Probabilistic data structures. Part 2. Cardinality

Presentation · July 2016

DOI: 10.13140/RG.2.1.2476.8244

Andrii Gakhov
V. N. Karazin Kharkiv National University

PROBABILISTIC DATA STRUCTURES

ALL YOU WANTED TO KNOW BUT WERE AFRAID TO ASK

PART 2: CARDINALITY

Andrii Gakhov
tech talk @ ferret

CARDINALITY

Agenda:

- Linear Counting
- LogLog, SuperLogLog, HyperLogLog, HyperLogLog++

THE PROBLEM

- To determine the *number of distinct elements*, also called the **cardinality**, of a large set of elements where duplicates are present

Calculating the exact cardinality of a multiset requires an amount of memory proportional to the cardinality, which is impractical for very large data sets.

LINEAR COUNTING

LINEAR COUNTING: ALGORITHM

- Linear counter is a **bit map** (hash table) of size **m** (all elements set to 0 at the beginning).
- Algorithm consists of a few steps:
  - for every element calculate hash function and set the appropriate bit to 1
  - calculate the fraction **V** of empty bits in the structure (divide the number of empty bits by the bit map size **m**)
  - estimate cardinality as $\mathbf{n \approx -m \ln V}$

LINEAR COUNTING: EXAMPLE

- Consider linear counter with 16 bits (m=16)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- Consider **MurmurHash3** as the hash function **h**
  (to calculate the appropriate index, we divide result by mod 16)

- Set of 10 elements: "*bernau*", "*bernau*", "*bernau*", "*berlin*", "*kiev*", "*kiev*", "*new york*", "*germany*", "*ukraine*", "*europe*" (NOTE: the real cardinality n = 7)

h("bernau") = 4, h("berlin") = 4, h("kiev") = 6, h("new york") = 6, h("germany") = 14, h("ukraine") = 7, h("europe") = 9

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	1	1	0	1	0	0	0	0	1	0

LINEAR COUNTING: EXAMPLE

0	0	0	0	1	0	1	1	0	1	0	0	0	0	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

number of empty bits: 11

m = 16

V = $^{11}/_{16}$ = 0.6875

- Cardinality estimation is

n ≈ - 16 * ln (0.6875) = 5.995

LINEAR COUNTING: READ MORE

- http://dblab.kaist.ac.kr/Prof/pdf/Whang1990(linear).pdf
- http://www.codeproject.com/Articles/569718/CardinalityplusEstimationplusinplusLinearplusTimep

HYPERLOGLOG

HYPERLOGLOG: INTUITION

- The cardinality of a multiset of *uniformly distributed numbers* can be estimated by the maximum number of leading zeros in the binary representation of each number. If such value is **k**, then the number of distinct elements in the set is $\mathbf{2^k}$

	0	1	2	3												
f =	0	0	**1**	0	0	0	1	0	0	0	1	0	0	0	0	0

leading zeros

rank = number of leading zeros + 1, e.g. rank(f) = 3

P(rank=1) = 1/2 - probability to find a binary representation, that starts with 1

$\mathbf{P(rank = 2) = 1/2^2}$ - probability to find a binary representation, that start with 01

…

$\mathbf{P(rank=k) = 1/2^k}$

- Therefore, for $\mathbf{2^k}$ binary representations we shell find <u>at least one</u> representation with **rank = k**
- If we remember the maximal rank we've seen and it's equal to **k**, then we can use $\mathbf{2^k}$ as the approximation of the number of elements

HYPERLOGLOG

- proposed by Flajolet et. al., 2007
- an extension of the Flajolet–Martin algorithm (1985)
- HyperLogLog is described by 2 parameters:
  - **p** – number of bits that determine a bucket to use averaging ($\mathbf{m = 2^p}$ is the number of buckets/substreams)
  - **h** - hash function, that produces *uniform* hash values

- The HyperLogLog algorithm is able to estimate cardinalities of $> 10^9$ with a typical error rate of 2%, using 1.5kB of memory (Flajolet, P. et al., 2007).

HYPERLOGLOG: ALGORITHM

- HyperLogLog uses **randomization** to approximate the cardinality of a multiset. This randomization is achieved by using hash function **h**
- Observe the maximum **number of leading zeros** that for all hash values:
  - If the bit pattern $\mathbf{0^{L-1}\,1}$ is observed at the beginning of a hash value (so, rank = L), then a *good estimation of the size of the multiset* is $\mathbf{2^L}$.

HYPERLOGLOG: ALGORITHM

- **Stochastic averaging** is used to reduce the large variability:
  - The input stream of data elements S is divided into **m** substreams $\mathbf{S_i}$ using the first **p** bits of the hash values (**m = 2^p^**).
  - In each substream, the rank (after the initial **p** bits that are used for substreaming) is measured independently.
  - These numbers are kept in an array of registers **M**, where **M[i]** stores the maximum rank it seen for the substream with index **i**.
- The **cardinality estimation** is calculated computes as the normalized bias corrected harmonic mean of the estimations on the substreams

$$DV_{HLL} = const(m) \cdot m^2 \cdot \left(\sum_{j=1}^{m} 2^{-M_j} \right)^{-1}$$

HYPERLOGLOG: EXAMPLE

- Let's use **p=3** bits to define a bucket (then $m=2^3=$**8 buckets**).

	0	1	2	3	4	5	6	7
M	0	0	0	0	0	0	0	0

- Consider **L=8** bits hash function **h**
- Index elements "berlin" and "ferret":

h("berlin") = **011**0111 **h**("ferret") = **110**0011

- Define buckets and calculate values to store:

(use first p =3 bits for buckets and least L - p = 5 bits for ranks)

- **bucket**("berlin") = 011 = 3 **value**("berlin") = rank(0111) = 2
- **bucket**("ferret") = 110 = 6 **value**("ferret") = rank(0011) = 3

	0	1	2	3	4	5	6	7
M	0	0	0	**2**	0	0	**3**	0

HYPERLOGLOG: EXAMPLE

- Index element "kharkov":
  - **h**("kharkov") = **110**0001
  - **bucket**("kharkov") = 110 = 6 **value**("kharkov") = **rank**(0001) = 4
  - M[6] = **max**(M[6], 4) = **max**(3, 4) = 4

	0	1	2	3	4	5	6	7
M	0	0	0	**2**	0	0	**4**	0

- Estimate the cardinality be the HLL formula ($C \approx 0.66$):

$$DV_{HLL} \approx 0.66 * 8^2 / (2^{-2} + 2^{-4}) = 0.66 * 204.8 \approx 135 \neq 3$$

NOTE: For small cardinalities HLL has a strong bias!!!

HYPERLOGLOG: PROPERTIES

- **Memory requirement doesn't grow linearly with L** (unlike MinCount or Linear Counting) - for hash function of **L** bits and precision **p**, required memory:

$$\lceil \log_2(L+1-p) \rceil \cdot 2^p \ bits$$

  - original HyperLogLog uses 32 bit hash codes, which requires $\mathbf{5 \cdot 2^p}$ **bits**
- **It's not necessary to calculate the full hash code** for the element
  - first **p** bits and number of leading zeros of the remaining bits are enough
- There are no evidence that some of popular hash functions (MD5, Sha1, Sha256, Murmur3) performs significantly better than others.

HYPERLOGLOG: PROPERTIES

- **Algorithm has large error for small cardinalities**.
  - For instance, for **n = 0** the algorithm always returns roughly **0.7m**
  - To achieve better estimates for small cardinalities, use *LinearCounting* below a threshold of **5m/2**

- The **standard error** can be estimated as:

$$\sigma = \frac{1.04}{\sqrt{2^p}}$$

so, if we use 16 bits (p=16) for bucket indices, we receive the standard error in **0.40625%**

HYPERLOGLOG: APPLICATIONS

- *PFCOUNT* in **Redis** returns the approximated cardinality computed by the HyperLogLog data structure (http://antirez.com/news/75)
    - Redis implementation uses 12Kb per key to count with a standard error of 0.81%, and there is no limit to the number of items you can count, unless you approach 2^{64} items

HYPERLOGLOG: READ MORE

- http://algo.inria.fr/flajolet/Publications/DuFl03-LNCS.pdf
- http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf
- https://stefanheule.com/papers/edbt13-hyperloglog.pdf
- https://highlyscalable.wordpress.com/2012/05/01/probabilistic-structures-web-analytics-data-mining/
- https://hal.archives-ouvertes.fr/file/index/docid/465313/filename/sliding_HyperLogLog.pdf
- http://stackoverflow.com/questions/12327004/how-does-the-hyperloglog-algorithm-work

HYPERLOGLOG++

HYPERLOGLOG++

- proposed by Stefan Heule et. al., 2013 for Google PowerDrill
- an improved version of HyperLogLog (Flajolet et. al., 2007)
- HyperLogLog++ is described by 2 parameters:
  - **p** – number of bits that determine a bucket to use averaging
    ($\mathbf{m = 2^p}$ is the number of buckets/substreams)
  - **h** - hash function, that produces *uniform* hash values

- The HyperLogLog++ algorithm is able to estimate cardinalities of ~ $7.9 \cdot 10^9$ with a typical error rate of 1.625%, using 2.56KB of memory (Micha Gorelick and Ian Ozsvald, High Performance Python, 2014).

HYPERLOGLOG++: IMPROVEMENTS

- **use 64-bit hash function**
  - algorithm that only uses the hash code of the input values is limited by the number of bits of the hash codes when it comes to accurately estimating large cardinalities
  - In particular, a hash function of **L** bits can at most distinguish $\mathbf{2^L}$ different values, and as the cardinality n approaches $\mathbf{2^L}$ hash collisions become more and more likely and accurate estimation gets impossible
  - if the cardinality approaches $2^{64} \approx 1.8 \cdot 10^{19}$, hash collisions become a problem
- **bias correction**
  - original algorithm overestimates the real cardinality for small sets, but most of the error is due to bias.
- **storage efficiency**
  - uses different encoding strategies for hash values, variable length encoding for integers, difference encoding

HYPERLOGLOG++ VS HYPERLOGLOG

- **accuracy is significantly better** for large range of cardinalities and equally good on the rest
- sparse representation allows for a **more adaptive use of memory**
- if the cardinality **n** is much smaller then **m**, then HyperLogLog++ **requires significantly less memory**
- For cardinalities between 12000 and 61000, the bias correction allows for a **lower error** and avoids a spike in the error when switching between sub-algorithms.
- 64 bit hash codes allow the algorithm to estimate cardinalities well beyond **1 billion**

HYPERLOGLOG++: APPLICATIONS

- *cardinality* metric in **Elasticsearch** is based on the HyperLogLog++ algorithm for big cardinalities (adaptive counting)
- **Apache DataFu**, collection of libraries for working with large-scale data in Hadoop, has an implementation of HyperLogLog++ algorithm

HYPERLOGLOG++: READ MORE

- http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/40671.pdf
- https://research.neustar.biz/2013/01/24/hyperloglog-googles-take-on-engineering-hll/

THANK YOU

- @gakhov
- linkedin.com/in/gakhov
- www.datacrucis.com