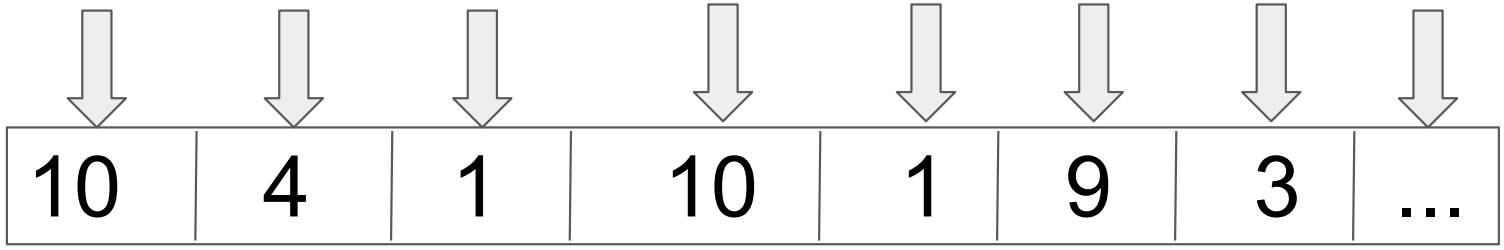
CS365
Foundations of Data Science

Charalampos E. Tsourakakis
ctsourak@bu.edu

The Streaming model



Input

-Stream of elements $\langle x_1, \ldots, x_m \rangle$, $x_i \in [n]$

- Order may be adversarial

- One pass over the stream

$n = 2^{c4}$

$[n] = \{1, \ldots, n\}$ or $\{0, \ldots, n-1\}$

k-server problem

see the data only once

Goals

- Compute as accurately as possible statistics of interest
    - E.g., Number of distinct elements appear in the stream, length of the stream, heavy hitters etc.
- Key constraint: space
- Ideally, also fast query and update times

Distinct elements

- A stream $\langle x_1, \ldots, x_m \rangle$, $x_i \in [n]$ is received, one element from the universe [n] at a time.
- **Number of distinct elements** is the number of items in the universe that appear at least once in the stream, i.e., $|\{i : f_i > 0,\ i \in [n]\}|$ where $f_i = |\{j : x_j = i,\ j \in [m]\}|$
- We wish to maintain a small *sketch S,* whose size is independent of m, so we can return an approximate value $\tilde{F_0}$ to the true value F_0 of distinct elements.
- There exist two broad families of algorithms for estimating F_0 in terms of the different types of guarantees they provide.
- Today: *insert - only* streams

Elements only arrives

Why not compute the distinct elements as follows?

>> sort -u filename | wc -l

```
babis: 44Wed Feb 15~/Desktop$
sort hyperloglog.hpp | wc -l
     374
babis: 45Wed Feb 15~/Desktop$
sort -u hyperloglog.hpp | wc -l
     184
babis: 46Wed Feb 15~/Desktop$
```

Distinct elements in practice

```
character (-z option).  A record can contain any printable or unprintable characters.  Comparisons are based on one or more sort keys extracted from
each line of input, and are performed lexicographically, according to the current locale's collating rules and the specified command-line options
that can tune the actual sorting behavior.  By default, if keys are not given, sort uses entire lines for comparison.

The command line options are as follows:

-c, --check, -C, --check=silent|quiet
        Check that the single input file is sorted.  If the file is not sorted, sort produces the appropriate error messages and exits with code 1,
        otherwise returns 0.  If -C or --check=silent is specified, sort produces no output.  This is a "silent" version of -c.

-m, --merge
        Merge only.  The input files are assumed to be pre-sorted.  If they are not sorted the output order is undefined.

-o output, --output=output
        Print the output to the output file instead of the standard output.

-S size, --buffer-size=size
        Use size for the maximum size of the memory buffer.  Size modifiers %,b,K,M,G,T,P,E,Z,Y can be used.  If a memory limit is not explicitly
        specified, sort takes up to about 90% of available memory.  If the file size is too big to fit into the memory buffer, the temporary disk
        files are used to perform the sorting.

-T dir, --temporary-directory=dir
        Store temporary files in the directory dir.  The default path is the value of the environment variable TMPDIR or /var/tmp if TMPDIR is not
        defined.

-u, --unique
        Unique keys.  Suppress all lines that have a key that is equal to an already processed one.  This option, similarly to -s, implies a stable
        sort.  If used with -c or -C, sort also checks that there are no lines with duplicate keys.

-s
        Stable sort.  This option maintains the original record order of records that have an equal key.  This is a non-standard feature, but it is
        widely accepted and used.

--version
        Print the version and silently exits.
```

Key constraints:

1) Limited space
2) One pass over the data/stream
3) Few operations per value (update time)
4) Accurate as possible
5) Fast query time (typically at the end of the stream, but at-all-times variations exist)

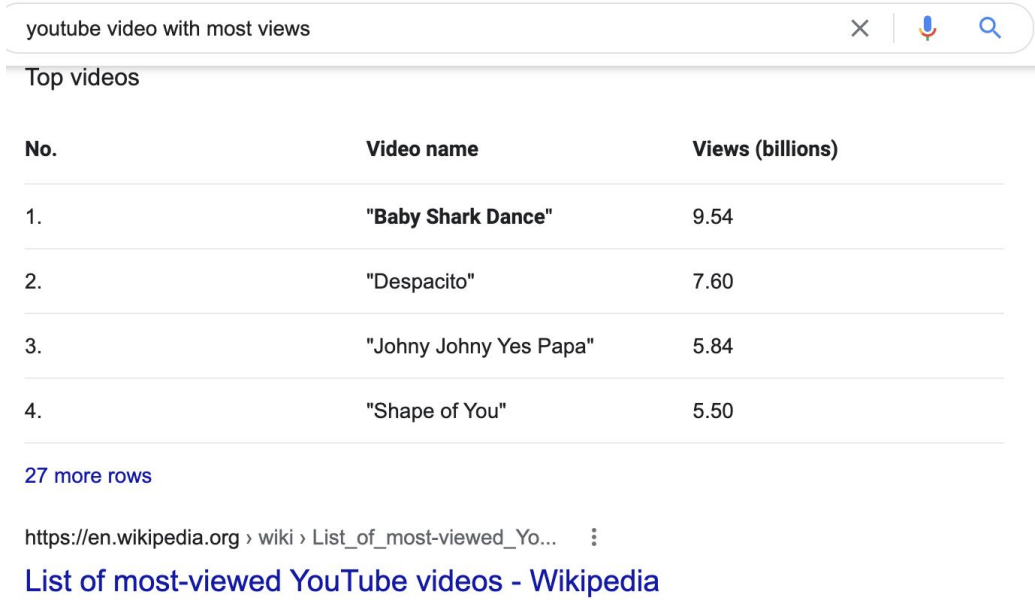
Realistic setting

Suppose we wish to count the number of distinct users who viewed “Baby Shark”.

- Say that the number of such users is 4B.
- If each user id is an Int64, storing those using ids in a set

  64 bits x 4 x 10^9 = 32GBs !

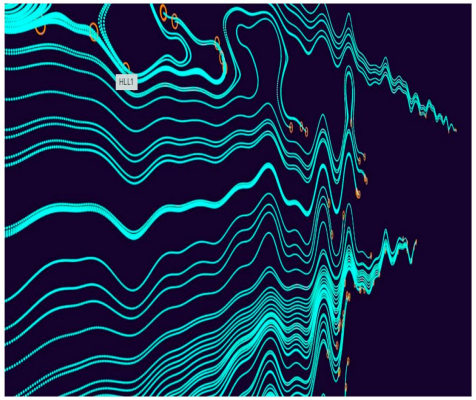
- Just for “Baby Shark”, we need 32 Gbs of storage (just one video).



engineering.fb.com/2018/12/13/data-infrastructure/hyperloglog/

POSTED ON DECEMBER 13, 2018 TO DATA INFRASTRUCTURE, OPEN SOURCE

HyperLogLog in Presto: A significantly faster way to handle cardinality estimation



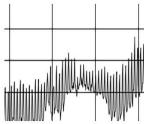
Related Posts

Nov 13, 2018

Data @Scale - Boston recap

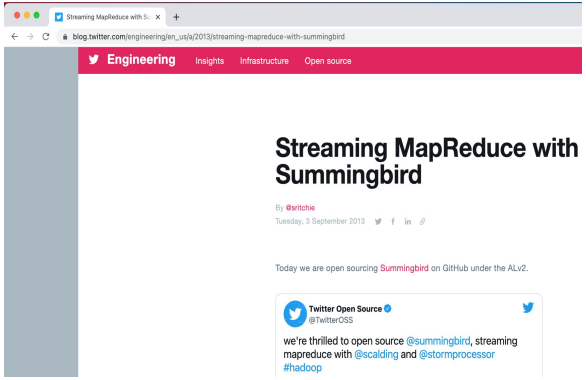
Sep 21, 2018

Open-sourcing LogDevice: A distributed data store for sequential data

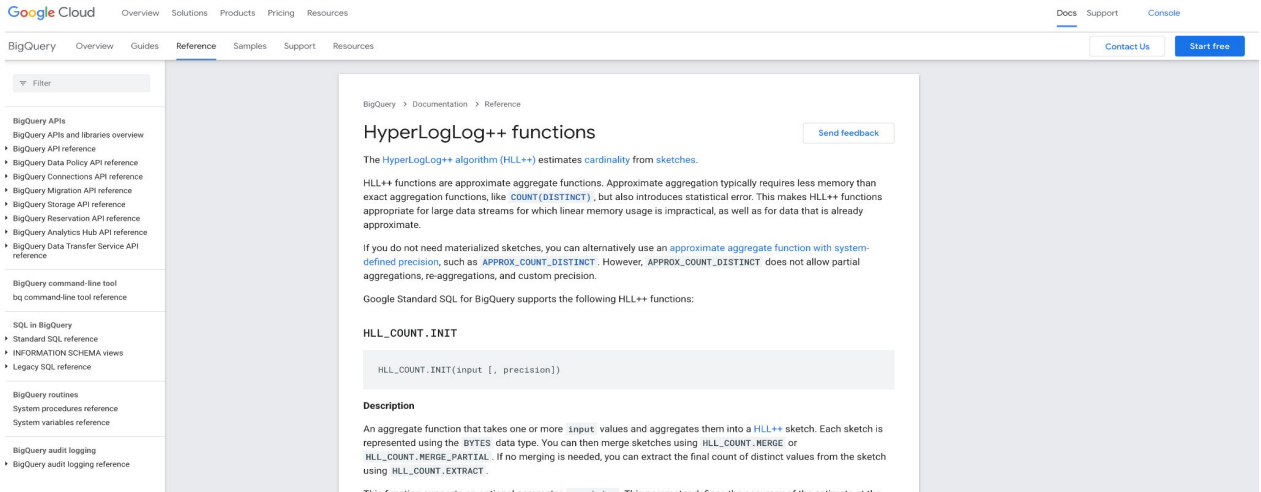


Jun 12, 2018

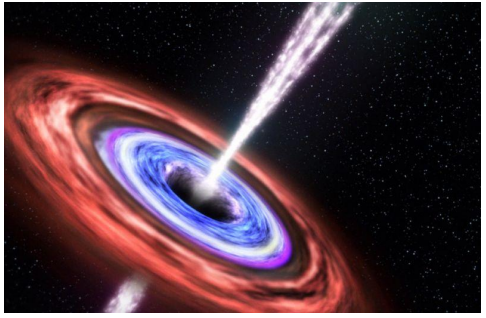
Building data science teams to have an impact at scale



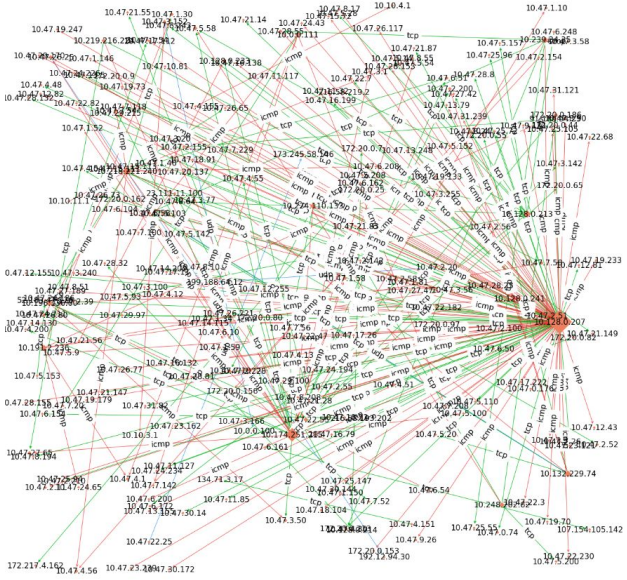

```java
package com.hll.redis;

import java.util.UUID;
import redis.clients.jedis.Jedis;

public class RedisHLL {
    static int max = 100000;
    static int batchSize = 1000;
    static String usersKey = "userids";

    public static void populaterRedis(Jedis jedis, String redisKey) {
        int i;
        for (i = 0; i <= max / batchSize; i++) {

            String[] arr = new String[batchSize];
            for (int j = 0; j < batchSize; j++) {
                String uuid = UUID.randomUUID().toString();
                arr[j] = uuid;
            }

            jedis.pfadd(redisKey, arr);
        }
    }

    public static void main(String[] args) {

        try {
            Jedis jedis = new Jedis("redis://localhost:6379");

            String usersKey0, usersKey1, usersKey2, usersKey3, usersKey4, usersKey5 = null;
            usersKey0 = usersKey + " 20190130-12-0";
            populaterRedis(jedis, usersKey0);
            usersKey1 = usersKey + " 20190130-12-1";
            populaterRedis(jedis, usersKey1);
            usersKey2 = usersKey + " 20190130-12-2";
            populaterRedis(jedis, usersKey2);
            usersKey3 = usersKey + " 20190130-12-3";
            populaterRedis(jedis, usersKey3);
            usersKey4 = usersKey + " 20190130-12-4";
            populaterRedis(jedis, usersKey4);
            usersKey5 = usersKey + " 20190130-12-5";
            populaterRedis(jedis, usersKey5);
            long start = System.currentTimeMillis();
            long count = jedis.pfcount(usersKey0, usersKey1, usersKey2, usersKey3, usersKey4, usersKey5);
            long end = System.currentTimeMillis();
            long tenMinCount = jedis.pfcount(usersKey0);
            System.out.println("The number of uniques for  10min are " + tenMinCount);
            System.out.println("The number of unique for hour are " + count);
            System.out.println("Time taken for uniques is  " + (end - start));
        } catch (Exception e) {
            System.out.println("Some problem with redis" + e.getMessage());
        }
    }
}
```



Applications



Astronomy



Network traffic

Actor	Movie	Year
DiCaprio	Django	2012
Pacino	Devil's Advocate	1997
DiCaprio	Inception	2010
Pacino	Scarface	1983
DiCaprio	Shutter Island	2010
Foxx	Django	2012

Query size estimation, query optimization

Unique Visitor

A unique visitor is a term used in marketing analytics which refers to a person who has visited the website at least once and is counted only once in the reporting time period. So if the user visits the web more than once, it counts as one visitor only. It's also called a "Unique User".

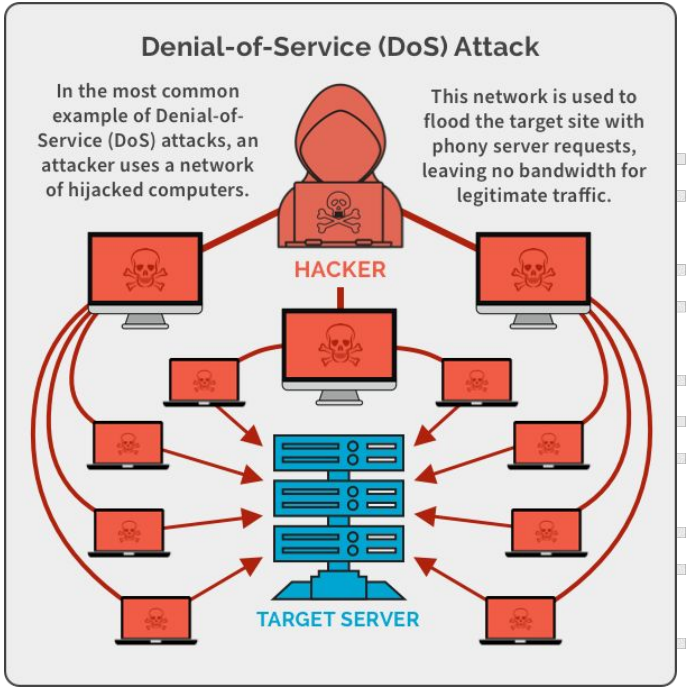
Marketing

Applications

- Compute the top-10 most frequent IP source/target over some router

  SELECT ip_source, ip_dest, **COUNT(*) AS** f
  FROM *Router*
  GROUP BY ip_source, ip_dest
  ORDER BY COUNT(*) DESC
  LIMIT 10

- What if your database has a continuous stream of Updates, i.e., tuples inserted, and deleted?



Moment estimation problem

- Distinct elements is a special case of computing p-th frequency moments

$$F_p = \sum_{i=1}^{n} f_i^p$$

- **p=0:** distinct elements (convention $0^0=0$)
- **p=1:** length of stream
- **p=2:** size of self-join in DB
- **p=+inf:** Here we define $F_\infty^\star = \max_i(f_i)$

In practice we will be interested in finding the heavy hitters, elements that appear frequently in the stream.

Two Warm-up Problems and a useful technique

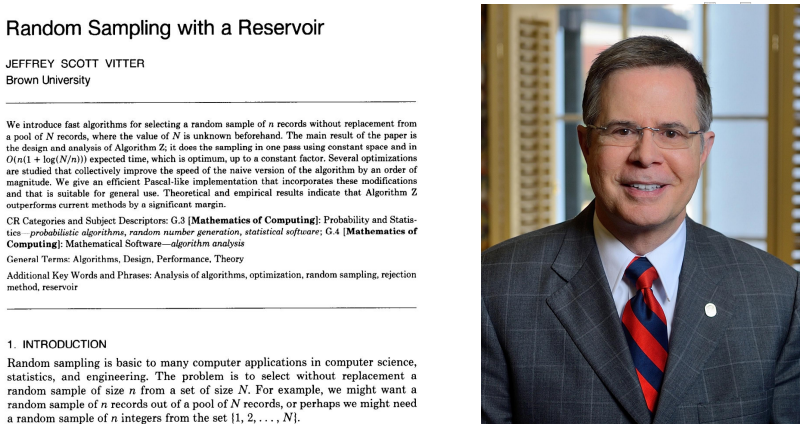
Missing number

- Let σ be an arbitrary permutation of {1,…,n}.
  - E.g., for n=6, σ=(5,2,3,6,1,4)
- An element **j** is removed from σ.
  - E.g, if j = 1, then the permutation σ_{-1}=(5,2,3,6,4)
- We get to see σ_{-j}, but we do not know what element j was removed.
- How much space do we need to find the missing number **j**?

Reservoir sampling

- How do we get a random sample (i.e., one element), from a stream of size n?
- What if the size n is unknown?
  - Reservoir sampling!
- **Algorithm**: When element x_j arrives we update our sample with value x_j with probability 1/j.

HW problem



Random Sampling with a Reservoir

JEFFREY SCOTT VITTER
Brown University

We introduce fast algorithms for selecting a random sample of n records without replacement from a pool of N records, where the value of N is unknown beforehand. The main result of the paper is the design and analysis of Algorithm Z; it does the sampling in one pass using constant space and in $O(n(1 + \log(N/n)))$ expected time, which is optimum, up to a constant factor. Several optimizations are studied that collectively improve the speed of the naive version of the algorithm by an order of magnitude. We give an efficient Pascal-like implementation that incorporates these modifications and that is suitable for general use. Theoretical and empirical results indicate that Algorithm Z outperforms current methods by a significant margin.

CR Categories and Subject Descriptors: G.3 [**Mathematics of Computing**]: Probability and Statistics—*probabilistic algorithms, random number generation, statistical software*; G.4 [**Mathematics of Computing**]: Mathematical Software—*algorithm analysis*

General Terms: Algorithms, Design, Performance, Theory

Additional Key Words and Phrases: Analysis of algorithms, optimization, random sampling, rejection method, reservoir

1. INTRODUCTION

Random sampling is basic to many computer applications in computer science, statistics, and engineering. The problem is to select without replacement a random sample of size n from a set of size N. For example, we might want a random sample of n records out of a pool of N records, or perhaps we might need a random sample of n integers from the set $[1, 2, \ldots, N]$.

A useful technique

Theorem: Let X be an unbiased estimator of a quantity Q. Let $\{X_{ij}\}_{i\in[t],j\in[k]}$ be a collection of independent RVs with X_{ij} distributed identically to X, where

$$t = O\left(\log \frac{1}{\delta}\right),\ k = O\left(\frac{Var[X]}{\epsilon^2 E[X]^2}\right)$$

Let $Z = median_{i\in[t]} \frac{1}{k}\sum_{j=1}^{k} X_{ij}$. Then, $\mathbf{Pr}(|Z - Q| \geq \epsilon Q) \leq \delta$.

Proof sketch: Chebyshev and Chernoff. (Homework problem)

SELECT COUNT(DISTINCT) Estimating F_0

Naive algorithmic solutions

- **Algorithm 1**
  - Maintain a bitmask with n bits, one per item in the universe.
  - When an element x appears in the stream, if BITMASK[x]=0, set it to 1.

Space complexity: O(n)

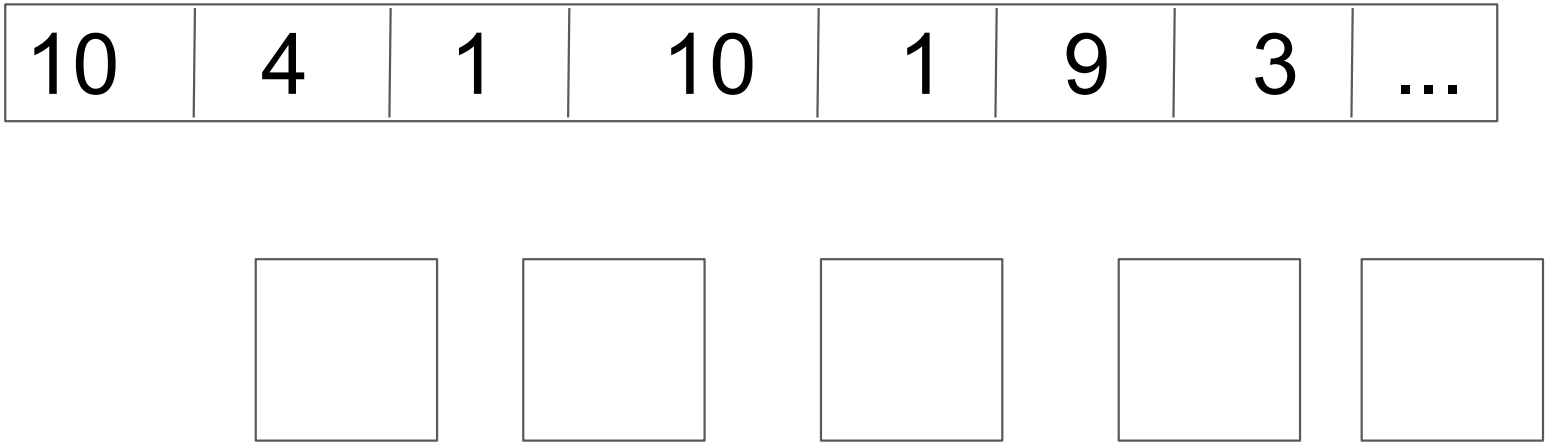
- **Algorithm 2**
  - Store the whole stream

Space complexity: O(mlog(n))

- **Algorithm 3**
  - d=Dict{}
  - For each x in stream
    - *if* x is in dictionary d, do nothing
    - *else* add x to d
  - Return size of d

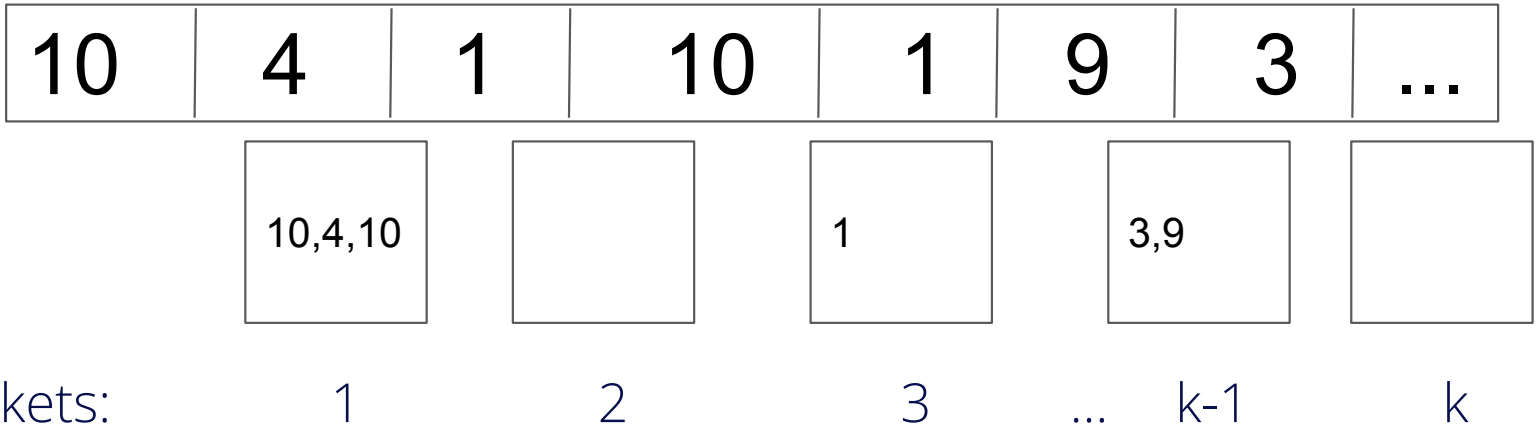
Worst-case space complexity: O(min(n,mlog(n))

Algorithm 1: Linear counting



Suppose we send each item x in the stream to a bucket according to some random (idealized for now) hash function.

Algorithm 1: Linear counting



Let Z be the number of buckets that didn't receive any item. In expectation, we obtain

$$E[Z] = k\left(1 - \frac{1}{k}\right)^{F_0}$$

Estimator: $F_0 = k \ln \frac{k}{z}$

Algorithm 1: Linear counting

- Setting the numbers of bins k requires knowing F_0, the quantity we wish to estimate
- By first computing the variance of Z, and then applying Chebyshev, we obtain the following corollary:

  Setting $k=F_0/12$ yields a standard error of less than 1%...

- .... However, F_0 can be O(n), so the space can also be prohibitively large using this approach.

Can we do better than this, and if yes, how much better can we do?

Algorithm 2: Idealized F_0 estimation

Suppose we have access to a random hash function h:U→[0,1].

- V ← +inf
- For each x
  - If h(x)<V then V← h(x) make V small
- At the end of the stream output 1/V-1 as our estimate for the number of distinct elements

Question: Why this specific output?

Minimum of n uniform random variables

Let's assume that the hash function is fully random.

- .
- Z=min(X_1,…,X_n)

$$X_i \sim U[0,1] \text{ for } i \in [n]$$

What is the expectation E[Z]?

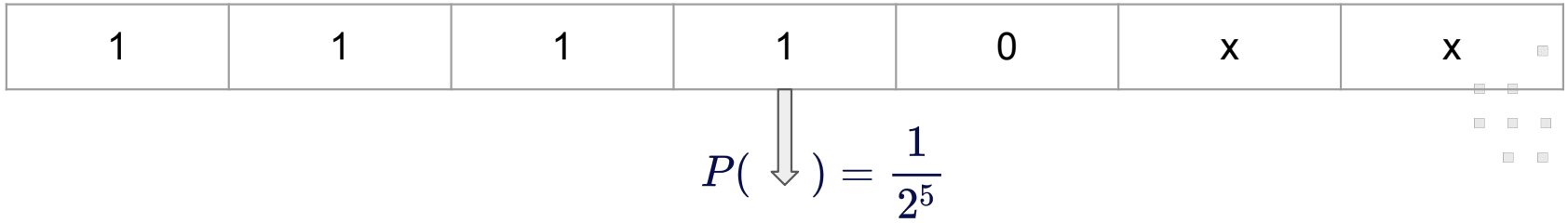
$$\mathbb{E}[\mathbb{Z}] = \int_0^1 \Pr(Z > t)dt = \int_0^1 \Pr\left(X_1 > t\right)^n = \int_0^1 (1-t)^n dt = \frac{1}{n+1}$$

Try to write sudo code

SELECT COUNT(DISTINCT)
Estimating F_0 in practice

Flajolet-Martin (FM) sketch

- **Key assumption**: We have an idealized hash function that maps each element of the universe into a string of random bits (i.e., Pr(bit=0)=Pr(bit=1)=½)



- **Intuition**: If we see the prefix **11110xx**, probably we have seen more than 32 distinct items.

- **Idea**: Keep track of prefixes of the form 1^k0

Flajolet-Martin (FM) sketch

Important functions

1. h(x) = hash function that transforms x into a uniform $\{0,1\}^{\infty}$ binary string
2. ρ(x) = position of leftmost 0 (e.g., ρ(**1110**10100000)=4), or in terms of the usual msb to lsb position of rightmost 0 ρ(00000101**0111**)=4)

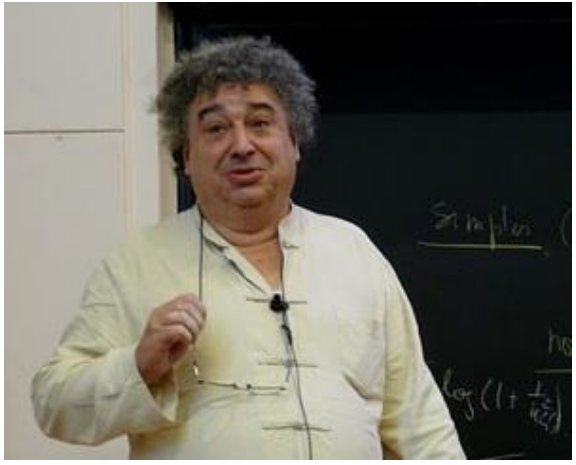
Algorithm

- Initialize a bitmask with L bits to 0, i.e., BITMASK

0	0	….	0

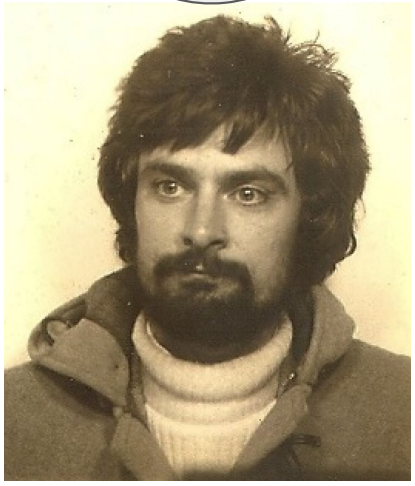
- For each element x in the stream
  - BITMASK[ρ(h(x))] ← 1
- Set R← ρ(BITMASK)
- Output $\lfloor \frac{2^R}{\underbrace{0.77351}_{\phi}} \rfloor$

"Without the analysis there is no algorithm"



Flajolet

"2^R seems a bit lower than F^0." Empirical correction?



Martin

Flajolet-Martin (FM) sketch

- **FM estimator**: Clearly, 2^R where R is the largest size of such a prefix approximates the number of distinct elements. However, it turns out there is some small bias:

$$E\left[2^R\right] \neq F_0$$

- Flajolet and Martin proved the following remarkable result:

$$E\left[2^R\right] = \phi \cdot F_0 \text{ where } \phi = \frac{e^\gamma \sqrt{2}}{3} \prod_{p=1}^{+\infty} \left(\frac{(4p+1)(4p+2)}{4p(4p+3)} \right)^{(-1)^{\nu(p)}}$$

where ν(p)=#bits equal to 1 in binary representation of p

- Thus, an unbiased (modulo negligible terms) estimator becomes $\left\lfloor \frac{2^R}{\underbrace{0.77351}_{\phi}} \right\rfloor$

- Fascinating analysis of an algorithm

Bit manipulation

Find first set

From Wikipedia, the free encyclopedia

In computer software and hardware, **find first set (ffs)** or **find first one** is a bit operation that, given an unsigned machine word,[nb 1] designates the index or position of the least significant bit set to one in the word counting from the least significant bit position. A nearly equivalent operation is cou
number of zero bits following the least significant one bit. The complementary operation that finds the index or position of the most significant set bit is **log base 2**, so called because it computes the binary logarithm $\lfloor \log_2(x) \rfloor$.[1] This is closely related to **count leading zeros** (**clz**) or **number of l**
significant one bit.[nb 2] There are two common variants of find first set, the POSIX definition which starts indexing of bits at 1,[2] herein labelled ffs, and the variant which starts indexing of bits at zero, which is equivalent to ctz and so will be called by that name.

Most modern CPU instruction set architectures provide one or more of these as hardware operators; software emulation is usually provided for any that aren't available, either as compiler intrinsics or in system libraries.

Contents [hide]

Examples [edit]

Given the following 32-bit word:

0000 0000 0000 0000 1000 0000 0000 1000

The count trailing zeros operation would return 3, while the count leading zeros operation returns 16. The count leading zeros operation depends on the word size: if this 32-bit word were truncated to a 16-bit word, count leading zeros would return zero. The find first set operation would return 4, i

Similarly, given the following 32-bit word, the bitwise negation of the above word:

1111 1111 1111 1111 0111 1111 1111 0111

The count trailing ones operation would return 3, the count leading ones operation would return 16, and the find first zero operation ffz would return 4.

If the word is zero (no bits set), count leading zeros and count trailing zeros both return the number of bits in the word, while ffs returns zero. Both log base 2 and zero-based implementations of find first set generally return an undefined result for the zero word.

Hardware support [edit]

Many architectures include instructions to rapidly perform find first set and/or related operations, listed below. The most common operation is count leading zeros (clz), likely because all other operations can be implemented efficiently in terms of it (see Properties and relations).

HAKMEM -- CONTENTS -- Di

inwap.com/pdp10/hbaker/hakmem/hakmem.html

Update

Massachusetts Institute of Technology
A. I. Laboratory
Artificial Intelligence Memo No. 239
February 29, 1972

HAKMEM

contents index

M. Beeler [beeler@bbn.com]
R. W. Gosper [rwg@newton.macsyma.com]
R. Schroeppel [rcs@cs.arizona.edu]

[Retyped and formatted in 'html' ('Web browser format) by Henry Baker, April, 1995. The goal of this 'html' document is to make HAKMEM available to the widest possible audience -- including those without bitmapped graphics browsers. Therefore, equations have been formatted to be readable even on ASCII browsers such as 'lynx'. Click here to get original AI Memo 239 in 400 dots/inch, 1 bit/pixel, Group 4 facsimile TIFF format (a single 5 megabyte gzip compressed tar file, AIM-239.tiff.tar.gz).]

Work reported herein was conducted at the Artificial Intelligence Laboratory, a Massachusetts Institute of Technology research program supported in part by the Advanced Research Projects Agency of the Department of Defense and monitored by the Office of Naval Research under Contract Number N00014-70-A-0362-0002.

Compiled with the hope that a record of the random things people do around here can save some duplication of effort -- except for fun.

Here is some little known data which may be of interest to computer hackers. The items and examples are so sketchy that to decipher them may require more sincerity and curiosity than a non-hacker can muster. Doubtless, little of this is new, but nowadays it's hard to tell. So we must be content to give you an insight, or save you some cycles, and to welcome further contributions of items, new or used.

The classification of items into sections is even more illogical than necessary. This is because later elaborations tend to shift perspective on many items, and this elaboration will (hopefully) continue after publication, since this text is retained in "machinable" form. We forgive in advance anyone deterred by this wretched typography.

People referred to are from the A. I. Lab:

```
Marvin Minsky          [minsky@ai.mit.edu]
Bill Gosper            [rwg@newton.macsyma.com]
Michael Beeler         [beeler@bbn.com]
Joe Cohen
Jerry Freiberg
John Roe
Rich Schroeppel        [rcs@cs.arizona.edu]
David Silver
Michael Speciner
Richard Stallman       [rms@ai.mit.edu]
Gerald Sussman         [gjs@ai.mit.edu]
David Waltz
```

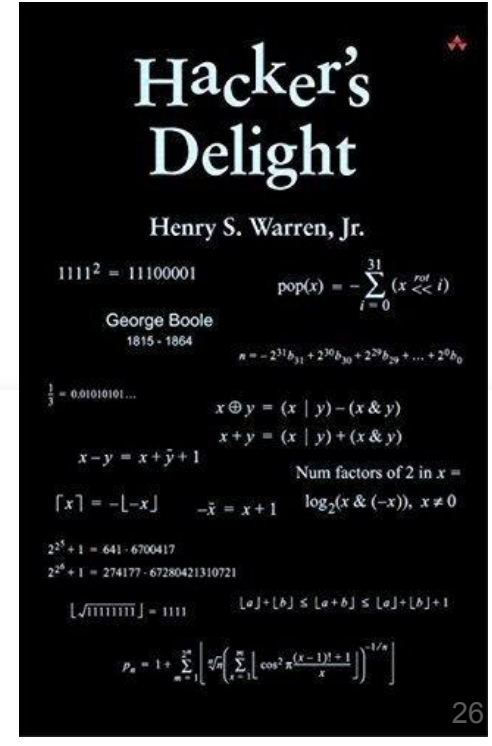
Once at the A. I. Lab but now elsewhere:

```
Jan Kok                    William Henneman
Ricl Liknaitzky            George Mitchell
Peter Samson               Stuart Nelson
Roger Banks                Rollo Silver
Mike Paterson              [Mike.Paterson@dcs.warwick.ac.uk]
```

Bit Twiddling Hacks

By Sean Eron Anderson
seander@cs.stanford.edu

Contents



Implementation details

- Bitmask operations can Be done very efficiently.

- E.g., R(x):
  x=1215
  '0b10010111111'
  x+1
  0b1001**1**000000
  x+1 & ~x
  0b0000**1**000000

In [6]:
```
function R(x)
    """
    Returns 2 to the power of trailing ones of input number x.
    """
    return ~x & (x+1)
end
```

Out[6]: R (generic function with 1 method)

In [7]:
```
function my_trailing_ones(x)
    """
    Returns the number of trailing ones of number x
    """
    return count_ones(R(x)-1)
end
```

Out[7]: my_trailing_ones (generic function with 1 method)

In [8]:
```
@benchmark my_trailing_ones(x)
```

Out[8]:
```
BechmarkTools.Trial: 10000 samples with 997 evaluations.
 Range (min … max):   13.789 ns … 92.621 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):      14.801 ns              ┊ GC (median):    0.00%
 Time  (mean ± σ):    16.231 ns ±  4.124 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%
```


```
Memory estimate: 0 bytes, allocs estimate: 0.
```

In [9]:
```
@benchmark trailing_ones(x)
```

Out[9]:
```
BechmarkTools.Trial: 10000 samples with 997 evaluations.
 Range (min … max):   14.823 ns … 121.835 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):      15.325 ns               ┊ GC (median):    0.00%
 Time  (mean ± σ):    16.414 ns ±   3.144 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%
```


```
Memory estimate: 0 bytes, allocs estimate: 0.
```

Flajolet-Martin (FM) sketch

- Is the constant $1/\varphi$ unexpected in the estimator?
  - **Intuition**: since if we have a prefix 1111…10 there are likely to be more than 2^R and less than 2^{R+1} elements.

- Variance of the estimator is 1.257, which in practice means it can be off by a factor of 2.

- **Question**: how do we reduce variance?

  - **Idea 1**: use multiple hash functions

  - **Idea 2**: stochastic averaging

Stochastic Averaging

- We improve the accuracy of the algorithm by a multiplicative factor of $\frac{1}{\sqrt{k}}$ by taking the average of k different hash function ⇒ expensive

- **Better idea**: use the first bits to create substreams!

  Split the elements in $k=2^l$ substreams by using the first l bits of the hash. E.g, for l=2
  h(v) = $\mathbf{b_1 b_2}$ $\mathbf{b_3 b_4 b_5}$...
  Substream id hash value

$$h(x) = \begin{cases} 00b_3b_4\ldots & \Rightarrow bitmap_{00}[\rho(b_3b_4\ldots)] = 1 \\ 01b_3b_4\ldots & \Rightarrow bitmap_{01}[\rho(b_3b_4\ldots)] = 1 \\ 10b_3b_4\ldots & \Rightarrow bitmap_{10}[\rho(b_3b_4\ldots)] = 1 \\ 11b_3b_4\ldots & \Rightarrow bitmap_{11}[\rho(b_3b_4\ldots)] = 1 \end{cases}$$

Flajolet-Martin theorem

The estimator Z is asymptotically unbiased, i.e., $E_n[Z] \rightarrow n$ and the coefficient of variation

using k bitmasks is $\frac{\sigma_n(Z)}{n} = \frac{0.78}{\sqrt{k}}$

Memory: O(klogn)

Example: can count cardinalities up to 10^9 with error <=6% using 4kBs of memory (=4096 bytes)**

"Caveat" of their work: Practical implementations of the hash function are not discussed.

** See also Mitzenmacher-Vadhan**: Why simple hash functions work: Exploiting the entropy in a data stream**

Hyperloglog (HLL)

Important functions

1. h(x) = hash function that maps each element to a counter
2. g(x) = ideal hash function
3. z(x) = 1+#leading 0s (e.g., z(**000**10100000)=3+1=4)

Init(m)

Pick hash functions h,g, and store m

for i ← 1 to m

    C[i] ← 0

Update(x)

C[h(x)] ← max(C[h(x)], z(g(x))

Hyperloglog (HLL)

$$Z := \left(\sum_{j=1}^{m} 2^{-M^{(j)}} \right)^{-1}.$$

It then returns a normalized version of the harmonic mean of the $2^{M^{(j)}}$ in the form,

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

query()

X ← 0

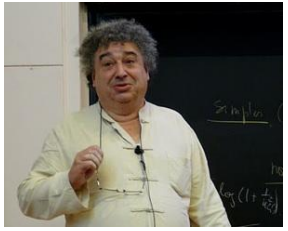
for j ← 1 to m

X ← X+$2^{-C[j]}$

Return $\alpha_m m^2/X$ where

Example: Suppose m=3

$$\alpha_m = \frac{0.7213}{1 + \frac{1.079}{m}}$$



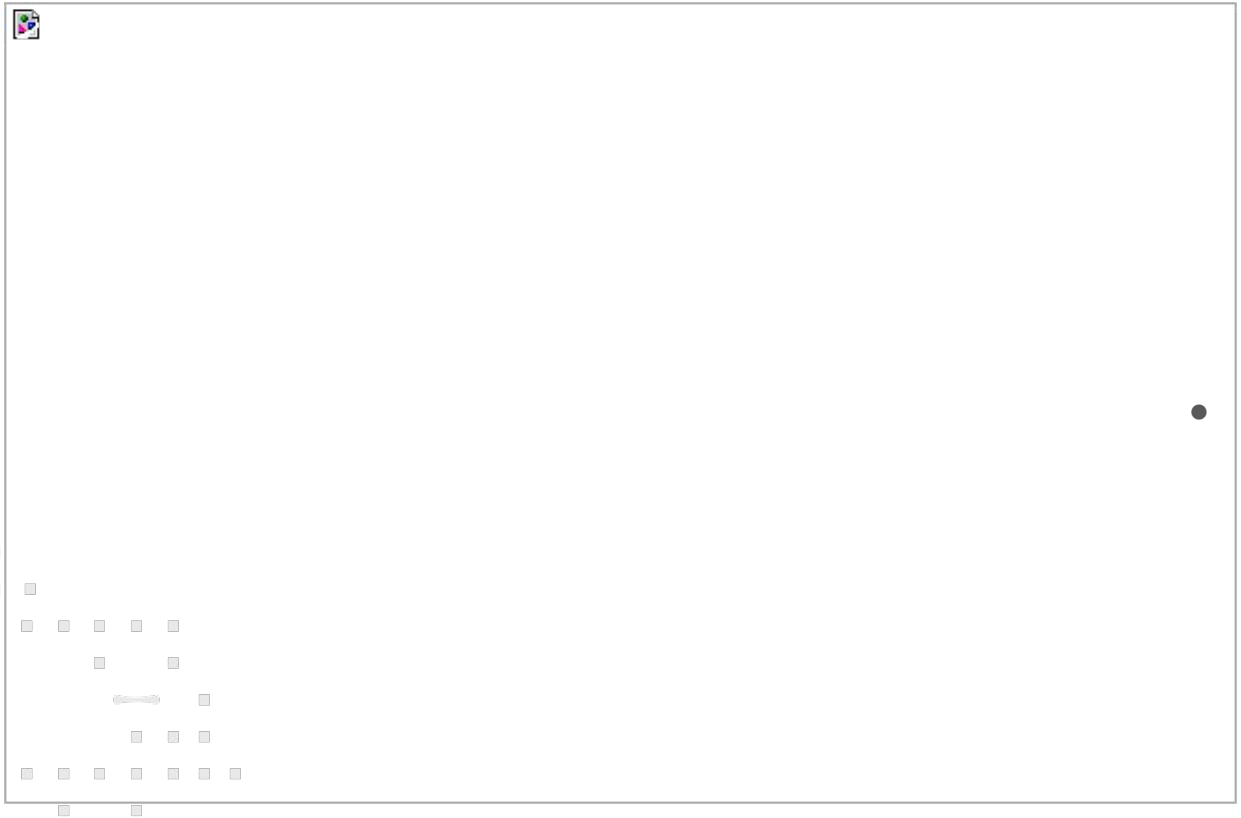
x	e_1	e_2	e_3	e_4	e_5	e_6
h(x)	1	2	3	3	1	2
g(x)	01010	00101	01100	10000	00011	01101

Hyperloglog (HLL)

x	e_1	e_2	e_3	e_4	e_5	e_6
h(x)	1	2	3	3	1	2
g(x)	**0**1010	**00**101	**0**1100	10000	**000**11	**0**1101
z(g(x))	2	3	2	1	4	2

Hyperloglog (HLL)

https://djharper.dev/demos/hyperloglog/



- Why is the estimator ~m^2/X?
  - Assuming h is ideal, each counter will be “responsible” ~F0/m distinct elements.
  - The proposed estimator instead of the usual arithmetic mean, is the harmonic mean m/X

Hyperloglog (HLL) - Remarks

- In the original Flajolet et al. HLL paper, g hashes from 0..31, i.e., 5 bits needed to represent the value g(x).
  - This suffices to count 2^{32}=4,294,967,296, i.e., inputs with billions of entries, i.e., then we do not expect to have more than 32 trailing 0s!

- High end correction
  - When dealing with large cardinalities, the HLL paper proposes a correction that accounts for the large number of collisions

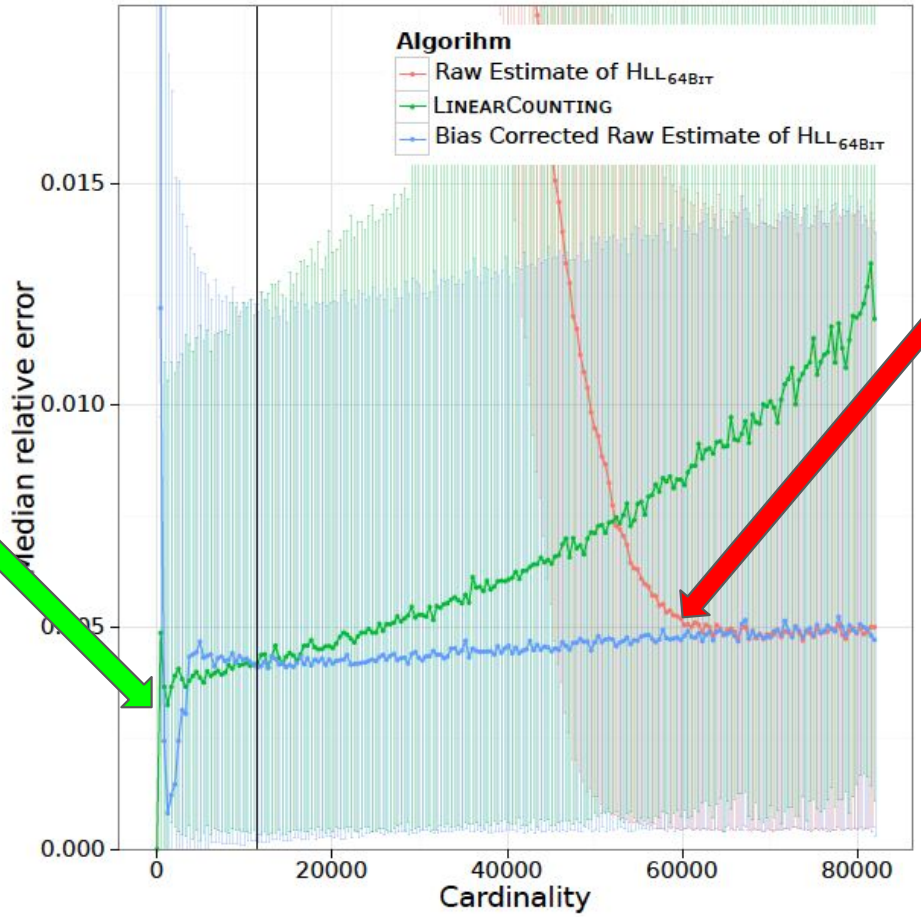
HyperLogLog++ (HLL++) [Heule-Nunkesser-Hall]

- In the HLL++ paper, g hashes from 0..63, i.e., 6 bits needed to represent the value g(x).
  - This suffices to count 2^{64}, i.e., inputs with trillions of entries!
  - With this simple addition of a bit, the high end correction of Flajolet et al. becomes redundant for current benchmarks/datasets
  - If we use 8 bits instead, then we can count approximately up to $2^{256} \sim 10^{77}$ >natural quantities in the world

- They introduce bias-corrected raw estimates

Low end correction

For small cardinalities, linear counting is better than HLL



For large cardinalities HLL is better than linear counting

Remarks

- For small counts, the HLL sketch can be (near-)empty, obtaining a harmonic average close 1, and thus receiving an estimate of approximate 0.7m for the distinct elements >> true F_0

- Heule, Nunkesser, and Hall not only use linear counting in the “low” end regime, but also learn the bias!
  - They build a tabulation between estimated and true values, and then interpolate.
  - Good practice when reported F_0 is <5m.

- They also use efficient sparse representation of the counters, useful when space is at a very high premium

Accuracy evaluation

- **relative error metric**: $\frac{\left|\tilde{F_0} - F_0\right|}{F_0}$ (misleading wrt under- vs over-estimate)
- **ratio error metric:** $\max\left(\frac{\tilde{F_0}}{F_0}, \frac{F_0}{\tilde{F_0}}\right)$

- **standard error metric**: the *standard error* of an estimator Y with standard deviation $\sigma_Y(F_0)$ is $\sigma_Y(F_0)/F_0$.

- **(ε,δ)-approximation scheme** for F_0 is a randomized procedure that given any positive ε<1, δ<1, outputs an estimate $\tilde{F_0}$ in $[(1-\varepsilon)F_0, (1+\varepsilon)F_0]$ with probability > 1-δ.

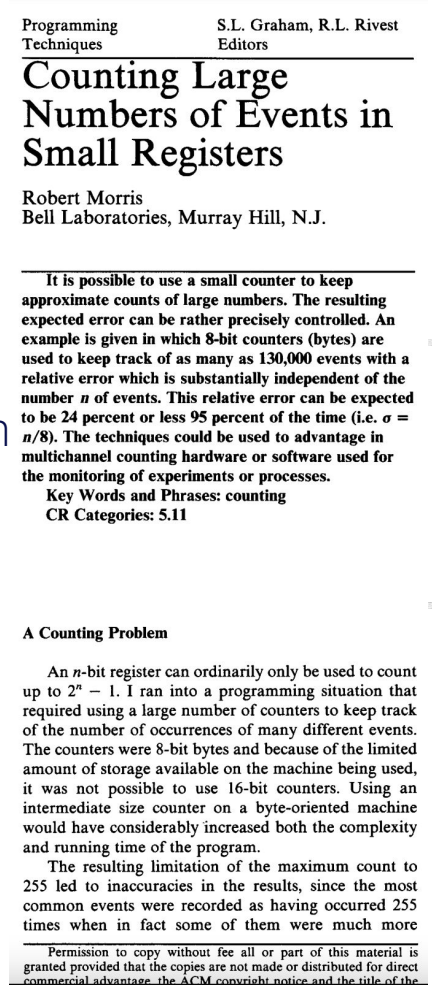
Algorithms for estimating F_1

Morris algorithm

- Robert Morris has been credited historically with the first streaming algorithm for estimating the F1 norm of a stream.
  - Cash register model/insertion only

- **Key idea**: instead of maintaining the actual length of the stream m, keep the logarithm.
  - E.g., if m=145, then by knowing the order of magnitude ~10^2 we can tell that our number is between 100 and 999

- This allows us to use loglog(m) bits to represent m ***approximately***



Programming Techniques | S.L. Graham, R.L. Rivest Editors

Counting Large Numbers of Events in Small Registers

Robert Morris
Bell Laboratories, Murray Hill, N.J.

It is possible to use a small counter to keep approximate counts of large numbers. The resulting expected error can be rather precisely controlled. An example is given in which 8-bit counters (bytes) are used to keep track of as many as 130,000 events with a relative error which is substantially independent of the number n of events. This relative error can be expected to be 24 percent or less 95 percent of the time (i.e. $\sigma = n/8$). The techniques could be used to advantage in multichannel counting hardware or software used for the monitoring of experiments or processes.

Key Words and Phrases: counting
CR Categories: 5.11

A Counting Problem

An n-bit register can ordinarily only be used to count up to $2^n - 1$. I ran into a programming situation that required using a large number of counters to keep track of the number of occurrences of many different events. The counters were 8-bit bytes and because of the limited amount of storage available on the machine being used, it was not possible to use 16-bit counters. Using an intermediate size counter on a byte-oriented machine would have considerably increased both the complexity and running time of the program.

The resulting limitation of the maximum count to 255 led to inaccuracies in the results, since the most common events were recorded as having occurred 255 times when in fact some of them were much more

How to save 1 bit?

- Maintain a counter **c** (aka Morris counter)
- init(): **c**← 0
- process()
  For each item in the stream
  - Increase **c** with probability 1/2
  - o/w keep same value
- Output estimate 2c

Let Z be the value of the counter after m increments.

- Z~Bin(m,½)
  - E[Z]=m/2
  - Var[Z] =m/4
  - Space complexity: lg(m/2)=lg(m)**-1** → we saved one bit at the cost of accuracy.

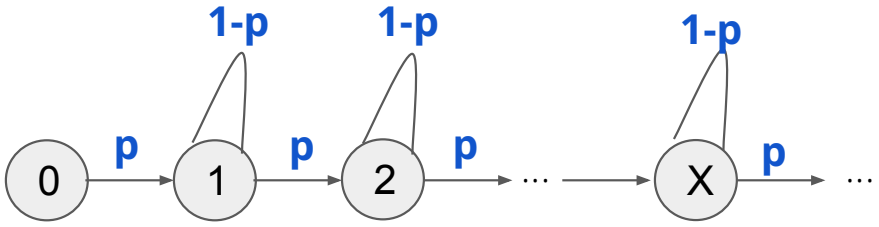
How to save k bits?

- Maintain a counter **c** (aka Morris counter)
- init(): **c**← 0
- process()
  For each item in the stream
  - Increase **c** with probability $1/2^{\mathbf{k}}$
  - o/w keep same value
- Output estimate $2^{\mathbf{k}}c$

Let Z be the value of the counter after m increments.

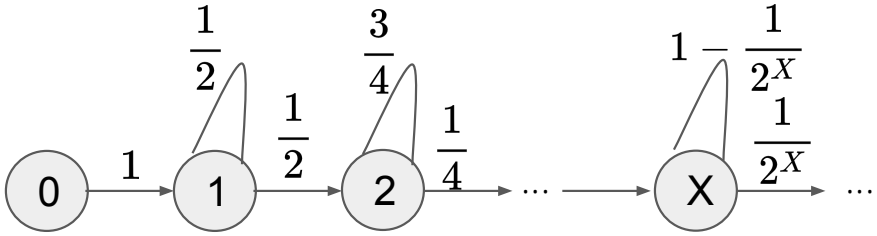
- $Z \sim Bin(m, 2^{-k})$
  - $E[Z]=m/2^k$
  - $Var[Z] \sim m/2^k$
  - Space complexity: $\lg(m/2)=\lg(m)-\mathbf{k}$ → we saved k bits

How to save k bits?



- Another perspective as a birth process
- Counter values follow a binomial distribution: $P(C_m = k) = \binom{m}{k} p^k (1-p)^{m-k}$

Morris algorithm → birth process with adaptive sampling



- Maintain a log-counter **c** (aka Morris counter)
- init(): **c**← 0
- process()
  For each item in the stream
  - Increase **c** with probability $1/2^{\mathbf{c}}$
  - o/w keep same value
- Output estimate $2^{\mathbf{c}}-1$

Why this estimator?

Claim: Define X_n to be the value of the counter after n increments. Then, $E[2^{X_n}]=n+1$.

Proof (induction)

Base case: If n=0, $X_n=0$ and thus the claim holds.

Inductive step: By conditional expectation rule $E[2^{X_{n+1}}]=E[E[2^{X_{n+1}} \mid X_n]]$ and the inductive hypothesis, we obtain the following expression:

$$
\begin{aligned}
E\left[2^{X_{n+1}}\right] &= \sum_{j=0}^{+\infty} P(X_n = j) E\left[2^{X_{n+1}} \mid X_n = j\right] \\
&= \sum_{j=0}^{+\infty} P(X_n = j)\left[2^j\left(1 - \frac{1}{2^j}\right) + 2^{j+1}\frac{1}{2^j}\right] \\
&= \sum_{j=0}^{+\infty} P(X_n = j)\left(2^j + 1\right) = E\left[2^{X_n}\right] + \sum_{j=0}^{+\infty} \Pr(X_n = j) \\
&= E\left[2^{X_n}\right] + 1
\end{aligned}
$$

Properties of Morris algorithm

1. The expectation of the variable $Z=2^{X_m}$ satisfies the following:

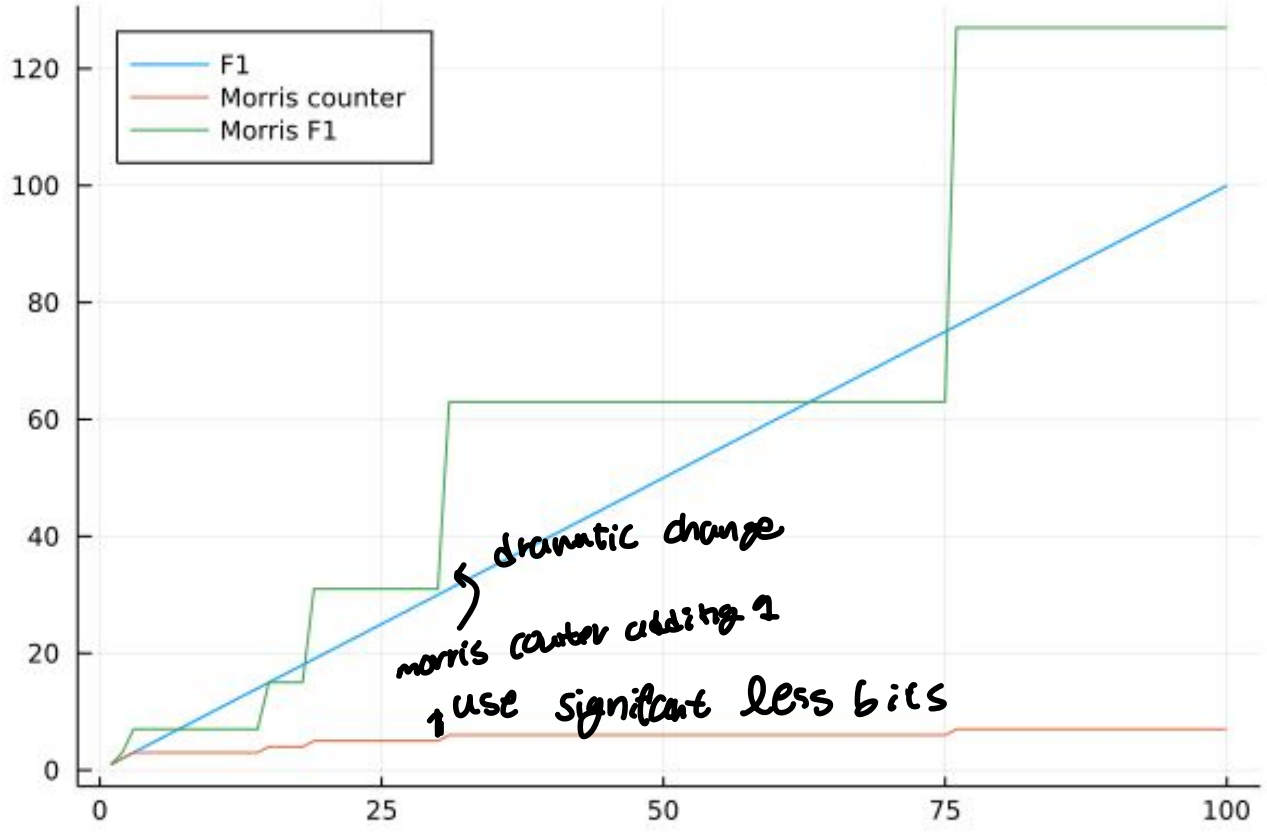
$$E[Z]=m+1$$

Corollary: Morris algorithm outputs an unbiased estimator of m.

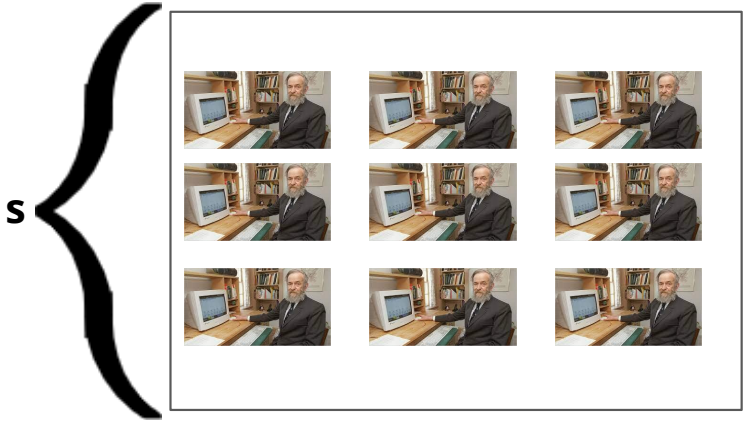
1. The variance of Z is equal to **$Var[Z]=m(m-1)/2$**

Observation: No improvement in terms of concentration as m grows since $Var(Z)/E(Z)^2$ is constant.

Morris algorithm



Morris+



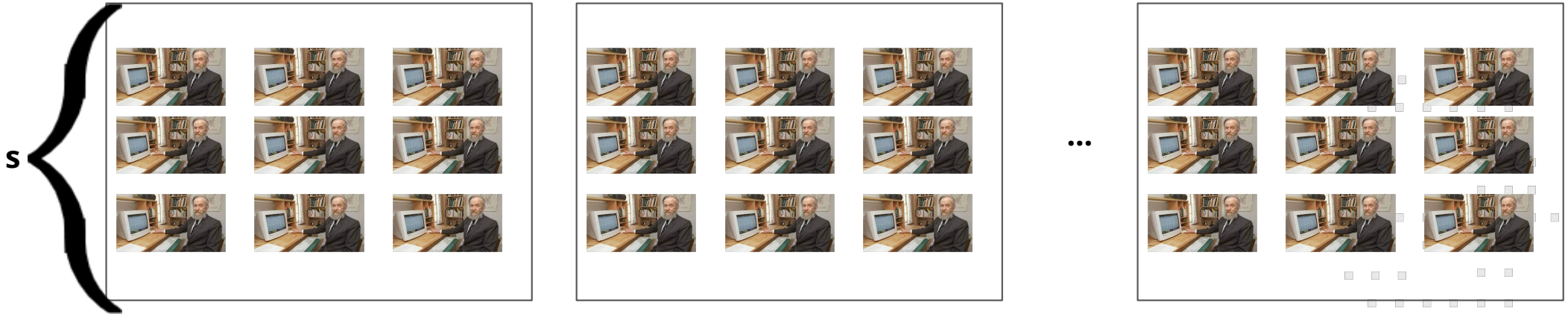
- Suppose we run **s** Morris counters and we output the average $\frac{1}{s}\sum_{i=1}^{s} m_i$
  - The average of this estimator remains the same but…
  - the variance scales down by a factor of 1/s

Reducing variance: Morris++

Moris+: avg of s independent Morris counters

Moris+: avg of s independent Morris counters

Moris+: avg of s independent Morris counters

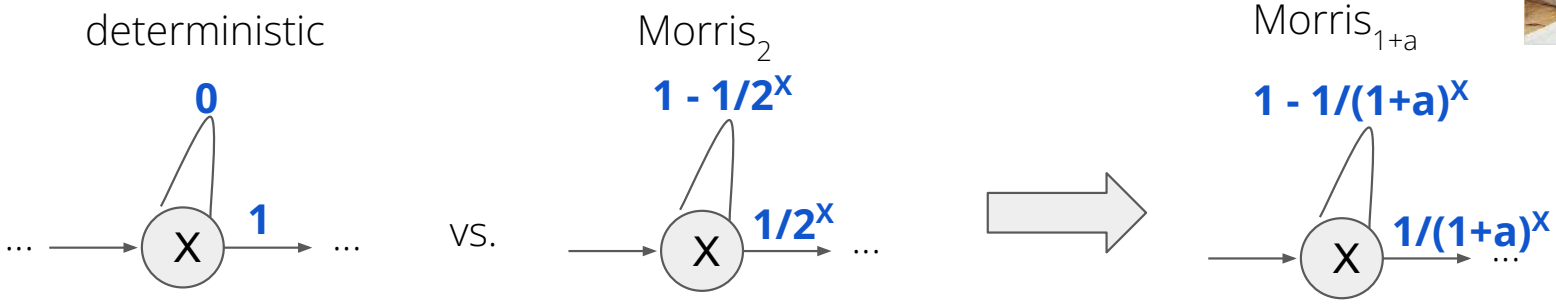


$\mathbf{avg_1}$ $\mathbf{avg_2}$ $\mathbf{avg_t}$

Morris++: Median of t **Moris+**'s where $s = \Theta\left(\frac{1}{\epsilon^2}\right), t = \Theta\left(\log \frac{1}{\delta}\right)$

Claim: The space complexity with probability 1-δ is $O\left(\frac{1}{\epsilon^2} \lg\left(\frac{1}{\delta}\right) \lg\lg\left(\frac{n}{\epsilon\delta}\right)\right)$ and we obtain an (ε,δ)-approximation scheme to F1 for insert-only streams.

Reducing variance: change basis



-

- The expectation of the variable $Z=((1+a)^{X_m}-1)/a$ satisfies the following:

$$E[Z]=m$$

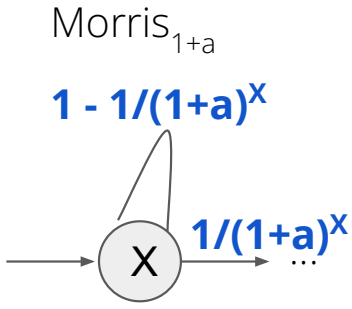
- The variance of Z is equal to **$Var[Z]=a\ m(m-1)/2$**

How to set a?

- Set **$a=2\varepsilon^2\delta$** and apply Chebyshev's inequality

$$\Pr(|Z-m| \geq \epsilon m) \leq \frac{Var(Z)}{\epsilon^2 m^2} = \frac{\frac{m(m-1)}{2} 2\delta\epsilon^2}{\epsilon^2 m^2} \leq \delta$$

- **Space complexity:** $O\left(\log\log n + \log\frac{1}{\epsilon} + \log\frac{1}{\delta}\right)$



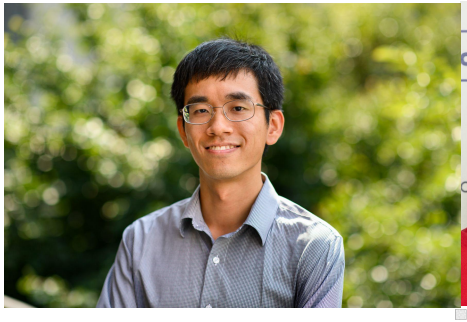
Optimal Algorithm for F1

- Nelson and Yu proved recently that Morris algorithm Is optimal by
  - Tightening the analysis of the space complexity

$$O\left(\log\log n + \log\frac{1}{\epsilon} + \log\log\frac{1}{\delta}\right)$$

for an (ε,δ)-approximation scheme to F_1.

  - Proving a tight lower bound, and thus practically nailing down the problem.



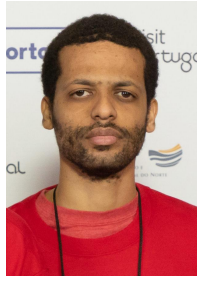
Turnstile model

- **Attempt**: Suppose we have a strict turnstile model. Can we use one Morris counter for insertions, and one for deletions and somehow combine them?

  No! If z^+, z^- are the approximate histograms of x^+,x^- of additions and deletions respectively, $|z^+-z^-|_1$ can be $O(\varepsilon m)$ off in terms of additive error from the desired $|x^+-x^-|_1$ (i.e., $O(\varepsilon)$ additive error per update in the stream).

- Optimal solution for the general turnstile model by:


Kane

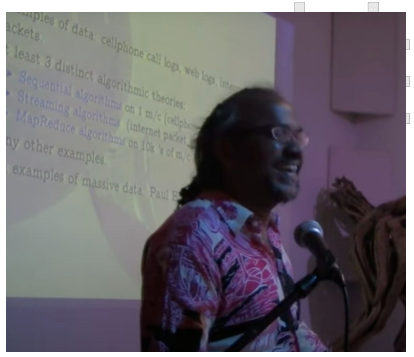

Nelson


Woodruff

Count-Min Sketch for Heavy Hitters

What is a heavy hitter?

- Let $[f_1, f_2, \ldots, f_n]$ be the vector of frequencies of the set of elements [n] after seeing a stream of length $m = f_1 + \ldots + f_n$
- Given 0<φ<1, a heavy hitter (φ-HH) is an element i such that $f_i \geq \phi m$
- The goal is to return all the approximate heavy hitters { i: such that $f_i \geq (\phi - \epsilon)m$}
- An elegant solution: Count Min (CM) sketch



Data structure

	1	2	…	b
1	0	0		0
…				
r	0	0		0

struct CMSketch

r::Int64

b::Int64

cm::Matrix{UInt64} //2d array with r rows and b buckets/columns

end

Each row **j** is associated with a hash function $\mathbf{h_j}$ and the functions are paiwise independent.

Basic API

- **init**(CMSketch)
  Set all counters in CMSketch.cm to 0
- **insert**(CMSketch, i)
  Update the data structure when element i arrives in the stream
- **query**(CMSketch, i)
  Obtain an estimate of the true frequency of element i

Insert(CM, i)

To update the data structure, we hash element i **r** times with each $\mathbf{h_j}$ hash function for j=1..r. Specifically, we update the data structure as follows

$$cm[j, \mathbf{h_j}(i)] = cm[j, \mathbf{h_j}(i)]+1 \text{ for } j=1..r$$

E.g., suppose 100 is the first element to arrive and $h_1(100)=2$ and $h_2(100)=4$. Then:

	1	2	3	4
1		+1		
2				+1

See also the Demo

query(CM,i)

To obtain an estimate of the frequency fi of element i we query the data structure. Specifically we hash element i with each $\mathbf{h_j}$ hash function for j=1..r and we return the smallest count among the entries of the cells CM[j, $\mathbf{h_j}$[i]] over all j.

For example, the estimated frequency of element 100 is
min(CM[1,h_1(100)], CM[2, h_2(100)] = 110

	1	2	3	4
1	50	110	20	5
2				130

Theoretical guarantees (proof on blackboard)

Suppose m is the length of the stream. Let the number of buckets B (#cols) be equal to $\lceil \frac{e}{\epsilon} \rceil$ and the number of repetitions (rows) set to **log(1/δ)**. Then the estimated frequency $\tilde{f}_i$ of the true frequency f_i satisfies the following guarantees:

1. $f_i \leq \tilde{f}_i$
2. $\tilde{f}_i \leq f_i + \epsilon m$ With probability at least 1-δ.

Readings

- Material, proofs and slides taught during the lectures
- Sections 6.1, 6.2, 6.3 from the Foundations of Data Science Book
- Reservoir sampling (Wikipedia article)
- Approximate counting (Morris Algorithm)
- Count Min sketch

More resources

- http://blog.notdot.net/2012/09/Dam-Cool-Algorithms-Cardinality-Estimation
- C++ implementation http://dialtr.github.io/libcount/
- https://datasketches.apache.org/
- http://dimacs.rutgers.edu/~graham/pubs/papers/cmencyc.pdf
- https://stackoverflow.com/questions/6811351/explaining-the-count-sketch-algorithm
- https://florian.github.io/count-min-sketch/