@ThanosMetrics

San Diego, 20th November 2019

Thanos

Inside a Distributed Monitoring System

Frederic Branczyk

brancz *fredbrancz*

Bartłomiej Płotka

bwplotka

Speakers

@ThanosMetrics

Frederic Branczyk

Principal Software Engineer @ Red Hat; OpenShift Monitoring Team

Prometheus Maintainer; Thanos Maintainer; SIG Instrumentation Lead

Bartek Plotka

Principal Software Engineer @ Red Hat; OpenShift Monitoring Team

Prometheus Maintainer; Thanos Maintainer

Agenda

- Quick intro, reiterate quickly on components
- StoreAPI
  - Querier (discovery, fanout, filtering)
  - Producer vs Browser
  - Integrations: OpenTSDB
- Downsampling
- Horizontal Query scaling
- Summary

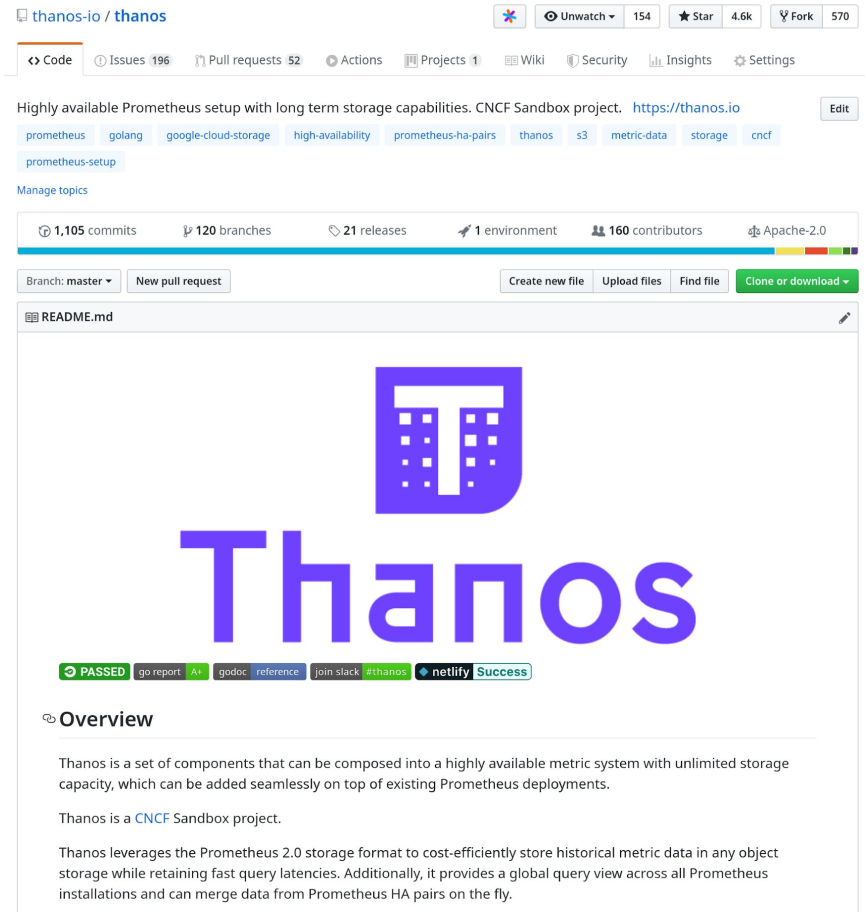
@ThanosMetrics

Thanos Community

- Fully open source from start
- Started in Nov 2017
- Part of **CNCF Sandbox**

- 4600+ Github stars
- 160+ contributors
- ~500 slack users
- 8 maintainers, 3 triagers from 7 different companies.

- Transparent Governance
- Prometheus Ecosystem


thanos-io / thanos

Unwatch 154 | Star 4.6k | Fork 570

Code | Issues 196 | Pull requests 52 | Actions | Projects 1 | Wiki | Security | Insights | Settings

Highly available Prometheus setup with long term storage capabilities. CNCF Sandbox project. https://thanos.io

prometheus golang google-cloud-storage high-availability prometheus-ha-pairs thanos s3 metric-data storage cncf prometheus-setup

Manage topics

1,105 commits | 120 branches | 21 releases | 1 environment | 160 contributors | Apache-2.0

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

README.md

PASSED | go report A+ | godoc reference | join slack #thanos | netlify Success

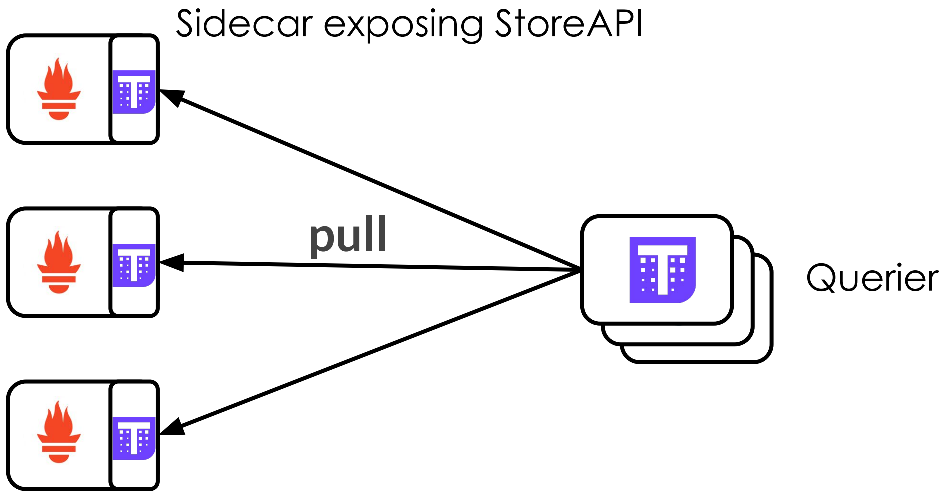
Overview

Thanos is a set of components that can be composed into a highly available metric system with unlimited storage capacity, which can be added seamlessly on top of existing Prometheus deployments.

Thanos is a CNCF Sandbox project.

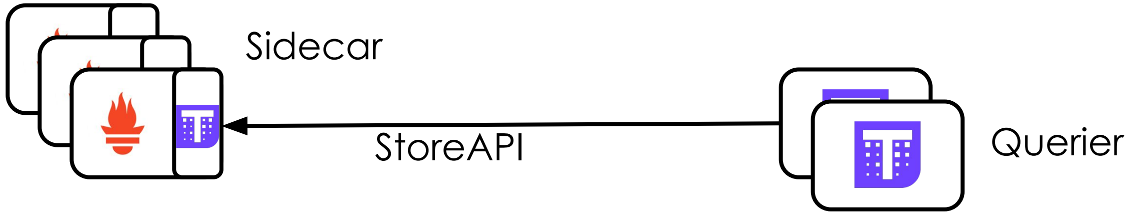
Thanos leverages the Prometheus 2.0 storage format to cost-efficiently store historical metric data in any object storage while retaining fast query latencies. Additionally, it provides a global query view across all Prometheus installations and can merge data from Prometheus HA pairs on the fly.

Let's quickly reiterate on Thanos

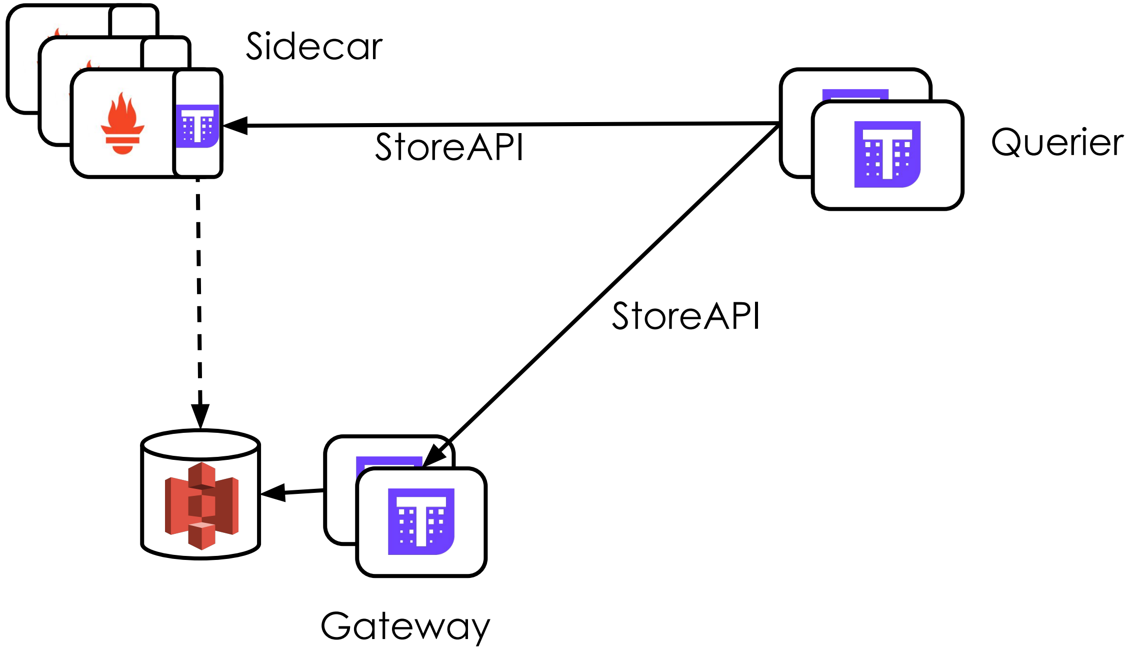


Let’s quickly reiterate on Thanos

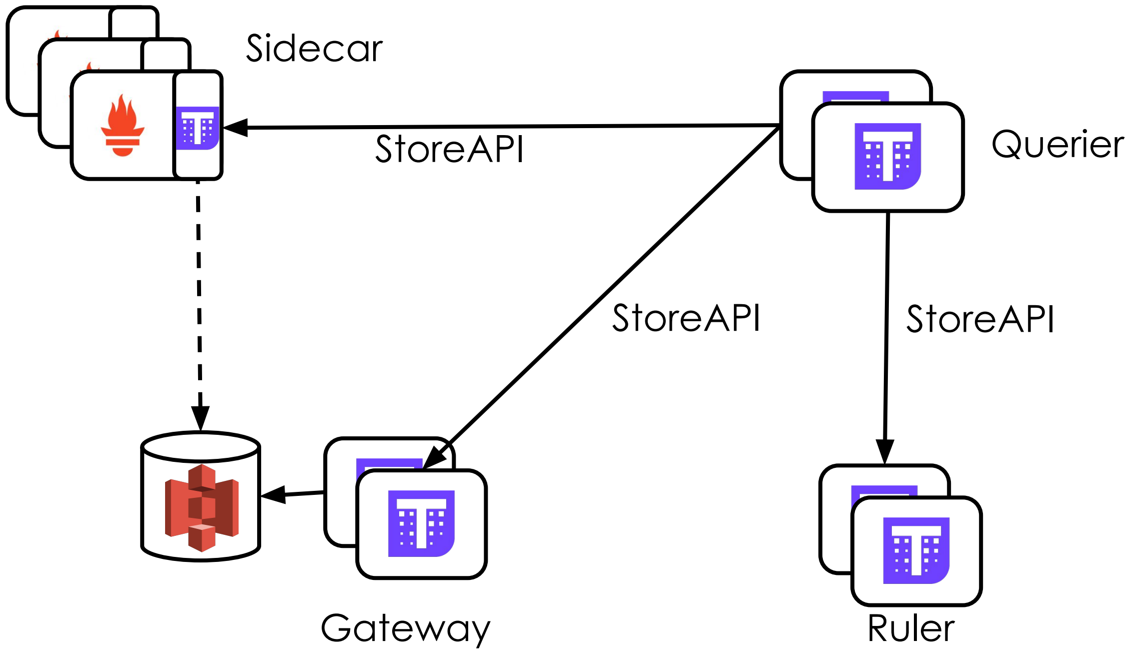
@ThanosMetrics



Let's quickly reiterate on Thanos

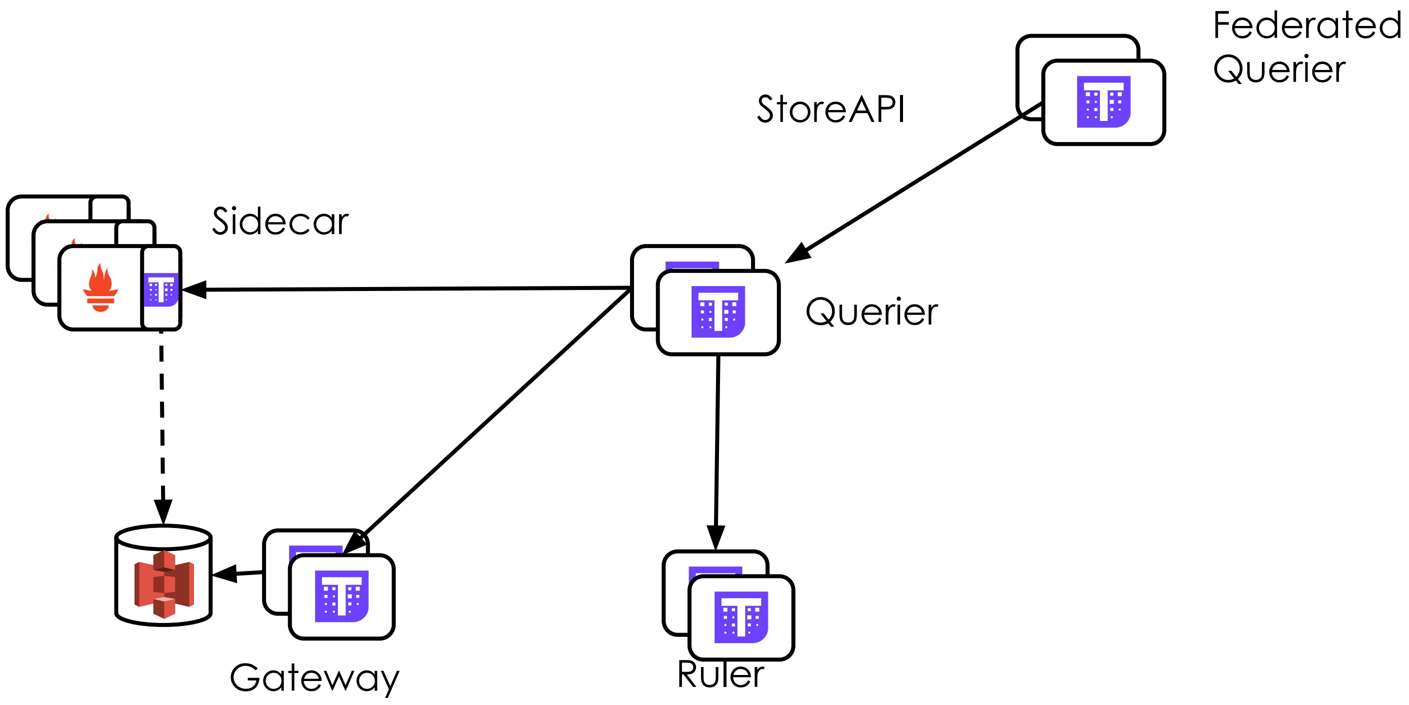


Let’s quickly reiterate on Thanos



Let's quickly reiterate on Thanos

@ThanosMetrics



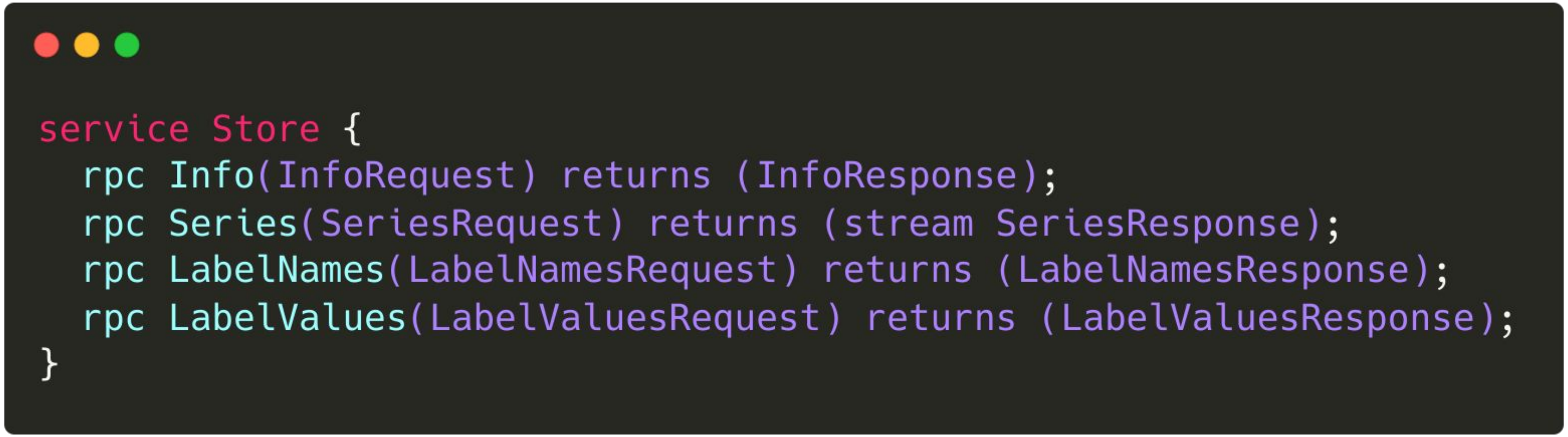
There was something common in all these architectures

StoreAPI

StoreAPI

- Every component in Thanos serves data via gRPC StoreAPI
  - sidecar
  - store
  - rule
  - receive (experimental component)
  - query
- Integrations! https://thanos.io/integrations.md/
  - OpenTSDB as StoreAPI: https://github.com/G-Research/geras

StoreAPI

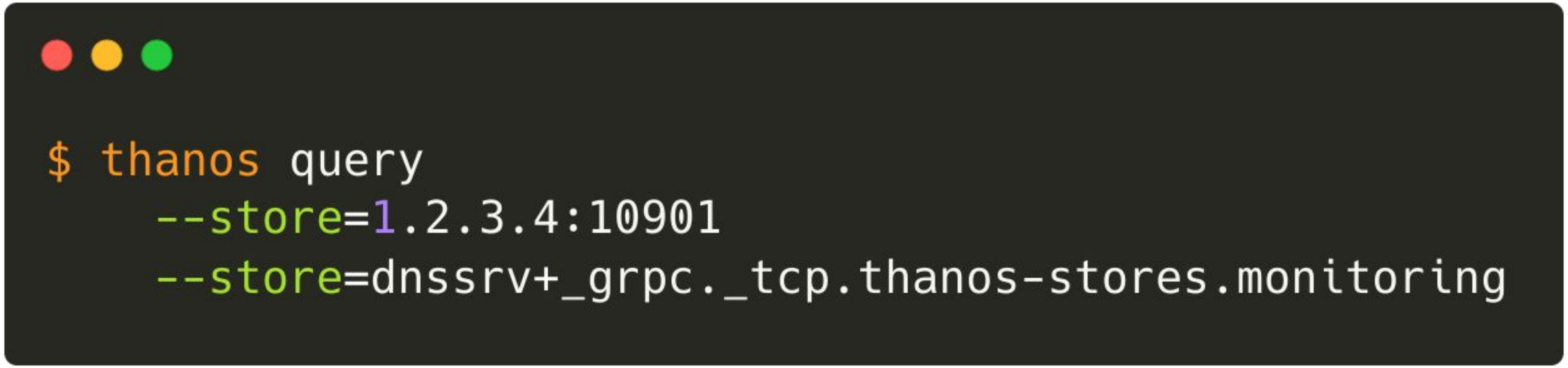


```
service Store {
  rpc Info(InfoRequest) returns (InfoResponse);
  rpc Series(SeriesRequest) returns (stream SeriesResponse);
  rpc LabelNames(LabelNamesRequest) returns (LabelNamesResponse);
  rpc LabelValues(LabelValuesRequest) returns (LabelValuesResponse);
}
```

From: rpc.proto

Thanos Query: Store Discovery

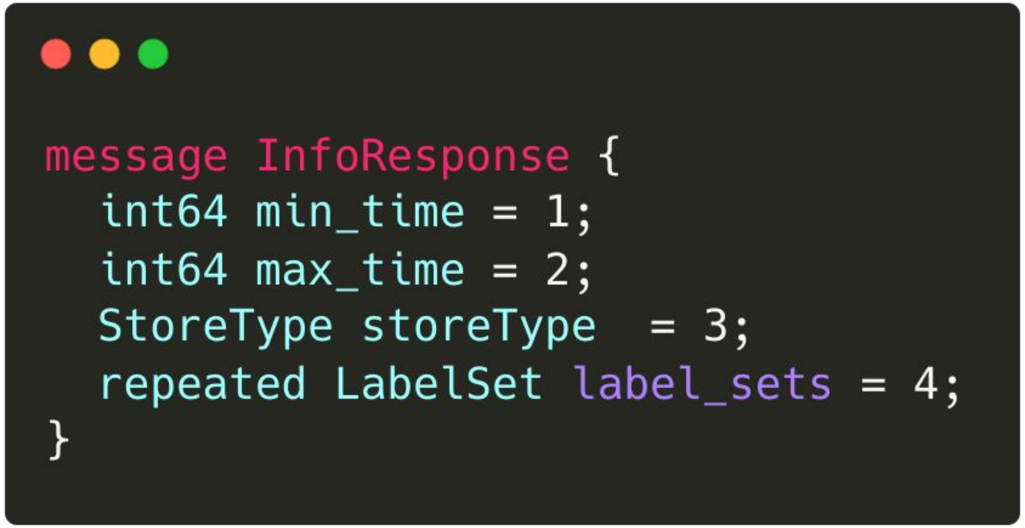
- --store flag
    - Exact endpoints
    - DNS discovery: A, AAAA, SRV



```
$ thanos query
    --store=1.2.3.4:10901
    --store=dnssrv+_grpc._tcp.thanos-stores.monitoring
```

Thanos Query: Store Infos

- Every 10s requests Info endpoint
- Healthiness
- Metadata propagation

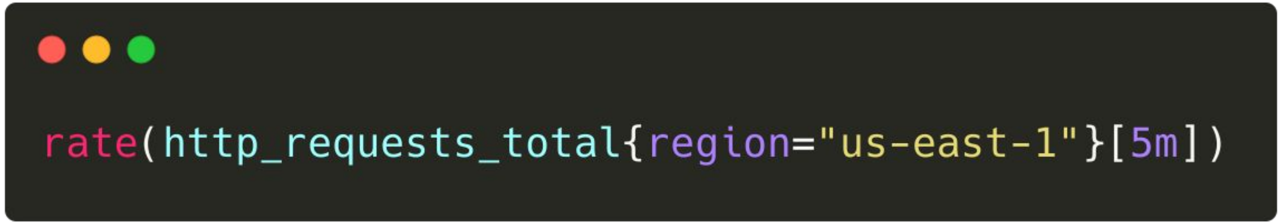


```
message InfoResponse {
  int64 min_time = 1;
  int64 max_time = 2;
  StoreType storeType  = 3;
  repeated LabelSet label_sets = 4;
}
```

@ThanosMetrics

Thanos Query: Life of a query

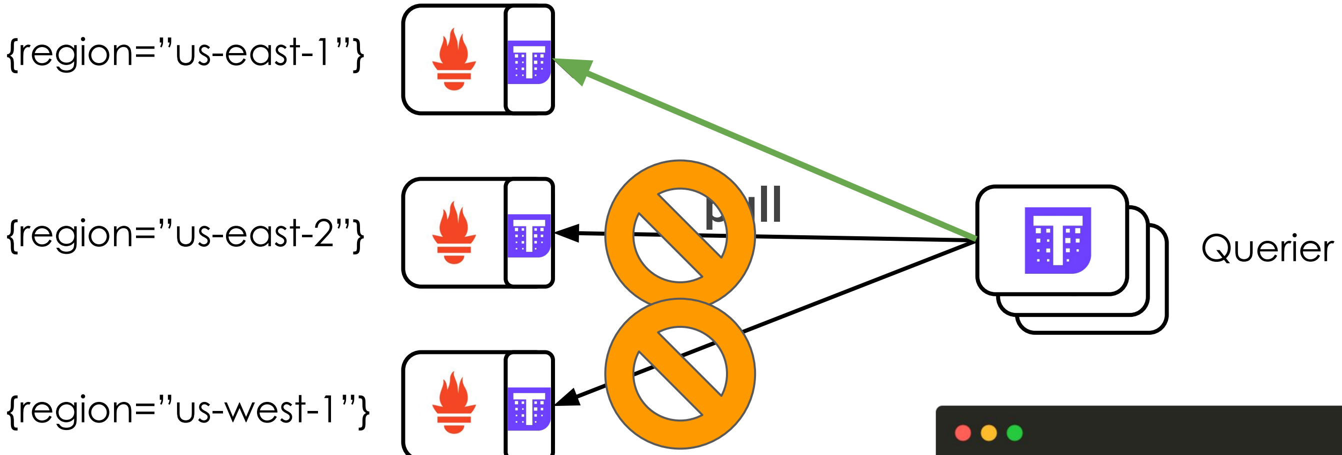
- Query
  - Select possible stores
  - Fan out to gather data
  - Process query

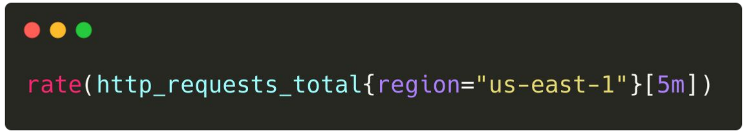


```
rate(http_requests_total{region="us-east-1"}[5m])
```

Thanos Query: Life of a query

@ThanosMetrics



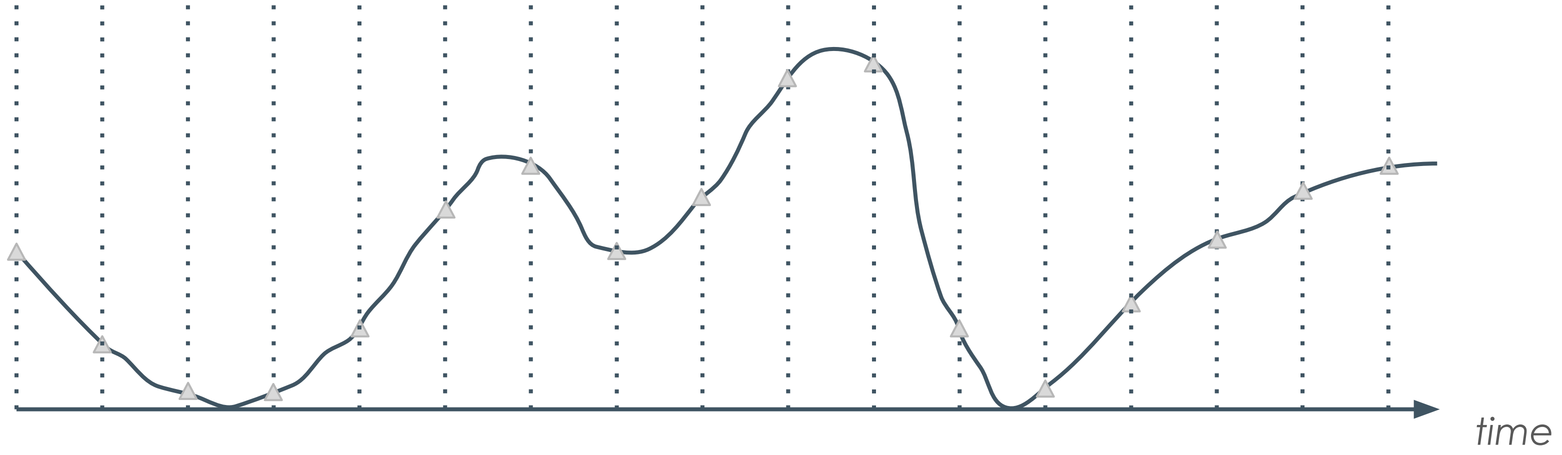


```
rate(http_requests_total{region="us-east-1"}[5m])
```

ProxyStore

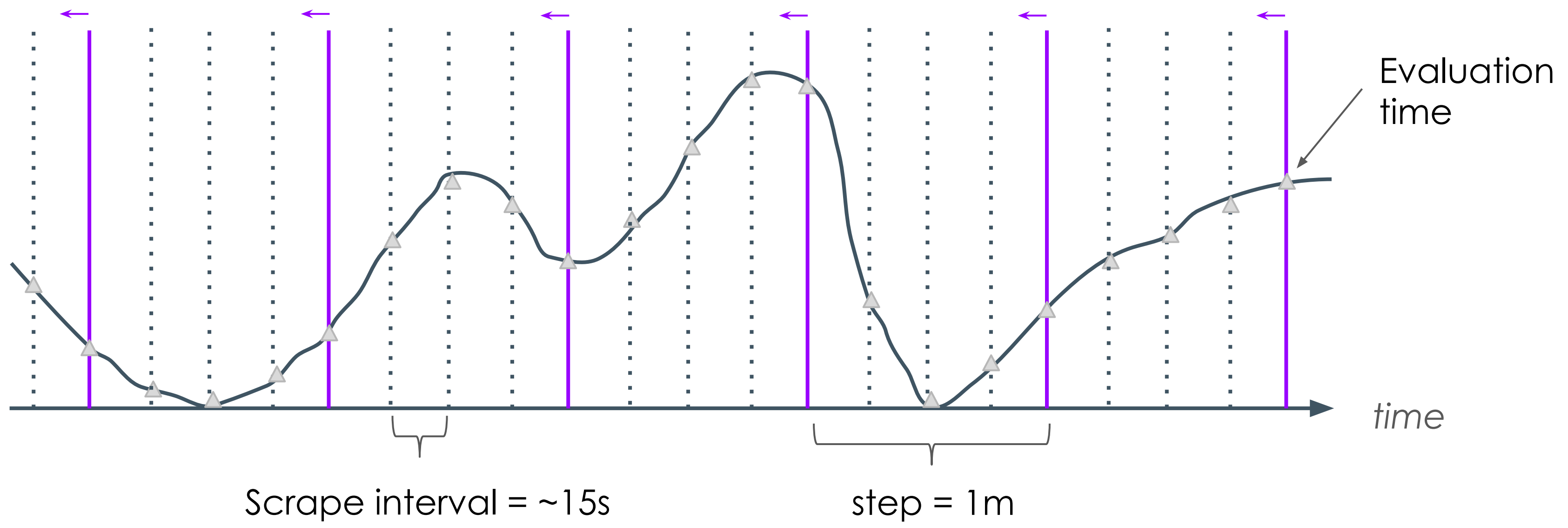
Challenges of Querying Years of Data

Query Resolution



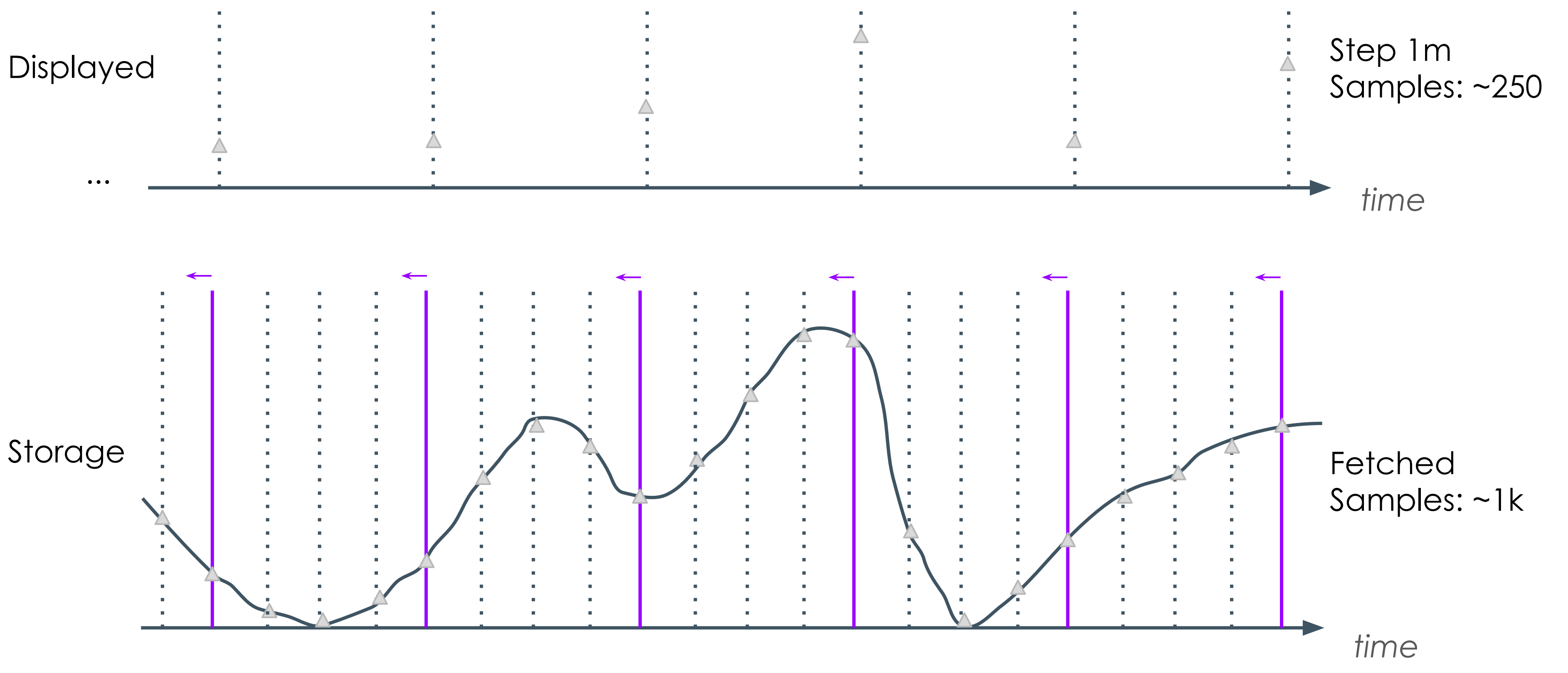
- Typical scrape period of Prometheus is 15s
- Querying 30 days means ~170k samples

Query Resolution



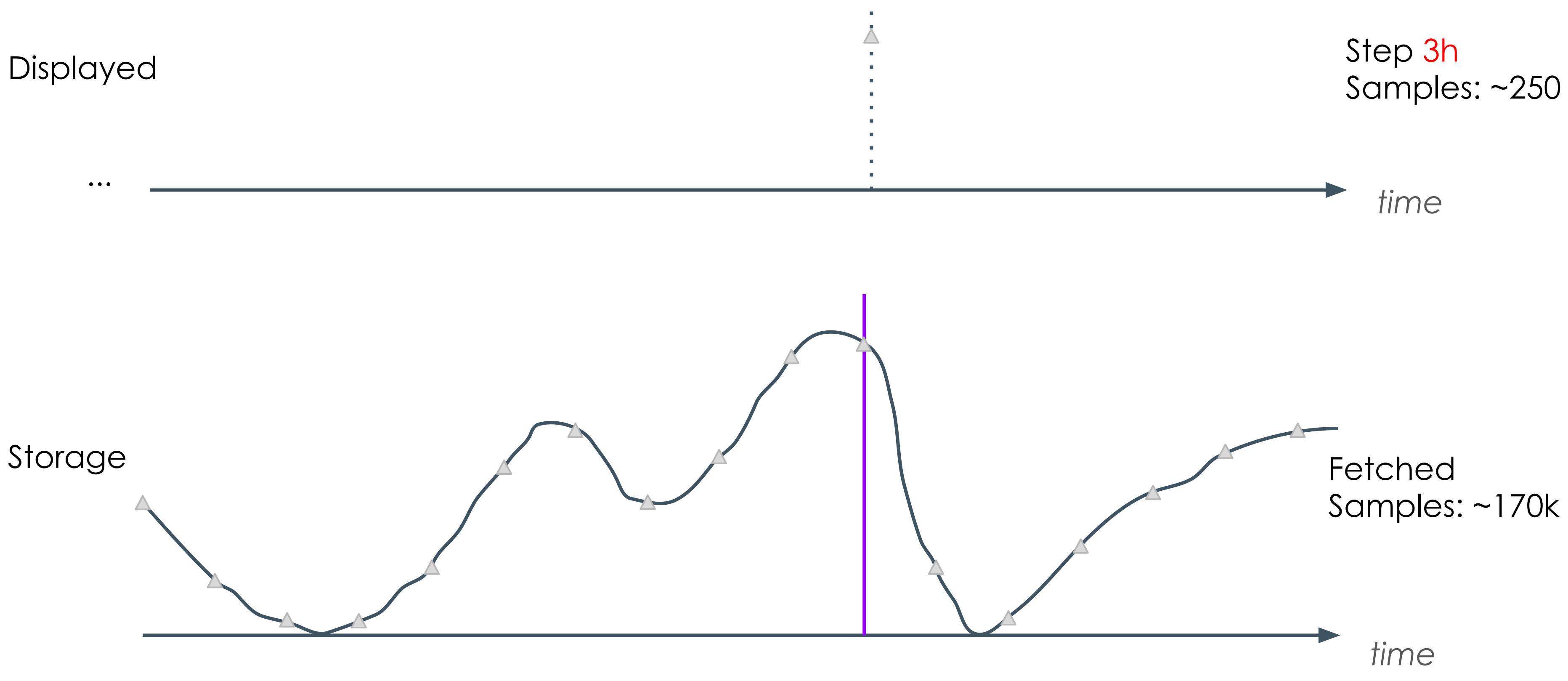
Query Resolution: 5h range

@ThanosMetrics



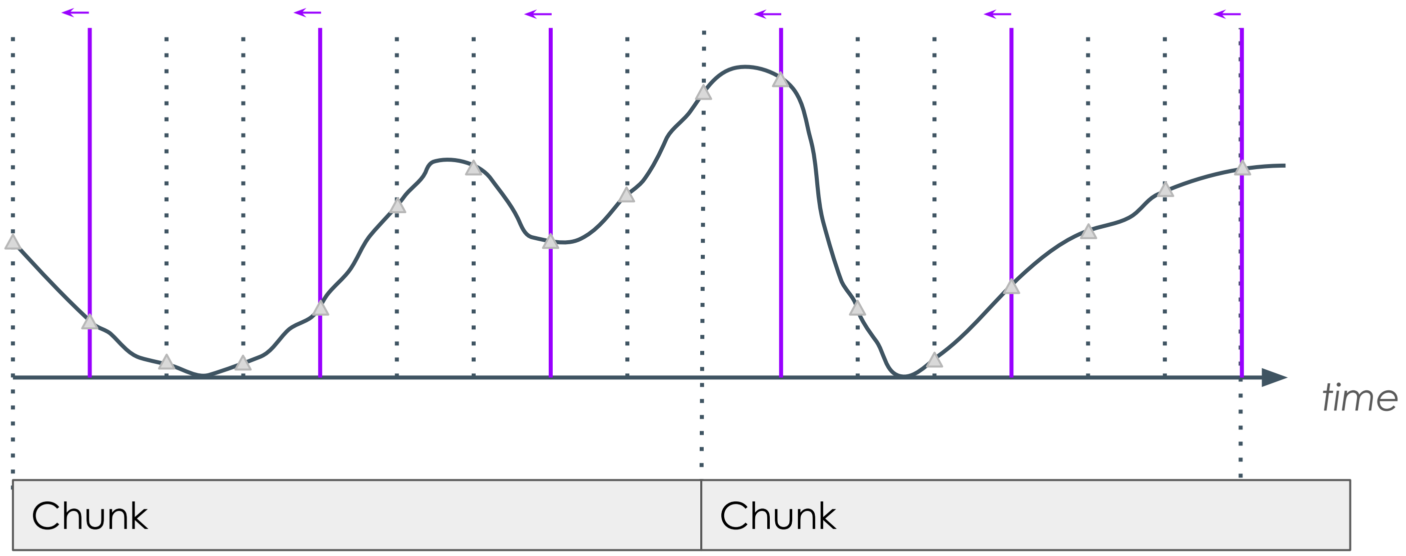
Query Resolution: 30d range

@ThanosMetrics



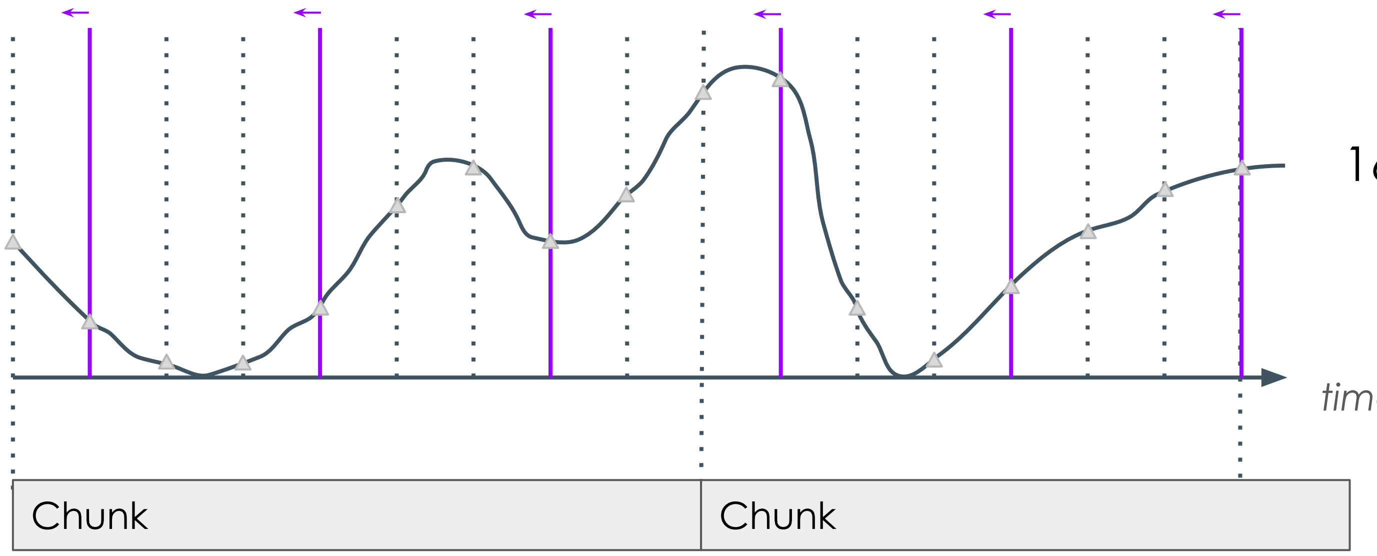
Chunks

@ThanosMetrics



Samples are stored in chunks

Chunks



16 bytes/sample

1.3 bytes/sample

Samples are stored in chunks

Chunk tradeoff

Decompressing one sample takes 10-40 nanoseconds

Chunk tradeoff

Decompressing one sample takes 10-40 nanoseconds

Query Range	Samples for 1000 series	Decompression latency	Chunk data size
30m	~120 000	~5ms	~160KB
1d	~6 millions	~240ms	~8MB

Chunks tradeoff

Decompressing one sample takes 10-40 nanoseconds

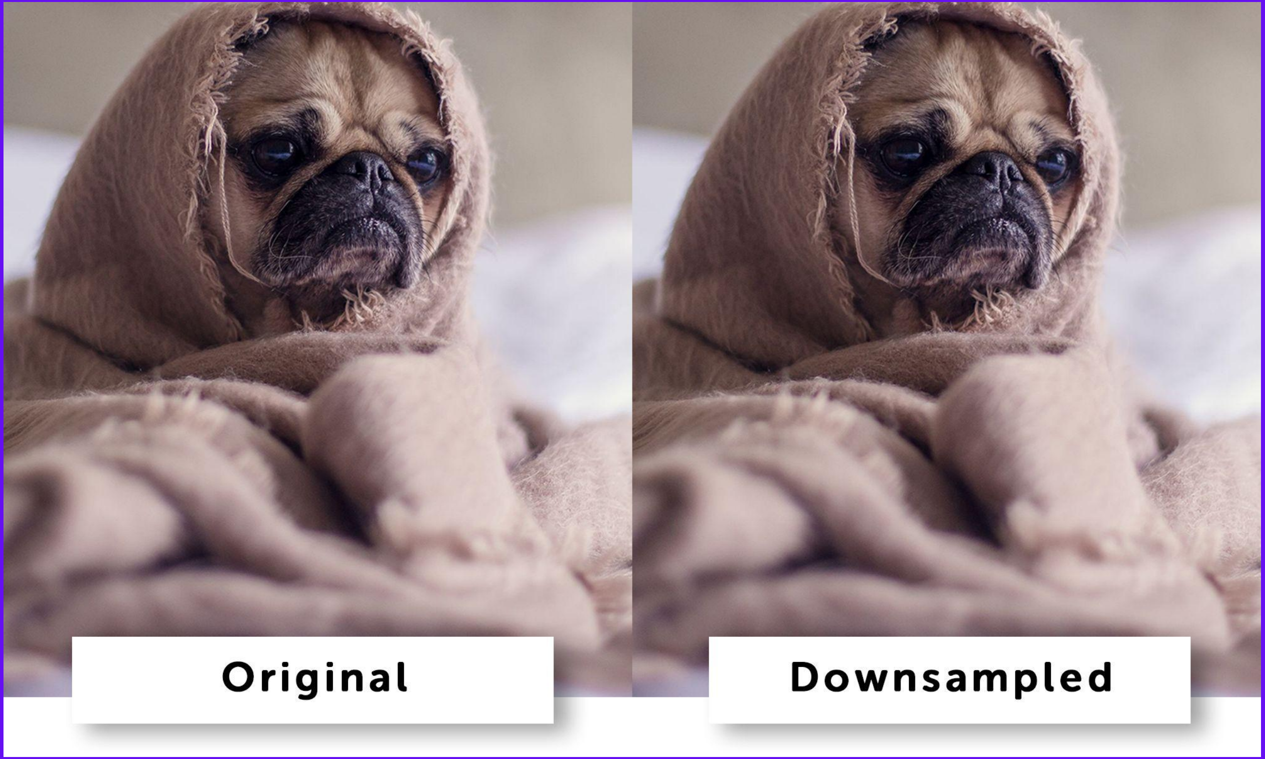
Query Range	Samples for 1000 series	Decompression latency	Chunk data size
30m	~120 000	~5ms	~160KB
1d	~6 millions	~240ms	~8MB
30d	~170 millions	~7s	~240MB

Chunks tradeoff

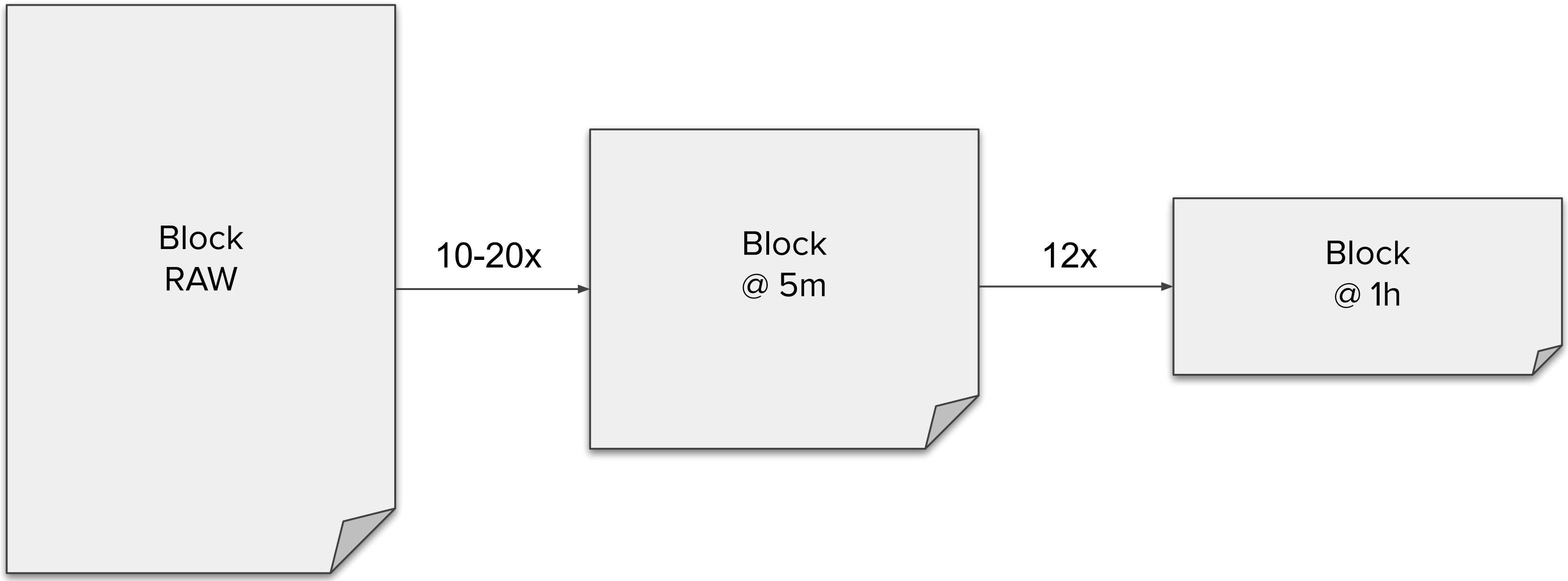
Decompressing one sample takes 10-40 nanoseconds

Query Range	Samples for 1000 series	Decompression latency	Chunk data size
30m	~120 000	~5ms	~160KB
1d	~6 millions	~240ms	~8MB
30d	~170 millions	~7s	~240MB
1y	~2 billions	~1m20s	~2GB 😱

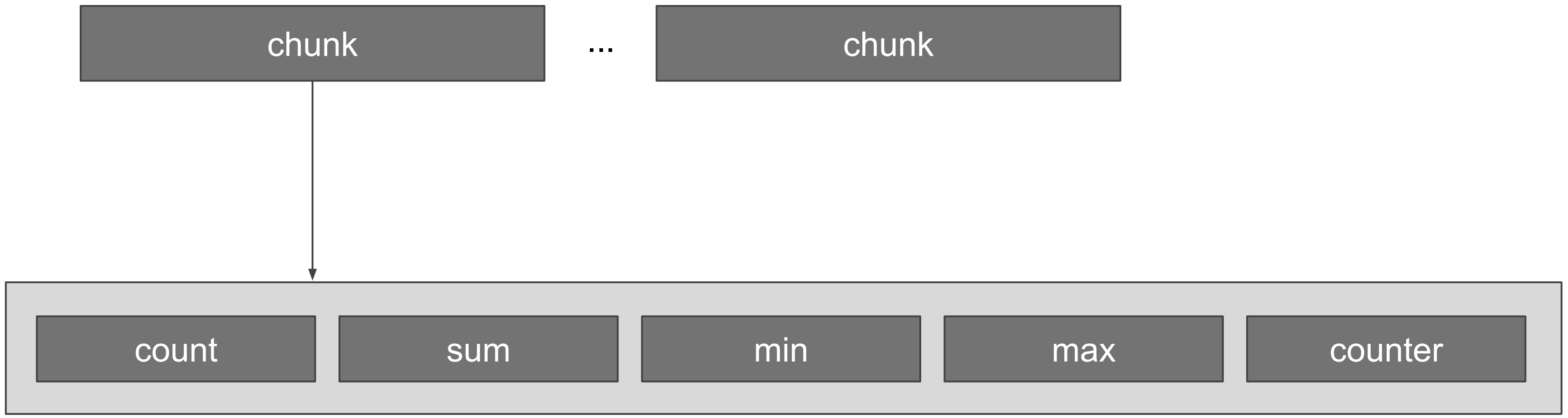
Downsampling



Downsampling

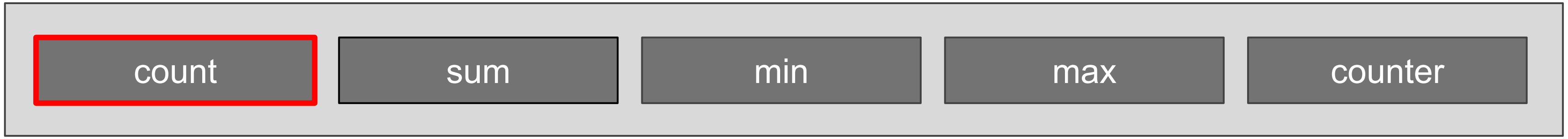


Downsampling



Downsampling

@ThanosMetrics

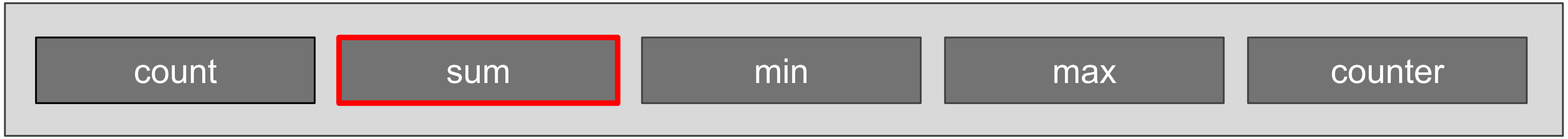


count(requests_total)

count_over_time(requests_total[1h])

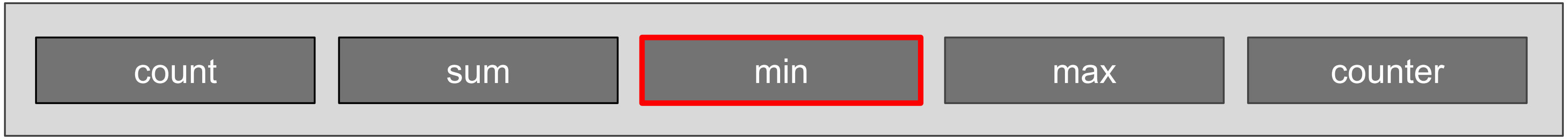
Downsampling

@ThanosMetrics



sum_over_time(requests_total[1h])

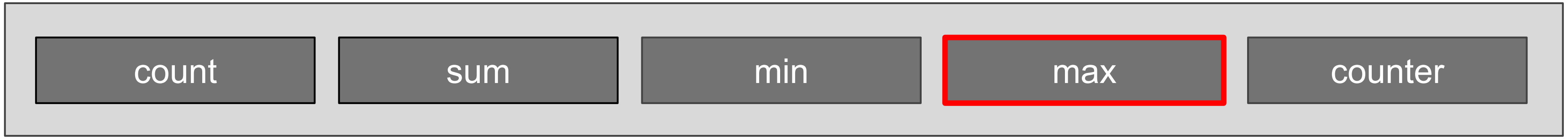
Downsampling



min(requests_total)

min_over_time(requests_total[1h])

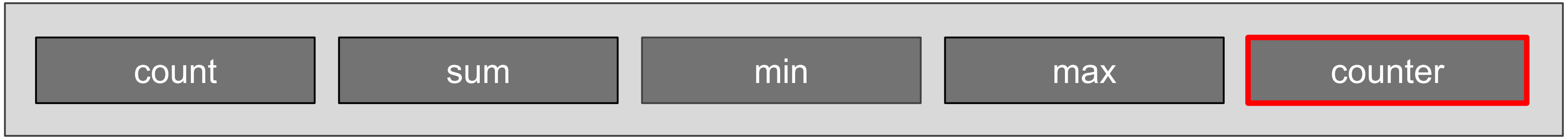
Downsampling



max(requests_total)

max_over_time(requests_total[1h])

Downsampling

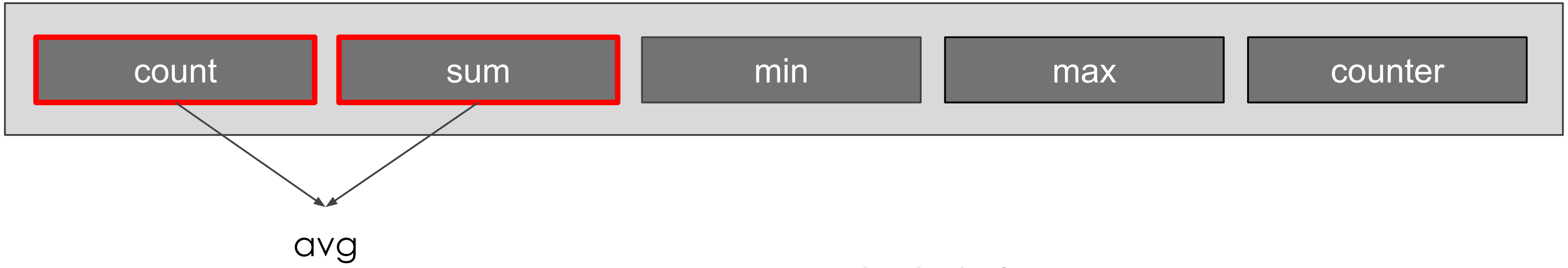


rate(requests_total[1h])

increase(requests_total[1h])

@ThanosMetrics

Downsampling



requests_total

avg(requests_total)

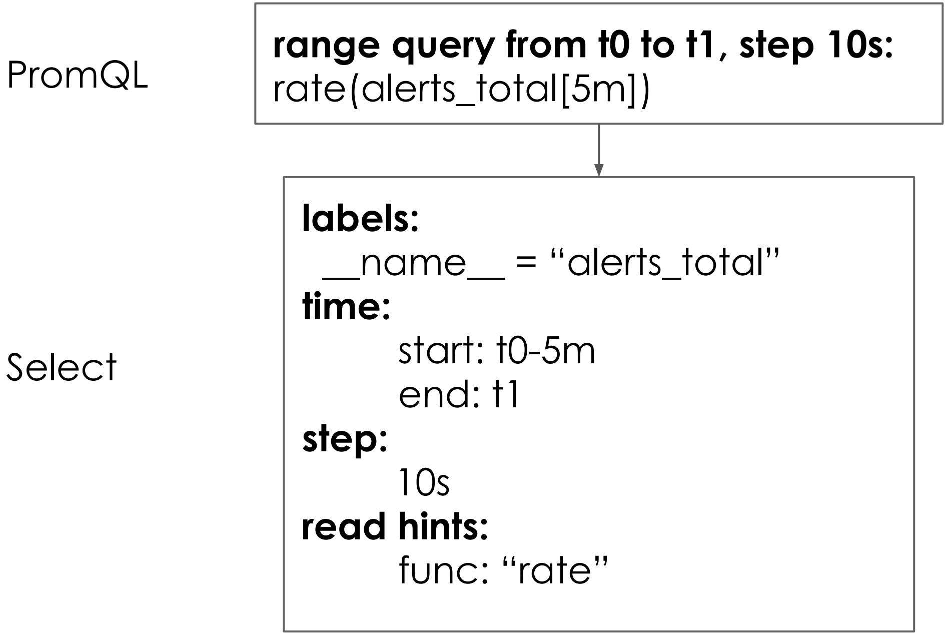
sum(requests_total)

Downsampling: What chunk to use on query?

PromQL

range query from t0 to t1, step 10s:
rate(alerts_total[5m])

Downsampling: What chunk to use on query?



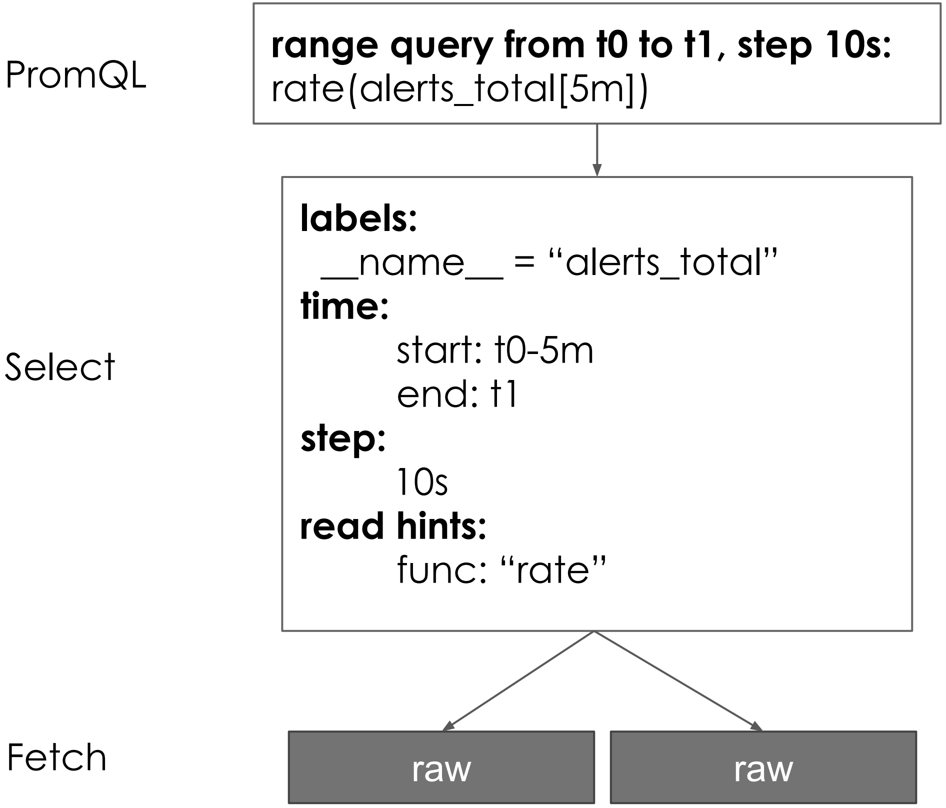
PromQL

range query from t0 to t1, step 10s:
rate(alerts_total[5m])

Select

labels:
  __name__ = “alerts_total”
time:
  start: t0-5m
  end: t1
step:
  10s
read hints:
  func: “rate”

Downsampling: What chunk to use on query?



Downsampling: What chunk to use on query?

PromQL

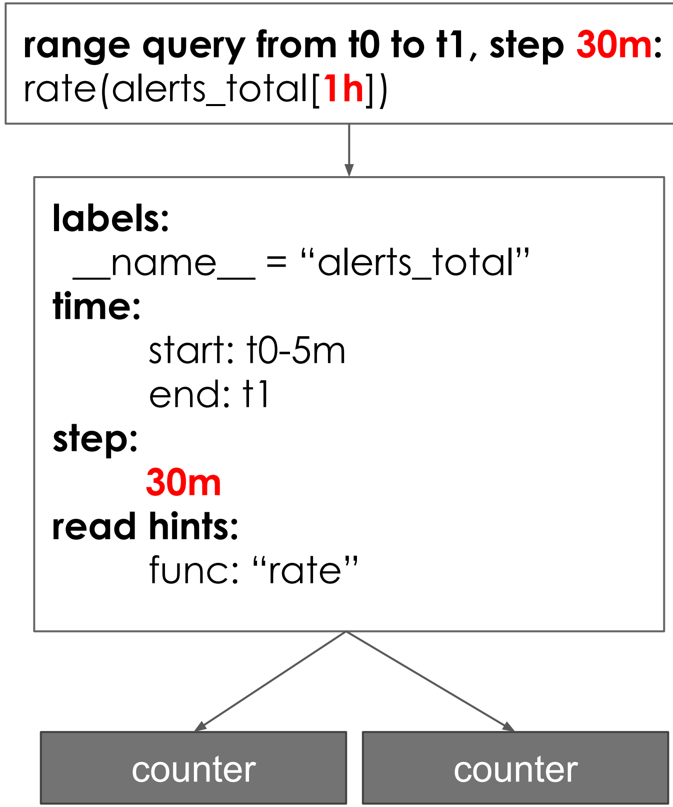
range query from t0 to t1, step 30m:
rate(alerts_total[**1h**])

Select

labels:
  __name__ = “alerts_total”
time:
    start: t0-5m
    end: t1
step:
    30m
read hints:
    func: “rate”

Can we fit 5 samples for this step with lower resolution?

Downsampling: What chunk to use on query?



PromQL

range query from t0 to t1, step 30m:
rate(alerts_total[**1h**])

Select

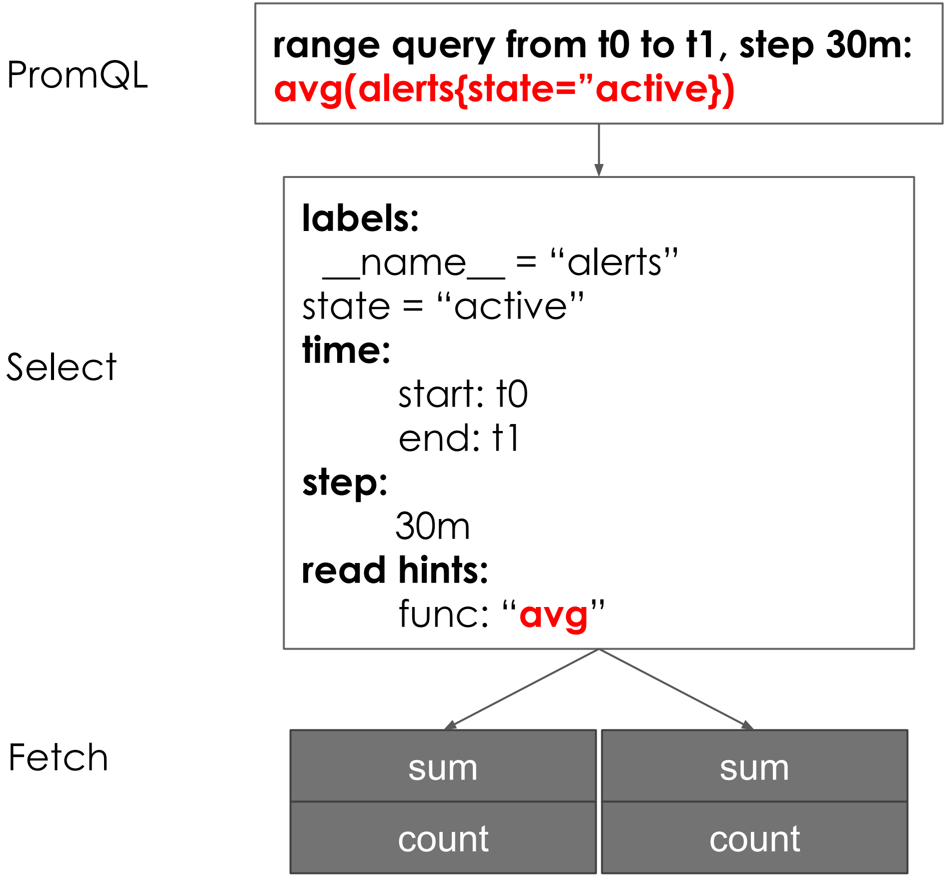
labels:
  __name__ = “alerts_total”
time:
  start: t0-5m
  end: t1
step:
  30m
read hints:
  func: “rate”

Can we fit 5 samples for this step with lower resolution?

Fetch

yes for 5m resolution!

Downsampling: What chunk to use on query?



PromQL

range query from t0 to t1, step 30m:
avg(alerts{state="active})

Select

labels:
 __name__ = "alerts"
state = "active"
time:
 start: t0
 end: t1
step:
 30m
read hints:
 func: "**avg**"

Fetch

sum	sum
count	count

Downsampling

Query Range	Samples for 1000 series	Decompression latency	Fetched chunks size	
30m	~120 000	~5ms	~160KB	
1d	~6 millions	~240ms	~8MB	
30d	~170 millions	~7s	~240MB	
30d	**~8 millions**	**~300ms**	**~9MB**	5m resolution [~5d+ queries]
1y	~2 billions	~80s	~2GB	
1y	**~8 millions**	**~300ms**	**~9MB**	1h resolution [~50d+ queries]

Downsampling: Caveats

- Thanos/Prometheus UI: **Step** (evaluation interval in seconds)
- Grafana: **Resolutions** (1/x samples per pixel)
- rate[<5m] vs rate[1h] / rate[5h] / rate[$_interval]
- Storing **only** downsampled data and trying to zoom-in

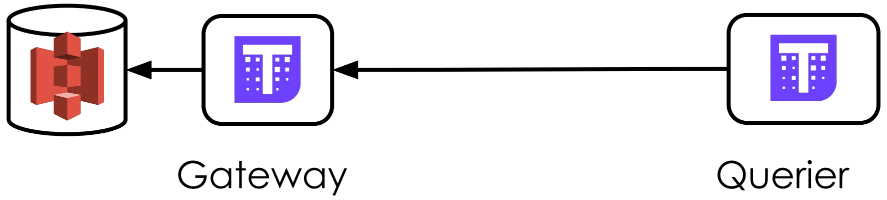
Downsampling: Caveats

- Thanos/Prometheus UI: **Step** (evaluation interval in seconds)
- Grafana: **Resolutions** (1/x samples per pixel)
- rate[<5m] vs rate[1h] / rate[5h] / rate[$_interval]
- Storing **only** downsampled data and trying to zoom-in

Standardize downsampling?

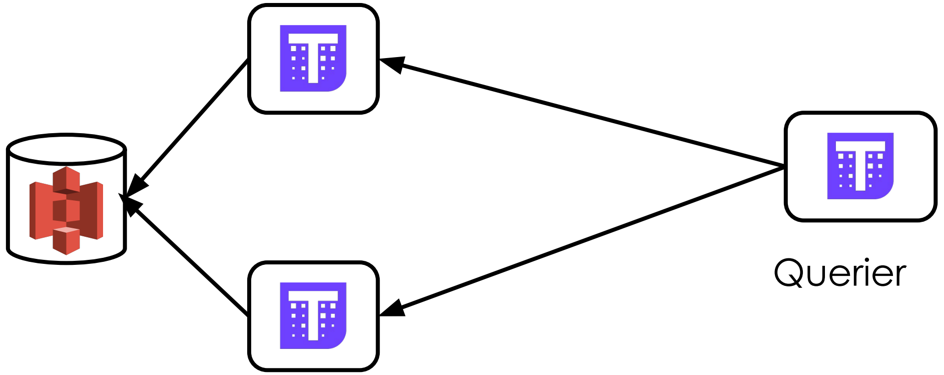
Horizontal Scaling of Long Term Storage Read Path

Querying long term storage backend



Time partitioning

Gateway: `--min-time=1y --max-time=150d`

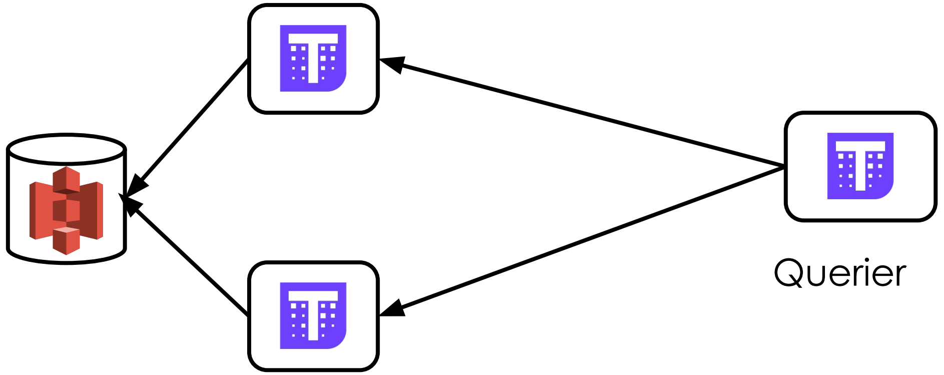


Gateway: `--min-time=150d`

Block Sharding

Gateway: `--selector.relabel-config=`

```
- action: keep
  regex: "eu.*"
  source_labels:
  - region
```



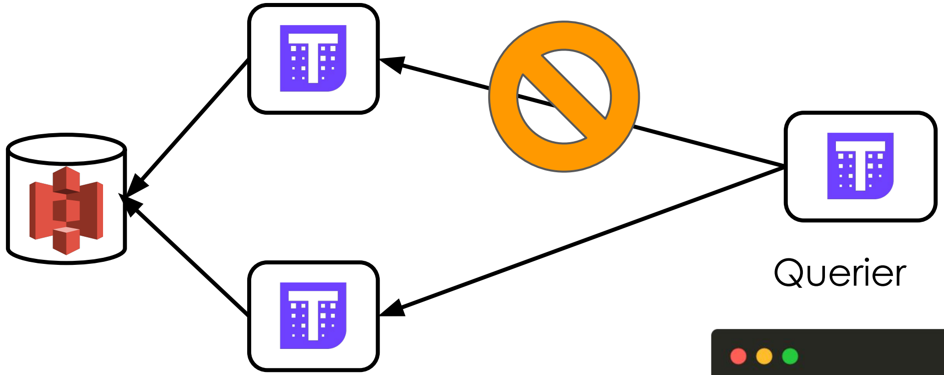
Gateway: `--selector.relabel-config=`

```
- action: keep
  regex: "us.*"
  source_labels:
  - region
```

Block Sharding

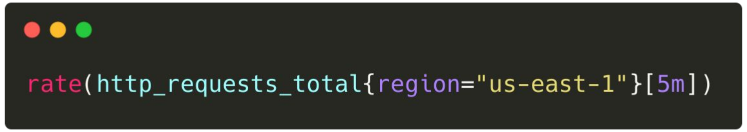
Gateway: `--selector.relabel-config=`

```
- action: keep
  regex: "eu.*"
  source_labels:
  - region
```



Gateway: `--selector.relabel-config=`

```
- action: keep
  regex: "us.*"
  source_labels:
  - region
```



```
rate(http_requests_total{region="us-east-1"}[5m])
```

Summary

Common StoreAPI

Downsampling

Horizontal Scaling of Long Term Storage

@ThanosMetrics

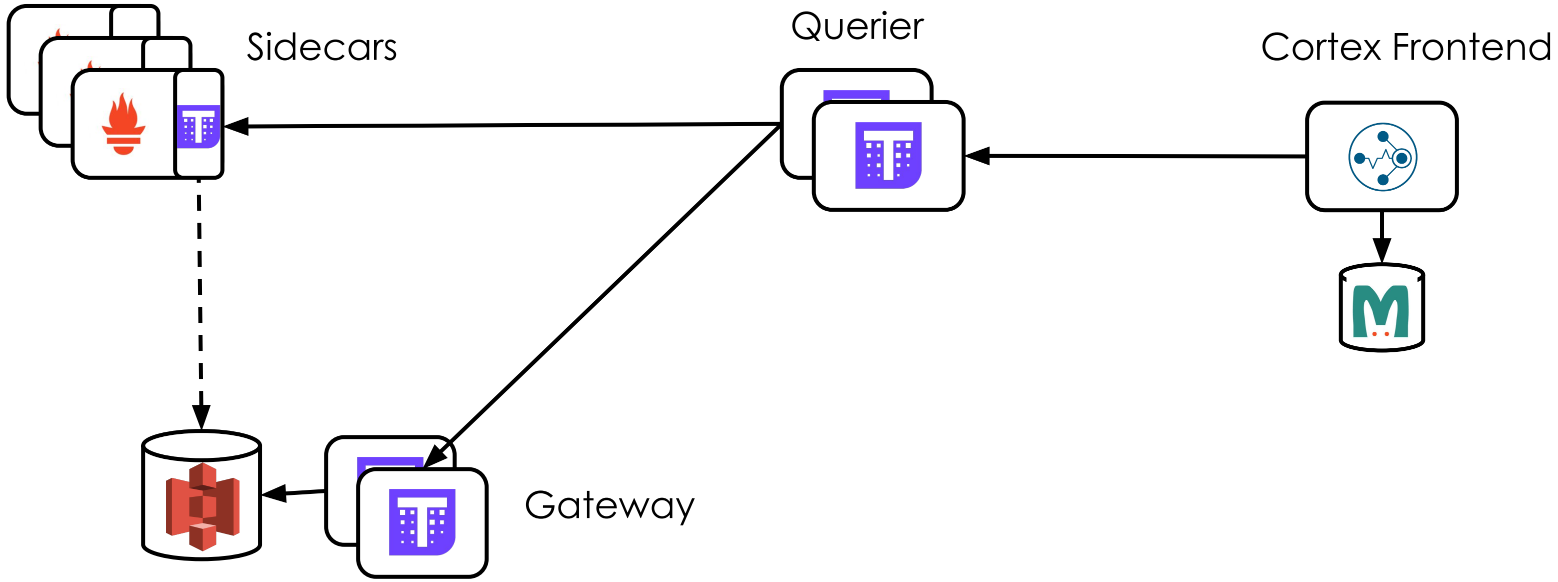
Thank You!

https://thanos.io

Bonus: Caching

@ThanosMetrics

Caching



Response Caching: Challenges

- Extremely useful for rolling windows (e.g Grafana “last 1h”)
- Dynamically changing StoreAPIs
- Downsampling
- Partial Response
- Backfilling/Deletion