Data Structures and Algorithms
Bloom Filters

CS 225
Brad Solomon

November 18, 2024

Why store

$\boxed{\frac{K,V}{K,V}}$

when you can store

$\boxed{\frac{1}{0}}$

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Announcements

MP_mosaic survey EC reached

MP_traversal survey EC **not reached** (Have until 11/20 to submit!)

MP_puzzle released, due after break. **Break doesn't count as a week**

↳ lab this week

Learning Objectives

review

Review when you would prefer different data structures

Build a conceptual understanding of a bloom filter

Review probabilistic data structures and one-sided error

Formalize the math behind the bloom filter

Wednesday

Which collision resolution strategy is better?

- Big Records: Open hashing (Seperate chaining)
  - ↳ can pass by ref
  - ↳ closed hashing cant allocate at this scale

K, V
&
externally

- Structure Speed: Closed hashing (Double hashing)

What structure do hash tables implement?

Dictionaries

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

① Probabilistic!

② Simple uniform hashing assumption (SUHA)

③ Pseudo-amortized

↳ B/c resize when $\alpha < 1$

Why talk about BSTs at all?

↳ Some data is not hashable

↳ Ordered dataset useful (Nearest neighbor)

Running Times

Exact lookup only

Trees in general

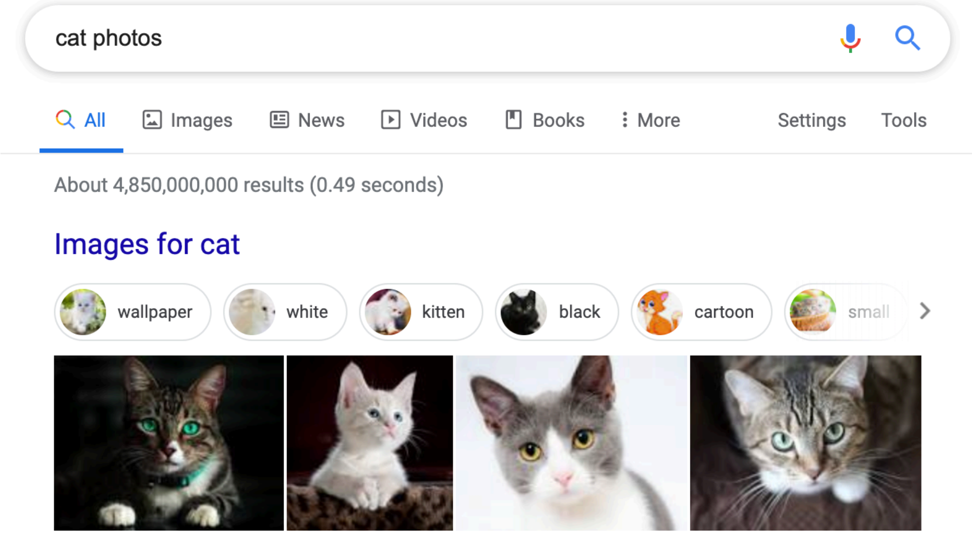
Approx match (KD Tree)

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

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

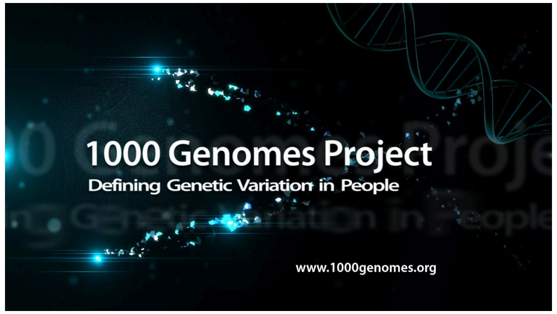
Google Universe Estimate (2013): >130 trillion webpages

Memory-Constrained Data Structures

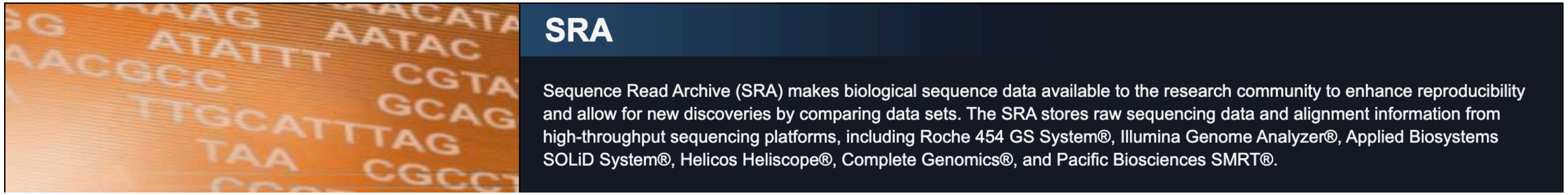
What method would you use to build a search index on a collection of objects *in a memory-constrained environment*?

Constrained by Big Data (Large N)

ENA
European Nucleotide Archive



GTEx



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)

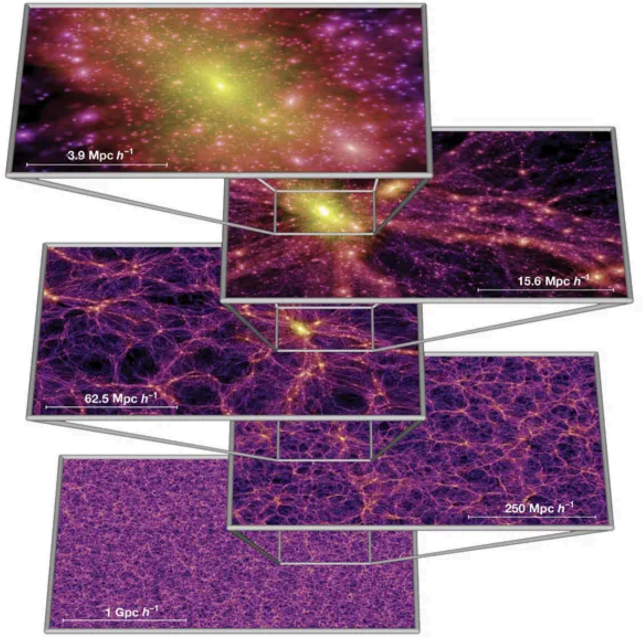


Image: https://doi.org/10.1038/nature03597

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

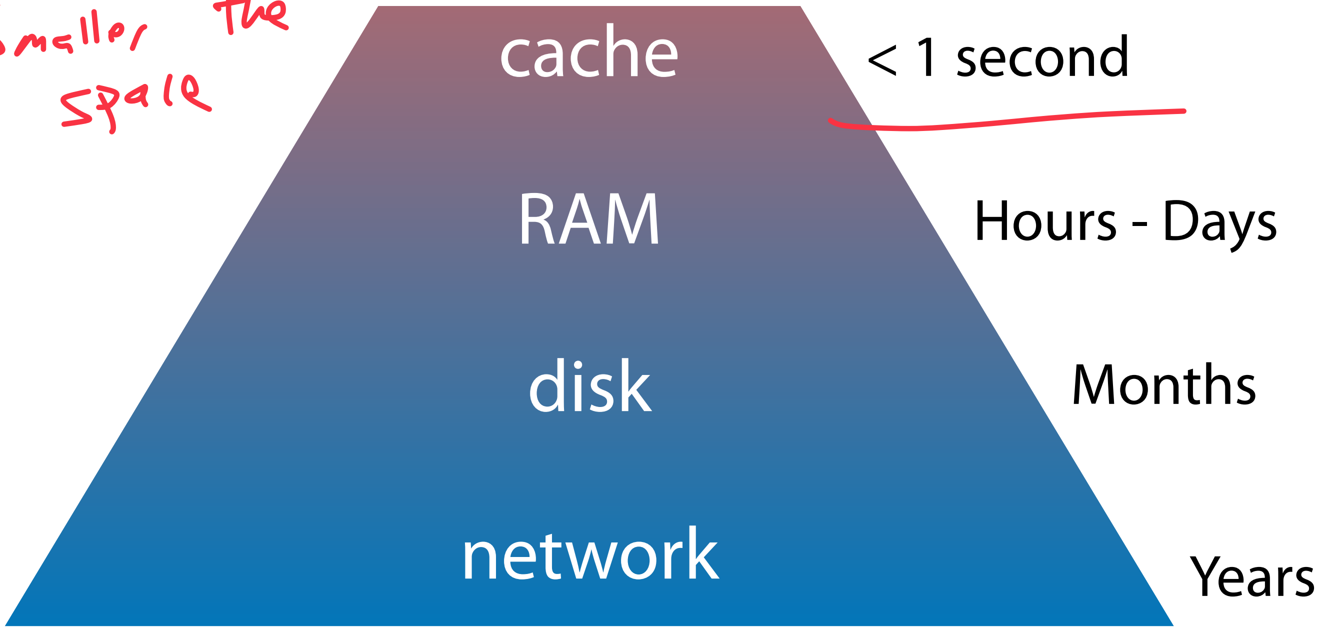
Estimated total volume of one array: 4.6 EB

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

Smaller the space



(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*?

1) Make a space-efficient encoding (compress the data)

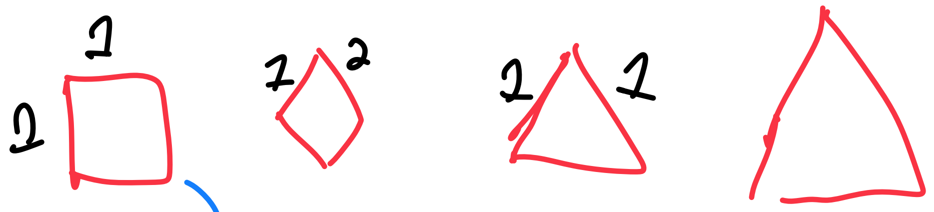
2) Throw away data we dont care about

3) Make a distributed network

Make a bloom filter

Reducing storage costs

100 billion



1) Throw out information that isn't needed

\# of sides 4 4 3

Length of perimeter : X X

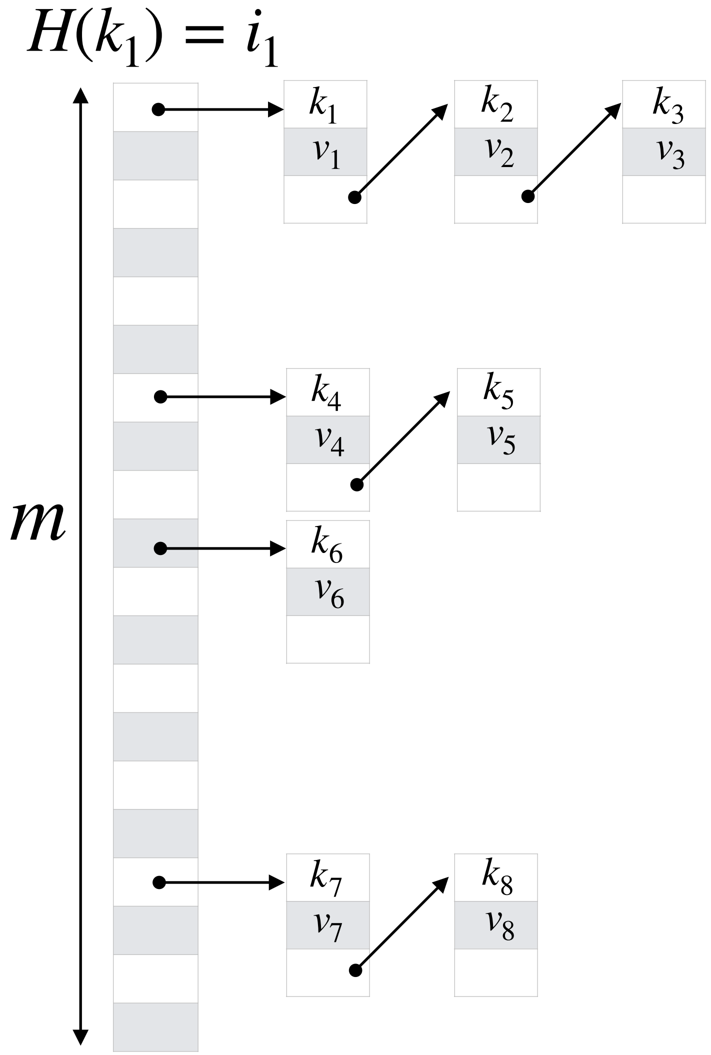
Use data stream to process each item individually

2) Compress the dataset "AAAGGG"

A3 G3 = run length encoding

Reducing a hash table

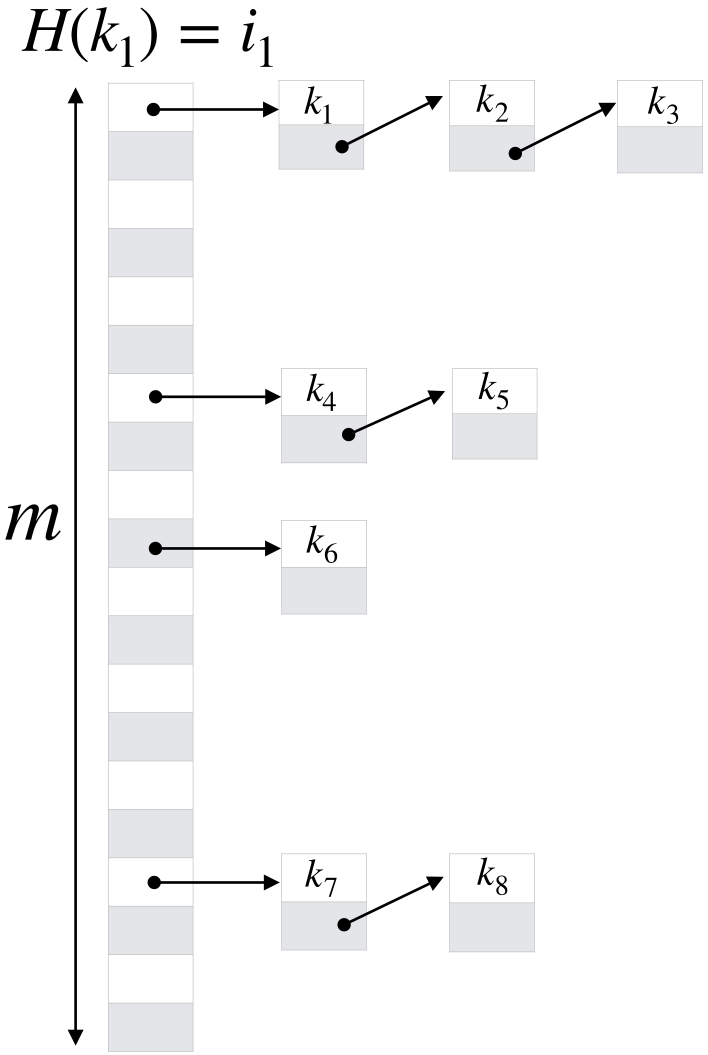
What can we remove from a hash table?



Reducing a hash table

What can we remove from a hash table?

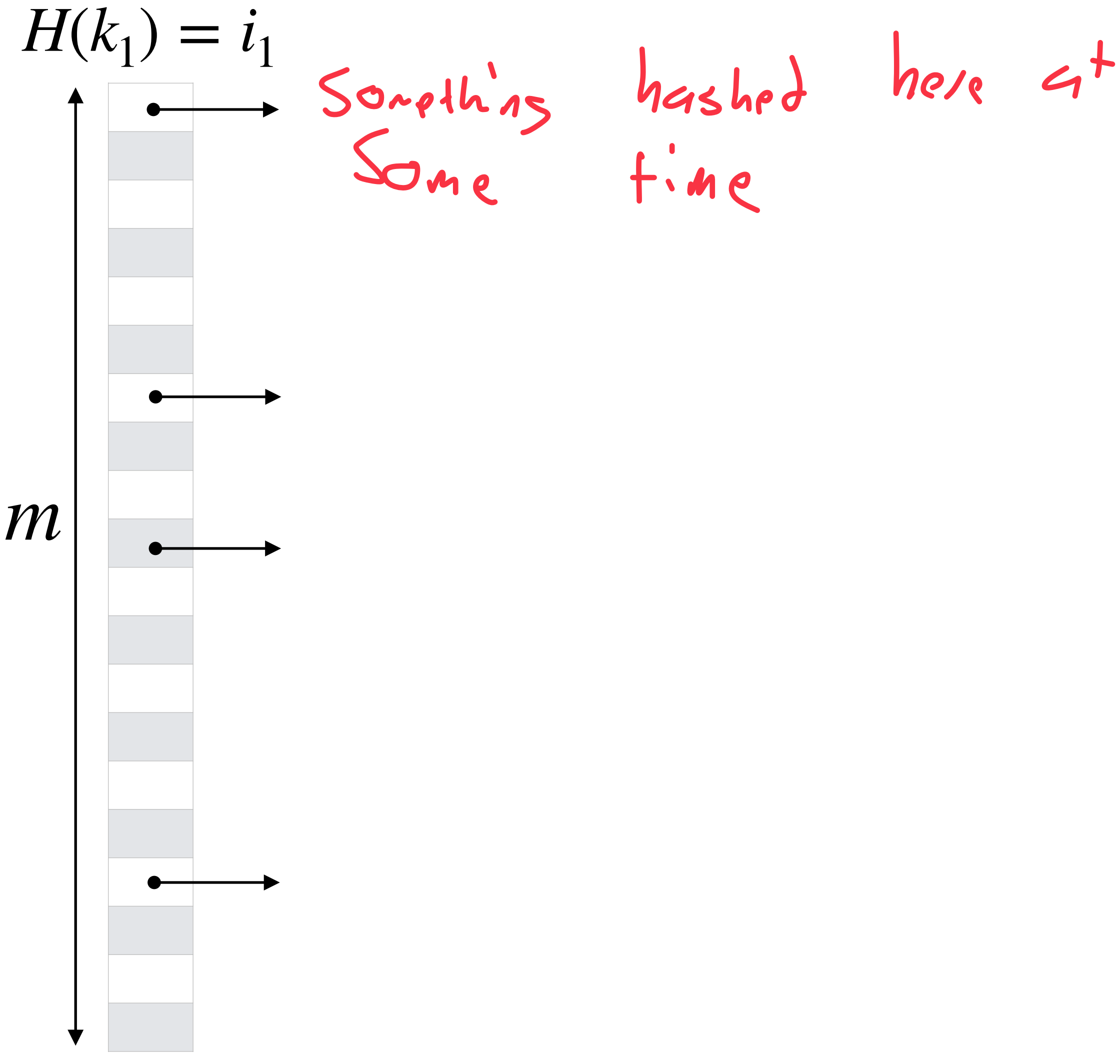
Take away values



Reducing a hash table

What can we remove from a hash table?

Take away values and keys



Reducing a hash table

What can we remove from a hash table?

Take away values and keys

This is a ***bloom filter***

$H(k_1) = i_1$

Something hashed here

Nothing hashed here

m

1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0

$O(m)$ storage (small constant 1 bit per block)

Bloom Filter ADT

Constructor

Insert

Find

Bloom Filter: Insertion

Empty BF is vector/array of 0s

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

1) Hash key to hash value (address)

h(k) = k % 7

2) Set bit to 1 at address

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

← h(29) = 1

No collision possible

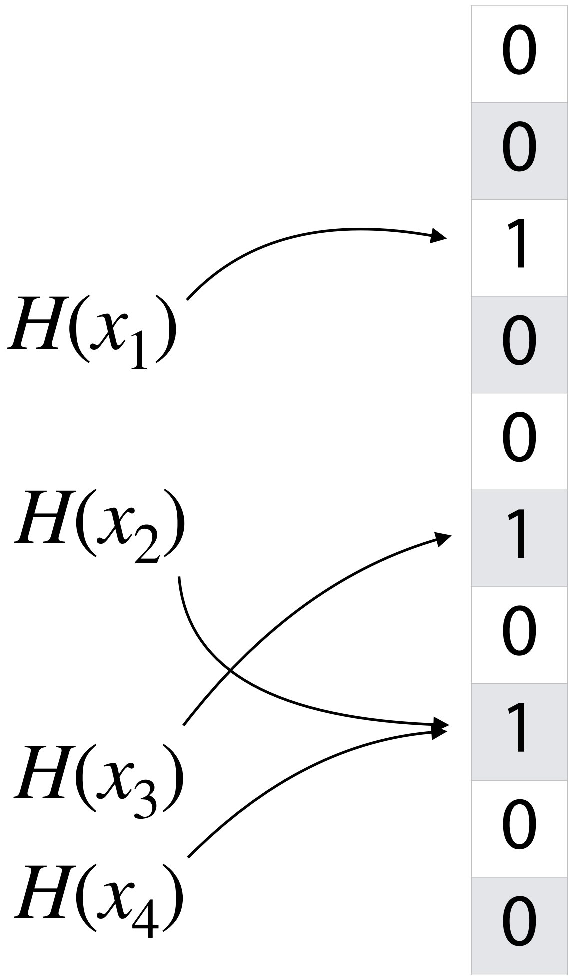
1 says "Something hashed at some point"

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

↳ Each 1 is one or more inserts



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

Oh no!

`_delete(13)`

1) Hash key

2) Set bit to 0

`_delete(29)`

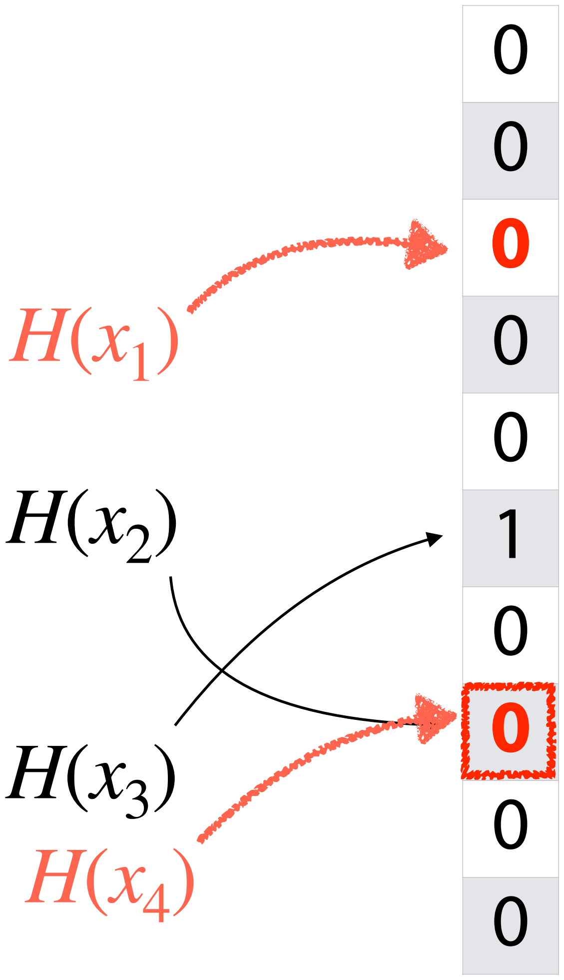
_find(8)

Bloom Filter: Deletion

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

① We never know if it is safe to set 1 to 0.

② We need the property that 1 stays 1 forever



Bloom Filter: Search

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

6

h(k) = k % 7

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

`_find(16)` → true! ✓ correct

1) Hash key

2) Lookup value

`_find(20)` → True ✗ False

↳ 20 % 7 = 6

`_find(3)` → False ✓ Correct

Probabilistic accuracy!

Bloom Filter: Search

The bloom filter is a *probabilistic* data structure!

If the value in the BF is 0:

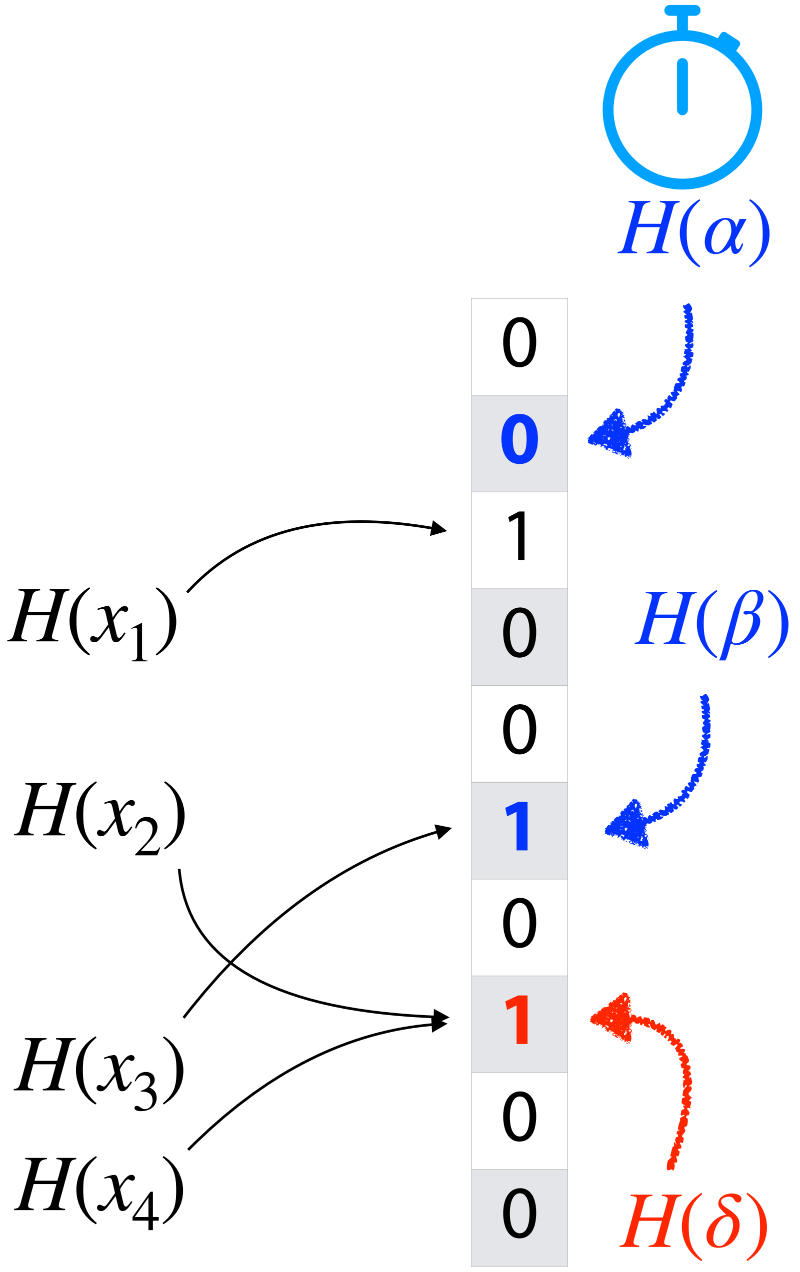
100% of the time item not in set

If the value in the BF is 1:

item might be in set

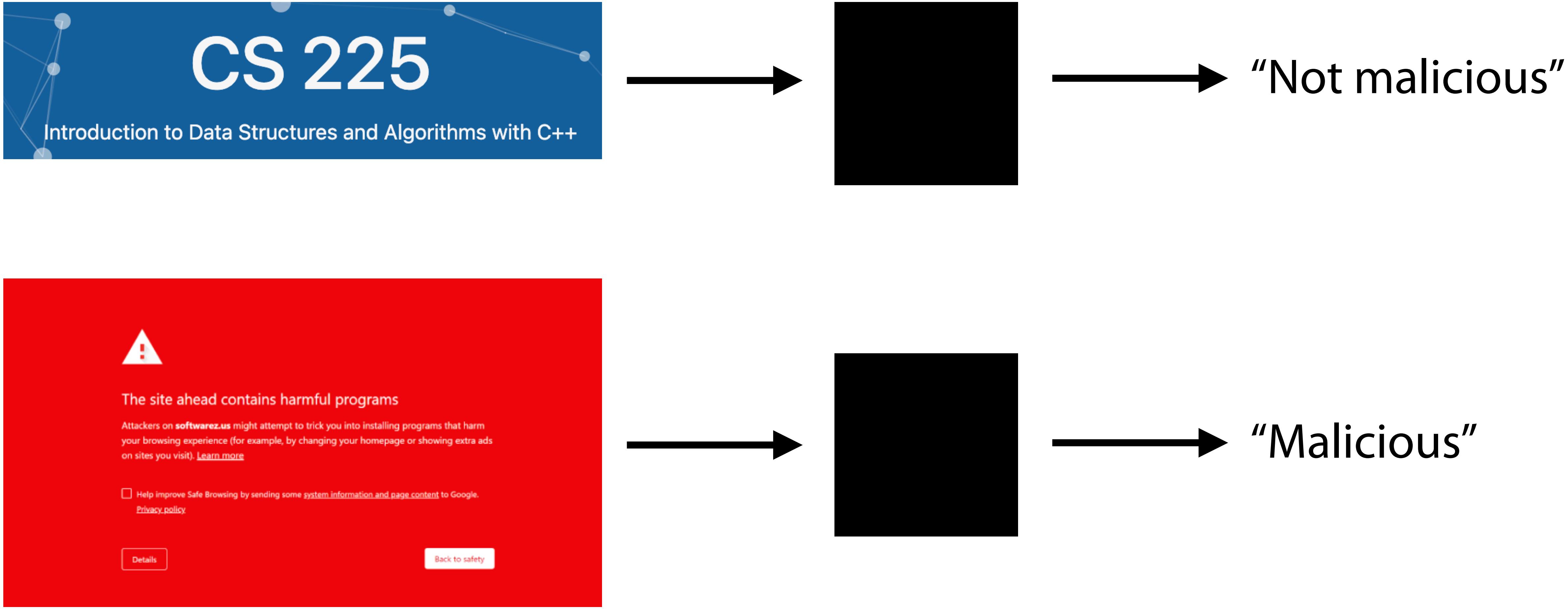
or

is a hash collision



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: Oracle says m | actual website is m

False Positive: Oracle says m | actual not m

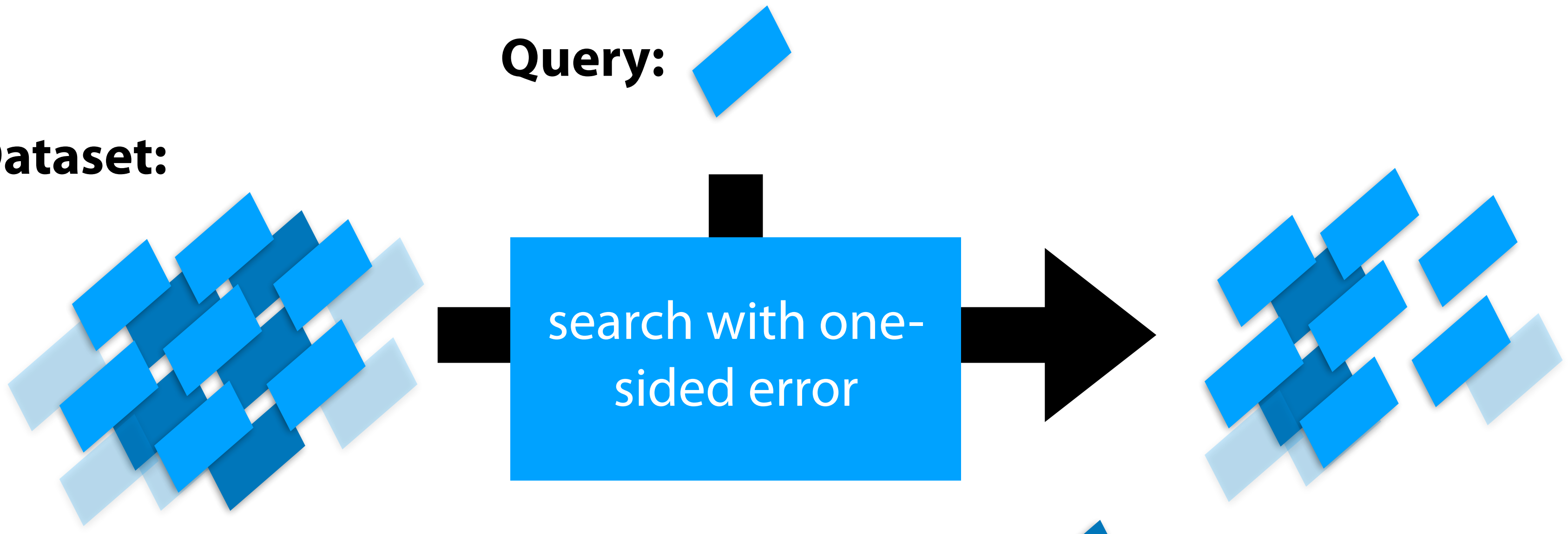
False Negative: not m | actual m

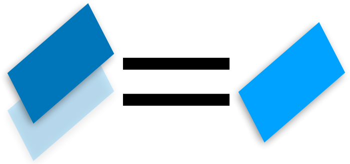
True Negative: not m | not m

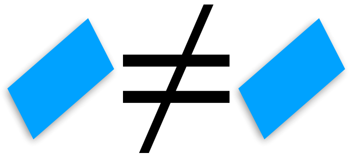
Imagine we have a **bloom filter** that **stores malicious sites…**

	Bit Value = 1	**Bit Value = 0**
Item Inserted	$H(z)$ 0 1 **'Yes'** 0 0 1 True Positive	$H(z)$ was inserted 0 0 **'No'** 0 0 1 False Negative Not possible
Item NOT inserted	0 1 **'Yes'** 0 0 1 False Positive	0 0 **'No'** 0 0 1 True Negative

Probabilistic Accuracy: One-sided error

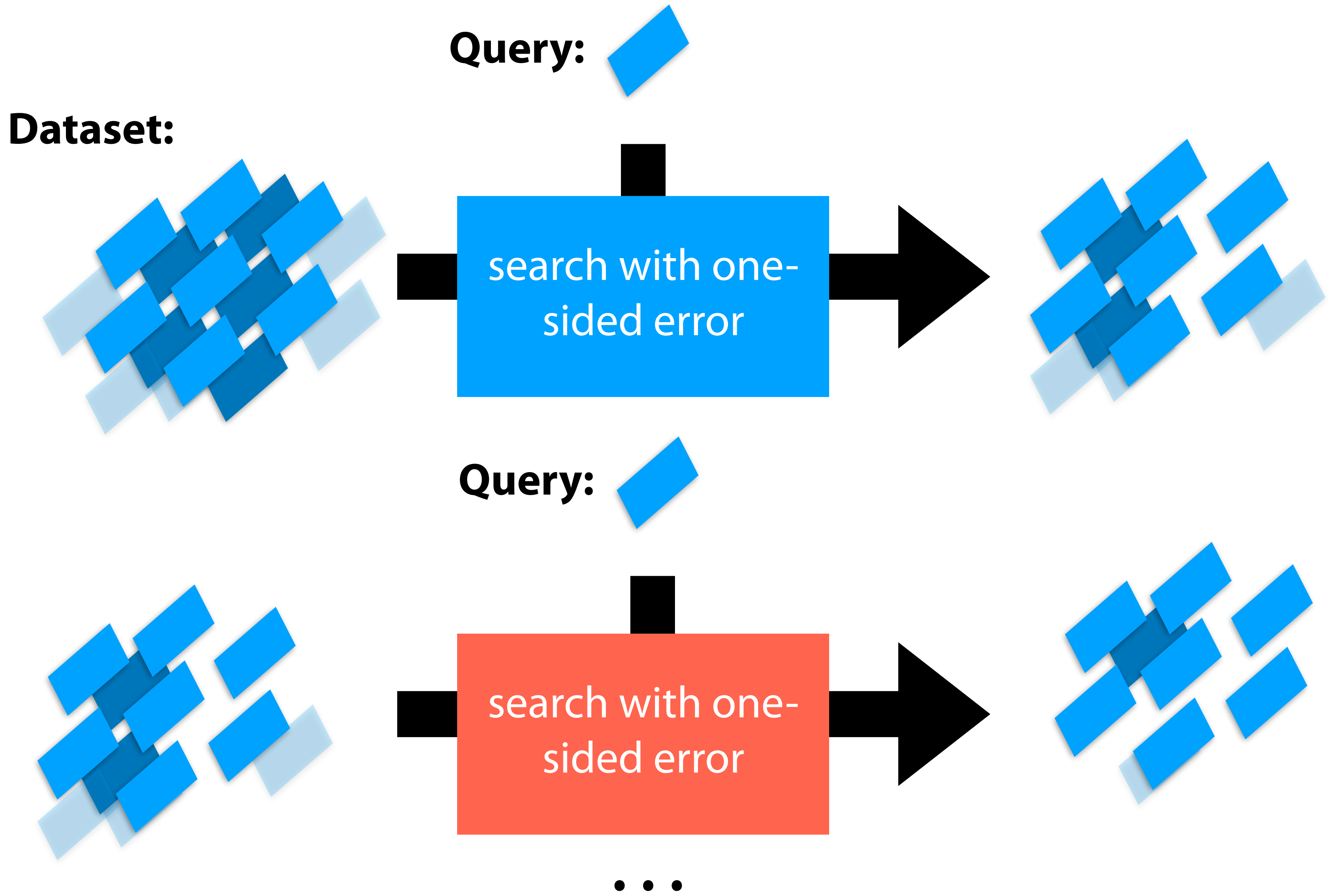


We will get some False Positives: 

We will NEVER have a False Negative: 

will remove some bad but no good items

Probabilistic Accuracy: One-sided error

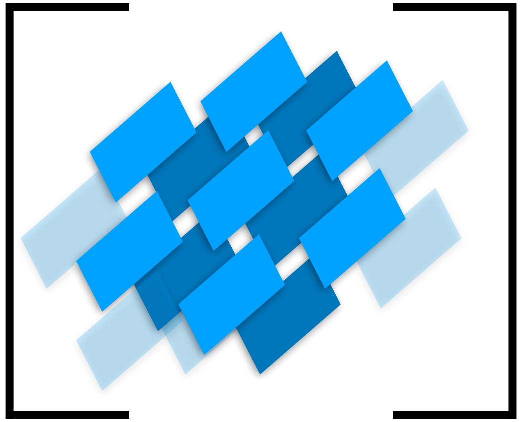


Bloom Filter: Repeated Trials

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

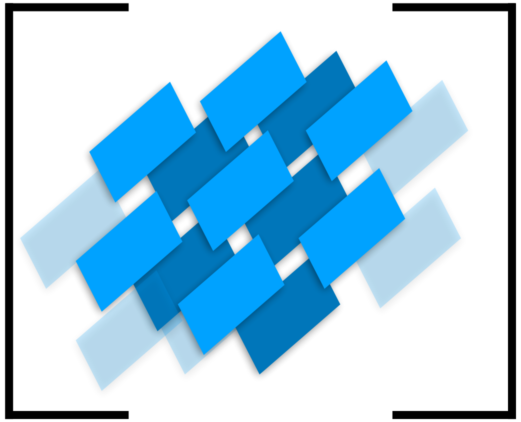
h_1



Bloom Filter: Repeated Trials

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

h_1	h_2
0	0
1	0
0	0
1	**1**
0	0
0	0
0	0
1	**1**
0	0
1	**1**
1	0
0	0
1	**1**
0	**1**
1	**1**
1	0
0	0
1	**1**
0	0
1	0

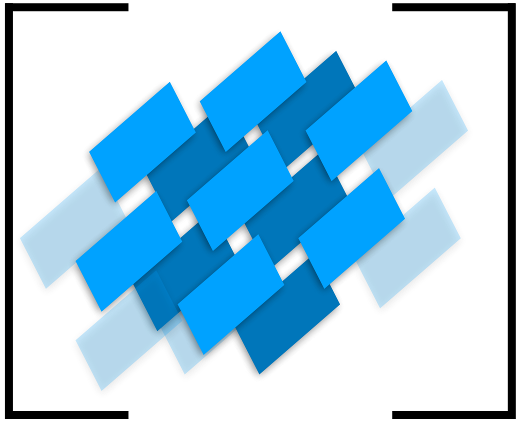


Bloom Filter: Repeated Trials

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

h_1	h_2	
0	0	0
1	0	**1**
0	0	**1**
1	1	**1**
0	0	0
0	0	0
0	0	**1**
1	1	**1**
0	0	0
1	1	**1**
1	0	**1**
0	0	0
1	1	**1**
0	1	0
1	1	**1**
1	0	**1**
0	0	0
1	1	**1**
0	0	0
1	0	**1**

h_3



Bloom Filter: Repeated Trials

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

h_1	h_2	h_3	...	h_k
0	0	0		0
1	0	1		1
0	0	1		1
1	1	1		1
0	0	0		1
0	0	0		1
0	0	1		0
1	1	1		0
0	0	0		0
1	1	1		1
1	0	1		0
0	0	0		1
1	1	1		0
0	1	0		0
1	1	1		1
1	0	1		1
0	0	0		1
1	1	1		1
0	0	0		1
1	0	1		1

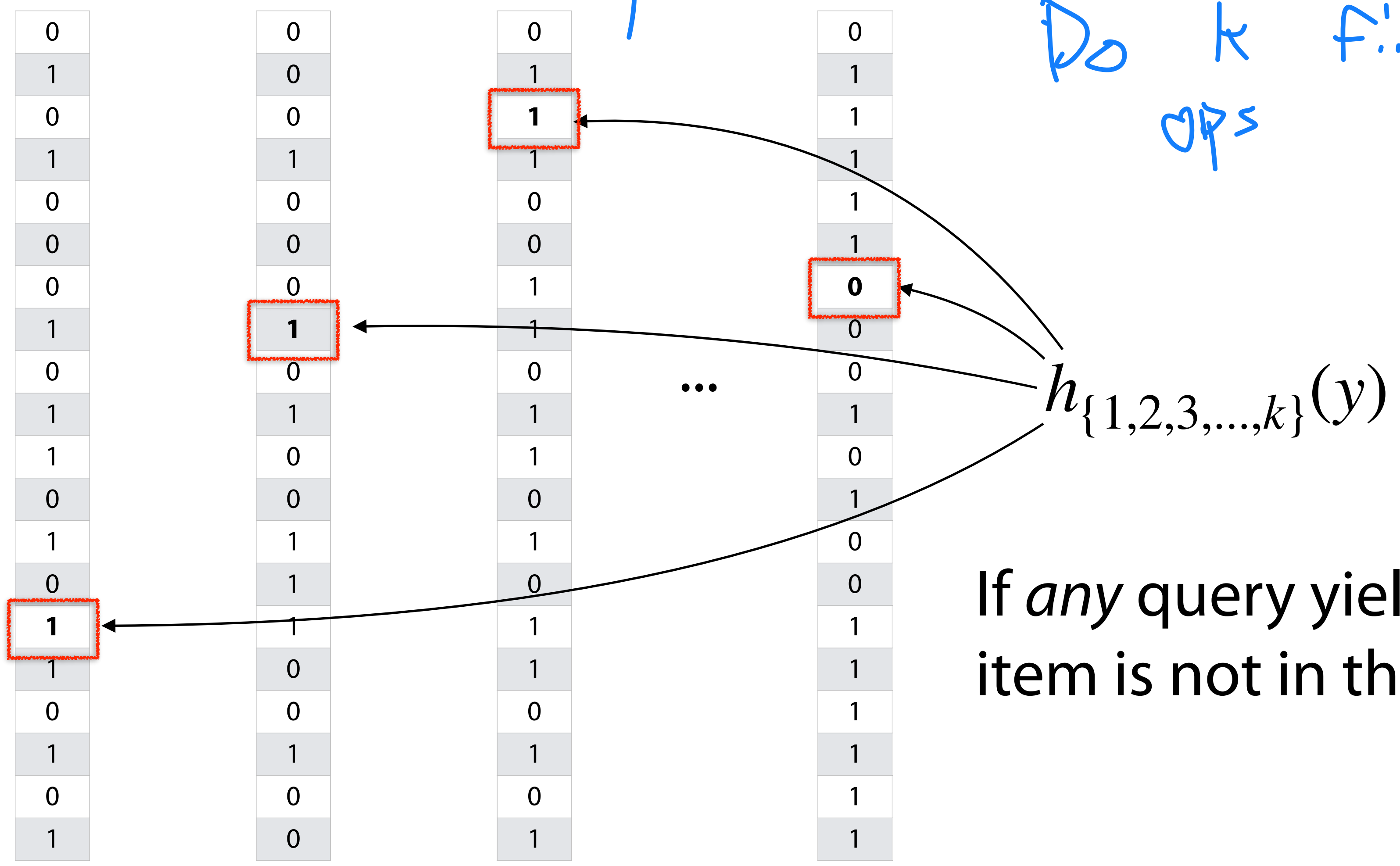
Bloom Filter: Repeated Trials

0	0	0		0
1	0	1		1
0	0	1		1
1	1	1		1
0	0	0		1
0	0	0		1
0	0	1		0
1	1	1		0
0	0	0	…	0
1	1	1		1
1	0	1		0
0	0	0		1
1	1	1		0
0	1	0		0
1	1	1		1
1	0	1		1
0	0	0		1
1	1	1		1
0	0	0		1
1	0	1		1

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

Bloom Filter: Repeated Trials

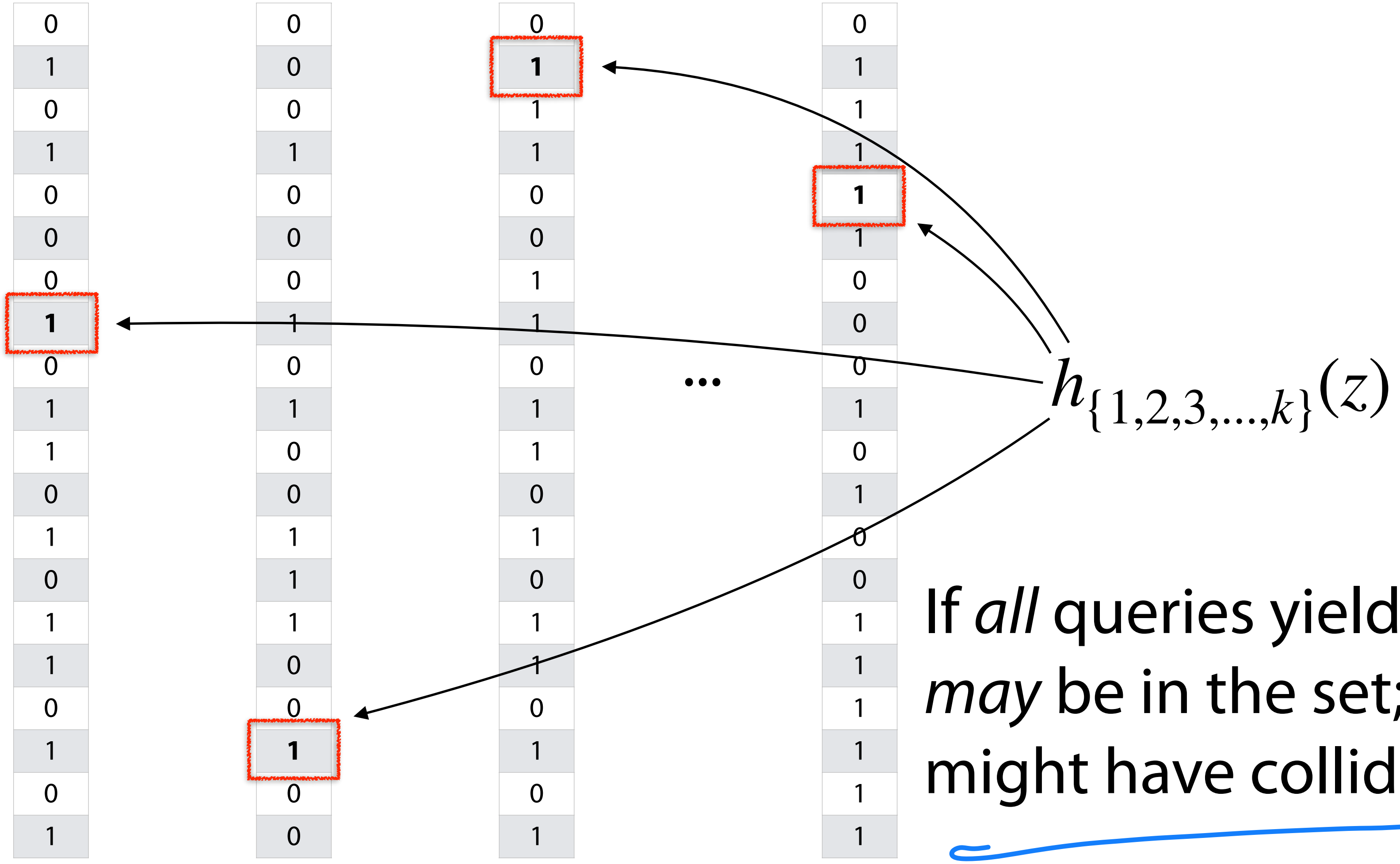
Do k find() ops



random collision

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

1
0

$k=10$

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?

$p^k \rightarrow p = 0.5 \quad (0.5)^{10} \approx 0.00097$

Bloom Filter: Repeated Trials

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

m

h_1	h_2	h_3	…	h_k
0	0	0		0
1	0	1		1
0	0	1		1
1	1	1		1
0	0	0		1
0	0	0		1
0	0	1		0
1	1	1		0
0	0	0		0
1	1	1		1
1	0	1		0
0	0	0		1
1	1	1		0
0	1	0		0
1	1	1		1
1	0	1		1
0	0	0		1
1	1	1		1
0	0	0		1
1	0	1		1

current storage

$k \cdot m$

can be good

vs (n)

Bloom Filter: Repeated Trials

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

S = { 6, 8, 4 }

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

$h_1(x) = x \% 10$	$h_2(x) = 2x \% 10$	$h_3(x) = (5+3x) \% 10$
6	2	3
8	6	9
4	8	7

1) Hash k times

2) Insert k times

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)` 1 2 8

1) Hash k times

2) Do k lookups

[find(1) → False]

`_find(16)` 6 2 3

↳ True

(This is false positive!)

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