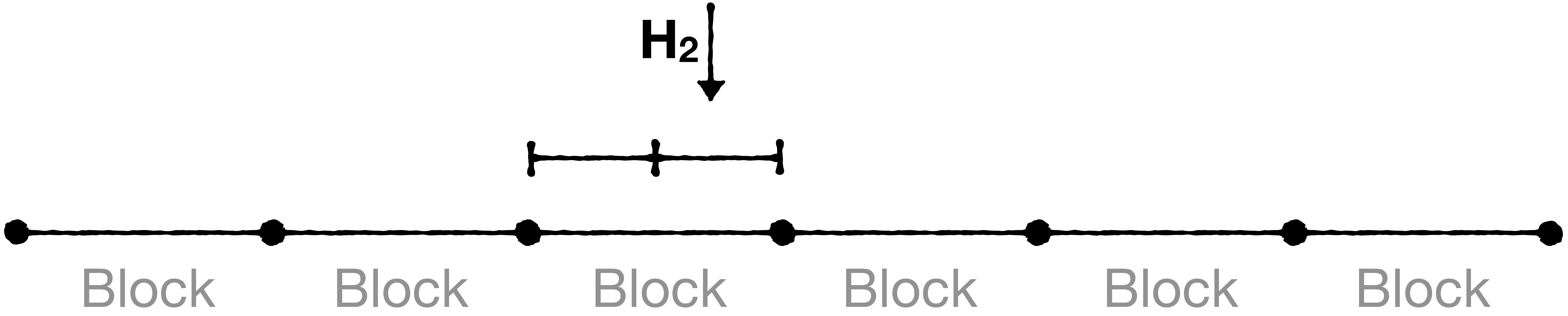
e.g., 4 hash functions, 2 words per block



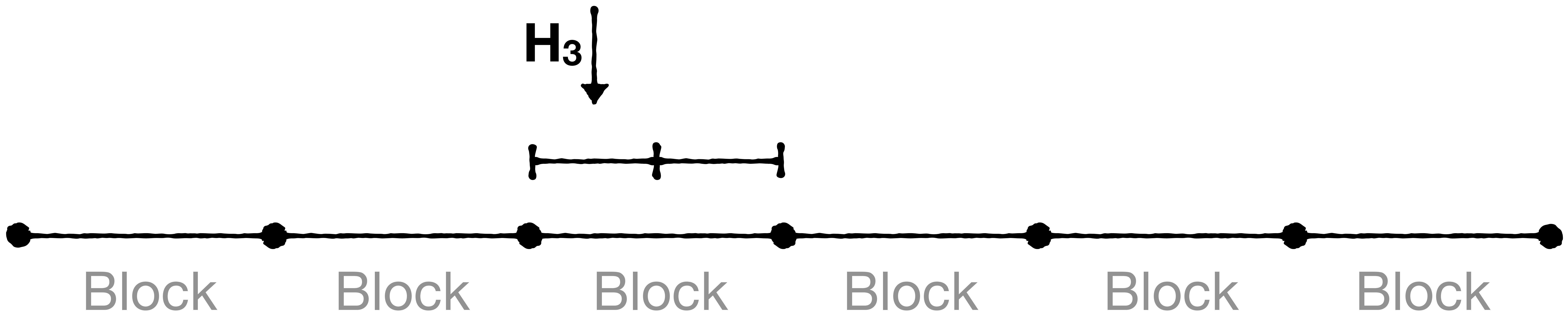
e.g., 4 hash functions, 2 words per block



e.g., 4 hash functions, 2 words per block

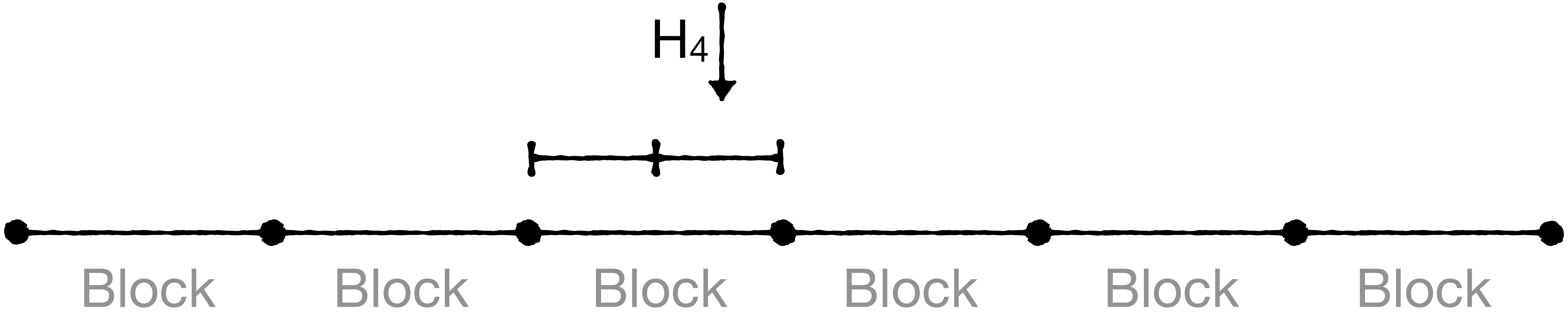


e.g., 4 hash functions, 2 words per block



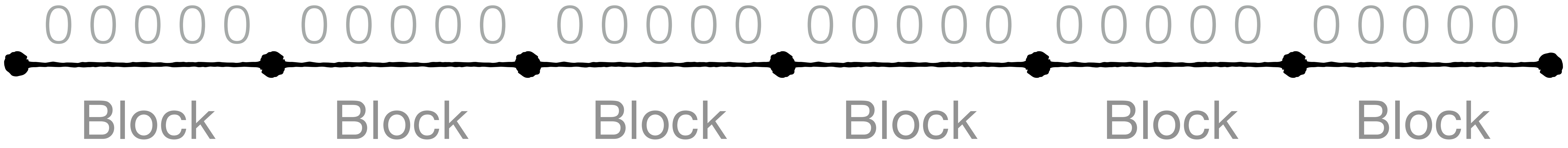
Solutions?

e.g., 4 hash functions, 2 words per block



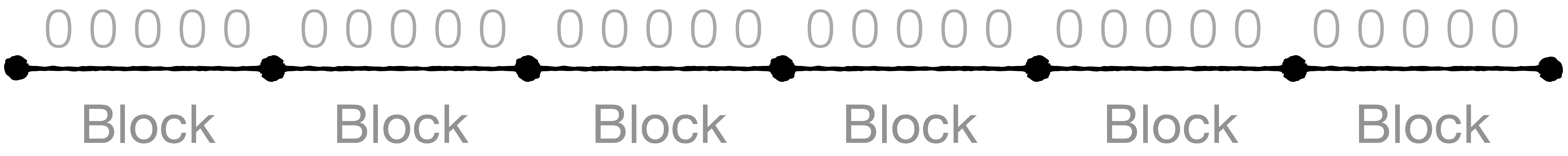
Split block Bloom filters. Jim Apple. Arxiv 2023.

Used in the Impala system as of 2016

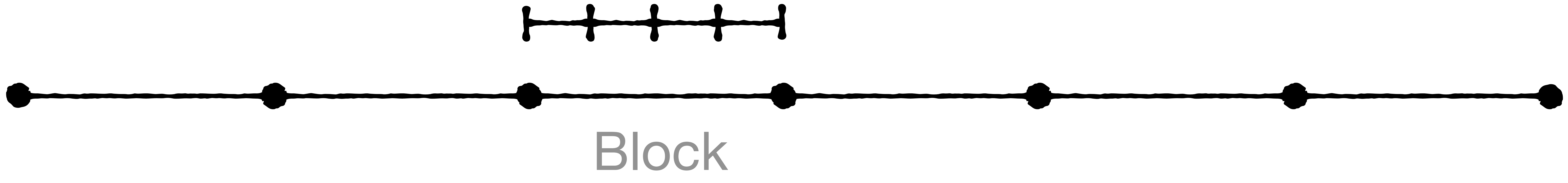


Split block Bloom filters. Jim Apple. Arxiv 2023.

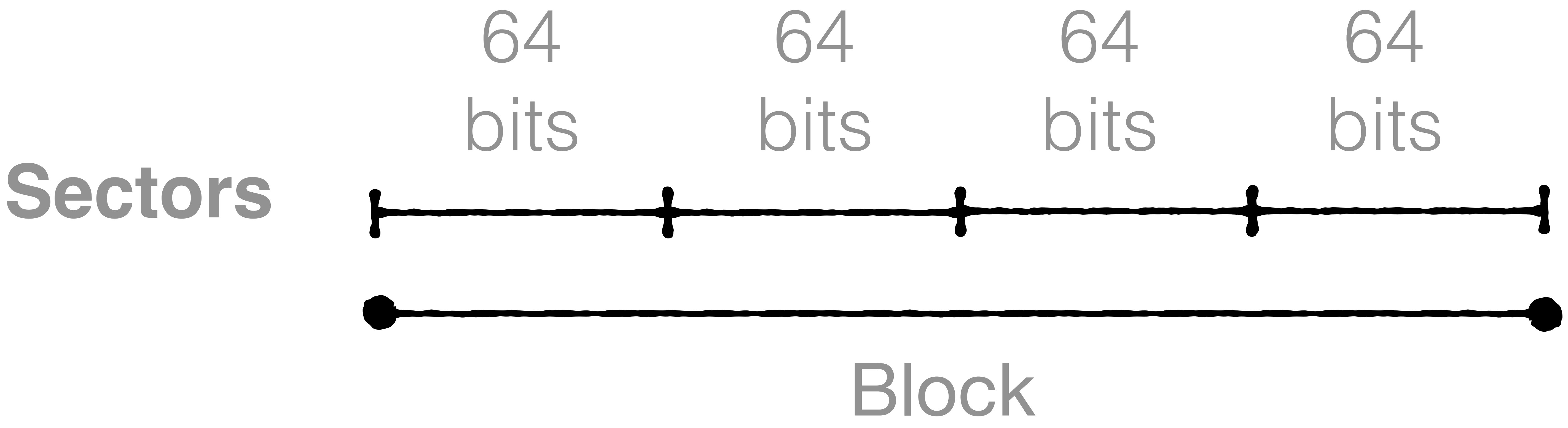
Performance-Optimal Filtering: Bloom Overtakes Cuckoo at High Throughput. Harald Lang, Thomas Neumann, Alfons Kemper, Peter Boncz. **VLDB 2019.**



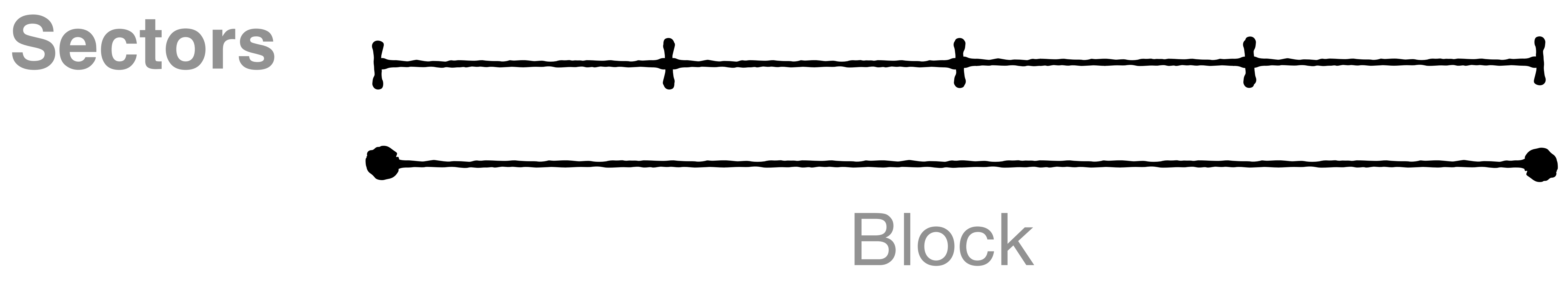
Partition into s sectors, each the size of a word (e.g., 64 bits)



Example: s = 4 sectors



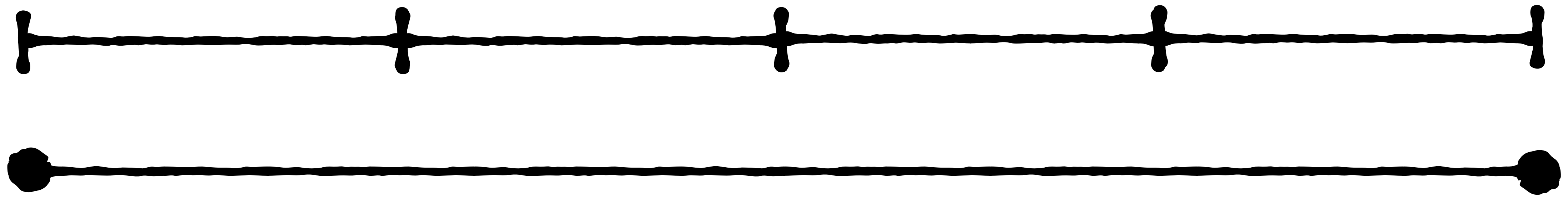
Insertion: hashes to k / s bits per sector sequentially



Insertion: hashes to k / s bits per sector sequentially

e.g., s=4 sectors and k=8 hashes

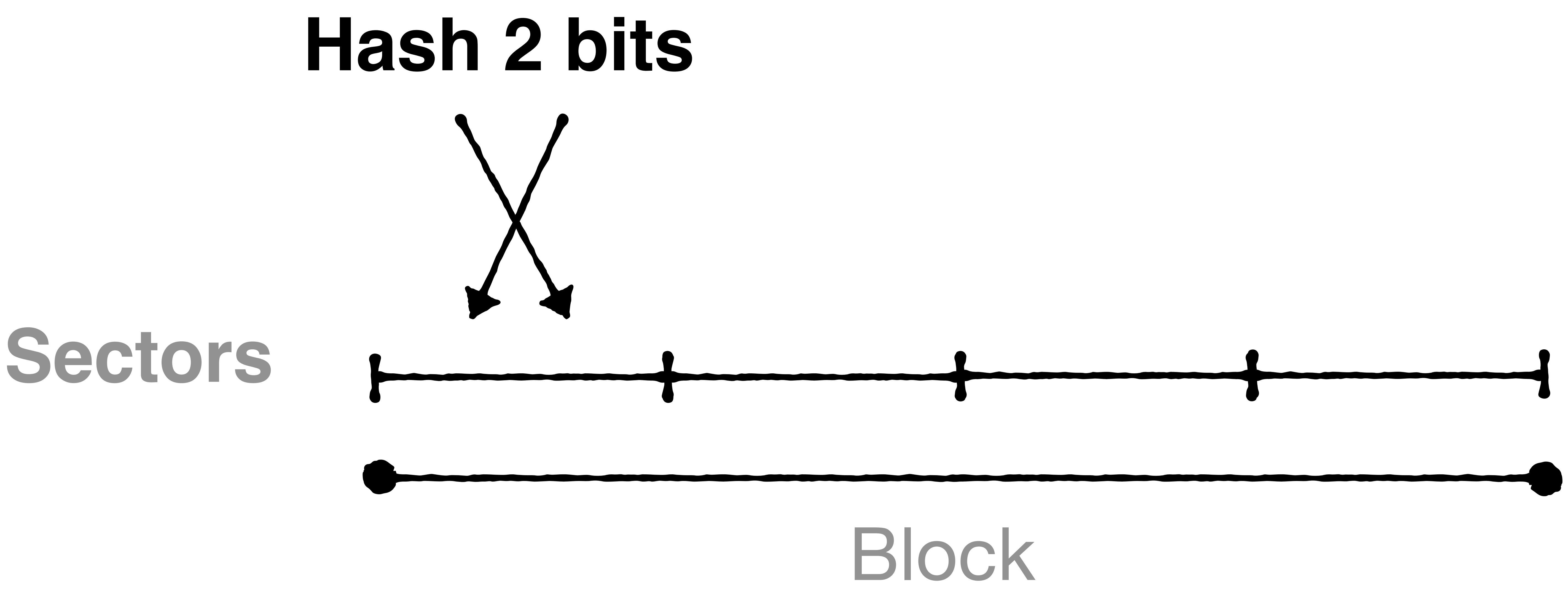
Sectors



Block

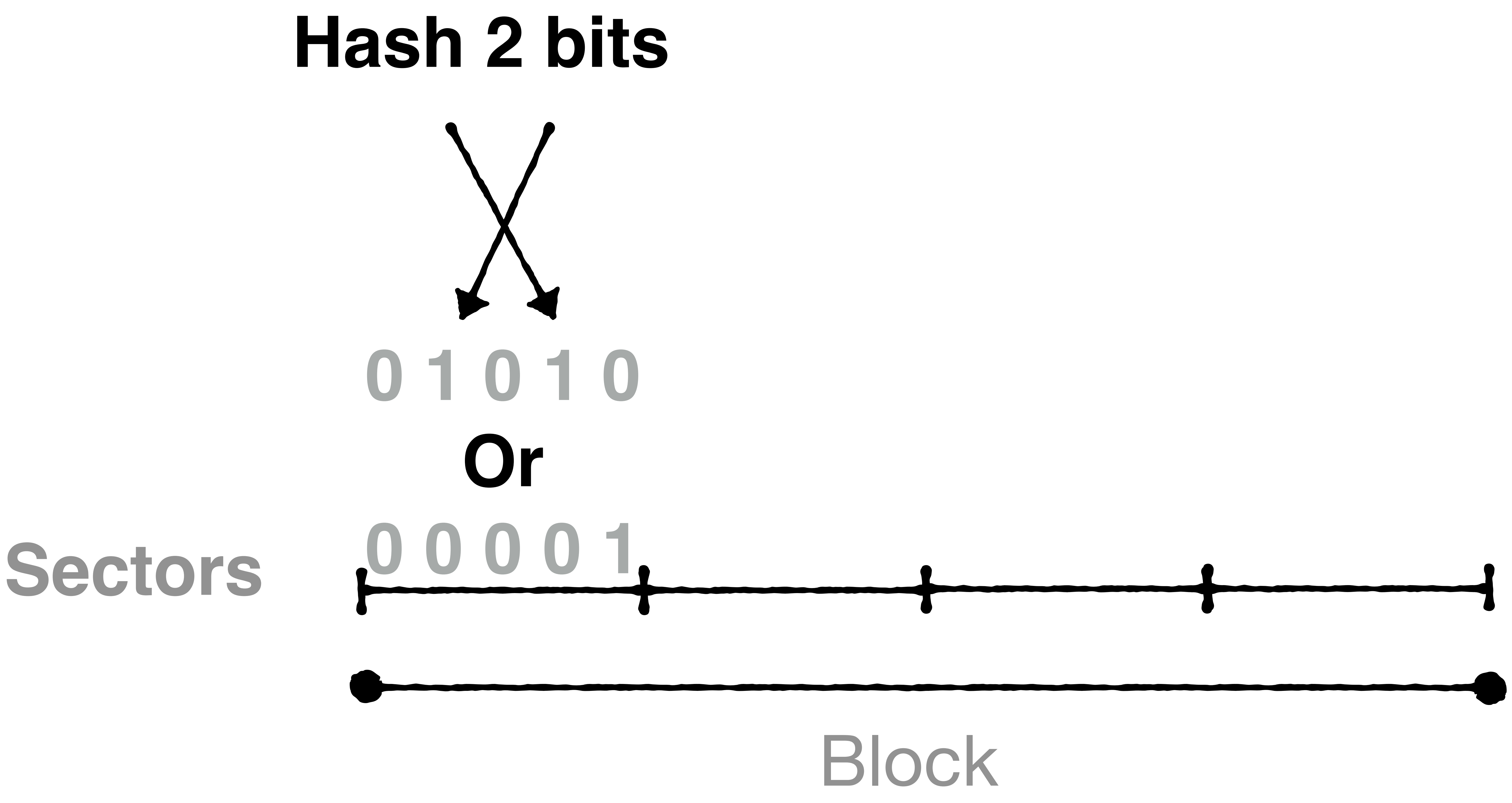
Insertion: hashes to k / s bits per sector sequentially

e.g., s=4 sectors and k=8 hashes



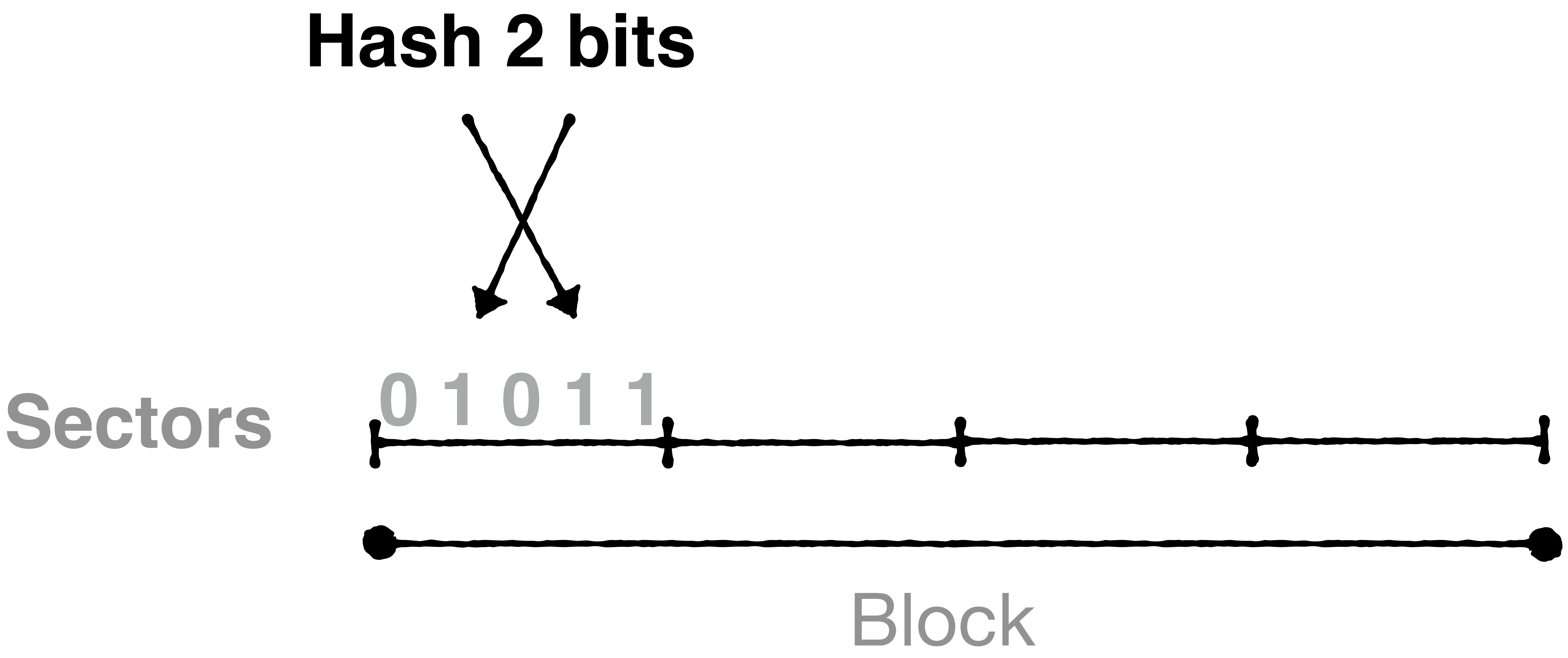
Insertion: hashes to k / s bits per sector sequentially

e.g., s=4 sectors and k=8 hashes



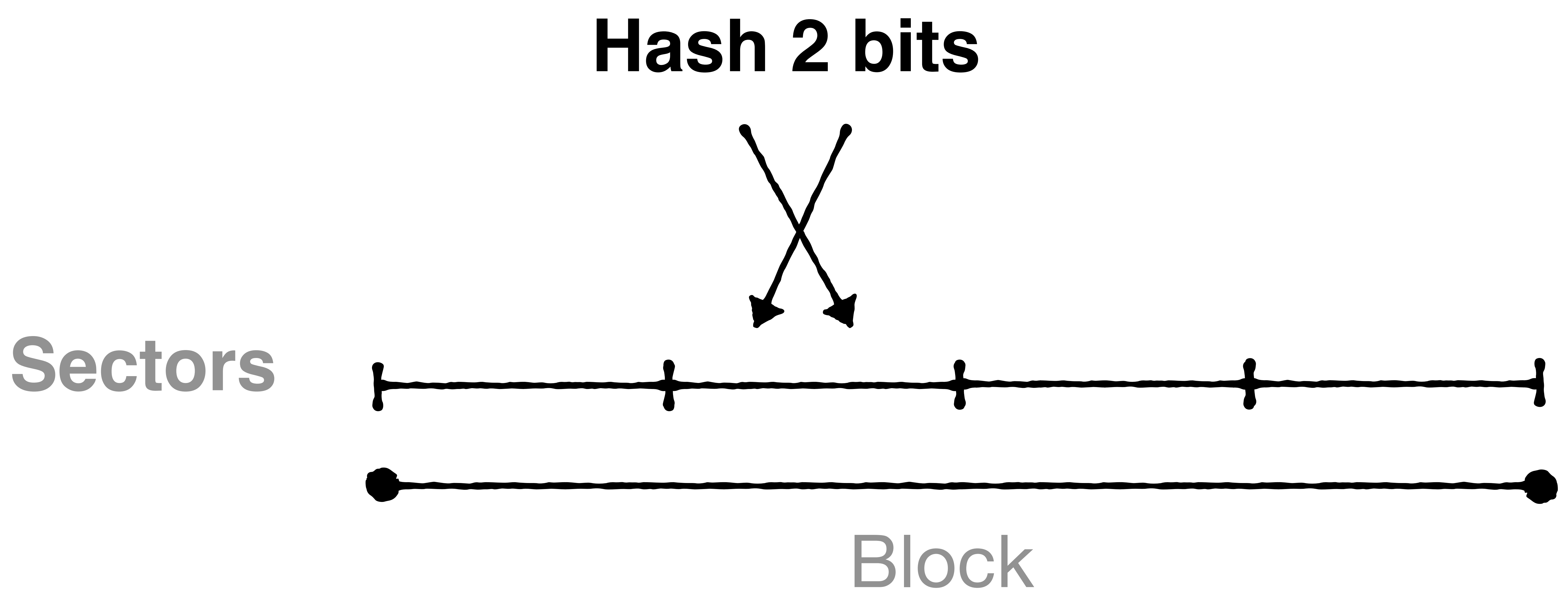
Insertion: hashes to k / s bits per sector sequentially

e.g., s=4 sectors and k=8 hashes



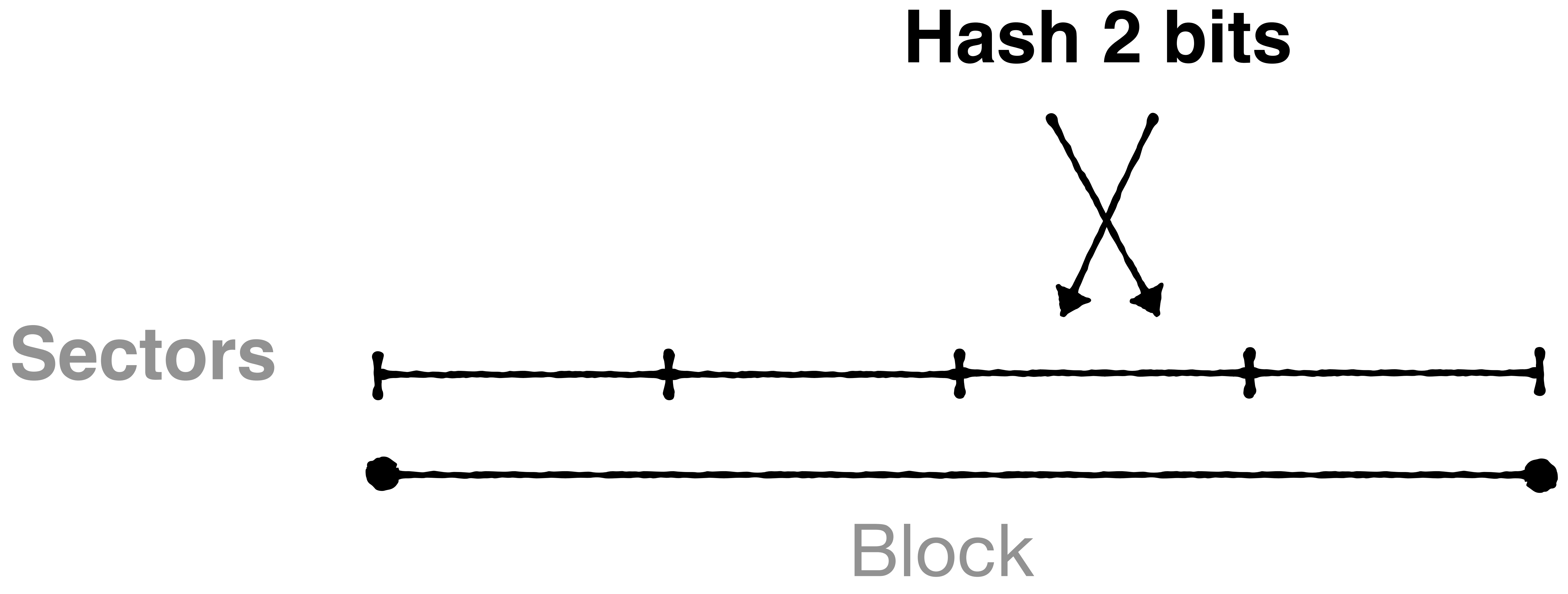
Insertion: hashes to k / s bits per sector sequentially

e.g., s=4 sectors and k=8 hashes



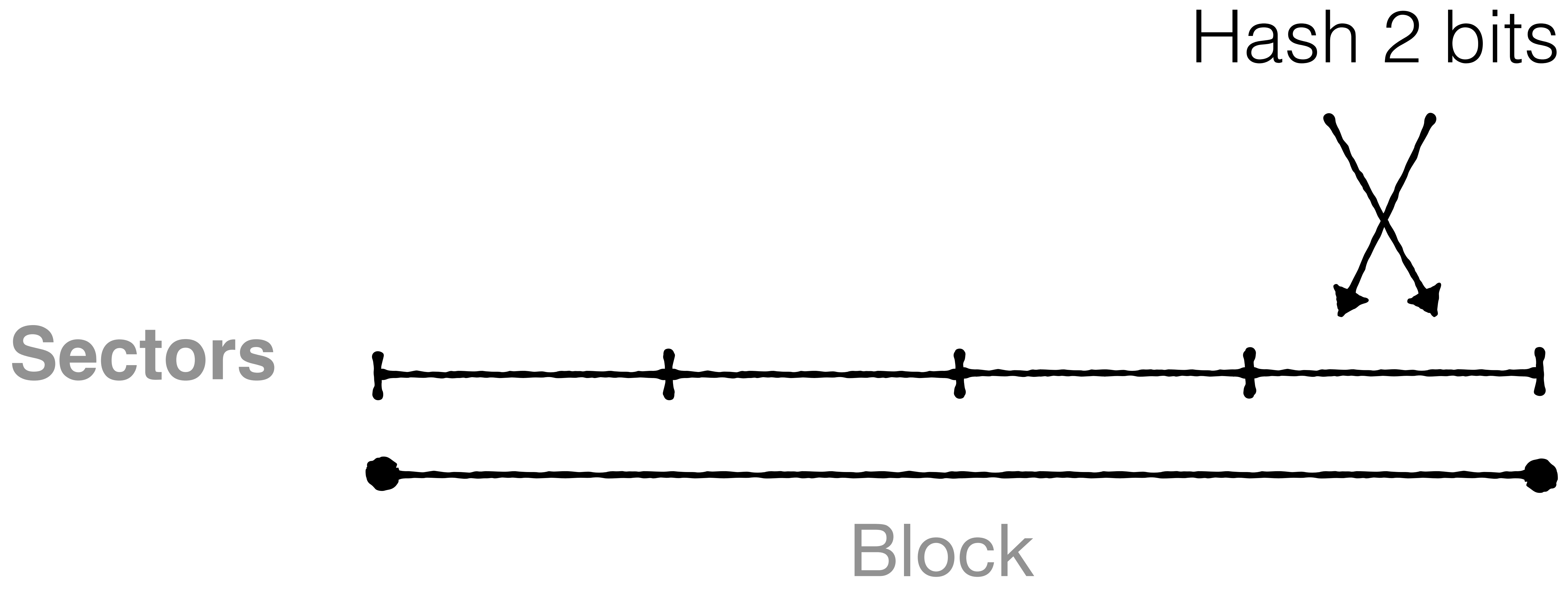
Insertion: hashes to k / s bits per sector sequentially

e.g., s=4 sectors and k=8 hashes

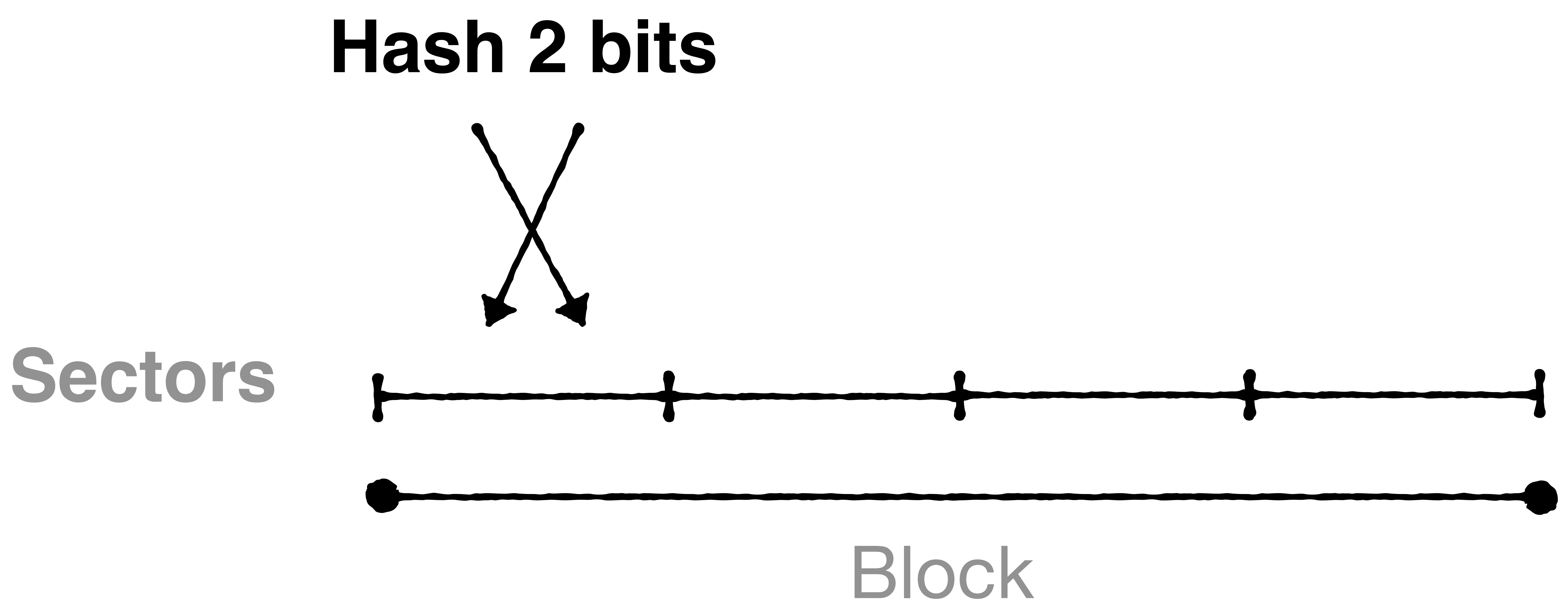


Insertion: hashes to k / s bits per sector sequentially

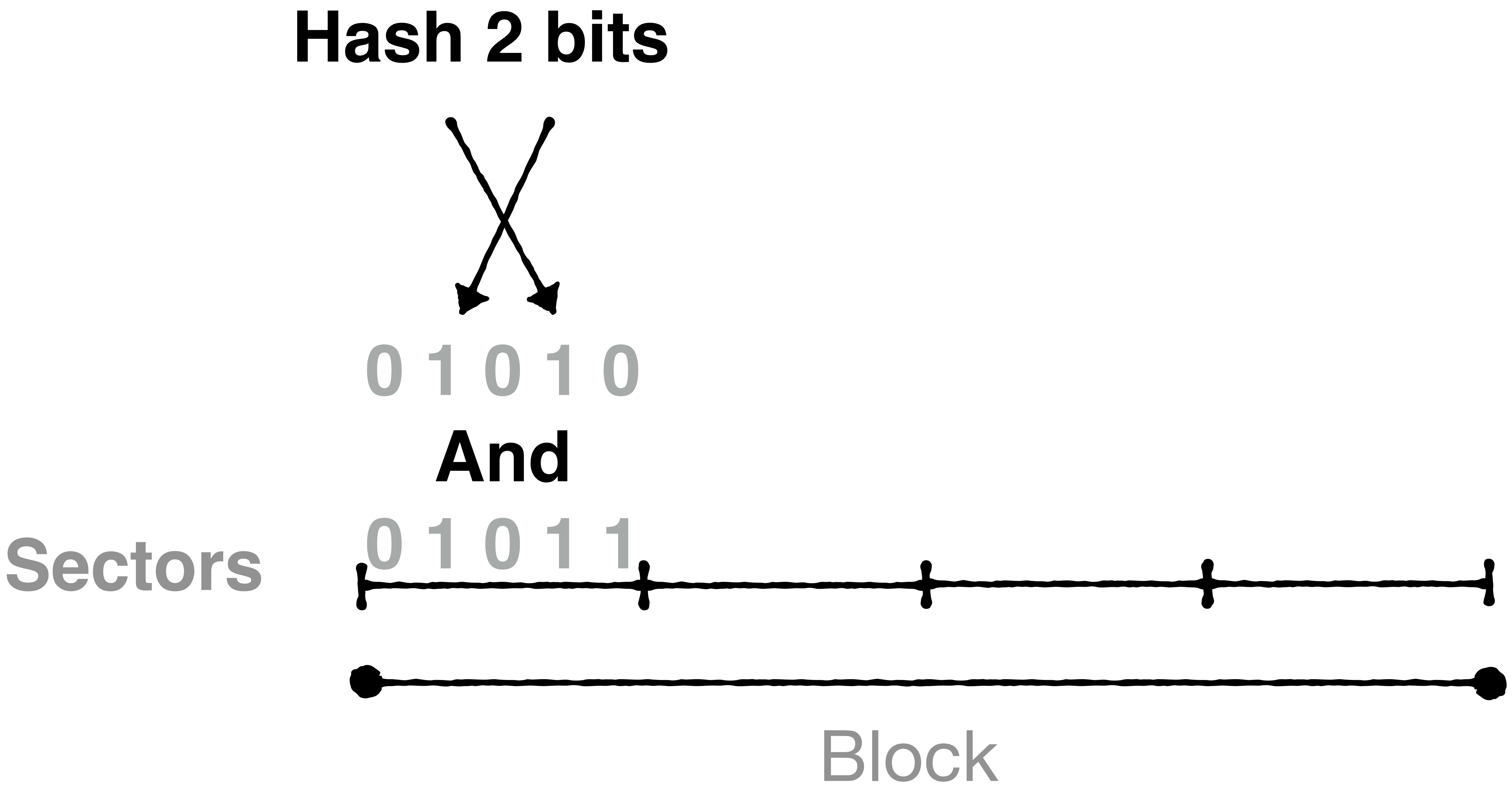
Read/written 4 rather than 8 registers



Query: check k / s hashes per sector sequentially



Query: check k / s hashes per sector sequentially

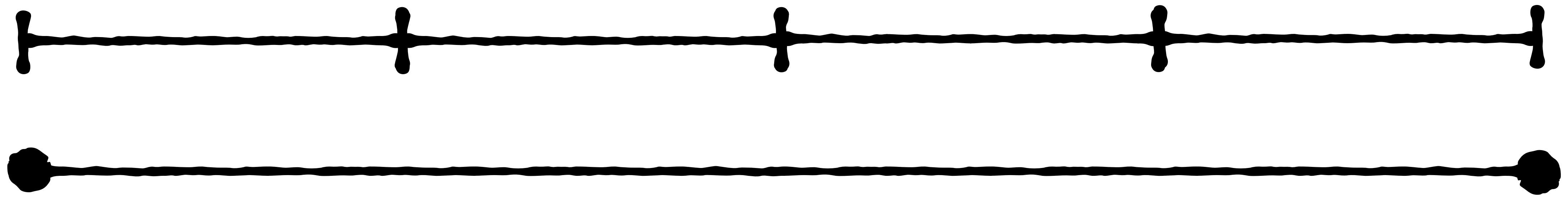


Query: check k / s hashes per sector sequentially

Hash 2 bits

(0 1 0 1 0 **And** 0 1 0 1 1 **) ==** 0 1 0 1 0

Sectors



Block

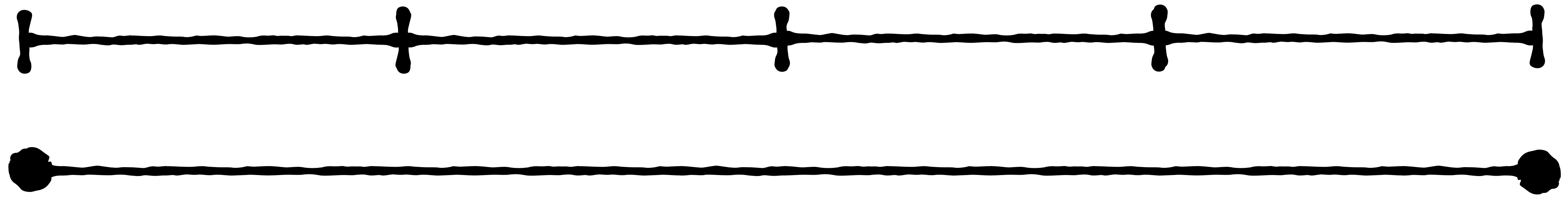
Query: check k / s hashes per sector sequentially

Hash 2 bits

if (0 1 0 1 0 And 0 1 0 1 1) != 0 1 0 1 0

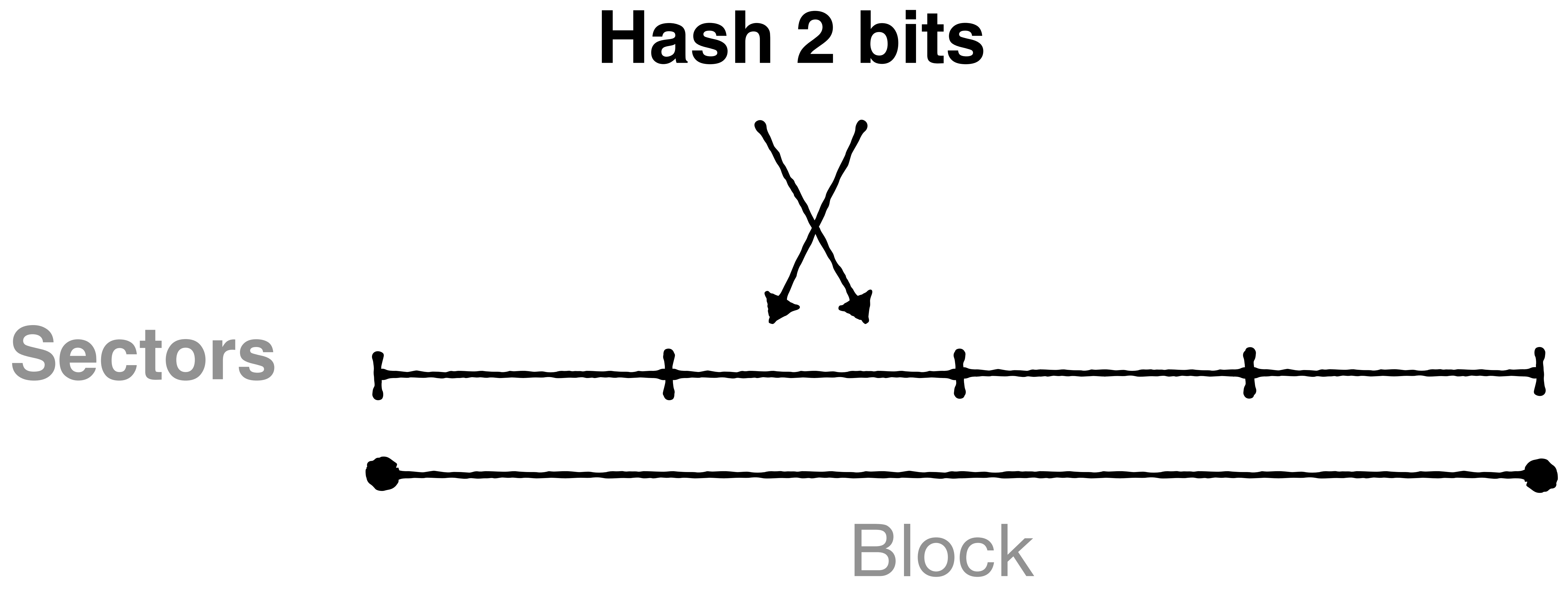
Return false

Sectors

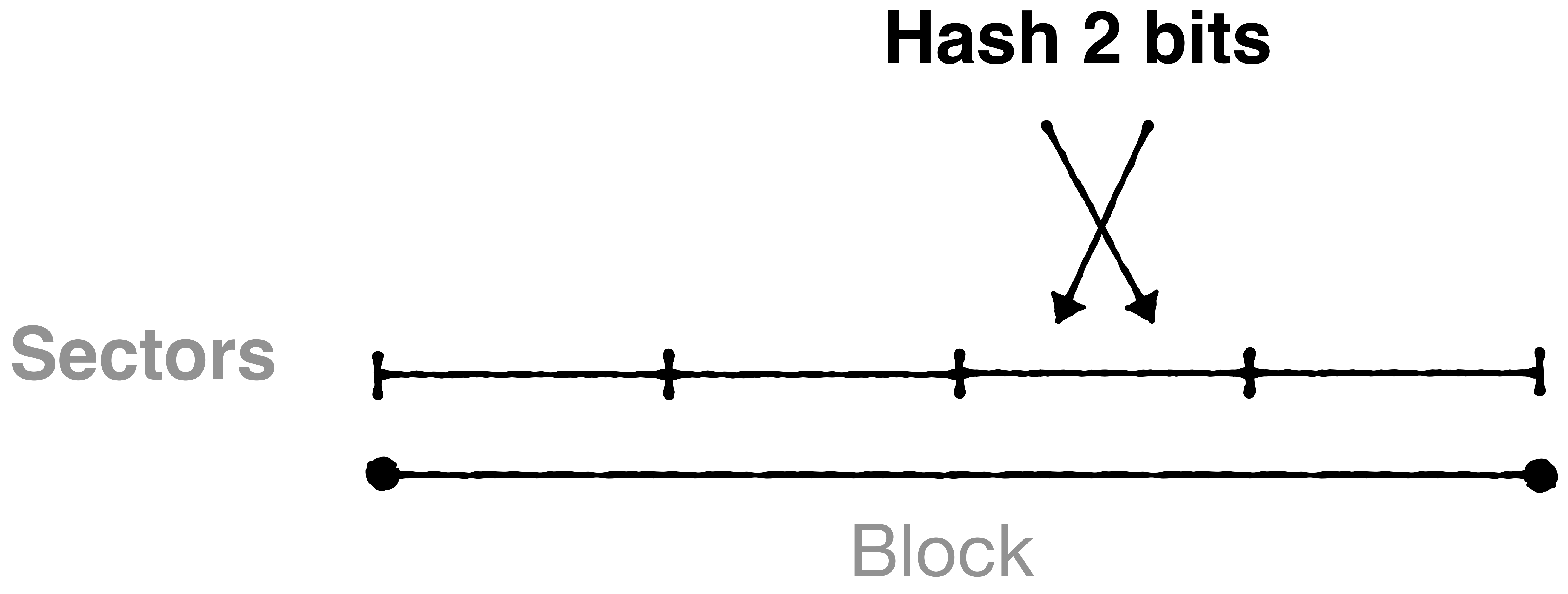


Block

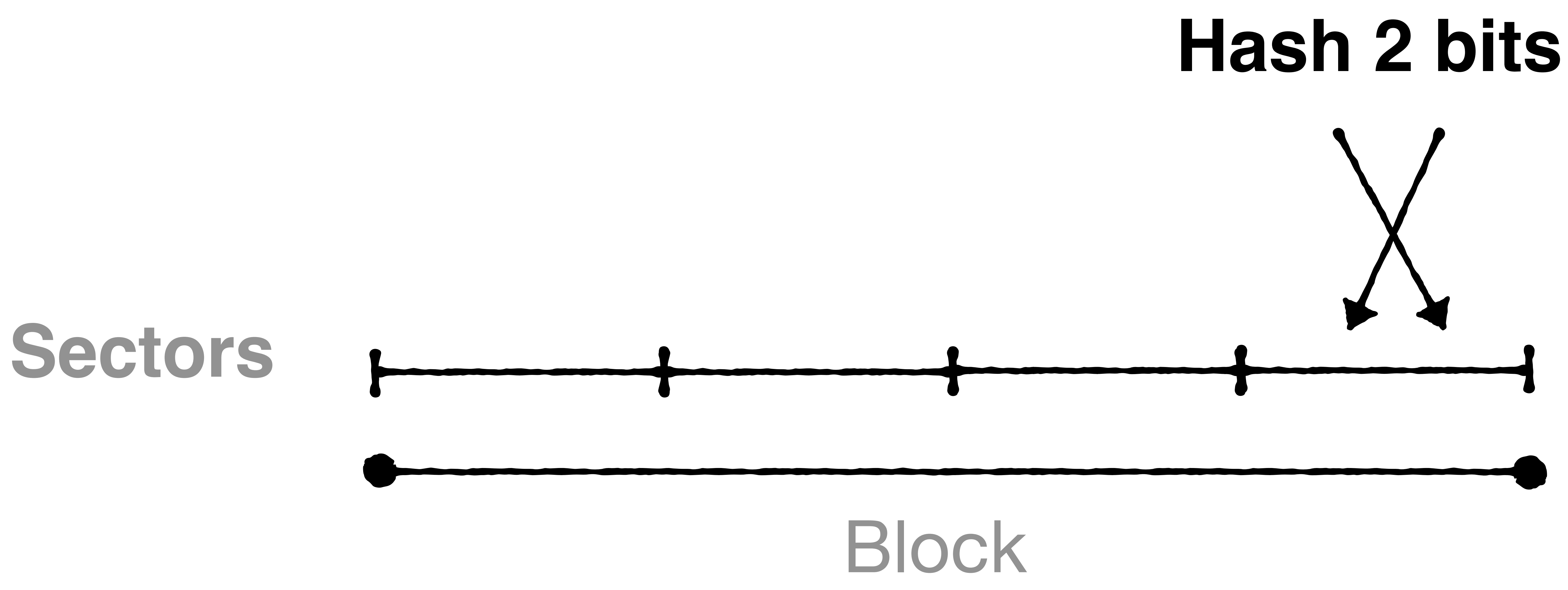
Query: check k / s hashes per sector sequentially



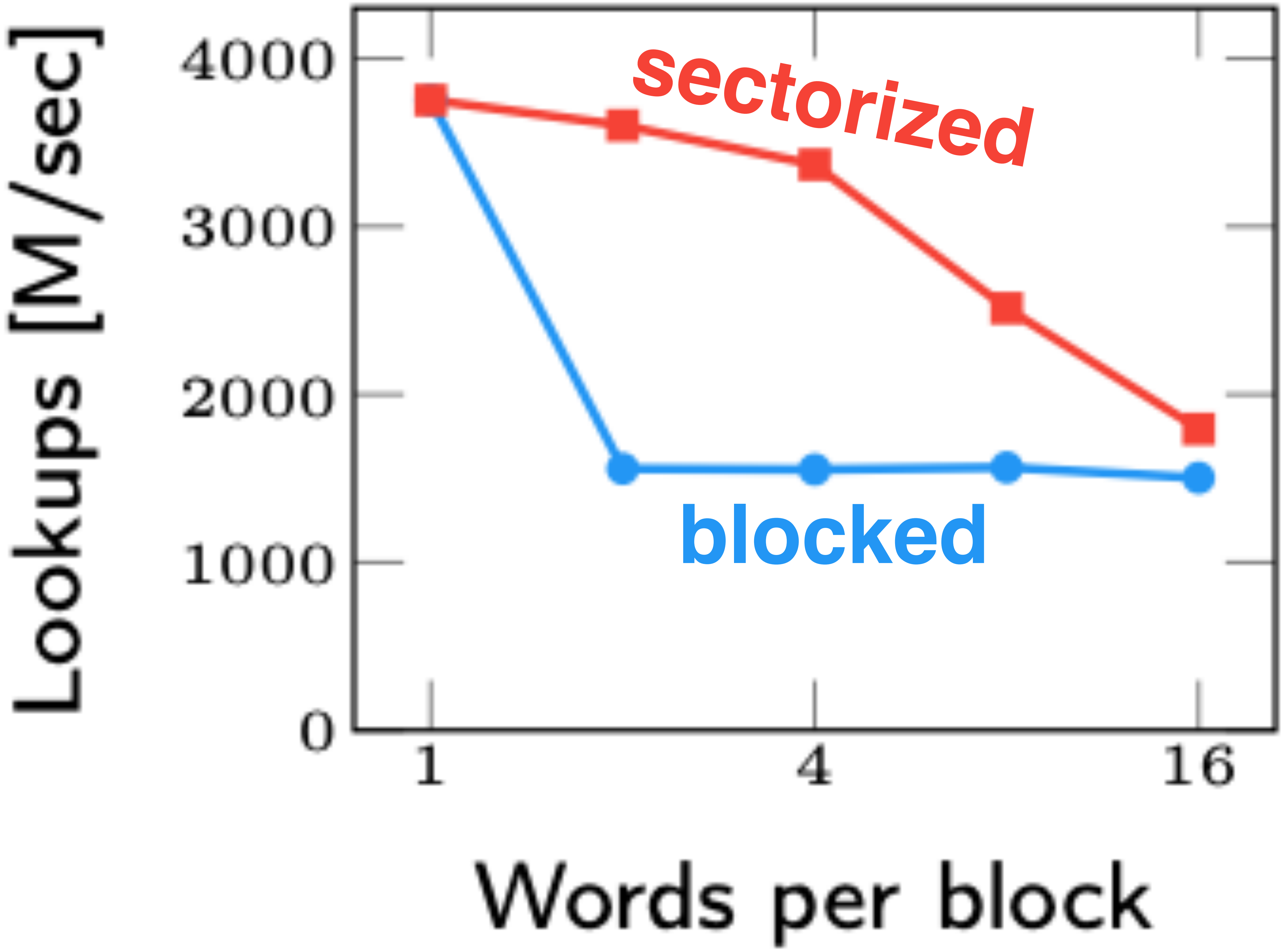
Query: check k / s hashes per sector sequentially



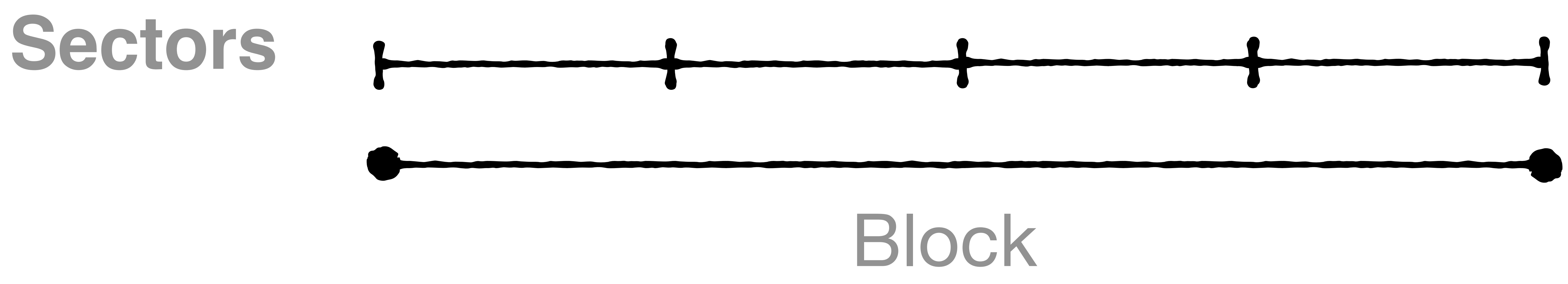
Query: check k / s hashes per sector sequentially



Query: check k / s hashes per sector sequentially

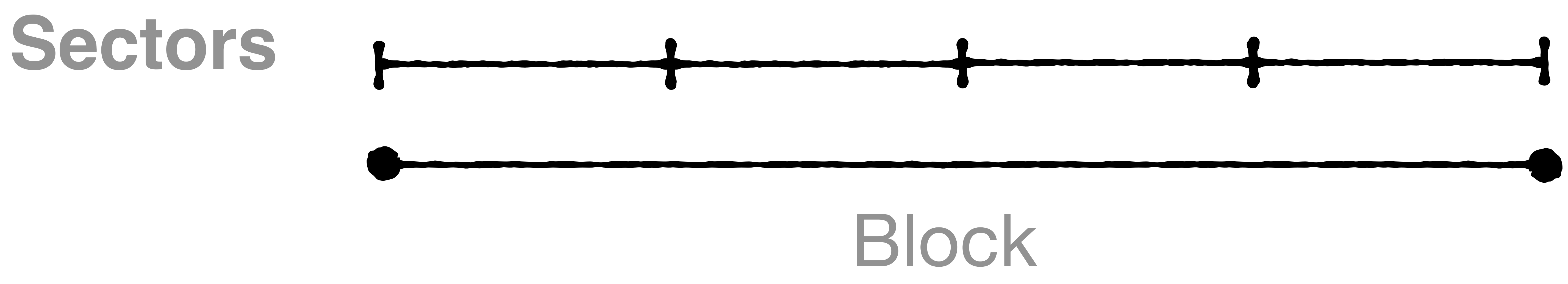


Further ways to optimize?



Further ways to optimize?

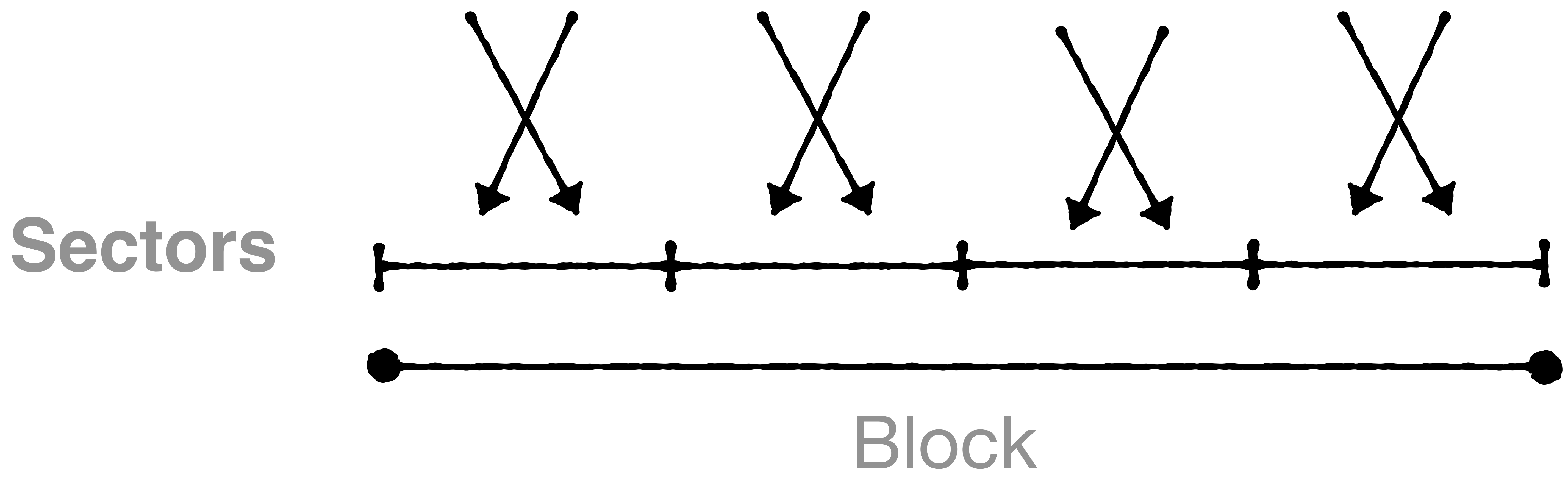
AVX - SIMD



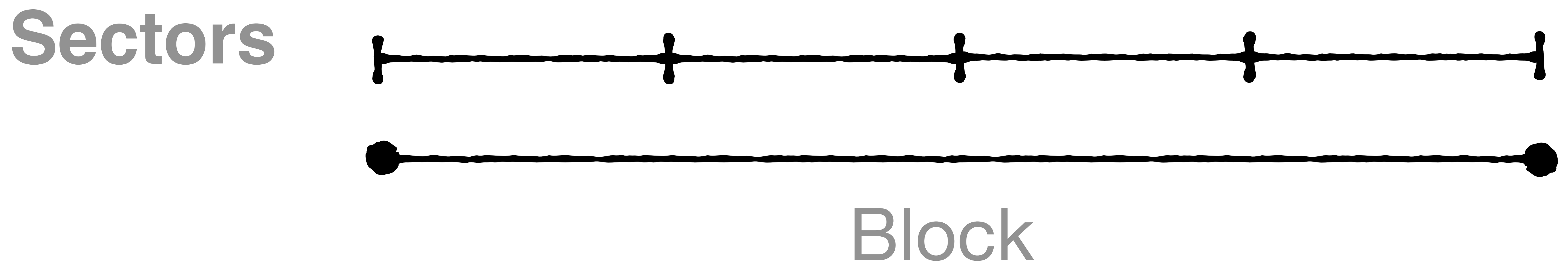
Further ways to optimize?

AVX - SIMD

Operate on each sector in parallel

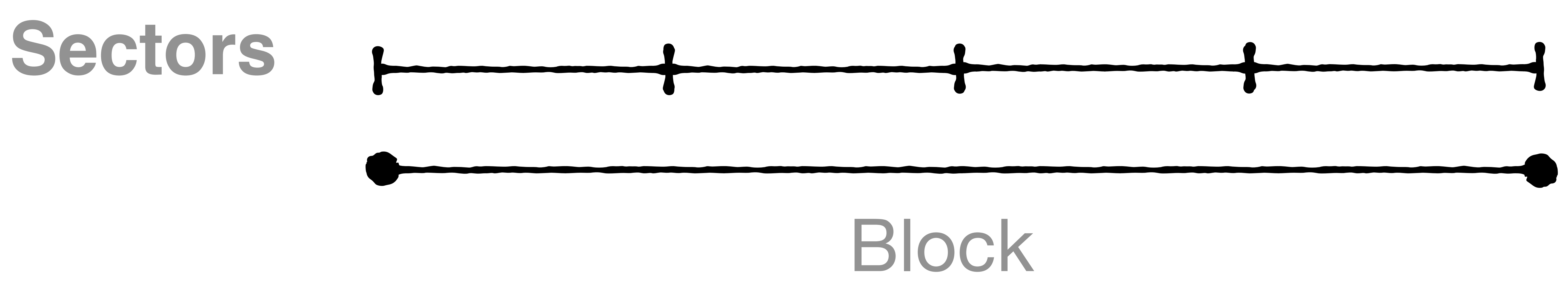


Problems with Sectorized Bloom filters?



Problems with Sectorized Bloom filters?

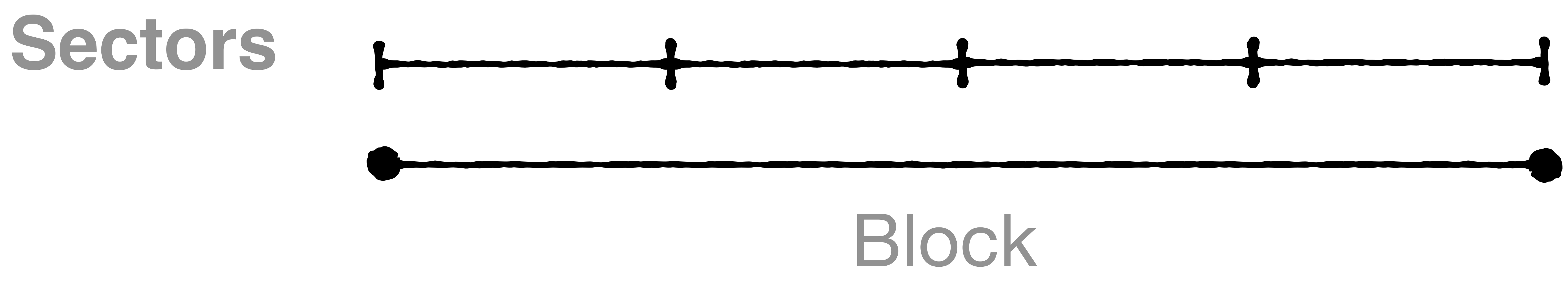
 # hashes must be multiple of # sectors



Problems with Sectorized Bloom filters?

hashes must be multiple of # sectors

Why is this bad?

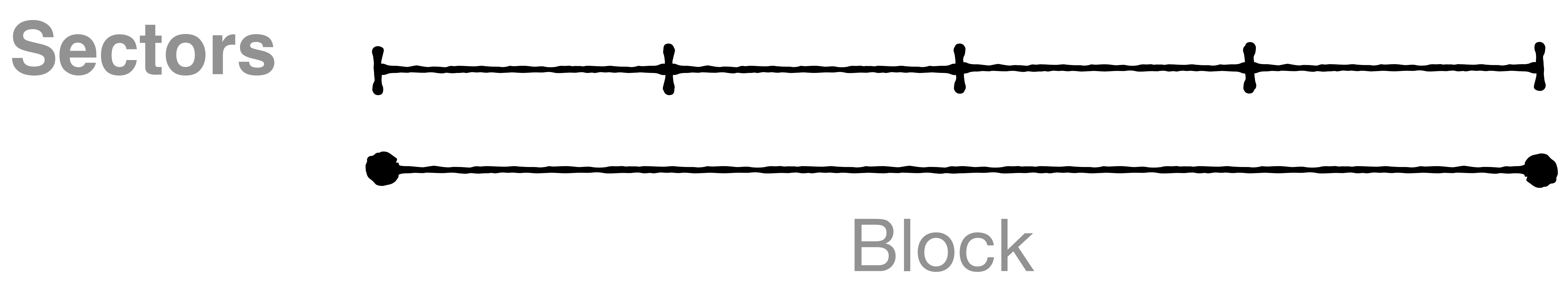


Problems with Sectorized Bloom filters?

hashes must be multiple of # sectors

Why is this bad?

Force using sub-optimal # hash functions



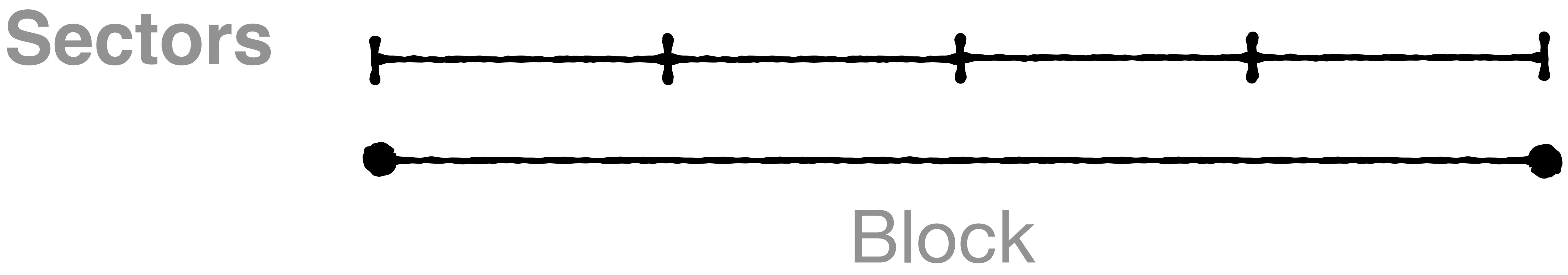
Problems with Sectorized Bloom filters?

hashes must be multiple of # sectors

Why is this bad?

Force using sub-optimal # hash functions

Harm FPR



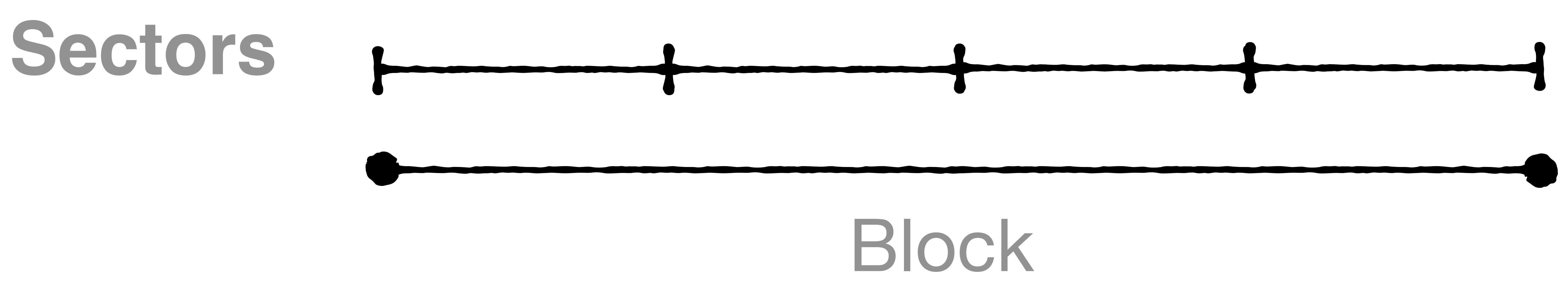
Problems with Sectorized Bloom filters?

hashes must be multiple of # sectors

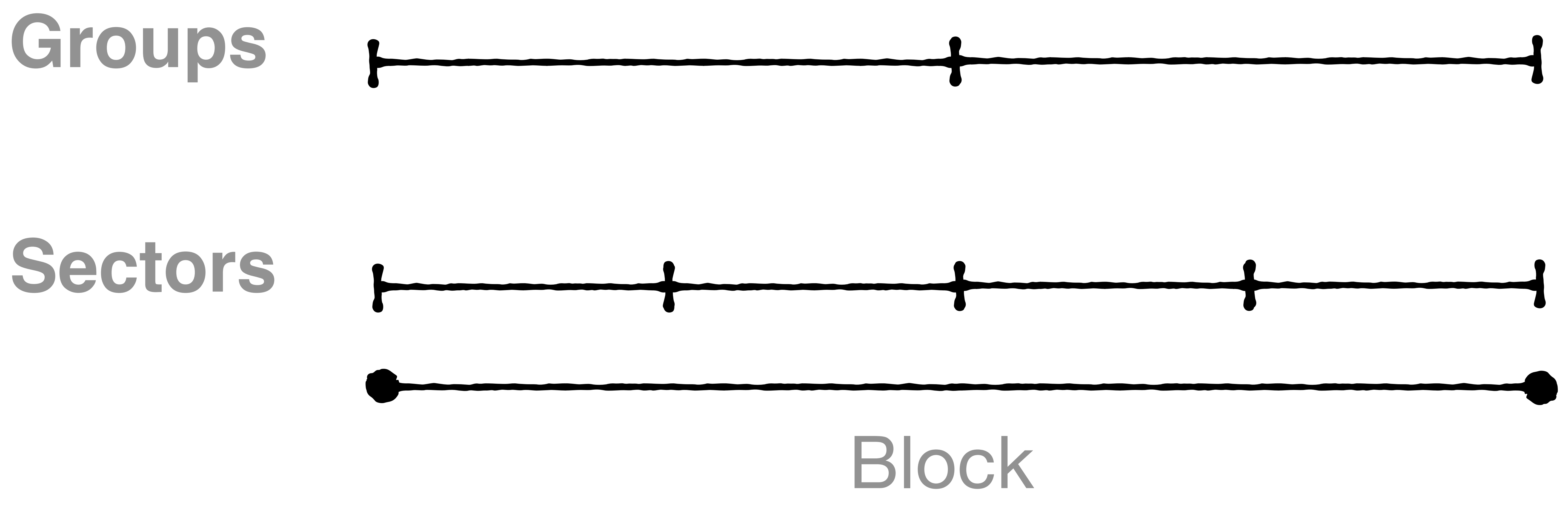
Why is this bad?

Force using sub-optimal # hash functions

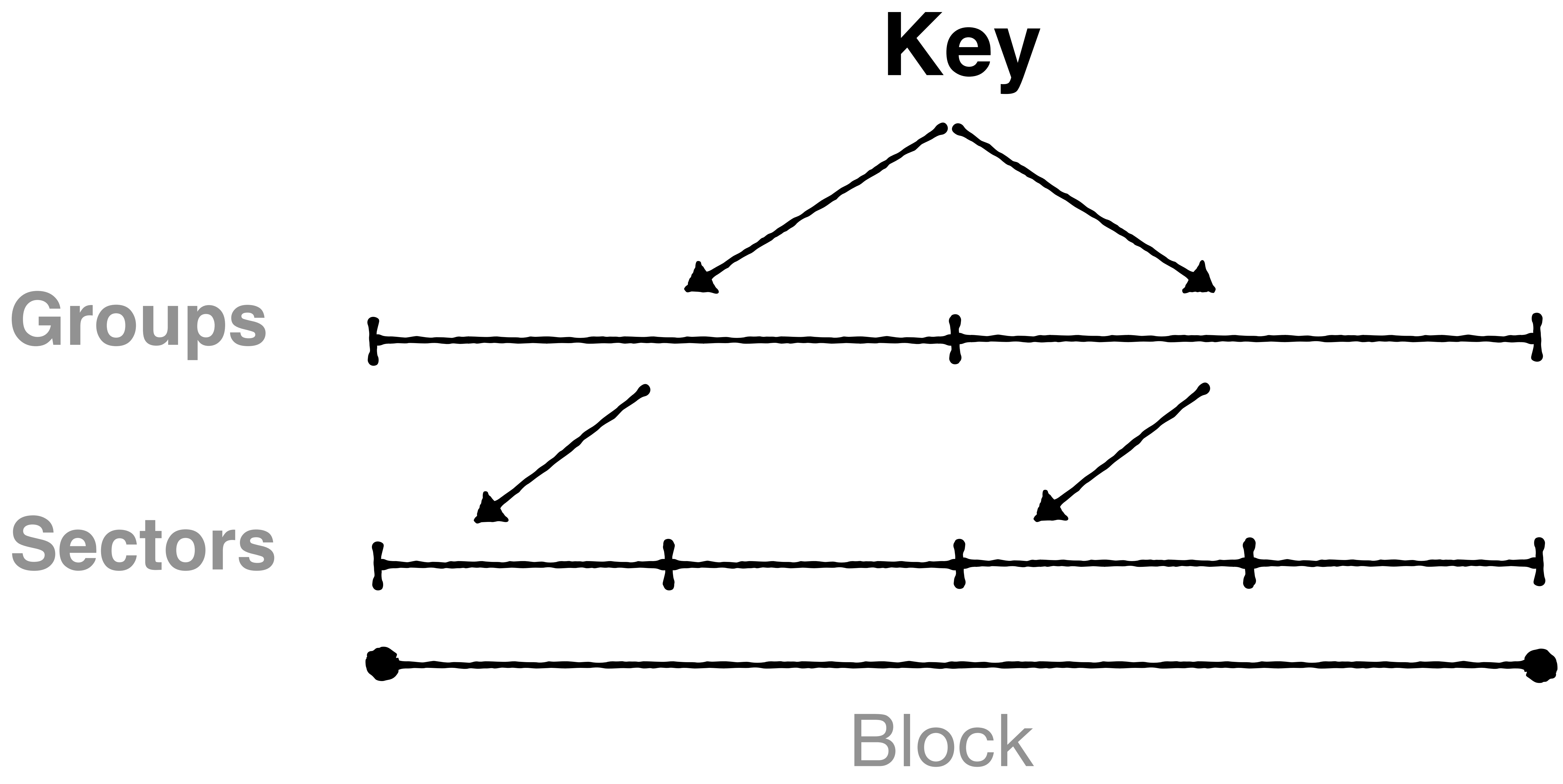
Harm FPR - solutions?



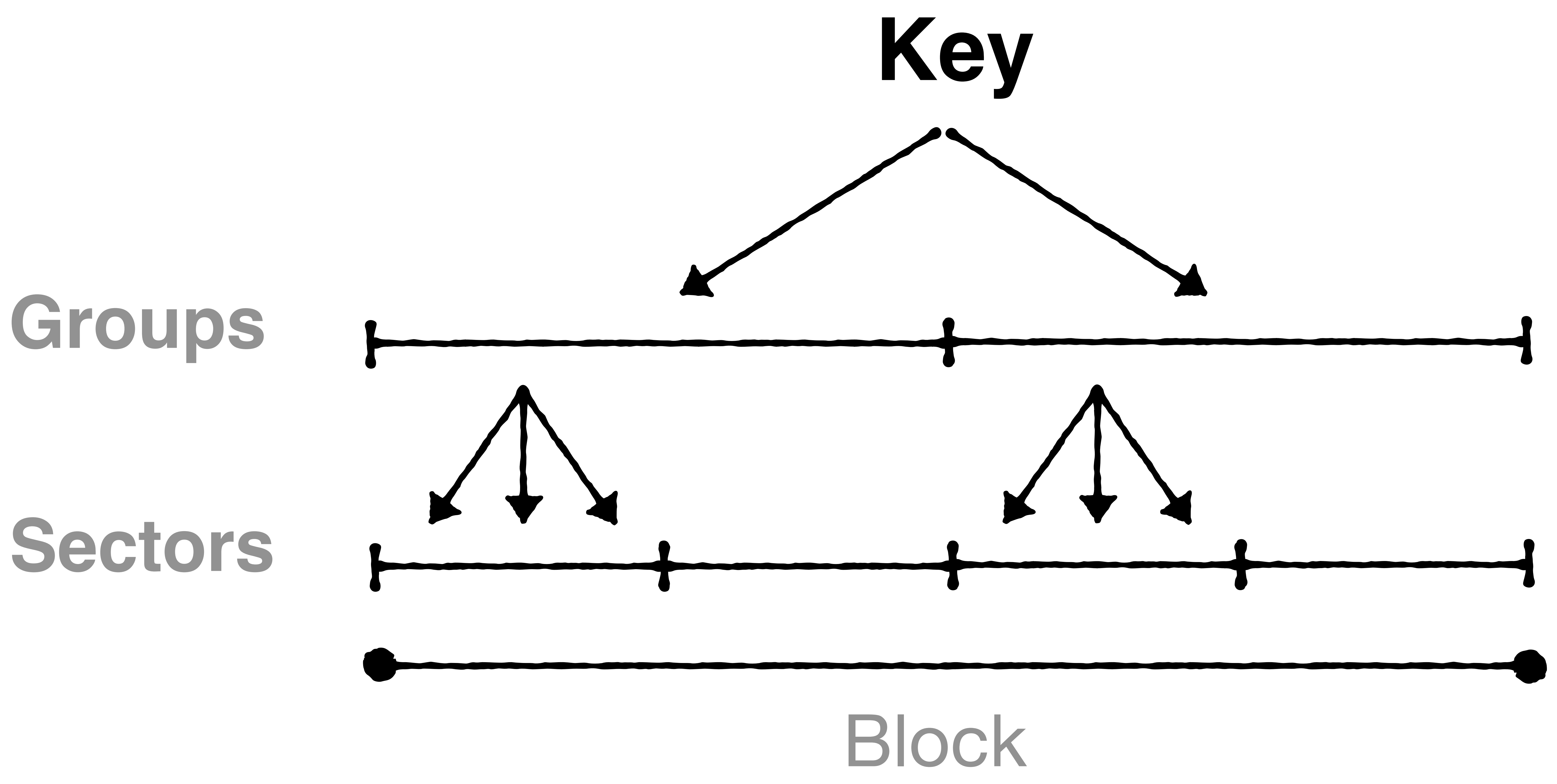
Divide sectors into Z groups



Map each key to one sector per group based on its hash

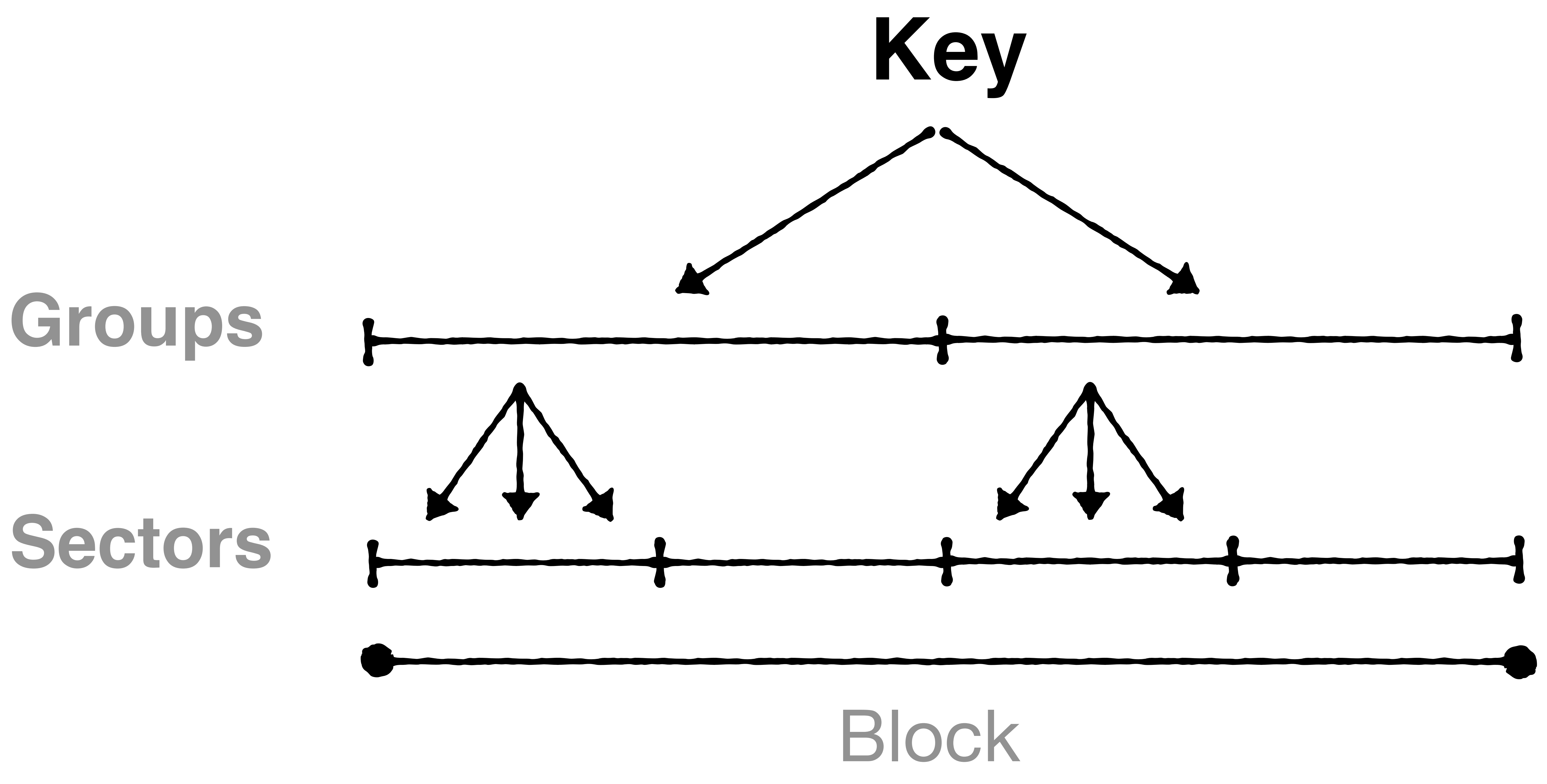


Hash bits only to relevant sector in each group

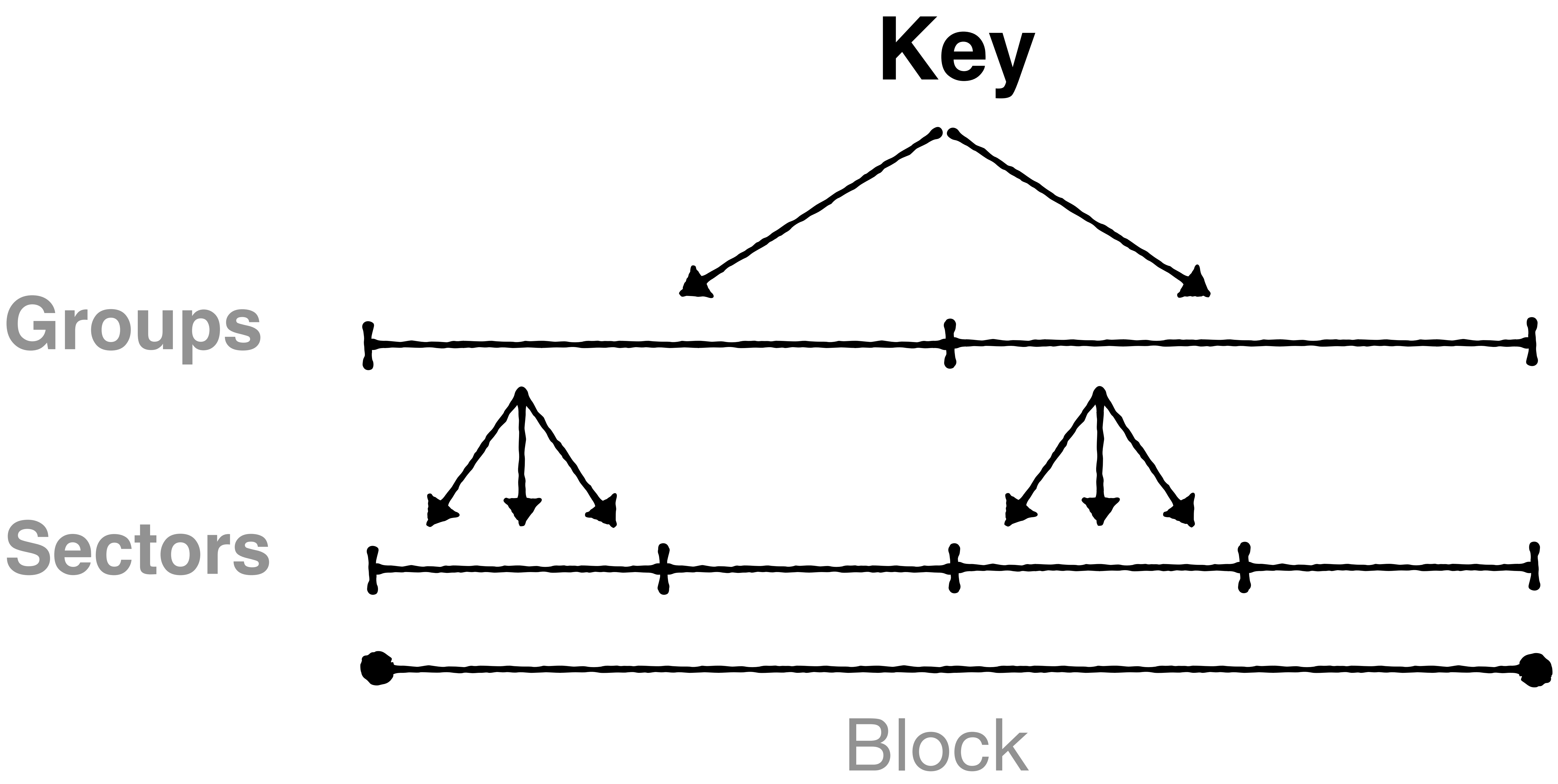


Hash bits only to relevant sector in each group

e.g., here we can use 6 hashes :)



Hash bits only to relevant sector in each group



Nearly best of both worlds :) faster & low FPR

Break

Bloom

$$\boldsymbol{FPR\ \varepsilon \approx 2^{-M/N \cdot \ln(2)}}$$

Bloom

Lower Bound

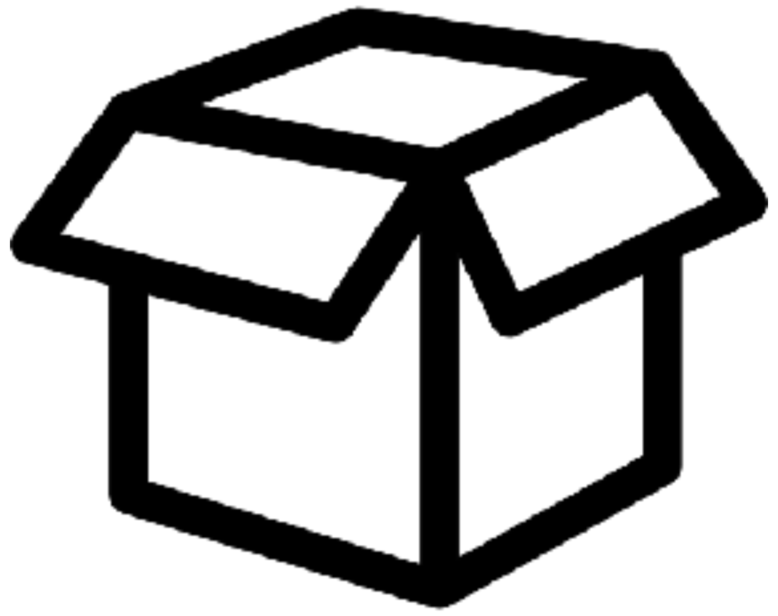
$FPR\ \varepsilon \approx 2^{-M/N \cdot \ln(2)}$

???

Lower Bound for Filter Memory

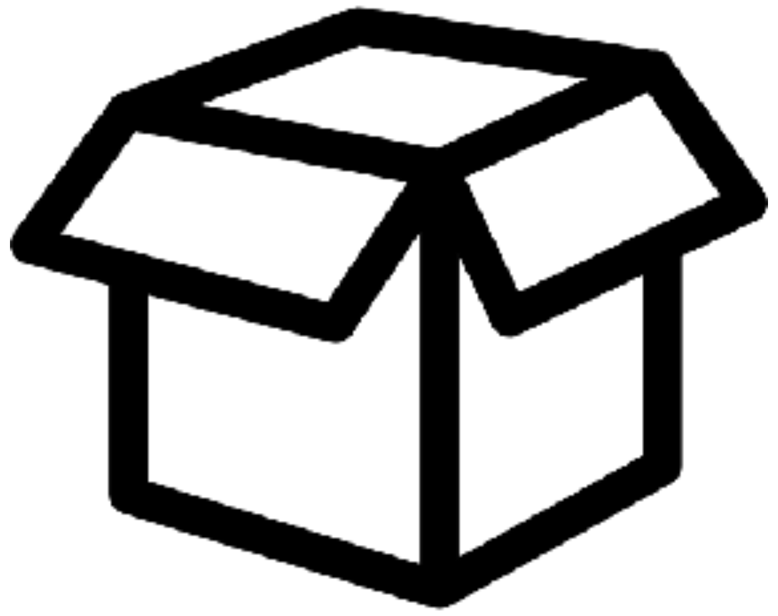
Lower Bound for Filter Memory

Assume nothing about implementation



Lower Bound for Filter Memory

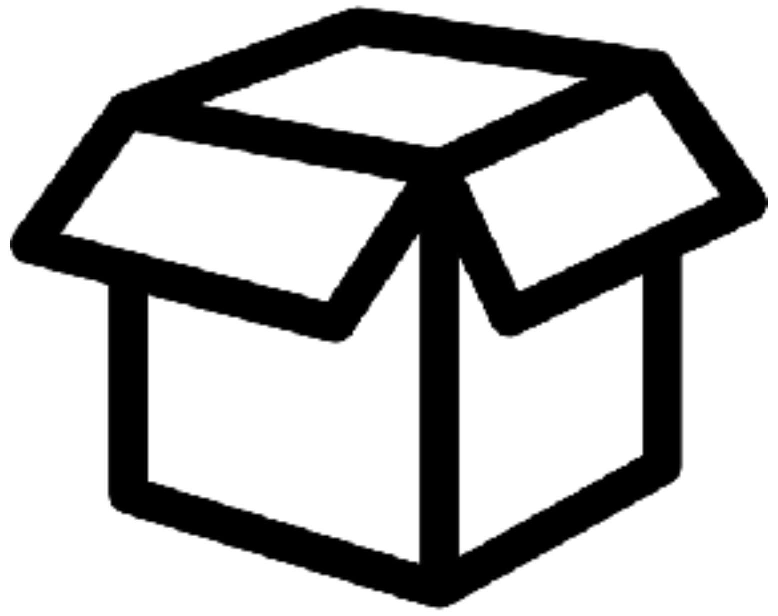
Assume nothing
about implementation



**Analyze with respect to
filter specification**

Lower Bound for Filter Memory

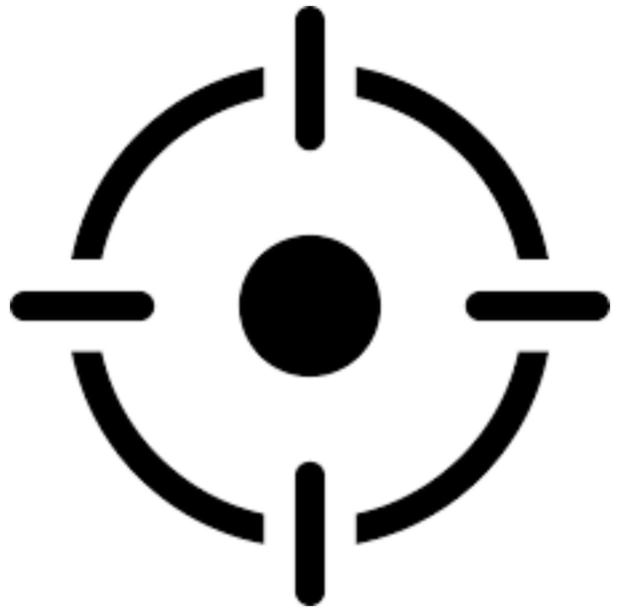
Assume nothing
about implementation



Analyze with respect to
filter specification

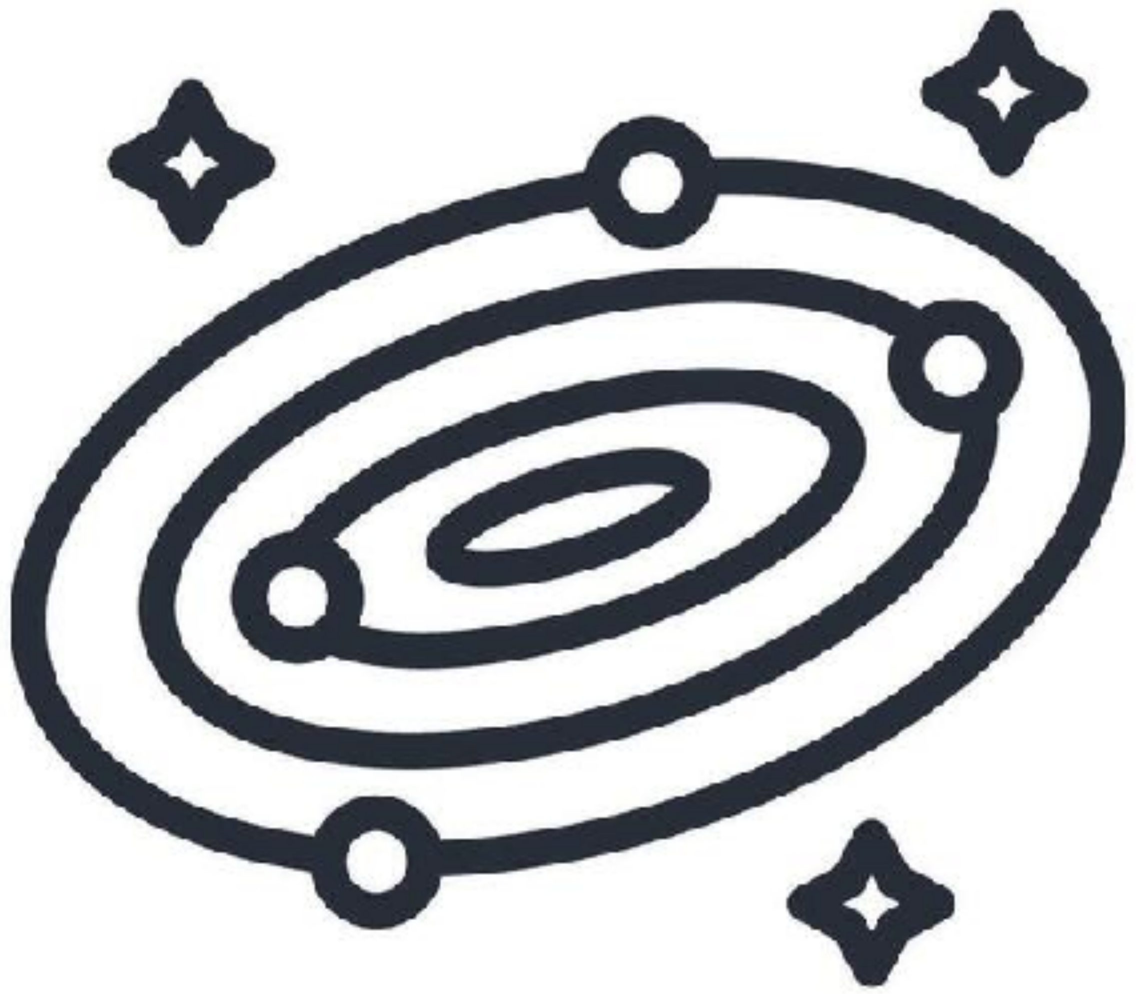
ε - FPR
N - # entries
U - Universe size

Lower Bound for **Exact Set**



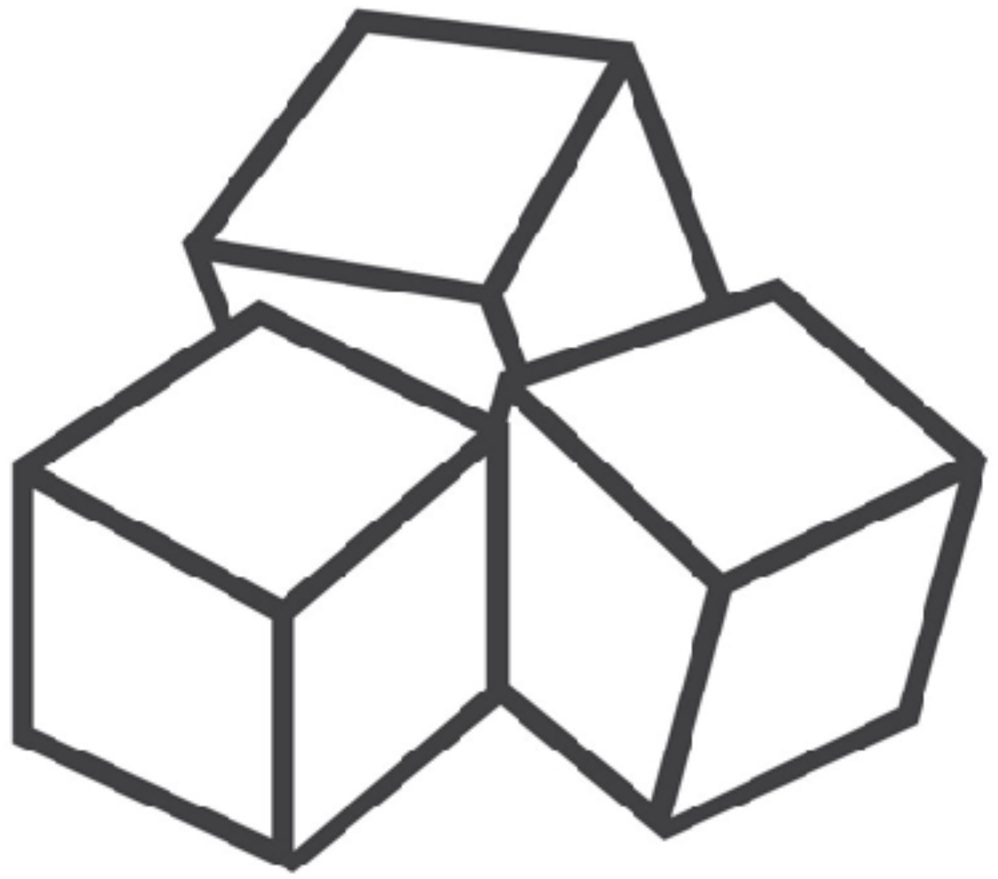
Lower Bound for Exact Set

Out of Universe U, store N entries



Lower Bound for Exact Set

(U choose N) combinations



Lower Bound for Exact Set

$\log_2$(U choose N) bits

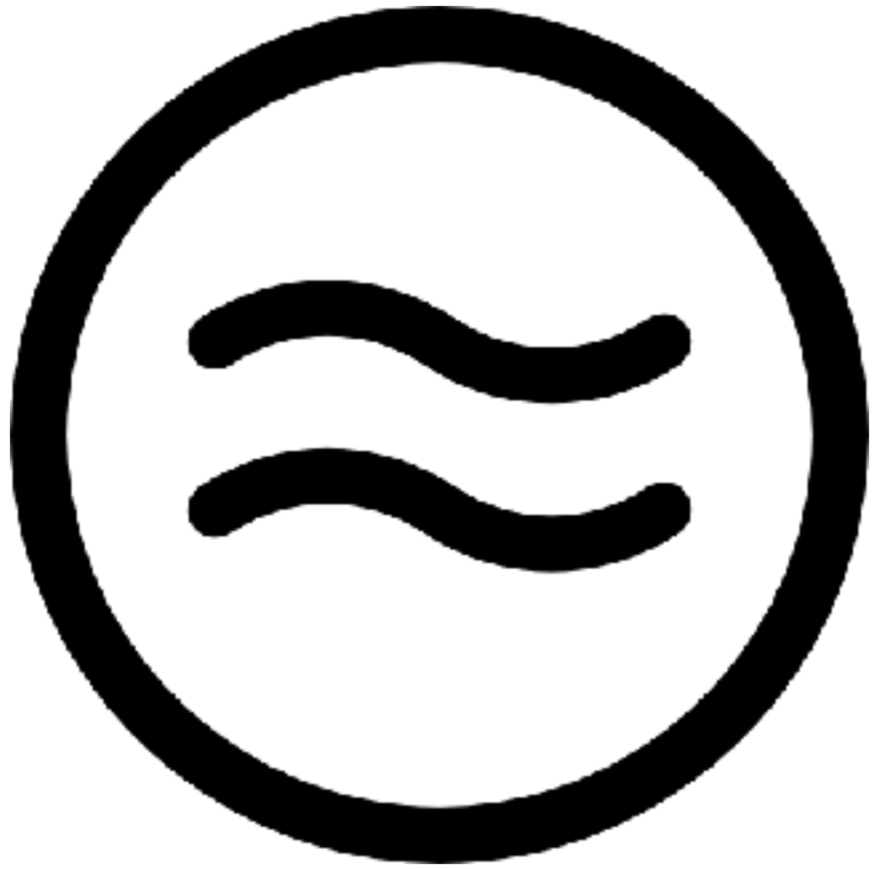
to encode a unique combination

Lower Bound for Exact Set

$\log_2(U \text{ choose } N) \approx \mathbf{N \cdot \log_2(U / N)}$ bits

for large U

Lower Bound for **Filter - Approximate Set**



Lower Bound for **Filter**

$$|\text{Filter}| + |\text{Disambiguation}| \geq |\text{Exact Set}|$$

$$|Filter| \; + \; |Disambiguation| \; \geq \; |Exact\ Set|$$

Legend

ε - FPR
N - # entries
U - Universe size

|Filter| + **|Disambiguation|** ≥ |Exact Set|

What information must we add the filter to turn it into an exact set?

ε - FPR
N - # entries
U - Universe size

$$|\text{Filter}| + |\text{Disambiguation}| \geq N \cdot \log_2(U / N)$$

Plug in

ε - FPR
N - # entries
U - Universe size

$$|\textbf{Filter}| + |\text{Disambiguation}| \geq N \cdot \log_2(U / N)$$

Query filter U times

ε - FPR
N - # entries
U - Universe size

$$|\mathbf{Filter}| + |\text{Disambiguation}| \geq N \cdot \log_2(U / N)$$

Tells us all positive keys

$$\mathbf{N + \varepsilon \cdot U}$$

ε - FPR
N - # entries
U - Universe size

$$|\mathbf{Filter}| + |\text{Disambiguation}| \geq N \cdot \log_2(U / N)$$

Tells us all positive keys

$\boldsymbol{\varepsilon \cdot U}$

ε - FPR
N - # entries
U - Universe size

$$|\text{Filter}| \;+\; |\textbf{Disambiguation}| \;\geq\; N \cdot \log_2(U / N)$$

Of all positives $\varepsilon \cdot U$, which keys are true positives?

ε - FPR
N - # entries
U - Universe size

$$|\text{Filter}| + |\textbf{Disambiguation}| \geq N \cdot \log_2(U / N)$$

$\varepsilon \cdot U$ choose N

ε - FPR
N - # entries
U - Universe size

$$|\text{Filter}| + |\textbf{Disambiguation}| \geq N \cdot \log_2(U / N)$$

$$\uparrow$$

$$\boldsymbol{\log_2(\varepsilon \cdot U \text{ choose } N)}$$

ε - FPR
N - # entries
U - Universe size

$$|\text{Filter}| + |\textbf{Disambiguation}| \geq N \cdot \log_2(U / N)$$

$$\mathbf{N \cdot \log_2((\varepsilon \cdot U) / N)}$$

ε - FPR
N - # entries
U - Universe size

$$|\text{Filter}| \;+\; \mathbf{N \cdot \log_2((\varepsilon \cdot U) / N)} \;\geq\; N \cdot \log_2(U / N)$$

Legend

ε - FPR
N - # entries
U - Universe size

$$|\text{Filter}| \geq N \cdot \log_2(U / N) - \mathbf{N \cdot \log_2((\varepsilon \cdot U) / N)}$$

ε - FPR
N - # entries
U - Universe size

$$|\text{Filter}| \geq N \cdot \log_2(1 / \varepsilon)$$

ε - FPR
N - # entries
U - Universe size

$$\varepsilon \geq 2^{-M/N}$$

ε - FPR
N - # entries
U - Universe size

Bloom

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

Lower bound

$\boldsymbol{2^{-M/N}}$

Bloom

$\approx 2^{-M/N \cdot \mathbf{0.69}}$

Lower bound

$2^{-M/N}$

Bloom

$\approx 2^{-M/N \cdot 0.69}$

???

Lower bound

$2^{-M/N}$

Bloom	**XOR Filter**	Lower bound
$\approx 2^{-M/N \cdot 0.69}$	$\approx \mathbf{2^{-M/N \cdot 0.81}}$	$\approx 2^{-M/N}$

XOR Filter

$$\oplus$$

Xor Filters: Faster and Smaller Than Bloom Filters

Thomas Mueller Graf, Daniel Lemire

Journal of Experimental Algorithmics, 2020

XOR Operator

Input 1	0	0	1	1
		$\oplus$		
Input 2	0	1	0	1
		=		
Parity	0	1	1	0

Parity can help recover any input

Suppose we lost input 2

Input 1	0	0	1	1
Parity	0	1	1	0

Parity can help recover any input

Suppose we lost input 2

Input 1	0	0	1	1
		$\oplus$		
Parity	0	1	1	0
		$=$		
Input 2	0	1	0	1

Recovered

Parity can help recover any input

Or suppose we lost input 1

Input 2	0	1	0	1
Parity	0	1	1	0

Parity can help recover any input

Or suppose we lost input 1

Input 2	0	1	0	1	
		$\oplus$			
Parity	0	1	1	0	
		=			
Input 1	0	0	1	1	**Recovered**

XOR is commutative and associative

Input 1	0	0	0	0	1	1	1	1
				$\oplus$				
Input 2	0	0	1	1	0	0	1	1
				$\oplus$				
Input 3	0	1	0	1	0	1	0	1
				=				
Parity	0	1	1	0	1	0	0	1

XOR is commutative and associative

Input 1	0	0	0	0	1	1	1	1
				$\oplus$				
Input 2	0	0	1	1	0	0	1	1
				$\oplus$				
Input 3	0	1	0	1	0	1	0	1
				=				
Parity	0	1	1	0	1	0	0	1

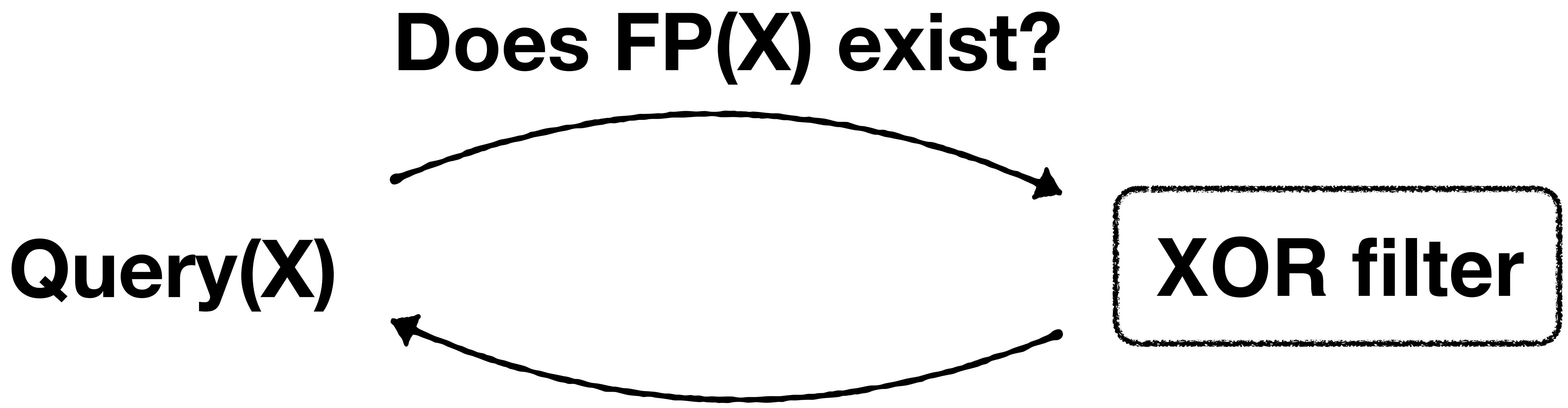
Parity can recover any input, as long as we also have all the other inputs

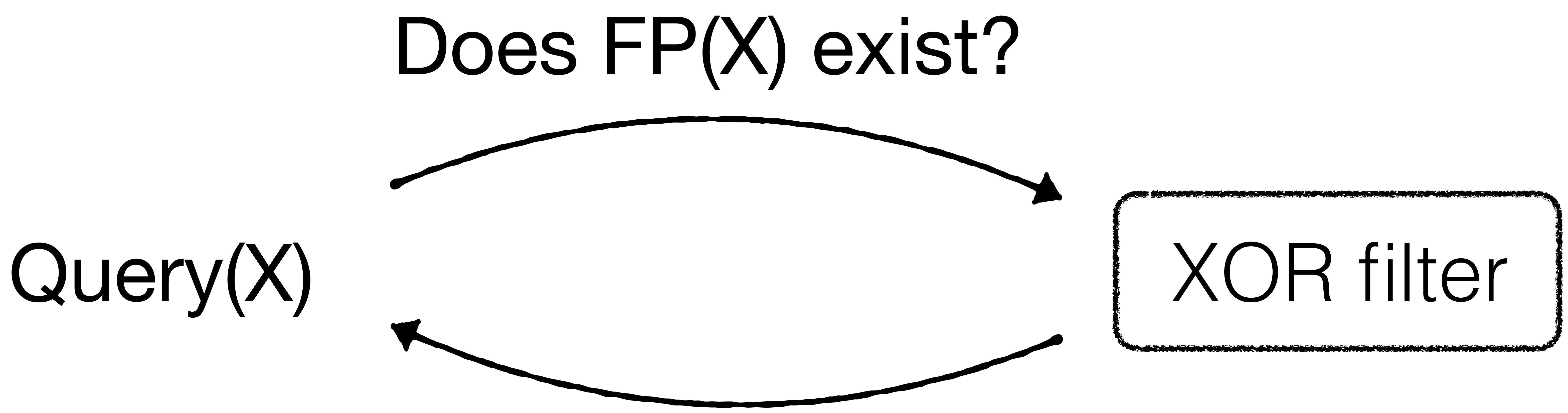
XOR filter stores a fingerprint for each key

FP() = 

XOR filter stores a fingerprint for each key

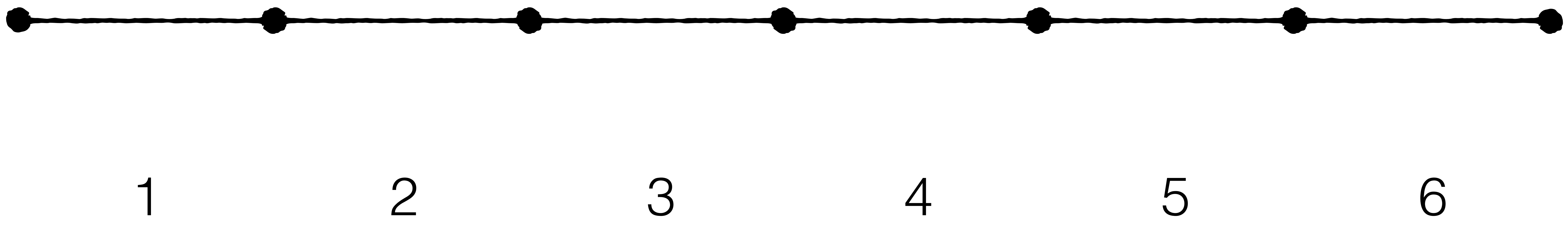
Example: $$\mathbf{FP(\ X\ ) =}\ \overbrace{0100}^{\textbf{F bits}}$$


Does FP(X) exist?
Query(X)
XOR filter

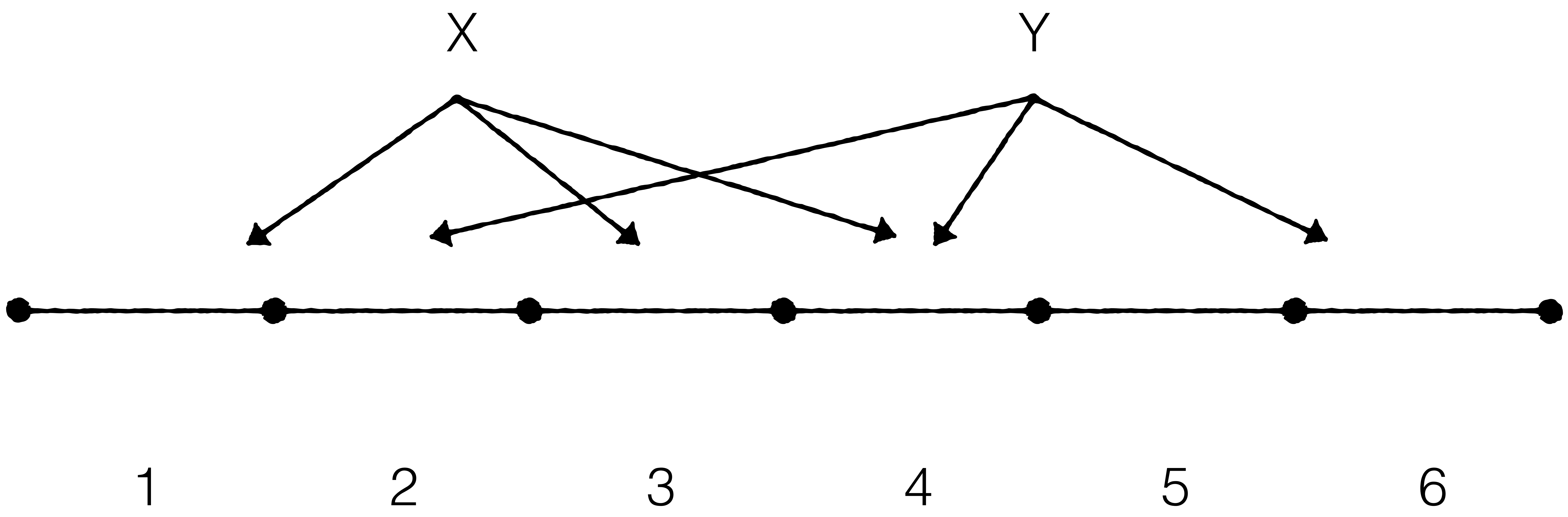


1. **True negative**
2. **True positive**
3. **False positive with probability 2^{-F}**

How does XOR filter store its fingerprints?

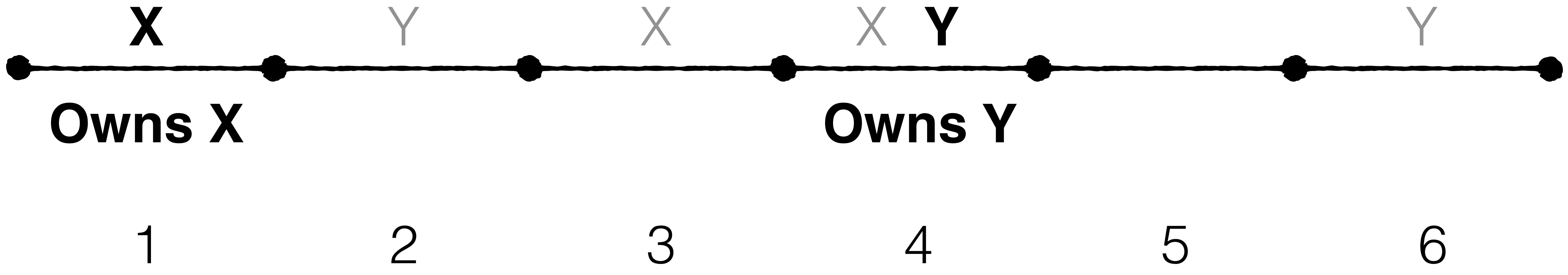


Hash each entry to three slots

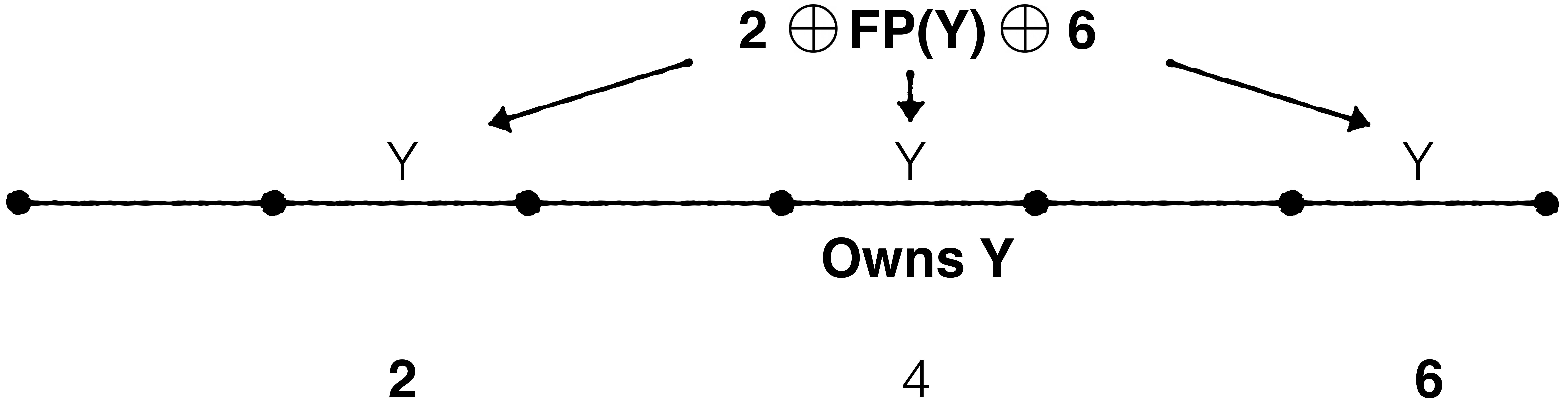


Assign one slot to uniquely own each entry

(More on this shortly)

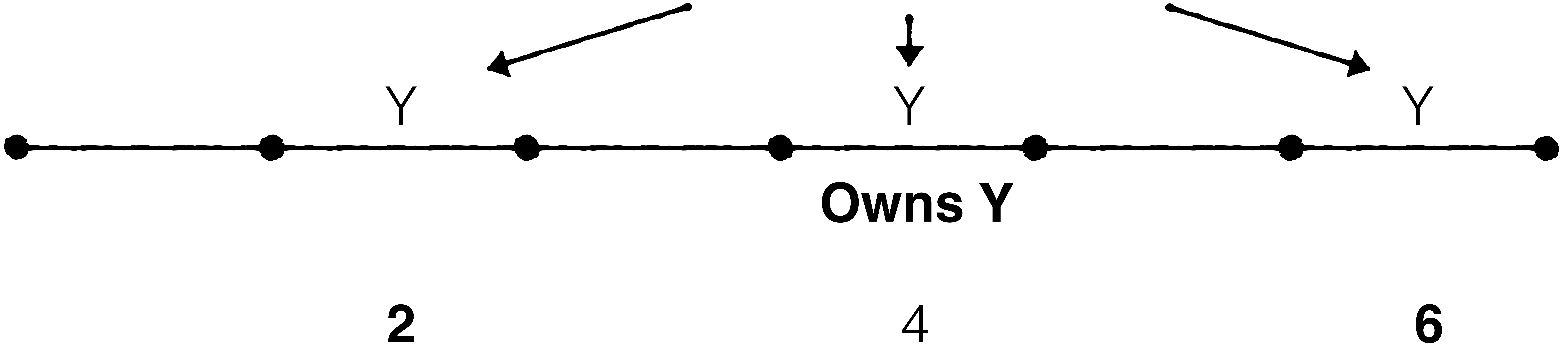


Each bucket stores XOR of fingerprint and other two slots

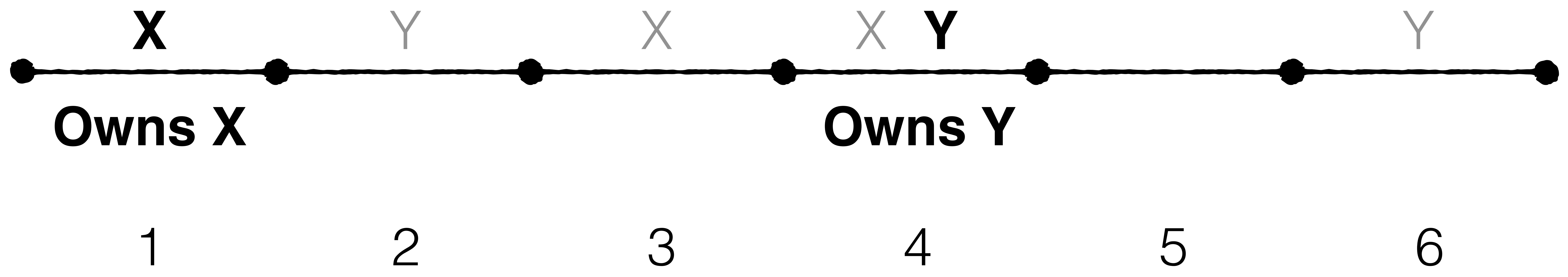


During queries, recover fingerprints by xoring three slots

get(Y) returns true if $\mathbf{FP(Y) = 2 \oplus 4 \oplus 6}$



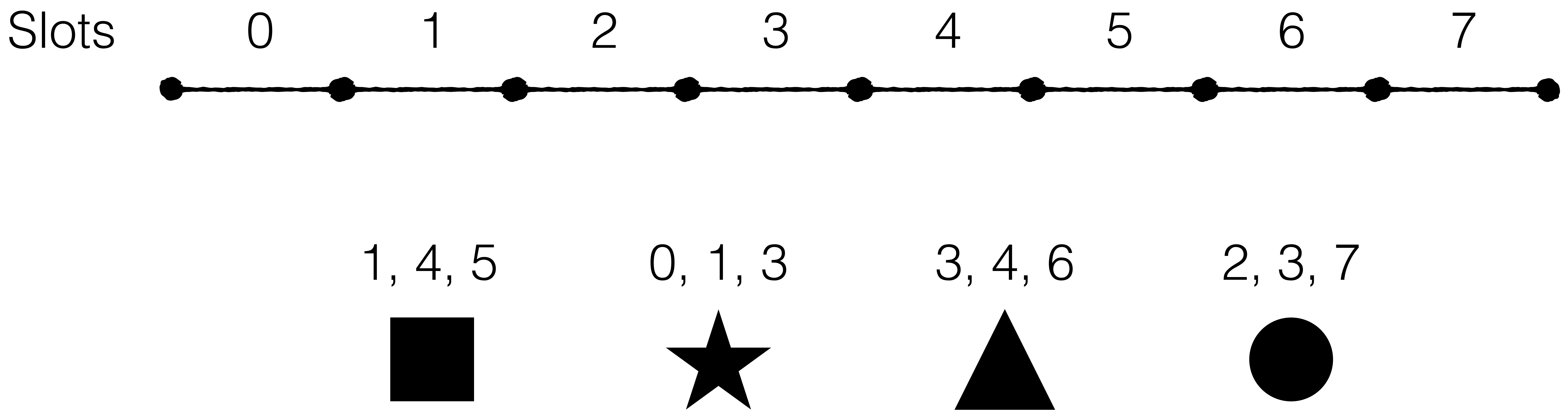
How to assign slots to own different entries?



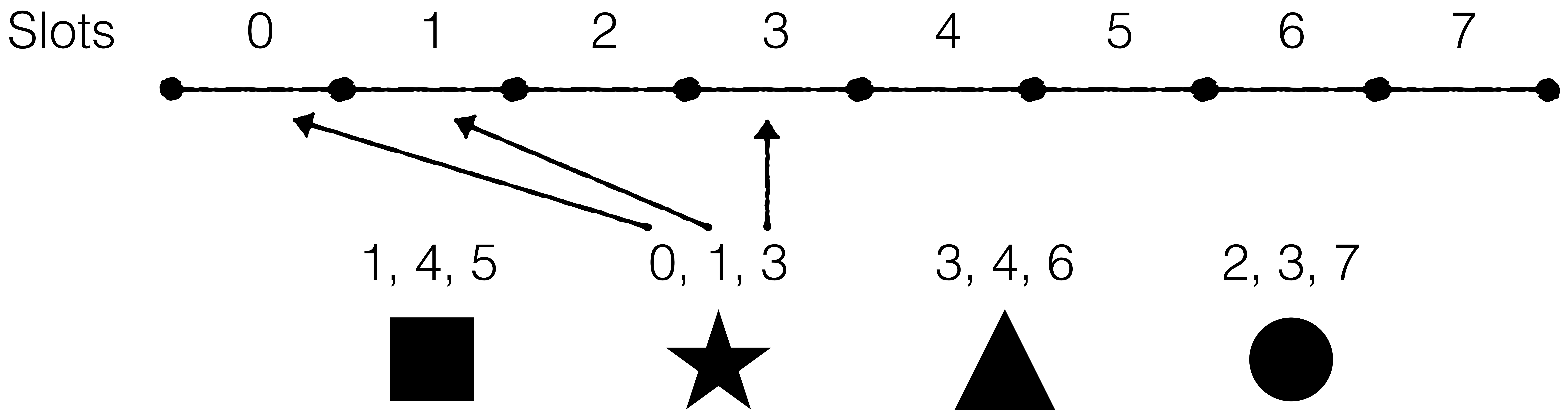
How to assign slots to own different entries?

Peeling

Peeling

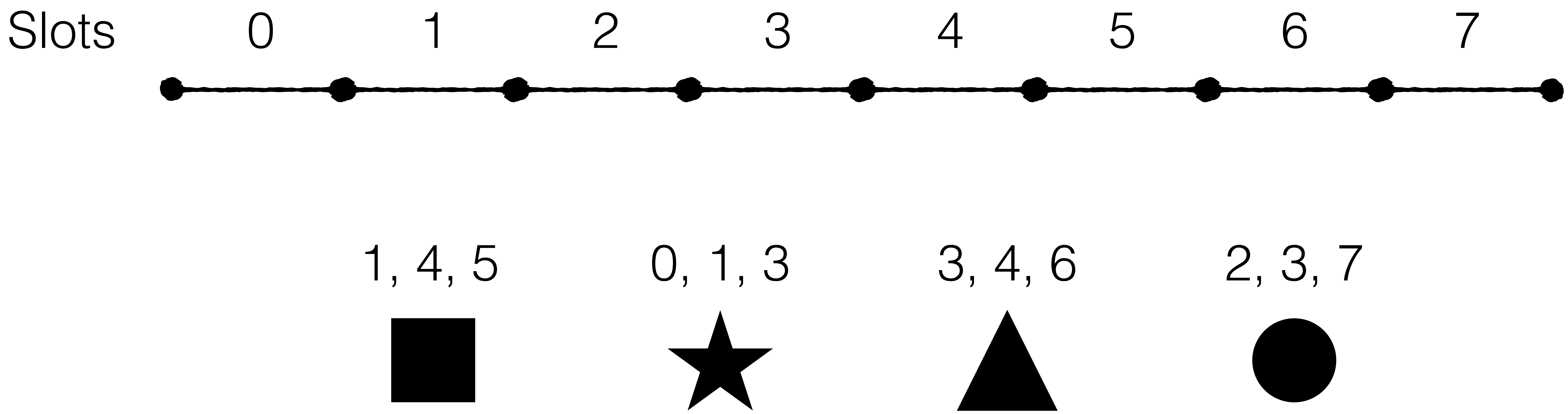


Peeling



Peeling

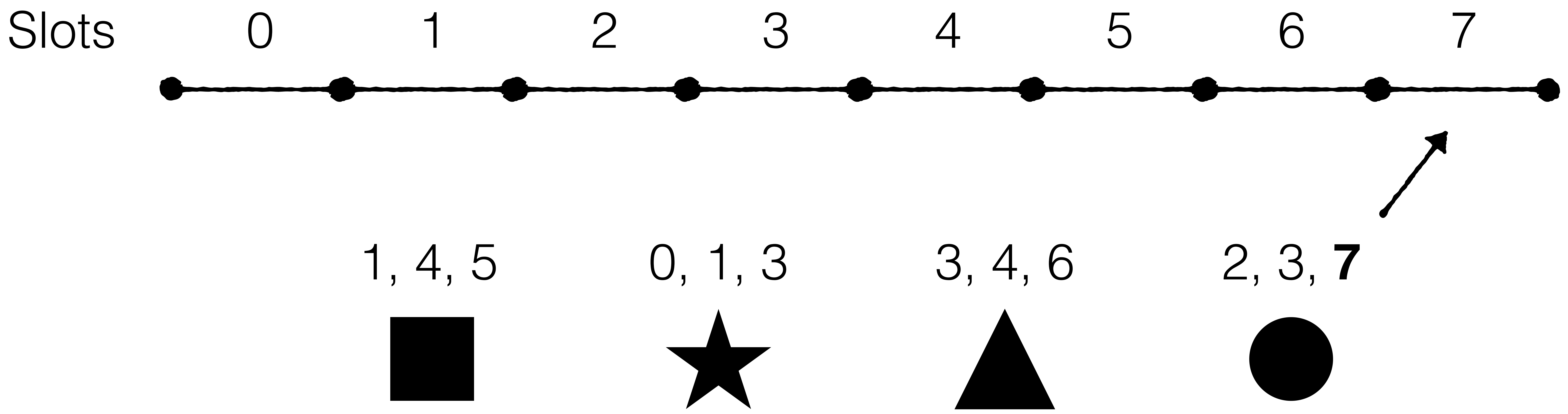
While not all keys have been assigned to a slot



Peeling

While not all keys have been assigned to a slot

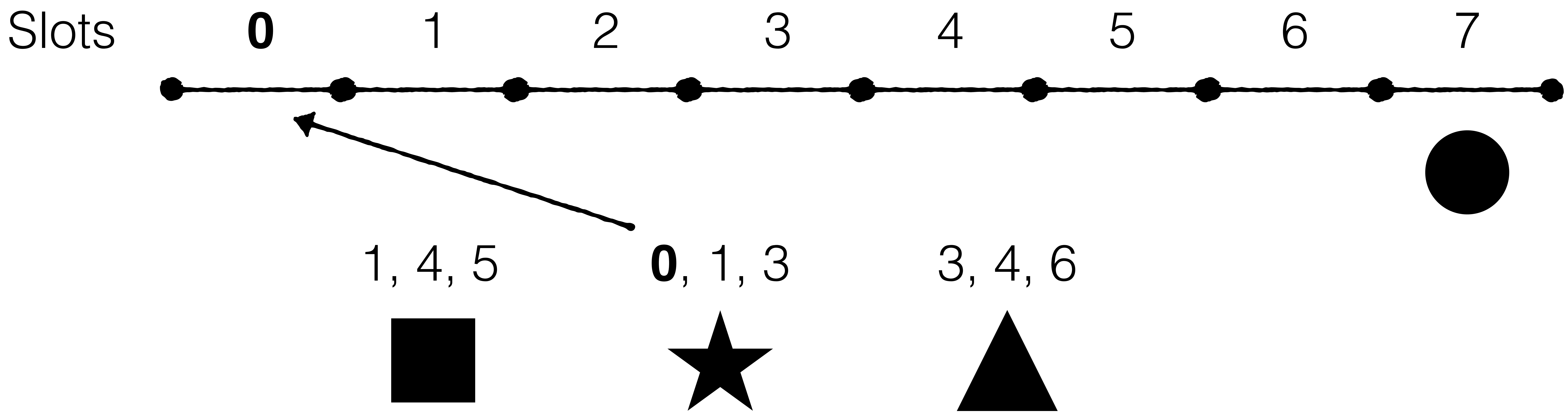
Assign some entry x to some slot y that only entry x maps to



Peeling

While not all keys have been assigned to a slot

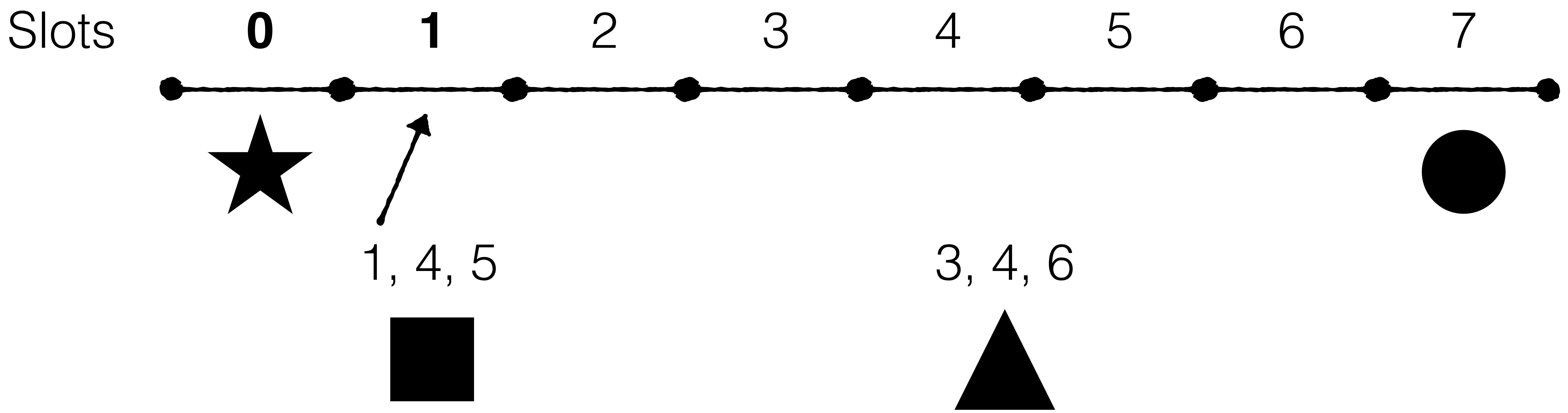
Assign some entry x to some slot y that only entry x maps to



Peeling

While not all keys have been assigned to a slot

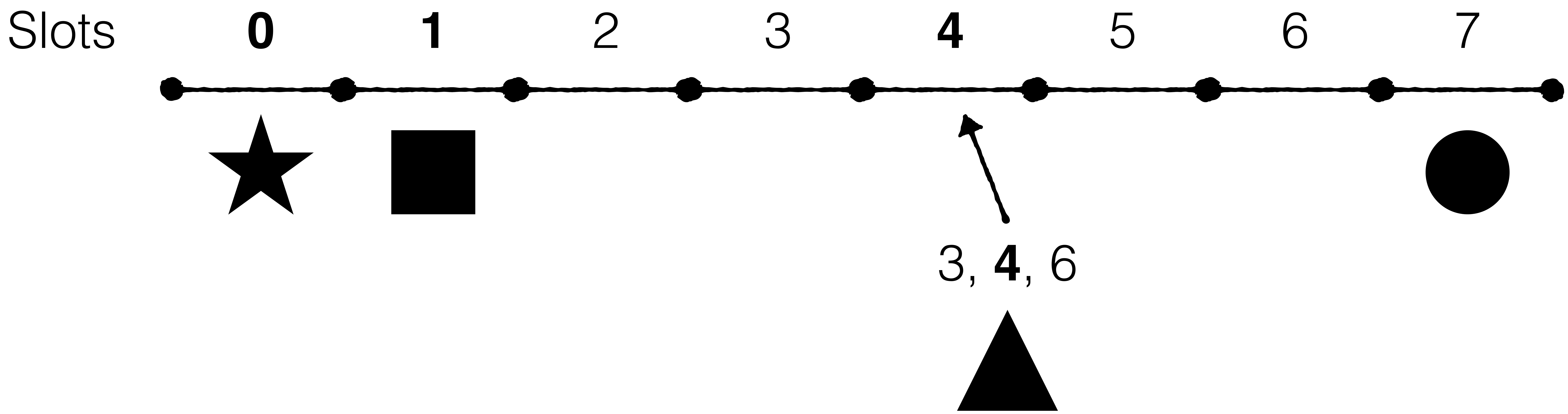
Assign some entry x to some slot y that only entry x maps to

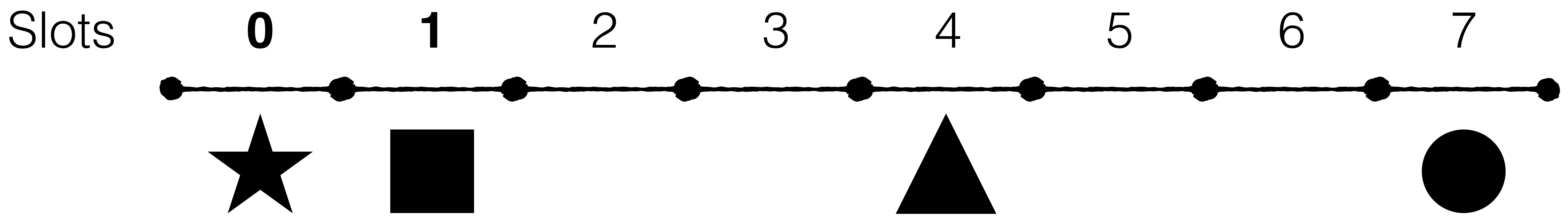


Peeling

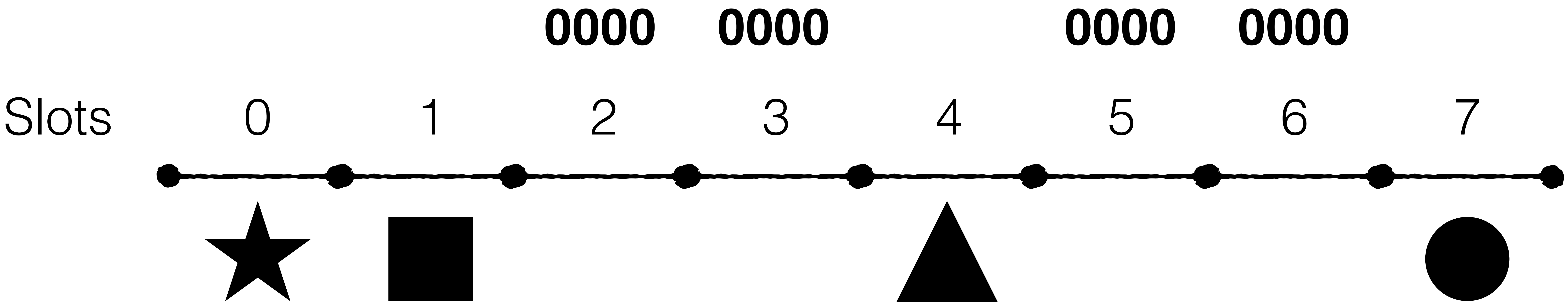
While not all keys have been assigned to a slot

Assign some entry x to some slot y that only entry x maps to



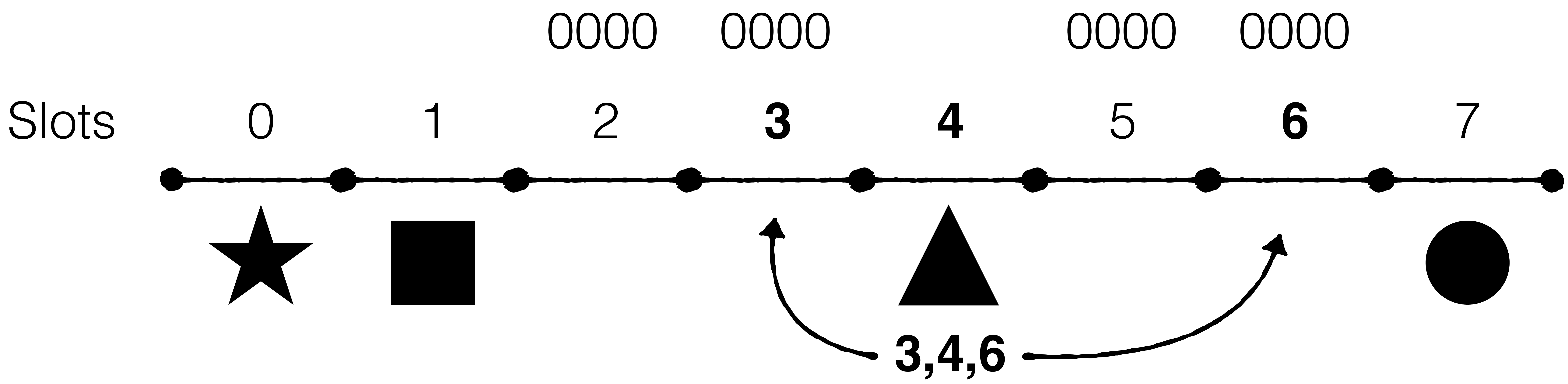

Slots
0
1
2
3
4
5
6
7

Populate all unassigned slots with zeros



Populate all unassigned slots with zeros

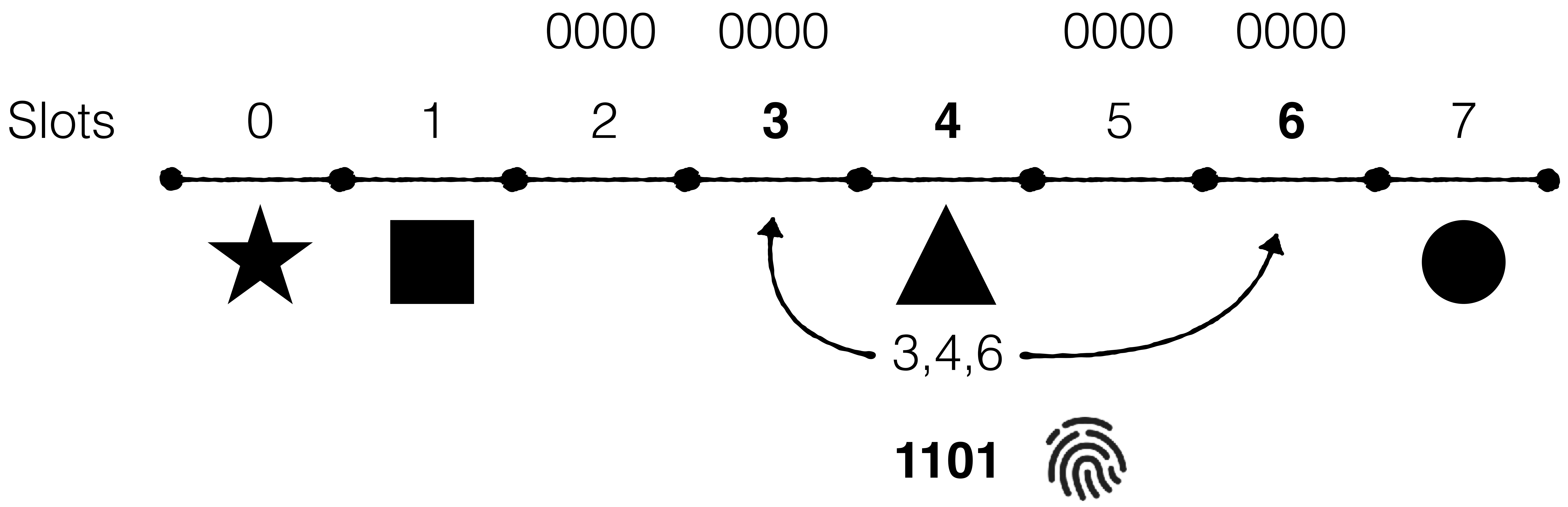
Find some entry whose only other candidate slots are populated



Populate all unassigned slots with zeros

Find some entry whose only other candidate slots are filled

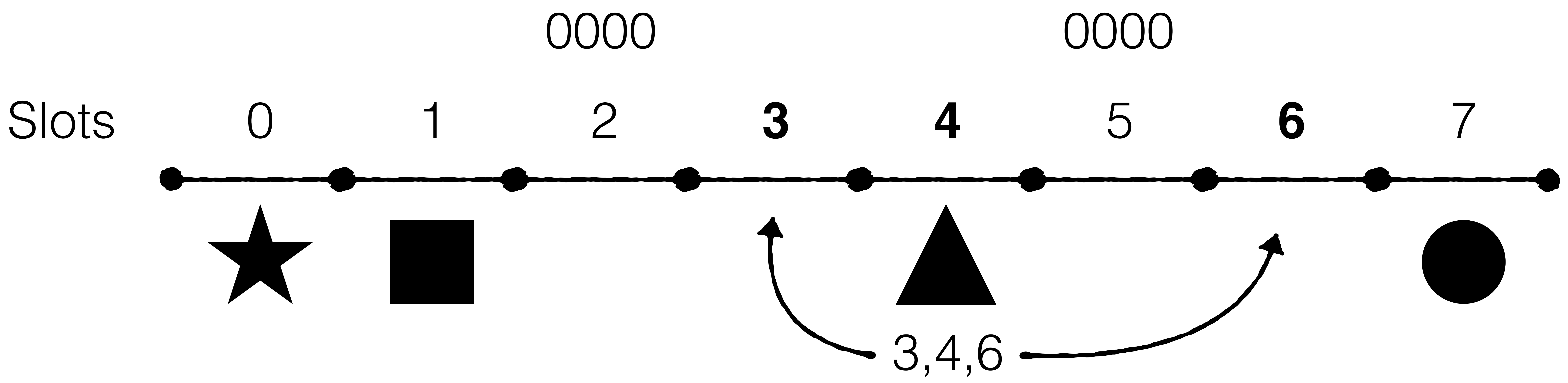
Xor its fingerprint with content at other slots and store



Populate all unassigned slots with zeros

Find some entry whose only other candidate slots are filled

Xor its fingerprint with content at other slots and store

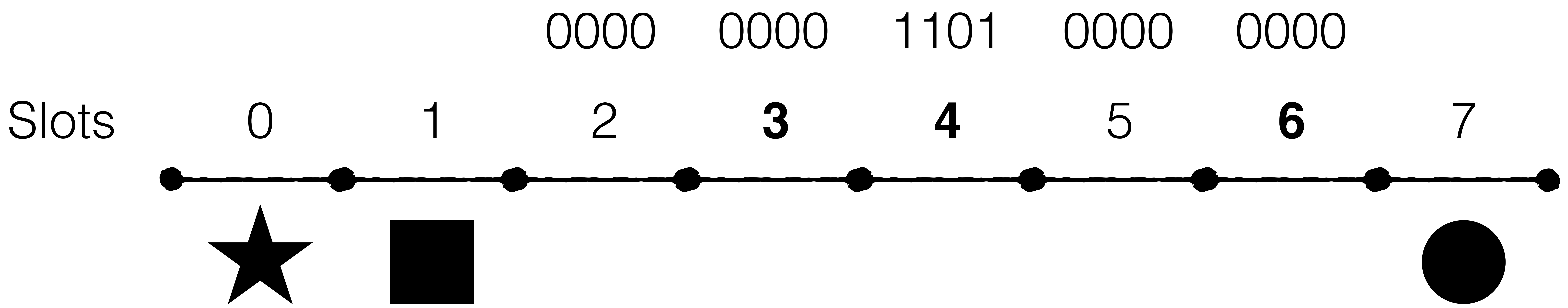


0000 ⊕ **1101** ⊕ 0000 = **1101**

Populate all unassigned slots with zeros

Find some entry whose only other candidate slots are filled

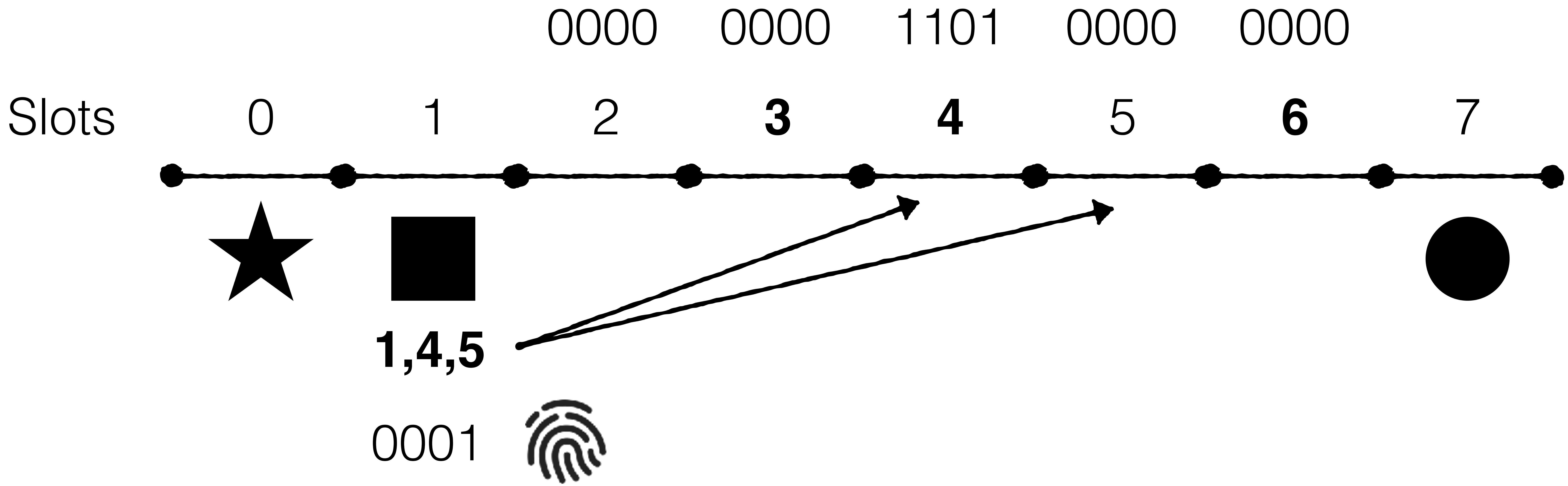
Xor its fingerprint with content at other slots and store



Populate all unassigned slots with zeros

Find some entry whose only other candidate slots are filled

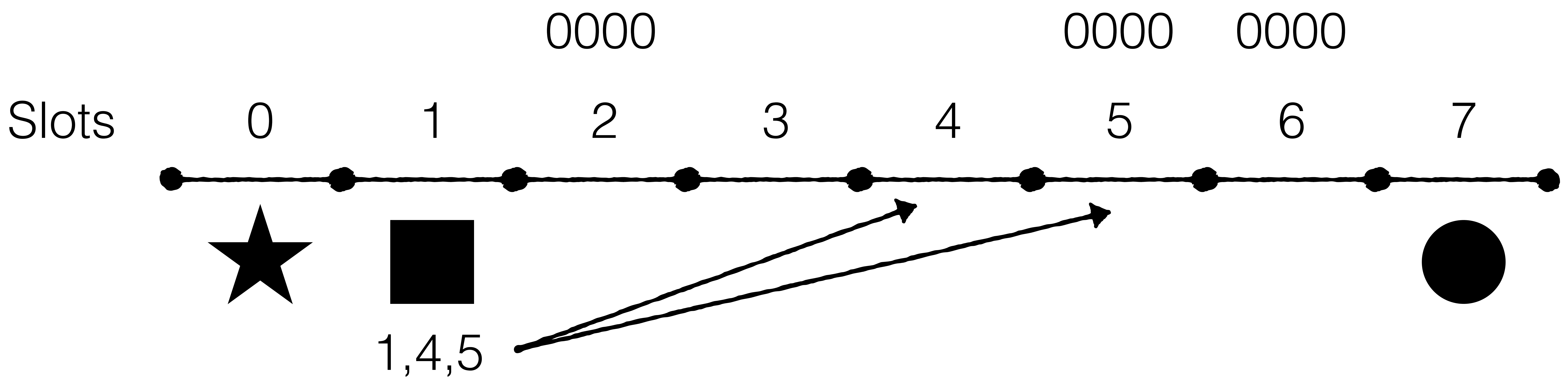
Xor its fingerprint with content at other slots and store



Populate all unassigned slots with zeros

Find some entry whose only other candidate slots are filled

Xor its fingerprint with content at other slots and store

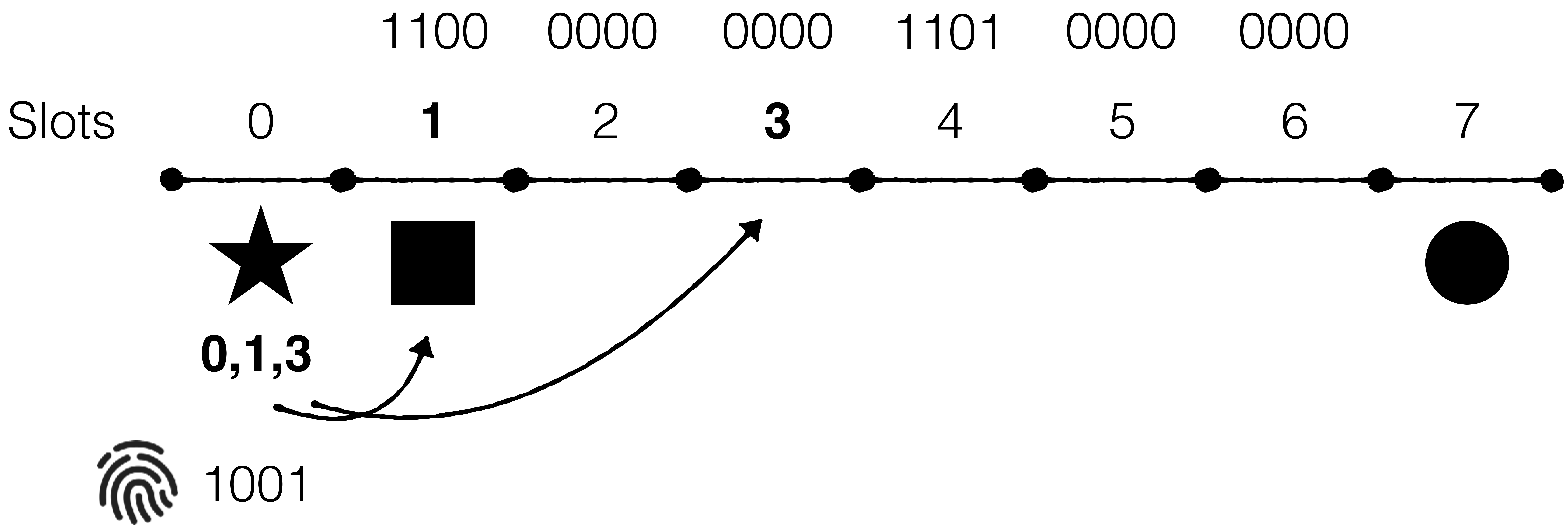


$0000 \oplus 0001 \oplus 1101 = \mathbf{1100}$

Populate all unassigned slots with zeros

Find some entry whose only other candidate slots are filled

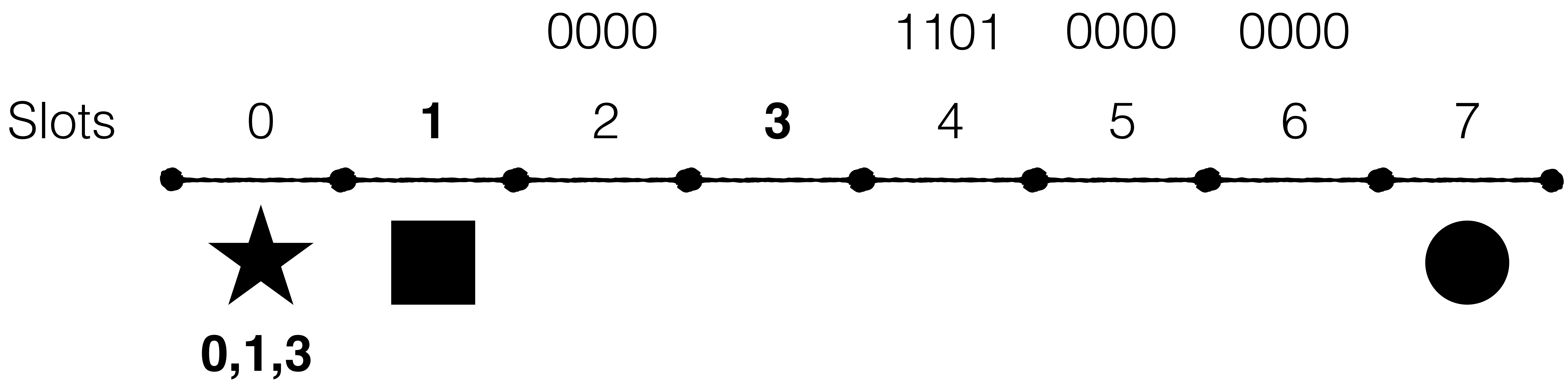
Xor its fingerprint with content at other slots and store



Populate all unassigned slots with zeros

Find some entry whose only other candidate slots are filled

Xor its fingerprint with content at other slots and store

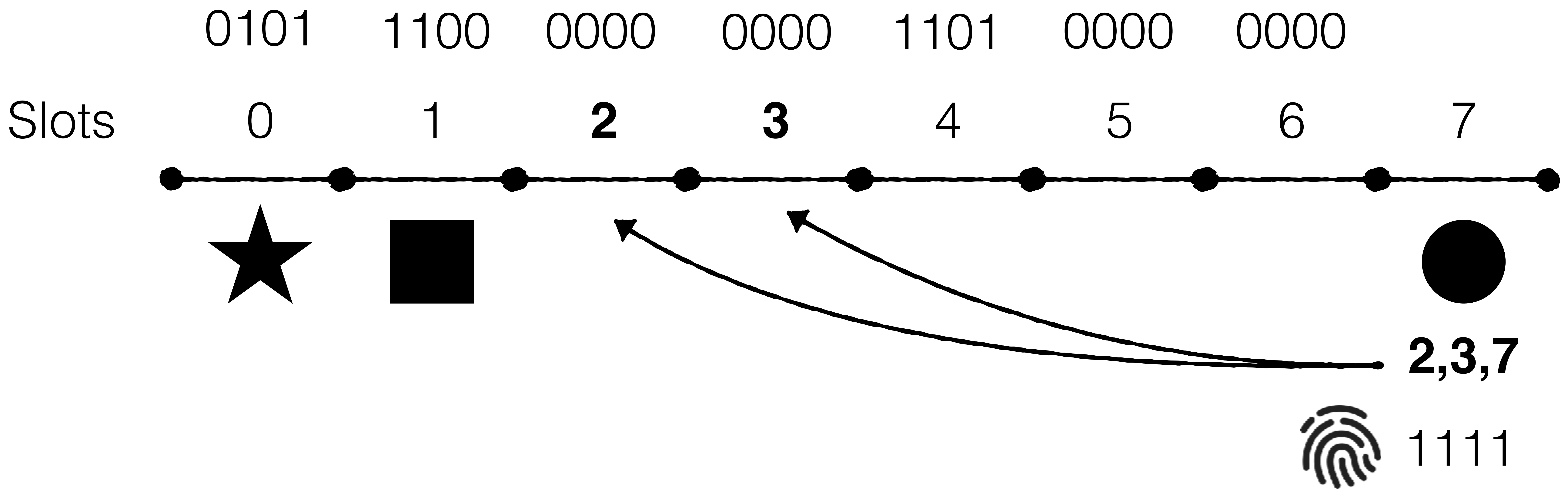


1100 ⊕ 1001 ⊕ 0000 = **0101**

Populate all unassigned slots with zeros

Find some entry whose only other candidate slots are filled

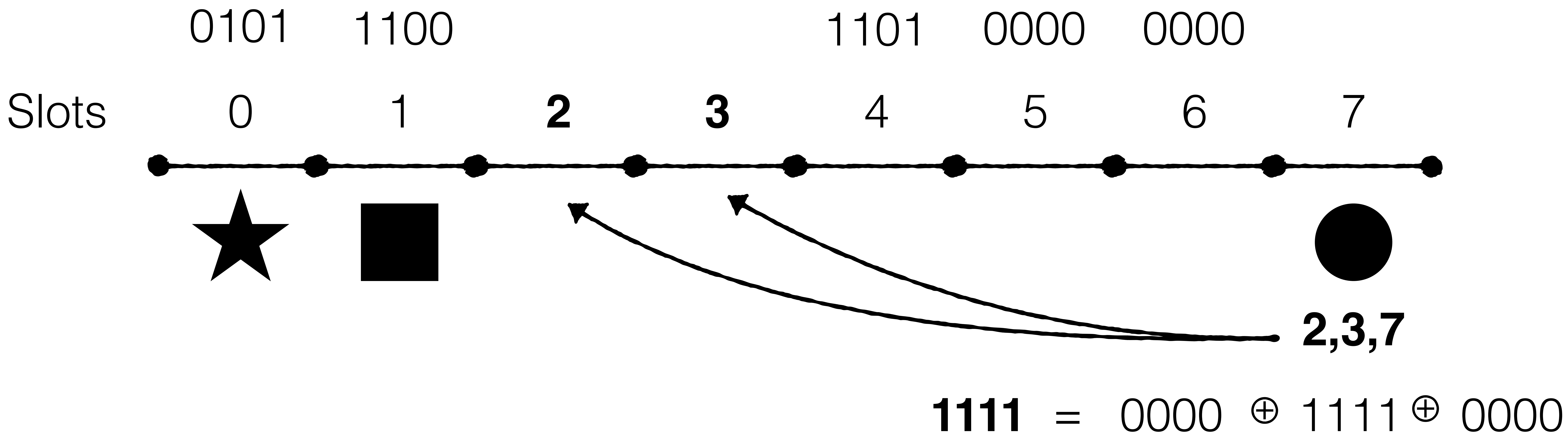
Xor its fingerprint with content at other slots and store



Populate all unassigned slots with zeros

Find some entry whose only other candidate slots are filled

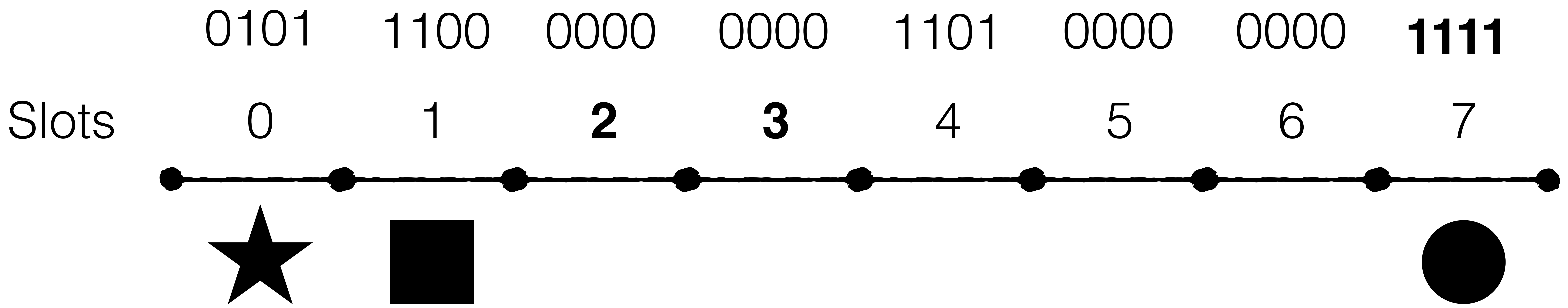
Xor its fingerprint with content at other slots and store



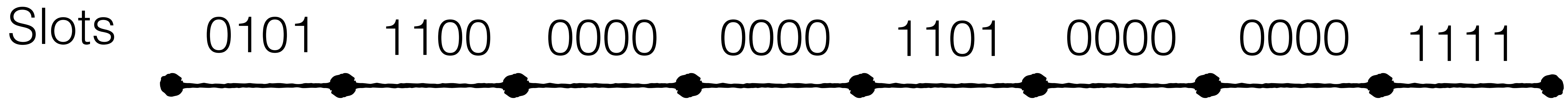
Populate all unassigned slots with zeros

Find some entry whose only other candidate slots are filled

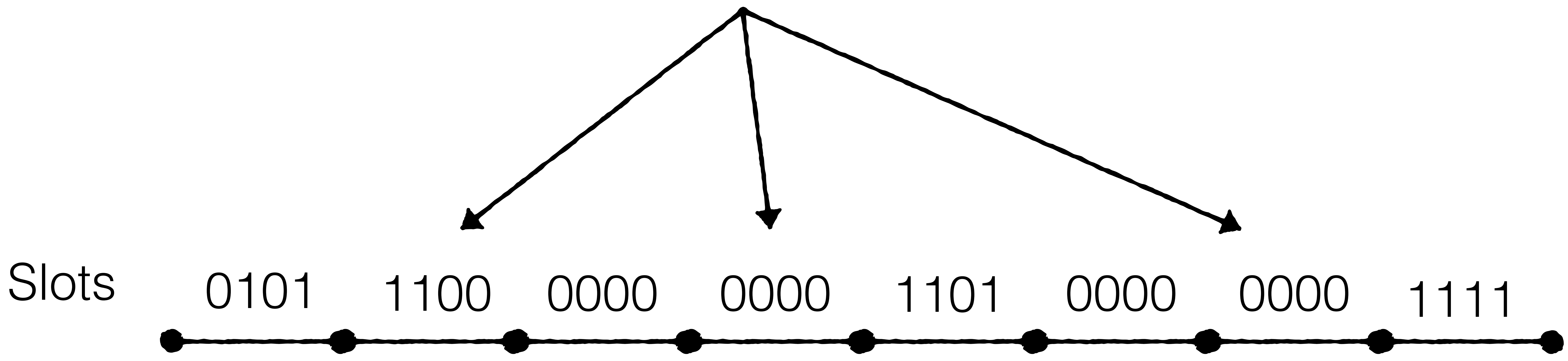
Xor its fingerprint with content at other slots and store



We’re done :)



Query(X) where FP(X) = 1001



Query(X) where FP(X) = **1001**

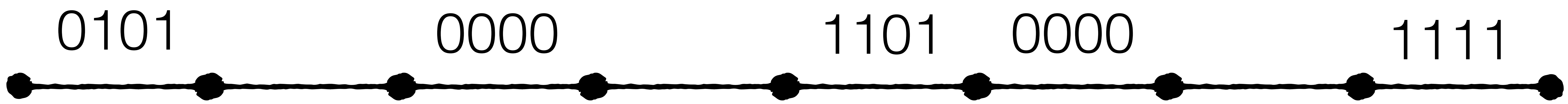
$1100 \oplus 0000 \oplus 0000 = \mathbf{1100}$

Not a fingerprint match so return negative

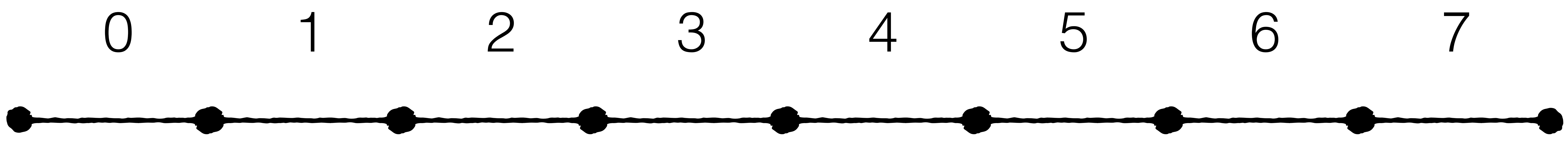
Query(X) where FP(X) = **1001**

1100 $\oplus$ 0000 $\oplus$ 0000 = **1100**

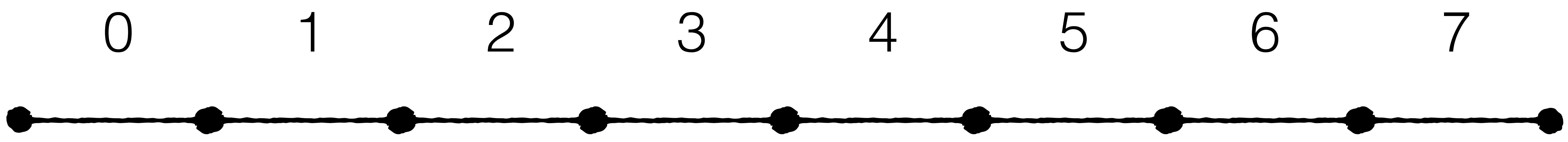
Slots



Construction can fail if there is no entry we can peel

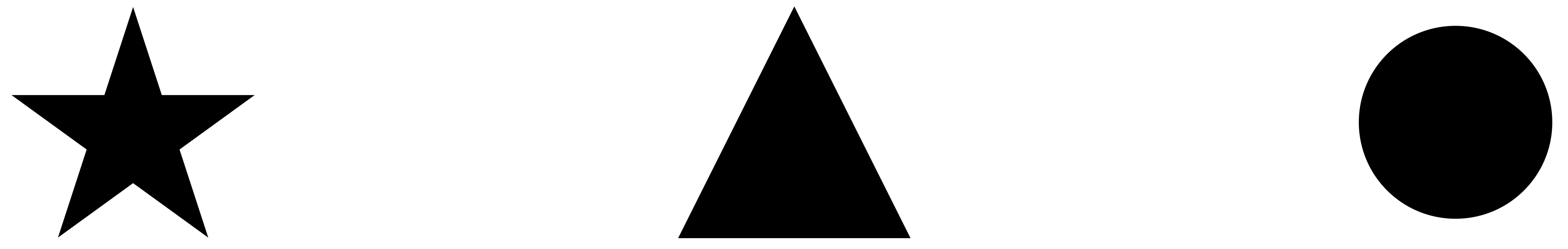


Construction can fail if there is no entry we can peel

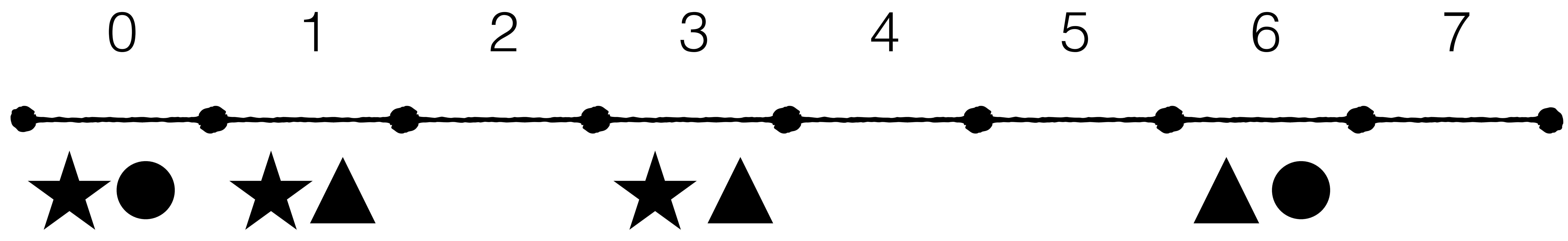


0, 1, 3   1, 3, 6   3, 6, 0

Example:

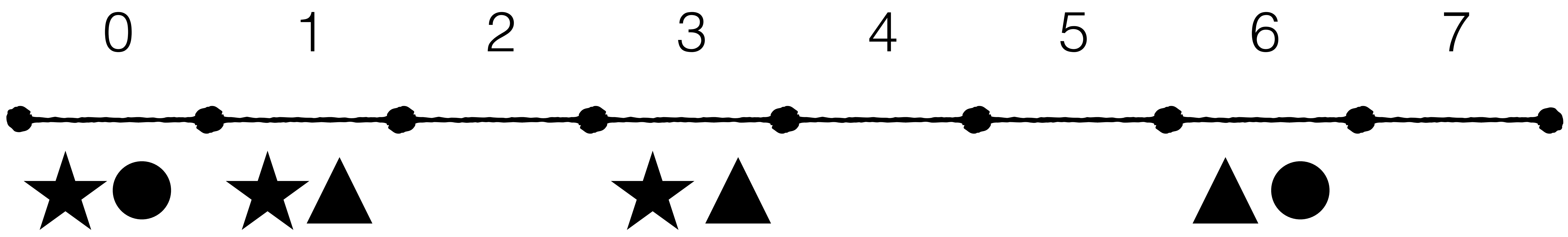


Construction can fail if there is no entry we can peel



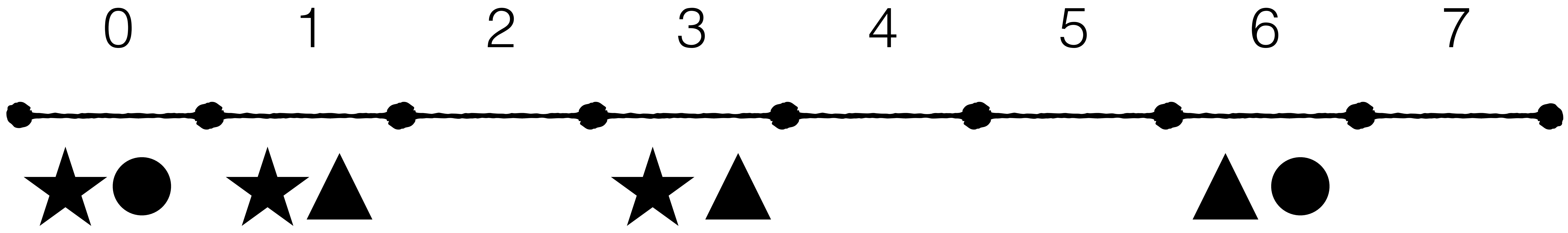
No slot has one entry uniquely mapping to it

If we fail, we must restart from scratch.



If we fail, we must restart from scratch.

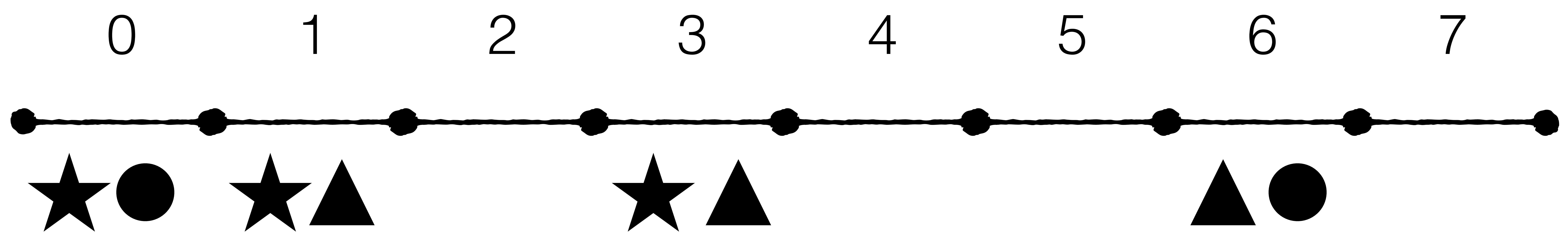
free space is necessary to succeed with high probability



If we fail, we must restart from scratch.

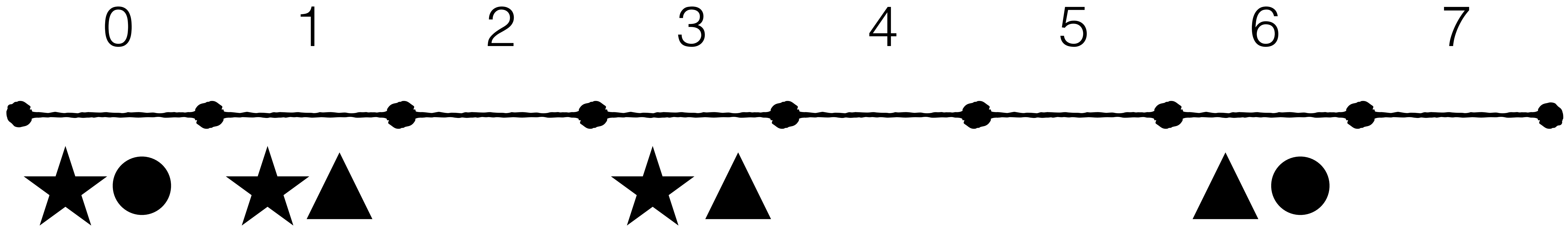
free space is necessary to succeed with high probability

What's the interplay between free space and # number of hash functions?



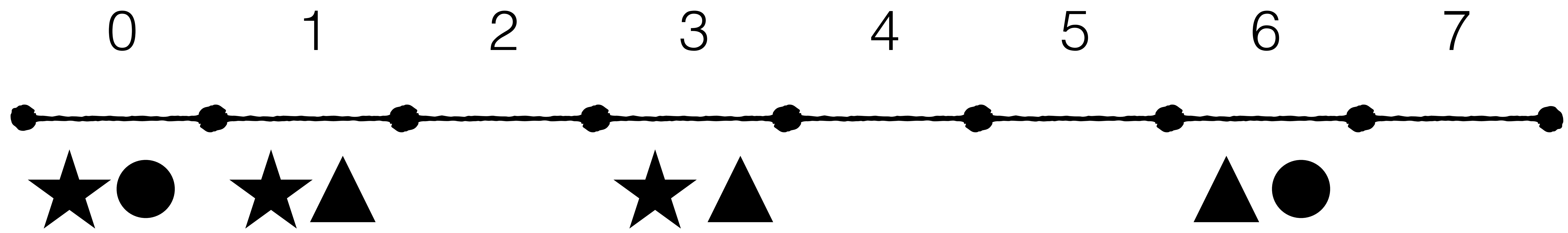
What's the interplay between free space and # number of hash functions?

Too few hash functions e.g., 1?



What's the interplay between free space and # number of hash functions?

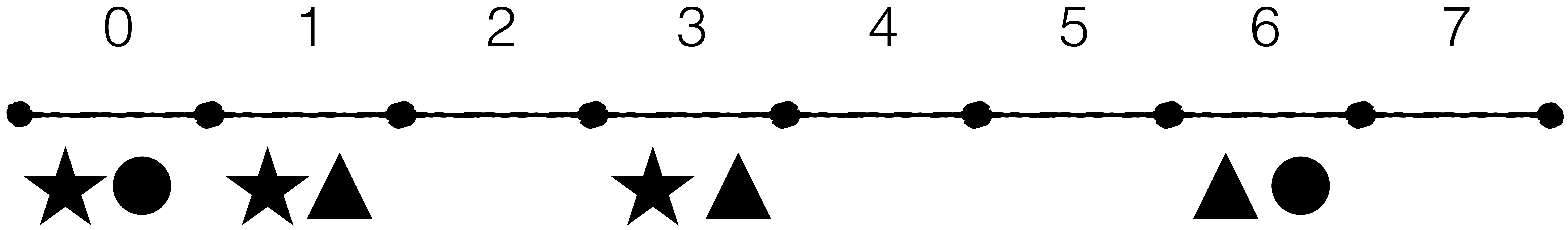
Too few hash functions e.g., 1? **Any collision makes us fail**



What's the interplay between free space and # number of hash functions?

Too few hash functions e.g., 1? Any collision makes us fail

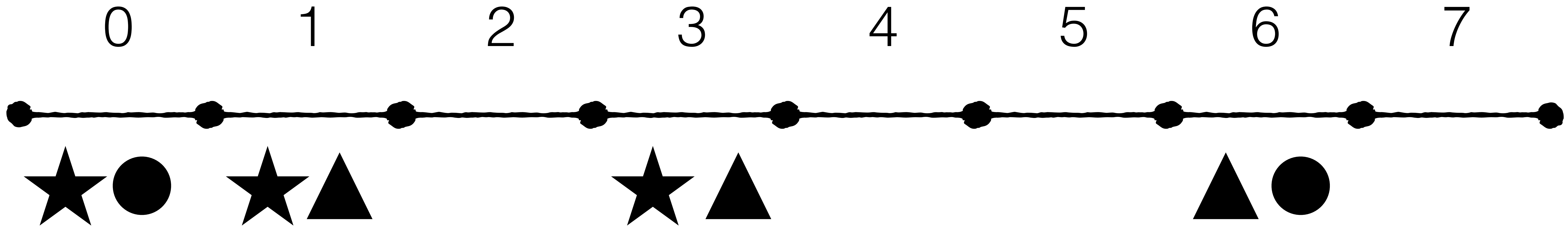
Too many hash functions e.g., n?

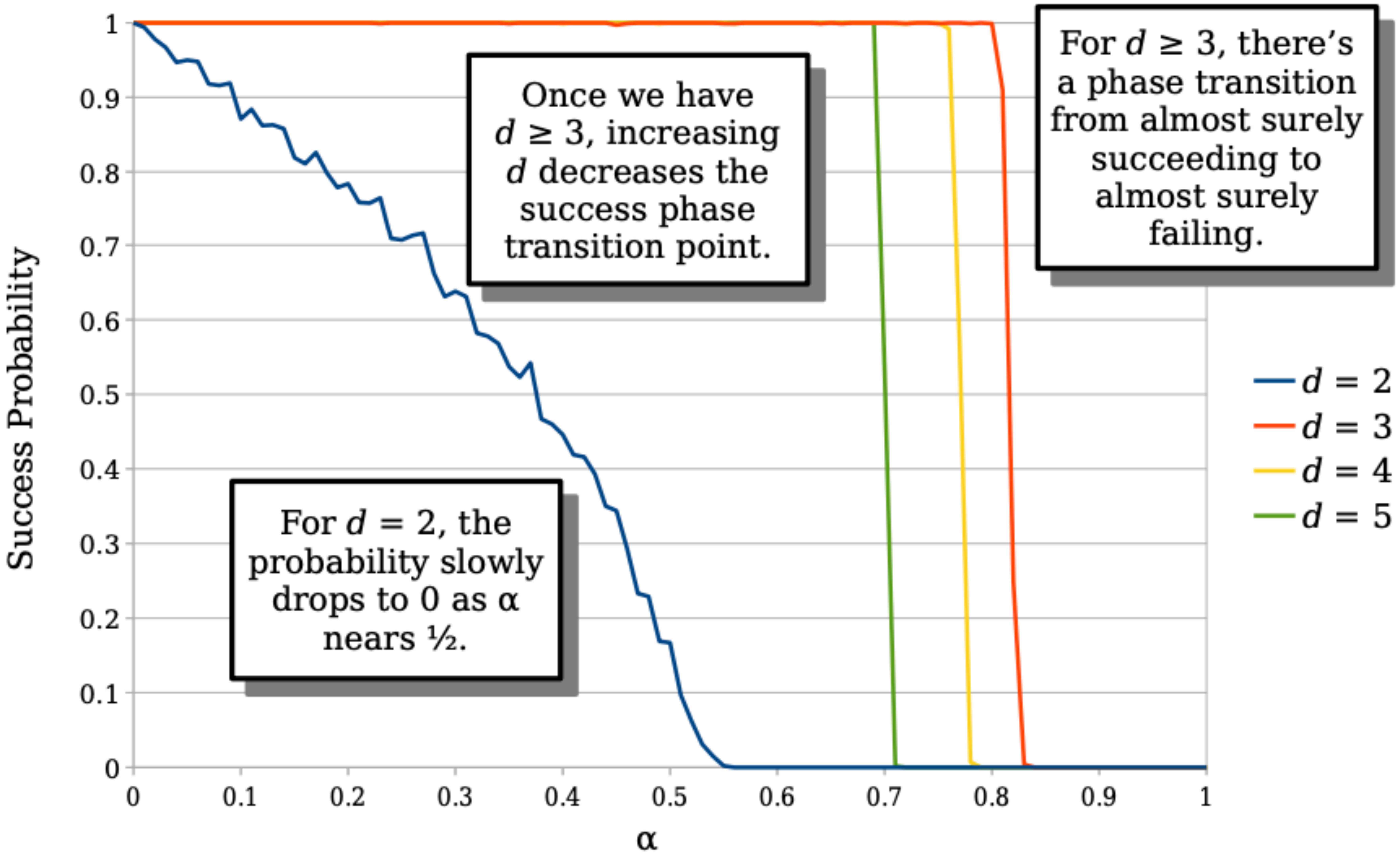


What's the interplay between free space and # number of hash functions?

Too few hash functions e.g., 1? Any collision makes us fail

Too many hash functions e.g., n? **Nothing is peelable**

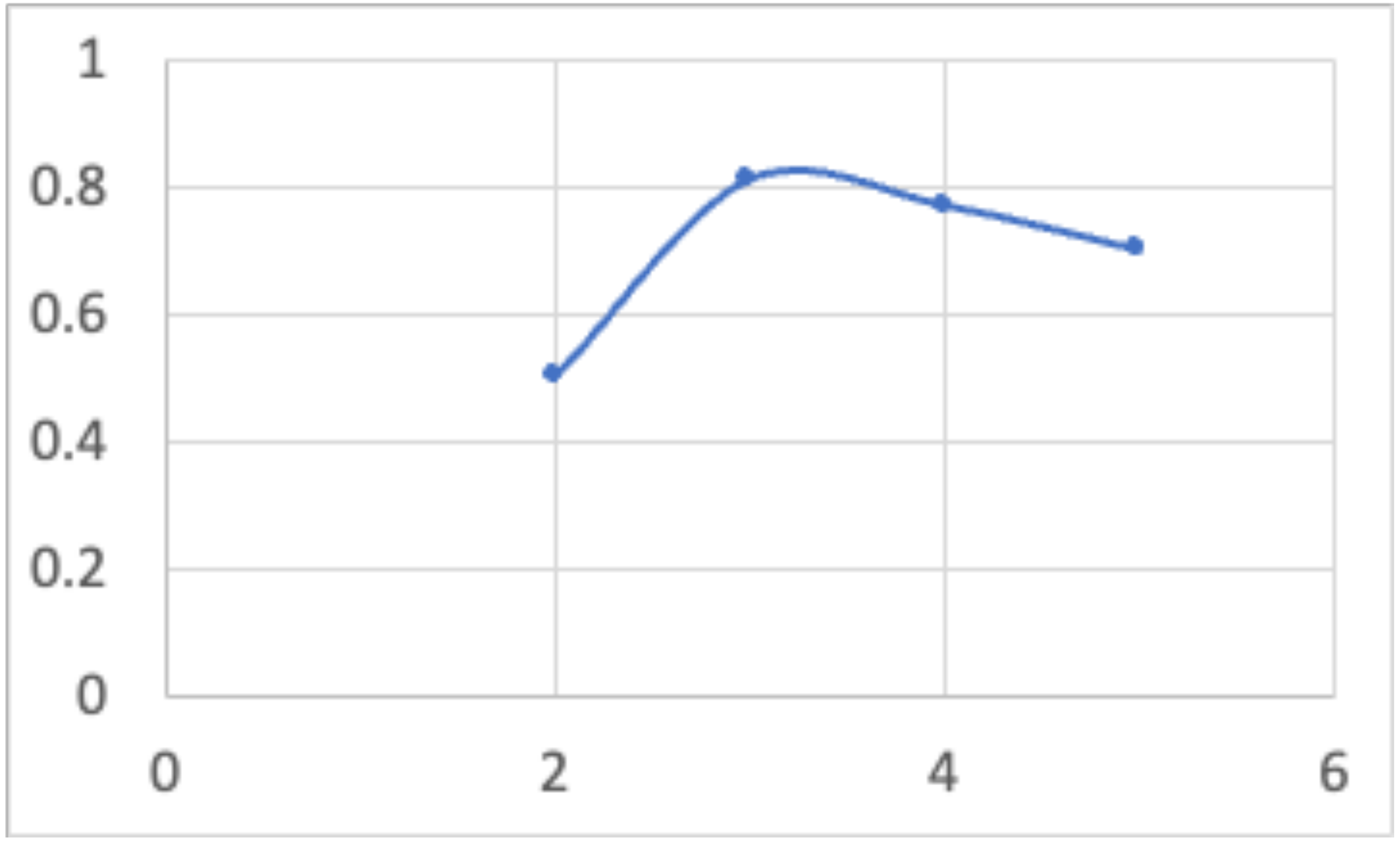




Create a table of m slots and place $n = \alpha m$ items into it.
Each item has d hashes.
What is the probability that the peeling algorithm succeeds?

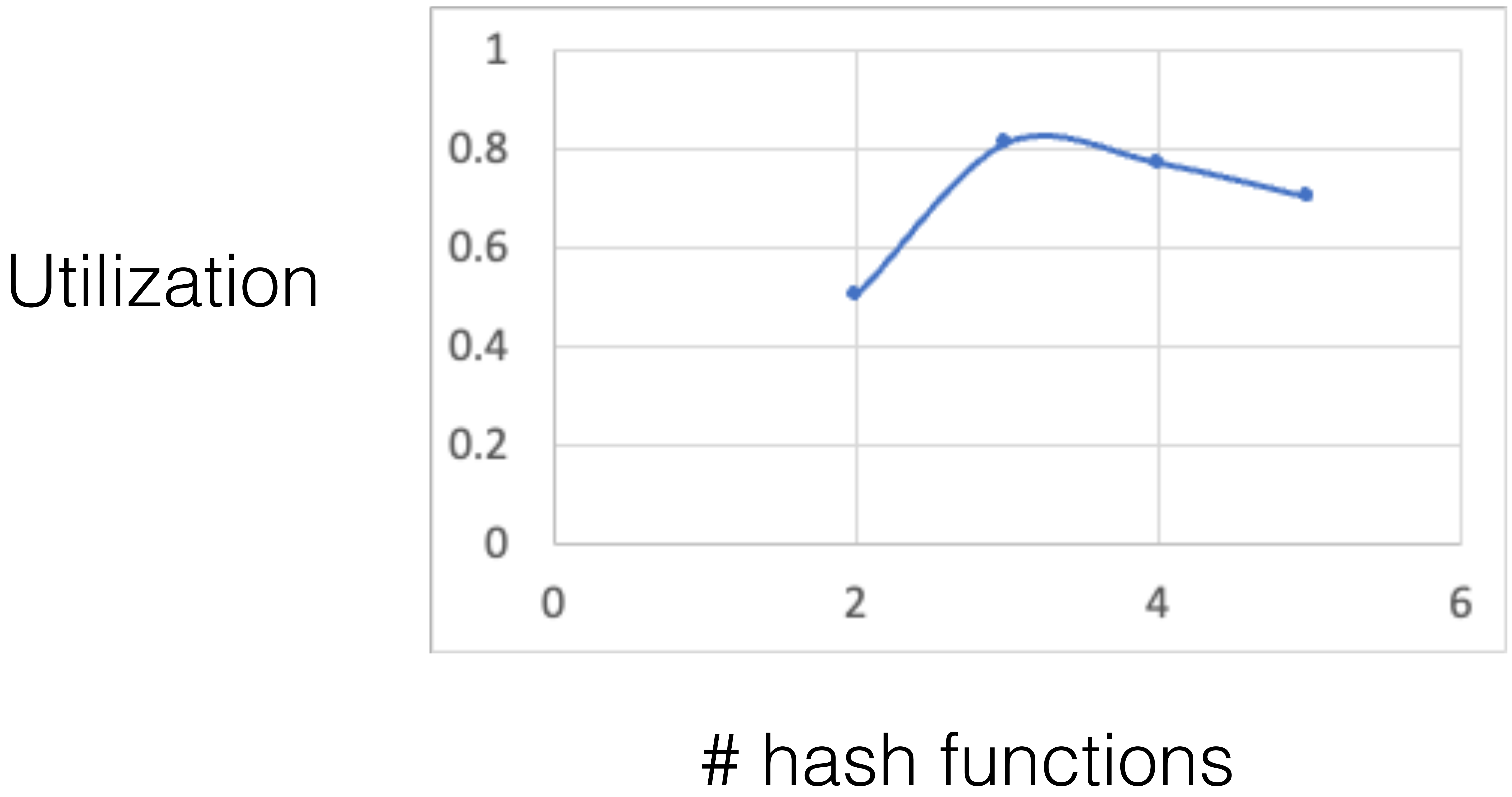
Slide by Keith Schwarz of Stanford

Utilization

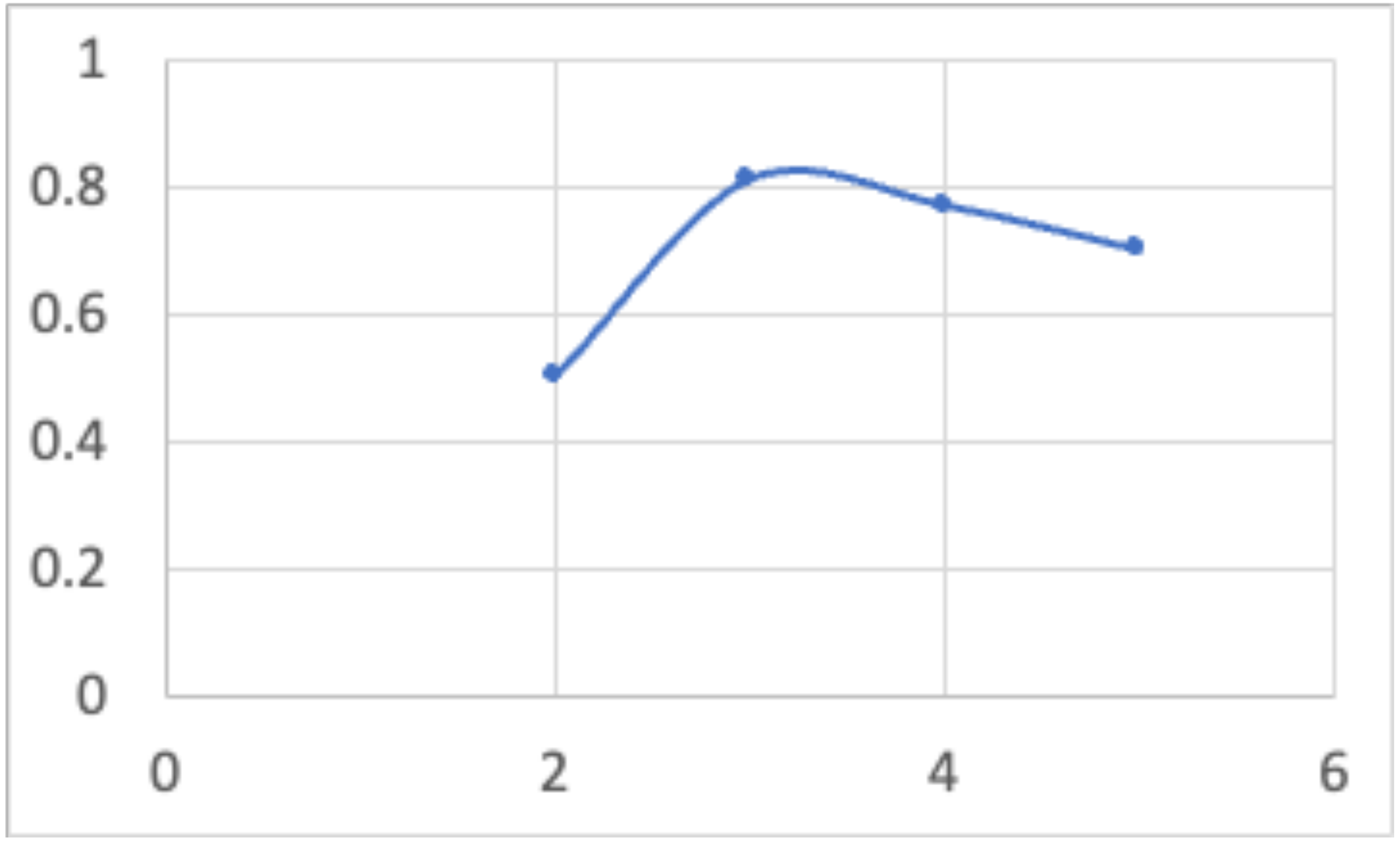


hash functions

Not enough placement flexibility



Too many items hashing to each slot

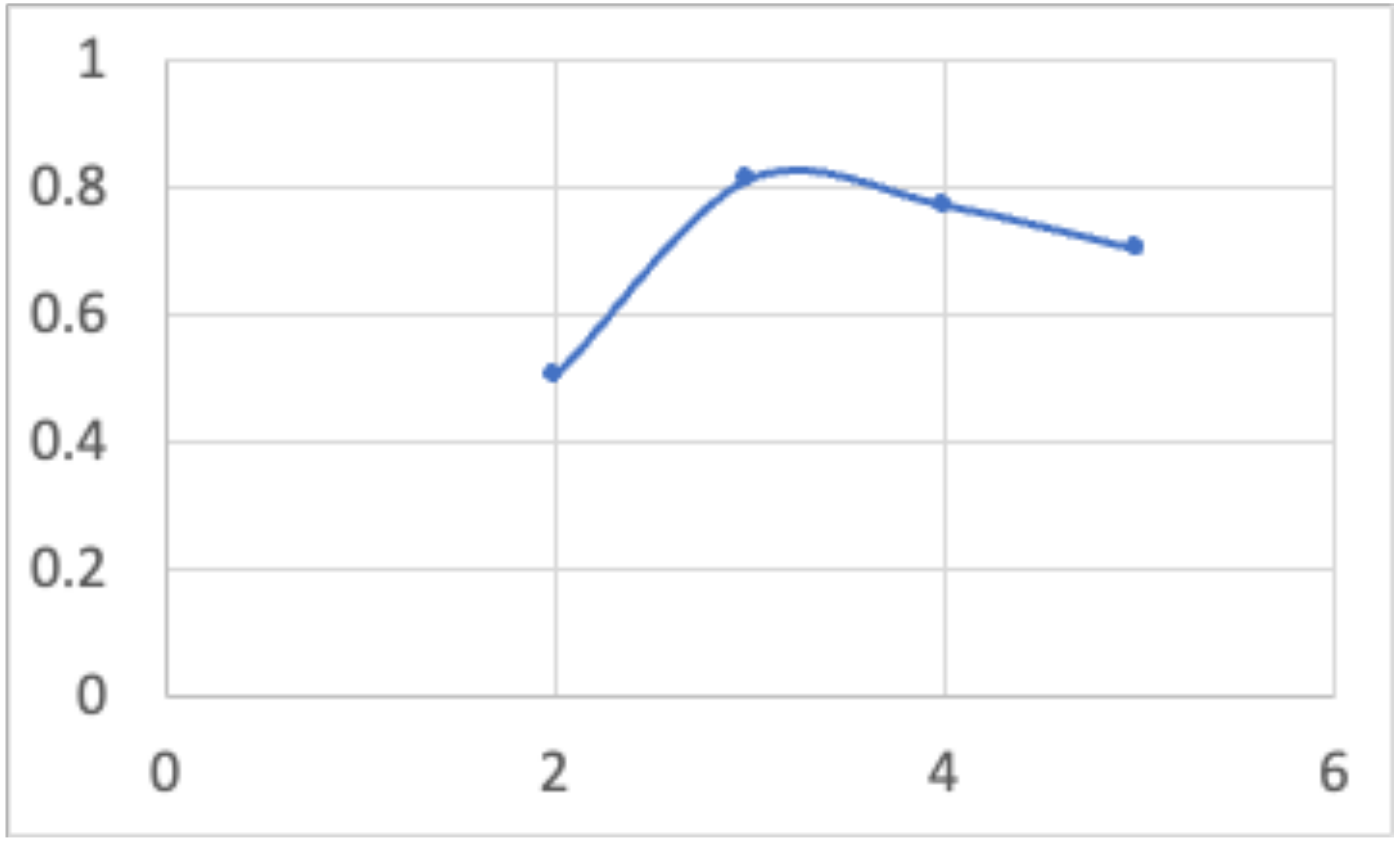


Utilization

hash functions

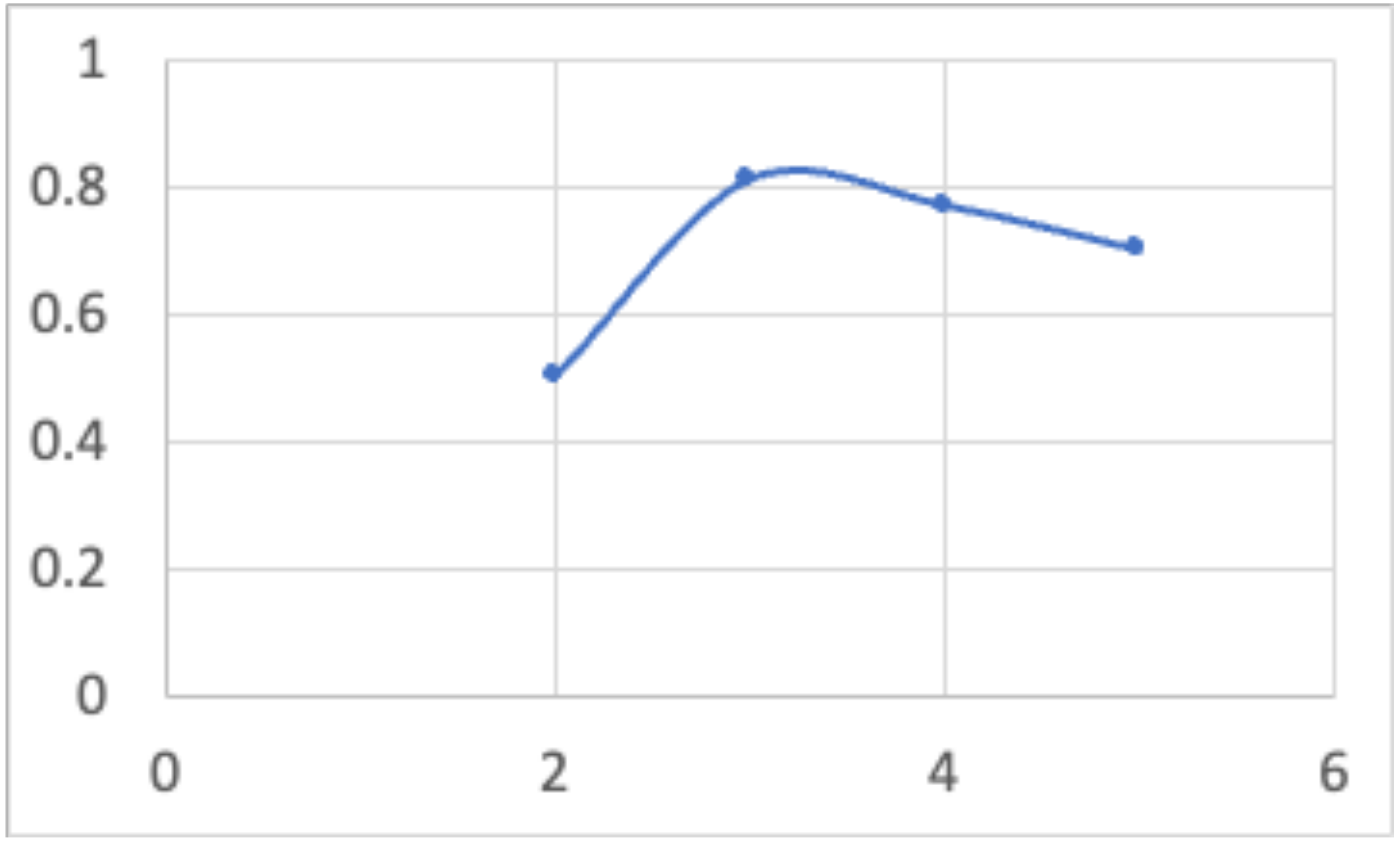
Optimal 0.81

Utilization



hash functions

Optimal 0.81

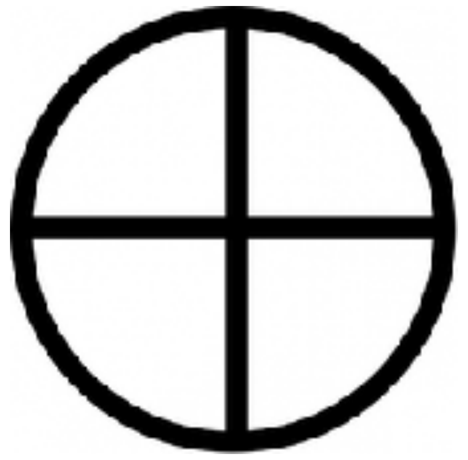


Similar to finding the optimal # hash functions with Bloom filters :)

Bloom

$\approx 2^{-M/N \cdot 0.69}$

XOR



$\boldsymbol{\approx 2^{-M/N \cdot 0.81}}$

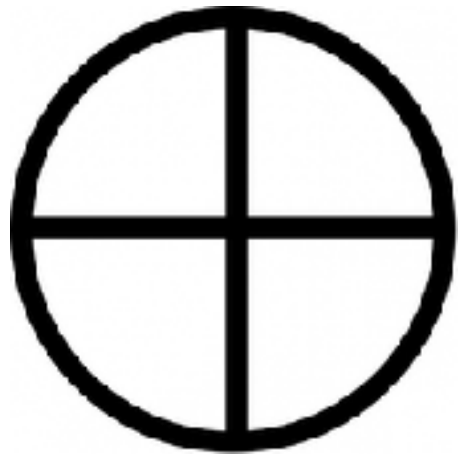
Idealized

$\approx 2^{-M/N}$

Bloom

$\approx 2^{-M/N \cdot 0.69}$

XOR



$\approx 2^{-M/N \cdot 0.81}$

Ribbon

$\boldsymbol{\approx 2^{-M/N \cdot 0.92}}$

Denser XOR filter

Idealized

$\approx 2^{-M/N}$

Bloom	XOR	**Ribbon**	Idealized
$\approx 2^{-M/N \cdot 0.69}$	$\approx 2^{-M/N \cdot 0.81}$	$\boldsymbol{\approx 2^{-M/N \cdot 0.92}}$	$\approx 2^{-M/N}$

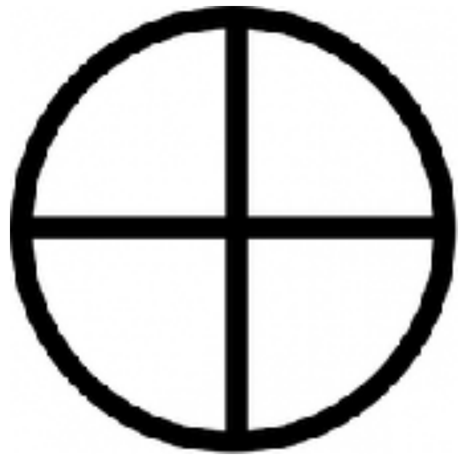
Denser XOR filter

In RocksDB since 2020

Bloom

$\approx 2^{-M/N \cdot 0.69}$

XOR

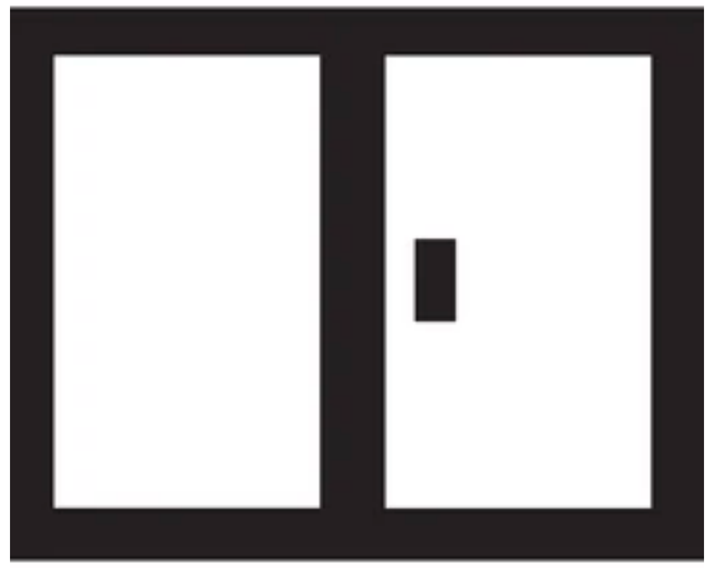


$\approx 2^{-M/N \cdot 0.81}$

Ribbon

$\approx 2^{-M/N \cdot 0.92}$

XOR filter w. Spatial Coupling



$\approx 2^{-M/N}$

Approach ideal

Operation Costs (in hash functions computed)

Construction =

Positive Query =

Avg. Negative Query =

Operation Costs (in hash functions computed)

 Construction = **O(N)**

 Positive Query =

 Avg. Negative Query =

Operation Costs (in hash functions computed)

Construction = O(N)

Positive Query = **3**

Avg. Negative Query = **3**

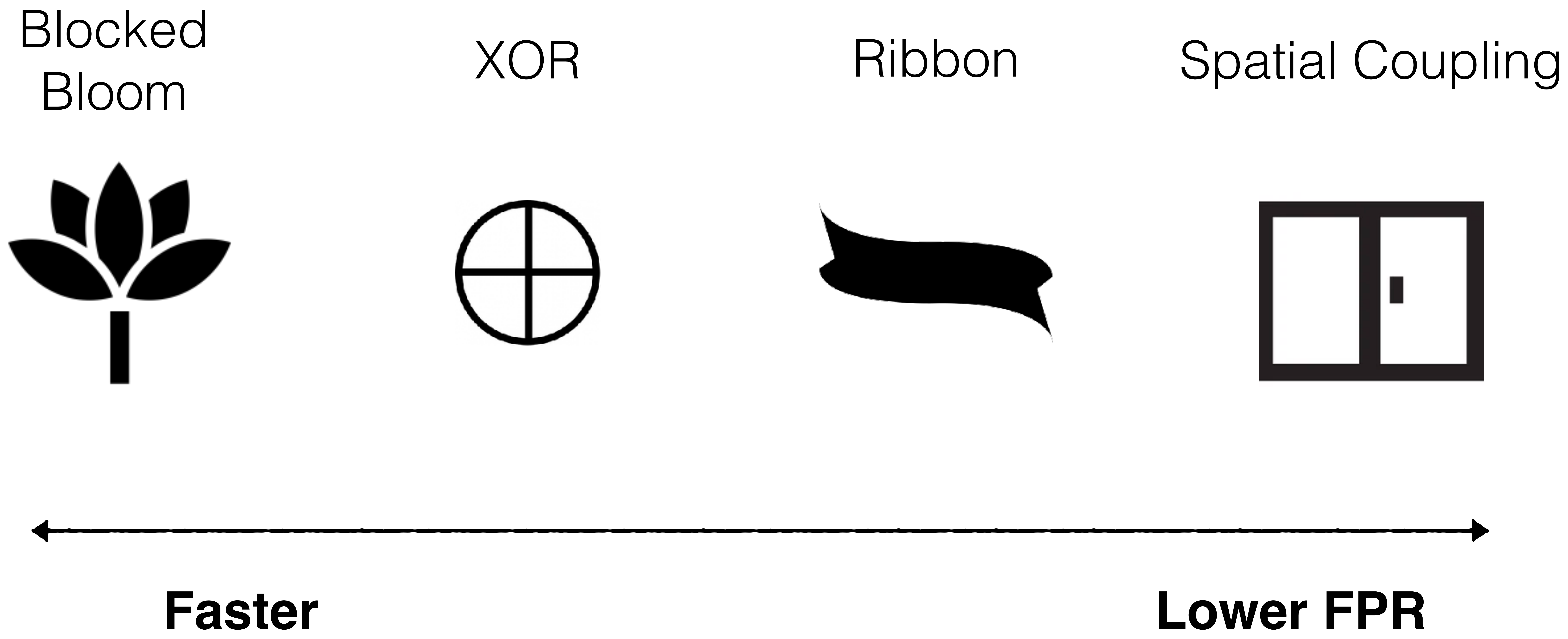
Operation Costs (in hash functions computed)

Construction = O(N)

Positive Query = **3**

Avg. Negative Query = **3**

Not as good as blocked Bloom filters



Blocked Bloom
XOR
Ribbon
Spatial Coupling
Faster
Lower FPR

And now: office hours :)