Labels

Labels are a key part of Prometheus, and one of the things that make it powerful. In this chapter you will learn what labels are, where they come from, and how you can add them to your own metrics.

What Are Labels?

- Labels are key-value pairs associated with time series that, in addition to the metric name, uniquely identify them.
- If you had a metric for HTTP requests that was broken out by path, you might try putting the path in the metric name:
  - `http_requests_login_total`
  - `http_requests_logout_total`
  - `http_requests_adduser_total`
  - `http_requests_comment_total`
  - `http_requests_view_total`

What Are Labels?

- These would be difficult for you to work with in PromQL.
- In order to calculate the total requests you would either need to know every possible HTTP path or do some form of potentially expensive matching across all metric names.
- Accordingly, this is an antipattern you should avoid.
- Instead, to handle this common use case, Prometheus has labels. In the preceding case you might use a path label:

```
http_requests_total{path="/login"}
http_requests_total{path="/logout"}
http_requests_total{path="/adduser"}
http_requests_total{path="/comment"}
http_requests_total{path="/view"}
```

What Are Labels?

- You can then work with the `http_requests_total` metric with all its path labels as one.
- With PromQL you could get an overall aggregated request rate, the rate of just one of the paths, or what proportion each request is of the whole.
- You can also have metrics with more than one label.
- There is no ordering on labels, so you can aggregate by any given label while ignoring the others, or even aggregate by several of the labels at once.

Instrumentation and Target Labels

- Labels come from two sources, instrumentation labels and target labels.
- When you are working in PromQL there is no difference between the two, but it's important to distinguish between them in order to get the most benefits from labels.
- **Instrumentation labels**, as the name indicates, come from your instrumentation.
- They are about things that are known inside your application or library, such as the type of HTTP requests it receives, which databases it talks to, and other internal specifics.
- **Target labels** identify a specific monitoring target; that is, a target that Prometheus scrapes.
- A target label relates more to your architecture and may include which application it is, what datacenter it lives in, if it is in a development or production environment, which team owns it, and of course, which exact instance of the application it is.
- Target labels are attached by Prometheus as part of the process of scraping metrics.
- Different Prometheus servers run by different teams may have different views of what a "team," "region," or "service" is, so an instrumented application should not try to expose such labels itself.
- Accordingly, you will not find any features in client libraries to add labels across all metrics of a target.

Metric

- As you may have noticed by now, the word metric is a bit ambiguous and means different things depending on context.
- It could refer to a metric family, a child, or a time series:
  - # HELP latency_seconds Latency in seconds.
  - # TYPE latency_seconds summary
  - latency_seconds_sum{path="/foo"} 1.0
  - latency_seconds_count{path="/foo"} 2.0
  - latency_seconds_sum{path="/bar"} 3.0
  - latency_seconds_count{path="/bar"} 4.0
- latency_seconds_sum{path="/bar"} is a time series, distinguished by a name and labels. This is what PromQL works with.
- latency_seconds{path="/bar"} is a child. For a summary it contains both the _sum and _count time series with those labels.
- latency_seconds is a metric family. It is only the metric name and its associated type. This is the metric definition when using a client library.
- For a gauge metric with no labels, the metric family, child, and time series are the same.

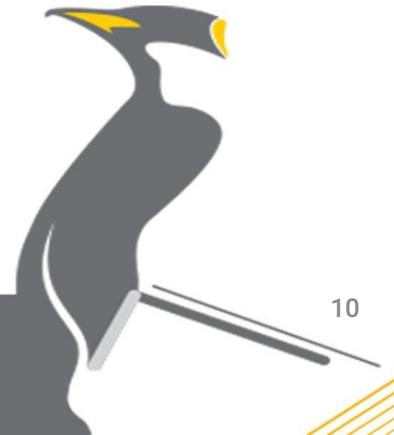
Label Patterns

- Prometheus only supports 64-bit floating-point numbers as time series values, not any other data types such as strings.
- But label values are strings, and there are certain limited use cases where it is okay to (ab)use them without getting too far into logs based monitoring.

INTRODUCTION TO PROMQL

PromQL is the **Prometheus Query Language**. While it ends in QL, you will find that it is not an SQL-like language, as SQL languages tend to lack expressive power when it comes to the sort of calculations you would like to perform on time series.

Labels are a key part of PromQL, and you can use them not only to do arbitrary aggregations but also to join different metrics together for arithmetic operations against them. There are a wide variety of functions available to you from prediction to date and math functions.



Aggregation Basics

Gauge

- Gauges are a snapshot of state, and usually when aggregating them you want to take a **sum**, **average**, **minimum**, or **maximum**.
- Calculate total filesystem size on each machine with:

```
sum without(device, fstype, mountpoint)(node_filesystem_size_bytes)
```

- We can test the value with calculating all values of `node_filesystem_free_bytes` expression.

Gauge

- Since there is only one Node exporter being scraped by Prometheus there is only one result, but if you were scraping more then you would have a result for each of the Node exporters.
- Let's go a step further and remove the instance label with:

```
sum without(device, fstype, mountpoint, instance)(node_filesystem_size_bytes)
```

- `max` would tell you the size of the biggest mounted filesystem on each machine:

```
max without(device, fstype, mountpoint)(node_filesystem_size_bytes)
```

- This predictability in what labels are returned is important for vector matching with operators.
- You are not limited to aggregating metrics about one type of job.
- For example, to find the average number of file descriptors open across all your jobs you could use:

```
avg without(instance, job)(process_open_fds)
```

Counter

- Counters track the number or size of events, and the value your applications expose on their /metrics is the total since it started.
- But that total is of little use to you on its own, **what you really want to know is how quickly the counter is increasing over time.**
- This is usually done using the `rate` function, though the increase and `rate` functions also operate on counter values.
- Calculate the amount of network traffic received per second:

```
rate(node_network_receive_bytes_total[5m])
```

- Some estimation is used to fill in the gaps between the data points you have and the boundaries of the range.

Counter

- The output of rate is a gauge, so the same aggregations apply as for gauges.

```
sum without(device)(rate(node_network_receive_bytes_total[5m]))
```

```
sum without(instance)(rate(node_network_receive_bytes_total{device="eth0"}[5m]))
```

- There is no ordering or hierarchy within labels, allowing you to aggregate by as many or as few labels as you like.

Summary

- A summary metric will usually contain both a `_sum` and `_count`, and sometimes a time series with no suffix with a quantile label.
- The _sum and _count are both counters.
- Prometheus exposes a `http_response_size_bytes` summary, for the amount of data some of its HTTP APIs return.
- `http_response_size_bytes_count` tracks the number of requests, and as it is a counter you must use rate before aggregating away its handler label:

```
sum without(handler)(rate(http_response_size_bytes_count[5m]))
```

- This gives you the total per-second HTTP request rate, and as the Node exporter also returns this metric you will see both jobs in the result:

Summary

- Similarly, `http_response_size_bytes_sum` is a counter with the number of bytes each handle has returned, so the same pattern applies:

```
sum without(handler)(rate(http_response_size_bytes_sum[5m]))
```

- This will return results with the same labels as the previous query, but the values are larger as responses tend to return many bytes.
- The power of a summary is that it allows you to calculate the average size of an event, in this case the average amount of bytes that are being returned in each response.

Summary

- If you had three responses of size 1, 4, and 7, then the average would be their sum divided by their count, which is to say 12 divided by 3.
- The same applies to the summary.
- You divide the `_sum` by the `_count` to get an average over a time period:

```
sum without(handler)(rate(http_response_size_bytes_sum[5m]))
/
sum without(handler)(rate(http_response_size_bytes_count[5m]))
```

- The division operator matches the time series with the same labels and divides, giving you the same two time series out but with the average response size over the past 5 minutes as a value.

Summary

- When calculating an average, it is important that you first aggregate up the sum and count, and only as the last step perform the division.
- Otherwise, you could end up averaging averages, which is not statistically valid.
- For example, if you wanted to get the average response size across all instances of a job, you could do:

```
sum without(instance)(sum without(handler)(rate(http_response_size_bytes_sum[5m])))
/
sum without(instance)(sum without(handler)(rate(http_response_size_bytes_count[5m])))
```

- However, it'd be incorrect to do:

```
avg without(instance)(sum without(handler)(rate(http_response_size_bytes_sum[5m]))
/
sum without(handler)(rate(http_response_size_bytes_count[5m])))
```

- It is incorrect to average an average, and both the division and avg would be calculating averages.

Histogram

- Histogram metrics allow you to track the distribution of the size of events, allowing you to calculate quantiles from them.
- Use histograms to calculate the 0.9 quantile (which is also known as the 90th percentile) latency.
- Prometheus 2.2.1 exposes a histogram metric called `prometheus_tsdb_compaction_duration_seconds` that tracks how many seconds compaction takes for the time series database.
- This histogram metric has time series with a `_bucket` suffix called `prometheus_tsdb_compaction_duration_seconds_bucket`.
- Each bucket has a `le` label, which is a counter of how many events have a size less than or equal to the bucket boundary.
- This is an implementation detail you largely need not worry about as the histogram_quantile function takes care of this when calculating quantiles.

Histogram

- For example, the 0.90 quantile would be:

```
histogram_quantile(0.90,rate(prometheus_tsdb_compaction_duration_seconds_bucket[1d]))
```

- As `prometheus_tsdb_compaction_duration_seconds_bucket` is a counter you must first take a rate.
- Compaction usually only happens every two hours, so a one-day time range is used here so you will see a result in the expression browser such as:

```
{instance="localhost:9090",job="prometheus"} 7.720000000000001
```

- This indicates that the 90th percentile latency of compactions is around 7.72 seconds.
  - ✓ As there will usually only be 12 compactions in a day, the 90th percentile says that 10% of compactions take longer than this, which is to say one or two compactions.

Histogram

- This is something to be aware of when using quantiles.
- For example, if you want to calculate a 0.999 quantile you should have several thousand data points to work with in order to produce a reasonably accurate answer.
- If you have fewer than that, single outliers could greatly affect the result, and you should consider using lower quantiles to avoid making statements about your system that you have insufficient data to backup.

Histogram

- Similar to when taking averages, using histogram_quantile should be the last step in a query expression.
- Quantiles cannot be aggregated, or have arithmetic performed upon them, from a statistical standpoint.
- Accordingly, when you want to take a histogram of an aggregate, first aggregate up with sum and then use histogram_quantile:

```
histogram_quantile(0.90,sum
without(instance)(rate(prometheus_tsdb_compaction_duration_bucket[1d])))
```

- This calculates the 0.9 quantile compaction duration across all of your Prometheus servers, and will produce a result without an instance label:

```
{job="prometheus"} 7.720000000000001
```

Histogram

- Histogram metrics also include _sum and _count metrics, which work exactly the same as for the summary metric.
- You can use these to calculate average event sizes, such as the average compaction duration:

```
sum
without(instance)(rate(prometheus_tsdb_compaction_duration_sum[1d]))
/
sum
without(instance)(rate(prometheus_tsdb_compaction_duration_count[1d]))
```

- This would produce a result like:

```
{job="prometheus"} 3.1766430400714287
```

Selectors

- Working with all the different time series with different label values for a metric can be a bit overwhelming, and potentially confusing if a metric is coming from multiple different types of server.
- Usually you will want to narrow down which time series you are working on.
- You almost always will want to limit by job label, and depending on what you are up to, you might want to only look at one instance or one handler, for example.
- This limiting by labels is done using selectors.

```
process_resident_memory_bytes{job="node"}
```

- is a selector that will return all time series with the name `process_resident_memory_bytes` and a job label of node.
- This particular selector is most properly called a instant vector selector as it returns the values of the given time series at a given instant.
- Vector here basically means a one-dimensional list, as a selector can return zero or more time series, and each time series will have one sample.
- The job="node" is called a matcher, and you can have many matchers in one selector that are ANDed together.

Matchers

- There are four matchers.
- =
  - ✓ is the equality matcher; for example, job="node".
  - ✓ With this you can specify that the returned time series has a label name with exactly the given label value.
  - ✓ As an empty label, value is the same as not having that label, you could use foo="" to specify that the foo label not be present.

Matchers

- !=
  - ✓ is the negative equality matcher; for example, job!="node". With this you can specify that the returned time series do not have a label name with exactly the given label value.
- =~
  - ✓ is the regular expression matcher; for example, job=~"n.*".
  - ✓ With this you specify that for the returned time series the given label's value will be matched by the regular expression.
  - ✓ The regular expression is fully anchored, which is to say that the regular expression a will only match the string a, and not xa or ax.
  - ✓ You can prepend or suffix your regular expression with .* if you do not want this behaviour.
  - ✓ As with relabelling, the RE2 regular expression engine is used.

Matchers

- !~
  - ✓ is the negative regular expression matcher.
  - ✓ RE2 does not support negative lookahead expressions, so this provides you with an alternative way to exclude label values based on a regular expression.
  - ✓ You can have multiple matchers with the same label name in a selector, which can be a substitute for negative lookahead expressions.
  - ✓ For example, to find the size of all filesystems mounted under /run but not /run/user, you could use:

```
node_filesystem_size_bytes{job="node",mountpoint=~"/run/.*",
mountpoint!~"/run/user/.*"}
```

- Internally the metric name is stored in a label called `__name__`, so

```
process_resident_mem ory_bytes{job="node"} =
{__name__="process_resident_memory_bytes",job="node"}
```

Matchers

- The selector {} returns an error, which is a safety measure to avoid accidentally returning all the time series inside the Prometheus server as that could be expensive.
- To be more precise, at least one of the matchers in a selector must not match the empty string.
  - ✓ So `{foo=""}`, `{foo!=""}`, and `{foo=~".*"}` will return an error, while `{foo="",bar="x"}` or `{foo=~".+"}` are permitted.

Instant Vector

- An instant vector selector returns an instant vector of the most recent samples before the query evaluation time, which is to say a list of zero or more time series.
- Each of these time series will have one sample, and a sample contains both a value and a timestamp.
- while the instant vector returned by an instant vector selector has the timestamp of the original data, any instant vectors returned by other operations or functions will have the timestamp of the query evaluation time for all of their values.
- When you ask for current memory usage you do not want samples from an instance that was turned down days ago to be included, a concept known as staleness.

Instant Vector

- In Prometheus 1.x this was handled by returning time series that had a sample no more than 5 minutes before the query evaluation time. This largely worked but had downsides such as double counting if an instance restarted with a new instance label within that 5-minute window.
- Prometheus 2.x has a more sophisticated approach.
- If a time series disappears from one scrape to the next, or if a target is no longer returned from service discovery, a special type of sample called a stale marker is appended to the time series.

Instant Vector

- When evaluating an instant vector selector, all time series satisfying all the matchers are first found, and the most recent sample in the 5 minutes before the query evaluation time is still considered.
- If the sample is a normal sample, then it is returned in the instant vector, but if it is a stale marker, then that time series will not be included in that instant vector.
- The outcome of all of this is that when you use an instant vector selector, time series that have gone stale are not returned.

Range Vector

- There is a second type of selector you have already seen, called the range vector selector.
- Unlike an instant vector selector which returns one sample per time series, a range vector selector can return many samples for each time series.
- Range vectors are always used with the rate function; for example:
  - ✓ `rate(process_cpu_seconds_total[1m])`
  - ✓ `process_cpu_seconds_total[1m]`

- This is because range vectors preserve the actual timestamps of the samples, and the scrapes for different targets are distributed in order to spread load more evenly.
- While you can control the frequency of scrapes and rule evaluations, you cannot control their phase or alignment. If you have a 10-second scrape interval and hundreds of targets, then all those targets will be scraped at different points in a given 10-second window. Put another way, your time series all have slightly different ages.
- This generally won't matter to you in practice, but can lead to artifacts as fundamentally metrics-based monitoring systems like Prometheus produce (quite good) estimates rather than exact answers.
- You will very rarely look at range vectors directly. It only comes up when you need to see raw samples when debugging. Almost always you will use a range vector with a function such as rate or avg_over_time that takes a range vector as an argument.
- Staleness and stale markers have no impact on range vectors; you will get all the normal samples in a given range. Any stale markers also in that range are not returned by a range vector selector.

Durations

- Durations in Prometheus as used in PromQL and the configuration file support several units.
- You have already seen m for minute.
- Suffix Meaning You can use one unit with integers, so 90m is valid but 1h30m and 1.5h are not.
- Leap years and leap seconds are ignored, 1y is always 60*60*24*365 seconds.

Durations

ms	Milliseconds
s	Seconds, which have 1,000 milliseconds
M	Minutes, which have 60 seconds
H	Hours, which have 60 minutes
D	Days, which have 24 hours
W	Weeks, which have 7 days
Y	Years, which have 365 days

Offset

- There is a modifier you can use with either type of vector selector called offset.
- Offset allows you to take the evaluation time for a query, and on a per-selector basis put it further back in time.

```
process_resident_memory_bytes{job="node"} offset 1h
```

- Offset is not used much in simple queries like this, as it would be easier to change the evaluation time for the whole query instead.
- Where this can be useful is when you only want to adjust one selector in a query expression.

```
process_resident_memory_bytes{job="node"}
-
process_resident_memory_bytes{job="node"} offset 1h
```

- The same approach works with range vectors:

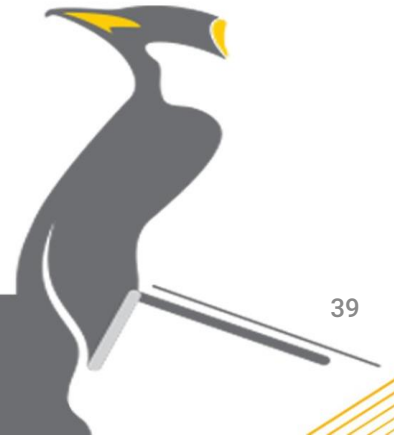
```
rate(process_cpu_seconds_total{job="node"}[5m])
-
rate(process_cpu_seconds_total{job="node"}[5m] offset 1h)
```

- offset only allows you to look further back into the past.
- This is because having “historical” graphs change as new data comes in would be counterintuitive.
- If you wish to have a negative offset, you can instead move the query evaluation time forward and add offsets to all the other selectors in an expression.

AGGREGATION OPERATORS

Aggregation is important. With applications with thousands or even just tens of instances it’s not practical for you to sift through each instance’s metrics individually. Aggregation allows you to summaries metrics not just within one application, but across applications too.

There are 11 aggregation operators in all, with 2 optional clauses: without and by.



Grouping

- Before talking about the aggregation operators themselves, you need to know about how time series are grouped.
- Aggregation operators work only on instant vectors, and they also output instant vectors.
- Let's say you have the following time series in Prometheus:

```
node_filesystem_size_bytes{device="/dev/sda1",fstype="vfat",instance="localhost:9100",job="node",mountpoint="/boot/efi"} 100663296
node_filesystem_size_bytes{device="/dev/sda5",fstype="ext4",instance="localhost:9100",job="node",mountpoint="/"} 90131324928
```

- There are three instrumentation labels: device, fstype, and mountpoint.
- There are also two target labels: job and instance. Target and instrumentation labels are a notion that you and I have, but which PromQL knows nothing about.
- All labels are the same when it comes to PromQL, no matter where they originated from.

without

- Generally you will always know the instrumentation labels, as they rarely change.
- But you do not always know the target labels in play, as an expression you write might be used by someone else on metrics originating from different scrape configs or Prometheus servers that might also have added in other target labels across a job, such as a env or cluster label.
- You might even add in such target labels yourself at some point, and it'd be nice not to have to update all your expressions.
- When aggregating metrics you should usually try to preserve such target labels, and thus you should use the without clause when aggregating to specify the specific labels you want to remove.
- sum without(fstype, mountpoint)(node_filesystem_size_bytes)

without

- will group the time series, ignoring the fstype and mountpoint labels, into three groups:
- # Group {device="/dev/sda1",instance="localhost:9100",job="node"}
- # Group {device="/dev/sda5",instance="localhost:9100",job="node"}
- # Group {device="tmpfs",instance="localhost:9100",job="node"}
- and the sum aggregator will apply within each of these groups, adding up the values of the time series and return one sample per group.
- Notice that the instance and job labels are preserved, as would be any other labels that had been present.
- This is useful because any alerts you created that included this expression somehow would have additional target labels like env or cluster preserved.
- This provides context for your alerts and makes them more useful (also useful when graphing).

without

- The metric name has also been removed, as this is an aggregation of the node_file system_size_bytes metric rather than the original metric.
- When a PromQL operator or function could change the value or meaning of a time series, the metric name is removed.
- It is valid to provide no labels to the without.

sum without()(node_filesystem_size_bytes)

- will give you the same result as:

node_filesystem_size_bytes

- with the only difference being the metric name being removed.

by

- In addition to without there is also the by clause.
- Where without specifies the labels to remove, by specifies the labels to keep.
- Accordingly, some care is required when using by to ensure you don't remove target labels that you would like to propagate in your alerts or use in your dashboards.
- You cannot use both by and without in the same aggregation.

```
sum by(job, instance, device)(node_filesystem_size_bytes)
```

- will produce the same result as the querying in the preceding section using without:

```
{device="/dev/sda1",instance="localhost:9100",job="node"} 100663296
{device="/dev/sda5",instance="localhost:9100",job="node"} 90131324928
{device="tmpfs",instance="localhost:9100",job="node"} 2486128640
```

by

- However, if instance or job had not been specified, then they wouldn't have defined the group and would not be in the output.
- Generally, you should prefer to use `without` rather than `by` for this reason.
- There are two cases where you might find by more useful.
- The first is that unlike `without`, `by` does not automatically drop the __name__ label.
- This allows you to use expressions like:

```
sort_desc(count by(__name__)({__name__=~".+"}))
```

- to investigate how many time series have the same metric names.

Operators

- All 11 aggregation operators use the same grouping logic.
- You can control this with one of without or by.
- What differs between aggregation operators is what they do with the grouped data.

sum

- sum is the most common aggregator; it adds up all the values in a group and returns that as the value for the group.

```
sum without(fstype, mountpoint, device)(node_filesystem_size_bytes)
```

- would return the total size of the filesystems of each of your machines.
- When dealing with counters, it is important that you take a rate before aggregating with sum:

```
sum without(device)(rate(node_disk_read_bytes_total[5m]))
```

- If you were to take a sum across counters directly, the result would be meaningless, as different counters could have been initialized at different times depending on when the exporter started, restarted, or any particular children were first used.

count

- The count aggregator counts the number of time series in a group, and returns it as the value for the group.

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

- would return the number of disk devices a machine has.
  - ✓ Here it is okay not to use rate with a counter, as you care about the existence of the time series rather than its value.

Unique label values

- You can also use count to count how many unique values a label has. For example, to count the number of CPUs in each of your machines you could use:
- `count without(cpu)(count without (mode)(node_cpu_seconds_total))`
- The inner count removes the other instrumentation label, mode, returning one time series per CPU per instance:

```
{cpu="0",instance="localhost:9100",job="node"} 8
{cpu="1",instance="localhost:9100",job="node"} 8
{cpu="2",instance="localhost:9100",job="node"} 8
{cpu="3",instance="localhost:9100",job="node"} 8
```

- The outer count then returns the number of CPUs that each instance has:

```
{instance="localhost:9100",job="node"} 4
```

- If you didn't want a per-machine breakdown, such as if you were investigating if certain labels had high cardinality, you could use the by modifier to look at only one label:

```
count(count by(cpu)(node_cpu_seconds_total))
```

- Which would produce a single sample with no labels, such as:

```
{} 4
```

avg

- The avg aggregator returns the average of the values of the time series in the group as the value for the group.

```
avg without(cpu)(rate(node_cpu_seconds_total[5m]))
```

- This gives you the exact same result as:

```
sum without(cpu)(rate(node_cpu_seconds_total[5m]))
/
count without(cpu)(rate(node_cpu_seconds_total[5m]))
```

- but it is both more succinct and more efficient to use avg.

avg

- When using avg sometimes you may find that a NaN in the input is causing the entire result to become NaN.
  - ✓ This is because any floating-point arithmetic that involves NaN will have NaN as a result.
- You may wonder how to filter out these NaNs in the input, but that is the wrong question to ask.
- Usually this is due to attempting to average averages, and one of the denominators of the first averages was 0.5 Averaging averages is not statistically valid, so what you should do instead is aggregate using sum and then finally divide.

stddev and stdvar

- The standard deviation is a statistical measure of how spread out a set of numbers is.
- For example, if you had the numbers [2,4,6] then the standard deviation would be 1.633.
- The numbers [3,4,5] have the same average of 4, but a standard deviation of 0.816.
- The main use of the standard deviation in monitoring is to detect outliers.
- In normally distributed data you would expect that about 68% of samples would be within one standard deviation of the mean, and 95% within two standard deviations.
- If one instance in a job has a metric several standard deviations away from the average, that's a good indication that something is wrong with it.

stddev and stdvar

- For example, you could find all instances that were at least two standard deviations above the average using an expression such as:

```
some_gauge > ignoring (instance) group_left() (avg
without(instance)(some_gauge)
+
2 * stddev without(instance)(some_gauge))
```

stddev and stdvar

- This uses one-to-many vector matching, which will be discussed in “Many-to-One and group_left”.
- If your values are all tightly bunched then this may return some time series that are more than two standard deviations away, but still operating normally and close to the average.
- You could add an additional filter that the value has to be at least say 20% higher than the average to protect against this.
- This is also a rare case where it is okay to take an average of an average, such as if you applied this to average latency.
- The standard variance is the standard deviation squared8 and has statistical uses.

min and max

- The `min` and `max` aggregators return the minimum or maximum value within a group as the value of the group, respectively.
- The same grouping rules apply as elsewhere, so the output time series will have the labels of the group.

```
max without(device, fstype,
mountpoint)(node_filesystem_size_bytes)
```

- will return the size of the biggest filesystem on each instance.
- The max and min aggregators will only return NaN if all values in a group are NaN.

topk and bottomk

- `topk` and `bottomk` are different from the other aggregators discussed so far in three ways.
  - ✓ First, the labels of time series they return for a group are not the labels of the group;
  - ✓ second, they can return more than one time series per group;
  - ✓ and third, they take an additional parameter.
- topk returns the k time series with the biggest values.

```
topk without(device, fstype, mountpoint)(2, node_filesystem_size_bytes)
```

- would return up to two time series per group, such as:

```
node_filesystem_size_bytes{device="/dev/sda5",fstype="ext4",instance="localhost
:9100",job="node",mountpoint="/"} 90131324928
node_filesystem_size_bytes{device="tmpfs",fstype="tmpfs",
instance="localhost:9100",job="node",mountpoint="/run"} 826961920
```

topk and bottomk

- As you can see, topk returns input time series with all their labels, including the __name__ label, which holds the metric name.
- The result is also sorted.
- bottomk is the same as topk, except that it returns the k time series with the smallest values rather than the k biggest values.
- Both aggregators will where possible avoid returning time series with NaN values.

quantile

- The quantile aggregator returns the specified quantile of the values of the group as the group's return value.
- As with topk, quantile takes a parameter.
- So, for example, if I wanted to know across the different CPUs in each of my machines what the 90th percentile of the system mode CPU usage is I could use:

```
quantile without(cpu)(0.9, rate(node_cpu_seconds_total{mode="system"}[5m]))
```

- which produces a result like:

```
{instance="localhost:9100",job="node",mode="system"} 0.024558620689654007
```

- This means that 90% of my CPUs are spending at least 0.02 seconds per second in the system mode. This would be a more useful query if I had tens of CPUs in my machine, rather than the four it actually has.

quantile

- In addition to the mean, you could use quantile to show the median, 25th, and 75th percentiles on your graphs.

```
# average, arithmetic mean
avg without(instance)(0.5, rate(process_cpu_seconds_total[5m]))
# 0.25 quantile, 25th percentile, 1st or lower quartile
quantile without(instance)(0.25, rate(process_cpu_seconds_total[5m]))
# 0.5 quantile, 50th percentile, 2nd quartile, median
quantile without(instance)(0.5, rate(process_cpu_seconds_total[5m]))
# 0.75 quantile, 75th percentile, 3rd or upper quartile
quantile without(instance)(0.75, rate(process_cpu_seconds_total[5m]))
```

- This would give you a sense of how your different instances for a job are behaving, without having to graph each instance individually.
- This allows you to keep your dashboards readable as the number of underlying instances grows.
- Personally I find that per-instance graphs break down somewhere around three to five instances.

quantile, histogram_quantile, and quantile_over_time

- As you may have noticed by now, there is more than one PromQL function or operator with quantile in the name.
- The quantile aggregator works across an instant vector in an aggregation group.
- The `quantile_over_time` function works across a single time series at a time in a range vector.
- The `histogram_quantile` function works across the buckets of one histogram metric child at a time in an instant vector.

count_values

- The final aggregation operator is `count_values`.
- Like `topk` it takes a parameter and can return more than one time series from a group.
- What it does is build a frequency histogram of the values of the time series in the group, with the count of each value as the value of the output time series and the original value as a new label.

count_values

- That’s a bit of a mouthful, so I’ll show you an example software_version{instance="a",job="j"} 7

```
software_version{instance="b",job="j"} 4
software_version{instance="c",job="j"} 8
software_version{instance="d",job="j"} 4
software_version{instance="e",job="j"} 7
software_version{instance="f",job="j"} 4
```

- If you evaluated the query:
- count_values without(instance)("version", software_version)
- on these time series you would get the result:

```
{job="j",version="7"} 2
{job="j",version="8"} 1
{job="j",version="4"} 3
```

count_values

- There is no bucketing involved when the frequency histogram is created; the exact values of the time series are used.
- Thus this is only really useful with integer values and where there will not be too many unique values.
- This is most useful with version numbers, or with the number of objects of some type that each instance of your application sees.
- If you have too many versions deployed at once, or different applications are continuing to see different numbers of objects, something might be stuck somewhere.
- `count_values` can be combined with count to calculate the number of unique values for a given aggregation group.
- For example, the number of versions of software that are deployed can be calculated with:

```
count without(version)(count_values without(instance)("version",
software_version))
```

count_values

- You could also combine `count_values` with count in the other direction; for example, to see how many of your machines had how many disk devices:

```
count_values without(instance)("devices",count
without(device) (node_disk_io_now))
```

- In my case I have one machine with five disk devices:

```
{devices="5",job="node"} 1
```