PromQL Cheatsheet

Prometheus is a monitoring and alerting system with a text based metric format, a multidimensional data model and a powerful query language. It's now widely used and is the de facto standard for monitoring Kubernetes.

Its metrics can be pulled from different sources, known as 'targets'. Also, they are available even under failure conditions. Furthermore, Prometheus has broad coverage in OSS communities, so there is a lot of integrations and documentation available.

Test these expressions in the 'Try PromQL' tab of our PromQL Playground
https://learn.sysdig.com/promql-playground

FUNCTIONS

MATH FUNCTIONS

```
log10(process_resident_memory_bytes)
```

It takes in a vector and returns a vector, and generally does what you'd expect. Available math functions are: `abs`, `ceil`, `exp`, `floor`, `ln`, `log2`, `log10`, `round`, `sqrt`.

```
round(node_cpu_seconds_total,10)
```

In addition, `round` takes a second optional scalar argument to change what you're rounding to.

CLAMPING

```
clamp_min(go_goroutines,20)
```

It sets a bound on the returned values. This can be useful if you sometimes get spurious values, or incorrectly implemented counters. Two bounds can be set: a lower bound (`clamp_min`) and an upper bound (`clamp_max`).

TIMESTAMPS

```
time() - process_start_time_seconds
```

The `time` function returns the Unix time in seconds when the query has been executed. Useful to calculate elapsed time from a returned timestamp of a metric.

TIME AND DATES

```
day_of_month()
```

You can break a timestamp into multiple time measurements. Available functions are: `day_of_month`, `day_of_week`, `days_in_month`, `hour`, `minute`, `month`, `year`.

GAUGE RANGE VECTORS

```
changes(process_start_time_seconds[15m])
```

It's the number of times each time series changed value.

```
deriv(process_resident_memory_bytes[1h])
```

This uses a least-squares regression to estimate per-second change in a time series.

```
predict_linear(node_filesystem_free_bytes[4h],
3600)
```

It also uses a least-squares regression, and uses it to predict where the time series will be in the given amount of seconds. Other available functions are `holt_winters` for smoothing a time series based on past data, and `idelta` that returns the difference between the last two samples.

AGGREGATING ACROSS TIME

```
avg_over_time
(process_resident_memory_bytes[10m])
```

There are range vector functions that work across time to aggregate each time series, and returns an instant vector. Available functions are:
`avg_over_time`, `sum_over_time`, `count_over_time`, `min_over_time`, `max_over_time`, `stddev_over_time`, `stdvar_over_time`, `quantile_over_time`.

```
quantile_over_time(0.95,
process_resident_memory_bytes[10m])
```

This function takes an additional parameter to indicate the quantile. In this example, `0.95` gives the 95th percentile.

COUNTER RANGE VECTORS

```
rate(process_cpu_seconds_total[1m])
```

This calculates per-second increase of a counter, allowing for resets and extrapolating at edges to provide better results. This is the most common function you will use in PromQL, and sometimes can be found with function `sum`; always use them in this order: `sum(rate(foo))`. It supports counter metrics resets. Try out the difference between with and without `rate`.

```
increase(process_cpu_seconds_total[15m])
```

It returns the increase across the period. In the example this is a per quarter of hour increase. Only use `increase` for display, use `rate` in rules and alerts.

```
irate(http_requests_total{status_code="401"}
[15m])
```

This only looks at the last two data points and returns the per-second rate. It produces very responsive graphs, but doesn't do well for alerting or longer time frames.

```
resets(http_requests_total[1h])
```

It counts the number of counter resets. Useful mostly for debugging. If you want to track how often a process restarts, a timestamp gauge and `changes` is better.

SUBQUERYING

```
max_over_time
(rate(process_cpu_seconds_total[5m])[1h:])
```

A subquery takes an instant vector expression and evaluates it at various points in a time range, producing a range vector.

HISTOGRAMS

```
histogram_quantile(0.95, rate
(request_duration_seconds_bucket
{status_code="401"}[10m]))
```

It calculates the given quantile. It should be passed a gauge instant vector, which means applying `rate` or `irate` first.

SWITCHING TYPES

```
scalar(round(vector(time()), 3600))
```

You can convert scalars to vectors with `vector`, and vectors to scalars with `scalar`.

ALTERING LABELS

```
label_replace
(node_filesystem_size_bytes,"example",
"$1","device","(.*)")
```

The most complicated function in PromQL. It allows you to set a label based on a regex applied to a label. If the regex doesn't match, the original time series is returned. It takes a vector, the new label, a replacement string, the old label and a regex for the label.

```
label_join(node_filesystem_size_bytes
{device="tmpfs",fstype="tmpfs"}, "foo", ",",
"device", "fstype")
```

This joins the values of multiple labels with the given separator. It takes a vector, the new label, a separator string, and existing labels.

SORTING

```
sort(node_filesystem_size_bytes)
```

It returns a sorted (ascendingly) vector. A descendent version exists called `sort_desc`. NaNs always sort to the end, so these functions aren't simple reverses of each other.

MISSING VALUES

```
absent(up{job="node"})
```

This returns nothing if there are any time series in the vector. If there are no time series, it returns `1` with labels taken from the selector - so use this directly on selectors.

SELECTORS

BASIC

```
node_cpu_seconds_total
```

It shows the time-series of a metric. Same as `{__name__="node_cpu_seconds_total"}`.

```
node_cpu_seconds_total{mode="idle"}
```

Selector with label to return matching time series. Available matchers are: `=` (equal), `!=` (not equal), `=~` (matches a regex), `!~` (doesn't match a regex). Multiple labels can be specified separated by comma.

RANGE VECTOR

```
node_cpu_seconds_total[5m]
```

A vector selector, with an added duration. This is an inclusive match over time, so all samples from exactly 10 minutes ago up as far as the query evaluation time will be included. A single suffix must be provided (`ms` milliseconds, `s`econds, `m`inutes, `h`ours, `d`ays, `w`eeks, `y`ears).

OFFSET MODIFIER

```
up offset 10m
```

When a query is executed, a time is provided. This defaults to now. Instant vectors return the most recent sample before this, in accordance with staleness. Range vectors return everything from the query time looking back as far as the duration. This uses the same duration syntax as range vectors.

AGGREGATORS & OPERATORS

AGGREGATORS

```
count(up{job="demo"})
```

They take an instant vector, and return an instant vector. Available aggregators are: `sum`, `count`, `count_values`, `min`, `max`, `avg`, `stddev`, `stdvar`, `topk`, `bottomk`, `quantile`. Special cases are `topk` and `bottomk` that also take a number of lines, `quantile` lines that takes the percentile, and `count_values` that needs a specified value to count.

CHOOSING AGGREGATION LABELS

```
count without(device)
(node_disk_read_bytes_total)
```

The without modifier says to use all the labels in the output - except the ones listed. Another modifier is `by`, which says to only output the given labels.

OPERATORS

```
prometheus_tsdb_head_active_appenders + 2
```

Basic arithmetic (`+ - * / % ^`), comparison (`== != > >= < <=`), and logical (`and unless or`) operators as found in many programing languages, are also found here.

```
node_filesystem_files_free /
node_filesystem_files
```

Prometheus has to match up time series from the left hand side (LHS) and right hand side (RHS). If everything matches up perfectly ignoring metric names on LHS and RHS, you'll get an output time series.

MATCHING TIME SERIES

```
  sum(rate(node_cpu_seconds_total{mode="idle"}
[1m])) / on (instance, job, mode)
sum(rate(node_cpu_seconds_total[1m]))
```

If things don't match up, we can use the `on` aggregator. With it, we can specify where the two time series are going to match. It needs to have a set of labels that doesn't result in an ambiguous match.

```
  sum without (cpu)(rate
(node_cpu_seconds_total{mode="idle"}[5m]))
/ ignoring (mode)  sum without (cpu,mode)
(rate(node_cpu_seconds_total[5m]))
```

If it's easier for you to specify where two time series do not match, then the `ignoring` aggregator is just for you. If there's exactly one time series in matching buckets on both sides, operation will go ahead. If there's zero on one side, operation will not go ahead due to no match. If there's many time series in a bucket, you'll get an error.

MANY TO ONE

```
  sum without (cpu)(rate
(node_cpu_seconds_total{mode="idle"}[5m])) /
ignoring(mode) group_left   sum without
(mode, cpu)(rate(node_cpu_seconds_total[5m]))
```

Sometimes you want to do a many-to-one match. We would use the `group_left` modifier. Many is the left side. We keep all the labels on the many side, and copy over any labels listed in the `group_left`. Aggregator `group_right` is similar, switching LHS and RHS. This should be accompanied by `on` or `ignoring`, as specifying matching labels is needed.

COMPARISON OPERATORS

```
up != 0
```

Matching works the same as for binary operators. `Vector` and `vector` will keep the LHS time series if the comparison is true. `Scalar` and `vector` will keep the vector time series if the comparison is true. `Scalar` and `scalar` is an error. No sane vector can be returned here.

BOOL VS FILTERING

```
count(process_max_fds < 1024)
```

The default mode of comparisons is to filter. You can add the `bool` modifier to return `0`/`1` instead. 1 means true. Modifier `bool` also allows scalar/scalar comparisons to work. Filtering is what you want for alerts. However it is difficult to work with elsewhere as you can end up with no time series in a bucket if everything is filtered.

LOCAL/SET OPERATORS

```
rate(process_cpu_seconds_total[5m]) unless
process_resident_memory_bytes < 1e6
```

These are many-to-many operators. Operators `and`, `unless`, and `or` don't follow quite the same matching rules. Operator `and` returns LHS time series if there's at least one matching time series on RHS. Operator `unless` is opposite of and, needs there to be no match on RHS. Operator `or` returns the LHS if present, otherwise the RHS.

For more information on Prometheus or PromQL, please refer to:
https://prometheus.io/docs/prometheus/latest/querying/basics

Learn how Sysdig simplifies Prometheus and PromQL on:
https://sysdig.com/product/monitor/prometheus

To discover Prometheus based monitoring integrations, don't miss:
https://promcat.io/

sysdig