BLOOMIN' MARVELLOUS

WHY PROBABLY CAN BE BETTER THAN DEFINITELY

Adrian Colyer, @adriancolyer

AGENDA

- Introduction & motivation
- Bloom filters
- Tuning
- Hashing
- Related applications of PDSs

TRAFFIC SURVEILLANCE

For every traffic camera in London, for every 24 hour period, answer the question 'did a vehicle with plate <license no> pass this camera?'

(assume we have reliable video feed -> license number conversion available for each camera)

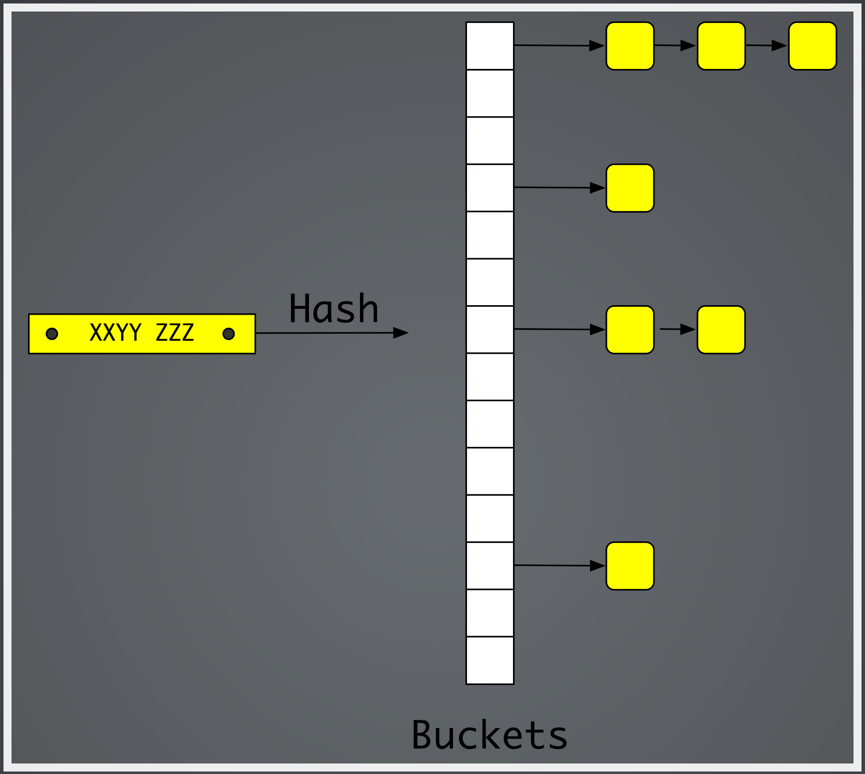
SET MEMBERSHIP

Given the set of all vehicles that passed a camera, we want an efficient membership test.

APPROACHES (PER CAMERA SITE)

- Look-up table: $LicensePlate \rightarrow Bool$
- Keep a list of every plate we see
- Keep a HashSet of every plate we see

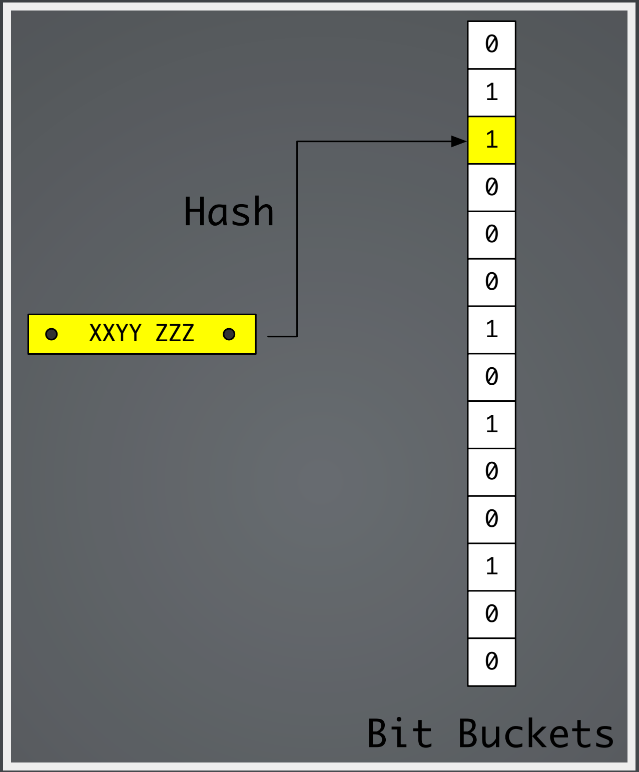
HASHSET



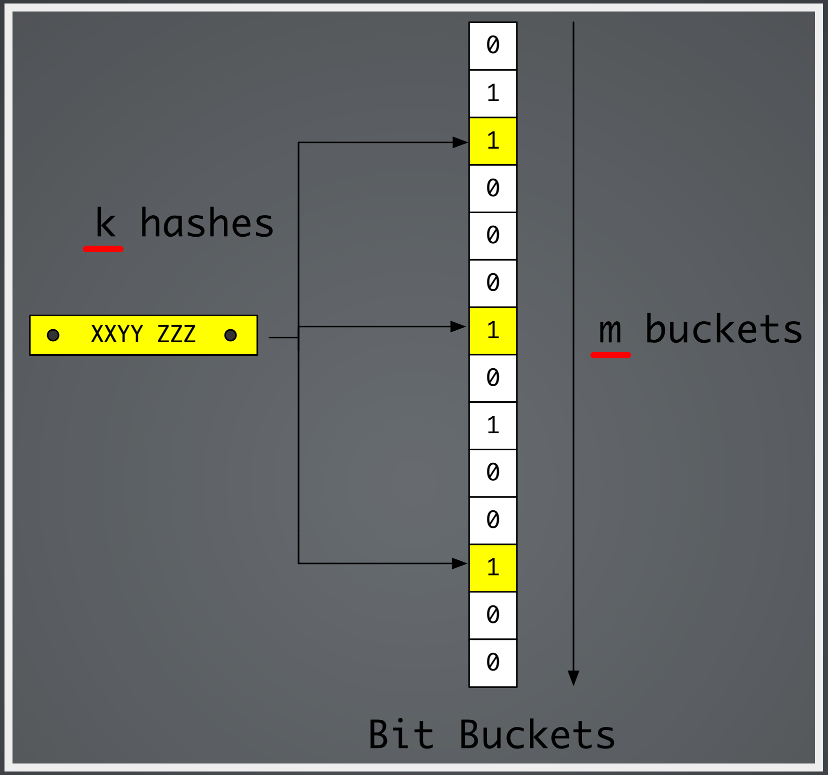
CAN WE DO BETTER?

- Time: avg. O(1), worst O(n)
- Space: O(n)
- We never need to enumerate the members...

JUST THE HASH - MUCH LESS SPACE



COPING WITH HASH COLLISIONS



BLOOM FILTERS

- m-bit vector
- k independent hashes
- to add an element: set bit for each hash
- membership test: hash and verify all bits set

BLOOM FILTER PROPERTIES

- No false negatives
- May generate false positives
  - Error rate can be tuned by varying m and k
- Constant in both space and time, regardless of number of items in the set
- Can only *add* items
- Very useful as a cheap guard in front of an expensive operation

IN THE WILD

- HBase, BigTable, Cassandra, ...
- Distributed IMDG
- Bloom joins
- Malicious URL identification in Chrome
- Networking (e.g. loop detection in routing)
- ...

TUNING BLOOM FILTER ACCURACY

Given an expected number of members n, m bits, and k hash functions, how should we choose m and k in order to achieve an acceptable false positive rate?

Consider the insertion of an element, and an individual hash function. The probability that a given bit is set is $1/m$. Therefore the probability that a given bit is not set is:

$$1 - \frac{1}{m}$$

And the probability that a given bit is not set by all k hash functions is:

$$\left(1 - \frac{1}{m}\right)^k$$

The probability that a given bit is *not* set after inserting n elements is simply:

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

and the probability an indidividual bit *is* set is therefore

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

What is the probability p we test an element that is *not* in the set, and get back all 1s? (A false positive).

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

(all k bits must be 1)

and the optimal value of k given m and n (so as to minimise p) is

$$k = \frac{m}{n} \ln 2$$

We always want to be optimal! Substituting for k in the formula for p and then solving for m gives:

$$m = -\frac{n \ln p}{(\ln 2)^2}$$

APPLYING THESE RESULTS:

1. Decide on an acceptable false positive rate p, and estimate number of members in the set, n.
2. Set $m = -\frac{n \ln p}{(\ln 2)^2}$
3. Set $k = \frac{m}{n} \ln 2$

EXAMPLES

Set size	False positive %	m	k	bits per member
100,000	1%	~960,000 (117KB)	7	9.6
100,000	0.1%	~1,440,000 (176KB)	10	14.4
10M	1%	~96M (11.4MB)	7	9.6
10M	0.1%	~144M (17MB)	10	14.4

URL USE CASE COMPARISON

Assume an average URL is 35 characters, 10M URLs...

- HashSet requires at least 350MB to store
- Bloom Filter with 1% false positive requires 11.4MB
    - About 3% of the space!

BACK TO OUR TRAFFIC PROBLEM...

There are 110 count points in Westminster alone

~20,000 vehicles/day/point

~23.5Kb per count point (1% false positive)

Only 2.5MB per day for all of Westminster!

A HASHING DIGRESSION

- Where can we find k independent hash algorithms?
- And how good does the hash have to be?

INDEPENDENCE

Events A and B are independent if

$$Pr(A \cap B) = Pr(A).Pr(B)$$

In other words:

$$Pr(A|B) = Pr(A)$$

and

$$Pr(B|A) = Pr(B)$$

MUTUAL INDEPENDENCE

Given a set of random variables $X_1, X_2, \ldots, X_n$, any subset $I \subseteq [1, n]$ and any values $x_i, i \in I$

Then $X_1, X_2, \ldots, X_n$ are mutually independent if

$$Pr\left(\bigcap_{i \in I} X_i = x_i\right) = \prod_{i \in I} Pr(X_i = x_i)$$

K-WISE INDEPENDENCE*

Restrict $|I| \leq k$, then our set of random variables $X_1, X_2, \ldots, X_n$ is k-wise independent if, for all subsets of k variables or fewer

$$Pr\left(\bigcap_{i \in I} X_i = x_i\right) = \prod_{i \in I} Pr(X_i = x_i)$$

When $k = 2$ we call this *pairwise independence*

* this is not the same k as the k hash functions in our bloom filter!

PAIRWISE EXAMPLE

Consider three variables a, b and x, where a and b are truly random, and $x = a + b$.

- Pairwise-independence
- But not 3-wise

THEORY AND PRACTICE

In theory, hash functions have uniform distribution over the range, and independence of hash values over the domain.

In practice such hash functions are expensive to compute and store. For non-cryptographic applications we can use more efficient algorithms with weaker guarantees.

STRONGLY UNIVERSAL HASH FUNCTIONS

Consider a set U (the universe) of values we want to hash, and a *family* of hash functions $\mathcal{H}$ that create an n-bit hash.

- For any k elements $x_1, x_2, \ldots, x_k \in U$
- And for any randomly selected hash function $h \in \mathcal{H}$
- Uniform distribution:
$$Pr(h(x_1) = y_1) = 1/n$$

AND K-WISE INDEPENDENCE

Given k elements $x_1, x_2, \ldots, x_k \in U$ and k output values $y_1, y_2, \ldots, y_k$

$$Pr\left(\bigcap_{i=1}^{k} h(x_i) = y_i \right) = \frac{1}{n^k}$$

when $k = 2$ we have a 2-universal or *pairwise independent* hash family

2-UNIVERSAL GOOD ENOUGH?

- Mitzenmacher and Vadhan show that with minimal entropy in data items, 2-universal hashes perform as predicted for truly random hashes.
- Bloom Filters and *non-cryptographic applications*
    - use k hash functions from a 2-universal family
- Caution required when influenced by external input: hash DoS attacks can exploit collisions

2-UNIVERSAL: SIMPLE IN PRINCIPLE

$$h(x) = ax + b \mod p$$

- p is a prime
- a and b chosen uniformly between 0 and $p - 1$ for each hash function in family

SELECTED HASH ALGORITHMS

- MurmurHash3
  - Can hash about 5GB/sec on dual-core 3.0GHz x64
  - Very good key distribution
- xxhash
  - Very good performance and distribution
- SipHash
  - competitive performance
  - protects against hash DoS attacks

EFFICIENT BLOOM FILTER IMPLEMENTATION

- Hashing is the most expensive operation
- Kirsch and Mitzenmacher show that we can simulate k independent hash functions using only 2 base functions.
- Extended double hashing: to hash input $u \in U$
    - $(h_1(u) + ih_2(u) + f(i)) \mod m$
    - for $i \in 1..k$
    - and $f(i)$ is a total function from $[k] \rightarrow [m]$
- See Cassandra implementation notes (Ellis)

RELATED APPLICATIONS

STREAM SUMMARIES WITH SKETCHES

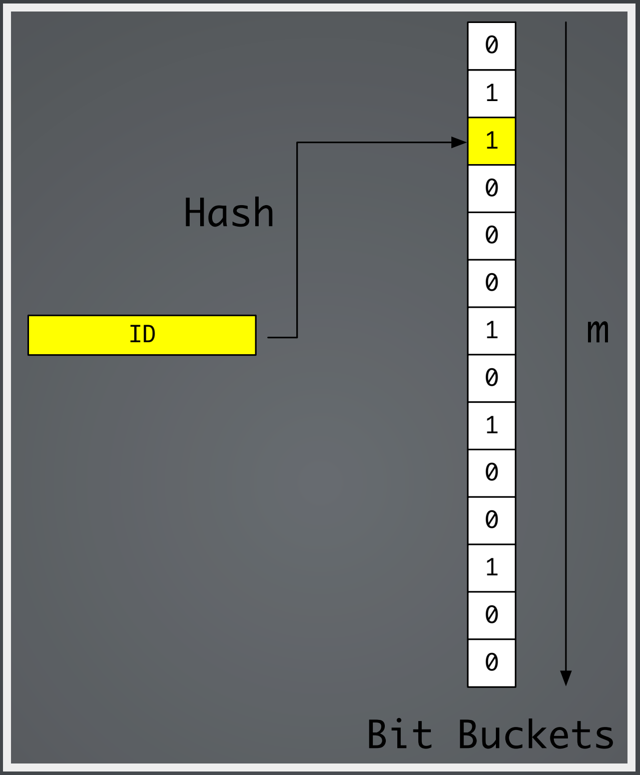
- Estimating cardinality (# of distinct values seen)
- Estimating frequency with which values appear in a stream
- Finding heavy hitters (top-k most frequent items)
- Quantile estimations
- Range estimations
- ...

CARDINALITY ESTIMATION

Clearspring case study: 16 character Ids, 3 billion events/day, how many distinct ids in the logs?

- HashSet with 1 in 3 unique Ids still needs at least 119GB
- Simple solution - linear counting
- Very space efficient solution - HyperLogLog

LINEAR COUNTING



- Estimate the number of distinct elements n using:

$$n = -m \ln \frac{m-w}{m}$$

- where w is the *weight* of the bitset, i.e. the number of 1s
- Rule of thumb for choosing m: about 0.1 bits per expected upper bound of measured cardinality
- ~12MB for the ID problem (vs 119GB)

HYPERLOGLOG

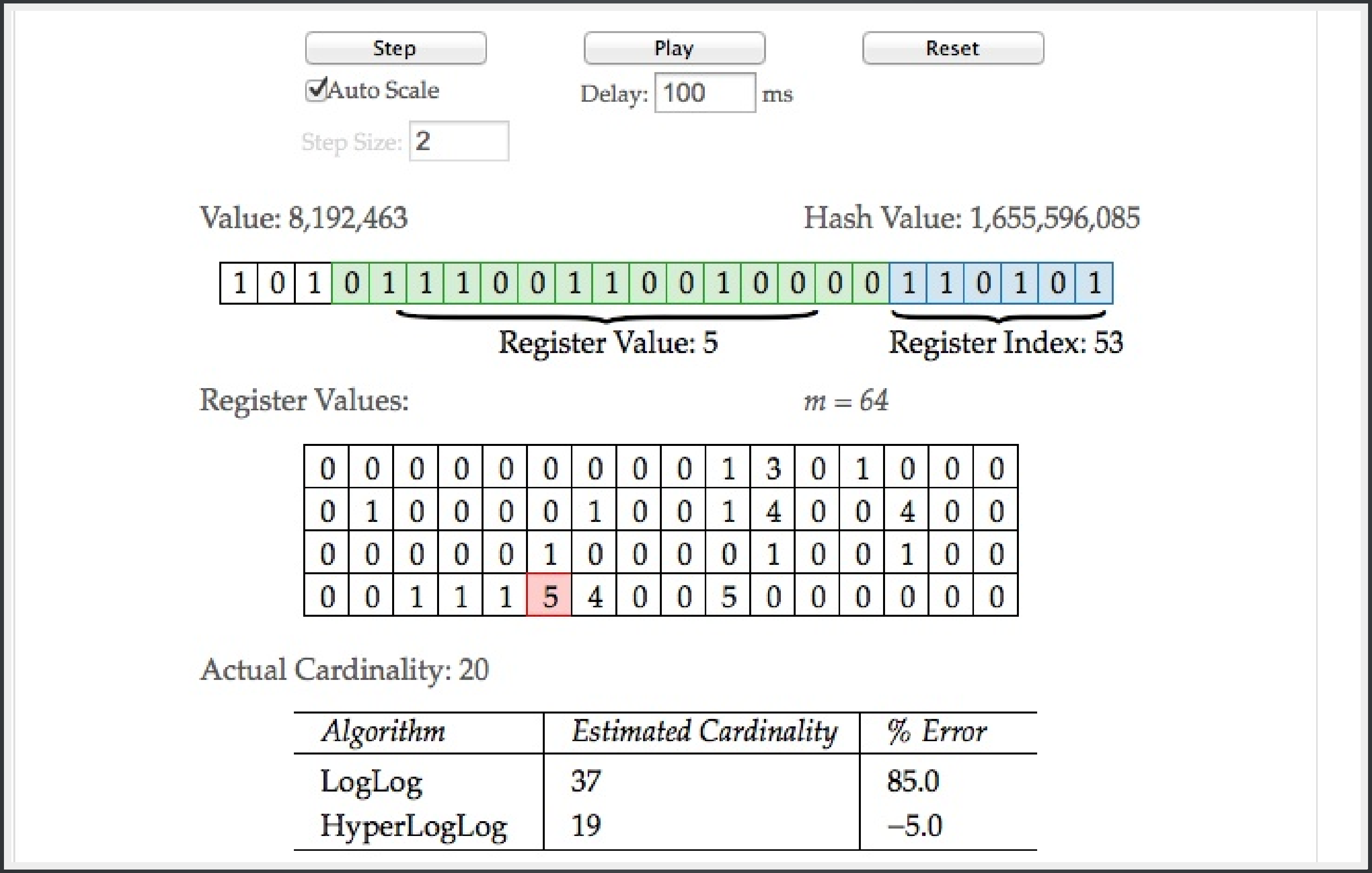
- More sophisticated, but still based on hashing and probabilities
- To estimate cardinalities up to 1 billion, with a% accuracy needs m bits:

$$m = 5\left(\frac{1.04}{a}\right)^2$$

- 2% accuracy, ~ 1.5KB!

INTERACTIVE DEMONSTRATION

AK Tech blog



☑ Auto Scale

Delay: 100 ms

Step Size: 2

Value: 8,192,463

Hash Value: 1,655,596,085

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

Register Value: 5

Register Index: 53

Register Values:

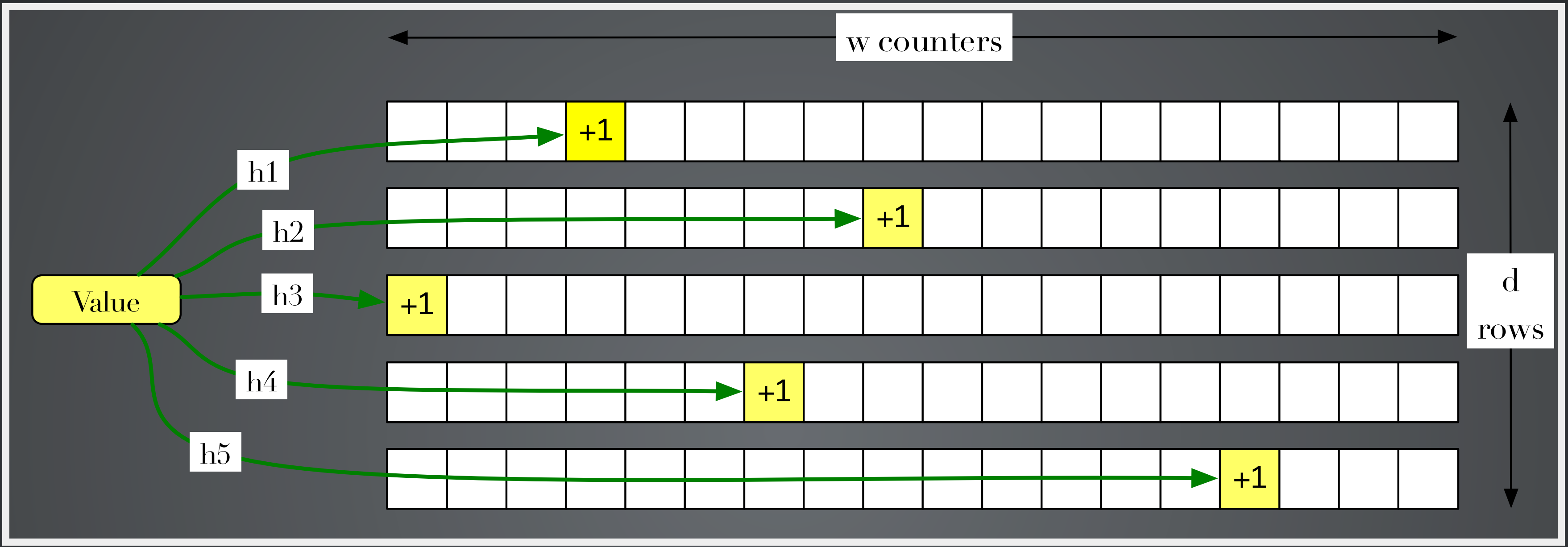
$m = 64$

0	0	0	0	0	0	0	0	0	1	3	0	1	0	0	0
0	1	0	0	0	0	1	0	0	1	4	0	0	4	0	0
0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0
0	0	1	1	1	5	4	0	0	5	0	0	0	0	0	0

Actual Cardinality: 20

Algorithm	*Estimated Cardinality*	*% Error*
LogLog	37	85.0
HyperLogLog	19	–5.0

COUNT-MIN SKETCH



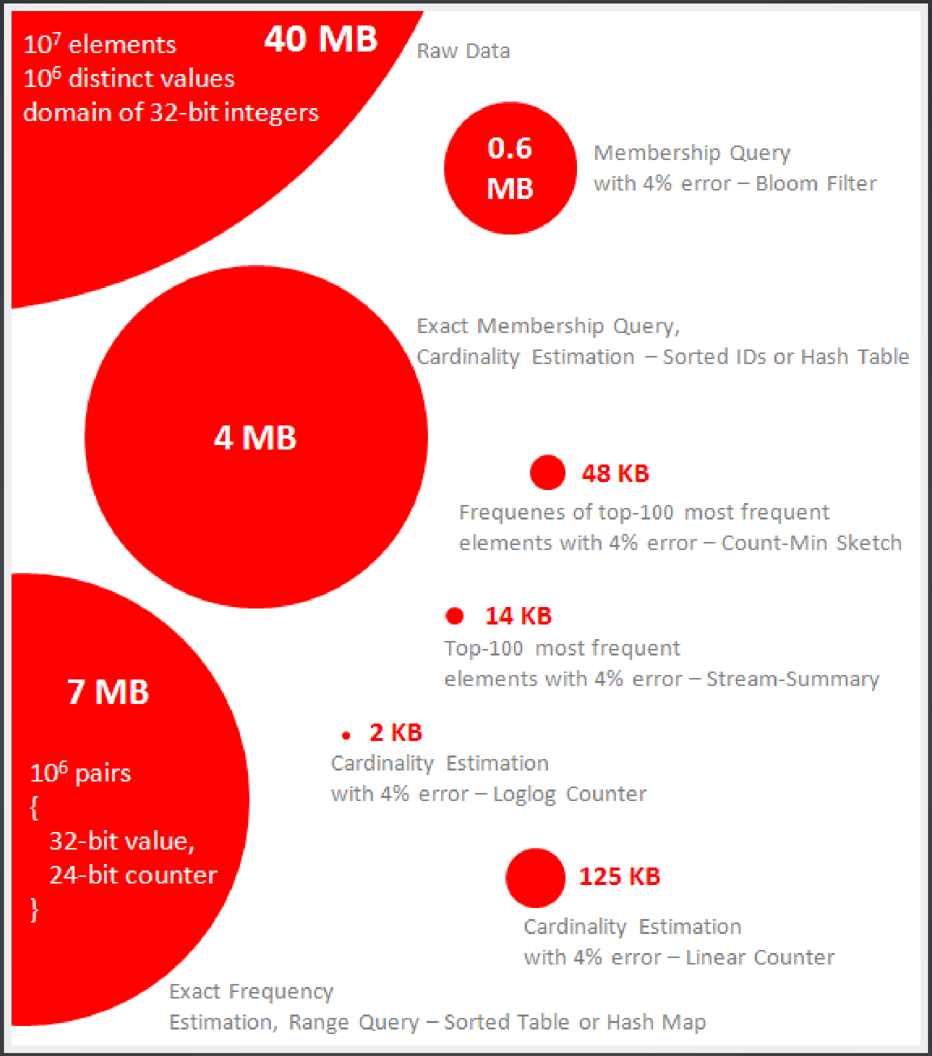
Pairwise independent hash functions

FREQUENCY ESTIMATION OF ITEM i

- Lowest count at hash locations

$$f(i) = \min_{j=1..d} C[j, h_j(i)]$$

- Improve accuracy by factoring in adjacent counter in each row score (Count sketch)
  - Subtract value to the left for even rows, to the right for odd rows
  - Accounts better for random noise
- See also Count-Mean-Min variation...
- This family of algorithms work best with highly skewed data



10^7 elements
10^6 distinct values
domain of 32-bit integers
40 MB
Raw Data
0.6 MB
Membership Query
with 4% error – Bloom Filter
4 MB
Exact Membership Query,
Cardinality Estimation – Sorted IDs or Hash Table
48 KB
Frequenes of top-100 most frequent
elements with 4% error – Count-Min Sketch
14 KB
Top-100 most frequent
elements with 4% error – Stream-Summary
7 MB
10^6 pairs
{
32-bit value,
24-bit counter
}
2 KB
Cardinality Estimation
with 4% error – Loglog Counter
125 KB
Cardinality Estimation
with 4% error – Linear Counter
Exact Frequency
Estimation, Range Query – Sorted Table or Hash Map

Probabilistic Data Structures for Web Analytics and Data Mining | Highly Scalable Blog - Ilya Katsov

RESOURCES

- Bloom filters by Example (Bill Mill)
- Probabilistic Data Structures for Web Analytics and Data Mining | Highly Scalable Blog - Ilya Katsov
- Sketch Techniques for Approximate Query Processing (Cormode)
- Probability and Computing (Mitzenmacher and Upfal)
- stream-lib - Apache 2.0 Licensed Java implementations