Better with Fewer Bits: Improving the Performance of Cardinality Estimation of Large Data Streams

Qingjun Xiao
School of Computer Science & Engineering, Southeast University, China.
Email: csqjxiao@seu.edu.cn

You Zhou, Shigang Chen
Dept. of Computer and Information Science & Engineering, University of Florida, USA.
Email: {youzhou, sgchen}@cise.ufl.edu
What is the cardinality estimation problem?

Elements occur multiple times, we want to count the number of distinct elements.

- Number of distinct element is \( n \) (\( = 6 \) in example)
- Number of elements in this example is 11

32, 12, 14, 32, 7, 12, 32, 7, 6, 12, 4,
A list of applications

Counting the number of unique visitors (100m+ daily visits) -- the most important metric in online advertising
  • See Redis HyperLogLog data structure

ISP measurement of traffic usage
Routers traffic in the range of Terabits/sec \(10^{12} \text{ b/s}\)

Internet-scale data measurement: Google indexes 6+ billions pages, and counts the number of accesses to each pages, and also counts the number of searches towards each keyword

Cardinality in DB queries optimization
  • the number of rows returned by each operation in an execution plan
Rules of the game

- **Limited memory**: cannot store all the stream elements; can use just one page of memory footprint, about 4 kB, in order to fit into the high-speed working memory.
- **Limited time**: online processing of the stream data, read the data by a single pass, or read-once data.

- Allow to generate an **approximate estimate** of the cardinality $n$, rather than compute its exact value.
- Assume there is a **uniform hash function** $h : \mathcal{D} \rightarrow [0, 1]$, to map the stream elements uniformly and pseudo-randomly.
One of fundamental stream processing techniques:

Give a (large) sequence of data values over a (large) domain \( \mathcal{D} \)

\[
S = s_1 s_2 \cdots s_\ell, \quad s_j \in \mathcal{D}
\]

View the stream \( S \) as a multiset \( MS \):

\[
MS = e_1^{f_1} e_2^{f_2} \cdots e_n^{f_n}, \quad s_j \in \mathcal{D}
\]

Element \( e_i \) has \( f_i \) repetitions, or say the frequency of \( e_i \) is \( f_i \).

Stream Processing Problems:

▶ Size Estimation: What is the size \( \ell \) of the stream?

▶ Cardinality Estimation: How many different elements are present?

▶ Elephants Identification: What are the elements with absolute frequencies above a threshold, e.g., \( f_i > 500 \)?

▶ Icebergs Identification: What are the elements with relative frequencies above a threshold, e.g., \( \frac{1}{\ell} f_i > \frac{1}{100} \)?

▶ Frequency moment estimation: \((\sum f_i^r)^{1/r}\)
Probabilistic counting with stochastic averaging

Philippe Flajolet, 1948–2011,
mathematician and computer scientist extraordinaire


Contributions of PCSA
• Introduced the problem
• Idea of streaming algorithm
• Idea of “small” sketch of “big” data
• Detailed analysis that yields tight bounds on accuracy
Two key ideas of PCSA (1)

Probabilistic Counting Register

• Use a hash function to map incoming stream elements to a bit array with **exponentially deceasing probabilities**
• Find the position $M'$ of the leftmost zero bit
• The bit array, or called a PC register, can generate a pretty coarse estimation for the cardinality of input stream as $2^{M'}$
Two key ideas of PCSA (2)

Stochastic Averaging

- Use a second hash function to divide the input stream into $2^m$ independent substreams
- Apply a PC register to each substream

Keypoint: Equal items all go to the same substream

- Compute $mean =$ average position of the left-most zero bits in the $2^m$ registers
- Return the result: $2^{mean}/0.77531$
Space-accuracy tradeoff for PCSA

Relative Accuracy: $\frac{0.78}{\sqrt{m}}$, where $m$ is the number of registers, and each register is given 32 bits memory.

But the space-accuracy tradeoff of PCSA is no satisfactory. A plethora of algorithms are proposed for improvement.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Std. Err.(σ)</th>
<th>Mem Units</th>
<th>Mem(σ=2%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MinCount</td>
<td>$1.00/\sqrt{m}$</td>
<td>32-bit keys</td>
<td>10000 bytes</td>
</tr>
<tr>
<td>PCSA</td>
<td>$0.78/\sqrt{m}$</td>
<td>32-bit registers</td>
<td>6084 bytes</td>
</tr>
<tr>
<td>MultiresBitmap</td>
<td>$\approx 4.4/\sqrt{m}$</td>
<td>1 bit</td>
<td>6050 bytes</td>
</tr>
<tr>
<td>LogLog</td>
<td>$1.30/\sqrt{m}$</td>
<td>5-bit registers</td>
<td>2641 bytes</td>
</tr>
<tr>
<td>HyperLogLog</td>
<td>$1.04/\sqrt{m}$</td>
<td>5-bit registers</td>
<td>1690 bytes</td>
</tr>
</tbody>
</table>

HyperLogLog is the state-of-the-art!!! It can reduce memory cost by 72% for attaining the same accuracy.
Valuable Ideas of HyperLogLog

• Use **loglog registers**, which are of log log(n) bits each

• Use **harmonic averaging**, instead of geometric mean, to summarize the estimation results of m loglog registers

**Key point:** Use a LogLog register to record the position of rightmost one-bit

HyperLogLog warmly embraced by industries

It appears difficult to further improve the space-accuracy tradeoff of HyperLogLog

HyperLogLog+ fixes some minor problems

HyperLogLog has two major shortcomings

We discover that the HyperLogLog register values exhibit a right-skewed distribution, implying the following two facts.

• **Outliers with large values** exist in the rightside long tail
• **Inefficient to use 5 bits** to encode the register histogram

Our technique: truncate the right-side tail to reject the outliers, and use less than five bits to encode the histogram.
HyperLogLog has another minor shortcoming:

- HyperLogLog has a **small biased region** from $2m$ to $5m$, since it uses LinearCounting for cardinalities $n < 2.5m$.
- Our HLL-TC and HLL-TC+ can remove the bias
- We greatly improve accuracy at the same memory cost

---

![Graphs showing estimation bias and standard deviation for HLL, HLL+, HLL-TC, and HLL-TC+ estimators.](image)

(a) Estimation bias

(b) Standard deviation

Fig. 6. Compare cardinality estimators with the same 24.58k bits memory.
What do we promise?

- We will propose two algorithms
  - HLL-TailCut needs $m$ registers of four bits each, and provides relative accuracy $\frac{1.04}{\sqrt{m}}$
  - HLL-TailCut+ needs $m$ registers of three bits each, and provides relative accuracy $\frac{1.00}{\sqrt{m}}$
- Both can support the counting of Tera- and Peta-scale data, while HyperLogLog can only support Giga-scale data.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Std. Err. ($\sigma$)</th>
<th>Mem Units</th>
<th>Mem ($\sigma=2%$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HyperLogLog</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HLL-TailCut</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HLL-TailCut+</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

HLL-TC needs 20% less memory

HLL-TC+ needs 45% less memory
How do we implement HLL-TailCut?

- **Improve the space-accuracy tradeoff (20% less memory cost)**, by reducing the size of a register to four bits only
- Use a base register $B$ to keep track of the value of the smallest HyperLogLog register
- Use $m$ offset registers to record the offsets of $m$ HyperLogLog registers relative to the base $B$
- Each offset register is given four bits memory
- A negligibly small portion of the long tail has been cut off

A Sliding Window of Width $K = 2^4$

Large overflowed values will be rounded down to $B + K - 1$
How do we implement HLL-TailCut? (conti.)

- **Have addressed the problem of small biased region**
  - When the estimated cardinality is larger than $5m$, we still use the HyperLogLog equation
    \[ \hat{n} = \alpha_m \cdot m^2 \cdot \left( \sum_{0 \leq j < m} 2^{-(B + \tilde{M}_j)} \right)^{-1} \quad (9) \]
  - When the estimated cardinality is smaller than $2m$, we still use the LinearCounting equation
  - When the estimated cardinality is **between $2m$ and $5m$**, we use the following maximum likelihood estimation formula
    \[ \hat{n} = \arg \max_n \log \mathcal{L}(n \mid N_0, N_1, \ldots, N_{K-1}) \quad (5) \]
    \[ \mathcal{L}(n \mid N_0, N_1, \ldots, N_{K-1}) \approx \frac{m!}{N_1!N_2!\ldots N_{K-1}!} \prod_{k=0}^{K-1} Pr\{M_j = k\}^{N_k} \quad (4) \]
    \[ Pr\{M_j = k\} \approx \begin{cases} (1 - \frac{1}{m})^n & \text{if } k = 0, \\ (1 - \frac{1}{m2^k})^n - (1 - \frac{1}{m2^{k-1}})^n & \text{if } k \geq 1. \end{cases} \quad (3) \]
How do we implement HLL-TailCut+?

- Further improve the space-accuracy tradeoff (i.e., 45% less memory cost), by reducing the size of a register to three bits only, i.e., $\mathcal{K} = 2^3 = 8$

- When $\mathcal{K} = 8$, a non-negligible proportion of the right-side long tail has been truncated

- MUST consider the dynamically increasing process of the base $\mathcal{B}$

- Use a maximum likelihood estimator to determine the cardinality of newly arrived stream elements, when the base $\mathcal{B}$ equals 0, 1, 2, 3, respectively
Beyond space-accuracy tradeoff: Mergeability of multiple sketches

Stream 1

Stream 2

Stream 1 + 2

Sketch 1

Sketch 2

Sketch 12

Enough to consider merging two sketches
The side-effect of our HLL-TC+ algorithm

- Our HLL-TC is able to merge multiple sketches.
- But our HLL-TC+ may not support the sketch merging.
- Our HLL-TC+ more fits low-end devices that desire the highest memory-efficiency and does not need mergability.
Simulation Result on Space-Accuracy Efficiency

Coarse accuracy comparison given the same memory

Fine accuracy comparison given the same memory

Fig. 5. Compare cardinality estimators with the same 1.54k bits memory.

Fig. 6. Compare cardinality estimators with the same 24.58k bits memory.
Summary

• Propose two new cardinality estimation algorithms, HLL-TailCut and HLL-TailCut+, and upload source code
  • https://www.dropbox.com/s/l0eaexhzvi34x9u/HLLPlus.zip

• Improve the space-accuracy tradeoff of HyperLogLog
  • HLL-TailCut needs 20% less memory at the same accuracy
  • HLL-TailCut needs 45% less memory at the same accuracy

• Address the small biased region problem of HyperLogLog

• Extend the effective operating range of HyperLogLog from Giga-scale data streams to Peta-scale or even Tera-scale data streams

• HLL-TailCut can support the merging of multiple sketches
Question and Answer