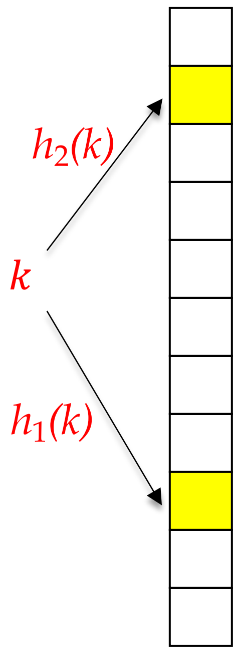
Cuckoo hashing

(prefect hashing with open addressing)

Cuckoo hashing

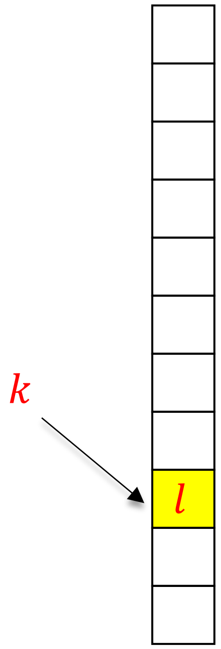
- Introduced by Pagh&Rodler in 2001
- uses two independent hash functions h_1 and h_2
- LOOKUP(x): check buckets $T[h_1(x)]$ and $T[h_2(x)]$



Cuckoo hashing

- Introduced by Pagh&Rodler in 2001
- uses two independent hash functions h_1 and h_2
- LOOKUP(x): check buckets $T[h_1(x)]$ and $T[h_2(x)]$
- INSERT(x):

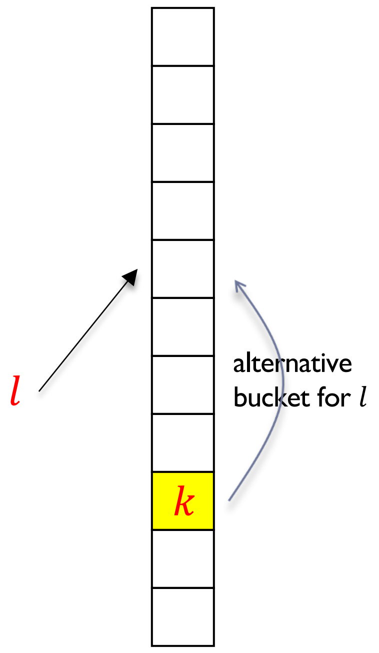
```
$pos \leftarrow h_1(x)$
**loop**
    **if** $T[pos]$ is empty **then**
        $T[pos] \leftarrow x$; **return**
    **swap** values of $x$ and $T[pos]$
    $pos \leftarrow$ alternative position for $x$
```



Cuckoo hashing

- Introduced by Pagh&Rodler in 2001
- uses two independent hash functions h_1 and h_2
- LOOKUP(x): check buckets $T[h_1(x)]$ and $T[h_2(x)]$
- INSERT(x):

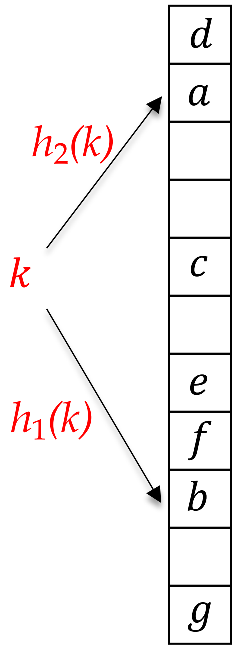
```
$pos \leftarrow h_1(x)$
loop
    if $T[pos]$ is empty then
        $T[pos] \leftarrow x$; return
    swap values of $x$ and $T[pos]$
    $pos \leftarrow$ alternative position for $x$
```



Cuckoo hashing

- Introduced by Pagh&Rodler in 2001
- uses two independent hash functions h_1 and h_2
- LOOKUP(x): check buckets $T[h_1(x)]$ and $T[h_2(x)]$
- INSERT(x):

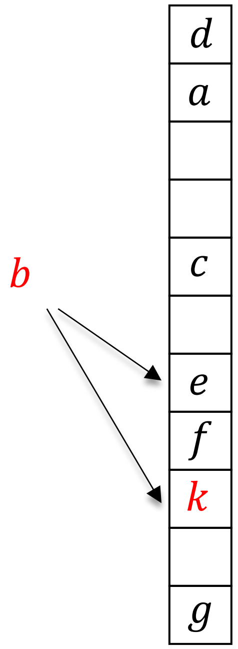
```
$pos \leftarrow h_1(x)$
**loop**
    **if** $T[pos]$ is empty **then**
        $T[pos] \leftarrow x$; **return**
    **swap** values of $x$ and $T[pos]$
    $pos \leftarrow$ alternative position for $x$
```



Cuckoo hashing

- Introduced by Pagh&Rodler in 2001
- uses two independent hash functions h_1 and h_2
- LOOKUP(x): check buckets $T[h_1(x)]$ and $T[h_2(x)]$
- INSERT(x):

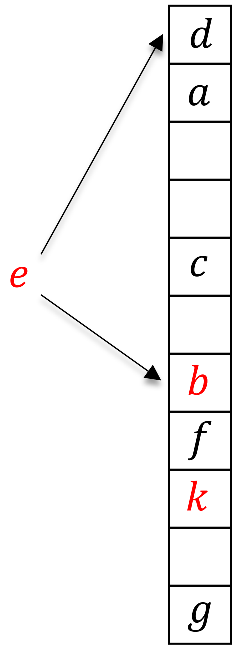
```
$pos \leftarrow h_1(x)$
loop
    if $T[pos]$ is empty then
        $T[pos] \leftarrow x$; return
    swap values of $x$ and $T[pos]$
    $pos \leftarrow$ alternative position for $x$
```



Cuckoo hashing

- Introduced by Pagh&Rodler in 2001
- uses two independent hash functions h_1 and h_2
- LOOKUP(x): check buckets $T[h_1(x)]$ and $T[h_2(x)]$
- INSERT(x):

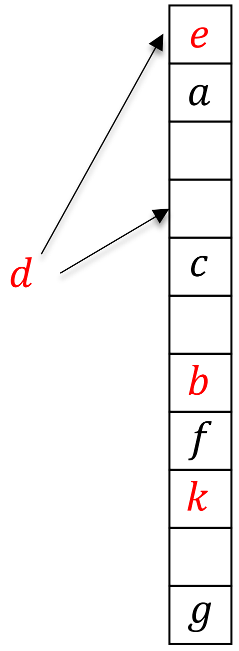
```
$pos \leftarrow h_1(x)$
**loop**
    **if** $T[pos]$ is empty **then**
        $T[pos] \leftarrow x$; **return**
    **swap** values of $x$ and $T[pos]$
    $pos \leftarrow$ alternative position for $x$
```



Cuckoo hashing

- Introduced by Pagh&Rodler in 2001
- uses two independent hash functions h_1 and h_2
- LOOKUP(x): check buckets $T[h_1(x)]$ and $T[h_2(x)]$
- INSERT(x):

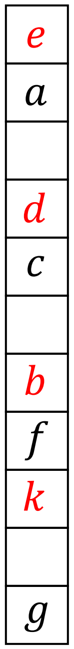
```
$pos \leftarrow h_1(x)$
**loop**
    **if** $T[pos]$ is empty **then**
        $T[pos] \leftarrow x$; **return**
    **swap** values of $x$ and $T[pos]$
    $pos \leftarrow$ alternative position for $x$
```



Cuckoo hashing

- Introduced by Pagh&Rodler in 2001
- uses two independent hash functions h_1 and h_2
- LOOKUP(x): check buckets $T[h_1(x)]$ and $T[h_2(x)]$
- INSERT(x):

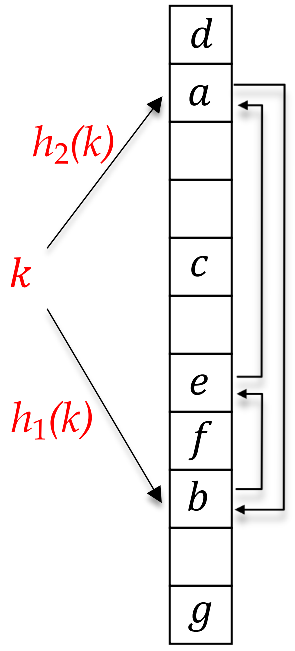
```
$pos \leftarrow h_1(x)$
loop
    if $T[pos]$ is empty then
        $T[pos] \leftarrow x$; return
    swap values of $x$ and $T[pos]$
    $pos \leftarrow$ alternative position for $x$
```



Cuckoo hashing

- Introduced by Pagh&Rodler in 2001
- uses two independent hash functions h_1 and h_2
- LOOKUP(x): check buckets $T[h_1(x)]$ and $T[h_2(x)]$
- INSERT(x):

```
$pos \leftarrow h_1(x)$
loop $n$ times
    if $T[pos]$ is empty then
        $T[pos] \leftarrow x$; return
    swap values of $x$ and $T[pos]$
    $pos \leftarrow$ alternative position for $x$
rehash from scratch
```



Cost of insertions: How many iterations?

- Random graphs:
  - nodes = buckets (m)
  - edges = inserted elements (n)
- If $n < m/2$, the graph contains very small connected components without cycles w.h.p.
- *Theorem*: if $n < \frac{m}{2(1+\delta)}$, then P[shortest path from i to j is of length d]$\leq \frac{1}{m}(1+\delta)^{-d}$
- $\Rightarrow$ P[j is accessible from i]$= O(\frac{1}{m})$
- $\Rightarrow$ number of accessible nodes is $n \cdot O\left(\frac{1}{m}\right) = O(1)$

Proof of the Theorem

- $d = 1$
  - P[an edge connects i and j] $\leq \frac{2}{m^2}$ $\Rightarrow$ P[exists an edge between i and j]$\leq \frac{2n}{m^2} < \frac{1}{m}(1+\delta)^{-1}$
- *induction:* $d \Rightarrow d+1$
  - for a fixed k, $\frac{1}{m}(1+\delta)^{-d} \cdot \frac{1}{m}(1+\delta)^{-1} = \frac{1}{m^2}(1+\delta)^{-(d+1)}$
  - summing over m possibilities for k, we obtain $\frac{1}{m}(1+\delta)^{-(d+1)}$

Cost of rehashing: how likely is a cycle?

- If $n < \frac{m}{2(1+\delta)}$, careful analysis shows that the probability of a rehash is $O\left(\frac{1}{n^2}\right)$
- A rehash involves n insertions, each taking expected $O(1)$ time $\Rightarrow$ amortized cost of rehashing is $O\left(\frac{1}{n}\right)$ time per insertion
- for more details see

  https://www.itu.dk/people/pagh/papers/cuckoo-undergrad.pdf

  http://www.cs.utoronto.ca/~noahfleming/CuckooHashing.pdf

Cuckoo hashing: summary

- LOOKUP, DELETE: worst-case $O(1)$ (two probes)
- INSERT: expected $O(1)$

- deletions supported
- no dynamic memory allocation (as in chaining)
- reasonable memory use, but load factor $< 1/2$
- generalization: (H, b)-cuckoo hashing
  - H hash functions (instead of 2), each bucket carries b items
  - admits a much higher load factor, e.g. (3,4)-tables admits load factor of over 99.9% [Walzer, ICALP 2018]

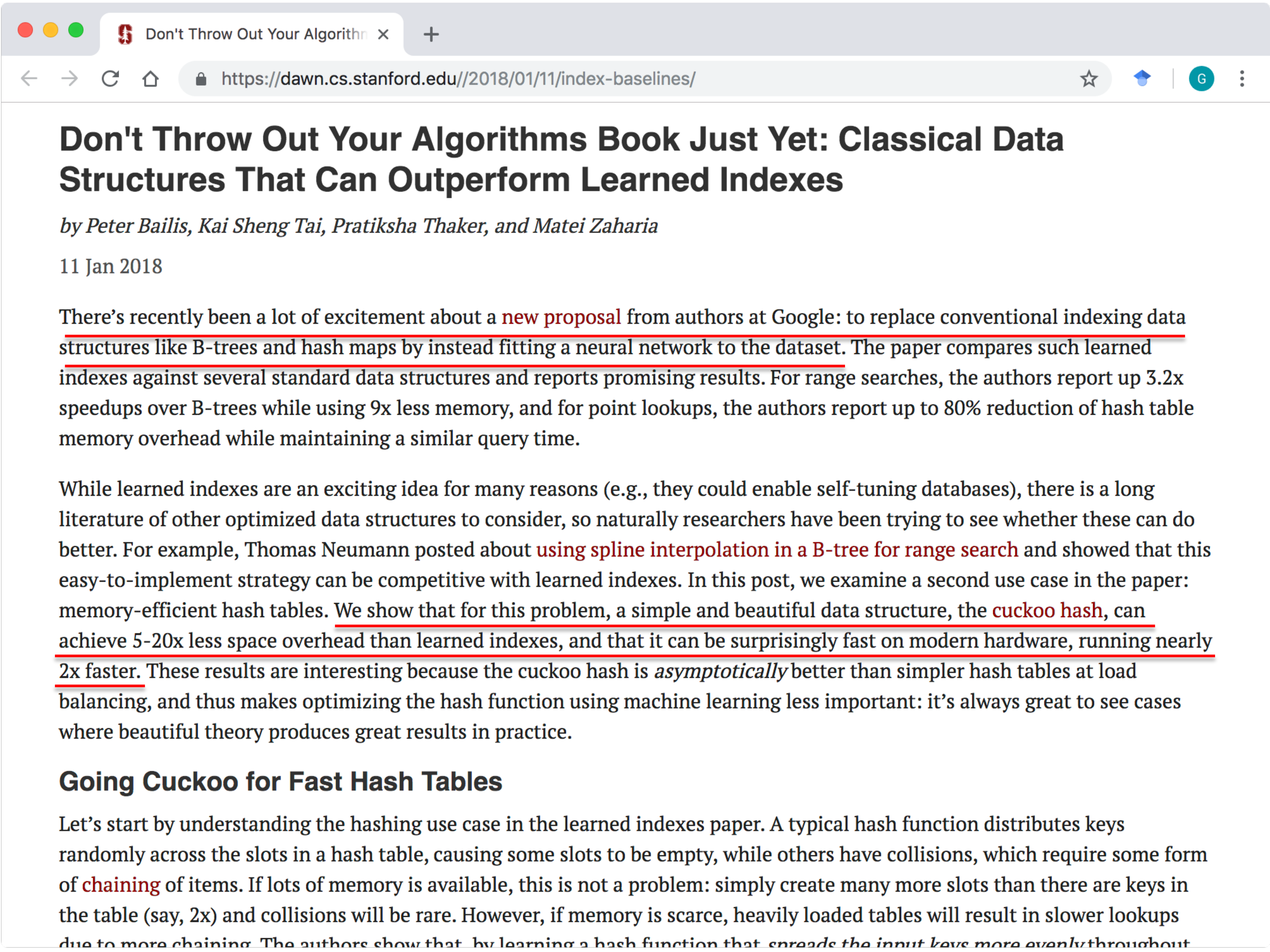
Load for $h > 2$ or $b > 1$

$b = 1$

h	2	3	4	5	6	7
load	0.5	0.918	0.976	0.992	0.997	0.999

$h = 2$

b	1	2	3	4	5	8	10
load	0.5	0.897	0.959	0.980	0.989	0.997	0.999



Don't Throw Out Your Algorithn ×

https://dawn.cs.stanford.edu//2018/01/11/index-baselines/

Don't Throw Out Your Algorithms Book Just Yet: Classical Data Structures That Can Outperform Learned Indexes

by Peter Bailis, Kai Sheng Tai, Pratiksha Thaker, and Matei Zaharia

11 Jan 2018

There's recently been a lot of excitement about a new proposal from authors at Google: to replace conventional indexing data structures like B-trees and hash maps by instead fitting a neural network to the dataset. The paper compares such learned indexes against several standard data structures and reports promising results. For range searches, the authors report up 3.2x speedups over B-trees while using 9x less memory, and for point lookups, the authors report up to 80% reduction of hash table memory overhead while maintaining a similar query time.

While learned indexes are an exciting idea for many reasons (e.g., they could enable self-tuning databases), there is a long literature of other optimized data structures to consider, so naturally researchers have been trying to see whether these can do better. For example, Thomas Neumann posted about using spline interpolation in a B-tree for range search and showed that this easy-to-implement strategy can be competitive with learned indexes. In this post, we examine a second use case in the paper: memory-efficient hash tables. We show that for this problem, a simple and beautiful data structure, the cuckoo hash, can achieve 5-20x less space overhead than learned indexes, and that it can be surprisingly fast on modern hardware, running nearly 2x faster. These results are interesting because the cuckoo hash is *asymptotically* better than simpler hash tables at load balancing, and thus makes optimizing the hash function using machine learning less important: it's always great to see cases where beautiful theory produces great results in practice.

Going Cuckoo for Fast Hash Tables

Let's start by understanding the hashing use case in the learned indexes paper. A typical hash function distributes keys randomly across the slots in a hash table, causing some slots to be empty, while others have collisions, which require some form of chaining of items. If lots of memory is available, this is not a problem: simply create many more slots than there are keys in the table (say, 2x) and collisions will be rare. However, if memory is scarce, heavily loaded tables will result in slower lookups due to more chaining. The authors show that, by learning a hash function that *spreads the input keys more evenly* throughout

Bloom filters

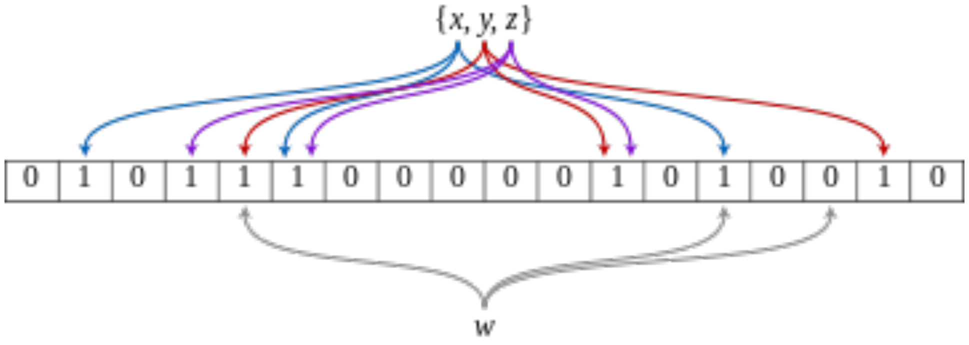
Approximate membership data structures

Bloom filters: generalities

- Bloom (1970)
- generalizes the bitmap representation of sets
- *approximate membership data structure*: supports INSERT and LOOKUP
- LOOKUP only checks for the presence, no satellite data
- produces false positives (with low probability)
- cannot iterate over the elements of the set
- DELETE is not supported (in the basic variant)
- *very* space efficient, keys themselves are ***not*** stored
- *Example*: forbidden passwords

Bloom filter: how it works

- U : universe of possible elements
- K : subset of elements, $|K| = n$
- m : size of allocated **bit array**



- define d hash functions $h_1, \ldots, h_d \colon U \rightarrow \{0, \ldots, m-1\}$

- INSERT(k): set $h_i(k) = 1$ for all i
- LOOKUP(k): check $h_i(k) = 1$ for all i

- false positives but no false negatives

Bloom filters: analysis

- P[specific bit of filter is 0] $= (1 - 1/m)^{dn} \approx e^{-dn/m} \equiv p$

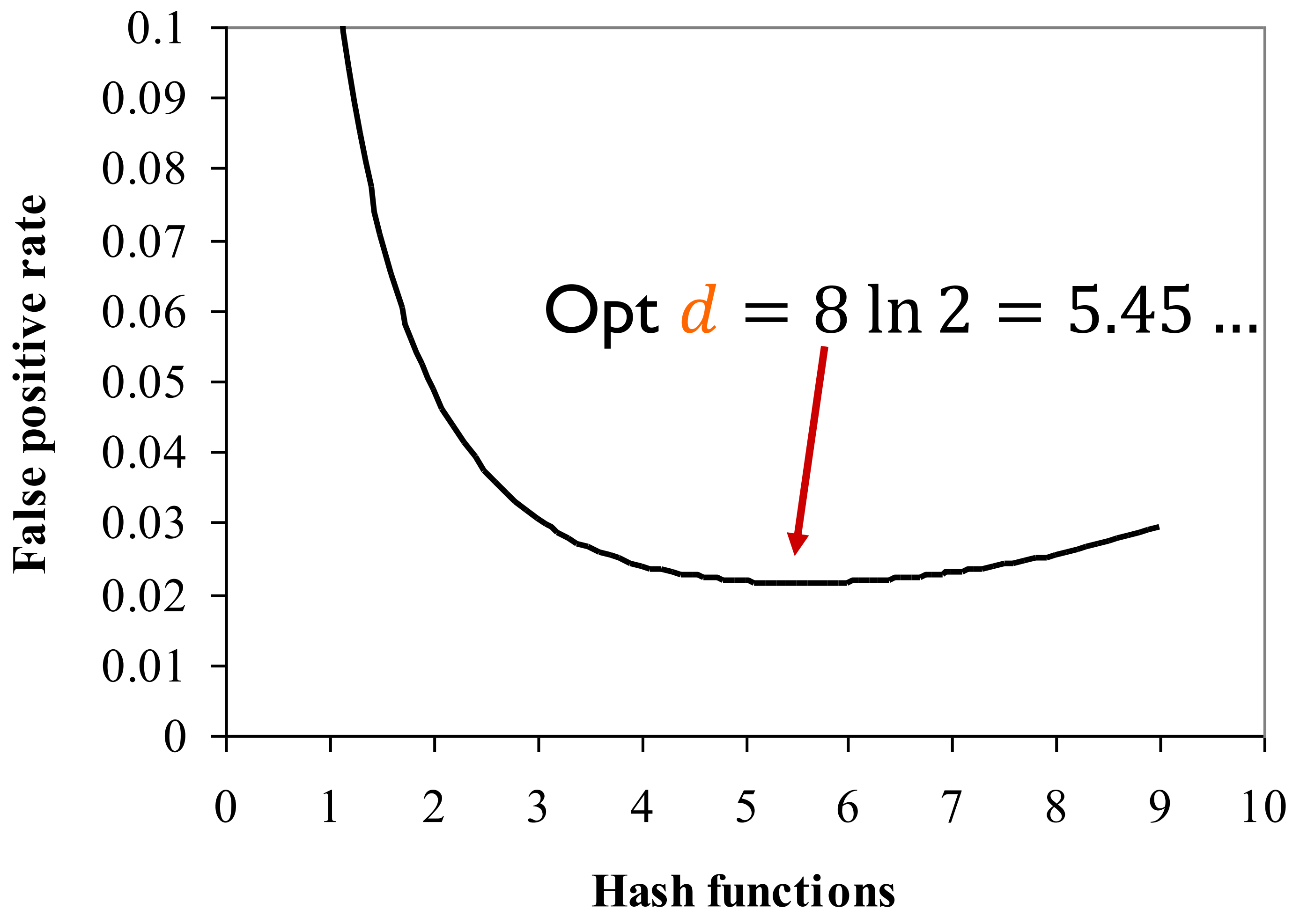
- P[false positive] $= (1-p)^{d} = (1 - e^{-dn/m})^{d}$

- Optimal number d of hash functions: $d = \ln 2 \cdot \frac{m}{n} \approx 0.693 \cdot \frac{m}{n}$
- Therefore, for the optimal number of hash functions,

$$\text{P[false positive]} = 2^{-\ln 2 \cdot \frac{m}{n}} \approx 0.6185^{\frac{m}{n}}$$

- E.g. with 10 bits per element, P[false positive] is less than 1%
- To insure the FP rate ε: $m = \log_2 e \cdot n \cdot \log_2 \frac{1}{\varepsilon} \approx 1.44 \cdot n \cdot \log_2 \frac{1}{\varepsilon}$

Dependence on the nb of hash functs



$m/n = 8$

n elements
m bits
d hash functions

Lower bound on the size of approximate membership data structures (AMD)

- Bloom filter takes $1.44 \cdot \log\frac{1}{\varepsilon}$ bits per key, is this optimal?

- How many AMDs are there to store all sets of size n drawn from universe U with FPP ε?
- Each AMD specifies a set of size $\varepsilon|U|$ (assuming $|U|$ large) containing a set of size n
- Any set of size n should be covered, and the number of such sets is $\geq \binom{|U|}{n} / \binom{\varepsilon|U|}{n} \approx \left(\frac{1}{\varepsilon}\right)^n$ (cf Erdős&Spencer 74, Rödl 85)
- $\Rightarrow$ each FPP must take $\geq n \cdot \log\frac{1}{\varepsilon}$ bits

Bloom filter: properties/operations

- For the optimal number of hash function, about a half of the bits is 1 [*immedate from the formula*]
- The Bloom filter for the union is the OR of the Bloom filters
- Is similar true for the intersection? [*explain*]
- If a Bloom filter is sparse, it is easy to halve its size

Bloom filters: applications

- Bloom filters are very easy to implement
- Used e.g. for
  - spell-checkers (in early UNIX-systems)
  - unsuitable passwords, "approximate" unsuitable passwords (Manber&Wu 1994)
  - online applications (traffic monitoring, …)
  - distributed databases
  - malicious sites in Google Chrome
  - read articles in publishing systems (Medium)
  - Google Bigtable, Apache HBase, Bitcoin, bioinformatics, …
- Sometimes (when the set of possible queries is limited) it is possible to store the set of false positives in a separate data structure