Cardinality Estimation

Robert Sedgewick
Princeton University

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

Philippe Flajolet 1948–2011
Analysis of Algorithms

“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.

how many times do we have to turn the crank?

Analytic Engine
Analysis of Algorithms (present-day context)

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

AofA
- Theorems \textit{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-71-104-94-246.1sanca.dsl-w.verizon.net
117.222.48.163
pool-71-104-94-246.1sanca.dsl-w.verizon.net
1.23.193.58
188.134.45.71
1.23.193.58
gsearch.CS.Princeton.EDU
pool-71-104-94-246.1sanca.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
... ... ...

6 million strings

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

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
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>4</th>
<th>109.108.229.102</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>81.95.186.98.freenet.com.ua</td>
</tr>
<tr>
<td>12</td>
<td>117.222.48.163</td>
</tr>
<tr>
<td>5</td>
<td>1.23.193.58</td>
</tr>
<tr>
<td>5</td>
<td>188.134.45.71</td>
</tr>
<tr>
<td>5</td>
<td>1.23.193.58</td>
</tr>
<tr>
<td>14</td>
<td>81.95.186.98.freenet.com.ua</td>
</tr>
<tr>
<td>14</td>
<td>81.95.186.98.freenet.com.ua</td>
</tr>
<tr>
<td>1</td>
<td>117.211.88.36</td>
</tr>
<tr>
<td></td>
<td>msnbot-131-253-46-251.msn.com</td>
</tr>
<tr>
<td></td>
<td>msnbot-131-253-46-251.msn.com</td>
</tr>
<tr>
<td></td>
<td>gsearch-CS.Princeton.EDU</td>
</tr>
<tr>
<td></td>
<td>118-171-27-8.dynamic.hinet.net</td>
</tr>
<tr>
<td></td>
<td>cpe-76-170-182-222.socal.rr.com</td>
</tr>
<tr>
<td></td>
<td>203-88-22-144.live.vodafone.in</td>
</tr>
<tr>
<td></td>
<td>117.211.88.36</td>
</tr>
</tbody>
</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();
}
```

http://algs4.cs.princeton.edu/34hash/LinearProbingHashST.java
**Theorem.** Expected time and space cost is linear.

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

The average number of probes needed by Algorithm L, assuming that all $M^N$ hash sequences are equally likely, is

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

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

where

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

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

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

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");
}
```

% 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

**Q. Is hashing with linear probing effective?**

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

% 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. 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).

---

Dietzfelbinger, Karlin, Mehlhorn, Meyer auf der Heide, Rohnert, and Tarjan
*Dynamic Perfect Hashing: Upper and Lower Bounds*
*SICOMP* 1994.

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

Typical modern applications
- Can only afford to process each value once.
- Estimate of count still very useful.

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-121-218-151-176.1nse3.cht.bigpond.net.au
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.

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 AoA.
**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.

```java
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.** 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. \( R(x) = 2^{r(x)} \)

<table>
<thead>
<tr>
<th></th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p(x) )</td>
<td>12</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( r(x) )</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( R(x) )</td>
<td>10</td>
<td>5</td>
<td>32</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

Beeler, Gosper, and Schroeppel

HAKMEM item 169, MIT AI Laboratory AIM 239, 1972

http://www.inwap.com/pdp10/hbaker/hakmem/hakmem.html

see also Knuth volume 4A
Probabilistic counting (Flajolet and Martin, 1983)

Maintain a single-word sketch that summarizes a data stream $x_0, x_1, ..., x_N, ...$

- 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 $\log N$.

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

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

---

**typical sketch**

$N = 10^6$

| sketch | $x_N$ | $R(x_N)$ | $R(x_N)$ sketch | estimate of $\log N$ |
|--------|------|---------|----------------|
| 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 1 1 1 1 1 | 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 1 0 0 0 1 1 1 | 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 | 20 | 9 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |

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

leading bits almost surely 0

trailing bits almost surely 1

**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>0110001001100111101111011110</td>
<td>2</td>
<td>100</td>
<td>00000000000000000000000000000000100</td>
</tr>
<tr>
<td>011001110010011000111110000000101</td>
<td>1</td>
<td>10</td>
<td>0000000000000000000000000000000010</td>
</tr>
<tr>
<td>0001000100011110011011011010011</td>
<td>2</td>
<td>100</td>
<td>00000000000000000000000000000000110</td>
</tr>
<tr>
<td>01000100011110111100000001110</td>
<td>5</td>
<td>100000</td>
<td>00000000000000000000000000000000100110</td>
</tr>
<tr>
<td>011010000101100010111000100100</td>
<td>0</td>
<td>1</td>
<td>00000000000000000000000000000000100111</td>
</tr>
<tr>
<td>0011011110110000000001010010101011</td>
<td>1</td>
<td>10</td>
<td>0000000000000000000000000000000010011</td>
</tr>
<tr>
<td>00110100011000111101011111111000</td>
<td>0</td>
<td>1</td>
<td>0000000000000000000000000000000010011</td>
</tr>
<tr>
<td>0001100001000100010111000101101110</td>
<td>3</td>
<td>1000</td>
<td>00000000000000000000000000000000101111</td>
</tr>
<tr>
<td>000110011001110011110100000011111</td>
<td>6</td>
<td>1000000</td>
<td>000000000000000000000000000000001101111</td>
</tr>
<tr>
<td>01000101110001001010110011111000</td>
<td>0</td>
<td>1</td>
<td>000000000000000000000000000000001101111</td>
</tr>
</tbody>
</table>

$R(\text{sketch}) = 10000_2$

$= 16$
Probabilistic counting (first try)

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)

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

class Sketch
{
    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!
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 $\text{mean} =$ average number of trailing bits in the sketches.
- Return $2^{\text{mean}}/77531$.

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

**Use initial m bits for second hash**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>100</td>
<td>0000000000000000</td>
<td>00000000000000100</td>
<td>0000000000000000</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>0000000000000010</td>
<td>0000000000000010</td>
<td>0000000000000000</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>0000000000000010</td>
<td>0000000000000010</td>
<td>0000000000000000</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>10000</td>
<td>000000000010010</td>
<td>000000000010010</td>
<td>0000000000000000</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1000</td>
<td>000000000010010</td>
<td>000000000010010</td>
<td>0000000000000000</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>100</td>
<td>0000000000000010</td>
<td>0000000000000010</td>
<td>0000000000000000</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>0000000000000010</td>
<td>0000000000000010</td>
<td>0000000000000000</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1000</td>
<td>0000000000000010</td>
<td>0000000000000010</td>
<td>0000000000000000</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1000</td>
<td>0000000000000010</td>
<td>0000000000000010</td>
<td>0000000000000000</td>
<td></td>
</tr>
</tbody>
</table>

**r(sketch[])**

<table>
<thead>
<tr>
<th>2</th>
<th>1</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010101101110</td>
<td>0000000000000010</td>
<td>0000000000000011</td>
<td>0000000000000000</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;
    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 $mean = \text{average} \#$ trailing 1 bits in the sketches.
- Return $2^{mean}/.77351$.

*Flajolet-Martin 1983*
Theoretical analysis of PCSA

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

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

$$E[\beta^q] = n^{1/8}(q + P_q(\log_2 n)) + o(n^{1/8}).$$

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

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

subject to $h > 0$.

- $1/m$; let $N_i$ be the number of elements that fall into distribution

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

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

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

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

$$\Pr \left[ \bigcup_{1 \leq j \leq m} \left| N_j - \frac{n}{m} \right| > \sqrt{n \log n} \right] \ll m \Pr \left[ \left| N_1 - \frac{n}{m} \right| > \sqrt{n \log n} \right].$$

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

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

- **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
```
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).**

<1% larger than actual value

Q. Is PCSA effective?

A. ABSOLUTELY!

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

---

`log.07.f3.txt`

| 109.108.229.102 |
| pool-71-104-94-246.lsanka.dsl-w.verizon.net |
| 117.222.48.163 |
| pool-71-104-94-246.lsanka.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.lsanka.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-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?

A poster child for AofA/AC

“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*

*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

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* $\frac{1.30}{\sqrt{M}}$.

$Ig N$ bits can save a value (PCSA)

$Ig \ lg N$ 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)

```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);
    }
    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 $1.02/\sqrt{M}$.
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*

*Upper bound*

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

**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* \( O(1/\sqrt{M}) \).

*Lower bound*

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 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>768</td>
</tr>
<tr>
<td>BetterAlgorithm</td>
<td>a few dozen</td>
<td>a few hundred</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(sketche) > 31)
            { sketch = sketch2; lgN++; sketch2 = 0; }
    }
    return (int) (Math.pow(2, lgN + 5.4 + p(sketche)/32.0));
}

Idea.
• lgN is estimate of \( \log N \)
• sketch is 64 indicators whether to increment \( \log N \)
• sketch2 is is 64 indicators whether to increment \( \log N \) by 2
• Update when half the bits in sketch are 1

bias (determined empirically)
Initial experiments

**Exact values for web log example**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>% java Hash 1000000 &lt; log.07.f3.txt</td>
<td>242601</td>
<td></td>
<td></td>
</tr>
<tr>
<td>% java Hash 2000000 &lt; log.07.f3.txt</td>
<td>483477</td>
<td></td>
<td></td>
</tr>
<tr>
<td>% java Hash 4000000 &lt; log.07.f3.txt</td>
<td>883071</td>
<td></td>
<td></td>
</tr>
<tr>
<td>% java Hash 6000000 &lt; log.07.f3.txt</td>
<td>1097944</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**HyperBitBit estimates**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>% java HyperBitBit 1000000 &lt; log.07.f3.txt</td>
<td>234219</td>
<td></td>
<td></td>
</tr>
<tr>
<td>% java HyperBitBit 2000000 &lt; log.07.f3.txt</td>
<td>499889</td>
<td></td>
<td></td>
</tr>
<tr>
<td>% java HyperBitBit 4000000 &lt; log.07.f3.txt</td>
<td>916801</td>
<td></td>
<td></td>
</tr>
<tr>
<td>% java HyperBitBit 6000000 &lt; log.07.f3.txt</td>
<td>1044043</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></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>Exact</td>
<td>242,601</td>
<td>483,477</td>
<td>883,071</td>
<td>1,097,944</td>
</tr>
<tr>
<td>HyperBitBit</td>
<td>234,219</td>
<td>499,889</td>
<td>916,801</td>
<td>1,044,043</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>Contribution</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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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>Authors/Reference</th>
<th>Method</th>
<th>Hashing Assumption</th>
<th>Feasible and Validated?</th>
<th>Memory Bound</th>
<th>Relative Accuracy</th>
<th># Bits for 10% Accuracy</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>[Theorem]</td>
<td>strong universal</td>
<td>X</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>X</td>
<td>$O(M \log \log N)$</td>
<td>O(1)</td>
<td>?</td>
</tr>
<tr>
<td>2003</td>
<td>Durand-Flajolet</td>
<td>LogLog</td>
<td>uniform</td>
<td>✓</td>
<td>$M \log \log N$</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>$M \log \log N$</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>X</td>
<td>$O(M) + \log \log N$</td>
<td>O(1)</td>
<td>?</td>
</tr>
<tr>
<td>2016?</td>
<td>HyperBitBit</td>
<td></td>
<td></td>
<td></td>
<td>$2M + \log \log N$</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