Bloom Filters, Cuckoo Hashing, Cuckoo Filters, Adaptive Cuckoo Filters and Learned Bloom Filters

Michael Mitzenmacher
With many various others
over the years

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: 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

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 $\quad$ $m = cn$ bits $\quad$ k hash functions

False Positive Probability

- Pr(specific bit of filter is 0) is

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

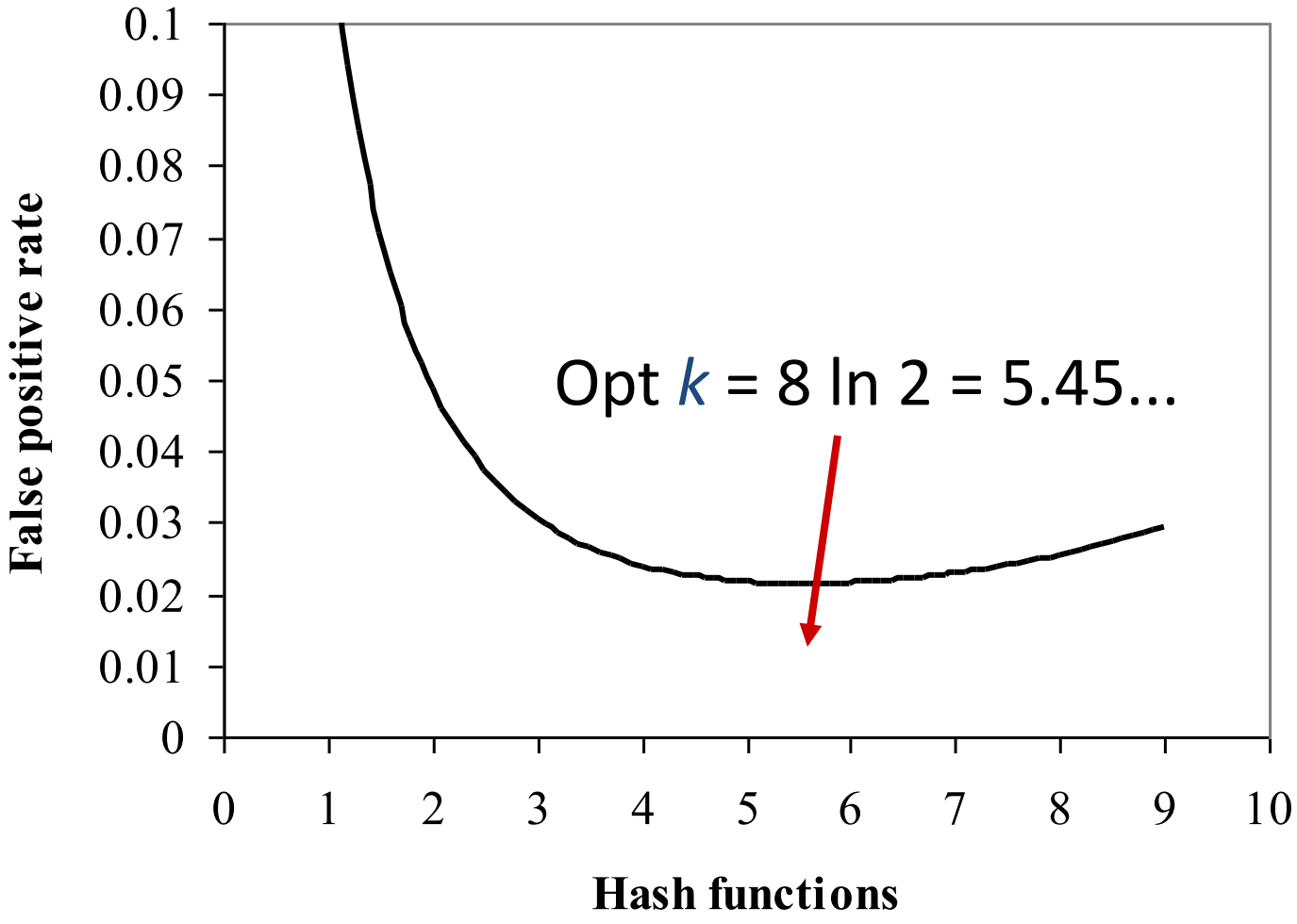
- If ρ is fraction of 0 bits in the filter then false positive probability is

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

- Approximations valid as ρ is concentrated around $E[\rho]$.
  - 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

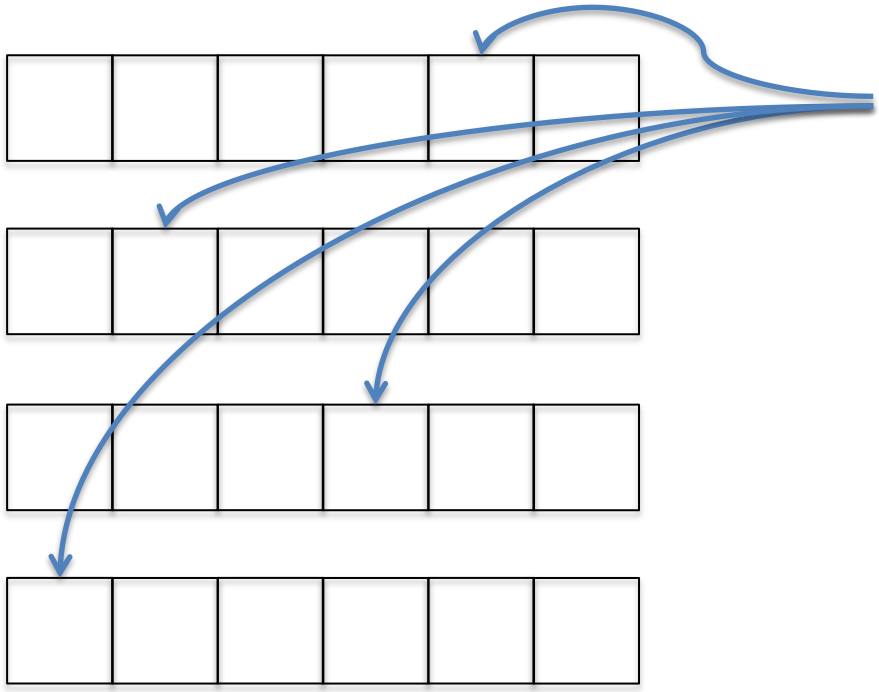
Example



$m/n = 8$

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

Split Bloom Filters



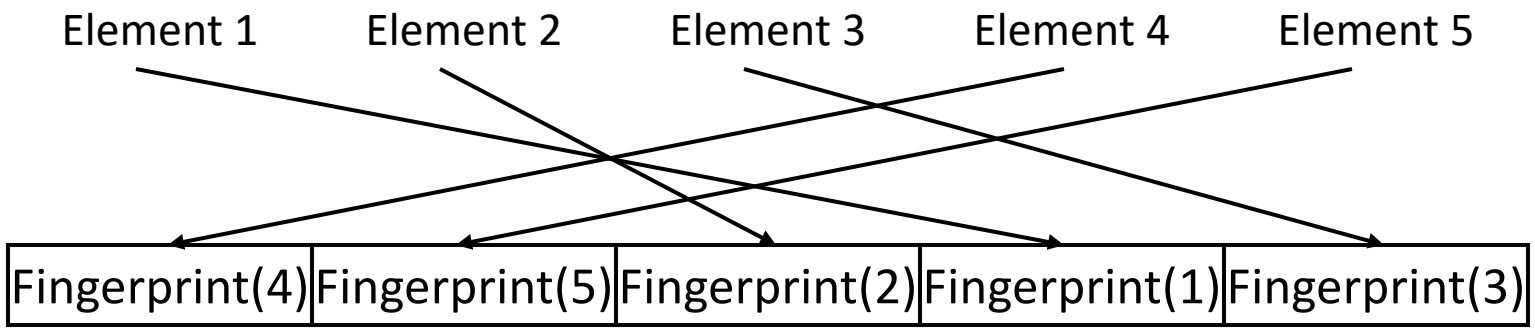
Key hashed to *k* cells

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

one has per group

“same” performance, easier to parallelize

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.

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]

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.

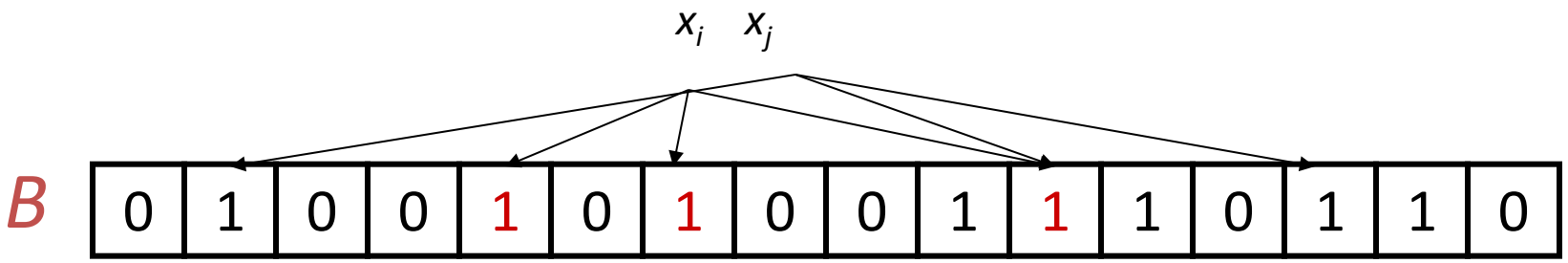
The main point

- 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):

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

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

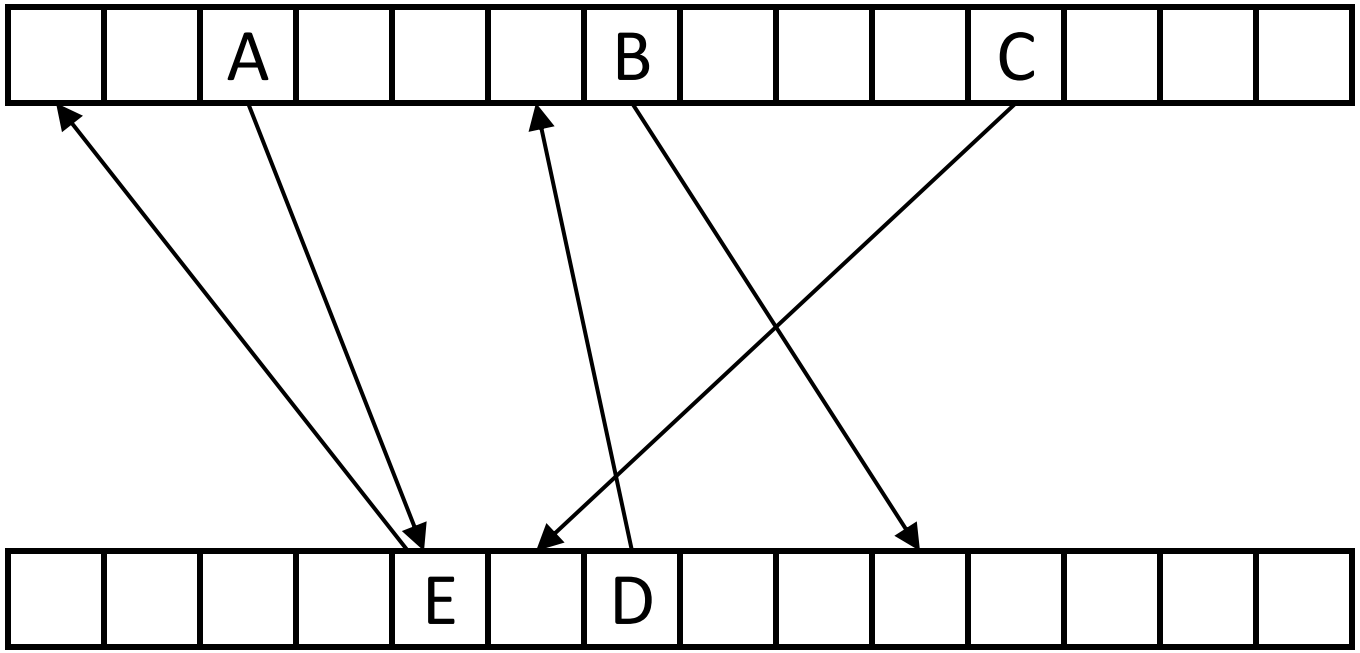
Counting Bloom Filters In Practice

- 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.

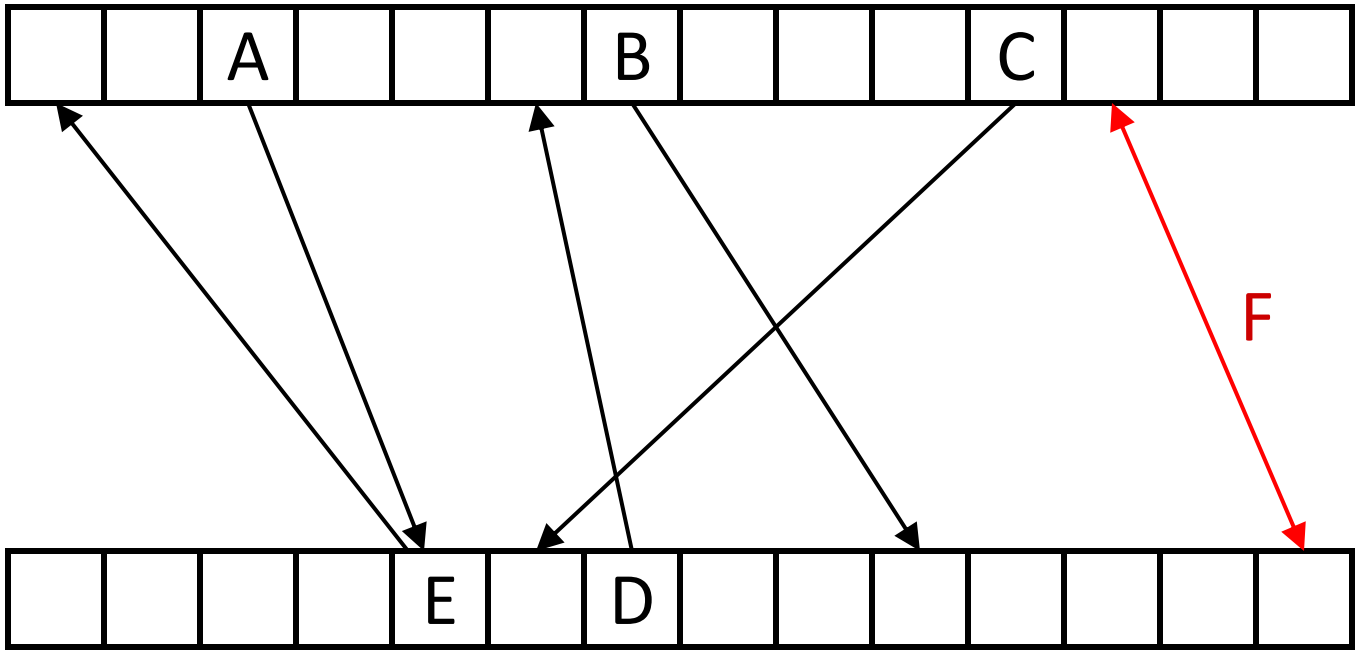
Cuckoo Hashing

- Basic scheme: each element gets two possible locations (uniformly at random).
- To insert *x*, check both locations for *x*. If one is empty, insert.
- If both are full, *x* kicks out an old element *y*. Then *y* moves to its other location.
- If that location is full, *y* kicks out *z*, and so on, until an empty slot is found.

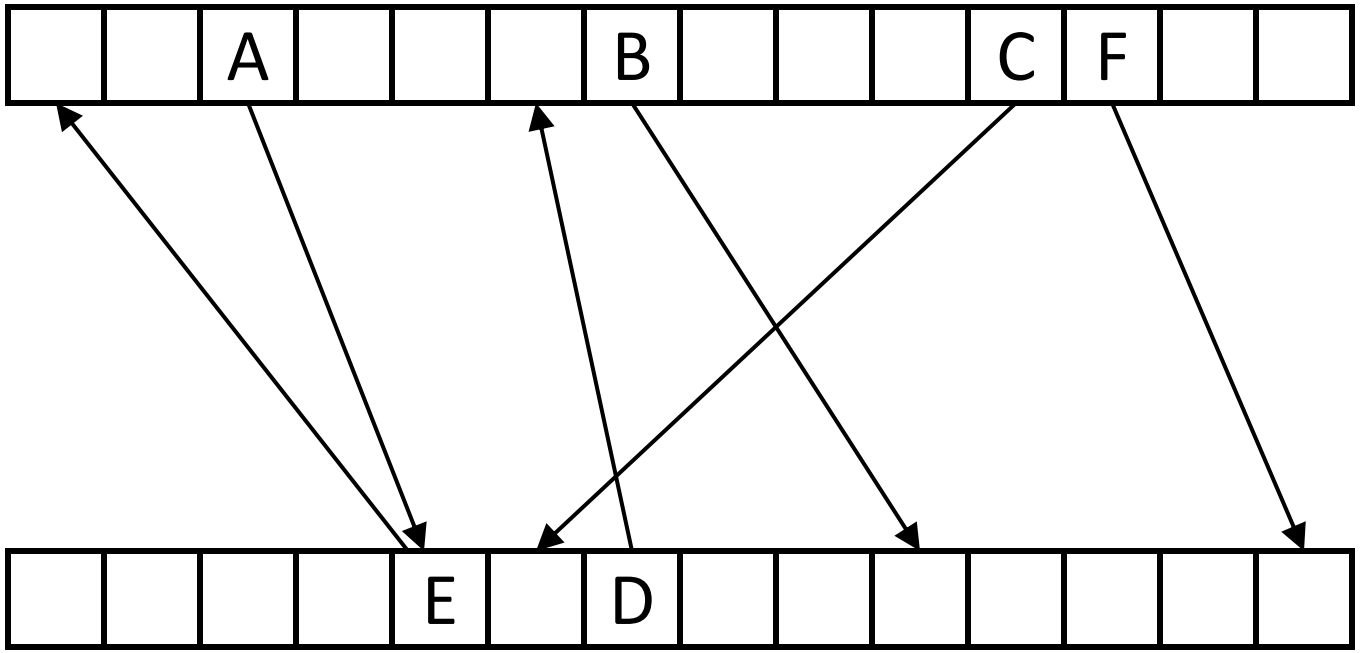
Cuckoo Hashing Examples



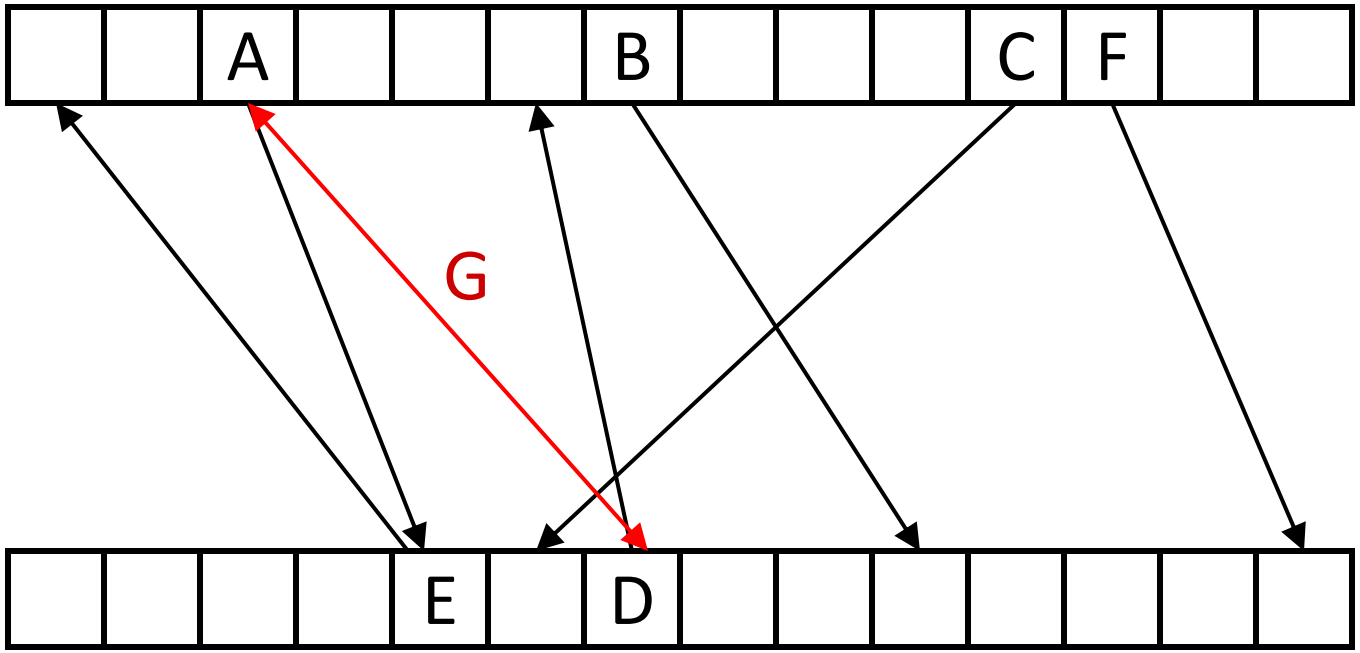
Cuckoo Hashing Examples



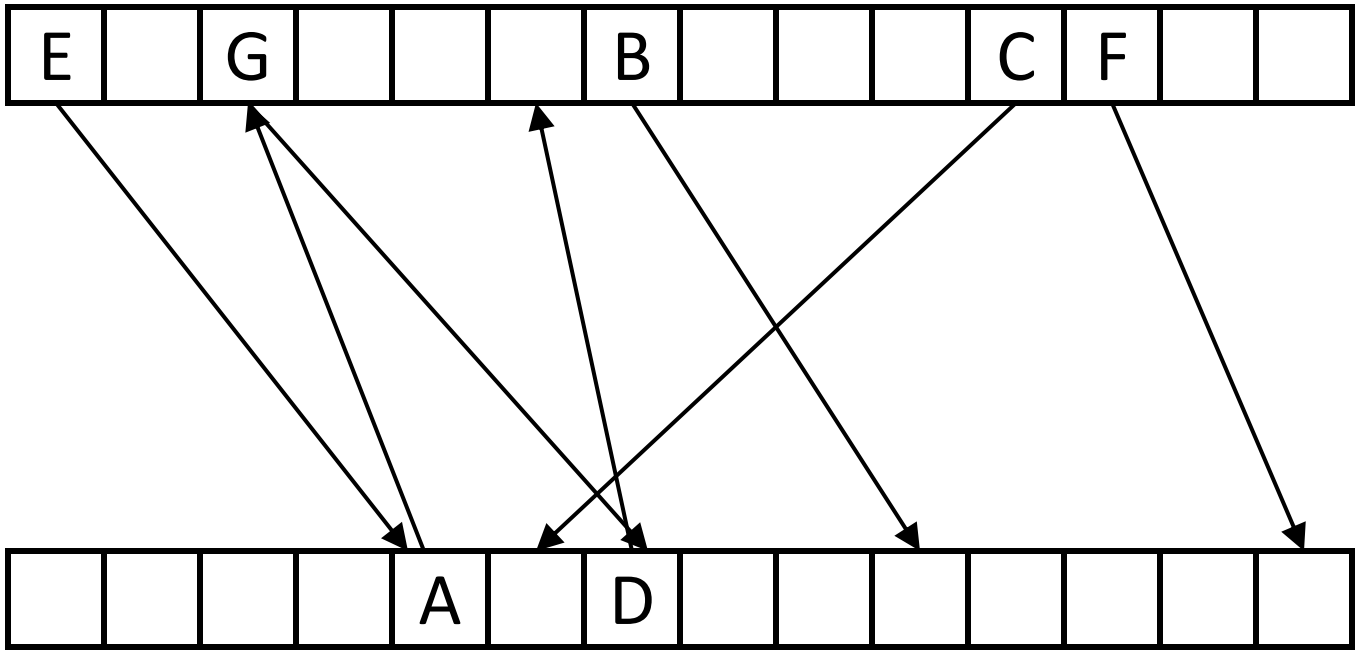
Cuckoo Hashing Examples



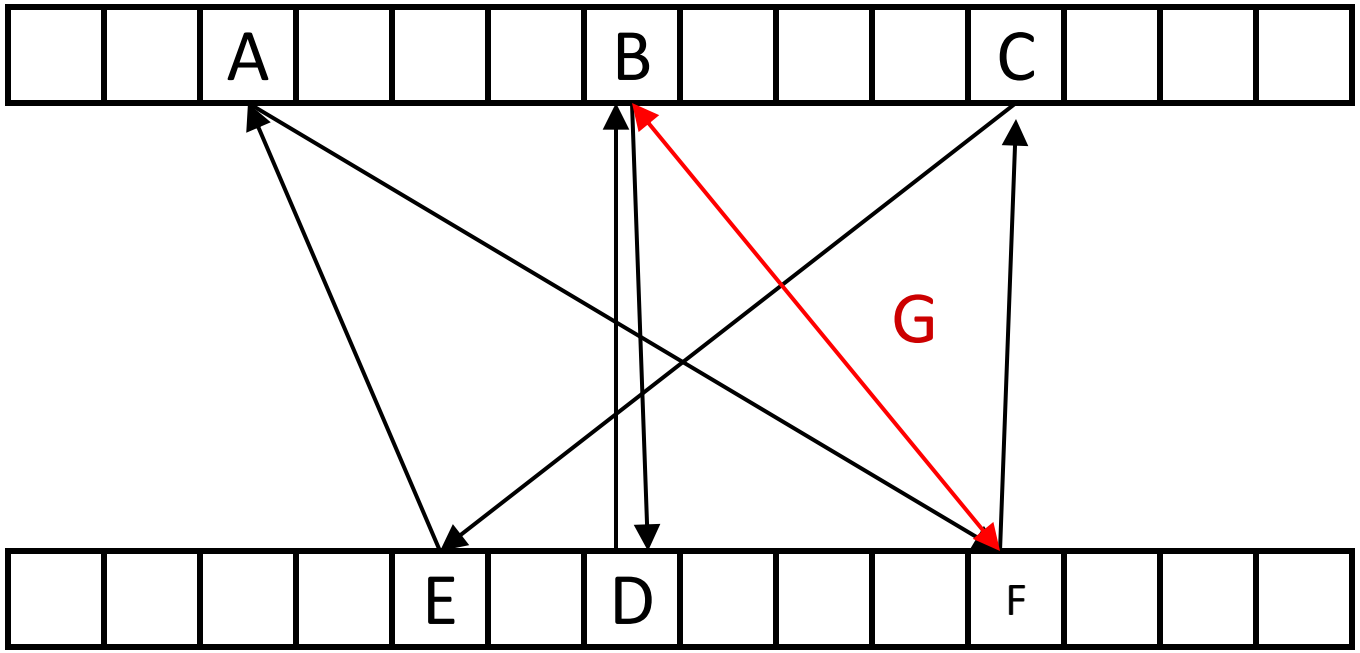
Cuckoo Hashing Examples



Cuckoo Hashing Examples



Cuckoo Hashing Examples



Good Properties of Cuckoo Hashing

- *Worst case constant lookup time.*
- High memory utilizations possible.
- Simple to build, design.

Cuckoo Hashing Failures

- Bad case 1: inserted element runs into cycles.
- Bad case 2: inserted element has very long path before insertion completes.
  - Could be on a long cycle.
- Bad cases occur with very small probability when load is sufficiently low.
- Theoretical solution: re-hash everything if a failure occurs.

Basic Performance

- For 2 choices, load less than 50%, n elements gives failure rate of $\Theta(1/n)$; maximum insert time $O(\log n)$.
- Related to random graph representation.
  - Each element is an edge, buckets are vertices.
  - Edge corresponds to two random choices of an element.
  - Small load implies small acyclic or unicyclic components, of size at most $O(\log n)$.

Natural Extensions

- More than 2 choices per element.
  - Very different : hypergraphs instead of graphs.
  - D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis.
  - Space efficient hash tables with worst case constant access time.
- More than 1 element per bucket.
  - M. Dietzfelbinger and C. Weidling.
  - Balanced allocation and dictionaries with tightly packed constant size bins.

Thresholds

Bucket Size 1

Choices	2	3	4	5	6	7
Load	0.5	0.918	0.976	0.992	0.997	0.999

2 Choices

Bucket size	1	2	3	4	5	8	10
Load	0.5	0.897	0.959	0.980	0.989	0.997	0.999

Stashes

- A failure in cuckoo hashing occurs whenever one element can't be placed.
- Is that really necessary?
- What if we could keep one element unplaced? Or eight? Or $O(\log n)$? Or εn?
- Goal : Reduce the failure probability.

A Simple Experiment

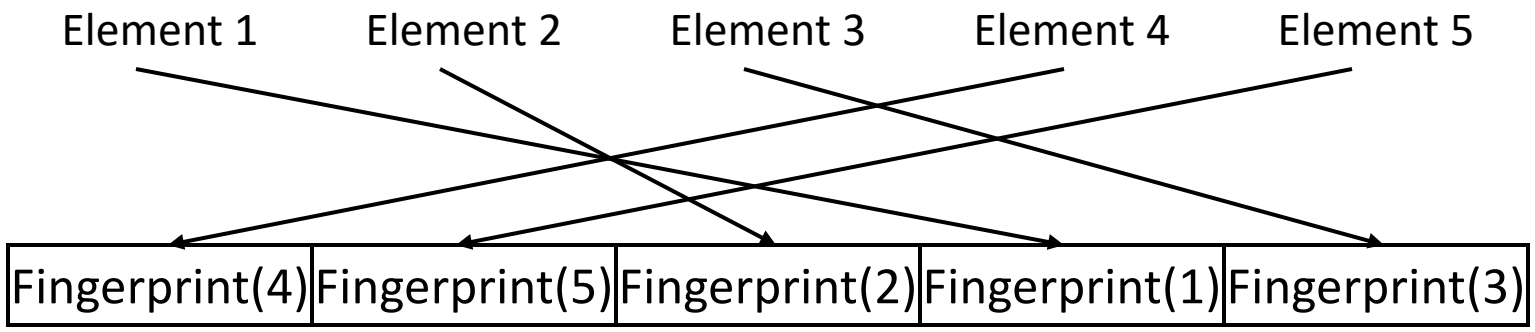
- 10,000 elements, table of size 24,000, 2 choices per element, 10^7 trials.

Stash Size	Needed Trials
0	9989861
1	10040
2	97
3	2
4	0

Bloom Filters via Hash Tables

- Recall one could obtain an optimal static Bloom filter using perfect hashing
- Can we use multiple-choice hashing/cuckoo hashing to get a “near-perfect” hash table for a Bloom filter type object?

Perfect Hashing Approach



Modern Update : Cuckoo Filters

- Use a cuckoo hash table to obtain a near-perfect hash table
- Store a fingerprint in the hash table
- Can support insertion **and deletion** of keys
- Very space efficient
  - From cuckoo hash table construction, with buckets that hold multiple keys.

Cuckoo Filters : Issues

- Consider cuckoo hash table, 2 choice per key, 4 fingerprints of keys per bucket.
- Buckets fill, an item has to be moved.
- How do we know where to move it?
  - We don’t have the key any more.
  - Just the fingerprint.

Partial-key Cuckoo Hashing

- Can’t use the key when moving a key.
- So we have to use the fingerprint instead.

$$h_1(x) = \text{hash}(x)$$

$$h_2(x) = h_1(x) \oplus \text{hash}(x\text{'s fingerprint})$$

- Note fingerprint is the same in both locations.
- Can compute h_1 from h_2 and vice versa with the stored fingerprint.

Partial-key Cuckoo Hashing

- Can't use the key when moving a key.
- So we have to use the fingerprint instead.

$$h_1(x) = \text{hash}(x)$$

$$h_2(x) = h_1(x) \oplus \text{hash}(x\text{'s fingerprint})$$

- Note fingerprint is the same in both locations.
- Can compute h_1 from h_2 and vice versa with the stored fingerprint.
- But now the two choices are limited, not completely random. Will this still work?

Partial-key Cuckoo Hashing

- Does it work?
- In practice, yes.
  - Essentially no discernible change in the threshold under reasonable settings.
- In theory, no.
  - You “need” logarithmic sized fingerprints…
  - But with a small constant factor.
  - So in practice it ends up OK.
- Provable bounds on performance of partial-key cuckoo hashing.
  - Eppstein has shown a simplification actually gets same performance as cuckoo hashing. (Just xor with fingerprint gives partitioned filter.)

Bit-Saving Tricks

- Every bit counts for space purposes.
- Bucket size of 4.
- Sort the fingerprints.
- Take the first 4 most significant bits.
- After sorting there are 3876 possible outcomes.
  - Less that 2^{12}.
  - So use only 12 bits to represent these 16.
  - Saves 1 bit per item.

Cuckoo Filter Performance

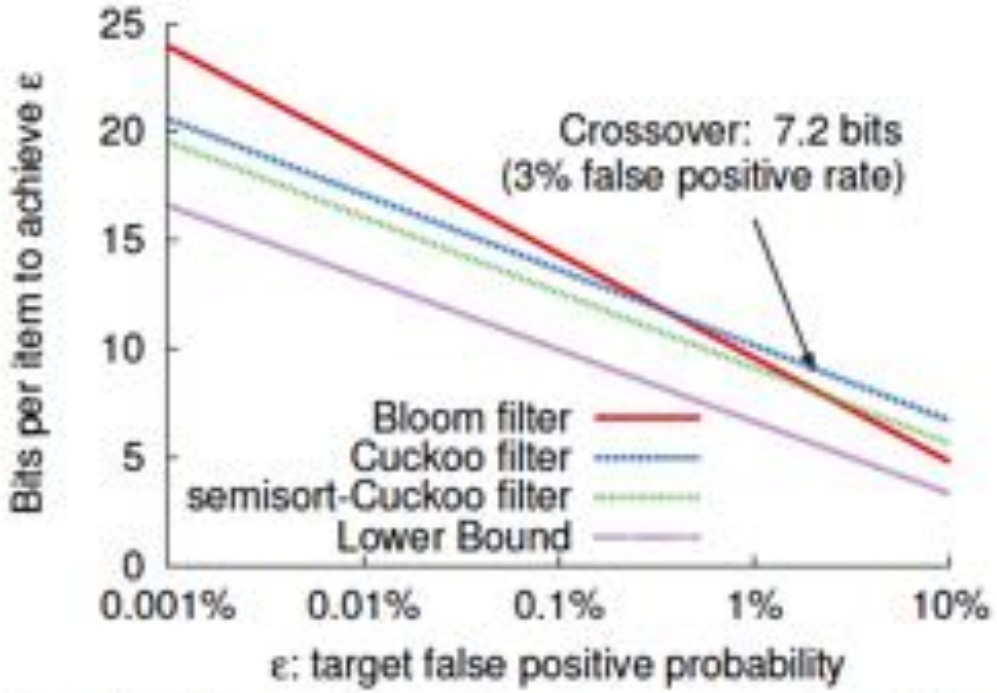


Figure 4: False positive rate vs. space cost per element. For low false positive rates (< 3%), cuckoo filters require fewer bits per element than the space-optimized Bloom filters. The load factors to calculate space cost of cuckoo filters are obtained empirically.

Motivation: Adaptivity

- "Bloom filter" data structures for membership checks are often used in situations with repeated queries
  - Whitelists / blacklists
  - Network security processing
- So "false positive probability" is not the right measure.
  - Rather, false positive *rate* over data stream.
- Goal: reduce/remove duplicated false positives for repeated queries.

Adaptive Cuckoo Filters : Key Differences

- Must keep a cuckoo hash table that mirrors the cuckoo filter, with elements in corresponding locations.
  - To see when we have false positives, and respond.
  - Can be kept in slower memory; only used to check for false positive, or insertion/deletion.
- Allows fingerprints to change, using different hash functions.

Adaptive : Buckets of Size 1

- Use 4 hash functions per element.
  - Achieves high loads, 95+ percent.
- Use *s* extra bits per cell, to keep track of hash function used for the current fingerprint.
- On lookup:
  - Check element against fingerprints in each of 4 hash buckets, using hash function from the *s* bits.
  - In case of match, check cuckoo table to make sure it's a true positive.
  - If false positive, increment the hash function counter and change the fingerprint.
    - Should remove the false positive.
    - Might create new ones.

Theoretical Analysis

- Requires simplifying assumptions
  - Assume requests for *A* non-set items are independently and uniformly generated over an interval.
    - More skewed requests are better for us.
  - Probability of a false positive on a bucket on a request depends on number of elements that hash to that bucket.
    - Approximately Poisson number of elements.
    - False positive with prob. = 2^{-#bits used in fingerprint}

Markov Chain Analysis

- Consider a single bucket.
- State is current hash function number.
- Will either
  - reach a state with no more false positives
  - cycle through all the fingerprints (at a slow rate)
- Can derive a corresponding (ugly, calculable) expression for false positive rate.

Adaptive : Buckets of Size > 1

- Use 2 hash functions, 4 cells per bucket.
  - Achieves high loads, 95+ percent.
- Use a different fingerprint for each cell location.
- On lookup:
  - Check element against all fingerprints in both cells.
  - In case of match, check cuckoo table to make sure it's a true positive.
  - If false positive, swap the element to new cell within the bucket.
    - Should remove the false positive.
    - Might create new ones.

Theoretical Analysis

- Requires simplifying assumptions
  - Assume requests for *A* non-set items are independently and uniformly generated over an interval.
    - More skewed requests are better for us.
  - Probability of a false positive on a cell on a request depends on number of elements that hash to that cell.
    - Approximately Poisson number of elements.
    - False positive with prob. = 2^{-#bits used in fingerprint}

Markov Chain Analysis

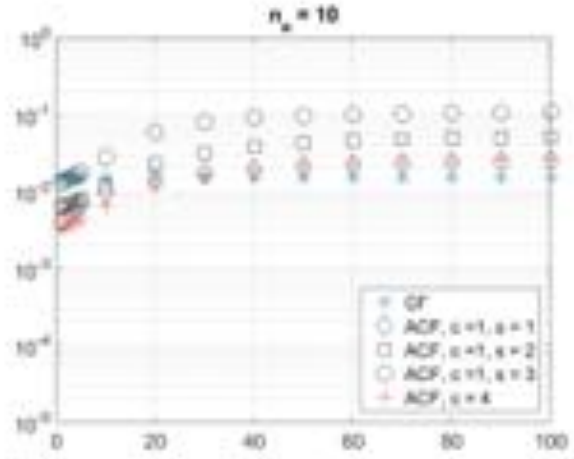
- Consider a single bucket.
- State is configuration of (up to) 4 elements within the bucket.
- Will either
  - reach a state with no more false positives
  - do random walk on set of 24 configurations
- Can derive a corresponding (ugly, calculable) expression for false positive rate.
  - But expression is too big to calculate efficiently; depends on number of elements hashing to each cell.
  - Use sampling to approximate expression for false positive rate.

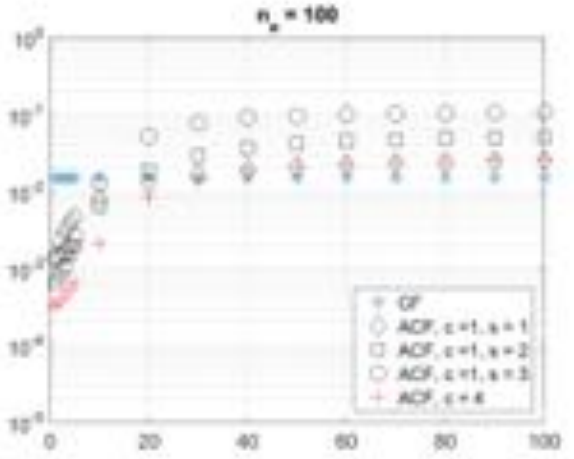
Simulation Setup

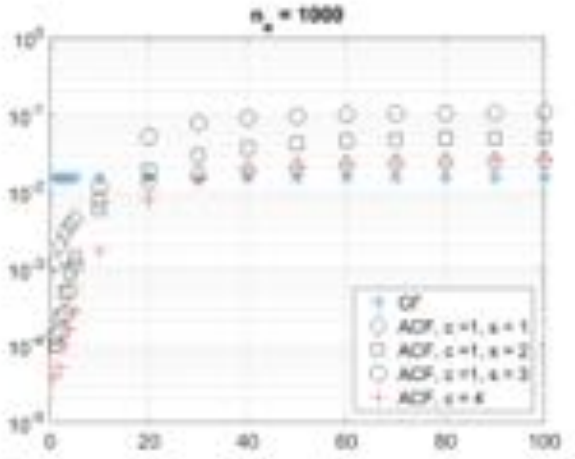
- Random experiments
  - Vary bits/cell (includes bits for storing hash function if needed): 8, 16
  - Vary bits per hash function counter: 1, 2, 3
  - Vary *A*/*S* ratio: [not-set-elements/set-elements] : 1, 2, 3, 4, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100.
  - Vary queries/element ratio: 10, 100, 1000
- CAIDA traces
  - 3 traces, each about 1 minute
  - Use flow 5-tuple as key
  - Some flows placed into filter, rest not, based on varying *A*/*S* ratio
  - Hundreds of thousands of total flows
  - Very skewed packets/flow distribution

Results for Random Tests

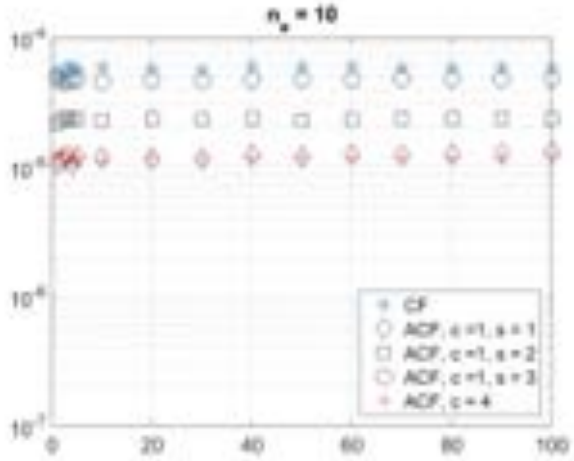
8 bits per cell

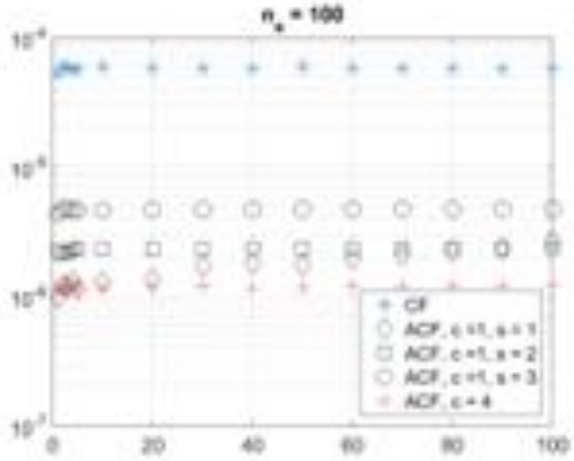


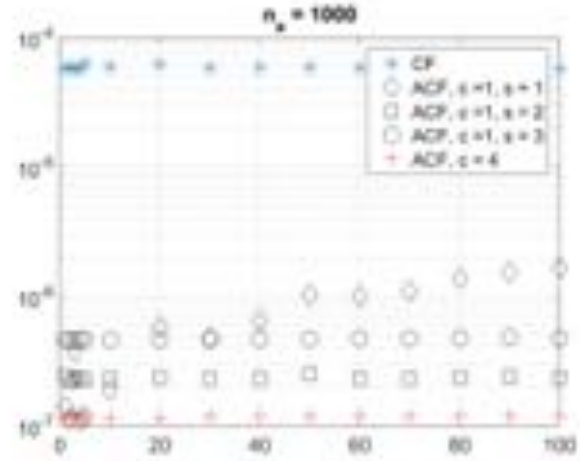




16 bits per cell

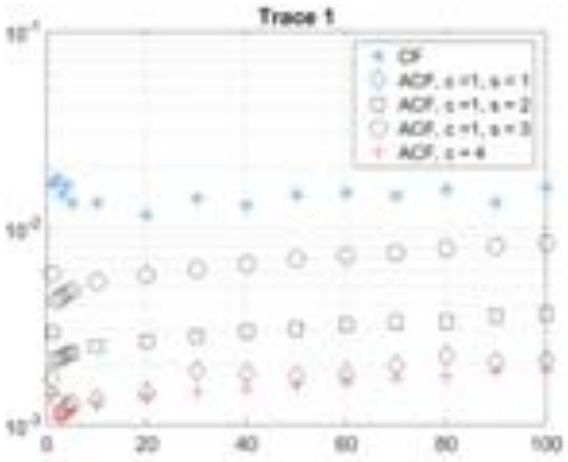


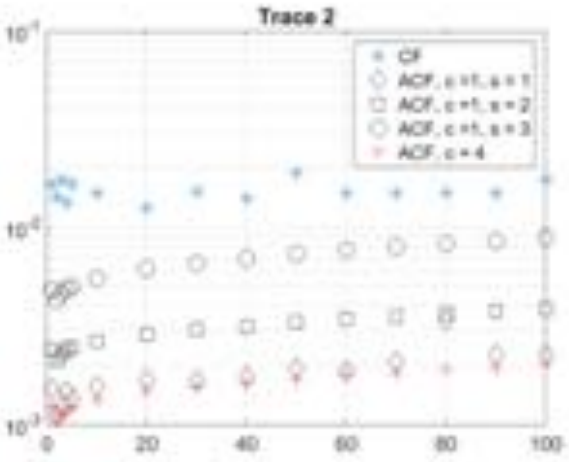


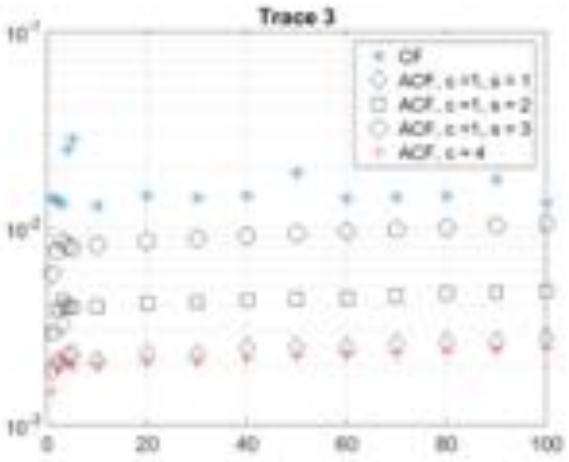


Results for CAIDA Tests

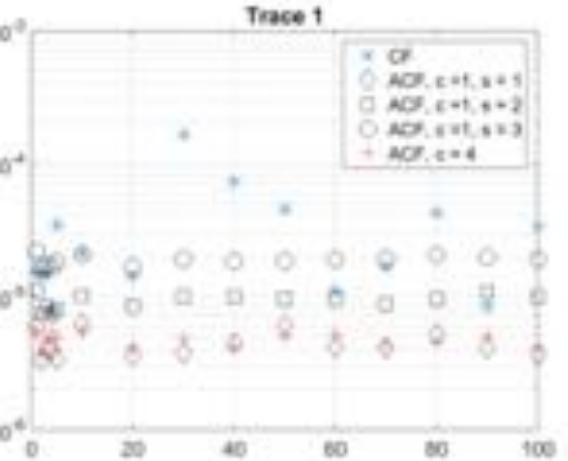
8 bits per cell

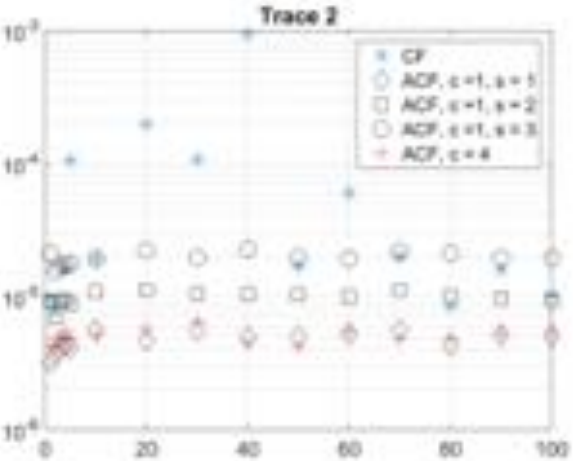


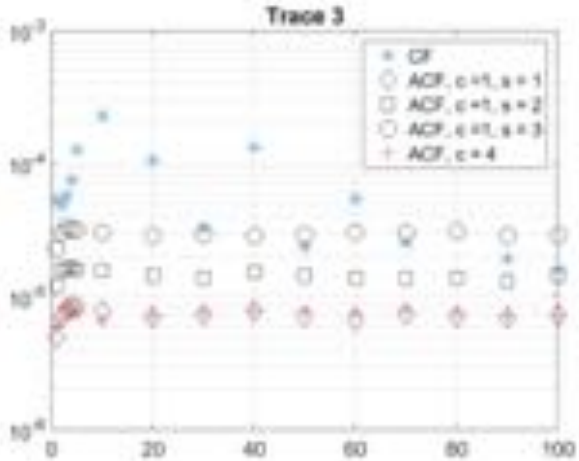




16 bits per cell





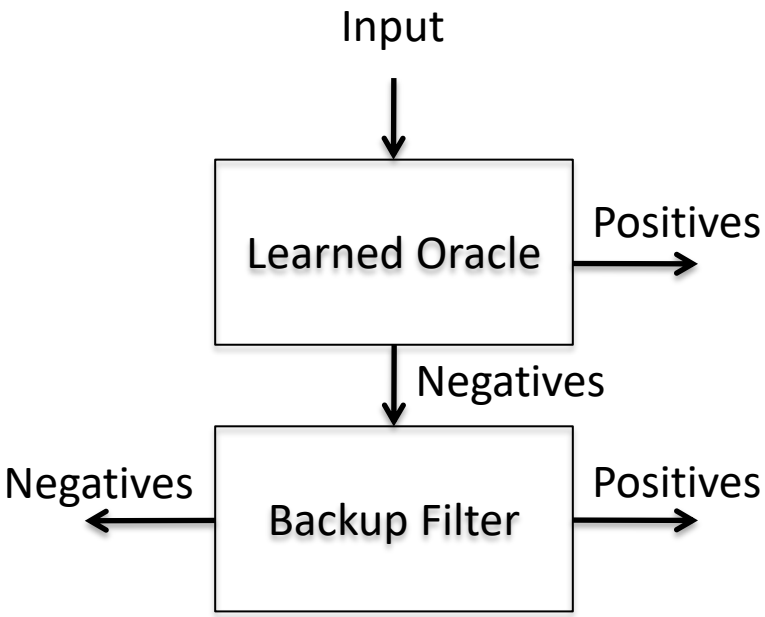


Experiment Summary

- ACFs perform better for lower *A*/*S* ratios, and for larger number of queries/element.
  - Small number of bits/cell, random queries only bad case for ACFs.
- Markov chain estimates are pretty accurate.
- Even better performance on skewed distributions of CAIDA traces.
  - In particular, *variance* is much less, because false positives are removed.
- 2 choices, 4 elements per cell often best or near-best configuration.
- Can reduce false positives by an order of magnitude.

Learned Bloom Filters

- Google Brain suggests can do better than standard Bloom filters
  - In a data dependent way
  - Assuming you can “learn” the set from the Bloom filter
- Use machine learning to develop a small-size oracle that provides probability an element is in a set
  - Oracle should (hopefully) give few false positives
  - And you need a backup to catch any false negatives.



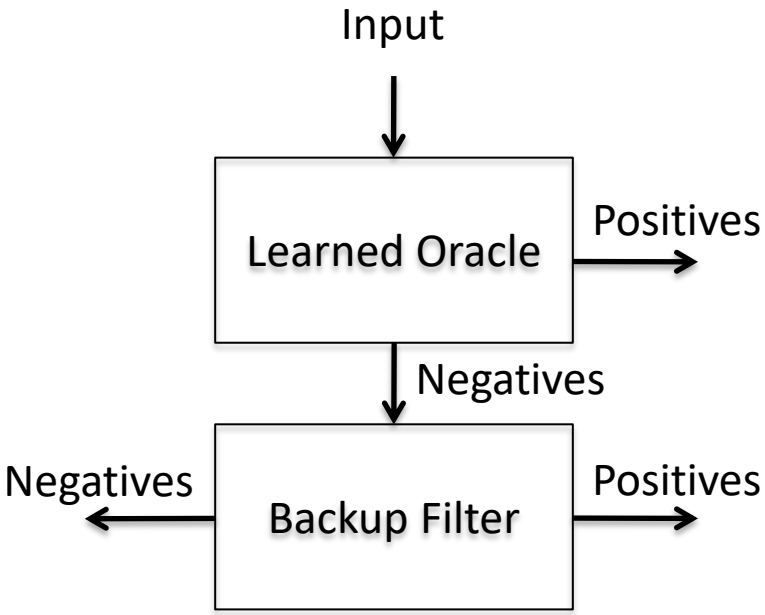
Items that the oracle says are very likely positives are treated as positives. Might be some false positives.

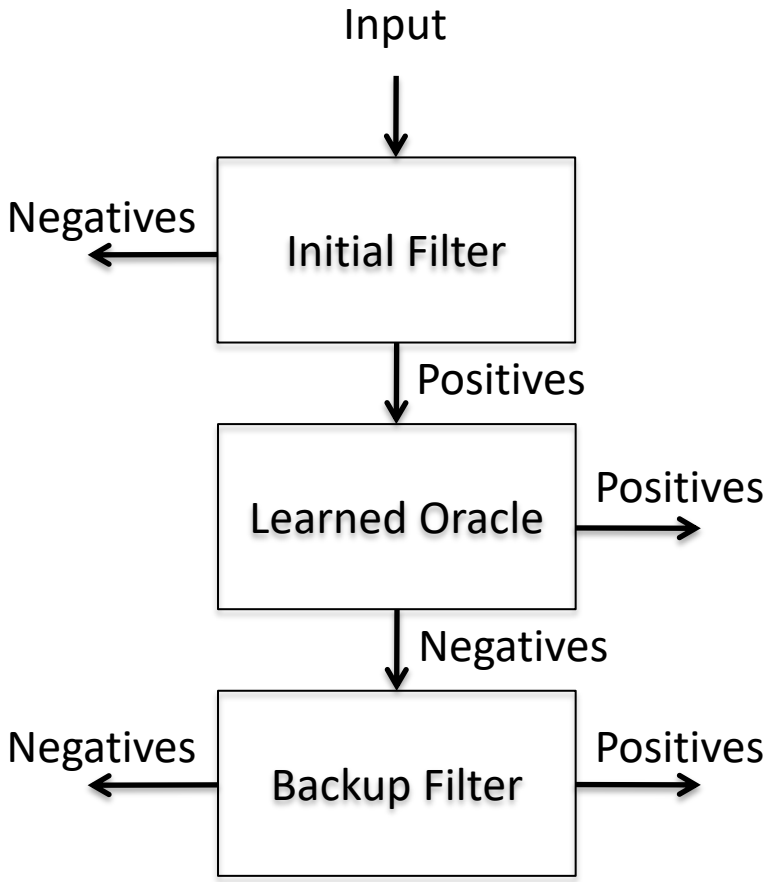
Items that are negatives might include possible false negatives! So we have a backup Bloom filter to catch oracle negatives in the set.

This might create additional false positives.

New Stuff

- Checkmy blog/arxiv for
  - Explanation of the different types of “guarantees” offered by Bloom filters and learned Bloom filters
    - Learned Bloom filters have “guarantees” only if test query data looks like future query data
    - Standard Bloom filter guarantees are stronger
  - Understanding of size needed for learned Bloom filter.
    - Equations to tell you how small/good the oracle needs to be.
  - Optimization of learned Bloom filters by using sandwiching.
    - Better to put a Bloom filter before and after the oracle.
    - Remove false positives at the source!
  - Extended to learned Bloomier filter analysis.


Input
Learned Oracle
Positives
Negatives
Negatives
Backup Filter
Positives


Input
Negatives
Initial Filter
Positives
Learned Oracle
Positives
Negatives
Negatives
Backup Filter
Positives

Conclusions / Future Work

- Cuckoo filter improves on Bloom filter constructions.
- Adaptivity appears useful for set membership data structures.
- Can maintain simplicity, high load while having adaptivity.
  - Downside: requires space to store all elements (in an offline structure).
  - Upside: large reduction in false positives.
- Analysis is difficult, and context dependent.
- Other structures/uses for adaptivity?
- Learned Bloom filters, and related data structures?