How to Measure PromQL/MetricsQL Expression Complexity

Roman Khavronenko | github.com/hagen1778

Roman Khavronenko

Software engineer with experience in distributed systems, monitoring and high-performance services.

Co-founder of VictoriaMetrics

https://github.com/hagen1778

https://twitter.com/hagen1778

VICTORIA METRICS

The High Performance
Open Source Time Series Database & Monitoring Solution

13k stars org
200+ contributors

Apache 2.0 license

Grammarly, CERN,
Roblox, Adidas, Wix

Use cases	Monitoring, alerting	Monitoring, alerting
Is open-source	Yes	Yes
Query language	PromQL	MetricsQL
Grafana integration	Yes	Yes
Scalability	Vertical	Horizontal and Vertical

Developer waiting for Grafana dashboard to load…

Prometheus Alerts Graph Status Help

Slow query:

Use local time | Enable query history | Enable autocomplete | Enable highlighting | Enable linter

```
max(max_over_time(node_systemd_unit_state[3d]))
```

Execute

Load time: 3426ms Resolution: 14s Result series: 1

Table Graph

Prometheus Alerts Graph Status Help

Fast query:

Use local time | Enable query history | Enable autocomplete | Enable highlighting | Enable linter

```
max(max_over_time(node_cooling_device_cur_state[3d]))
```

Execute

Load time: 53ms Resolution: 14s Result series: 1

Table Graph

Evaluation time

{}	0

Because those queries scan different amounts of data!

ChatGPT 4o

R

How to optimize SQL query?

Optimizing SQL queries is crucial for improving the performance of your database operations. Here are some general tips and techniques to help you optimize your SQL queries:

1. Indexing

How to optimize SQL query?

```
> SELECT * FROM bar;
```

1. **Use Indexes**, but avoid unnecessary Indexes
2. Select only **necessary columns**
3. Use **WHERE** - filter data as early as possible
4. **Database design**: normalize or denormalize data
5. **Partition tables** into smaller, manageable pieces

How to optimize SQL query?

Optimized query:

```
SELECT foo FROM bar                # select one column
WHERE date = '2024-06-01' # partition by time
AND user = 'baz';                  # index by user
```

Can we apply the same tips to PromQL/MetricsQL?

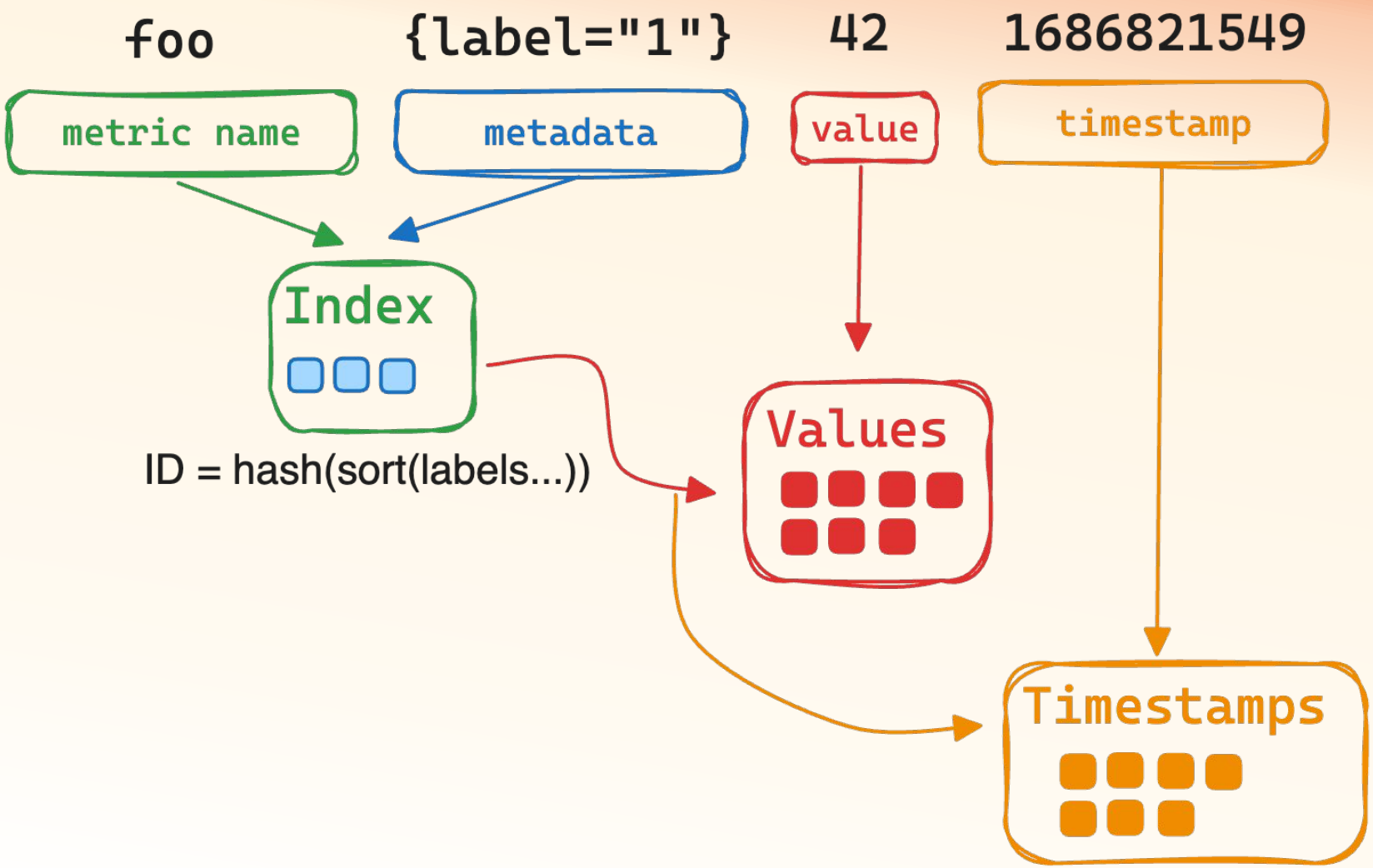
1. ⊘ Use Indexes, but avoid unnecessary Indexes
2. ⊘ Select only necessary columns
3. ✔ Use WHERE - filter data as early as possible
4. ⊘ Database design: normalize or denormalize data
5. ⊘ Partition tables into smaller, manageable pieces

Data model in Prometheus/VictoriaMetrics

```
foo{label="1"} 42 1686821549
foo{label="2"} 12 1686821549
foo{label="3"} 24 1686821549
foo{label="4"} 8  1686821549
```

foo	{label="1"}	42	1686821549
metric name	metadata	value	timestamp

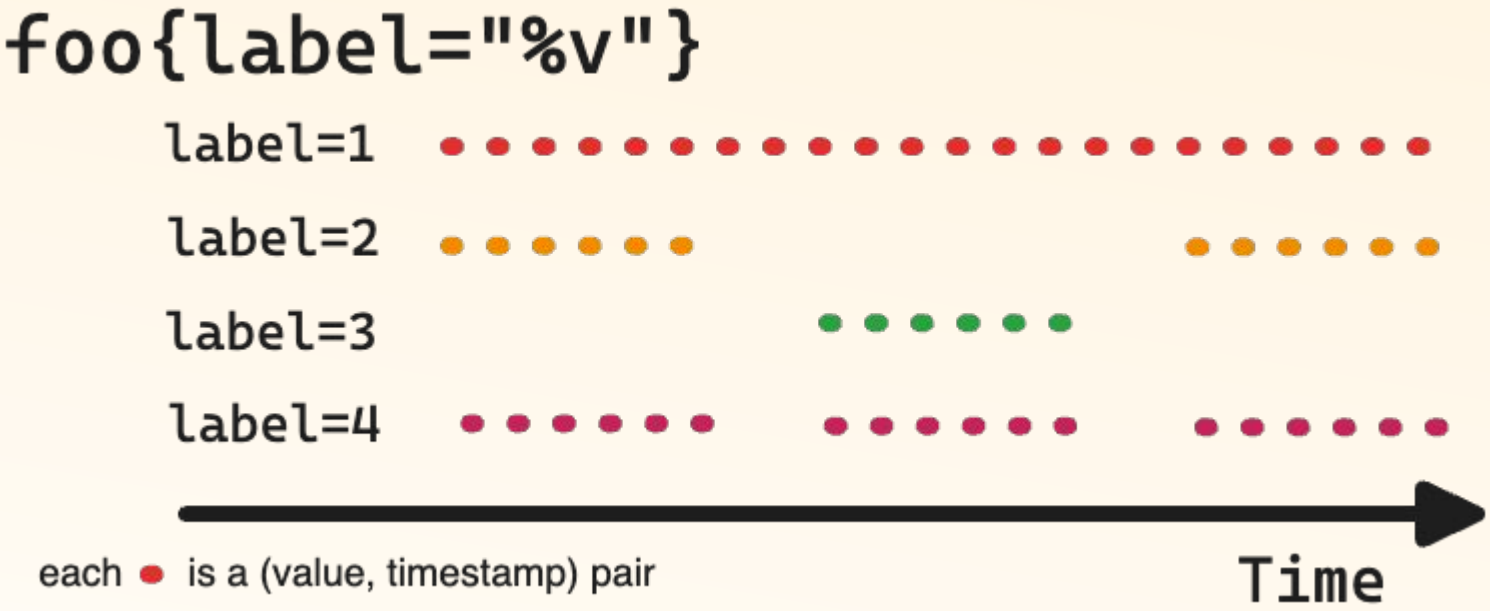
https://docs.victoriametrics.com/keyconcepts/#data-model


foo
{label="1"}
42
1686821549
metric name
metadata
value
timestamp
Index
ID = hash(sort(labels...))
Values
Timestamps

Data model in Prometheus/VictoriaMetrics

1. Data model is pre-defined and **can't be changed**
2. **Indexes** are created automatically
3. Data blocks are **partitioned by time**
4. The stored data types are strings (**name** and **metadata**) and numerics (**value** and **timestamp**)

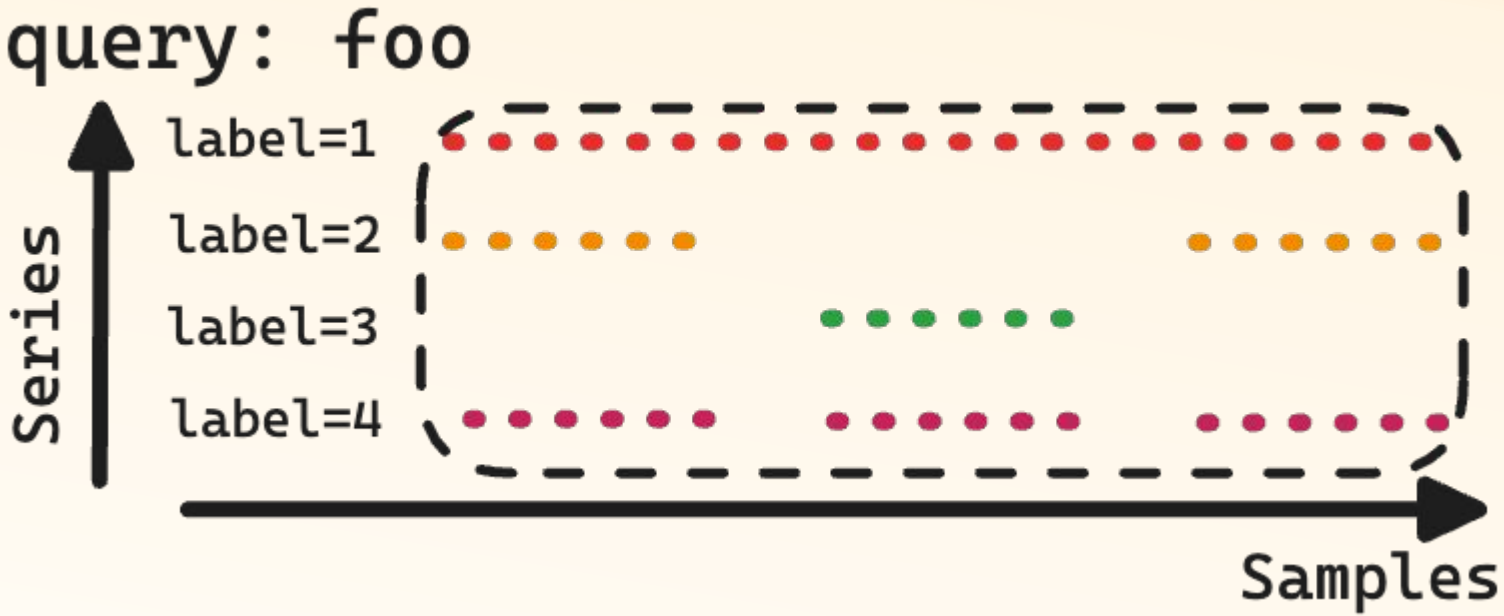
What is a time series?



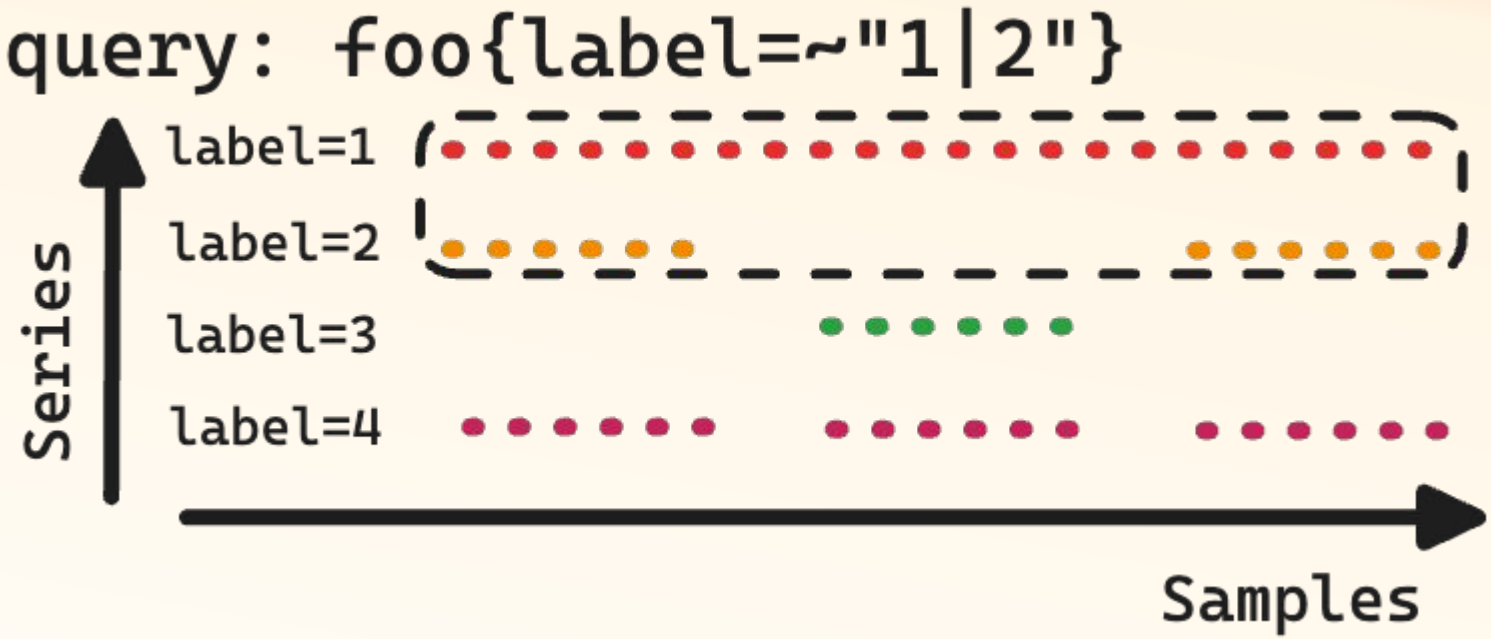
When PromQL/MetricsQL query can be slow?

1. When it selects big number of **time series**.
2. When it selects big number of **data samples**.

Select all series and samples of metric "foo"



Select subset of series and samples of metric "foo"



Select subset of series and samples for last 5m

query: foo{label=~"1|2"}&start=-5m

Selecting **less data** is the **most effective way** to optimize the query performance

How many series query selects?

1. Use the combination of count and last_over_time functions over series selector from the query
2. For instant query:
   a. **count(last_over_time(<series selector>[5m]))**
3. For range query:
   a. **count(last_over_time(<series selector>[<range>]))**

How many series query selects?

```
count(last_over_time(node_systemd_unit_state[3d]))
```

Execute

Table | Graph

Slow query

Load time: 3390ms Resolution: 14s Result series: 1

Evaluation time

{}	990

```
count(last_over_time(node_cooling_device_cur_state[3d]))
```

Execute

Table | Graph

Fast query

Load time: 61ms Resolution: 14s Result series: 1

Evaluation time

{}	1

How many samples query selects?

1. Use the combination of sum and count_over_time functions over series selector from the query
2. For instant query:
   a. **sum(count_over_time(<series selector>[5m]))**
3. For range query:
   a. **sum(count_over_time(<series selector>[<range>]))**

How many samples query selects?

```
sum(count_over_time(node_systemd_unit_state[3d]))
```

Execute

Table | Graph

Slow query

Load time: 2935ms Resolution: 14s Result series: 1

Evaluation time
{}	16690915

```
sum(count_over_time(node_cooling_device_cur_state[3d]))
```

Execute

Table | Graph

Fast query

Load time: 48ms Resolution: 14s Result series: 1

Evaluation time
{}	17280

No matter what your query is…

- the more series you select
- and the more data samples you process

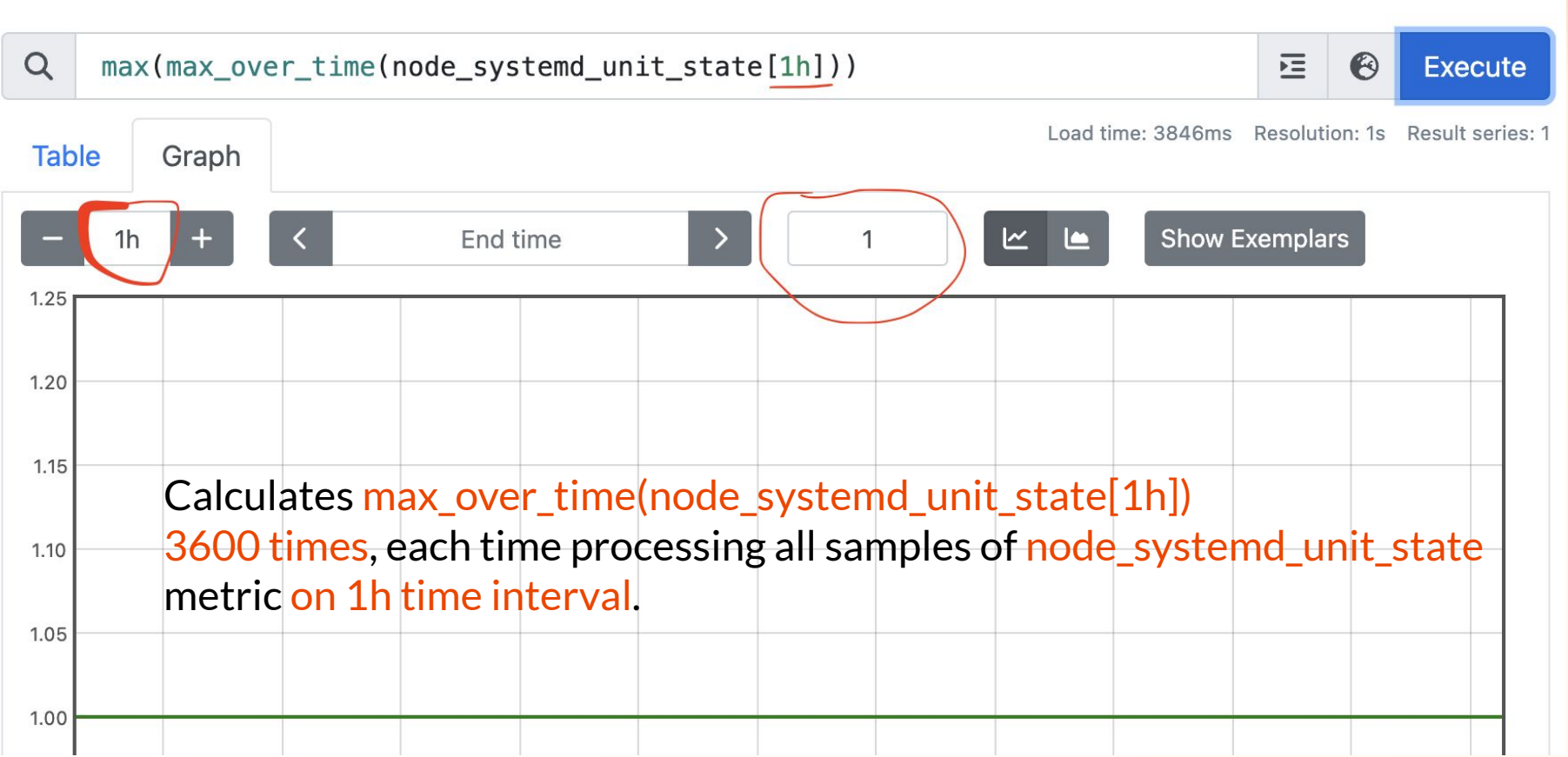
The slower this query will be!

Selected samples != Processed samples

Usually, the number of processed raw samples matches the number of selected raw samples. Unless:

- You use **range query** (i.e. for plotting graphs in Grafana)
- The lookbehind window [<duration>] exceeds the **step param**

Selected samples != Processed samples



What about functions? How slow are they?

1. Label manipulation functions such as label_replace, label_join and label_set.
2. Transform functions such as abs, round and time
3. Aggregate functions such as sum, count, avg, min, max.
4. Rollup functions such as rate, increase, min_over_time and and quantile_over_time.

** Ordered from the least expensive to the most expensive*

What about functions? How slow are they?

5. Subqueries such as:

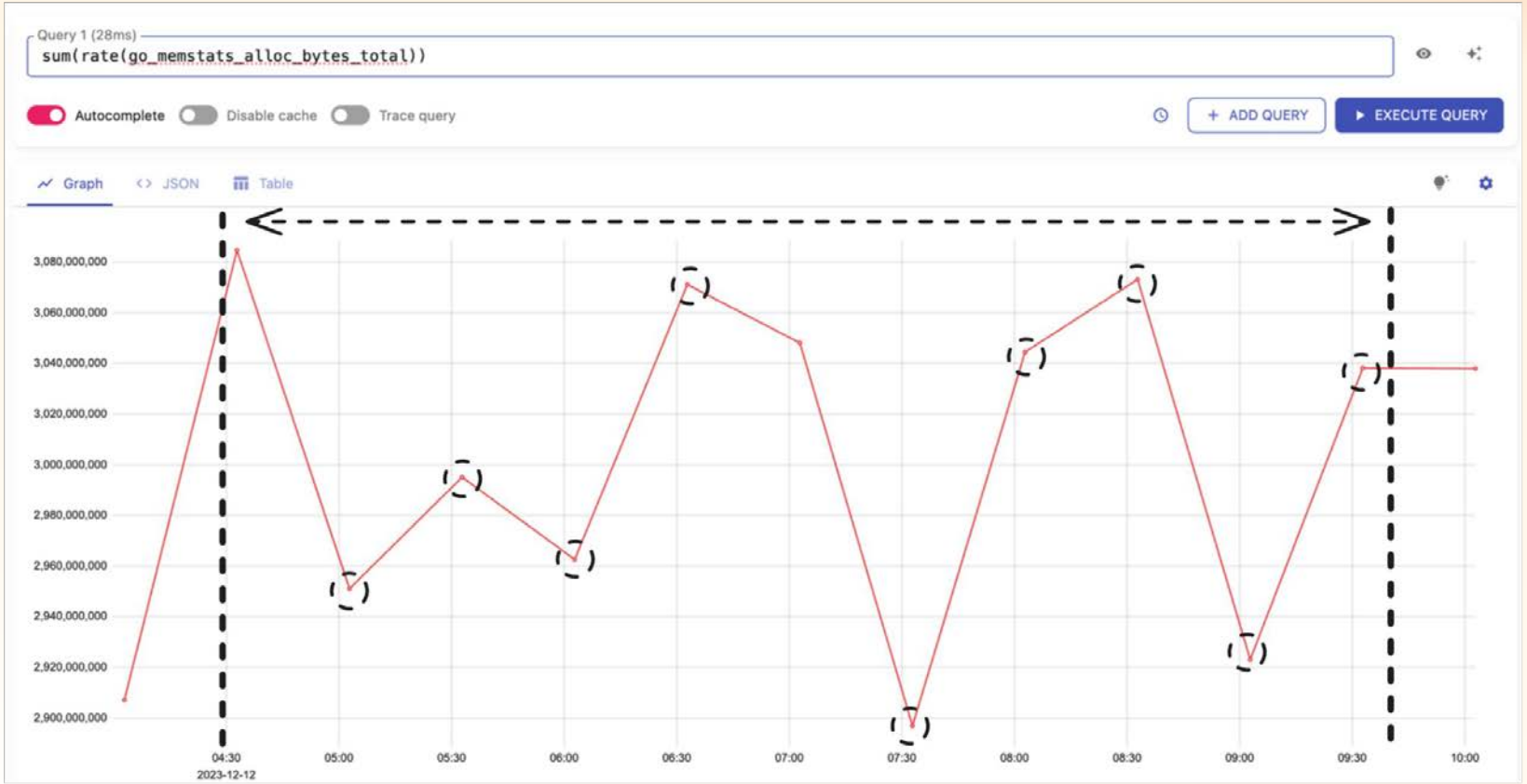
- `avg_over_time((rate(errors_total[5m]) > bool 10)[1h:1m])`
- `max_over_time(deriv(rate(traveled_meters_total[1m])[5m:])[1h:])`
- `min_over_time(rate(requests_total[5m])[30m:])`

subqueries:
> are **expensive**: the inner query is evaluated many times
> are **complicated**: hard to write, hard to read and understand

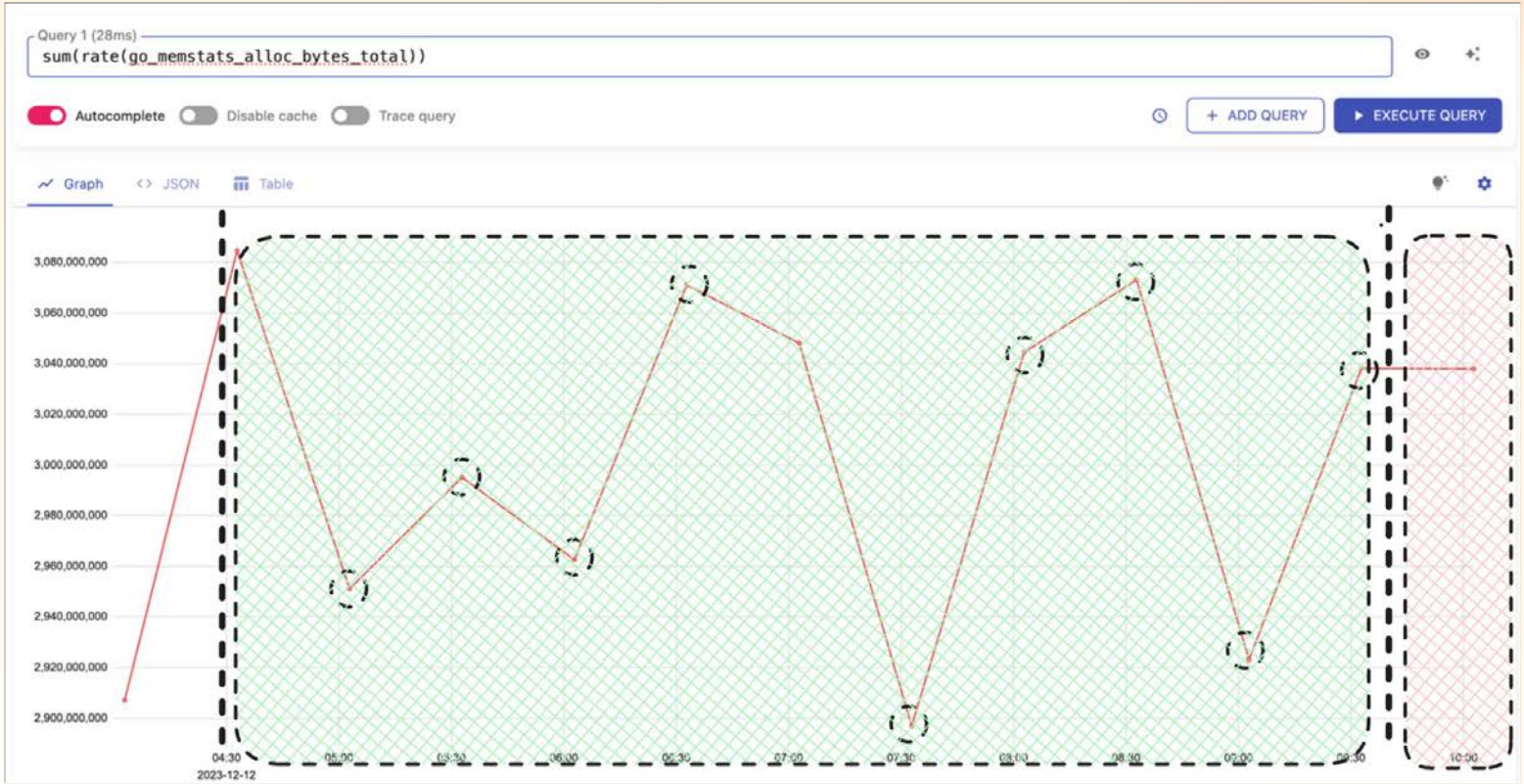
Performance improvement tips: query caching

- Prometheus doesn't support **caching** out-of-the-box
- Caching **reverse-proxies** to the rescue:
  - https://github.com/jacksontj/promxy
  - Thanos query frontend

Performance improvement tips: query caching



Performance improvement tips: query caching



Served from cache Queried from DB

Performance improvement tips: filters pushdown

```
min_over_time(node_systemd_unit_state{name="apparmor.service"}[3d])
/
max_over_time(node_systemd_unit_state[3d])
```

Execute

Table Graph

Load time: 3078ms Resolution: 1s Result series: 5

Apply filters to both parts of expression

```
min_over_time(node_systemd_unit_state{name="apparmor.service"}[3d])
/
max_over_time(node_systemd_unit_state{name="apparmor.service"}[3d])
```

Execute

Table Graph

Load time: 73ms Resolution: 1s Result series: 5

Performance improvement tips: recording rules

- Pre-compute time series with recording rules
- Pros:
  - Queries over pre-computed series **are faster**
- Cons:
  - Constant **read pressure** on database
  - **Extra time series** to process and store
  - Recording rules need to be **maintained**

Summary

- **Measure complexity** of the PromQL/MetricsQL queries
- **Use caching** frontend to reduce pressure on database
- Carefully **craft queries** to get optimal performance
- Use **recording rules** for performance-critical queries

Can VictoriaMetrics make it easier?

Can VictoriaMetrics make it easier?

Yes!

- Cardinality explorer
- Query tracing
- Built-in caching
- Filters pushdown
- Stream aggregation

VictoriaMetrics: cardinality explorer

VMUI | Query | Explore | Tools | Dashboards | 2023-09-06

Time series selector

Focus label

Limit entries: 10

Total series
72,491 ↑0.31%

Documentation | RESET | EXECUTE QUERY

Metric names with the highest number of series

Table | Graph

Metric name	Number of series	Share in total
github_downloads_total	3496	4.82% ↓-0.01%
storage_operation_duration_seconds_bucket	2240	3.09% ↓-0.01%
flag	2046	2.82% ↓-0.01%
grpc_server_handled_total	1564	2.16% ↓-0.01%

VictoriaMetrics: cardinality explorer

VictoriaMetrics allows exploring time series cardinality to identify:

- **Metric names** with the highest **number of series**
- **Labels** with the highest **number of series**
- **Values** with the highest **number of series for the selected label**
- **label=name** pairs with the highest **number of series**
- **Labels** with the highest **number of unique values**

➔ *Available built-in in VictoriaMetrics components*

➔ *Supports specifying Prometheus URL*

VictoriaMetrics: query tracing

VictoriaMetrics supports query tracing for detecting bottlenecks during query processing.

This is like **EXPLAIN ANALYZE** from Postgresql!

Query (4955ms)

```
sum(vm_http_requests_total)
```

Autocomplete | Disable cache | Trace query | + ADD QUERY | ▶ EXECUTE QUERY

Trace for **sum(vm_http_requests_total)**

100.00%

4s955ms: vmselect-20240425-151811-tags-v1.101.0-enterprise-cluster-0-gea1c63525: /select/0/prometheus/api/v1/query_range: start=1714225499956, end=1716817499956, step=1800000, query="sum(vm_http_requests_total)": series=1

99.99%

4s954ms: eval: query=sum(vm_http_requests_total), timeRange=[2024-04-27T13:44:59.956Z..2024-05-27T13:44:59.956Z], step=1800000, mayCache=false: series=1, points=1441, pointsPerSeries=1441

99.99%

4s954ms: aggregate sum(): series=1

0.00%

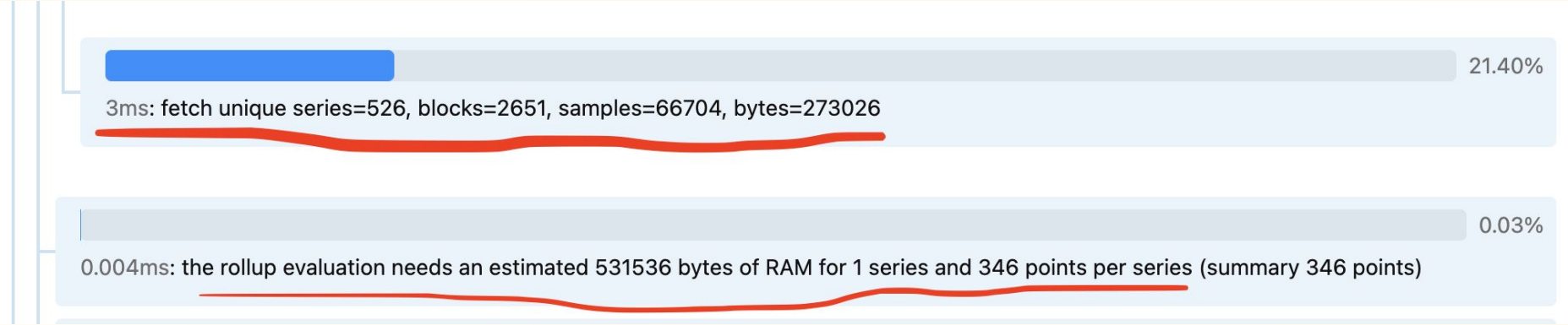
0.007ms: do not fetch series from cache, since it is disabled in the current context

99.99%

4s954ms: rollup sum(vm_http_requests_total): timeRange=[2024-04-27T13:44:59.956Z..2024-05-27T13:44:59.956Z], step=1800000, window=0

VictoriaMetrics: count number of series and samples

Trace shows exactly how many series and samples was selected and processed by query



VictoriaMetrics: built-in caching

VictoriaMetrics automatically caches Range and Instant queries. No proxies needed!

98.60%

15ms: eval: query=sum(rate(vm_http_requests_total[5m])), timeRange=[2024-05-27T12:34:10Z..2024-05-28T12:31:40Z], step=250000, mayCache=true: series=1, points=346, pointsPerSeries=346

98.58%

15ms: aggregate sum(): series=1

0.41%

0.066ms: rollup cache get series: query=sum(rate(vm_http_requests_total[5m])), timeRange=[2024-05-27T12:34:10Z..2024-05-28T12:31:40Z], window=300000, step=250000

0.09%

0.015ms: load compressed entry from cache with size 3109 bytes

0.27%

0.043ms: unpack the entry into 5532 bytes

VictoriaMetrics: filters pushdown

VictoriaMetrics automatically performs filters pushdown

Query (872ms)

```
rate(vm_http_requests_total{cluster="sandbox"}[3d])
/
rate(vm_http_request_errors_total[3d])
```

Autocomplete | Disable cache | Trace query | + ADD QUERY | ▶ EXECUTE QUERY

Trace for **rate(vm_http_requests_total{cluster="sandbox"}[3d]) / rate(vm_http_request_errors_total[3d])**

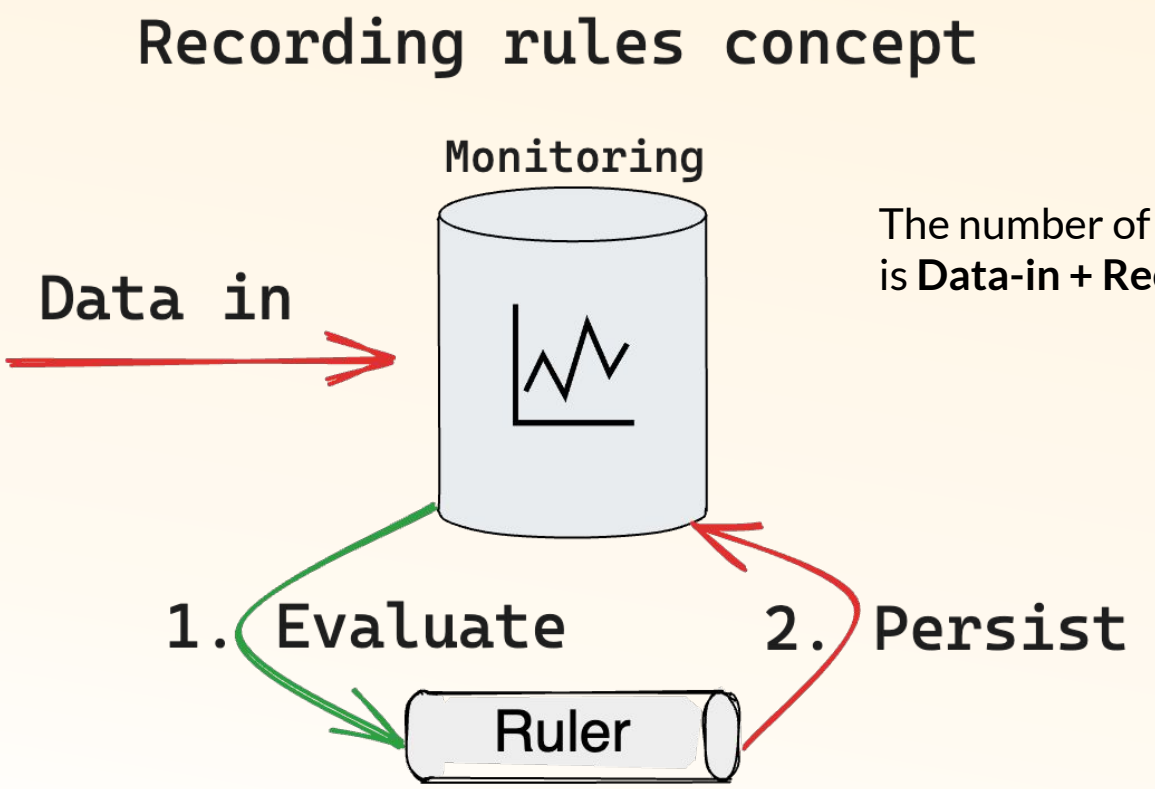
100.00%

872ms: vmselect-20240425-151811-tags-v1.101.0-enterprise-cluster-0-gea1c63525: /select/0/prometheus/api/v1/query: query=rate(vm_http_requests_total{cluster="sandbox"}[3d]) / rate(vm_http_request_errors_total[3d]) , time=1716900181833: series=32

99.95%

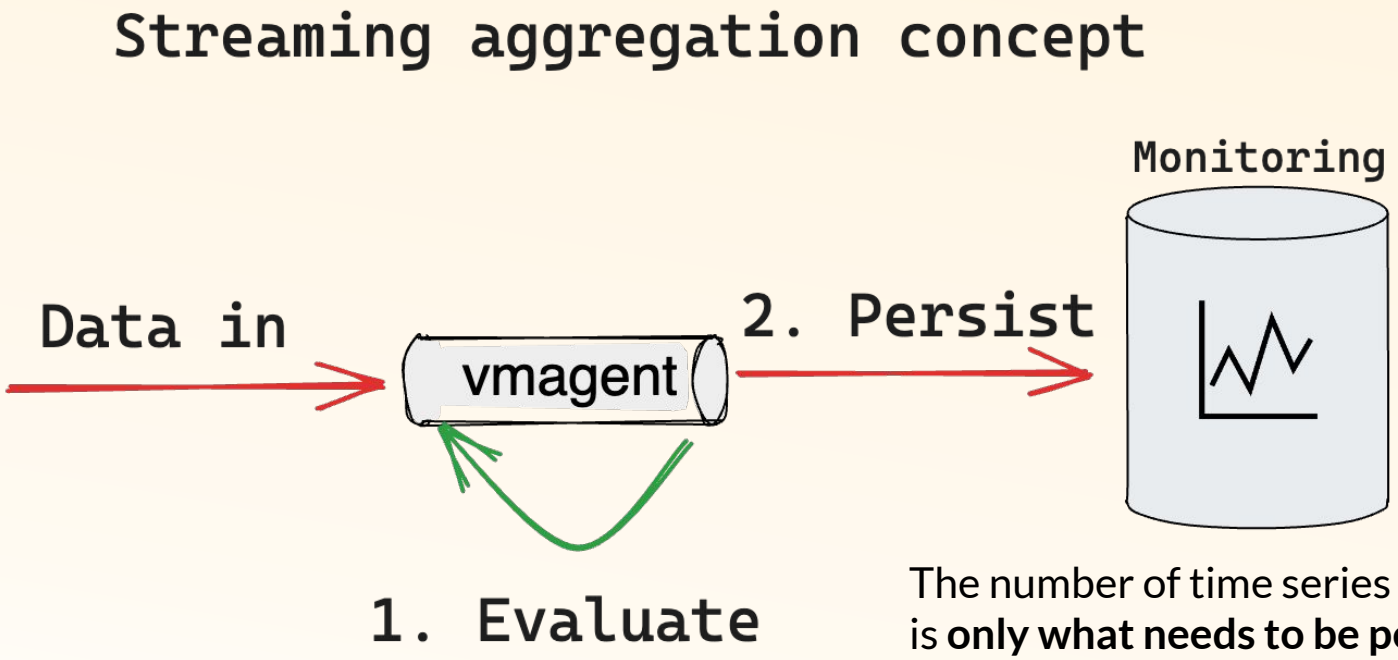
872ms: eval: query=rate(vm_http_requests_total{cluster="sandbox"}[3d]) / rate(vm_http_request_errors_total{cluster="sandbox"}[3d]), timeRange=[2024-05-28T12:43:01.833Z..2024-05-28T12:43:01.833Z], step=250000, mayCache=true: series=251, points=251, pointsPerSeries=1

VictoriaMetrics: Stream aggregation vs Recording rules



The number of time series stored in TSDB is **Data-in + Recording Rules results**

VictoriaMetrics: Stream aggregation vs Recording rules



The number of time series stored in TSDB is **only what needs to be persisted**

Additional materials

1. How to optimize PromQL and MetricsQL queries
2. Prometheus Subqueries in VictoriaMetrics
3. Query tracing
4. Streaming aggregation
5. VictoriaMetrics playground
6. Documentation

Questions?

- https://github.com/VictoriaMetrics
- https://github.com/hagen1778