BINARY OPERATORS

You will want to do more with your metrics than simply aggregate them, which is where the binary operators come in. Binary operators are operators that take two operands, such as the addition and equality operators.

Binary operators in Prometheus allow for more than simple arithmetic on instant vectors; you can also apply a binary operator to two instant vectors with grouping based on labels. This is where the real power of PromQL comes out, allowing classes of analysis that few other metrics systems offer.

PromQL has three sets of binary operators: **arithmetic operators**, **comparison operations**, and **logical operators**.

Working with Scalars

- In addition to instant vectors and range vectors, there is another type of value known as a scalar.
  - **Scalars** are single numbers with no dimensionality.
- For example, 0 is a scalar with the value zero, while {} 0 is an instant vector containing a single sample with no labels and the value zero.

Arithmetic Operators

- You can use scalars in arithmetic with an instant vector to change the values in the instant vector.

```
process_resident_memory_bytes / 1024
```

  - ✓ Which is the process memory usage, in kilobytes.

- You will note that the division operator was applied to all time series in the instant vector returned by the `process_resident_memory_bytes` selector and that the metric name was removed as it is no longer the metric `process_resident_memory_bytes`.

Arithmetic Operators

- `+` addition
- `-` subtraction
- `*` multiplication
- `/` division
- `%` modulo
- `^` exponentiation

✓ The modulo operator is a floating-point modulo and can return non-integer results if you provide it with non-integer input.

```
5 % 1.5 = 0.5
```

Binary Arithmetic Operators

- You can also use binary arithmetic operators when both operands are scalars and the result will be a scalar.
- This is mostly useful for readability,
  - ✓ `(1024 * 1024 * 1024)` than it is `1073741824`.
- In addition, you can put the scalar operand on the left side of the operator and an instant vector on the right, so:

```
1e9 - process_resident_memory_bytes
```

Comparison Operators

- `==` equals
- `!=` not equals
- `>` greater than
- `<` less than
- `>=` greater than or equal to
- `<=` less than or equal to

✓ The comparison operators in PromQL are filtering.

```
process_open_fds{instance="localhost:9090",job="prometheus"} 14
process_open_fds{instance="localhost:9100",job="node"} 7
```

- And used an instant vector in a comparison with a scalar such as in the expression:

```
process_open_fds > 10
```

- Then you would get the result:

```
process_open_fds{instance="localhost:9090",job="prometheus"} 14
```

Comparison Operators

- As the value can't change, the metric name has been preserved.
- When comparing a scalar and an instant vector it doesn't matter which side each is on, it is always elements of the instant vector that are returned.
  - ✓ You cannot do a filtering comparison between two scalars, as to be consistent with arithmetic operations between two scalars it'd have to return a scalar.
    - This doesn't allow for filtering, as there's no way to have an empty scalar like you can have an empty instant vector.

bool modifier

- Filtering comparisons are primarily useful in alerting rules, and generally to be avoided elsewhere.
- To see how many of your processes for each job had more than ten open file descriptors.

```
count without(instance)(process_open_fds > 10)
{job="prometheus"} 1
```

- This correctly indicates that there is one Prometheus process with more than 10 open file descriptions.
- It does not report that the Node exporter has zero such processes.
- This is can be a subtle gotcha because as long as one time series is not filtered away everything seems to be okay.

bool modifier

- What you need is some way to do the comparison but not have it filter.
- This is what the bool modifier does; for each comparison it returns a 0 for false or a 1 for true.

```
process_open_fds > bool 10
```

  - ✓ Has one output sample per sample in the input instant vector.

- You can `sum` up to get the number of processes for each job that have more than 10 open file descriptors:

```
sum without(instance)(process_open_fds > bool 10)
{job="prometheus"} 1
{job="node"} 0
```

bool modifier

- You could use a similar approach to find the proportion of machines with more than four disk devices:

```
avg without(instance)(count without(device)(node_disk_io_now) > bool 4)
```

- This works by first using a count aggregation to find the number of disks reported by each Node exporter, then seeing how many have more than four, and finally averaging across machines to get the proportion.
- The trick here is that the values returned by the bool modifier are all 0 and 1, so the `count` is the total number of machines, and the `sum` is the number of machines meeting the criteria.
- The avg is the `count` divided by the `sum`, giving you a ratio or proportion.

bool modifier

- The bool modifier is the only way you can compare scalars, as:

```
42 <= bool 13
```

- Will return:

```
0
```

- Where the 0 indicates false.

Vector Matching

- Using operators between scalars and instant vectors will cover many of your needs, but using operators between two instant vectors is where PromQL's power really starts to shine.
- When you have a scalar and an instant vector, it is obvious that the scalar can be applied to each sample in the vector.
- With two instant vectors, which samples should apply to which other samples?
  - ✓This matching of the instant vectors is known as **vector matching**.

One-to-One

- In the simplest cases there will be a one-to-one mapping between your two vectors.

```
process_open_fds{instance="localhost:9090",job="prometheus"} 14
process_open_fds{instance="localhost:9100",job="node"} 7
process_max_fds{instance="localhost:9090",job="prometheus"} 1024
process_max_fds{instance="localhost:9100",job="node"} 1024
```

- Then when you evaluated the expression:

```
process_open_fds / process_max_fds
```

- you would get the result:

```
{instance="localhost:9090",job="prometheus"} 0.013671875
{instance="localhost:9100",job="node"} 0.0068359375
```

One-to-One

- What has happened here is that samples with exactly the same labels, except for the metric name in the label `__name__`, were matched together.
- That is to say that the two samples with the labels `{instance="localhost:9090",job="prometheus"}` got matched together, and the two samples with the labels `{instance="localhost:9100",job="node"}` got matched together.
- In this case there was a perfect match, with each sample on both sides of the operator being matched.
- If a sample on one side had no match on the other side, then it would not be present in the result, as binary operators need two operands.

One-to-One

- **Sometimes you will want to match two instant vectors whose labels do not quite match.**
  - ✓ Similar to how aggregation allows you to specify which labels matter, vector matching also has clauses controlling which labels are considered.
- **You can use the `ignoring` clause to ignore certain labels when matching, similar to how `without` works for aggregation.**
  - ✓ Say you were working with `node_cpu_seconds_total`, which has `cpu` and `mode` as instrumentation labels, and wanted to know what proportion of time was being spent in the **idle** mode for each instance.

```
sum without(cpu)(rate(node_cpu_seconds_total{mode="idle"}[5m]))
/
ignoring(mode) sum without(mode, cpu)(rate(node_cpu_seconds_total[5m]))
{instance="localhost:9100",job="node"} 0.8423353718871361
```

One-to-One

- Here the first sum produces an instant vector with a `mode="idle"` label, whereas the second `sum` produces an instant vector with no mode label.
- Usually vector matching will fail to match the samples, but with `ignoring(mode)` the mode label is discarded when the vectors are being grouped, and matching succeeds.
- As the `mode` label was not in the match group it is not in the output.

One-to-One

- The `on` clause allows you to consider only the labels you provide, similar to how `by` works for aggregation.

```
sum by(instance, job)(rate(node_cpu_seconds_total{mode="idle"}[5m]))
/ on(instance, job)
sum by(instance, job)(rate(node_cpu_seconds_total[5m]))
```

- will produce the same result as the preceding expression, but as with `by`, the `on` clause has the disadvantage that you need to know all labels that are currently on the time series or that may be present in the future in other contexts.
- The value that is returned for the arithmetic operators is the result of the calculation, but you may be wondering what happens for the comparison operators when there are two instant vectors.

One-to-One

- The answer is that the value from the left-hand side is returned.

```
process_open_fds > (process_max_fds * .5)
```

- will return for you the value of process_open_fds for all instances whose open file descriptors are more than halfway to the maximum.
- If you had instead used:

```
(process_max_fds * .5) < process_open_fds
```

- you would get half the maximum file descriptors as the return value.
- While the result will have the same labels, this value is less useful when alerting or when used in a dashboard!
- In general, a current value is more informative than the limit, so you should try to structure your math so that the most interesting number is on the left hand side of a comparison.

Many-to-One and group_left

- If you were to remove the matcher on mode from the preceding section and try to evaluate:

```
sum without(cpu)(rate(node_cpu_seconds_total[5m]))
/ ignoring(mode) sum without(mode, cpu)(rate(node_cpu_seconds_total[5m]))
```

  - ✓ **you would get the error.**
- many-to-one matching must be explicit (group_left/group_right).
- This is because the samples no longer match one-to-one, as there are multiple samples with different mode labels on the left-hand side for each sample on the righthand side.
- This can be a subtle failure mode, as a time series may appear later on that breaks your expression.
- You must specifically request that you want to do many-to-one matching using the `group_left` modifier.

group_left

- `group_left` lets you specify that there can be multiple matching samples in the group of the left-hand operand.

```
sum without(cpu)(rate(node_cpu_seconds_total[5m]))
/ ignoring(mode) group_left
sum without(mode, cpu)(rate(node_cpu_seconds_total[5m]))
```

- will produce one output sample for each different mode label within each group on the left-hand side:
- `group_left` always takes all of its labels from samples of your operand on the left hand side.
  - ✓ This ensures that the extra labels that are on the left side that require this to be many-to-one vector matching are preserved.
- You can use this approach to determine the proportion each label value within metric represents of the whole, as shown in the preceding example, or to compare a metric from a leader of a cluster against the replicas.

group_right

- There is also a `group_right` modifier that works in the same way as `group_left` except that the one and the many side are switched, with the many side now being your operand on the right-hand side.
- Any labels you specify in the `group_right` modifier are copied from the left to the right.
- For the sake of consistency, you should prefer `group_left`.

Many-to-Many and Logical Operators

- There are three logical or set operators you can use:
  - ✓ or union
  - ✓ and intersection
  - ✓ unless set subtraction
- All the logical operators work in a many-to-many fashion, and they are the only operators that work many-to-many.
- They are different from the arithmetic and comparison operators you have already seen in that no math is performed; all that matters is whether a group contains samples.

or operator

- Metrics with inconsistent children, or whose children are not always present, are tricky to work with, but you can deal with them using the `or` operator.
- How the `or` operator works is that for each group where the group on the left-hand side has samples, then they are returned; otherwise, the samples in the group on the right-hand side are returned.
- `or` can be used to substitute the missing time series from `node_hwmon_sensor_label`.
- All you need is some other time series that has the labels you need, which in this case is `node_hwmon_temp_celsius`.
- `node_hwmon_temp_celsius` does not have the label label, but all the other labels match up so you can ignore this using ignoring:

```
node_hwmon_sensor_label or ignoring(label) (node_hwmon_temp_celsius * 0 + 1)
```

or operator

- **The vector matching produced three groups.**
  - ✓ The first two groups had a sample from node_hwmon_sensor_label so that was what was returned, including the metric name as there was nothing to change it.
  - ✓ For the third group, however, which included sensor="temp1", there was no sample in the group for the left-hand side, so the values in the group from the right-hand side were used.
- **Because arithmetic operators were used on the value, the metric name was removed.**

or operator

- This expression can now be used in the place of `node_hwmon_sensor_label`:

```
node_hwmon_temp_celsius * ignoring(label)
group_left(label) (node_hwmon_sensor_label
or ignoring(label) (node_hwmon_temp_celsius * 0 + 1) )
```

  - ✓ The sample with sensor="temp1" is now present in your result.
  - ✓ It has no label called label, which is the same as saying that that label label has the empty string as a value.

or operator

- In simpler cases you will be working with metrics without any instrumentation labels.
- For example, you might be using the textfile collector, and expecting it to expose a metric called `node_custom_metric`.
- In the event that metric doesn't exist you would like to return 0 instead.
  - ✓ In cases like this, you can use the up metric that is associated with every target:

```
node_custom_metric or up * 0
```

- This has a small problem in that it will return a value even for a failed scrape, which is not how scraped metrics work.
- It will also return results for other jobs. You can fix this with a matcher and some filtering:

```
node_custom_metric or (up{job="node"} == 1) * 0
```

or operator

- Another way you can use the or operator is to return the larger of two series:

```
(a >= b) or b
```

  - ✓If **a** is larger it will be returned by the comparison, and then the `or` operator since the group on the left-hand side was not empty.
  - ✓If on the other hand **b** is larger, then the comparison will return nothing, and or will return **b** as the group on the left-hand side was empty.

unless operator

- The `unless` operator does vector matching in the same way as the `or` operator, working based on whether groups from the right and left operands are empty or have samples.
- The `unless` operator returns the left-hand group, unless the right-hand group has members, in which case it returns no samples for that group.
- You can use `unless` to restrict what time series are returned based on an expression.
- If you wanted to know the average CPU usage of processes except those using less than 100 MB of resident memory you could use the expression:

```
rate(process_cpu_seconds_total[5m]) unless
process_resident_memory_bytes < 100 * 1024 * 1024
```

unless operator

- `unless` can also be used to spot when a metric is missing from a target. For example:

```
up{job="node"} == 1 unless node_custom_metric
```

- would return a sample for every instance that was missing the `node_custom_metric` metric, which you could use in alerting.
- By default, as with all binary operators, `unless` looks at all labels when grouping.
- If `node_custom_metric` had instrumentation labels you could use on or ignoring to check that at least one relevant time series existed without having to know the values of the other labels:

```
up == 1 unless on (job, instance) node_custom_metric
```

- Even if there are multiple samples from the right operand in a group, this is okay as unless uses many-to-many matching.

and operator

- The `and` operator is the opposite of the `unless` operator.
- It returns a group from the left-hand operand only if the matching right-hand group has samples; otherwise, it returns no samples for that match group.
- You can think of it as an `if` operator.
- You will use the `and` operator most commonly in alerting to specify more than one condition.
- To return when both latency is high and there is more than a trickle of user requests.
  - ✓ To do this for Prometheus for handlers that were taking over a second on average and had at least one request per second you could use:

```
(rate(http_request_duration_microseconds_sum{job="prometheus"}[5m])
/ rate(http_request_duration_microseconds_count{job="prometheus"}[5m])
) > 1000000 and rate(http_request_duration_microseconds_count{job="prometheus"}[5m]) > 1
```

- This will return a sample for every individual handler on every prometheus job, so it could get a little spammy even with the one request per second restriction.
  - ✓ Usually you would want to aggregate across a job when alerting.

and operator

- You can use `on` and `ignoring` with the `and` operator, as you can with the other binary operators.
- In particular, `on()` can be used to have a condition that has no common labels at all between the two operands.
- To limit the time of day an expression will return results for:

```
(rate(http_request_duration_microseconds_sum{job="prometheus"}[5m])
/ rate(http_request_duration_microseconds_count{job="prometheus"}[5m])) >
1000000
and rate(http_request_duration_microseconds_count{job="prometheus"}[5m]) > 1
and on() hour() > 9 < 17
```

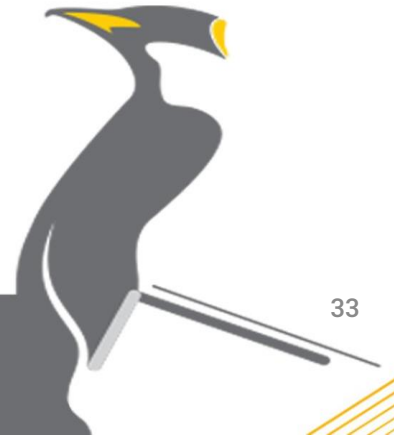
  - ✓ It returns an instant vector with one sample with no labels and the hour of the UTC day of the query evaluation time as the value.

Operator Precedence

- When evaluating an expression with multiple binary operators, PromQL does not simply go from left to right.
- Instead, there is an order of operators that is largely the same as the order used in other languages:
  1. ^
  2. * / %
  3. + -
  4. == != > < >= <=
  5. unless and
  6. or
- For example, a or b * c + d is the same as a or ((b * c) + d).
- All operators except ^ are left-associative.
  - ✓ That means that a / b * c is the same as (a / b) * c, but a ^ b ^ c is a ^ (b ^ c).
- You can use parentheses to change the order of evaluation.

FUNCTIONS

PromQL has 46 functions as of 2.2.1, and offers you a wide variety of functionality, from common math to functions specifically for dealing with counter and histogram metrics. In this chapter you will learn about how all the functions work and how they can be used.



Functions

- Almost all PromQL functions return instant vectors, and the two that don't (`time` and `scalar`) return scalars.
- No functions return range vectors, though multiple functions, including `rate` and `avg_over_time` that you have already seen, take a range vector as input.
- Put another way, functions generally work either across the samples of a single time series at a time or across the samples of an instant vector.
  - ✓ There is no single function or feature of PromQL that you can use to process an entire range vector at once.
- PromQL is statically typed, functions do not change their return value based on the input types.
  - ✓ In fact, the input types for each function are also fixed.
- Where a function needs to work with two different types, two different names are used. For example, you use the avg aggregator on instant vectors and the avg_over_time function on range vectors.

Changing Type

- At times you will have a vector but need a scalar, or vice versa.
- There are two functions that allow you to do so.
  - ✓Vector
  - ✓Scalar

vector

- The `vector` function takes a scalar value, and converts it into an instant vector with one **labelless** sample and the given value.

```
vector(1)
```

- will produce:

```
{} 1
```

- This is useful if you need to ensure an expression returns a result, but can't depend on any particular time series to exist.

```
sum(some_gauge) or vector(0)
```

- will always return one sample, even if some_gauge has no samples.
  - ✓ Depending on the use case, "`bool modifier`" may be a better choice than the `or` operator.

scalar

- The scalar function takes an instant vector with a single sample and converts it to a scalar with the value the input sample had.
  - ✓ If there is not exactly one sample, then **NaN** will be returned to you.
- This is mostly useful when working with scalar constants, but you should use math functions that only work on instant vectors.
- if you wanted the natural logarithm of two as a scalar, rather than typing out 0.693147180559453 and hoping anyone reading it recognized the significance of number, you could use:

```
scalar(ln(vector(2)))
```

- This can also make certain expressions simpler to write.
- if you wanted to see which servers were started in the current year, you could do:

```
year(process_start_time_seconds) == scalar(year())
```

- rather than:

```
year(process_start_time_seconds) == on() group_left year()
```

scalar

- But use of the `scalar` function should be limited because using scalar loses all of your labels and with it your ability to do vector matching.

```
sum(rate(node_cpu_seconds_total{mode!="idle",instance="localhost:9090"}[5m]))
/ scalar(count(node_cpu_seconds_total{mode="idle",instance="localhost:9090"))
```

- will give you the proportion of time a machine's CPU is not idle, but you would then have to alter and reevaluate this expression for every single instance.

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

- and calculate the proportion of nonidle CPU for all your machines at once.

Math

- The math functions perform standard mathematical operations on instant vectors, such as calculating absolute values or taking a logarithm.
- Each sample in the instant vector is handled independently, and the metric name is removed in the return value.

abs

- `abs` takes an instant vector and returns the absolute value for each of its values, which is to say any negative numbers are changed to positive numbers.
- The expression:

```
abs(process_open_fds - 15)
```

- will return how far away each process's open file descriptors count is from 15.
- Counts of 5 and 25 would both return 10.

ln, log2, and log10

- The functions `ln`, `log2`, and `log10` take an instant vector and return the logarithm of the values and use different bases for the logarithm, Euler's number e, 2, and 10, respectively.
- `ln` is also known as the natural logarithm.
- These functions can be used to get an idea of the different orders of magnitude of numbers.
- For example, to calculate the number of 9s of successes an API endpoint had over the past hour, you could do:

```
log10(
1 - (
sum without(instance)(rate(requests_failed_total[1h]))
/
sum without(instance)(rate(requests_total[1h]))
)
) * -1
```

- These can also be useful for graphing in certain circumstances where normal linear graphs can’t suitably represent a large variance in values.
- However, it is usually best to rely on the in-built logarithm graphing options in tools such as Grafana rather than using these functions, as they tend to gracefully handle edge cases such as negative logarithms returning NaN

- If you want a logarithm to a different base, you can use the change of base formula.
- For example, for a logarithm base three on the instant vector x, you would use:

```
ln(x) / ln(3)
```

exp

- The `exp` function provides the natural exponent, and is the inverse to the ln function.

```
exp(vector(1))
```

- returns:

```
{} 2.718281828459045
```

  - ✓ which is Euler's number, e.

sqrt

- The `sqrt` function returns a square root of the values in an instant vector.

```
sqrt(vector(9))
```

- will return:

```
{} 3
```

- sqrt predates the exponent operator ^, so this is equivalent to:

```
vector(9) ^ 0.5
```

- The cube or third root can be calculated with:

```
vector(9) ^ (1/3)
```

ceil and floor

- `ceil` and `floor` allow you to round the values in an instant vector.
- `ceil` always rounds up to the nearest integer, and `floor` always rounds down.

```
ceil(vector(0.1))
```

- will return:

```
{} 1
```

round

- `round` rounds the values in an instant vector to the nearest integer.
  - ✓ If you provide a value that is exactly halfway between two integers, it is rounded up.

```
round(vector(5.5))
```

```
{} 6
```

- The additional argument is a scalar, and the values will be rounded to the nearest multiple of this number:

```
round(vector(2446), 1000)
```

```
{} 2000
```

- This is equivalent to:

```
round(vector(2446) / 1000) * 1000
```

clamp_max and clamp_min

- Sometimes you will find that a metric returns spurious values well outside the normal range, such as a gauge that you expect to be positive occasionally being massively negative.
- `clamp_max` and `clamp_min` allow you to put upper and lower bounds, respectively, on the values in an instant vector.
- For example, if you didn't believe that your processes could have fewer than 10 open file descriptors you could use:

```
clamp_min(process_open_fds, 10)
```

Time and Date

- Prometheus offers you several functions dealing with time, most of which are convenience functions around time to save you from having to implement date-related logic yourself.
- Prometheus works entirely in UTC, and has no notion of time zones.

time

- The `time` function is the most basic time-related function.
- It returns the evaluation time of the query as seconds since the Unix epoch2 as a scalar.

```
time()
```

- If you were to use time with the query_range endpoint, then every result would be different, as each step has a different evaluation time.
- The Prometheus best practice is to expose the Unix time in seconds at which something of interest happened, and not how long it has been since it happened.
- This is more reliable, as it's not susceptible to failure to update the metric.
- The time function then lets you convert these to durations.
- To see how long your processes have been running you would use:

```
time() - process_start_time_seconds
```

- If you had a batch job pushing the last time it succeeded to the Pushgateway, you could find jobs that hadn't succeeded in the past hour with:

```
time() - my_job_last_success_seconds > 3600
```

minute, hour, day_of_week, day_of_month, days_in_month, month, and year

- time covers most use cases, but sometimes you will want to have logic based on the clock or calendar.
- Converting to minutes and hours from time isn't too difficult, but beyond that you have to consider issues like **leap days**.
- All of these functions return the given value for the query evaluation time as an instant vector with one sample and no labels.

minute, hour, day_of_week, day_of_month, days_in_month, month, and year

- `day_of_week` starts with 0 for Sunday.
- If you wanted to check if today was the last day of the month you could compare the output of `day_of_month` to `days_in_month`.
- Why these functions don't return scalars, as that'd seem more convenient to work with?
- The answer is that these functions all take an optional argument so that you can pass in instant vectorsTo see what year your processes started in you could use:

```
year(process_start_time_seconds)
```

- Count how many processes were started this month:

```
sum((year(process_start_time_seconds) == bool scalar(year()))
*
(month(process_start_time_seconds) == bool scalar(month())))
```

timestamp

- The `timestamp` function is different from the other time functions in that it looks at the timestamp of the samples in an instant vector rather than the values.
- The timestamps for samples returned from all operators, functions, the query_range HTTP API, and query HTTP API when it returns an instant vector will be the query evaluation time.
- However, the timestamp of samples in an instant vector from an instant vector selector will be the actual timestamps.
- The timestamp function allows you to access these.
- To see when the last scrape started for each target with:

```
timestamp(up)
```

- This is because the default timestamp for data from a scrape is the time that the scrape started.

timestamp

- If you want to see raw data with samples for debugging, using a range vector selector with the query HTTP API is best, but timestamp does have some uses.

```
node_time_seconds - timestamp(node_time_seconds)
```

- would return the difference between when the scrape of the Node exporter was started by Prometheus and what time the Node exporter thought was the current time.
- While this isn't 100% accurate (it will vary with machine load), it will allow you to know if time is out of sync by a few seconds without needing a 1-second scrape interval.

Labels

- In an ideal world the label names and label values used by different parts of your system would be consistent, so you wouldn't have customer in one place and cust in another.
- While it is best to resolve such inconsistencies in the source code, or failing that with `metric_relabel_configs`, this is not always possible.
- Thus the two label functions allow you to change labels.

label_replace

- `label_replace` allows you to do regular expression substitution on label values.
- For example, if you needed the device label on `node_disk_read_bytes_total` to be dev instead for vector matching to work as you needed you could do:

```
label_replace(node_disk_read_bytes_total, "dev", "${1}", "device", "(.*)")
```

- which would return a result like:

```
node_disk_read_bytes_total{dev="sda",device="sda",instance="localhost:9100", job="node"}
4766305792
```

- Unlike most functions, `label_replace` does not remove the metric name, as it is presumed that you are doing something unusual if you have to resort to `label_replace`, and removing the metric name could make that harder for you.
- The arguments to `label_replace` are the instant vector input, the name of the output label, the replacement, the name of the source label, and the regular expression.
- `label_replace` is similar to the replace relabelling action, but you can only use one label as a source label.
- If the regular expression does not match for a given sample, then that sample is returned unchanged.

label_join

- `label_join` allows you to join label values together, similarly to how `source_labels` is handled in relabelling.
- If you wanted to join the job and instance labels together into a new label you could do:

```
label_join(node_disk_read_bytes_total, "combined", "-", "instance", "job")
node_disk_read_bytes_total{combined="localhost:9100-node",device="sda",
instance="localhost:9100",job="node"} 4766359040
```

- As with `label_replace`, `label_join` does not remove the metric name.
  - ✓ The arguments are the **instant vector input**, the **name of the output label**, the **separator**, and then **zero or more label names**.
- You could combine `label_join` with `label_replace` to provide the full functionality of the replace relabel action, but at that point you should seriously consider `metric_relabel_configs` or fixing the source metrics instead.

Missing Series and absent

- As mentioned, the `absent` function plays the role of a not operator.
  - ✓ If you pass a nonempty instant vector, it returns an empty instant vector.
  - ✓ If you pass an empty instant vector, it returns an instant vector with one sample and a value of 1.
- This sample has no labels, since there are no labels to work with.
- However, `absent` is a little smarter than that, and if the argument is an instant vector selector, it uses the labels from any equality matchers present.

Missing Series and absent

Expression	Result
absent(up)	empty instant vector
absent(up{job="prometheus"})	empty instant vector
absent(up{job="missing"})	{job="missing"} 1
absent(up{job=~"missing"})	{} 1
absent(non_existent)	{} 1
absent(non_existent{job="foo”,env="dev"})	{job="foo”,env="dev"} 1
absent(non_existent{job="foo”,env="dev"} * 0)	{} 1

- **`absent` is useful for detecting if an entire job has gone missing from service discovery.**
  - ✓ Alerting on up == 0 doesn't work too well when you have no targets to produce up metrics!
- **Even when using `static_configs` it can be wise to have such an alert in case generation of your `prometheus.yml` goes awry.**
  - ✓ If you want instead to alert on specific metrics that are missing from a target, you can use `unless`.

Sorting with sort and sort_desc

- PromQL generally does not specify the order of elements within an instant vector, so it can change from evaluation to evaluation.
- But if you use `sort` or `sort_desc` as the last thing that is evaluated in a PromQL expression, then the instant vector will be sorted by value.

```
sort(node_filesystem_size_bytes)
```

- The effect of these functions is cosmetic, but may save you some effort in reporting scripts.
- `NaN`s are always sorted to the end, so sort and `sort_desc` are not quite the reverse of each other.

Histograms with histogram_quantile

- The `histogram_quantile` is internally a bit like an aggregator, since it groups samples together like a `without(le)` clause would and then calculates a quantile from their values.

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

  - ✓ Would calculate the 0.9 quantile (also known as the 90th percentile) latency of Prometheus's compaction latency over the past day.

- Values outside of the range from zero to one do not make sense for quantiles, and will result in infinities.
- You must always use `rate` first for buckets exposed by Prometheus's histogram metric type, as `histogram_quantile` needs gauges to work on.

Counters

- Counters include not just the counter metric, but also the `_sum`, `_count`, and `_bucket` time series from summary and histogram metrics.
  - ✓ **Counters can only go up.**
- When an application starts or restarts, counters will initialize to 0, and the counter functions take this into account automatically.
- The values of counters are not particularly useful on their own;
  - ✓ you will almost always want to **convert them to gauges** using one of the counter-related functions.
- Functions working on counters all take a range vector as an argument and return an instant vector.
  - ✓ Each of the time series in the range vector is processed individually, and returns at most one sample.
- If there is only one sample for one of your time series within the range you provide, you will get no output for it when using these functions.

rate

- The `rate` function is the primary function you will use with counters, and indeed likely the main function you will use from PromQL.
- `rate` returns how fast a counter is increasing per second for each time series in the range vector passed to it.

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

- `rate` automatically handles counter resets, and any decrease in a counter is considered to be a counter reset.
  - ✓ If you had a time series that had values [5,10,4,6], it would be treated as though it was [5,10,14,16].
- `rate` presumes that the targets it is monitoring are relatively long-lived compared to a scrape interval, as it cannot detect multiple resets in a short period of time.
  - ✓ If you have targets that are expected to regularly live for less than a handful of scrape intervals you may wish to consider a log-based monitoring solution instead.
- `rate` has to handle scenarios like time series appearing and disappearing, such as if one of your instances started up and then later crashed.
  - ✓ For example, if one of your instances had a counter that was incrementing at a rate of around 10 per second, but was only running for half an hour, then a `rate(x_total[1h])` would return a result of around 5 per second.

rate

- Since scrapes for different targets happen at different times, there can be jitter over time, the steps of a `query_range` call will rarely align perfectly with scrapes, and scrapes are expected to fail every now and then.
- `rate` is not intended to catch every single increment, as it is expected that increments will be lost, such as if an instance dies between scrapes.
- This may cause artifacts if you have very slow moving counters, such as if they're only incremented a few times an hour.
- rate can also only deal with changes in counters, because if a counter time series appears with a value of 100, rate has no idea if those increments were just now or if the target has been running for years and has only just started being returned by service discovery to be scraped.
- It is recommended to use a range for your range vector that is at least four times your scrape interval. This will ensure that you always have two samples to work with even if scrapes are slow, ingestion is slow, and there has been a single scrape failure. Such issues are a fact of life in real-world systems, so it is important to be resilient. For example, for a 1-minute scrape interval you might use a 4-minute rate, but usually that is rounded up to a 5-minute rate.
- Generally you should aim to have the same range used on all your rate functions within a Prometheus for the sake of sanity, since outputs from rates over different ranges are not comparable and tend to be hard to keep track of.
- You may wonder with all these implementation details and caveats if rate could be changed to be simpler. There are several ways you can approach this problem, but at the end of the day they all have both advantages and disadvantages. If you fix one apparent problem, you will cause a different problem to pop up. The rate function is a good balance across all of these concerns, and provides a robust solution suitable for operational monitoring. If you run into a situation where any rate-like function isn't giving you quite what you need, I would suggest continuing your debugging based on logs data, which does not have these particular concerns and can produce exact answers.

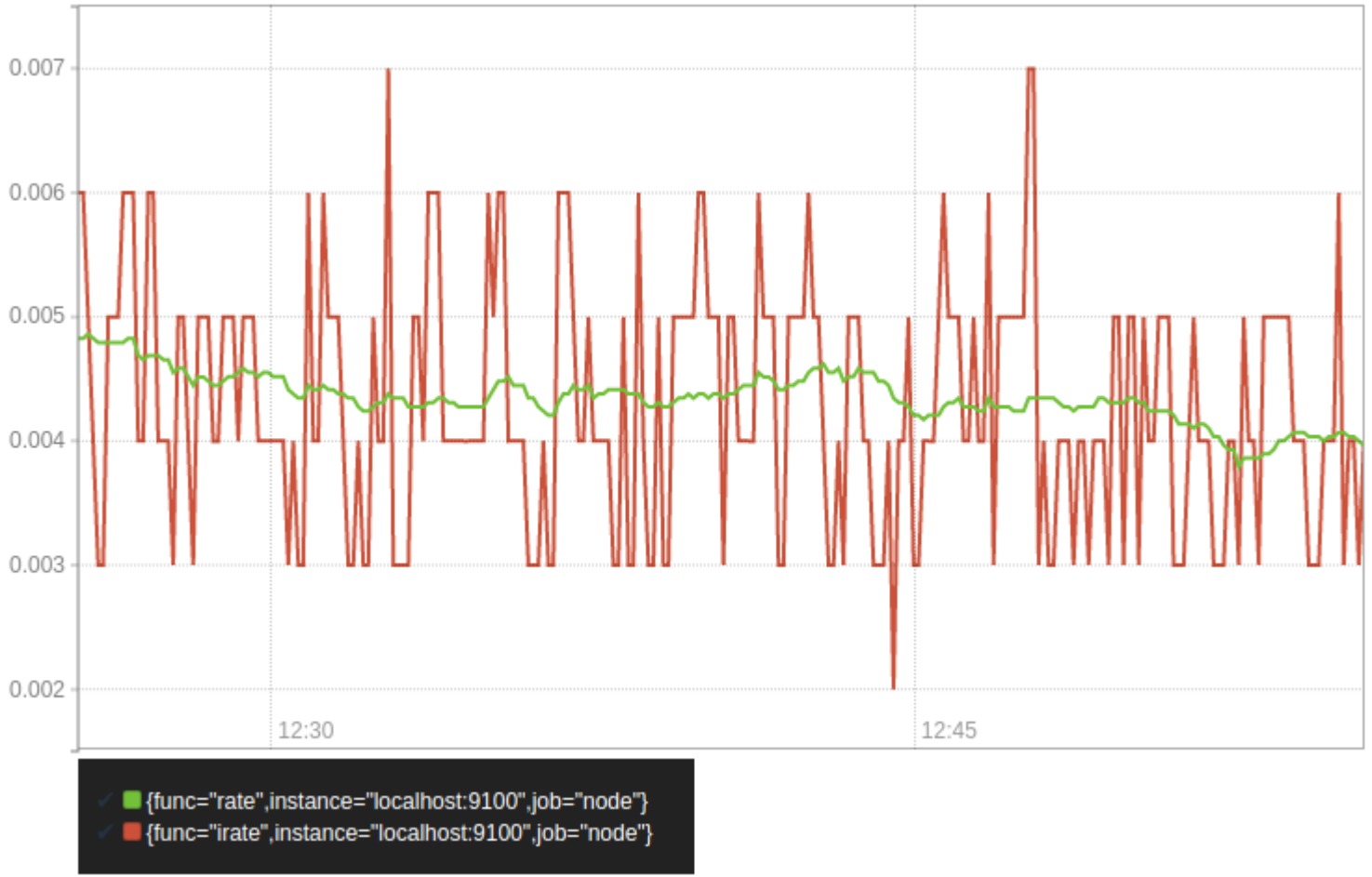
increase

- `increase` is merely syntactic sugar on top of `rate`.
- `increase(x_total[5m])` is exactly equivalent to `rate(x_total[5m]) * 300`, which is to say the result of `rate` multiplied by the range of the range vector.
  - ✓ The logic is otherwise identical.
- Seconds are the base unit for Prometheus, so you should use `increase` only when displaying values to humans.
- Within your recording rules and alerts it is best to stick to rate for consistency.
- One of the outcomes of the robustness of `rate` and `increase` is that they can return noninteger results when given integer inputs.
- Consider that you had the following data points for a time series:
- 21@2, 22@7, 24@12
  - ✓ And you were to calculate increase(x_total[15s]) with a query time of 15 seconds, The increase here is 3 over a period of 10 seconds, so you might expect a result of 3.
- However, the rate was taken over a 15-second period, so to avoid underestimating the correct answer, the 10 seconds of data you have is extrapolated out to 15 seconds, producing a result of 4.5 for the increase.
- rate and increase presume that a time series continues beyond the bound of the range if the first/last samples is within 110% of the average interval of the data. If this is not the case, it is presumed the time series exists for 50% of an interval beyond the samples you have, but not with the value going below zero.

irate

- `irate` is like `rate` in that it returns the per-second rate at which a counter is increasing.
- The algorithm it uses is much simpler though;
  - ✓ it only looks at the **last two samples** of the range vector it is passed.
- This has the advantage that it is much more responsive to changes and you don't have to care so much about the relationship between the vector's range and the scrape interval, but comes with the corresponding disadvantage that as it is only looking at two samples, it can only be safely used in graphs that are fully zoomed in.
  - ✓ Due to the lack of averaging that irate brings, the graphs can be more volatile and harder to read.
- It is not advisable to use irate in alerts due to it being sensitive to brief spikes and dips; use `rate` instead.

irate



resets

- You may sometimes suspect that a counter is resetting more often than it should be.
- The `resets` function returns how many times each time series in a range vector has reset.

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

  - ✓ Will indicate how many times the CPU time of the process has reset in the past hour.
- This should be the number of times the process has restarted, but if you had a bug that was causing it to go backwards, the value would be higher.
- `resets` is intended as a debugging tool for counters, since counters might reset too often and nonmonotonic counters will cause artifacts in the form of large spikes in your graphs.
- However, some users have found occasional uses for it when they want to know how many times a gauge has been seen to decrease.

Changing Gauges

- Unlike counters, the values of gauges are useful on their own and you can use binary operators and aggregators directly on them.
  - ✓ But sometimes you will want to analyze the history of a gauge, and there are several functions for this purpose.
- As with the counter functions, these functions also take a range vector and return an instant vector with at most one sample for each time series in your input.

changes

- Some gauges are expected to change very rarely.
- For example, the start time of a process does not change in the lifetime of a process.
- The `changes` function allows you to count how many times a gauge has changed value, so `changes(process_start_time_seconds[1h])` will tell you how many times your process has restarted in the past hour.
  - ✓ If you aggregated this across entire applications it would allow you to spot if your applications were in a slow crash loop.
    - Due to the fundamental nature of metrics sampling, Prometheus may not scrape often enough to see every possible change.
  - ✓ However, if a process is restarting that frequently, you will still detect it either via this method or by up being 0.
- You can use `changes` beyond `process_start_time_seconds` for other situations where the fact that a gauge has changed is interesting to you.

deriv

- Often you will want to know how quickly a gauge is changing.
  - ✓ For example, how quickly a backlog is increasing if it is increasing at all.
- This would allow you to alert on not only the backlog being higher than you'd like but also that it has not already started to go down.
- You could do `x - x offset 1h`, but this only uses two samples, and thus lacks robustness because it is susceptible to individual outlier values.
- The `deriv` function uses least-squares regression to estimate the slope of each of the time series in a range vector.

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

- would calculate how fast resident memory is changing per second based on samples from the past hour.

predict_linear

- `predict_linear` goes a step further than `deriv` and predicts what the value of a gauge will be in the future based on data in the provided range.

```
predict_linear(node_filesystem_free_bytes{job="node"}[1h], 4 * 3600)
```

- Would predict how much free space would be left on each filesystem in four hours based on the past hour of samples.
- This expression is roughly equivalent to:

```
deriv(node_filesystem_free_bytes{job="node"}[1h]) * 4 * 3600
+ node_filesystem_free_bytes{job="node"}[1h]
```

- But `predict_linear` is slightly more accurate because it uses the intercept from the regression.

predict_linear

- `predict_linear` is useful for resource limit alerts, where static thresholds such as 1 GB free or percentage thresholds such as 10% free tend to have false positives and false negatives depending on whether you are working with relatively large or small filesystems.
- A 1 GB threshold on a 1 TB filesystem would alert you too late, but would also alert you too early on a 2 GB filesystem.
  - ✓ `predict_linear` works better across **all sizes**.
- It can take some tweaking to choose good values for the range and to determine how far to predict forward.
- If there was a regular sawtooth pattern in the data you would want to ensure that the range was long enough not to extrapolate the upward part of the cycle out indefinitely.

delta

- `delta` is similar to `increase`, but without the counter reset handling.
- This function should be avoided as it can be overly affected by single outlier values.
- You should use `deriv` instead, or x - x offset 1h if you really want to compare with the value a given time ago.

idelta

- `idelta` takes the last two samples in a range and returns their difference.
- `idelta` is intended for advanced use cases.
- For example, the way rate and irate work is not to everyone's personal tastes, so using `idelta` and recording rules allows users to implement what they'd like without polluting PromQL with various subtle variations of the rate function.

holt_winters

- The `holt_winters` function implements Holt-Winters double exponential smoothing.
- Gauges can at times be very spiky and hard to read so some smoothing is often good.
- At the simplest you could use `avg_over_time`, but you might want something more sophisticated.
- This function works through the samples for a time series and tracks the smoothed value so far and provides an estimate of the trend in the data.
- Each new sample is taken into account based on the smoothing factor, which indicates how much old data is important relative to new data, and the trend factor, which controls how important the trend is.

```
holt_winters(process_resident_memory_bytes[1h], 0.1, 0.5)
```

- would smooth memory usage with a smoothing factor of 0.1 and a trend factor of 0.5.
- Both factors must be between 0 and 1.

Aggregation Over Time

- Aggregators such as `avg` work across the samples in an instant vector.
- There is also a set of functions such as `avg_over_time` that apply the same logic, but across the values of a time series in a range vector.
  - ✓ `sum_over_time`
  - ✓ `count_over_time`
  - ✓ `avg_over_time`
  - ✓ `stddev_over_time`
  - ✓ `stdvar_over_time`
  - ✓ `min_over_time`
  - ✓ `max_over_time`
  - ✓ `quantile_over_time`
- To see the peak memory usage that Prometheus saw for a process you could use:

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

- And even go a step further and calculate that across the application:

```
max without(instance)(max_over_time(process_resident_memory_bytes[1h]))
```

Aggregation Over Time

- These functions only work from the values of the samples; there is no weighting based on the length of time between samples or any other logic relating to timestamps.
  - ✓ This means that if you change the scrape interval, for example, there will be a bias toward the time period with the more frequent scrapes for functions such as `avg_over_time` and `quantile_over_time`.
  - ✓ Similarly if there are failed scrapes for a period of time, that period will be less represented in your result.
- These functions are used with gauges.
- If you want to take an `avg_over_time` of a rate this isn't possible as that function returns instant rather than range vectors.
- However, rate already calculates an average over time, so you can increase the range on the rate.
- Instead of trying to do:

```
avg_over_time(rate(x_total[5m])[1h])
```

- Which will produce a parse error, you can instead do:

```
rate(x_total[1h])
```