Cardinality Estimation

Robert Sedgewick
Princeton University

with special thanks to Jérémie Lumbroso
Philippe Flajolet, mathematician and computer scientist extraordinaire

Philippe Flajolet 1948–2011
“PEOPLE WHO ANALYZE ALGORITHMS have double happiness. First of all they experience the sheer beauty of elegant mathematical patterns that surround elegant computational procedures. Then they receive a practical payoff when their theories make it possible to get other jobs done more quickly and more economically.”

– Don Knuth

Understood since Babbage: AofA is a *scientific* endeavor.

- Start with a working program (algorithm implementation).
- Develop mathematical model of its behavior.
- Use the *model* to formulate hypotheses on resource usage.
- Use the *program* to validate hypotheses.
- Iterate on basis of insights gained.
Analysis of Algorithms (present-day context)

**AofA**
- Theorems *and* code
- Precise math models
- Experiment, validate, iterate

**Theoretical computer science**
- Theorems
- Abstract math models
- Limited experimentation

**Practical computing**
- Real code on real machines
- Thorough validation
- Limited math models
Cardinality Estimation

- Warmup: exact cardinality count
- Probabilistic counting
- Stochastic averaging
- Refinements
- 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**

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
119.249.79.23

**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.
Standard “optimal” solution: Use a hash table

Hashing with linear probing
- Create a table of size $M$.
- Transform each value into a “random” table index.
- Move right to find space if value collides.
- Count values new to the table.

example: multiply by a prime, then take remainder after dividing by $M$.

<table>
<thead>
<tr>
<th>small example data stream</th>
<th>P J J E K J L C K O M T P G L J I F K C</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash values (x-'A')%97%17</td>
<td>15 6 6 14 1 6 13 7 1 15 8 7 15 4 13 6 11 9 1 7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>hash table (M = 17)</th>
<th>K</th>
<th>G</th>
<th>J</th>
<th>C</th>
<th>M</th>
<th>I</th>
<th>L</th>
<th>E</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Additional (key) idea. Keep searches short by doubling table size when it becomes half full.
Mathematical analysis of exact cardinality count with linear probing

**Theorem.** Expected time and space cost is linear.

**Proof.** Follows from classic Knuth Theorem 6.4.K.

**Theorem K.** The average number of probes needed by Algorithm 1, assuming that all \( M^N \) hash sequences (35) are equally likely, is

\[
C_N = \frac{1}{2} (1 + Q_0 (M, N-1)) \quad \text{(successful search), (40)}
\]

\[
C_N' = \frac{1}{2} (1 + Q_1 (M, N)) \quad \text{(unsuccessful search), (41)}
\]

where

\[
Q_r (M, N) = \binom{r}{0} + \binom{r+1}{1} \frac{N}{M} + \binom{r+2}{2} \frac{N(N-1)}{M^2} + \cdots
\]

\[
= \sum_{k \geq 0} \binom{r+k}{k} \frac{N(N-1)}{M} \cdots \frac{N-k+1}{M}. \quad (42)
\]

**Proof.** Details of the calculation are worked out in exercise 27. (For the variance, see exercises 28, 67, and 68.)

"I first formulated [this] derivation in 1962. Since this was the first nontrivial algorithm I had ever analyzed satisfactorily, it had a strong influence on the structure of these books. Ever since that day, the analysis of algorithms has in fact been one of the major themes of my life."

-- Knuth, TAOCP volume 3

---

Q. Do the hash functions that we use *uniformly* and *independently* distribute keys in the table?

A. Not likely.
Scientific validation of exact cardinality count with linear probing

**Hypothesis.** Time and space cost is *linear for the hash functions we use and the data we have.*

**Quick experiment.** Doubling the problem size should double the running time.

---

**Driver to read N strings and count distinct values**

```java
public static void main(String[] args) {
    int N = Integer.parseInt(args[0]);
    String stream = new StringStream(N);
    long start = System.currentTimeMillis();
    StdOut.println(count(stream));
    long now = System.currentTimeMillis();
    double time = (now - start) / 1000.0;
    StdOut.println(time + " seconds");
}
```

---

Q. Is hashing with linear probing effective?

A. Yes. Validated in countless applications for *over half a century.*

---

% java Hash 2000000 < log.07.f3.txt
483477
3.322 seconds

% java Hash 4000000 < log.07.f3.txt
883071
6.55 seconds

% java Hash 6000000 < log.07.f3.txt
1097944
9.49 seconds

---

% sort -u log.07.f3 | wc -l
1097944

**sort-based method takes about 3 minutes**
Complexity of exact cardinality count

Q. Does there exist an optimal algorithm for this problem?
A. Depends on definition of “optimal”.

**Guaranteed linear-time?** NO. Linearithmic lower bound.

**Guaranteed linearithmic?** YES. Balanced BSTs or mergesort.

**Linear-time with high probability assuming the existence of random bits?**
YES. Dynamic perfect hashing.

Dietzfelbinger, Karlin, Mehlhorn, Meyer auf der Heide, Rohnert, and Tarjan
*Dynamic Perfect Hashing: Upper and Lower Bounds*

**Within a small constant factor of the cost of touching the data in practice?**
YES. Hashing with linear probing.

M. Mitzenmacher and S. Vadhan
*Why Simple Hash Functions Work: Exploiting the Entropy in a Data Stream.*
*SODA* 2008.

**Hypothesis.** Hashing with linear probing is “optimal”.

*Note: uniformity of hash function affects only the running time (not the value computed).*
Exact cardinality count requires linear space

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

A. Bad news: You cannot get an exact count.

A. Good news: You can get an accurate estimate (stay tuned).
Cardinality Estimation

• Warmup: exact cardinality count
• Probabilistic counting
• Stochastic averaging
• Refinements
• Final frontier
Cardinality estimation is a fundamental problem with many applications *where memory is limited.*

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

**Constraints**
- 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**
- How many unique visitors to my website?
- Which sites are the most/least popular?
- How many different websites visited by each customer?
- How many different values for a database join?

To fix ideas on scope: Think of *millions* of streams each having *trillions* of values.
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

Bottom line: Quintessential example of the effectiveness of scientific approach to algorithm design.
PCSA first step: Use hashing

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.

Example: Java
- All data types implement a hashCode() method (though we often override the default).
- String data type stores value (computed once).

Bottom line: Do cardinality estimation on streams of (binary) integers.

String value = “gsearch.CS.Princeton.EDU”
int x = value.hashCode();

“Random” except for the fact that some values are equal.

20th century: use 32 bits (millions of values)
21st century: use 64 bits (quadrillions of values)
Initial hypothesis

**Hypothesis.** Uniform hashing assumption is reasonable in this context.

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

No problem!

- AofA is a scientific endeavor (we always validate hypotheses).
- 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 AofA.**
Experimenting to validate hypotheses is **WHAT WE DO!**
Probabilistic counting starting point: three integer functions

**Definition.** \( p(x) \) is the **number of 1s** in the binary representation of \( x \).

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

**Definition.** \( R(x) = 2^r(x) \)

<table>
<thead>
<tr>
<th></th>
<th>( p(x) )</th>
<th>( r(x) )</th>
<th>( R(x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>12</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>8</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>10</td>
<td>5</td>
<td>32</td>
</tr>
</tbody>
</table>

\[ x \]

\[ \sim x \]

\[ x + 1 \]

\[ \sim x \land (x + 1) \]

**Bit-whacking magic:**

\( R(x) \) is easy to compute.

**Exercise:** Compute \( p(x) \) as easily.

**Note:** \( r(x) = p(R(x) - 1) \).

**Bottom line:** \( p(x) \), \( r(x) \), and \( R(x) \) all can be computed with just a few machine instructions.
Probabilistic counting (Flajolet and Martin, 1983)

Maintain a single-word *sketch* that summarizes a data stream $x_0, x_1, \ldots, x_N, \ldots$

- For each $x_N$ in the stream, update sketch by *bitwise or* with $R(x_N)$.
- Use *position of rightmost 0* (with slight correction factor) to estimate $\lg N$.

### typical sketch

<table>
<thead>
<tr>
<th>sketch</th>
<th>$x_N$</th>
<th>$R(x_N)$</th>
<th>sketch $\mid R(x_N)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 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</td>
<td>0 0 1 1 0 1 0 1 0 1 1 1 1 1 1 1 0 1 0 1 0 1 0 1 0 1 0 0 0 1 1 1</td>
<td>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</td>
<td></td>
</tr>
<tr>
<td>31 30 29 28 27 26 25 24 23 22 20 9 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td>leading bits almost surely 0</td>
<td>trailing bits almost surely 1</td>
<td></td>
</tr>
</tbody>
</table>

Rough estimate of $\lg N$ is $r(\text{sketch})$.

Rough estimate of $N$ is $R(\text{sketch})$.  

$R(x) = 2^k$ with probability $1/2^k$

estimate of $\lg N$

correction factor needed (stay tuned)
Probabilistic counting trace

<table>
<thead>
<tr>
<th>( x )</th>
<th>( r(x) )</th>
<th>( R(x) )</th>
<th>sketch</th>
</tr>
</thead>
<tbody>
<tr>
<td>01100010011000111010011110111011</td>
<td>2</td>
<td>100</td>
<td>00000000000000000000000000100</td>
</tr>
<tr>
<td>0110011100100011111011111000101</td>
<td>1</td>
<td>10</td>
<td>00000000000000000000001110</td>
</tr>
<tr>
<td>0001000100011101100110111011100011</td>
<td>2</td>
<td>100</td>
<td>0000000000000000000000011101</td>
</tr>
<tr>
<td>01000100011101110010000111100111111</td>
<td>5</td>
<td>100000</td>
<td>000000000000000000000100110</td>
</tr>
<tr>
<td>01101000010110001011100010001000</td>
<td>0</td>
<td>1</td>
<td>00000000000000000000001001111</td>
</tr>
<tr>
<td>00110111101110000000101001101001</td>
<td>1</td>
<td>11</td>
<td>0000000000000000000000101111</td>
</tr>
<tr>
<td>0011010001100011101101011111111100</td>
<td>0</td>
<td>1</td>
<td>00000000000000000000001011111</td>
</tr>
<tr>
<td>00110000100010000100001011100101</td>
<td>1</td>
<td>10</td>
<td>0000000000000000000000101111</td>
</tr>
<tr>
<td>0001100001000100001011101011011100</td>
<td>3</td>
<td>1000</td>
<td>0000000000000000000001001111</td>
</tr>
<tr>
<td>00011001100110011110010000111111</td>
<td>6</td>
<td>1000000</td>
<td>00000000000000000000011011111</td>
</tr>
<tr>
<td>010001011100010001010110111111100</td>
<td>0</td>
<td>1</td>
<td>000000000000000000000001101111</td>
</tr>
</tbody>
</table>

\[ R(\text{sketch}) = 100000_2 \]
\[ = 16 \]
Probabilistic counting (Flajolet and Martin, 1983)

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

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

Maintain a `sketch` of the data

- A single word
- OR of all values of R(x) in the stream
- 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

\[ \lg(\phi N) + P(\lg N) + o(1) \text{ where } \phi \doteq 0.77351 \]

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 AC

Kirschenhofer, Prodinger, and Szpankowski

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

In other words. In PC code, \( R(\text{sketch})/0.77351 \) 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.
Cardinality Estimation

- Rules of the game
- Probabilistic counting
- Stochastic averaging
- Refinements
- Final frontier
Stochastic averaging

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

Key point: equal values all go to the same stream
## PCSA trace

**Use initial m bits for second hash**

<table>
<thead>
<tr>
<th>( x )</th>
<th>( R(x) )</th>
<th>( \text{sketch}[0] )</th>
<th>( \text{sketch}[1] )</th>
<th>( \text{sketch}[2] )</th>
<th>( \text{sketch}[3] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010011110111011</td>
<td>100</td>
<td>0000000000000000</td>
<td>000000000000000100</td>
<td>000000000000000000</td>
<td>000000000000000000</td>
</tr>
<tr>
<td>0001111100000010</td>
<td>10</td>
<td>0000000000000000</td>
<td>000000000000000100</td>
<td>000000000000000000</td>
<td>000000000000000000</td>
</tr>
<tr>
<td>0110110110110011</td>
<td>100</td>
<td>0000000000000000</td>
<td>000000000000000100</td>
<td>000000000000000000</td>
<td>000000000000000000</td>
</tr>
<tr>
<td>0000000111011111</td>
<td>100000</td>
<td>000000000000</td>
<td>000000000000000100</td>
<td>000000000000000000</td>
<td>000000000000000000</td>
</tr>
<tr>
<td>0101110001000100</td>
<td>1</td>
<td>0000000000000000</td>
<td>000000000000000101</td>
<td>000000000000000000</td>
<td>000000000000000000</td>
</tr>
<tr>
<td>0000101001010101</td>
<td>10</td>
<td>0000000000000000</td>
<td>000000000000000100</td>
<td>000000000000000000</td>
<td>000000000000000000</td>
</tr>
<tr>
<td>101010111111100</td>
<td>1</td>
<td>0000000000000000</td>
<td>000000000000000101</td>
<td>000000000000000000</td>
<td>000000000000000000</td>
</tr>
<tr>
<td>0001011100110111</td>
<td>100</td>
<td>0000000000000000</td>
<td>000000000000000100</td>
<td>000000000000000000</td>
<td>000000000000000000</td>
</tr>
<tr>
<td>1110010000011111</td>
<td>100000</td>
<td>0000000000000000</td>
<td>000000000000000100</td>
<td>000000000000000000</td>
<td>000000000000000000</td>
</tr>
<tr>
<td>1010110011111101</td>
<td>10</td>
<td>0000000000000000</td>
<td>000000000000000101</td>
<td>000000000000000000</td>
<td>000000000000000000</td>
</tr>
<tr>
<td>0001110100110000</td>
<td>1</td>
<td>0000000000000000</td>
<td>000000000000000101</td>
<td>000000000000000000</td>
<td>000000000000000000</td>
</tr>
</tbody>
</table>

\( r(\text{sketch}[i]) \)

<table>
<thead>
<tr>
<th>2</th>
<th>1</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000101011</td>
<td>000000000000000101</td>
<td>000000000000000111</td>
<td>0000000000100000</td>
</tr>
</tbody>
</table>
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(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 averaging**
- Use second hash to split into $M = 2^m$ independent streams
- Use PC on each stream, yielding $2^m$ sketches.
- Compute $\text{mean} = \text{average } \# \text{trailing 1 bits in the sketches.}$
- Return $2^{\text{mean} / .77351}$.

**Flajolet-Martin 1983**

Q. Accuracy improves as $M$ increases.

Q. How much?

Theorem (paraphrased to fit context of this talk). Under the uniform hashing assumption, PCSA
- Uses $64M$ bits.
- Produces estimate with a relative accuracy close to $0.78 / \sqrt{M}$. 
Validation of PCSA analysis

Hypothesis. Value returned is accurate to $0.78/\sqrt{M}$ for random values from a large range.

Experiment. 1,000,000 31-bit random values, $M = 1024$ (10 trials)

% java PCSA 1000000 31 1024 10
964416
997616
959857
1024303
972940
985534
998291
996266
959208
1015329
Space-accuracy tradeoff for probabilistic counting with stochastic averaging

Relative accuracy: \( \frac{0.78}{\sqrt{M}} \)

**Bottom line.**
- Attain 10% relative accuracy with a sketch consisting of 64 words.
- Attain 2.4% relative accuracy with a sketch consisting of 1024 words.
Scientific 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).

log.07.f3.txt

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

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.

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

A poster child for AofA/AC
Small sample of work on related problems

<table>
<thead>
<tr>
<th>Year</th>
<th>Authors</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1970</td>
<td>Bloom</td>
<td>set membership</td>
</tr>
<tr>
<td>1984</td>
<td>Wegman</td>
<td>unbiased sampling estimate</td>
</tr>
<tr>
<td>1996–</td>
<td>many authors</td>
<td>refinements (stay tuned)</td>
</tr>
<tr>
<td>2000</td>
<td>Indyk</td>
<td>L1 norm</td>
</tr>
<tr>
<td>2004</td>
<td>Cormode-Muthukrishnan</td>
<td>frequency estimation deletion and other operations</td>
</tr>
<tr>
<td>2005</td>
<td>Giroire</td>
<td>fast stream processing</td>
</tr>
<tr>
<td>2012</td>
<td>Lumbroso</td>
<td>full range, asymptotically unbiased</td>
</tr>
<tr>
<td>2014</td>
<td>Helmi-Lumbroso-Martinez-Viola</td>
<td>uses neither sampling nor hashing</td>
</tr>
</tbody>
</table>
Cardinality Estimation

- Rules of the game
- Probabilistic counting
- Stochastic averaging
- Refinements
- 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 $2/c$.*

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
Interesting quote

“Flajolet and Martin [assume] that one may use in the algorithm an explicit family of hash functions which exhibits some ideal random properties. Since we are not aware of the existence of such a family of hash functions ...”

– Alon, Matias, and Szegedy

No! They hypothesized that practical hash functions would be as effective as random ones. They then validated that hypothesis by proving tight bounds that match experimental results.

Points of view re hashing

• **Theoretical computer science.** Uniform hashing assumption is not proved.
• **Practical computing.** Hashing works for many common data types.
• **AofA.** Extensive experiments have validated precise analytic models.

Points of view re random bits

• **Theoretical computer science.** Axiomatic that random bits exist.
• **Practical computing.** No, they don’t! And randomized algorithms are inconvenient, btw.
• **AofA.** More effective path forward is to validate precise analysis even if stronger assumptions are needed.
logs and loglogs

To improve space-time tradeoffs, we need to carefully count bits.

Relevant quantities
- $N$ is the number of items in the data stream.
- $\lg N$ is the number of bits needed to represent numbers less than $N$ in binary.
- $\lg \lg N$ is the number of bits needed to represent numbers less than $\lg N$ in binary.

For real-world applications
- $N$ is less than $2^{64}$.
- $\lg N$ is less than 64.
- $\lg \lg N$ is less than 8.

Typical PCSA implementations
- Could use $M \lg N$ bits, in theory.
- Use 64-bit words to take advantage of machine-language efficiencies.
- Use (therefore) $64 \times 64 = 4096$ bits with $M = 64$ (for 10% accuracy with $N < 2^{64}$).
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.

**Theorem** (paraphrased to fit context of this talk).

*With strongly universal hashing, there exists an algorithm that*

- Uses $O(M \log \log N)$ bits. 
  
  **PCS A uses $M \log N$ 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)

**Durand and Flajolet**

_LogLog Counting of Large Cardinalities_
ESA 2003; LNCS volume 2832.

Contributions (independent of BYJKST)
- Presents LogLog algorithm, an easy variant of PCSA
- Improves space-accuracy tradeoff *without* extra expense per value
- Full analysis, fully validated with experimentation

**Theorem** (paraphrased to fit context of this talk).

*Under the uniform hashing assumption, LogLog*
- Uses $M \lg \lg N$ bits.
- Achieves relative accuracy close to $1.30 / \sqrt{M}$.

**PCSA** saves sketches ($\lg N$ bits each)

00000000000000000000000000001101111

LogLog saves $r()$ values ($\lg \lg N$ bits each)

00100 ($ = 4$)

Not much impact on cardinality estimation in practice *only because*
- PCSA was effectively deployed in practical systems
- Idea led to a better algorithm a few years later (stay tuned)
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(x, M);
        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) (alpha * M * M / sum);
}

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

Theorem (paraphrased to fit context of this talk).
Under the uniform hashing assumption, **HyperLogLog**
- Uses \( M \log \log N \) bits.
- Achieves relative accuracy close to \( 1.02/\sqrt{M} \).
Space-accuracy tradeoff for HyperLogLog

Relative accuracy: \( \frac{1.02}{\sqrt{M}} \)

Bottom line (for \( N < 2^{64} \)).
- Attain 10\% relative accuracy with a sketch consisting of \( 108 \times 6 = 648 \) bits.
- Attain 3.1\% relative accuracy with a sketch consisting of \( 1024 \times 6 = 6144 \) bits.
Typical PCSA implementations
- Could use $M \ lg \ N$ bits, in theory.
- Use 64-bit words to take advantage of machine-language efficiencies.
- Use (therefore) $64 \times 64 = 4096$ bits with $M = 64$ (for 10% accuracy with $N < 2^{64}$).

Typical Hyperloglog implementations
- Could use $M \ lg \ lg \ N$ bits, in theory.
- Use 8-bit bytes to take advantage of machine-language efficiencies.
- Use (therefore) $108 \times 8 = 864$ bits with $M = 108$ (for 10% accuracy with $N < 2^{64}$).
Validation of Hyperloglog

S. Heule, M. Nunkesser and A. Hall

Extending Database Technology/International Conference on Database Theory 2013.
Philippe Flajolet, mathematician, data scientist, and computer scientist extraordinaire

Philippe Flajolet 1948–2011
Cardinality Estimation

- Rules of the game
- Probabilistic counting
- Stochastic averaging
- Refinements
- 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*

\[ \text{loglog} N \text{ improvement possible} \]

**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 \( O(M) \) bits.
- Achieves relative accuracy \( O(1/\sqrt{M}) \).

Unlikely to have impact on cardinality estimation in practice
- Tough to beat HyperLogLog’s low stream-processing expense.
- Constants hidden in \( O \)-notation not likely to be \(< 6\)
- No validation
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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Machine instructions per stream element</th>
<th>Memory bound</th>
<th>Memory bound when $N &lt; 2^{64}$</th>
<th># bits for 10% accuracy when $N &lt; 2^{64}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>HyperLogLog</td>
<td>20–30</td>
<td>$M \log \log N$</td>
<td>$6M$</td>
<td>648</td>
</tr>
<tr>
<td>BetterAlgorithm</td>
<td><em>a few dozen</em></td>
<td><em>a few hundred</em></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Also, results need to be validated through extensive experimentation.
A proposal: HyperBitBit (Sedgewick, 2016)

public static long estimate(Iterable<String> stream, int M) {
    int lgN = 5;
    long sketch = 0;
    long sketch2 = 0;
    for (String x : stream) {
        long x = hash(s);
        int k = hash2(x, 64);
        if (r(x) > lgN) sketch = sketch | (1L << k);
        if (r(x) > lgN + 1) sketch2 = sketch2 | (1L << k);
        if (p(sketch) > 31) {
            sketch = sketch2; lgN++; sketch2 = 0;
        }
    }
    return (int) (Math.pow(2, lgN + 5.4 + p(sketch)/32.0));
}

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

Q. Does this even work?
Initial experiments

**Exact values for web log example**

| % java Hash 1000000 < log.07.f3.txt |
| % java Hash 2000000 < log.07.f3.txt |
| % java Hash 4000000 < log.07.f3.txt |
| % java Hash 6000000 < log.07.f3.txt |

<table>
<thead>
<tr>
<th>1,000,000</th>
<th>2,000,000</th>
<th>4,000,000</th>
<th>6,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact</td>
<td>242,601</td>
<td>483,477</td>
<td>883,071</td>
</tr>
<tr>
<td>HyperBitBit</td>
<td>234,219</td>
<td>499,889</td>
<td>916,801</td>
</tr>
<tr>
<td>ratio</td>
<td>1.05</td>
<td>1.03</td>
<td>0.96</td>
</tr>
</tbody>
</table>

**HyperBitBit estimates**

| % java HyperBitBit 1000000 < log.07.f3.txt |
| % java HyperBitBit 2000000 < log.07.f3.txt |
| % java HyperBitBit 4000000 < log.07.f3.txt |
| % java HyperBitBit 6000000 < log.07.f3.txt |

Conjecture. On practical data, **HyperBitBit**, for $N < 2^{64}$,

- Uses $128 + 6$ bits.
- Estimates cardinality within 10% of the actual.

**Next steps.**

- Analyze.
- Experiment.
- Iterate
## Summary/timeline for cardinality estimation

<table>
<thead>
<tr>
<th>Year</th>
<th>Authors</th>
<th>Algorithm</th>
<th>Hashing Assumption</th>
<th>Feasible and Validated?</th>
<th>Memory Bound (bits)</th>
<th>Relative Accuracy Constant</th>
<th># Bits for 10% Accuracy when $N &lt; 2^{64}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1985</td>
<td>Flajolet-Martin</td>
<td>PCSA</td>
<td>uniform</td>
<td>✓</td>
<td>$M \log N$</td>
<td>0.78</td>
<td>4096</td>
</tr>
<tr>
<td>1996</td>
<td>Alon–Matias–Szegedy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[theorem]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td>Durand–Flajolet</td>
<td>LogLog</td>
<td>uniform</td>
<td>✓</td>
<td>$M \lg \lg N$</td>
<td>1.30</td>
<td>1536</td>
</tr>
<tr>
<td>2007</td>
<td>Flajolet–Fusy–Gandouet–Meunier</td>
<td>HyperLogLog</td>
<td>uniform</td>
<td>✓</td>
<td>$M \lg \lg N$</td>
<td>1.04</td>
<td>648</td>
</tr>
<tr>
<td>2010</td>
<td>Kane–Nelson–Woodruff</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[theorem]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2018+</td>
<td>RS–?</td>
<td>HyperBitBit</td>
<td>uniform</td>
<td>✓ (?)</td>
<td>$2M + \lg \lg N$</td>
<td>?</td>
<td>134 (?)</td>
</tr>
</tbody>
</table>
Happy Birthday, Don!
Cardinality Estimation

Robert Sedgewick
Princeton University

with special thanks to Jérémie Lumbroso