

Designs of Algorithms and Programming for Massive Data

DS8001

Andriy Miranskyy

Oct. 17, 2016

Midterm

- This is the last lecture that will be covered on the midterm
- I am out of office next Monday – Mefta will give a tutorial on R, Spark, and fundamentals of parallelization
- If you have questions about midterm – ask them now or via email later. Alternatively, send me an email to setup an appointment.

Survey

- https://docs.google.com/a/ryerson.ca/forms/d/1QKRuXWNSjlZhLOUJ6BGeWY1P8j5zG1hw7ZG6w-Of9mA/edit?usp=drive_web

Outline

- Randomized Algorithms
- Probabilistic Algorithms and Data structures
- Estimators of
 - Cardinality (Count-distinct)
 - Frequency of elements
 - Presence of an element

Randomized Algorithms

Randomized Algorithms

- Randomized Algorithms: have some randomness as part of its logic*
- Las Vegas
 - Always produce correct result
- Probabilistic Algorithm (Monte Carlo)
 - May return incorrect result
 - Don't confuse with Monte Carlo Method

* Executed by probabilistic Turing machine

Example

- Input: unsorted array of integers A of length n
- Output: what is the location of 123 in A ? Assume that 123 is present in A .

Example

What happens when $\text{max_iteration_count} < n$?

- Las Vegas (randomized)

```
while(true):  
    i = randomly select an integer between 1 and n  
    if (A[i] == 123):  
        return i
```

- Monte Carlo* (probabilistic)

```
iteration_count = 0  
max_iteration_count = ... // set by a user  
while(true):  
    i = randomly select an integer between 1 and n  
    if (A[i] == 123):  
        return i  
    iteration_count = iteration_count + 1  
    if(iteration_count == max_iteration_count*):  
        break
```


We will focus on probabilistic algorithms and data structures today

Typical problems

- Given a number of elements (words, ids, etc.)
 - Count the number of distinct elements (cardinality)
 - Find how many times each element occurs in the set
 - Find if an element exists in the set

How would you approach these problems?

Typical problems

- Given a number of elements (words, ids, etc.)
 - Count the number of elements (cardinality)
 - Find if an element exists in the set
 - Find how many times each element occurs in the set
- But what if the number of elements is large?

Case study¹

- Count the number of distinct visitors to a site per day
- User id is represented by 16 byte string
 - e.g.: mseitg1ywi13fcn
- One² billion visits per day.
 - $16 * 1E9 / 1024^3 \approx 45\text{GB}$ is needed for storage
 - Assuming that we have 3 visits per user on average, then we will need $\approx 15\text{ GB}$ to store distinct ids if we load the data into array.
 - But array makes it difficult to track distinct values...

1. <http://highscalability.com/blog/2012/4/5/big-data-counting-how-to-count-a-billion-distinct-objects-us.html>

2. They actually had 3E9, but let's stick to a more conservative estimate of 1E9.

Case study*

- Assume that we use hash table and it needs at least 32 bits per distinct visitor id (key)*
 - Looking now for $15 * 4 \text{ GB} \approx 60 \text{ GB}$ of memory
- There exists specialized implementations of hash tables, e.g. sparse hash** using 1-2 bits per key
 - Looking now for $15 * 1.25 \text{ GB} \approx 19 \text{ GB}$ of memory
- Still a bit excessive...
- Can we do better?

* Some estimates were done for Java: <http://code-o-matic.blogspot.ca/2012/02/updated-memory-cost-per-javaguava.html> and the estimate is VERY conservative

** <https://github.com/sparsehash/sparsehash>

Other applications

- Detection of network attacks
 - By tracking the number of distinct network packets in a given time windows
- Mining massive datasets
 - natural language text
 - biological data
 - structured databases (e.g. internet graph)
 - etc.
- See [1] for review

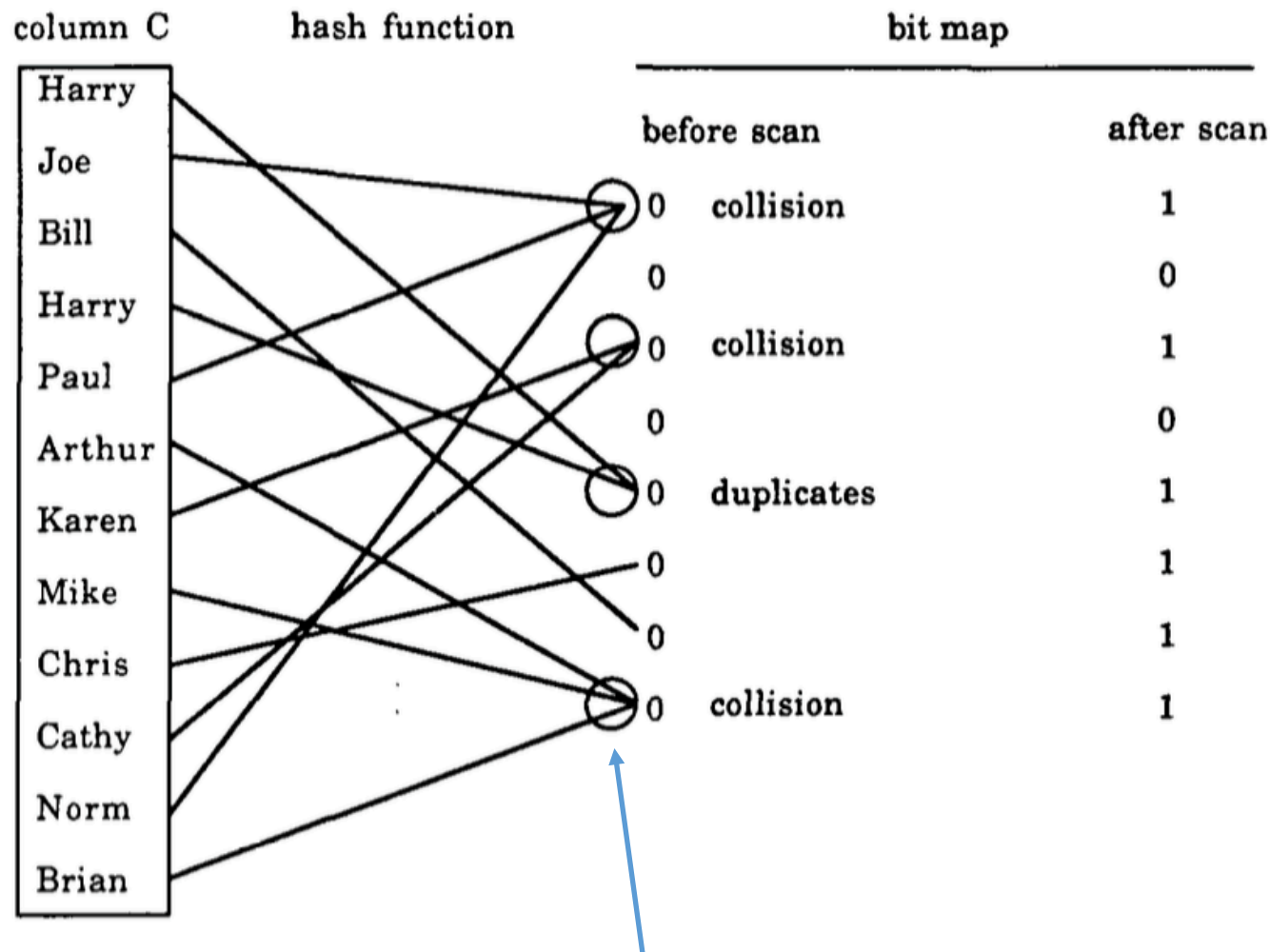
Intermezzo: Hash functions

- The following methods rely on a hash function that maps an elements to a reasonably long binary string, such that the bits in this string are random uniform and independent bits
- “It is theoretically impossible to define a hash function that creates random data from non-random data in actual files. But in practice it is not difficult to produce a pretty good imitation of random data” D. Knuth
- However, this does not imply that every implementation of a hash function available is a good one!
- Corollary: don’t grab the 1st possible implementation of a hash function that you find!
 - <http://spyced.blogspot.ca/2009/01/all-you-ever-wanted-to-know-about.html>

Linear Counting

The text, equation and figures are taken from [2]

Linear Counting Algorithm



Multiple values can map to the same bit or "urn"

Linear Counting Algorithm

Algorithm Basic Linear Counting:

Let key_i = the key for the i th tuple in the relation.

Initialize the bit map to “0”s.

for $i = 1$ to q do

$\text{hash_value} = \text{hash}(\text{key}_i)$

$\text{bit map}(\text{hash_value}) = \text{“1”}$

end for

U_n = number of “0”s in the bit map

$V_n = U_n/m$

$\hat{n} = -m \ln V_n$

Approximation of the cardinality. But where did this formula come from?

Why linear?

- The algorithm has $O(q)$ time complexity, where q is the number of elements
- Let's analyze space requirements...

Linear Counting Notation

Table I. Terminology and Notation

Urn problem	Linear counting
n : Number of distinct colors	True count of unique values in the column (column cardinality)
q : Number of balls	Total number of tuples in the relation or total number of values in the column including duplicates (relation cardinality)
m : Number of urns	Number of bits in the bit map
t : n/m	n/m = load factor
U_n : Random variable denoting the number of empty urns	Random variable denoting the number of “0” bits after the test
V_n : U_n/m -random variable denoting the fraction of empty urns	Random variable denoting the fraction of “0” bits after the test
p : mean of V_n	
$\hat{n} = -m \ln V_n$: maximum likelihood estimator of n (see Appendix B)	

Linear Counting

Derivation of the Estimator of n

Let A_j be the event that urn j is empty and let 1_{A_j} be the corresponding indicator random variable. Since the assignment of the balls is independent,

$$P(A_j) = \left(1 - \frac{1}{m}\right)^n$$

Since U_n is the number of empty urns,

$$U_n = \sum_{j=1}^m 1_{A_j}.$$

Linear Counting

Derivation of the Estimator of n

Thus,

$$E(U_n) = \sum_{j=1}^m P(A_j) = m \left(1 - \frac{1}{m}\right)^n \cong me^{-n/m} = me^{-t}, \quad \text{as } n, m \rightarrow \infty$$

Since $V_n = U_n/m$,

$$E(V_n) = e^{-t}, \quad \text{as } m, n \rightarrow \infty,$$

Linear Counting

Derivation of the Estimator of n

From Eq. (5), $E(V_n) = e^{-n/m}$, as $m, n \rightarrow \infty$. Replacing $E(V_n)$ and n by their representations in terms of observed variables, V_n and \hat{n} , we obtain

$$\hat{n} = -m \ln V_n, \quad (7)$$

where \hat{n} is our estimator for n . Theorem A4 in Appendix B shows that \hat{n} is the maximum likelihood estimator for n .



The authors show that V_n is normally distributed (as $n, m \rightarrow \infty$) and leverage this fact.

Linear Counting

Derivation of the Bias

We derive the bias for the ratio \hat{n}/n as follows. First, we expand the right hand side of Eq. (7) by its Taylor series about $p = E(V_n) = e^{-t}$, the mean of V_n . Denoting $-\ln(V_n)$ as $f(V_n)$,

$$\begin{aligned}\hat{n} &= m \times f(V_n) \\ &= m(f(p) + (V_n - p)f'(p) + \frac{1}{2}(V_n - p)^2f''(p) \\ &\quad + \frac{1}{6}(V_n - p)^3f'''(p) + \frac{1}{24}(V_n - p)^4f''''(p) \dots) \quad (8) \\ &= m\left(t - \frac{V_n - p}{p} + \frac{\frac{1}{2}(V_n - p)^2}{p^2} - \frac{\frac{1}{3}(V_n - p)^3}{p^3} + \frac{\frac{1}{4}(V_n - p)^4}{p^4} \dots\right).\end{aligned}$$

Linear Counting


Derivation of the Bias

We truncate Eq. (8) after the third term since the expected value of the second term is 0, and the third term is the first nonzero bias term. We shall discuss the error caused by this truncation in Appendix C. We then obtain the expected value of \hat{n} as follows:

$$E(\hat{n}) = mt + \frac{m}{2p^2} E(V_n - p)^2.$$

Since, from Eq. (6), $E(V_n - p)^2 = \text{Var}(V_n) = (1/m)e^{-t}(1 - (1 + t)e^{-t})$, we obtain

$$E(\hat{n}) = n + \frac{e^t - t - 1}{2}. \quad (9)$$


$$\text{Var}(V_n) = \frac{1}{m} e^{-t}(1 - (1 + t)e^{-t}), \quad \text{as } m, n \rightarrow \infty. \quad (6)$$

Linear Counting Bias vs n and t

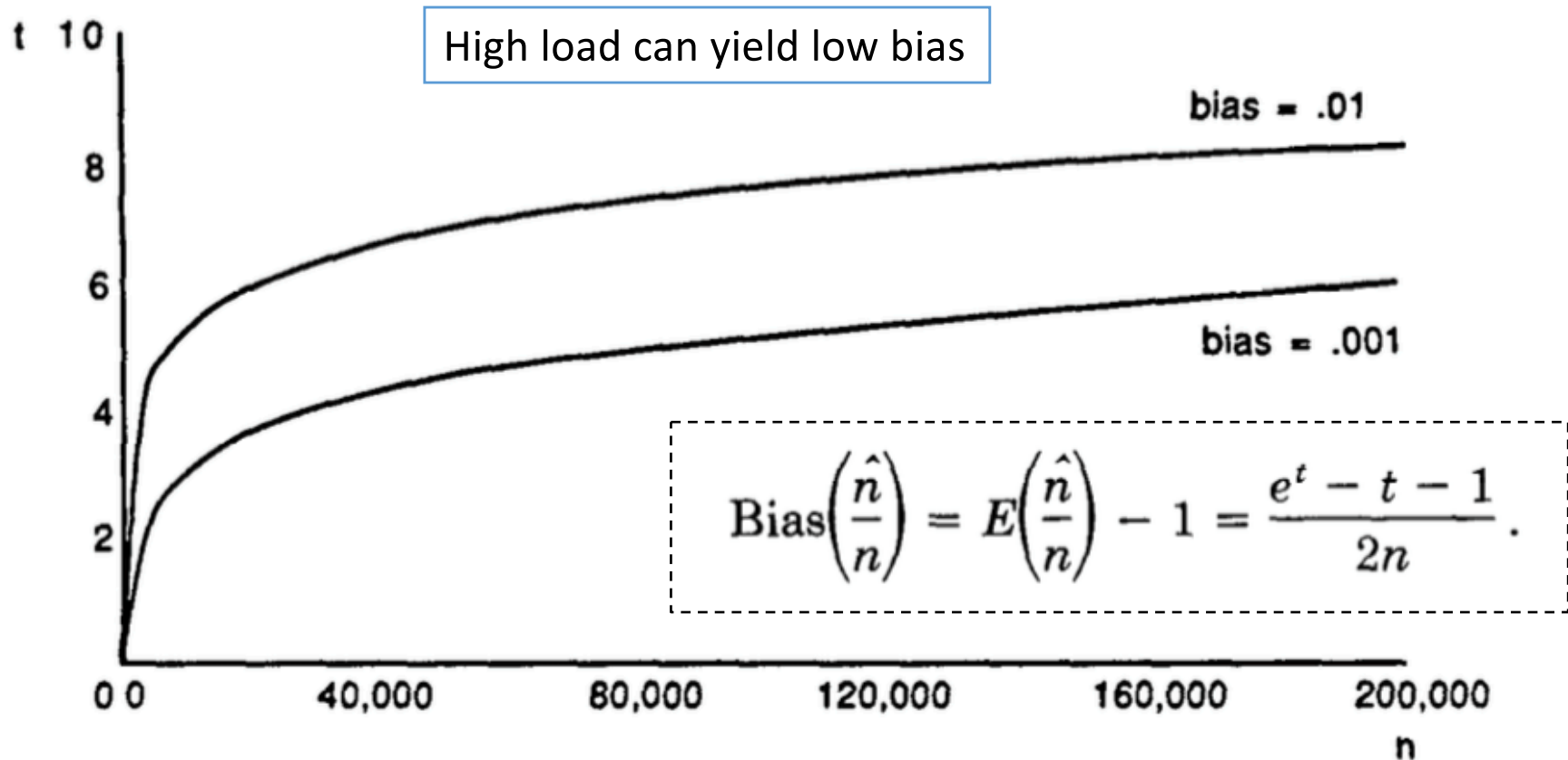


Fig. 3. Load factor that must be selected to achieve a specific bias.

Linear Counting

S.d. of \hat{n}/n

High load can yield low s.d. of the bias

The authors call s.d. standard error

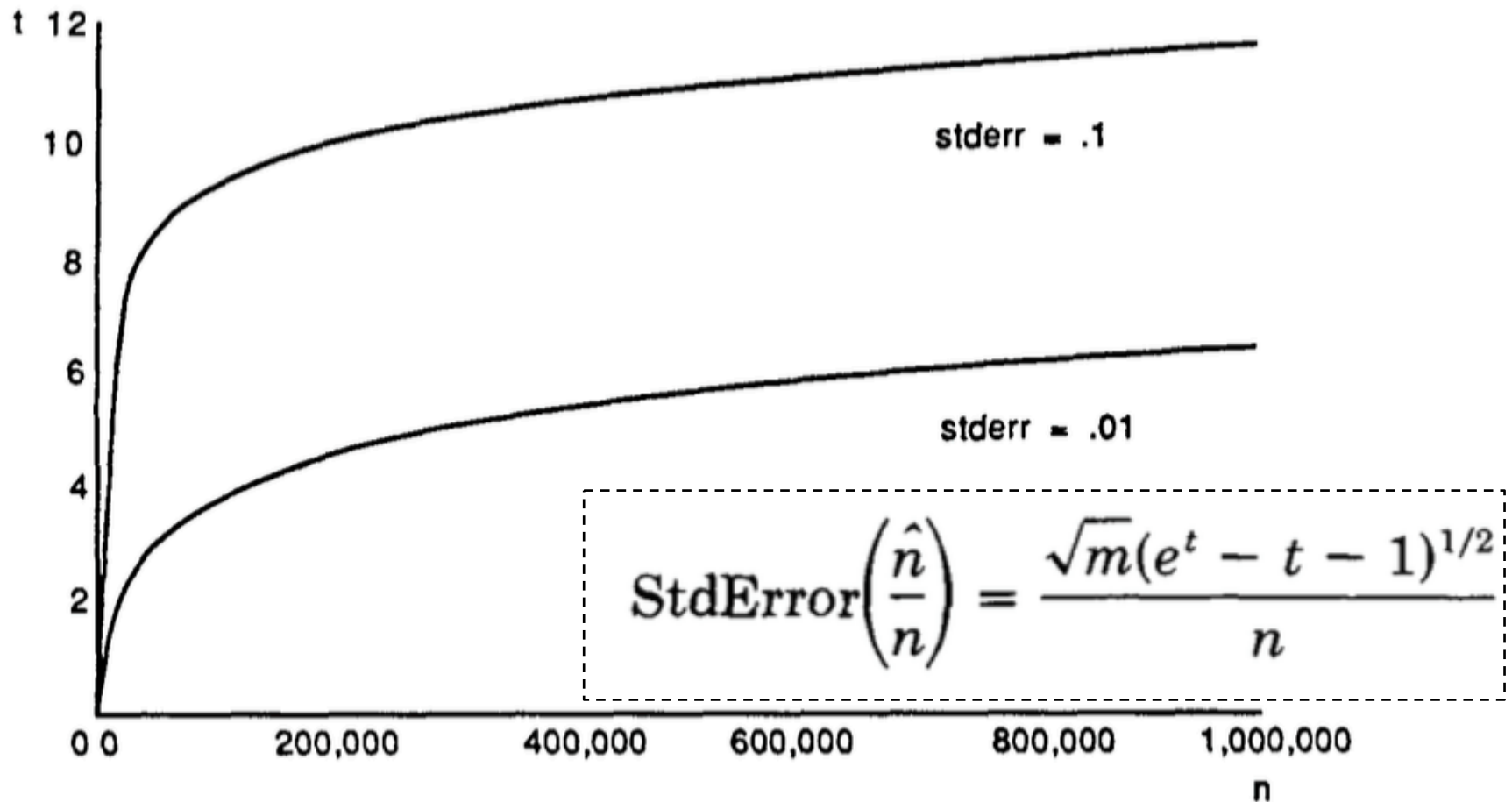


Fig. 4. Load factor that must be selected to achieve a specified standard error.

Linear Counting Selection of m

As we discuss in Section 4.2, $a = \sqrt{5}$ limits the probability of the map becoming full to 0.7 percent. From Eq. (14) and Eq. (16), we obtain

$$m > \beta(e^t - t - 1), \quad (17)$$

where $\beta = \max(a^2, 1/(\epsilon t)^2)$.

s.d.



Cannot solve (17) analytically as t depends on m

What happens if the map fills up, i.e. all bits are set to 1?

Linear Counting

Selection of m – case study

- Going back to our case study, we have $1\text{E}9$ records, i.e., $n = 1\text{E}9$
- Setting s.d. to 0.01 and solving the inequality numerically:
 - $m > 7.06\text{E}7$ bits ≈ 8.4 MB
 - Compare with 19 GB for sparse hash
 - $t = n/m \approx 1\text{E}9 / 7.06\text{E}7 \approx 14$

Linear Counting

Experimental Results

Table VI. Map Size = 100000

Load factor	Experimental $E[\hat{n}/n]$	Theoretical $E[\hat{n}/n]$	Experimental standard error	Theoretical standard error
0.25	0.999977	1.000001	0.002096	0.002333
0.50	0.999874	1.000001	0.002256	0.002439
0.75	1.000235	1.000002	0.002894	0.002554
1	0.999992	1.000004	0.002726	0.002680
2	1.000241	1.000011	0.003658	0.003312
3	0.999878	1.000027	0.004317	0.004228
4	0.999918	1.000062	0.005681	0.005568
5	1.000011	1.000142	0.007748	0.007548
6	1.001603	1.000330	0.010973	0.010494
7	1.000730	1.000778	0.014368	0.014905
8	1.000408	1.001857	0.023627	0.021549
9	1.010030	1.004496	0.035842	0.031609
10	1.011620	1.011008	0.051350	0.046921

Why experimental data differs from theoretical data?

Theory: approximations (e.g. Taylor series); Practice: imprecise hash functions, etc.

Linear Counting

Which tools are used?

- Probability theory
- Combinatorics
- Taylor series

Modern count-distinct algorithms

A family of LogLog Counters

- LogLog Counter (1984-2003) [3]
 - $O(\log \log n)$ bits of storage needed
- HyperLogLog Counter (2007) [1]
 - “estimate cardinalities well beyond 10^9 with a typical accuracy of 2% while using a memory of only 1.5 kilobytes”
- HyperLogLog++ Counter(2013) [5]
 - further improvement of accuracy and space requirements of HyperLogLog
- You can find Java implementations of this and other algorithms at <https://github.com/addthis/stream-lib/tree/master/src/main/java/com/clearspring/analytics/stream/cardinality>

Idea behind LogLog Counter

- Every element is converted to a binary string
- Probabilistically
 - $n/2$ of the strings will start with 0
 - $n/4$ will start with 01
 - $n/8$ will start with 001
 - ...
 - $n/2^r$ will start with $00\dots01$
 - Very few records (potentially 1) will be here
- Thus, the value of r can tell us something about cardinality. But it will be a very unstable one. The line of papers on the previous slide used a number of techniques to “stabilize” the estimate.

LogLog Counter Algorithm

```
Algorithm LOGLOG( $\mathfrak{M}$ : Multiset of hashed values;  $m \equiv 2^k$ )  
Initialise  $M^{(1)}, \dots, M^{(m)}$  to 0;  
let  $\rho(y)$  be the rank of first 1-bit from the left in  $y$ ;  
  for  $x = b_1 b_2 \dots \in \mathfrak{M}$  do  
    set  $j := \langle b_1 \dots b_k \rangle_2$  (value of first  $k$  bits in base 2)  
    set  $M^{(j)} := \max(M^{(j)}, \rho(b_{k+1} b_{k+2} \dots))$ ;  
return  $E := \alpha_m m 2^{\frac{1}{m}} \sum_j M^{(j)}$  as cardinality estimate.
```

FIGURE 2. The principle of the basic LOGLOG algorithm.

Taken from [3]

α – constant

m – number of “buckets”

M – function of the “rank”

HyperLogLog Counter Algorithm

Note the growth in complexity of the estimates as the authors tighten the approximation

Let $h : \mathcal{D} \rightarrow [0, 1] \equiv \{0, 1\}^\infty$ hash data from domain \mathcal{D} to the binary domain.
 Let $\rho(s)$, for $s \in \{0, 1\}^\infty$, be the position of the leftmost 1-bit ($\rho(0001 \dots) = 4$).
Algorithm HYPERLOGLOG (**input** \mathcal{M} : multiset of items from domain \mathcal{D}).
assume $m = 2^b$ with $b \in \mathbb{Z}_{>0}$;
initialize a collection of m registers, $M[1], \dots, M[m]$, to $-\infty$;
for $v \in \mathcal{M}$ **do**
 set $x := h(v)$;
 set $j = 1 + \langle x_1 x_2 \dots x_b \rangle_2$; {the binary address determined by the first b bits of x }
 set $w := x_{b+1} x_{b+2} \dots$; **set** $M[j] := \max(M[j], \rho(w))$;
compute $Z := \left(\sum_{j=1}^m 2^{-M[j]} \right)^{-1}$; {the “indicator” function}
return $E := \alpha_m m^2 Z$ with α_m as given by Equation (3).

$$E := \frac{\alpha_m m^2}{\sum_{j=1}^m 2^{-M^{(j)}}}, \quad \text{with} \quad \alpha_m := \left(m \int_0^\infty \left(\log_2 \left(\frac{2+u}{1+u} \right) \right)^m du \right)^{-1}. \quad (3)$$

Is the element in the set?

Bloom Filter

General idea

Is the element in the set?

- If the answer is “no”, then we are 100% certain that it is not there
- If the answer is “yes”, then we are not 100% certain that the element is present in the set
 - The real answer is “maybe”

Why do you need it?

Saves a lot of I/O operations

- Google BigTable, Apache Hbase, and Apache Cassandra database identifies non-existing rows and columns, avoiding disk lookup
 - Can also be used in data joins (Bloom join)
- Google Chrome web browser used* Bloom filter to detect malicious URLs. A URL was checked against a local Bloom filter. If it returned positive result, full checkup was performed
- Medium uses Bloom filters to avoid recommending articles a user has previously read.
- Check https://en.wikipedia.org/wiki/Bloom_filter#Extensions_and_applications for other examples

High-level overview of the algorithm

Create bit array A of length m and populate it with 0s

`add_element(element):`

`// Apply k different hash functions to an element`

`// Each hash function returns a value between 1 and m`

`for $i = 1$ to k :`

`$A[\text{hash}_i(\text{element})] = 1$`

`is_element_present(element):`

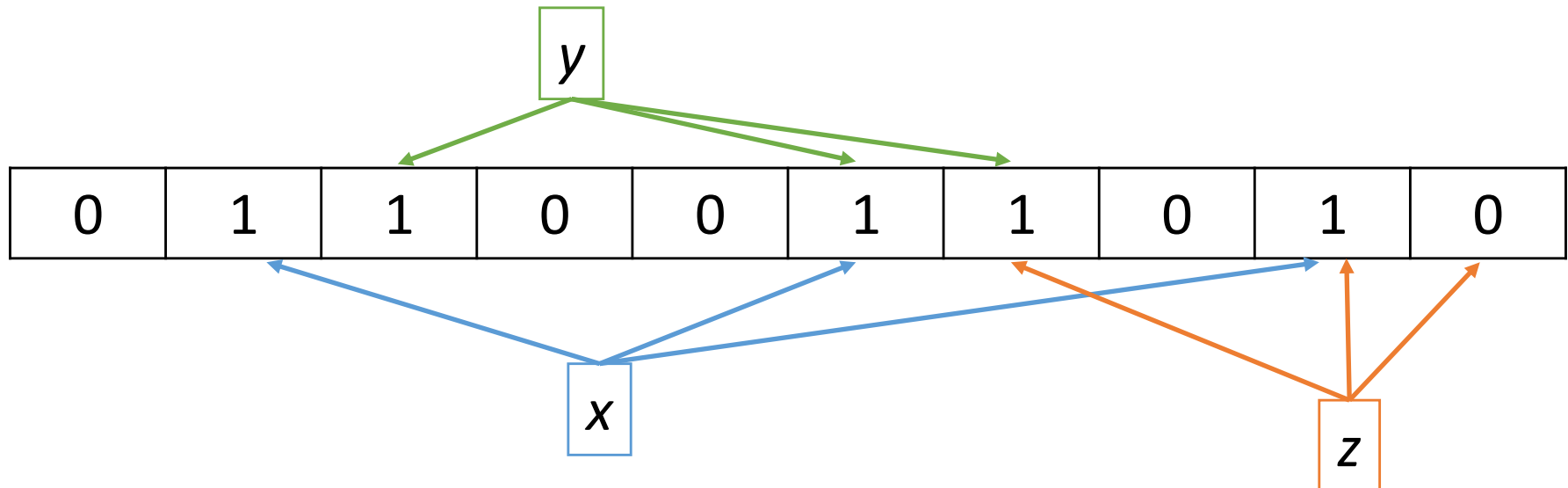
`//if at least one element in A is 0 - the element is absent`

`for $i = 1$ to k :`

`if($A[\text{hash}_i(\text{element})] == 0$): return false`

`return true`

High-level overview of the algorithm



- Number of hash functions $k = 3$
- Length of bit array $m = 10$
- x and y are probably present
- z is definitely absent

Probability of false positives

- Assume independence of the probabilities of each bit being set
- Probability that a given bit is not set to 1 by a hash function is
 - $(1 - 1/m)$
- Probability that it is not set to 1 by any hash function is
 - $(1 - 1/m)^k$
- After inserting n elements, probability that it is still zero is
 - $(1 - 1/m)^{kn}$
- Thus, probability that it is 1 is
 - $1 - (1 - 1/m)^{kn}$

Probability of false positives

- Thus, probability that it is 1 is
 - $1 - (1 - 1/m)^{kn}$
- Let's check for false positive, i.e., element not present in the set, but all the bits are set to 1. The probability of occurrence of such event is:
 - $[1 - (1 - 1/m)^{kn}]^k \approx [1 - \exp(-kn/m)]^k$
- This formula is valid only if assumption of independence holds. However, you can arrive to the same formula even if you relax this assumption [11].

Optimal values of k and m

- Probability of false positives $p = [1 - \exp(-kn/m)]^k$ is minimized when $k = \ln(2) m/n$
 - How do you think this result is obtained?
- Substitution of k into $[1 - \exp(-kn/m)]^k$ yields upon simplification
 - $\ln(p) = -m/n (\ln(2))^2$
- Thus,
 - $m = -n \ln(p) / (\ln(2))^2$

Frequency Estimation

Count-Min Sketch [6]

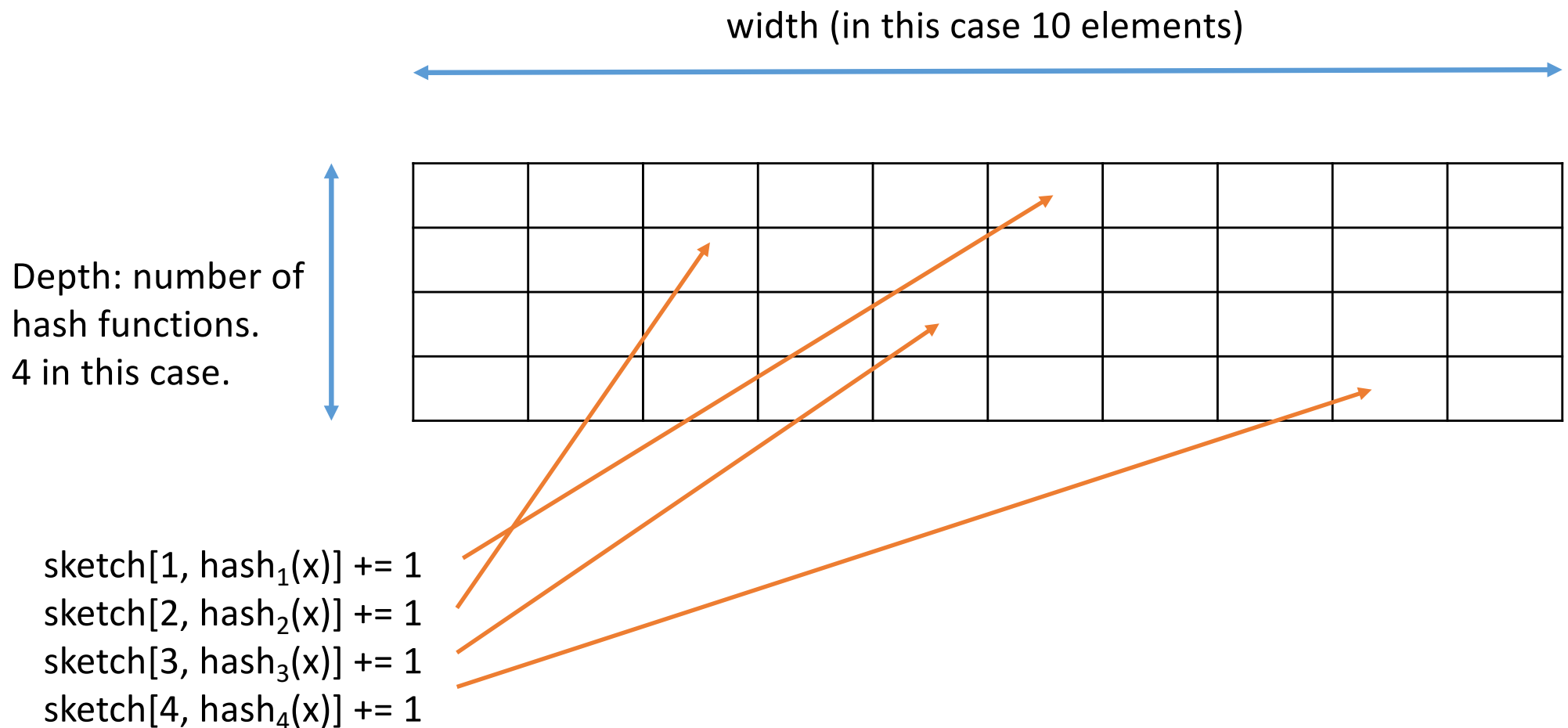
Count Min Sketch

- Find how many times each element occurs in the set
- Let's reuse the idea behind Bloom's filter, but now our array cell will store an integers rather than bit
- Sketch == Summary

Problem

- Get the frequency of elements [6]
 - Count of frequent events should be accurate
 - Count of infrequent event can be inaccurate

Count Min Sketch [6]



High-level overview of the algorithm

Create d by w matrix A and populate it with 0s

`add_element(element):`

`// Apply k different hash functions to an element`

`// Each hash function returns a value between 1 and w`

`for $i = 1$ to d :`

`$A[i, \text{hash}_i(\text{element})] = A[i, \text{hash}_i(\text{element})] + 1$`

`get_element_frequency(element):`

`//return the smallest value`

`cnts = array of length d`

`for $i = 1$ to d :`

`$\text{cnts}[i] = A[i, \text{hash}_i(\text{element})]$`

`return min(cnts)`

Why return min?

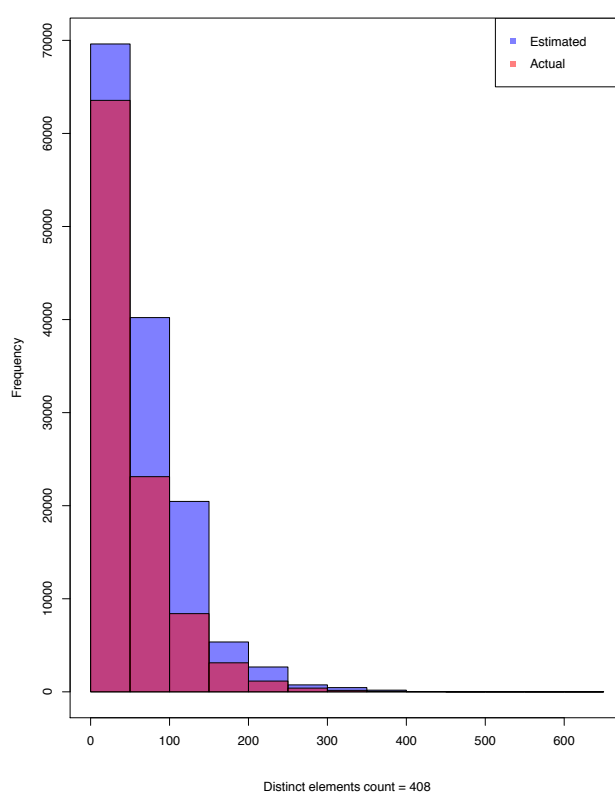
Optimal values of w and d

- “... accuracy guarantees will be made in terms of a pair of user specified parameters, ϵ and δ , meaning that the error in answering a query is within a factor of ϵ with probability δ . ” [6]
- From [6]*
 - $w = e / \epsilon$
 - $d = \ln(1 / \delta)$
- A more thorough analysis in the case of Zipf distribution is given in [7].

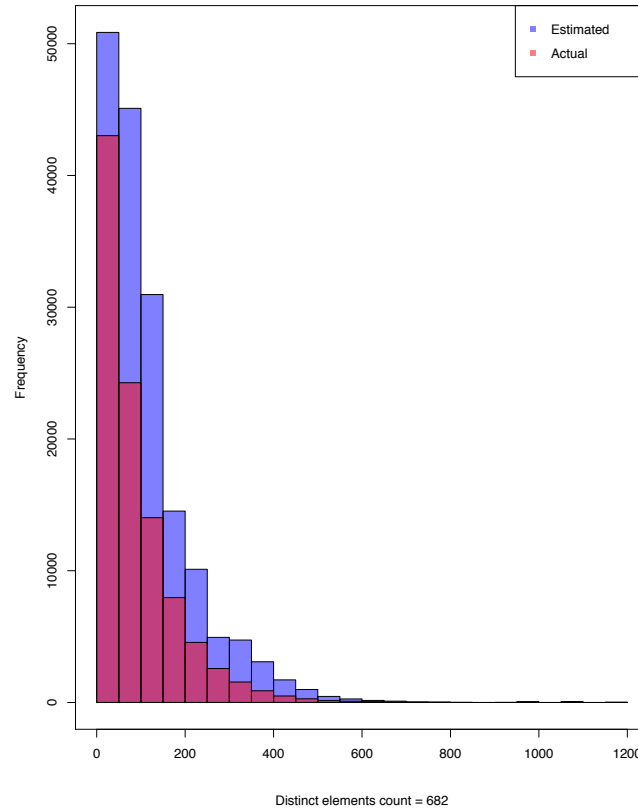
* I am not 100% certain if I interpreted the formulas on p.15 of [6] correctly...

Sample runs

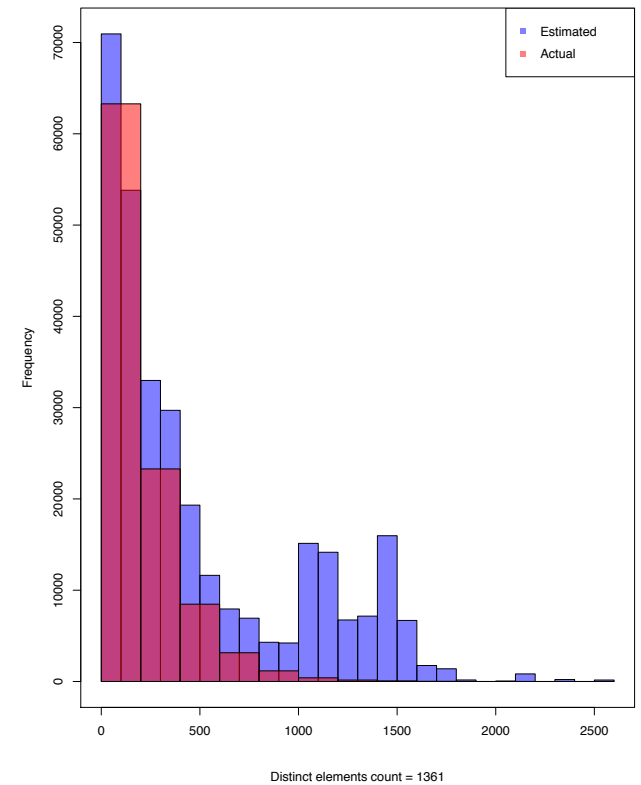
The code is available in `count_min.R` on D2L – please familiarize yourself with it



Distinct Count = 408



Distinct Count = 682



Distinct Count = 1361

Number of samples = 100,000; sketch width = 200, depth = 3

Extensions - Accuracy

- As you can see from sample runs – results are not spectacular.
- There are extensions trying to reduce error, e.g., correction algorithm trying to estimate “noise” of hash functions and “adjust” the values was suggested in [8].
 - Pre-print only

Extensions -- Applications

- Keep track of most frequent elements (top-K) [6, 9]
- Estimate quantiles [6]

Additional Probabilistic Algorithms/Structures

- Quantile estimators Q-digest[10]
 - <https://github.com/addthis/stream-lib/blob/master/src/main/java/com/clearspring/analytics/stream/quantile/QDigest.java>
 - Suggested to be used in sensor network
- Sketch distribution
 - a mix of k-means and q-digest
 - provides part per million accuracy for extreme quantiles and typically <1000 ppm accuracy for middle quantiles
 - Use if you need a meaningful distribution approximation and basic random sampling does not cut it
 - <https://github.com/addthis/stream-lib/blob/master/src/main/java/com/clearspring/analytics/stream/quantile/TDigest.java>

References

1. Flajolet P, Fusy É, Gandouet O, Meunier F. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. DMTCS Proceedings. 2008 Jan 20(1), 127-146.
2. Whang KY, Vander-Zanden BT, Taylor HM. A linear-time probabilistic counting algorithm for database applications. ACM Transactions on Database Systems (TODS). 1990 Jun 1;15(2):208-229.
3. Durand M, Flajolet P. Loglog counting of large cardinalities. In European Symposium on Algorithms 2003 Sep 16 (pp. 605-617). Springer Berlin Heidelberg.
4. Flajolet P, Fusy É, Gandouet O, Meunier F. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. DMTCS Proceedings. 2007, 127-146.
5. Heule S, Nunkesser M, Hall A. HyperLogLog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In Proceedings of the 16th International Conference on Extending Database Technology 2013 Mar 18 (pp. 683-692). ACM.
6. Cormode G, Muthukrishnan S. An improved data stream summary: the count-min sketch and its applications. Journal of Algorithms. 2005 Apr 30;55(1):58-75.
7. Cormode G, Muthukrishnan S. Summarizing and Mining Skewed Data Streams, (pre-print) <http://www.cs.rutgers.edu/~muthu/cmz-sdm.pdf>
8. Deng F, Rafiei D. New estimation algorithms for streaming data: Count-min can do more, (pre-print) <http://webdocs.cs.ualberta.ca/~fandeng/paper/cmm.pdf>
9. Metwally A, Agrawal D, El Abbadi A. Efficient computation of frequent and top-k elements in data streams. In International Conference on Database Theory 2005 Jan 5 (pp. 398-412). Springer Berlin Heidelberg.
10. Shrivastava N, Buragohain C, Agrawal D, Suri S. Medians and beyond: new aggregation techniques for sensor networks. In Proceedings of the 2nd international conference on Embedded networked sensor systems 2004 Nov 3 (pp. 239-249). ACM.
11. Mitzenmacher M, Upfal E. Probability and computing: Randomized algorithms and probabilistic analysis. Cambridge University Press; 2005 Jan 31.

Summary

- Randomized Algorithms
 - Probabilistic Algorithms and Data structures
 - Estimators of
 - Cardinality (Count-distinct)
 - Linear Counting and LogLog family
 - Presence of an element
 - Bloom filter
 - Frequency estimation
 - Count-min Sketch

References

1. Flajolet P, Fusy É, Gandouet O, Meunier F. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. DMTCS Proceedings. 2008 Jan 20(1), 127-146. <http://algo.inria.fr/flajolet/Publications/FIFuGaMe07.pdf>
2. Whang KY, Vander-Zanden BT, Taylor HM. A linear-time probabilistic counting algorithm for database applications. ACM Transactions on Database Systems (TODS). 1990 Jun 1;15(2):208-229. http://dblab.kaist.ac.kr/Publication/pdf/ACM90_TODS_v15n2.pdf
3. Durand M, Flajolet P. Loglog counting of large cardinalities. In European Symposium on Algorithms 2003 Sep 16 (pp. 605-617). Springer Berlin Heidelberg. <http://algo.inria.fr/flajolet/Publications/DuFl03.pdf>
4. Flajolet P, Fusy É, Gandouet O, Meunier F. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. DMTCS Proceedings. 2007, 127-146. <http://algo.inria.fr/flajolet/Publications/FIFuGaMe07.pdf>
5. Heule S, Nunkesser M, Hall A. HyperLogLog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In Proceedings of the 16th International Conference on Extending Database Technology 2013 Mar 18 (pp. 683-692). ACM. <http://research.google.com/pubs/pub40671.html>
6. Cormode G, Muthukrishnan S. An improved data stream summary: the count-min sketch and its applications. Journal of Algorithms. 2005 Apr 30;55(1):58-75. <https://web.archive.org/web/20060907232042/http://www.eecs.harvard.edu/~michaelm/CS222/countmin.pdf>
7. Cormode G, Muthukrishnan S. Summarizing and Mining Skewed Data Streams, (pre-print) <http://www.cs.rutgers.edu/~muthu/cmz-sdm.pdf>
8. Deng F, Rafiei D. New estimation algorithms for streaming data: Count-min can do more, (pre-print) <http://webdocs.cs.ualberta.ca/~fandeng/paper/cmm.pdf>
9. Metwally A, Agrawal D, El Abbadi A. Efficient computation of frequent and top-k elements in data streams. In International Conference on Database Theory 2005 Jan 5 (pp. 398-412). Springer Berlin Heidelberg. http://link.springer.com/chapter/10.1007/978-3-540-30570-5_27
10. Shrivastava N, Buragohain C, Agrawal D, Suri S. Medians and beyond: new aggregation techniques for sensor networks. In Proceedings of the 2nd international conference on Embedded networked sensor systems 2004 Nov 3 (pp. 239-249). ACM. <http://www.cs.virginia.edu/~son/cs851/papers/ucsb.sensys04.pdf>
11. Mitzenmacher M, Upfal E. Probability and computing: Randomized algorithms and probabilistic analysis. Cambridge University Press; 2005 Jan 31.

Notes and Backup

LogLog Counter Algorithm

- Assume that we have a good source of randomness that provides uniformly-distributed output
- Given a sequence of numbers x ,
- , a sequence of zero bits of length k will appear every 2^k times in a sequence of

Hash Table

- Note that Worst case is linear, and it is controlled by Load fact
 - https://en.wikipedia.org/wiki/Hash_table

Time complexity in big O notation		
	Average	Worst case
Space	$O(n)^{[1]}$	$O(n)$
Search	$O(1)$	$O(n)$
Insert	$O(1)$	$O(n)$
Delete	$O(1)$	$O(n)$

- In the simplest model, the hash function is completely unspecified and the table does not resize. For the best possible choice of hash function, a table of size k with open addressing has no collisions and holds up to k elements, with a single comparison for successful lookup, and a table of size k with chaining and n keys has the minimum $\max(0, n - k)$ collisions and $O(1 + n/k)$ comparisons for lookup. For the worst choice of hash function, every insertion causes a collision, and hash tables degenerate to linear search, with $\Omega(n)$ amortized comparisons per insertion and up to n comparisons for a successful lookup.
- Adding rehashing to this model is straightforward. As in a dynamic array, geometric resizing by a factor of b implies that only n/b^i keys are inserted i or more times, so that the total number of insertions is bounded above by $bn/(b - 1)$, which is $O(n)$. By using rehashing to maintain $n < k$, tables using both chaining and open addressing can have unlimited elements and perform successful lookup in a single comparison for the best choice of hash function.
- In more realistic models, the hash function is a random variable over a probability distribution of hash functions, and performance is computed on average over the choice of hash function. When this distribution is uniform, the assumption is called "simple uniform hashing" and it can be shown that hashing with chaining requires $\Theta(1 + n/k)$ comparisons on average for an unsuccessful lookup, and hashing with open addressing requires $\Theta(1/(1 - n/k))$.^[22] Both these bounds are constant, if we maintain $n/k < c$ using table resizing, where c is a fixed constant less than 1.

- Even with million buckets
- Hash collisions are practically unavoidable when hashing a random subset of a large set of possible keys. For example, if 2,450 keys are hashed into a million buckets, even with a perfectly uniform random distribution, according to the birthday problem there is approximately a 95% chance of at least two of the keys being hashed to the same slot.

- What if we get very large?
- Try sparse hash:
<https://github.com/sparsehash/sparsehash>
- And if that fails?
 - Distributed key-value store!
 - Add Hash paper
 - From practical perspective – use one of NoSQL databases
- And if you do not want to spend too much resources?
 - Probabilistic data structures!
 - Bloom filter is one of the more interesting ones, but let's start with simpler ones

Other

- Trie (a.k.a. radix tree , digital tree, prefix tree)
 - <https://en.wikipedia.org/wiki/Trie>
 - Keys are strings, useful for, e.g., spellcheckers
- Judy Array
 - Highly optimized trie
 - Complex code, difficulties with porting
 - <http://judy.sourceforge.net>

Load Factor

- If you know what you are doing, you can set it manually.
 - Say, for Java default initial capacity is 16 and default load factor 0.75
 - <http://docs.oracle.com/javase/6/docs/api/java/util/HashMap.html>
 - <http://stackoverflow.com/questions/10901752/what-is-the-significance-of-load-factor-in-hashmap>
 - <https://www.quora.com/Whats-the-purpose-of-load-factor-in-hash-tables>

- Lab -- try to implement probabilistic structure
 - see if existing labs available
 - <https://github.com/addthis/stream-lib>

Outline

- Probabilistic Algorithms
- randomized algo
 - https://en.wikipedia.org/wiki/Randomized_algorithm
- search
 - random
 - MC?
- probabilistic turing machine
 - https://en.wikipedia.org/wiki/Probabilistic_Turing_machine
- random sampling – estimate distr
- Cardinality

Outline

- Sampling data
 - Not good for outliers
- Bitmaps
 - <http://blog.getspool.com/2011/11/29/fast-easy-realtime-metrics-using-redis-bitmaps/>
- Cardinality Estimators
 - <https://highlyscalable.wordpress.com/2012/05/01/probabilistic-structures-web-analytics-data-mining/>
 - Linear Counter
 - <http://highscalability.com/blog/2012/4/5/big-data-counting-how-to-count-a-billion-distinct-objects-us.html>
 - LogLog
 - <https://highlyscalable.wordpress.com/2012/05/01/probabilistic-structures-web-analytics-data-mining/>
 - <http://algo.inria.fr/flajolet/Publications/DuFI03-LNCS.pdf>
 - HyperLogLog
 - <http://highscalability.com/blog/2012/4/5/big-data-counting-how-to-count-a-billion-distinct-objects-us.html>
 - <http://algo.inria.fr/flajolet/Publications/FIFuGaMe07.pdf>
 - Count-min
 - Bloom filter