Data Structures and Algorithms

Hash tables 2 & Bloom Filters

CS 225

Brad Solomon

November 1, 2023

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Lots of extra credit projects to grade

Learning Objectives

Finish discussion of hash table ADT

Build a conceptual understanding of a bloom filter

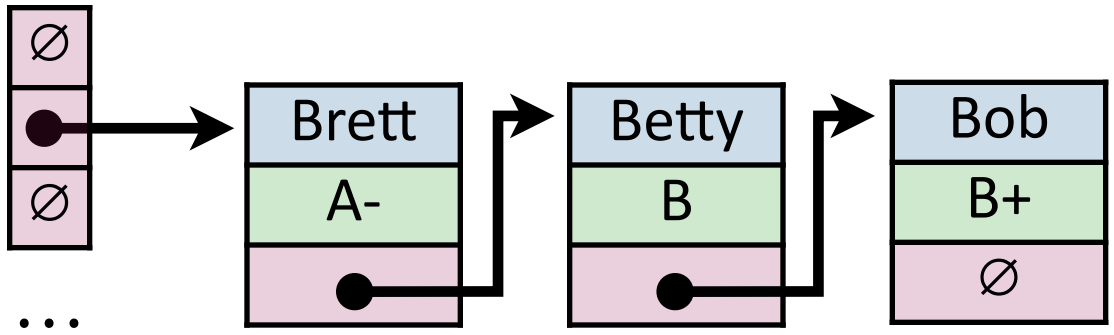
Review probabilistic data structures and one-sided error

Formalize the math behind the bloom filter

Open vs Closed Hashing

Addressing hash collisions depends on your storage structure.

- **Open Hashing:** store *k,v* pairs externally



CL
↳ infinite size!

- **Closed Hashing:** store *k,v* pairs in the hash table

fixed size

K, V
K, V
K, V

look for next available

Collision Handling: Double Hashing

S = { 16, 8, 4, 13, 29, 11, 22 }

$h_1(k) = k \% 7$

$h_2(k) = 5 - (k \% 5)$

|S| = n

|Array| = m

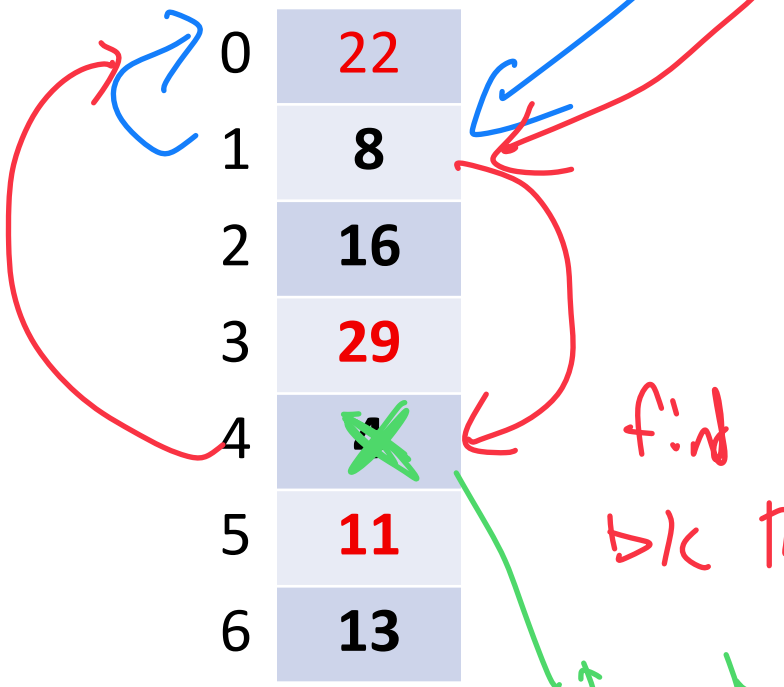
$h(k, i) = (h_1(k) + i*h_2(k)) \% 7$

Try $h(k) = (k + 0*h_2(k)) \% 7$, if full...

Try $h(k) = (k + 1*h_2(k)) \% 7$, if full...

Try $h(k) = (k + 2*h_2(k)) \% 7$, if full...

Try ...



find works b/c this is deterministic!

Tombstone! Record that Something was here

22 % 7 = 1

1

3

22

1

1 + 3 = 4

1 + 6 = 7 = 0

Running Times

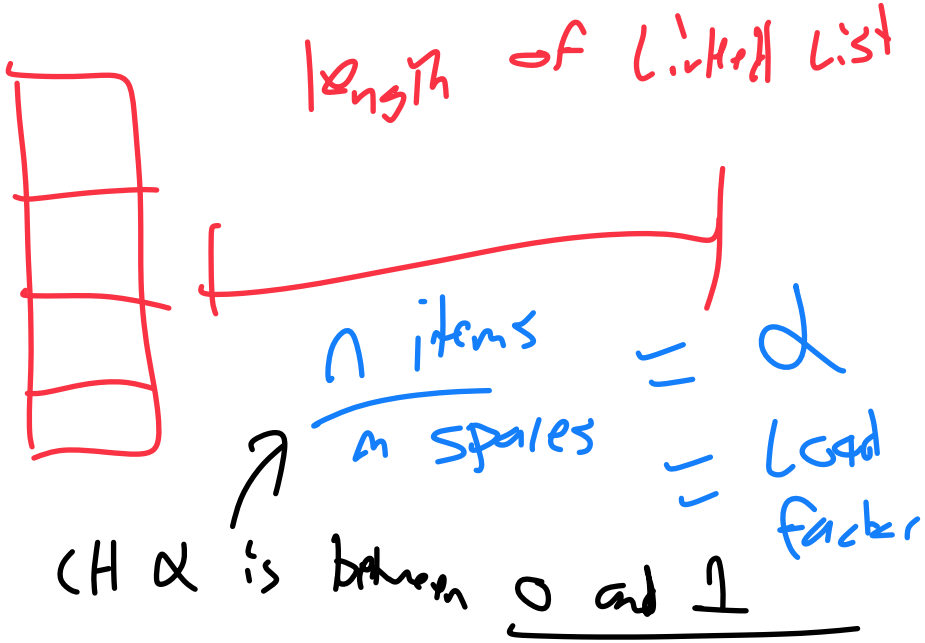
(Expectation under SUHA)

(Don't memorize these equations, no need.)

Open Hashing:

insert: 1.

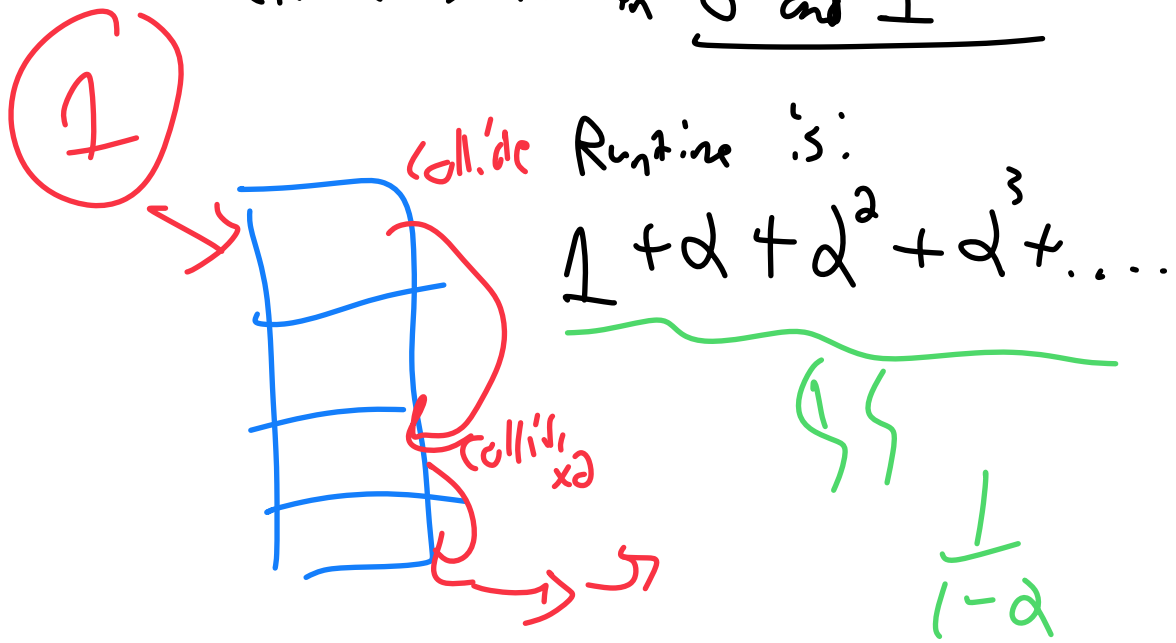
find/ remove: $1 + \alpha$.



Closed Hashing:

insert: $\frac{1}{1-\alpha}$.

find/ remove: $\frac{1}{1-\alpha}$.



Running Times *(Don't memorize these equations, no need.)*

The expected number of probes for find(key) under SUHA

Linear Probing:

- Successful: **½(1 + 1/(1-α))**
- Unsuccessful: **½(1 + 1/(1-α))²**

Double Hashing:

- Successful: **1/α * ln(1/(1-α))**
- Unsuccessful: **1/(1-α)** ← 1/1-1 = 1/0 !!

Separate Chaining:

- Successful: **1 + α/2**
- Unsuccessful: **1 + α**

OH: $\alpha \to \infty$

CH: $0 \leq \alpha < 1$ (Don't let it be 1)

Instead, observe:

- As α increases:

Search time → ∞

- If α is constant:

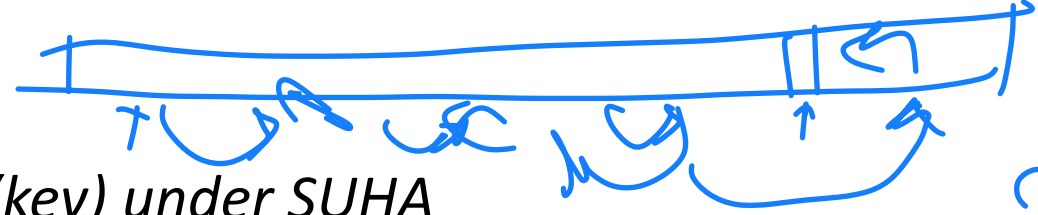
↳ Runtime is constant

IF $n = 100$, $m > 100$

$m = 200$:)

Running Times

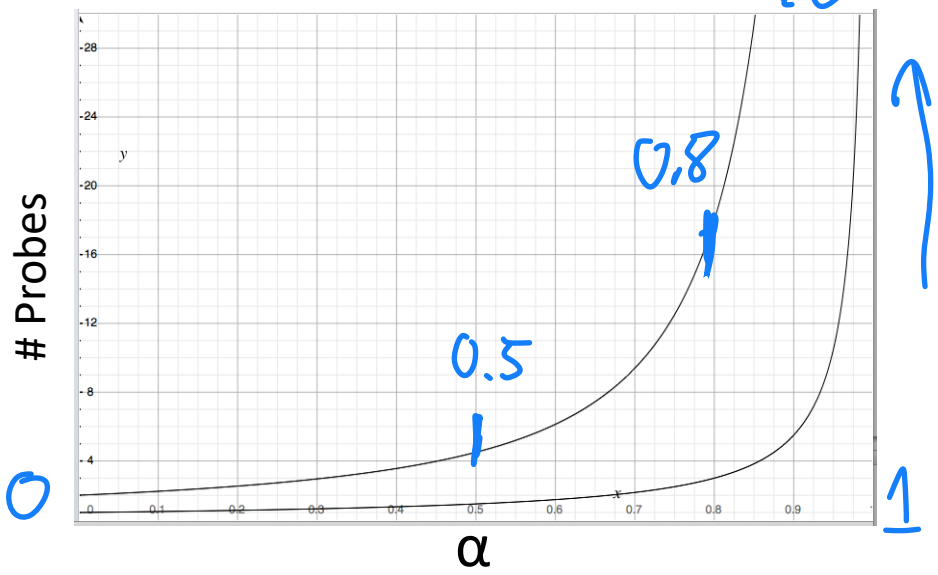
The expected number of probes for find(key) under SUHA

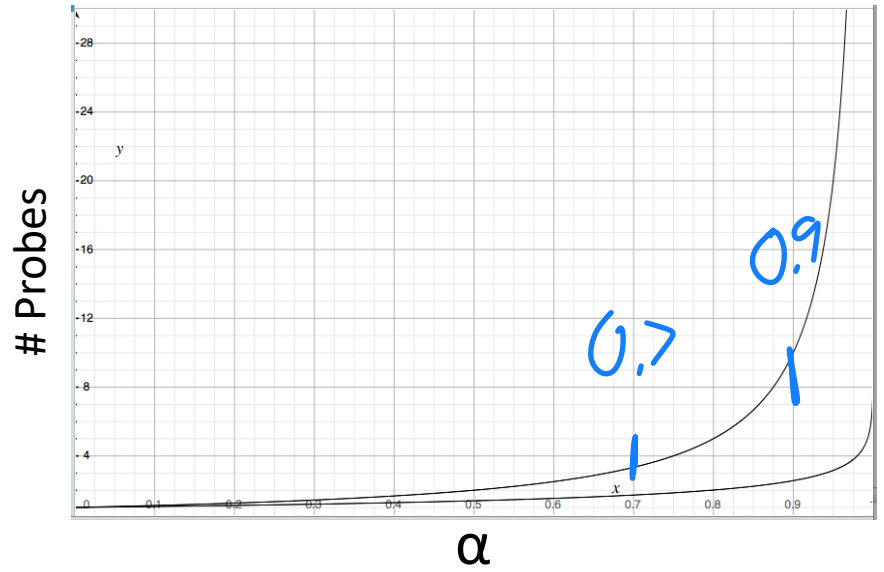


- **Linear Probing:**
  - Successful: **½(1 + 1/(1-α))**
  - Unsuccessful: **½(1 + 1/(1-α))²**

- **Double Hashing:**
  - Successful: **1/α * ln(1/(1-α))**
  - Unsuccessful: **1/(1-α)**

When do we resize?





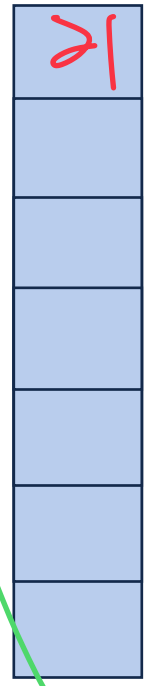
Resizing a hash table

How do you resize?

1) When we resize we have to reinsert every item

H%7

21 %7 = 0

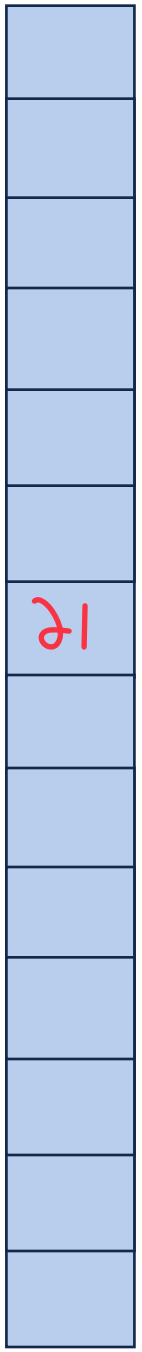


hash! compression

↓ ↓

H%14

21%14 @ 7 →



Pseudo-amortized

1) We resize when $\alpha < 1$

2) Universal SUHA

3) Expectation

↓

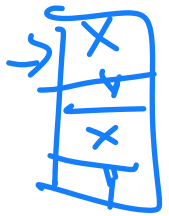
$O(1)^{***}$

h1 = x

h2 = d

h2 has not factor w/ m

x is odd



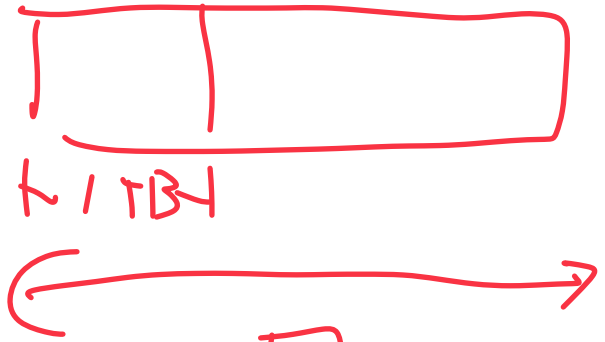
m = 4 x is even

Slope tangent exception

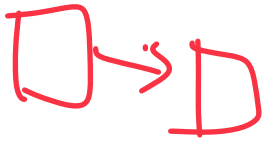
0 + 1 + 2 + 3 + 4 . . . I

Which collision resolution strategy is better?

- Big Records: Open hashing (Separate chaining)
- Structure Speed: Closed hashing



BTree! like



What structure do hash tables implement?

Dictionary! (key, value)

What constraint exists on hashing that doesn't exist with BSTs?

↳ Amortized, probabilistic (expectation), SUHA → Faster in practice!

Why talk about BSTs at all?

↳ An ordered data set is better in tree | guaranteed performance | nearest neighbor

↳ min & max

Running Times

log 2

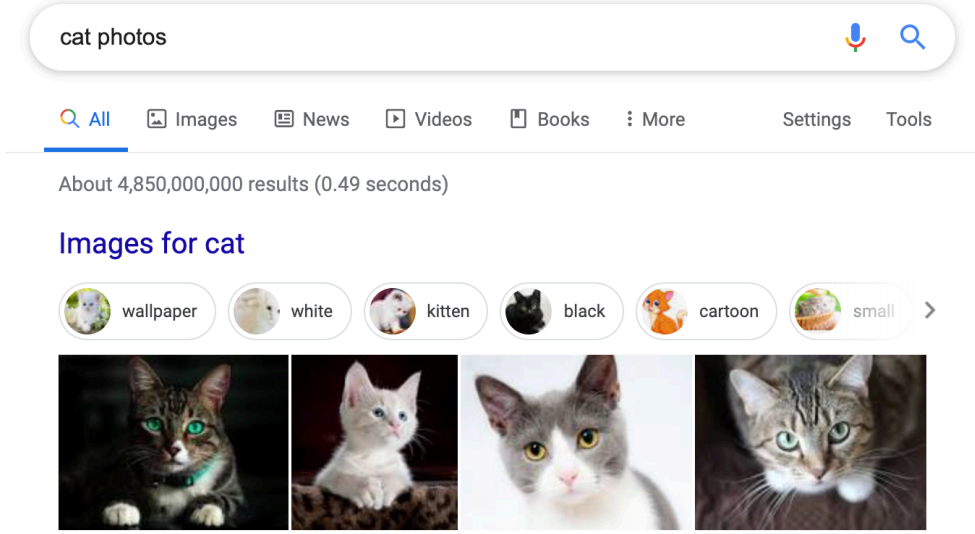
	Hash Table	AVL	Linked List
Find	Expectation*: $O(1)$ Worst Case: $O(n)$	$O(\log n)$	$O(n)$
Insert	Expectation*: $O(1)$ Worst Case: $O(n)$	$O(\log n)$	$O(1)$
Storage Space	$O(m) \sim O(n)$	$O(n)$	$O(n)$

control this

Memory-Constrained Data Structures

What method would you use to build a search index on a collection of objects *in a memory-constrained environment*?

Constrained by Big Data (Large N)



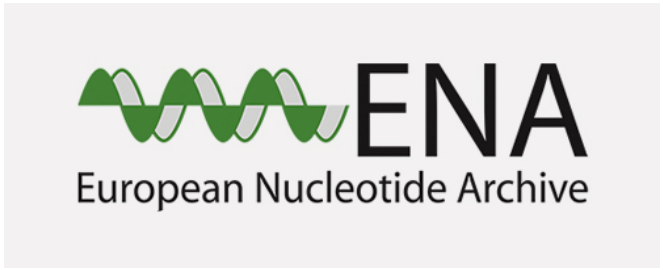
Google Index Estimate: >60 billion webpages

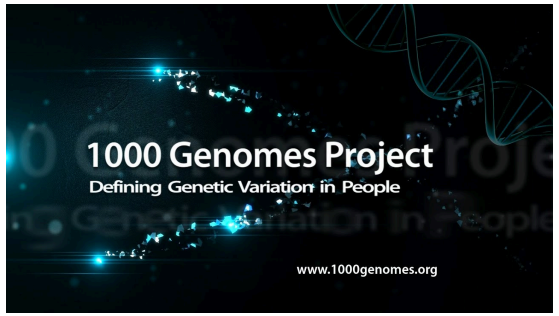
Google Universe Estimate (2013): >130 trillion webpages

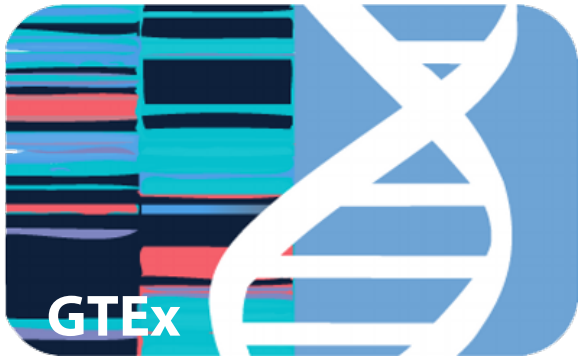
Memory-Constrained Data Structures

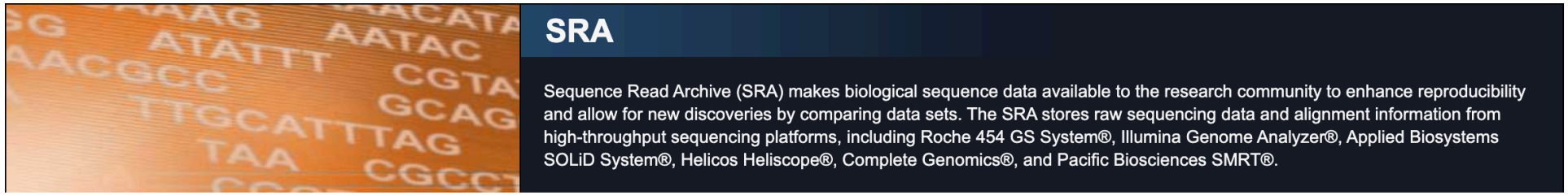
What method would you use to build a search index on a collection of objects *in a memory-constrained environment*?

Constrained by Big Data (Large N)









Sequence Read Archive Size: >60 petabases (10^{15})

Memory-Constrained Data Structures

What method would you use to build a search index on a collection of objects *in a memory-constrained environment*?

Constrained by Big Data (Large N)

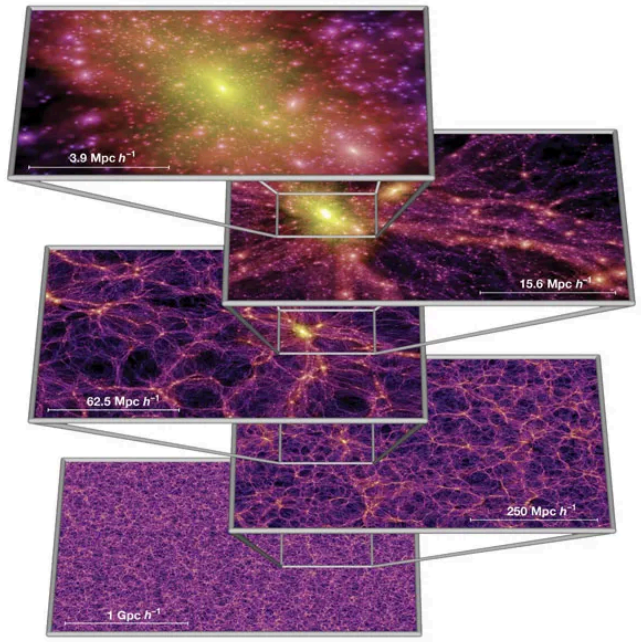


Image: https://doi.org/10.1038/nature03597

Sky Survey Projects	Data Volume
DPOSS (The Palomar Digital Sky Survey)	3 TB
2MASS (The Two Micron All-Sky Survey)	10 TB
GBT (Green Bank Telescope)	20 PB
GALEX (The Galaxy Evolution Explorer)	30 TB
SDSS (The Sloan Digital Sky Survey)	40 TB
SkyMapper Southern Sky Survey	500 TB
PanSTARRS (The Panoramic Survey Telescope and Rapid Response System)	~ 40 PB expected
LSST (The Large Synoptic Survey Telescope)	~ 200 PB expected
SKA (The Square Kilometer Array)	~ 4.6 EB expected

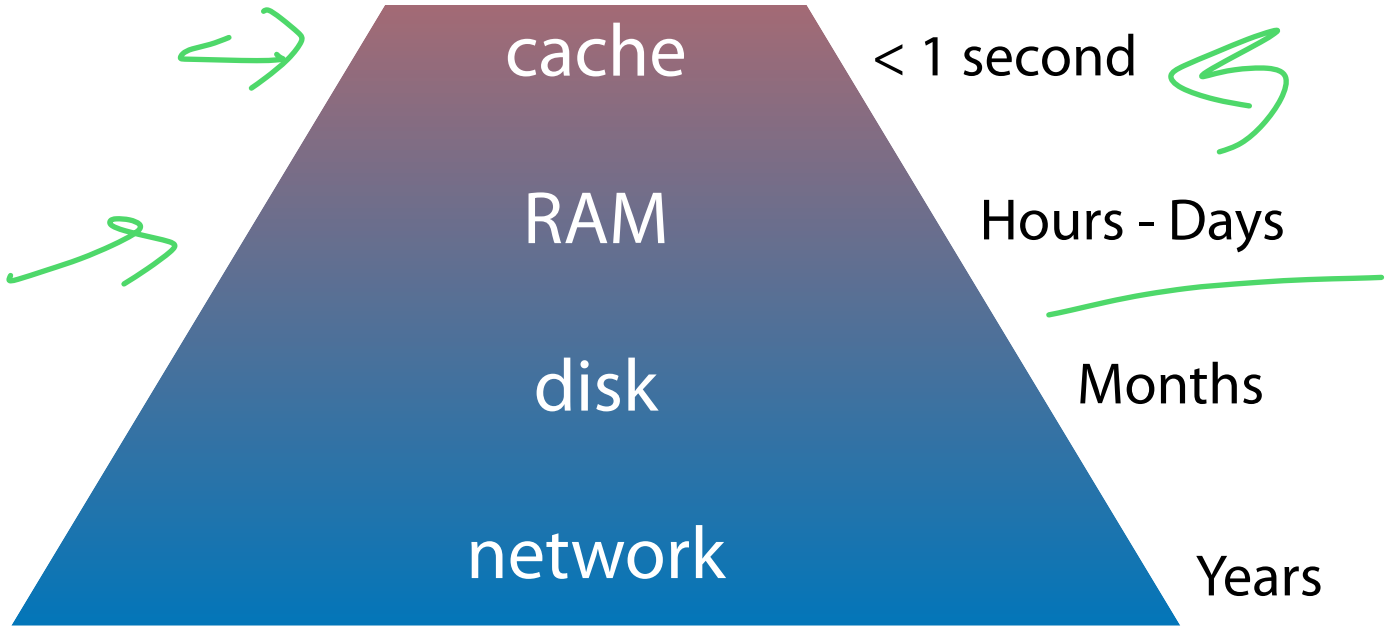
Table: http://doi.org/10.5334/dsj-2015-011

Estimated total volume of one array: 4.6 EB

Memory-Constrained Data Structures

What method would you use to build a search index on a collection of objects *in a memory-constrained environment*?

Constrained by resource limitations



(Estimates are Time x 1 billion courtesy of https://gist.github.com/hellerbarde/2843375)

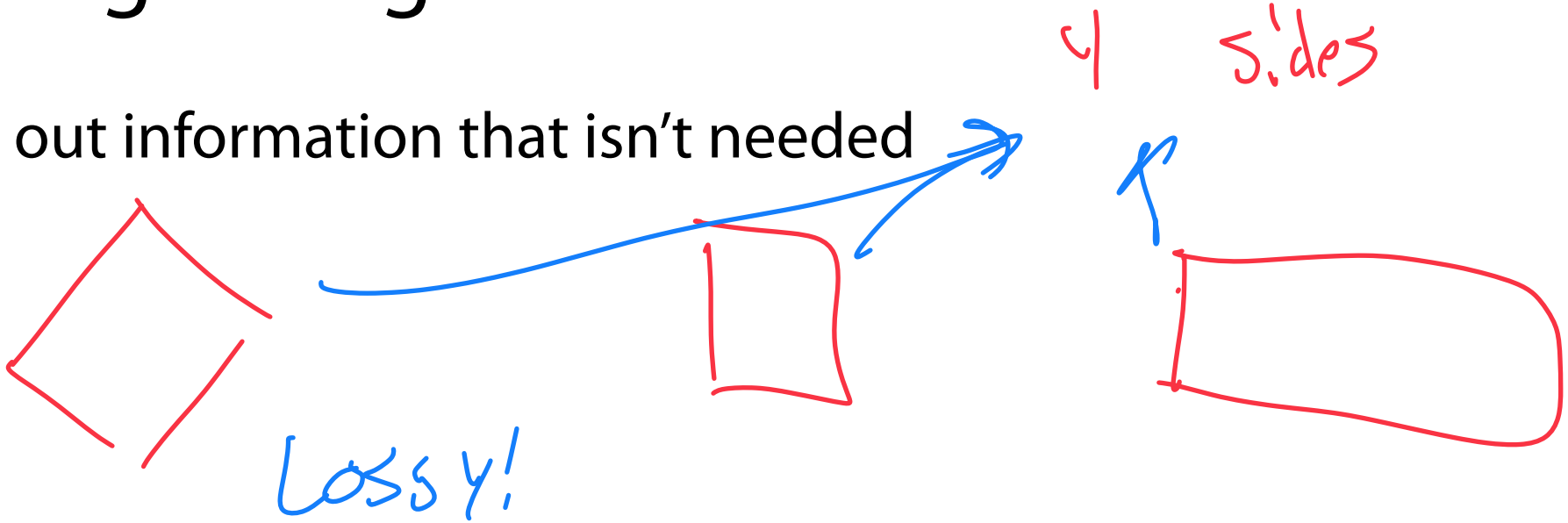
Memory-Constrained Data Structures

What method would you use to build a search index on a collection of objects *in a memory-constrained environment*?

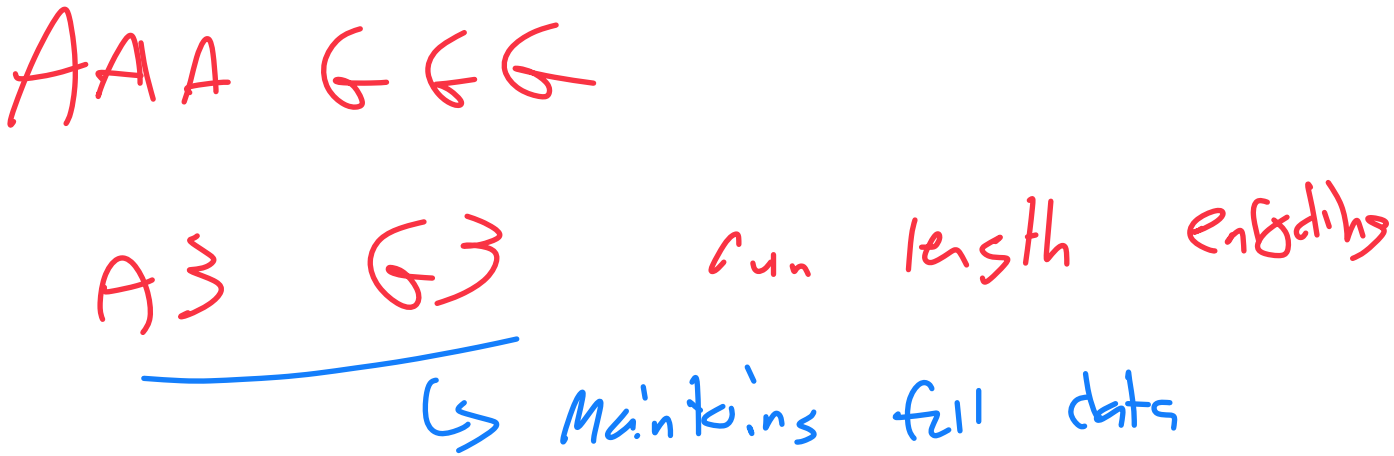
use least recently used → track? [Analyze part at a time]

Reducing storage costs

1) Throw out information that isn't needed



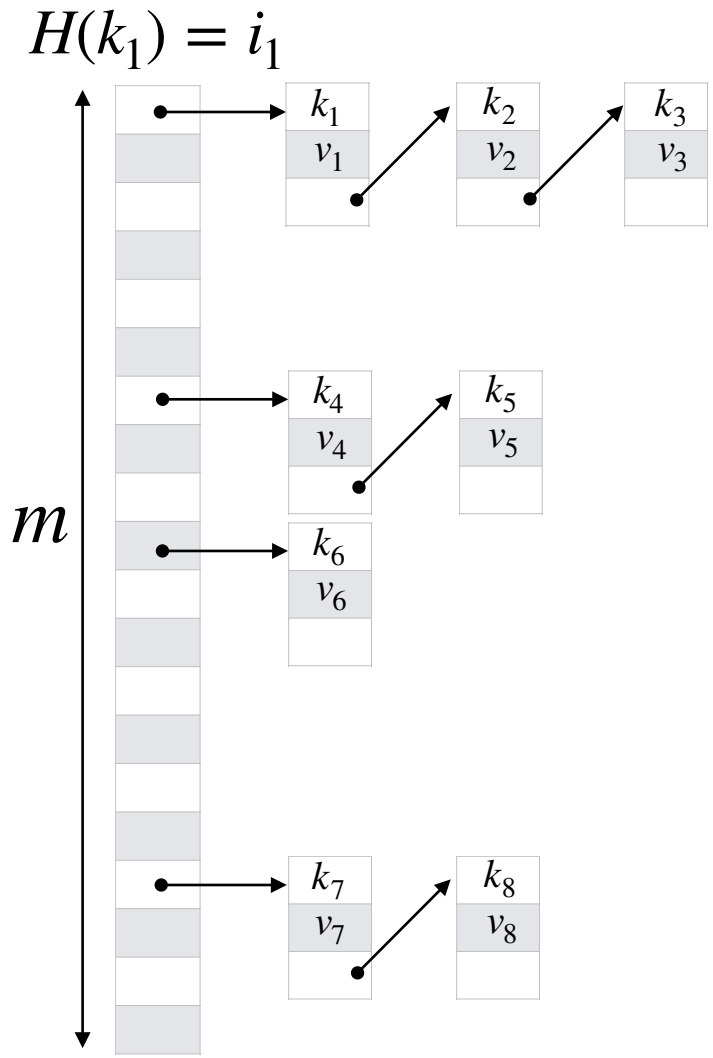
2) Compress the dataset



Reducing a hash table

Dictionary : K, V pair

What can we remove from a hash table?

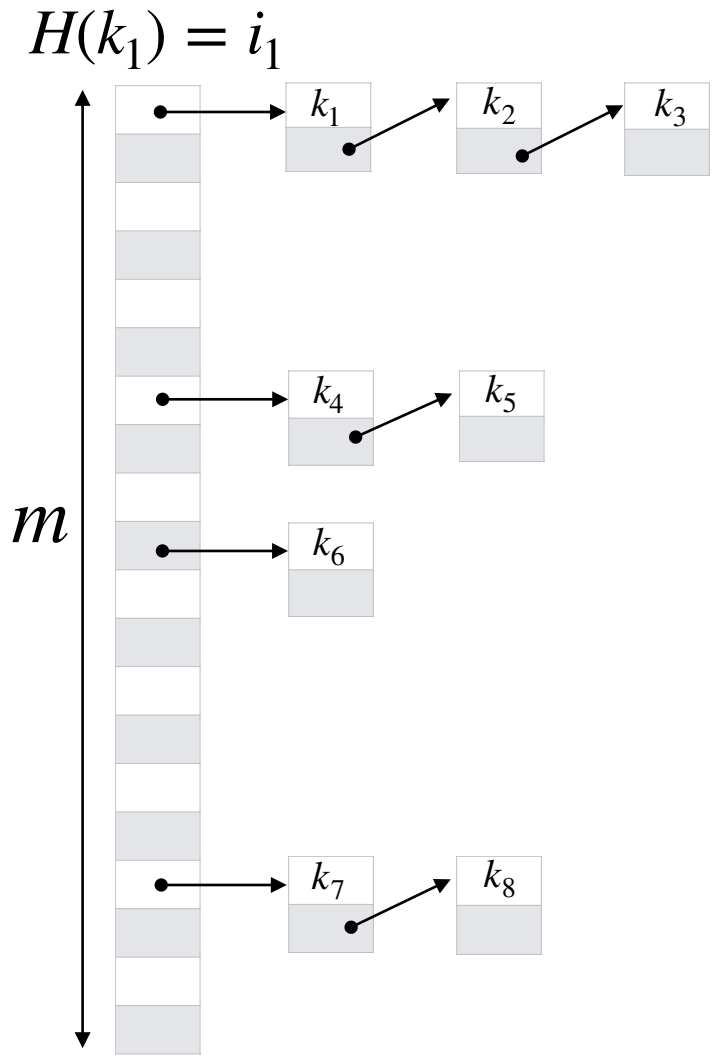


Reducing a hash table

Keys are hash set

What can we remove from a hash table?

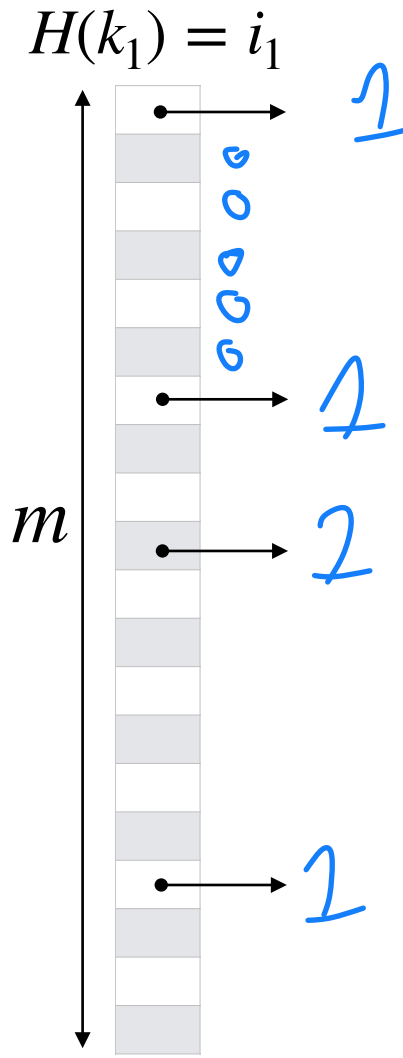
Take away values



Reducing a hash table

What can we remove from a hash table?

Take away values and keys



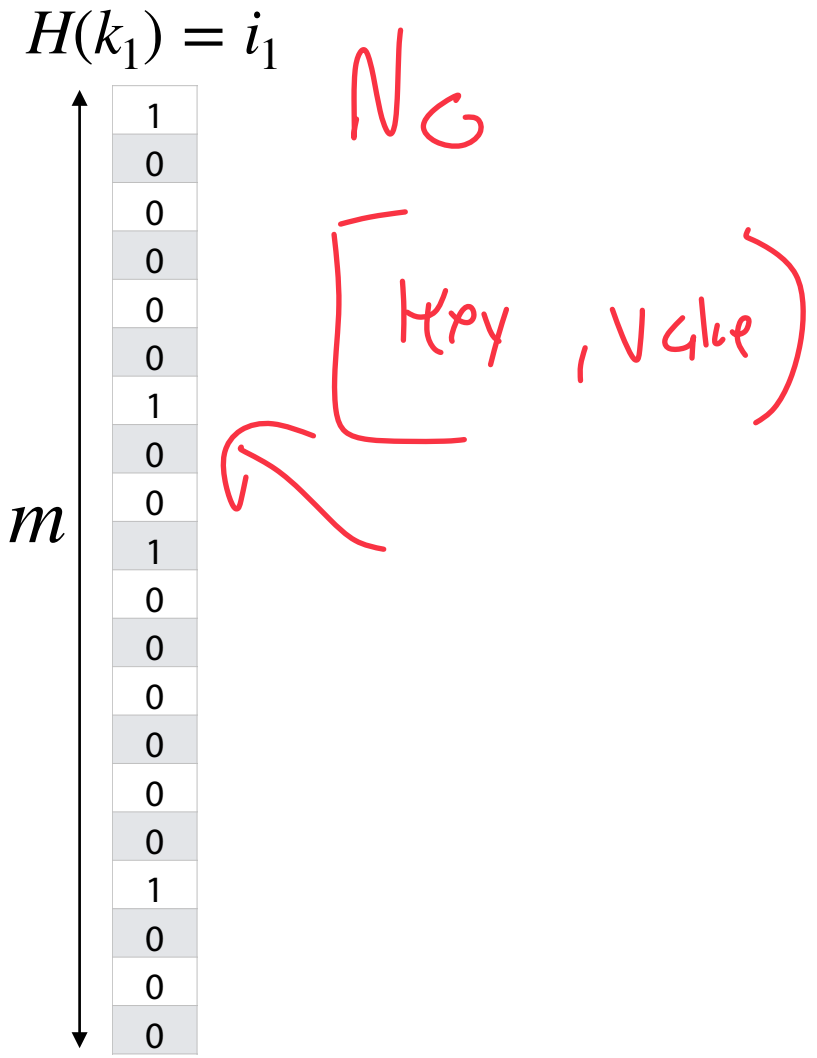
Approximate hash set

Reducing a hash table

What can we remove from a hash table?

Take away values and keys

This is a ***bloom filter***



Bloom Filter ADT

Constructor (Hash function (s) / One or more) , bit vector size m (bits) / array size m → buckets 1 bit each

Insert (Value / data)

Delete is missing!

Find ← Is probabilistic on existence (present / absent)

Bloom Filter: Insertion

O(1)

S = { 16, 8, 4, 13, 29, 11, 22 }

h(k) = k % 7

11 % 7 = 4

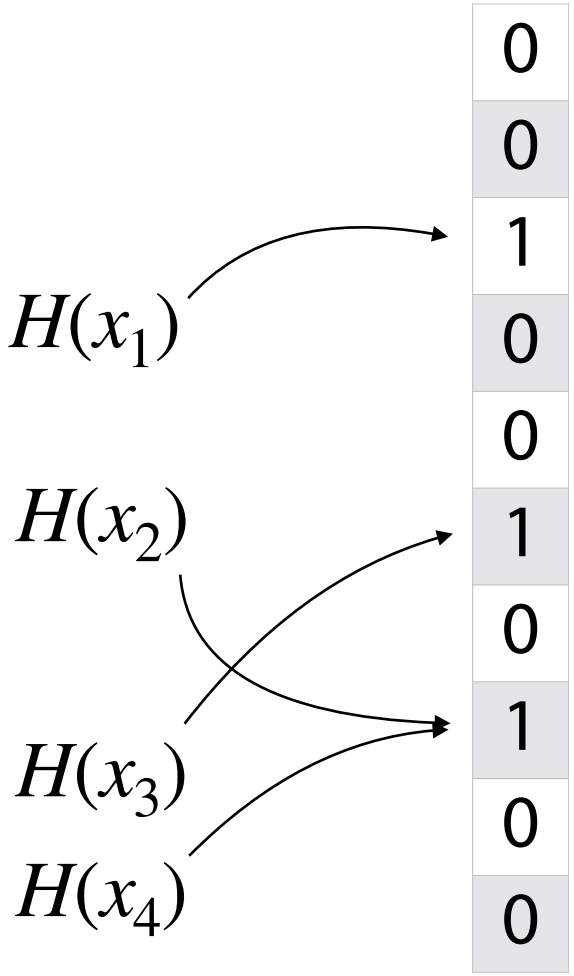
0	0
1	~~0~~ 1
2	~~0~~ 1
3	0
4	~~0~~ 1
5	0
6	~~0~~ 1

No collisions here! Bit stays 1

Bloom Filter: Insertion

An item is inserted into a bloom filter by hashing and then setting the hash-valued bit to 1

If the bit was already one, it stays 1



Bloom Filter: Deletion

S = { 16, 8, 4, 13, 29, 11, 22 }

h(k) = k % 7

0	0
1	~~1~~ 0
2	1
3	0
4	1
5	0
6	~~1~~ 0

_delete(13)

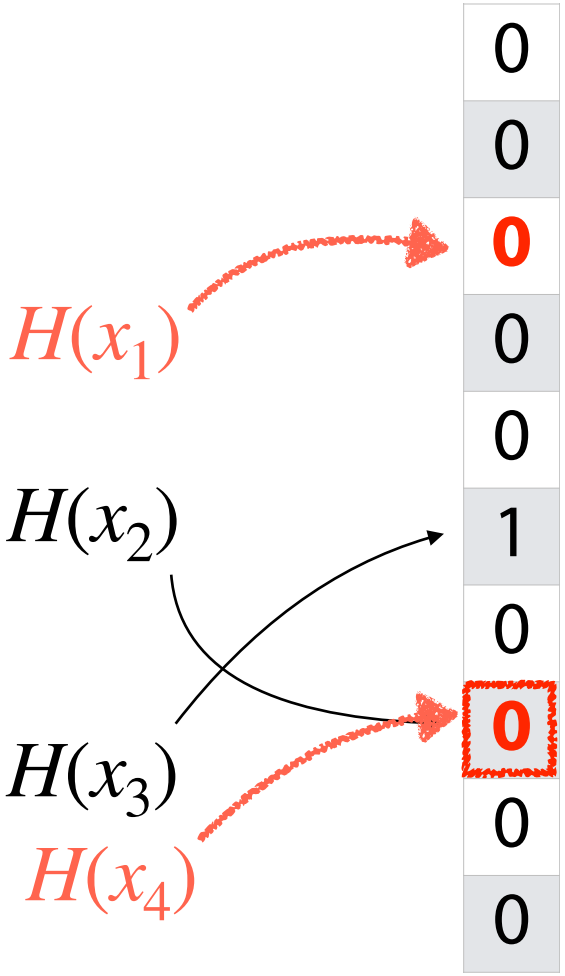
We cannot delete b/c we don't know what was inserted!

_delete(29)

find (8)
↳ return false;

Bloom Filter: Deletion

Due to hash collisions and lack of information, items cannot be deleted!



Bloom Filter: Search

S = { 16, 8, 4, 13, 29, 11, 22 }

h(k) = k % 7

0	0
1	1
2	1
3	0
4	1
5	0
6	1

```
_find(16)
```

```
_find(20)
```

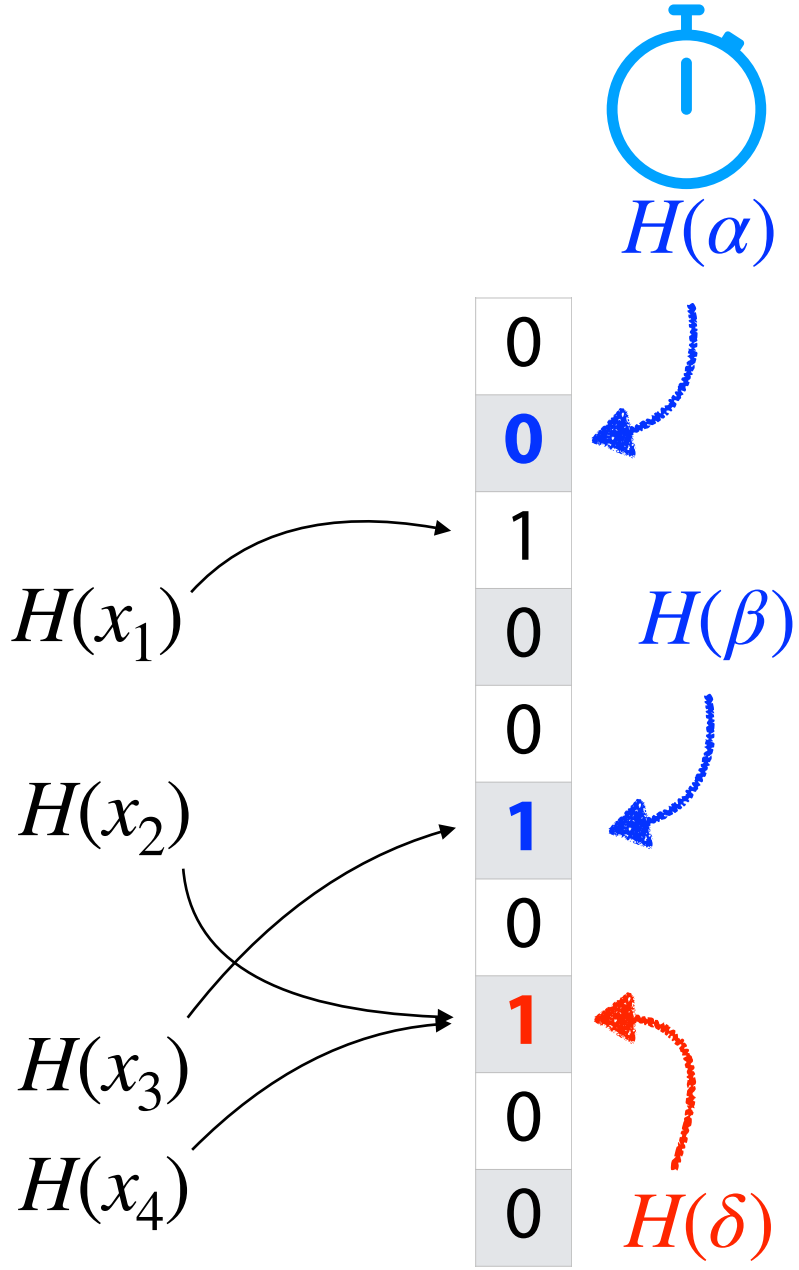
```
_find(3)
```

Bloom Filter: Search

The bloom filter is a *probabilistic* data structure!

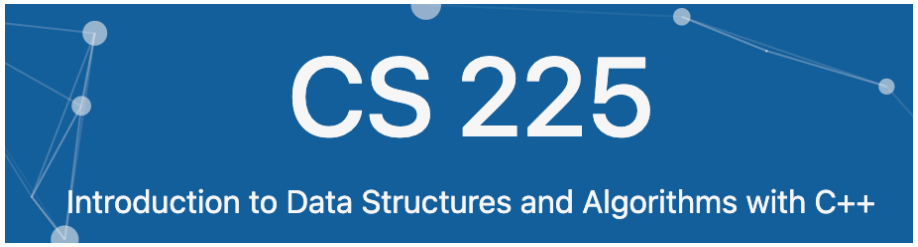
If the value in the BF is 0:

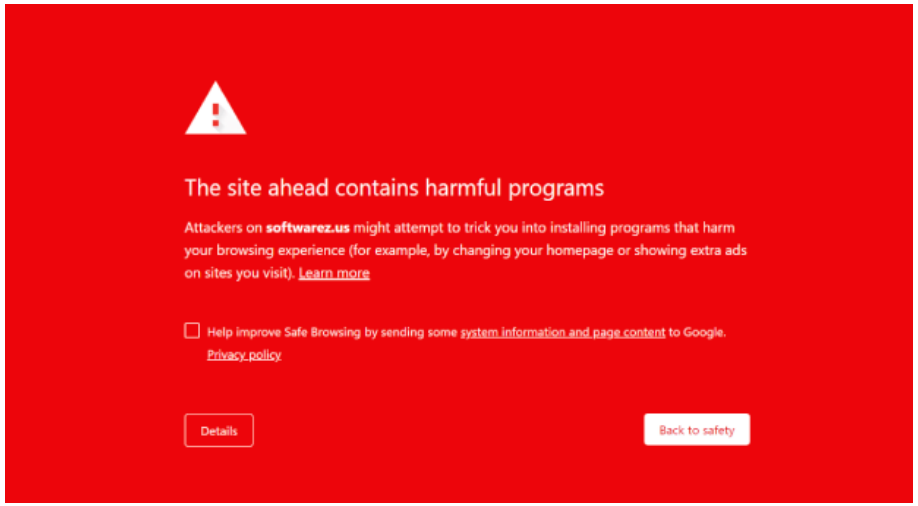
If the value in the BF is 1:

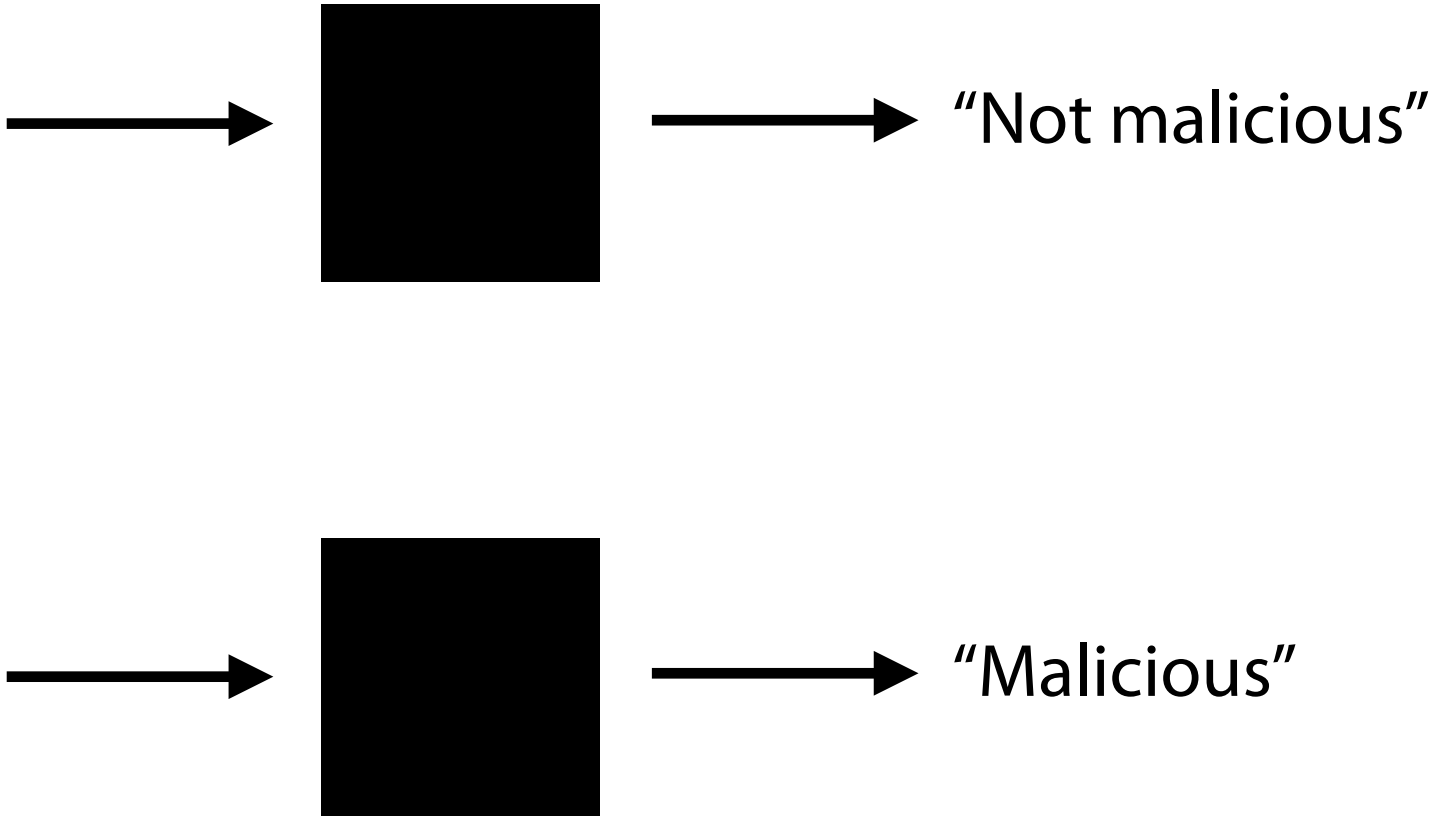


Probabilistic Accuracy: Malicious Websites

Imagine we have a detection oracle that identifies if a site is malicious







Probabilistic Accuracy: Malicious Websites

Imagine we have a detection oracle that identifies if a site is malicious

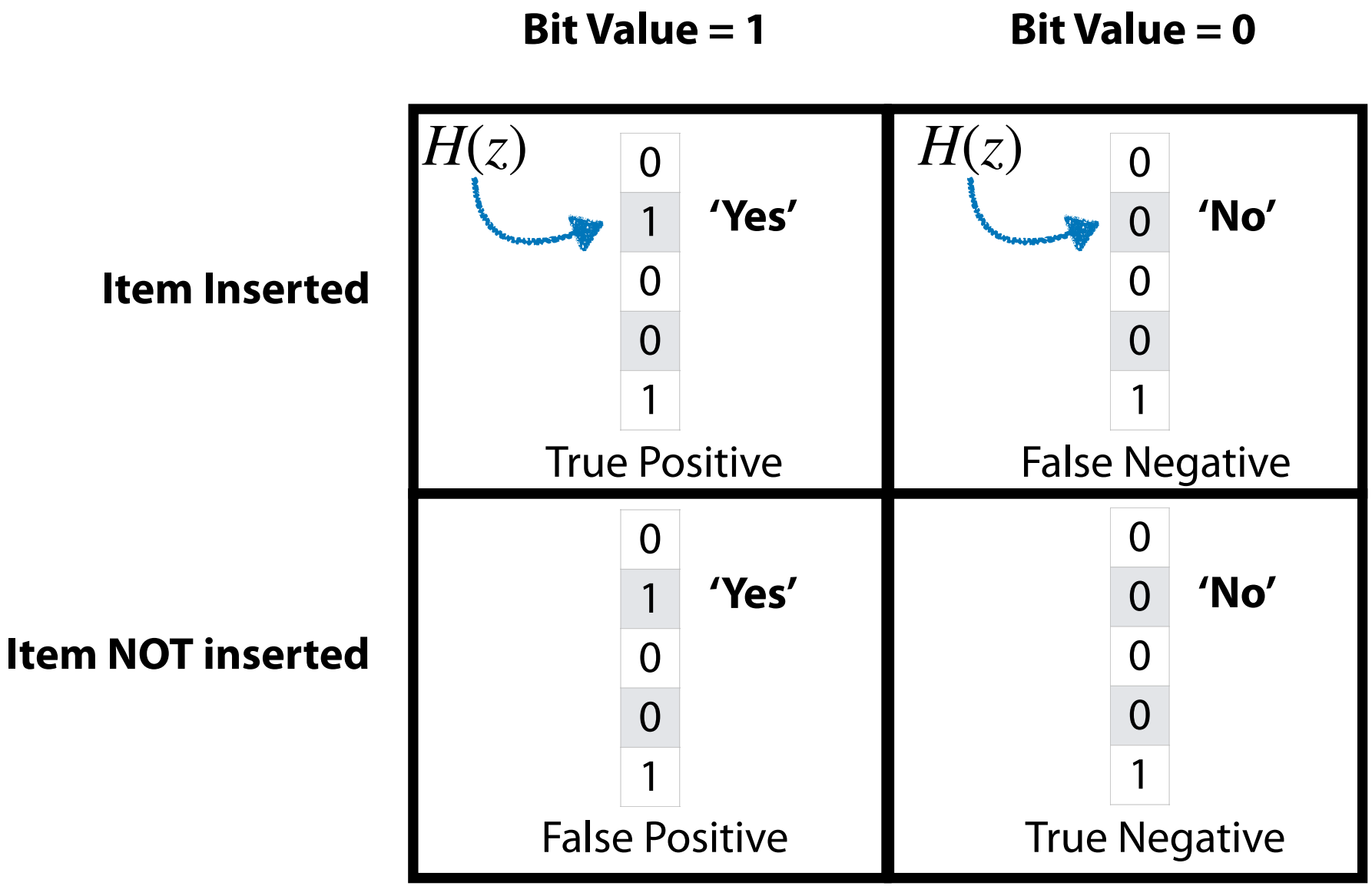
True Positive:

False Positive:

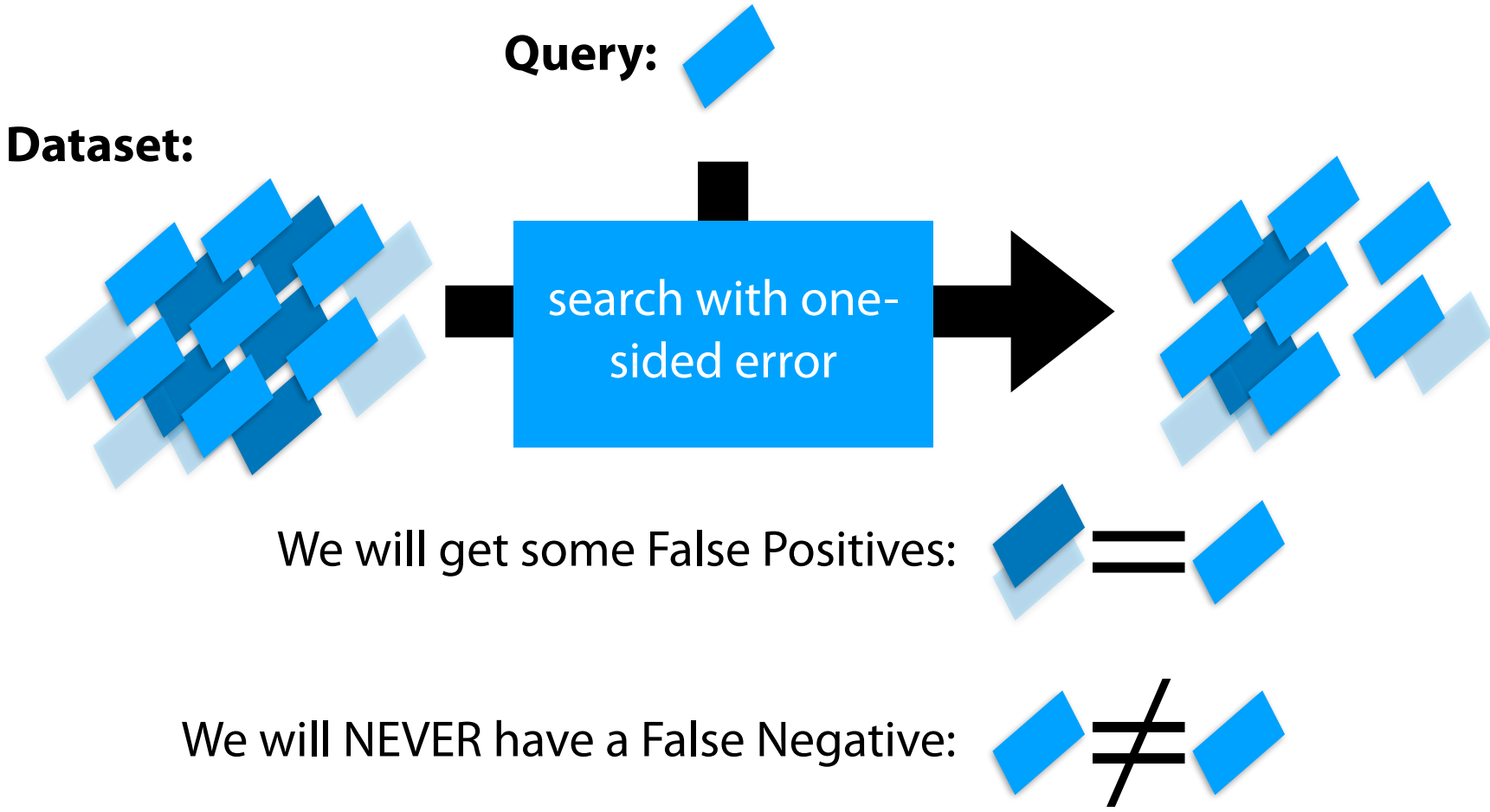
False Negative:

True Negative:

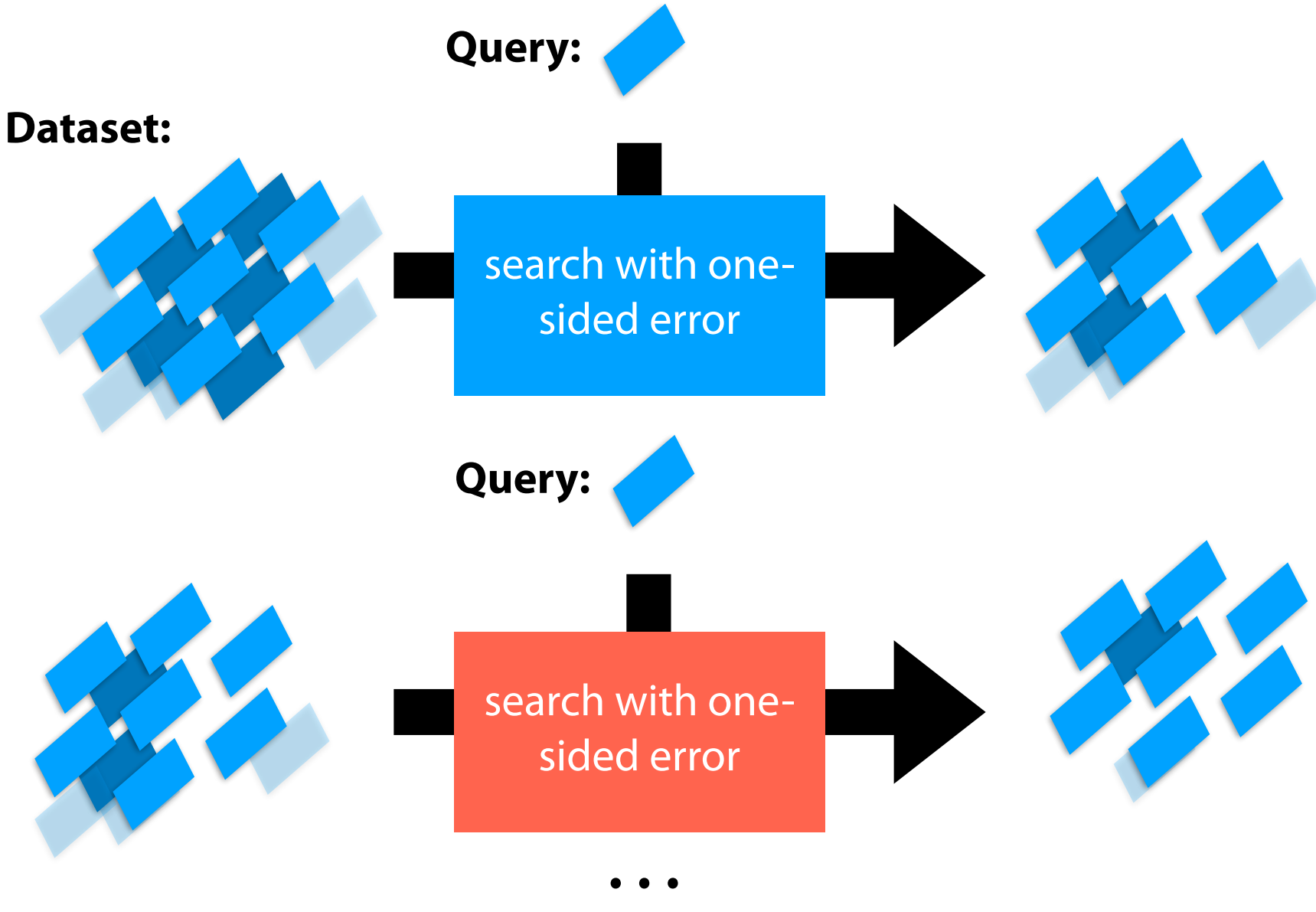
Imagine we have a **bloom filter** that **stores malicious sites…**



Probabilistic Accuracy: One-sided error



Probabilistic Accuracy: One-sided error

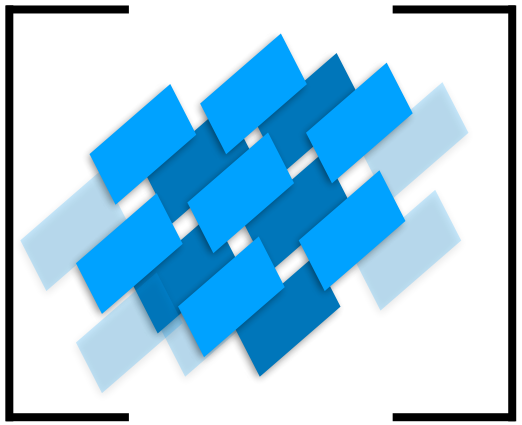


Bloom Filter: Repeated Trials

Use many hashes/filters; add each item to each filter

0
1
0
1
0
0
0
1
0
1
1
0
1
0
1
1
0
1
0
1

h_1



Bloom Filter: Repeated Trials

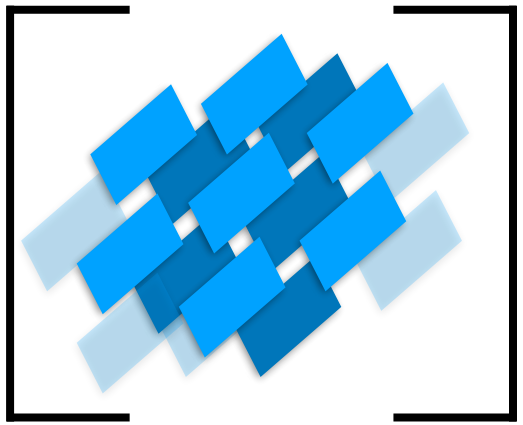
Use many hashes/filters; add each item to each filter

h_1

0
1
0
1
0
0
0
1
0
1
1
0
1
0
1
1
0
1
0
1

0
0
0
1
0
0
0
1
0
1
0
0
1
1
1
0
0
1
0
0

h_2



Bloom Filter: Repeated Trials

Use many hashes/filters; add each item to each filter

h_1

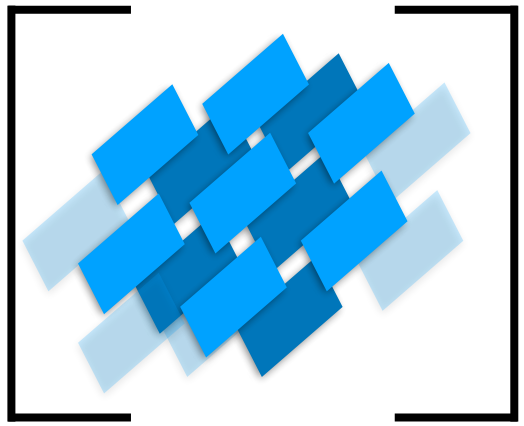
0
1
0
1
0
0
0
1
0
1
1
0
1
0
1
1
0
1
0
1

h_2

0
0
0
1
0
0
0
1
0
1
0
0
1
1
1
0
0
1
0
0

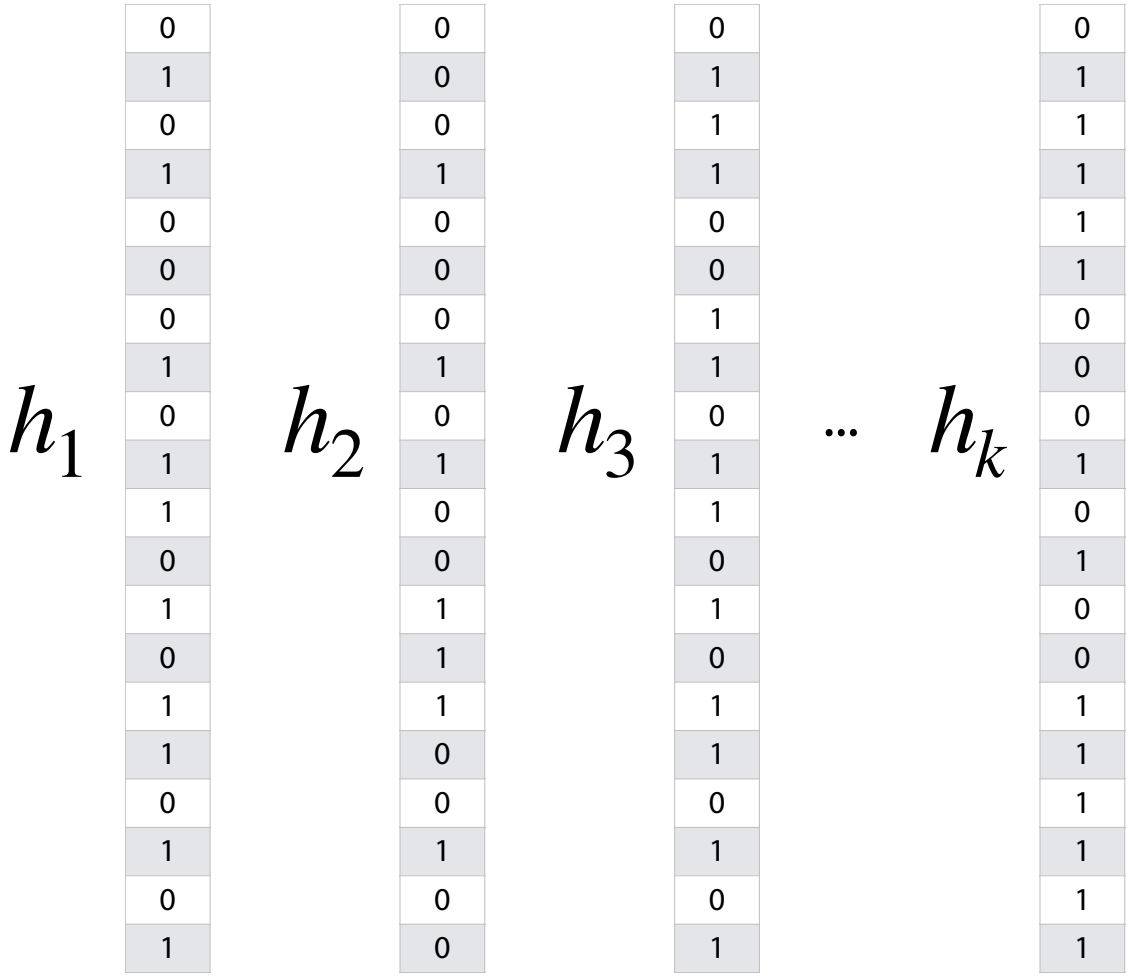
h_3

0
1
1
1
0
0
1
1
0
1
1
0
1
0
1
1
0
1
0
1

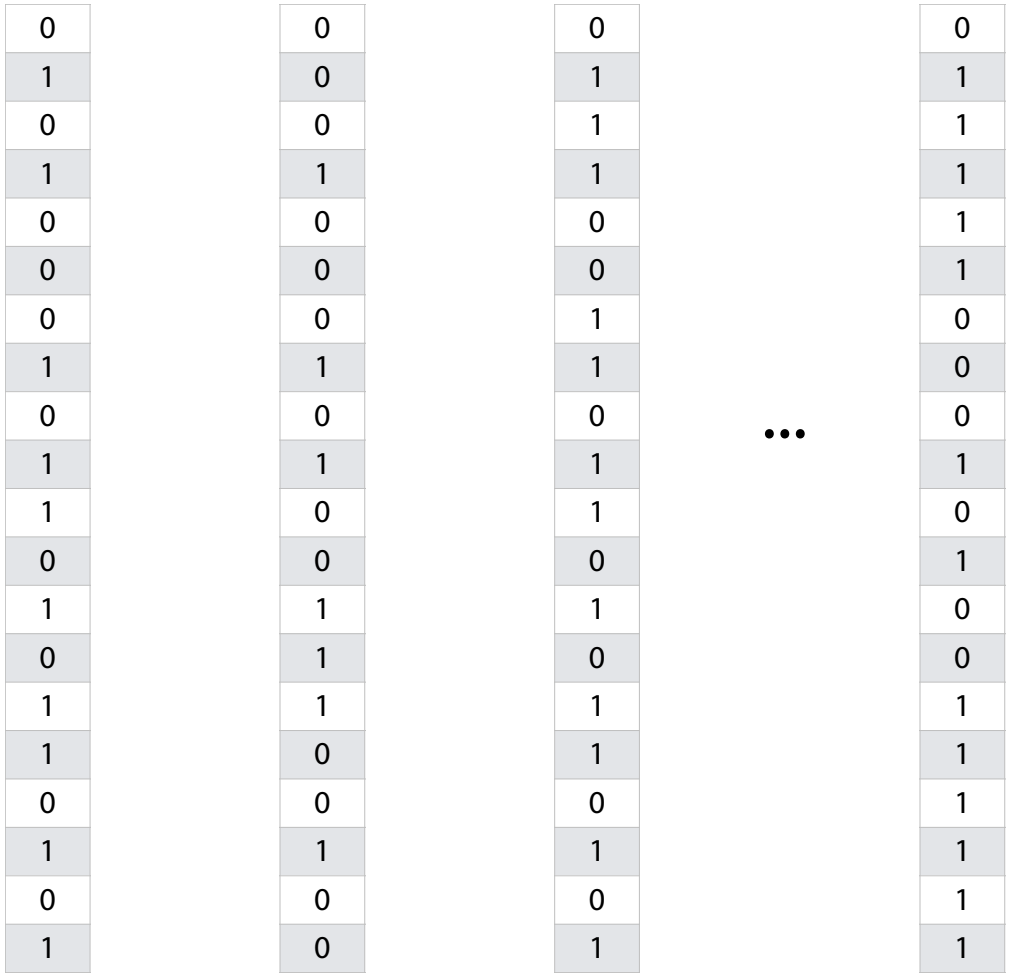


Bloom Filter: Repeated Trials

Use many hashes/filters; add each item to each filter

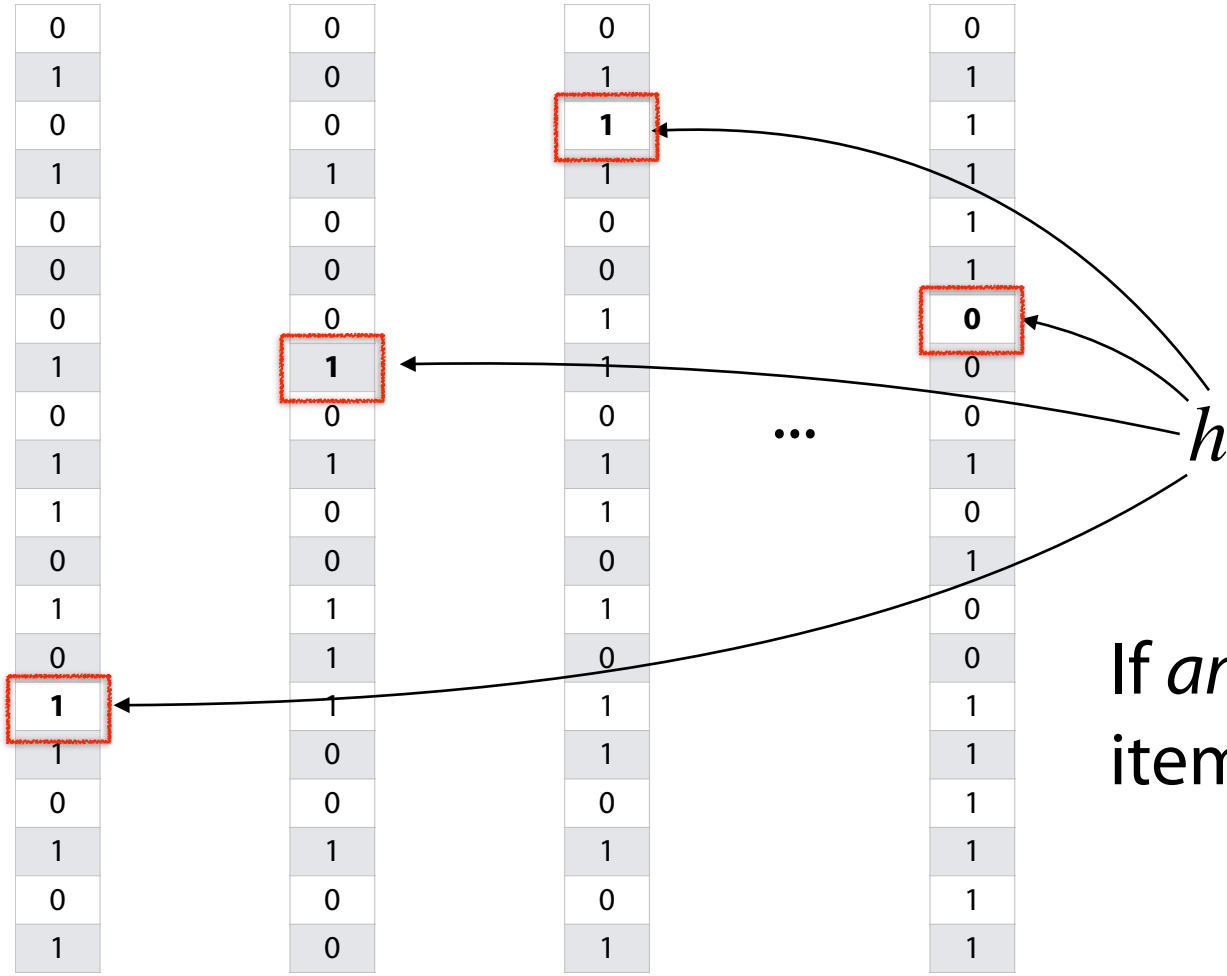


Bloom Filter: Repeated Trials



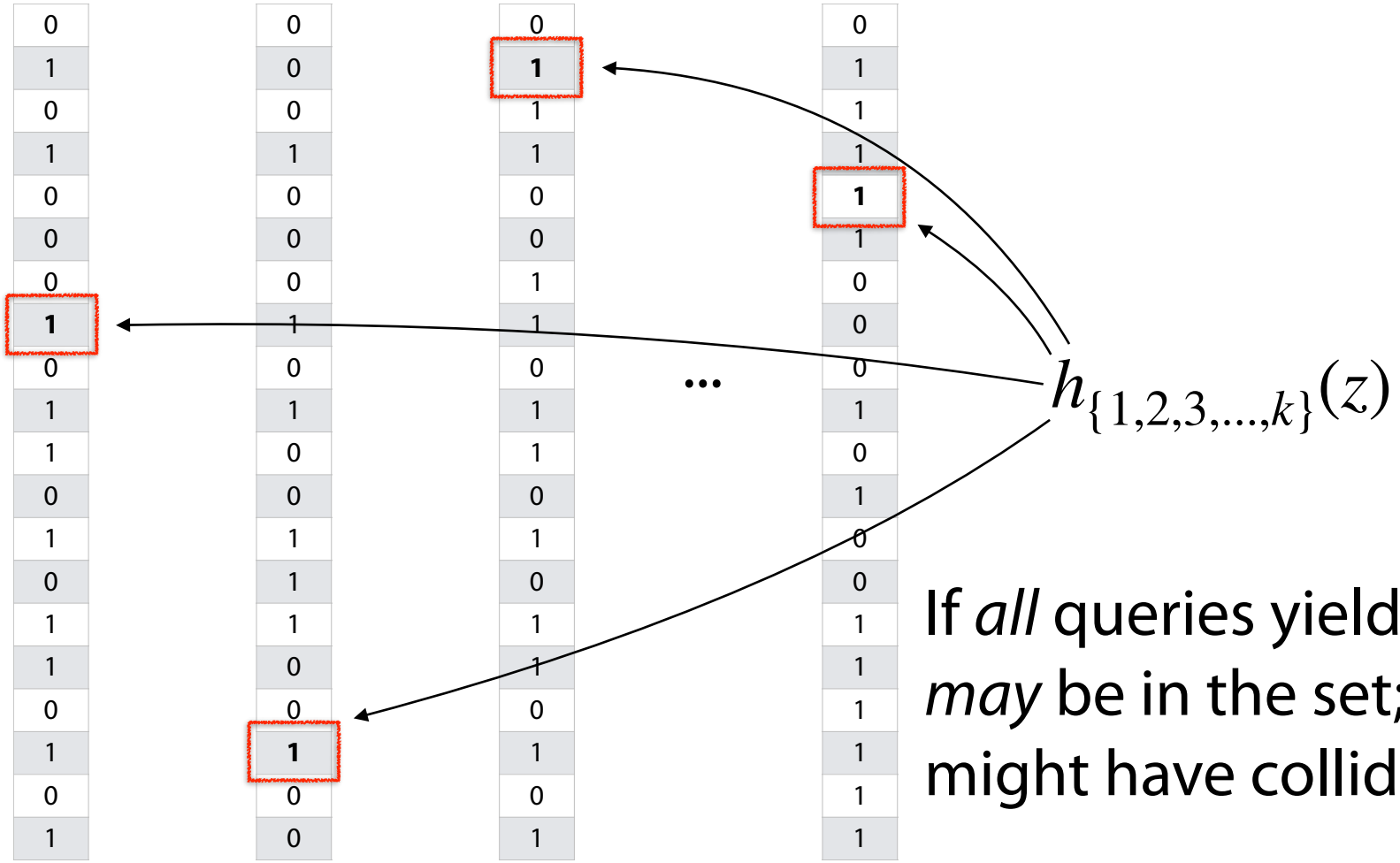
$$h_{\{1,2,3,\ldots,k\}}(y)$$

Bloom Filter: Repeated Trials



If *any* query yields 0, item is not in the set

Bloom Filter: Repeated Trials



If *all* queries yield 1, item *may* be in the set; or we might have collided *k* times

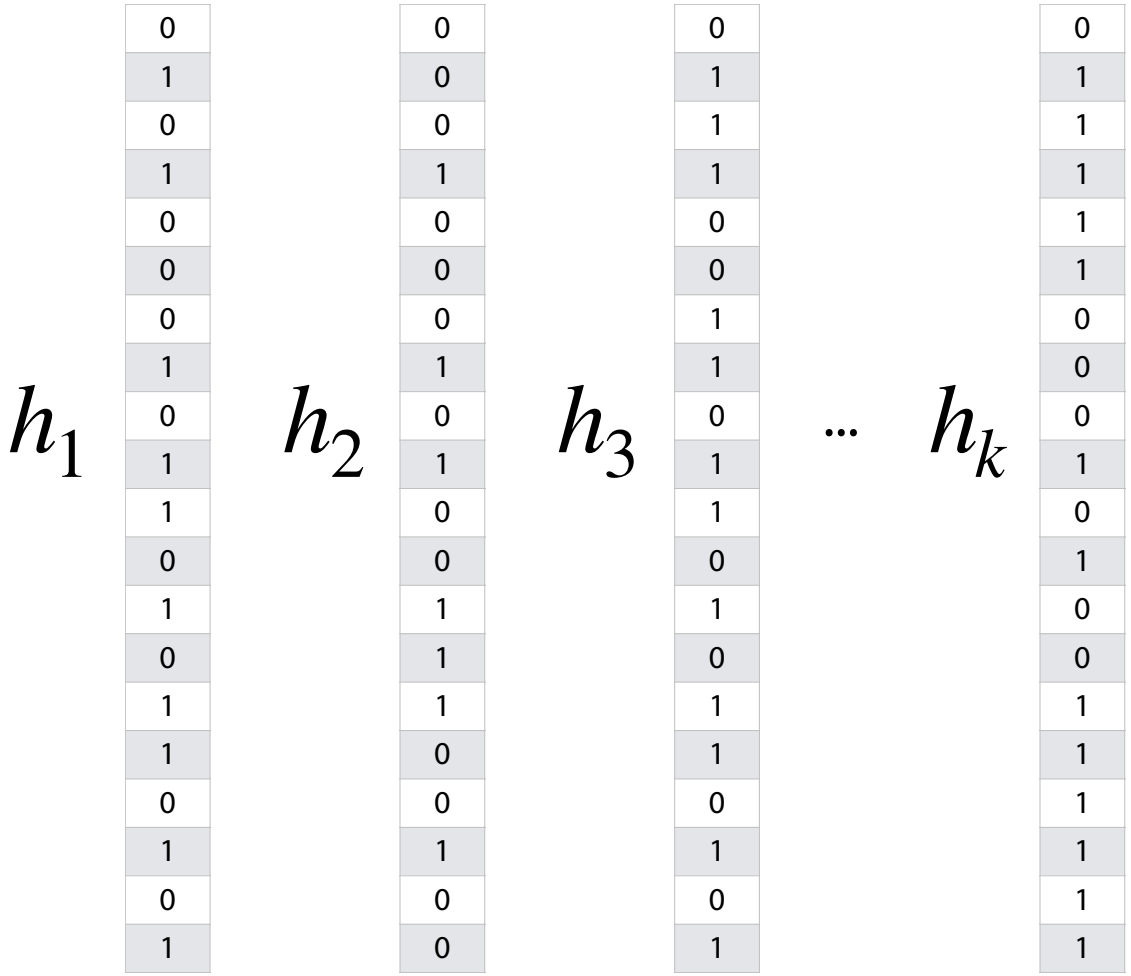
Bloom Filter: Repeated Trials

Using repeated trials, even a very bad filter can still have a very low FPR!

If we have k bloom filter, each with a FPR p, what is the likelihood that ***all*** filters return the value '1' for an item we didn't insert?

Bloom Filter: Repeated Trials

But doesn't this hurt our storage costs by storing k separate filters?



Bloom Filter: Repeated Trials

Rather than use a new filter for each hash, one filter can use k hashes

S = { 6, 8, 4 }

$h_1(x) = x \% 10$ **$h_2(x) = 2x \% 10$** **$h_3(x) = (5+3x) \% 10$**

Bloom Filter: Repeated Trials

Rather than use a new filter for each hash, one filter can use k hashes

0	0
1	0
2	1
3	1
4	1
5	0
6	1
7	1
8	1
9	1

$h_1(x) = x \% 10$ **$h_2(x) = 2x \% 10$** **$h_3(x) = (5+3x) \% 10$**

```
_find(1)
```

```
_find(16)
```

Bloom Filter

$H = \{h_1, h_2, \ldots, h_k\}$

A probabilistic data structure storing a set of values

Built from a bit vector of length m and k hash functions

Insert / Find runs in: _____

Delete is not possible (yet)!

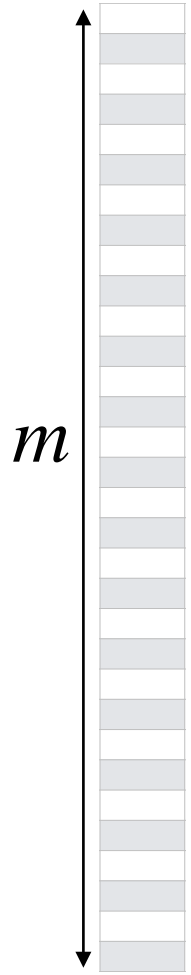
0
0
1
0
0
1
0
1
0
0

Bloom Filter: Error Rate

$h_{\{1,2,3,\ldots,k\}}$

Given bit vector of size m and k SUHA hash function

What is our expected FPR after n objects are inserted?

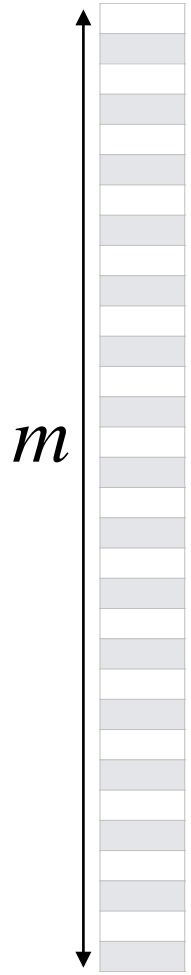


Bloom Filter: Error Rate

$h_{\{1,2,3,\ldots,k\}}$

Given bit vector of size m and 1 SUHA hash function

What's the probability a specific bucket is 1 after one object is inserted?



Same probability given k SUHA hash function?

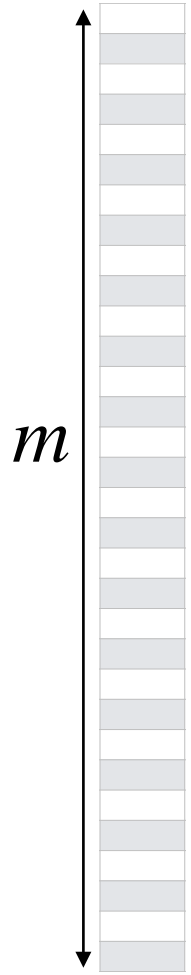
Bloom Filter: Error Rate

$h_{\{1,2,3,\ldots,k\}}$

Given bit vector of size m and k SUHA hash function

Probability a specific bucket is 0 after one object is inserted?

After n objects are inserted?

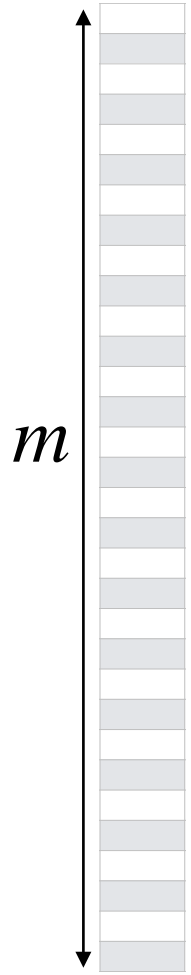


Bloom Filter: Error Rate

$h_{\{1,2,3,\ldots,k\}}$

Given bit vector of size m and k SUHA hash function

What's the probability a specific bucket is 1 after n objects are inserted?



Bloom Filter: Error Rate

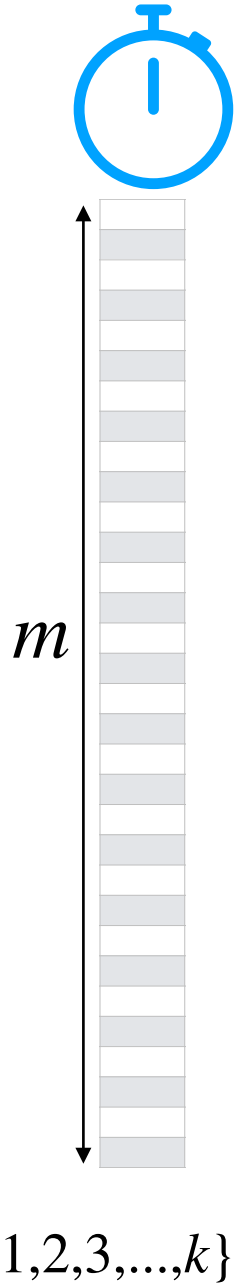
Given bit vector of size m and k SUHA hash function

What is our expected FPR after n objects are inserted?

The probability my bit is 1 after n objects inserted

$$\left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^{k}$$

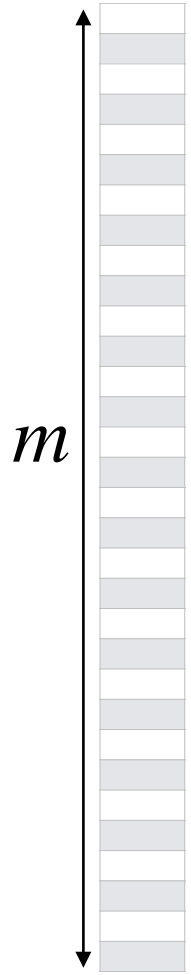
The number of [assumed independent] trials



Bloom Filter: Error Rate

$h_{\{1,2,3,\ldots,k\}}$

Vector of size m, k SUHA hash function, and n objects



To minimize the FPR, do we prefer…

(A) large k **(B) small** k

$$\left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^{k}$$

Bloom Filter: Error Rate

Vector of size m, k SUHA hash function, and n objects

(A) large k

$$\left(1-\left(1-\frac{1}{m}\right)^{nk}\right)^{k}$$

As k increases, this gets smaller!

(B) small k

$$\left(1-\left(1-\frac{1}{m}\right)^{nk}\right)^{k}$$

As k decreases, this gets smaller!

Bloom Filter: Optimal Error Rate

To build the optimal hash function, fix **m** and **n**!

Claim: The optimal hash function is when $k^* = ln\ 2 \cdot \frac{m}{n}$

(1) $$\left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^k \approx \left(1 - e^{\frac{-nk}{m}}\right)^k$$

(2) $$\frac{d}{dk}\left(1 - e^{\frac{-nk}{m}}\right)^k \approx \frac{d}{dk}\left(k\ \ ln(1 - e^{\frac{-nk}{m}})\right)$$

Bloom Filter: Optimal Error Rate

Claim 1: $\left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^{k} \approx \left(1 - e^{\frac{-nk}{m}}\right)^{k}$

$$\left(1 - \frac{1}{m}\right)^{nk} = e^{ln\left[\left(1 - \frac{1}{m}\right)^{nk}\right]}$$

$$= e^{ln\left[\left(1 - \frac{1}{m}\right)\right]nk}$$

$$\approx e^{\frac{-nk}{m}}$$

Bloom Filter: Optimal Error Rate

Claim 2: $$\frac{d}{dk}\left(1 - e^{\frac{-nk}{m}}\right)^k \approx \frac{d}{dk}\left(k \;\; ln(1 - e^{\frac{-nk}{m}})\right)$$

Fact: $$\frac{d}{dx}\ln\ f(x) = \frac{1}{f(x)}\frac{df(x)}{dx}$$

TL;DR: $$min\left[f(x)\right] = min\left[\ln\ f(x)\right]$$

Derivative is zero when $k^* = \ln 2 \cdot \frac{m}{n}$

Bloom Filter: Error Rate

$$\left(1 - e^{\frac{-nk}{m}}\right)^k$$

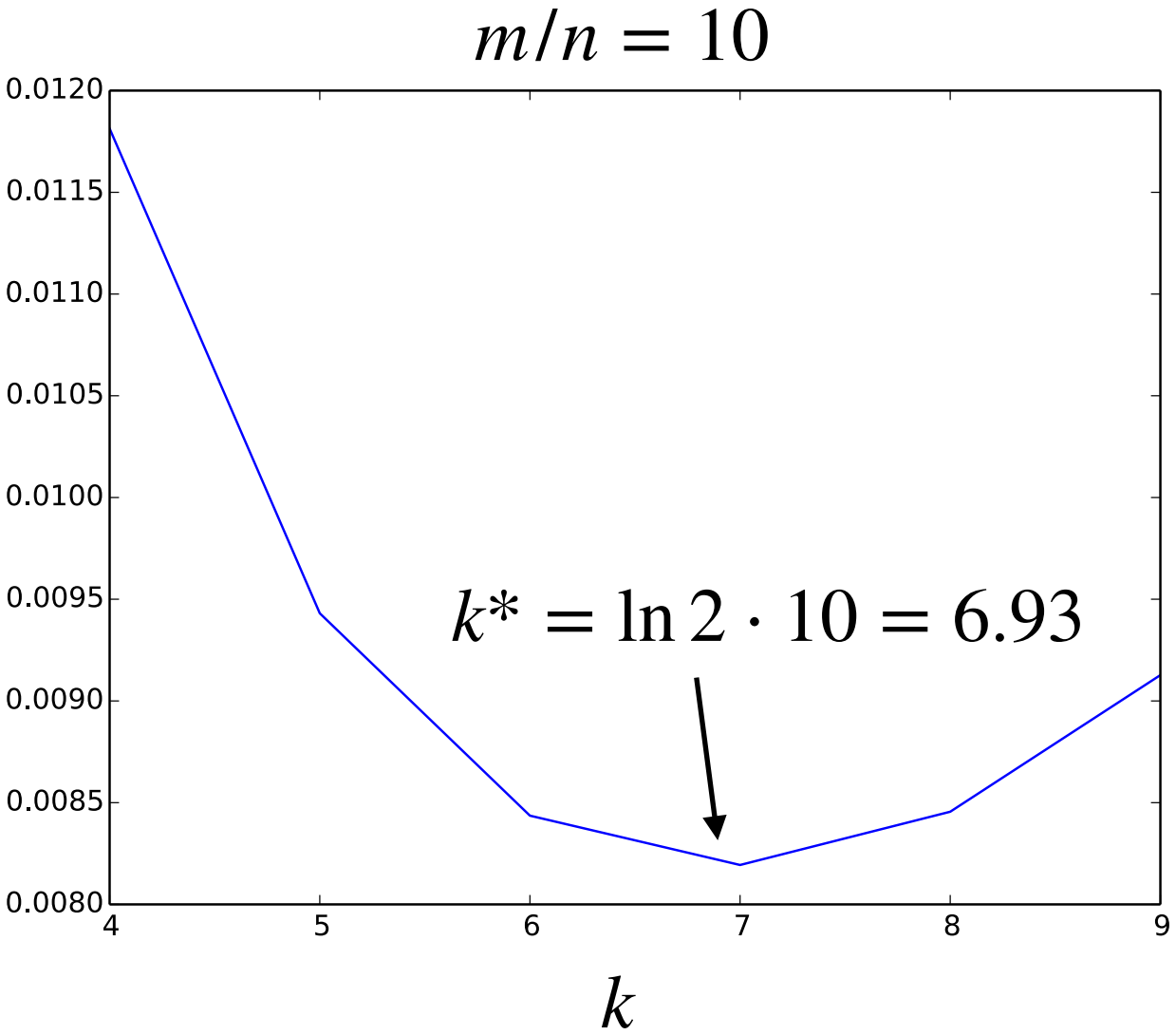


Figure by Ben Langmead

Bloom Filter: Optimal Parameters

$$k^* = \ln 2 \cdot \frac{m}{n}$$

Given any two values, we can optimize the third

$n = 100$ items $\quad k = 3$ hashes $\quad m =$

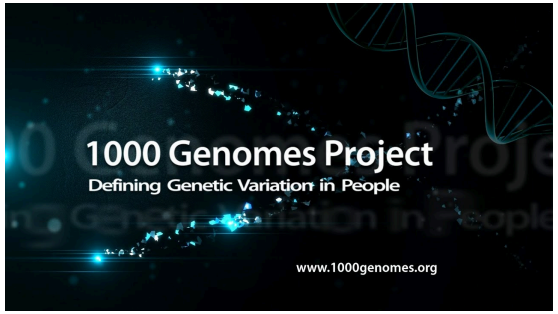
$m = 100$ bits $\quad n = 20$ items $\quad k =$

$m = 100$ bits $\quad k = 2$ items $\quad n =$

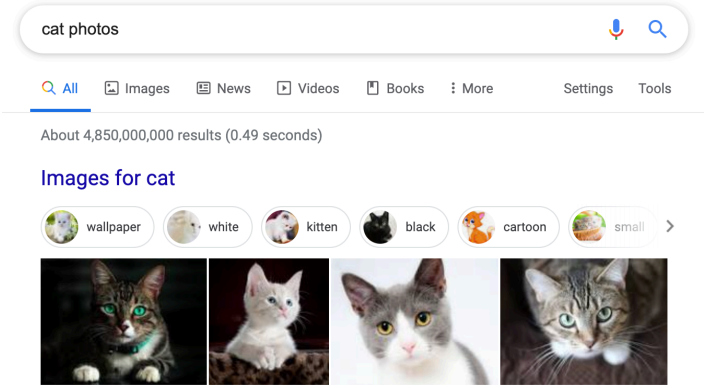
Bloom Filter: Optimal Parameters

$$m = \frac{nk}{\ln 2} \approx 1.44 \cdot nk$$

Optimal hash function is still O(m)!

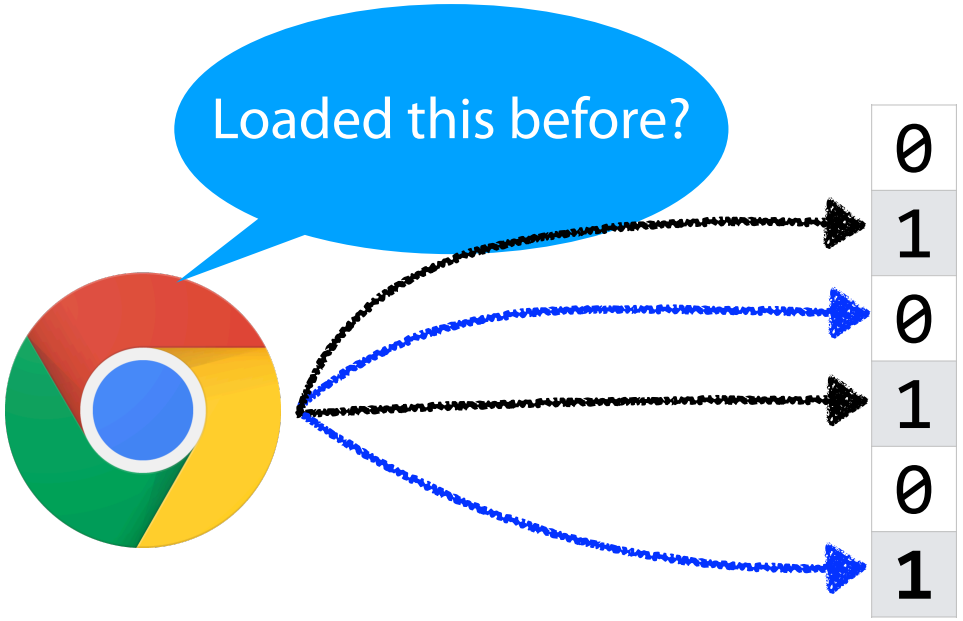


n = 250,000 files vs ~10^{15} nucleotides vs 260 TB



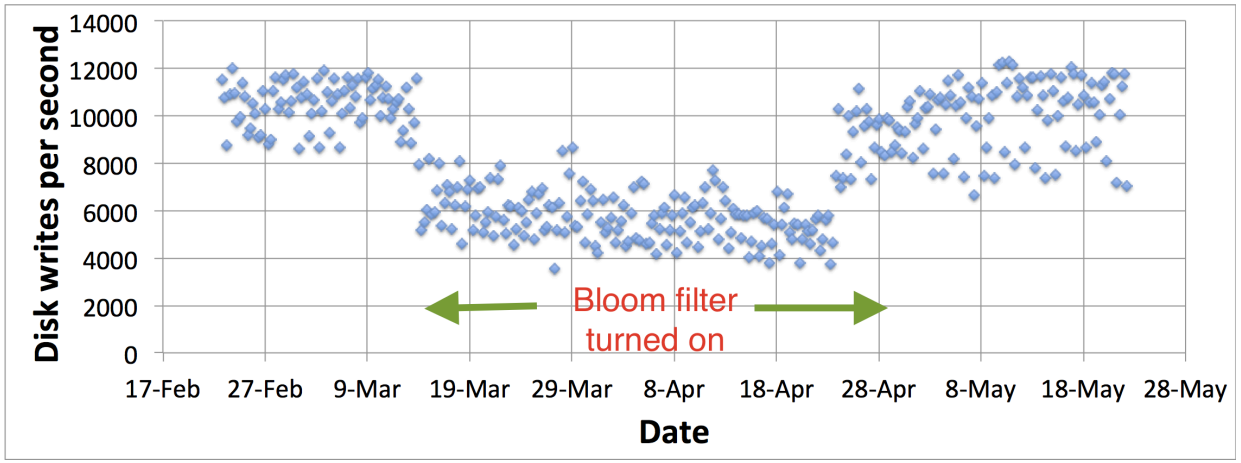
n = 60 billion — 130 trillion

Bloom Filter: Website Caching



Cache this page!

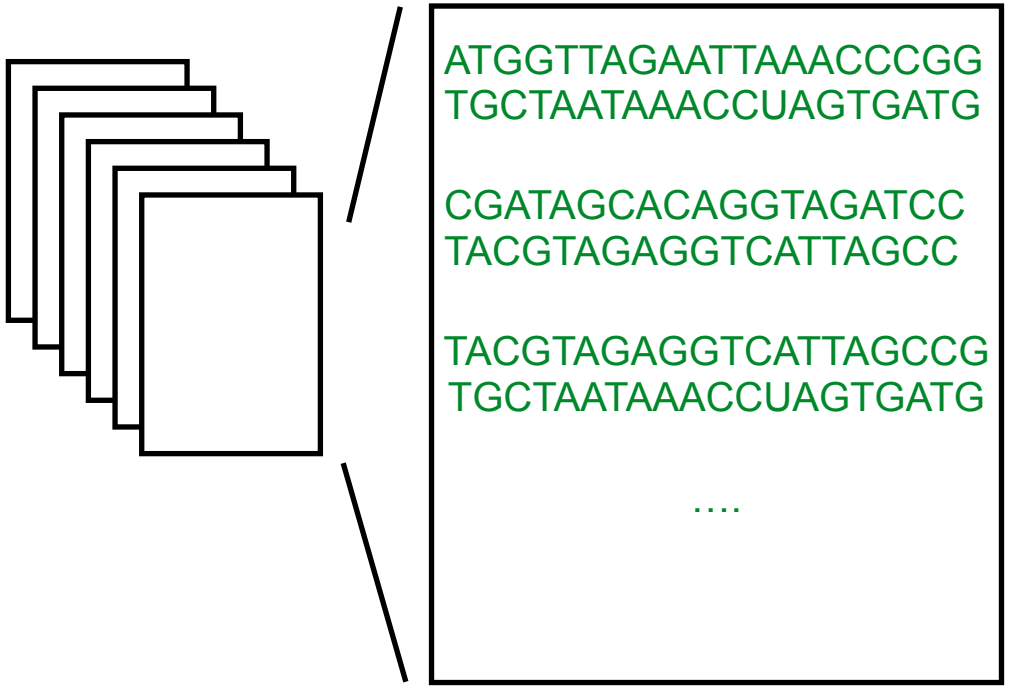
Add to filter (but don't cache!)



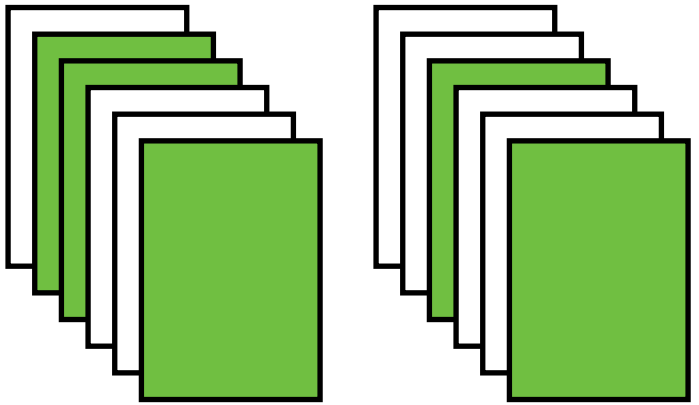
Maggs, Bruce M., and Ramesh K. Sitaraman. **Algorithmic nuggets in content delivery.** *ACM SIGCOMM Computer Communication Review* 45.3 (2015): 52-66.

Sequence Bloom Trees

Imagine we have a large collection of text…

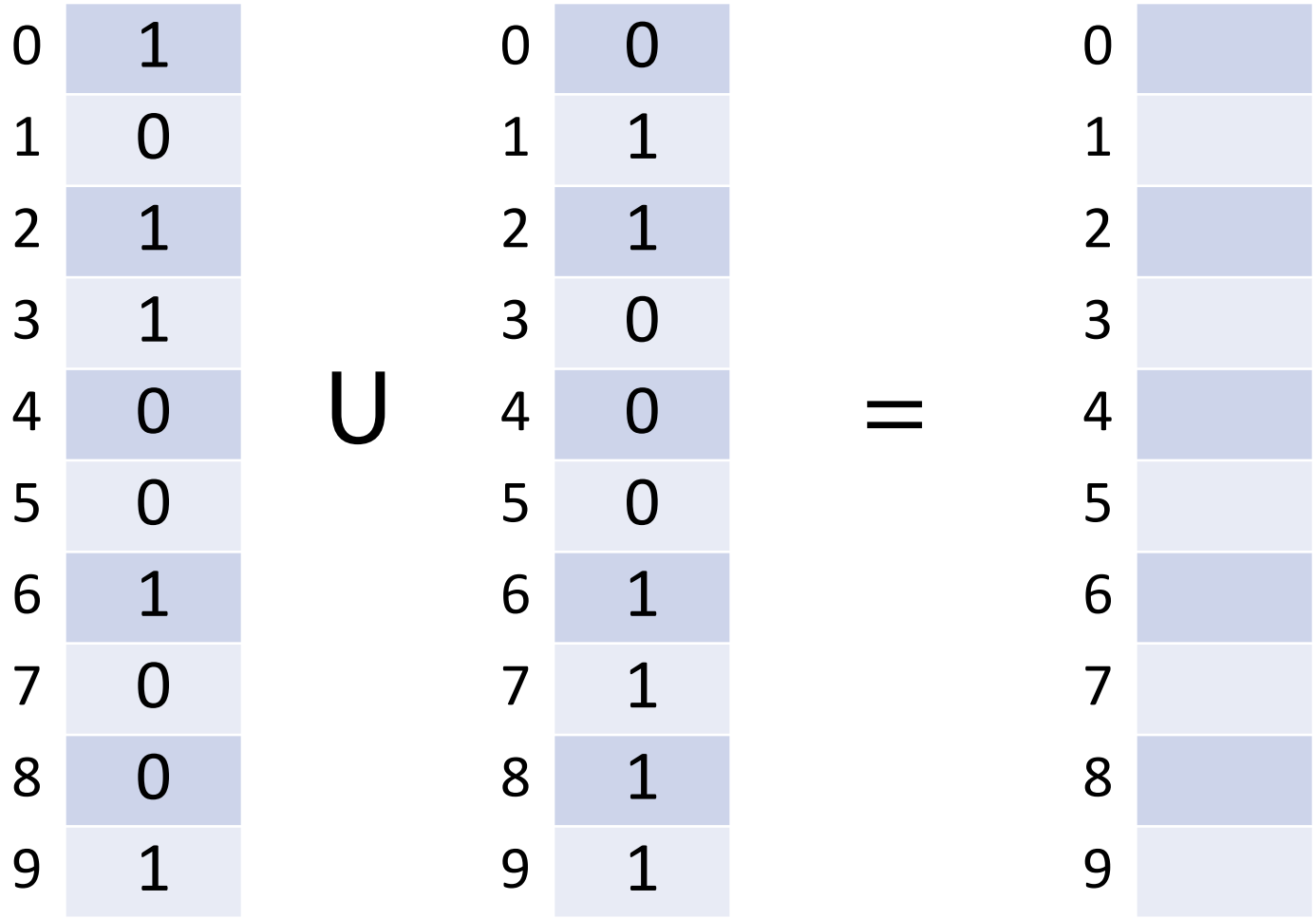


And our goal is to search these files for a query of interest…



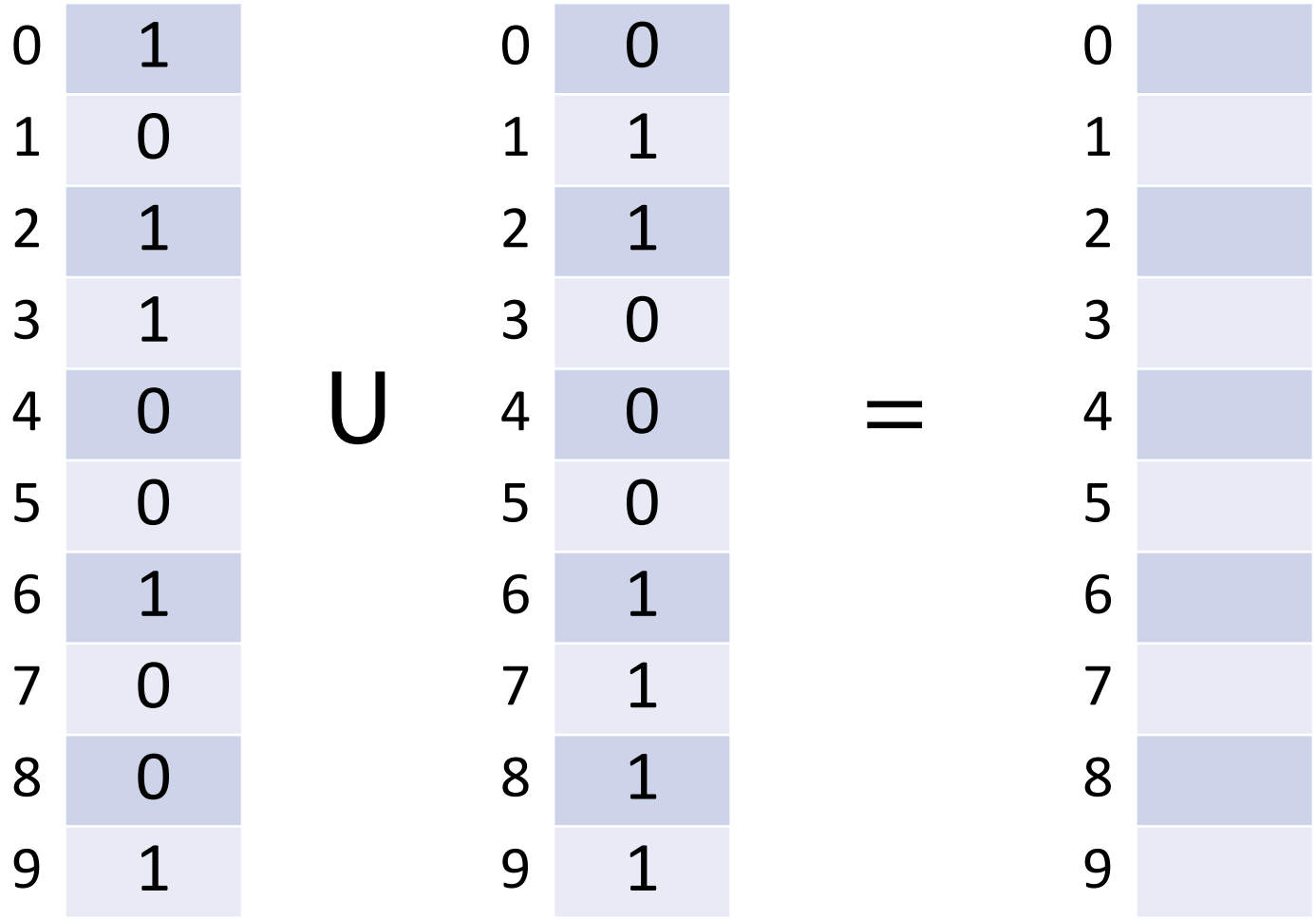
Bloom Filters: Unioning

Bloom filters can be trivially merged using bit-wise union.



Bloom Filters: Intersection

Bloom filters can be trivially merged using bit-wise intersection.



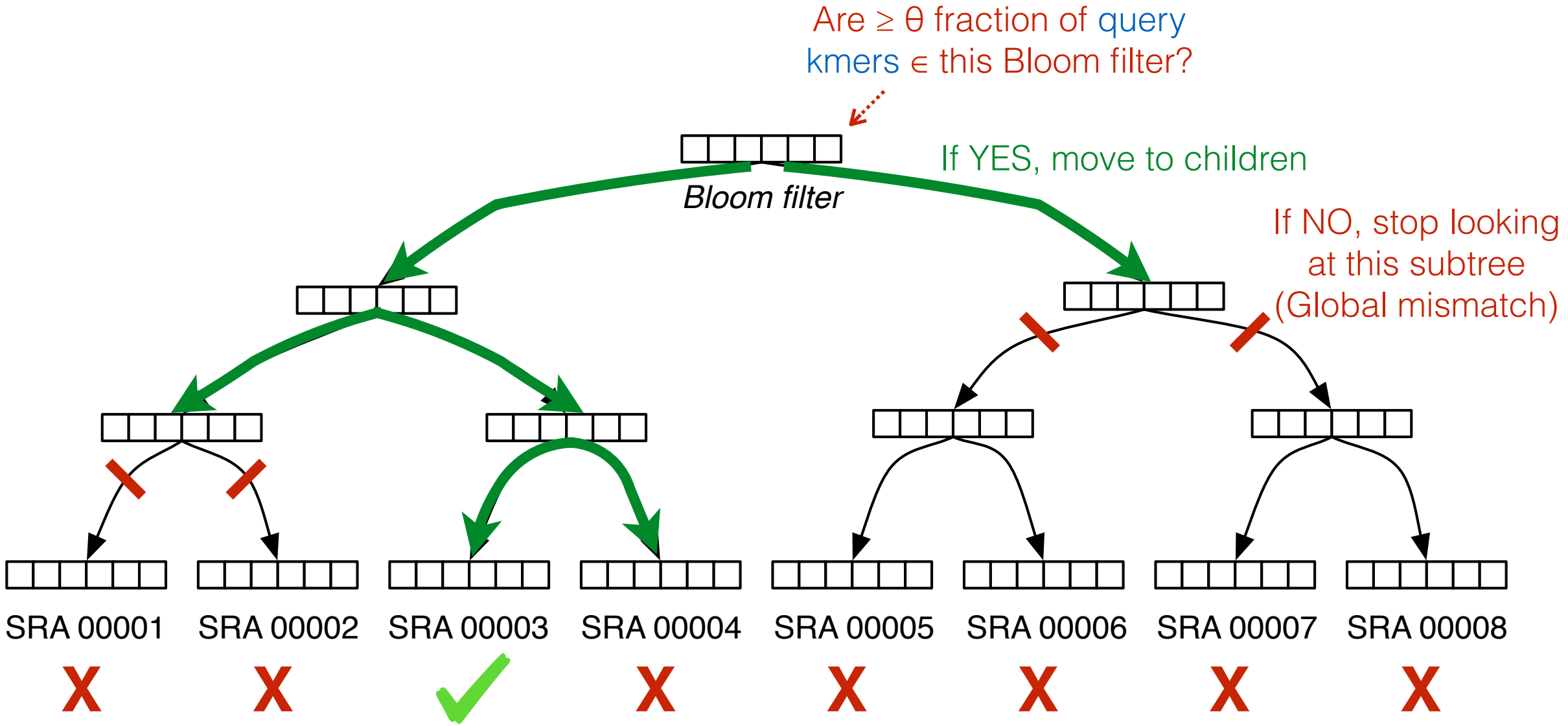
Bit Vector Merging

What is the conceptual meaning behind **union** and **intersection**?

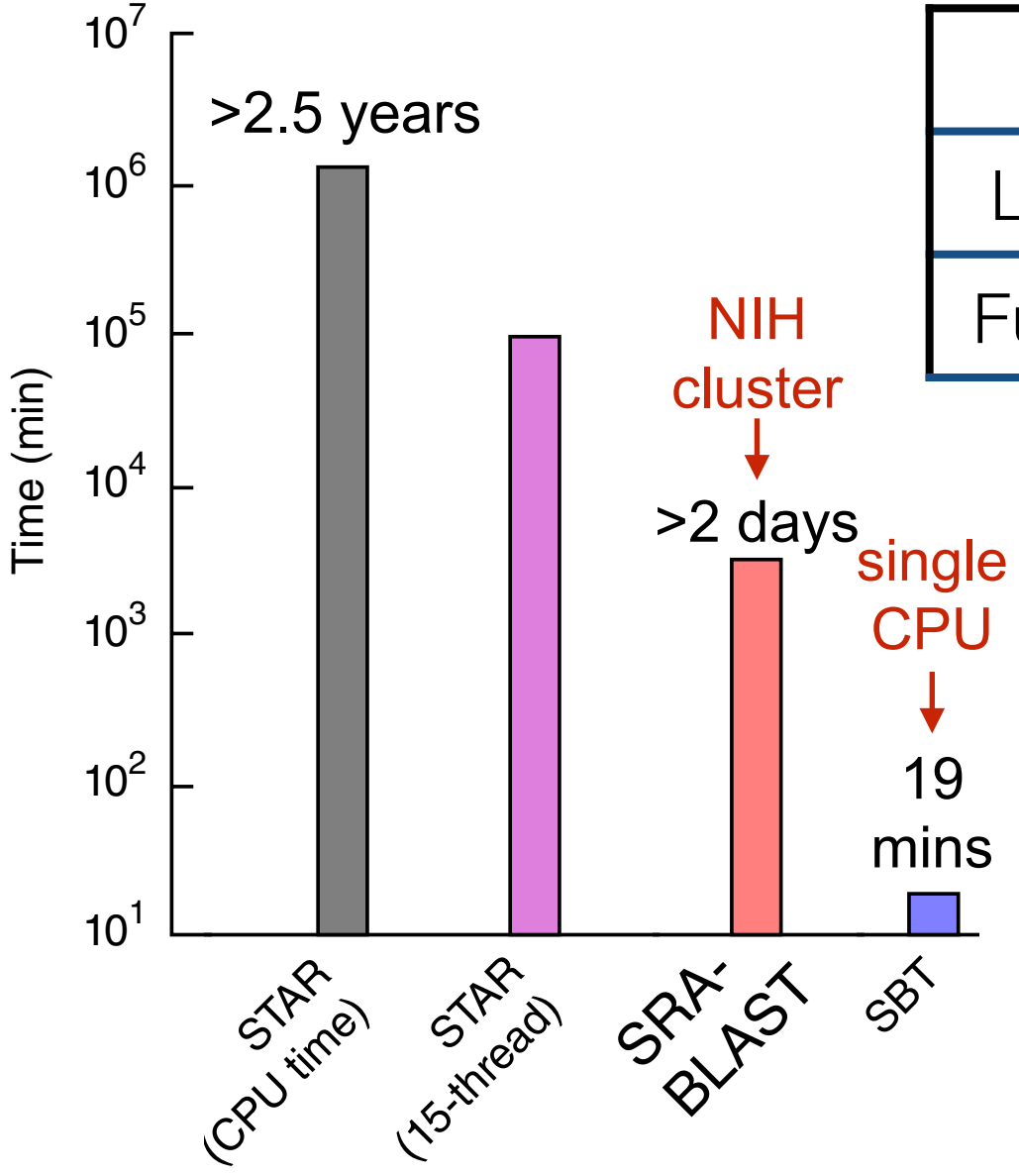
SRA 00001 SRA 00002 SRA 00003 SRA 00004 SRA 00005 SRA 00006 SRA 00007 SRA 00008

Sequence Bloom Trees

Sequence Bloom Trees



Sequence Bloom Trees



	SRA	FASTA.gz	SBT
Leaves	4966 GB	2692 GB	63 GB
Full Tree	-	-	200 GB

Solomon, Brad, and Carl Kingsford. "Fast search of thousands of short-read sequencing experiments." *Nature biotechnology* 34.3 (2016): 300-302.

Solomon, Brad, and Carl Kingsford. "Improved search of large transcriptomic sequencing databases using split sequence bloom trees." *International Conference on Research in Computational Molecular Biology*. Springer, Cham, 2017.

Sun, Chen, et al. "Allsome sequence bloom trees." *International Conference on Research in Computational Molecular Biology*. Springer, Cham, 2017.

Harris, Robert S., and Paul Medvedev. "Improved representation of sequence bloom trees." *Bioinformatics* 36.3 (2020): 721-727.

Bloom Filters: Tip of the Iceberg

Cohen, Saar, and Yossi Matias. "Spectral bloom filters." *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 2003.

Fan, Bin, et al. "Cuckoo filter: Practically better than bloom." *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. 2014.

Nayak, Sabuzima, and Ripon Patgiri. "countBF: A General-purpose High Accuracy and Space Efficient Counting Bloom Filter." *2021 17th International Conference on Network and Service Management (CNSM)*. IEEE, 2021.

Mitzenmacher, Michael. "Compressed bloom filters." *IEEE/ACM transactions on networking* 10.5 (2002): 604-612.

Crainiceanu, Adina, and Daniel Lemire. "Bloofi: Multidimensional bloom filters." *Information Systems* 54 (2015): 311-324.

Chazelle, Bernard, et al. "The bloomier filter: an efficient data structure for static support lookup tables." *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*. 2004.

There are many more than shown here…