CSE 312

Foundations of Computing II

Lecture 10: Bloom Filters

Announcements

- PSet 3 due today
- PSet 2 returned yesterday
- PSet 4 posted this evening
  - Last PSet prior to midterm (midterm is in exactly two weeks from now)
  - Midterm info will follow soon
  - PSet 5 will only come after the midterm in two weeks

Today

- An Application: Bloom Filters!

Basic Problem

Problem: Store a subset S of a large set U.

Example. $U =$ set of 128 bit strings $\quad |U| \approx 2^{128}$

$S =$ subset of strings of interest $\quad |S| \approx 1000$

Two goals:

1. **Very fast** (ideally constant time) answers to queries "Is $x \in S$?" for any $x \in U$.
2. **Minimal storage** requirements.

Naïve Solution I – Constant Time

Idea: Represent S as an array A with $|U|$ entries.

$$\mathrm{A}[x] = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{if } x \notin S \end{cases}$$

$S = \{0,2,\ldots,K\}$

0	1	2	…	K	…		
1	0	1	0	1	…	0	0

Membership test: To check. $x \in S$ just check whether $\mathrm{A}[x] = 1$.

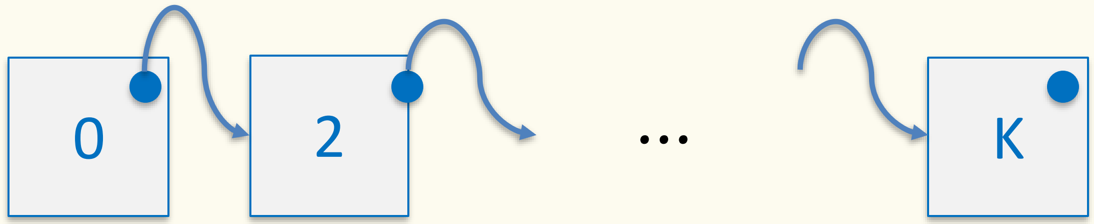
$\rightarrow$ **constant time!** 👍 😀

Storage: Require storing $|U|$ bits, even for small S. 👎 😢

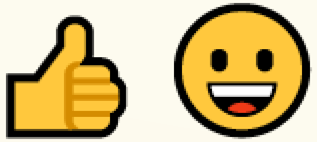
Naïve Solution II – Small Storage

Idea: Represent S as a list with $|S|$ entries.

$S = \{0,2,\dots,K\}$



Storage: Grows with $|S|$ <u>only</u>



Membership test: Check $x \in S$ requires time linear in $|S|$

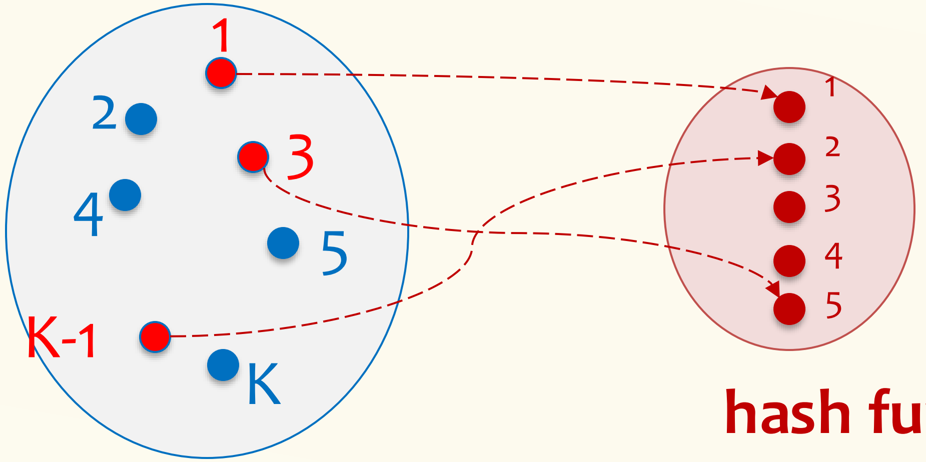
(Can be made logarithmic by using a tree)

Less naïve solution – Hash Table

Idea: Map elements in S into an array A of size m using a hash function $\mathbf{h}$

Membership test: To check $x \in S$ just check whether $A[\boldsymbol{h}(x)] = x$

Storage: m elements (size of array)



hash function $\boldsymbol{h}$: $U \to [m]$ $[m] = \{1, .., m\}$

Less naïve solution – Hash Table

Idea: Map elements in S into an array A of size m using a hash function $\mathbf{h}$

Membership test: To check $x \in S$ just check whether $A[\mathbf{h}(x)] = x$

Storage: m elements (size of array)

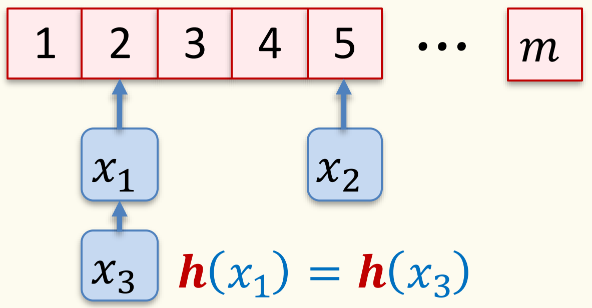
Challenge 1: Ensure $\boldsymbol{h}(x) \neq \boldsymbol{h}(y)$ *for most* $x, y \in S$

Challenge 2: Ensure $m = O(|S|)$

Hashing: collisions

Collisions occur when $\boldsymbol{h}(x) = \boldsymbol{h}(y)$ for some distinct $x, y \in S$, i.e., two elements of set map to the same location

- Common solution: <u>chaining</u> – at each location (bucket) in the table, keep linked list of all elements that hash there.



Good hash functions to keep collisions low

- The hash function h is good iff it
  - distributes elements uniformly across the m array locations so that
  - pairs of elements are mapped independently

"Universal Hash Functions" – see CSE 332

Hashing: summary

Hash Tables

- They store the data itself
- With a good hash function, the data is well distributed in the table and lookup times are small.
- However, they need at least as much space as all the data being stored, i.e., $m = \Omega(|S|)$

In some cases, $|S|$ is huge, or not known a-priori …

Can we do better!?

Bloom Filters

to the rescue

(Named after Burton Howard Bloom)

Bloom Filters – Main points

- Probabilistic data structure.
- Close cousins of hash tables.
  - But: Ridiculously space efficient
- Occasional errors, specifically false positives.

Bloom Filters

- Stores information about a set of elements $S \subseteq U$.
- Supports two operations:
  1. **`add`**(x) - adds $x \in U$ to the set S
  2. **`contains`**(x) – ideally: true if $x \in S$, false otherwise

Instead, relaxed guarantees:

- False $\rightarrow$ **definitely** not in S
- True $\rightarrow$ **possibly** in S

  [i.e. we could have *false positives*]

Bloom Filters – Why Accept False Positives?

- **Speed** – both `add` and `contains` very very fast.
- **Space** – requires a miniscule amount of space relative to storing all the actual items that have been added.
  - Often just 8 bits per inserted item!
- **Fallback mechanism** – can distinguish false positives from true positives with extra cost
  - Ok if mostly negatives expected + low false positive rate

Bloom Filters: Application

- Google Chrome has a database of malicious URLs, but it takes a long time to query.
- Want an in-browser structure, so needs to be efficient and be space-efficient
- Want it so that can check if a URL is in structure:
  - If return False, then definitely not in the structure (don’t need to do expensive database lookup, website is safe)
  - If return True, the URL may or may not be in the structure. Have to perform expensive lookup in this rare case.

Bloom Filters – More Applications

- Any scenario where space and efficiency are important.
- Used a lot in networking
- In distributed systems when want to check consistency of data across different locations, might send a Bloom filter rather than the full set of data being stored.
- Google BigTable uses Bloom filters to reduce disk lookups
- Internet routers often use Bloom filters to track blocked IP addresses.
- And on and on…

What you can't do with Bloom filters

- There is no delete operation
  - Once you have added something to a Bloom filter for S, it stays

- You can't use a Bloom filter to name any element of S

But what you ***can*** do makes them very effective!

Bloom Filters – Ingredients

Basic data structure is a $k \times m$ binary array - "the Bloom filter"

- k rows $t_1, \dots, t_k$, each of size m
- Think of each row as an m-bit vector

k different hash functions $\boldsymbol{h}_1, \dots, \boldsymbol{h}_k : U \to [m]$

Bloom Filters – Three operations

- Set up Bloom filter for $S = \emptyset$

function INITIALIZE(k, m)
  for $i = 1, \dots, k$: **do**
    t_i = new bit vector of m 0s

- Update Bloom filter for $S \leftarrow S \cup \{x\}$

function ADD(x)
  for $i = 1, \dots, k$: **do**
    $t_i[h_i(x)] = 1$

- Check if $x \in S$

function CONTAINS(x)
  return $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \cdots \wedge t_k[h_k(x)] == 1$

Bloom Filters - Initialization

Number of hash functions

Size of array associated to each hash function.

function INITIALIZE(k, m)

for $i = 1, \ldots, k$: **do**

t_i = new bit vector of m 0s

for each hash function, initialize an empty bit vector of size m

Bloom Filters: Example

Bloom filter of length $m = 5$ that uses $k = 3$ hash functions

function INITIALIZE(k, m)
    for $i = 1, \ldots, k$: **do**
        t_i = new bit vector of m 0s

Index $\rightarrow$	1	2	3	4	5
t_1	0	0	0	0	0
t_2	0	0	0	0	0
t_3	0	0	0	0	0

Bloom Filters: Add

$$\boxed{\begin{array}{l}\textbf{function } \text{ADD}(x) \\ \quad \textbf{for } i = 1, \ldots, k\textbf{: do} \\ \qquad \underbrace{t_i[h_i(x)]}_{} = 1\end{array}}$$

for each hash function h_i

Index into i-th bit-vector, at index produced by hash function and set to 1

$h_i(x) \rightarrow$ result of hash function h_i on x

Bloom Filters: Example

Bloom filter of length $m = 5$ that uses $k = 3$ hash functions

$$\textbf{function } \text{ADD}(x)$$
$$\quad \textbf{for } i = 1, \ldots, k\textbf{: do}$$
$$\quad\quad t_i[h_i(x)] = 1$$

add("thisisavirus.com")

h_1("thisisavirus.com") $\rightarrow$ 3

Index $\rightarrow$	1	2	3	4	5
t_1	0	0	0	0	0
t_2	0	0	0	0	0
t_3	0	0	0	0	0

Bloom Filters: Example

Bloom filter of length $m = 5$ that uses $k = 3$ hash functions

$$\textbf{function } \text{ADD}(x)$$
$$\quad \textbf{for } i = 1, \ldots, k\textbf{: do}$$
$$\qquad t_i[h_i(x)] = 1$$

add("thisisavirus.com")

h_1("thisisavirus.com") → 3

h_2("thisisavirus.com") → 2

Index →	1	2	3	4	5
t_1	0	0	1	0	0
t_2	0	0	0	0	0
t_3	0	0	0	0	0

Bloom Filters: Example

Bloom filter of length $m = 5$ that uses $k = 3$ hash functions

$$\begin{aligned} &\textbf{function } \text{ADD}(x) \\ &\quad \textbf{for } i = 1, \dots, k\textbf{: do} \\ &\qquad t_i[h_i(x)] = 1 \end{aligned}$$

add(“thisisavirus.com”)

h_1(“thisisavirus.com”) $\rightarrow$ 3

h_2(“thisisavirus.com”) $\rightarrow$ 2

h_3(“thisisavirus.com”) $\rightarrow$ 5

Index $\rightarrow$	1	2	3	4	5
t_1	0	0	1	0	0
t_2	0	1	0	0	0
t_3	0	0	0	0	0

Bloom Filters: Example

Bloom filter of length $m = 5$ that uses $k = 3$ hash functions

function ADD(x)
  for $i = 1, \ldots, k$: **do**
    $t_i[h_i(x)] = 1$

add("thisisavirus.com")

h_1("thisisavirus.com") $\rightarrow 3$

h_2("thisisavirus.com") $\rightarrow 2$

h_3("thisisavirus.com") $\rightarrow 5$

Index $\rightarrow$	1	2	3	4	5
t_1	0	0	1	0	0
t_2	0	1	0	0	0
t_3	0	0	0	0	1

Bloom Filters: Contains

function CONTAINS(x)
  return $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \cdots \wedge t_k[h_k(x)] == 1$

Returns True if the bit vector t_i for each hash function has bit 1 at index determined by $h_i(x)$,
Returns False otherwise

Bloom Filters: Example

Bloom filter of length $m = 5$ that uses $k = 3$ hash functions

function CONTAINS(x)
  return $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \cdots \wedge t_k[h_k(x)] == 1$

contains("thisisavirus.com")

Index $\rightarrow$	1	2	3	4	5
t_1	0	0	1	0	0
t_2	0	1	0	0	0
t_3	0	0	0	0	1

Bloom Filters: Example

Bloom filter of length $m = 5$ that uses $k = 3$ hash functions

function CONTAINS(x)
    return $t_1[h_1(x)] == 1 \land t_2[h_2(x)] == 1 \land \cdots \land t_k[h_k(x)] == 1$

True

contains("thisisavirus.com")

h_1("thisisavirus.com") $\rightarrow$ 3

Index $\rightarrow$	1	2	3	4	5
t_1	0	0	1	0	0
t_2	0	1	0	0	0
t_3	0	0	0	0	1

Bloom Filters: Example

Bloom filter of length $m = 5$ that uses $k = 3$ hash functions

function CONTAINS(x)
  return $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \cdots \wedge t_k[h_k(x)] == 1$

True True

contains("thisisavirus.com")

h_1("thisisavirus.com") $\rightarrow$ 3

h_2("thisisavirus.com") $\rightarrow$ 2

Index $\rightarrow$	1	2	3	4	5
t_1	0	0	1	0	0
t_2	0	1	0	0	0
t_3	0	0	0	0	1

Bloom Filters: Example

Bloom filter of length $m = 5$ that uses $k = 3$ hash functions

function CONTAINS(x)
  return $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \cdots \wedge t_k[h_k(x)] == 1$

True True True

contains("thisisavirus.com")

h_1("thisisavirus.com") $\rightarrow$ 3

h_2("thisisavirus.com") $\rightarrow$ 2

h_3("thisisavirus.com") $\rightarrow$ 5

Index $\rightarrow$	1	2	3	4	5
t_1	0	0	1	0	0
t_2	0	1	0	0	0
t_3	0	0	0	0	1

Bloom Filters: Example

Bloom filter of length $m = 5$ that uses $k = 3$ hash functions

function CONTAINS(x)
  return $t_1[h_1(x)] == 1 \land t_2[h_2(x)] == 1 \land \cdots \land t_k[h_k(x)] == 1$

True True True

contains("thisisavirus.com")

h_1("thisisavirus.com") $\rightarrow$ 3

h_2("thisisavirus.com") $\rightarrow$ 2

h_3("thisisavirus.com") $\rightarrow$ 5

Index $\rightarrow$	1	2	3	4	5
t_1	[illegible]	[illegible]	1	0	0
t_2	0	1	0	0	0
t_3	0	0	0	0	1

Since all conditions satisfied, returns True (correctly)

Bloom Filters: False Positives

Bloom filter of length $m = 5$ that uses $k = 3$ hash functions

add("totallynotsuspicious.com")

$$\textbf{function } \text{ADD}(x)$$
$$\quad \textbf{for } i = 1, \dots, k\text{: } \textbf{do}$$
$$\qquad t_i[h_i(x)] = 1$$

Index →	1	2	3	4	5
t_1	0	0	1	0	0
t_2	0	1	0	0	0
t_3	0	0	0	0	1

Bloom Filters: False Positives

Bloom filter of length $m = 5$ that uses $k = 3$ hash functions

$$
\begin{aligned}
&\textbf{function } \text{ADD}(x) \\
&\quad \textbf{for } i = 1, \dots, k\textbf{: do} \\
&\quad\quad t_i[h_i(x)] = 1
\end{aligned}
$$

add(“totallynotsuspicious.com”)

h_1(“totallynotsuspicious.com”) $\rightarrow$ 2

Index $\rightarrow$	1	2	3	4	5
t_1	0	0	1	0	0
t_2	0	1	0	0	0
t_3	0	0	0	0	1

Bloom Filters: False Positives

Bloom filter of length $m = 5$ that uses $k = 3$ hash functions

function ADD(x)
  for $i = 1, \ldots, k$: **do**
    $t_i[h_i(x)] = 1$

add("totallynotsuspicious.com")

h_1("totallynotsuspicious.com") → 2

h_2("totallynotsuspicious.com") → 1

Index →	1	2	3	4	5
t_1	0	1	1	0	0
t_2	0	1	0	0	0
t_3	0	0	0	0	1

Bloom Filters: False Positives

Bloom filter of length $m = 5$ that uses $k = 3$ hash functions

function ADD(x)
  for $i = 1, \dots, k$: **do**
    $t_i[h_i(x)] = 1$

add("totallynotsuspicious.com")

h_1("totallynotsuspicious.com") → 2

h_2("totallynotsuspicious.com") → 1

h_3("totallynotsuspicious.com") → 5

Index →	1	2	3	4	5
t_1	0	1	1	0	0
t_2	1	1	0	0	0
t_3	0	0	0	0	1

Bloom Filters: False Positives

Bloom filter of length $m = 5$ that uses $k = 3$ hash functions

function ADD(x)
    for $i = 1, \ldots, k$: **do**
        $t_i[h_i(x)] = 1$

add("totallynotsuspicious.com")

h_1("totallynotsuspicious.com") $\rightarrow$ 2

h_2("totallynotsuspicious.com") $\rightarrow$ 1

h_3("totallynotsuspicious.com") $\rightarrow$ 5

Index $\rightarrow$	1	2	3	4	5
t_1	0	1	1	0	0
t_2	1	1	0	0	0
t_3	0	0	0	0	1

Bloom Filters: False Positives

Bloom filter of length $m = 5$ that uses $k = 3$ hash functions

function CONTAINS(x)
  return $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \cdots \wedge t_k[h_k(x)] == 1$

contains("verynormalsite.com")

Index →	1	2	3	4	5
t_1	0	1	1	0	0
t_2	1	1	0	0	0
t_3	0	0	0	0	1

Bloom Filters: False Positives

Bloom filter of length $m = 5$ that uses $k = 3$ hash functions

function CONTAINS(x)
  return $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \cdots \wedge t_k[h_k(x)] == 1$

True

contains("verynormalsite.com")

h_1("verynormalsite.com") $\to$ 3

Index $\to$	1	2	3	4	5
t_1	0	1	1	0	0
t_2	1	1	0	0	0
t_3	0	0	0	0	1

Bloom Filters: False Positives

Bloom filter of length $m = 5$ that uses $k = 3$ hash functions

function CONTAINS(x)
  return $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \cdots \wedge t_k[h_k(x)] == 1$

True True

contains("verynormalsite.com")

h_1("verynormalsite.com") → 3

h_2("verynormalsite.com") → 1

Index →	1	2	3	4	5
t_1	0	1	1	0	0
t_2	1	1	0	0	0
t_3	0	0	0	0	1

Bloom Filters: False Positives

Bloom filter of length $m = 5$ that uses $k = 3$ hash functions

function CONTAINS(x)
    return $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \cdots \wedge t_k[h_k(x)] == 1$

True   True   True

contains("verynormalsite.com")

h_1("verynormalsite.com") → 3

h_2("verynormalsite.com") → 1

h_3("verynormalsite.com") → 5

Index →	1	2	3	4	5
t_1	0	1	1	0	0
t_2	1	1	0	0	0
t_3	0	0	0	0	1

Bloom Filters: False Positives

Bloom filter of length $m = 5$ that uses $k = 3$ hash functions

function CONTAINS(x)
  return $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \cdots \wedge t_k[h_k(x)] == 1$

True True True

contains("verynormalsite.com")

h_1("verynormalsite.com") $\rightarrow$ 3

h_2("verynormalsite.com") $\rightarrow$ 1

h_3("verynormalsite.com") $\rightarrow$ 5

Index $\rightarrow$	1	2	3	4	5
			1	0	0
t_2	1	1	0	0	0
t_3	0	0	0	0	1

Since all conditions satisfied, returns True (incorrectly)

Brain Break

Analysis: False positive probability

Question: For an element $x \in U$, what is the probability that `contains`(x) returns true if `add`(x) was never executed before?

Probability over what?! Over the choice of the $\boldsymbol{h}_1, \ldots, \boldsymbol{h}_k$

Assumptions for the analysis (somewhat stronger than for ordinary hashing):

- Each $\boldsymbol{h}_i(x)$ is uniformly distributed in $[m]$ for all x and i
- Hash function outputs for each $\boldsymbol{h}_i$ are mutually independent (not just in pairs)
- Different hash functions are independent of each other

False positive probability – Events

Assume we perform $\mathbf{add}(x_1), \ldots, \mathbf{add}(x_n)$
+ $\mathbf{contains}(x)$ for $x \notin \{x_1, \ldots, x_n\}$

Event E_i holds iff $\boldsymbol{h}_i(x) \in \{\boldsymbol{h}_i(x_1), \ldots, \boldsymbol{h}_i(x_n)\}$

$$P(\text{false positive}) = P(E_1 \cap E_2 \cap \cdots \cap E_k) = \prod_{i=1}^{k} P(E_i)$$

$\boldsymbol{h}_1, \ldots, \boldsymbol{h}_k$ independent

False positive probability – Events

Event E_i holds iff $\boldsymbol{h}_i(x) \in \{\boldsymbol{h}_i(x_1), \dots, \boldsymbol{h}_i(x_n)\}$

Event E_i^c holds iff $\boldsymbol{h}_i(x) \neq \boldsymbol{h}_i(x_1)$ and … and $\boldsymbol{h}_i(x) \neq \boldsymbol{h}_i(x_n)$

$$P(E_i^c) = \sum_{z=1}^{m} P(\boldsymbol{h}_i(x) = z) \cdot P(E_i^c \mid \boldsymbol{h}_i(x) = z)$$

LTP

False positive probability – Events

Event E_i^c holds iff $\boldsymbol{h}_i(x) \neq \boldsymbol{h}_i(x_1)$ and … and $\boldsymbol{h}_i(x) \neq \boldsymbol{h}_i(x_n)$

$$P(E_i^c \mid \boldsymbol{h}_i(x) = z) = P(\boldsymbol{h}_i(x_1) \neq z, \dots, \boldsymbol{h}_i(x_n) \neq z \mid \boldsymbol{h}_i(x) = z)$$

Independence of values of $\boldsymbol{h}_i$ on different inputs

$$= P(\boldsymbol{h}_i(x_1) \neq z, \dots, \boldsymbol{h}_i(x_n) \neq z)$$

$$= \prod_{j=1}^{n} P(\boldsymbol{h}_i(x_j) \neq z)$$

Outputs of $\boldsymbol{h}_i$ uniformly spread

$$= \prod_{j=1}^{n} \left(1 - \frac{1}{m}\right) = \left(1 - \frac{1}{m}\right)^n$$

$$\Rightarrow P(E_i^c) = \sum_{z=1}^{m} P(\boldsymbol{h}_i(x) = z) \cdot P(E_i^c \mid \boldsymbol{h}_i(x) = z) = \left(1 - \frac{1}{m}\right)^n$$

False positive probability – Events

Event E_i holds iff $\boldsymbol{h}_i(x) \in \{\boldsymbol{h}_i(x_1), \dots, \boldsymbol{h}_i(x_n)\}$

Event E_i^c holds iff $\boldsymbol{h}_i(x) \neq \boldsymbol{h}_i(x_1)$ *and … and* $\boldsymbol{h}_i(x) \neq \boldsymbol{h}_i(x_n)$

$$P(E_i^c) = \left(1 - \frac{1}{m}\right)^n$$

$$\Rightarrow \quad \mathrm{FPR} = \prod_{i=1}^{k} \left(1 - P(E_i^c)\right) = \left(1 - \left(1 - \frac{1}{m}\right)^n\right)^k$$

False Positivity Rate – Example

$$\text{FPR} = \left(1 - \left(1 - \frac{1}{m}\right)^{n}\right)^{k}$$

e.g., $n = 5{,}000{,}000$

$k = 30$

$m = 2{,}500{,}000$

→ FPR = 1.28%

Comparison with Hash Tables - Space

- Google storing 5 million URLs, each URL 40 bytes.
- Bloom filter with $k = 30$ and $m = 2{,}500{,}000$

Hash Table

(optimistic)
$5{,}000{,}000 \times 40B = 200\mathrm{MB}$

Bloom Filter

$2{,}500{,}000 \times 30 = 75{,}000{,}000$ bits

< 10 MB

Time

- Say avg user visits 102,000 URLs in a year, of which 2,000 are malicious.
- 0.5 seconds to do lookup in the database, 1ms for lookup in Bloom filter.
- Suppose the false positive rate is 3%

0.5 seconds DB lookup

false positives

$$1\text{ms} + \frac{100000 \times 0.03 \times 500\text{ms} + 2000 \times 500 \text{ ms}}{102000} \approx 25.51\text{ms}$$

total URLs

malicious URLs

Bloom filter lookup

Bloom Filters typical of….

… randomized algorithms and randomized data structures.

- **Simple**
- **Fast**
- **Efficient**
- **Elegant**
- **Useful!**