

Native Histograms in Prometheus: A Better Histogram Experience for Everyone

*Julius Volz ([@juliusvolz](https://twitter.com/juliusvolz)) – PromLabs
KCD Zurich, June 15, 2023*

All Credit Goes To...



[Björn Rabenstein](#)



[Ganesh Vernekar](#)



[Dieter Plaetinck](#)

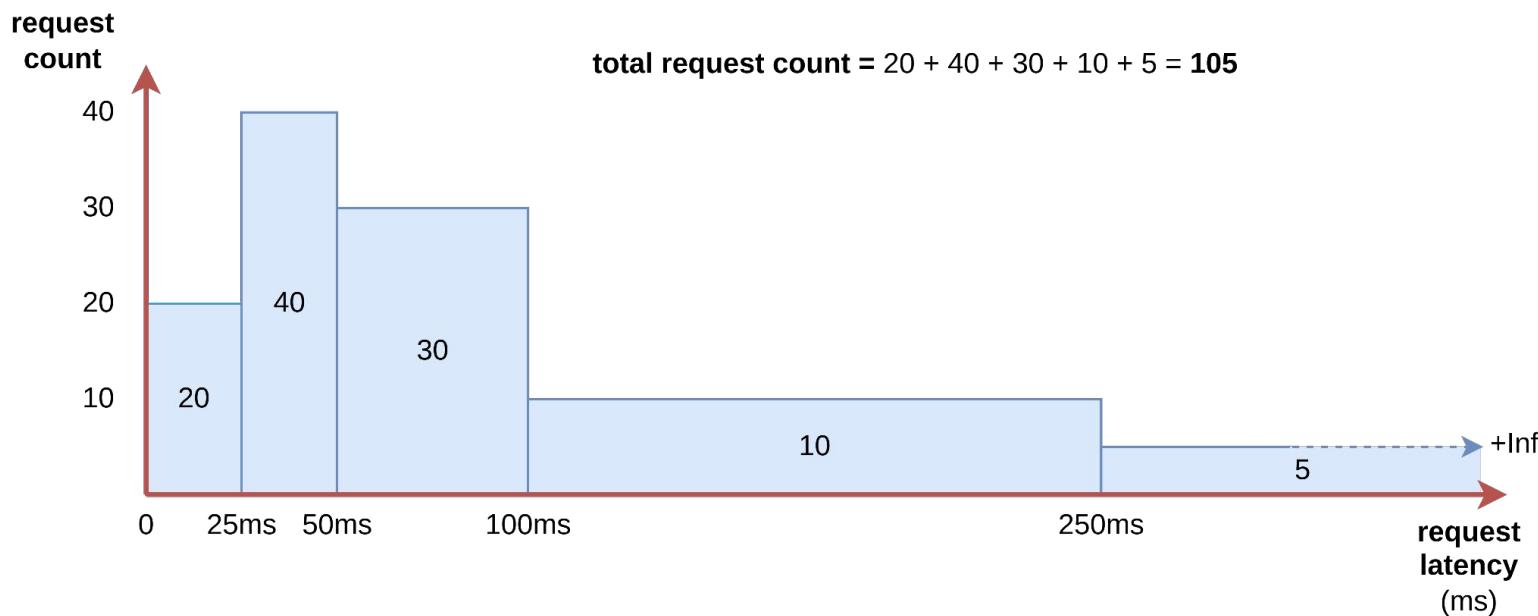
...and others!

Quick Recap:

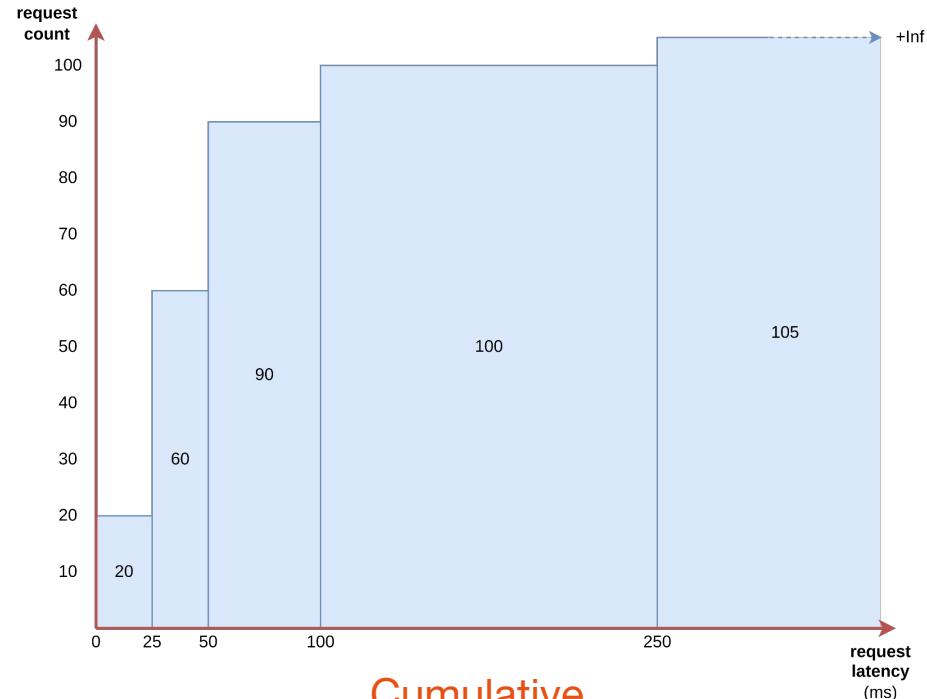
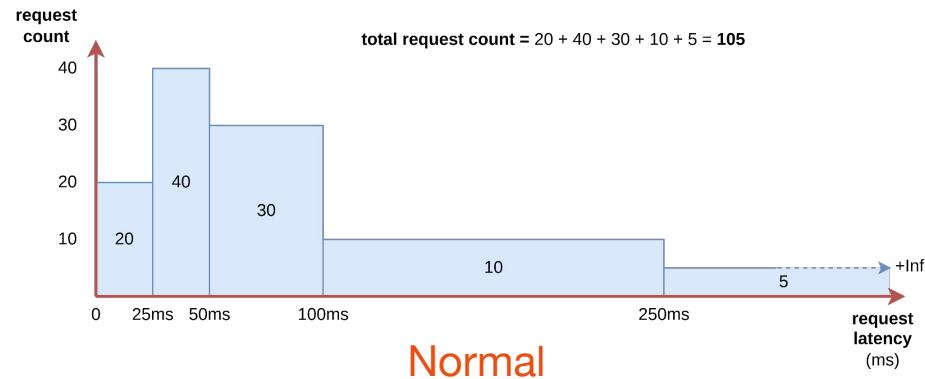
Classic Histograms

Histograms In General

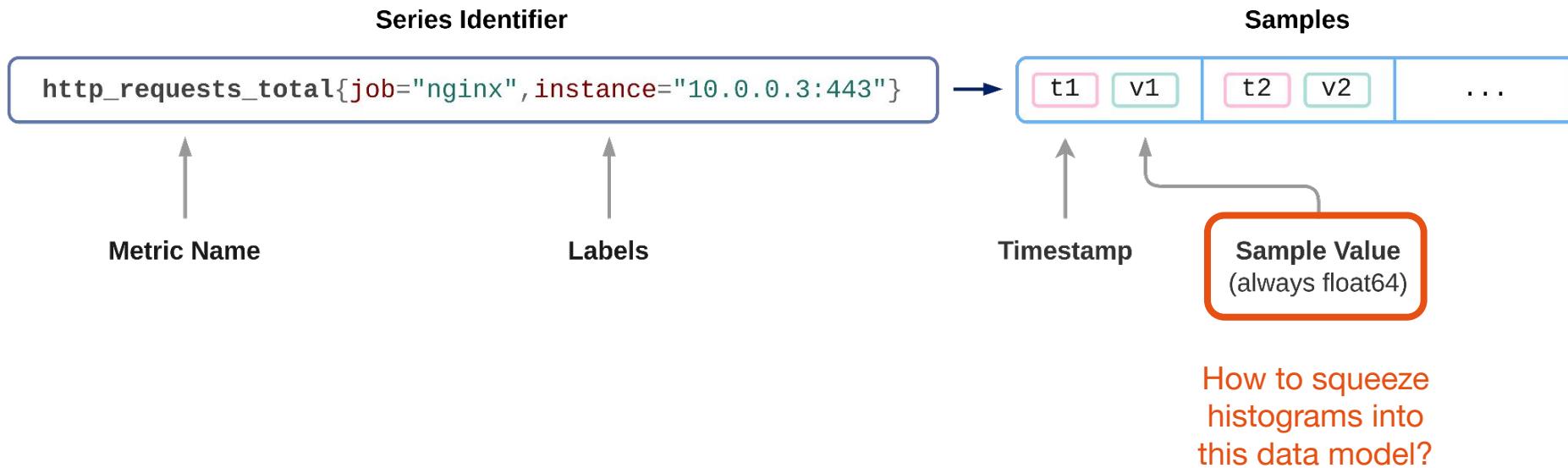
Way to track the distribution of a set of values, e.g. request durations:



Cumulative Representation

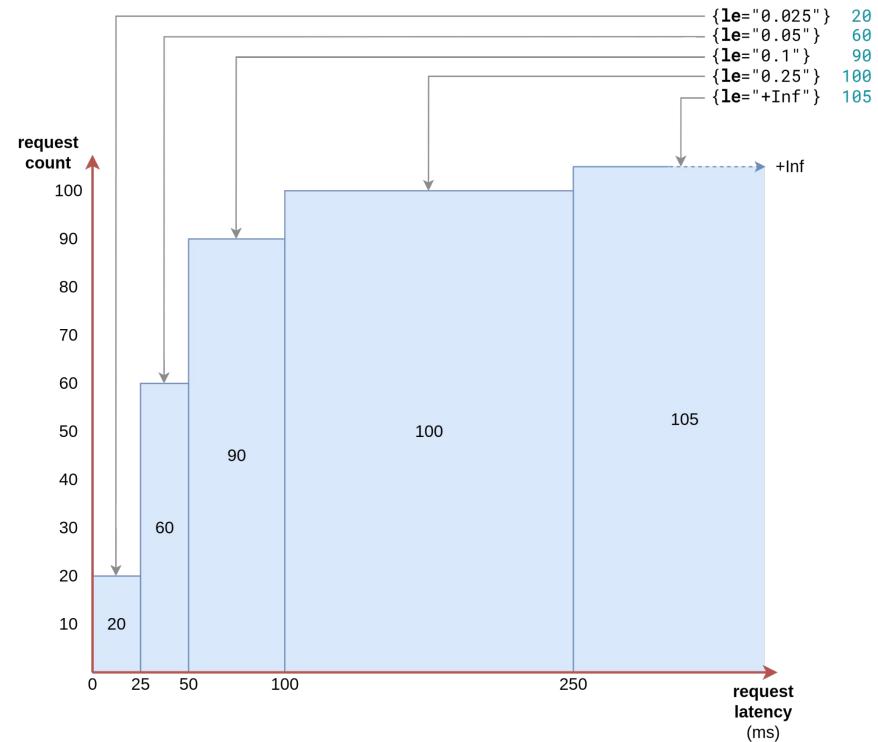


Classic Prometheus Data Model



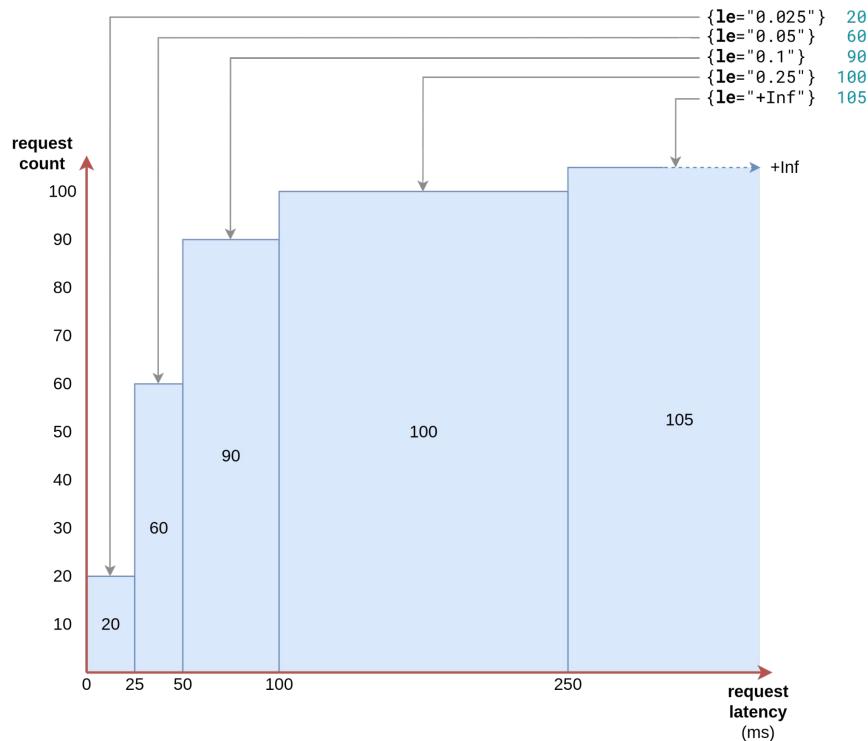
Classic Histograms

(le = "less than or equal to")



- Cumulative histogram
- One counter series per bucket
- Manual bucket configuration

Serializing Classic Histograms



```
# le = "less than or equal"
foo_bucket{le="0.025"} 20
foo_bucket{le="0.05"} 60
foo_bucket{le="0.1"} 90
foo_bucket{le="0.25"} 100
foo_bucket{le="+Inf"} 105
foo_sum 21.322
foo_count 105
```

What's Great About Classic Histograms?

- Efficient at **high-frequency** observations
- Computing **percentages / ratios / averages**
- Computing **percentiles**
- **Partitioning** by dimensions (with caveats!)
- **Aggregating** over dimensions

What Sucks About Classic Histograms?

- **Expensive!** (one series per bucket)
- **Manual + static** bucket configuration

In practice, **huge tradeoff** between **cost** and
resolution / dimensional partitioning.

Lots of overloaded Prometheus servers :(

Can We Improve That While
Keeping the Great Features?

Enter: Native Histograms!

First Question

One series per bucket is expensive.

Can we store an **entire histogram**
in the **sample value** of a single series?

Yes! New Prometheus Data Model:

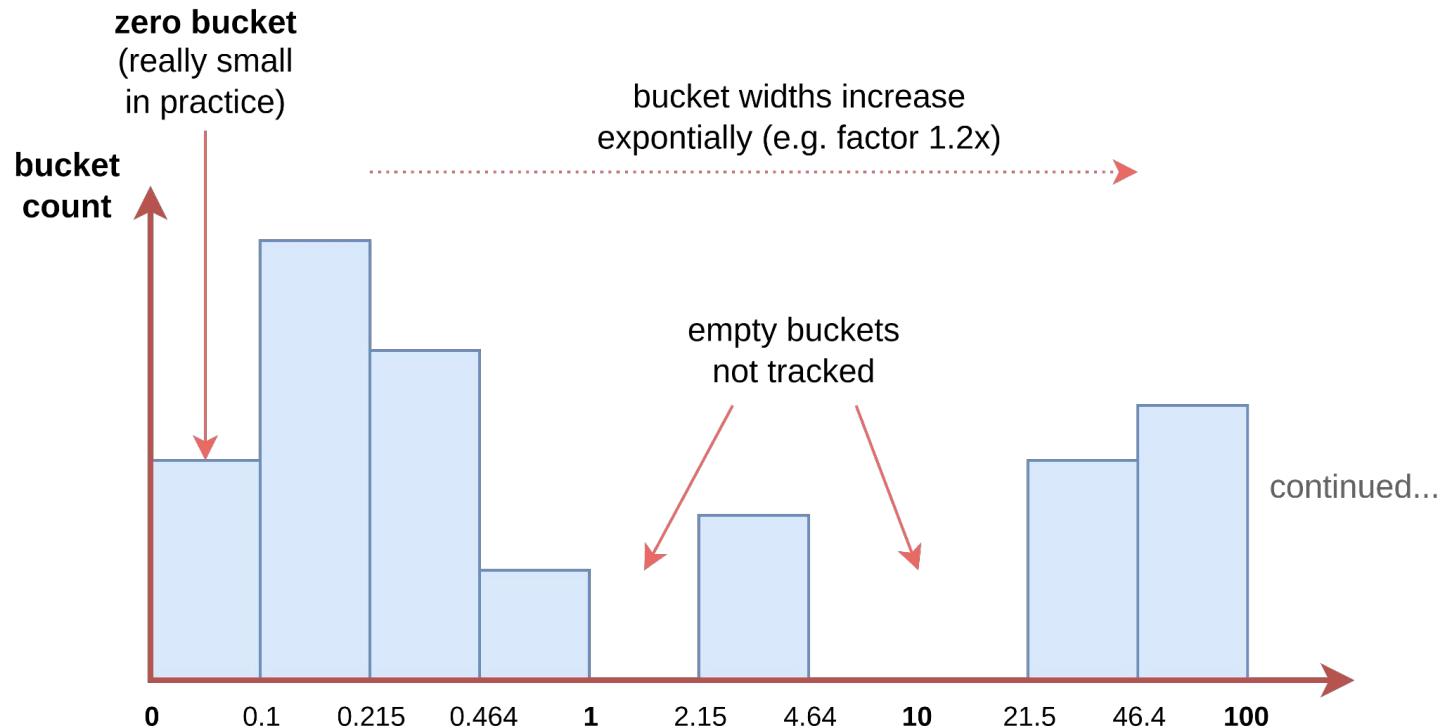


Second Question

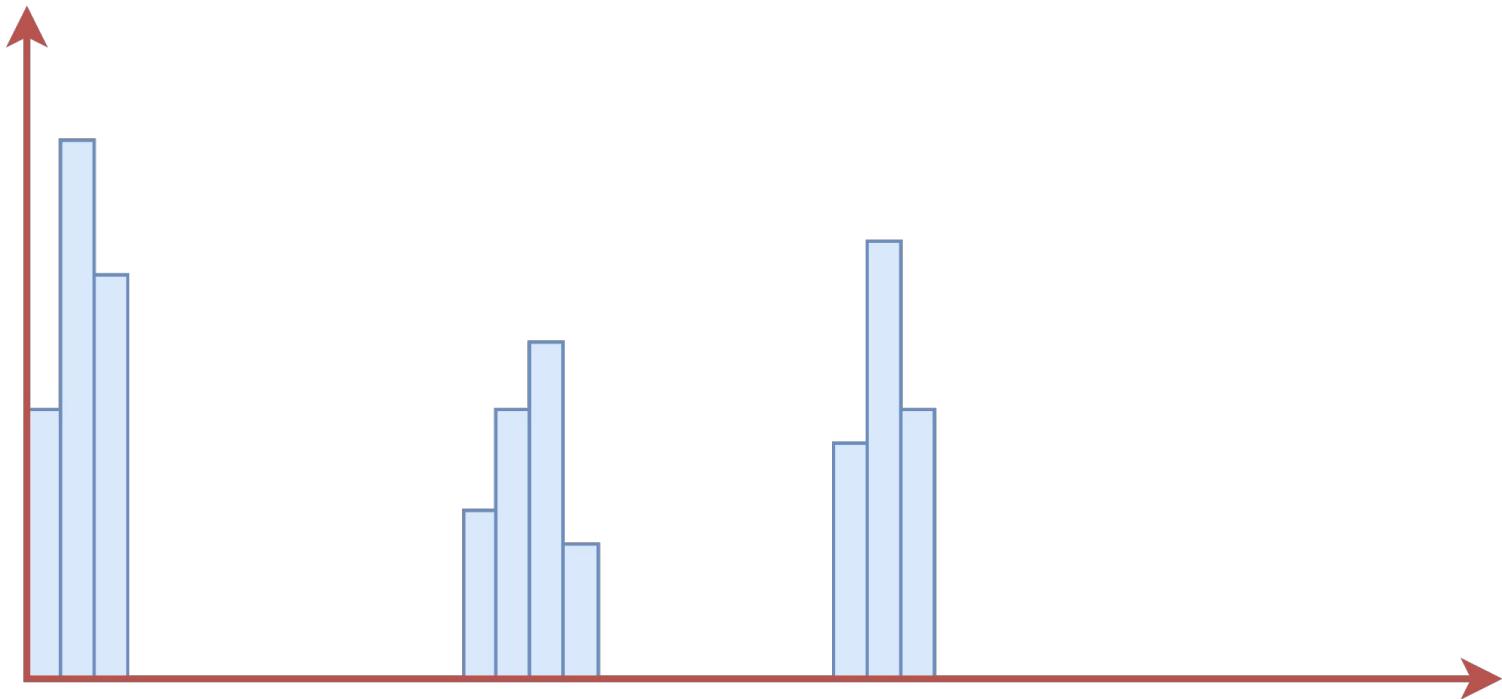
Can we have **sparse, high-resolution histograms** with no **manual bucket config**?

These exist in other systems, but how to make these work with Prometheus' **pull model**?

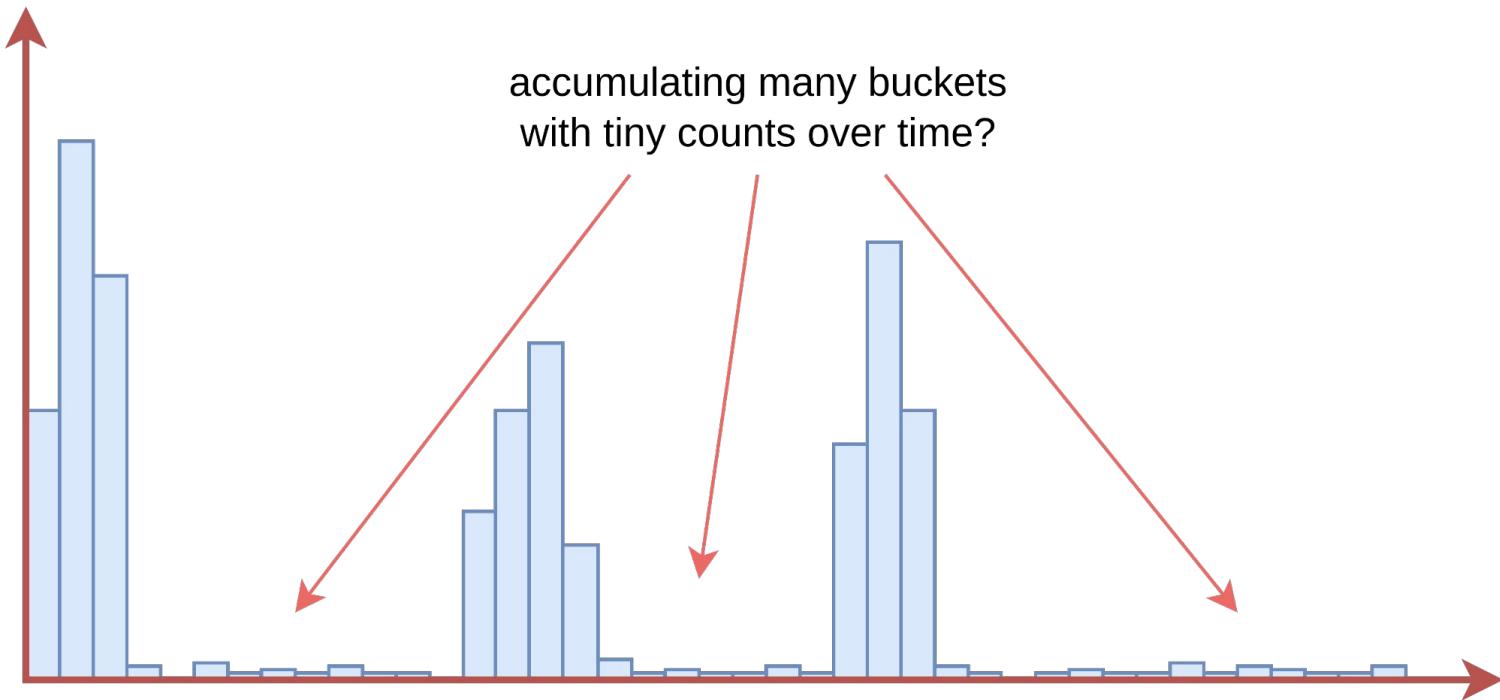
Native Histograms



Sparse Bucket Usage



Accumulating Entropy Over Time?



Solution: Resets!

- Buckets are counters.
- Counters should always be rate-ed before use.
- Counters may occasionally reset anyway.
- `rate()` and friends deal fine with resets.

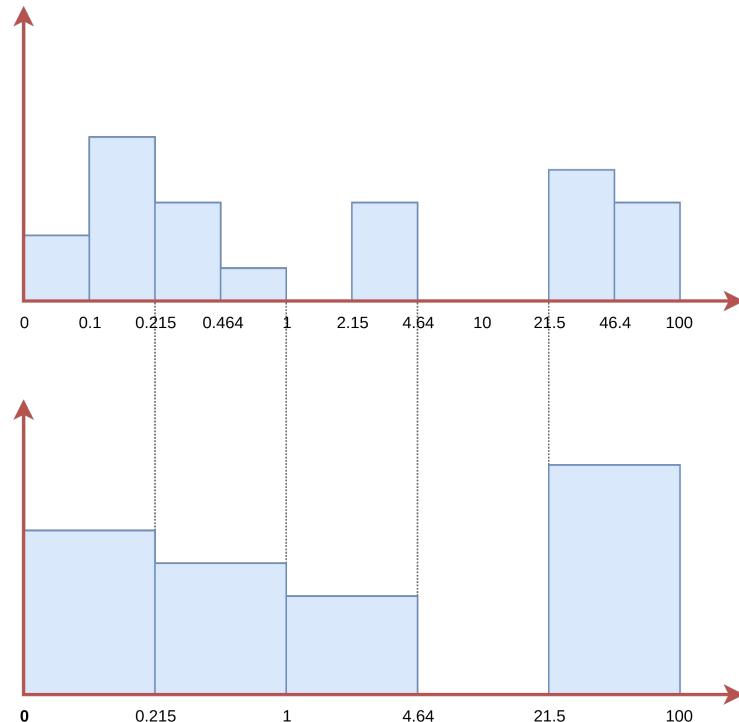
Just **reset the histogram** completely once in a while!

Resolution Downgrade + Aggregability

Real bucket factors are always $2^{2^{-n}}$.

⇒ All possible schemas vary by multiples of 2 in their bucket density, so can be downgraded / aggregated!

n	Factor
8	1.002711275
7	1.005429901
6	1.010889286
5	1.021897149
4	1.044273782
3	1.090507733
2	1.189207115
1	1.414213562
0	2
-1	4
-2	16
-3	256
-4	65536



Native Histogram Advantages

- Way fewer time series
- Much better bucket encoding
- Sparse - Unused buckets are free
- Much higher resolution possible
- All schemas aggregatable
- Easier configuration

Enabling Native Histograms in Prometheus

Still experimental!

Enable content negotiation + ingestion using:

```
./prometheus --enable-feature=native-histograms
```

Scraping Native Histograms

No text format support yet, currently requires new **protobuf-based scrape format** (implemented in Go client library).

Server negotiates format:

```
Accept: application/vnd.google.protobuf;proto=io.prometheus.client.  
MetricFamily;encoding=delimited,application/openmetrics-text;versio  
n=1.0.0;q=0.8,application/openmetrics-text;version=0.0.1;q=0.75,  
tex  
t/plain;version=0.0.4;q=0.5,*/*;q=0.1
```

Native Histograms In Instrumentation

Classic histograms:

```
requestDurations := prometheus.NewHistogram(prometheus.HistogramOpts{  
    Name:      "http_request_duration_seconds",  
    Help:      "A histogram of the HTTP request durations in seconds.",  
    // Manual configuration of upper bucket bounds.  
    Buckets: []float64{0.05, 0.1, 0.25, 0.5, 1, 2.5, 5, 10},  
})
```

Native Histograms In Instrumentation

Native histograms:

```
requestDurations := prometheus.NewHistogram(prometheus.HistogramOpts{  
    Name:      "http_request_duration_seconds",  
    Help:      "A histogram of the HTTP request durations in seconds.",  
    // Increase each subsequent bucket width by a factor of ~1.1x.  
    NativeHistogramBucketFactor: 1.1,  
})
```

Optional Settings

Optional settings to configure details:

```
// Zero bucket upper bound.  
NativeHistogramZeroThreshold float64  
// Max bucket number.  
NativeHistogramMaxBucketNumber uint32  
// Minimum duration before resetting buckets.  
NativeHistogramMinResetDuration time.Duration  
// Maximum upper bound to increase zero bucket to (when >max buckets).  
NativeHistogramMaxZeroThreshold float64
```

Very simplified! For full details, see:

https://pkg.go.dev/github.com/prometheus/client_golang/prometheus#HistogramOpts

Native Histograms in PromQL

Querying a Native Histogram

demo_api_request_duration_seconds

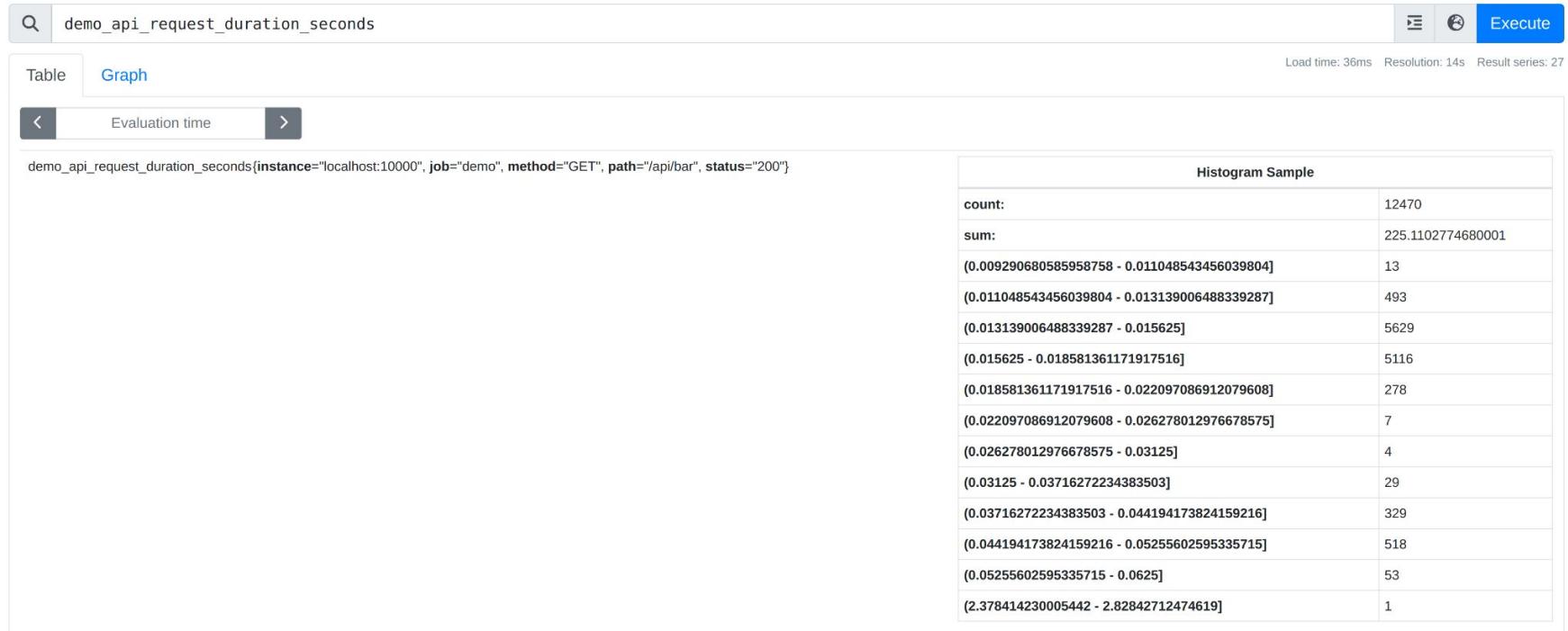
Load time: 31ms Resolution: 14s Result series: 27

Table Graph

Evaluation time < >

demo_api_request_duration_seconds{instance="localhost:10000", job="demo", method="GET", path="/api/bar", status="200"}	{ count:999 sum:19.90158758199998 (0.011048543456039804,0.013139006488339287]:37 (0.013139006488339287,0.015625]:396 (0.015625,0.018581361171917516]:395 (0.018581361171917516,0.022097086912079608]:27 (0.03125,0.03716272234383503]:5 (0.03716272234383503,0.044194173824159216]:52 (0.044194173824159216,0.05255602595335715]:82 (0.05255602595335715,0.0625]:5 }
demo_api_request_duration_seconds{instance="localhost:10000", job="demo", method="GET", path="/api/bar", status="500"}	{ count:10 sum:0.3622726599999999 (0.013139006488339287,0.015625]:2 (0.015625,0.018581361171917516]:1 (0.03716272234383503,0.044194173824159216]:4 (0.044194173824159216,0.05255602595335715]:2 (0.05255602595335715,0.0625]:1 }
demo_api_request_duration_seconds{instance="localhost:10000", job="demo", method="GET", path="/api/foo", status="200"}	{ count:1660 sum:26.039021972999993 (0.006569503244169644,0.0078125]:5 (0.0078125,0.009290680585958758]:105 (0.009290680585958758,0.011048543456039804]:659 (0.011048543456039804,0.013139006488339287]:455 (0.013139006488339287,0.015625]:11 (0.018581361171917516,0.022097086912079608]:1 (0.022097086912079608,0.026278012976678575]:30 (0.026278012976678575,0.03125]:224 (0.03125,0.03716272234383503]:165 (0.03716272234383503,0.044194173824159216]:5 }
demo_api_request_duration_seconds{instance="localhost:10000", job="demo", method="GET", path="/api/foo", status="500"}	{ count:28 sum:0.747962609 (0.009290680585958758,0.011048543456039804]:3 (0.011048543456039804,0.013139006488339287]:2 (0.022097086912079608,0.026278012976678575]:5 (0.026278012976678575,0.03125]:8 (0.03125,0.03716272234383503]:9 (0.03716272234383503,0.044194173824159216]:1 }

Working on a Better Visualization...



Calculating Rates

rate(demo_api_request_duration_seconds[5m])		Load time: 46ms Resolution: 14s Result series: 27
Table	Graph	
	Evaluation time	< >
{instance="localhost:10000", job="demo", method="GET", path="/api/bar", status="200"}	{ count:10.495643000000001 sum:0.17934558302538336 (0.009290680585958758,0.011048543456039804]:0.007334481481481483 (0.011048543456039804,0.013139006488339287]:0.4437361296296297 (0.013139006488339287,0.015625]:4.70140262962963 (0.015625,0.018581361171917516]:4.540044037037037 (0.018581361171917516,0.022097086912079608]:0.2897120185185186 (0.026278012976678575,0.03125]:0.0036672407407407413 (0.03125,0.03716272234383503]:0.014668962962962965 (0.03716272234383503,0.044194173824159216]:0.17236031481481484 (0.044194173824159216,0.05255602595335715]:0.311715462962963 (0.05255602595335715,0.0625]:0.011001722222222223 }	
{instance="localhost:10000", job="demo", method="GET", path="/api/bar", status="500"}	{ count:0.03300516666666667 sum:0.0009249296615506663 (0.013139006488339287,0.015625]:0.0036672407407407413 (0.015625,0.018581361171917516]:0.014668962962962965 (0.03716272234383503,0.044194173824159216]:0.01100172222222223 (0.044194173824159216,0.05255602595335715]:0.0036672407407407413 }	
{instance="localhost:10000", job="demo", method="GET", path="/api/foo", status="200"}	{ count:19.00364151851852 sum:0.2160229423373014 (0.006569503244169644,0.0078125]:0.07701205555555557 (0.0078125,0.009290680585958758]:1.9473048333333336 (0.009290680585958758,0.011048543456039804]:9.369800092592595 (0.011048543456039804,0.013139006488339287]:6.67804538888889 (0.013139006488339287,0.015625]:0.19069651851851854 (0.018581361171917516,0.022097086912079608]:0.007334481481481483 (0.022097086912079608,0.026278012976678575]:0.05500861111111112 (0.026278012976678575,0.03125]:0.326384425925926 (0.03125,0.03716272234383503]:0.34105338888888889 (0.03716272234383503,0.044194173824159216]:0.011001722222222223 }	
{instance="localhost:10000", job="demo", method="GET", status="200"}	{ count:0.19069651851854 sum:0.0035245753210655564 (0.0078125,0.009290680585958758]:0.01100172222222223 (0.009290680585958758,0.011048543456039804]:0.06967757407407409 (0.011048543456039804,0.013139006488339287]:0.02933792592592593 }	

Aggregating Over Dimensions

sum by(path, method) (rate(demo_api_request_duration_seconds[5m]))

Load time: 38ms Resolution: 14s Result series: 5

Table Graph

Evaluation time < >

{method="GET", path="/api/bar"}	{ count:45.5788413673182 sum:0.8028927489200216 (0.0078125,0.009290680585958758]:0.003508771929824561 (0.009290680585958758,0.011048543456039804]:0.07017534010533874 (0.011048543456039804,0.013139006488339287]:1.9263114066173985 (0.013139006488339287,0.015625]:20.122760061886915 (0.015625,0.018581361171917516]:19.242061262235524 (0.018581361171917516,0.022097086912079608]:1.1719267467128884 (0.022097086912079608,0.026278012976678575]:0.003508771929824561 (0.026278012976678575,0.03125]:0.003508747307036442 (0.03125,0.03716272234383503]:0.08421030471069639 (0.03716272234383503,0.044194173824159216]:1.042102800879083 (0.044194173824159216,0.05255602595335715]:1.719293936626114 (0.05255602595335715,0.0625]:0.18947321637755205 }
{method="GET", path="/api/foo"}	{ count:83.47348971512295 sum:0.9728647667771257 (0.006569503244169644,0.0078125]:0.31227988920237804 (0.0078125,0.009290680585958758]:7.999980720356901 (0.009290680585958758,0.011048543456039804]:40.715695193028495 (0.011048543456039804,0.013139006488339287]:29.26659863390309 (0.013139006488339287,0.015625]:0.7859630901943808 (0.018581361171917516,0.022097086912079608]:0.01754378578075845 (0.022097086912079608,0.026278012976678575]:0.29473607879883146 (0.026278012976678575,0.03125]:2.2315739489424327 (0.03125,0.03716272234383503]:1.7894693998453937 (0.03716272234383503,0.044194173824159216]:0.05964897507028882 }
{method="GET", path="/api/nonexistent"}	{ count:2.5157836134606377 sum:0.00008186569528666895 (0.000004536465129862675,0.000005394796609394436]:0.003508747307036442 (0.000006415530511884418,0.00000762939453125]:0.003508771929824561 (0.00000762939453125,0.00000907293025972535]:0.003508771929824561 (0.000012831061023768835,0.0000152587890625]:0.0210525823337113 (0.0000152587890625,0.0000181458605194507]:0.08421045244742512 (0.0000181458605194507,0.000021579186437577742]:0.18596432133378688 (0.000021579186437577742,0.00002566212204753767]:0.22105213912318494 0 00002566212204753767 0 0000305175781251 0 6070161403607263 0 000030517578125 0 00003629172102890141 0 8215502612262055

Calculating Quantiles

```
histogram_quantile(  
    0.9,  
    sum by(method, path) (  
        rate(demo_api_request_duration_seconds[5m])  
    )  
)
```

Q Table Graph Load time: 21ms Resolution: 3s Result series: 5

Execute

Evaluation time	
{method="GET", path="/api/bar"}	0.0197084802952615
{method="GET", path="/api/foo"}	0.012671338816069253
{method="GET", path="/api/nonexistent"}	0.00004158369478303058
{method="POST", path="/api/bar"}	0.06001322530904167
{method="POST", path="/api/foo"}	0.0243032422412312

Calculating Quantiles (Classic vs. Native)



Fraction of Observations Within Range

```
# Within the last 5 minutes, which fraction of
# my requests finished within 50ms (interpolated)?
histogram_fraction(
    0,
    0.05,
    sum by(method, path) (
        rate(demo_api_request_duration_seconds[5m])
    )
)
```

Table Graph Load time: 36ms Resolution: 3s Result series: 5

Execute

< Evaluation time >

{method="GET", path="/api/bar"}	0.9860660782306007
{method="GET", path="/api/foo"}	1
{method="GET", path="/api/nonexistent"}	1
{method="POST", path="/api/bar"}	0.4352938236934613
{method="POST", path="/api/foo"}	1

Accessing the Count and Sum Fields

```
# Extract the "count" field, which is equivalent
# to the per-(path/method) request rate in this case.
histogram_count(
    sum by(path, method) (
        rate(demo_api_request_duration_seconds[5m])
    )
)
```

Load time: 21ms Resolution: 3s Result series: 5
 Execute

TableGraph

Evaluation time< >

Query	Value
{method="GET", path="/api/bar"}	63.3436370106538
{method="GET", path="/api/foo"}	112.72241064129149
{method="GET", path="/api/nonexistent"}	4.059634829328666
{method="POST", path="/api/bar"}	15.077139982090676
{method="POST", path="/api/foo"}	12.361360000457982

Accessing the Count and Sum Fields

```
# Average request handling duration.  
histogram_sum(rate(demo_api_request_duration_seconds[5m]))  
/  
histogram_count(rate(demo_api_request_duration_seconds[5m]))
```

Load time: 51ms Resolution: 3s Result series: 27

TableGraphExecute

<Evaluation time>

{instance="localhost:10000", job="demo", method="GET", path="/api/bar", status="200"}	0.017171290258045088
{instance="localhost:10000", job="demo", method="GET", path="/api/bar", status="500"}	0.029652652590909063
{instance="localhost:10000", job="demo", method="GET", path="/api/foo", status="200"}	0.0123609184425159
{instance="localhost:10000", job="demo", method="GET", path="/api/foo", status="500"}	0.0223724411320752
{instance="localhost:10000", job="demo", method="GET", path="/api/nonexistent", status="404"}	0.00003528826349892014
{instance="localhost:10000", job="demo", method="POST", path="/api/bar", status="200"}	0.06115579723658063
{instance="localhost:10000", job="demo", method="POST", path="/api/bar", status="500"}	0.09069541970000003
{instance="localhost:10000", job="demo", method="POST", path="/api/foo", status="200"}	0.026676985996028392
{instance="localhost:10000", job="demo", method="POST", path="/api/foo", status="500"}	0.047206546692307644
{instance="localhost:10001", job="demo", method="GET", path="/api/bar", status="200"}	0.01718281653899948

Load time: 51ms Resolution: 3s Result series: 27

Binary Operators

rate(demo_api_request_duration_seconds{method="GET"}[5m]) + ignoring(method) rate(demo_api_request_duration_seconds{method="POST"}[5m]) Execute

Table Graph Load time: 38ms Resolution: 14s Result series: 12

Evaluation time	
<	>

{instance="localhost:10000", job="demo", path="/api/bar", status="200"} { count:16.5859649122807 sum:0.4376887746035095 (0.009290680585958758,0.011048543456039804]:0.021052631578947364 (0.011048543456039804,0.013139006488339287]:0.5929824561403508 (0.013139006488339287,0.015625]:6.028070175438596 (0.015625,0.018581361171917516]:5.368421052631579 (0.018581361171917516,0.022097086912079608]:0.3157894736842105 (0.03125,0.03716272234383503]:0.02456140350877193 (0.03716272234383503,0.044194173824159216]:0.6912280701754385 (0.044194173824159216,0.05255602595335715]:2.1929824561403506 (0.05255602595335715,0.0625]:0.975438596491228 (0.0625,0.07432544468767006]:0.017543859649122806 (0.1051120519067143,0.125]:0.010526315789473682 (0.125,0.14865088937534013]:0.15087719298245614 (0.14865088937534013,0.17677669529663687]:0.1789473684210526 (0.17677669529663687,0.2102241038134286]:0.017543859649122806 }

{instance="localhost:10000", job="demo", path="/api/bar", status="500"} { count:0.10175438596491226 sum:0.007497379919298239 (0.013139006488339287,0.015625]:0.010526315789473682 (0.015625,0.018581361171917516]:0.017543859649122806 (0.018581361171917516,0.022097086912079608]:0.007017543859649122 (0.03716272234383503,0.044194173824159216]:0.007017543859649122 (0.044194173824159216,0.05255602595335715]:0.017543859649122806 (0.05255602595335715,0.0625]:0.003508771929824561 (0.0625,0.07432544468767006]:0.003508771929824561 (0.125,0.14865088937534013]:0.010526315789473682 (0.14865088937534013,0.17677669529663687]:0.021052631578947364 (0.17677669529663687,0.2102241038134286]:0.003508771929824561 }

{instance="localhost:10000", job="demo", path="/api/foo"} { count:27.47719298245614 sum:0.352869677592981 (0.006569503244169644,0.0078125]:0.09473684210526315 (0.0079125,0.00920069050507501,2.450640122907017]:0.00020069050507509 (0.0110410512456020020041,1.2.225097710200211]

Real-World Results

Detailed Talk by Björn

The image consists of two main parts. On the left is a screenshot of a presentation slide titled "Quantile calculation". It compares two methods: "Classic histograms: 348 ms" and "Native histograms: 433 ms". Each method is shown with a query example and a histogram graph. The histograms show two sharp peaks at approximately 100 and 400 seconds. On the right is a video frame of Björn Rabenstein, a man with dark hair and a beard, wearing a black t-shirt with a white logo, standing behind a podium and speaking. The podium has a sign that reads "Observability Day EUROPE". The background is a stage with blue lighting.

Quantile calculation

Classic histograms: 348 ms

```
Q histogram.quantile(0.9, sum by (le, request{status_code="200", route=~"(api/prometheus:api/v1/query){$m}}))
```

Native histograms: 433 ms

```
Q histogram.quantile(0.9, request.duration.seconds{status_code="200", route=~"(api/prometheus:api/v1/query){$m}})
```

Observability Day
EUROPE

[Prometheus Native Histograms in Production](#) - Björn Rabenstein

(Very) Rough Bottom Line

Storage: 10x resolution for half the price!

Query time: Roughly the same, some things a tad slower

Bucket entropy and resets were reasonable.

Grafana Heatmap Comparison



Still Experimental!

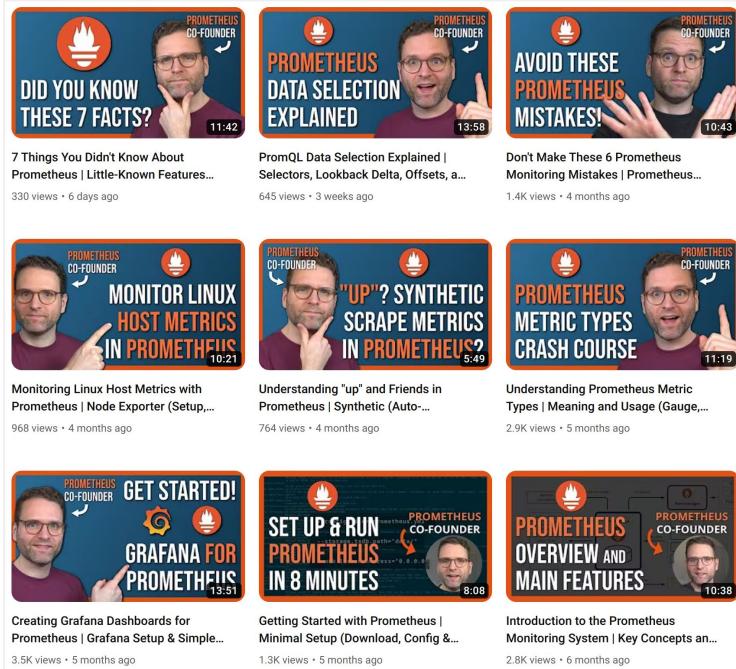
Try It Out and Give Feedback!

More Native Histogram Talks

- **Prometheus Native Histograms in Production**, Björn Rabenstein, Observability Day Amsterdam 2023
<https://www.youtube.com/watch?v=TgINvIK9SYc>
- **Native Histograms in Prometheus**, Ganesh Vernekar, PromCon EU 2022
<https://www.youtube.com/watch?v=AcmABV6NCYk>
- **PromQL for Native Histograms**, Björn Rabenstein, PromCon EU 2022
<https://www.youtube.com/watch?v=fikCqhlUOmQ>
- **Prometheus Sparse High-Resolution Histograms in Action**, Ganesh Vernekar, KubeCon NA 2022
<https://www.youtube.com/watch?v=T2GvcYNth9U>
- **A New Kid in Histogram Town**, Björn Rabenstein, KubeCon NA 2021
<https://www.youtube.com/watch?v=rM8tR2SPJ4M>
- **Better Histograms in Prometheus**, Björn Rabenstein, KubeCon NA 2020
<https://www.youtube.com/watch?v=HG7uzON-IDM>
- **Prometheus Histograms – Past, Present, and Future**, Björn Rabenstein, PromCon EU 2019
<https://www.youtube.com/watch?v=7sQFkaMCyEI>

Thanks! Learn More:

YouTube Channel:
<https://youtube.com/@PromLabs>



Self-Paced Prometheus Courses:
<https://training.promlabs.com/>

The screenshot shows the PromLabs Training homepage. At the top, there's a navigation bar with links for 'Trainings', 'PCA Certification', 'Team Plans', and 'Sign In'. Below the navigation, a section titled 'Learn Prometheus with PromLabs' is displayed. It includes a brief description: 'Learn how to monitor your systems and services more effectively with online trainings created by the **co-founder of Prometheus**. All trainings are self-paced and available without time limit.' Below this, there are two buttons: 'Start Learning' and 'Team Plans'. To the right, a screenshot of a Prometheus dashboard shows a graph with several data series over time. At the bottom, a circular profile picture of a man with glasses is shown with the caption 'By the creator of Prometheus' and a link 'About Julius'.

30% OFF! For the first 20 people
with coupon code **KCD-ZRH-2023**