Bit-array-based alternatives to HyperLogLog

Svante Janson
Uppsala University

Jérémie Lumbroso
University of Pennsylvania

Robert Sedgewick
Princeton University

This work is dedicated to the memory of Philippe Flajolet

Philippe Flajolet 1948-2011

Bit-array-based alternatives to HyperLogLog

- A fundamental problem in data science
- Simple, elegant and efficient solutions
- A simple algorithm
- HyperBitBit
- Memory vs. accuracy comparisons

Cardinality counting: a fundamental problem in data science

Q. In a given stream of data values, how many *different* values are present?

Reference application. How many unique visitors in a web log ?

log.07.f3.txt

```
117.222.48.163
pool-71-104-94-246.lsanca.dsl-w.verizon.net
1.23.193.58
188.134.45.71
1.23.193.58
gsearch.CS.Princeton.EDU
pool-71-104-94-246.lsanca.dsl-w.verizon.net
81.95.186.98.freenet.com.ua
81.95.186.98.freenet.com.ua
81.95.186.98.freenet.com.ua
CPE-121-218-151-176.lnse3.cht.bigpond.net.au
117.211.88.36
```

6 million strings

UNIX (1970s-present)

```
% sort -u log.07.f3.txt | wc -l
1112365
```

"unique"

SQL (1970s-present)

```
SELECT
DATE_TRUNC('day',event_time),
COUNT(DISTINCT user_id),
COUNT(DISTINCT url)
FROM weblog
```

State of the art in the wild for decades. Sort, then count.

"Optimal" solution. Use a hash table. ← order of magnitude faster than sort-based solution

Q. I can't use a hash table. The stream is ***much too big*** to fit all values in memory. Now what?

Cardinality *estimation*

A. Look for a way to *estimate* the value of ***N***, the number of distinct values in the stream.

Practical cardinality estimation problem

- Make *one pass* through the stream.
- Use *as few operations per value* as possible
- Use *as little memory* as possible.
- Produce *as accurate an estimate* as possible.

typical applications where exact count is not really necessary

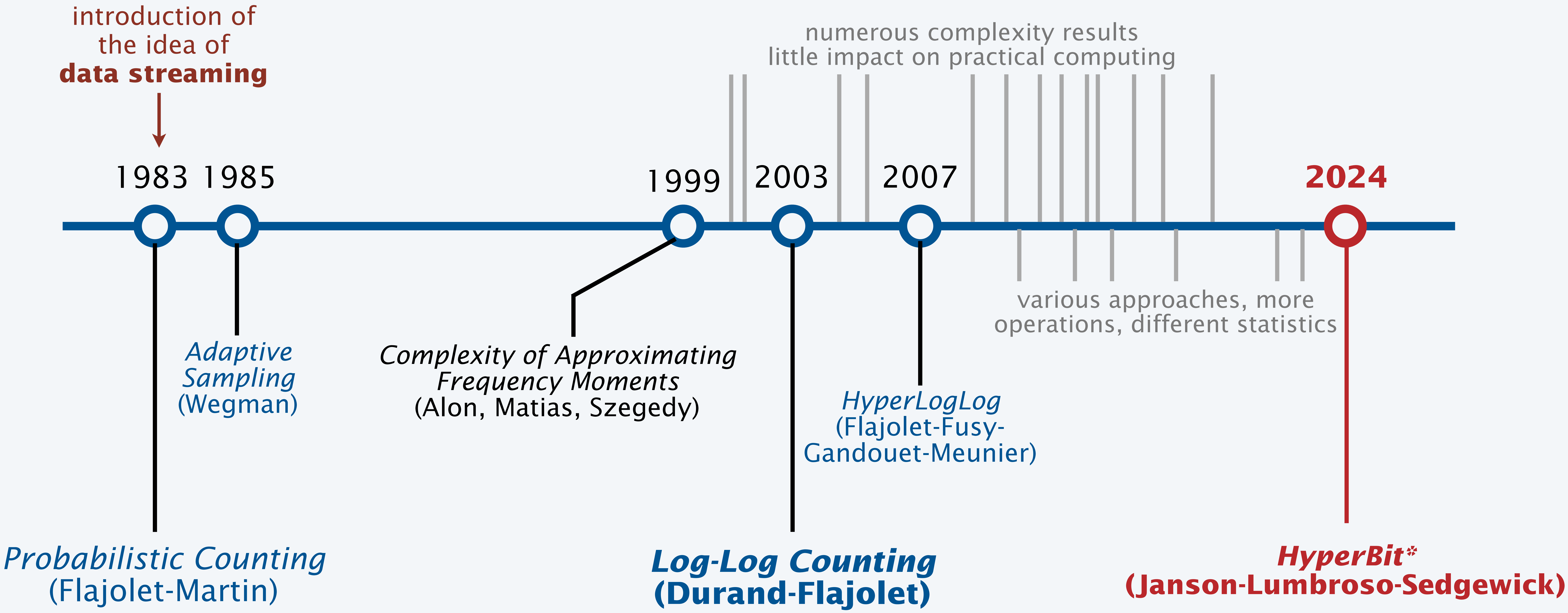
- How many unique visitors to my website?
- How many different cars passed here this year?
- How many different IP addresses hit this node?
- How many different values for a database join?

To fix ideas on scope (202x): Think of *billions* of streams each having *trillions* of values.

Q. How much memory is needed to estimate ***N*** to within, say, 10% accuracy?

A. Much less than you might think!

Timeline of milestones in cardinality estimation



For some details, see "*The Story of HyperLogLog: How Flajolet Processed Streams with Coin Flips*" J. Lumbroso, 2013.

Bit-array-based alternatives to HyperLogLog

- A fundamental problem in data science
- **Simple, elegant and efficient solutions**
- A simple algorithm
- HyperBitBit
- Memory vs. accuracy comparisons

Simple, elegant, and efficient solutions

1983 — Flajolet and Martin, *Probabilistic Counting Algorithms for Data Base Applications*

2003 — Durand and Flajolet, *LogLog Counting of Large Cardinalities*

2007 — Flajolet, Fusy, Gandouet, and Meunier, *HyperLogLog: Analysis of a near-optimal cardinality estimation algorithm*

Key steps

- **Hash** each item so as to work with "random" values.
- Develop a **sketch** that enables cardinality estimation.
- **Split** stream into ***M*** substreams and record their estimates.
- Average the estimates and precisely **analyze** the bias.

Probabilistic counting sketches are ***M*** ***64-bit*** values. ← 21st century value

LogLog algorithm sketches are M 8-bit values. ← packing/unpacking 6-bit values generally not worth the trouble

First step: Hash the values

Transform value to a "random" computer word.

- Compute a *hash function* that transforms data value into a 32- or 64-bit value.
- Cardinality count is unaffected (with high probability).
- Built-in capability in modern systems.
- *Allows use of fast machine-code operations.*

20th century: use 32 bits (millions of values)
21st century: use 64 bits (quintillions of values)

State-of-the-art-"Mersenne twister" uses only a few machine-code instructions.

Bottom line: Do cardinality estimation on streams of (binary) integers, not arbitrary value types.

```
01111000100111110111000111001000
01111000100111110111000111001000
01110101010110110000000011011010
00110100010001111100010100111010
00010000111001101000111010010011
00001001011011100000010010010111
00001001011011100000010010010111
00111000101001001011010101001100
00111000101001001011010101001100
01101001001000011100110100110011
00001000011101100110110010100101
```

"Random" *except* for the fact that some values are equal.

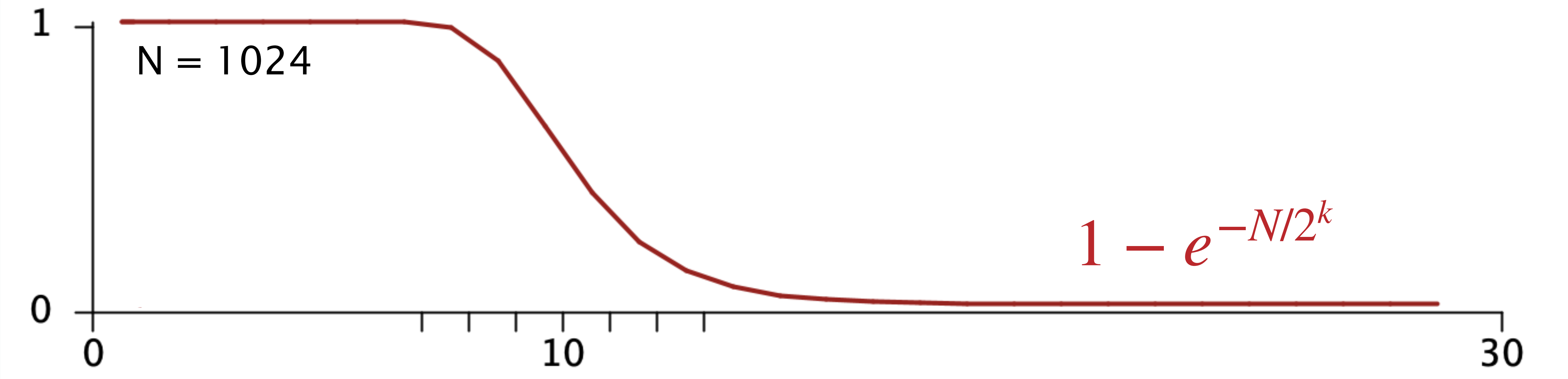
Second step: Focus on the trailing 1s

Let **X** be the max number of trailing 1s in a random stream of random distinct binary values.

Pr { no value has k trailing 1s } $= \left(1 - \frac{1}{2^k}\right)^N \sim e^{-N/2^k}$ = Pr { **X** <= k }

Pr { **X** > k } $\sim 1 - e^{-N/2^k}$ ← ~1 when k is small, ~0 when k is large

E(X) $\sim \sum_{k \geq 0} \left(1 - e^{-N/2^k}\right)$



$\sum_{k \geq 0} (1 - e^{-N/2^k}) =$ 1+1+1+1+1+1+1+1+1+1+1 + . . . + 0+0+0+0+0+0+0+0+0+...

~lg N terms are ~1 — a few are not close to 0 or 1 — the rest are all ~0

1111001111111110010...
1111000101001111010...
0111001101001100011...
0111001101001100011...
0111001101001100011...
0110000111010011101...
0111001101001100011 ..
1100000000011011010...
0111001101001100011...
0111001101001100011...
0010011100101000000...
1111000101001111010...
1111010101101100001...
0001110001110010000...
0001110001110010000...
1100000000011011010...
1111000101001111010...
0110001110100100011...
1000000100100101111...
1000000100100101111...
0010110101010011100...

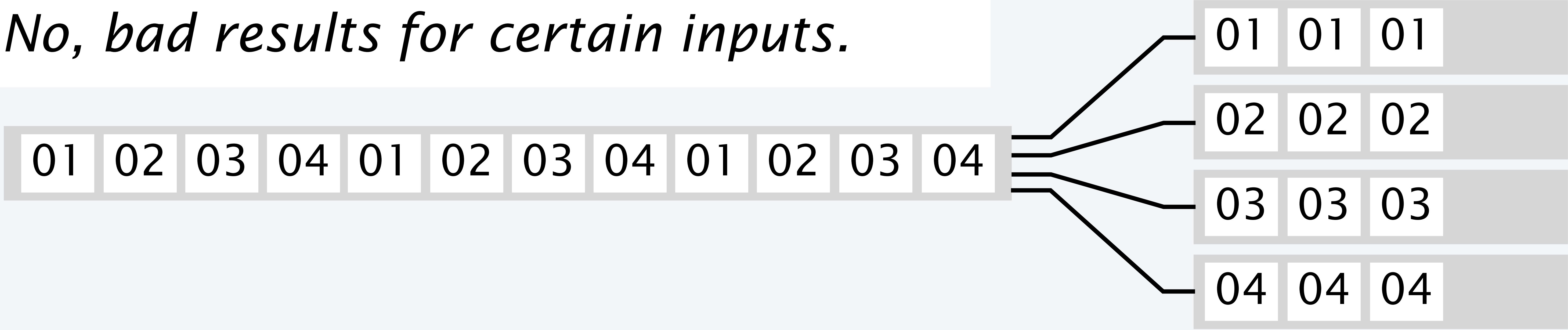
Takeaway. **E(X)** is slightly larger than lg N

Third step: stochastic splitting

Goal: Perform M independent experiments.

Alternative 1: M independent hash functions? *No, too expensive.*

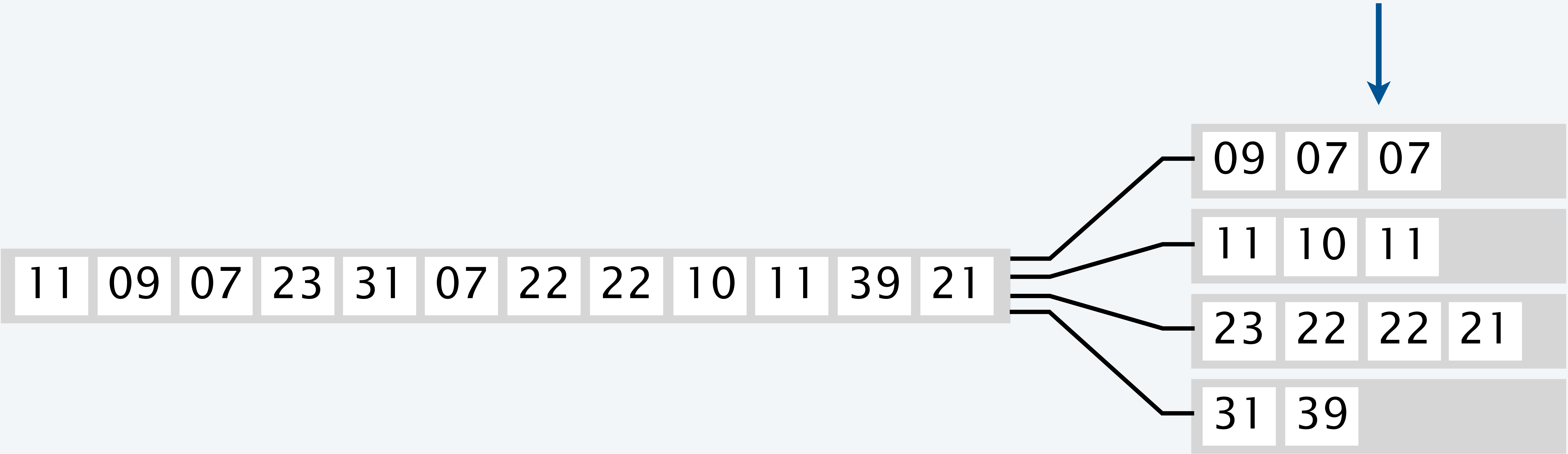
Alternative 2: M-way alternation? *No, bad results for certain inputs.*



Alternative 3 (Flajolet-Martin): **Stochastic splitting.**

Use second hash to divide stream into 2^m independent streams

key point: equal values all go to the same stream



Fourth step: average and analyze

LogLog: Use **arithmetic mean** of max # trailing 1s in the substreams.

bias: $e^{-\gamma}\sqrt{2} \doteq .794028$

std error: $\sim c_M/\sqrt{M}$ where $c_M \sim \sqrt{(\ln 2)^2/12 + \pi^2/6} \approx 1.30$

memory: ***8M*** bits (*M* numbers, each about $\lg N$ and stored in an 8-bit byte)

HyperLogLog: Use **geometric mean** of max # trailing 1s in the substreams.

bias: $\dfrac{1}{2\ln 2} \doteq .72134$

std error: $\sim \beta_\infty/\sqrt{M}$ where $\beta_\infty \sim \sqrt{3\ln 2 - 1} \approx 1.04$

memory: ***8M*** bits (*M* numbers, each about $\lg N$ and stored in an 8-bit byte)

Goal: Optimal use of memory

HyperLogLog

bias: $\dfrac{1}{2\ln 2} \doteq .72134$

std error: $\sim \beta_\infty/\sqrt{M}$ where $\beta_\infty \sim \sqrt{3\ln 2 - 1} \approx 1.04$

memory: ***8M*** bits (*M* numbers, each about $\lg N$ and stored in an 8-bit byte)

HyperBit?

bias: **???**

std error: **???**

memory: ***M*** bits (one bit per stream)

Bit-array-based alternatives to HyperLogLog

- A fundamental problem in data science
- Simple, elegant and efficient solutions
- **A simple algorithm**
- HyperBitBit
- Memory vs. accuracy comparisons

A simple algorithm: HyperBitT

Idea: Start with a rough estimate **T** of lg(N/M)
Compute fraction **beta** of M substreams with no values having > T trailing 1s.
Use 2^T to estimate N/M, modified by the bias factor ln(1/beta) (proof to follow).

```
public static long
estimate(Iterable<String> stream, int M, int T)
{
   bit[] sketch = new bit[M];
   for (String s : stream)
   {
      long x = hash1(s);       // 64-bit hash
      int  k = hash2(s, M);    // (lg M)-bit hash
      if (r(x) >= T) sketch[k] = 1;
   }
   double beta = 1.0 - 1.0*p(sketch)/M;
   return (long) (Math.pow(2, T)*M*Math.log(1/beta));
}
```

Details.

- **M** is the number of substreams
- **T** is an estimate of lg(N/M)
- **sketch** is an **M**-bit array (initialized to all 0s)
- **r(x)** is # of trailing 1s in x
- **p(x)** is # of 1s in x
- **beta** is fraction of 0s in sketch

Effective only when T is not large or small (stay tuned).

Notes.

- no bit array in Java, use shift/mask in arrays of integers
- **r(x)>=T** is easily computed
- **p()** computation is easily avoided

HyperBitT mean value analysis (elementary)

If there are $M\beta$ 0s in the sketch, what is the expected number of values that have been processed?

In a data stream with ***v*** distinct values

- Pr {*a given value has at least **T** trailing 1s*} $= 1/2^T$
- Pr {*no item has at least **T** trailing 1s*} $= \left(1 - \dfrac{1}{2^T}\right)^v \sim e^{-v/2^T}$

corresponding bit in sketch is 0

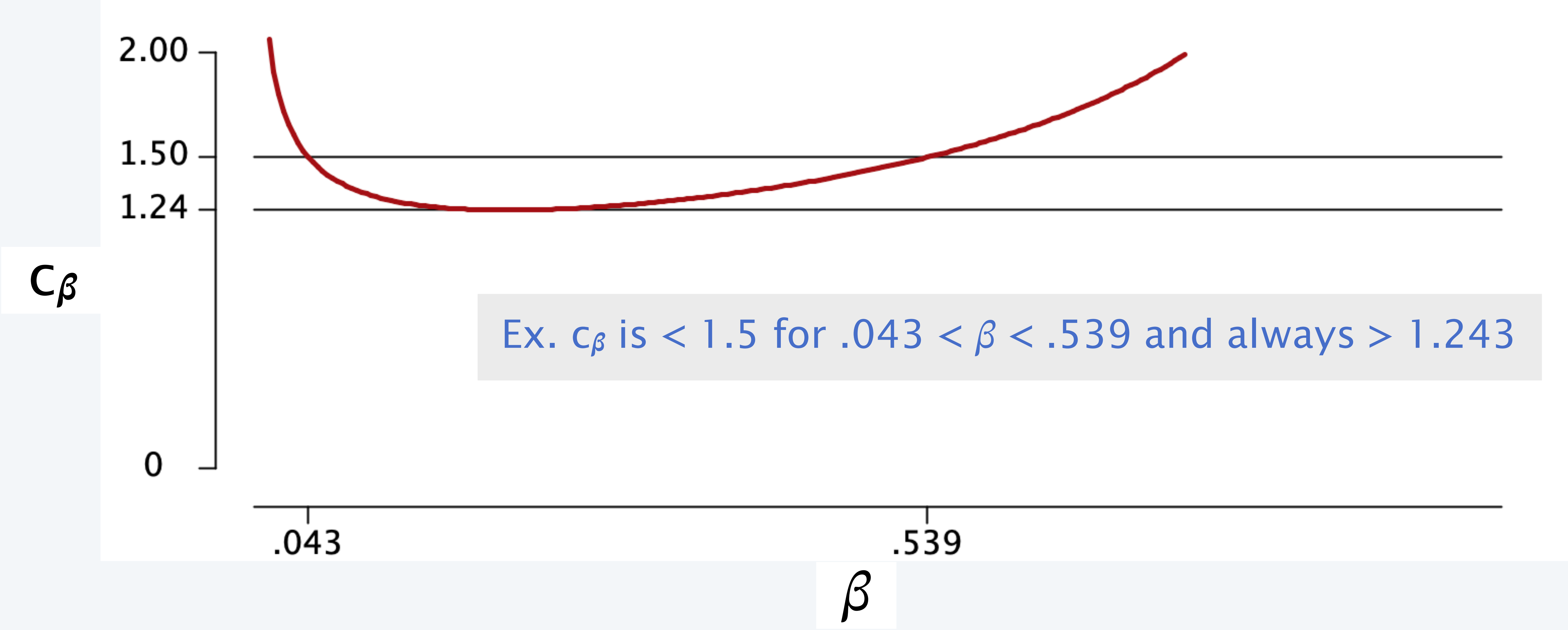
After ***Mv*** distinct values (approximately ***v*** per stream) have been processed

- distribution of # of 0s in sketch is binomial $B(M, e^{-v/2^T})$
- expected number of 0s in sketch is $\sim Me^{-v/2^T}$
- Algorithm terminates with $Me^{-v/2^T} = M\beta$, or $v = 2^T \ln(1/\beta)$

Theorem. Expected number of values processed is $\sim Mv = M \cdot 2^T \cdot \ln(1/\beta)$

HyperBitT distribution analysis (Janson)

Theorem. The distribution of the number of values processed is ***asymptotically normal*** with mean $\overline{N} = M \cdot 2^T \cdot \ln(1/\beta)$ and standard error $\dfrac{\sqrt{1/\beta - 1}}{\sqrt{M} \cdot \ln(1/\beta)} = \dfrac{c_\beta}{\sqrt{M}}$ where $c_\beta \equiv \dfrac{\sqrt{1/\beta - 1}}{\ln 1/\beta}$.



Ex. c_β is < 1.5 for .043 < β < .539 and always > 1.243

Range of reasonable accuracy depends on M and T but is quite large

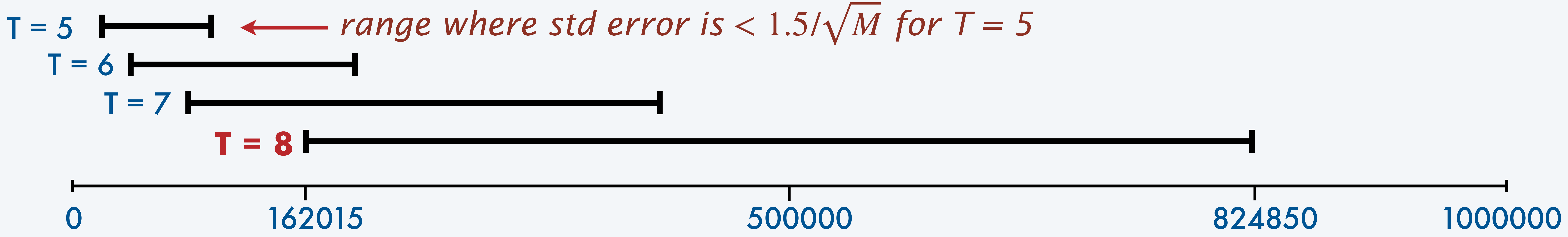
Expected number of values processed is $M \cdot 2^T \cdot \ln(1/\beta)$

"Reasonable accuracy": Standard error is less than $1.5/\sqrt{M}$ (when $.043 < \beta < .539$)

Ex. ***M*** = 1024

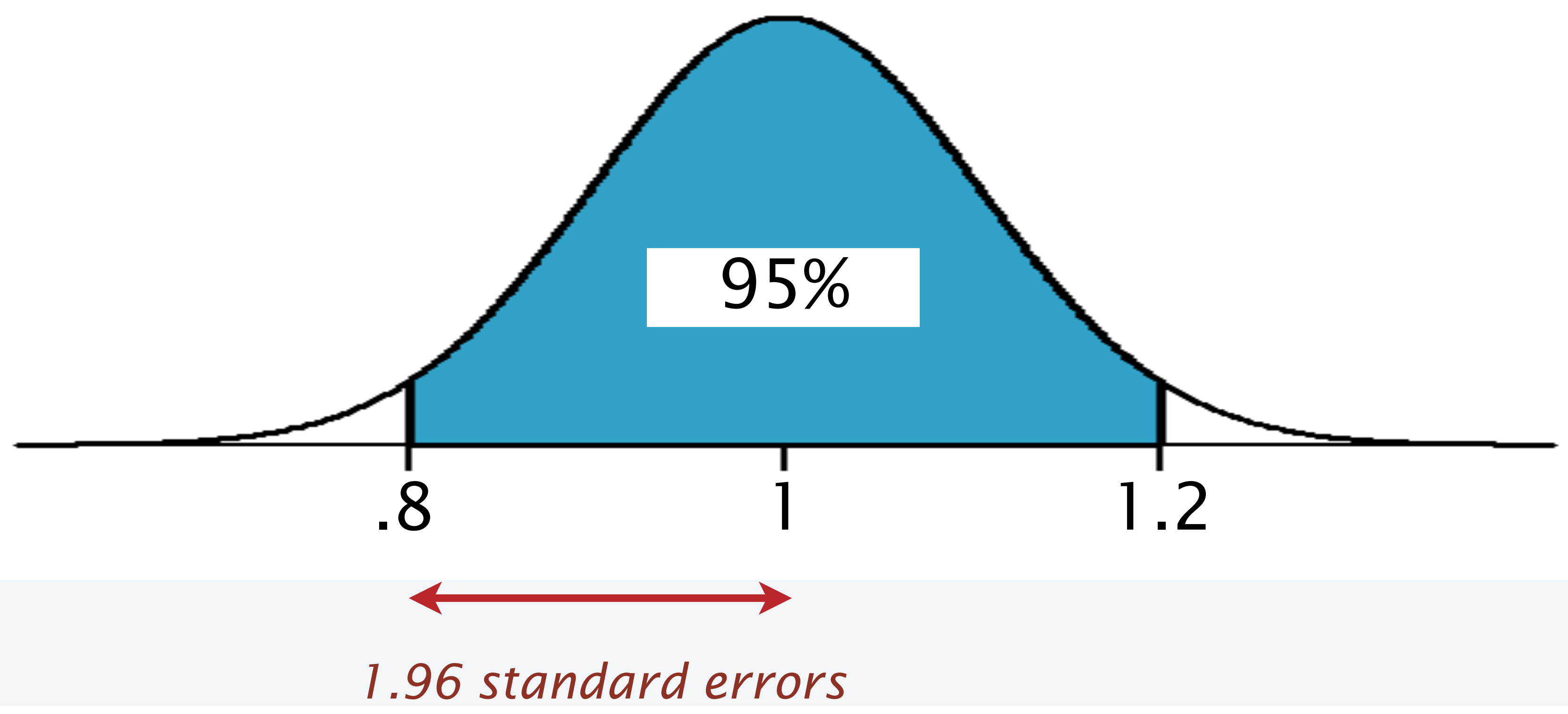
$M \cdot 2^T \cdot \ln(1/\beta)$

T	2^T	$\beta = .539$	$\beta = .043$
5	32	20251	103106
6	64	40503	206212
7	128	81007	412425
8	256	162015	824850



Application example

How many different values in my web log (1 million entries)?



Q. How accurate an answer do you want?

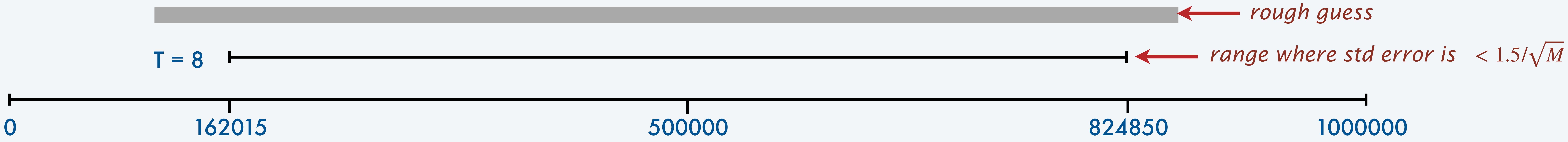
A. 95% sure to be within 10%.

Recomendation 1. Take M = 1024 to get standard error $\frac{1.5}{32}$ and 95% sure to be within $1.96 \cdot \frac{1.5}{32} < 10\,\%$

Q. What's your rough guess?

A. Somewhere between 100,000 and 900,000.

Recommendation 2. Take T = 8 for result to be valid unless it is much smaller or larger than that.



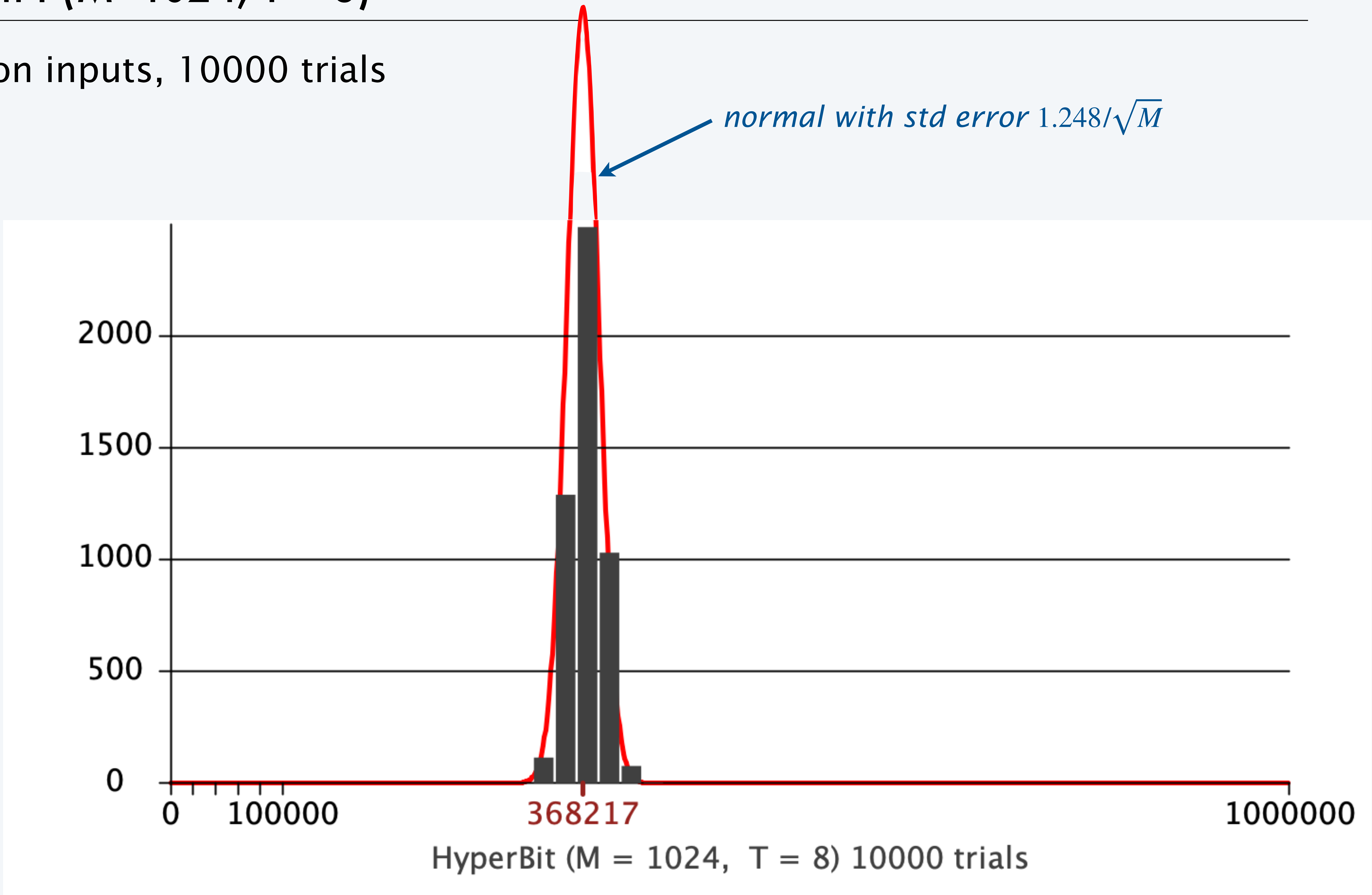
HyperBitT validation I (M=1024, T = 8)

Experiment. 1million inputs, 10000 trials

normal with std error $1.248/\sqrt{M}$

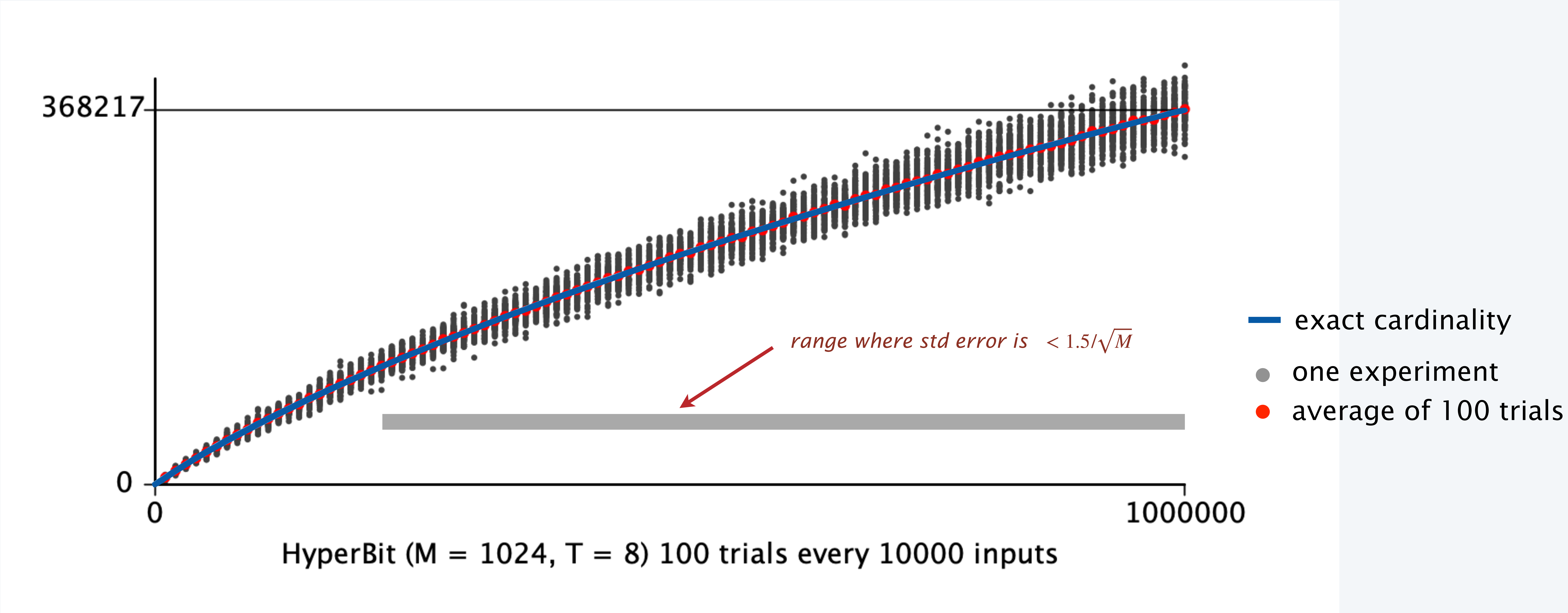
50-interval histogram

height of each bar is # of estimates in its interval



HyperbitT validation II (M=1024, T = 8)

Experiment. 100 trials for x*10000 inputs for x from 1 to 100 (10000 trials) with $M = 1024$



Bit-array-based alternatives to HyperLogLog

- A fundamental problem in data science
- Simple, elegant and efficient solutions
- A simple algorithm
- **HyperBitBit**
- Memory vs. accuracy comparisons

HyperBitBit

Unfortunate truth. While very useful in many contexts, HyperBitT is NOT a streaming algorithm.

Goal. Eliminate need to provide rough estimate of cardinality.

Simple idea (HyperBitBit, RS, 2015): Make T a variable and increment as needed.

- Start at T=1
- Maintain a second sketch for T+1.
- When sketch is half full, increment T
- Then set sketch1 = sketch2 and set sketch2 to 0.
- Try to estimate the error inherent in resetting sketch2 to 0.

Questions.

- Why half full?
- Why T+1?
- What's the bias?
- What's the standard error?

Good news. HyperBitT analysis provides proper settings and the answers to these questions.

HyperBitBit

Idea: Keep track of sketches for T and T+**4**. When sketch for T fills, increment T by 4 and update sketches.

```
public static long estimate(Iterable<String> stream)
{
   bit[] sketch1 = new bit[M];
   bit[] sketch2 = new bit[M];
  for (String s : stream)
   {
      long x = hash1(s);       // 64-bit hash
      int  k = hash2(s, M);    // (lg M)-bit hash
      if (r(x) >= T  ) sketch1[k] = 1;
      if (r(x) >= T+4) sketch2[k] = 1;
     if (p(sketch1) > .988*M)
      {
         T = T+4;
         sketch1 = sketch2;
         sketch2 = new bit[M];
      }
   }
   double beta = 1.0 - 1.0*p(sketch1)/M;
   return (long) (Math.pow(2, T)*M*Math.log(1/beta));
}
```

Key questions: Why 4? Why .988? Why $\ln(1/\beta)$?

Details.

- **T** is an estimate of lg(N/M)
- **sketch1/2**are bit arrays (initialized to all 0s)
- **r(x)** is # of trailing 1s in x
- **p(x)** is # of 1s in x
- **beta** is fraction of 0s in sketch
- Correct at end with bias factor *(a function of beta)*

Notes.

- sketch for **T+8** is likely all 0s
- **r(x)>=T** is easily computed
- **p()** computation is easily avoided

Parameter values for HyperBitBit

Q1. How much to increment ***T*** ?

$$M2^T \ln(1/\beta) = M2^{T+i} \ln(1/\beta_i)$$

estimated cardinality

fraction of 0s in sketch for T

fraction of 0s in sketch for T+i

$$\beta_i = \exp\left(-\ln(1/\beta)/2^i\right)$$

i	0	1	2	3	4	5	6	7	8
β_i	**.03**	.17	.42	.64	**.80**	.90	.95	.97	**.99**

A1. With T+4, sketch for T+8 would be nearly all 0s even when sketch for T is 97% 1s.

HyperBitT analysis applies throughout.

Q2. When to increment **T** ?

A2. When std error for T+4 equals std error for T — (do the math) — when sketch is 98.8% full.

Q3. Reported cardinality count ?

A3. $M2^T \ln(1/\beta)$.

Q4. Relative std error ?

A4. About $1.46/\sqrt{M}$, on average.

Bit-array-based alternatives to HyperLogLog

- A fundamental problem in data science
- Simple, elegant and efficient solutions
- A simple algorithm
- HyperBitBit
- Memory vs. accuracy comparisons

Algorithm comparisons: memory vs. accuracy

Q. Given M^* bits of memory, how do the algorithms compare??

A. Using b bits per item, the number of streams is M^*/b and the relative error is $\dfrac{c}{\sqrt{M^*/b}} = \dfrac{c\sqrt{b}}{\sqrt{M^*}}$

	memory use	*b*	*c*	*std error*
HyperLogLog	M bytes	8	1.05	$\dfrac{2.35}{\sqrt{M^*}}$
HyperBitT	M bits	1	1.32	$\dfrac{1.32}{\sqrt{M^*}}$
HyperBitBit	$2M$ bits	2	1.46	$\dfrac{2.06}{\sqrt{M^*}}$

HyperBitBitBit and HyperTwoBits

HyperBitBit Drawback. Too many nonzero bits in T+8 sketch for large M.

Fix. HyperBitBitBit.

HyperBitBitBit Drawback. Uses $3M$ bits.

Fix. Use array of 2-bit values (#1s in corresponding position in sketches) instead.

Ex. ($M = 64$, increment = 4)

	*sketch for **T***	1111111111011101111111111111111111111101110111111001111111111111
before	*sketch for **T+4***	0001001110100000000000001000011001011000000111100001000000000000
	*sketch for **T+8***	0000000100000000000000000000000000010000000011000010000000000000
	two-bit value	1112112321201110111111121111221121231011102233110031111111111111
	*sketch for **T***	0001001110100000000000001000011001011000000111100001000000000000
*after **T+=4***	*sketch for **T+4***	0000000100000000000000000000000000010000000011000010000000000000
	*sketch for **T+8***	0000000000000000000000000000000000000000000000000000000000000000
	two-bit value	0001001210100000000000001010011001012000000112200002000000000000

Exercise in hacking. Implement with code on real machines (see appendix in paper)

Algorithm comparisons: memory vs. accuracy

Q1. Given M^* bits, what accuracy is expected?

A1. With *b* bits per item, accuracy is $c/\sqrt{M^*/b}$.

	b	*c*	*M**=128	*M**=8K
HyperLogLog	8	1.05	26%	3.3%
HyperBitT	1	1.32	12%	1.5%
HyperTwoBits	2	1.46	18%	2.3%

accuracy $x = c/\sqrt{M^*/b}$ *when using M* bits*

Q2. How many bits to achieve a given accuracy x ?

A2. Solve $x = c/\sqrt{M^*/b}$ for M^* to get $M^* = b(c/x)^2$.

	b	*c*	*x* = 2%	*x* = 20%
HyperLogLog	8	1.05	21632	216
HyperBitT	1	1.32	4356	44
HyperTwoBits	2	1.46	10658	106

memory $M^* = b(c/x)^2$ *for accuracy within* $1 \pm x$

Still open: HyperBit

```
public static long
estimate(Iterable<String> stream, int M)
{
   int T = 1;
   double beta;
   bit[] sketch = new bit[M];
   for (String s : stream)
   {
      long x = hash1(s);      // 64-bit hash
      int  k = hash2(s, M);   // (lg M)-bit hash
      if (r(x) >= T) sketch[k] = 1;
      beta = 1.0 - 1.0*p(sketch)/M;
      if (beta > THRESHHOLD)
      {
         T += INCREMENT;
         sketch = new sketch[];
      }
   }
  return (int) (Math.pow(2, T)*M*BIAS);
}
```

Details.

- **M** is the number of substreams
- **T** is an estimate of lg(N/M)
- **sketch** is an **M**-bit array (initialized to all 0s)

some choices test well empirically

Open: Analysis proving values for threshhold, increment, and bias

This work is dedicated to the memory of Philippe Flajolet

Philippe Flajolet 1948-2011

Bit-array-based alternatives to HyperLogLog

Svante Janson
Uppsala University

Jérémie Lumbroso
University of Pennsylvania

Robert Sedgewick
Princeton University