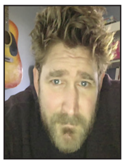
iVoyeur
BPF and Histograms

DAVE JOSEPHSEN



Dave Josephsen is a book author, code developer, and monitoring expert who works for Fastly. His continuing mission: to help engineers worldwide close the feedback loop. dave-usenix@skeptech.org

Argus Panoptes was the all-seeing primordial giant and slayer of the mother of all monsters, Echidna, in Greek mythology. In some tellings he has 100 eyes, some combination of which are always open, though the Renaissance painters (perhaps to save on paint?) always depict him with just the two.

Whether he was a many-eyed giant or merely a very astute, tall man, his powers of observation were so legendary that Hera herself entrusted to Argus the task of guarding the promiscuous nymph Io (inexplicably disguised as a cow) in order to keep her away from Zeus, Hera's unfaithful husband and king of the gods.

Setting aside for a moment the questionable rational of hiring a P.I. to track your cheating husband's *lover* rather than the man himself, Argus proved to be so effective at this task, watching Io day and night, and never once letting her out of his sight for a microsecond, that Zeus eventually had him murdered in order to reunite with his mistress.

I guess the all-seeing Argus *didn't see that comin'*.

That it's often possible to see everything and yet still fail to comprehend is, I think, one of several invaluable lesson Argus Panoptes gave his life to teach us. Every bit as true today as it was in the golden age.

Let's say for example you have a spreadsheet of latency times and other metrics from a front-end web server. With five minutes' worth of samples, you can scroll around and probably tease out some patterns. But with a full day of data, things become vastly more opaque. Ironically, the more you see, the less you begin to comprehend.

In my last column, we talked about the various data structures eBPF uses to pass data from protected kernel space into userspace where we can get our hands on it, and we discovered that our sample BCC tool, `biolatency`, was using a built-in histogram data-structure to summarize data in kernel space. In this issue, as promised, we're going to talk a little about histogram theory and how histograms enable us to make sense of massive amounts of data, thereby achieving comprehension rather than mere observation.

As I'm sure you probably already know, a histogram is a visual representation of a pile of data. Instead of plotting each value in the set, we depict a series of buckets or "bins" which are sized to indicate how many measurements in the sample set fell within the bounds of each bucket. Figure 1 is your obligatory example histogram of 500 measurements, ranging in value from 0 to 100. As you see, it looks like a bar graph, except rather than representing a single metric, each bar represents *the magnitude* of measurements whose value fell between the bar before it, and the bar after it.

Histograms are pretty great because we can depict what the overall data set *looks* like independent of its size. It doesn't matter if the set contains five minutes of data, or five days, we can use the same amount of pixel-space to represent it. Further, histograms are far more representative of the distribution than any combination of statistical reference metrics like

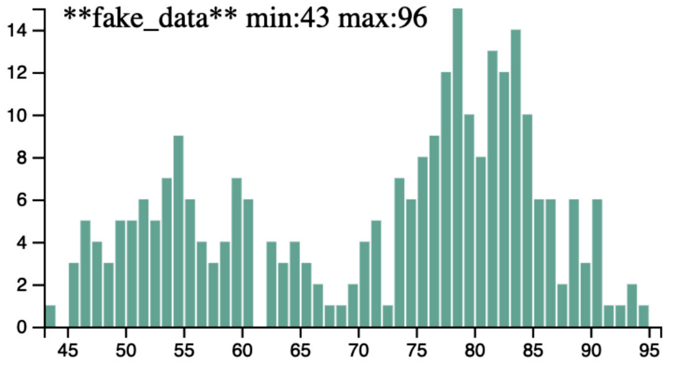


Figure 1: Obligatory example histogram of 500 measurements, ranging from 0 to 100

average, min/max, sum, and p-values, and because they amount to a struct of counters, histograms are super cheap to compute and store.

A histogram's "resolution" is said to vary with its bin-width. You can't accurately represent the value 2.63 if you have bins spaced at integer intervals, for example. That 2.63 would *resolve* to a "3" if our bucket-widths were integer-spaced. So it's important to put some thought into how best to situate the width and total number of bins for a given data set. Obviously, the bin-width decreases (resolution improves) as the total number of bins increases. There are, as you can probably imagine, quite a few strategies to find the optimal total value of bins, or *k value*, for a given data set.

One of the most basic and popular ways of computing the optimal number of bins for a given distribution is the "powers of two" rule, which says the optimal number of bins is the square root of the total number of samples in the set, or for a group of samples s:

$k = \sqrt{s}$

This is the formula used internally by Excel histograms and many other simple implementations when we want to provide a cheap, hands-off way of choosing a bin-number. With the powers of two formula, you'd wind up with 16 buckets for a data set with 256 samples in it, for example. There are various takes on this, like Sturges' formula, which uses the base-2 logarithm of the max value in the data set.

$k = \lceil \log_2 n \rceil + 1$

There is no universally correct number of bins, and every algorithm carries tradeoffs. Sturges, for example, gives poor results for data sets where n<30 but works well on sets with a large range. It's no accident, however, that we've immediately wandered into the land of squares and base-2 logarithms. As it turns out, base-2 logarithms and histograms share something of a magic relationship in computerland, where the underlying representation of basically everything is a binary number.

So far, we've been talking about histograms whose bins are all the same width, aka "Linear Histograms." But there's no particular reason that this should be true. It's absolutely possible to vary the bin-widths within a set number of bins.

In fact, if we were to vary our histogram bin-widths along $\log_2$ boundaries instead of making them all the same width, each boundary between our bins would represent an order of magnitude increase in the sample set. Another way to state that is: every bucket would represent a consecutive bit in a binary number. Therefore, a base-2 "log linear histogram" can use 64 bins to represent a 64-bit int, which is a very large set [0 - 18,446,744,073,709,551,615].

Again, these buckets are not the same size. Instead, they become exponentially fatter at each boundary. The first bin represents the numbers between 0 and 1. The second, 2 and 4, the next 8 and 16 and so on. Now consider this structure in the context of kernel-based metrics like the disk I/O latency we are trying to measure with `biolatency`, and I think you'll realize that this sort of structure is kind of ideal for our problem domain. Our disk I/O is going to *usually* be great, somewhere on the order of tens of milliseconds, where a powers-of-two histogram's resolution is going to be very good. Then we're going to have a small number of outliers on the order of seconds, or possibly tens-of-seconds.

Most in-kernel latency metrics distribute like this: a vast number of very small-value measurements and then a rare smattering of exponentially larger outliers. Many network metrics fit this pattern even more, with normal measurements near 0 and outliers in the billions. The pattern is so pervasive, that BPF has a helper function, called `bpf_log2l()`, which returns the base-2 log of a given measurement, so you can convert any measured value to $\log_2$ scale in-kernel before passing it into the histogram.

Wait what? Convert the value? I thought we were talking about bin boundaries, not modifying the value of our measurements, Dave.

Well, we are. But you'll remember that both the probe code itself as well as the histogram storage `struct` are in-kernel. The histogram implementation [1] is a bare-bones linear histogram, with a statically configured number of same-sized bins. There is no way to create variable-sized bins. But we can simulate the same effect by creating a 64-bucket in-kernel linear histogram and converting (compressing) our measurements to $\log_2$ scale values before storing them.

Remember, we're not storing the actual values, we're merely incrementing *counters* within buckets that roughly align to our values. So if we compress the scale of our values to $\log_2$, we are effectively creating $\log_2$ indexes; we can come back at print-time in userspace and recompute the *indexes* using the in-kernel bucket values as a base-2 exponent, and all the counts will neatly line up with the correct magnitude.

Taking a look at `biolatency`'s storage code [2], we see that every time biolatency commits a value to the data-structure, it uses the `bpf_log2l()` helper function. This is somewhat obfuscated by the find/replace pattern in the Python BCC tools, but the invocation to create the `HISTOGRAM` looks like this:

```
BPF_HISTOGRAM(dist);
```

It's possible to pass in a bin number and a data-structure to represent the index values, but the log-linear use case is so pervasive in BCC that the defaults are aligned to our use case, and the above invocation will create a 64-bin, int-indexed histogram called "dist". We write to it like so:

```
dist.increment(bpf_log2l(delta));
```

Where "delta" is a u64 representing the difference in nanoseconds between the `blk_account_io_start` and `blk_account_io_done` kprobes firing. The kernel will use the delta value to find or create the appropriate bucket and add a +1 to it.

At the end of script-execution, when the userspace side of `biolatency` catches a `Ctl-C`, it grabs the histogram from kernelspace [3] using `get_table()`, the same function we used to grab `structs` from userspace in my previous article. Python BCC has a print-function that knows how to re-compute the indexes of log-linear histograms for us, so all we need to do is pass the dict [4] into the `print_log2_hist()` function, passing along the appropriate labels to make the resulting graph more human readable.

I sometimes wonder what Argus Panoptes would make of the modern world. Thinking about him as a sort of spherical-cow of observation, the hypothetically perfect monitoring machine. Would he be blinded by the abundance of spread-spectrum data being flung in every direction about our heads? Would he be mesmerized to the point of paralysis at the sight of a computer, with its unending infinitesimal internal machinations?

Personally, I vastly prefer to work with instrumentation systems like eBPF, which reward system-knowledge and rely on an interrogative question/answer relationship between operator and machine, over the packaged *measure everything* monitoring systems of the world. I think this is probably true of most engineers. Certainly the ancient Greeks agree, who valued as priceless the oracles, while relegating the spherical-cow of observation to, well, cow-watching. I think that puts us in pretty good company.

Take it easy.

References

[1] https://github.com/torvalds/linux/blob/master/Documentation/trace/histogram.rst.

[2] https://github.com/iovisor/bcc/blob/master/tools/biolatency.py#L127.

[3] https://github.com/iovisor/bcc/blob/master/tools/biolatency.py#L198.

[4] https://github.com/iovisor/bcc/blob/master/tools/biolatency.py#L210.