Analysis of `HdrHistogram`

David Andrzejewski
`david@sumologic.com`

June 13, 2016

1 Introduction

`HdrHistogram`[1] (HDR stands for "High Dynamic Range") is a popular and practical method for constructing histograms over streaming data such as latency metrics. It defines a variable-width histogram approach with several appealing properties:

- tunable *relative* error bounds
- suitable for streaming data input
- high-speed inserts
- small, fixed memory footprint
- lossless merges

The performance is impressive. We can process a range of 1 million distinct values with 1% relative error using fewer than 1800 entries. Using 32-bit integers as counters, this would be roughly 7 kilobytes of storage (excluding overhead).

The key downside is that we must pre-define the maximum and minimum observed values.

To my knowledge, there is little to no formal documentation regarding its internal mechanics aside from the reference implementations available in many different programming languages. This document attempts to provide a detailed description of *how* this approach actually works.

2 Usage and external API

2.1 Configuration parameters

To construct an `HdrHistogram`, the user supplies 3 values:

- x_{min} = the smallest possible observed value

[1] https://github.com/HdrHistogram/HdrHistogram

- x_{max} = the largest possible observed value
- s = the number of significant figures (1 to 5)

The meanings of x_{min} and x_{max} should be clear. The significant figures parameter s controls the *relative* error induced by mapping observations to histogram bins. For $s = 1$, every observation x should be placed in a bin with boundaries no further than 10% from x. That is, letting b_k be the lower boundary of bin k containing x where $b_k \leq x < b_{k+1}$, we should have both $0.9x < b_k$ and $b_{k+1} < 1.1x$. Likewise, $s = 2$ should be accurate to 1%, $s = 3$ to 0.1%, and so on.

2.2 Usage

It is natural to consider two basic usage operations: recording a new observed value, and querying the histogram.

2.2.1 Recording an observation

Consider a mapping function $m(x) = k$ the assigns an observed value $x_{min} \leq x \leq x_{max}$ to its appropriate bin k, having the property $b_k \leq x \leq b_{k+1}$.

The core `HdrHistogram` data structure is simply a *fixed-size* vector v of K integer counts, with one counter for each bin $k = 0 \ldots K-1$. Letting $v[k]$ be the k^{th} element, recording an observation x is simply then a matter of doing:

$$\begin{aligned} k &\leftarrow m(x) \\ v[k] &\leftarrow v[k] + 1. \end{aligned}$$

2.2.2 Querying the histogram

Given the interpretation of v as giving us a variable-width histogram of counts with bin boundaries b_k, all the standard histogram query operations are supported:

- exact number of observations between some range b_k and $b_{k'}$
- exact total number of observations
- approximate cumulative distribution function (CDF)
- approximate quantiles

As mentioned earlier, the *approximate* nature of the CDF and quantiles is determined by the significant figures parameter s.

2.2.3 Merging histograms

Given these definitions and mechanics, we get a property which can be crucially useful in practice: `HdrHistograms` (with identical configurations) can be *losslessly* merged.

Let a and b be two observed datasets (ie, mutlisets of numerical observations between some x_{min} and x_{max}), and let ϕ be the `HdrHistogram` construction operation. Then we have

$$\phi(ab) = \phi(a)\phi(b) \tag{1}$$

where ab is the multiset sum and $\phi(a)\phi(b)$ is the formed by simply taking the element-wise sum of the underlying `HdrHistogram` vectors. These combination operations each form commutative monoids over their respective domains (which are, by definition, associative), and the `HdrHistogram` computation ϕ forms a *monoid homomorphism* between the two.

The practical usefulness of this property in large-scale distributed data processing is difficult to overstate as it affords us tremendous flexibility in computing `HdrHistograms`. We can distribute calculations, snapshot partial results, and aggregate by external keys, all without losing information or introducing (additional) inaccuracy.

3 Internal mechanics

How does `HdrHistogram` provide these behaviors and guarantees? We will now dig a bit deeper into a few questions we that have swept under the rug thus far:

- How do the configuration parameters determine the number of bins K and the bin boundaries b_k?
- How is the mapping function $m(x) = k$ rapidly computed?

For simplicity we focus on *integer-valued* observation data, although the ideas generalize to the real-valued case.

3.1 Defining the bucketing

`HdrHistogram` specifies each bin in terms of two indices: the *bucket* i (which indexes *exponentially* over the full range of values) and the *sub-bucket* j (which indexes *linearly* within a bucket). This decomposition (similar to IEEE 754 floating point representation) is what underpins the error bound: the bin widths grow as the values themselves grow, allowing us to achieve constant *relative* error. But how *exactly* do we accomplish this?

3.1.1 Bin boundaries

We define the bin minimum $b(i,j)$ as a function of two indices for bucket i and sub-bucket j, where

$$b(i,j) = (j + C)(2^i)$$

and C is a magic term to be defined later.

We are using base 2 exponentiation for the *bucket* index i, meaning that the total width of bucket i is 2^i.

For simplicity, we had previously defined the bin mapping function from an observation x to a linear bin index k as $m(x) = k$. Going forward we simply assume the existence of some simple mapping (eg, row-major order) $r(i,j) = k$ between these composite (i,j) coordinates and the linear index k.

3.1.2 Number of sub-buckets and buckets

Into how many sub-buckets must we divide this bucket in order to guarantee that a given observation x is no more than, say $0.1x$, from its bin boundaries $m(x)$ and $m(x) + 1$?

The worst *relative* error will be given by sub-bucket $j = 0$, whose boundaries will be

$$\begin{aligned} b(i,0) &= C2^i \\ b(i,1) &= (1 + C)2^i. \end{aligned}$$

The worst case error is then no worse than $1/C$, given by the bucket width 2^i divided by the bucket minimum $C2^i$. For reasons we will come to later, let's say C must be a power of 2, that is $C = 2^D$ for some D.

For s significant figures we want to relative error to be less than 10^{-s}. We can therefore solve for D as:

$$\begin{aligned} 2^{-D} &\leq 10^{-2} \\ 2^D &\geq 10^2 \\ D &\geq log_2(10^2). \end{aligned}$$

This value $C = 2^D$ also controls the *number* of sub-buckets. Notice that at $j = C$ we have bin boundary

$$\begin{aligned} b(i,C) &= (C + C)2^i \\ &= (2C)2^i \\ &= C2^{i+1} \\ &= (0 + C)2^{i+1} \\ &= b(i + 1, 0). \end{aligned}$$

That is, at $j = C$ we "roll over" to the next bucket $i + 1$. This means that j ranges from $0, \ldots, C - 1$ and there are C sub-buckets per bucket.

But what about i? How many buckets do we need to cover the range $[x_{min}, x_{max}]$? For sake of simplicity, let's say $x_{min} = 0$. We then need to find the smallest value of i such that $C2^i > x_{max}$, which can be computed as

$$i_{max} = \lceil log_2 \frac{x_{max}}{C} \rceil.$$

3.2 Assigning points to buckets

Given the `HdrHistogram` configuration (x_{min}, x_{max}, s), we can now compute the

- number of buckets i_{max}
- number of sub-buckets C
- bin boundary function $b(i, j)$
- coordinate mapping function $r(i, j) = k$

Given a new observation x, it remains to be determined how we can rapidly compute the bin mapping function $m(x) = k$. Given r, this becomes a matter of assigning the appropriate bucket and sub-bucket indices, say via function $a(x) = (i, j)$.

A naive approach could simply iterate through all valid (i, j) pairs until the appropriate bin is identified, but this would be inefficient and unsuitable for high-throughput monitoring applications. Instead, we would like a simple, closed-form function.

Let α be a "mask" value defined by $2C - 1$, which is the largest value which can be represented in the first bucket $i = 0$, and recall that $C = 2^D$.

For observation x, we compute the bucket index i by

1. identifying the (one-based) index of the leftmost 1 in the binary representation of $x|\alpha$ where $|$ is bitwise OR
2. subtracting $D + 1$ from this quantity

Let's unpack this. If $x \leq 2C - 1$, then the leftmost 1 is going to be at position $D + 1$, giving a result of 0 (as we'd expect). For any $x \geq 2C$, the leading 1 will be in some position $\geq D + 2$, and subtracting $D + 1$ will recover the bucket index i.

Given the bucket index i, we can easily recover $(j + C)$ by simply right-shifting x by i (an operation equivalent to simply dividing by 2^i).

This procedure is made possible by the general form of the bin boundaries as well as the earlier restriction that C be a power of 2. This formulation allows us to rapdily compute (i, j) in terms of a handful of simple arithmetic and bitwise operations, without resorting to any expensive division or logarithms. The result is that this approach supports insert times "as low as 3-6 nanoseconds on modern (circa 2012) Intel CPUs."

4 Additional resources

Implementations:

- Java (original): https://github.com/HdrHistogram/HdrHistogram
- Go: https://github.com/codahale/hdrhistogram
- C: https://github.com/HdrHistogram/HdrHistogram_c
- Haskell: https://github.com/joshbohde/hdr-histogram
- Erlang: https://github.com/HdrHistogram/hdr_histogram_erl

Blog posts / etc:

- http://kamon.io/core/metrics/instruments/
- http://psy-lob-saw.blogspot.com/2015/02/hdrhistogram-better-latency-capture.html