Bloom Filters and Such

Michael Mitzenmacher
Harvard University

Good Hash Functions

- There's a whole theory on good hash functions that are close to random in suitable ways.
- We will not explore that here. Just take random hash function as an assumption unless otherwise noted.

Things Tie Together

- Ideas early on will keep re-appearing in somewhat different and sometimes more complex ways.
- Hashing themes repeat.

Bloom Filters: Approximate Membership Queries

- Given a set $S = \{x_1, x_2, x_3, \ldots x_n\}$ on a universe U, want to answer *membership queries* of the form:

$$\text{Is } y \in S.$$

- Data structure should be:
  - Fast (Faster than searching through S).
  - Small (Smaller than explicit representation).
- To obtain speed and size improvements, allow some probability of error.
  - False positives: $y \notin S$ but we report $y \in S$
  - False negatives: $y \in S$ but we report $y \notin S$

Bloom Filters

- Given a set $S = \{x_1, x_2, x_3, \ldots x_n\}$ on a universe U, want to answer queries of the form:

$$\text{Is } y \in S.$$

- Bloom filter provides an answer in
  - "Constant" time (time to hash).
  - Small amount of space.
  - But with some probability of being wrong.

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

n items $m = cn$ bits k hash functions

False Positive Probability

- Pr(specific bit of filter is 0) is

$$p' \equiv (1-1/m)^{kn} \approx \mathrm{e}^{-kn/m} \equiv p$$

- If ρ is fraction of 0 bits in the filter then false positive probability is

$$(1-\rho)^k \approx (1-p')^k \approx (1-p)^k = (1-\mathrm{e}^{-k/c})^k$$

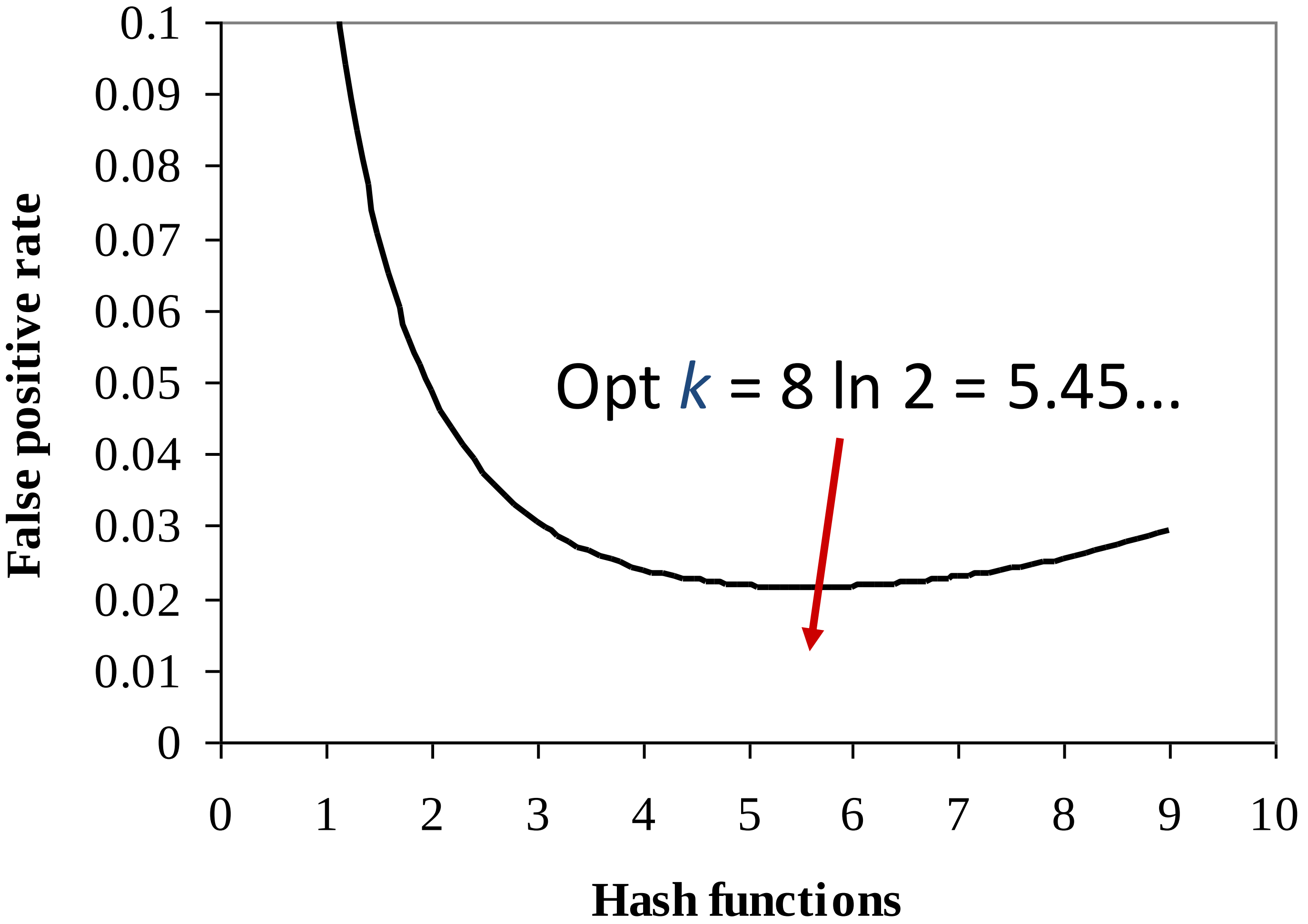
- Approximations valid as ρ is concentrated around E[ρ].
  - Martingale argument suffices.
- Find optimal at $k = (\ln 2)m/n$ by calculus.
  - So optimal fpp is about $(0.6185)^{m/n}$

n items $m = cn$ bits k hash functions

More Sophisticated Analyses

- We have *nk* hashes into *m* buckets. Can calculate the distribution of the number of empty/non-empty buckets exactly.
  - Stirling numbers of the 2nd kind.
- Can get the "exact" false positive probability.
  - Really, a distribution over possible false positive probabilities.
- Overkill except for very small filters.
  - Where concentration isn't tight enough.

Example



$m/n = 8$

n items $m = cn$ bits k hash functions

A Useful Framework

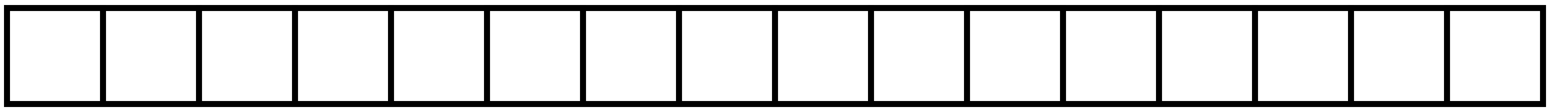
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Each Binomial(nk,1/m)

Not independent

n items $m = cn$ bits k hash functions

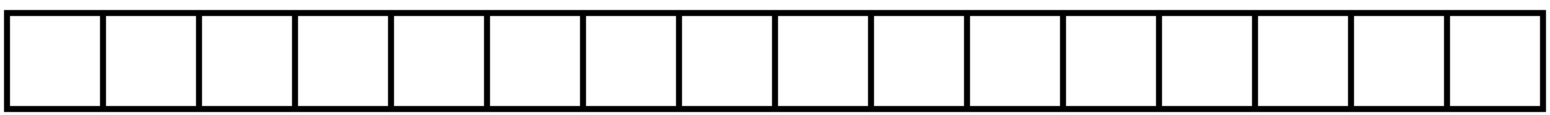
A Useful Framework



Each Binomial(nk,1/m)

Not independent

$n \rightarrow \infty$



Each Poisson(nk/m)

Independent

n items $m = cn$ bits k hash functions

Binomial to Poisson

- Strong theorems regarding the binomial to Poisson connection for hashing.
  - Conditioned on Poissons having right number of items, distribution is binomial.
  - Chernoff bounds from Poisson setting can be applied to binomial setting.
- When helpful, think of independent Poisson and work out details later.

A Useful Framework

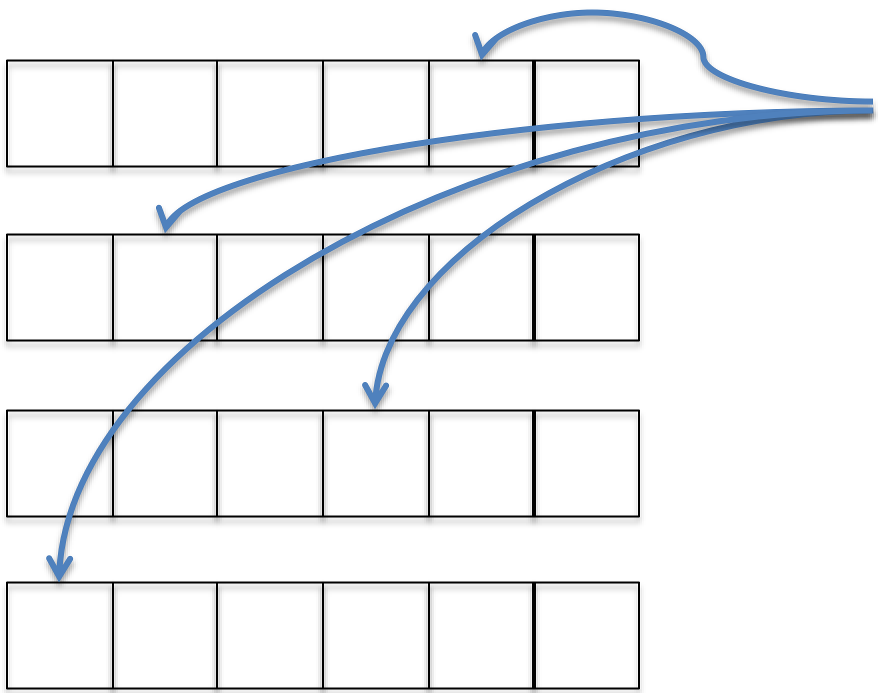
- Pr(specific bit of filter is 0) is

$$Pr(\mathrm{Poisson}(kn/m) = 0) = e^{-kn/m}$$

- Approximate independence gives false positive probability is approximately

$$(1 - e^{-kn/m})^k$$

Split Bloom Filters



Key hashed to *k* cells

m bits split into *m*/*k* disjoint groups

one has per group

"same" performance, easier to parallelize

False Positive Probability
Split Bloom Filter

- Pr(specific bit of filter is 0) is

$$p' \equiv (1-k/m)^n \approx \mathrm{e}^{-kn/m} \equiv p$$

- Note the k subgroups are truly independent, and always have k distinct choices. So false positive probability is truly

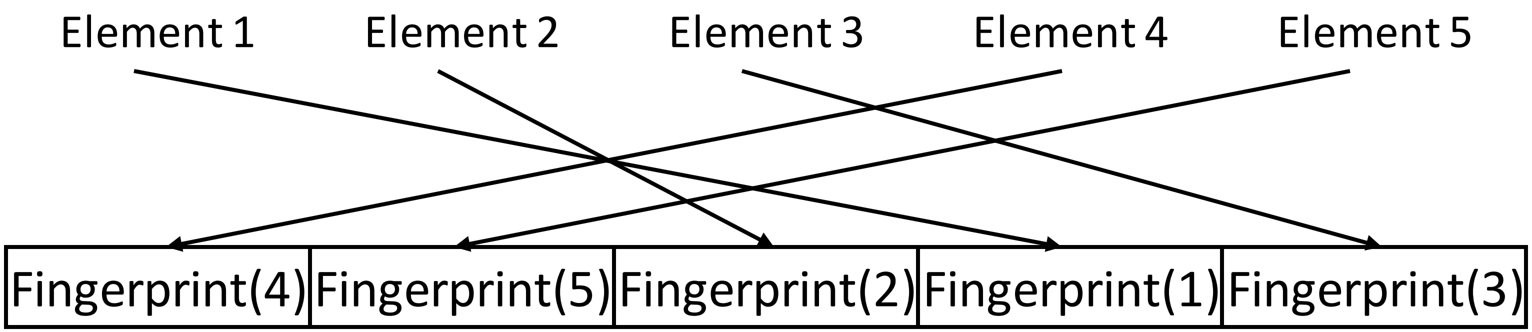
$$(1-p')^k$$

- Asymptotically the same, though the "bound" is slightly "worse", as

$$(1-k/m)^n \leq (1-1/m)^{kn}$$

n items $m = cn$ bits k hash functions

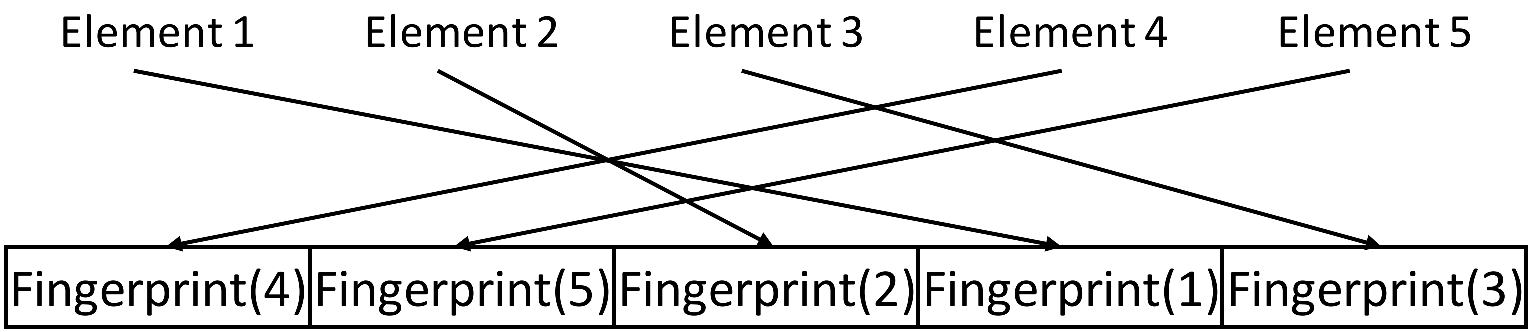
Perfect Hashing Approach



Alternative Construction

- Bloom filters are NOT optimal.
  - In terms of space vs. error tradeoff.
- Given a set of n elements, compute a *perfect hash function* mapping them to an array of n cells.
  - Perfect hash function = 1 cell per element.
- Store a $\log 1/\varepsilon$ -bit fingerprint of the element at each cell. (Determined by random hash function.)
- To test y for set membership, hash to find its cell, then hash to check its fingerprint.
  - False positive probability of $(0.5)^{m/n} = \varepsilon$, if $m = n \log 1/\varepsilon$ bits used
- Constant factor less space (about 40% less).
- Less flexible solution: can't add new elements.

Perfect Hashing Approach



So Why Use Bloom Filters?

- In the real world, there is a 4-dimensional tradeoff space.

So Why Use Bloom Filters?

- In the real world, there is a 4-dimensional tradeoff space.
  - Time.
  - Space.
  - Correctness (error probability).

So Why Use Bloom Filters?

- In the real world, there is a 4-dimensional tradeoff space.
  - Time.
  - Space.
  - Correctness (error probability).
  - Programmer Time.

Classic uses of BF: Spell-Checking

- *Once upon a time, memory was scarce...*
- **`/usr/dict/words`** -- about 210KB, 25K words
- Use 25 KB Bloom filter
  - 8 bits per word.
  - 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], [Manber & Wu, 94]
  - Keyword driven ads in web search engines, etc.

Classic uses of BF: Data Bases

- **Join:** Combine two tables with a common domain into a single table
- **Semi-join:** A join in distributed DBs in which only the joining attribute from one site is transmitted to the other site and used for selection. The selected records are sent back.
- **Bloom-join:** A semi-join where we send only a BF of the joining attribute.

Modern Use of BF: Large-Scale Signature Detection

- Monitor all traffic going through a router, checking for signatures of bad behavior.
  - Strings associated with worms, viruses, etc.
- Must be fast – operate at line speed.
  - Run easily on hardware.
- Solution : Separate signatures by length, build a Bloom filter for each length, in parallel check all strings of each length each time a new character comes through.
- Signature found : send off to analyzer for action.
  - False positive = extra work along the slow path.
- [Dharmapurikar, Krishnamurthy, Sproull, Lockwood]

Hardware Framework

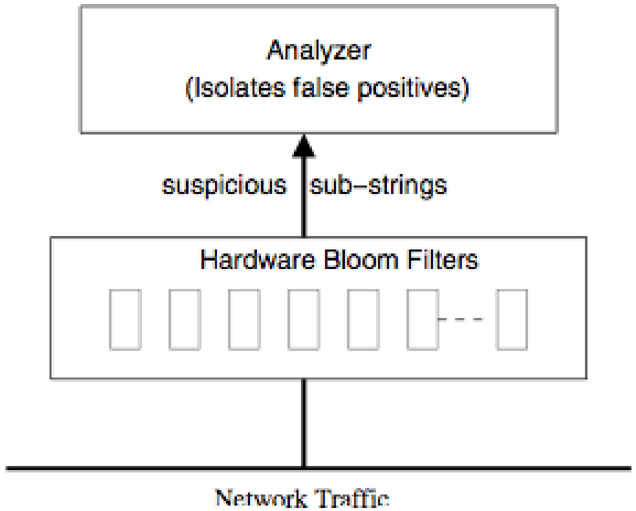


Figure 1. Bloom filters scanning all traffic on multi-gigabit network for predefined signatures

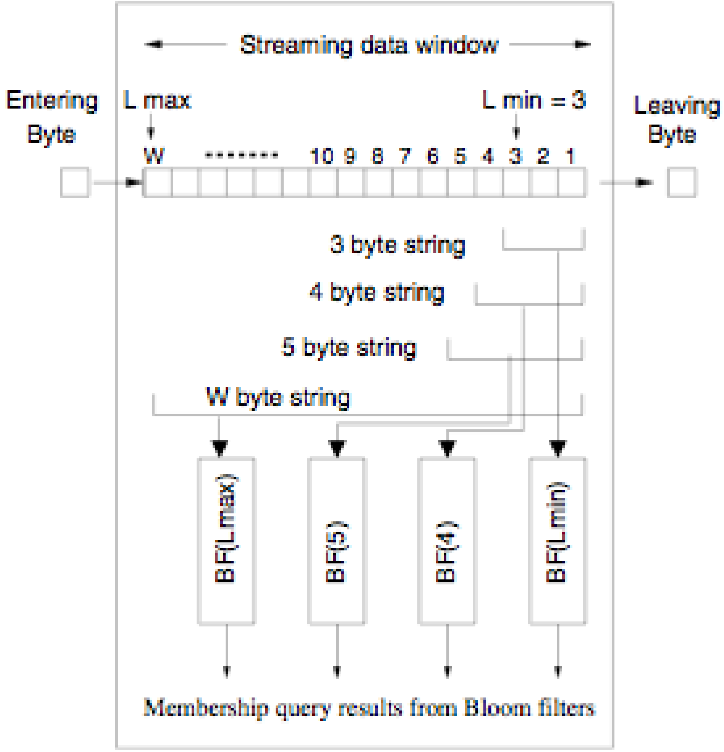


Figure 2. A window of streaming data containing strings of length from $L_{min} = 3$ to $L_{max} = W$. Each string is examined by a Bloom filter

Modern Uses

- All over networking : see my surveys
  - Broder/Mitzenmacher : Network Applications of Bloom Filters
  - Kirsch/Mitzenmacher/Varghese : Hash-Based Techniques for High-Speed Packet Processing
- But more and more every day.

Why Bloom Filters Are Not Taught in Algorithms 101?

- With optimal k the upper bound on expected number of false positives for $z = \text{poly}(n)$ elements is:

$$\#errors = z(0.61)^{m/n}$$

- For theoretical analyses we usually want

$$\#errors = O(1) \text{ or even } \#errors = o(1)$$

- This requires $m/n = \Omega(\log n)$.
- Not interesting: can be done by hashing.
- Bloom filters allow constant bits/element and constant false positive probability.
  - Good enough for many applications.

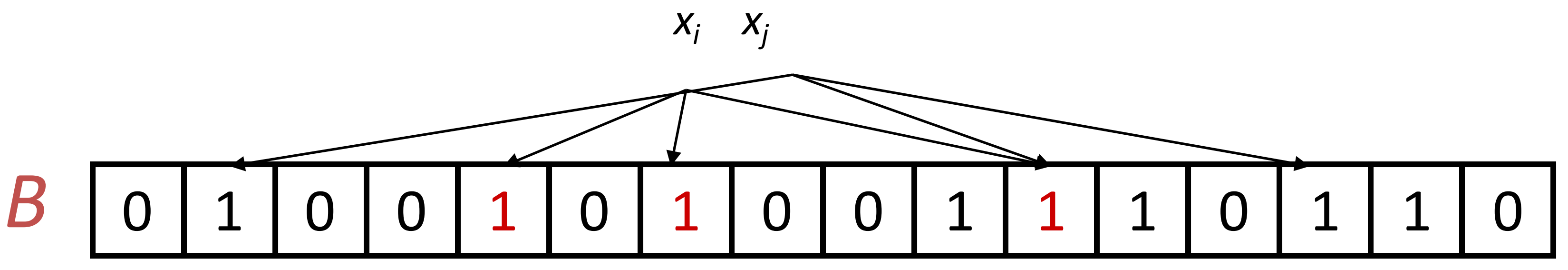
The main point

- Whenever you have a set or list, and space is an issue, a Bloom filter may be a useful alternative.

- Just be sure to consider the effects of the false positives!

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

n items $m = cn$ bits k hash functions

Counting Bloom Filters: Overflow

- Must choose counters large enough to avoid overflow.
- Poisson approximation suggests 4 bits/counter.
  - Average load using $k = (\ln 2)m/n$ counters is $\ln 2$.
  - Probability a counter has load 16 (Poisson approx):

$$\approx e^{-\ln 2}(\ln 2)^{16}/16! \approx 6.78\mathrm{E}-17$$

- Failsafes possible.
- Generally 4 bits/counter.
  - Can do better with slower, multilevel scheme.

Counting Bloom Filters In Practice

- If insertions/deletions are rare compared to lookups
  - Keep a CBF in "off-chip memory"
  - Keep a BF in "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 described.

Variation: Double Hashing

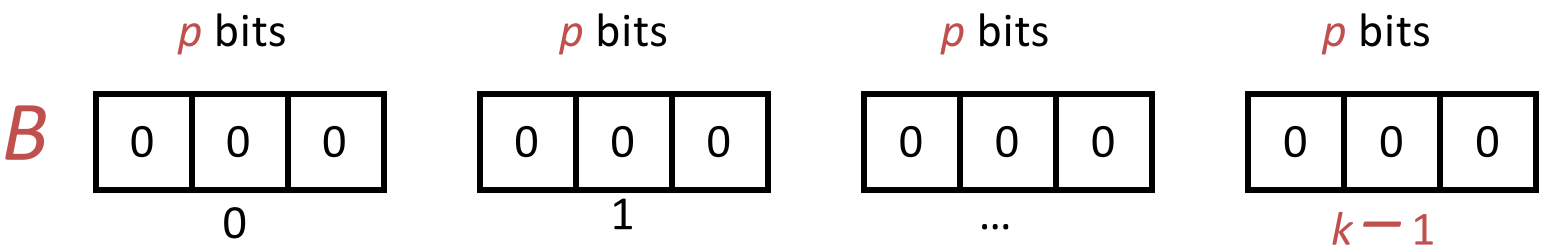
- [DillingerManolios],[KirschMitzenmacher]
- Let h_1 and h_2 be hash functions.
- For $i = 0, 1, 2, \ldots, k-1$ and some f, let

$$g_i(x) = h_1(x) + ih_2(x) \bmod m$$

  - So 2 hash functions can mimic k hash functions.
- Dillinger/Manolios show experimentally, and we prove, no difference in asymptotic false positive probability.

A Simpler Framework

- Consider the “split” Bloom filter.
- Suppose $m = kp = cn$ for p prime.



$$g_i(x) = h_1(x) + ih_2(x) \bmod p$$

- Here h_1, h_2 map universe to numbers mod p.
- Set $g_i(x)$th bit of ith subarray.

n items $m = kp = cn$ bits k hash functions

Collisions

- A collision for x,y is an i : $g_i(x) = g_i(y)$.
- In the simple example, number of collisions for any pair x,y is only 0, 1, or k, since:

$$g_i(x) = h_1(x) + ih_2(x) = h_1(y) + ih_2(y) = g_i(y) \bmod p$$
$$g_j(x) = h_1(x) + jh_2(x) = h_1(y) + jh_2(y) = g_j(y) \bmod p$$

implies, for distinct i, j

$$h_1(x) = h_1(y), h_2(x) = h_2(y), \text{so } g_l(x) = g_l(y) \forall l$$

Ignoring k-Collisions

- Consider some $z \notin S$. False positive occurs if for every i, $g_i(x) = g_i(z)$ for some $x \in S$.
- Bad case [k-collison]: for some $x \in S$,

$$h_1(x) = h_1(z), h_2(x) = h_2(z)$$

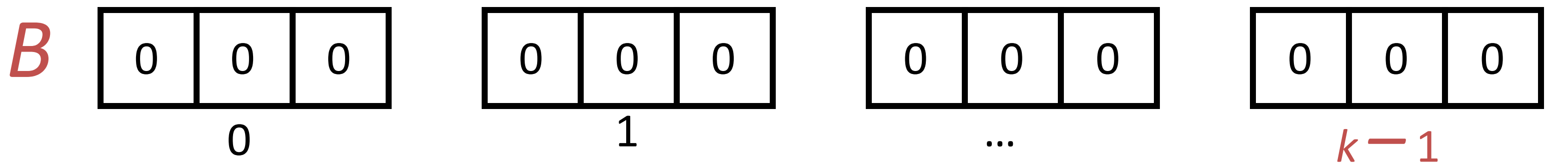
- k-collision occurs with probability at most $n/p^2 = o(1)$.
- Now ignore (condition on) no k-collision.

n items $m = kp = cn$ bits k hash functions

Back to Poisson Framework

Each Binomial(n,k/m)

Not independent



Each Poisson(kn/m)

Independent Enough

$$\Pr(\text{false positive}) \rightarrow (1 - e^{-kn/m})^k$$

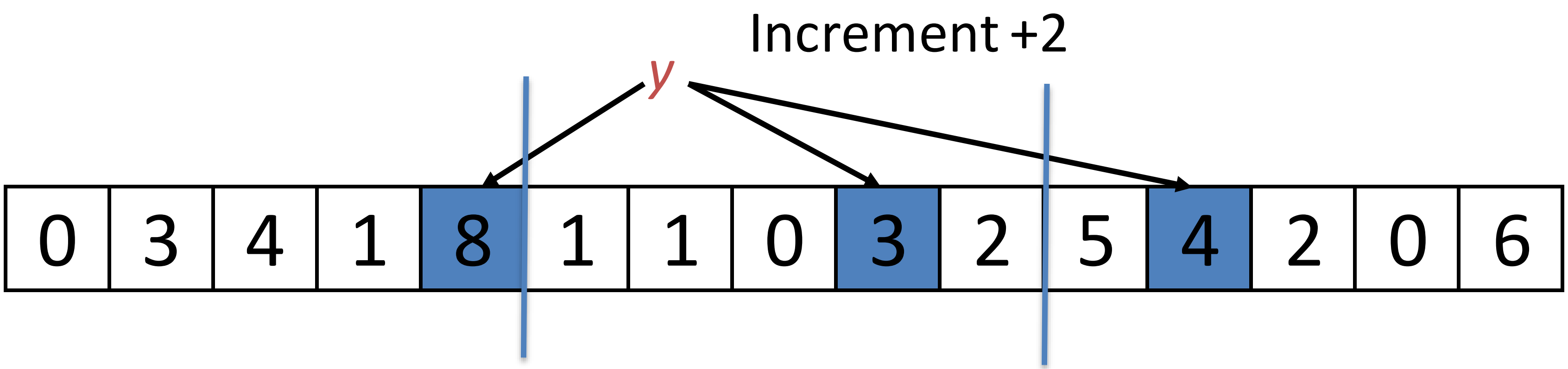
Double Hashing

- Many available Bloom filter applications now use double hashing technique.
  - BFs have false positives anyway.
  - Negligible change in false positives, simpler/faster.

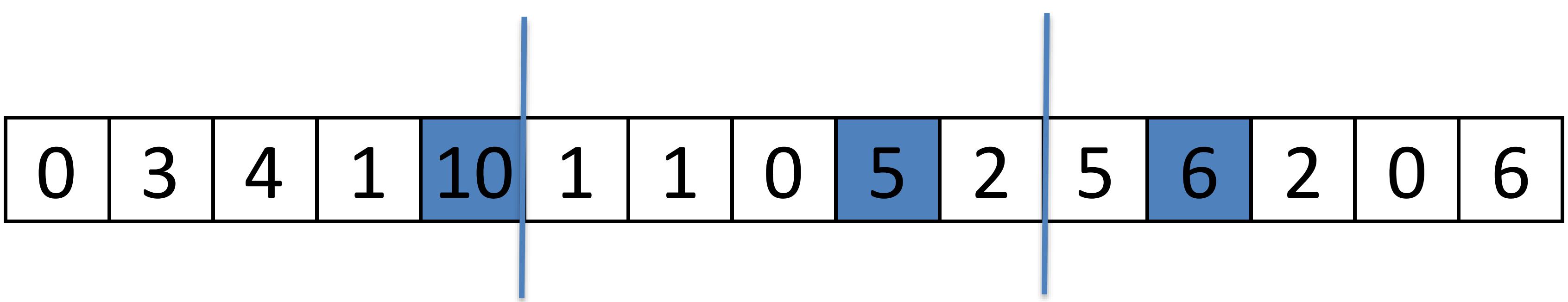
Count Min Sketch

- Variation of counting Bloom filters.
- Item have associated counts.
- Hash items to k locations.
- Increment count at those locations.
- Estimate of item = minimum of counters.

Count Min Sketch



Easy again to think of the hash table broken into *k* subtables, with one hash in each subtable. Makes analysis slightly easer.



Analysis : Count-Min Sketch

- Expected amount per cell = (Total**k*/*m*)
  - For *m* cells, *k* subtables
- As long as hash functions are pairwise independent, for any item *x*, and any bin *B(x)* that *x* hashes to

$$\Pr(\text{Extra Count at } B(x) \ \geq cTk/m) \leq 1/c$$

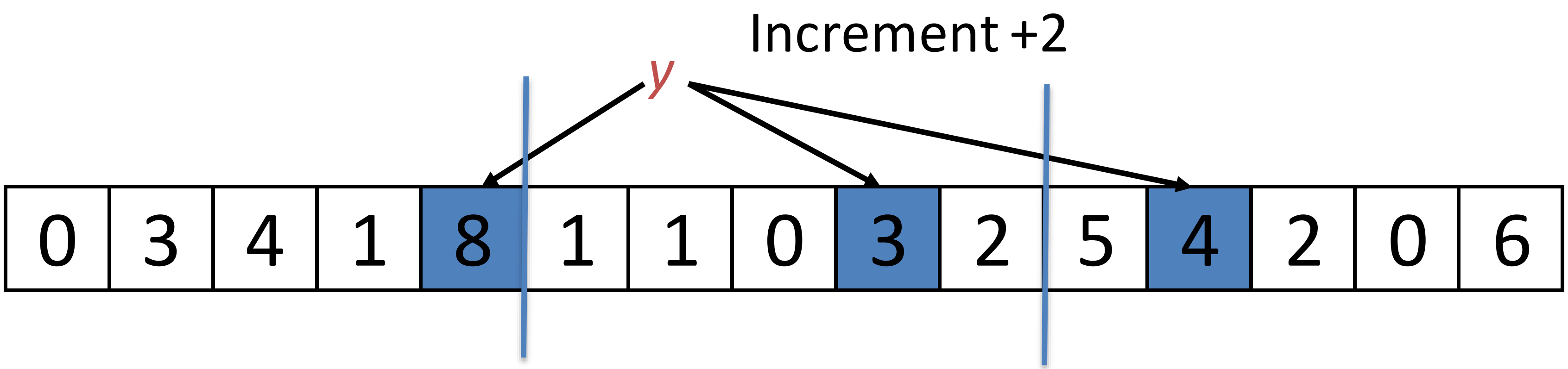
- Hence for *Est*(*x*) = minimum count of the bins *x* hashes to,

$$\Pr(Est(x) \ \geq Count(x) + cTk/m) \leq (1/c)^k$$

Count-Min Notes

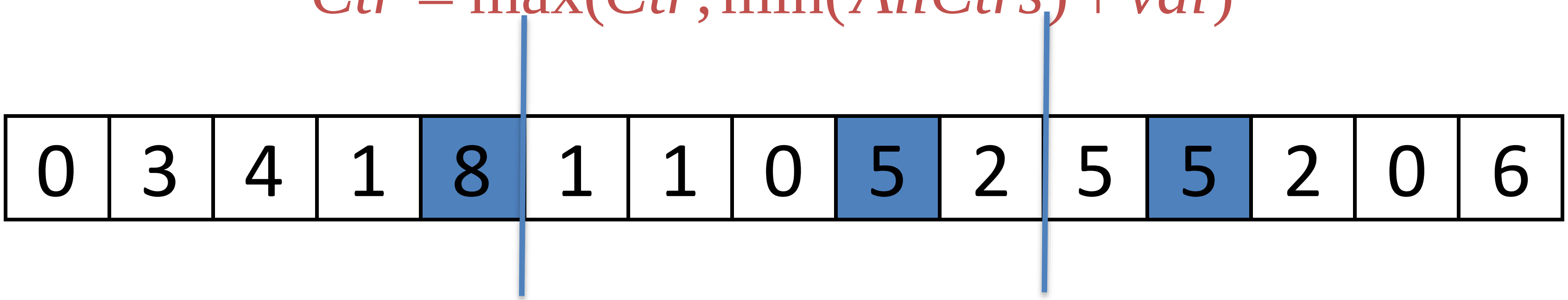
- Lots of uses:
  - Finding heavy hitters
  - Other approximate count situations
    - Approximate dot-products, etc.
- Better analysis available for skewed data streams (with few large values)
  - Common setting in practice
- Improvement possible in “no deletion” setting

Conservative Update



The flow associated with *y* can only have been responsible for 3 packets; counters should be updated to 5.

$$Ctr = \max(Ctr, \min(AllCtrs) + val)$$



Stragglers' Problem

- Consider data streams that insert/delete a lot of pairs.
  - Flows through a router, people entering/leaving a building.
- We want listing not at all times, but at "reasonable" or "off-peak" times, when the current working set size is bounded.
  - If we do all the N insertions, then all the N-M deletions, and want a list at the end, we want…
- Data structure size should be proportional to **listing size**, not maximum size.
  - Proportional to M, not to N!
  - Proportional to size you want to be able to list, not number of pairs your system has to handle.

Set Reconciliation Problem

- Alice and Bob each hold a set of keys, with a large overlap.
  - Example: Alice is your smartphone phone book, Bob is your desktop phone book, and new entries or changes need to be synched.
- Want one/both parties to learn the set difference.
- Goal: communication is proportional to the size of the difference.

Simple Bloom Filter Solution

- Alice and Bob create Bloom filters of their sets.
- Trade Bloom filters.
- Alice can check for elements not in Bob’s filter, and vice versa, and send those.
- False positives?

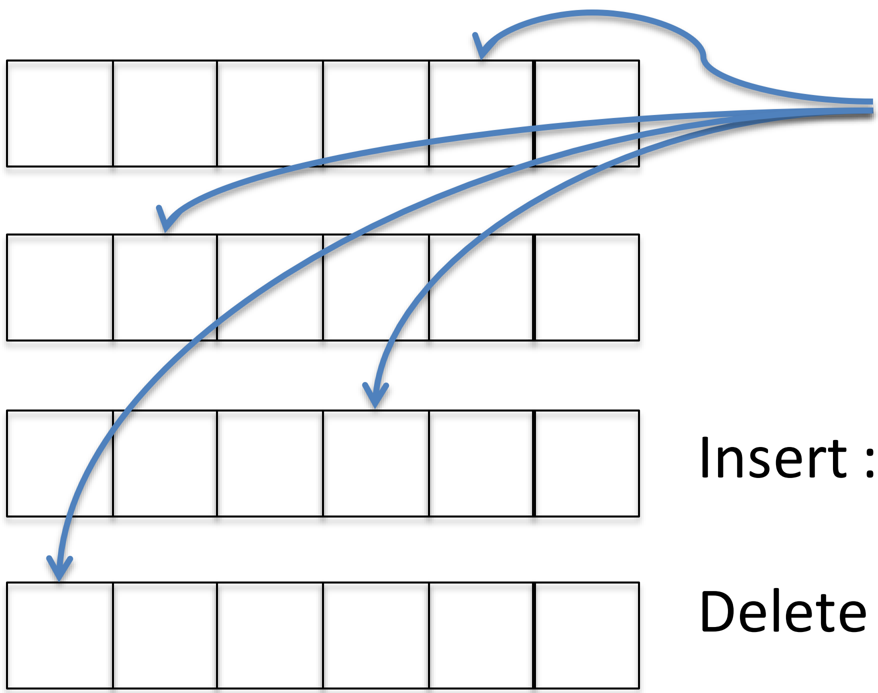
Simple Bloom Filter Solution

- Alice and Bob create Bloom filters of their sets.
- Trade Bloom filters.
- Alice can check for elements not in Bob's filter, and vice versa, and send those.
- False positives will prevent some elements in the difference from being sent.
  - But a small number; can repeat this over multiple filters in various ways.
- Transmission is proportional to set size, not set difference!

Invertible Bloom Lookup Tables Functionality

- IBLT operations
  - Insert (k)
  - Delete (k)
  - ListEntries()
    - Lists all current keys
    - Succeeds as long as **current** load is not too high
      - Design threshold

Invertible Bloom Lookup Tables



Key *k* pair hashed to *j* cells

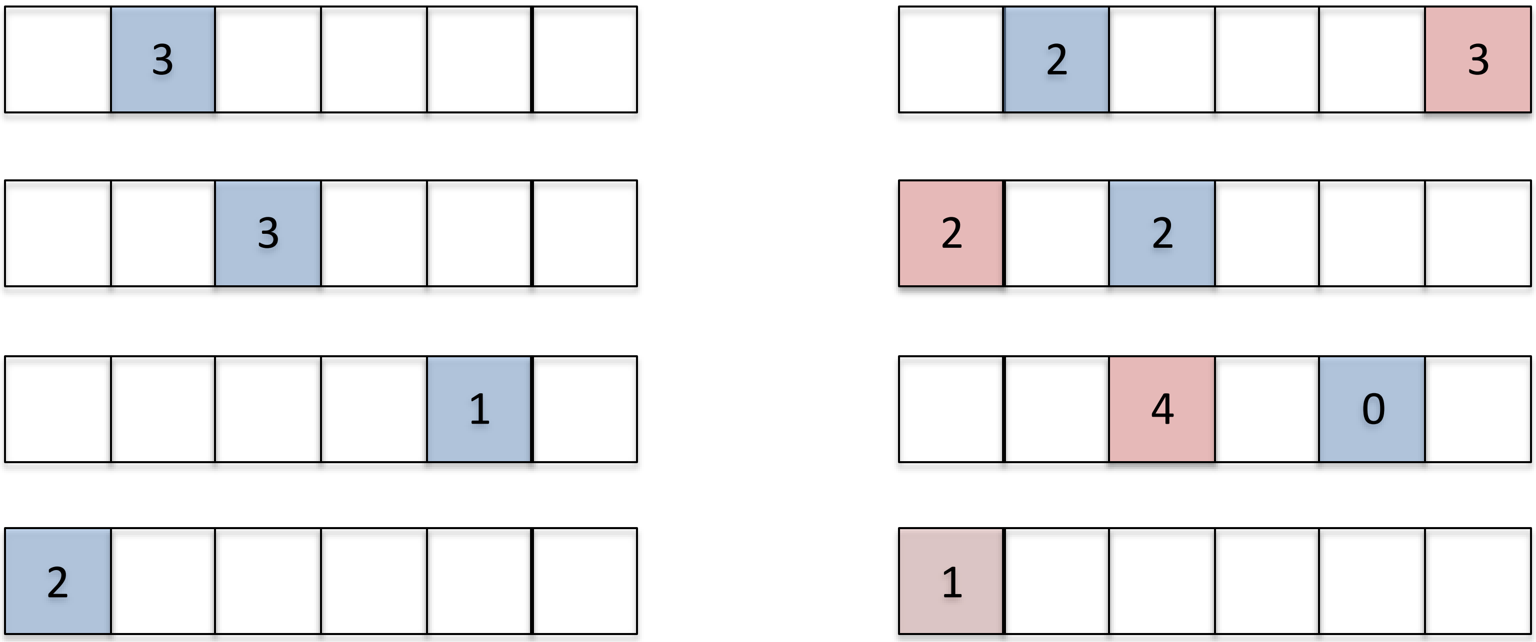
Count
KeySum
CheckSum

Insert : Update KeySum and CheckSum
XOR in new values

Delete : Update KeySum and CheckSum
XOR out values

Peel away keys by finding cell
where KeySum and CheckSum match
and Count =1; e.g. just one key in the cell.

Listing Example



Solving Set Reconciliation

- Alice and Bob create IBLTs.
  - Using predetermined size, hash function.
- Trade IBLTs.
- Alice “deletes” her items from Bob’s IBLT.
- Remaining IBLT holds the set difference (with “positive” count for Bob’s items, negative for Alice’s items).
- Peel to recover items.
  - Can also peel when count is -1 in this setting.
- Theorem: space required is O(|set difference|).

Set Similarity

- Bloom filter can be used to estimate similarity of two sets.
- Take the dot product of their Bloom filters to estimate the intersection (or union).
- What is the probability a bit is set?

Set Similarity

- Bloom filter can be used to estimate similarity of two sets.
- Take the dot product of their Bloom filters to estimate the intersection (or union).
- What is the probability a bit is set?

$$1-(1-1/m)^{k|S_1|}-(1-1/m)^{k|S_2|}+(1-1/m)^{k|S_1 \cup S_2|}$$

Odd Sketches

- Better similarity sketch when similarity is high.
- Use just 1 hash function.
- Cells don't keep count, but keep *parity* of number of elements hashed there.
- Hence

$$\mathrm{OddSketch}(S_1) \oplus \mathrm{OddSketch}(S_2) = \mathrm{OddSketch}(S_1 \oplus S_2)$$

Odd Sketches

- How many 1s in

$$\mathrm{OddSketch}(S_1) \oplus \mathrm{OddSketch}(S_2) = \mathrm{OddSketch}(S_1 \oplus S_2)$$

- Use Poisson approximation.
- Elements in both sets cancel each other out.
- So $n = |S_1 \oplus S_2|$ elements hashed.
- Probability a bit is 1 = probability a Poisson random variable with mean *n*/*m* is odd.

$$(1 - e^{-2n/m})/2$$

- Estimate *n* from the number of 1 bits.

Odd Sketches

- Odd Sketches tested on natural “high similarity” problems.
  - Web duplicate detection.
  - Association rule learning.

Dynamic Bloom Filters

- Bloom filters allow addition of items.
- Suppose we want to allow addition of items, from an empty filter up to n items, so that at every intermediate point we use at most m bits, and have false positive at most ε.
- Requires at least $C(\varepsilon) n \log_2(1/\varepsilon)$ bits for $C(\varepsilon) > 1$.
- Even stronger lower bounds when number of items not known in advance.

Sliding Bloom Filters

- Keep a Bloom filter over last n items of a stream.
- Useful generalization: must answer yes on last n items, and on previous m items any answer is OK; for items of age $n+m$ false positive probability should be ε.
- Recent results: constant ε can be done in space $n \log(n/m) + O(n)$.
- Practical?

Conclusion

- Bloom filter idea: use multiple hashing
  - To get exponential decrease in false positives
  - Simple approach to build a data structure
- The idea is so basic, it can be applied in lots of ways.
  - Countless variations.
  - Just the tip of the iceberg.
  - Very powerful paradigm.
- Usually theoretically interesting to find “better” ways.
  - Though not necessarily useful in practice.

Next Lecture

- Other uses of “multiple choices” in hashing
  - Balanced allocations
  - Cuckoo hashing

Exercise

- Derive that for two sets S_1 and S_2, if we take the dot product of their Bloom filters, the fraction of bits set to 1 is (approximately)

$$1-(1-1/m)^{k|S_1|}-(1-1/m)^{k|S_2|}+(1-1/m)^{k|S_1\cup S_2|}$$

Exercise

- Derive that for a Poisson random variable with mean *x*, the probability it takes on an odd value is

$$(1-e^{-2x})/2$$

Exercise

- Consider a universe with universe size $u >> n$. Show that any structure that uses m bits to represent sets of n elements from the universe with false positive probability at most ε requires at least (approximately) $n \log_2(1/\varepsilon)$ bits.

Exercise

- Consider a counting Bloom filter with a secondary structure. The counter only uses 3 bits per cell. If the count is greater than 6, we use a value of 7 to represent that the count for that cell must be kept in a secondary structure. Suggest a suitable secondary structure, and estimate the reduction in size from this approach.

Open Exercise

- The false positive rate for a Bloom filter is different from the false positive probability. Given a set S for a Bloom filter, and a multiset T of queries disjoint from S, the false positive rate is the fraction of false positives on T. It can be highly variable, if the multiset yields a false positive on frequent items.
- Can we have a Bloom filter "adapt" to lower the false positive rate by avoiding false positives on frequent items of T?