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)

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

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

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

Example. 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

$UNIX (1970s-present)$

% 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

6 million strings

State of the art in the wild for decades. Sort, then count.
Warmup: exact cardinality count using linear probing

Hashing with linear probing

- Compute a hash function that transforms data value into a table index.
- Scan to find value or an empty slot.
- Keep table less than half full by resizing.

<table>
<thead>
<tr>
<th>cnt</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
</table>
Warmup: exact cardinality count using a hash table

Hashing with linear probing
- Compute a *hash function* that transforms data value into a table index.
- Scan to find value or an empty slot.
- Keep table less than half full by resizing.

Textbook method, implemented in most modern programming environments.

Exact cardinality count in Java
- Input is an “iterable”
- HashSet implements linear probing
- add() adds new value (noop if already there)
- size() gives number of distinct values added

```java
public static long count(Iterable<String> stream) {
    HashSet<String> hset = new HashSet<String>();
    for (String x : stream) {
        hset.add(x);
    }
    return hset.size();
}
```
Mathematical analysis of exact cardinality count with linear probing

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

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

---

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

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

\[
C'_N = \frac{1}{2} \left(1 + Q_1(M, N)\right) \quad \text{(unsuccessful search),}
\]

where

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

\[
= \sum_{k=0}^{\infty} \binom{r+k}{k} \cdot \frac{N}{M} \cdot \frac{N-1}{M} \cdot \frac{N-k+1}{M}.
\]

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

---

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]);
    StringStream 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”.

<table>
<thead>
<tr>
<th>Guaranteed linear-time?</th>
<th>NO. Linearithmic lower bound.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guaranteed linearithmic?</td>
<td>YES. Balanced BSTs or mergesort.</td>
</tr>
</tbody>
</table>

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

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

**Hypothesis.** Standard 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).

109.108.229.102
tel-71-104-94-246.lsanca.dsl-w.verizon.net
117.222.48.163
tel-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
tel-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.Inse3.cht.bigpond.net.au

**Typical modern applications**

- Can only afford to process each value *once*.  
- *Estimate* of count still very useful.
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.
Probabilistic counting with stochastic averaging (PCSA)

Flajolet and Martin
*Probabilistic Counting Algorithms for Data Base Applications*

**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 AoFoA.
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.

```
011110001001111101110001111001000
011110010011111011100111001000
011101010110110110000000011011010
001101000011111001001111101100111010
00010000111001101100011101110110100111
0000100101110111000000010100100111
00001100101111000000011100011101110111
```

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

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

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

Current Java default is 32-bit int value
Initial hypothesis

**Hypothesis.** Knuth’s assumption about hash functions 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. \( R(x) = 2^{r(x)} \)

**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 $\log N$.

**typical sketch**

$N = 10^6$

<table>
<thead>
<tr>
<th>$x_N$</th>
<th>$R(x_N)$</th>
<th>sketch</th>
<th>$\log N$ estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 1 0 1 1 1 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0</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</td>
<td>20</td>
<td>16</td>
</tr>
</tbody>
</table>

Rough estimate of $\log N$ is $r(sketch)$.

Rough estimate of $N$ is $R(sketch)$.

*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>01100010011001111010011111011101110</td>
<td>2</td>
<td>100</td>
<td>0000000000000000000000000000100</td>
</tr>
<tr>
<td>01100111001000110011111000000101</td>
<td>1</td>
<td>10</td>
<td>000000000000000000000000000010</td>
</tr>
<tr>
<td>000100010001111000110110110110011</td>
<td>2</td>
<td>100</td>
<td>0000000000000000000000000000110</td>
</tr>
<tr>
<td>010001000111011110000000111011111</td>
<td>5</td>
<td>100000</td>
<td>0000000000000000000000000000110</td>
</tr>
<tr>
<td>011010000010110010111000100100100</td>
<td>0</td>
<td>1</td>
<td>0000000000000000000000000000100</td>
</tr>
<tr>
<td>00110111101100000000101001010101</td>
<td>1</td>
<td>10</td>
<td>0000000000000000000000000000111</td>
</tr>
<tr>
<td>00110100011001110101011111111000</td>
<td>0</td>
<td>1</td>
<td>0000000000000000000000000000111</td>
</tr>
<tr>
<td>000110000100001000010111001011011</td>
<td>3</td>
<td>1000</td>
<td>0000000000000000000000000000111</td>
</tr>
<tr>
<td>000110011001100111100010000111111</td>
<td>6</td>
<td>100000</td>
<td>0000000000000000000000000000110</td>
</tr>
<tr>
<td>01000101110001001010110011111100</td>
<td>0</td>
<td>1</td>
<td>0000000000000000000000000000111</td>
</tr>
</tbody>
</table>

$R(sketch) = 100000_2$

$= 16$
Probabilistic counting (first try)

```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);
}
```

Maintain a *sketch* of the data
- A single word
- OR of all values of R(x) in the stream
Estimate is smallest value not seen.

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

Q. (Martin) Seems a bit low. How much?

A. (unsatisfying) Obtain empirically.
Probabilistic counting (Flajolet and Martin)

```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
Estimate is smallest value not seen.

with correction for bias

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

Q. Value of correction factor? How accurate?

A. (Flajolet) Analytic combinatorics!

Magic is something you make
Mathematical analysis of probabilistic counting

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

\[ \lg(\phi N) + P(\lg N) + o(1) \quad \text{where } \phi \equiv .77351 \]

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

*Analysis of a splitting process arising in probabilistic counting and other related algorithms*


**In other words.** In PC code, \( R(\text{sketch})/.77351 \) 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.
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>101011011110111011</td>
<td>100</td>
<td>0000000000000000000000000000000000</td>
<td>0000000000000000000000000000000000</td>
<td>100</td>
<td>0000000000000000000000000000000000</td>
</tr>
<tr>
<td>001110010000001011</td>
<td>10</td>
<td>0000000000000000000000000000000000</td>
<td>0000000000000000000000000000000000</td>
<td>10</td>
<td>0000000000000000000000000000000000</td>
</tr>
<tr>
<td>010110101101101101</td>
<td>100</td>
<td>0000000000000000000000000000000000</td>
<td>0000000000000000000000000000000000</td>
<td>100</td>
<td>0000000000000000000000000000000000</td>
</tr>
<tr>
<td>0000000111011111</td>
<td>1000000</td>
<td>0000000000000000000000000000000000</td>
<td>0000000000000000000000000000000000</td>
<td>1000000</td>
<td>0000000000000000000000000000000000</td>
</tr>
<tr>
<td>0101110001000100</td>
<td>1</td>
<td>0000000000000000000000000000000000</td>
<td>0000000000000000000000000000000000</td>
<td>1000000</td>
<td></td>
</tr>
<tr>
<td>000001001001010101</td>
<td>10</td>
<td>0000000000000000000000000000000000</td>
<td>0000000000000000000000000000000000</td>
<td>1000000</td>
<td></td>
</tr>
<tr>
<td>1010101111111100</td>
<td>1</td>
<td>0000000000000000000000000000000000</td>
<td>0000000000000000000000000000000000</td>
<td>1000000</td>
<td></td>
</tr>
<tr>
<td>000101101101011011</td>
<td>1000000</td>
<td>0000000000000000000000000000000000</td>
<td>0000000000000000000000000000000000</td>
<td>1000000</td>
<td></td>
</tr>
<tr>
<td>1110010000111111</td>
<td>1000000</td>
<td>0000000000000000000000000000000000</td>
<td>0000000000000000000000000000000000</td>
<td>1000000</td>
<td></td>
</tr>
<tr>
<td>1010110011111101</td>
<td>10</td>
<td>0000000000000000000000000000000000</td>
<td>0000000000000000000000000000000000</td>
<td>1000000</td>
<td></td>
</tr>
<tr>
<td>0001110100110100</td>
<td>1</td>
<td>0000000000000000000000000000000000</td>
<td>0000000000000000000000000000000000</td>
<td>1000000</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>2</th>
<th>1</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000000000101101</td>
<td>0000000000000000000000000000000000</td>
<td>0000000000000000000000000000000000</td>
<td>0000000000000000000000000000000000</td>
</tr>
</tbody>
</table>

\( M = 4 \)
Probabilistic counting with stochastic averaging in Java

```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(x);
    }
    int sum = 0;
    for (int k = 0; k < M; k++)
        sum += r(sketch[k]);
    double mean = 1.0 * sum / M;
    double phi = .77351;
    return (int) (M * Math.pow(2, mean)/phi);
}
```

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

*Flajolet-Martin 1983*
Theoretical analysis of PCSA

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

**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** $0.78/\sqrt{M}$.

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

$$E[\beta^n] = n^{1/q}(\delta_q + \psi_n(\log_2 n)) + o(n^{1/q}).$$

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

$$\Pr(N_1 = k) = \binom{n}{k} p^k q^{n-k},$$

and taking logarithms of (30), for $k = pn + \delta$ and $\delta \ll n$, one finds

$$\Pr(N_1 = pn + \delta) = \exp \left( -\frac{\delta^2 + O(\delta)}{2npq} + O\left(\frac{\delta^3}{n^2}\right)\right).$$

If $\delta = \sqrt{n \log n}$, the probability (30) is exponentially small. We conclude the proof by observing that the binomial distribution is unimodal and

$$\Pr \left( \bigcup_{1 \leq i < m} \left| N_i - \frac{n}{m} \right| > \sqrt{n \log n} \right) < m \Pr \left( \left| N_1 - \frac{n}{m} \right| > \sqrt{n \log n} \right).$$

We can now conclude the proof of the first part of Theorem 4. Let $S$ denote the sum $R^{(1)} + R^{(2)} + \cdots + R^{(m)}$. We have

$$\Pr(S = k) = \sum_{n_1 + n_2 + \cdots + n_m = n} \frac{1}{m!} p_{n_1, k_1} p_{n_2, k_2} \cdots p_{n_m, k_m}.$$
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)
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).

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

<1% larger than actual value

Q. Is PCSA effective?

A. ABSOLUTELY!

log.07.f3.txt

109.108.229.102
pool-71-104-94-246.lsanco.dsl-w.verizon.net
117.222.48.163
pool-71-104-94-246.lsanco.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.lsanco.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
Summary: PCSA (Flajolet-Martian, 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
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

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

Replaces “uniform hashing” assumption with “random bit existence” assumption
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.** 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.
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.  
  *PCSA 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 $1.30/\sqrt{M}$.

$\lg N$ bits can save a value (PCSA)
$\lg \lg N$ bits can save a bit index in a value

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): LogLog algorithm (2003)

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);
    }
    int sum = 0;
    for (int k = 0; k < M; k++)
        sum += Bits.r(sketch[k]);
    double mean = 1.0 * sum / M;
    double alpha = .77351;
    return (int) (alpha * M * Math.pow(2, mean + 1.0));
}

Idea. Keep track of min(r(x)) (with stochastic averaging).

Durand and Flajolet
LogLog Counting of Large Cardinalities
ESA 2003; LNCS volume 2832.
We can do better (in theory and in practice): HyperLogLog algorithm (2007)

```java
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. Use Harmonic mean.

Flajolet-Fusy-Gandouet-Meunier 2007

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 $1.02/\sqrt{M}$. 

Flajolet, Fusy, Gandouet, and Meunier

Space-accuracy tradeoff for HyperLogLog

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

Bottom line (for \( N < 2^{64} \)).
- Attain 12.5% relative accuracy with a sketch consisting of \( 64 \times 6 = 396 \) bits.
- Attain 3.1% relative accuracy with a sketch consisting of \( 1024 \times 6 = 6144 \) bits.
Validation of Hyperloglog

Practical Data Science - Amazon Announces HyperLogLog

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.*
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*

Lower bound

**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*

Theoretical Computer Science
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></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><strong>HyperLogLog</strong></td>
<td>20–30</td>
<td>$M \log\log N$</td>
<td>$6M$</td>
<td>768</td>
</tr>
<tr>
<td><strong>BetterAlgorithm</strong></td>
<td><em>a few dozen</em></td>
<td></td>
<td></td>
<td><em>a few hundred</em></td>
</tr>
</tbody>
</table>

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

```java
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.
- \(\text{lgN}\) is estimate of \(\log N\)
- sketch is 64 indicators whether to increment \(\text{lgN}\)
- sketch2 is is 64 indicators whether to increment \(\text{lgN}\) by 2
- Update when half the bits in sketch are 1

Bias (determined empirically)
Initial experiments

**Exact values for web log example**

<table>
<thead>
<tr>
<th>% java Hash</th>
<th>10 000 000</th>
<th>&lt; log.07.f3.txt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>242 601</td>
<td></td>
</tr>
<tr>
<td>% java Hash</td>
<td>20 000 000</td>
<td>&lt; log.07.f3.txt</td>
</tr>
<tr>
<td></td>
<td>483 477</td>
<td></td>
</tr>
<tr>
<td>% java Hash</td>
<td>40 000 000</td>
<td>&lt; log.07.f3.txt</td>
</tr>
<tr>
<td></td>
<td>88 307 1</td>
<td></td>
</tr>
<tr>
<td>% java Hash</td>
<td>60 000 000</td>
<td>&lt; log.07.f3.txt</td>
</tr>
<tr>
<td></td>
<td>109 794 4</td>
<td></td>
</tr>
</tbody>
</table>

**HyperBitBit estimates**

| % java HyperBitBit | 10 000 000 | < log.07.f3.txt |
|                    | 23 421 9   |                |
| % java HyperBitBit | 20 000 000 | < log.07.f3.txt |
|                    | 49 988 9   |                |
| % java HyperBitBit | 40 000 000 | < log.07.f3.txt |
|                    | 91 680 1   |                |
| % java HyperBitBit | 60 000 000 | < log.07.f3.txt |
|                    | 104 404 3  |                |

<table>
<thead>
<tr>
<th>Exact</th>
<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>HyperBitBit</td>
<td>242,601</td>
<td>483,477</td>
<td>883,071</td>
<td>1,097,944</td>
</tr>
<tr>
<td>ratio</td>
<td>1.05</td>
<td>1.03</td>
<td>0.96</td>
<td>1.03</td>
</tr>
</tbody>
</table>

**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.**
Small sample of work on related problems

<table>
<thead>
<tr>
<th>Year</th>
<th>Author(s)</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>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>
## Summary/timeline for cardinality estimation

<table>
<thead>
<tr>
<th>Year</th>
<th>Name</th>
<th>Method</th>
<th>Hashing Assumption</th>
<th>Feasible and Validated?</th>
<th>Memory Bound</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><strong>PCSA</strong></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>[theorem]</td>
<td>strong universal</td>
<td>✗</td>
<td>( O(M \log N) )</td>
<td>O(1)</td>
<td>?</td>
</tr>
<tr>
<td>2002</td>
<td>Bar-Yossef-Jayram-Kumar-Sivakumar-Trevisan</td>
<td>[theorem]</td>
<td>strong universal</td>
<td>✗</td>
<td>( O(M \log \log N) )</td>
<td>O(1)</td>
<td>?</td>
</tr>
<tr>
<td>2003</td>
<td>Durand-Flajolet</td>
<td><strong>LogLog</strong></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><strong>HyperLogLog</strong></td>
<td>uniform</td>
<td>✓</td>
<td>( M \lg \lg N )</td>
<td>1.02</td>
<td>768</td>
</tr>
<tr>
<td>2010</td>
<td>Kane-Nelson-Woodruff</td>
<td>[theorem]</td>
<td>strong universal</td>
<td>✗</td>
<td>( O(M) + \lg \lg N )</td>
<td>O(1)</td>
<td>?</td>
</tr>
<tr>
<td>2016?</td>
<td><strong>HyperBitBit</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>134 (?)?</td>
</tr>
</tbody>
</table>
Philippe Flajolet, mathematician, data scientist, and computer scientist extraordinaire

Philippe Flajolet 1948–2011
Cardinality Estimation

Robert Sedgewick
Princeton University

with special thanks to Jérémie Lumbroso