DevOpsSchool

Lets Learn, Share & Practice DevOps

Prometheus Query

PromQL

- Non-SQL Query Language
- Better for metrics computation
- Only does reads

Non-SQL Query Language

PromQL: `rate(api_http_requests_total[5m])`

SQL: `SELECT job, instance, method, status, path, rate(value, 5m) FROM api_http_requests_total`

PromQL: `avg by(city) (temperature_celsius{country="germany"})`

SQL: `SELECT city, AVG(value) FROM temperature_celsius WHERE country="germany" GROUP BY city`

PromQL: `rate(errors{job="foo"}[5m]) / rate(total{job="foo"}[5m])`

SQL:

```
SELECT errors.job, errors.instance, [...more labels...], rate(errors.value, 5m) /
rate(total.value, 5m) FROM errors JOIN total ON [...all the label equalities...] WHERE
errors.job="foo" AND total.job="foo"
```

Prometheus provides a functional expression language called PromQL

- Provides built in operators and functions
- Vector-based calculation like Excel
- Expressions over time-series vectors
- PromQL is read-only
- Example:

```
100 - (avg by (instance) (irate(node_cpu_seconds_total{job='node_exporter',mode="idle"}[5m])) * 100)
```

Resolution

- A range query's **query step(aka Resolution**) is completely independent from any range durations specified in the PromQL expression it evaluates.
- If you have 'rate(http_requests_total[5m])', you can evaluate this at a **query step(aka Resolution**) of 15 seconds and Prometheus doesn't care either way.
- What happens is that every 15 seconds, you look back 5 minutes and take the rate between **then** and **now**.

Resolution of the graph

- The resolution there is the resolution of the graph, which is **automatically** determined by the width of the graph and the **time period** it covers. It bears **no relation** to the **scrape interval**.
- If you want to make it lower, either zoom out or set it manually in the res textbox.

Operators

Prometheus supports many binary and aggregation operators.

Binary operators

- Arithmetic binary operators
- Comparison binary operators
- Logical/set binary operators

Vector matching

- One-to-one vector matches
- Many-to-one and one-to-many vector matches

Aggregation operators

Operators

- **Arithmetic binary operators**
  Example: - (subtraction), * (multiplication), / (division), % (modulo), ^ (power/exponentiation)

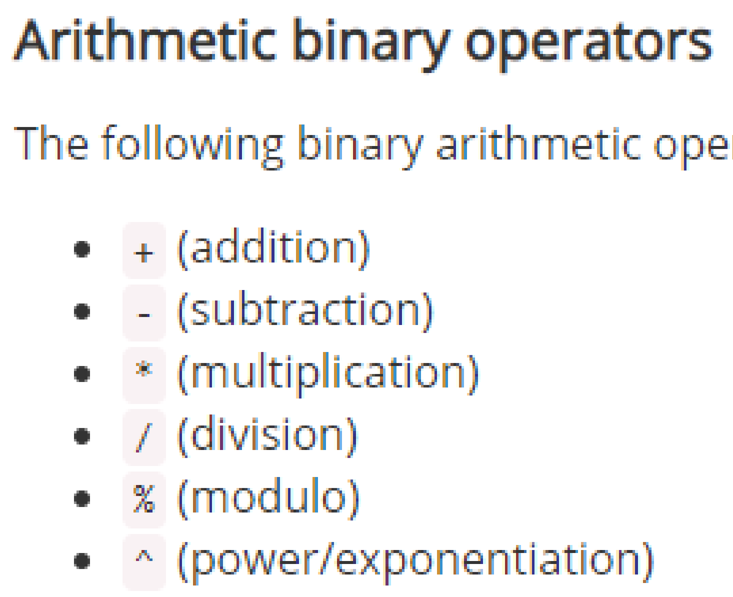
- **Comparison binary operators**
  Example: == (equal), != (not-equal), > (greater-than), < (less-than) ,>= (greater-or-equal), <= (less-or-equal)

- **Logical/set binary operators**
  Example: and (intersection), or (union), unless (complement)

- **Aggregation operators**
  Example:**sum** (calculate sum over dimensions), **min** (select minimum over dimensions) ,**max** (select maximum over dimensions), **avg** (calculate the average over dimensions), **stddev** (calculate population standard deviation over dimensions), **stdvar** (calculate population standard variance over dimensions), **count** (count number of elements in the vector), **count_values** (count number of elements with the same value), **bottomk** (smallest k elements by sample value), **topk** (largest k elements by sample value), **quantile** (calculate ϕ-quantile (0 ≤ ϕ ≤ 1) over dimensions)

Operators

Prometheus supports many binary and aggregation operators.



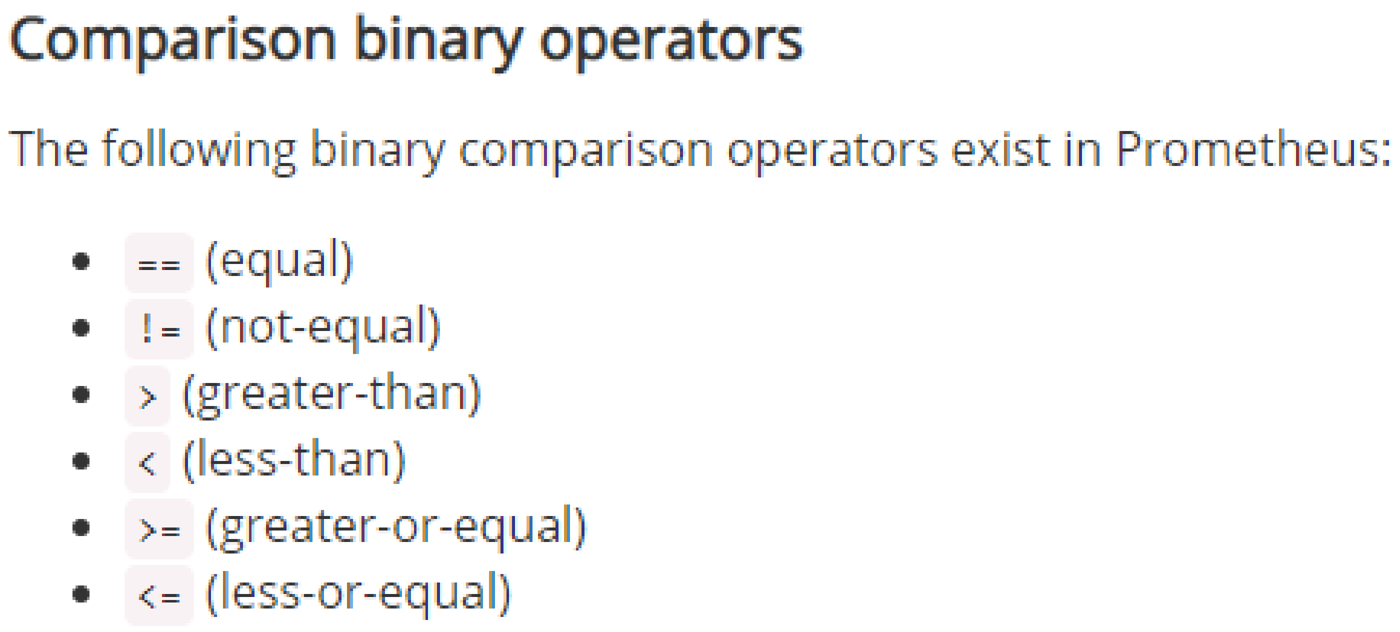
Arithmetic binary operators

The following binary arithmetic operators exist in Prometheus:

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

Operators

Prometheus supports many binary and aggregation operators.



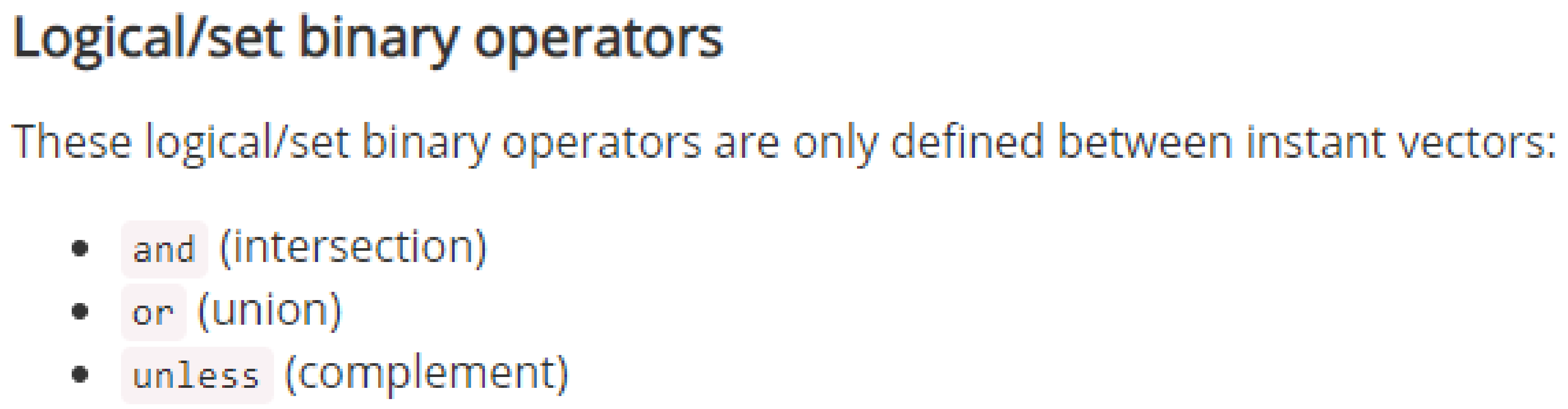
Comparison binary operators

The following binary comparison operators exist in Prometheus:

- `==` (equal)
- `!=` (not-equal)
- `>` (greater-than)
- `<` (less-than)
- `>=` (greater-or-equal)
- `<=` (less-or-equal)

Operators

Prometheus supports many binary and aggregation operators.



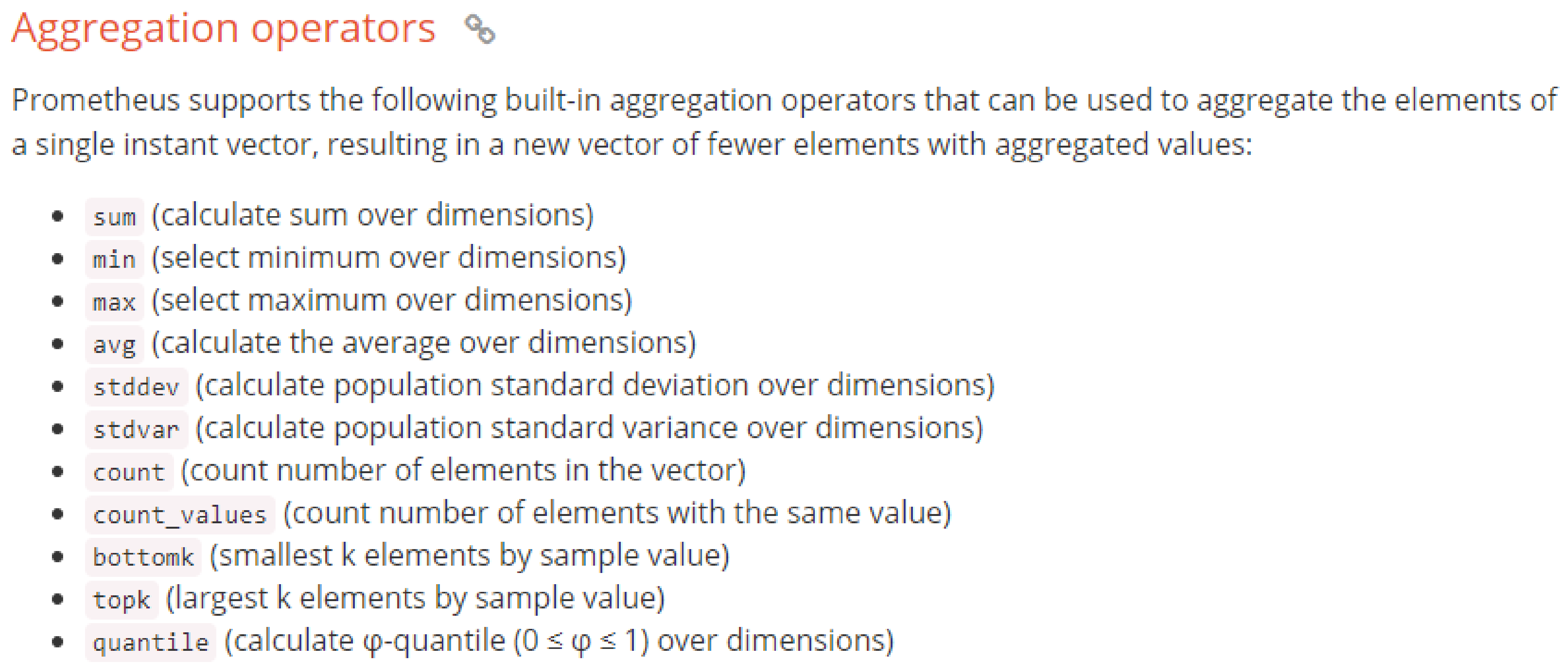
Logical/set binary operators

These logical/set binary operators are only defined between instant vectors:

- `and` (intersection)
- `or` (union)
- `unless` (complement)

Operators

Prometheus supports many binary and aggregation operators.



Aggregation operators

Prometheus supports the following built-in aggregation operators that can be used to aggregate the elements of a single instant vector, resulting in a new vector of fewer elements with aggregated values:

- `sum` (calculate sum over dimensions)
- `min` (select minimum over dimensions)
- `max` (select maximum over dimensions)
- `avg` (calculate the average over dimensions)
- `stddev` (calculate population standard deviation over dimensions)
- `stdvar` (calculate population standard variance over dimensions)
- `count` (count number of elements in the vector)
- `count_values` (count number of elements with the same value)
- `bottomk` (smallest k elements by sample value)
- `topk` (largest k elements by sample value)
- `quantile` (calculate φ-quantile (0 ≤ φ ≤ 1) over dimensions)

Operators

```
sum without (instance) (http_requests_total)
```

Which is equivalent to:

```
 sum by (application, group) (http_requests_total)
```

If we are just interested in the total of HTTP requests we have seen in **all** applications, we could simply write:

```
sum(http_requests_total)
```

To count the number of binaries running each build version we could write:

```
count_values("version", build_version)
```

To get the 5 largest HTTP requests counts across all instances we could write:

```
topk(5, http_requests_total)
```

Operators

Binary operator precedence

The following list shows the precedence of binary operators in Prometheus, from highest to lowest.

1. `^`
2. `*`, `/`, `%`
3. `+`, `-`
4. `==`, `!=`, `<=`, `<`, `>=`, `>`
5. `and`, `unless`
6. `or`

Operators on the same precedence level are left-associative. For example, `2 * 3 % 2` is equivalent to `(2 * 3) % 2`. However `^` is right associative, so `2 ^ 3 ^ 2` is equivalent to `2 ^ (3 ^ 2)`.

Functions

abs()
absent()
absent_over_time()
ceil()
changes()
clamp_max()
clamp_min()
day_of_month()
day_of_week()
days_in_month()
delta()
deriv()
exp()
floor()
histogram_quantile()

holt_winters()
hour()
idelta()
increase()
irate()
label_join()
label_replace()
ln()
log2()
log10()
minute()
month()
predict_linear()
rate()
resets()

round()
scalar()
sort()
sort_desc()
sqrt()
time()
timestamp()
vector()
year()
<aggregation>_over_time()

DevOpsSchool
Lets Learn, Share & Practice DevOps

PromQL - Operators

+	(addition)	==	(equal)
-	(substraction)	!=	(not-equal)
*	(multiplication)	>	(greater-than)
/	(division)	<	(less-than)
%	(modulo)	>=	(greater-or-equal)
^	(exponentiation)	<=	(less-or-equal)
and	(intersection)	or	(union)
unless	(complement)		

... and vector matching

DevOpsSchool
Lets Learn, Share & Practice DevOps

PromQL - Aggregation Operators

```
sum          stddev          bottomk
min          stdvar          topk
max          count           quantile
avg          count_values
```

PromQL - Examples

```
rate(api_http_requests_total[5m])

errors{job="foo"} / total{job="foo"}
```

Using functions, operators, etc.

Return the per-second rate for all time series with the `http_requests_total` metric name, as measured over the last 5 minutes:

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

Assuming that the `http_requests_total` time series all have the labels `job` (fanout by job name) and `instance` (fanout by instance of the job), we might want to sum over the rate of all instances, so we get fewer output time series, but still preserve the `job` dimension:

```
sum by (job) (
  rate(http_requests_total[5m])
)
```

If we have two different metrics with the same dimensional labels, we can apply binary operators to them and elements on both sides with the same label set will get matched and propagated to the output. For example, this expression returns the unused memory in MiB for every instance (on a fictional cluster scheduler exposing these metrics about the instances it runs):

If we have two different metrics with the same dimensional labels, we can apply binary operators to them and elements on both sides with the same label set will get matched and propagated to the output. For example, this expression returns the unused memory in MiB for every instance (on a fictional cluster scheduler exposing these metrics about the instances it runs):

```
(instance_memory_limit_bytes - instance_memory_usage_bytes) / 1024 / 1024
```

The same expression, but summed by application, could be written like this:

```
sum by (app, proc) (
  instance_memory_limit_bytes - instance_memory_usage_bytes
) / 1024 / 1024
```

If the same fictional cluster scheduler exposed CPU usage metrics like the following for every instance:

```
instance_cpu_time_ns{app="lion", proc="web", rev="34d0f99", env="prod", job="cluster-manager"}
instance_cpu_time_ns{app="elephant", proc="worker", rev="34d0f99", env="prod", job="cluster-manager"}
instance_cpu_time_ns{app="turtle", proc="api", rev="4d3a513", env="prod", job="cluster-manager"}
instance_cpu_time_ns{app="fox", proc="widget", rev="4d3a513", env="prod", job="cluster-manager"}
...
```

...we could get the top 3 CPU users grouped by application (`app`) and process type (`proc`) like this:

```
topk(3, sum by (app, proc) (rate(instance_cpu_time_ns[5m])))
```

Assuming this metric contains one time series per running instance, you could count the number of running instances per application like this:

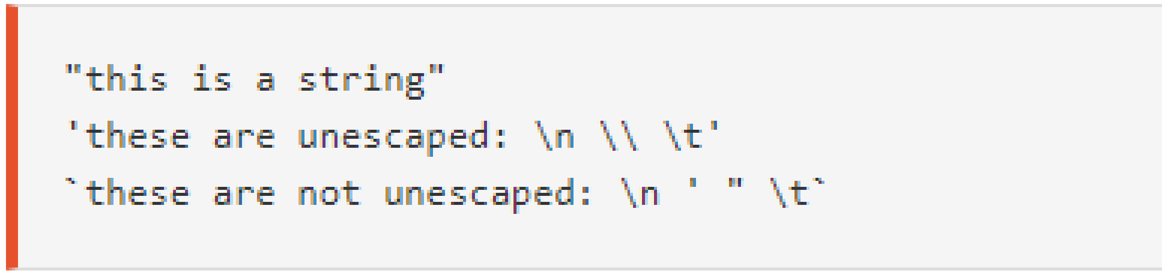
```
count by (app) (instance_cpu_time_ns)
```

String literals

Strings may be specified as literals in single quotes, double quotes or backticks.

In single or double quotes a backslash begins an escape sequence, which may be followed by a, b, f, n, r, t, v or \.

Example:



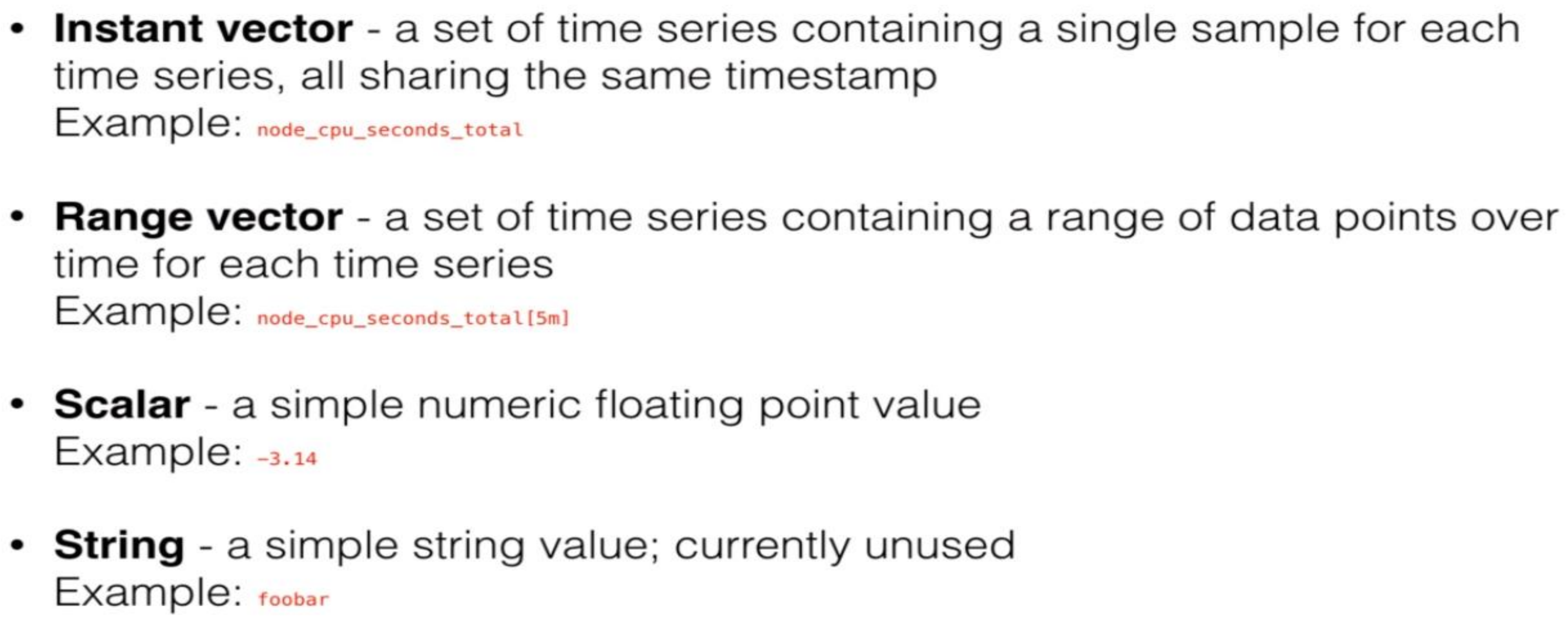
```
"this is a string"
'these are unescaped: \n \\ \t'
`these are not unescaped: \n ' " \t`
```

Float literals

Scalar float values can be literally written as numbers of the form [-](digits)[.(digits)].

```
-2.43
```

An expression or sub-expression can evaluate to one of four types



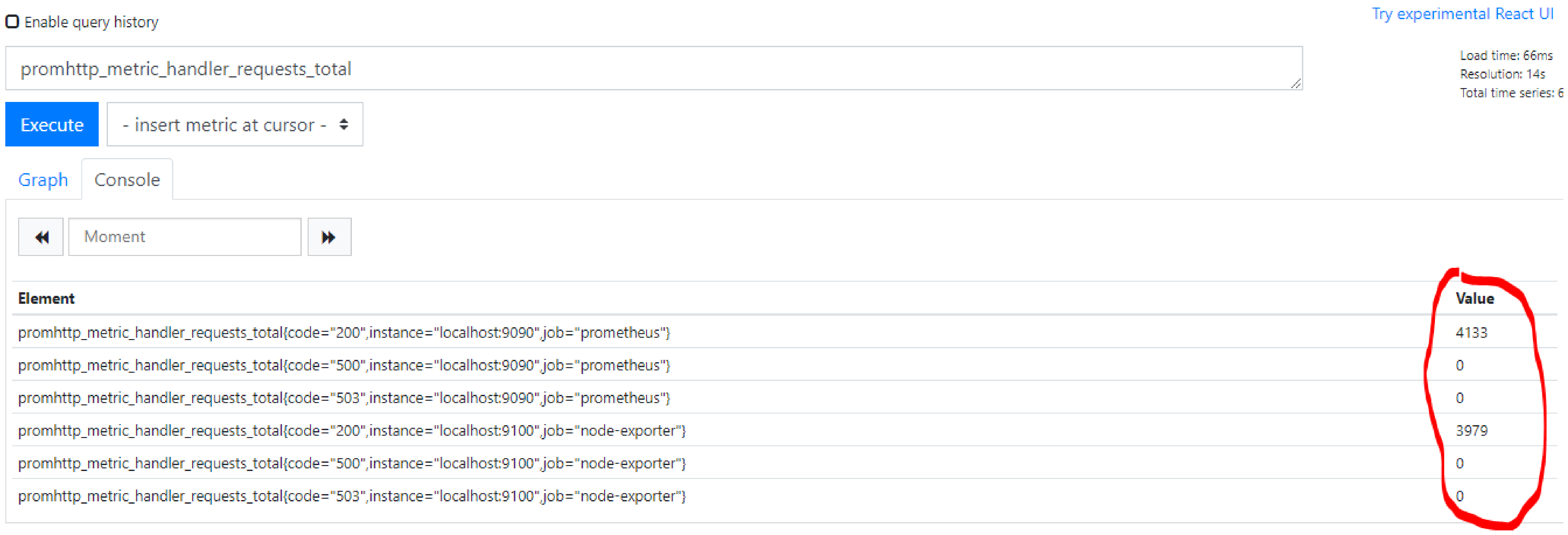
- **Instant vector** - a set of time series containing a single sample for each time series, all sharing the same timestamp
  Example: `node_cpu_seconds_total`

- **Range vector** - a set of time series containing a range of data points over time for each time series
  Example: `node_cpu_seconds_total[5m]`

- **Scalar** - a simple numeric floating point value
  Example: `-3.14`

- **String** - a simple string value; currently unused
  Example: `foobar`

Instant vector Selectors



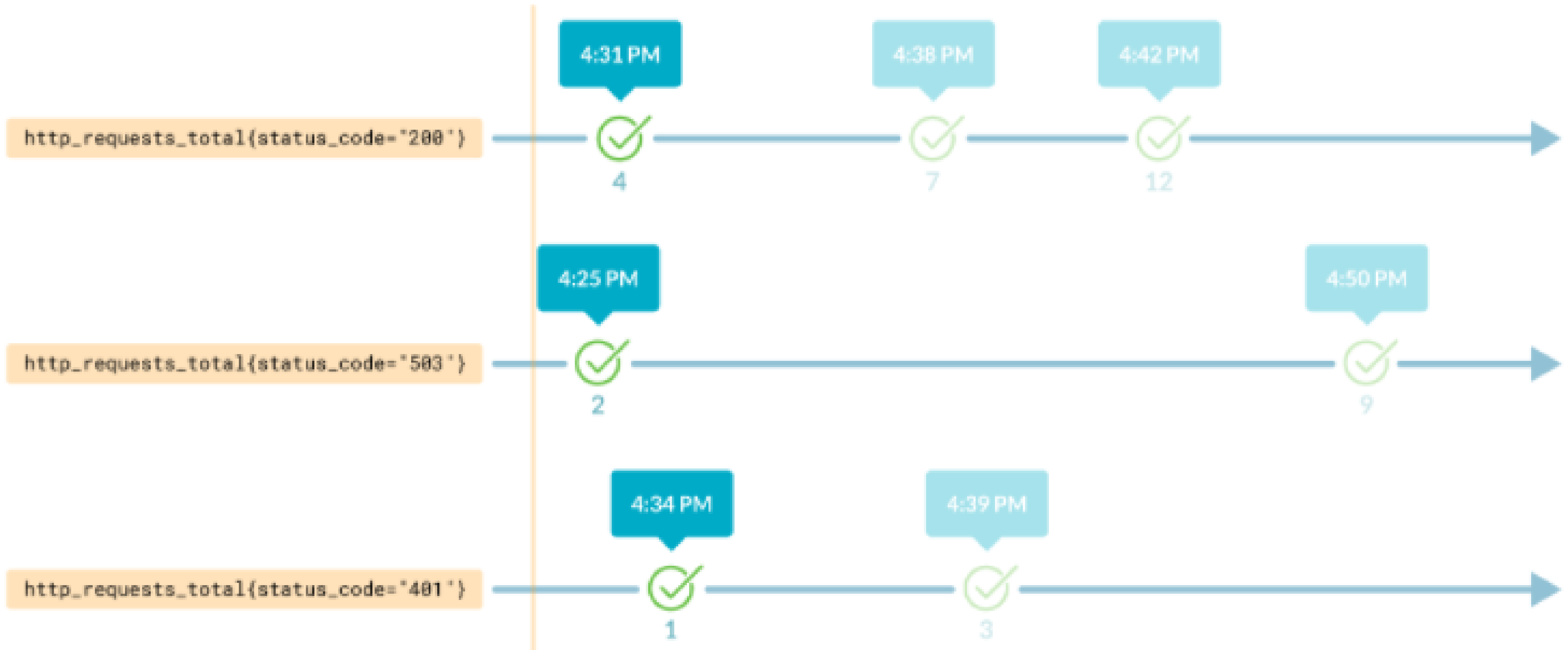
```
promhttp_metric_handler_requests_total
```

Load time: 66ms
Resolution: 14s
Total time series: 6

Element	Value
promhttp_metric_handler_requests_total{code="200",instance="localhost:9090",job="prometheus"}	4133
promhttp_metric_handler_requests_total{code="500",instance="localhost:9090",job="prometheus"}	0
promhttp_metric_handler_requests_total{code="503",instance="localhost:9090",job="prometheus"}	0
promhttp_metric_handler_requests_total{code="200",instance="localhost:9100",job="node-exporter"}	3979
promhttp_metric_handler_requests_total{code="500",instance="localhost:9100",job="node-exporter"}	0
promhttp_metric_handler_requests_total{code="503",instance="localhost:9100",job="node-exporter"}	0

Instant vector Selectors

This is called an **instant vector**, the earliest value for every series at the moment specified by the query. As the samples are taken at random times, Prometheus has to make approximations to select the samples. If no time is specified, then it will return the last available value.



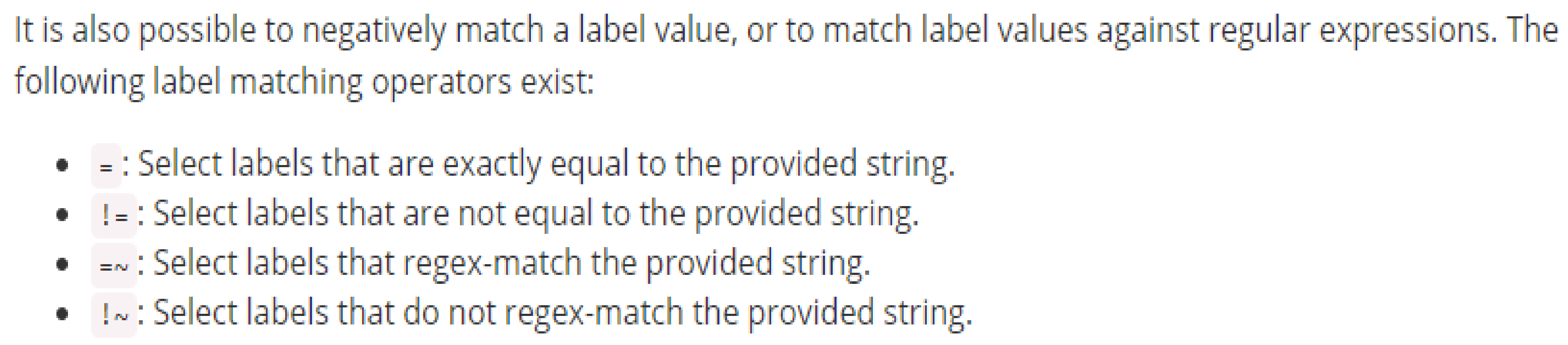
http_requests_total{status_code="200"}
4:31 PM
4
4:38 PM
7
4:42 PM
12
http_requests_total{status_code="503"}
4:25 PM
2
4:50 PM
9
http_requests_total{status_code="401"}
4:34 PM
1
4:39 PM
3

Instant vector Selectors

Example

```
promhttp_http_requests_total
promhttp_http_requests_total{job="prometheus",group="canary"}
promhttp_http_requests_total{environment=~"staging|testing|development",method!="GET"}
promhttp_http_requests_total{job=~".*"} # Bad!
promhttp_http_requests_total{job=~".+"} # Good!
promhttp_http_requests_total{job=~".*",method="get"} # Good!

# Label matchers can also be applied to metric names by matching against the internal __name__ label.
http_requests_total is equivalent to {__name__="http_requests_total"}
{__name__=~"job:.*"}
```

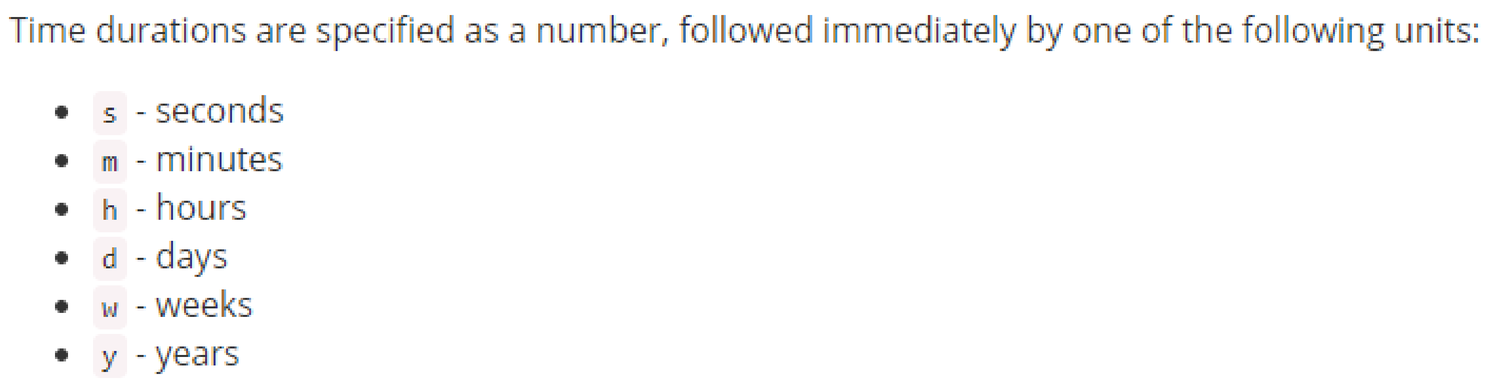


It is also possible to negatively match a label value, or to match label values against regular expressions. The following label matching operators exist:

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

Range Vector Selectors

Range vector literals work like instant vector literals, except that they select a range of samples back from the current instant. Syntactically, a range duration is appended in square brackets ([]) at the end of a vector selector to specify how far back in time values should be fetched for each resulting range vector element.

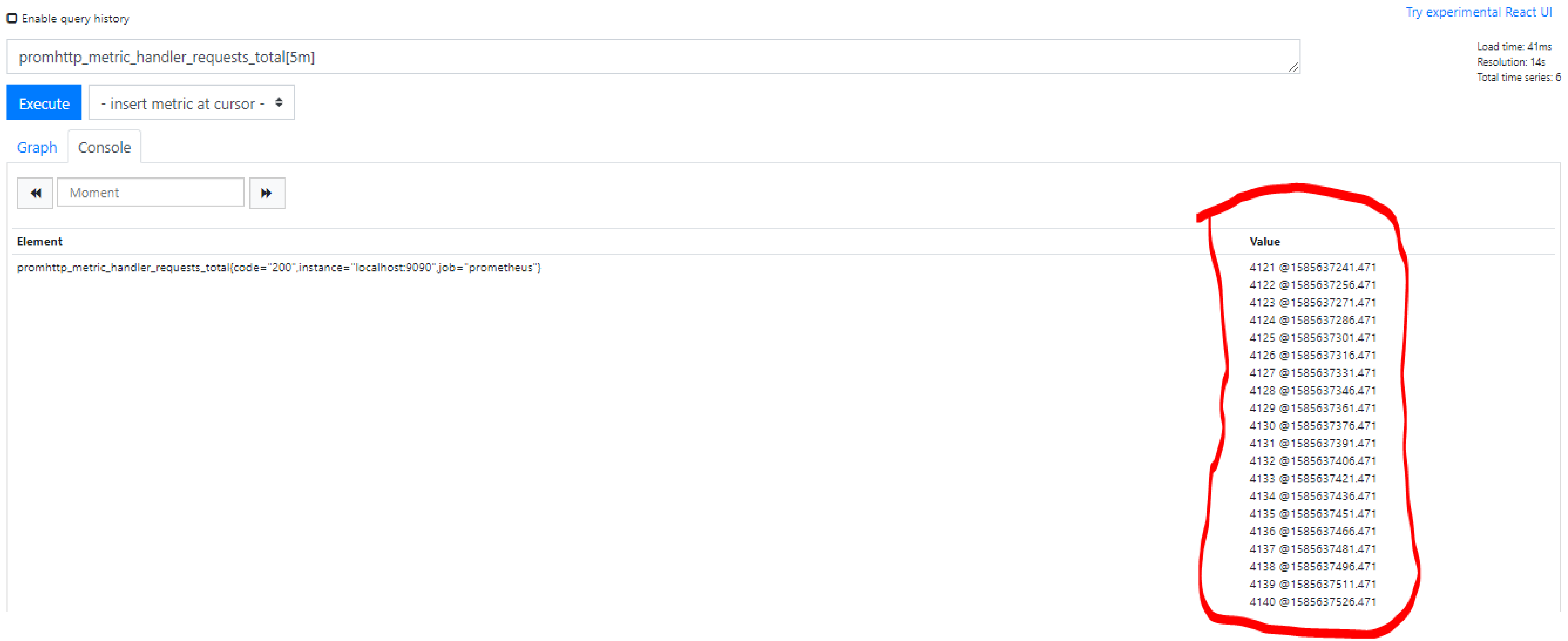

Time durations are specified as a number, followed immediately by one of the following units:

- `s` - seconds
- `m` - minutes
- `h` - hours
- `d` - days
- `w` - weeks
- `y` - years

Example

promhttp_http_requests_total[5m]
promhttp_http_requests_total{job="prometheus",group="canary"}[2h]
promhttp_http_requests_total{environment=~"staging|testing|development",method!="GET"}[60m]

Range Vector Selectors



Enable query history

Try experimental React UI

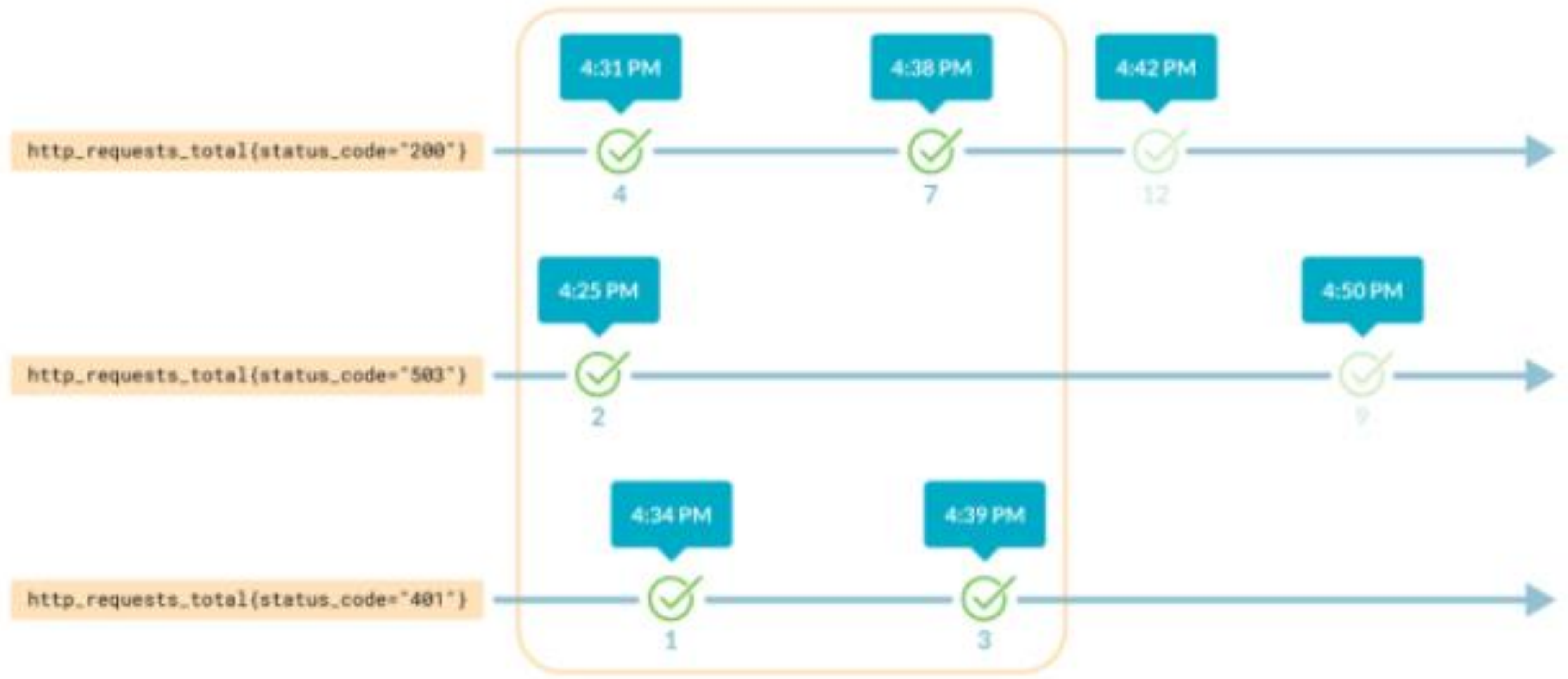
promhttp_metric_handler_requests_total[5m]

Load time: 41ms
Resolution: 14s
Total time series: 6

Element	Value
promhttp_metric_handler_requests_total{code="200",instance="localhost:9090",job="prometheus"}	4121 @1585637241.471
	4122 @1585637256.471
	4123 @1585637271.471
	4124 @1585637286.471
	4125 @1585637301.471
	4126 @1585637316.471
	4127 @1585637331.471
	4128 @1585637346.471
	4129 @1585637361.471
	4130 @1585637376.471
	4131 @1585637391.471
	4132 @1585637406.471
	4133 @1585637421.471
	4134 @1585637436.471
	4135 @1585637451.471
	4136 @1585637466.471
	4137 @1585637481.471
	4138 @1585637496.471
	4139 @1585637511.471
	4140 @1585637526.471

Range Vector Selectors

This is called a range vector: all the values for every series within a range of timestamps.



http_requests_total{status_code="200"}
4:31 PM
4
4:38 PM
7
4:42 PM
12
http_requests_total{status_code="503"}
4:25 PM
2
4:50 PM
9
http_requests_total{status_code="401"}
4:34 PM
1
4:39 PM
3

Offset modifier

The offset modifier allows changing the time offset for individual instant and range vectors in a query.

The following expression returns the value of prometheus_http_requests_total the current query evaluation time:

```
prometheus_http_requests_total
```

The following expression returns the value of http_requests_total 5 minutes in the past relative to the current query evaluation time:

```
prometheus_http_requests_total offset 5m
```

You can use offset to change the time for Instant and Range Vectors. This can be helpful for comparing current usage to past usage when determining the conditions of an alert.

```
sum(rate(prometheus_http_requests_total[5m] offset 5m))
```

Offset modifier

For example, the following expression returns the value of `http_requests_total` 5 minutes in the past relative to the current query evaluation time:

```
http_requests_total offset 5m
```

Note that the `offset` modifier always needs to follow the selector immediately, i.e. the following would be correct:

```
sum(http_requests_total{method="GET"} offset 5m) // GOOD.
```

While the following would be *incorrect*:

```
sum(http_requests_total{method="GET"}) offset 5m // INVALID.
```

The same works for range vectors. This returns the 5-minute rate that `http_requests_total` had a week ago:

```
rate(http_requests_total[5m] offset 1w)
```

Subquery

Subquery allows you to run an instant query for a given range and resolution. The result of a subquery is a range vector.

Syntax: `<instant_query> '[' <range> ':' [<resolution>] ']' [ offset <duration> ]`

- `<resolution>` is optional. Default is the global evaluation interval.

  - Offsets may be added at any place of the query. The following query returns the number of cache requests for the previous day:

```
(rate(hits_total[5m]) + rate(miss_total[5m])) offset 1d
```

This is equivalent to the following PromQL query:

```
rate(hits_total[5m] offset 1d) + rate(miss_total[5m] offset 1d)
```

Subquery

Return the 5-minute rate of the `http_requests_total` metric for the past 30 minutes, with a resolution of 1 minute.

```
rate(http_requests_total[5m])[30m:1m]
```

This is an example of a nested subquery. The subquery for the `deriv` function uses the default resolution. Note that using subqueries unnecessarily is unwise.

```
max_over_time(deriv(rate(distance_covered_total[5s])[30s:5s])[10m:])
```

Demo

1. https://www.devopsschool.com/blog/prometheus-promql-example-query-node-exporter/
2. https://www.devopsschool.com/blog/prometheus-promql-example-query/
3. https://www.devopsschool.com/blog/prometheus-promql-example-query-monitoring-kubernetes/