CS6931 Database Seminar

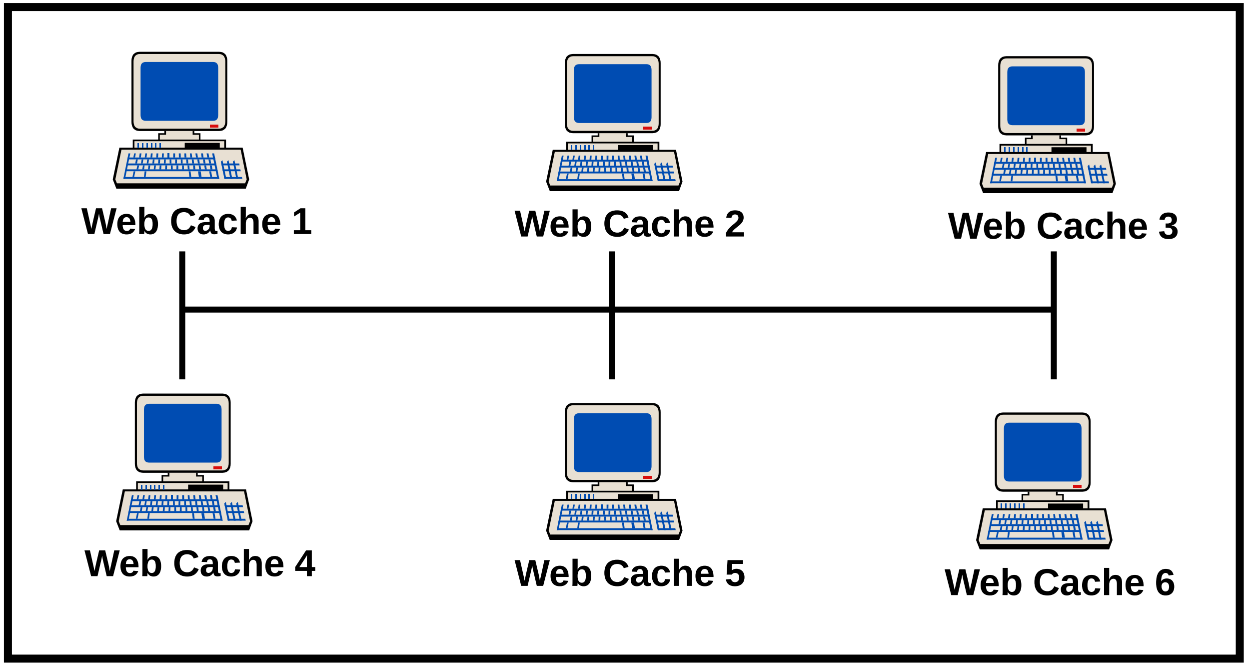
Lecture 7: Set Operations on Massive Data (Continued)

Membership Testing (Bloom Filter)

Bloom Filter

- Problem: membership testing
  - Does item x from an universe [U] belong to a set S ?
- Assumption: the great majority of items tested will not belong to the given set
- Data structure should be:
  - Fast (faster than searching through S).
  - Small (smaller than explicit representation).
- The “price”: allow some probability of error
  - Allow false positive errors
  - Don’t allow false negative errors

Sample Application: Distributed Web Caches



- Summary Cache: [Fan, Cao, Almeida, & Broder]

  If local caches know each other's content...

  …try local cache before going out to Web
- The idea: each cache keeps a summary of the content of each participating cache
- Store each summary in a Bloom Filter

Why Bloom Filters?

- Size is very economical
- Efficient query time
- Percentage of false positives is 1%-2% for 8 bits per entry
- False positives are possible
  - Penalty is a wasted cache query. Small cost.
- No false negatives
  - Never miss a cache hit. Big potential gain.

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 $H_i(x_j) = a$, set $B[a] = 1$.

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

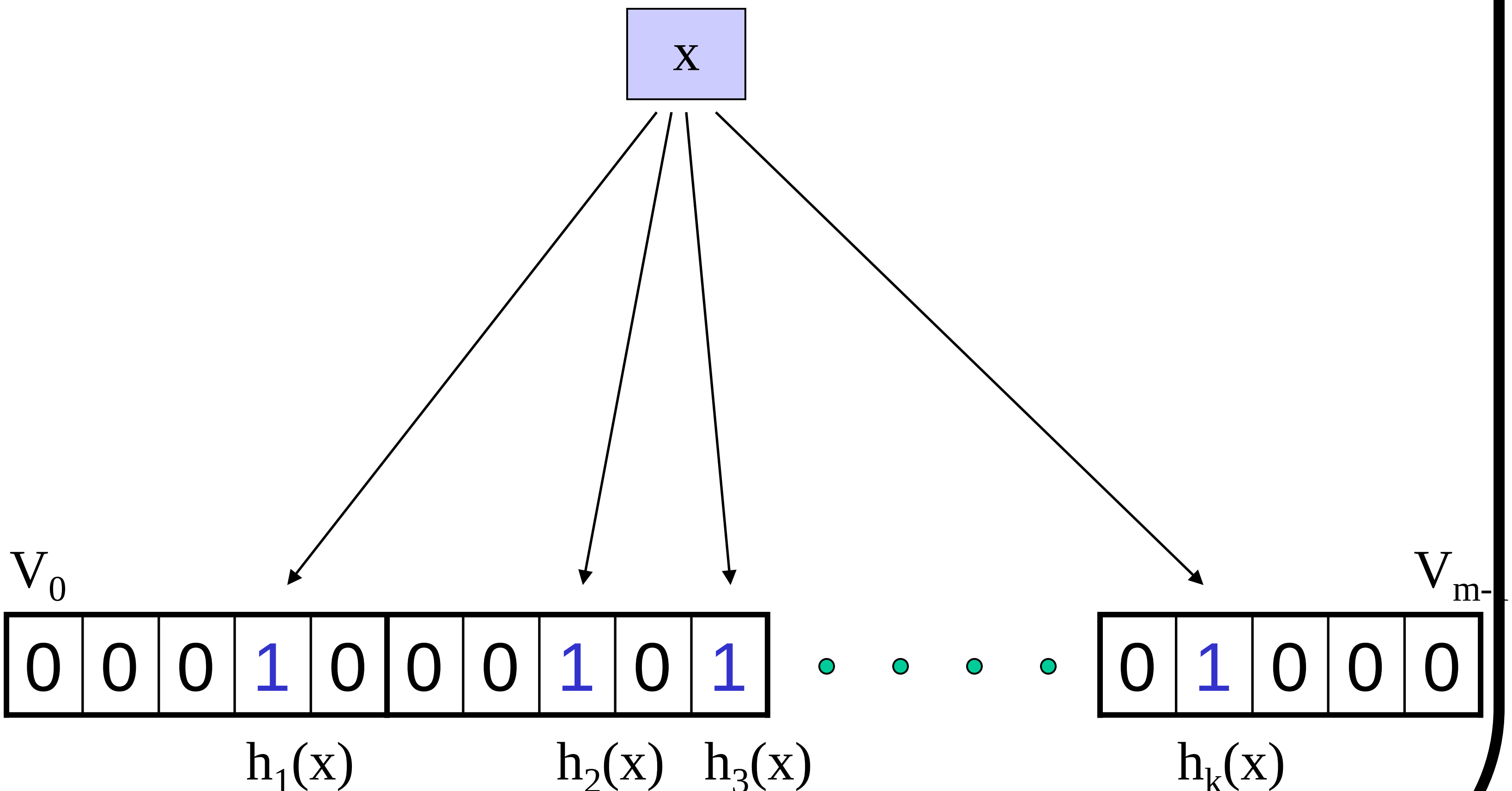
To check if y is in S, check B at $H_i(y)$. All k values must be 1.

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

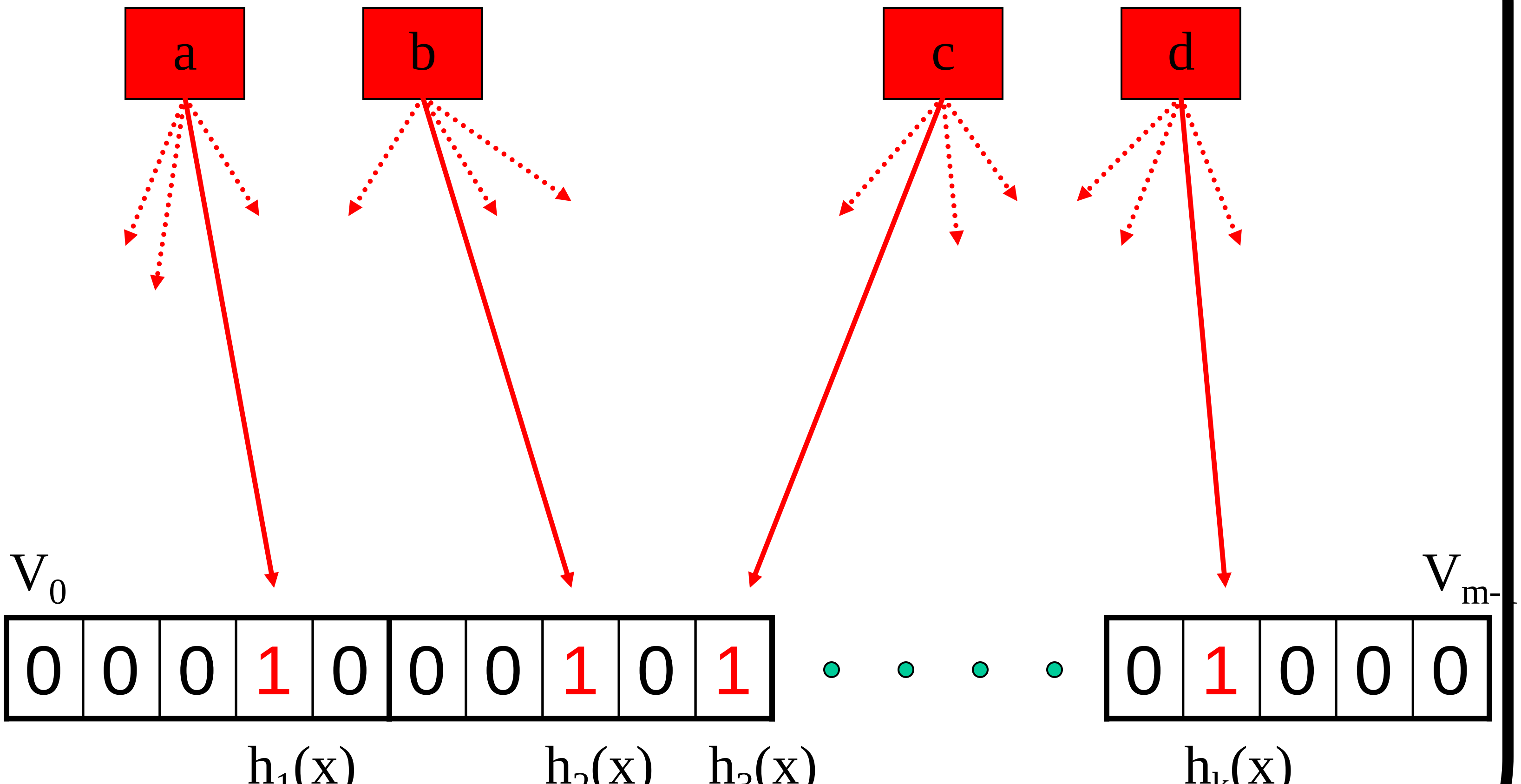
Possible to have a false positive; all k values are 1, but y is not in S.

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

Bloom Filter



Bloom Errors



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

Computational Factors

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

Error Estimation

- Assumption: Hash functions are perfectly random
- Probability of a bit being 0 after hashing all elements:

$$(1-1/m)^{kn} \approx e^{-kn/m} = e^{-\gamma}, \gamma = \frac{nk}{m}$$

- Let $p=e^{-kn/m}$, probability of a false positive is:

$$f = \left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^{k} \approx \left(1-e^{-kn/m}\right)^{k} = (1-p)^{k}$$

- Assuming we are given m and n, the optimal k is:

$$f = \exp\left(k\ln\left(1-e^{-kn/m}\right)\right)$$

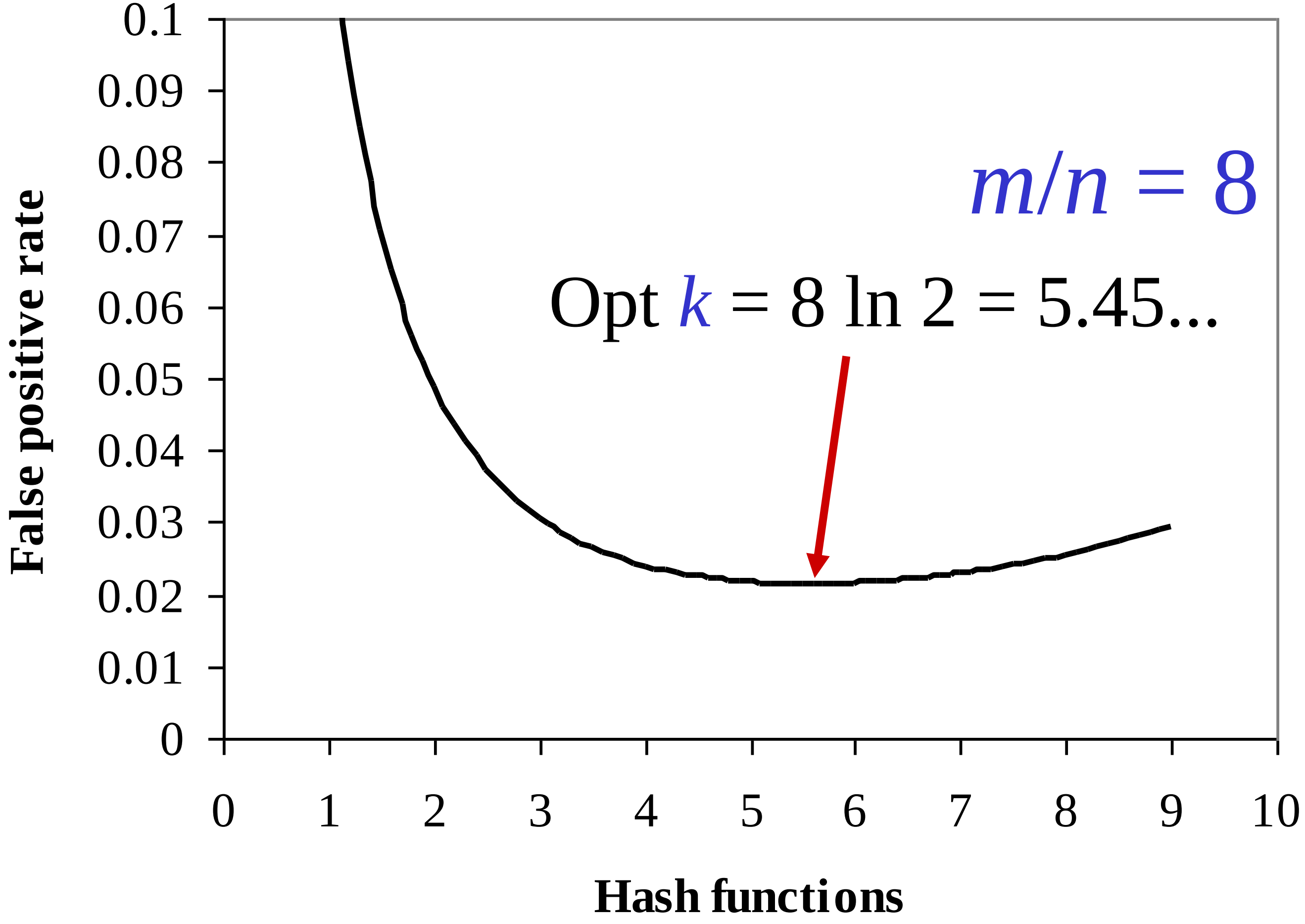
$$g = k\ln\left(1-e^{-kn/m}\right)$$

$$\frac{dg}{dk} = \ln\left(1-e^{-kn/m}\right) + \frac{kn}{m}\frac{e^{-kn/m}}{1-e^{-kn/m}}$$

$$\frac{dg}{dk} = 0 \Rightarrow k_{\min} = (\ln 2)\left(\frac{m}{n}\right)$$

$$f(k_{\min}) = (1/2)^{k} = (0.6185)^{m/n}$$

Example



False positive rate
0.1
0.09
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
0 1 2 3 4 5 6 7 8 9 10
Hash functions
m/n = 8
Opt k = 8 ln 2 = 5.45...

Bloom Filter Tradeoffs

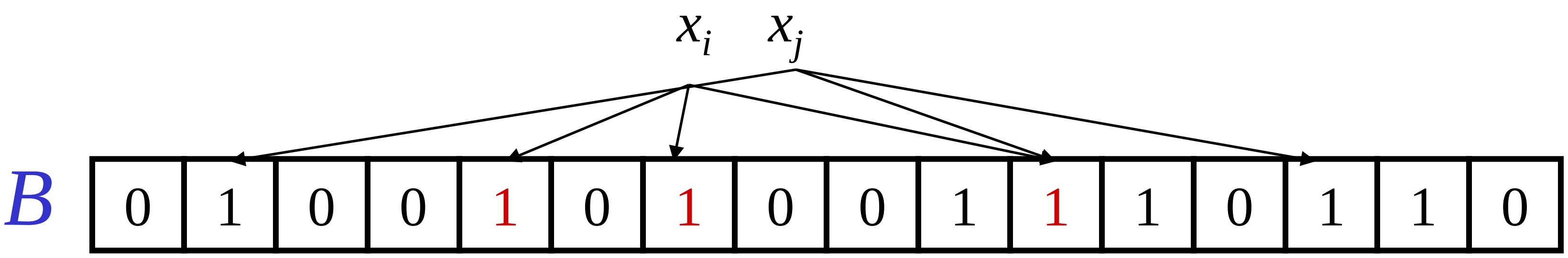
- Three factors: m,k and n.
- Normally, n and m are given, and we select k.
  - More hash functions yields more chances to find a 0 bit for elements not in S
  - Fewer hash functions increases the fraction of the bits that are 0.
- Not surprisingly, when k is optimal, the “hit ratio” (ratio of bits flipped in the array) is 0.5 .

Bloom Filters and Deletions

- Cache contents change
  - Items both inserted and deleted.
- Insertions are easy – add bits to BF
- Can Bloom filters handle deletions?
  - Use Counting Bloom Filters to track insertions/deletions

Handling Deletions

- Bloom filters can handle insertions, but not deletions.
- If deleting x_i means resetting 1s to 0s, then deleting x_i will "delete" x_j.



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