17. Ranking Nodes

37. Nodes ranked by memory usage:

```
promql

  rank(aerospike_node_memory_used) by (node)
```

---

18. Correlation and Ratios

38. Compare CPU and memory usage per node:

```
promql

  aerospike_node_cpu_usage / aerospike_node_memory_used
```

---

19. Custom Derived Metrics

39. Average write latency:

```
promql

  rate(aerospike_node_write_latency_sum[5m]) / rate(aerospike_node_write_latency_count[5
  m])
```

I can expand these with even more details if you'd like! Which area should I focus on?

apart from this just explain all the function in prometheus from order wise like by avg rate

Here's an **ordered list of Prometheus functions**, categorized by their functionality, with clear explanations and examples to understand how each is used. I'll use Aerospike-related examples wherever applicable.

---

1. Aggregation Functions

These functions aggregate data across labels or over time.

- `sum()`
  Sums up the values of a metric across dimensions.
  Example:

```promql
sum(aerospike_node_memory_used)
```

Total memory used across all nodes.

- `avg()`
  Computes the average of values.
  Example:

```promql
avg(rate(aerospike_node_read_ops_total[5m]))
```

Average read operations per second over the last 5 minutes.

- `min()`
  Finds the minimum value across dimensions.
  Example:

```promql
min(aerospike_node_memory_used)
```

Node with the lowest memory usage.

- `max()`
  Finds the maximum value across dimensions.
  Example:

```promql
max(aerospike_node_memory_used)
```

Node with the highest memory usage.

- `count()`
  Counts the number of time series matching a query.
  Example:

```promql
count(aerospike_node_memory_used)
```

Number of nodes reporting memory usage.

- `stddev()`
  Computes the standard deviation of values.

Example:

```promql
stddev(aerospike_node_memory_used)
```

Variability in memory usage across nodes.

- `stdvar()`
  Computes the variance of values.
  Example:

```promql
stdvar(aerospike_node_memory_used)
```

- `count_values()`
  Counts the occurrences of each unique value.
  Example:

```promql
count_values("memory", aerospike_node_memory_used)
```

- `quantile()`
  Computes a specific quantile (e.g., median, 95th percentile).
  Example:

```promql
quantile(0.95, aerospike_node_memory_used)
```

---

2. Rate and Increase

These functions analyze how metrics change over time.

- `rate()`
  Calculates the per-second rate of change for a counter metric.
  Example:

```promql
rate(aerospike_node_read_ops_total[5m])
```

Rate of read operations per second over the last 5 minutes.

- `irate()`
  Calculates the instantaneous per-second rate of change.
  Example:

```promql
irate(aerospike_node_write_ops_total[5m])
```

- `increase()`
  Computes the total increase in a counter metric over a given time.
  Example:

```promql
increase(aerospike_node_read_ops_total[1h])
```

  Total read operations in the last hour.

- `delta()`
  Computes the difference between the first and last value in a time range.
  Example:

```promql
delta(aerospike_node_disk_used[1h])
```

- `idelta()`
  Computes the instantaneous difference between two consecutive samples.
  Example:

```promql
idelta(aerospike_node_disk_used[1h])
```

---

3. Histogram and Summary Functions

These functions work with histogram metrics.

- `histogram_quantile()`
  Calculates a quantile (e.g., 90th percentile) from histogram buckets.
  Example:

```promql
histogram_quantile(0.90, sum(rate(aerospike_node_read_latency_bucket[5m])) by (le))
```

4. Time Series Manipulation

These functions work on time-series data, modifying or filtering them.

- **`avg_over_time()`**
  Computes the average of a time-series over a range.
  Example:

```promql
avg_over_time(aerospike_node_memory_used[1h])
```

- **`min_over_time()`**
  Computes the minimum value of a time-series over a range.
  Example:

```promql
min_over_time(aerospike_node_memory_used[1h])
```

- **`max_over_time()`**
  Computes the maximum value of a time-series over a range.
  Example:

```promql
max_over_time(aerospike_node_memory_used[1h])
```

- **`sum_over_time()`**
  Computes the sum of a time-series over a range.
  Example:

```promql
sum_over_time(aerospike_node_read_ops_total[1h])
```

- **`count_over_time()`**
  Counts the number of samples in a time-series over a range.
  Example:

```promql
count_over_time(aerospike_node_memory_used[1h])
```

- **`quantile_over_time()`**
  Computes a quantile over a range.
  Example:

```promql
quantile_over_time(0.95, aerospike_node_memory_used[1h])
```

---

5. Sorting and Ranking

These functions sort or rank metrics.

- **`topk()`**
  Returns the top `k` metrics by value.
  Example:

```promql
topk(3, aerospike_node_memory_used)
```

- **`bottomk()`**
  Returns the bottom `k` metrics by value.
  Example:

```promql
bottomk(3, aerospike_node_memory_used)
```

- **`sort()`**
  Sorts metrics by value in ascending order.
  Example:

```promql
sort(aerospike_node_memory_used)
```

- **`sort_desc()`**
  Sorts metrics by value in descending order.
  Example:

```promql
sort_desc(aerospike_node_memory_used)
```

---

6. Conditional Functions

- **`abs()`**
  Returns the absolute value of each metric.
  Example:

```promql
abs(aerospike_node_memory_used)
```

- **`ceil()`**
  Rounds values up to the nearest integer.
  Example:

```promql
ceil(aerospike_node_memory_used)
```

- **`floor()`**
  Rounds values down to the nearest integer.
  Example:

```promql
floor(aerospike_node_memory_used)
```

- **`clamp_min()`**
  Limits values to a minimum threshold.
  Example:

```promql
clamp_min(aerospike_node_memory_used, 1000)
```

- **`clamp_max()`**
  Limits values to a maximum threshold.
  Example:

```promql
clamp_max(aerospike_node_memory_used, 5000)
```

---

7. Label Manipulation

- `label_replace()`
  Replaces or creates a label in a metric.
  Example:

```
promql

  label_replace(aerospike_node_memory_used, "new_label", "value", "existing_label", ".*")
```

- `label_join()`
  Joins multiple labels into a single label.
  Example:

```
promql

  label_join(aerospike_node_memory_used, "joined_label", ",", "label1", "label2")
```

---

8. Vector Matching

- `on()` and `ignoring()`
  These modify how metrics are matched by labels during binary operations.
  Example:

```
promql

  aerospike_node_memory_used / on(node) aerospike_node_memory_limit
```

- `group_left` and `group_right`
  Modify the grouping in vector operations.
  Example:

```
promql

  aerospike_node_memory_used / on(node) group_left aerospike_node_memory_limit
```

This covers all core Prometheus functions with examples. Let me know if you want a deep dive into any specific group!

now take an example data scenario and explain about about each function in below
sum(): Adds values across dimensions.
avg(): Computes the average.
max(): Finds the maximum value.
min()()

```
count()
rate
```

Let's take a hypothetical **Aerospike metrics dataset** and use the functions `sum()`, `avg()`, `max()`, `min()`, `count()`, and `rate()` to explain each function with clarity.

Scenario: Hypothetical Data

We have an Aerospike cluster with **3 nodes** (`node1`, `node2`, `node3`) and two namespaces (`namespace1`, `namespace2`). Below is the sample data collected:

Metric: aerospike_node_read_ops_total

This metric represents the **total read operations** on each node (a cumulative counter):

Time (T)	node	namespace	Value (`aerospike_node_read_ops_total`)
T1	node1	namespace1	1000
T1	node1	namespace2	800
T1	node2	namespace1	1500
T1	node2	namespace2	1200
T1	node3	namespace1	2000
T1	node3	namespace2	1000

Function Examples with Explanations

1. `sum()`: Adds the values across all dimensions (e.g., node, namespace).

- **Goal**: Total read operations across all nodes and namespaces.
- **Query**:

```
promql


  sum(aerospike_node_read_ops_total)
```

- **Result**:
  $1000 + 800 + 1500 + 1200 + 2000 + 1000 = 7800$
- **Interpretation**: The total number of read operations across all nodes and namespaces is **7,800**.
- **Another Example (Group by namespace)**:

```
promql


  sum(aerospike_node_read_ops_total) by (namespace)
```

Result:

- `namespace1`: $1000 + 1500 + 2000 = 4500$
- `namespace2`: $800 + 1200 + 1000 = 3000$

---

2. `avg()`: Computes the average across all dimensions.

- **Goal**: Find the average number of read operations across all nodes and namespaces.
- **Query**:

```
promql


  avg(aerospike_node_read_ops_total)
```

- **Result**:
  $\frac{1000+800+1500+1200+2000+1000}{6} = 1300$
- **Interpretation**: The average number of read operations per node-namespace pair is **1,300**.
- **Another Example (Group by node)**:

```
promql


  avg(aerospike_node_read_ops_total) by (node)
```

Result:

- `node1`: $\frac{1000+800}{2} = 900$
- `node2`: $\frac{1500+1200}{2} = 1350$
- `node3`: $\frac{2000+1000}{2} = 1500$

---

3. `max()`: Finds the maximum value across dimensions.

- **Goal**: Find the maximum number of read operations across all nodes and namespaces.
- **Query**:

```
promql


  max(aerospike_node_read_ops_total)
```

- **Result**:
  $2000 \text{ (from node3, namespace1)}$
- **Interpretation**: The maximum read operations for a single node-namespace combination is **2,000**.
- **Another Example (Group by namespace)**:

```
promql

  max(aerospike_node_read_ops_total) by (namespace)
```

Result:

- `namespace1`: 2000 (node3)
- `namespace2`: 1200 (node2)

---

4. `min()`: Finds the minimum value across dimensions.

- **Goal**: Find the minimum number of read operations across all nodes and namespaces.
- **Query**:

```
promql

  min(aerospike_node_read_ops_total)
```

- **Result**:
  800 (from node1, namespace2)
- **Interpretation**: The minimum read operations for a single node-namespace combination is **800**.
- **Another Example (Group by node)**:

```
promql

  min(aerospike_node_read_ops_total) by (node)
```

Result:

- `node1`: 800
- `node2`: 1200
- `node3`: 1000

---

5. `count()`: Counts the number of unique time-series.

- **Goal**: Find how many node-namespace pairs are reporting read operations.
- **Query**:

```
promql

  count(aerospike_node_read_ops_total)
```

- **Result**:
  6 (one for each node-namespace combination)

- **Another Example (Group by namespace)**:

```
promql


  count(aerospike_node_read_ops_total) by (namespace)
```

Result:

- `namespace1`: 3 (node1, node2, node3)
- `namespace2`: 3 (node1, node2, node3)

---

6. `rate()`: Calculates the per-second rate of change for a counter metric.

- **Goal**: Calculate the per-second read operations for each node and namespace over the last 5 minutes.
- **Query**:

```
promql


  rate(aerospike_node_read_ops_total[5m])
```

- **Result**:
  Assuming the following increases over 5 minutes:
  - `node1, namespace1`: +50
  - `node1, namespace2`: +40
  - `node2, namespace1`: +100
  - `node2, namespace2`: +60
  - `node3, namespace1`: +150
  - `node3, namespace2`: +80

  Rate = Increase / Duration = Increase / 300 seconds:
  - `node1, namespace1`: $\frac{50}{300} = 0.166$
  - `node1, namespace2`: $\frac{40}{300} = 0.133$
  - `node2, namespace1`: $\frac{100}{300} = 0.333$
  - `node2, namespace2`: $\frac{60}{300} = 0.200$
  - `node3, namespace1`: $\frac{150}{300} = 0.500$
  - `node3, namespace2`: $\frac{80}{300} = 0.266$

Summary of Results

Function	Description	Result
`sum()`	Total read operations	7,800
`avg()`	Average read operations	1,300
`max()`	Maximum read operations	2,000
`min()`	Minimum read operations	800

Function	Description	Result
`count()`	Number of node-namespace pairs	6
`rate()`	Per-second read operations rate	Calculated per pair (see above)

Let me know if you'd like more examples or deeper explanations for any of these!

same like add someother important fucntion combineed

Let's extend the explanation with **combined Prometheus functions**, including some other important and frequently used ones. We'll build on the same Aerospike dataset from before.

Dataset Recap

Time (T)	node	namespace	Value (`aerospike_node_read_ops_total`)
T1	node1	namespace1	1000
T1	node1	namespace2	800
T1	node2	namespace1	1500
T1	node2	namespace2	1200
T1	node3	namespace1	2000
T1	node3	namespace2	1000

---

1. Using `sum()` with `rate()`

Goal: Total read operations per second across all nodes and namespaces.

- **Query**:

```promql
promql


  sum(rate(aerospike_node_read_ops_total[5m]))
```

- **Explanation**:
  - `rate(aerospike_node_read_ops_total[5m])`: Computes the per-second rate of change for each node-namespace pair.
  - `sum()`: Aggregates these rates to calculate the total across all nodes and namespaces.
- **Result**: Total read operations per second across all nodes.

---

2. Using `avg()` with `rate()`

Goal: Average read operations per second per node across namespaces.

- **Query**:

```promql
avg(rate(aerospike_node_read_ops_total[5m])) by (node)
```

- **Explanation**:
  - `rate(aerospike_node_read_ops_total[5m])`: Computes the per-second rate of change for each node-namespace pair.
  - `avg() by (node)`: Calculates the average rate for each node across all namespaces.
- **Result**:
  - `node1`: Average rate per second across `namespace1` and `namespace2`.
  - Similarly for `node2` and `node3`.

---

3. Using `max()` and `rate()`

Goal: Find the node-namespace pair with the highest read operation rate.

- **Query**:

```promql
max(rate(aerospike_node_read_ops_total[5m]))
```

- **Explanation**:
  - `rate(aerospike_node_read_ops_total[5m])`: Computes the per-second rate of change for each node-namespace pair.
  - `max()`: Returns the highest rate among all pairs.
- **Result**: Maximum read operation rate (e.g., from `node3, namespace1`).

---

4. Using `bottomk()` with `sum()`

Goal: Find the 2 nodes with the least total read operations.

- **Query**:

```promql
bottomk(2, sum(rate(aerospike_node_read_ops_total[5m])) by (node))
```

- **Explanation**:
  - `rate(aerospike_node_read_ops_total[5m])`: Calculates the per-second rate for each node-namespace pair.
  - `sum() by (node)`: Aggregates the rate across all namespaces for each node.
  - `bottomk(2, ...)`: Selects the 2 nodes with the least total read operation rates.
- **Result**: The 2 nodes with the lowest total read operation rates.

---

5. Using `histogram_quantile()` and `rate()`

Goal: Calculate the 95th percentile latency for read operations across all nodes.

- **Dataset Update**: Assume we have a histogram metric, `aerospike_read_latency_bucket`, representing read latencies.
- **Query**:

```
promql


  histogram_quantile(0.95, sum(rate(aerospike_read_latency_bucket[5m])) by (le))
```

- **Explanation**:
  - `rate(aerospike_read_latency_bucket[5m])`: Computes the per-second rate of latency events for each latency bucket (`le`).
  - `sum() by (le)`: Aggregates the latency bucket data across nodes.
  - `histogram_quantile(0.95, ...)`: Calculates the 95th percentile latency.
- **Result**: The 95th percentile latency for read operations.

---

6. Using `clamp_max()` with `rate()`

Goal: Limit read operation rates to a maximum threshold of 100 operations per second.

- **Query**:

```
promql


  clamp_max(rate(aerospike_node_read_ops_total[5m]), 100)
```

- **Explanation**:
  - `rate(aerospike_node_read_ops_total[5m])`: Computes the per-second rate of change for each node-namespace pair.
  - `clamp_max(..., 100)`: Ensures that no value exceeds 100.
- **Result**: Read operation rates, but capped at 100 for any pair exceeding this threshold.

---

7. Using `label_replace()` with `sum()`

Goal: Add a new label (`cluster`) based on an existing label (`node`) while summing read operations.

- **Query**:

```
promql


  sum(aerospike_node_read_ops_total) by (cluster)
```

  With label_replace:

```
promql


  label_replace(sum(aerospike_node_read_ops_total), "cluster", "cluster_$1", "node", "(.
  *)")
```

- **Explanation**:
  - `sum(aerospike_node_read_ops_total)`: Aggregates read operations across nodes.
  - `label_replace()`: Adds a new label `cluster` by transforming the `node` label.
- **Result**: Aggregated read operations with the new `cluster` label.

---

8. Using `topk()` and `rate()`

Goal: Find the 3 namespaces with the highest total read operation rates.

- **Query**:

```
promql


  topk(3, sum(rate(aerospike_node_read_ops_total[5m])) by (namespace))
```

- **Explanation**:
  - `rate(aerospike_node_read_ops_total[5m])`: Calculates the per-second rate for each node-namespace pair.
  - `sum() by (namespace)`: Aggregates the rate across nodes for each namespace.
  - `topk(3, ...)`: Selects the 3 namespaces with the highest total rates.
- **Result**: Top 3 namespaces with the most read operations.

---

9. Using `stddev()` with `rate()`

Goal: Calculate the standard deviation of read operation rates across nodes.

- **Query**:

```
promql


  stddev(rate(aerospike_node_read_ops_total[5m])) by (node)
```

- **Explanation**:
  - `rate(aerospike_node_read_ops_total[5m])`: Computes the per-second rate for each node-namespace pair.
  - `stddev() by (node)`: Calculates the standard deviation of rates for each node across namespaces.
- **Result**: Variability of read operation rates across namespaces for each node.

---

10. Using `quantile_over_time()`

Goal: Find the 90th percentile of read operations over the last hour for each node.

- **Query**:

```promql
quantile_over_time(0.90, aerospike_node_read_ops_total[1h])
```

- **Explanation**:
  - `aerospike_node_read_ops_total[1h]`: Fetches the total read operations for the last 1 hour.
  - `quantile_over_time(0.90, ...)`: Calculates the 90th percentile value for this range.
- **Result**: 90th percentile of read operations for each node.

Summary of Combined Functions and Goals

Function	Goal
`sum() + rate()`	Total read operation rates across all nodes and namespaces.
`avg() + rate()`	Average read operation rate per node across namespaces.
`max() + rate()`	Maximum rate of read operations for a single node-namespace pair.
`bottomk() + sum()`	Identify nodes with the least total read operations.
`histogram_quantile()`	Calculate 95th percentile latency for read operations.
`clamp_max()`	Limit read operation rates to a threshold.
`label_replace()`	Add new labels or modify existing ones.
`topk() + sum()`	Identify top namespaces with the highest read operations.
`stddev() + rate()`	Measure variability of read operation rates across nodes.
`quantile_over_time()`	Calculate percentiles over time for read operations.

Let me know if you'd like further examples or clarification!

Messages beyond this point are only visible to you