Click Here

Prometheus Functions and Their Application The Prometheus monitoring stack offers various functions to analyze data trends, with rate() being one of the most used but misunderstood. This function calculates the per-second average rate of change over time for any value. It is essential in predicting trends, especially when analyzing server request rates or CPU usage. Understanding how rate() works internally is crucial before applying it to specific metrics. PromQL uses two types of vectors: range and instant. Range vectors have a time dimension, making them necessary for trend analysis with functions like rate(). The optimal time range depends on the use case and desired level of detail in data. Prometheus Functions Overview Before diving into Prometheus functions, let's establish some basic concepts. PromQL is a custom query language that filters and searches through Prometheus' time series data. Every metric has four components: name, labels (key-value pairs for identification), value (64-bit floating point number), and timestamp with millisecond precision. Prometheus also includes three primary data types in its expression language: scalar (representing a single floating-point value), instant vector (a set of time series data at a specific moment), and range vector (a set of time series data over a period). The client libraries categorize metrics into four types: counter (increasing values that reset on restart), gauge (counting or measuring rising/falling values), histogram (sampling observations in buckets for summing up observed data), and unknown (unaccounted for). For a more comprehensive understanding of how to apply these functions, especially rate(), it is recommended to sign up for the Hosted Graphite free trial or schedule a demo. Similar to a histogram, this metric records observations and observed values while calculating configurable quantities within a sliding time window. A Prometheus query can return either an instant vector or a range vector depending on the metric type and desired result. Let's discuss Prometheus functions, which are used in the PromQL language to query a Prometheus database. These functions will be explored further in the next section. To understand how the rate() function works, let's examine its arguments. In PromQL, there are two types of arguments: range and instant vectors. Range vectors define a time dimension, whereas instant vectors do not. The rate() function requires at least two points of data to calculate changes, which is why it takes an argument of the range type. If less than two samples are available, no results will be returned. To determine the optimal time range for the rate() function, consider using a minimum of twice the scrape interval. However, this can result in very sharp changes being reflected in the results, making it harder to spot spikes or trends. A more practical approach is to define a variable for the time range within Grafana, such as 1m, 5m, 15m, or 2h. This allows you to choose the best value for your specific use case when trying to identify spikes or trends. Alternatively, you can use the $__interval variable in Grafana, which divides the time range by the step's size. While this includes all data points between each step, it also has its limitations. The rate function calculates the difference in value over a specified time interval and divides it by that interval, considering all data points. This is useful for constantly increasing metrics like counters but can be misleading when used with gauges. The rate function automatically adjusts for resets, making it suitable for counters. However, if another aggregation is used with rate(), it must be applied first to avoid counter reset issues. Using rate() with gauges can lead to incorrect results due to misinterpretation of value decreases as counter resets. For example, calculating the rate over 60 seconds may yield different results when compared to multiplying the average rate by the interval. This discrepancy occurs because real-world data rarely aligns perfectly due to random delays between scrapes or failed data points. Understanding that rate() performs extrapolation is crucial for avoiding future issues. Given article text here Looking at the rate() function in a given context, even if it is applied multiple times within its time range. In such cases, the rate() calculates the value with the available data and subsequently fills any gaps using either the first or last data points to provide more accurate insights. Therefore, results may vary despite having all integer values, making this function suitable for spotting trends and spikes in data while alerting on any unusual changes. Given article text here the past 5 minutes, e.g.: increase(http_requests_total{job="api-server"}[5m]) Linear regression Given a range vector and a scalar t, the predict_linear() Prometheus function uses simple linear regression to predict the future value of the time series t seconds. Regular expressions The label_replace() Prometheus function searches through time series to find one that matches the given regex (regular expression), and then. Calculating percentiles Given a histogram, the histogram_quantile() Prometheus function. The highest bucket in the histogram must have an upper bound of +Inf (positive infinity). Calculating differences Given a range vector, the delta() Prometheus function calculates the difference between two quantities. For example, to calculate the difference in CPU temperature between now and 2 hours ago: delta(cpu_temp_celsius{host="zeus"}[2h]) To enhance your use of Prometheus functions, you can also integrate Prometheus with Grafana, a web application for data analytics and visualization. MetricFire's hosted Grafana service makes it easy for any Prometheus user to enjoy high-quality, informative data visualizations. When you choose MetricFire, you also get access to our Hosted Graphite data source and platform and 24/7 support for all your Graphite and Grafana needs. Sign up today! How MetricFire Can Help MetricFire is a cloud infrastructure and application monitoring platform that makes understanding your data at a glance simple. We offer a Hosted Graphite service that handles issues such as scaling, data storage, and support and maintenance—so that you can spend more time on the metrics and less on the technical details. To learn more, check out the Prometheus posts on our blog or book a demo to discuss your business needs and objectives. Want to get started with your monitoring right away? The MetricFire free trial is an excellent way to explore what you can do with hosted Graphite. Community Guides Questions Comparisons Blog Docs Understanding the rate and increase functions in Prometheus is crucial for accurately analyzing time-series data. Here's a detailed breakdown of both functions and their differences to ensure you have a solid grasp of how they work. Centralize & visualize your logs. Query everything with SQL. Rate Function Definition: The rate function calculates the average per-second rate of increase of a counter over a specified time range. Usage: It is primarily used for counters, which are metrics that only increase (or reset). The function is written as: rate(metric_name[time_range]) Example: If you have a counter that tracks the number of HTTP requests, you could use: rate(http_requests_total[5m]) This returns the average number of requests per second over the last 5 minutes. Key Points: The result is a rate (requests per second) based on the specified time interval. It handles counter resets automatically (e.g., when the service restarts). Useful for generating smooth graphs over time. Increase Function Definition: The increase function The total metric increase within a specific timeframe, commonly referred to as `increase(metric_name[time_range])`. For instance, considering HTTP request metrics, you could utilize: `increase(http_requests_total[5m])` This yields the cumulative number of requests that occurred over the preceding 5 minutes. Essential Points: * The outcome represents an absolute count of the metric increase within the specified time frame. * It automatically handles counter resets. * Suitable for calculating total occurrences (e.g., overall requests) over a designated period. Key Differences: * `rate`: Displays the per-second rate of increase (e.g., requests per second). * `increase`: Returns the total increase over the specified time (e.g., total requests). Use Cases: * Employ `rate` when evaluating speed or performance changes (e.g., request arrival frequency). * Utilize `increase` when seeking to determine the overall amount of something that occurred (e.g., total requests within a timeframe).

Prometheus rate increase 区别. Prometheus rate by. Prometheus rate 和 increase. Prometheus rate irate increase.