PromQL for Prometheus: A Complete Guide

# 1. Introduction to Prometheus

Prometheus is an open-source monitoring and alerting toolkit designed for reliability and scalability. It collects and stores metrics as time series data, recording real-time performance metrics from various services and systems. Prometheus uses a time series database where each data point is stored with a timestamp and key-value pairs known as labels.

## 1.1 Key Concepts

- Metrics: Numerical measurements over time, such as CPU usage, memory usage, HTTP request count, etc.  
- Time Series: A set of data points indexed by time, often representing a single metric (e.g., http\_requests\_total).  
- Labels: Key-value pairs that provide additional metadata about a metric (e.g., status="200").  
- PromQL: Prometheus Query Language, used to query time series data stored in Prometheus.

# 2. Basics of PromQL

PromQL is used to query time series data stored in Prometheus. It supports:  
- Selecting Time Series: Through metric names and labels.  
- Aggregation: Grouping and summarizing data.  
- Operations: Arithmetic and logical operations on time series data.  
- Functions: Built-in functions for operations like rate calculations, averages, etc.

## 2.1 Simple Query Example

A basic query in PromQL looks like this:  
```promQL  
http\_requests\_total  
```  
This will return all time series of the http\_requests\_total metric.

## 2.2 Filtering by Labels

You can filter results based on labels. For example:  
```promQL  
http\_requests\_total{status="200"}  
```  
This query filters the http\_requests\_total metric for only those series where the status label is "200".

# 3. PromQL Operators and Functions

## 3.1 Arithmetic Operators

PromQL supports standard arithmetic operators:  
- `+`, `-`, `\*`, `/`  
For example:  
```promQL  
http\_requests\_total{status="200"} / http\_requests\_total  
```  
This would return the ratio of successful HTTP requests (status="200") to total HTTP requests.

## 3.2 Aggregation Operators

Prometheus provides aggregation operators to summarize data. Common ones include:  
- `sum()`  
- `avg()`  
- `min()`  
- `max()`  
- `count()`  
Example:  
```promQL  
sum(rate(http\_requests\_total{status="200"}[5m]))  
```  
This sums the rate of successful HTTP requests over the last 5 minutes.

## 3.3 Range Vectors and Instant Vectors

- \*\*Instant Vector\*\*: A single time series at a given point in time.  
- \*\*Range Vector\*\*: A set of time series over a range of time (e.g., http\_requests\_total[5m]).  
Range vectors allow operations over a time window:  
```promQL  
rate(http\_requests\_total{status="200"}[5m])  
```  
This returns the per-second rate of http\_requests\_total over the last 5 minutes.

# 4. Time Series Manipulation

## 4.1 Counting Events

You can count the number of events over a period with:  
```promQL  
count(http\_requests\_total{status="500"})  
```  
This counts how many HTTP 500 errors have occurred.

## 4.2 Rate Calculation

To calculate the rate of events over a period:  
```promQL  
rate(http\_requests\_total[1m])  
```  
This calculates the per-second rate of requests over the last minute.

# 5. Advanced PromQL Queries

## 5.1 Using `by` and `without` for Grouping

You can group results by specific labels using `by`:  
```promQL  
sum(rate(http\_requests\_total[5m])) by (status)  
```  
This sums the rate of requests over the last 5 minutes and groups by the `status` label.

## 5.2 Subqueries

Subqueries allow for more advanced calculations:  
```promQL  
avg(rate(http\_requests\_total[5m])) by (status) [1h]  
```  
This calculates the average rate of requests per second over the last 5 minutes, then over the last hour.

# 6. PromQL in Practice

## 6.1 Create Real-World Queries

- Query for the rate of requests with different status codes:  
```promQL  
rate(http\_requests\_total{status="200"}[5m])  
```  
- Query for the maximum memory usage across all nodes:  
```promQL  
max(memory\_usage\_bytes) by (instance)  
```  
- Calculate the average memory usage over the last hour for each instance:  
```promQL  
avg(memory\_usage\_bytes[1h]) by (instance)  
```

## 6.2 Setting Up Alerts

PromQL is also used for setting up alerts in Prometheus. For example, creating an alert for when HTTP request errors exceed a certain threshold:  
```promQL  
sum(rate(http\_requests\_total{status="500"}[5m])) > 10  
```  
This creates an alert if there are more than 10 HTTP 500 errors per second over the last 5 minutes.

# 7. Best Practices for PromQL Queries

## 7.1 Query Optimization

- Use range vectors wisely; avoid large time windows unless necessary.  
- Limit the use of `count()` and `avg()` for very high cardinality data.  
- Avoid frequent use of `by` when grouping on many labels.

## 7.2 Avoiding Common Pitfalls

- Watch out for queries that could result in high cardinality (e.g., grouping by many labels).  
- Be mindful of how metrics are scraped to avoid missing crucial data.  
- Ensure queries return meaningful data and not just raw numbers.

# 8. Conclusion

PromQL is a powerful query language that allows you to extract meaningful insights from Prometheus time series data. By understanding basic queries, aggregation, and time series manipulation, you can perform complex analysis and set up effective alerts to monitor the health of your systems.