Fingerprinting and Bloom Filters

Paolo Giaccone

Notes for the class on "Computer-aided simulations lab"

Politecnico di Torino

December 2022

Outline

Section 1

Set Membership

Set Membership

Set membership problem

Given a set of elements $S = \{s_1, \ldots, s_m\}$ on a universe U (i.e., $S \subseteq U$), determine, given some $x \in U$, whether $x \in S$:

1. if $x \in S$, return TRUE always
2. if $x \notin S$, return FALSE always

- storage is $m \times L$ where L is the average size of each element

Possible solutions

- compare x with all elements in S: $O(m)$ search
- keep sorted elements of S and perform binary search: $O(\log(m))$ search
- hash tables (traditional, multiple-choice, cuckoo) storing $(x, 1)$: $O(1)$ search

Approximated Set Membership

Approximated set membership problem

Given a set of elements $S = \{s_1, \ldots, s_m\}$ on a universe U (i.e., $S \subseteq U$), determine, given some $x \in U$, whether $x \in S$:

1. if $x \in S$, return TRUE always
2. if $x \notin S$, return FALSE with high probability

- false positive when the algorithm returns TRUE even if $x \notin S$
- storage is $\theta(m)$, independently from the size of the stored elements

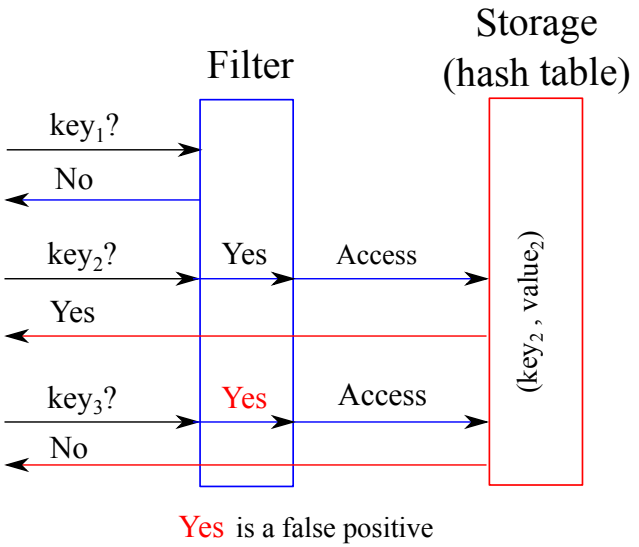
Solutions

1. Bit string hashing
2. Bloom filters
3. Cuckoo filters

Filtering storage access

Instead of accessing directly the hash table, determine if the key is already inside the hash or not (set membership problem).

The filter may lead to false positives but they are not a problem.



Possible applications:

- when the hash table is too large and it is expensive to access it in terms of time or bandwidth, the filter can be used to understand if it worth to check the hash table
- the hash table may be stored in different entities in the network
- data deduplication in data centers

Section 2

Fingerprinting

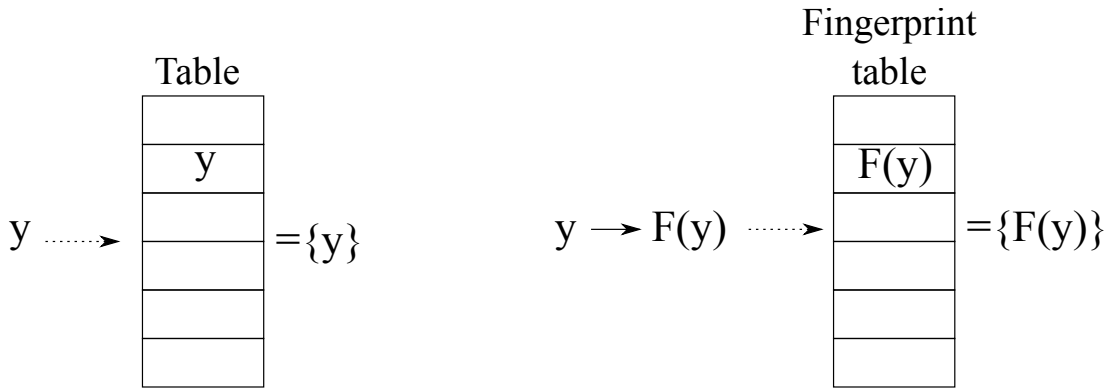
Fingerprinting

Instead of storing $y \in S$, store its fingerprint $F(y)$ (which is actually a hash function)

- y generic/variable number of bits for its representation
- $F(y)$ fixed number of bits (say b)
- total storage is $m \times b$ for a standard fingerprint table

Storage policy

- For every element $y \in S$, store only $F(y)$ (i.e., y's fingerprint)
- To check whether $x \in S$, check if $F(x)$ is in the fingerprint table
- Use the data structures considered so far



False positives

When two or more fingerprints collide (i.e., $F(x) = F(y)$ for $x \neq y$), fingerprinting may lead to false positives

Toy example

- assume to store only x, thus the table will be $T = \{F(x)\}$
- checking if y is in the table will give a false positive answer if $F(x) = F(y)$

Deletion support

To support deletion and keep the false-positive property, if two elements x and y with the same fingerprints must be stored ($F(x) = F(y)$), two copies of the fingerprint must be stored

- otherwise (just one copy), if $F(x)$ was deleted, also y would be deleted and then false-negative events would occur

Example of fingerprinting - without deletion support

Assume (i) $F(x)$ maps any string to a 8-bit value $\in [0, 255]$, (ii) only one copy of the same fingerprint is stored

Fingerprint map

x	$F(x)$
"Andrea Bianco"	34
"Carla Fabiana Chiasserini"	11
"Emilio Leonardi"	70
"Marco Mellia"	34
"Marco Ajmone Marsan"	121
"Paolo Giaccone"	121
"Politecnico di Torino"	78

Insertion operations

x	Table state
"Paolo Giaccone"	$\{121\}$
"Andrea Bianco"	$\{121, 34\}$
"Politecnico di Torino"	$\{121, 78, 34\}$
"Marco Mellia"	$\{121, 78, 34\}$

Final table: $\{121, 78, 34\}$

- Search("Paolo Giaccone") returns TRUE
- Search("Marco Mellia") returns TRUE
- Search("Emilio Leonardi") returns FALSE
- Search("Marco Ajmone Marsan") returns TRUE (i.e., false positive)

Example of fingerprinting - with deletion support

Assume (i) $F(x)$ maps any string to a 8-bit value $\in [0, 255]$, (ii) many copies of the same fingerprint are stored for different contents

Fingerprint map

x	$F(x)$
"Andrea Bianco"	34
"Carla Fabiana Chiasserini"	11
"Emilio Leonardi"	70
"Marco Mellia"	34
"Marco Ajmone Marsan"	121
"Paolo Giaccone"	121
"Politecnico di Torino"	78

Insertion operations

x	Table state
"Paolo Giaccone"	$\{121\}$
"Andrea Bianco"	$\{121, 34\}$
"Politecnico di Torino"	$\{121, 78, 34\}$
"Marco Mellia"	$\{121, 78, 34, 34\}$

Final table: $T = \{121, 78, 34, 34\}$

- Delete("Andrea Bianco") and then $T = \{121, 78, 34\}$
- Search("Andrea Bianco") returns TRUE (i.e., false positive)
- Search("Marco Mellia") returns TRUE
- Delete("Marco Mellia") and then $T = \{121, 78\}$
- Search("Andrea Bianco") and Search("Marco Mellia") return FALSE

Fingerprint size

Key question

How many bits for the fingerprint?

- if too small, many false positive
- if too large, space waste

Note that the fingerprint sizing is independent from the data structure adopted for storage

Fingerprint size

Probability of false positive

If m elements are stored through a fingerprint with range $F(\cdot) \in [0, n)$:

$$Pr(\text{false positive}) = 1 - \left(1 - \frac{1}{n}\right)^m \quad (1)$$

Since the possible values for the fingerprint are n, the collision probability between two distinct elements $x \in S$ and $y \in S$, with $x \neq y$, is:

$$\Pr(F(x) = F(y)) = \frac{1}{n}$$

Assume that m elements are already stored in set S. Now consider x outside S ($x \notin S$). To avoid false positive for x, $F(x)$ must be different from any $F(y)$, $\forall y \in S$, and thus

$$Pr(\text{false positive}) = 1 - \left(1 - \frac{1}{n}\right)^m$$

Fingerprint sizing

Minimum number of bits per fingerprint

To store m elements with $Pr(\text{false positive}) \leq \epsilon$, for some small $\epsilon > 0$, the number of bits b for the fingerprint must satisfy:

$$b \geq \log_2 \frac{m}{\epsilon} \quad \text{bits per fingerprint}$$

i.e., the range of the fingerprint $F(\cdot) \in [0, n)$ must satisfy:

$$n \geq \frac{m}{\epsilon}$$

Fingerprint sizing – proof

Given some small ϵ, thanks to (1):

$$\Pr(\text{false positive}) = 1 - \left(1 - \frac{1}{n}\right)^m \leq \epsilon$$

For large n,

$$1 - \frac{1}{n} \approx e^{-\frac{1}{n}}$$

and (1) becomes:

$$1 - \epsilon \leq e^{-\frac{m}{n}} \qquad \Rightarrow \qquad \log(1 - \epsilon) \leq -\frac{m}{n}$$

By approximating $\log(1 - \epsilon) \approx -\epsilon$,

$$-\epsilon \leq -\frac{m}{n} \qquad \Rightarrow \qquad n \geq \frac{m}{\epsilon}$$

Finally,

$$b = \log_2 n \geq \log_2 \frac{m}{\epsilon}$$

Special cases for fingerprint

Negligible probability of false positive

Let $\alpha > 0$, if

$$\Pr(\text{false positive}) \leq \frac{1}{m^{\alpha}}$$

then

$$b \geq (\alpha + 1)\log_2 m \text{ bits per fingerprint} \quad \Rightarrow \quad b_{\min} = \theta(\log_2 m)$$

$$F(\cdot) \in [0, n) \quad \text{with} \quad n \geq m^{\alpha+1}$$

Non-sufficient fingerprint

If $b = \log_2 m$, then $n = m$ and

$$Pr(\text{false positive}) = 1 - \left(1 - \frac{1}{m}\right)^{m} = 1 - e^{-1} \approx 0.63$$

Storage for a fingerprint table

Fingerprint table: each whole fingerprint is stored in a table (e.g., hash table, binary tree, etc.)

Total storage for a fingerprint table

Assume b is the minimum number of bits in the fingerprint to get a probability of false positive equal to ϵ. To store m fingerprints on a table the total storage will be

$$m \times b = m \times \log_2 \frac{m}{\epsilon}$$

independently from the universe size $|U|$

Section 3

Bit String Hashing

Bit string hashing

Data structure is bit array BA of size n:

- $BA[i] \in \{0, 1\}$, for $i = 0, \ldots, n-1$
- initialization: $BA[i] = 0$ for all i
- use hash function $h(\cdot) \in [0, n-1]$ for fingerprinting

Insertion

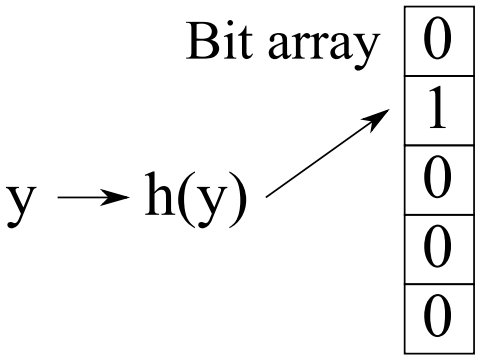
Add y into S

- set $BA[h(y)] = 1$

Search

Check if $x \in S$

- if $BA[h(x)] = 1$, return TRUE
  - may lead to FALSE POSITIVE
- otherwise, return FALSE



Bit string hashing: storage

Total storage

- let b be the number of bits for each fingerprint
- $h(y) \in [0, 2^b - 1]$
- total storage is $n = 2^b$ bits

Required storage for bit string hashing

Let m be the number of elements to store. By combining previous results:

Storage size

1. with bit array of size $n = m$ bits, then Pr(false positive) ≈ 0.63
   - indeed, if $b = \log_2 m$ bit/fingerprint, then $n = 2^b = m$ bits
2. with bit array of size $n = m/\epsilon$ bits, then Pr(false positive) $= \epsilon$
   - indeed, if $b = \log_2(m/\epsilon)$ bit/fingerprint, then $n = 2^b = m/\epsilon$ bits
3. with bit array of size $n = m^2$ bits, then Pr(false positive) $\approx \frac{1}{m}$
   - indeed, if $b = 2\log_2 m$ bit/fingerprint, then $n = 2^b = m^2$ bits

Section 4

Bloom filters

Bloom filters

Main idea

Instead of using one hash function as in bit string hashing, use k independent hash function $h_1, \ldots, h_k$ each with range $[1, n]$

Data structure is bit array BF of size n:

- $BF[i] \in \{0, 1\}$, for $i = 1, \ldots, n$
- initialization: $BF[i] = 0$ for all i

Operations

Insertion

Add y into S

- $BF[h_j(y)] = 1$, for $j = 1, \ldots, k$
  - independently from the previous value of $BF[h_j(y)]$

function INSERT(x)
  for $i = 1 \rightarrow k$ **do**
    $BF[h_j(x)] = 1$
  end for
end function

Operations

Search

Check if $x \in S$

- evaluate all k hash function $h_j(x)$
- only if all the corresponding bits in BF are 1, then $x \in S$
  - i.e., $BF[h_j(x)] = 1$ for all $j = 1 \dots k$

```
function SEARCH(x)
    ans=TRUE
    for j = 1 → k do
        if BF[h_j(x)] = 0 then
            ans=FALSE
            break
        end if
    end for
    return ans
end function
```

Operations

Union

Union of BF_1 and BF_2

- easily computed by: $BF_1[i]$ OR $BF_2[i]$, for all i

```
function UNION(BF1, BF2)
    for i = 1 → n do
        BF[i] = BF1[i] OR BF2[i]
    end for
    return BF
end function
```

Deletion

Deleting an element is not possible

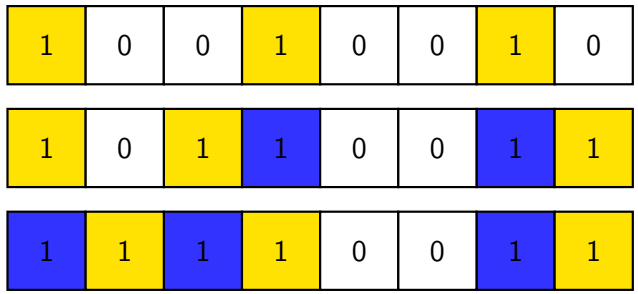
- other data structures must be considered (e.g., *counting* Bloom filters, Cuckoo filters)

Example

Data insert:

x	$h_1(x)$	$h_2(x)$	$h_3(x)$
x_1	1	4	7
x_2	1	3	8
x_3	2	4	8

Bloom filter BF



$S = \{x_1\}$

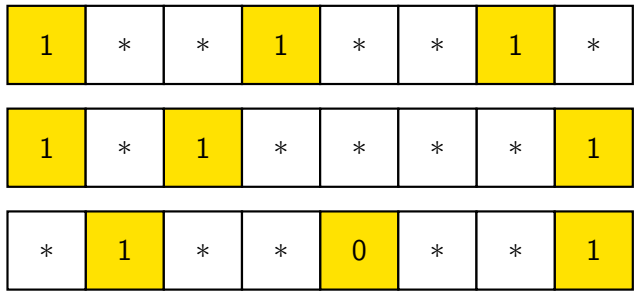
$S = \{x_1, x_2\}$

$S = \{x_1, x_2, x_3\}$

Search:

x	$h_1(x)$	$h_2(x)$	$h_3(x)$
x_1	1	4	7
x_4	1	3	8
x_5	2	5	8

Bloom filter BF



TRUE

FALSE POSITIVE

FALSE

Performance

Assume

- Bloom filter of length n bits
- k hash functions
- m elements already stored

The probability that a certain bit is 0 is

$$p = \left(1 - \frac{1}{n}\right)^{mk} \approx e^{-km/n}$$

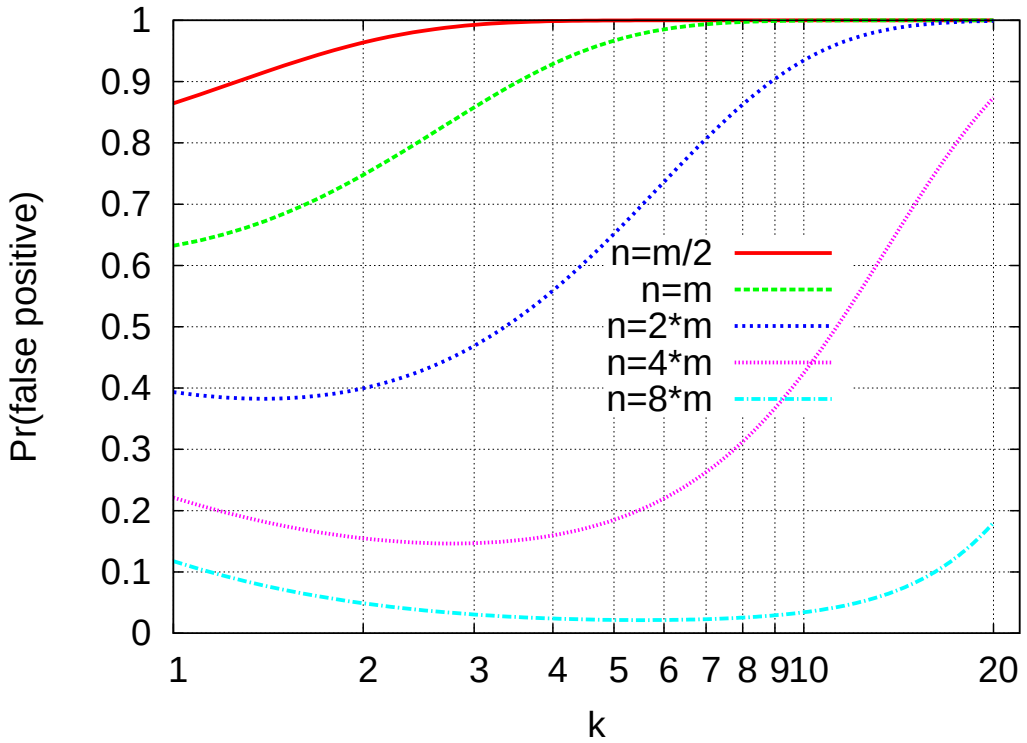
Assume that exactly np of the bits are 0 and $n(1-p)$ are 1

- good approximation thanks to Azuma-Hoeffding inequality for martingales

Performance

Probability of false positive

$$\Pr(\text{false positive}) = (1-p)^k = \left(1 - \left(1 - \frac{1}{n}\right)^{mk}\right)^k \approx \left(1 - e^{\frac{-km}{n}}\right)^k$$



Optimal design of Bloom filters

Given m, n, what is the optimal k that minimizes Pr(false positive)?

- small k increases the fraction of 0 bits in the arrays, available for an element that is not a member of S
- large k increases the probability of finding at least a 0 bit for an element that is not a member of S

Using some algebra:

$$\frac{d\,\Pr(\text{false positive})}{dk} = 0 \qquad \Rightarrow \qquad k_{opt} = \frac{n}{m}\log(2)$$

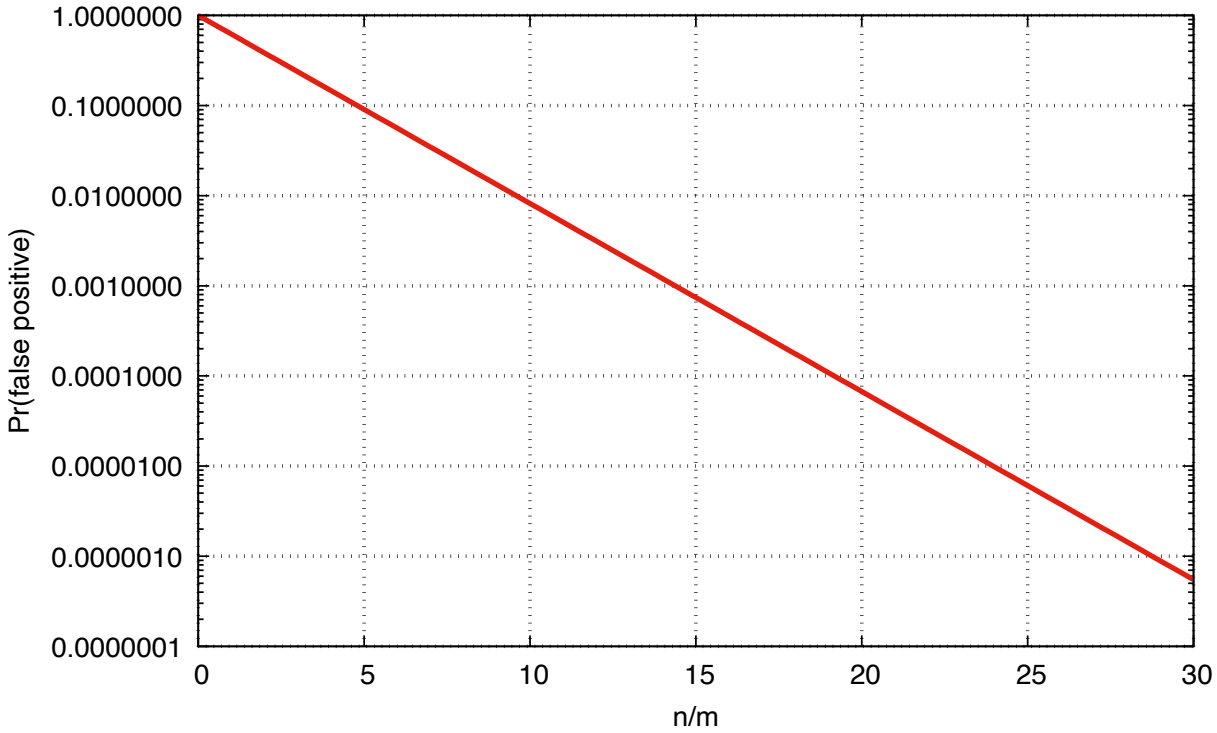
Optimal number of hash functions

$$k_{opt} = \frac{n}{m}\log(2)$$

Performance for optimally designed Bloom filters

Optimal performance

$$\Pr(\text{false positive}) = \left(1 - e^{\frac{-k_{opt}\, m}{n}}\right)^{k_{opt}} = \left(\frac{1}{2}\right)^{k_{opt}} = (0.6185)^{n/m}$$



Performance for optimally designed Bloom filters

By setting in the above formula Pr(false positive) $= \epsilon$, we obtain

$$(0.6185)^{n/m} = \epsilon \;\Rightarrow\; \frac{n}{m}\log_2(0.6185) = \log_2 \epsilon \;\Rightarrow\; \frac{n}{m} = \frac{-\log_2 \epsilon}{-\log_2(0.6195)}$$

Hence we can claim

Required storage for the whole filter

$$n = m \times 1.44 \log_2(1/\epsilon) \text{ bits}$$

Amortized space cost per item

$$n/m = 1.44 \log_2(1/\epsilon) \text{ bits per content}$$

which is independent from the number of stored elements

Near-to-optimality of Bloom filters

Bloom filter

To obtain Pr(false positive) $= \epsilon$, the total storage is $n = m \times 1.44 \log_2(1/\epsilon)$ bits for an optimally designed Bloom filter

We provide the following property, without proof:

Lower bound for storage

To obtain Pr(false positive) $= \epsilon$, the total storage is at least $n = m \times \log_2(1/\epsilon)$ bits

Thus, the storage of Bloom filters is within a 1.44 factor with respect to the lower bound

Bloom filter occupancy

Number of stored elements

It can be shown that the number of distinct elements stored in a bloom filter with n bits and k hash functions can be estimated as

$$-\frac{n}{k} \ln\left(1 - \frac{N_1}{n}\right)$$

where N_1 is the actual number of bits equal to 1 in the bloom filter.

Hashing vs. Bloom filters

Storage comparison

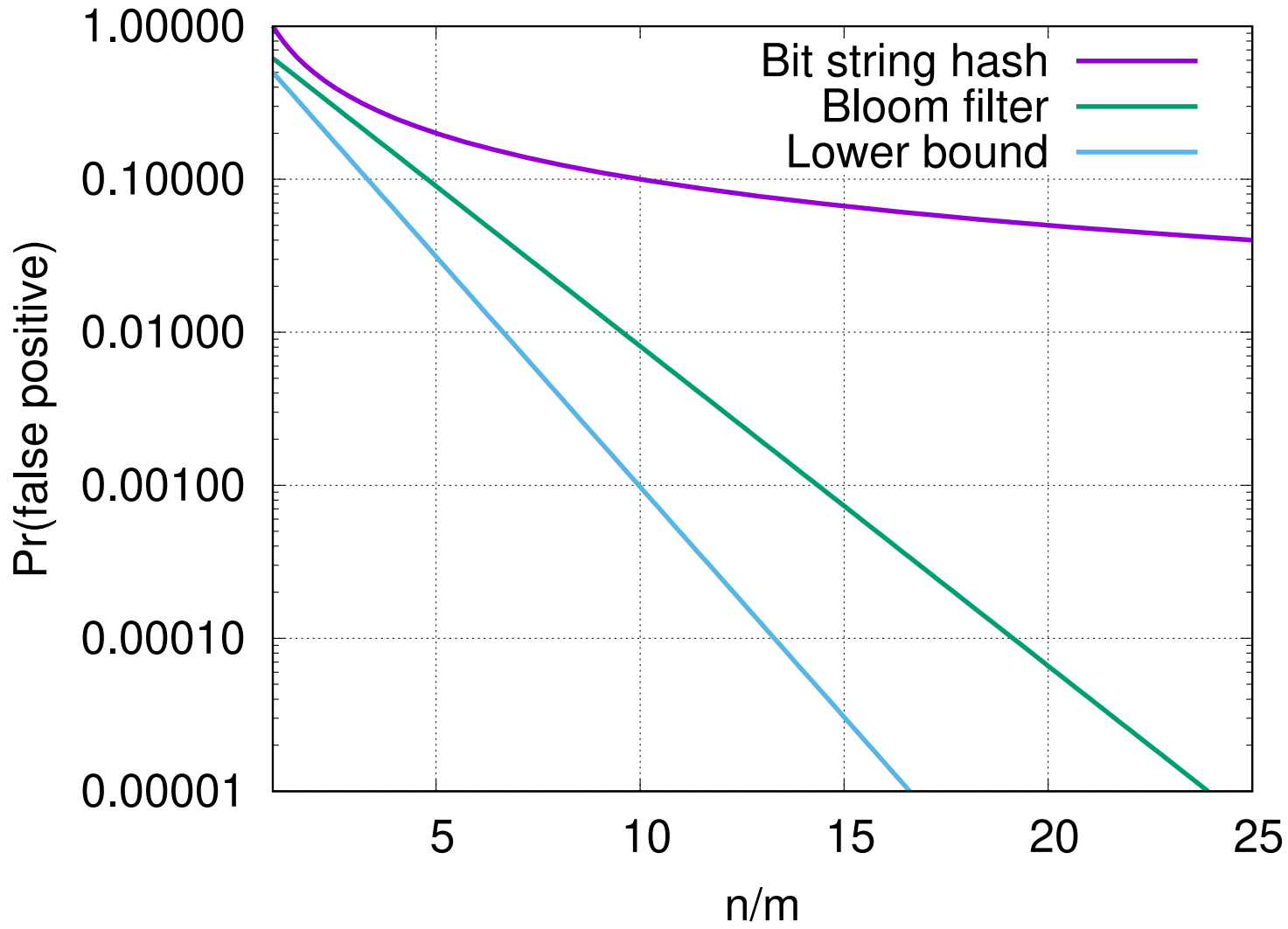
Given

- m is the number of elements to store
- Pr(false positive) $= \epsilon$

Method	Storage requirement n [bit]
Bit string hash	$m \times (1/\epsilon)$
Bloom filter	$m \times 1.44 \log_2(1/\epsilon)$
Lower bound	$m \times \log_2(1/\epsilon)$

In both bit string hashes and bloom filters, the storage grows as $\theta(m)$, independently from the universe size $|U|$.

Hashing vs. Bloom filters



Overall storage comparison

- m is the number of elements to store
- L is the average size of each elements (in bits)

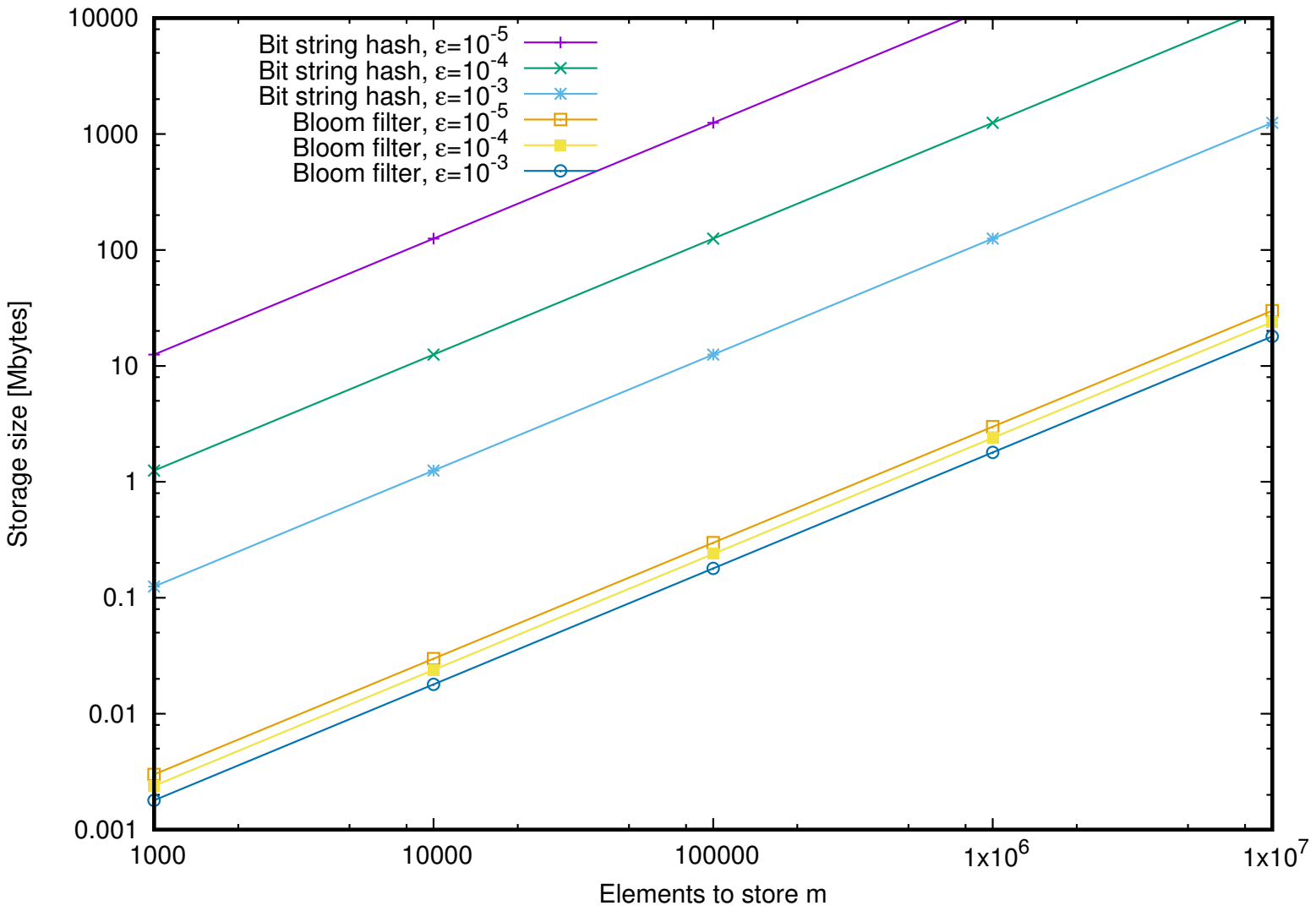
Scheme	Prob. false positive	Total storage [bit]
Store element in a table	0	$m \times L$
Store fingerprint in a table	ϵ	$m \times \log_2 \frac{m}{\epsilon}$
Store fingerprint in a bit string array	ϵ	$m \times \frac{1}{\epsilon}$
Bloom filter	ϵ	$m \times 1.44 \log_2 \frac{1}{\epsilon}$
Lower bound	ϵ	$m \times \log_2 \frac{1}{\epsilon}$

Examples of design

Universe U with $|U| \approx 8\ 10^{28}$ elements

1. $m = 1,000$ elements to store, Pr(false positive) $= 10^{-3}$
   - bit string hash: $n = 10^6$ bits (125 kbytes)
   - Bloom filter: $n = 15\ 10^3$ bits (1.9 kbytes)
2. $m = 10,000$ elements to store, Pr(false positive) $= 10^{-4}$
   - bit string hash: $n = 10^8$ bits (12.5 Mbytes)
   - Bloom filter: $n = 191\ 10^3$ bits (23.9 kbytes)

Examples of design



Overall storage comparison

- m is the number of elements to store
- L is the average size of each elements (in bits)

Scheme	Prob. false positive	Total storage [bit]
Store element in a table	0	$m \times L$
Store fingerprint in a table	ϵ	$m \times \log_2(m/\epsilon)$
Store fingerprint in a bit string array	ϵ	$m \times 1/\epsilon$
Bloom filter	ϵ	$m \times 1.44 \log_2(1/\epsilon)$
Cuckoo filter	ϵ	$m \times (\log_2(1/\epsilon) + 3)$
Lower bound	ϵ	$m \times \log_2 1/\epsilon$