HyperBit

A Memory-Efficient Alternative to HyperLogLog

[work in progress]

Robert Sedgewick
Princeton University

with thanks to Jérémie Lumbroso and Svante Janson

Philippe Flajolet, mathematician and computer scientist extraordinaire

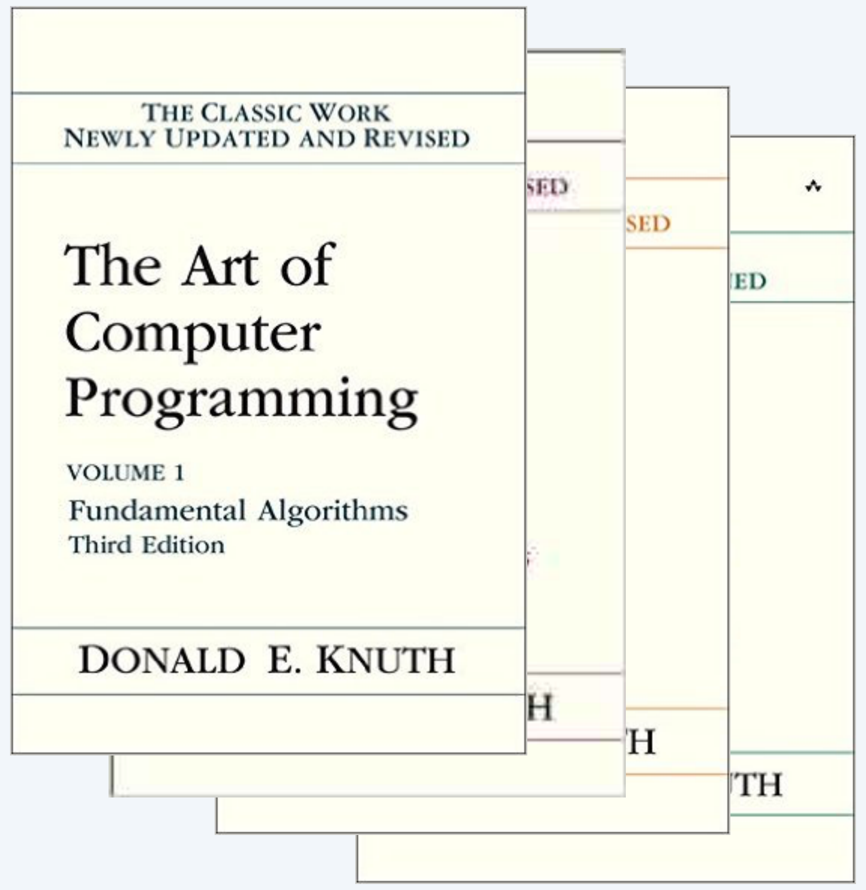
Philippe Flajolet 1948-2011

Algorithm science (Knuth, 1960s–present; Sedgewick, 1980s–present)

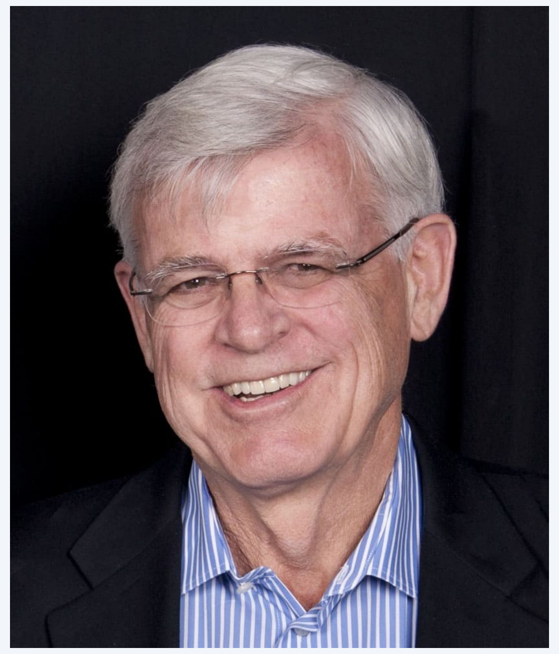
aka "analysis of algorithms"

A ***scientific*** basis for studying algorithms

- Implement and run on realistic inputs
  [Is the algorithm effective in the real world?].
- Develop a mathematical model
- Use model to formulate hypotheses on performance
- Test hypotheses with real-world experiments
- **Iterate**



Knuth



Sedgewick

BENEFIT: Enabled creation of our software infrastructure.

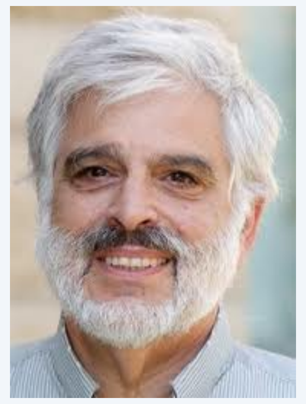
DRAWBACKS: Model can be unavailable, unrealistic, or excessively detailed and complicated.
Mathematical analysis can be prohibitively challenging.

Theory of Algorithms (AHU, 1970s; CLRS, 1980s–present)

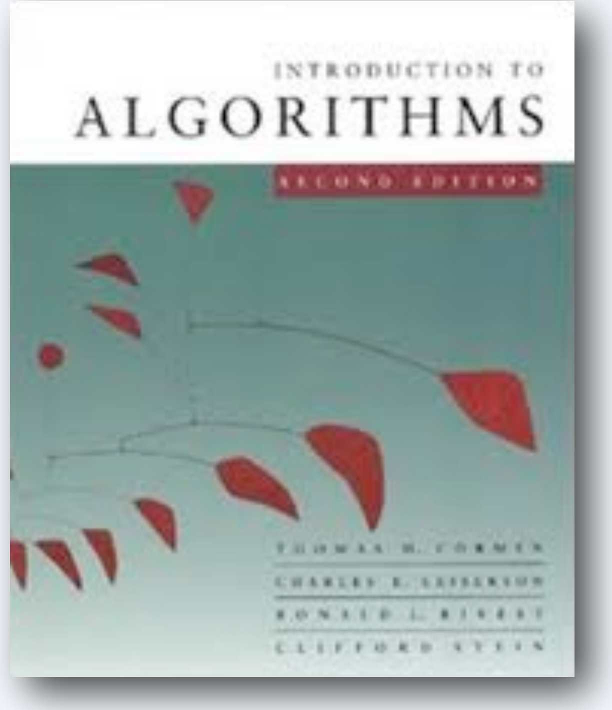
A mathematical basis for studying algorithms

- Analyze worst-case cost
  [takes model out of the picture].
- Use O-notation for upper bounds
  [takes detail out of analysis].
- Classify algorithms by these costs.

Aho, Hopcroft and Ullman



Cormen, Leiserson, Rivest, and Stein



BENEFIT: Enabled a new Age of (Theoretical) Algorithm Design.

DRAWBACKS: Analysis is typically *unsuitable* for scientific studies.
Algorithms are often *not useful* in the real world.

(Elementary facts that are often overlooked!)

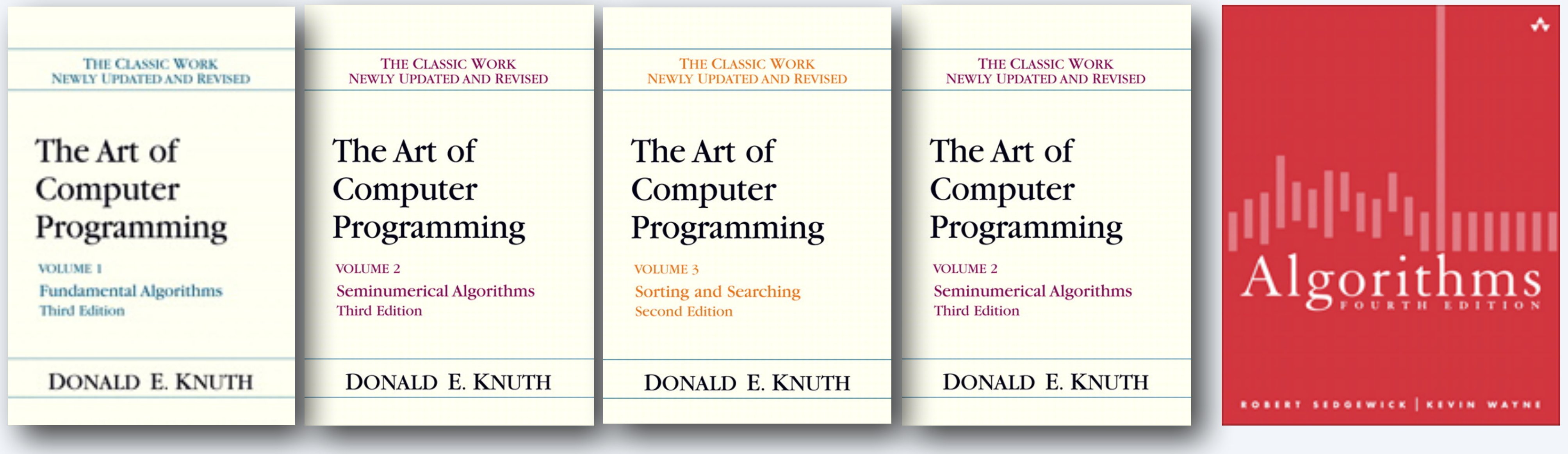
Analytic combinatorics context

Drawbacks of AHU/CLRS approach:

- Worst-case performance may not be relevant.
- Cannot use O- upper bounds to predict or compare.

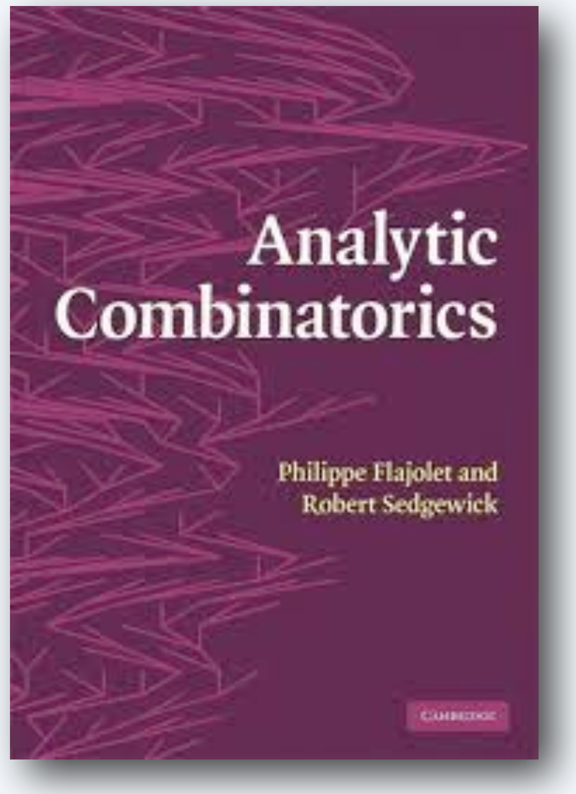
Drawbacks of Knuth/Sedgewick approach:

- Model may be unrealistic.
- Analysis may be detailed and difficult.



Analytic combinatorics can provide a basis for scientific studies.

- A calculus for developing models.
- Universal laws that encompass the detail in the analysis.
- *Applies to many sciences, not just algorithm science.*



Hyperbit: A Memory-Efficient Alternative to HyperLogLog

- **The problem**
- A solution
- Another approach
- Final frontier

Cardinality counting

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.

This talk. Estimate ***N*** to within 10% accuracy 99% of the time using thousands of bits of memory.

Posted on Facebook, 2018

"Computing the count of distinct elements in massive data sets is often necessary but computationally intensive.

Say you need to determine the number of distinct people visiting Facebook in the past week using a single machine.

*With a traditional SQL query on the Facebook data sets this would take **days and terabytes of memory**. "*

Hyperbit: A Memory-Efficient Alternative to HyperLogLog

- The problem
- **A solution**
- Another approach
- Final frontier

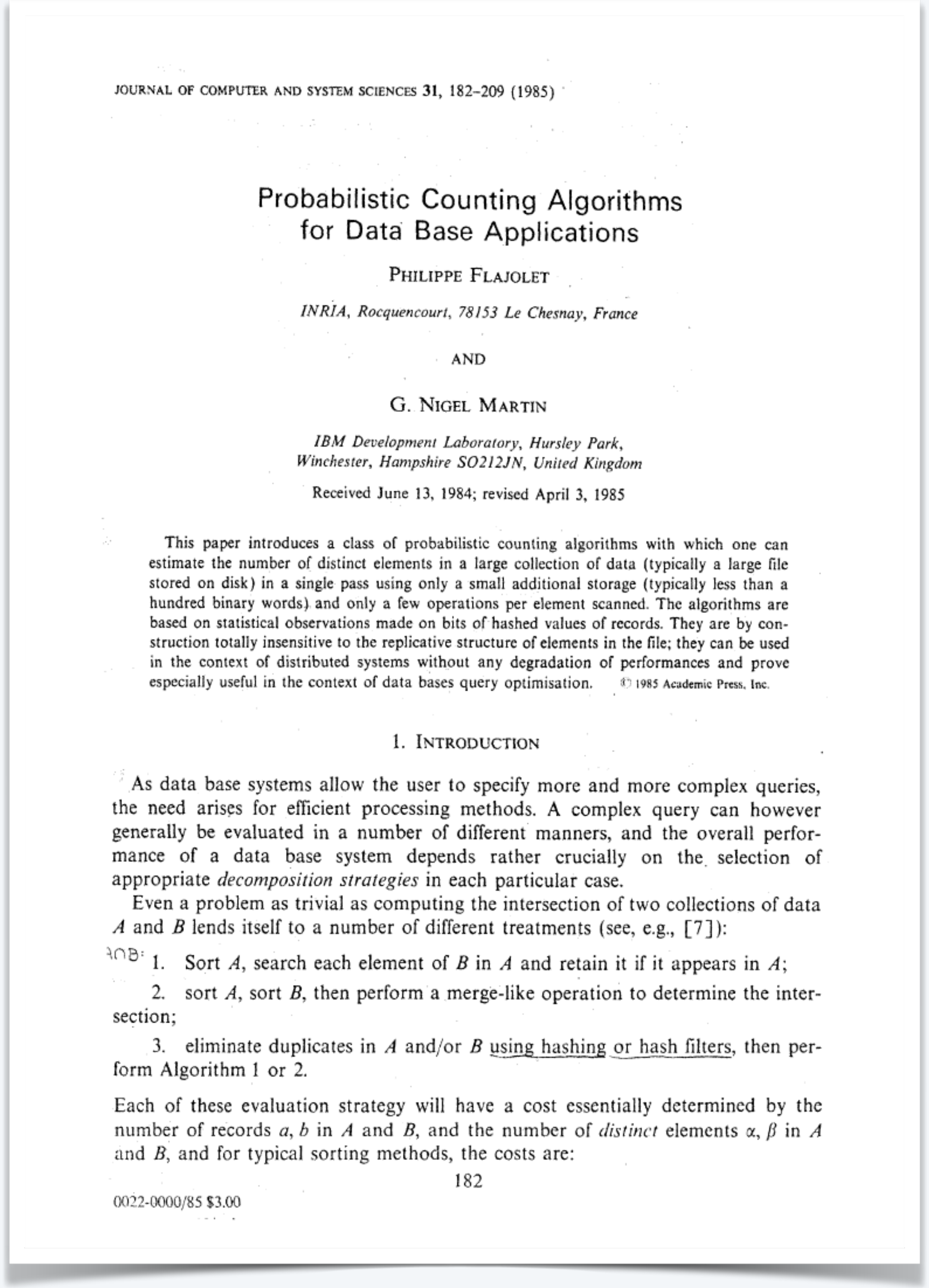
Probabilistic counting with stochastic averaging (PCSA)

Flajolet and Martin, *Probabilistic Counting Algorithms for Data Base Applications FOCS* 1983, *JCSS* 1985.

Philippe Flajolet 1948-2011

Contributions

- Introduced problem
- Idea of *streaming algorithm*
- Idea of “small” *sketch* of “big” data
- Detailed analysis that yields tight bounds on accuracy
- Full validation of mathematical results with experimentation
- Practical algorithm that has remained effective for decades



JOURNAL OF COMPUTER AND SYSTEM SCIENCES **31**, 182–209 (1985)

Probabilistic Counting Algorithms for Data Base Applications

PHILIPPE FLAJOLET

INRIA, Rocquencourt, 78153 Le Chesnay, France

AND

G. NIGEL MARTIN

IBM Development Laboratory, Hursley Park, Winchester, Hampshire SO212JN, United Kingdom

Received June 13, 1984; revised April 3, 1985

This paper introduces a class of probabilistic counting algorithms with which one can estimate the number of distinct elements in a large collection of data (typically a large file stored on disk) in a single pass using only a small additional storage (typically less than a hundred binary words) and only a few operations per element scanned. The algorithms are based on statistical observations made on bits of hashed values of records. They are by construction totally insensitive to the replicative structure of elements in the file; they can be used in the context of distributed systems without any degradation of performances and prove especially useful in the context of data bases query optimisation. © 1985 Academic Press, Inc.

1. INTRODUCTION

As data base systems allow the user to specify more and more complex queries, the need arises for efficient processing methods. A complex query can however generally be evaluated in a number of different manners, and the overall performance of a data base system depends rather crucially on the selection of appropriate *decomposition strategies* in each particular case.

Even a problem as trivial as computing the intersection of two collections of data A and B lends itself to a number of different treatments (see, e.g., [7]):

$A \cap B$: 1. Sort A, search each element of B in A and retain it if it appears in A;

2. sort A, sort B, then perform a merge-like operation to determine the intersection;

3. eliminate duplicates in A and/or B using hashing or hash filters, then perform Algorithm 1 or 2.

Each of these evaluation strategy will have a cost essentially determined by the number of records a, b in A and B, and the number of *distinct* elements α, β in A and B, and for typical sorting methods, the costs are:

182

0022-0000/85 $3.00

Bottom line.
Quintessential example of the effectiveness of algorithm science and analytic combinatorics.

Starting point: three integer functions

Def. $r(x)$ is the **number of *trailing* 1s** in the binary representation of x.

position of rightmost 0

Def. $R(x) = 2^{r(x)}$

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	$r(x)$	$R(x)$	$R(x)_2$
1	0	1	1	1	1	0	1	1	1	1	1	0	1	0	1	**1**	**2**	**1 0**
1	0	1	0	1	0	1	0	1	0	0	0	1	1	1	0	**0**	**1**	**1**
0	1	1	0	1	0	0	1	0	1	0	**1**	**1**	**1**	**1**	**1**	**5**	**32**	**1 0 0 0 0 0**

Bit-whacking magic:
$R(x)$ is easy to compute.

0	**1**	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**1**	**1**	**1**	**1**	**1**	x
1	0	0	1	0	1	1	0	1	0	1	0	0	0	0	0	~x
0	1	1	0	1	0	0	1	0	1	1	0	0	0	0	0	x + 1
0	0	0	0	0	0	0	0	0	0	**1**	**0**	**0**	**0**	**0**	**0**	~x & (x + 1)

3 instructions on a typical computer

Def. $p(x)$ is the **number of 1s** in the binary representation of x.

for bit-whacking magic for this and r(x) see Knuth volume 4A

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 instruictions.

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.

Initial hypothesis

Fact. Hash values are *not* random.

Hypothesis. Hash values are "sufficiently" random.

Implication. Need to run experiments to validate any hypotheses about performance.

No problem!

- We *always* validate hypotheses in algorithm science.
- End goal is development of algorithms that are useful in practice.
- It is the responsibility of the *designer* to validate utility before claiming it.
- After decades of experience, discovering a performance problem due to a bad hash function would be a significant research result.

Unspoken bedrock principle of algorithm science.

Experimenting to validate hypotheses is **WHAT WE DO!**

Probabilistic counting (Flajolet and Martin, 1983)

Maintain a single-word *sketch* that summarizes a data stream x_0, x_1, …, x_i, …

- For each x_i in the stream, update sketch by *bitwise or* with **R(xi)** [$2^{r(x_i)}$] .
- Use **`r(sketch)`** [*number of trailing 1s in the sketch*] to estimate lg N_i
- Equivalently, use **`R(sketch)`**[$2^{r(sketch)}$] to estimate N_i
- Refine with a correction factor, informed by analysis

estimate of lg N_i

typical sketch
N_i ~ 10^6

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sketch	0	0	0	0	0	0	0	1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
x_i	0	0	1	1	0	1	0	1	0	1	1	1	1	1	1	0	1	0	1	0	1	0	1	0	1	0	0	0	1	1	1	1
R(x_i)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
sketch | R(x_i)	0	0	0	0	0	0	0	1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

$r(x_i) = 4$

$R(x_i) = 16$

no change (usually)

leading bits almost surely 0

trailing bits almost surely 1

estimate of N ⟶ 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 $= 2^{20} \approx 10^6$

Example Probablilistic Counting actions (32-bit values)

x	R(x)	sketch
00110010100000000110011110111111	1000000	00000000000000000011001111111111
		1000000
no change with high probability		00000000000000000011001111111111
00110010100000000111111111111111	1000000000000000	00000000000000000110011111111111
		1000000000000000
no change with low probability		00000000000000000110011111111111
00110010100000001111111111111111	10000000000000000	00000000000000000110011111111111
		1000000000000000
sketch changes but not r(sketch)		00000000000000001110011111111111
00110010100000001000111111111111	100000000000	00000000000000000110011111111111
		100000000000
sketch changes and r(sketch) increases by 1		00000000000000000110111111111111
00110010100000001001111111111111	1000000000000	00000000000000000110111111111111
		1000000000000
sketch changes and r(sketch) increases by more than 1		00000000000000000111111111111111

estimate of N ⟶ 1000000000000000

Probabilistic counting (Flajolet and Martin, 1983)

```
public long R(long x)
{ return ~x & (x+1); }

public long estimate(Iterable<String> stream)
{
   long sketch;
   for (s : stream)
      sketch = sketch | R(hash(s));
   return R(sketch) /.77351;
}
```

Maintain a *sketch* of the data

- A single word
- OR of all values of R(hash(s))
- Return smallest value not seen

  with correction for bias

Early example of “a simple algorithm whose analysis isn’t”

Q. (Martin) Estimate seems a bit low. How much?

A. (unsatisfying) Obtain correction factor empirically.

A. (Flajolet) "*Without the analysis, there is no algorithm!*"

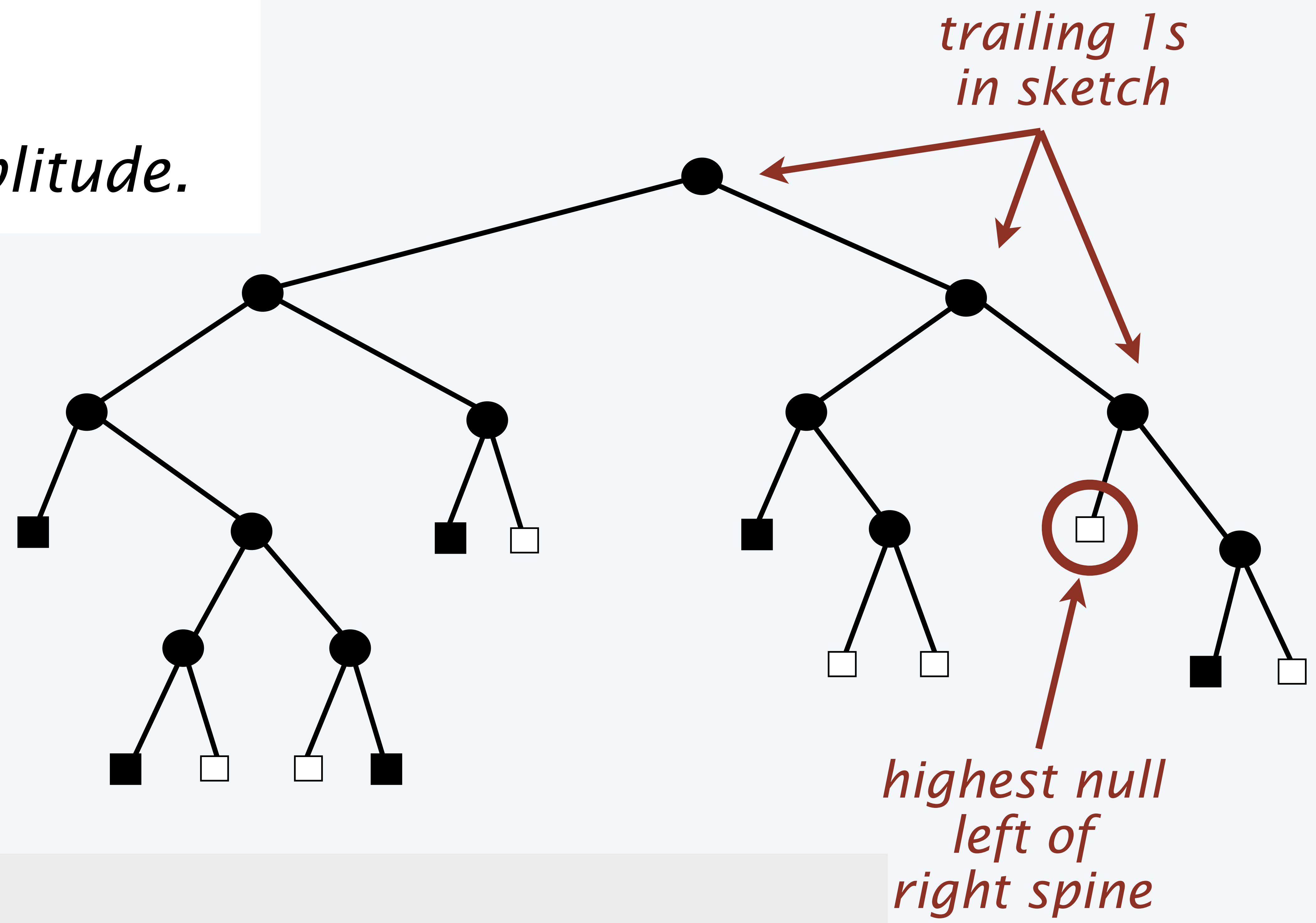
Magic is something you make

Mathematical analysis of probabilistic counting

Theorem. *The expected number of trailing 1s in the PC sketch is*

$$\lg(\phi N) + P(\lg N) + o(1) \quad \textit{where} \;\; \phi \doteq .77351$$

and P is an oscillating function of lg *N of very small amplitude.*



Proof (omitted).

1980s: Flajolet *tour de force*

1990s: trie parameter

21st century: standard analytic combiantorics

Kirschenhofer, Prodinger, and Szpankowski
Analysis of a splitting process arising in probabilistic counting and other related algorithms, ICALP 1992.

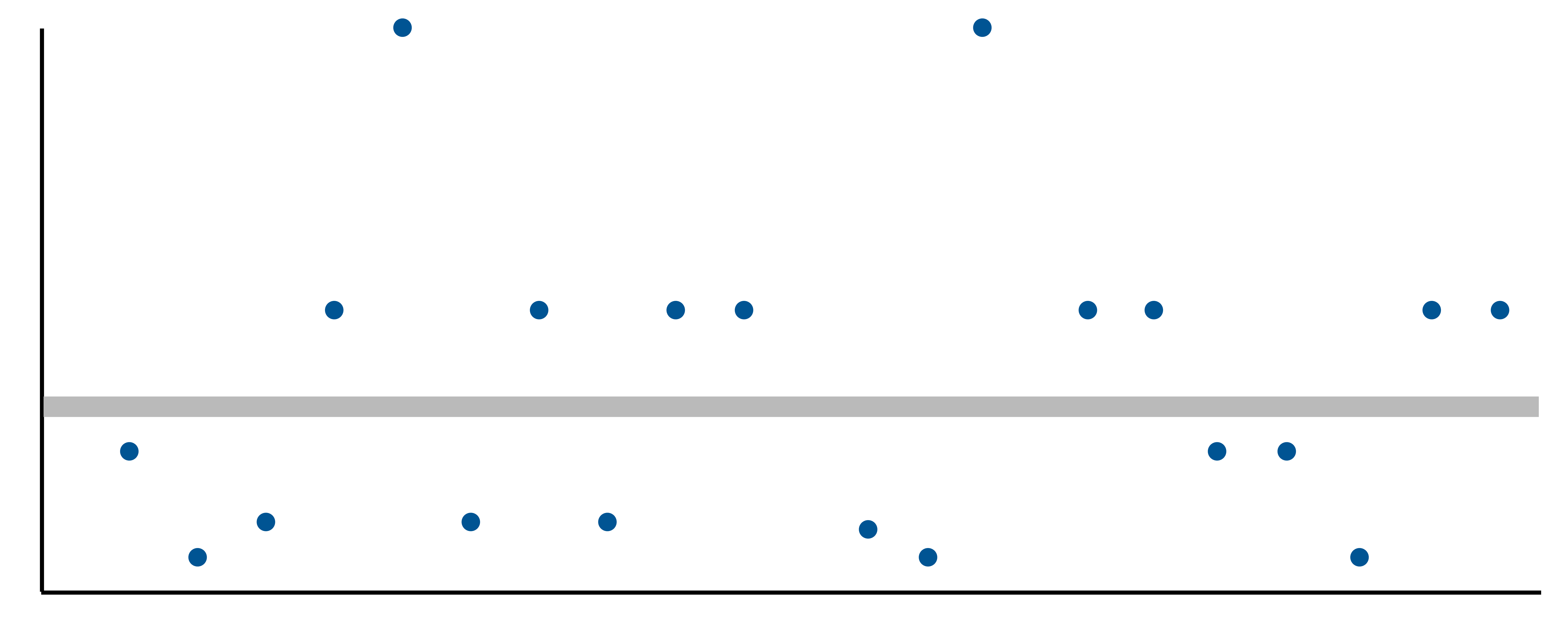
Jacquet and Szpankowski
Analytical depoissonization and its applications, TCS 1998.

In other words. In PC code, `R(sketch)/.77351` is an *unbiased statistical estimator* of *N*.

Validation of probabilistic counting

Hypothesis. Expected value returned is *N* *for random values from a large range.*

Quick experiment. 100,000 31-bit random values (20 trials)



Flajolet and Martin: Result is "typically one binary order of magnitude off."

Of course! (Always returns a power of 2 divided by .77351.)

Need to incorporate more experiments for more accuracy.

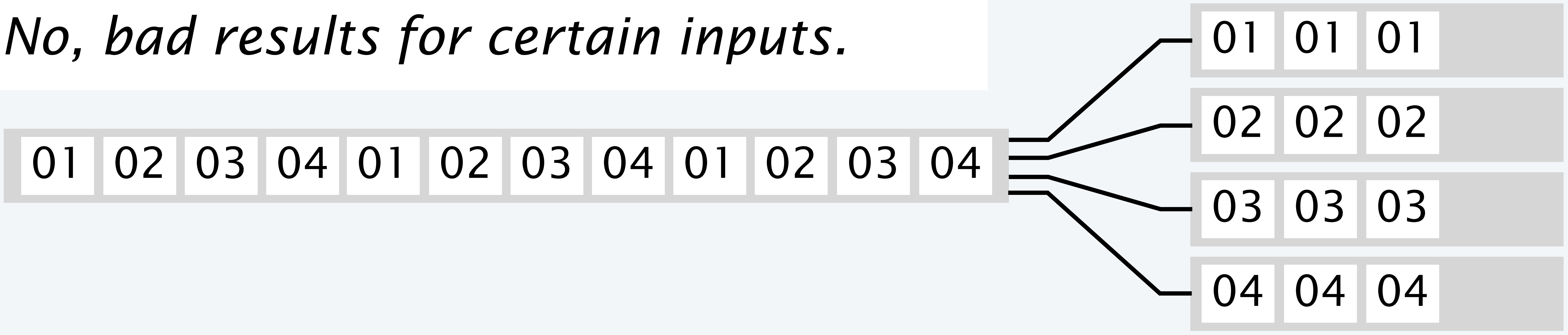
16384/.77351 = 21181
32768/.77351 = 42362
65536/.77351 = 84725
...

Stochastic splitting

Goal: Perform *M* independent PC experiments and average results.

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

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

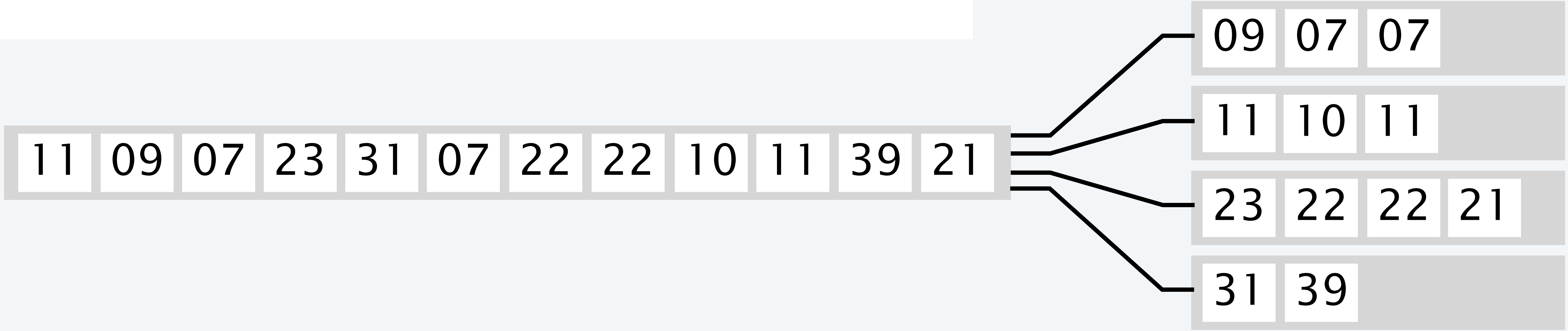


Alternative 3: **Stochastic splitting**

- Use second hash to divide stream into 2^m independent streams
- Use PC on each stream, yielding 2^m sketches .
- Compute *mean* = average number of trailing bits in the sketches.
- Return $2^{mean}/.77531$.

key point: equal values all go to the same stream

original paper calls it stochastic "averaging" later developments make "splitting" more apt



Probabilistic counting with stochastic splitting in Java

```
public static long estimate(Iterable<Long> stream, int M)
{
   long[] sketch = new long[M];
   for (long x : stream)
   {
      int k = hash2(x, M);
      sketch[k] = sketch[k] | R(hash(x));
   }
   int sum = 0;
   for (int k = 0; k < M; k++)
      sum += r(sketch[k]);
   double mean = 1.0 * sum / M;
   return (int) (M * Math.pow(2, mean)/.77351);
}
```

Idea. *Stochastic splitting*

- Use second hash to split into $M = 2^m$ independent streams
- Use PC on each stream, yielding 2^m sketches .
- Compute *mean* = average # trailing 1 bits in the sketches.
- Return $2^{mean}/.77351$.

Q. Accuracy obviously improves as M increases, but by how much?

Theoretical analysis of PCSA

Definition. The *relative accuracy* is the standard deviation of the estimate divided by the actual value.

LEMMA 4. *Setting $\beta = 2^{1/q}$, with $q \geq 1$, one has for fixed q*

$$E[\beta^{R_n}] = n^{1/q}(d_q + P_q(\log_2 n)) + o(n^{1/q}),$$

where

LEMMA 5. *If n elements are distributed into m cells (m fixed), where the probability that any element goes to a given cell has probability $1/m$, then the probability that at least one of the cells has a number of elements N satisfying*

$$|N - n/m| > \sqrt{n} \log n$$

Theorem (paraphrased to fit context of this talk).

Under appropriate assumptions about the hash function, PCSA

- *Uses 64M bits.*
- *Produces estimate with a relative accuracy close to* $0.78/\sqrt{M}$.

Proof (another quintessential Flajolet tour de force, omitted).

exact analysis via Mellin transform techniques

precise asymptotic estimates

uniform bounds computed with MACSYMA

and the same bound applies if 2 is replaced by β in the above sum.

We now consider the error that comes from the replacement of the $p_{n,k}$ by their asymptotic equivalent for "small" k. From the bounds of Theorem 2, one finds

$$\sum_{k \leq (5/4)\log_2 n} \beta^k \left[p_{n,k} - \psi\left(\frac{n}{2^k}\right) + \psi\left(\frac{n}{2^{k+1}}\right)\right] = O\left(\frac{n^{5/4q}}{n^{0.49}}\right) = O(n^{0.76/q}), \qquad (29)$$

proof of the first part of Theorem 4. Let S denote the sum $R^{\langle 1 \rangle} + R^{\langle 2 \rangle} + \cdots + R^{\langle m \rangle}$. We have

$$\Pr(S = k) = \sum_{\substack{n_1 + n_2 + \cdots + n_m = n \\ k_1 + k_2 + \cdots + k_m = k}} \frac{1}{m^n} \binom{n}{n_1, n_2, \ldots, n_m} p_{n_1,k_1} p_{n_2,k_2} \cdots p_{n_m,k_m}. \qquad (31)$$

Preliminary validation of PCSA

Hypothesis. Accuracy is as specified *for the hash functions we use and the data we have.*

Validation (Flajolet and Martin, 1985). Extensive reproducible scientific experiments (!)

Validation (RS, this morning).

```
% java PCSA 6000000 1024 < log.07.f3.txt
1106474
```

<1% larger than actual value

log.07.f3.txt

```
109.108.229.102
pool-71-104-94-246.lsanca.dsl-w.verizon.net
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
```

Q. Is PCSA effective?

A. ABSOLUTELY!

Summary: PCSA (Flajolet-Martin, 1983)

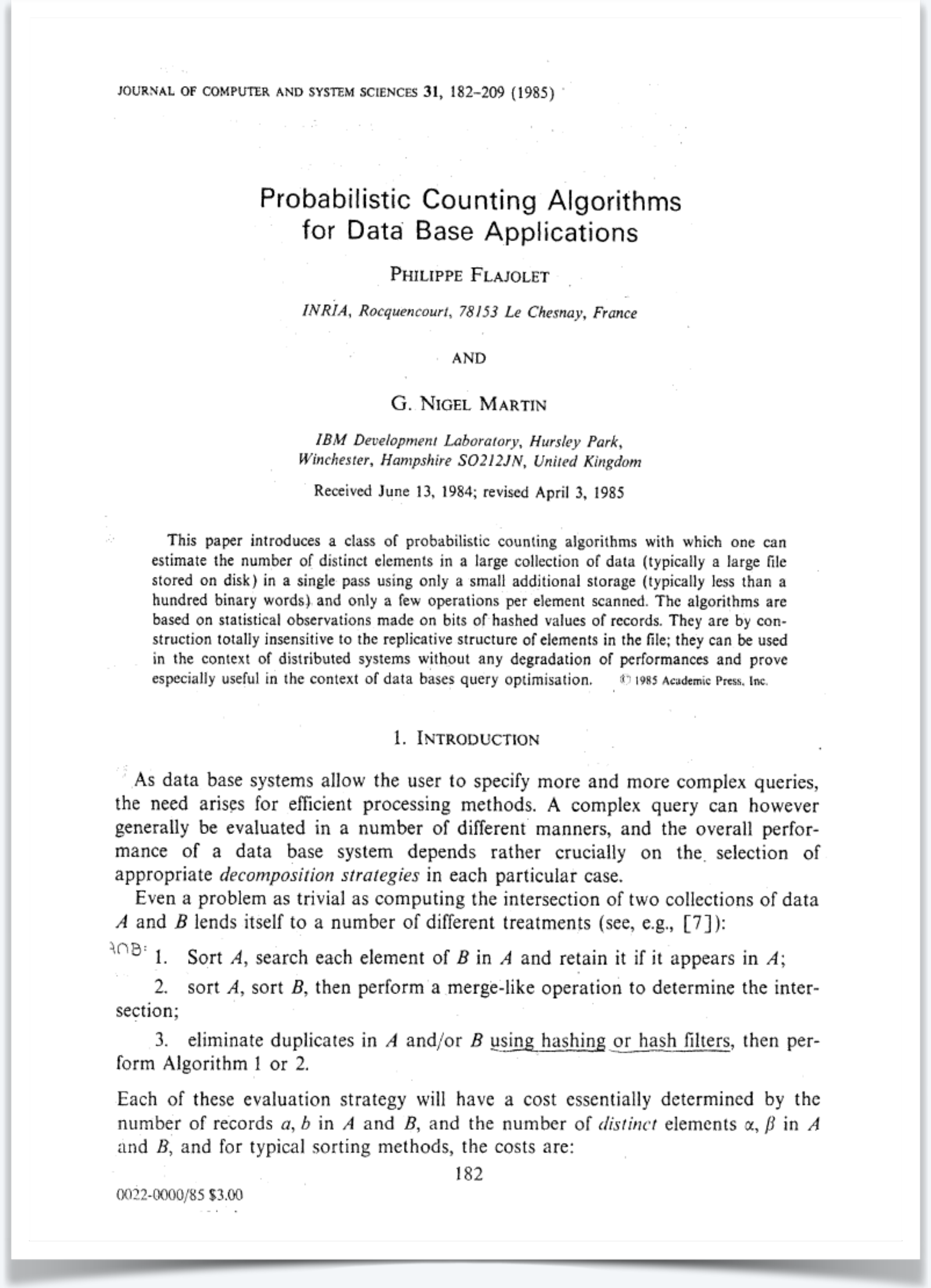
is a *demonstrably* effective approach to cardinality estimation

Q. *About* how many different values are present in a given stream?

PCSA

- Makes *one pass* through the stream.
- Uses *a few machine instructions per value*
- Uses M words to achieve relative accuracy $0.78/\sqrt{M}$

Results validated through extensive experimentation.



JOURNAL OF COMPUTER AND SYSTEM SCIENCES **31**, 182–209 (1985)

Probabilistic Counting Algorithms for Data Base Applications

PHILIPPE FLAJOLET

INRIA, Rocquencourt, 78153 Le Chesnay, France

AND

G. NIGEL MARTIN

IBM Development Laboratory, Hursley Park, Winchester, Hampshire SO212JN, United Kingdom

Received June 13, 1984; revised April 3, 1985

This paper introduces a class of probabilistic counting algorithms with which one can estimate the number of distinct elements in a large collection of data (typically a large file stored on disk) in a single pass using only a small additional storage (typically less than a hundred binary words) and only a few operations per element scanned. The algorithms are based on statistical observations made on bits of hashed values of records. They are by construction totally insensitive to the replicative structure of elements in the file; they can be used in the context of distributed systems without any degradation of performances and prove especially useful in the context of data bases query optimisation. © 1985 Academic Press, Inc.

1. INTRODUCTION

As data base systems allow the user to specify more and more complex queries, the need arises for efficient processing methods. A complex query can however generally be evaluated in a number of different manners, and the overall performance of a data base system depends rather crucially on the selection of appropriate *decomposition strategies* in each particular case.

Even a problem as trivial as computing the intersection of two collections of data A and B lends itself to a number of different treatments (see, e.g., [7]):

$A \cap B$: 1. Sort A, search each element of B in A and retain it if it appears in A;

2. sort A, sort B, then perform a merge-like operation to determine the intersection;

3. eliminate duplicates in A and/or B using hashing or hash filters, then perform Algorithm 1 or 2.

Each of these evaluation strategy will have a cost essentially determined by the number of records a, b in A and B, and the number of *distinct* elements α, β in A and B, and for typical sorting methods, the costs are:

182

0022-0000/85 $3.00

A poster child for AS/AC

Open questions

- Better space-accuracy tradeoffs?
- Support other operations?

"IT IS QUITE CLEAR that other observable regularities on hashed values of records could have been used... *– Flajolet and Martin*

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

Hyperbit: A Memory-Efficient Alternative to HyperLogLog

- The problem
- A solution
- **A better solution**
- Another approach
- Final frontier

We can do better (in theory)

Alon, Matias, and Szegedy

The Space Complexity of Approximating the Frequency Moments

STOC 1996; JCSS 1999.

Contributions

- Studied problem of estimating higher moments
- Formalized idea of randomized streaming algorithms
- Won Gödel Prize in 2005 for “foundational contribution”

Theorem (paraphrased to fit context of this talk).

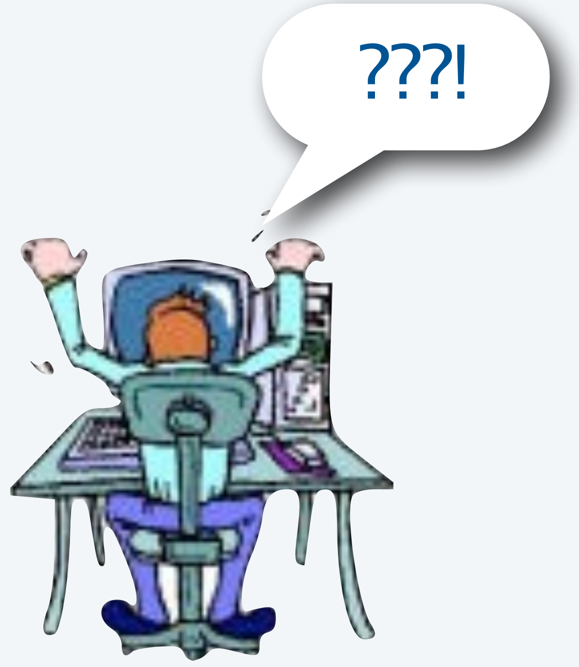
With strongly universal hashing, PC, for any $c > 2$,

- *Uses $O(\log N)$ bits.*
- Is accurate to a factor of c, *with probability at least $2/c$*.

Replaces “uniform hashing” assumption with “random bit existence” assumption

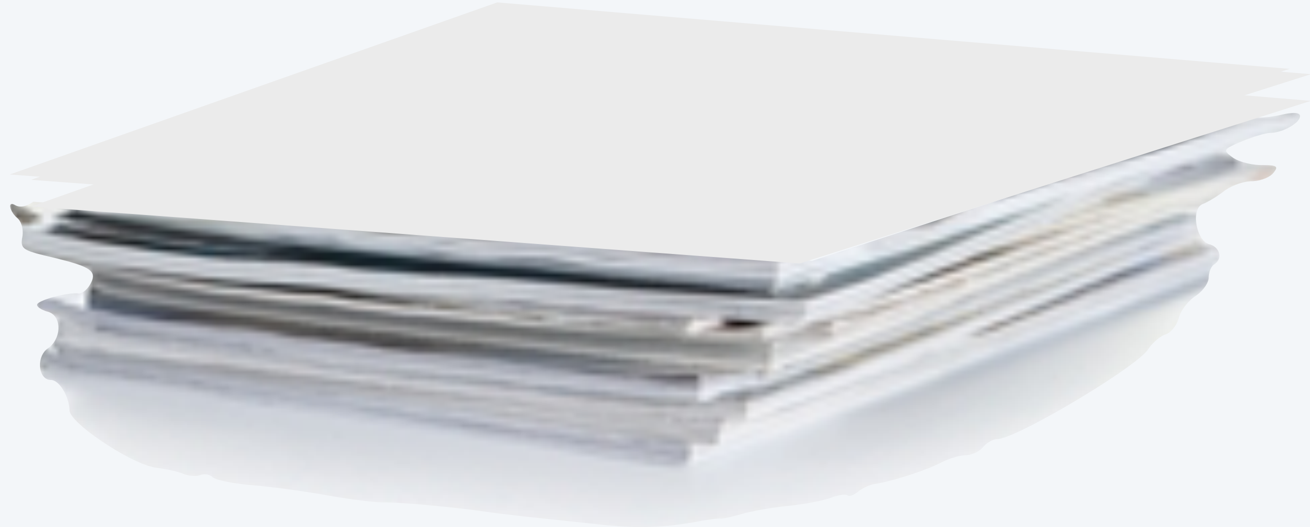
BUT, no impact on cardinality estimation in practice

- “Algorithm” just changes hash function for PC
- Accuracy estimate is too weak to be useful
- No validation

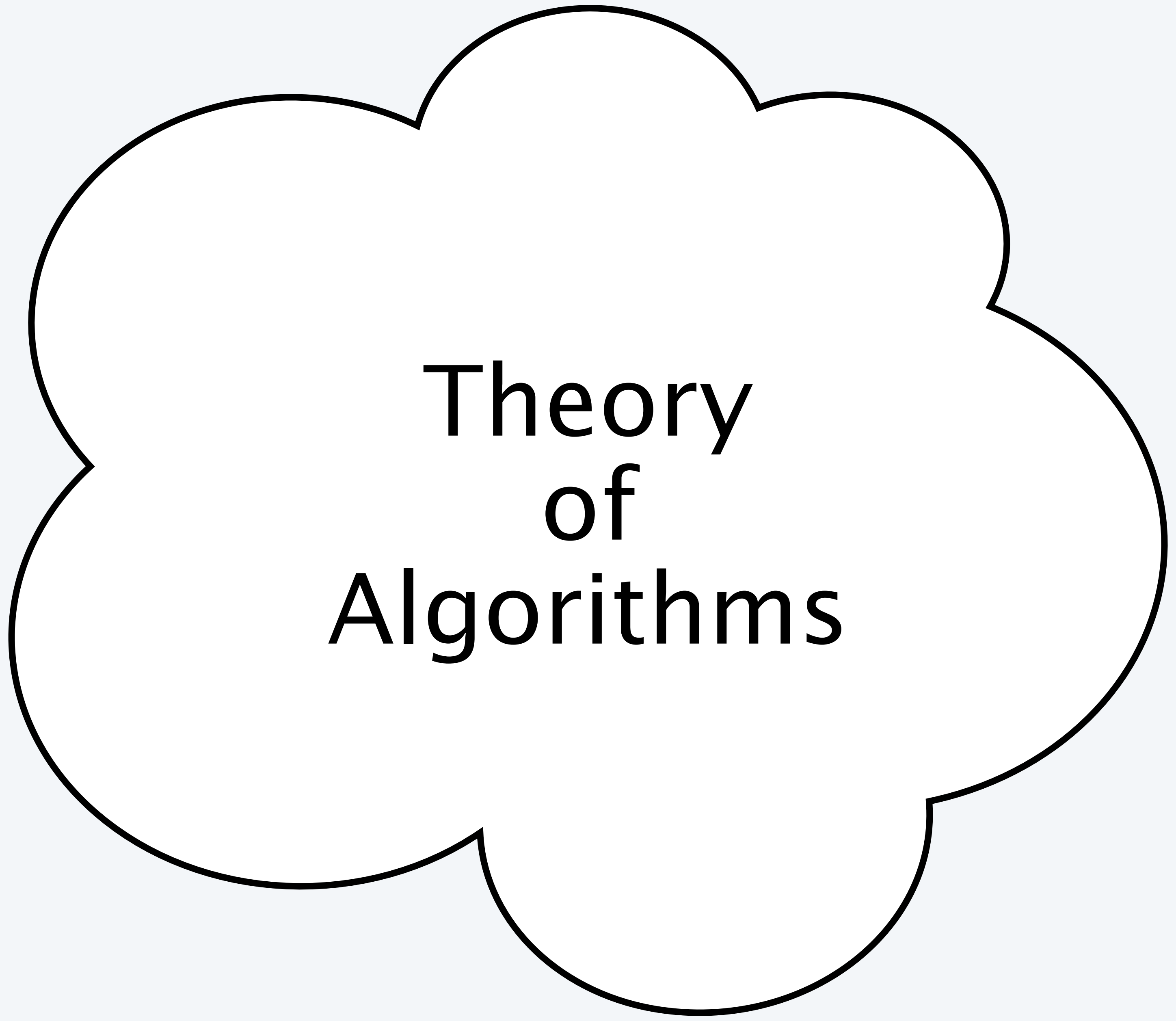


We can do better (in theory)

papers about cardinality estimation and other streaming algorithms



papers about streaming algorithms having validated implementations



We can do better (in theory)

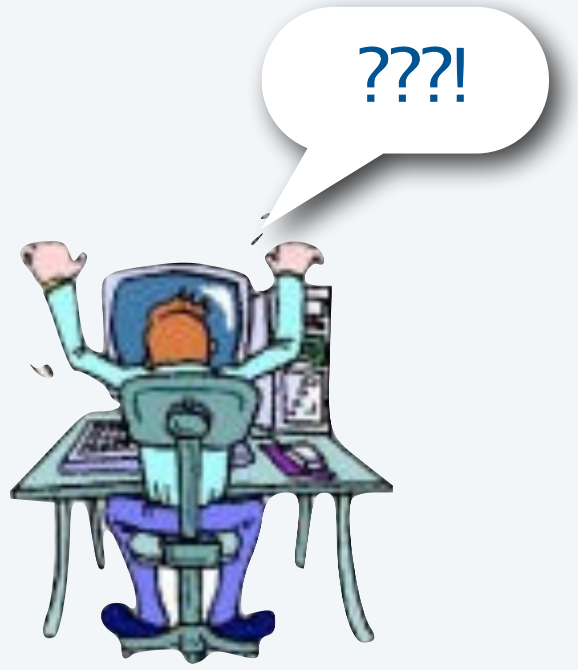
Bar-Yossef, Jayram, Kumar, Sivakumar, and Trevisan
Counting Distinct Elements in a Data Stream
RANDOM 2002.

Contribution
Improves space-accuracy tradeoff at extra stream-processing expense.

Theory of Algorithms

Theorem (paraphrased to fit context of this talk).
With strongly universal hashing, there exists an algorithm that
- *Uses O(M log log N) bits.* ⟵ *PCSA uses M lg N bits*
- Achieves relative accuracy $O(1/\sqrt{M})$.

STILL no impact on cardinality estimation in practice
- Infeasible because of high stream-processing expense.
- Big constants hidden in O-notation
- No validation

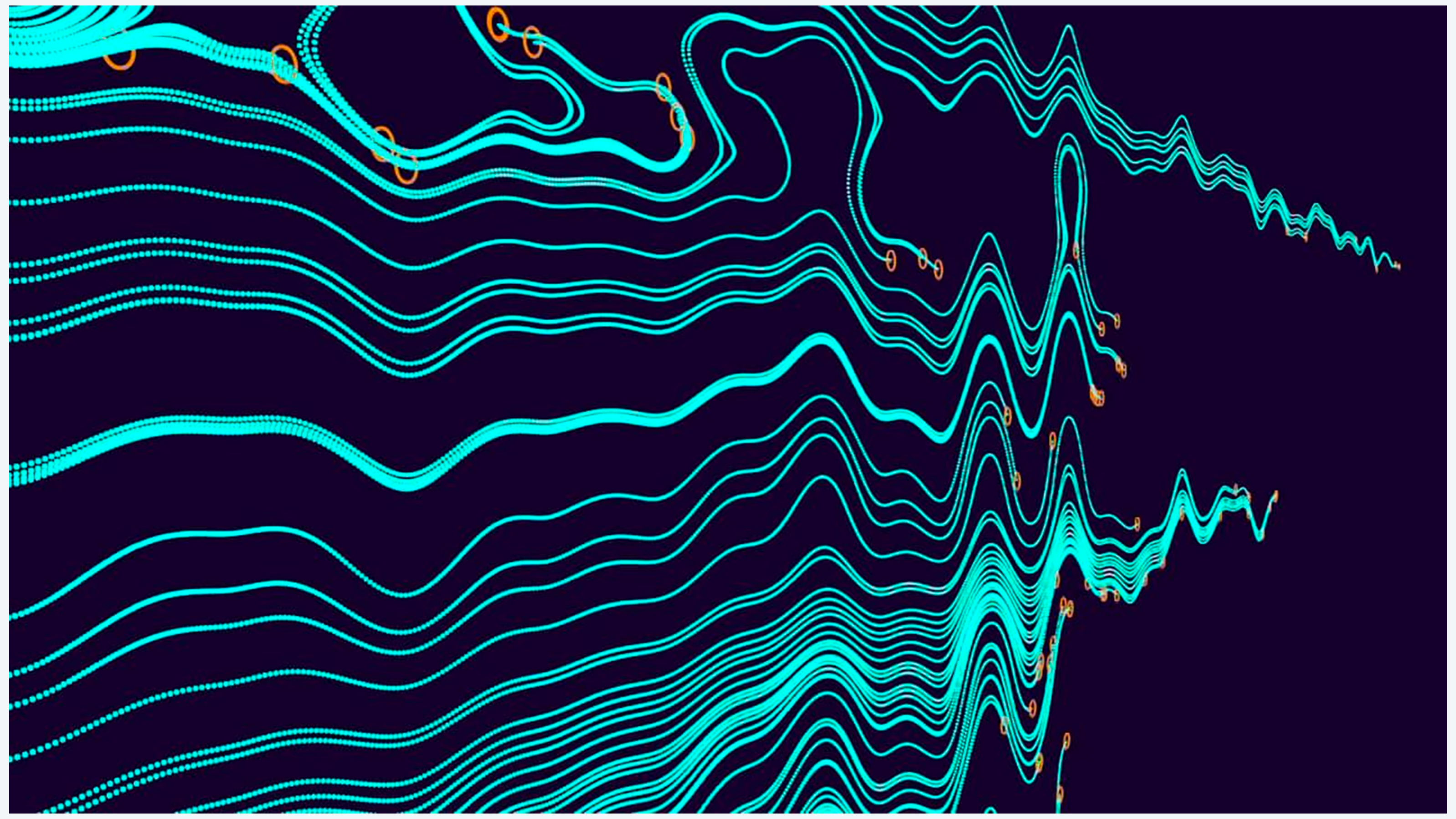


We can do better (in theory *and* in practice)

Flajolet, Fusy, Gandouet, and Meunier
HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm
AofA 2007; DMTCS 2007.

Contributions

- Presents **HyperLogLog** algorithm
- Easy variant of PCSA that uses a much smaller sketch
- Idea: Harmonic mean of `r()` values
- Reduces memory used *without* extra expense
- Full analysis, fully validated with experimentation



PCSA saves sketches (lg N bits each)

00000000000000000000000001101**111**

HyperLogLog saves **`r()`** *values (lglg N bits each)*

00100 (= 4)

We can do better (in theory and in practice): HyperLogLog algorithm (2007)

```
public static long estimate(Iterable<Long> stream, int M)
{
   int[] bytes = new int[M];
   for (long x : stream)
   {
      int k = hash2(s, M);
      int x = hash(s);
      if (bytes[k] < Bits.r(x)) bytes[k] = Bits.r(x);
   }
   double sum = 0.0;
   for (int k = 0; k < M; k++)
      sum += Math.pow(2, -1.0 - bytes[k]);
   return (int) (bias * M * M / sum);
}
```

8-bit bytes (code to pack into M lglgN bits omitted)

about .709 for M = 64

Flajolet-Fusy-Gandouet-Meunier 2007

Idea. *Harmonic mean of r() values*

- Use stochastic splitting
- Keep track of min($r(x)$) for each stream
- Return *harmonic mean.*

Flajolet, Fusy, Gandouet, and Meunier
HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm
AofA 2007; DMTCS 2007.

Theorem (paraphrased to fit context of this talk).

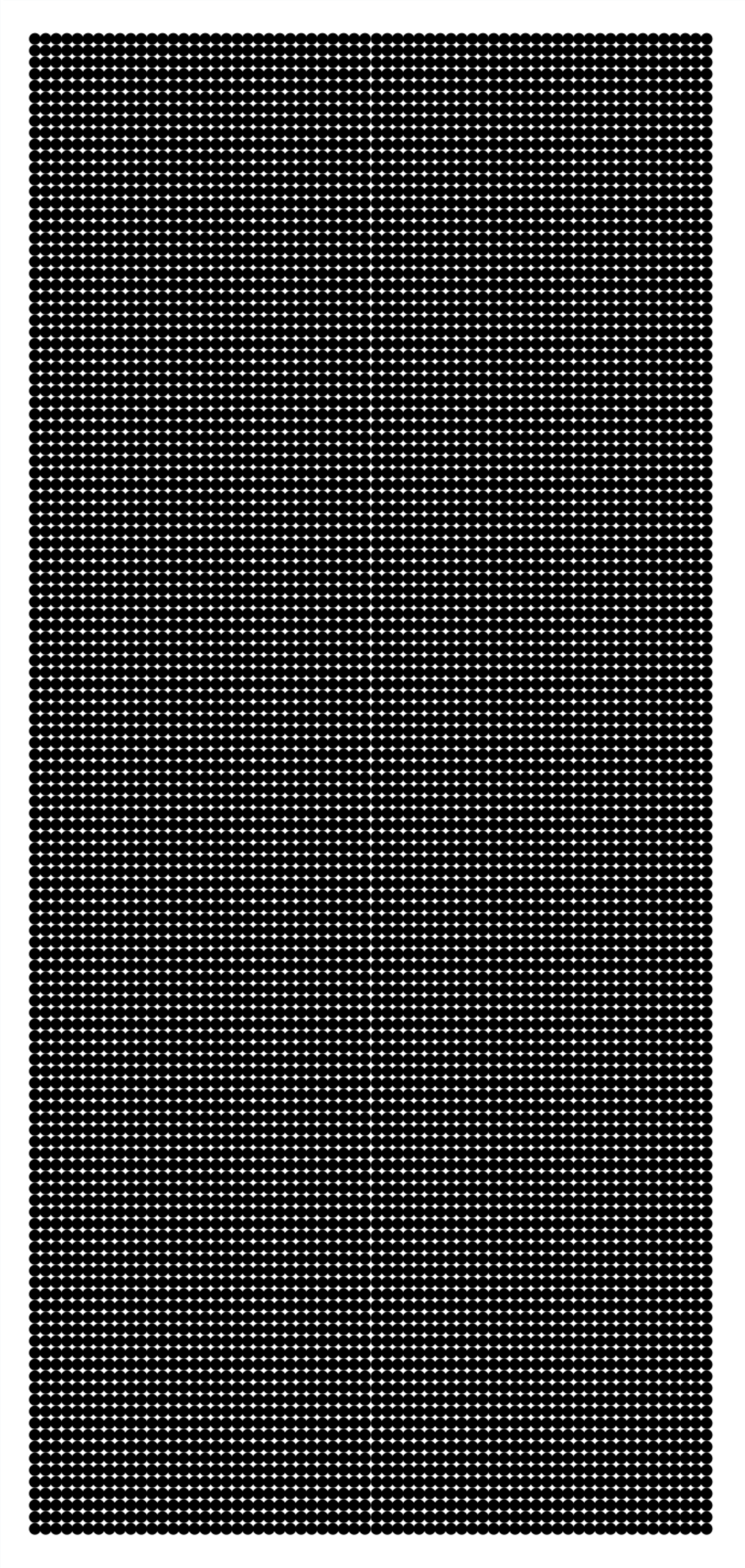
Under appropriate assumptions about the hash function, ***HyperLogLog***

- *Uses M lg lg N bits (6 in the real world).*
- *Achieves relative accuracy close to* $1.079/\sqrt{M}$.

Memory use for cardinality estimation algorithms with M-way stochastic splitting

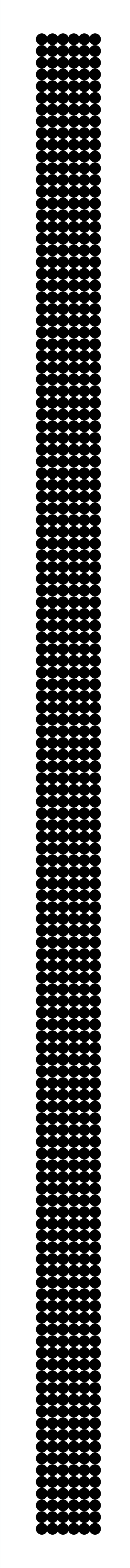
Probabilistic Counting

M 64-bit words



HyperLogLog

M 6-bit bytes



Pictured: M = 128

HyperLogLog accuracy hypothesis

Theorem (Flajolet, Fusy, Gandouet, and Meunier).
Let $H_{LL}(S, M)$ be the harmonic mean of the sketch computed by HyperLogLog *for a stream S having N distinct values when using M substreams. Then the statistic*

$$c_1 M H_{LL}(S, M) \ \textit{where}\ c_1 = \frac{1}{\ln 4} \doteq 0.721$$

is approximately Gaussian with mean N and variance $\sigma^2 \sim c_2 / M$ where $c_2 = 3 \ln 2 - 1 \doteq 1.079$.

Hypothesis. *The reported estimate will be within 3σ of the actual count 99% of the time.*

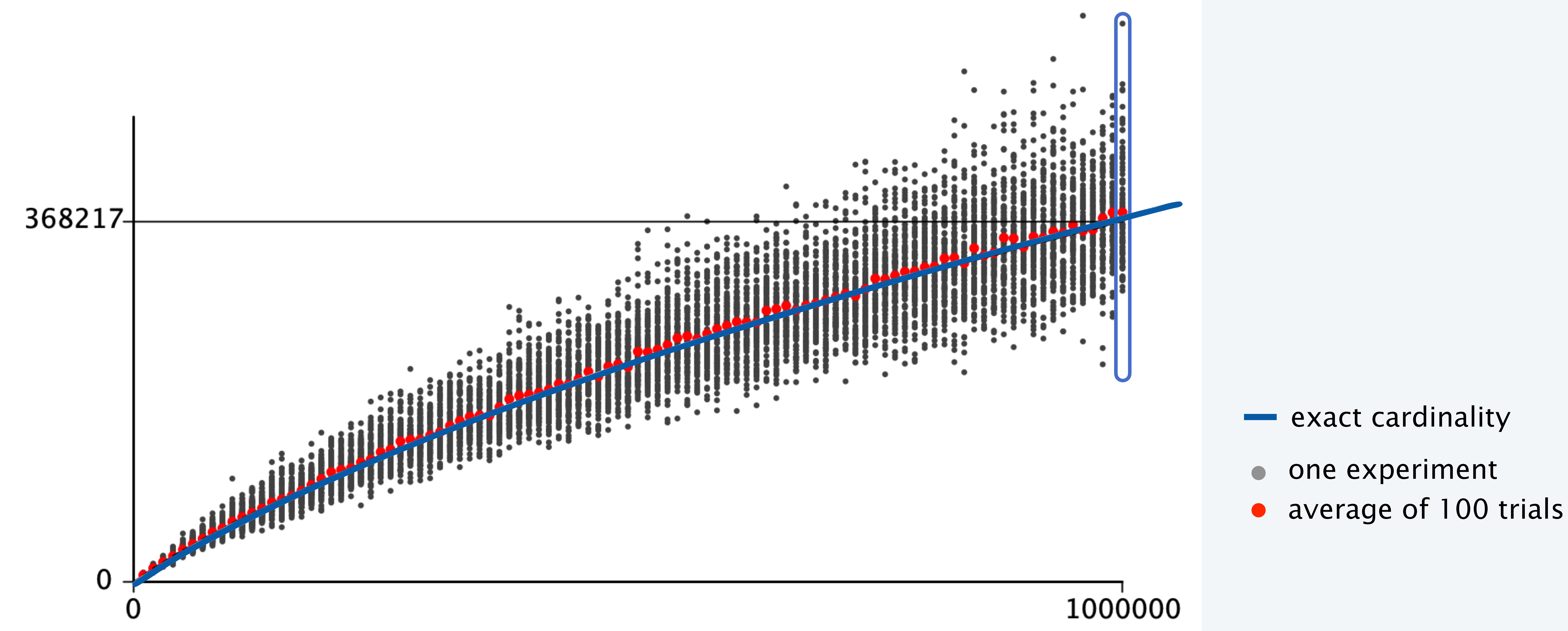
Consequence. *HLL can solve the practical cardinality count problem with 6144 bits.*

$$M = 1024$$

$$\sigma = \sqrt{3 \ln 2 - 1}/32 \doteq .032$$

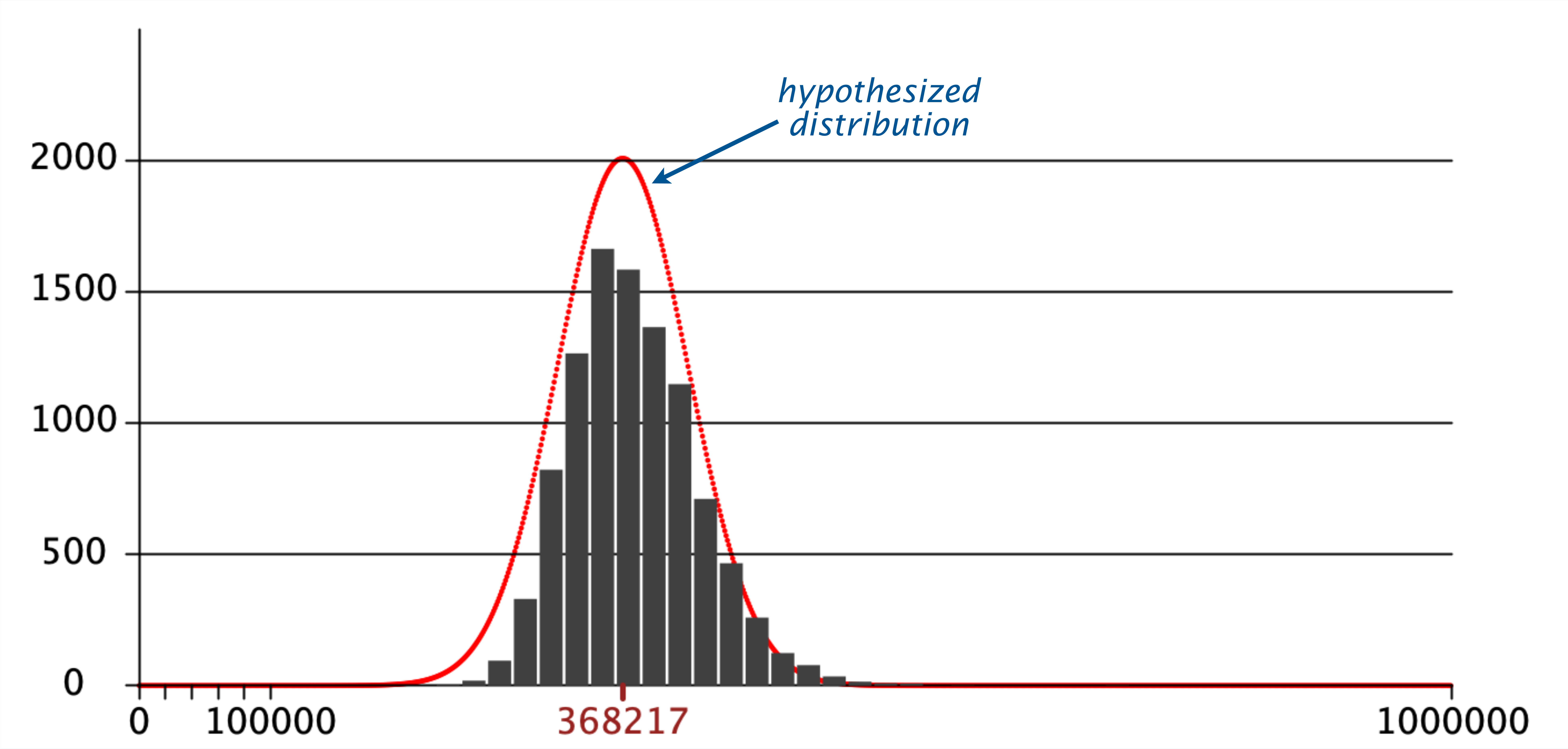
HyperLogLog validation I

Experiment. 100 trials for x*10000 inputs for x from 1 to 100 (10000 trials)



HyperLogLog validation II

Experiment. 10000 trials for 1 million inputs



*Histogram of number of estimates between x*2000 and (x+1)*2000*

Posted on Facebook, 2018 (continued)

Computing the count of distinct elements in massive data sets is often necessary but computationally intensive.

Say you need to determine the number of distinct people visiting Facebook in the past week using a single machine.

With a traditional SQL query on the data sets we use at Facebook this would take days and terabytes of memory.

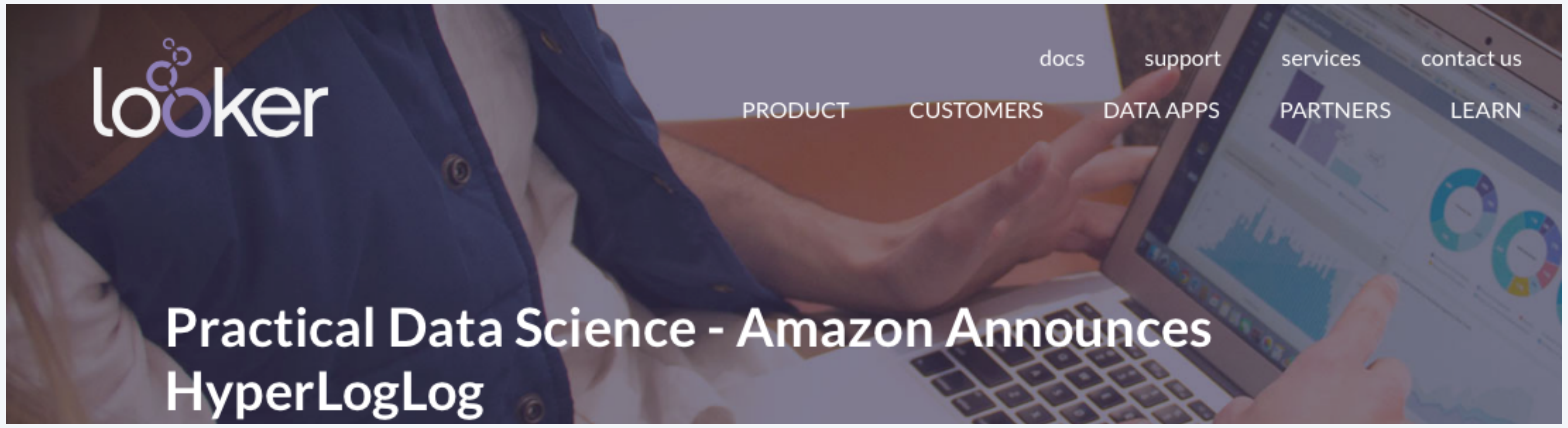
To speed up these queries, we implemented HyperLogLog (HLL) in Presto, a distributed SQL query engine.

HLL works by providing an approximate count of distinct elements.

With HLL, we can perform the same calculation in 12 hours with ***less than 1 MB of memory****.*

We have seen great improvements, with some queries being run ***within minutes****.*

Hyperloglog validation in the Real World



cradlepoint

redis

BigQuery

Periscope DATA

neustar

Research at Google

S. Heule, M. Nunkesser and A. Hall
HyperLogLog in Practice: Algorithmic Engineering of a State of The Art Cardinality Estimation Algorithm.
Extending Database Technology/International Conference on Database Theory 2013.

AMAZON REDSHIFT

Philippe Flajolet, mathematician and algorithm scientist extraordinaire

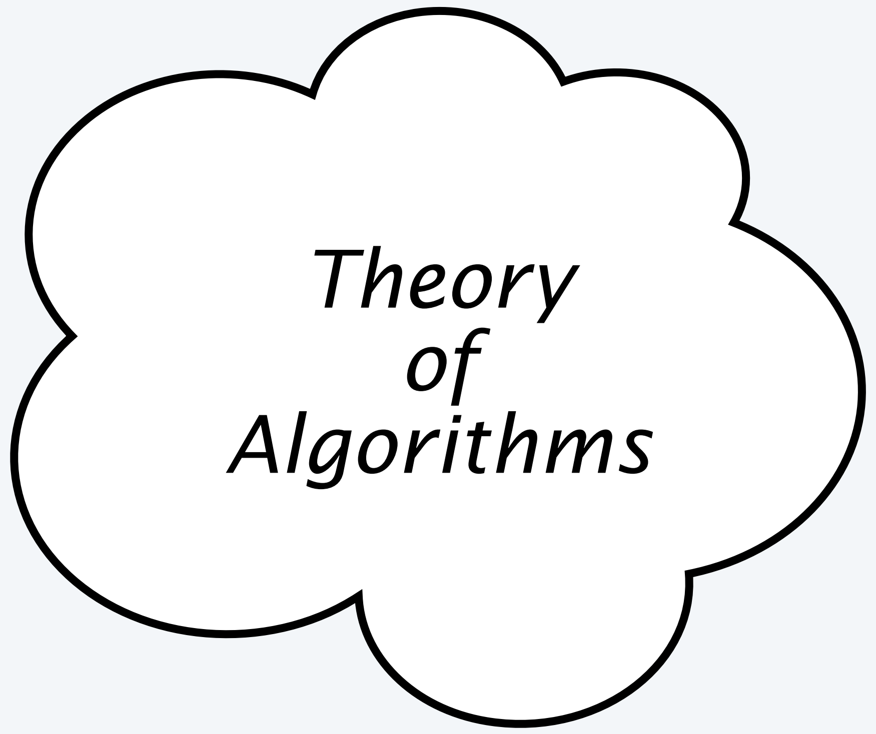
Hyperbit: A Memory-Efficient Alternative to HyperLogLog

- The problem
- A solution
- A better solution
- **Another approach**
- Final frontier

We can do a bit better (in theory) *but not much better*

Indyk and Woodruff

Tight Lower Bounds for the Distinct Elements Problem, FOCS 2003.



Theorem (paraphrased to fit context of this talk).

Any algorithm that achieves relative accuracy $O(1/\sqrt{M})$ *must use* $\Omega(M)$ *bits*

Kane, Nelson, and Woodruff

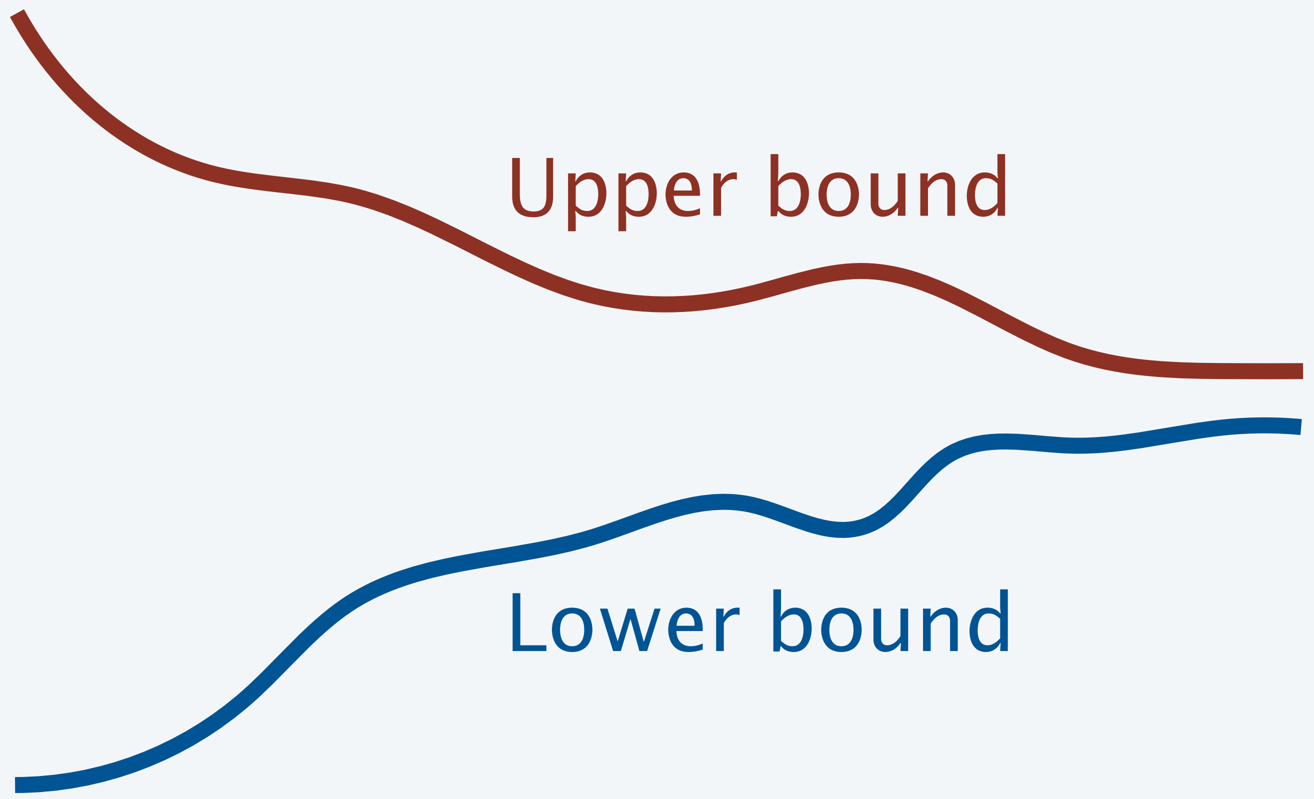
Optimal Algorithm for the Distinct Elements Problem, PODS 2010.

Theorem (paraphrased to fit context of this talk).

With strongly universal hashing there exists an algorithm that

- *Uses* O(*M + loglog N*) *bits.*
- *Achieves relative accuracy* $O(1/\sqrt{M})$.

optimal



Not a practical algorithm (never implemented, no validation)

- Tough to beat HyperLogLog's low stream-processing expense.
- Constants hidden in O-notation not likely to be small (need to be <6)

Open: Does there exist an "optimal" algorithm for the **practical** cardinality estimation problem?

Can we beat HyperLogLog in practice?

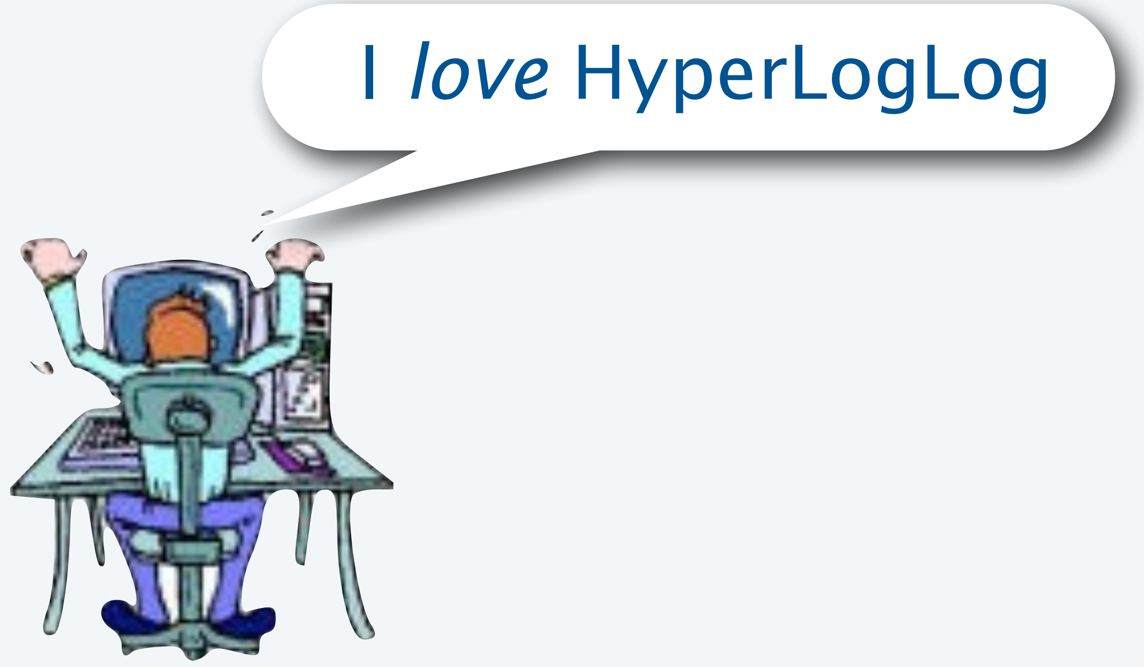
Necessary characteristics of a better algorithm

- Makes *one pass* through the stream.
- Uses *a few dozen machine instructions per value*
- Uses *a few hundred bits*
- Achieves 10% relative accuracy or better

> *" I've long thought that there should be a simple algorithm that uses a small constant times M bits..."*
>
> *– Jérémie Lumbroso*

	machine instructions per stream element	*memory bound*	*memory bound when $N < 2^{64}$*	*# bits for 10% accuracy when $N < 2^{64}$*
HyperLogLog	20–30	$M \log\log N$	$6M$	6144
BetterAlgorithm	*a few dozen*	cM	$2M$ or $3M$	*a few thousand*

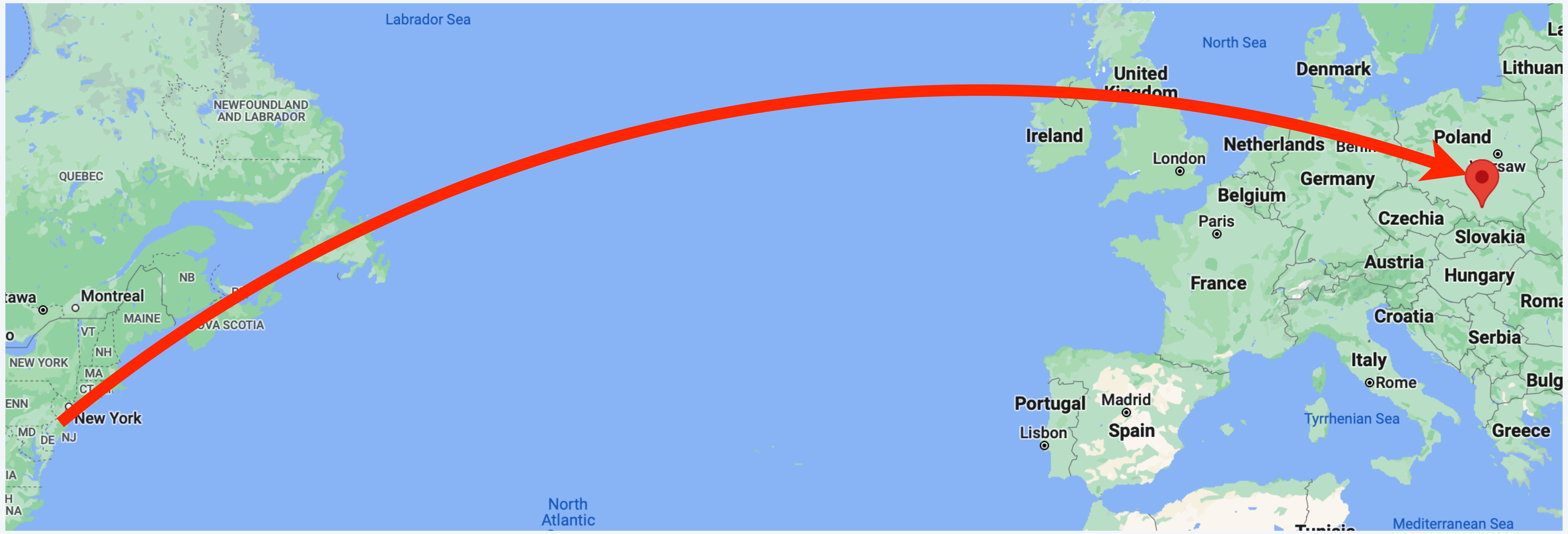
Also, results need to be validated through experimentation.



Krakow, 2016

27th AofA, July 3-8

Trip to Krakow



A proposal: HyperBitBit (Sedgewick, 2016)

```
public static long estimate(Iterable<String> stream, int M)
{
   int T = 0;
   long sketch = 0;
   long sketch2 = 0;
   for (String x : stream)
   {
      long x = hash(s);
      int k = hash2(x, 64);
      if (r(x) > T)      sketch  = sketch  | (1L << k);
      if (r(x) > T + 1)  sketch2 = sketch2 | (1L << k);
      if (p(sketch) >= 32)
      { sketch = sketch2; T++; sketch2 = 0; }
   }
   return (int) (Math.pow(2, T + bias + p(sketch)/32.0));
}
```

M = 64 likely to be value of choice

bias factor (to be determined empirically)

Idea.

- T is estimate of $\lg N$
- sketch is 64 indicators whether to increment T
- sketch2 is is 64 indicators whether to increment T *by 2*
- Update when half the bits in sketch are 1
- correct with p(sketch) *and bias factor*

recall that p(x) is the number of 1 bits in x

Example HyperBitBit actions (M=8)

substream → k(x); *# trailing 1s* → r(x); *estimate of lg N* → T

	x	k(x)	r(x)	T	sketch	sketch2
					7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
1 0 0	... 1 1 1 0 0 0 1 1 0 1 0 0 0 1 1 0 0 1 1 1 1 0 1 **0** 1 1	4	2	5	0 0 1 0 0 0 0 0	0 0 1 0 0 0 0 0
	no change because **r(x)<=T**			5	0 0 1 0 0 0 0 0	0 0 1 0 0 0 0 0
1 0 1	... 1 1 1 0 0 0 1 1 0 1 0 0 0 1 1 0 0 1 1 1 1 1 1 1 1 1	5	9	5	0 0 1 0 0 0 0 0	0 0 1 0 0 0 0 0
	no change because both sketch bits are set					
0 0 1	... 1 1 1 0 0 0 1 1 0 1 0 0 0 1 1 0 0 1 1 0 1 1 1 1 1 1	1	6	5	0 0 1 0 0 0 0 0	0 0 1 0 0 0 0 0
	set sketch bit because **r(x)>T**			5	0 0 1 0 0 0 1 0	0 0 1 0 0 0 0 0
1 1 0	... 1 1 1 0 0 0 1 1 0 1 0 0 0 1 1 0 0 1 0 1 1 1 1 1 1 1	6	7	5	0 0 1 0 0 0 1 0	0 0 1 0 0 0 0 0
	set both sketch bits because **r(x)>T+1**			5	0 1 1 0 0 0 1 0	0 1 1 0 0 0 0 0
0 0 0	... 1 1 1 0 0 0 1 1 0 1 0 0 0 1 1 0 0 1 1 0 1 1 1 1 1 1	0	6	5	0 1 1 0 0 0 1 0	0 1 1 0 0 0 0 0
	set sketch bit because **r(x)>T**			5	0 1 1 0 0 0 1 1	0 1 1 0 0 0 0 0
	increment T and reset sketches because half the bits in sketch are set			6	0 1 1 0 0 0 0 0	0 0 0 0 0 0 0 0

A proposal: HyperBitBit (Sedgewick, 2016)

```
public static long estimate(Iterable<String> stream, int M)
{
   int T = 0;
   long sketch = 0;
   long sketch2 = 0;
   for (String x : stream)
   {
      long x = hash(s);
      int k = hash2(x, 64);
      if (r(x) > T)     sketch  = sketch  | (1L << k);
      if (r(x) > T + 1) sketch2 = sketch2 | (1L << k);
      if (p(sketch) >= 32)
      { sketch = sketch2; T++; sketch2 = 0; }
   }
   return (int) (Math.pow(2, T + bias + p(sketch)/32.0));
}
```

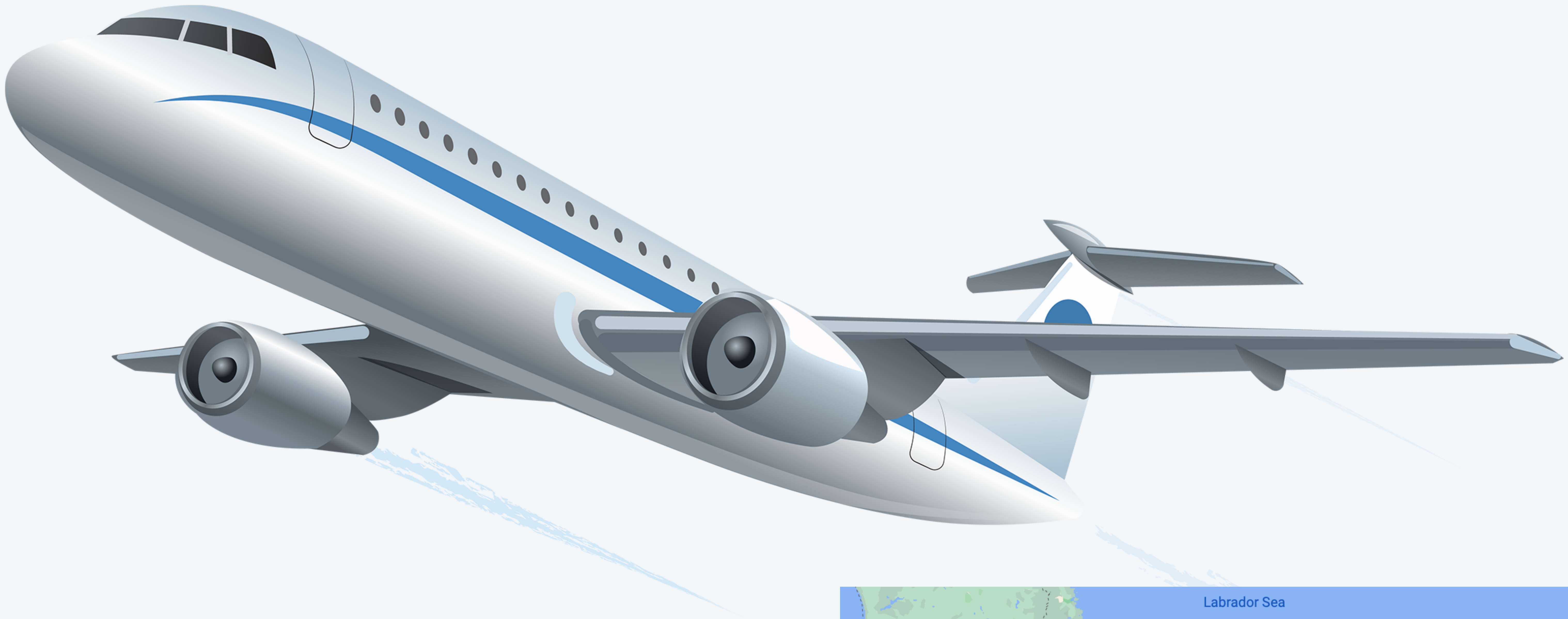
Idea.

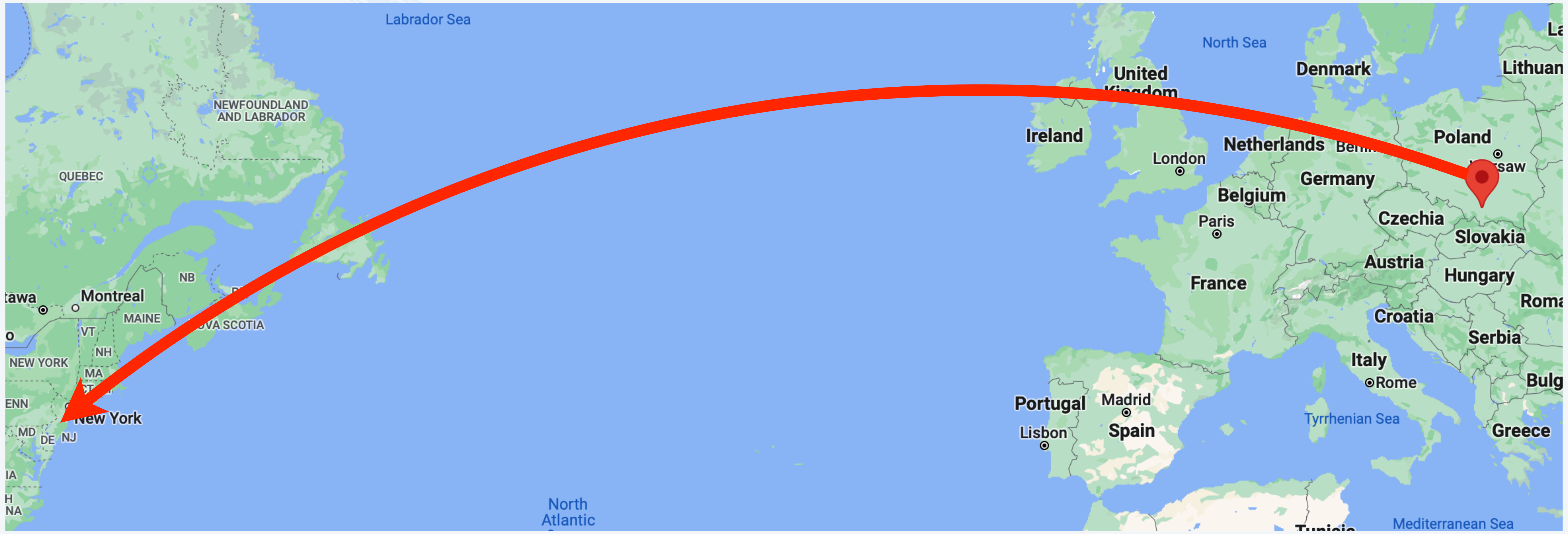
- T is estimate of $\lg N$
- sketch is 64 indicators whether to increment T
- sketch2 is is 64 indicators whether to increment T *by 2*
- Update when half the bits in sketch are 1
- correct with p(sketch) *and bias factor*

Q. What is the bias factor?

Q. Does this even work?

Return trip from Krakow





HyperBitBit preliminary validation

. . . after some hacking to settle on **`bias = 5.4`** . . .

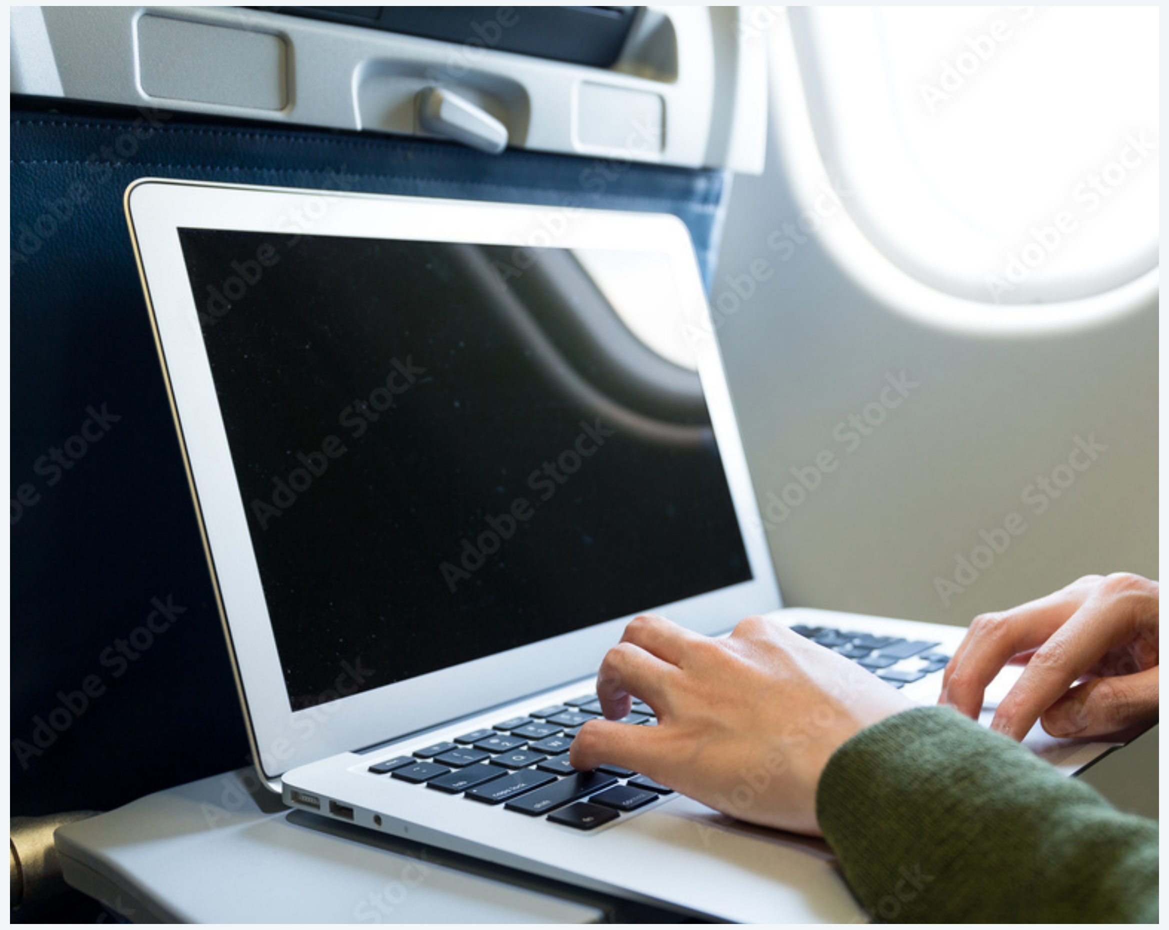
Exact values for web log example

```
% java Hash 1000000 < log.07.f3.txt
242601
% java Hash 2000000 < log.07.f3.txt
483477
% java Hash 4000000 < log.07.f3.txt
883071
% java Hash 6000000 < log.07.f3.txt
1097944
```

HyperBitBit estimates

```
% java HyperBitBit 1000000 < log.07.f3.txt
234219
% java HyperBitBit 2000000 < log.07.f3.txt
499889
% java HyperBitBit 4000000 < log.07.f3.txt
916801
% java HyperBitBit 6000000 < log.07.f3.txt
1044043
```

	1,000,000	2,000,000	4,000,000	6,000,000
Exact	242,601	483,477	883,071	1,097,944
HyperBitBit	234,219	499,889	916,801	1,044,043
ratio	1.05	1.03	0.96	1.03



It *does* seem to work!

Next challenge: analyze HyperBitBit

2016

2017

Sweden, 2018

Knuth 80, January 8–10

ALGORITHMS
COMBINATORICS
INFORMATION

COLLOQUIUM FOR
DON KNUTH'S
80TH BIRTHDAY

29th AofA, June 25–29

UPPSALA
UNIVERSITET

HyperBitBit analysis (Janson, 2018)

Key observation: the process obeys a *Poisson distribution.*

event: "next item in the data stream has more than T trailing 1s"

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

- Pr {*a given item has more than **T** trailing 1s* } $= 1/2^{T+1}$
- Pr {*no item has more than **T** trailing 1s* } $\sim e^{-v/2^{T+1}}$

$$\left(1-\frac{1}{2^{T+1}}\right)^v$$

corresponding bit in sketch is 0

Each HyperBitBit **phase** begins when ***T*** is incremented

- sketch2 is set to 0
- sketch is set to sketch2, say it has ***qM*** 0s
- After ***Mv*** distinct values (approximately ***v*** per stream) are added
  - number of 0s in each sketch is binomially distributed
  - expected number of 0s in sketch2 is $\sim Me^{-v/2^{T+2}}$
  - expected number of 0s in sketch is $\sim Mqe^{-v/2^{T+1}}$

```
1010011110111011
0001111100000101
0110110110110011
0000000111011111
0101110001000100
0000101001010101
1010101111111100
0001011100110111
1110010000111111
1010110011111100
0110001001100011
0110011100100011
0001000100011100
0100010001110111
0110100000101100
0011011110110000
1111000100111110
0001111100010100
1010001000100011
0010101010111111
1110101110001000
0110000110111101
0101010110110110
1001010101111111
```

HyperBitBit analysis (continued)

Def. Let q_T be the expected proportion of 0s in sketch, at the beginning of phase T.

Def. Let v_T be the expected number of values added to each stream during phase T.

Expected proportion of 0s in sketch2	$e^{-v_T/2^{T+2}}$
Expected proportion 0s in sketch	$q_T e^{-v_T/2^{T+1}}$
Phase T ends after Mv_T new values where	$q_T e^{-v_T/2^{T+1}} = 1/2$
Solve for v_T	$v_T = 2^{T+1}\ln(2q_T)$
Expected proportion of 0s in sketch2 at that point	$e^{-\ln(2q_T)/2}$
THEREFORE	$q_{T+1} = \dfrac{1}{\sqrt{2q_T}}$ **!!**

q_0	1.0000
q_1	0.7071
q_2	0.8408
q_3	0.7711
q_4	0.8052
q_5	0.7879
q_6	0.7966
q_7	0.7923
q_8	0.7944
q_9	0.7933

$q = 2^{-1/3} \doteq 0.7937$

Lemma 1. As T increases, proportion of 0s in sketch approaches $2^{-1/3}$ (solution of $q = 1/\sqrt{2q}$).

Lemma 2. Expected number of values in phase T is $Mv_T \sim 2M\ln 2^{-1/3}2^{T+1} = M\cdot(4/3)\ln 2\cdot 2^T$

HyperBitBit analysis accounting summary

Lemma 2. Expected number of values *in* phase ***T*** is $\sim M\dfrac{4\ln 2}{3}2^T$

Lemma 3. Expected number of values *before* phase ***T*** is $\sim M\dfrac{4\ln 2}{3}2^T$

$$M\frac{4\ln 2}{3}\sum_{0\le i<T} 2^i$$

Lemma 4. If there are βM 0s in the sketch on termination, then the expected number of values in the last phase is $M(\ln 2^{-1/3} - \ln\beta)\ 2^{T+1}$

Mv where v satisfies

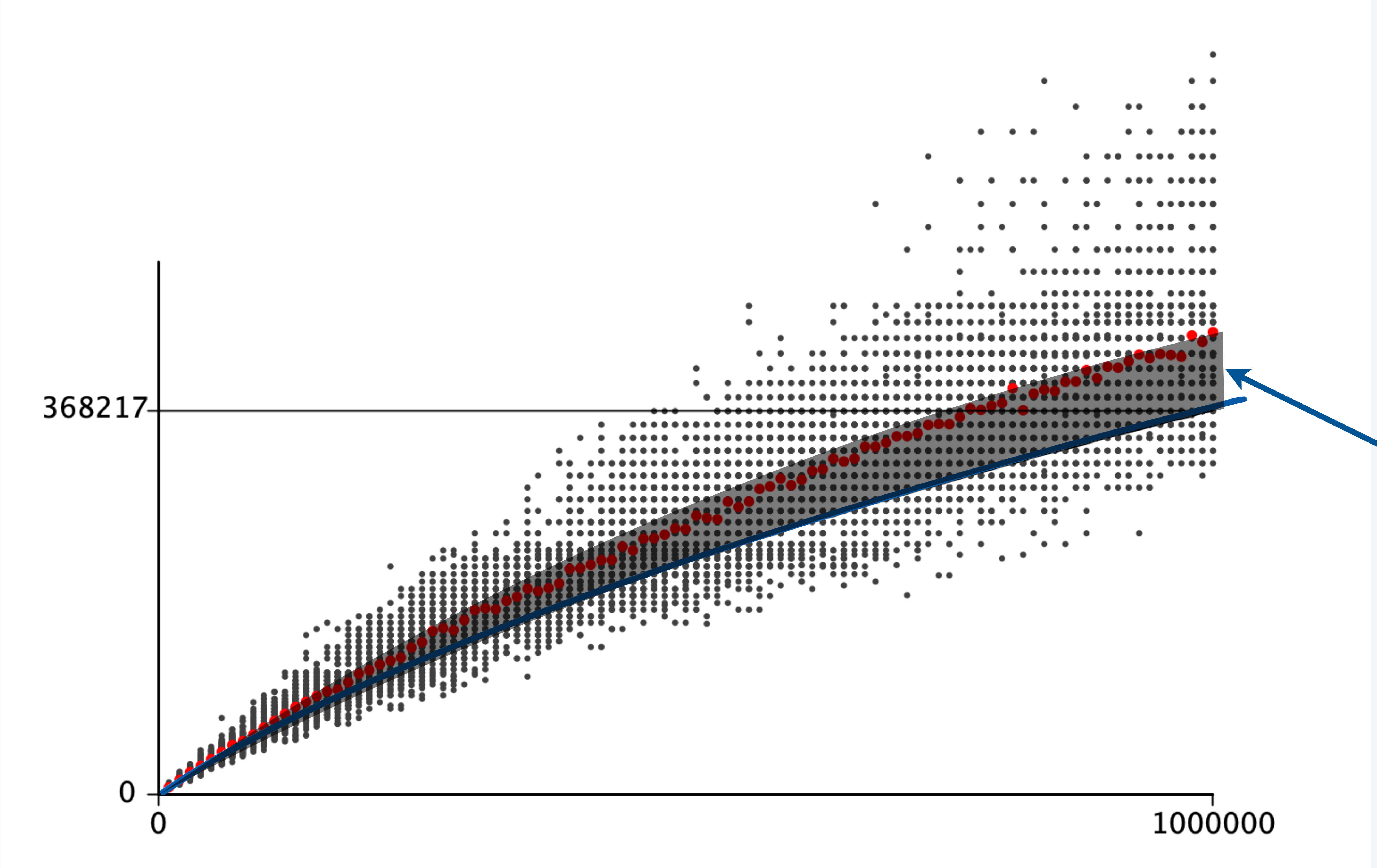
$$qe^{-v/2^{T+1}} = \beta$$

and $q = 2^{-1/3}$

Theorem. When `HyperBitBit` terminates with βM 0s in sketch in phase ***T***, then *N*/*M* is $\sim \Big(\dfrac{2\ln 2}{3} - 2\ln\beta\Big)2^T$

increases from .9242 to 1.8484
as β decreases from .7933 to .5

HyperBitBit validation (?)



OBVIOUSLY the estimate is too high because values with > **T+1** zeros are *recounted* later on.

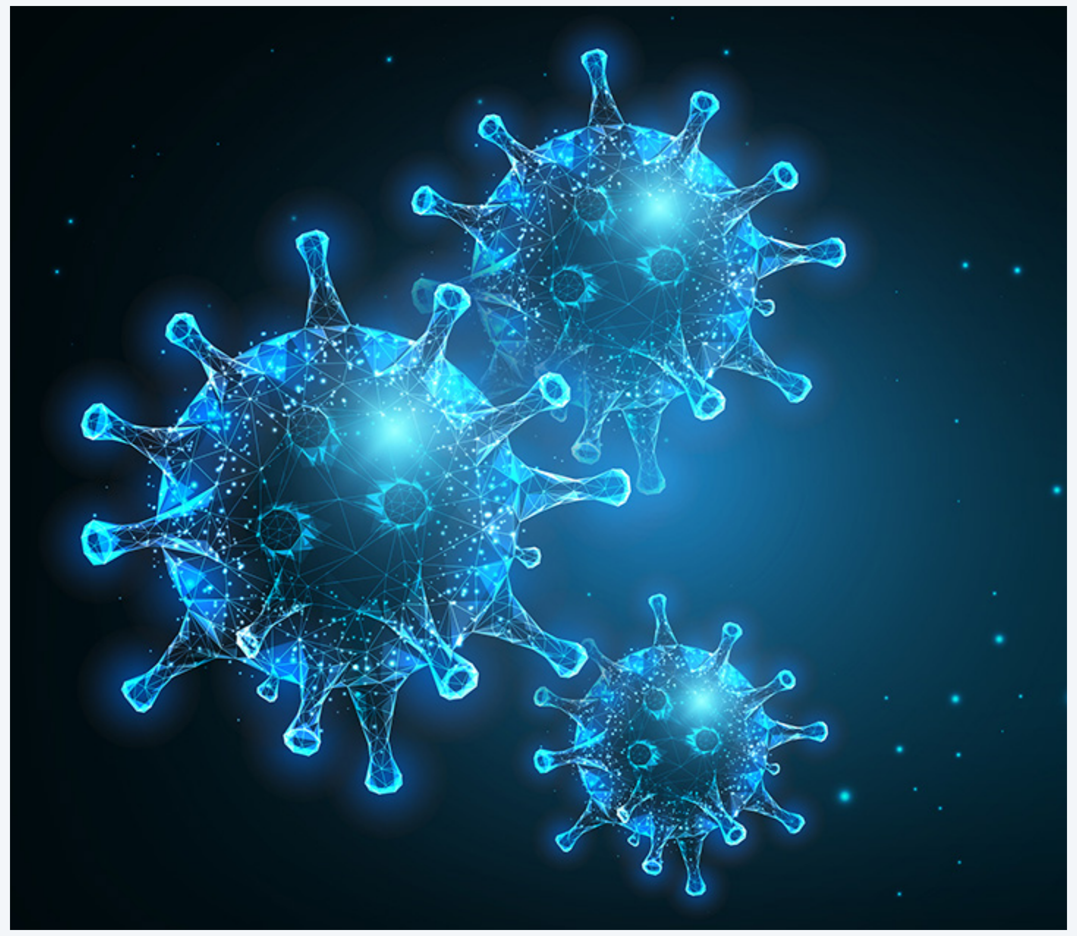
There are too many recounted values to ignore.

HyperBitBitBit ? No. Would be better, but still a problem.

Next challenge: estimate the number of recounts in HyperBitBit

2019

2022



2021

2020

Hyperbit: A Memory-Efficient Alternative to HyperLogLog

- The problem
- A solution
- A better solution
- Another approach
- Final frontier

OF

A simpler algorithm: HyperBit

Insight: We need to estimate *all* the forgotten values—why bother keeping track of them for ***T+1*** ?

```
public static long estimate(Iterable<String> stream, int M)
{
   int T = 0;
   long sketch = 0;
   for (String x : stream)
   {
      long x = hash(s);
      int k = hash2(x, 64);
      if (r(x) > T) sketch  = sketch  | (1L << k);
      if (p(sketch) >= 32)
      { T++; sketch = 0; }
   }
   return (int) (Math.pow(2, T)*M* ?? );
}
```

M = 64

bias factor (to be analyzed)

Idea.

- T is estimate of $\lg N$
- sketch is M indicators whether to increment T
- Set a sketch bit when r(x)>T
- Update when half the bits in sketch are 1
- Correct at end with *bias factor that is a function of p(sketch)*

Preliminary experimental validation inconclusive—but maybe analyzing this will be informative.

Example Hyperbit actions (M=8)

substream → k(x)   *# trailing 1s* → r(x)   *estimate of* lg *N* → T

	x	k(x)	r(x)	T	sketch
					7 6 5 4 3 2 1 0
					1 0 1 0 0 0 0 0
1 0 0	... 1 1 1 0 0 0 1 1 0 1 0 0 0 1 1 0 0 1 1 1 1 0 1 **0** 1 1	4	2	5	1 0 1 0 0 0 0 0
	no change because **r(x)<=T**			5	1 0 1 0 0 0 0 0
1 0 1	... 1 1 1 0 0 0 1 1 0 1 0 0 0 1 1 0 0 1 1 1 1 1 1 1 1	5	9	5	1 0 1 0 0 0 0 0
	no change because sketch bit for substream is already set				1 0 1 0 0 0 0 0
0 0 1	... 1 1 1 0 0 0 1 1 0 1 0 0 0 1 1 0 0 1 1 0 1 1 1 1 1 1	1	6	5	1 0 1 0 0 0 **0** 0
	set sketch bit for substream because **r(x)>T**			5	1 0 1 0 0 0 **1** 0
0 0 0	... 1 1 1 0 0 0 1 1 0 1 0 0 0 1 1 0 0 0 1 1 1 1 1 1 1 1	0	8	5	1 0 1 0 0 0 1 **0**
	set sketch bit for substream because **r(x)>T**			5	1 0 1 0 0 0 1 **1**
	half the sketch bits are set so **increment T and reset sketch**			**6**	**0 0 0 0 0 0 0 0**

HyperBit analysis

Starting point is the same as for HyperBitBit, but simpler

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

- Pr {*a given item has more than **T** trailing 1s* } $= 1/2^{T+1}$
- Pr {*no item has more than **T** trailing 1s* } $\sim e^{-v/2^{T+1}}$

$$\left(1 - \frac{1}{2^{T+1}}\right)^v$$

corresponding bit in sketch is 0

Each HyperBit **phase** begins when ***T*** is incremented

- sketch is set to 0
- After ***Mv***$_T$ distinct values (approximately v_T per stream) are added
  - number of 0s in sketch is binomially distributed
  - expected number of 0s in sketch is $\sim Me^{-v/2^{T+1}}$
  - phase ends when $e^{-v_T/2^{T+1}} = 1/2$, or $v_T = 2^{T+1}\ln 2$

Lemma. Expected number of values in phase ***T*** is $\sim Mv_T = M \cdot \ln 4 \cdot 2^T$

1010011110111011
0001111100000101
0110110110110011
0000001110111111
0101110001000100
0000101001010101
1010101111111100
0001011100110111
1110010000111111
1010110011111100
0110001001100011
0110011100100011
0001000100011100
0100010001110111
0110100000101100
0011011110110000
1111000100111110
0001111100010100
1010001000100011
0010101010111111
1110101110001000
0110000110111101
0101010110110110
1001010101111111

HyperBit analysis (estimating the values that will be recounted)

Idea. Estimate the number of values accounted for in phase T that *will be recounted* in phase $T+1$.

Q. How many such values?

A. *Half of them.*

. . . 0111110111111111111 ⟵ *counted*

. . . 0111111111111111111 ⟵ *will be counted again in the next phase*

If My_T values will be recounted on average then y_T satisfies $e^{-y_T/2^{T+1}} = 3/4$ and $y_T = 2^{T+1} \ln 4/3$

Lemma 1. Expected number of values in phase T that *will* be recounted is $M \cdot 2^{T+1} \cdot (\ln 4 - \ln 3)$

Lemma 2. Expected number of values in phase T that *will not* be recounted is $M \cdot 2^{T} \cdot (2 \ln 3 - \ln 4)$

$$\underbrace{M \cdot 2^T \cdot \ln 4}_{\text{total count (last slide)}} \quad - \quad \underbrace{M \cdot 2^{T+1} \cdot (\ln 4 - \ln 3)}_{\text{will be recounted (above)}} \quad = \quad M \cdot 2^T \cdot (2 \ln 3 - \ln 4)$$

HyperBit analysis (last phase)

Q. How many values need to be accounted for in the last (unfinished) phase ?

A. It depends on β (proportion of 0s in the sketch on termination).

Three observations complete the analysis

1. As usual, the algorithm accounts for ***Mx*** values, where $e^{-x/2^{T+1}} = \beta$ so $x = 2^{T+1}\ln(1/\beta)$

2. Add back the recount estimate $M \cdot 2^T \cdot (\ln 4 - \ln 3)$ from phase ***T−1*** (it is too high).

recount estimate for previous phase
 # values that generate half the 1s

3. Replace that estimate with ***My*** where $e^{-y/2^T} = 1 - (\frac{1-\beta}{2})$ so $y = 2^T \ln \frac{1+\beta}{2}$

Lemma 2. Expected # of values to count in the last phase is $M \cdot 2^T\left(\ln 4 - \ln 3 - 2\ln\beta + \ln\frac{1+\beta}{2}\right)$

HyperBit analysis final accounting

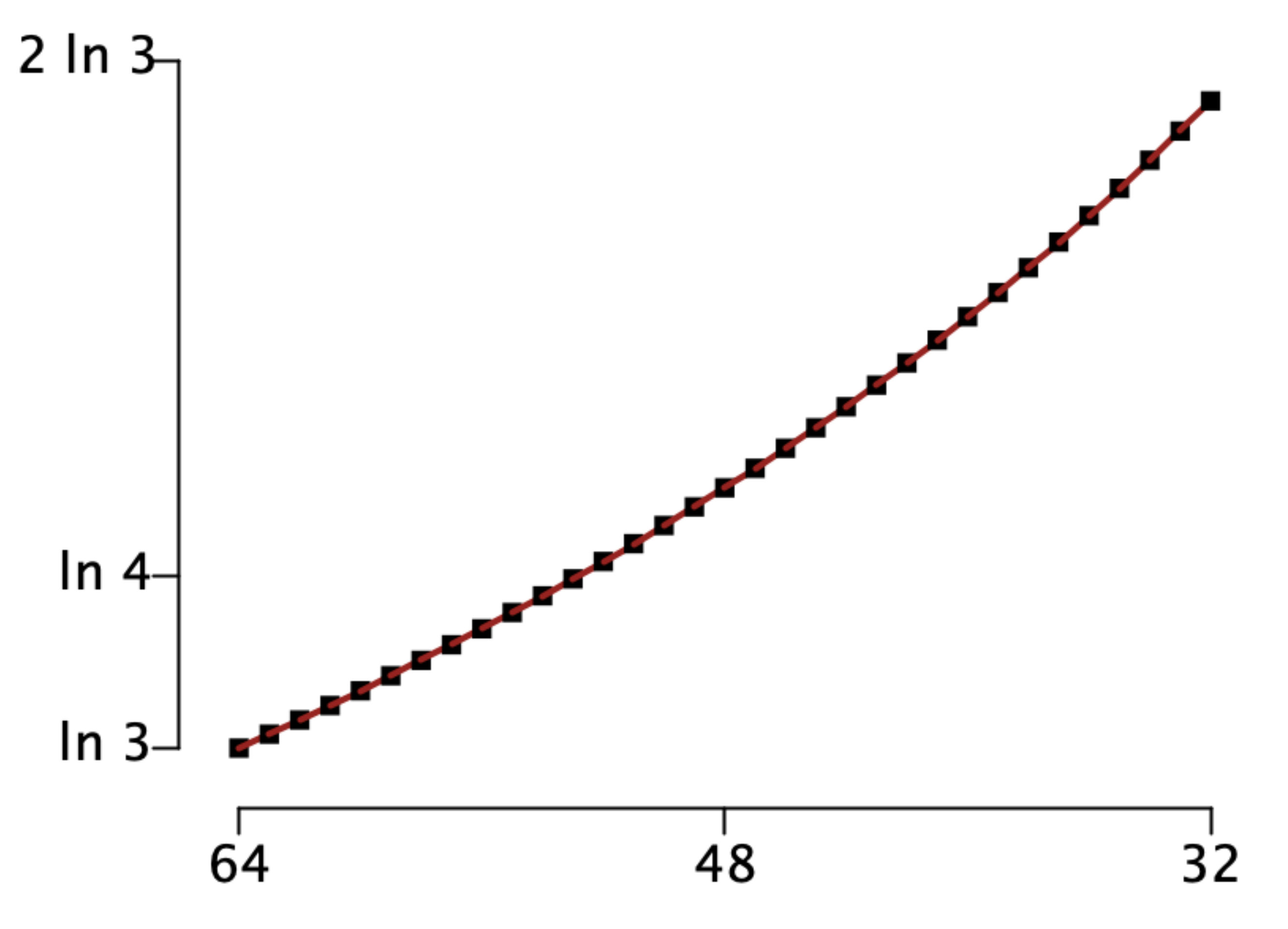
Expected # of values accounted for *in* phase ***T*** is $M \cdot 2^T \cdot (2\ln 3 - \ln 4)$

Expected # of values accounted for *before* phase ***T*** is $M \cdot 2^T \cdot (2\ln 3 - \ln 4)$ $\quad M(2\ln 3 - \ln 4)\sum_{0 \le i < T} 2^i$

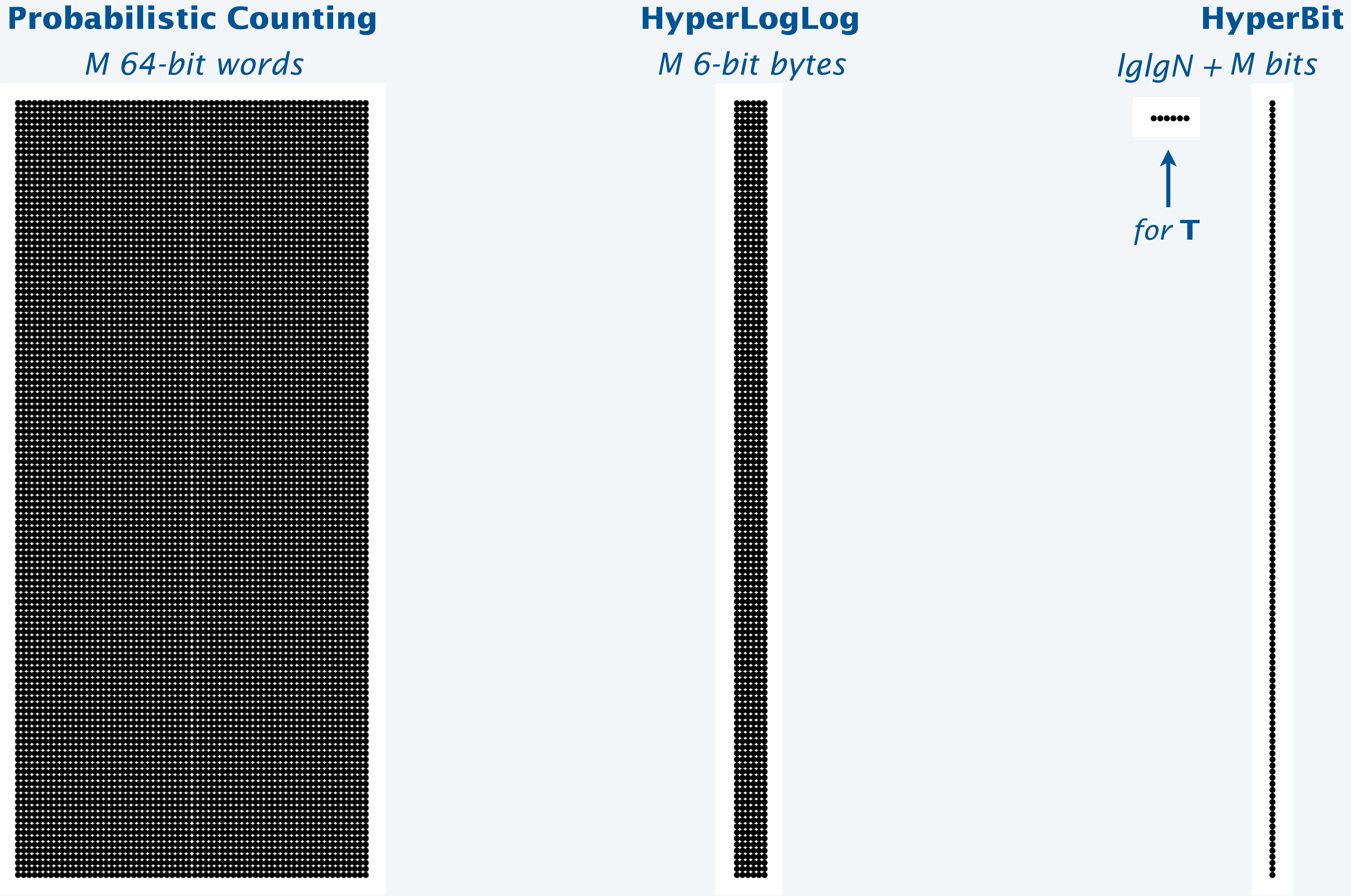
Expected # of values accounted for when T is the last (unfinished) phase is

$$M \cdot 2^T\left(\ln 4 - \ln 3 - 2\ln\beta + \ln\frac{1+\beta}{2}\right)$$

Theorem. The expected number of values seen when `HyperBit` terminates after completing ***T*** phases with βM 0s in sketch is $\sim M \cdot 2^T \cdot \big(\ln 3 - 2\ln\beta + \ln((1+\beta)/2)\big)$



Memory use for cardinality estimation algorithms



Pictured: M = 128

HyperBit accuracy hypothesis

Theorem. The expected number of values seen when `HyperBit` terminates after completing ***T*** phases with βM 0s in sketch is $\sim M \cdot 2^T \cdot \left(\ln 3 - 2\ln\beta + \ln((1+\beta)/2)\right)$

Conjecture. *The statistic is approximately Gaussian with variance* $\sigma^2 \sim c/M$ *where* $c \approx 1$.

suggested by experiments and history

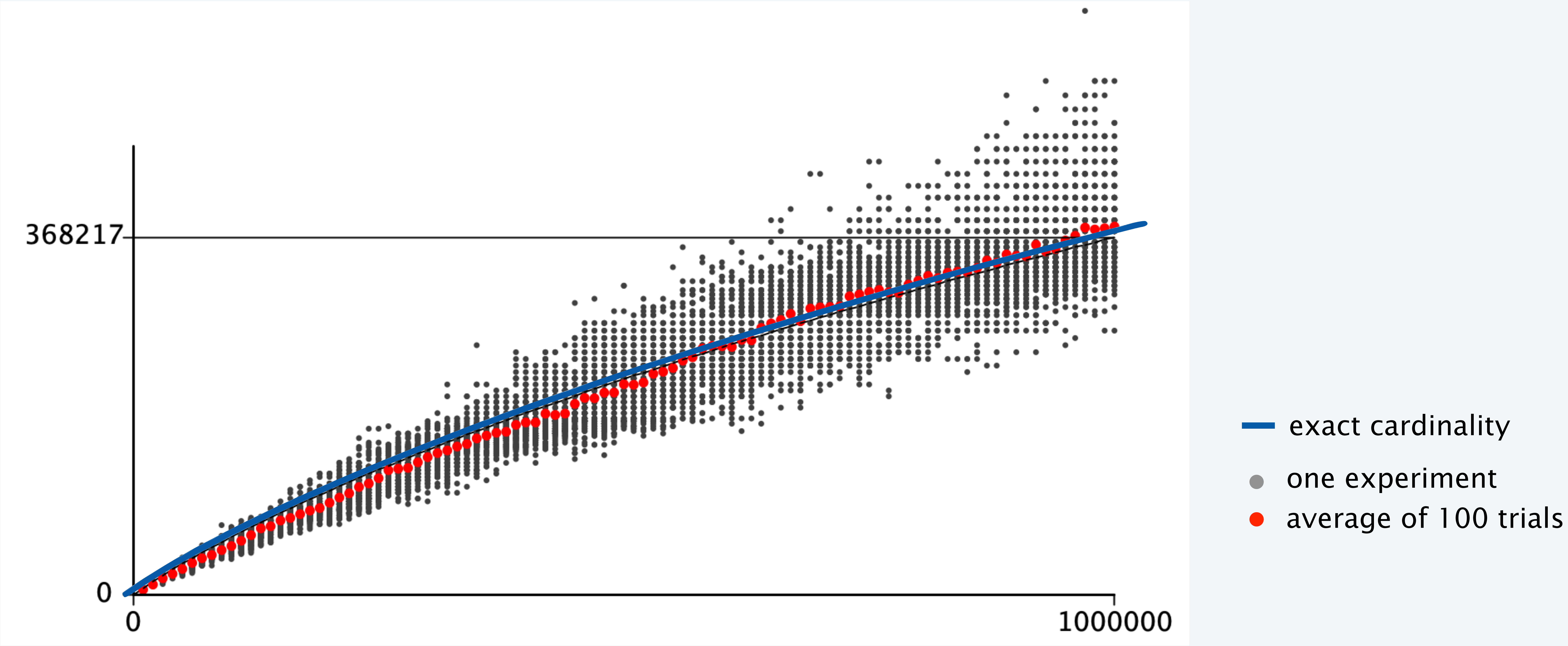
Hypothesis. *The reported estimate will be within* 3σ *of the actual count 99% of the time.*

Consequence. HyperBit *solves the practical cardinality estimation problem with 1030 bits.*

within 10% accuracy 99% of the time for $M = 1024$

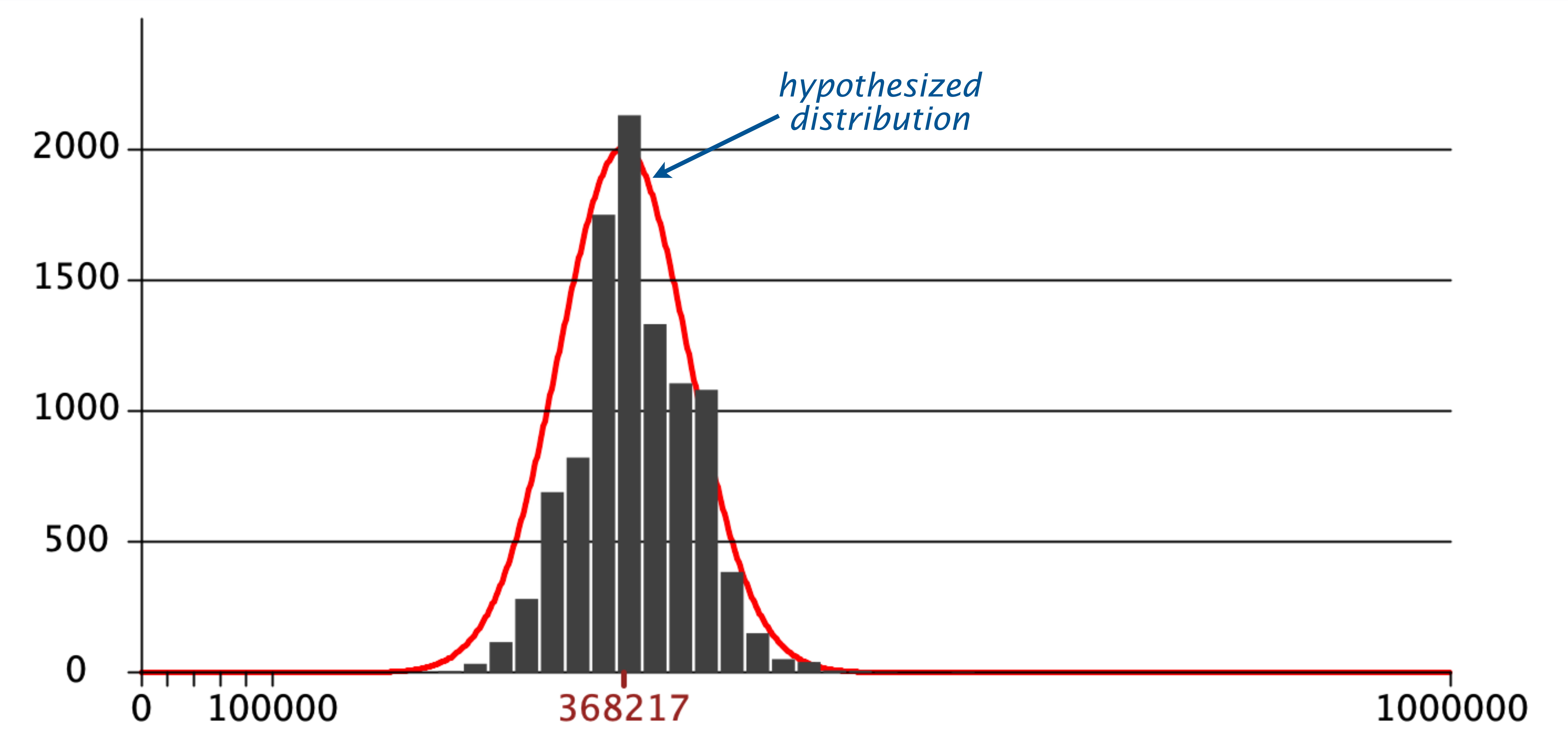
HyperBit validation I

Experiment. 100 trials for x*10000 inputs for x from 1 to 100 (10000 trials)



HyperBit validation II

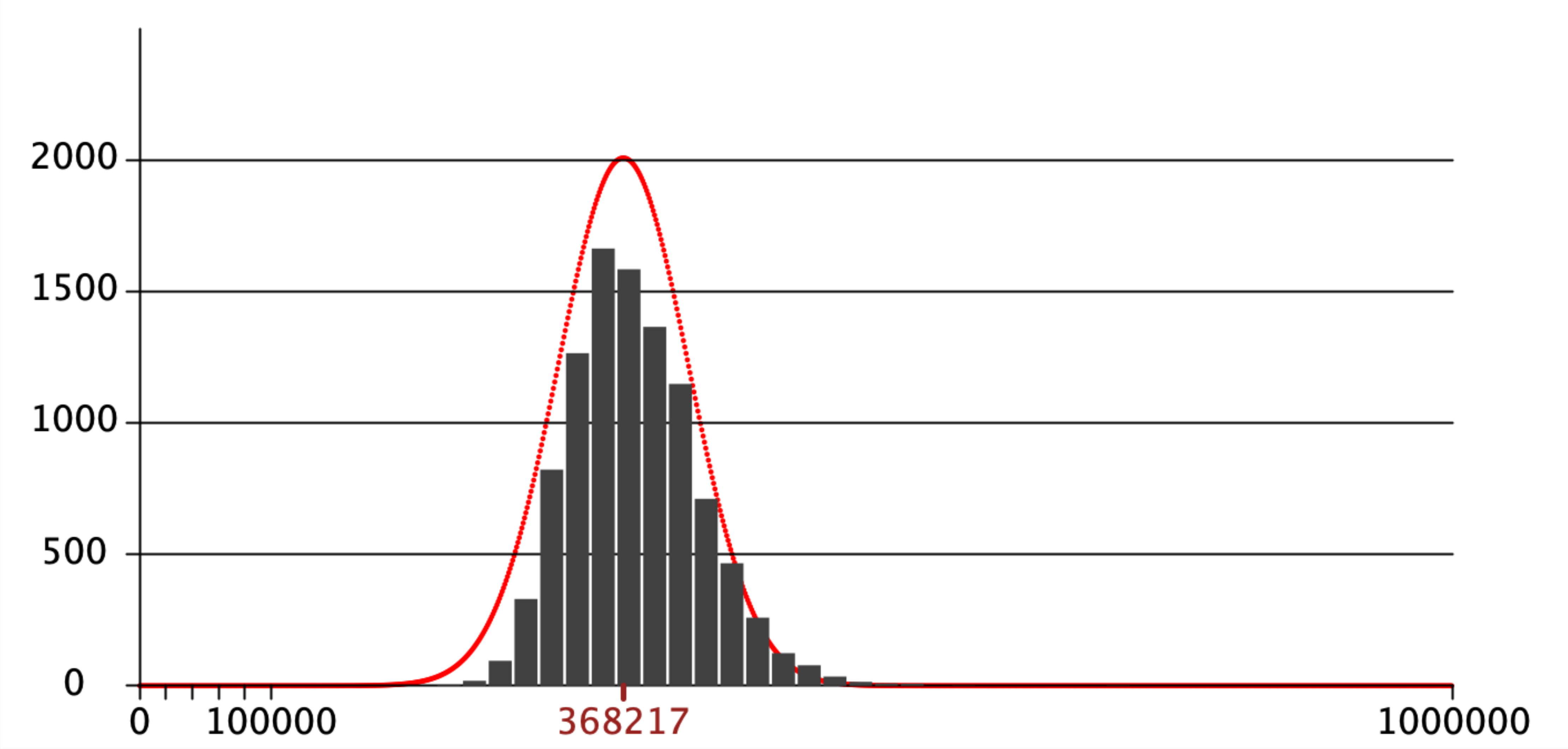
Experiment. 10000 trials for 1 million inputs



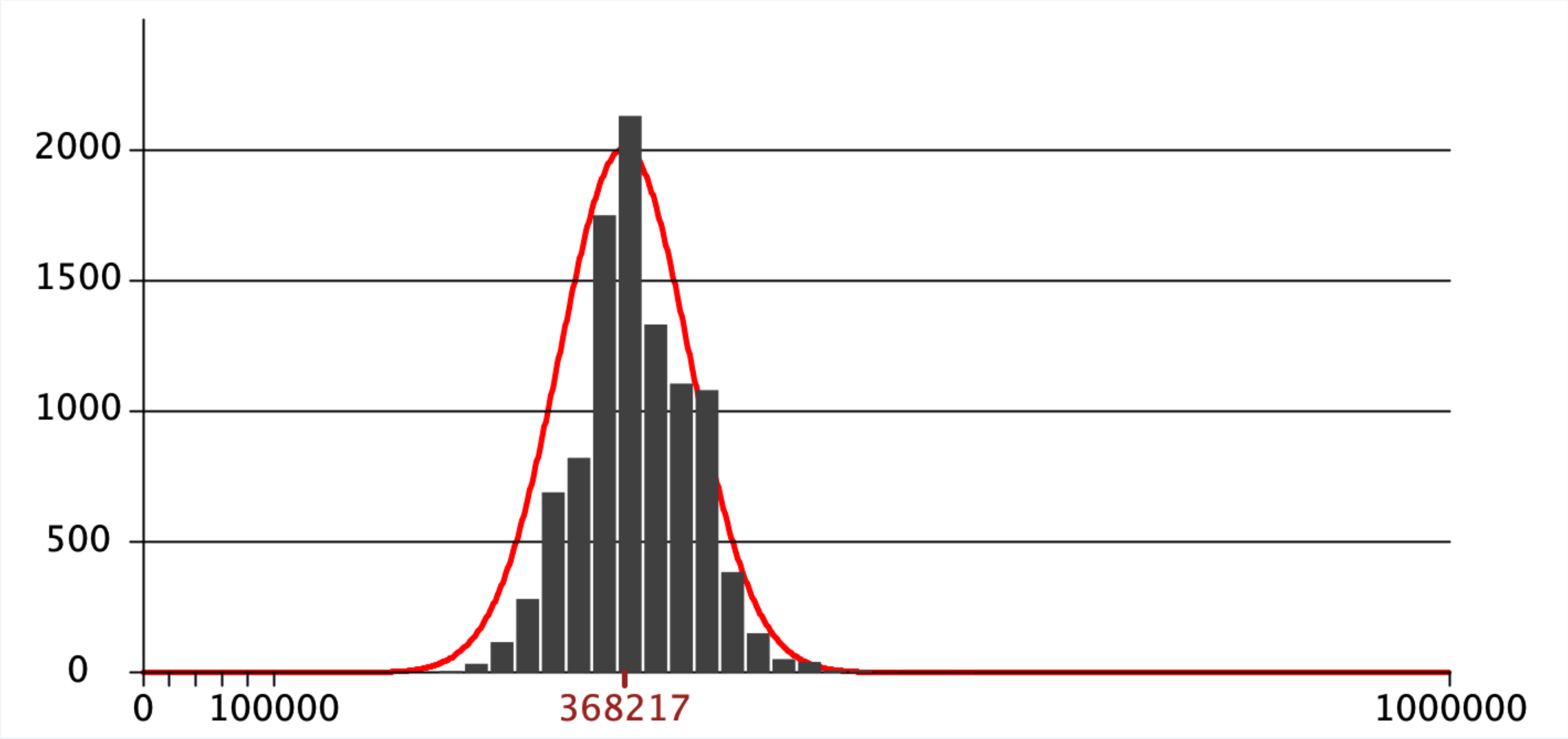
*Histogram of number of estimates between x*2000 and (x+1)*2000*

HyperBit vs. HyperLogLog

HLL



HB



Bottom line. Comparable accuracy with one-sixth as much memory.

Optimal ?

Pictured: M = 64

What's next?

Fully analyze relative accuracy of HyperBit

HyperBit vs HyperBitBit ?

Determine optimal values of parameters

Continue to validate results

Algorithm science for other streaming algorithms

HyperBit

A Memory-Efficient Alternative to HyperLogLog

Robert Sedgewick
Princeton University

with thanks to Jérémie Lumbroso and Svante Janson