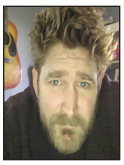
iVoyeur
Prometheus (Part Two)

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

I'm writing to you from the warm afterglow of Monitorama PDX 2019, a conference where we are invariably treated to at least one talk concerning itself with the ever-noble cause of statistics for anomaly detection.

If that sounds a bit sarcastic, it probably is, but only a little bit. If giving talks with titles like "statistics for sysadmin" is a crime, it's one I myself am guilty of many times over, and I fully admit that, like elk grazing their way across a hillside, no matter how many times I see it, it never fails to fascinate.

The talk invariably begins the same way, with a baseline introduction of the normal distribution and an accompanying graphic depicting our old and steadfast friend the bell curve, along with a rundown of some of our very favorite actors, like standard deviation from the mean, z-score, and the like.

But the speaker has a secret that you can probably guess if you are a regular reader of this column, which is this: system metrics are rarely normally distributed. So, really, there are two paths this talk can walk.

In the first, the speaker has gotten lucky and found a use-case for which the input signal happens to be normally distributed, and has therefore been able to apply straightforward statistical analysis to achieve a successful predictive model. The speaker will subsequently encounter a litany of follow-on problems that are sure to entertain us, including unexpected seasonality like cyber-Monday and unpredictable aperiodic events such as labor-union strikes and the like.

If the speaker's problem is not easily represented by a normally distributed metrics signal, things get interesting pretty quickly. This path descends into the land of custom models, advanced math, and data science, which is always fun. But even if the speaker is ultimately successful, the results are rarely directly applicable to our own peculiar set of problems, or are nontrivial to implement if they are.

Well, that's not exactly fair. It's true that complex anomaly detection models are difficult to implement, but that's also true of simple techniques that work on normally distributed signals. Aside from some commercial offerings like Circonus and SignalFx, and a handful of rapidly aging, very basic tools like the Holt-Winters predictive analysis features built into RRDTool, there haven't really been any tools in the monitoring world you can pick up and use to experiment with anomaly detection on time series.

That's why I was delighted to see a pair of talks this year whose content could succinctly be described as: "My Prometheus Queries: Let Me Show You Them!" One is a lightning talk by Jack Neely called "Five Neat Prometheus Tricks" [1], and the other, a full-length talk by Andrew Newdigate entitled "Practical Anomaly Detection Using Prometheus" [2].

As promised, Jack shows us some neat tricks, including overriding the `avg()` function to express things that aren't averages (like ratios), and he helpfully explains how to use operators creatively to craft up/down alerts that don't fire if the host has only been up for five minutes, and measure metrics like memory usage as a function of things like OS or Go version.

Andrew meanwhile dives into the nitty-gritty of anomaly detection, showing us how to compute z-scores on moving averages of Prometheus vectors. Both talks are well worth seeing, but what has me excited is the more general pattern of using PromQL to express an answer, or set of answers, to common monitoring problems. This is especially true in the context of anomaly detection, where we have seen so many talks on general principles without being able to lay our hands on anything like a functional language driving a visualization engine capable of expressing anomaly detection primitives like the z-score.

In my last article, I detailed Prometheus's data model and commented that I was enamored of the tool's ability to pull together different types of engineers by providing a system-agnostic monitoring signal that everyone could "get behind." The simplicity and ubiquity of Prometheus's data model is a huge success, which I believe likely to outgrow the tool itself.

Prometheus Query Language

Prometheus's query language, PromQL, is another great success, as evidenced by the fact that engineers are using it in conference talks as if it were a specification language to communicate techniques and general solutions to common monitoring problems. While the language is certainly more tightly coupled to Prometheus itself than the data model, and has its limitations, I think it was designed sufficiently well that it's already doing a pretty great job of scaling beyond the imagination of its creators.

The simplest Prometheus query is the literal name of a metric. One metric that will probably be available in every Prometheus install is "up." The query syntax is very simple:

```
up
```

The `up` metric is built into every off-the-shelf Prometheus exporter and displays a "1" if the exporter could be contacted by the poller or "0" if it could not. The `job` label shows the name of the exporter that generated each particular `up` metric.

We can filter the output of this query by label, by adding the label name in braces. Node_exporter [3] is the de facto Prometheus system agent, so its `up` metric is a pretty solid metric for host availability in general. We could filter for only the `up` metrics exported by node_exporter like so:

```
up{job="node_exporter"}
```

Internally, every query is actually implemented in this way, with a comma-separated list of label-name equality-operator and value surrounded by braces. Our first query was actually a shortcut for:

```
{__name__="up"}
```

and our second query:

```
{__name__="up",job="node_exporter"}
```

Our equality operator doesn't have to be =. In fact, PromQL supports the following range of equality operators:

- =: Select labels that are exactly equal to the provided string
- !=: Select labels that are not equal to the provided string
- =~: Select labels that regex-match the provided string
- !~: Select labels that do not regex-match the provided string

Regex in PromQL is RE2 [4] syntax, and generally every query that uses a regex must either specify a name or at least one label matcher that does not match the empty string. You can also match the same label multiple times, so an admittedly convoluted way to match every `up` metric from `node_`, except those from `node_blarg` could be:

```
up{job=~"node_.*", job!="node_blarg"}
```

What if, instead of the output of the latest poll, we wanted to see the last five minutes of samples from the poller?

```
up{job="node_exporter"}[5m]
```

By adding a range duration in square brackets to our query, we express to Prometheus that we want to see every sample within the duration for every returned result. Prometheus refers to this output (confusingly) as a "range vector," as opposed to a single-sample response or "instant vector." In the example above, we've expressed our desired duration in minutes, but you can use seconds, minutes, hours, days, weeks, or years instead.

Durations and range-vector results give us the opportunity to begin measuring aggregations of samples over time. For example, to find hosts that have been unavailable any time in the last hour, we can use a function to retrieve the minimum value of the `up` metric over a duration of samples from the last hour (this will return `0` for hosts who have been down any time in the duration):

```
min_over_time(up{job="node_exporter"}[1h])
```

PromQL supports the aggregations you'd expect as well as a few you might not have predicted:

- avg_over_time(range-vector): the average value of all points in the specified interval
- min_over_time(range-vector): the minimum value of all points in the specified interval
- max_over_time(range-vector): the maximum value of all points in the specified interval
- sum_over_time(range-vector): the sum of all values in the specified interval
- count_over_time(range-vector): the count of all values in the specified interval
- quantile_over_time(scalar, range-vector): the φ-quantile ($0 \le \varphi \le 1$) of the values in the specified interval

- stddev_over_time(range-vector): the population standard deviation of the values in the specified interval
- stdvar_over_time(range-vector): the population standard variance of the values in the specified interval

PromQL also has first-class support for "offsets," meaning it's easy to express, for a given query, that you want to see the samples from last week or two hours ago instead of the current samples.

```
up{job="node_exporter"} offset 1w
```

This would give you the instant-vector value of the node_exporter's up metric from exactly one week ago. The syntax works the same for range-vector outputs like so:

```
up{job="node_exporter"}[5m] offset 1w
```

And for function invocations across range vectors:

```
min_over_time(up{job="node_exporter"}[1h] offset 1w)
```

Finally, myriad operators [5] are supported. These allow you to perform mathematical operations and/or filter the results by the return values themselves and enable a lot of other more advanced functionality I won't have space to get into here. If, for example, we *just* wanted to see the hosts that had been down in the last hour, rather than a complete list of hosts with 0s and 1s to indicate their respective status, we could use a binary comparison operator to filter out the "OK" hosts like so:

```
min_over_time(up{job="node_exporter"}[1h]) < 1
```

That should get you started exploring Prometheus metrics with PromQL, but there's a lot more to learn. The aforementioned talks are a great way to sample some of PromQL's outer limits, and of course the docs [6] are well written and expansive.

Take it easy.

References

[1] https://vimeo.com/341145117#t=24m17s.

[2] https://vimeo.com/341141334.

[3] Node_exporter: https://github.com/prometheus/node_exporter.

[4] Syntax: https://github.com/google/re2/wiki/Syntax.

[5] Operators: https://prometheus.io/docs/prometheus/latest/querying/operators/.

[6] Basics: https://prometheus.io/docs/prometheus/latest/querying/basics/.