ECEN 758 Data Mining & Analysis: Hash Functions & Hash Tables Bloom Filters & Approximate Data Structures

Adapted from material by

- Michael Mitzenmacher
- Slides at http://hemo600.esy.es and elsewhere, original source not determined

TEXAS A&M UNIVERSITY
Department of Electrical & Computer Engineering

TEXAS A&M
Institute of Data Science

Hash Functions

- **A hash function is a function that:**
  - When applied to an Object, returns a number
  - When applied to *equal* Objects, returns the *same* number for each
  - When applied to *unequal* Objects, is *very unlikely* to return the same number for each
- **Hash functions turn out to be very important for searching, that is, looking things up fast**

Properties of Good Hash Functions

Hash function f: {keys}→Range = {indices}

Distributes keys (approximately) uniformly over Range

- **Efficiently computable**
- **Does not waste space**
  - Load factor lambda λ = (number of keys / RangeSize)
- **Should minimize collisions**

  = different keys hashing to same index
- **Breaks up clusters in key space**
  - Cryptographic hashes: difficult to infer keys from indices

Searching

- **Consider the problem of searching an array for a given value**
- **If the array is not sorted, the search requires O(n) time**
  - If the value isn’t there, we need to search all n elements
  - If the value is there, we search n/2 elements on average
- **If the array is sorted, we can do a binary search (see**
  - A binary search requires O(log n) time
  - About equally fast whether the element is found or not
- **It doesn’t seem like we could do much better**
  - How about an O(1), that is, constant time search?
  - We can do it *if* the array is organized in a particular way

Hashing

- **Suppose we were to come up with a "magic function" that, given a value to search for, would tell us exactly where in the array to look**
  - If it's in that location, it's in the array
  - If it's not in that location, it's not in the array
- **This function would have no other purpose**
- **If we look at the function's inputs and outputs, there wouldn't be an evident relation between them**
- **This function is called a <u>hash function</u> because it "makes hash" of its inputs**

Examples of Hash Functions

- **Simple hash functions for integers**
  - h(x) = a * x mod p for p prime
    - Note p limits size of hash ouput
    - Use p prime to break up clusters in the input space
- **CRC32**
  - Combination of shifts and XOR
  - Quick to compute; used for checksums
- **Cryptographic hashes**
  - Used for sensitive applications e.g. encryption, password verification
  - Infeasible to invert or find two inputs with same hash

Example (ideal) hash function

- **Suppose a hash function gives the following values:**

```
hashCode("apple") = 5
hashCode("watermelon") = 3
hashCode("grapes") = 8
hashCode("cantaloupe") = 7
hashCode("kiwi") = 0
hashCode("strawberry") = 9
hashCode("mango") = 6
hashCode("banana") = 2
```

0	kiwi
1	
2	banana
3	watermelon
4	
5	apple
6	mango
7	cantaloupe
8	grapes
9	strawberry

Sets and tables

- **Sometimes we just want a set of things—objects are either in it, or they are not in it**
- **Sometimes we want a map—a way of looking up one thing based on the value of another**
  - We use a *key* to find a place in the map
  - The associated *value* is the information we are trying to look up
- **Hashing works the same for both sets and maps**

...	*key*	*value*
141		
142	robin	robin info
143	sparrow	sparrow info
144	hawk	hawk info
145	seagull	seagull info
146		
147	bluejay	bluejay info
148	owl	owl info

Finding the hash function

- **How can we come up with the hash function?**
- **In general, we cannot--there is no such magic function**
- **In a few specific cases, where all the possible values are known in advance, it has been possible to compute a perfect hash function**
- **What is the next best thing?**
  - A perfect hash function would tell us exactly where to look
  - The next best we can do is a function that tells us where to *start* looking

Example imperfect hash function

- **Suppose our hash function gave us the following values:**
  - hash("apple") = 5
    hash("watermelon") = 3
    hash("grapes") = 8
    hash("cantaloupe") = 7
    hash("kiwi") = 0
    hash("strawberry") = 9
    hash("mango") = 6
    hash("banana") = 2
    hash("honeydew") = 6

0	kiwi
1	
2	banana
3	watermelon
4	
5	apple
6	mango
7	cantaloupe
8	grapes
9	strawberry

Collisions

- **When two values hash to the same array location, this is called a collision**
- **Collisions are normally treated as “first come, first served”—the first value that hashes to the location gets it**
- **We have to find something to do with the second and subsequent values that hash to this same location**

Handling collisions

- **What can we do when two different values attempt to occupy the same place in an array?**
  - **Solution #1:** Search from there for an empty location
    - Can stop searching when we find the value *or* an empty location
    - Search must be end-around
  - **Solution #2:** Use a second hash function
    - ...and a third, and a fourth, and a fifth, ...
  - **Solution #3:** Use the array location as the header of a linked list of values that hash to this location
- **All these solutions work, provided:**
  - We use the same technique to *add* things to the array as we use to *search* for things in the array

Insertion, I

- **Suppose you want to add seagull to this hash table**
- **Also suppose:**
  - hashCode(seagull) = 143
  - table[143] is not empty
  - table[143] != seagull
  - table[144] is not empty
  - table[144] != seagull
  - table[145] is empty
- **Therefore, put seagull at location 145**

. . .	
141	
142	robin
143	sparrow
144	hawk
145	seagull
146	
147	bluejay
148	owl
. . .	

Searching, I

- **Suppose you want to look up seagull in this hash table**
- **Also suppose:**
  - hashCode(seagull) = 143
  - table[143] is not empty
  - table[143] != seagull
  - table[144] is not empty
  - table[144] != seagull
  - table[145] is not empty
  - table[145] == seagull !
- **We found seagull at location 145**

. . .	
141	
142	robin
143	sparrow
144	hawk
145	seagull
146	
147	bluejay
148	owl
. . .	

Searching, II

- **Suppose you want to look up cow in this hash table**
- **Also suppose:**
  - hashCode(cow) = 144
  - table[144] is not empty
  - table[144] != cow
  - table[145] is not empty
  - table[145] != cow
  - table[146] is empty
- **If cow were in the table, we should have found it by now**
- **Therefore, it isn't here**

. . .	
141	
142	robin
143	sparrow
144	hawk
145	seagull
146	
147	bluejay
148	owl
. . .	

Insertion, II

- **Suppose you want to add hawk to this hash table**
- **Also suppose**
  - hashCode(hawk) = 143
  - table[143] is not empty
  - table[143] != hawk
  - table[144] is not empty
  - table[144] == hawk
- **hawk is already in the table, so do nothing**

. . .	
141	
142	robin
143	sparrow
144	hawk
145	seagull
146	
147	bluejay
148	owl
. . .	

Insertion, III

- **Suppose:**
  - You want to add cardinal to this hash table
  - hashCode(cardinal) = 147
  - The last location is 148
  - 147 and 148 are occupied
- **Solution:**
  - Treat the table as circular; after 148 comes 0
  - Hence, cardinal goes in location 0 (or 1, or 2, or ...)

. . .	
141	
142	robin
143	sparrow
144	hawk
145	seagull
146	
147	bluejay
148	owl

Membership Testing

- **Set of m items $S = \{x_1, x_2, \ldots, x_m\}$**
- **For a new item x we want to determine whether $x \in S$**
- **Examples:**
  - Does IP address s in a packet match a list of suspicious addresses
  - Does a proposed password match a list of well-known insecure passwords

Approximate Membership Testing

- **Exact membership queries**
  - For set of n-items requires O(n) memory
- **Cost is content dependent**
  - Testing every packet in a router is expensive in resources
    - Fast router memory to keep up with interface line rate
  - Testing packets in a cloud-based server
    - Slower, cheaper, but OK if not every packet needs testing
- **Approximate Membership Testing Data Strcture**
  - Fast (Faster than searching through S).
  - Small (Smaller than explicit representation).
- **To obtain speed and size improvements, allow some probability of error.**
  - False positives: $x \notin S$ but we report $x \in S$
  - False negatives: $x \in S$ but we report $x \notin S$

Approximate Membership Testing in Networks

- **Some network management apps test packets for set membership**
- **Network security app**
  - Set = list of IP addresses that have been deemed suspicious
- **Testing**
  - Does SrcIP or DstIP match list of suspected compromised host?
  - Does SrcIP or DstIP match list of suspected DDoS attackers
- **Actions**
  - Strong:
    - Configure filter in router to block traffic
    - Downside: may block legitimate traffic
  - Moderate:
    - Route packet to traffic scrubber for further inspection
    - Downside: more consumption of resources at scrubber, increased latency
- **Approximate membership testing at router**
  - Can save router expensive router memory resources
    - If implementable in smaller space than exact membership tests

Hash-based membership queries

- **Instead of storing words, store hash of words**
- **Compute hashes h(S) = {h(w) : w in S}**
- **Store h(S) in sorted list**
  - Space saving: assume h(w) takes up less space than w
  - Example:
    - Suppose password has minimum length 8 characters = 64 bits
    - Hash values are e.g. 32 bits
- **Testing:**
  - declare test word u unacceptable if h(u) in h(S)
- **False positives:**
  - hash values are not unique
  - collisions: h(u) in h(S) = for u not in S

Computation of false positive rate

- **Use b bit hash, $n = 2^b$ is number of distinct hash values**
- **Model hashes as uniformly randomly distributed**
  - Hash m items, each with hash distributed randomly in {1,2,….,m}
- **Pr[u has a collisions with S] = $1- (1-1/n)^m \approx 1-\exp(-m/n)$**
- **Suppose we want single item collision probability no larger than ε**
  - $1-\exp(-m/n) \approx \varepsilon \implies n \approx m / \log(1/(1- \varepsilon)$
  - $b = \mathrm{ceil}(\log_2 (m / \log(1/(1- \varepsilon)))$
- **Example: m = 100k words, ε =1% $\implies$ b = 24bits**

Bloom Filters

- **Approximate membership queries**
  - Build on idea of hashing
- **Bloom filter provides an answer combining**
  - Fixed cost to hash
  - Small amount of space.
  - But with some probability of being wrong.

Bloom Filters

Start with an n bit array, filled with 0s.

B

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Hash each item x_j in S k times. If $H_i(x_j) = a$, set $B[a] = 1$.

B

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

To check if y is in S, check B at $H_i(y)$. All k values must be 1.

B

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

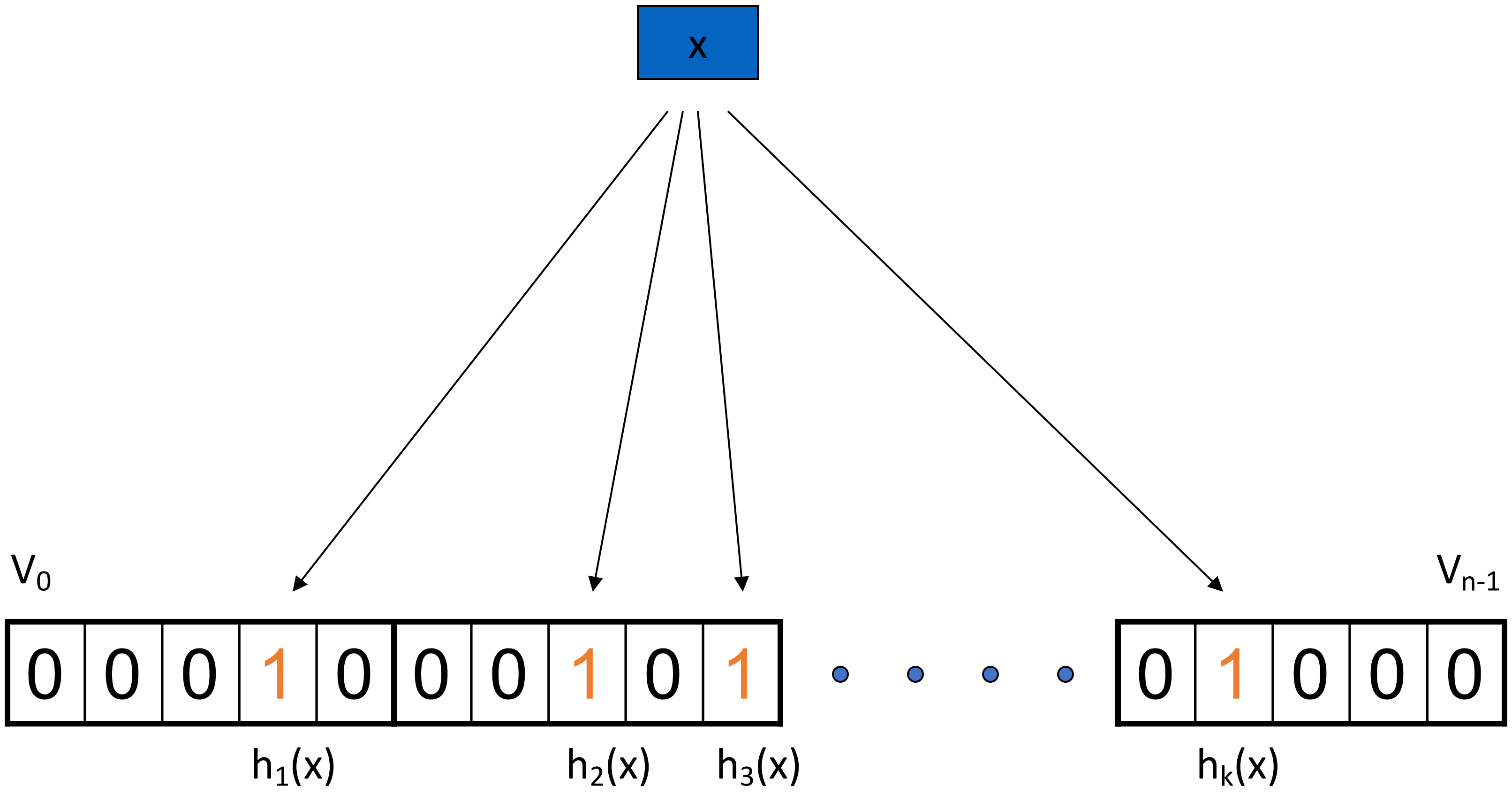
Possible to have a false positive; all k values are 1, but y is not in S.

B

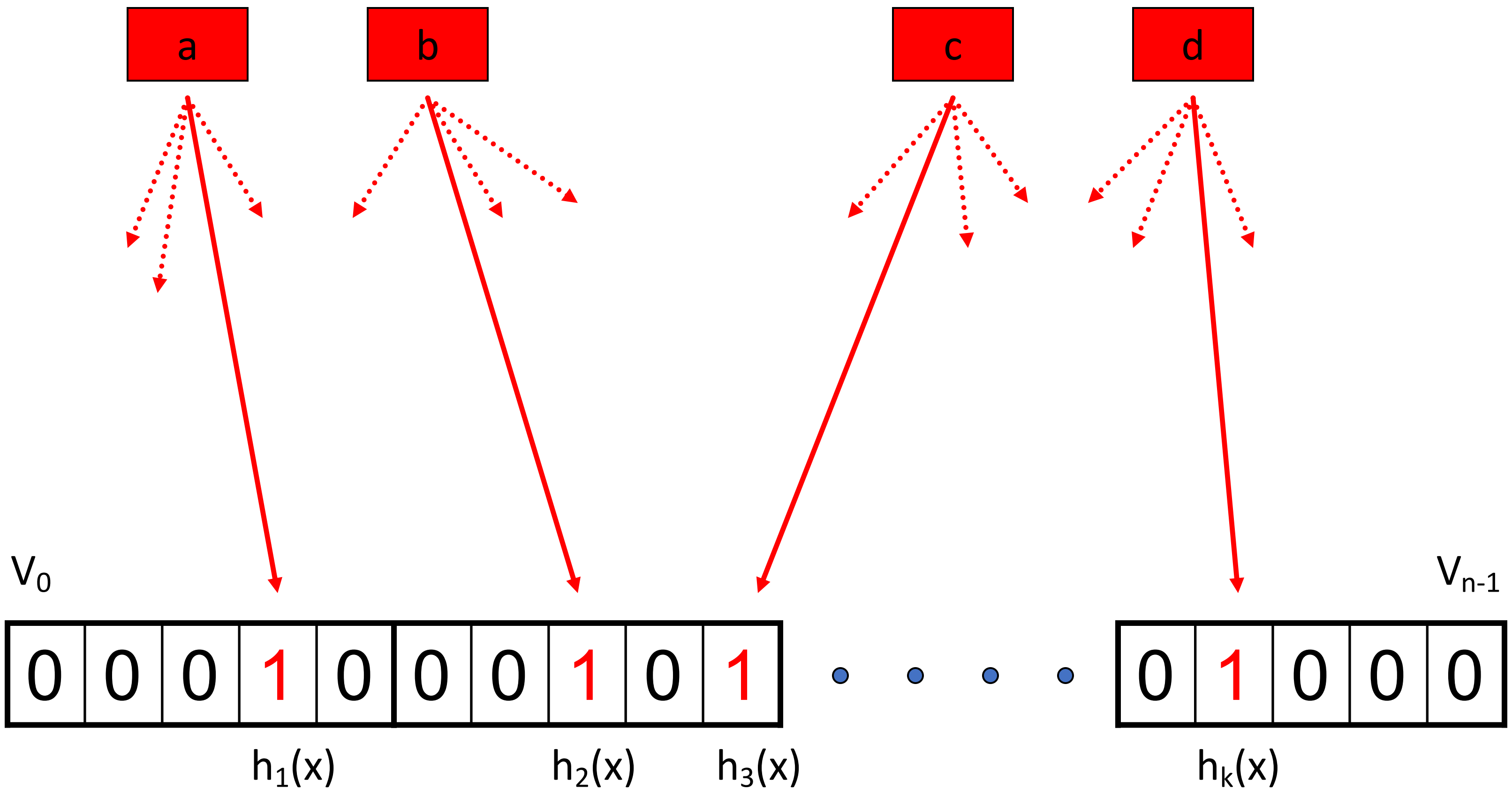
0	1	0	0	1	0	1	0	0	1	1	1	0	1	1	0

m items $n = cm$ bits k hash functions

Bloom Filter Insertion



Bloom Filter False Positives



x didn't appear, yet its bits are already set

Bloom Filter Properties

- **No overflow**
  - Degradation: false positive rate grows with #items stored
- **Union and intersection of Bloom Filters**
  - bitwise OR and AND respectively
- **Compression**
  - size $n = 2^b$: b bit output of hash function
  - bitwise AND first and second halves together
    - AND positions k and $k+2^{b-1}$ for $k < 2^{b-1}$
  - Mask highest order bit in hash output
    - Identifies positions k and $k+2^{b-1}$ for $k < 2^{b-1}$

Computational Factors

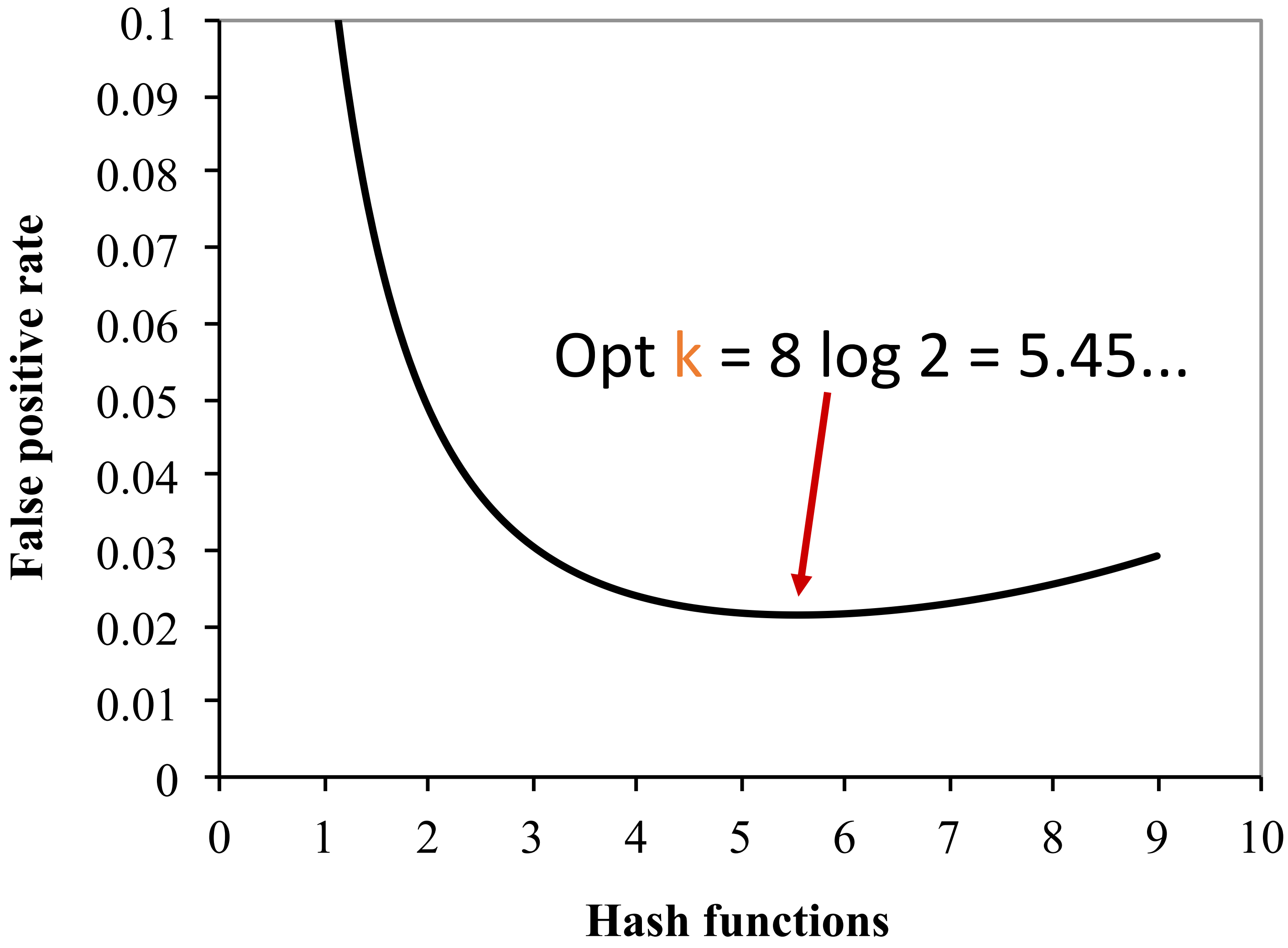
- **Size *n/m* : bits per item.**
  - $|S| = m$: Number of elements to encode.
  - h_i: $S \rightarrow [1..n]$: Maintain a Bit Vector V of size n
- **Cost *k* : number of hash functions.**
  - Use k hash functions ($h_1..h_k$)
- **Error *f* : false positive probability.**

False Positive Probability Analysis

- **Pr[any bit=0] = $(1-1/n)^{km} \approx e^{-km/n} = e^{-k/c}$ (no hash hits)**
- **False Positives**
  - Happen if all k hash function find bit=1
  - Probability $\approx f(k) = (1-e^{-k/c})^k$ (all bits set = 1)
- **Trade-off when increasing k**
  - More likely to get collision in specific bit
  - More bits have to be set for collision to occur
- **Analysis**
  - Derivative $(d/dk) \log f(k) = \log(1-e^{-k/c}) + (k/c)\, e^{-k/c} / (1 - e^{-k/c})$
- **Optimal #hash functions**
  - $f'(k) = 0$ when $k = k^* = c \log(2) = (n/m) \log(2)$
  - Optimal choice of k depends only on c = number of bits per item
- **Optimal False Positive Probability**
  - $f(k^*) = ((1/2)^{\log(2)})^{n/m} \sim 0.6185^{n/m}$
  - FPP depends only on bits per item c = n/m, regardless of #items m

m items
n = cm bits
k hash functions

Example



m items
n = cm bits
k hash functions

n/m = 8

n/m = 8

m items *n = cm* bits *k* hash functions

Optimal Randomization

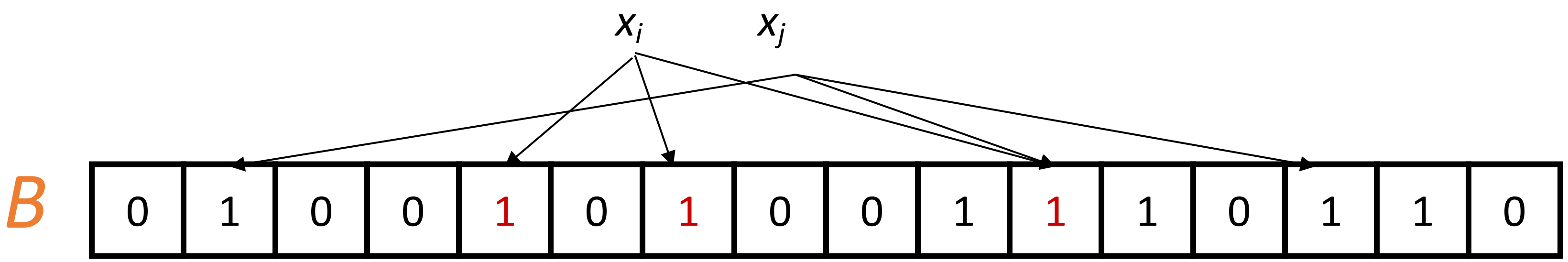
- **At optimum Pr[bit=0] $\approx e^{-k^*/c} = 1/2$**
  - Each bit of the Bloom filter is 0 with probability 1/2.
  - An optimized Bloom filter looks like a random bit string.

Classic uses of Bloom Filter: Spell-Checking

- **Once upon a time, memory was scarce...**
- **/usr/dict/words -- about 210KB, 25K words**
- **Use 25 KB Bloom filter**
  - 8 bits per words
  - Optimal 5 hash functions.
- **Probability of false positive about 2%**
- **False positive = accept a misspelled word**
- **BFs still used to deal with list of words**
  - Password security
    - [Spafford 1992 http://dl.acm.org/citation.cfm?id=134593]
  - Keyword driven ads in web search engines, etc.

Bloom Filters and Deletions

- **Memory Contents change**
  - Items both inserted and deleted.
- **Insertions are easy – add bits to BF**
- **Bloom filters can handle insertions, but not deletions**
  - No consistent way of deleting
  - If deleting x = setting $h_l(x) = 0$ for all hashes l = 1,…,k
  - Then deleting x_i causes deletion of x_j (some bits no longer set)



Counting Bloom Filters

Start with an m bit array, filled with 0s.

B	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Hash each item x_j in S k times. If $H_i(x_j) = a$, add 1 to $B[a]$.

B	0	3	0	0	1	0	2	0	0	3	2	1	0	2	1	0

To delete x_j decrement the corresponding counters.

B	0	2	0	0	0	0	2	0	0	3	2	1	0	1	1	0

Can obtain a corresponding Bloom filter by reducing to 0/1.

B	0	1	0	0	0	0	1	0	0	1	1	1	0	1	1	0

TEXAS A&M UNIVERSITY
Department of Electrical
& Computer Engineering

TEXAS A&M
Institute of
Data Science

Counting Bloom Filters In Practice

- **If insertions/deletions are rare compared to lookups**
  - Keep a CBF in slow "off-chip memory"
  - Keep a BF in fast "on-chip memory"
  - Update the BF when the CBF changes
- **Keep space savings of a Bloom filter**
- **But can deal with deletions**
- **Popular design for network devices**
  - E.g. pattern matching application

Reference & Acknowledgement

- **Bloom, B. Space/time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM, 13* (7). 422-426, 1970**
- **These notes adapt material from**
  - Michael Mitzenmacher
  - Slides at http://hemo600.esy.es
    - and elsewhere, original source not determined