QUERYING PROMETHEUS

Prometheus provides a functional query language called PromQL (Prometheus Query Language) that lets the user select and aggregate time series data in real time.

When you send a query request to Prometheus, it can be an instant query, evaluated at one point in time, or a range query at equally-spaced steps between a start and an end time. PromQL works exactly the same in each cases; the range query is just like an instant query run multiple times at different timestamps.

In the Prometheus UI, the "Table" tab is for instant queries and the "Graph" tab is for range queries.

Other programs can fetch the result of a PromQL expression via the HTTP API.

Examples

This document is a Prometheus basic language reference. For learning, it may be easier to start with a couple of examples.

Expression language data types

In Prometheus's expression language, 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
- Range vector - a set of time series containing a range of data points over time for each time series
- Scalar - a simple numeric floating point value
- String - a simple string value; currently unused

Depending on the use-case (e.g. when graphing vs. displaying the output of an expression), only some of these types are legal as the result of a user-specified expression. For instant queries, any of the above data types are allowed as the root of the expression. Range queries only support scalar-typed and instant-vector-typed expressions.

Notes about the experimental native histograms:

- Ingesting native histograms has to be enabled via a feature flag.
- Once native histograms have been ingested into the TSDB (and even after disabling the feature flag again), both instant vectors and range vectors may now contain samples that aren't simple floating point numbers (float samples) but complete histograms (histogram samples). A vector may contain a mix of float samples and histogram samples.

Literals

String literals

String literals are designated by single quotes, double quotes or backticks.

PromQL follows the same escaping rules as Go. For string literals in single or double quotes, a backslash begins an escape sequence, which may be followed by `a`, `b`, `f`, `n`, `r`, `t`, `v` or `\`. Specific characters can be provided using octal (`\nnn`) or hexadecimal (`\xnn`, `\unnnn` and `\Unnnnnnnn`) notations.

Conversely, escape characters are not parsed in string literals designated by backticks. It is important to note that, unlike Go, Prometheus does not discard newlines inside backticks.

Example:

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

Float literals and time durations

Scalar float values can be written as literal integer or floating-point numbers in the format (whitespace only included for better readability):

```
[-+]?(
      [0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?
    | 0[xX][0-9a-fA-F]+
    | [nN][aA][nN]
    | [iI][nN][fF]
```

```
)
```
Examples:

```
23
-2.43
3.4e-9
0x8f
-Inf
NaN
```
Additionally, underscores (_) can be used in between decimal or hexadecimal digits to improve readability.

Examples:

```
1_000_000
.123_456_789
0x_53_AB_F3_82
```
Float literals are also used to specify durations in seconds. For convenience, decimal integer numbers may be combined with the following time units:

- `ms` – milliseconds
- `s` – seconds – 1s equals 1000ms
- `m` – minutes – 1m equals 60s (ignoring leap seconds)
- `h` – hours – 1h equals 60m
- `d` – days – 1d equals 24h (ignoring so-called daylight saving time)
- `w` – weeks – 1w equals 7d
- `y` – years – 1y equals 365d (ignoring leap days)

Suffixing a decimal integer number with one of the units above is a different representation of the equivalent number of seconds as a bare float literal.

Examples:

```
1s # Equivalent to 1.
2m # Equivalent to 120.
1ms # Equivalent to 0.001.
-2h # Equivalent to -7200.
```
The following examples do not work:

```
0xABm # No suffixing of hexadecimal numbers.
1.5h # Time units cannot be combined with a floating point.
+Infd # No suffixing of ±Inf or NaN.
```

Multiple units can be combined by concatenation of suffixed integers. Units must be ordered from the longest to the shortest. A given unit must only appear once per float literal.

Examples:

```
1h30m # Equivalent to 5400s and thus 5400.
12h34m56s # Equivalent to 45296s and thus 45296.
54s321ms # Equivalent to 54.321.
```

Time series selectors

These are the basic building-blocks that instruct PromQL what data to fetch.

Instant vector selectors

Instant vector selectors allow the selection of a set of time series and a single sample value for each at a given timestamp (point in time). In the simplest form, only a metric name is specified, which results in an instant vector containing elements for all time series that have this metric name.

The value returned will be that of the most recent sample at or before the query's evaluation timestamp (in the case of an instant query) or the current step within the query (in the case of a range query). The `@` modifier allows overriding the timestamp relative to which the selection takes place. Time series are only returned if their most recent sample is less than the lookback period ago.

This example selects all time series that have the `http_requests_total` metric name, returning the most recent sample for each:

```
http_requests_total
```

It is possible to filter these time series further by appending a comma-separated list of label matchers in curly braces (`{}`).

This example selects only those time series with the `http_requests_total` metric name that also have the `job` label set to `prometheus` and their `group` label set to `canary`:

```
http_requests_total{job="prometheus",group="canary"}
```

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.

Regex matches are fully anchored. A match of `env=~"foo"` is treated as `env=~"^foo$"`.

For example, this selects all `http_requests_total` time series for `staging`, `testing`, and `development` environments and HTTP methods other than `GET`.

```
http_requests_total{environment=~"staging|testing|
development",method!="GET"}
```

Label matchers that match empty label values also select all time series that do not have the specific label set at all. It is possible to have multiple matchers for the same label name.

For example, given the dataset:

```
http_requests_total
http_requests_total{replica="rep-a"}
http_requests_total{replica="rep-b"}
http_requests_total{environment="development"}
```

The query `http_requests_total{environment=""}` would match and return:

```
http_requests_total
http_requests_total{replica="rep-a"}
http_requests_total{replica="rep-b"}
```

and would exclude:

```
http_requests_total{environment="development"}
```

Multiple matchers can be used for the same label name; they all must pass for a result to be returned.

The query:

```
http_requests_total{replica!="rep-a",replica=~"rep.*"}
```

Would then match:

```
http_requests_total{replica="rep-b"}
```
Vector selectors must either specify a name or at least one label matcher that does not match the empty string. The following expression is illegal:

```
{job=~".*"} # Bad!
```
In contrast, these expressions are valid as they both have a selector that does not match empty label values.

```
{job=~".+"}              # Good!
{job=~".*",method="get"} # Good!
```
Label matchers can also be applied to metric names by matching against the internal `__name__` label. For example, the expression `http_requests_total` is equivalent to `{__name__="http_requests_total"}`. Matchers other than `=` (`!=`, `=~`, `!~`) may also be used. The following expression selects all metrics that have a name starting with `job:`:

```
{__name__=~"job:.*"}
```
The metric name must not be one of the keywords `bool`, `on`, `ignoring`, `group_left` and `group_right`. The following expression is illegal:

```
on{} # Bad!
```
A workaround for this restriction is to use the `__name__` label:

```
{__name__="on"} # Good!
```
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 float literal is appended in square brackets (`[]`) at the end of a vector selector to specify for how many seconds back in time values should be fetched for each resulting range vector element. Commonly, the float literal uses the syntax with one or more time units, e.g. `[5m]`. The range is a left-open and right-closed interval, i.e. samples with timestamps coinciding with the left boundary of the range are excluded from the selection, while samples coinciding with the right boundary of the range are included in the selection.

In this example, we select all the values recorded less than 5m ago for all time series that have the metric name `http_requests_total` and a `job` label set to `prometheus`:

```
http_requests_total{job="prometheus"}[5m]
```

Offset modifier

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

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)
```

When querying for samples in the past, a negative offset will enable temporal comparisons forward in time:

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

Note that this allows a query to look ahead of its evaluation time.

@ modifier

The `@` modifier allows changing the evaluation time for individual instant and range vectors in a query. The time supplied to the `@` modifier is a unix timestamp and described with a float literal.

For example, the following expression returns the value of `http_requests_total` at `2021-01-04T07:40:00+00:00`:

```
http_requests_total @ 1609746000
```

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

```
sum(http_requests_total{method="GET"} @ 1609746000) // GOOD.
```

While the following would be incorrect:

```
sum(http_requests_total{method="GET"}) @ 1609746000 //
INVALID.
```

The same works for range vectors. This returns the 5-minute rate that `http_requests_total` had at `2021-01-04T07:40:00+00:00`:

```
rate(http_requests_total[5m] @ 1609746000)
```

The `@` modifier supports all representations of numeric literals described above. It works with the `offset` modifier where the offset is applied relative to the `@` modifier time. The results are the same irrespective of the order of the modifiers.

For example, these two queries will produce the same result:

```
# offset after @
http_requests_total @ 1609746000 offset 5m
# offset before @
http_requests_total offset 5m @ 1609746000
```

Additionally, `start()` and `end()` can also be used as values for the `@` modifier as special values.

For a range query, they resolve to the start and end of the range query respectively and remain the same for all steps.

For an instant query, `start()` and `end()` both resolve to the evaluation time.

```
http_requests_total @ start()
rate(http_requests_total[5m] @ end())
```

Note that the `@` modifier allows a query to look ahead of its evaluation time.

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>] ']' [ @ <float_literal> ] [ offset <float_literal> ]`

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

Operators

Prometheus supports many binary and aggregation operators. These are described in detail in the expression language operators page.

Functions

Prometheus supports several functions to operate on data. These are described in detail in the expression language functions page.

Comments

PromQL supports line comments that start with `#`. Example:

```
    # This is a comment
```

Regular expressions

All regular expressions in Prometheus use RE2 syntax.

Regex matches are always fully anchored.

Gotchas

Staleness

The timestamps at which to sample data, during a query, are selected independently of the actual present time series data. This is mainly to support cases like aggregation (`sum`, `avg`, and so on), where multiple aggregated time series do not precisely align in time. Because of their independence, Prometheus needs to assign a value at those timestamps for each relevant time series. It does so by taking the newest sample that is less than the lookback period ago. The lookback period is 5 minutes by default, but can be set with the `--query.lookback-delta` flag

If a target scrape or rule evaluation no longer returns a sample for a time series that was previously present, this time series will be marked as stale. If a target is removed, the previously retrieved time series will be marked as stale soon after removal.

If a query is evaluated at a sampling timestamp after a time series is marked as stale, then no value is returned for that time series. If new samples are subsequently ingested for that time series, they will be returned as expected.

A time series will go stale when it is no longer exported, or the target no longer exists. Such time series will disappear from graphs at the times of their latest collected sample, and they will not be returned in queries after they are marked stale.

Some exporters, which put their own timestamps on samples, get a different behaviour: series that stop being exported take the last value for (by default) 5 minutes before disappearing. The `track_timestamps_staleness` setting can change this.

Avoiding slow queries and overloads

If a query needs to operate on a substantial amount of data, graphing it might time out or overload the server or browser. Thus, when constructing queries over unknown data, always start building the query in the tabular view of Prometheus's expression browser until the result set seems reasonable (hundreds, not thousands, of time series at most). Only when you have filtered or aggregated your data sufficiently, switch to graph mode. If the expression still takes too long to graph ad-hoc, pre-record it via a recording rule.

This is especially relevant for Prometheus's query language, where a bare metric name selector like `api_http_requests_total` could expand to thousands of time series with different labels. Also, keep in mind that expressions that aggregate over many time series will generate load on the server even if the output is only a small number of time series. This is similar to how it would be slow to sum all values of a column in a relational database, even if the output value is only a single number.

OPERATORS

Binary operators

Prometheus's query language supports basic logical and arithmetic operators. For operations between two instant vectors, the matching behavior can be modified.

Arithmetic binary operators

The following binary arithmetic operators exist in Prometheus:

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

Binary arithmetic operators are defined between scalar/scalar, vector/scalar, and vector/vector value pairs.

Between two scalars, the behavior is obvious: they evaluate to another scalar that is the result of the operator applied to both scalar operands.

Between an instant vector and a scalar, the operator is applied to the value of every data sample in the vector. E.g. if a time series instant vector is multiplied by 2, the result is another vector in which every sample value of the original vector is multiplied by 2. The metric name is dropped.

Between two instant vectors, a binary arithmetic operator is applied to each entry in the left-hand side vector and its matching element in the right-hand vector. The result is propagated into the result vector with the grouping labels becoming the output label set. The metric name is dropped. Entries for which no matching entry in the right-hand vector can be found are not part of the result.

Trigonometric binary operators

The following trigonometric binary operators, which work in radians, exist in Prometheus:

- `atan2` (based on https://pkg.go.dev/math#Atan2)

Trigonometric operators allow trigonometric functions to be executed on two vectors using vector matching, which isn't available with normal functions. They act in the same manner as arithmetic 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)

Comparison operators are defined between scalar/scalar, vector/scalar, and vector/vector value pairs. By default they filter. Their behavior can be modified by providing `bool` after the operator, which will return `0` or `1` for the value rather than filtering.

Between two scalars, the `bool` modifier must be provided and these operators result in another scalar that is either `0` (`false`) or `1` (`true`), depending on the comparison result.

Between an instant vector and a scalar, these operators are applied to the value of every data sample in the vector, and vector elements between which the comparison result is `false` get dropped from the result vector. If the `bool` modifier is provided, vector elements that would be dropped instead have the value `0` and vector elements that would be kept have the value `1`. The metric name is dropped if the `bool` modifier is provided.

Between two instant vectors, these operators behave as a filter by default, applied to matching entries. Vector elements for which the expression is not true or which do not find a match on the other side of the expression get dropped from the result, while the others are propagated into a result vector with the grouping labels

becoming the output label set. If the `bool` modifier is provided, vector elements that would have been dropped instead have the value `0` and vector elements that would be kept have the value `1`, with the grouping labels again becoming the output label set. The metric name is dropped if the `bool` modifier is provided.

Logical/set binary operators

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

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

`vector1 and vector2` results in a vector consisting of the elements of `vector1` for which there are elements in `vector2` with exactly matching label sets. Other elements are dropped. The metric name and values are carried over from the left-hand side vector.

`vector1 or vector2` results in a vector that contains all original elements (label sets + values) of `vector1` and additionally all elements of `vector2` which do not have matching label sets in `vector1`.

`vector1 unless vector2` results in a vector consisting of the elements of `vector1` for which there are no elements in `vector2` with exactly matching label sets. All matching elements in both vectors are dropped.

Vector matching

Operations between vectors attempt to find a matching element in the right-hand side vector for each entry in the left-hand side. There are two basic types of matching behavior: One-to-one and many-to-one/one-to-many.

Vector matching keywords

These vector matching keywords allow for matching between series with different label sets providing:

- `on`
- `ignoring`

Label lists provided to matching keywords will determine how vectors are combined. Examples can be found in One-to-one vector matches and in Many-to-one and one-to-many vector matches

Group modifiers

These group modifiers enable many-to-one/one-to-many vector matching:

- `group_left`
- `group_right`

Label lists can be provided to the group modifier which contain labels from the "one"-side to be included in the result metrics.

Many-to-one and one-to-many matching are advanced use cases that should be carefully considered. Often a proper use of `ignoring(<labels>)` provides the desired outcome.

Grouping modifiers can only be used for comparison and arithmetic. Operations as *`and`*, *`unless`* and *`or`* operations match with all possible entries in the right vector by default.

One-to-one vector matches

One-to-one finds a unique pair of entries from each side of the operation. In the default case, that is an operation following the format `vector1 <operator> vector2`. Two entries match if they have the exact same set of labels and corresponding values. The `ignoring` keyword allows ignoring certain labels when matching, while the `on` keyword allows reducing the set of considered labels to a provided list:

```
<vector expr> <bin-op> ignoring(<label list>) <vector expr>
<vector expr> <bin-op> on(<label list>) <vector expr>
```

Example input:

```
method_code:http_errors:rate5m{method="get", code="500"}  24
method_code:http_errors:rate5m{method="get", code="404"}  30
method_code:http_errors:rate5m{method="put", code="501"}  3
method_code:http_errors:rate5m{method="post", code="500"} 6
```

```
method_code:http_errors:rate5m{method="post", code="404"} 21

method:http_requests:rate5m{method="get"}  600
method:http_requests:rate5m{method="del"}  34
method:http_requests:rate5m{method="post"} 120
```

Example query:

```
method_code:http_errors:rate5m{code="500"} / ignoring(code)
method:http_requests:rate5m
```

This returns a result vector containing the fraction of HTTP requests with status code of 500 for each method, as measured over the last 5 minutes. Without `ignoring(code)` there would have been no match as the metrics do not share the same set of labels. The entries with methods `put` and `del` have no match and will not show up in the result:

```
{method="get"}  0.04            //  24 / 600
{method="post"} 0.05            //   6 / 120
```

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

Many-to-one and one-to-many matchings refer to the case where each vector element on the "one"-side can match with multiple elements on the "many"-side. This has to be explicitly requested using the `group_left` or `group_right` modifiers, where left/right determines which vector has the higher cardinality.

```
<vector expr> <bin-op> ignoring(<label list>)
group_left(<label list>) <vector expr>
<vector expr> <bin-op> ignoring(<label list>)
group_right(<label list>) <vector expr>
<vector expr> <bin-op> on(<label list>) group_left(<label
list>) <vector expr>
<vector expr> <bin-op> on(<label list>) group_right(<label
list>) <vector expr>
```

The label list provided with the group modifier contains additional labels from the "one"-side to be included in the result metrics. For `on` a label can only appear in one of the lists. Every time series of the result vector must be uniquely identifiable.

Example query:

```
method_code:http_errors:rate5m / ignoring(code) group_left
method:http_requests:rate5m
```

In this case the left vector contains more than one entry per `method` label value. Thus, we indicate this using `group_left`. The elements from the right side are now matched with multiple elements with the same `method` label on the left:

```
{method="get", code="500"}  0.04            //  24 / 600
{method="get", code="404"}  0.05            //  30 / 600
{method="post", code="500"} 0.05            //   6 / 120
{method="post", code="404"} 0.175           //  21 / 120
```

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)
- `group` (all values in the resulting vector are 1)
- `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)
- `limitk` (sample n elements)
- `limit_ratio` (sample elements with approximately r ratio if r `> 0`, and the complement of such samples if r `= -(1.0 - r)`)

These operators can either be used to aggregate over all label dimensions or preserve distinct dimensions by including a `without` or `by` clause. These clauses may be used before or after the expression.

```
<aggr-op> [without|by (<label list>)] ([parameter,] <vector expression>)
```

or

`<aggr-op>([parameter,] <vector expression>) [without|by (<label list>)]`
`label list` is a list of unquoted labels that may include a trailing comma, i.e. both `(label1, label2)` and `(label1, label2,)` are valid syntax.

`without` removes the listed labels from the result vector, while all other labels are preserved in the output. `by` does the opposite and drops labels that are not listed in the `by` clause, even if their label values are identical between all elements of the vector.

`parameter` is only required for `count_values`, `quantile`, `topk`, `bottomk`, `limitk` and `limit_ratio`.

`count_values` outputs one time series per unique sample value. Each series has an additional label. The name of that label is given by the aggregation parameter, and the label value is the unique sample value. The value of each time series is the number of times that sample value was present.

`topk` and `bottomk` are different from other aggregators in that a subset of the input samples, including the original labels, are returned in the result vector. `by` and `without` are only used to bucket the input vector.

`limitk` and `limit_ratio` also return a subset of the input samples, including the original labels in the result vector, these are experimental operators that must be enabled with `--enable-feature=promql-experimental-functions`.

`quantile` calculates the φ-quantile, the value that ranks at number φ*N among the N metric values of the dimensions aggregated over. φ is provided as the aggregation parameter. For example, `quantile(0.5, ...)` calculates the median, `quantile(0.95, ...)` the 95th percentile. For φ = `NaN`, `NaN` is returned. For φ < 0, `-Inf` is returned. For φ > 1, `+Inf` is returned.

Example:

If the metric `http_requests_total` had time series that fan out by `application`, `instance`, and `group` labels, we could calculate the total number of seen HTTP requests per application and group over all instances via:

```
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)
```

To sample 10 timeseries, for example to inspect labels and their values, we could write:

```
limitk(10, http_requests_total)
```

To deterministically sample approximately 10% of timeseries we could write:

```
limit_ratio(0.1, http_requests_total)
```

Given that `limit_ratio()` implements a deterministic sampling algorithm (based on labels' hash), you can get the complement of the above samples, i.e. approximately 90%, but precisely those not returned by `limit_ratio(0.1, ...)` with:

```
limit_ratio(-0.9, http_requests_total)
```

You can also use this feature to e.g. verify that `avg()` is a representative aggregation for your samples' values, by checking that the difference between averaging two samples' subsets is "small" when compared to the standard deviation.

```
abs(
  avg(limit_ratio(0.5, http_requests_total))
  -
  avg(limit_ratio(-0.5, http_requests_total))
```

```
) <= bool stddev(http_requests_total)
```

Binary operator precedence

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

1. `^`
2. `*`, `/`, `%`, `atan2`
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)`.

Operators for native histograms

Native histograms are an experimental feature. Ingesting native histograms has to be enabled via a feature flag. Once native histograms have been ingested, they can be queried (even after the feature flag has been disabled again). However, the operator support for native histograms is still very limited.

Logical/set binary operators work as expected even if histogram samples are involved. They only check for the existence of a vector element and don't change their behavior depending on the sample type of an element (float or histogram). The `count` aggregation operator works similarly.

The binary `+` and `-` operators between two native histograms and the `sum` and `avg` aggregation operators to aggregate native histograms are fully supported. Even if the histograms involved have different bucket layouts, the buckets are automatically converted appropriately so that the operation can be performed. (With the currently supported bucket schemas, that's always possible.) If either operator has to aggregate a mix of histogram samples and float samples, the corresponding vector element is removed from the output vector entirely.

The binary `*` operator works between a native histogram and a float in any order, while the binary `/` operator can be used between a native histogram and a float in that exact order.

All other operators (and unmentioned cases for the above operators) do not behave in a meaningful way. They either treat the histogram sample as if it were a float sample of value 0, or (in case of arithmetic operations between a scalar and a vector) they leave the histogram sample unchanged. This behavior will change to a meaningful one before native histograms are a stable feature.

FUNCTIONS

Some functions have default arguments, e.g. `year(v=vector(time()) instant-vector)`. This means that there is one argument `v` which is an instant vector, which if not provided it will default to the value of the expression `vector(time())`.

Notes about the experimental native histograms:

- Ingesting native histograms has to be enabled via a feature flag. As long as no native histograms have been ingested into the TSDB, all functions will behave as usual.
- Functions that do not explicitly mention native histograms in their documentation (see below) will ignore histogram samples.
- Functions that do already act on native histograms might still change their behavior in the future.
- If a function requires the same bucket layout between multiple native histograms it acts on, it will automatically convert them appropriately. (With the currently supported bucket schemas, that's always possible.)

abs()

`abs(v instant-vector)` returns the input vector with all sample values converted to their absolute value.

absent()

`absent(v instant-vector)` returns an empty vector if the vector passed to it has any elements (floats or native histograms) and a 1-element vector with the value 1 if the vector passed to it has no elements.

This is useful for alerting on when no time series exist for a given metric name and label combination.

```
absent(nonexistent{job="myjob"})
# => {job="myjob"}

absent(nonexistent{job="myjob",instance=~".*"})
```

```
# => {job="myjob"}

absent(sum(nonexistent{job="myjob"}))
# => {}
```

In the first two examples, `absent()` tries to be smart about deriving labels of the 1-element output vector from the input vector.

absent_over_time()

`absent_over_time(v range-vector)` returns an empty vector if the range vector passed to it has any elements (floats or native histograms) and a 1-element vector with the value 1 if the range vector passed to it has no elements.

This is useful for alerting on when no time series exist for a given metric name and label combination for a certain amount of time.

```
absent_over_time(nonexistent{job="myjob"}[1h])
# => {job="myjob"}

absent_over_time(nonexistent{job="myjob",instance=~".*"}[1h])
# => {job="myjob"}

absent_over_time(sum(nonexistent{job="myjob"})[1h:])
# => {}
```

In the first two examples, `absent_over_time()` tries to be smart about deriving labels of the 1-element output vector from the input vector.

ceil()

`ceil(v instant-vector)` rounds the sample values of all elements in `v` up to the nearest integer value greater than or equal to v.

- `ceil(+Inf) = +Inf`
- `ceil(±0) = ±0`
- `ceil(1.49) = 2.0`
- `ceil(1.78) = 2.0`

changes()

For each input time series, `changes(v range-vector)` returns the number of times its value has changed within the provided time range as an instant vector.

clamp()

`clamp(v instant-vector, min scalar, max scalar)` clamps the sample values of all elements in `v` to have a lower limit of `min` and an upper limit of `max`.

Special cases:

- Return an empty vector if `min > max`
- Return `NaN` if `min` or `max` is `NaN`

clamp_max()

`clamp_max(v instant-vector, max scalar)` clamps the sample values of all elements in `v` to have an upper limit of `max`.

clamp_min()

`clamp_min(v instant-vector, min scalar)` clamps the sample values of all elements in `v` to have a lower limit of `min`.

day_of_month()

`day_of_month(v=vector(time()) instant-vector)` returns the day of the month for each of the given times in UTC. Returned values are from 1 to 31.

day_of_week()

`day_of_week(v=vector(time()) instant-vector)` returns the day of the week for each of the given times in UTC. Returned values are from 0 to 6, where 0 means Sunday etc.

day_of_year()

`day_of_year(v=vector(time()) instant-vector)` returns the day of the year for each of the given times in UTC. Returned values are from 1 to 365 for non-leap years, and 1 to 366 in leap years.

days_in_month()

`days_in_month(v=vector(time()) instant-vector)` returns number of days in the month for each of the given times in UTC. Returned values are from 28 to 31.

delta()

`delta(v range-vector)` calculates the difference between the first and last value of each time series element in a range vector `v`, returning an instant vector with the given deltas and equivalent labels. The delta is extrapolated to cover the full time range as specified in the range vector selector, so that it is possible to get a non-integer result even if the sample values are all integers.

The following example expression returns the difference in CPU temperature between now and 2 hours ago:

```
delta(cpu_temp_celsius{host="zeus"}[2h])
```

`delta` acts on native histograms by calculating a new histogram where each component (sum and count of observations, buckets) is the difference between the respective component in the first and last native histogram in `v`. However, each element in `v` that contains a mix of float and native histogram samples within the range, will be missing from the result vector.

`delta` should only be used with gauges and native histograms where the components behave like gauges (so-called gauge histograms).

deriv()

`deriv(v range-vector)` calculates the per-second derivative of the time series in a range vector `v`, using simple linear regression. The range vector must have at least two samples in order to perform the calculation. When `+Inf` or `-Inf` are found in the range vector, the slope and offset value calculated will be `NaN`.

`deriv` should only be used with gauges.

exp()

`exp(v instant-vector)` calculates the exponential function for all elements in `v`. Special cases are:

- `Exp(+Inf) = +Inf`
- `Exp(NaN) = NaN`

`floor()`

`floor(v instant-vector)` rounds the sample values of all elements in `v` down to the nearest integer value smaller than or equal to v.

- `floor(+Inf) = +Inf`
- `floor(±0) = ±0`
- `floor(1.49) = 1.0`
- `floor(1.78) = 1.0`

`histogram_avg()`

This function only acts on native histograms, which are an experimental feature. The behavior of this function may change in future versions of Prometheus, including its removal from PromQL.

`histogram_avg(v instant-vector)` returns the arithmetic average of observed values stored in a native histogram. Samples that are not native histograms are ignored and do not show up in the returned vector.

Use `histogram_avg` as demonstrated below to compute the average request duration over a 5-minute window from a native histogram:

```
histogram_avg(rate(http_request_duration_seconds[5m]))
```

Which is equivalent to the following query:

```
  histogram_sum(rate(http_request_duration_seconds[5m]))
/
  histogram_count(rate(http_request_duration_seconds[5m]))
```

`histogram_count()` and `histogram_sum()`

Both functions only act on native histograms, which are an experimental feature. The behavior of these functions may change in future versions of Prometheus, including their removal from PromQL.

`histogram_count(v instant-vector)` returns the count of observations stored in a native histogram. Samples that are not

native histograms are ignored and do not show up in the returned vector.

Similarly, `histogram_sum(v instant-vector)` returns the sum of observations stored in a native histogram.

Use `histogram_count` in the following way to calculate a rate of observations (in this case corresponding to “requests per second”) from a native histogram:

```
histogram_count(rate(http_request_duration_seconds[10m]))
```

`histogram_fraction()`

This function only acts on native histograms, which are an experimental feature. The behavior of this function may change in future versions of Prometheus, including its removal from PromQL.

For a native histogram, `histogram_fraction(lower scalar, upper scalar, v instant-vector)` returns the estimated fraction of observations between the provided lower and upper values. Samples that are not native histograms are ignored and do not show up in the returned vector.

For example, the following expression calculates the fraction of HTTP requests over the last hour that took 200ms or less:

```
histogram_fraction(0, 0.2,
rate(http_request_duration_seconds[1h]))
```

The error of the estimation depends on the resolution of the underlying native histogram and how closely the provided boundaries are aligned with the bucket boundaries in the histogram.

`+Inf` and `-Inf` are valid boundary values. For example, if the histogram in the expression above included negative observations (which shouldn't be the case for request durations), the appropriate lower boundary to include all observations less than or equal 0.2 would be `-Inf` rather than `0`.

Whether the provided boundaries are inclusive or exclusive is only relevant if the provided boundaries are precisely aligned with bucket boundaries in the underlying native histogram. In this case,

the behavior depends on the schema definition of the histogram. The currently supported schemas all feature inclusive upper boundaries and exclusive lower boundaries for positive values (and vice versa for negative values). Without a precise alignment of boundaries, the function uses linear interpolation to estimate the fraction. With the resulting uncertainty, it becomes irrelevant if the boundaries are inclusive or exclusive.

`histogram_quantile()`

`histogram_quantile(φ scalar, b instant-vector)` calculates the φ-quantile (0 ≤ φ ≤ 1) from a classic histogram or from a native histogram. (See histograms and summaries for a detailed explanation of φ-quantiles and the usage of the (classic) histogram metric type in general.)

Note that native histograms are an experimental feature. The behavior of this function when dealing with native histograms may change in future versions of Prometheus.

The float samples in `b` are considered the counts of observations in each bucket of one or more classic histograms. Each float sample must have a label `le` where the label value denotes the inclusive upper bound of the bucket. (Float samples without such a label are silently ignored.) The other labels and the metric name are used to identify the buckets belonging to each classic histogram. The histogram metric type automatically provides time series with the `_bucket` suffix and the appropriate labels.

The native histogram samples in `b` are treated each individually as a separate histogram to calculate the quantile from.

As long as no naming collisions arise, `b` may contain a mix of classic and native histograms.

Use the `rate()` function to specify the time window for the quantile calculation.

Example: A histogram metric is called `http_request_duration_seconds` (and therefore the metric name for the buckets of a classic histogram is

`http_request_duration_seconds_bucket`). To calculate the 90th percentile of request durations over the last 10m, use the following expression in case `http_request_duration_seconds` is a classic histogram:

```
histogram_quantile(0.9,
rate(http_request_duration_seconds_bucket[10m]))
```

For a native histogram, use the following expression instead:

```
histogram_quantile(0.9,
rate(http_request_duration_seconds[10m]))
```

The quantile is calculated for each label combination in `http_request_duration_seconds`. To aggregate, use the `sum()` aggregator around the `rate()` function. Since the `le` label is required by `histogram_quantile()` to deal with classic histograms, it has to be included in the `by` clause. The following expression aggregates the 90th percentile by `job` for classic histograms:

```
histogram_quantile(0.9, sum by (job, le)
(rate(http_request_duration_seconds_bucket[10m])))
```

When aggregating native histograms, the expression simplifies to:

```
histogram_quantile(0.9, sum by (job)
(rate(http_request_duration_seconds[10m])))
```

To aggregate all classic histograms, specify only the `le` label:

```
histogram_quantile(0.9, sum by (le)
(rate(http_request_duration_seconds_bucket[10m])))
```

With native histograms, aggregating everything works as usual without any `by` clause:

```
histogram_quantile(0.9,
sum(rate(http_request_duration_seconds[10m])))
```

In the (common) case that a quantile value does not coincide with a bucket boundary, the `histogram_quantile()` function interpolates the quantile value within the bucket the quantile value falls into. For classic histograms, for native histograms with custom bucket boundaries, and for the zero bucket of other native histograms, it assumes a uniform distribution of observations within the bucket (also called linear interpolation). For the non-zero-buckets of native histograms with a standard exponential bucketing schema, the interpolation is done under the assumption that the samples within

the bucket are distributed in a way that they would uniformly populate the buckets in a hypothetical histogram with higher resolution. (This is also called exponential interpolation.)

If `b` has 0 observations, `NaN` is returned. For $\phi < 0$, `-Inf` is returned. For $\phi > 1$, `+Inf` is returned. For $\phi =$ `NaN`, `NaN` is returned.

Special cases for classic histograms:

- If `b` contains fewer than two buckets, `NaN` is returned.
- The highest bucket must have an upper bound of `+Inf`. (Otherwise, `NaN` is returned.)
- If a quantile is located in the highest bucket, the upper bound of the second highest bucket is returned.
- The lower limit of the lowest bucket is assumed to be 0 if the upper bound of that bucket is greater than 0. In that case, the usual linear interpolation is applied within that bucket. Otherwise, the upper bound of the lowest bucket is returned for quantiles located in the lowest bucket.

Special cases for native histograms (relevant for the exact interpolation happening within the zero bucket):

- A zero bucket with finite width is assumed to contain no negative observations if the histogram has observations in positive buckets, but none in negative buckets.
- A zero bucket with finite width is assumed to contain no positive observations if the histogram has observations in negative buckets, but none in positive buckets.

You can use `histogram_quantile(0, v instant-vector)` to get the estimated minimum value stored in a histogram.

You can use `histogram_quantile(1, v instant-vector)` to get the estimated maximum value stored in a histogram.

Buckets of classic histograms are cumulative. Therefore, the following should always be the case:

- The counts in the buckets are monotonically increasing (strictly non-decreasing).

- A lack of observations between the upper limits of two consecutive buckets results in equal counts in those two buckets.

However, floating point precision issues (e.g. small discrepancies introduced by computing of buckets with `sum(rate(...))`) or invalid data might violate these assumptions. In that case, `histogram_quantile` would be unable to return meaningful results. To mitigate the issue, `histogram_quantile` assumes that tiny relative differences between consecutive buckets are happening because of floating point precision errors and ignores them. (The threshold to ignore a difference between two buckets is a trillionth (1e-12) of the sum of both buckets.) Furthermore, if there are non-monotonic bucket counts even after this adjustment, they are increased to the value of the previous buckets to enforce monotonicity. The latter is evidence for an actual issue with the input data and is therefore flagged with an informational annotation reading `input to histogram_quantile needed to be fixed for monotonicity`. If you encounter this annotation, you should find and remove the source of the invalid data.

`histogram_stddev()` and `histogram_stdvar()`

Both functions only act on native histograms, which are an experimental feature. The behavior of these functions may change in future versions of Prometheus, including their removal from PromQL.

`histogram_stddev(v instant-vector)` returns the estimated standard deviation of observations in a native histogram, based on the geometric mean of the buckets where the observations lie. Samples that are not native histograms are ignored and do not show up in the returned vector.

Similarly, `histogram_stdvar(v instant-vector)` returns the estimated standard variance of observations in a native histogram.

`double_exponential_smoothing()`

This function has to be enabled via the feature flag **`--enable-feature=promql-experimental-functions`**.

`double_exponential_smoothing(v range-vector, sf scalar, tf scalar)` produces a smoothed value for time series based on the range in `v`. The lower the smoothing factor `sf`, the more importance is given to old data. The higher the trend factor `tf`, the more trends in the data is considered. Both `sf` and `tf` must be between 0 and 1. For additional details, refer to NIST Engineering Statistics Handbook. In Prometheus V2 this function was called `holt_winters`. This caused confusion since the Holt-Winters method usually refers to triple exponential smoothing. Double exponential smoothing as implemented here is also referred to as "Holt Linear".

`double_exponential_smoothing` should only be used with gauges.

hour()

`hour(v=vector(time()) instant-vector)` returns the hour of the day for each of the given times in UTC. Returned values are from 0 to 23.

idelta()

`idelta(v range-vector)` calculates the difference between the last two samples in the range vector `v`, returning an instant vector with the given deltas and equivalent labels.

`idelta` should only be used with gauges.

increase()

`increase(v range-vector)` calculates the increase in the time series in the range vector. Breaks in monotonicity (such as counter resets due to target restarts) are automatically adjusted for. The increase is extrapolated to cover the full time range as specified in the range vector selector, so that it is possible to get a non-integer result even if a counter increases only by integer increments.

The following example expression returns the number of HTTP requests as measured over the last 5 minutes, per time series in the range vector:

`increase(http_requests_total{job="api-server"}[5m])`
`increase` acts on native histograms by calculating a new histogram where each component (sum and count of observations, buckets) is the increase between the respective component in the first and last native histogram in `v`. However, each element in `v` that contains a mix of float and native histogram samples within the range, will be missing from the result vector.

`increase` should only be used with counters and native histograms where the components behave like counters. It is syntactic sugar for `rate(v)` multiplied by the number of seconds under the specified time range window, and should be used primarily for human readability. Use `rate` in recording rules so that increases are tracked consistently on a per-second basis.

`info()` (experimental)

The *`info`* function is an experiment to improve UX around including labels from info metrics. The behavior of this function may change in future versions of Prometheus, including its removal from PromQL. *`info`* has to be enabled via the feature flag *`--enable-feature=promql-experimental-functions`*.

`info(v instant-vector, [data-label-selector instant-vector])` finds, for each time series in `v`, all info series with matching identifying labels (more on this later), and adds the union of their data (i.e., non-identifying) labels to the time series. The second argument `data-label-selector` is optional. It is not a real instant vector, but uses a subset of its syntax. It must start and end with curly braces (`{ ... }`) and may only contain label matchers. The label matchers are used to constrain which info series to consider and which data labels to add to `v`.

Identifying labels of an info series are the subset of labels that uniquely identify the info series. The remaining labels are considered data labels (also called non-identifying). (Note that Prometheus's concept of time series identity always includes all the labels. For the sake of the `info` function, we “logically” define info series identity in a different way than in the conventional Prometheus view.) The identifying labels of an info series are used to join it to regular (non-info) series, i.e. those series that have the

same labels as the identifying labels of the info series. The data labels, which are the ones added to the regular series by the `info` function, effectively encode metadata key value pairs. (This implies that a change in the data labels in the conventional Prometheus view constitutes the end of one info series and the beginning of a new info series, while the "logical" view of the `info` function is that the same info series continues to exist, just with different "data".)

The conventional approach of adding data labels is sometimes called a "join query", as illustrated by the following example:

```
  rate(http_server_request_duration_seconds_count[2m])
* on (job, instance) group_left (k8s_cluster_name)
  target_info
```

The core of the query is the expression `rate(http_server_request_duration_seconds_count[2m])`. But to add data labels from an info metric, the user has to use elaborate (and not very obvious) syntax to specify which info metric to use (`target_info`), what the identifying labels are (`on (job, instance)`), and which data labels to add (`group_left (k8s_cluster_name)`).

This query is not only verbose and hard to write, it might also run into an "identity crisis": If any of the data labels of `target_info` changes, Prometheus sees that as a change of series (as alluded to above, Prometheus just has no native concept of non-identifying labels). If the old `target_info` series is not properly marked as stale (which can happen with certain ingestion paths), the query above will fail for up to 5m (the lookback delta) because it will find a conflicting match with both the old and the new version of `target_info`.

The `info` function not only resolves this conflict in favor of the newer series, it also simplifies the syntax because it knows about the available info series and what their identifying labels are. The example query looks like this with the `info` function:

```
info(
  rate(http_server_request_duration_seconds_count[2m]),
  {k8s_cluster_name=~".+"}
)
```

The common case of adding all data labels can be achieved by omitting the 2nd argument of the `info` function entirely, simplifying the example even more:

```
info(rate(http_server_request_duration_seconds_count[2m]))
```

While `info` normally automatically finds all matching info series, it's possible to restrict them by providing a `__name__` label matcher, e.g. `{__name__="target_info"}`.

Limitations

In its current iteration, `info` defaults to considering only info series with the name `target_info`. It also assumes that the identifying info series labels are `instance` and `job`. `info` does support other info series names however, through `__name__` label matchers. E.g., one can explicitly say to consider both `target_info` and `build_info` as follows: `{__name__=~"(target|build)_info"}`. However, the identifying labels always have to be `instance` and `job`.

These limitations are partially defeating the purpose of the `info` function. At the current stage, this is an experiment to find out how useful the approach turns out to be in practice. A final version of the `info` function will indeed consider all matching info series and with their appropriate identifying labels.

`irate()`

`irate(v range-vector)` calculates the per-second instant rate of increase of the time series in the range vector. This is based on the last two data points. Breaks in monotonicity (such as counter resets due to target restarts) are automatically adjusted for.

The following example expression returns the per-second rate of HTTP requests looking up to 5 minutes back for the two most recent data points, per time series in the range vector:

```
irate(http_requests_total{job="api-server"}[5m])
```

`irate` should only be used when graphing volatile, fast-moving counters. Use `rate` for alerts and slow-moving counters, as brief changes in the rate can reset the `FOR` clause and graphs consisting entirely of rare spikes are hard to read.

Note that when combining `irate()` with an aggregation operator (e.g. `sum()`) or a function aggregating over time (any function ending in `_over_time`), always take a `irate()` first, then aggregate. Otherwise `irate()` cannot detect counter resets when your target restarts.

`label_join()`

For each timeseries in `v`, `label_join(v instant-vector, dst_label string, separator string, src_label_1 string, src_label_2 string, ...)` joins all the values of all the `src_labels` using `separator` and returns the timeseries with the label `dst_label` containing the joined value. There can be any number of `src_labels` in this function.

`label_join` acts on float and histogram samples in the same way.

This example will return a vector with each time series having a `foo` label with the value `a,b,c` added to it:

```
label_join(up{job="api-server",src1="a",src2="b",src3="c"},
"foo", ",", "src1", "src2", "src3")
```

`label_replace()`

For each timeseries in `v`, `label_replace(v instant-vector, dst_label string, replacement string, src_label string, regex string)` matches the regular expression `regex` against the value of the label `src_label`. If it matches, the value of the label `dst_label` in the returned timeseries will be the expansion of `replacement`, together with the original labels in the input. Capturing groups in the regular expression can be referenced with `$1`, `$2`, etc. Named capturing groups in the regular expression can be referenced with `$name` (where `name` is the capturing group name). If the regular expression doesn't match then the timeseries is returned unchanged.

`label_replace` acts on float and histogram samples in the same way.

This example will return timeseries with the values `a:c` at label `service` and `a` at label `foo`:

```
label_replace(up{job="api-server",service="a:c"}, "foo",
"$1", "service", "(.*):.*")
```
This second example has the same effect than the first example, and illustrates use of named capturing groups:
```
label_replace(up{job="api-server",service="a:c"},
"foo", "$name", "service", "(?P<name>.*):(?
P<version>.*)")
```

ln()

`ln(v instant-vector)` calculates the natural logarithm for all elements in `v`. Special cases are:

- `ln(+Inf) = +Inf`
- `ln(0) = -Inf`
- `ln(x < 0) = NaN`
- `ln(NaN) = NaN`

log2()

`log2(v instant-vector)` calculates the binary logarithm for all elements in `v`. The special cases are equivalent to those in `ln`.

log10()

`log10(v instant-vector)` calculates the decimal logarithm for all elements in `v`. The special cases are equivalent to those in `ln`.

minute()

`minute(v=vector(time()) instant-vector)` returns the minute of the hour for each of the given times in UTC. Returned values are from 0 to 59.

month()

`month(v=vector(time()) instant-vector)` returns the month of the year for each of the given times in UTC. Returned values are from 1 to 12, where 1 means January etc.

predict_linear()

`predict_linear(v range-vector, t scalar)` predicts the value of time series `t` seconds from now, based on the range vector `v`,

using simple linear regression. The range vector must have at least two samples in order to perform the calculation. When `+Inf` or `-Inf` are found in the range vector, the slope and offset value calculated will be `NaN`.

`predict_linear` should only be used with gauges.

`rate()`

`rate(v range-vector)` calculates the per-second average rate of increase of the time series in the range vector. Breaks in monotonicity (such as counter resets due to target restarts) are automatically adjusted for. Also, the calculation extrapolates to the ends of the time range, allowing for missed scrapes or imperfect alignment of scrape cycles with the range's time period.

The following example expression returns the per-second rate of HTTP requests as measured over the last 5 minutes, per time series in the range vector:

```
rate(http_requests_total{job="api-server"}[5m])
```

`rate` acts on native histograms by calculating a new histogram where each component (sum and count of observations, buckets) is the rate of increase between the respective component in the first and last native histogram in `v`. However, each element in `v` that contains a mix of float and native histogram samples within the range, will be missing from the result vector.

`rate` should only be used with counters and native histograms where the components behave like counters. It is best suited for alerting, and for graphing of slow-moving counters.

Note that when combining `rate()` with an aggregation operator (e.g. `sum()`) or a function aggregating over time (any function ending in `_over_time`), always take a `rate()` first, then aggregate. Otherwise `rate()` cannot detect counter resets when your target restarts.

`resets()`

For each input time series, `resets(v range-vector)` returns the number of counter resets within the provided time range as an

instant vector. Any decrease in the value between two consecutive float samples is interpreted as a counter reset. A reset in a native histogram is detected in a more complex way: Any decrease in any bucket, including the zero bucket, or in the count of observation constitutes a counter reset, but also the disappearance of any previously populated bucket, an increase in bucket resolution, or a decrease of the zero-bucket width.

`resets` should only be used with counters and counter-like native histograms.

If the range vector contains a mix of float and histogram samples for the same series, counter resets are detected separately and their numbers added up. The change from a float to a histogram sample is not considered a counter reset. Each float sample is compared to the next float sample, and each histogram is comprared to the next histogram.

round()

`round(v instant-vector, to_nearest=1 scalar)` rounds the sample values of all elements in `v` to the nearest integer. Ties are resolved by rounding up. The optional `to_nearest` argument allows specifying the nearest multiple to which the sample values should be rounded. This multiple may also be a fraction.

scalar()

Given a single-element input vector, `scalar(v instant-vector)` returns the sample value of that single element as a scalar. If the input vector does not have exactly one element, `scalar` will return `NaN`.

sgn()

`sgn(v instant-vector)` returns a vector with all sample values converted to their sign, defined as this: 1 if v is positive, -1 if v is negative and 0 if v is equal to zero.

sort()

`sort(v instant-vector)` returns vector elements sorted by their sample values, in ascending order. Native histograms are sorted by their sum of observations.

Please note that `sort` only affects the results of instant queries, as range query results always have a fixed output ordering.

`sort_desc()`

Same as `sort`, but sorts in descending order.

Like `sort`, `sort_desc` only affects the results of instant queries, as range query results always have a fixed output ordering.

`sort_by_label()`

This function has to be enabled via the feature flag **`--enable-feature=promql-experimental-functions`**.

`sort_by_label(v instant-vector, label string, ...)` returns vector elements sorted by the values of the given labels in ascending order. In case these label values are equal, elements are sorted by their full label sets.

Please note that the sort by label functions only affect the results of instant queries, as range query results always have a fixed output ordering.

This function uses natural sort order.

`sort_by_label_desc()`

This function has to be enabled via the feature flag **`--enable-feature=promql-experimental-functions`**.

Same as `sort_by_label`, but sorts in descending order.

Please note that the sort by label functions only affect the results of instant queries, as range query results always have a fixed output ordering.

This function uses natural sort order.

sqrt()

`sqrt(v instant-vector)` calculates the square root of all elements in `v`.

time()

`time()` returns the number of seconds since January 1, 1970 UTC. Note that this does not actually return the current time, but the time at which the expression is to be evaluated.

timestamp()

`timestamp(v instant-vector)` returns the timestamp of each of the samples of the given vector as the number of seconds since January 1, 1970 UTC. It also works with histogram samples.

vector()

`vector(s scalar)` returns the scalar `s` as a vector with no labels.

year()

`year(v=vector(time()) instant-vector)` returns the year for each of the given times in UTC.

<aggregation>_over_time()

The following functions allow aggregating each series of a given range vector over time and return an instant vector with per-series aggregation results:

- `avg_over_time(range-vector)`: the average value of all points in the specified interval.
- `min_over_time(range-vector)`: the minimum value of all points in the specified interval.
- `max_over_time(range-vector)`: the maximum value of all points in the specified interval.
- `sum_over_time(range-vector)`: the sum of all values in the specified interval.
- `count_over_time(range-vector)`: the count of all values in the specified interval.

- `quantile_over_time(scalar, range-vector)`: the φ-quantile (0 ≤ φ ≤ 1) of the values in the specified interval.
- `stddev_over_time(range-vector)`: the population standard deviation of the values in the specified interval.
- `stdvar_over_time(range-vector)`: the population standard variance of the values in the specified interval.
- `last_over_time(range-vector)`: the most recent point value in the specified interval.
- `present_over_time(range-vector)`: the value 1 for any series in the specified interval.

If the feature flag `--enable-feature=promql-experimental-functions` is set, the following additional functions are available:

- `mad_over_time(range-vector)`: the median absolute deviation of all points in the specified interval.

Note that all values in the specified interval have the same weight in the aggregation even if the values are not equally spaced throughout the interval.

`avg_over_time`, `sum_over_time`, `count_over_time`, `last_over_time`, and `present_over_time` handle native histograms as expected. All other functions ignore histogram samples.

Trigonometric Functions

The trigonometric functions work in radians:

- `acos(v instant-vector)`: calculates the arccosine of all elements in `v` (special cases).
- `acosh(v instant-vector)`: calculates the inverse hyperbolic cosine of all elements in `v` (special cases).
- `asin(v instant-vector)`: calculates the arcsine of all elements in `v` (special cases).
- `asinh(v instant-vector)`: calculates the inverse hyperbolic sine of all elements in `v` (special cases).
- `atan(v instant-vector)`: calculates the arctangent of all elements in `v` (special cases).
- `atanh(v instant-vector)`: calculates the inverse hyperbolic tangent of all elements in `v` (special cases).

- `cos(v instant-vector)`: calculates the cosine of all elements in `v` (special cases).
- `cosh(v instant-vector)`: calculates the hyperbolic cosine of all elements in `v` (special cases).
- `sin(v instant-vector)`: calculates the sine of all elements in `v` (special cases).
- `sinh(v instant-vector)`: calculates the hyperbolic sine of all elements in `v` (special cases).
- `tan(v instant-vector)`: calculates the tangent of all elements in `v` (special cases).
- `tanh(v instant-vector)`: calculates the hyperbolic tangent of all elements in `v` (special cases).

The following are useful for converting between degrees and radians:

- `deg(v instant-vector)`: converts radians to degrees for all elements in `v`.
- `pi()`: returns pi.
- `rad(v instant-vector)`: converts degrees to radians for all elements in `v`.

QUERY EXAMPLES

Simple time series selection

Return all time series with the metric `http_requests_total`:

```
http_requests_total
```

Return all time series with the metric `http_requests_total` and the given `job` and `handler` labels:

```
http_requests_total{job="apiserver", handler="/api/comments"}
```

Return a whole range of time (in this case 5 minutes up to the query time) for the same vector, making it a range vector:

```
http_requests_total{job="apiserver", handler="/api/comments"}[5m]
```

Note that an expression resulting in a range vector cannot be graphed directly, but viewed in the tabular ("Console") view of the expression browser.

Using regular expressions, you could select time series only for jobs whose name match a certain pattern, in this case, all jobs that end with `server`:

```
http_requests_total{job=~".*server"}
```

To select all HTTP status codes except 4xx ones, you could run:

```
http_requests_total{status!~"4.."}
```

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:])
```

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):

```
(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)
```
If we are exploring some metrics for their labels, to e.g. be able to aggregate over some of them, we could use the following:

```
limitk(10, app_foo_metric_bar)
```
Alternatively, if we wanted the returned timeseries to be more evenly sampled, we could use the following to get approximately 10% of them:

```
limit_ratio(0.1, app_foo_metric_bar)
```

HTTP API

The current stable HTTP API is reachable under `/api/v1` on a Prometheus server. Any non-breaking additions will be added under that endpoint.

Format overview

The API response format is JSON. Every successful API request returns a `2xx` status code.

Invalid requests that reach the API handlers return a JSON error object and one of the following HTTP response codes:

- `400 Bad Request` when parameters are missing or incorrect.
- `422 Unprocessable Entity` when an expression can't be executed (RFC4918).
- `503 Service Unavailable` when queries time out or abort.

Other non-`2xx` codes may be returned for errors occurring before the API endpoint is reached.

An array of warnings may be returned if there are errors that do not inhibit the request execution. An additional array of info-level annotations may be returned for potential query issues that may or may not be false positives. All of the data that was successfully collected will be returned in the data field.

The JSON response envelope format is as follows:

```
{
  "status": "success" | "error",
  "data": <data>,

  // Only set if status is "error". The data field may still hold
  // additional data.
  "errorType": "<string>",
  "error": "<string>",

  // Only set if there were warnings while executing the request.
  // There will still be data in the data field.
  "warnings": ["<string>"],
```

```
  // Only set if there were info-level annnotations while
executing the request.
  "infos": ["<string>"]
}
```
Generic placeholders are defined as follows:

- `<rfc3339 | unix_timestamp>`: Input timestamps may be provided either in RFC3339 format or as a Unix timestamp in seconds, with optional decimal places for sub-second precision. Output timestamps are always represented as Unix timestamps in seconds.
- `<series_selector>`: Prometheus time series selectors like `http_requests_total` or `http_requests_total{method=~"(GET|POST)"}` and need to be URL-encoded.
- `<duration>`: the subset of Prometheus float literals using time units. For example, `5m` refers to a duration of 5 minutes.
- `<bool>`: boolean values (strings `true` and `false`).

Note: Names of query parameters that may be repeated end with `[]`.

Expression queries

Query language expressions may be evaluated at a single instant or over a range of time. The sections below describe the API endpoints for each type of expression query.

Instant queries

The following endpoint evaluates an instant query at a single point in time:

```
GET /api/v1/query
POST /api/v1/query
```
URL query parameters:

- `query=<string>`: Prometheus expression query string.
- `time=<rfc3339 | unix_timestamp>`: Evaluation timestamp. Optional.
- `timeout=<duration>`: Evaluation timeout. Optional. Defaults to and is capped by the value of the `-query.timeout` flag.

- `limit=<number>`: Maximum number of returned series. Doesn't affect scalars or strings but truncates the number of series for matrices and vectors. Optional. 0 means disabled.

The current server time is used if the `time` parameter is omitted.

You can URL-encode these parameters directly in the request body by using the `POST` method and `Content-Type: application/x-www-form-urlencoded` header. This is useful when specifying a large query that may breach server-side URL character limits.

The `data` section of the query result has the following format:

```
{
  "resultType": "matrix" | "vector" | "scalar" | "string",
  "result": <value>
}
```

`<value>` refers to the query result data, which has varying formats depending on the `resultType`. See the expression query result formats.

The following example evaluates the expression `up` at the time `2015-07-01T20:10:51.781Z`:

```
$ curl 'http://localhost:9090/api/v1/query?
query=up&time=2015-07-01T20:10:51.781Z'
{
   "status" : "success",
   "data" : {
      "resultType" : "vector",
      "result" : [
         {
            "metric" : {
               "__name__" : "up",
               "job" : "prometheus",
               "instance" : "localhost:9090"
            },
            "value": [ 1435781451.781, "1" ]
         },
         {
            "metric" : {
               "__name__" : "up",
               "job" : "node",
               "instance" : "localhost:9100"
            },
            "value" : [ 1435781451.781, "0" ]
```

```
          }
        ]
    }
}
```

Range queries

The following endpoint evaluates an expression query over a range of time:

```
GET /api/v1/query_range
POST /api/v1/query_range
```

URL query parameters:

- `query=<string>`: Prometheus expression query string.
- `start=<rfc3339 | unix_timestamp>`: Start timestamp, inclusive.
- `end=<rfc3339 | unix_timestamp>`: End timestamp, inclusive.
- `step=<duration | float>`: Query resolution step width in `duration` format or float number of seconds.
- `timeout=<duration>`: Evaluation timeout. Optional. Defaults to and is capped by the value of the `-query.timeout` flag.
- `limit=<number>`: Maximum number of returned series. Optional. 0 means disabled.

You can URL-encode these parameters directly in the request body by using the `POST` method and `Content-Type: application/x-www-form-urlencoded` header. This is useful when specifying a large query that may breach server-side URL character limits.

The `data` section of the query result has the following format:

```
{
  "resultType": "matrix",
  "result": <value>
}
```

For the format of the `<value>` placeholder, see the range-vector result format.

The following example evaluates the expression `up` over a 30-second range with a query resolution of 15 seconds.

```
$ curl 'http://localhost:9090/api/v1/query_range?
query=up&start=2015-07-01T20:10:30.781Z&end=2015-07-01T20:11:
00.781Z&step=15s'
{
   "status" : "success",
   "data" : {
      "resultType" : "matrix",
      "result" : [
         {
            "metric" : {
               "__name__" : "up",
               "job" : "prometheus",
               "instance" : "localhost:9090"
            },
            "values" : [
               [ 1435781430.781, "1" ],
               [ 1435781445.781, "1" ],
               [ 1435781460.781, "1" ]
            ]
         },
         {
            "metric" : {
               "__name__" : "up",
               "job" : "node",
               "instance" : "localhost:9091"
            },
            "values" : [
               [ 1435781430.781, "0" ],
               [ 1435781445.781, "0" ],
               [ 1435781460.781, "1" ]
            ]
         }
      ]
   }
}
```

Formatting query expressions

The following endpoint formats a PromQL expression in a prettified way:

```
GET /api/v1/format_query
POST /api/v1/format_query
```

URL query parameters:

- `query=<string>`: Prometheus expression query string.

You can URL-encode these parameters directly in the request body by using the `POST` method and `Content-Type: application/x-www-form-urlencoded` header. This is useful when specifying a large query that may breach server-side URL character limits.

The `data` section of the query result is a string containing the formatted query expression. Note that any comments are removed in the formatted string.

The following example formats the expression `foo/bar`:

```
$ curl 'http://localhost:9090/api/v1/format_query?query=foo/bar'
{
   "status" : "success",
   "data" : "foo / bar"
}
```

Parsing a PromQL expressions into a abstract syntax tree (AST)

This endpoint is experimental and might change in the future. It is currently only meant to be used by Prometheus' own web UI, and the endpoint name and exact format returned may change from one Prometheus version to another. It may also be removed again in case it is no longer needed by the UI.

The following endpoint parses a PromQL expression and returns it as a JSON-formatted AST (abstract syntax tree) representation:

```
GET /api/v1/parse_query
POST /api/v1/parse_query
```

URL query parameters:

- `query=<string>`: Prometheus expression query string.

You can URL-encode these parameters directly in the request body by using the `POST` method and `Content-Type: application/x-www-form-urlencoded` header. This is useful when specifying a large query that may breach server-side URL character limits.

The `data` section of the query result is a string containing the AST of the parsed query expression.

The following example parses the expression `foo/bar`:

```
$ curl 'http://localhost:9090/api/v1/parse_query?query=foo/
bar'
{
   "data" : {
      "bool" : false,
      "lhs" : {
         "matchers" : [
            {
               "name" : "__name__",
               "type" : "=",
               "value" : "foo"
            }
         ],
         "name" : "foo",
         "offset" : 0,
         "startOrEnd" : null,
         "timestamp" : null,
         "type" : "vectorSelector"
      },
      "matching" : {
         "card" : "one-to-one",
         "include" : [],
         "labels" : [],
         "on" : false
      },
      "op" : "/",
      "rhs" : {
         "matchers" : [
            {
               "name" : "__name__",
               "type" : "=",
               "value" : "bar"
            }
         ],
         "name" : "bar",
         "offset" : 0,
         "startOrEnd" : null,
         "timestamp" : null,
         "type" : "vectorSelector"
      },
      "type" : "binaryExpr"
   },
   "status" : "success"
}
```

Querying metadata

Prometheus offers a set of API endpoints to query metadata about series and their labels.

NOTE: These API endpoints may return metadata for series for which there is no sample within the selected time range, and/or for series whose samples have been marked as deleted via the deletion API endpoint. The exact extent of additionally returned series metadata is an implementation detail that may change in the future.

Finding series by label matchers

The following endpoint returns the list of time series that match a certain label set.

```
GET /api/v1/series
POST /api/v1/series
```

URL query parameters:

- `match[]=<series_selector>`: Repeated series selector argument that selects the series to return. At least one `match[]` argument must be provided.
- `start=<rfc3339 | unix_timestamp>`: Start timestamp.
- `end=<rfc3339 | unix_timestamp>`: End timestamp.
- `limit=<number>`: Maximum number of returned series. Optional. 0 means disabled.

You can URL-encode these parameters directly in the request body by using the `POST` method and `Content-Type: application/x-www-form-urlencoded` header. This is useful when specifying a large or dynamic number of series selectors that may breach server-side URL character limits.

The `data` section of the query result consists of a list of objects that contain the label name/value pairs which identify each series.

The following example returns all series that match either of the selectors `up` or `process_start_time_seconds{job="prometheus"}`:

```
$ curl -g 'http://localhost:9090/api/v1/series?' --data-urlencode 'match[]=up' --data-urlencode 'match[]=process_start_time_seconds{job="prometheus"}'
{
```

```
   "status" : "success",
   "data" : [
      {
         "__name__" : "up",
         "job" : "prometheus",
         "instance" : "localhost:9090"
      },
      {
         "__name__" : "up",
         "job" : "node",
         "instance" : "localhost:9091"
      },
      {
         "__name__" : "process_start_time_seconds",
         "job" : "prometheus",
         "instance" : "localhost:9090"
      }
   ]
}
```

Getting label names

The following endpoint returns a list of label names:

```
GET /api/v1/labels
POST /api/v1/labels
```

URL query parameters:

- `start=<rfc3339 | unix_timestamp>`: Start timestamp. Optional.
- `end=<rfc3339 | unix_timestamp>`: End timestamp. Optional.
- `match[]=<series_selector>`: Repeated series selector argument that selects the series from which to read the label names. Optional.
- `limit=<number>`: Maximum number of returned series. Optional. 0 means disabled.

The `data` section of the JSON response is a list of string label names.

Here is an example.

```
$ curl 'localhost:9090/api/v1/labels'
{
    "status": "success",
    "data": [
```

```
        "__name__",
        "call",
        "code",
        "config",
        "dialer_name",
        "endpoint",
        "event",
        "goversion",
        "handler",
        "instance",
        "interval",
        "job",
        "le",
        "listener_name",
        "name",
        "quantile",
        "reason",
        "role",
        "scrape_job",
        "slice",
        "version"
    ]
}
```

Querying label values

The following endpoint returns a list of label values for a provided label name:

```
GET /api/v1/label/<label_name>/values
```

URL query parameters:

- `start=<rfc3339 | unix_timestamp>`: Start timestamp. Optional.
- `end=<rfc3339 | unix_timestamp>`: End timestamp. Optional.
- `match[]=<series_selector>`: Repeated series selector argument that selects the series from which to read the label values. Optional.
- `limit=<number>`: Maximum number of returned series. Optional. 0 means disabled.

The `data` section of the JSON response is a list of string label values.

This example queries for all label values for the `http_status_code` label:

```
$ curl http://localhost:9090/api/v1/label/http_status_code/
values
{
   "status" : "success",
   "data" : [
      "200",
      "504"
   ]
}
```

Label names can optionally be encoded using the Values Escaping method, and is necessary if a name includes the `/` character. To encode a name in this way:

- Prepend the label with `U__`.
- Letters, numbers, and colons appear as-is.
- Convert single underscores to double underscores.
- For all other characters, use the UTF-8 codepoint as a hex integer, surrounded by underscores. So becomes `_20_` and a `.` becomes `_2e_`.

More information about text escaping can be found in the original UTF-8 Proposal document.

This example queries for all label values for the `http.status_code` label:

```
$ curl http://localhost:9090/api/v1/label/
U__http_2e_status_code/values
{
   "status" : "success",
   "data" : [
      "200",
      "404"
   ]
}
```

Querying exemplars

This is experimental and might change in the future. The following endpoint returns a list of exemplars for a valid PromQL query for a specific time range:

```
GET /api/v1/query_exemplars
POST /api/v1/query_exemplars
```

URL query parameters:

- `query=<string>`: Prometheus expression query string.
- `start=<rfc3339 | unix_timestamp>`: Start timestamp.
- `end=<rfc3339 | unix_timestamp>`: End timestamp.

```
$ curl -g 'http://localhost:9090/api/v1/query_exemplars?
query=test_exemplar_metric_total&start=2020-09-14T15:22:25.47
9Z&end=2020-09-14T15:23:25.479Z'
{
    "status": "success",
    "data": [
        {
            "seriesLabels": {
                "__name__": "test_exemplar_metric_total",
                "instance": "localhost:8090",
                "job": "prometheus",
                "service": "bar"
            },
            "exemplars": [
                {
                    "labels": {
                        "trace_id": "EpTxMJ40fUus7aGY"
                    },
                    "value": "6",
                    "timestamp": 1600096945.479
                }
            ]
        },
        {
            "seriesLabels": {
                "__name__": "test_exemplar_metric_total",
                "instance": "localhost:8090",
                "job": "prometheus",
                "service": "foo"
            },
            "exemplars": [
                {
                    "labels": {
                        "trace_id": "Olp9XHlq763ccsfa"
                    },
                    "value": "19",
                    "timestamp": 1600096955.479
                },
                {
                    "labels": {
                        "trace_id": "hCtjygkIHwAN9vs4"
                    },
                    "value": "20",
                    "timestamp": 1600096965.489
                }
```

```
            ]
        }
    ]
}
```

Expression query result formats

Expression queries may return the following response values in the `result` property of the `data` section. `<sample_value>` placeholders are numeric sample values. JSON does not support special float values such as `NaN`, `Inf`, and `-Inf`, so sample values are transferred as quoted JSON strings rather than raw numbers.

The keys `"histogram"` and `"histograms"` only show up if the experimental native histograms are present in the response. Their placeholder `<histogram>` is explained in detail in its own section below.

Range vectors

Range vectors are returned as result type `matrix`. The corresponding `result` property has the following format:

```
[
  {
    "metric": { "<label_name>": "<label_value>", ... },
    "values": [ [ <unix_time>, "<sample_value>" ], ... ],
    "histograms": [ [ <unix_time>, <histogram> ], ... ]
  },
  ...
]
```

Each series could have the `"values"` key, or the `"histograms"` key, or both. For a given timestamp, there will only be one sample of either float or histogram type.

Series are returned sorted by `metric`. Functions such as `sort` and `sort_by_label` have no effect for range vectors.

Instant vectors

Instant vectors are returned as result type `vector`. The corresponding `result` property has the following format:

```
[
  {
```

```
    "metric": { "<label_name>": "<label_value>", ... },
    "value": [ <unix_time>, "<sample_value>" ],
    "histogram": [ <unix_time>, <histogram> ]
  },
  ...
]
```

Each series could have the `"value"` key, or the `"histogram"` key, but not both.

Series are not guaranteed to be returned in any particular order unless a function such as `sort` or `sort_by_label` is used.

Scalars

Scalar results are returned as result type `scalar`. The corresponding `result` property has the following format:

```
[ <unix_time>, "<scalar_value>" ]
```

Strings

String results are returned as result type `string`. The corresponding `result` property has the following format:

```
[ <unix_time>, "<string_value>" ]
```

Native histograms

The `<histogram>` placeholder used above is formatted as follows.

Note that native histograms are an experimental feature, and the format below might still change.

```
{
  "count": "<count_of_observations>",
  "sum": "<sum_of_observations>",
  "buckets": [ [ <boundary_rule>, "<left_boundary>",
"<right_boundary>", "<count_in_bucket>" ], ... ]
}
```

The `<boundary_rule>` placeholder is an integer between 0 and 3 with the following meaning:

- 0: “open left” (left boundary is exclusive, right boundary in inclusive)
- 1: “open right” (left boundary is inclusive, right boundary in exclusive)

- 2: “open both” (both boundaries are exclusive)
- 3: “closed both” (both boundaries are inclusive)

Note that with the currently implemented bucket schemas, positive buckets are “open left”, negative buckets are “open right”, and the zero bucket (with a negative left boundary and a positive right boundary) is “closed both”.

Targets

The following endpoint returns an overview of the current state of the Prometheus target discovery:

```
GET /api/v1/targets
```

Both the active and dropped targets are part of the response by default. Dropped targets are subject to `keep_dropped_targets` limit, if set. `labels` represents the label set after relabeling has occurred. `discoveredLabels` represent the unmodified labels retrieved during service discovery before relabeling has occurred.

```
$ curl http://localhost:9090/api/v1/targets
{
  "status": "success",
  "data": {
    "activeTargets": [
      {
        "discoveredLabels": {
          "__address__": "127.0.0.1:9090",
          "__metrics_path__": "/metrics",
          "__scheme__": "http",
          "job": "prometheus"
        },
        "labels": {
          "instance": "127.0.0.1:9090",
          "job": "prometheus"
        },
        "scrapePool": "prometheus",
        "scrapeUrl": "http://127.0.0.1:9090/metrics",
        "globalUrl": "http://example-prometheus:9090/
metrics",
        "lastError": "",
        "lastScrape": "2017-01-17T15:07:44.723715405+01:00",
        "lastScrapeDuration": 0.050688943,
        "health": "up",
        "scrapeInterval": "1m",
        "scrapeTimeout": "10s"
```

```
      }
    ],
    "droppedTargets": [
      {
        "discoveredLabels": {
          "__address__": "127.0.0.1:9100",
          "__metrics_path__": "/metrics",
          "__scheme__": "http",
          "__scrape_interval__": "1m",
          "__scrape_timeout__": "10s",
          "job": "node"
        },
      }
    ]
  }
}
```

The `state` query parameter allows the caller to filter by active or dropped targets, (e.g., `state=active`, `state=dropped`, `state=any`). Note that an empty array is still returned for targets that are filtered out. Other values are ignored.

```
$ curl 'http://localhost:9090/api/v1/targets?state=active'
{
  "status": "success",
  "data": {
    "activeTargets": [
      {
        "discoveredLabels": {
          "__address__": "127.0.0.1:9090",
          "__metrics_path__": "/metrics",
          "__scheme__": "http",
          "job": "prometheus"
        },
        "labels": {
          "instance": "127.0.0.1:9090",
          "job": "prometheus"
        },
        "scrapePool": "prometheus",
        "scrapeUrl": "http://127.0.0.1:9090/metrics",
        "globalUrl": "http://example-prometheus:9090/
metrics",
        "lastError": "",
        "lastScrape": "2017-01-17T15:07:44.723715405+01:00",
        "lastScrapeDuration": 50688943,
        "health": "up"
      }
    ],
```

```
    "droppedTargets": []
  }
}
```

The `scrapePool` query parameter allows the caller to filter by scrape pool name.

```
$ curl 'http://localhost:9090/api/v1/targets?
scrapePool=node_exporter'
{
  "status": "success",
  "data": {
    "activeTargets": [
      {
        "discoveredLabels": {
          "__address__": "127.0.0.1:9091",
          "__metrics_path__": "/metrics",
          "__scheme__": "http",
          "job": "node_exporter"
        },
        "labels": {
          "instance": "127.0.0.1:9091",
          "job": "node_exporter"
        },
        "scrapePool": "node_exporter",
        "scrapeUrl": "http://127.0.0.1:9091/metrics",
        "globalUrl": "http://example-prometheus:9091/
metrics",
        "lastError": "",
        "lastScrape": "2017-01-17T15:07:44.723715405+01:00",
        "lastScrapeDuration": 50688943,
        "health": "up"
      }
    ],
    "droppedTargets": []
  }
}
```

Rules

The `/rules` API endpoint returns a list of alerting and recording rules that are currently loaded. In addition it returns the currently active alerts fired by the Prometheus instance of each alerting rule.

As the `/rules` endpoint is fairly new, it does not have the same stability guarantees as the overarching API v1.

```
GET /api/v1/rules
```

URL query parameters:

- `type=alert|record`: return only the alerting rules (e.g. `type=alert`) or the recording rules (e.g. `type=record`). When the parameter is absent or empty, no filtering is done.
- `rule_name[]=<string>`: only return rules with the given rule name. If the parameter is repeated, rules with any of the provided names are returned. If we've filtered out all the rules of a group, the group is not returned. When the parameter is absent or empty, no filtering is done.
- `rule_group[]=<string>`: only return rules with the given rule group name. If the parameter is repeated, rules with any of the provided rule group names are returned. When the parameter is absent or empty, no filtering is done.
- `file[]=<string>`: only return rules with the given filepath. If the parameter is repeated, rules with any of the provided filepaths are returned. When the parameter is absent or empty, no filtering is done.
- `exclude_alerts=<bool>`: only return rules, do not return active alerts.
- `match[]=<label_selector>`: only return rules that have configured labels that satisfy the label selectors. If the parameter is repeated, rules that match any of the sets of label selectors are returned. Note that matching is on the labels in the definition of each rule, not on the values after template expansion (for alerting rules). Optional.
- `group_limit=<number>`: The `group_limit` parameter allows you to specify a limit for the number of rule groups that is returned in a single response. If the total number of rule groups exceeds the specified `group_limit` value, the response will include a `groupNextToken` property. You can use the value of this `groupNextToken` property in subsequent requests in the `group_next_token` parameter to paginate over the remaining rule groups. The `groupNextToken` property will not be present in the final response, indicating that you have retrieved all the available rule groups. Please note that there are no guarantees regarding the consistency of the response if the rule groups are being modified during the pagination process.

- `group_next_token`: the pagination token that was returned in previous request when the `group_limit` property is set. The pagination token is used to iteratively paginate over a large number of rule groups. To use the `group_next_token` parameter, the `group_limit` parameter also need to be present. If a rule group that coincides with the next token is removed while you are paginating over the rule groups, a response with status code 400 will be returned.

```
$ curl http://localhost:9090/api/v1/rules

{
    "data": {
        "groups": [
            {
                "rules": [
                    {
                        "alerts": [
                            {
                                "activeAt":
"2018-07-04T20:27:12.60602144+02:00",
                                "annotations": {
                                    "summary": "High request
latency"
                                },
                                "labels": {
                                    "alertname":
"HighRequestLatency",
                                    "severity": "page"
                                },
                                "state": "firing",
                                "value": "1e+00"
                            }
                        ],
                        "annotations": {
                            "summary": "High request latency"
                        },
                        "duration": 600,
                        "health": "ok",
                        "labels": {
                            "severity": "page"
                        },
                        "name": "HighRequestLatency",
                        "query":
"job:request_latency_seconds:mean5m{job=\"myjob\"} > 0.5",
                        "type": "alerting"
                    },
                    {
```

```
                        "health": "ok",
                        "name":
"job:http_inprogress_requests:sum",
                        "query": "sum by (job)
(http_inprogress_requests)",
                        "type": "recording"
                    }
                ],
                "file": "/rules.yaml",
                "interval": 60,
                "limit": 0,
                "name": "example"
            }
        ]
    },
    "status": "success"
}
```

Alerts

The `/alerts` endpoint returns a list of all active alerts.

As the `/alerts` endpoint is fairly new, it does not have the same stability guarantees as the overarching API v1.

```
GET /api/v1/alerts
$ curl http://localhost:9090/api/v1/alerts

{
    "data": {
        "alerts": [
            {
                "activeAt":
"2018-07-04T20:27:12.60602144+02:00",
                "annotations": {},
                "labels": {
                    "alertname": "my-alert"
                },
                "state": "firing",
                "value": "1e+00"
            }
        ]
    },
    "status": "success"
}
```

Querying target metadata

The following endpoint returns metadata about metrics currently scraped from targets. This is experimental and might change in the future.

`GET /api/v1/targets/metadata`
URL query parameters:

- `match_target=<label_selectors>`: Label selectors that match targets by their label sets. All targets are selected if left empty.
- `metric=<string>`: A metric name to retrieve metadata for. All metric metadata is retrieved if left empty.
- `limit=<number>`: Maximum number of targets to match.

The `data` section of the query result consists of a list of objects that contain metric metadata and the target label set.

The following example returns all metadata entries for the `go_goroutines` metric from the first two targets with label `job="prometheus"`.

```
curl -G http://localhost:9091/api/v1/targets/metadata \
    --data-urlencode 'metric=go_goroutines' \
    --data-urlencode 'match_target={job="prometheus"}' \
    --data-urlencode 'limit=2'
{
  "status": "success",
  "data": [
    {
      "target": {
        "instance": "127.0.0.1:9090",
        "job": "prometheus"
      },
      "type": "gauge",
      "help": "Number of goroutines that currently exist.",
      "unit": ""
    },
    {
      "target": {
        "instance": "127.0.0.1:9091",
        "job": "prometheus"
      },
      "type": "gauge",
      "help": "Number of goroutines that currently exist.",
      "unit": ""
    }
```

```
  ]
}
```
The following example returns metadata for all metrics for all targets with label `instance="127.0.0.1:9090"`.

```
curl -G http://localhost:9091/api/v1/targets/metadata \
    --data-urlencode
'match_target={instance="127.0.0.1:9090"}'
{
  "status": "success",
  "data": [
    // ...
    {
      "target": {
        "instance": "127.0.0.1:9090",
        "job": "prometheus"
      },
      "metric":
"prometheus_treecache_zookeeper_failures_total",
      "type": "counter",
      "help": "The total number of ZooKeeper failures.",
      "unit": ""
    },
    {
      "target": {
        "instance": "127.0.0.1:9090",
        "job": "prometheus"
      },
      "metric": "prometheus_tsdb_reloads_total",
      "type": "counter",
      "help": "Number of times the database reloaded block
data from disk.",
      "unit": ""
    },
    // ...
  ]
}
```

Querying metric metadata

It returns metadata about metrics currently scraped from targets. However, it does not provide any target information. This is considered experimental and might change in the future.

`GET /api/v1/metadata`
URL query parameters:

- `limit=<number>`: Maximum number of metrics to return.
- `limit_per_metric=<number>`: Maximum number of metadata to return per metric.
- `metric=<string>`: A metric name to filter metadata for. All metric metadata is retrieved if left empty.

The `data` section of the query result consists of an object where each key is a metric name and each value is a list of unique metadata objects, as exposed for that metric name across all targets.

The following example returns two metrics. Note that the metric `http_requests_total` has more than one object in the list. At least one target has a value for `HELP` that do not match with the rest.

```
curl -G http://localhost:9090/api/v1/metadata?limit=2
```

```
{
  "status": "success",
  "data": {
    "cortex_ring_tokens": [
      {
        "type": "gauge",
        "help": "Number of tokens in the ring",
        "unit": ""
      }
    ],
    "http_requests_total": [
      {
        "type": "counter",
        "help": "Number of HTTP requests",
        "unit": ""
      },
      {
        "type": "counter",
        "help": "Amount of HTTP requests",
        "unit": ""
      }
    ]
  }
}
```

The following example returns only one metadata entry for each metric.

```
curl -G http://localhost:9090/api/v1/metadata?
limit_per_metric=1
```

```
{
  "status": "success",
  "data": {
    "cortex_ring_tokens": [
      {
        "type": "gauge",
        "help": "Number of tokens in the ring",
        "unit": ""
      }
    ],
    "http_requests_total": [
      {
        "type": "counter",
        "help": "Number of HTTP requests",
        "unit": ""
      }
    ]
  }
}
```

The following example returns metadata only for the metric `http_requests_total`.

```
curl -G http://localhost:9090/api/v1/metadata?
metric=http_requests_total
```

```
{
  "status": "success",
  "data": {
    "http_requests_total": [
      {
        "type": "counter",
        "help": "Number of HTTP requests",
        "unit": ""
      },
      {
        "type": "counter",
        "help": "Amount of HTTP requests",
        "unit": ""
      }
    ]
  }
}
```

Alertmanagers

The following endpoint returns an overview of the current state of the Prometheus alertmanager discovery:

```
GET /api/v1/alertmanagers
```

Both the active and dropped Alertmanagers are part of the response.

```
$ curl http://localhost:9090/api/v1/alertmanagers
{
  "status": "success",
  "data": {
    "activeAlertmanagers": [
      {
        "url": "http://127.0.0.1:9090/api/v1/alerts"
      }
    ],
    "droppedAlertmanagers": [
      {
        "url": "http://127.0.0.1:9093/api/v1/alerts"
      }
    ]
  }
}
```

Status

Following status endpoints expose current Prometheus configuration.

Config

The following endpoint returns currently loaded configuration file:

```
GET /api/v1/status/config
```

The config is returned as dumped YAML file. Due to limitation of the YAML library, YAML comments are not included.

```
$ curl http://localhost:9090/api/v1/status/config
{
  "status": "success",
  "data": {
    "yaml": "<content of the loaded config file in YAML>",
  }
}
```

Flags

The following endpoint returns flag values that Prometheus was configured with:

`GET /api/v1/status/flags`
All values are of the result type `string`.

```
$ curl http://localhost:9090/api/v1/status/flags
{
  "status": "success",
  "data": {
    "alertmanager.notification-queue-capacity": "10000",
    "alertmanager.timeout": "10s",
    "log.level": "info",
    "query.lookback-delta": "5m",
    "query.max-concurrency": "20",
    ...
  }
}
```
New in v2.2

Runtime Information

The following endpoint returns various runtime information properties about the Prometheus server:

`GET /api/v1/status/runtimeinfo`
The returned values are of different types, depending on the nature of the runtime property.

```
$ curl http://localhost:9090/api/v1/status/runtimeinfo
{
  "status": "success",
  "data": {
    "startTime": "2019-11-02T17:23:59.301361365+01:00",
    "CWD": "/",
    "hostname" : "DESKTOP-717H17Q",
    "serverTime": "2025-01-05T18:27:33Z",
    "reloadConfigSuccess": true,
    "lastConfigTime": "2019-11-02T17:23:59+01:00",
    "timeSeriesCount": 873,
    "corruptionCount": 0,
    "goroutineCount": 48,
    "GOMAXPROCS": 4,
    "GOGC": "",
    "GODEBUG": "",
    "storageRetention": "15d"
```

```
  }
}
```
NOTE: The exact returned runtime properties may change without notice between Prometheus versions.
New in v2.14

Build Information

The following endpoint returns various build information properties about the Prometheus server:

```
GET /api/v1/status/buildinfo
```
All values are of the result type `string`.

```
$ curl http://localhost:9090/api/v1/status/buildinfo
{
  "status": "success",
  "data": {
    "version": "2.13.1",
    "revision": "cb7cbad5f9a2823a622aaa668833ca04f50a0ea7",
    "branch": "master",
    "buildUser": "julius@desktop",
    "buildDate": "20191102-16:19:59",
    "goVersion": "go1.13.1"
  }
}
```
NOTE: The exact returned build properties may change without notice between Prometheus versions.
New in v2.14

TSDB Stats

The following endpoint returns various cardinality statistics about the Prometheus TSDB:

```
GET /api/v1/status/tsdb
```
URL query parameters:

- `limit=<number>`: Limit the number of returned items to a given number for each set of statistics. By default, 10 items are returned.

The `data` section of the query result consists of:

- headStats: This provides the following data about the head block of the TSDB:
  - numSeries: The number of series.
  - chunkCount: The number of chunks.
  - minTime: The current minimum timestamp in milliseconds.
  - maxTime: The current maximum timestamp in milliseconds.
- seriesCountByMetricName: This will provide a list of metrics names and their series count.
- labelValueCountByLabelName: This will provide a list of the label names and their value count.
- memoryInBytesByLabelName This will provide a list of the label names and memory used in bytes. Memory usage is calculated by adding the length of all values for a given label name.
- seriesCountByLabelPair This will provide a list of label value pairs and their series count.

```
$ curl http://localhost:9090/api/v1/status/tsdb
{
  "status": "success",
  "data": {
    "headStats": {
      "numSeries": 508,
      "chunkCount": 937,
      "minTime": 1591516800000,
      "maxTime": 1598896800143,
    },
    "seriesCountByMetricName": [
      {
        "name": "net_conntrack_dialer_conn_failed_total",
        "value": 20
      },
      {
        "name":
"prometheus_http_request_duration_seconds_bucket",
        "value": 20
      }
    ],
    "labelValueCountByLabelName": [
      {
        "name": "__name__",
        "value": 211
      },
      {
```

```
        "name": "event",
        "value": 3
      }
    ],
    "memoryInBytesByLabelName": [
      {
        "name": "__name__",
        "value": 8266
      },
      {
        "name": "instance",
        "value": 28
      }
    ],
    "seriesCountByLabelValuePair": [
      {
        "name": "job=prometheus",
        "value": 425
      },
      {
        "name": "instance=localhost:9090",
        "value": 425
      }
    ]
  }
}
```

New in v2.15

WAL Replay Stats

The following endpoint returns information about the WAL replay:

```
GET /api/v1/status/walreplay
```

- read: The number of segments replayed so far.
- total: The total number segments needed to be replayed.
- progress: The progress of the replay (0 - 100%).
- state: The state of the replay. Possible states:
  - waiting: Waiting for the replay to start.
  - in progress: The replay is in progress.
  - done: The replay has finished.

```
$ curl http://localhost:9090/api/v1/status/walreplay
{
  "status": "success",
  "data": {
    "min": 2,
    "max": 5,
```

```
    "current": 40,
    "state": "in progress"
  }
}
```

NOTE: This endpoint is available before the server has been marked ready and is updated in real time to facilitate monitoring the progress of the WAL replay.

New in v2.28

TSDB Admin APIs

These are APIs that expose database functionalities for the advanced user. These APIs are not enabled unless the `--web.enable-admin-api` is set.

Snapshot

Snapshot creates a snapshot of all current data into `snapshots/<datetime>-<rand>` under the TSDB's data directory and returns the directory as response. It will optionally skip snapshotting data that is only present in the head block, and which has not yet been compacted to disk.

```
POST /api/v1/admin/tsdb/snapshot
PUT /api/v1/admin/tsdb/snapshot
```

URL query parameters:

- `skip_head=<bool>`: Skip data present in the head block. Optional.

```
$ curl -XPOST http://localhost:9090/api/v1/admin/tsdb/snapshot
{
  "status": "success",
  "data": {
    "name": "20171210T211224Z-2be650b6d019eb54"
  }
}
```

The snapshot now exists at `<data-dir>/snapshots/20171210T211224Z-2be650b6d019eb54`

New in v2.1 and supports PUT from v2.9

Delete Series

DeleteSeries deletes data for a selection of series in a time range. The actual data still exists on disk and is cleaned up in future compactions or can be explicitly cleaned up by hitting the Clean Tombstones endpoint.

If successful, a `204` is returned.

```
POST /api/v1/admin/tsdb/delete_series
PUT /api/v1/admin/tsdb/delete_series
```

URL query parameters:

- `match[]=<series_selector>`: Repeated label matcher argument that selects the series to delete. At least one `match[]` argument must be provided.
- `start=<rfc3339 | unix_timestamp>`: Start timestamp. Optional and defaults to minimum possible time.
- `end=<rfc3339 | unix_timestamp>`: End timestamp. Optional and defaults to maximum possible time.

Not mentioning both start and end times would clear all the data for the matched series in the database.

Example:

```
$ curl -X POST \
  -g 'http://localhost:9090/api/v1/admin/tsdb/delete_series?
match[]=up&match[]=process_start_time_seconds{job="prometheus
"}'
```

NOTE: This endpoint marks samples from series as deleted, but will not necessarily prevent associated series metadata from still being returned in metadata queries for the affected time range (even after cleaning tombstones). The exact extent of metadata deletion is an implementation detail that may change in the future.

New in v2.1 and supports PUT from v2.9

Clean Tombstones

CleanTombstones removes the deleted data from disk and cleans up the existing tombstones. This can be used after deleting series to free up space.

If successful, a `204` is returned.

```
POST /api/v1/admin/tsdb/clean_tombstones
```

`PUT /api/v1/admin/tsdb/clean_tombstones`
This takes no parameters or body.

```
$ curl -XPOST http://localhost:9090/api/v1/admin/tsdb/clean_tombstones
```
New in v2.1 and supports PUT from v2.9

Remote Write Receiver

Prometheus can be configured as a receiver for the Prometheus remote write protocol. This is not considered an efficient way of ingesting samples. Use it with caution for specific low-volume use cases. It is not suitable for replacing the ingestion via scraping and turning Prometheus into a push-based metrics collection system.

Enable the remote write receiver by setting `--web.enable-remote-write-receiver`. When enabled, the remote write receiver endpoint is `/api/v1/write`. Find more details here.

New in v2.33

OTLP Receiver

Prometheus can be configured as a receiver for the OTLP Metrics protocol. This is not considered an efficient way of ingesting samples. Use it with caution for specific low-volume use cases. It is not suitable for replacing the ingestion via scraping.

Enable the OTLP receiver by setting `--web.enable-otlp-receiver`. When enabled, the OTLP receiver endpoint is `/api/v1/otlp/v1/metrics`.

New in v2.47

OTLP Delta

Prometheus can convert incoming metrics from delta temporality to their cumulative equivalent. This is done using deltatocumulative from the OpenTelemetry Collector.

To enable, pass `--enable-feature=otlp-deltatocumulative`.

New in v3.2

Notifications

The following endpoints provide information about active status notifications concerning the Prometheus server itself. Notifications are used in the web UI.

These endpoints are experimental. They may change in the future.

Active Notifications

The `/api/v1/notifications` endpoint returns a list of all currently active notifications.

`GET /api/v1/notifications`
Example:

```
$ curl http://localhost:9090/api/v1/notifications
{
  "status": "success",
  "data": [
    {
      "text": "Prometheus is shutting down and gracefully
stopping all operations.",
      "date": "2024-10-07T12:33:08.551376578+02:00",
      "active": true
    }
  ]
}
```

New in v3.0

Live Notifications

The `/api/v1/notifications/live` endpoint streams live notifications as they occur, using Server-Sent Events. Deleted notifications are sent with `active: false`. Active notifications will be sent when connecting to the endpoint.

`GET /api/v1/notifications/live`
Example:

```
$ curl http://localhost:9090/api/v1/notifications/live
data: {
```

```
  "status": "success",
  "data": [
    {
      "text": "Prometheus is shutting down and gracefully
stopping all operations.",
      "date": "2024-10-07T12:33:08.551376578+02:00",
      "active": true
    }
  ]
}
```

Note: The `/notifications/live` endpoint will return a `204 No Content` response if the maximum number of subscribers has been reached. You can set the maximum number of listeners with the flag `--web.max-notifications-subscribers`, which defaults to 16.

```
GET /api/v1/notifications/live
204 No Content
```

New in v3.0