The Prometheus Query Language (PromQL) is a powerful and flexible language used to query and manipulate time-series metrics in Prometheus.

It forms the backbone of extracting insights from your metrics data. Whether you're building dashboards, setting up alerts, or exploring your data through the API, PromQL provides the tools to meet diverse monitoring needs.

This guide will introduce the core concepts of PromQL, covering its data types and fundamental query structures. With practical examples, you'll build a strong foundation for mastering Prometheus queries and effectively analyzing your metrics.

Let's get started!

# Prerequisites

- Prior knowledge of Prometheus metrics concepts.
- A recent version of Docker and Docker Compose installed on your machine.

# Setting up a local Prometheus sandbox

To experiment with PromQL queries, you need a monitoring service set up with Prometheus and a data source, such as Node Exporter, to collect system metrics.

In this section, you'll configure a local environment using Docker Compose to set up Prometheus and Node Exporter. This setup will provide a sandbox for learning and testing PromQL's capabilities.

Start by creating a `promql-tutorial` in your filesystem to place the configuration files and change into it:

```
mkdir promql-tutorial
```

```
cd promql-tutorial
```

Next, create a `docker-compose.yml` file with the following contents to define the services:

docker-compose.yml

```
services:
  node-exporter:
    image: prom/node-exporter:latest
    container_name: node-exporter
    restart: unless-stopped
    volumes:
      - /proc:/host/proc:ro
      - /sys:/host/sys:ro
      - /:/rootfs:ro
    command:
      - --path.procfs=/host/proc
      - --path.rootfs=/rootfs
      - --path.sysfs=/host/sys
      - --collector.filesystem.mount-points-exclude=^/(sys|
proc|dev|host|etc)($$|/)
    networks:
      - monitoring

  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    restart: unless-stopped
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
      - ./alerts.yml:/etc/prometheus/alerts.yml
```

```
      - prometheus_data:/prometheus
    command:
      - --config.file=/etc/prometheus/prometheus.yml
      - --storage.tsdb.path=/prometheus
      - --web.console.libraries=/etc/prometheus/
console_libraries
      - --web.console.templates=/etc/prometheus/consoles
      - --web.enable-lifecycle
    expose:
      - 9090
    ports:
      - 9090:9090
    networks:
      - monitoring

networks:
  monitoring:
    driver: bridge

volumes:
  prometheus_data:
```

This monitoring setup uses two services: `node-exporter` and `prometheus`. The `node-exporter` gathers hardware and system metrics from the machine it runs on, while `prometheus` collects these metrics and stores them for analysis.

The Prometheus UI has also been made accessible at `http://localhost:9090` so that you can query and visualize the collected data.

Within the same directory, create a `prometheus.yml` file to configure Prometheus to scrape metrics from itself and Node Exporter:

prometheus.yml

```
global:
  scrape_interval: 10s

scrape_configs:
  - job_name: prometheus
    static_configs:
      - targets:
          - 'localhost:9090'
```

```
  - job_name: node-exporter
    static_configs:
      - targets:
          - 'node-exporter:9100'
```

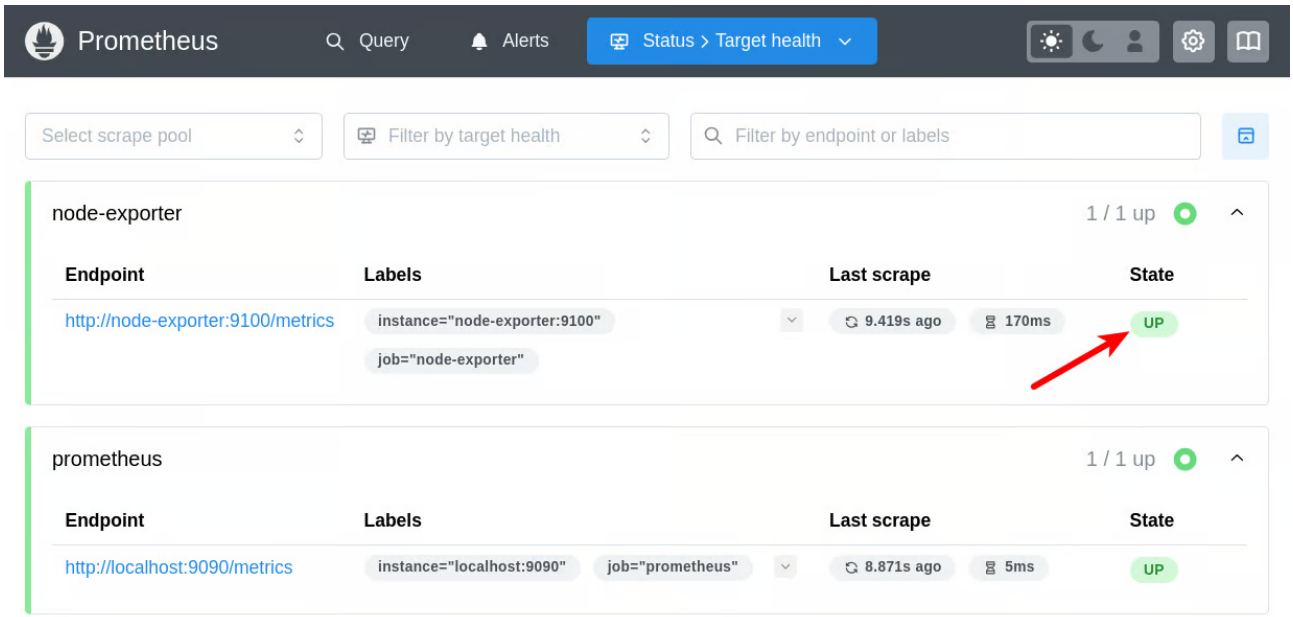
With this configuration in place, you're ready to launch both services with Docker Compose:

```
docker compose up -d
```

Output

```
[+] Running 2/2
 ✔ Container node-exporter  Running
0.4s
 ✔ Container prometheus     Running
0.5s
```

Once the setup is complete, the Prometheus UI should be accessible and you can confirm that the Node Exporter metrics are being scraped by heading to `http://localhost:9090/targets`:



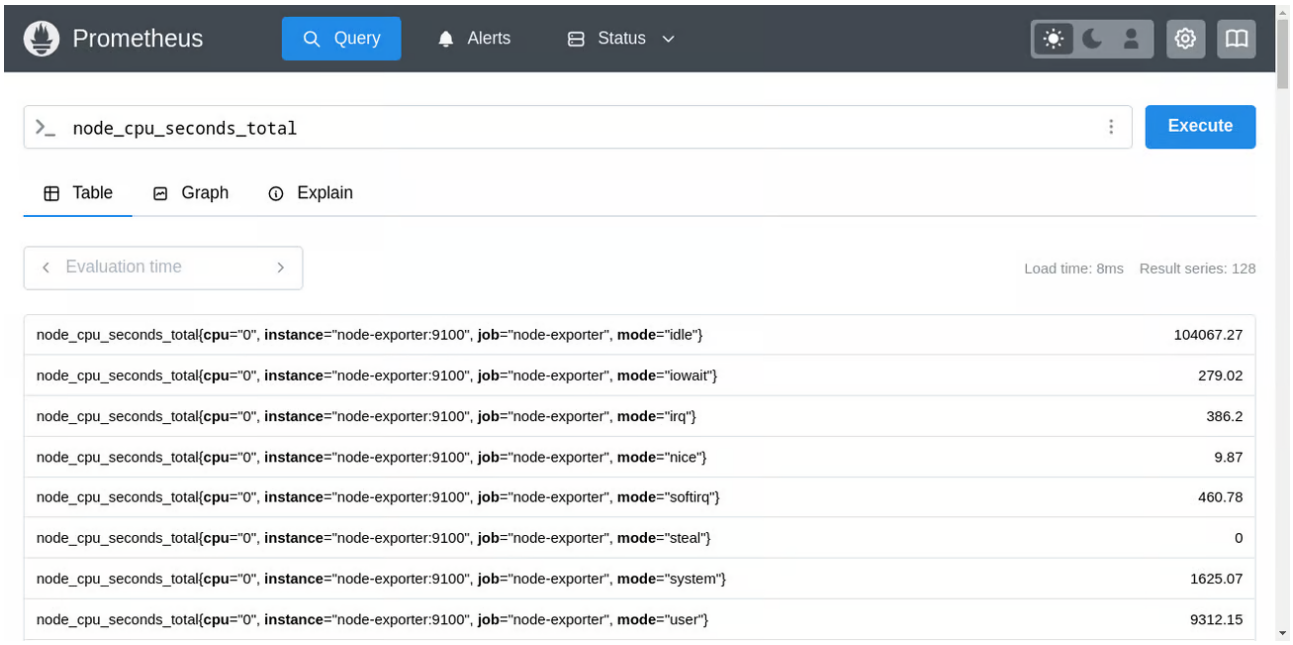
You are now ready to explore metrics and test PromQL queries in Prometheus. In the following sections, we'll explore several sample queries for you to experiment with.

# Getting started with PromQL

The most basic way to use PromQL is by typing a metric name into the expressions input. For example, you can type the metric below to get a snapshot of its value at that particular moment:

`node_cpu_seconds_total`

This results in an instant vector containing the current value of that metric for each labeled instance at query time (like the different CPU instances in this case).



It's called an instant vector because a single name may contain many values (as seen in the screenshot above) due to the use of labels as each one creates its own time series.

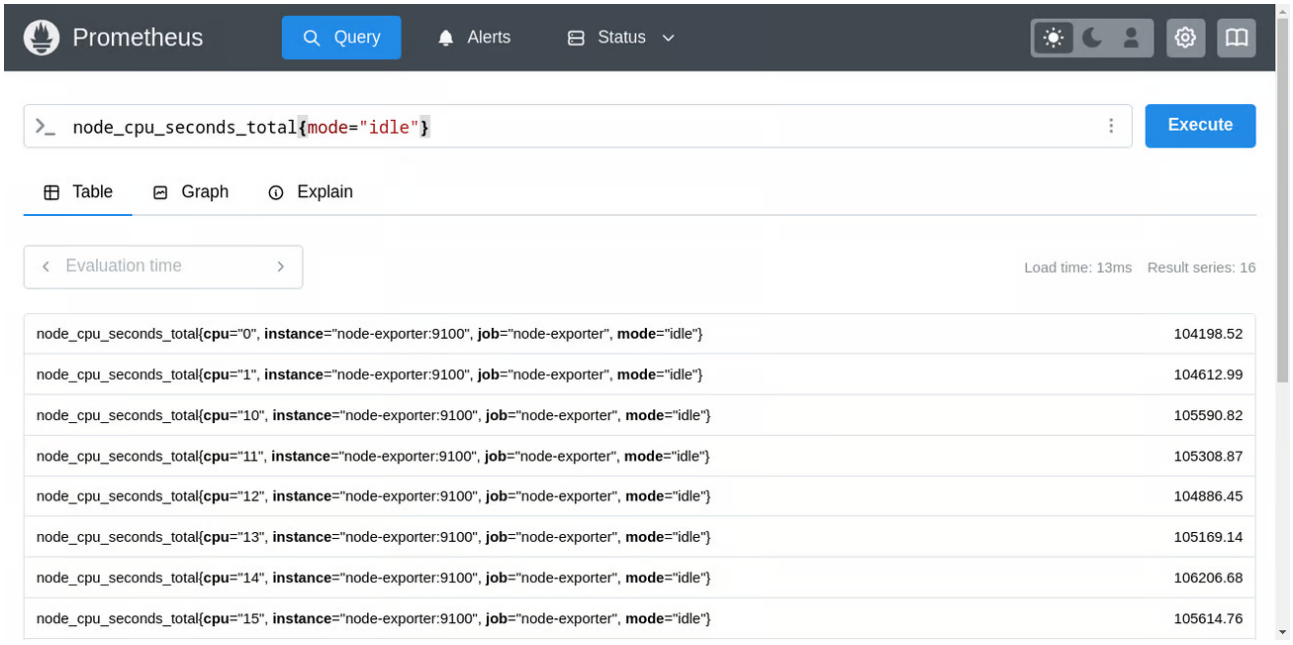
The syntax of an instant vector selector is:

```
<metric_name>{<label_selectors>}
```

You can filter the output by adding a comma-separated list of label matchers enclosed in curly braces {}.

For instance, to select only the time series representing the idle CPU time, you can use:

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




If you want to filter by multiple labels, you can separate them with commas:

```
node_cpu_seconds_total{mode="idle",cpu="0"}
```

Besides the `=` sign, which selects labels that are exactly equal to the provided string, you can also use the following label-matching operators:

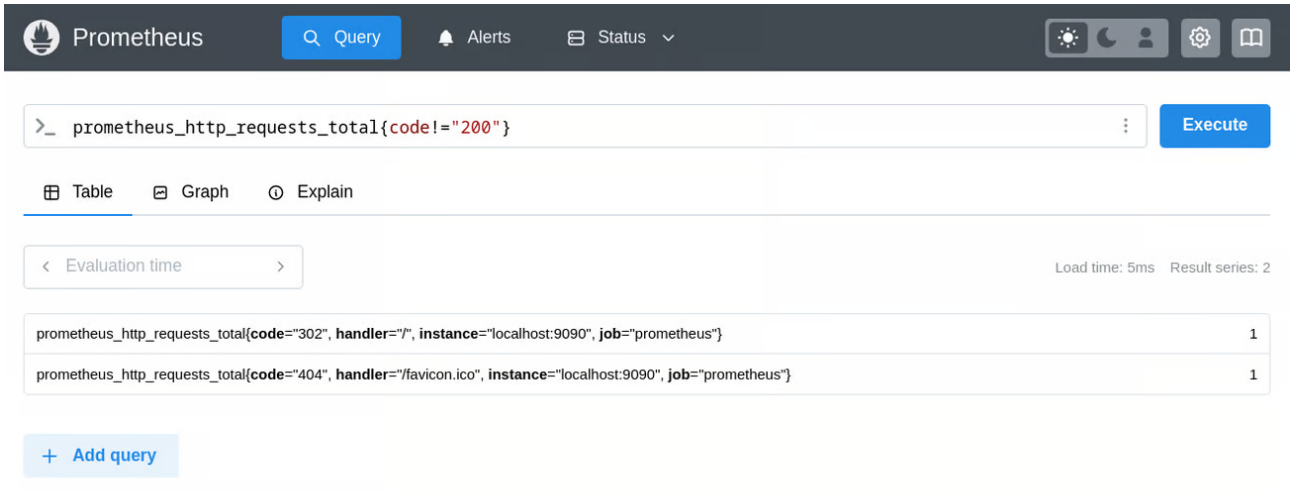
- `!=`: Select labels that are not equal to the provided string.
- `=~`: Select labels that match a regular expression.
- `!~`: Select labels that do not match a regular expression

For example, you can select the time series to paths under `/api/v1/` with the following PromQL:

```
prometheus_http_requests_total{handler=~"/api/v1/.*"}
```

Or you can filter out the time series that have a `code` label of `200` with:

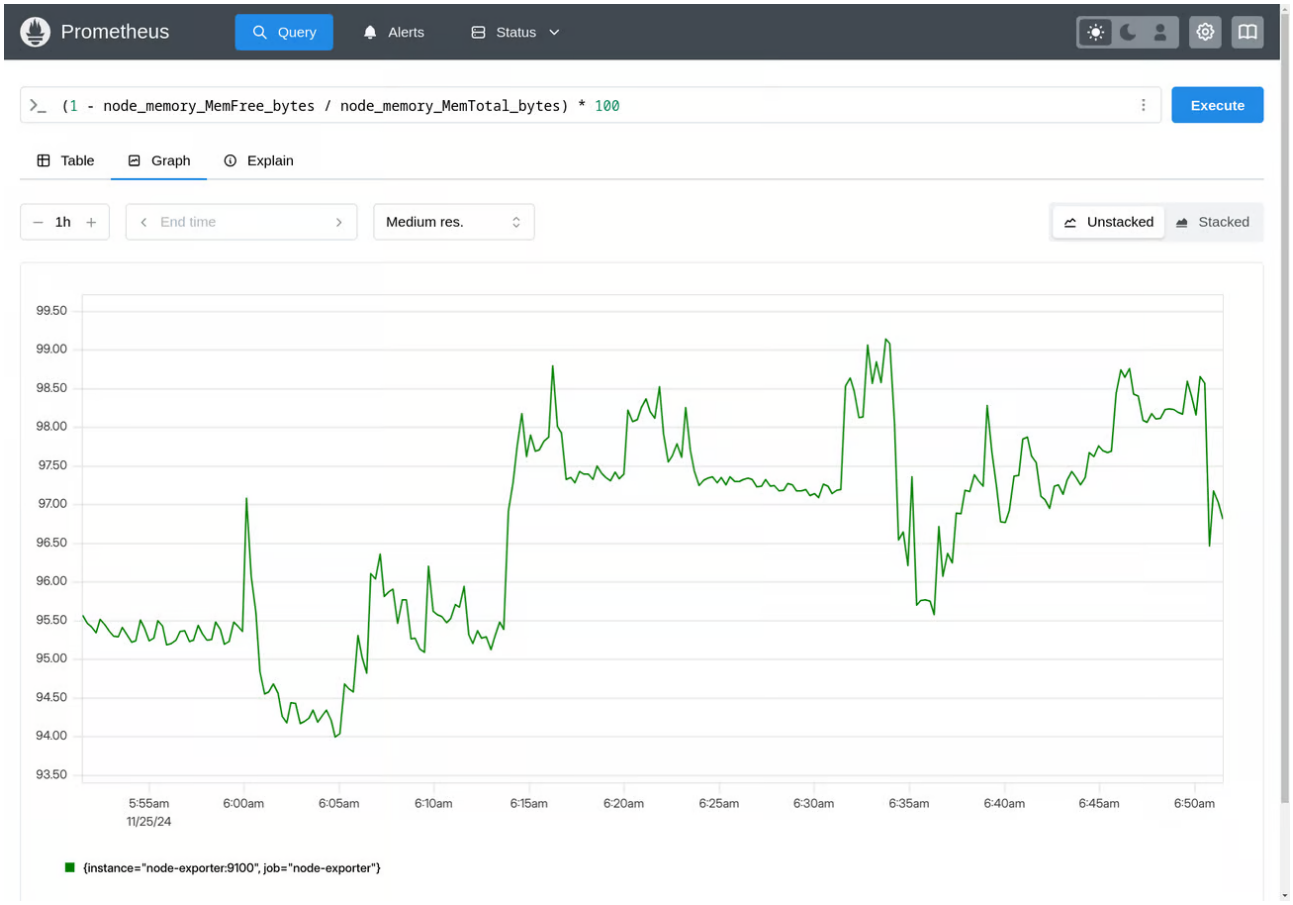
```
prometheus_http_requests_total{code!="200"}
```



Instead of just viewing a metric value at query time, you can select a range of samples back from the current instant with a range vector selector:

`<metric_name>{<label_selectors>}[<duration>]`
For example, the query below outputs the values of the `prometheus_http_requests_total` time series over the past five minutes for 4xx responses:

```
prometheus_http_requests_total{code=~"4.."}[5m]
```




The Prometheus expression browser cannot graph range vectors directly, so you'll usually wrap such expressions in a

function that calculates rates, averages, and trends over time such as rate() or increase():

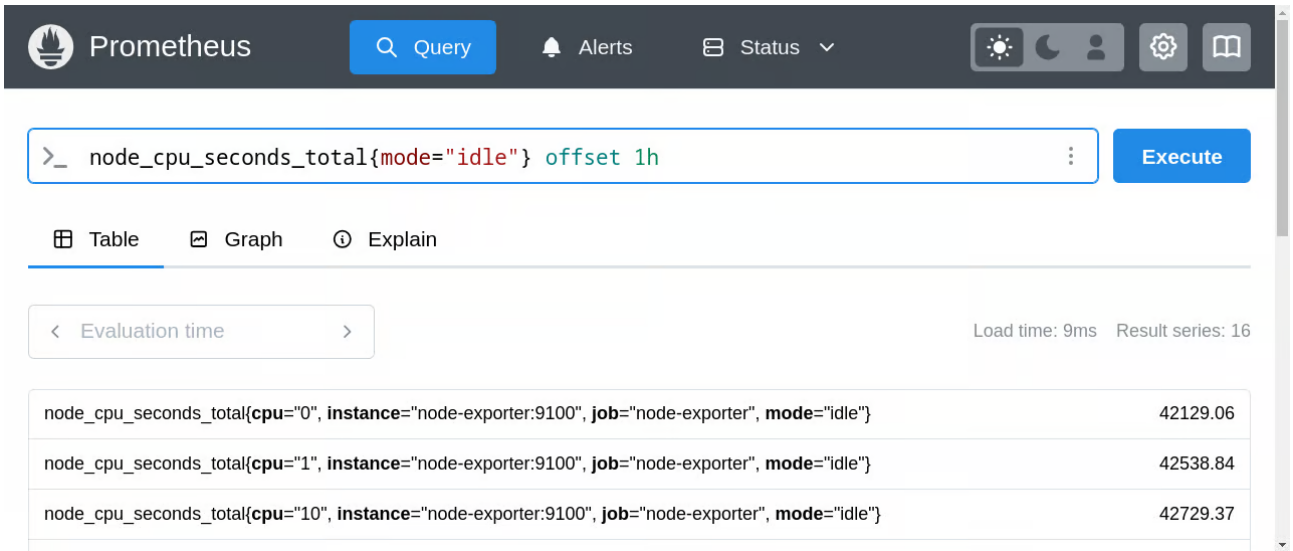
```
prometheus_http_requests_total{code=~"4.."}[5m]
```

With both instant and range vectors, you can use an offset modifier to query data from a specific point in the past relative to the current evaluation time. It's a useful way to compare current and past metrics, and validate if the reported values have improved compared to the previous period.

```
<instant_or_range_selector> offset <duration>
```

With instant vectors, an offset shifts its evaluation by a specified duration into the past. For example, the query below retrieves the value of node_cpu_seconds_total for the mode="idle" metric 1 hour ago instead of the current time.

```
node_cpu_seconds_total{mode="idle"} offset 1h
```



For range vectors, the offset applies to the start and end times of the specified range, shifting the entire range into the past.

```
rate(node_cpu_seconds_total{mode="idle"}[5m] offset 1h)
```

Here, the rate of increase for `node_cpu_seconds_total` is calculated over a 5-minute range, but the range starts 1 hour ago.

Another way to change the evaluation time for a query is by using the `@` modifier which allows you to evaluate a query at a specific, fixed Unix timestamp (in seconds):

```
<instant_or_range_selector> @ <unix_timestamp>
```

Here's how to calculate the request rate over five minutes, ending at a specific timestamp:

```
rate(http_requests_total[5m]) @ 1732616754
```

When using the `@` modifier with `offset`, the offset is applied relative to the time specified by the `@` modifier, no matter the order in which the modifiers appear in the query.

```
node_cpu_seconds_total{mode="idle"} offset 1h @ 1732616754
# This is equivalent to
node_cpu_seconds_total{mode="idle"} @ 1732616754 offset 1h
```

Both queries will retrieve data for `node_cpu_seconds_total` from 1 hour before the fixed timestamp `1732616754`.

# Exploring the PromQL grammar

Now that you have a basic idea of how to query your metrics data with PromQL, let's take a brief moment to understand its underlying structure and the key components of its grammar.

## Data types

PromQL operates on four fundamental data types:

- Scalar: A single numeric floating-point value (e.g., `10`, `-5.7`).
- Instant vector: A set of time series, each with a single value at a specific timestamp (e.g `http_requests_total`).
- Range vector: A set of time series, each with a range of data points over a specified time duration (e.g., `http_requests_total[5m]`).
- String: A simple string value.

## Expressions

PromQL queries are constructed from expressions. An expression can be a simple metric name (`http_requests_total`) or a complex combination of functions, operators, and selectors.

Expressions are evaluated to produce a result, which can be a scalar, an instant vector, or a range vector.

## Literals

PromQL supports string, scalar-float, and duration literals. String literals are designated by single quotes, double quotes, or backticks, while scalar-float values can be literal integers or floating-point numbers. Integers can also be combined with time units to specify durations (such as `1h`, `5m`, `1h30m` e.t.c.).

## Operators

PromQL provides arithmetic, comparison, logical, and aggregation operators for filtering and manipulating the metrics data. We'll take a closer look at these operators shortly.

## Functions

PromQL functions help you transform and analyze your time series data. They take instant vectors (or scalars and range vectors in some cases) as input and return a new value, usually another instant vector.

The most important functions are those that manipulate and analyze time series data, including rate calculations, deltas, and histogram analysis which we'll explore in some upcoming sections.

## Identifiers

Identifiers in PromQL include the names of the metrics you want to query such as `node_cpu_seconds_total` and the names of built-in functions like `rate()`, `deriv()` or `predict_linear()`.

## Comments

In PromQL, you can use comments to annotate queries, document logic, or clarify intentions. These comments start with the `#` character:

```
# Calculate the per-second rate of HTTP requests over the past 5 minutes
rate(http_requests_total[5m])
```

# Understanding the PromQL operators

Prometheus supports arithmetic, comparison, and logical/set operators, and you can use them between scalars, instant vectors, or combinations of both.

## Arithmetic operators

The arithmetic operators in PromQL include: `+`, `-`, `*`, `/`, `%`, and `^` (power). These operators can be used in the following scenarios:

### 1. Between two scalars

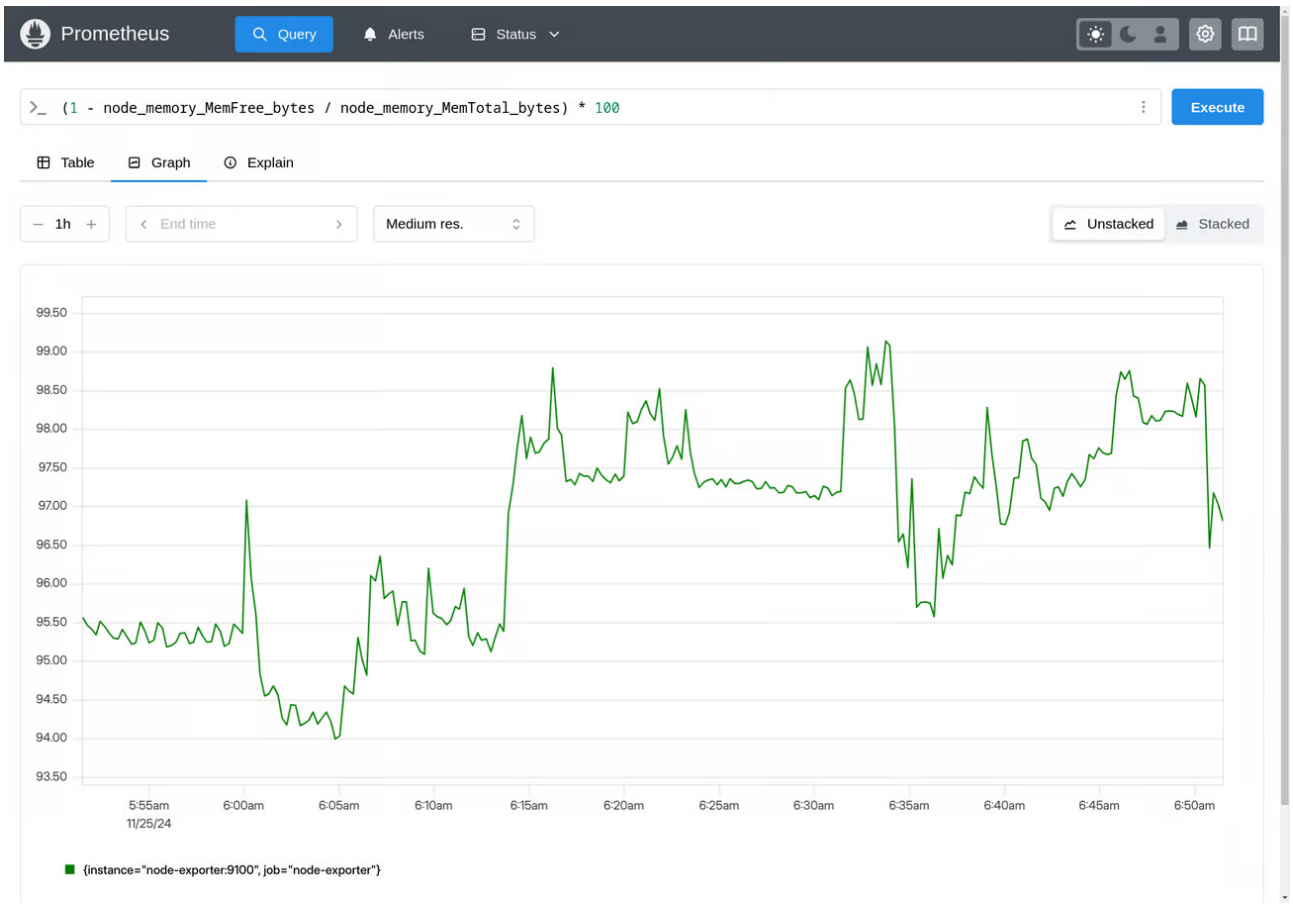
For example, let's calculate the percentage of memory used on a system through the following Node Exporter metrics:

- `node_memory_MemTotal_bytes`: Total physical memory in bytes.
- `node_memory_MemFree_bytes`: Free memory in bytes.

The PromQL query to compute the percentage of memory used is:

```
(1 - node_memory_MemFree_bytes / node_memory_MemTotal_bytes)
* 100
```

Here, `node_memory_MemFree_bytes` is divided by `node_memory_MemTotal_bytes` to produce the free and total memory ratio. This result is then subtracted from 1 to yield the proportion of used memory, and subsequently multiplied by 100 to convert the result to a percentage.

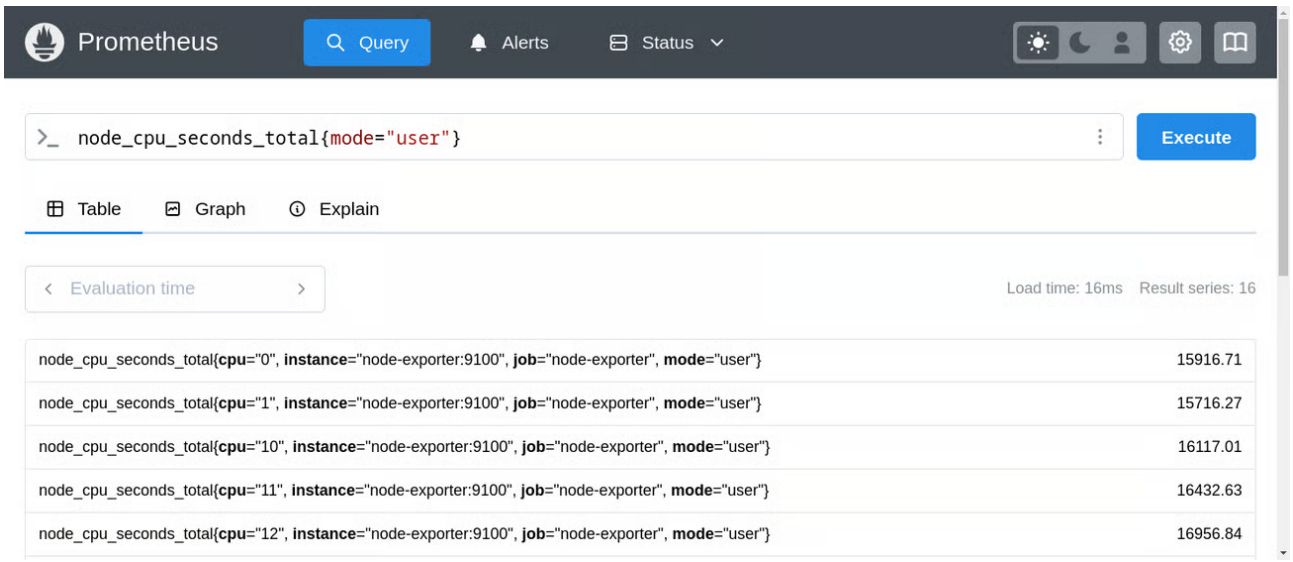



## 2. Between a scalar and an instant vector

When an arithmetic operator is applied between a scalar and an instant vector, each sample in the vector is transformed by the scalar, and the metric name is dropped in the resulting vector. This operation is helpful in scaling or normalizing metrics.

Let's use the `node_cpu_seconds_total` metric as an example. It represents the total time the CPU has spent in various states (such as `user`, `system`, `idle`) in seconds:

```
node_cpu_seconds_total{mode="user"}
```

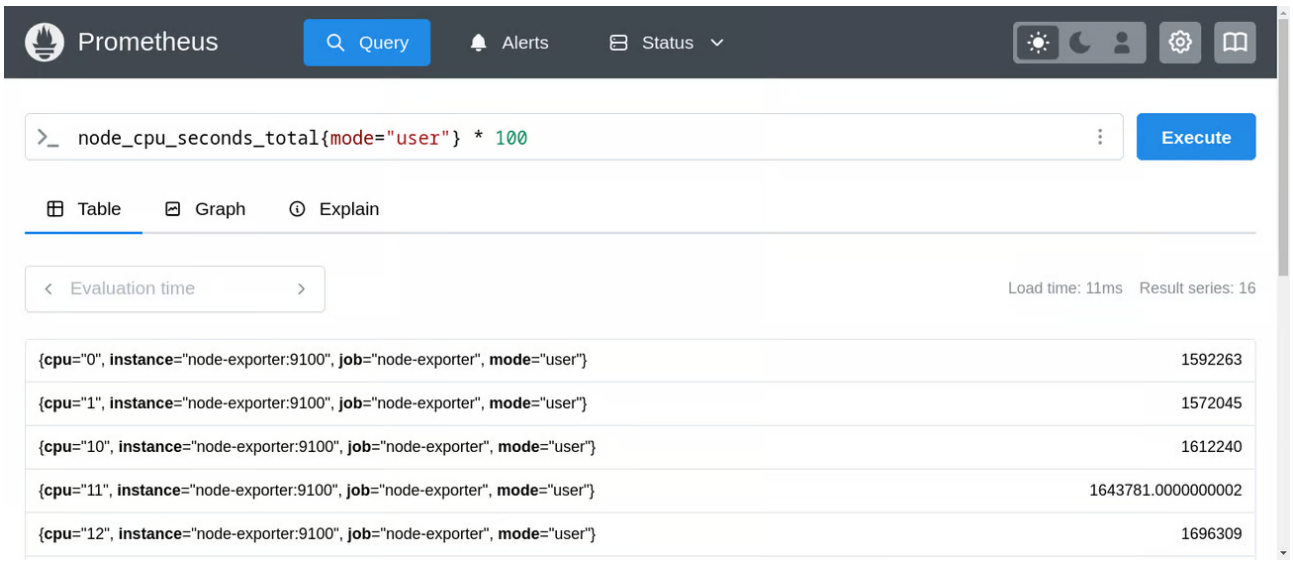



To scale these values (e.g., to express CPU time as a percentage of some unit), you can multiply the vector by a scalar. For example:

```
node_cpu_seconds_total{mode="user"} * 100
```

The result is a new instant vector where:

- The values are scaled.
- The labels (`instance`, `cpu`, `mode`) remain unchanged.
- The metric name (`node_cpu_seconds_total`) is dropped.




## 3. Between two instant vectors

When applying an arithmetic operator between two instant vectors, PromQL performs the operation on matching pairs of time series. Matches are based on identical label sets in both vectors.

Let's take the `node_network_receive_bytes_total` and `node_network_transmit_bytes_total` metrics for example. They represent the total number of bytes received and transmitted on a network interface respectively, and they both have a `device` label which identifies the network interface.

You can compute the difference between received and transmitted bytes for each network interface with the following query:

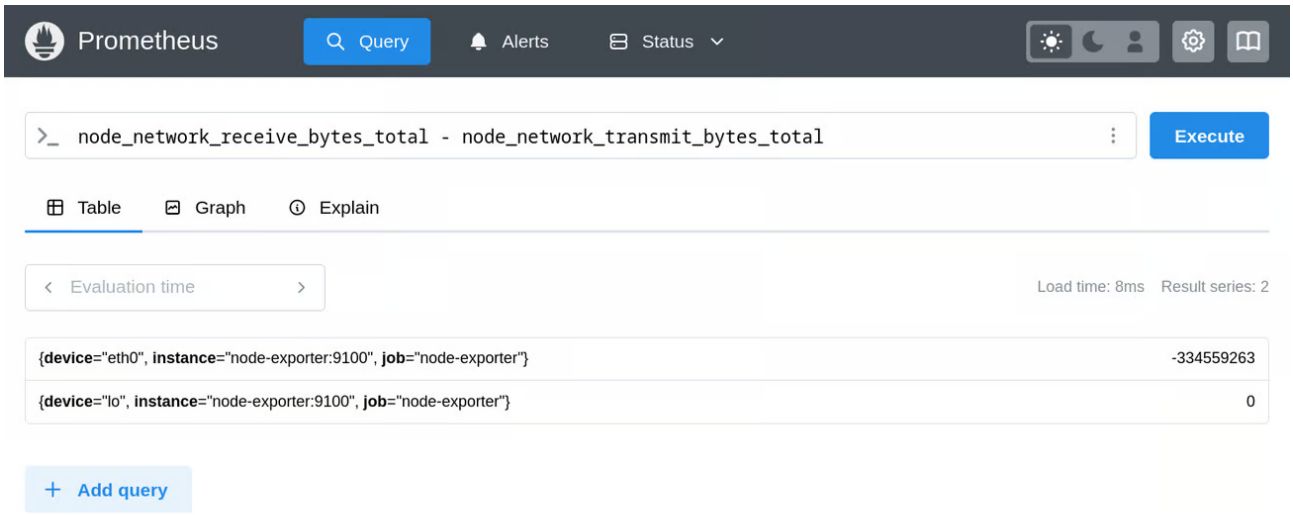
```
node_network_receive_bytes_total -
node_network_transmit_bytes_total
```

If the individual metrics report the following:

```
node_network_receive_bytes_total{device="eth0"}: 1000000
```

`node_network_transmit_bytes_total{device="eth0"}: 500000`
The output would be:

`{device="eth0"}: 500000`
Any label that does not exist in both vectors will be dropped in the resulting vector.




A negative result (as seen above) indicates that the bytes transmitted on a network interface exceeds the bytes received over the same period.

## Comparison operators

We also have comparison operators like `==`, `!=`, `>`, `<`, `>=`, and `<=` which can be used for filtering time series based on label values or metric values, or to return a boolean value (`0` or `1`) instead of filtering.

Here are the possible ways to use them:

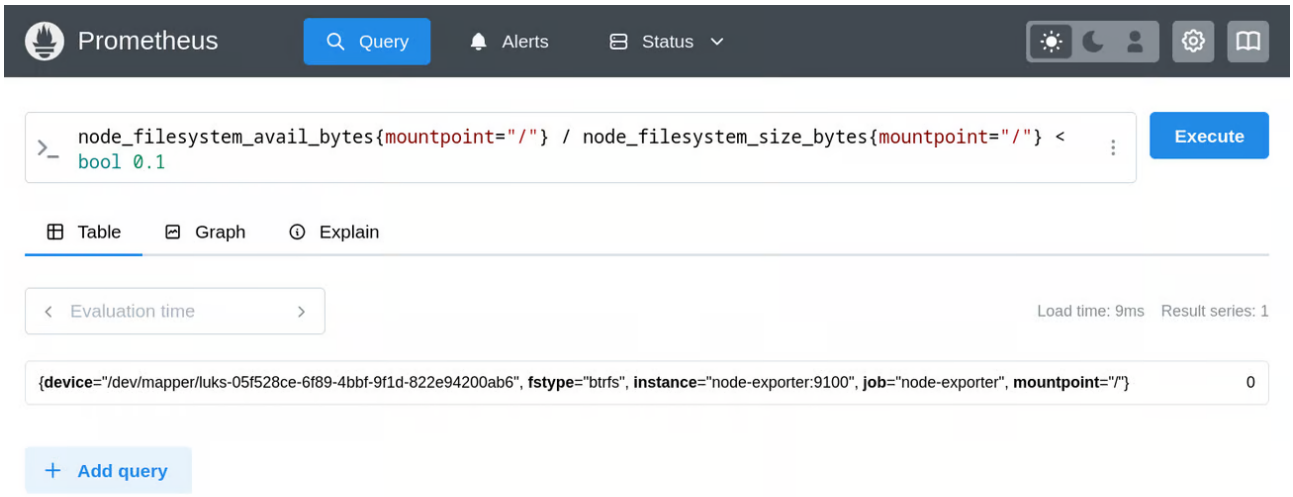
## 1. Comparing two scalars

Comparison operators are useful for comparing scalar values to determine if a metric meets a threshold or condition. When applied between scalars, the `bool` modifier must be used to return a boolean result (`0` or `1`).

Here's a query that uses the `<` operator to check if the available disk space is less than 10% of the total disk space:

```
node_filesystem_avail_bytes{mountpoint="/"} /
node_filesystem_size_bytes{mountpoint="/"} < bool 0.1
```

The calculated scalar value (the result of the division) is compared with the scalar literal (0.1), which is prefixed with the `bool` modifier so that `0` (false) or `1` (true) is returned.




This type of scalar comparison is often used in alerting rules where an alert is triggered if the specified condition surpasses a known threshold:

alerts.yml

```
- alert: LowDiskSpace
  expr: node_filesystem_avail_bytes{mountpoint="/"} /
node_filesystem_size_bytes{mountpoint="/"} < bool 0.1
  for: 5m
  labels:
    severity: critical
  annotations:
    summary: "Low disk space on {{ $labels.instance }}"
    description: "The root filesystem on
{{ $labels.instance }} has less than 10% available disk
space."
```

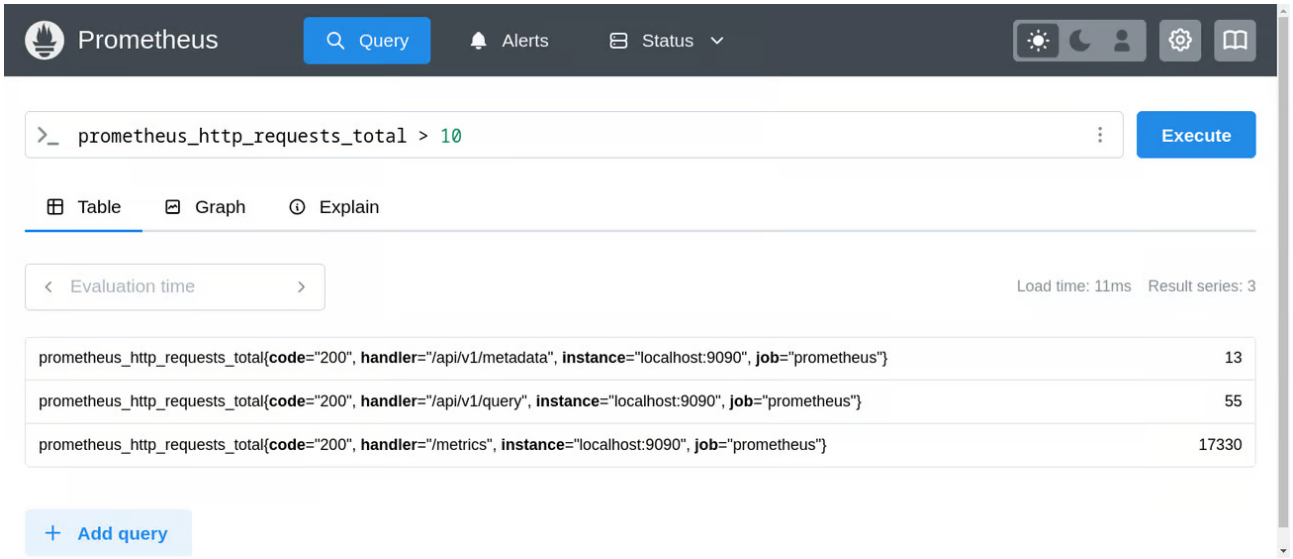
## 2. Comparing a scalar and an instant vector

When a comparison operator is applied between an instant vector and a scalar value, each element in the vector is evaluated against the scalar.

The time series that evaluate to `false` are dropped, while the values satisfying the condition remains.

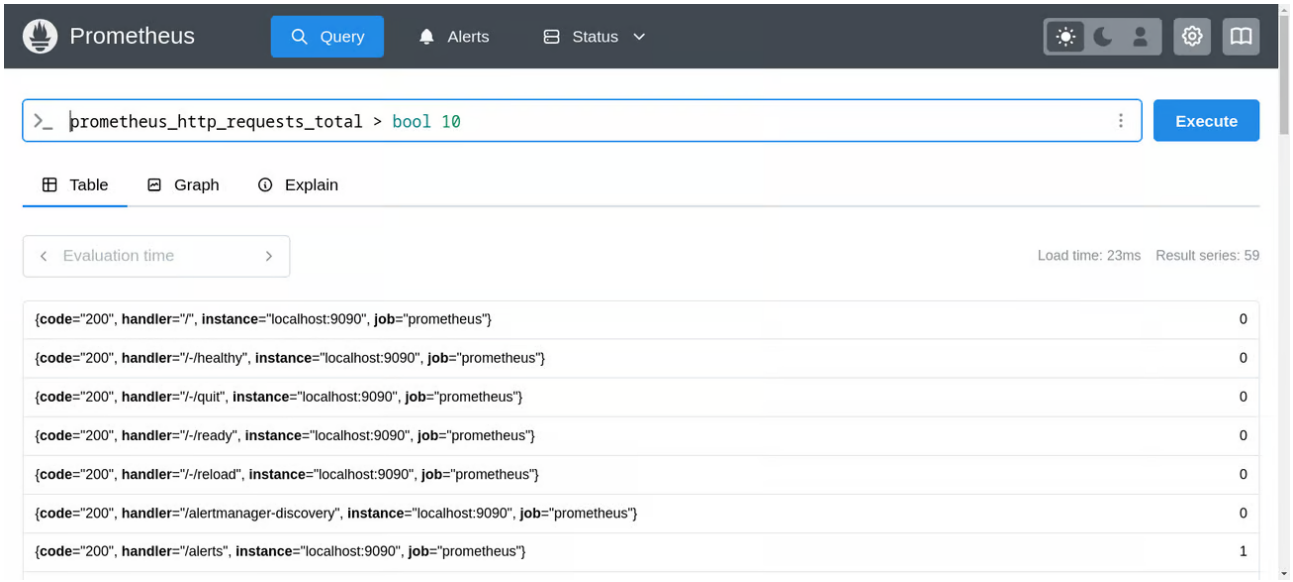
```
prometheus_http_requests_total > 10
```

With the above query, all matching time series whose value is less than or equal to 10 will be dropped:



If the `bool` modifier is provided, the vector elements that would have been dropped will be assigned a value of `0` while those that would have been kept are assigned the value of `1`. The metric name is also dropped in this case.

```
prometheus_http_requests_total > bool 10
```



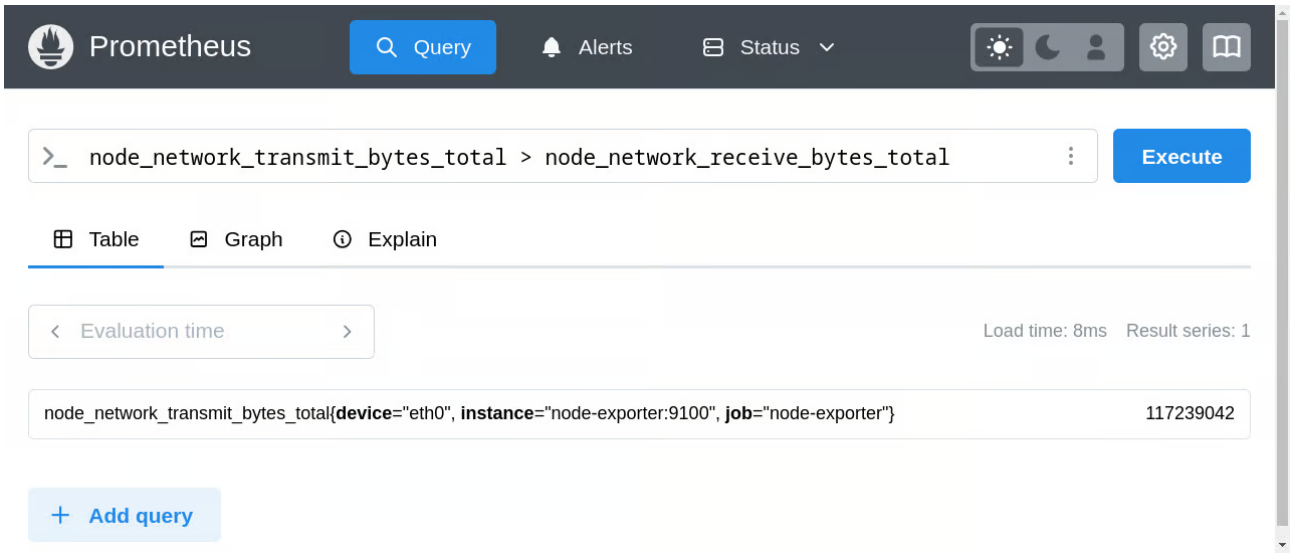

## 3. Comparing two instant vectors

You can also compare two instant vectors with a comparison operator. In this case, PromQL evaluates the elements of the vectors based on their labels and drops non-matching or `false` results.

For instance, let's compare the `node_network_receive_bytes_total` and `node_network_transmit_bytes_total` metrics to find network interfaces where transmitted bytes exceed received bytes:

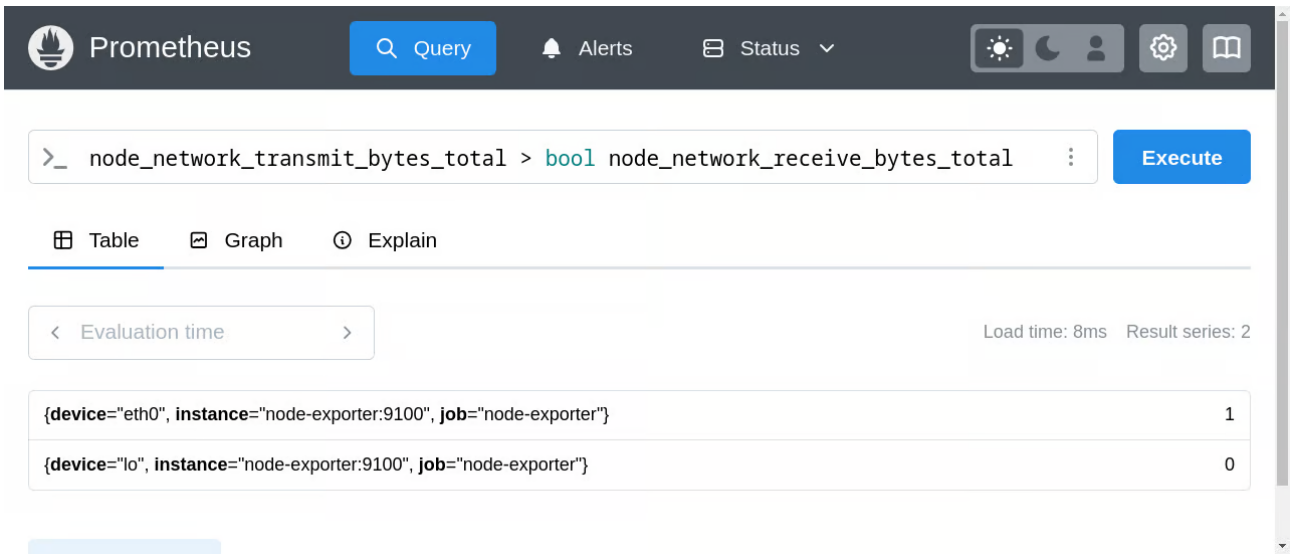
```
node_network_receive_bytes_total >
node_network_transmit_bytes_total
```

For each network interface (`device` label), PromQL compares the received bytes with transmitted bytes and returns only time series where transmitted bytes exceed received bytes:




With the `bool` modifier, all time series are retained but assigned `1` if transmitted bytes are greater or `0` otherwise. However, the metric name is dropped.

```
node_network_receive_bytes_total > bool
node_network_transmit_bytes_total
```




## Logical operators

The logical operators in PromQL (`and`, `or`, `unless`) are used to combine or filter instant vectors based on label sets and conditions. These operators operate only on instant vectors and help refine or manipulate query results.

Let's look at each one in turn.

### 1. The `and` operator

The `and` operator computes the intersection of two vectors, keeping only the time series from the left-hand vector (`vector1`) that exist in both vectors and have matching label sets.

For example, assume the following values for the `node_network_receive_bytes_total` and `node_network_transmit_bytes_total` metrics:

```
node_network_receive_bytes_total:
  {device="eth0"}: 1000000

node_network_transmit_bytes_total:
  {device="eth0"}: 500000
```

```
  {device="eth1"}: 20000
```
The following `and` expression:

```
node_network_receive_bytes_total and
node_network_transmit_bytes_total
```
Will produce the following result:

```
node_network_receive_bytes_total{device="eth0"}: 1000000
```
Both vectors have matching labels for `device="eth0"`, so the left-hand time series is included in the result. For `device="eth1"`, the `node_network_receive_bytes_total` metric does not exist, so this time series is excluded altogether.

### 2. The `or` operator

With the `or` operator, all-time series from the left-hand vector are retained alongside the elements of the right-hand vector which do not have matching label sets.

If you have the same values for the `node_network_receive_bytes_total` and `node_network_transmit_bytes_total` metrics in the previous section, applying the `or` operator:

```
node_network_receive_bytes_total or
node_network_transmit_bytes_total
```
Would yield:

```
node_network_receive_bytes_total:
  {device="eth0"}: 1000000
node_network_transmit_bytes_total:
  {device="eth1"}: 20000
```

For `device="eth0"`, the value from the left-hand vector is kept. For `device="eth1"`, the time series is taken from `node_network_transmit_bytes_total` because it has no match in the left-hand vector.

### 3. The `unless` operator

The `unless` operator is useful for identifying time series that are present in one metric but absent in another.

With the following metric values:

```
node_network_receive_bytes_total:
  {device="eth0"}: 1000000
  {device="eth1"}: 800000

node_network_transmit_bytes_total:
  {device="eth0"}: 500000
```

The result of the `unless` operator:

```
node_network_receive_bytes_total unless
node_network_transmit_bytes_total
```

Would be:

```
node_network_receive_bytes_total:
  {device="eth1"}: 800000
```

Since both vectors have matching time series for the `device="eth0"` label, it is excluded from the result, while `device="eth1"` is included since it is missing from the right-hand vector.

## Aggregation operators

PromQL's aggregation operators let you combine related time series, revealing broader trends in your metrics. Similar to

SQL's GROUP BY, these operators group time series by labels, applying functions like sum() or avg() to consolidate their values.

Here are the operators you're likely to encounter most often:

| Operator | Description |
| --- | --- |
| sum | Calculates the sum of values across time-series. |
| avg | Computes the average of values across time-series. |
| min | Finds the minimum value across time-series. |
| max | Finds the maximum value across time-series. |
| count | Counts the number of time-series. |
| stddev | Calculates the standard deviation across time-series. |
| stdvar | Calculates the variance across time-series. |
| quantile | Computes a specific quantile (e.g., 0.95). |
| topk | Selects the top k time-series by value. |
| bottomk | Selects the bottom k time-series by value. |

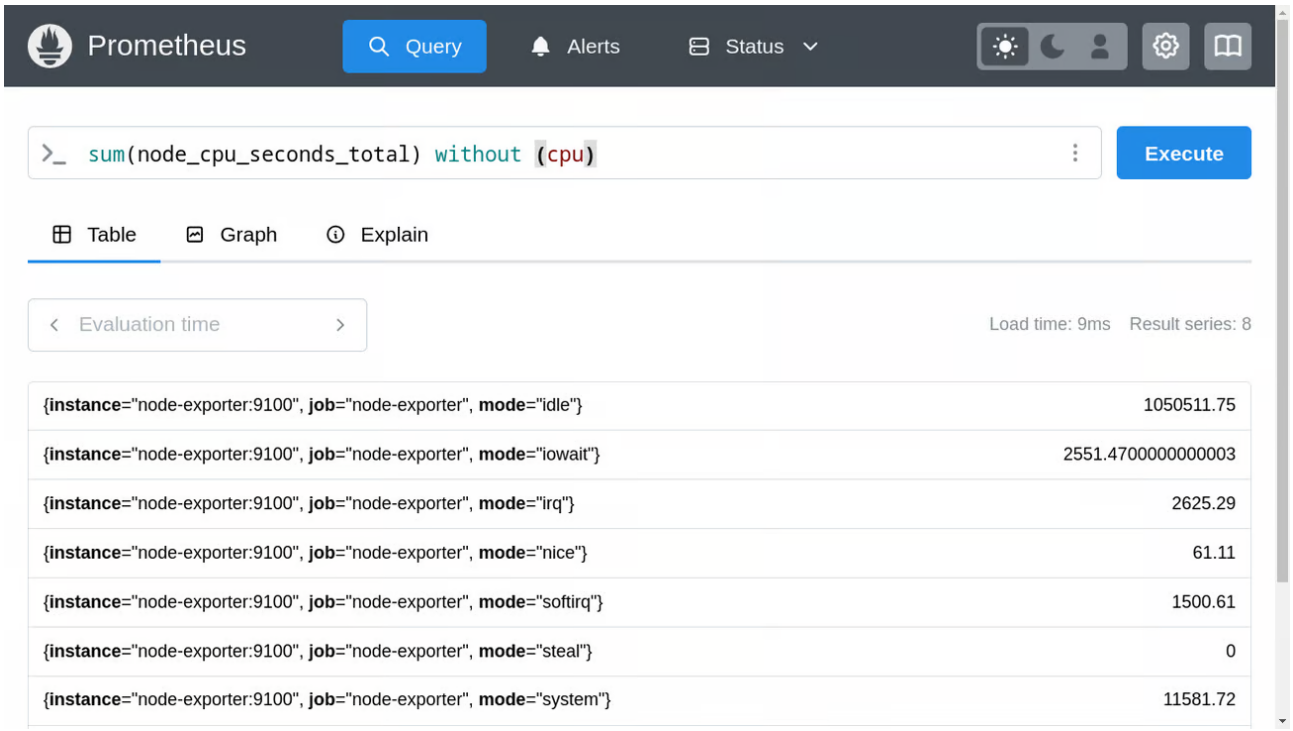
These operators can aggregate across all label dimensions or retain specific dimensions through the without or by clause:

```
<aggregation_operator> [without|by (<label_list>)]
([parameter,] <vector_expression>)
# or
<aggregation_operator>([parameter,] <vector_expression>)
[without|by (<label_list>)]
```

Here, by (<label_list>) groups the aggregation by the specified labels, while without (<label_list>) aggregates across all labels except the specified ones.

A basic example of an aggregation is summing the total CPU usage across all cores while removing the `cpu` label from the result:

```
sum(node_cpu_seconds_total) without (cpu)
```

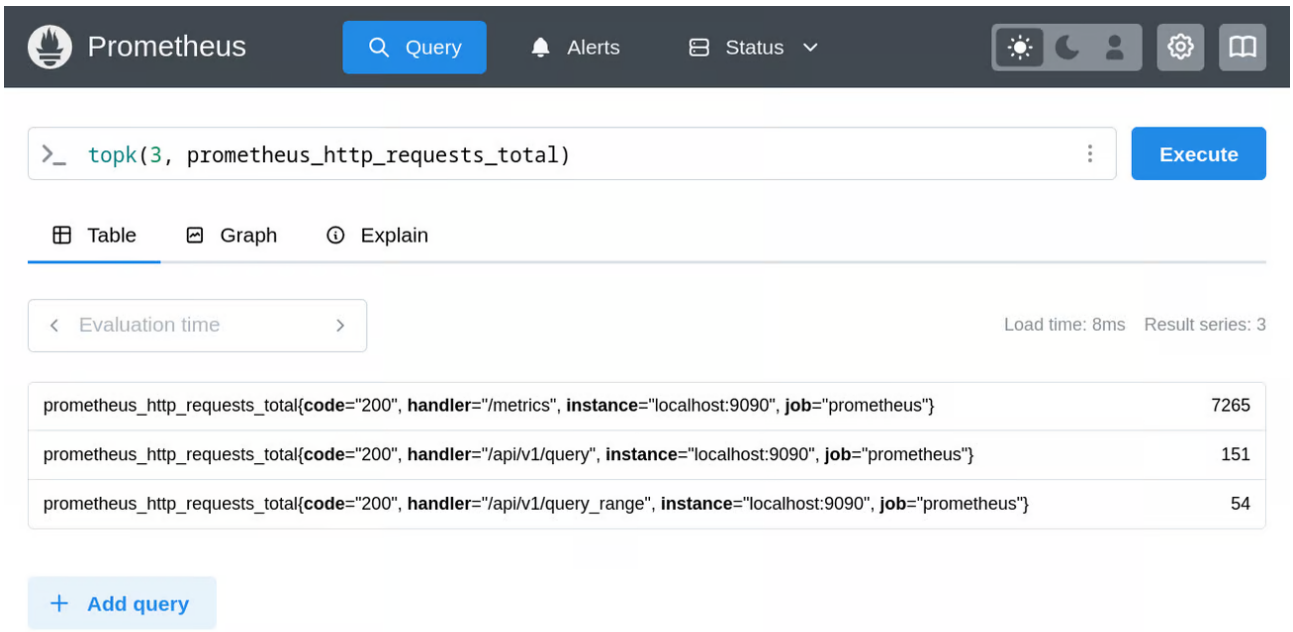



Or you can compute the average memory usage across all instances with:

```
avg(node_memory_MemAvailable_bytes / node_memory_MemTotal_bytes)
```

If you'd like to see the top 3 routes in terms of requests received, you can use:

```
topk(3, prometheus_http_requests_total)
```




# Querying counters

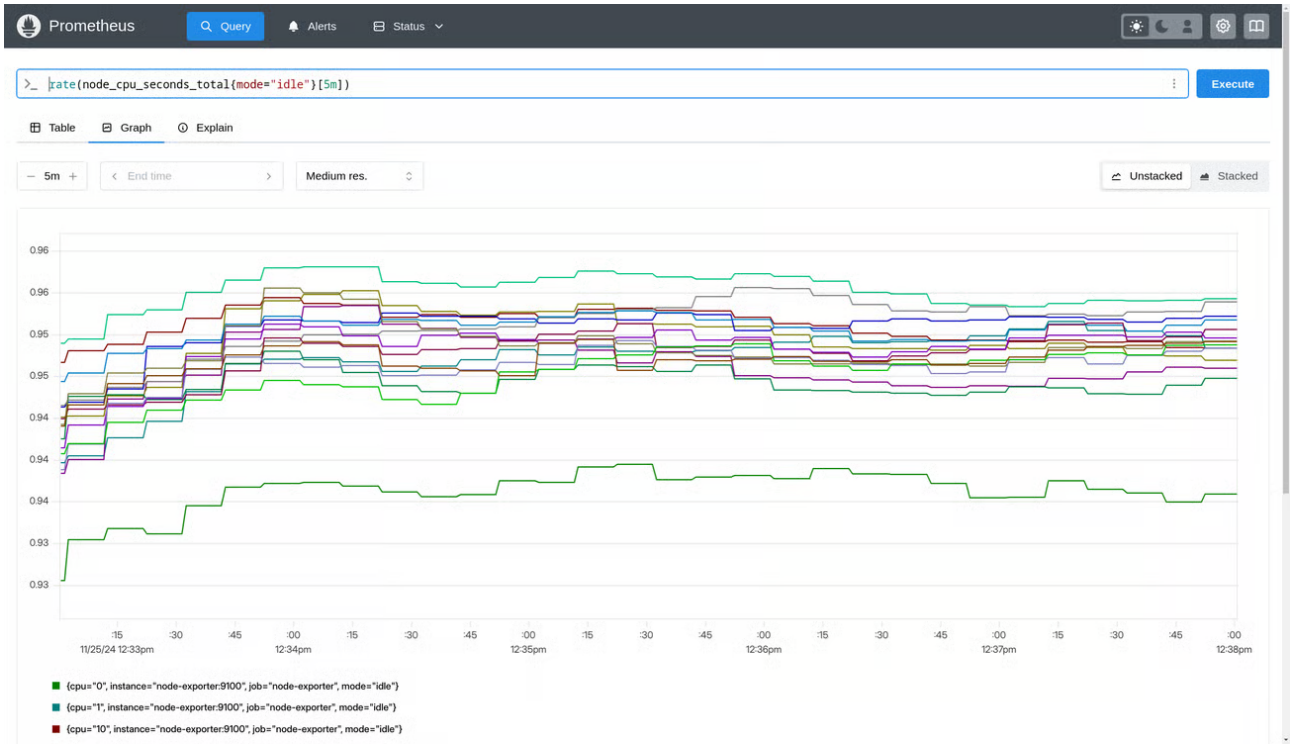
Counters are metrics that only increase or reset to zero (such as during a service restart). To extract meaningful insights from counters, PromQL offers functions like `rate()` and `increase()`, which are designed to handle counter resets gracefully.

Most of the time, you'll want to know how quickly the counter is changing over time. This is calculated with the `rate()` function:

```
rate(<metric_name>[<duration>])
```

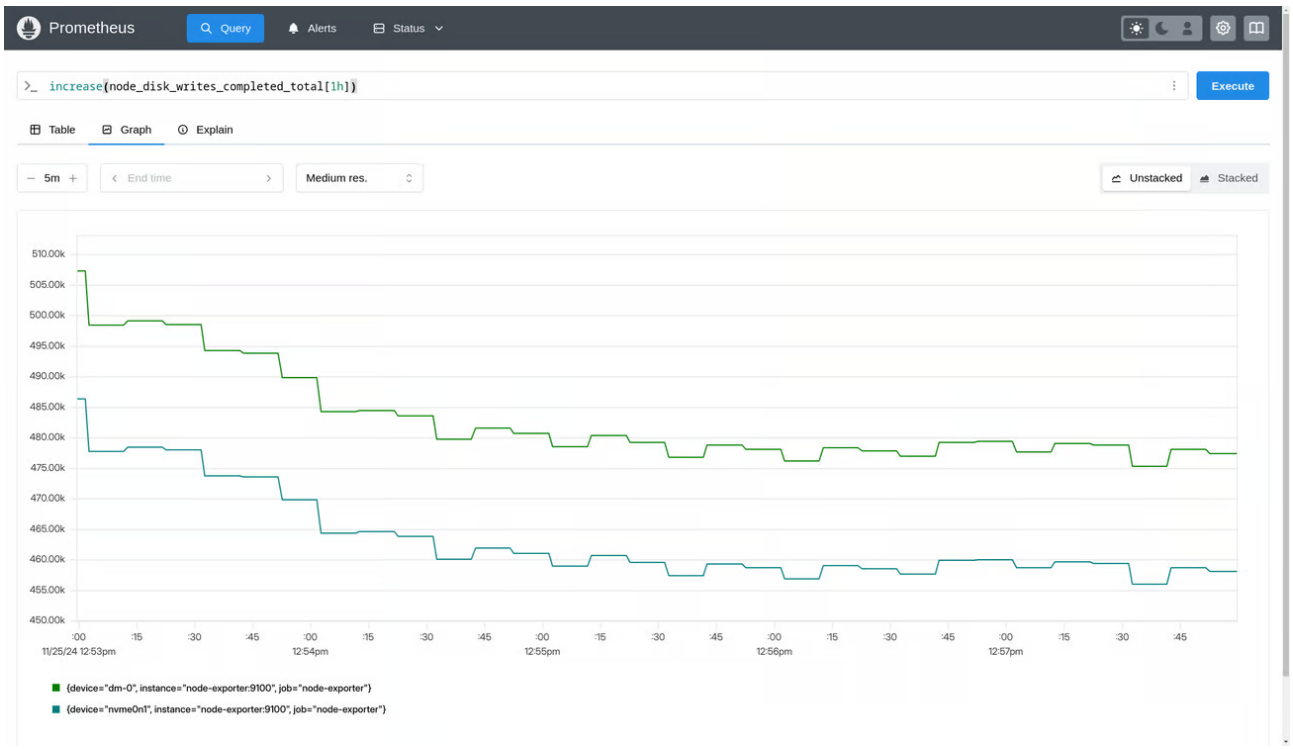
For example, you can see the per-second rate of change of idle CPU time for each CPU on your computer with:

```
rate(node_cpu_seconds_total{mode="idle"}[5m])
```




On the other hand, increase() shows you the total change in a counter over a time period. For example, you can see total number of disk writes completed in the last hour with:

```
increase(node_disk_writes_completed_total[1h])
```




Both functions graciously account for counter resets, such as when the service being monitored is restarted. In such cases, they effectively ignore the reset and continue calculating the rate of change based on the available data.

You can also use these functions for alerting. For instance, here's how to trigger an alert if your error rate exceeds 5%:

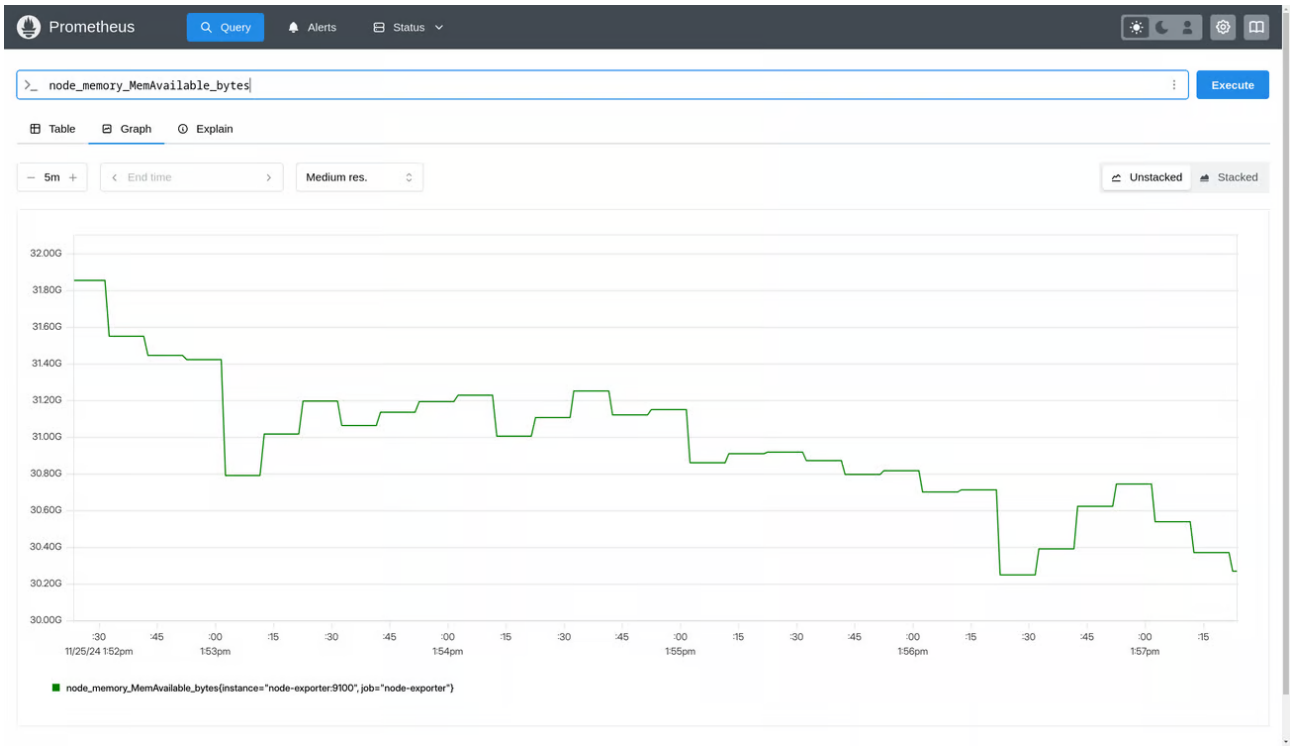
alerts.yml

```
- alert: HighErrorRate
  expr: rate(http_requests_total{code=~"5.."}[5m]) /
rate(http_requests_total[5m]) > 0.05
  for: 2m
  labels:
    severity: error
  annotations:
    summary: "High error rate detected"
    description: "The error rate is above 5% for
{{ $labels.instance }}"
```

# Querying gauges

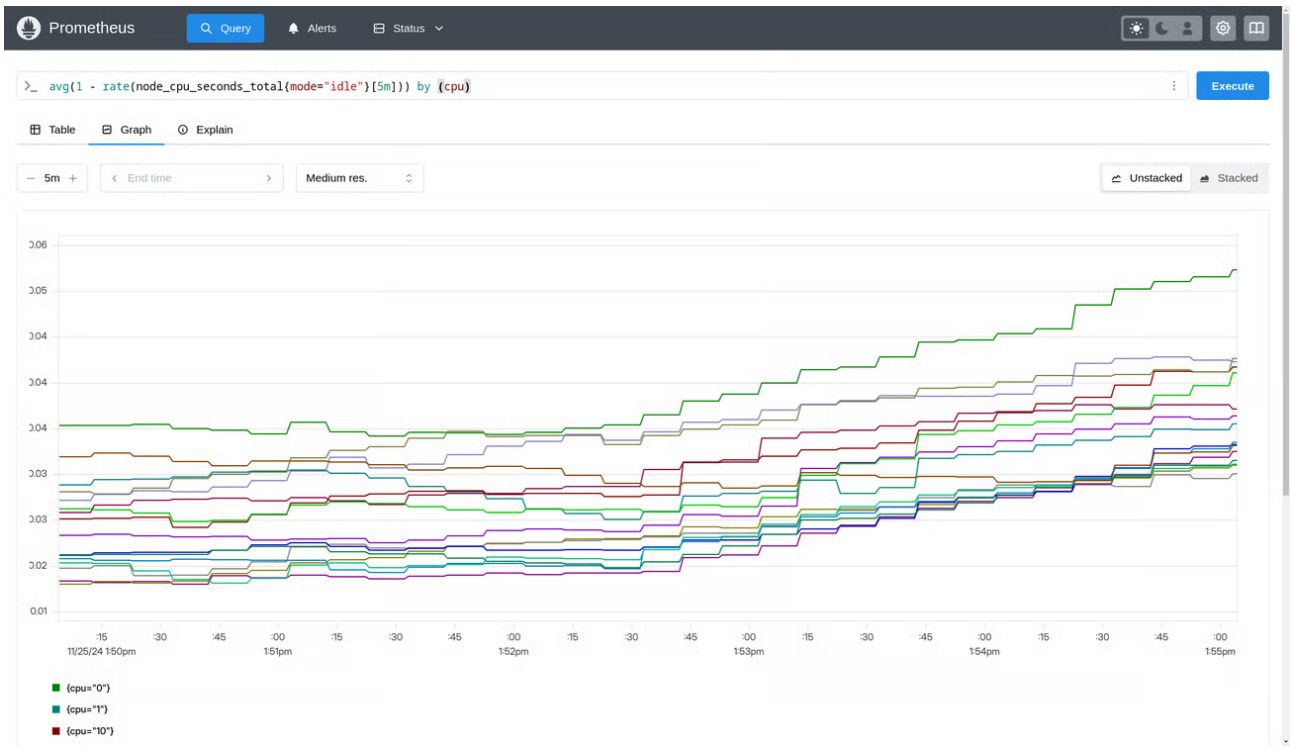
Unlike counters, gauge metrics produce a meaningful value when queried directly since they reflect the current state of the metric being monitored.

```
node_memory_MemAvailable_bytes
```




Aggregation functions like `avg()` or `sum()` can also be used to aggregate gauge values across multiple instances or labels, such as the query below which shows the average CPU usage rate as a fraction of each CPU core on a server:

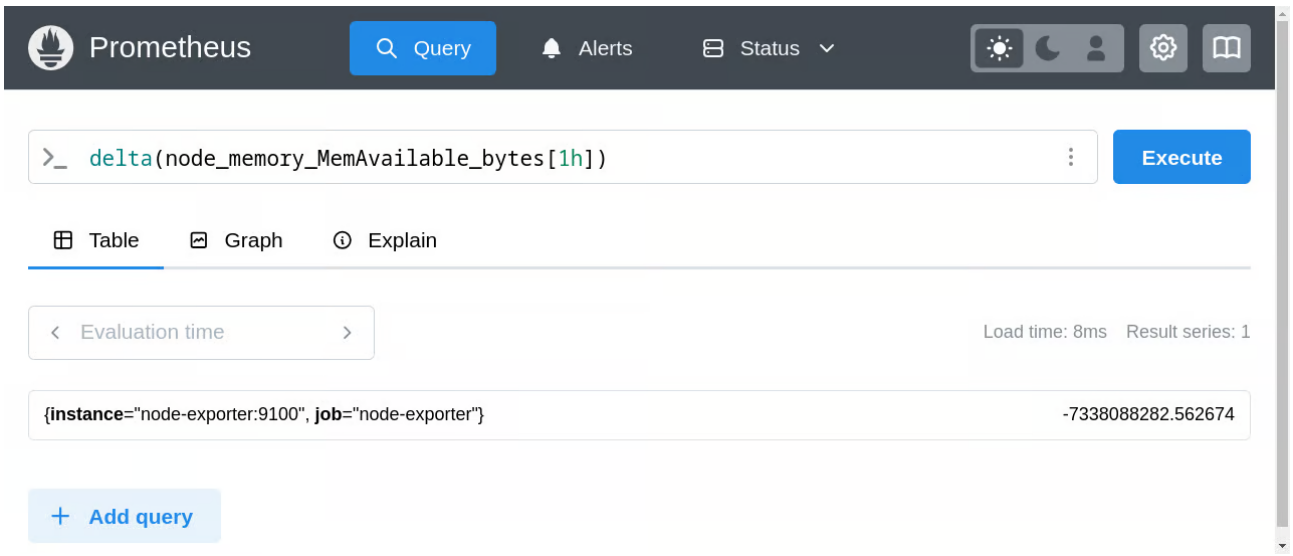
```
avg(1 - rate(node_cpu_seconds_total{mode="idle"}[5m])) by (cpu)
```




Another helpful function for gauges is delta(), which calculates the total change in a gauge value over a specific time window. To see how much available memory has changed in the last hour, you could use:

```
delta(node_memory_MemAvailable_bytes[1h])
```

When plotted on a graph, you'll see a visualization of how the available memory on your nodes has changed over time:




# Querying histograms

Histograms in Prometheus offer valuable insights into the distribution of your data by grouping observations into buckets.

They provide three key metric types: bucket time series (`_bucket`), sum time series (`_sum`), and count time series (`_count`).

Since these are counter metrics, you can apply functions like `rate()` and `increase()` to analyze their changes over time. For example:

```
rate(http_request_duration_seconds_bucket{le="1"}[5m])
```

This calculates the rate of requests with a duration less than or equal to 1 second over the past 5 minutes.

However, the most powerful function for histograms is `histogram_quantile()`. It estimates a specific quantile (percentile) from the bucket distribution:

`histogram_quantile(φ scalar, b instant-vector)`
Where `φ` is the quantile (e.g. 0.95 for the 95th percentile) and `b` is an instant vector of bucket values.

To calculate the 95th percentile request latency over five minutes, you would use:

```
histogram_quantile(0.95,
rate(http_request_duration_seconds_bucket[5m]))
```

This query first calculates the rate of change for each bucket over 5 minutes and then uses those rates to estimate the 95th percentile latency.

# Querying summaries

Summaries in Prometheus provide precomputed quantiles, allowing you to directly query specific percentiles without extra calculations.

For example, to get the 95th percentile of HTTP request durations, you can simply use:

```
http_request_duration_seconds{quantile="0.95"}
```

Summaries also expose `_sum` and `_count` metrics, which are counters. This means you can apply functions like `rate()` and `increase()` to analyze their change over time.

However, keep in mind that you cannot aggregate quantiles from summaries, as these are precomputed values. You can, however, aggregate the `_sum` and `_count` metrics if needed.

# Where can you use PromQL?

There are several places where you can use PromQL to query your Prometheus metrics including:

## 1. The Prometheus UI

Throughout this tutorial, we've typed PromQL queries into the Prometheus expression browser and viewed the query results directly as tables or graphs. This is a useful approach for testing and exploring raw metric data with rudimentary visualizations.

## 2. Alerting rules

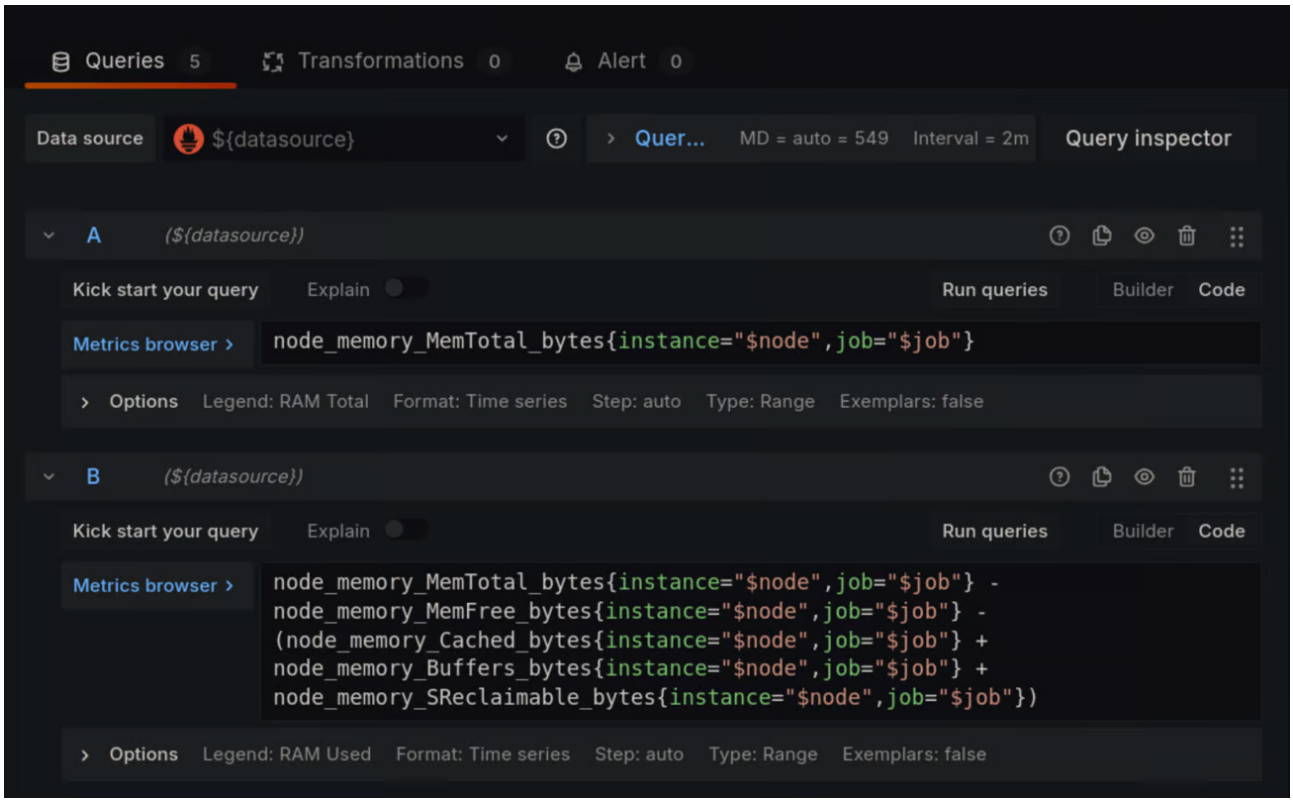
alerts.yml

```
groups:
- name: HighCPUAlert
  rules:
  - alert: HighCPUUsage
    expr: 100 - (avg by (instance)
(rate(node_cpu_seconds_total{mode="idle"}[5m])) * 100) > 80
    for: 10m
    labels:
      severity: critical
    annotations:
      summary: "High CPU usage on {{ $labels.instance }}"
      description: "CPU usage on {{ $labels.instance 1 }} is
above 80% for more than 10 minutes."
```

We've also seen a few examples of defining alerting conditions with PromQL expressions so that alerts are triggered when specific thresholds are breached or predefined patterns emerge.

You can find many more alerting examples here.

## 3. Grafana




Grafana is often paired with Prometheus for a more comprehensive visualization experience. You use PromQL within Grafana to query your Prometheus data source and create rich, informative dashboards with panels displaying various metrics and visualizations.

## 4. Other monitoring platforms


Several other monitoring and observability platforms have embraced PromQL or offer compatibility with it, allowing you to leverage your skills and knowledge across different tools.

For example, you can use it to query your metrics within Better Stack.