levelup.gitconnected.com

Conquer PromQL — How Rate and Increase Work - Level Up Coding

Guy Erez

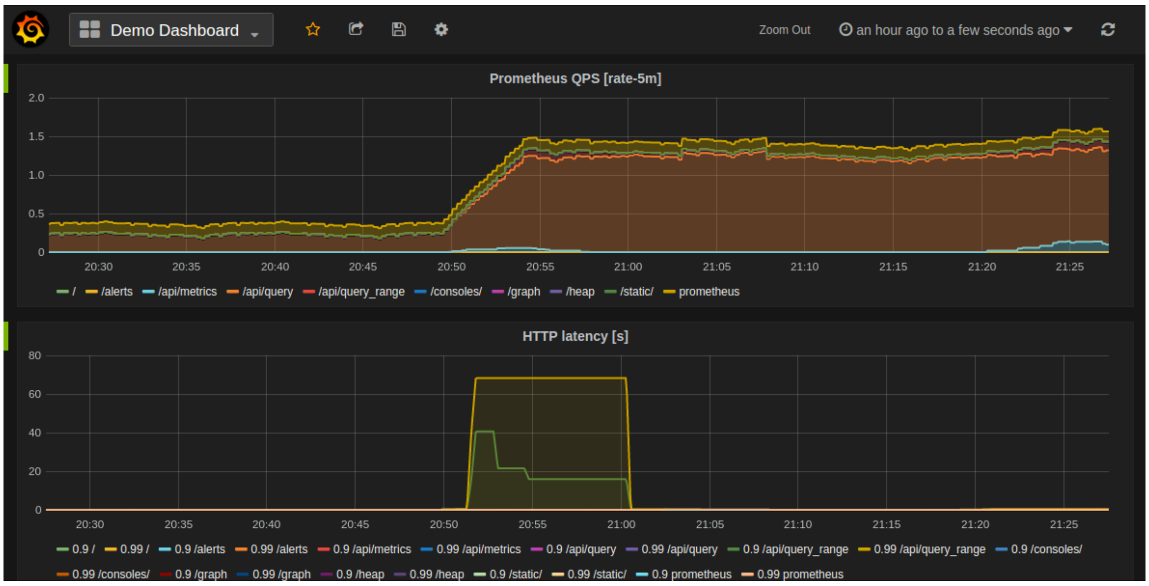
7–8 minutes

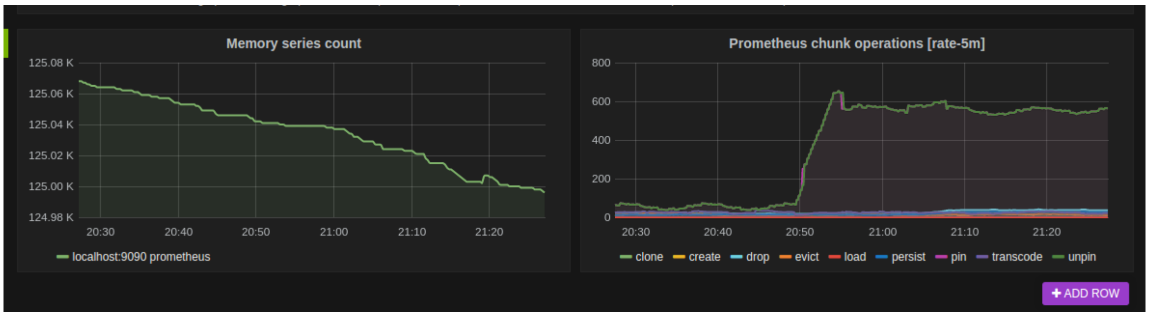
Conquer PromQL — How Rate and Increase Work

Looking to hide highlights? You can now hide them from the “•••” menu.

PromQL. No matter how good you are, there will come a time when it’ll make you want to bang your head against the keyboard and beg for mercy. However, today is NOT that day my friends. Today, we learn!

So… how do they work?


Demo Dashboard
Zoom Out
an hour ago to a few seconds ago
Prometheus QPS [rate-5m]
/ /alerts /api/metrics /api/query /api/query_range /consoles/ /graph /heap /static/ prometheus
HTTP latency [s]
0.9 / 0.99 / 0.9 /alerts 0.99 /alerts 0.9 /api/metrics 0.99 /api/metrics 0.9 /api/query 0.99 /api/query 0.9 /api/query_range 0.99 /api/query_range 0.9 /consoles/ 0.99 /consoles/ 0.9 /graph 0.99 /graph 0.9 /heap 0.99 /heap 0.9 /static/ 0.99 /static/ 0.9 prometheus 0.99 prometheus



Prometheus visualization — https://prometheus.io/docs/visualization/grafana/

The Difference Between Increase() And Rate()

Increase()

Let's start with **`increase()`** as it makes it much easier to understand **`rate()`. When I read the** documentation I felt like I needed a degree in statistics to make sense of it. **So let's break it down in simple terms**:

Let's say you're measuring the number of failed requests to your server. So you've initialized a counter metric called `failed_requests`, with the label `path`. If you decide to query Prometheus for the metric's value, it'll look something like this: `failed_requests{path="/cats"}` and the result will simply be an integer value - let's say it's 10.

Now, you wait for 10 minutes and measure it again — you get 12 this time. So how do you calculate the increase? Pretty simple: You call the **`increase()`** function, on the aforementioned metric & label, and add the timeframe. In our example, we waited for 10 minutes with our second measurement — so it'll be:

`increase(failed_requests{path="/cats"}[10m])`

The result? Simply the current value, minus the value 10 minutes ago (the timeframe). So that brings us to:

`increase(failed_requests{path="/cats"}[10m]) = 12`

`– 10 = 2`

That's all there is to it. Now, what do we do with this information?

Well, Assuming you're not expecting a lot of errors, but you are expecting a non-zero value (because your server is cat-like in its behavior), you can use **`increase()`** to set up an alert, like this:

`increase(failed_requests{path="/cats"}[10m] > 2` So if you get more than 2 alerts in a 10-minute timeframe, the alert will sound, and the SWAT team will be all over your server, catching it in the act of consuming way too much catnip.

Great, so now that we've mastered **`increase()`** Let's talk about **`rate()`**

Rate()

It's a bit more complicated, but just a bit.

Let's say we're looking at the same metric, but we don't care about absolute numbers (*Because Only a Sith deals in absolutes*). We could use it for a few common use cases:

1. Measuring trends in the metric's value — can help us spot sudden spikes/drops that might indicate we're having an issue.
2. Handling data that has a variable sample frequency — like response time (which is not a discrete metric like # of failed requests). It can help us normalize the data by calculating the rate over a fixed time period.

Cool, so you're convinced now. How does it work? Well, it's actually closely related to the **`increase()`** function. One could say it's the **`increase()`** function with extra steps, **one** extra step.

Let's go back to our example. So **`failed_requests{path="/cats"}`** had a value of 10, and 10 minutes later, it had a value of 12.

So**`increase(failed_requests{path="/cats"}[10m]) = 2`**

But we need to get the rate. Since the increase happened over a 10-minute timeframe, all we gotta do is divide it by 10 minutes (in seconds - because `rate()` is calculated per second).

```
rate(failed_requests{path=”/cats”}[10m]) =
increase(failed_requests{path=”/cats”}[10m]) / (10
* 60) = 2 / 600 = 0.00333...
```

So that’s our rate, the relative increase divided by that timeframe.
If we want to set up an alert, assuming we can’t stand a rate that’s higher than that, we can write a query that looks like this:

```
rate(failed_requests{path=”/cats”}[10m]) > 0.003
```

An important point to remember here is that `rate()` is heavily influenced by the metric’s value (the actual absolute numbers) **and the timeframe you’re looking at.**

I even have a story to illustrate this caveat, a true story mind you, about extension heartbeats.

How we almost missed a beat

It was a clear Monday afternoon, everything seemed quite alright, I was deep into a coding session when I got a Slack message from a co-worker. They happened to examine our Grafana dashboards, and it seemed like one of our tenant’s extensions **stopped sending heartbeats**. I was surprised to hear that because we didn’t get any PagerDuty alert on that issue.
However, graphs don’t lie — there seems to be a problem, and we caught it by coincidence.

After a quick investigation, it turned out to be a technical issue — that tenant’s ID was changed — as they became an active prospect. On the prospect’s side — nothing changed, the extensions simply reported a different value. We just didn’t clean the previous ID from our dashboards.

Phew, it wasn't a real incident, but that still doesn't explain why we didn't get an alert, a mystery indeed. The investigation continued.

Can you guess what happened? Y**up, it's very much related to the difference between increase and rate**. We did have an alert, but it wasn't sensitive enough.

That was our alert expression: **`(sum by (tenant_id) (rate(heartbeat[30m] offset 30m))) — (sum by (tenant_id)(rate(heartbeat[30m]))) > 2`**

It worked well, for relatively large numbers — that was a good fit for 100% of our tenants at the time. Over time, we started working with prospects that had a relatively small number of extensions deployed. So even if the rate dropped sharply, the absolute metric values were still quite small.

So the first part of the expression — the heartbeat rate 30 minutes ago, could easily have been 0.1. And even if the heartbeat rate now dropped to 0, we'd still be WAY below the **`> 2`** threshold.

Explanation below — beware — this section is going to involve math — feel free to skip to the next one

Explanation: A steady 6 heartbeats a minute, results in an increase of 30 * 6 = 180 heartbeats over 30 minutes. To get the 30-minute rate, we'd need to divide it by the number of seconds, which comes down to 30 * 60 = 1800. That brings us to 180/1800 = 0.1. And if the increase is 0 now (all extensions stopped sending data for that tenant ID), we'd just get a rate of 0.
And 0.1–0 is still 0.1 < 2.

So what did we do? Using **`rate()`** was still a good idea if we wanted to catch sudden spikes/drops. We just had to cover the edge cases. That meant considering the case where the heartbeats drop to zero and changing the expression to require different thresholds if the absolute heartbeat values are relatively small.

What we've learned today

PromQL is tough, but it doesn't have to be. **`Rate()`** and **`Increase()`**are pretty straightforward. **`Increase()` is simply the increase of the metric over time. `Rate()` is simply `Increase()` divided by the timeframe you're looking at.**

That's all there is to it, just make sure to consider edge cases, and you'll be on your way to conquering PromQL and becoming a true 10x engineer!