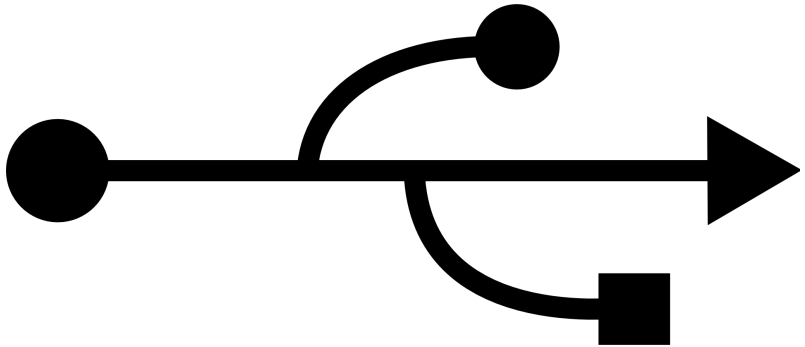
M3 and a new age of metrics and monitoring in an increasingly complex world

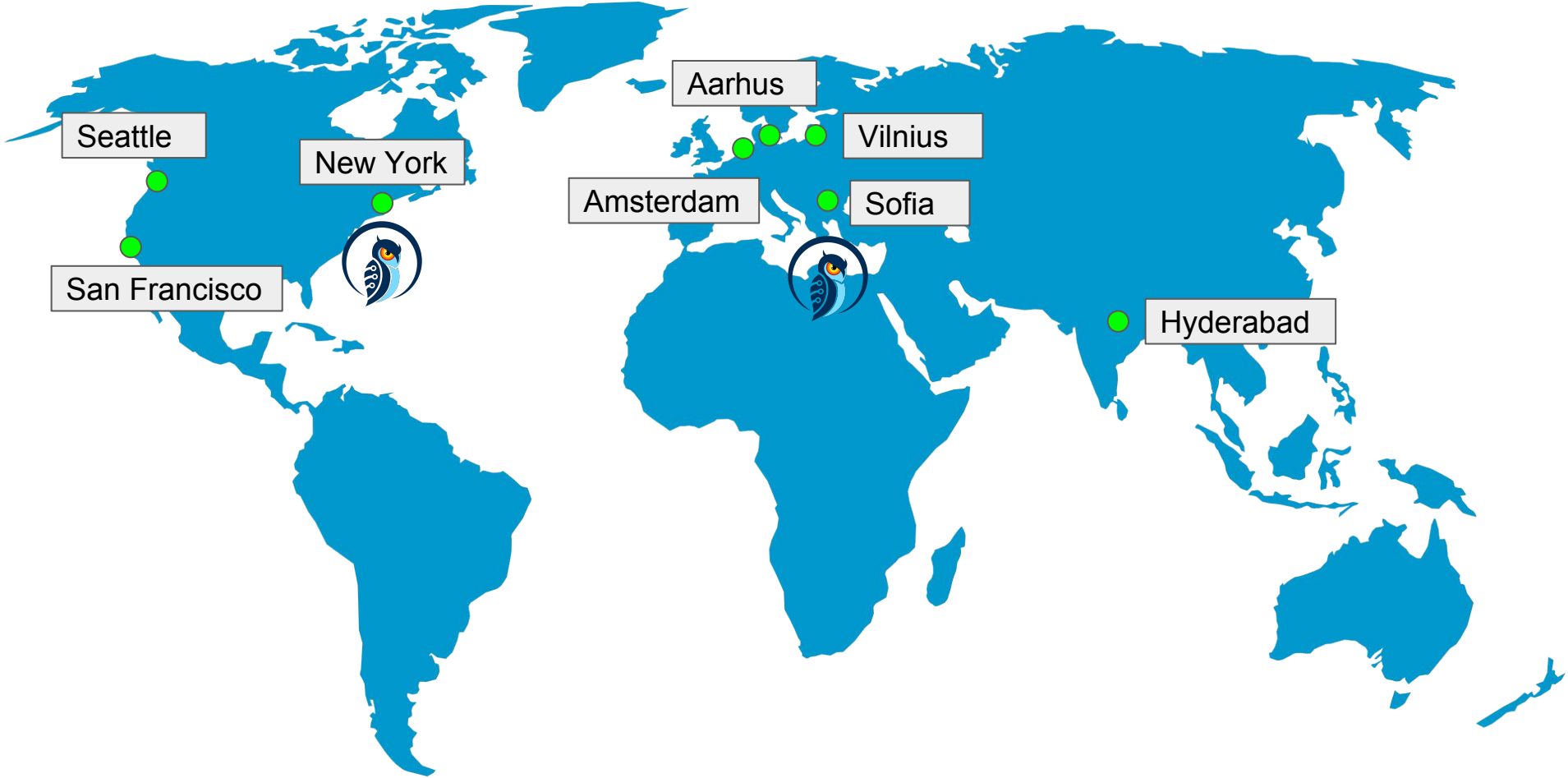
February 3, 2019
fosdem.org

Rob Skillington
Observability and M3, Uber NYC

Why am I here?

- Working on ~~monitoring~~ ~~“observability”~~ running computers now at Uber for 3 continuous years.
- Member of OpenMetrics, a CNCF project standardizing metrics exposition. See openmetrics.io for more info.




Seattle
New York
San Francisco
Aarhus
Vilnius
Amsterdam
Sofia
Hyderabad

Agenda

- High dimensionality metrics and monitoring in an increasingly complex world
- M3, Prometheus, Graphite

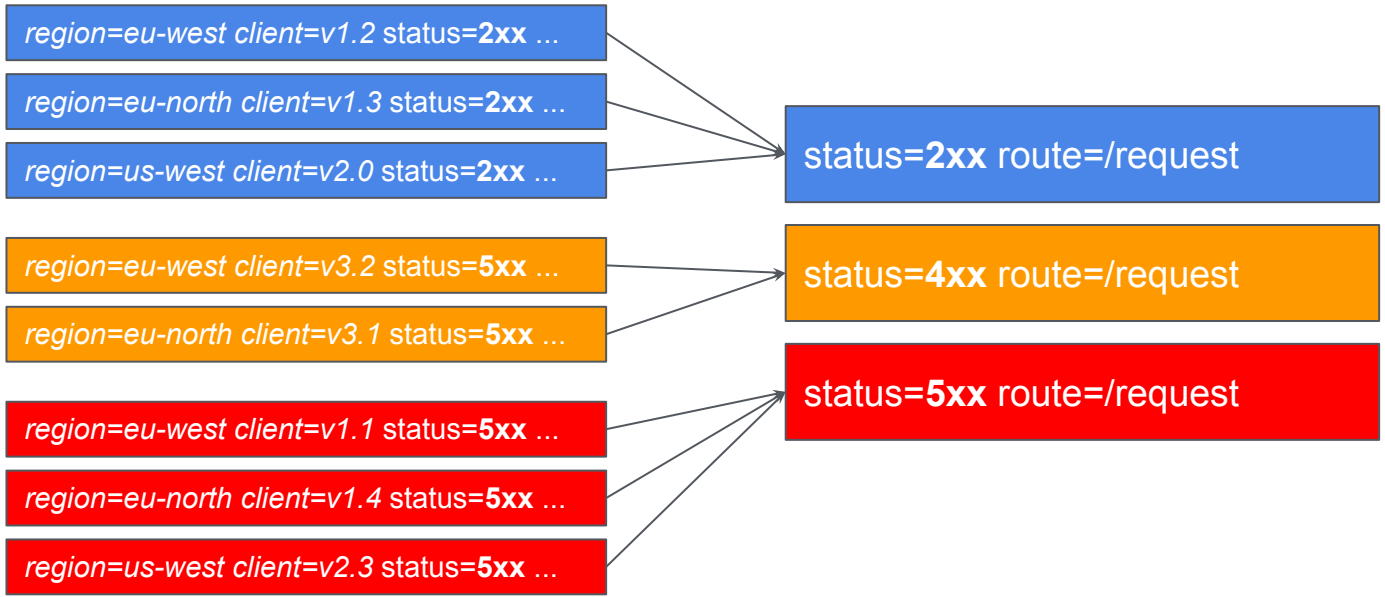
“High dimensionality metrics”
Huh?

High dimensionality metrics

- Take a single metric, such as status code delivered by our frontends
  - **http_status_code**
- At Uber, we have roughly a few hundred important HTTP routes we want to drill down into the status code with the following dimensions:
  - *Route (few hundred)*
  - *Status code (less than a few common status codes)*
  - *Places of operation (few hundred)*
    - *failures sometimes isolated to place of operation*
  - *App device version (few hundred)*

High dimensionality metrics

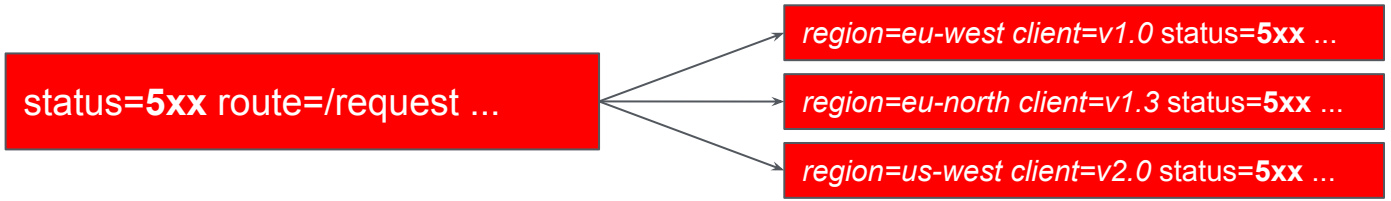
You can roll up metrics to make viewing fast. For example, view status codes by route, but across all regions for all app versions.



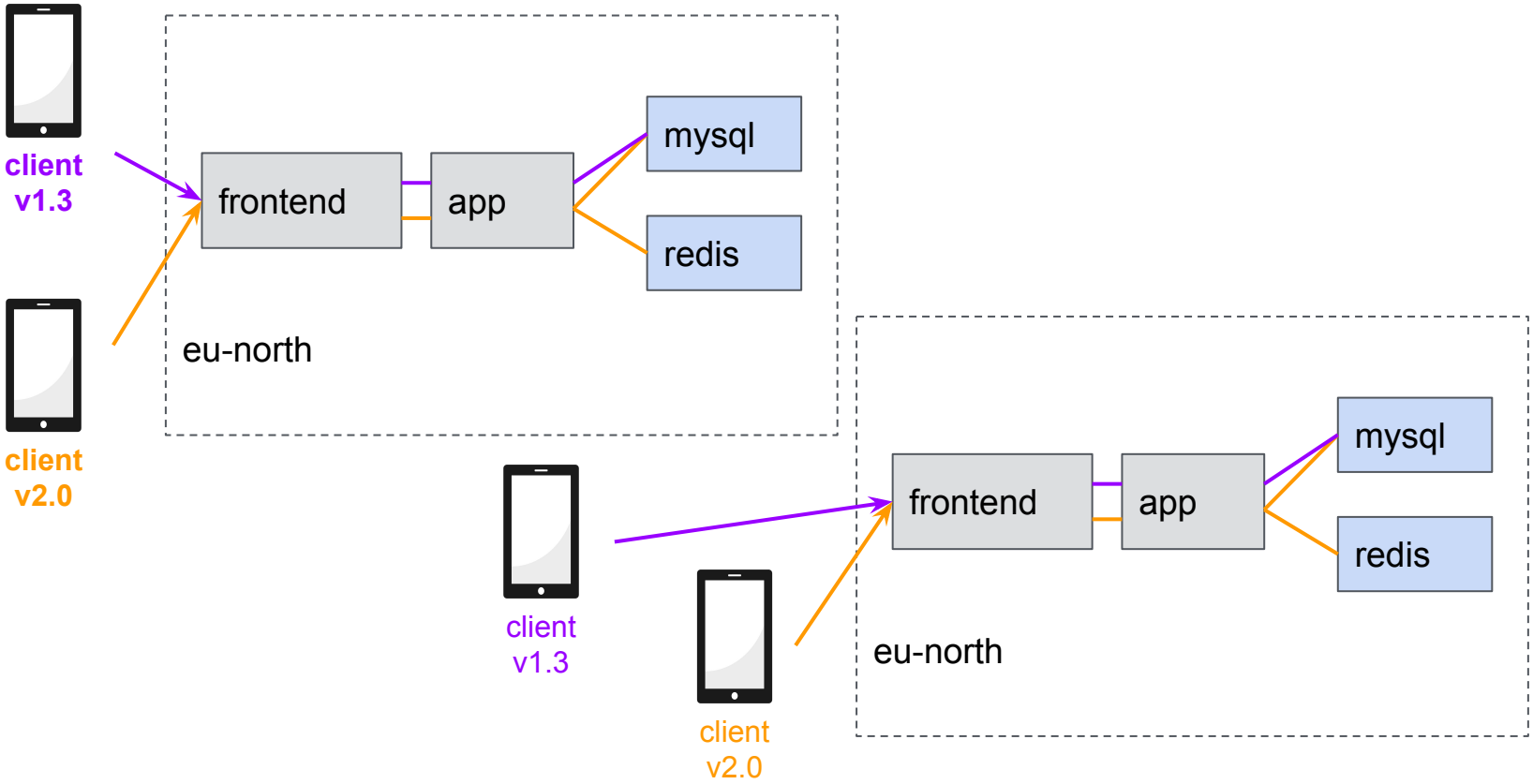
High dimensionality metrics

However, it is still incredibly high dimensional, just to store the raw data if you later want to drill down on.

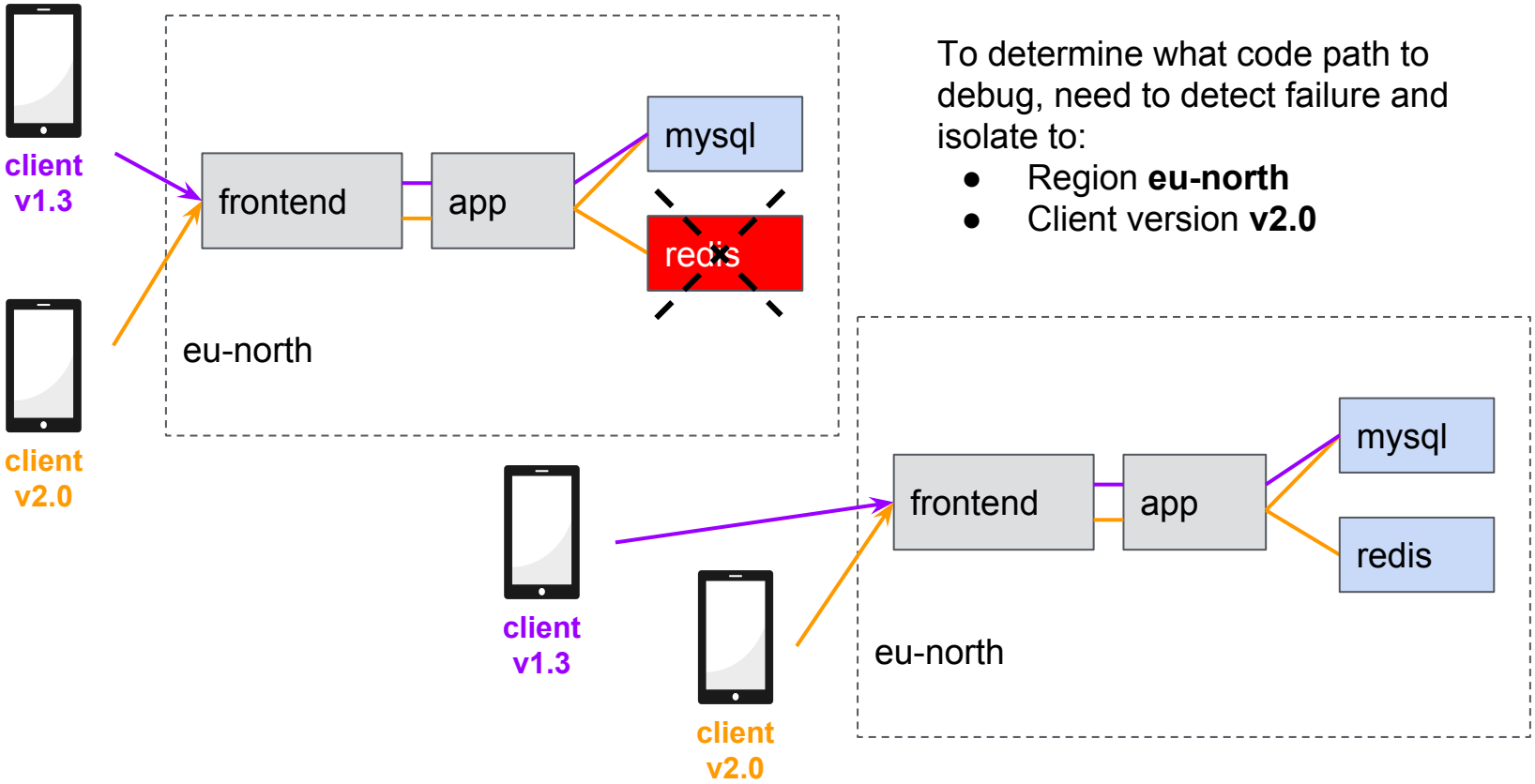
- Routes (eg 500) * Status code (eg 5) * Region (eg 5) * Client version (eg 20) = 250,000 unique time series
- Expensive but not too bad..? However add any other dimensions and it gets out of control (any multiplier on 250k explodes to millions quickly).



High dimensionality metrics



High dimensionality metrics



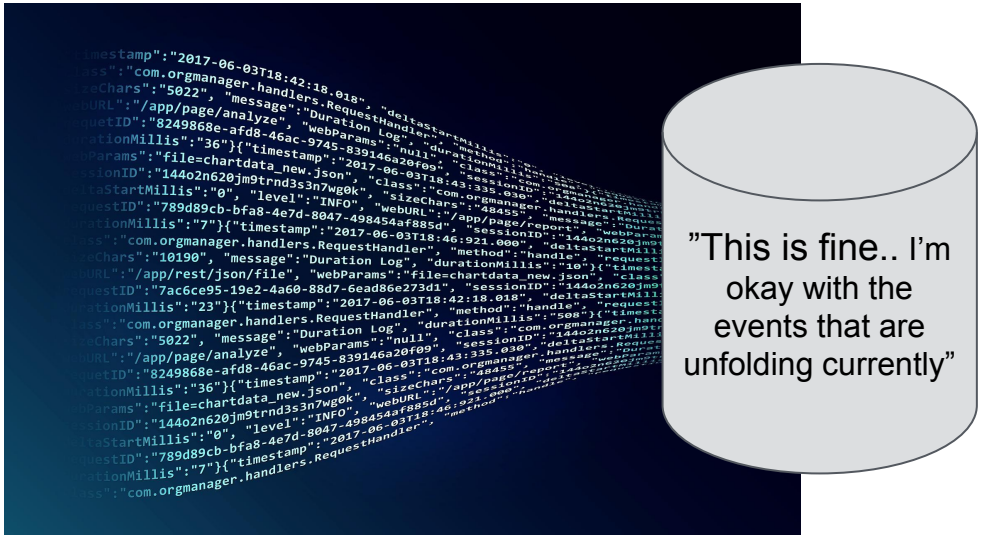
To determine what code path to debug, need to detect failure and isolate to:

- Region **eu-north**
- Client version **v2.0**

Ok great but what do I need?

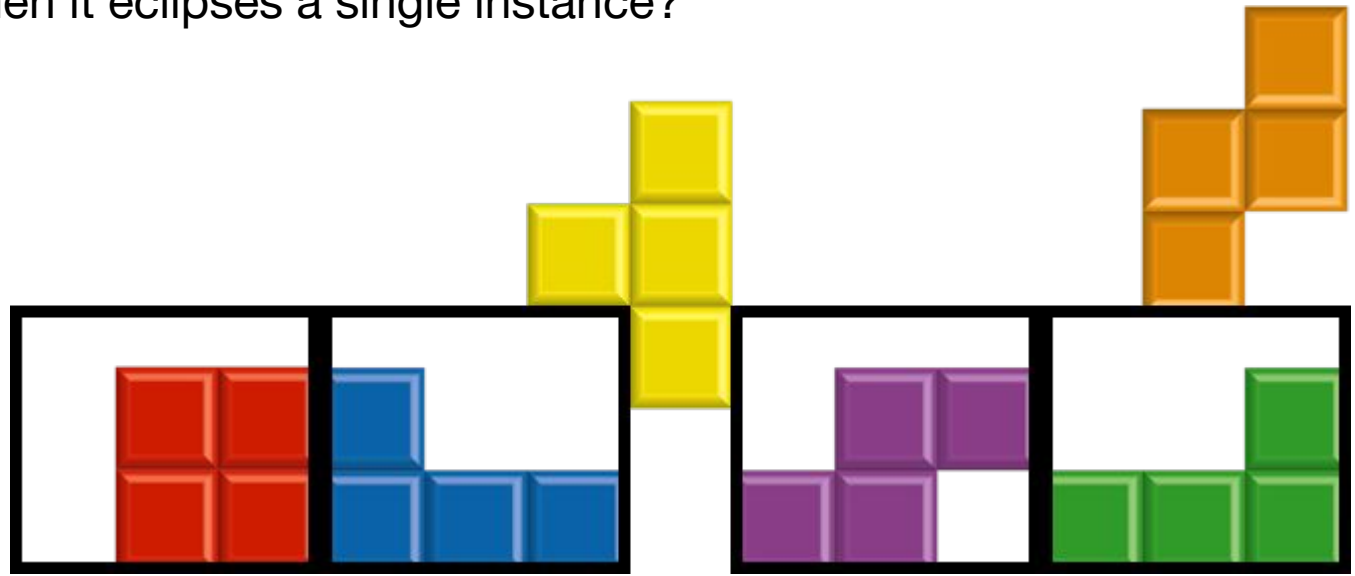
- A manageable way to scale out capacity
  - A single Prometheus instance can hold N million time series

    (where N does not usually reach the double digits, even on big machines)



Ok great but what do I need?

- It’s a little like playing tetris, can I fit my high cardinality metrics into an existing Prometheus instance or do I setup a new one, what happens when it eclipses a single instance?



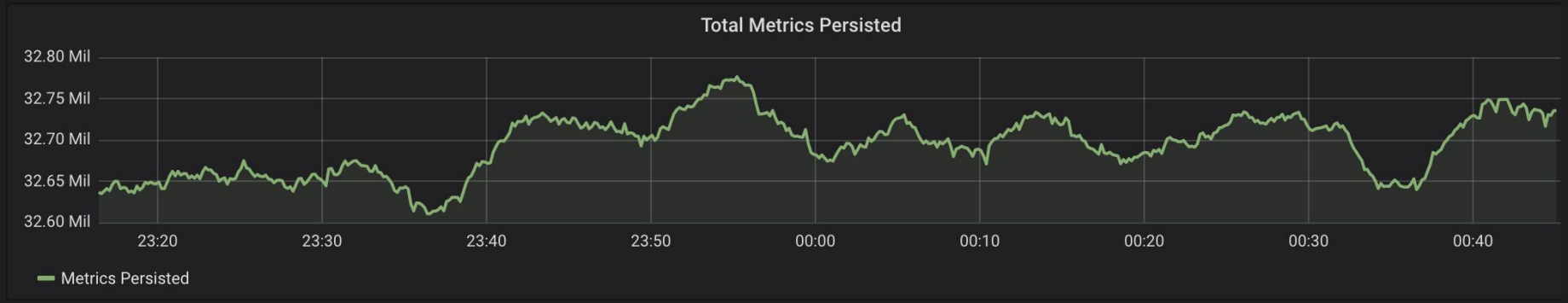
Metrics and Monitoring at Uber

Uber Workload - Ingress

~700M Pre-aggregated Metrics/s

(~130Gbps)

Storage

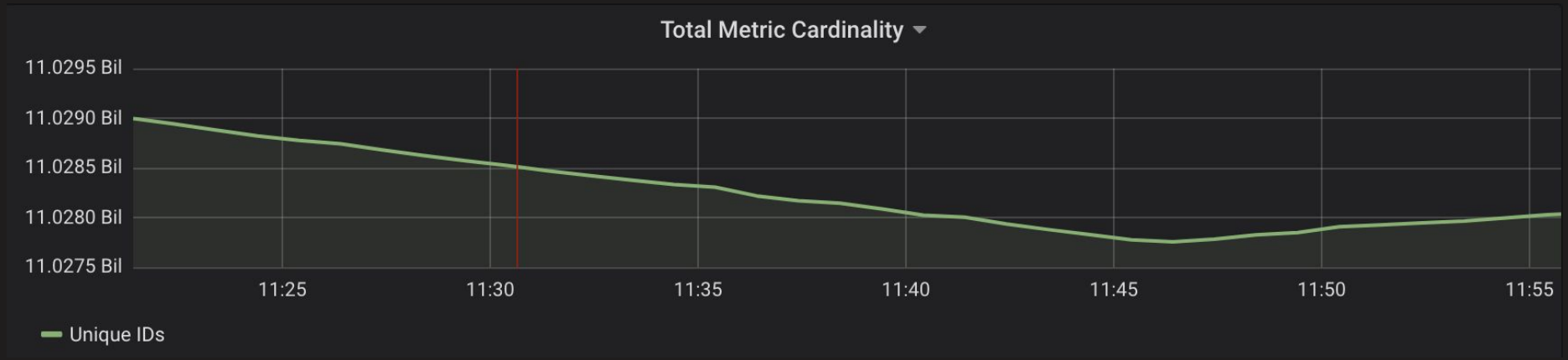


~33M Metrics Stored/s

(not including Replica Factor = 3)

(~50Gbps)

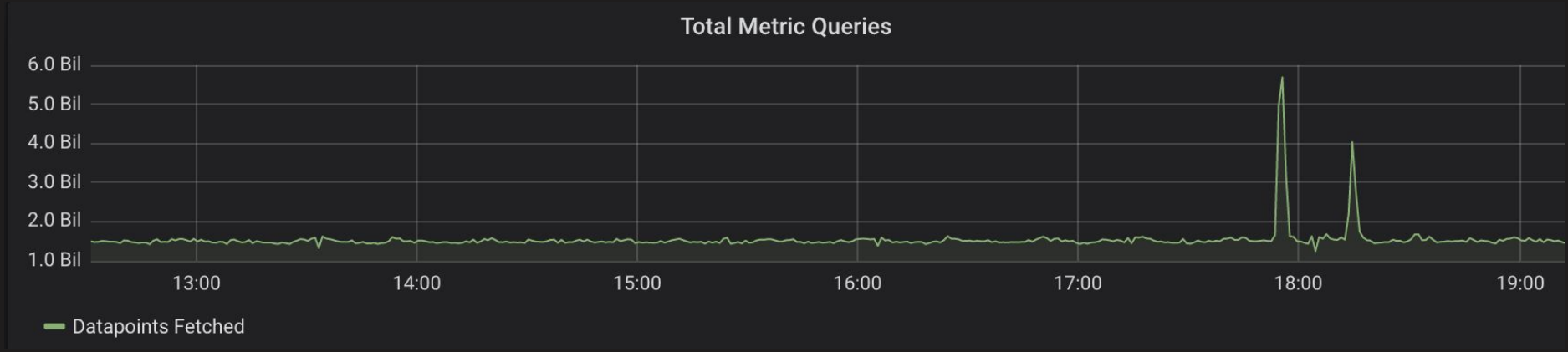
Index



~ 11B Unique Metric IDs

(Equivalent of thousands of Prom instances)

Egress



~ 1.5B data points per second (spikes to 5B+)

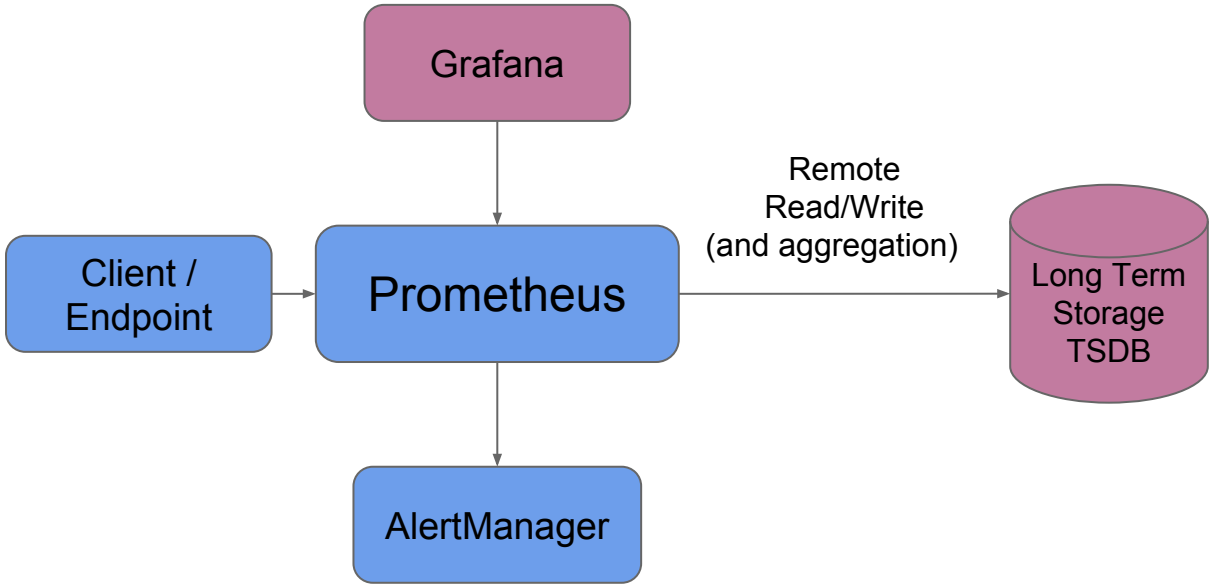
powers dashboards and 150,000 scheduled

“realtime” alerts

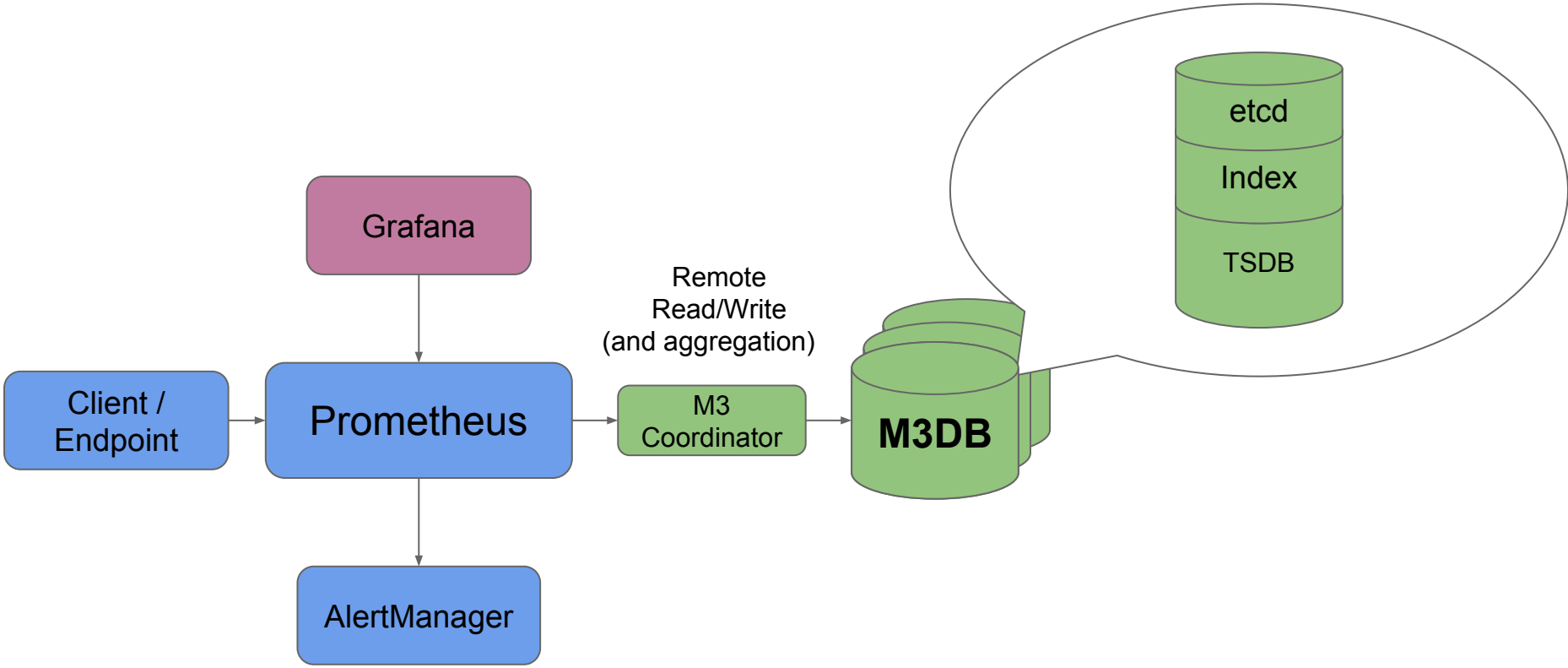
(~20 gigabits/sec)

M3, Prometheus and Graphite

Prometheus Long Term Storage



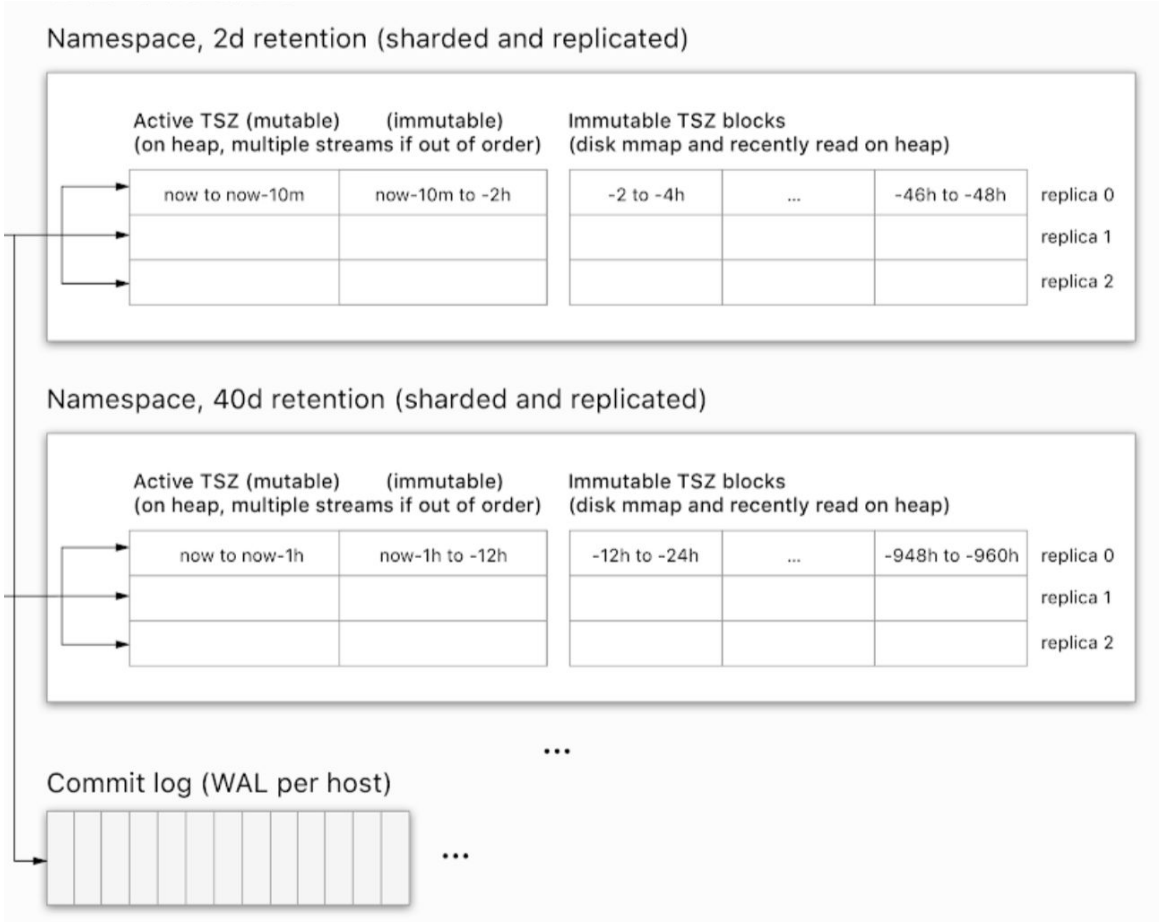
M3DB


Grafana
Client / Endpoint
Prometheus
AlertManager
Remote Read/Write (and aggregation)
M3 Coordinator
M3DB
etcd
Index
TSDB

M3DB High-Level Architecture

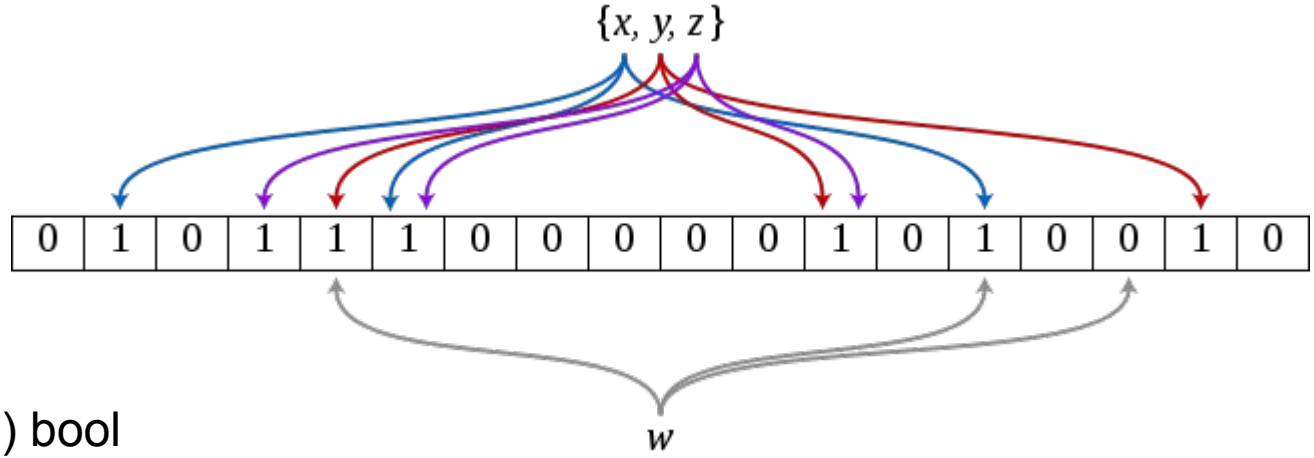
Think Log Structured Merge tree (LSM).. but with almost zero compaction.

Typical LSM will have levelled or size based compaction, M3DB has time window compaction which by will avoid any compaction of time series data files. Also downsampling is done as data is streamed to the aggregators.



Bloom filter for each time series data block

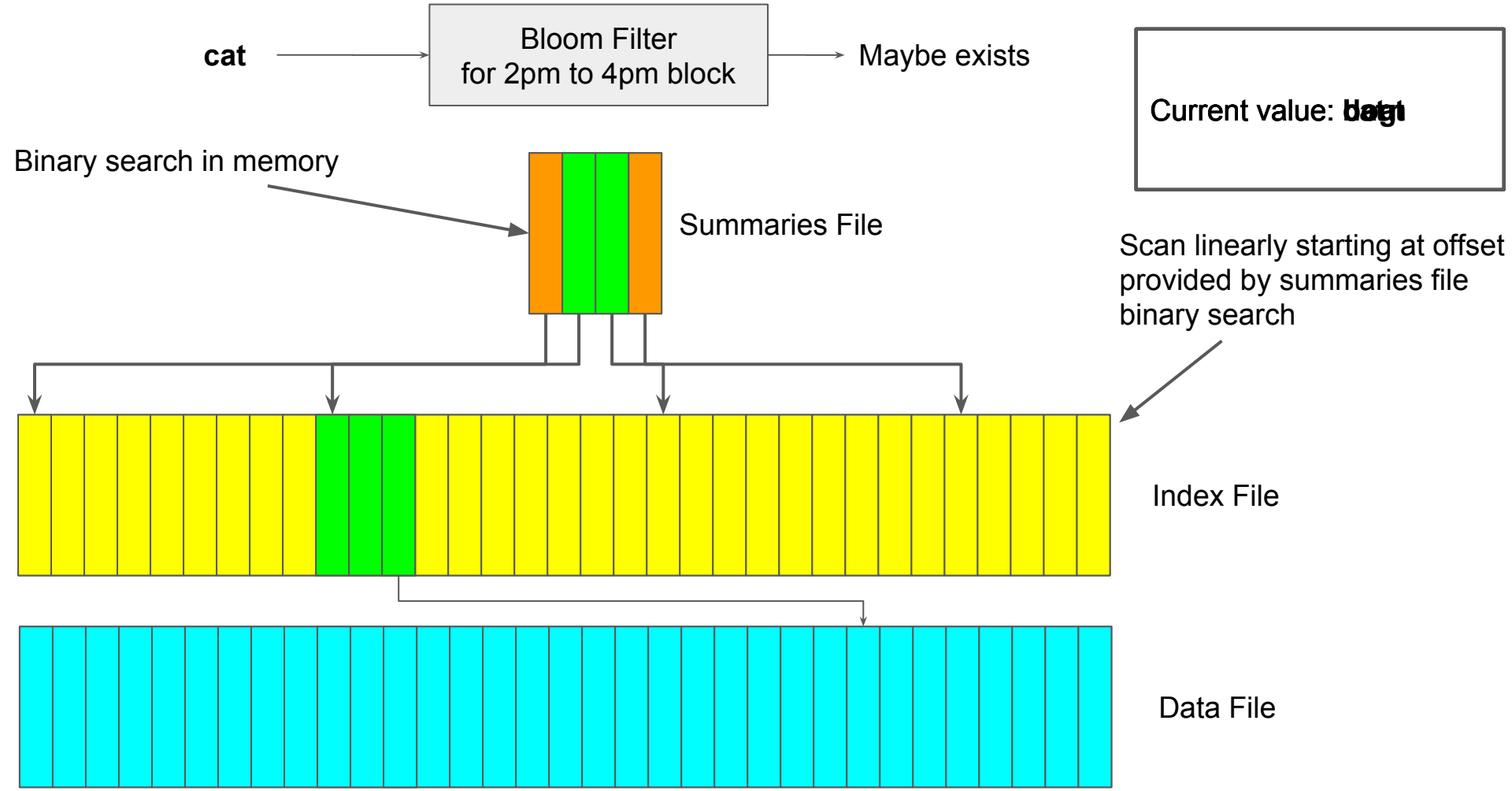
- Each time series data block contains a bloom filter, an index summaries file, a metadata per time series key and time series data file.
- The bloom filter is mmap'd and used for fast lookups to determine for large range scan, which blocks for should be searched on disk for a given metric.



Bloom filter API

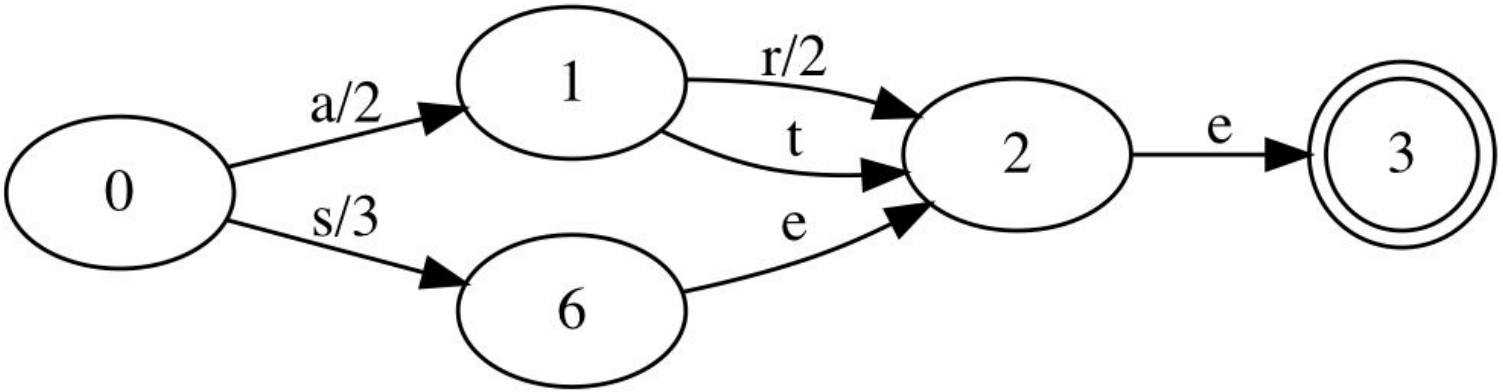
Add([]byte)

MaybeExists([]byte) bool


cat
Bloom Filter
for 2pm to 4pm block
Maybe exists
Current value: bat
Binary search in memory
Summaries File
Scan linearly starting at offset
provided by summaries file
binary search
Index File
Data File

Inverted Index

- The inverted index is similar to Lucene, it uses FST segments to built an efficient and compressed structure for fast regexp, using Roaring Bitmaps to capture metric IDs associated to a label value.
- Each metric's label/tag has it's own FST that when searched can find the Roaring Bitmap mmap offset for the set of metric IDs associated with a label value.



Postings Lists

- For every term combination in the form of `service="foo"` need to store a set of metric IDs (integers) that match, this is called a “postings list”.

- `{service="foo",endpoint="bar",client_version="3"}`
  - Do some index magic to find the 3 different sets that store `service="foo"`, `endpoint="bar"` and `client_version="3"`.
  - Calculate the intersection (AND) of those 3 sets, and then retrieve those documents

- Index is broken into blocks and segments, so also need to be able to calculate union (OR)
  - Find sets that store `service="foo"` postings list in the 12PM->2PM block and the 2PM->4 PM block and then calculate their union (OR)

Roaring Bitmap

- Bitmap Container
  - Exactly the same as a regular bitmap, but always from 0 -> 2^16
  - (2^16) / 8 -> Always uses exactly 8KiB of memory
- Sorted Array Container
  - Sorted array of uint16 -> `[]uint16`
  - Grows dynamically
  - 2 bytes of memory per value
    - Minimum size: 2 bytes!
    - Maximum size: 131 KiB :(
- “Run” Container
  - Optimized for long sequences of continuous values
  - Array of uint16 pairs -> `[]struct{start, length uint16}`
  - Ideal scenario: all numbers in a 2^16 range exist in the set -> 4 bytes of memory!

Roaring Bitmap

- Same APIs as a normal bitmap, but adapts to your workload
- Break the 2^32 number space into chunks of size 2^16
  - `(2^32) / (2^16) == 2^16 chunks of 2^16`
- For each chunk, store the values in that range within a “container”
  - Use a different container type for each chunk based on data density
- Can imagine it as a: `map[uint16]*Container`

Values Stored	0 -> 2^16	2^16 -> 2^17	2^17 -> 2^17 + 2^16	…	…	…	…	…
Key Prefix	0	1	2	3	4	5	6	7
Container	?	?	?	?	?	?	?	?

Only need keys/containers for chunks that have data in that range, otherwise we can leave them out entirely.

Roaring Bitmap

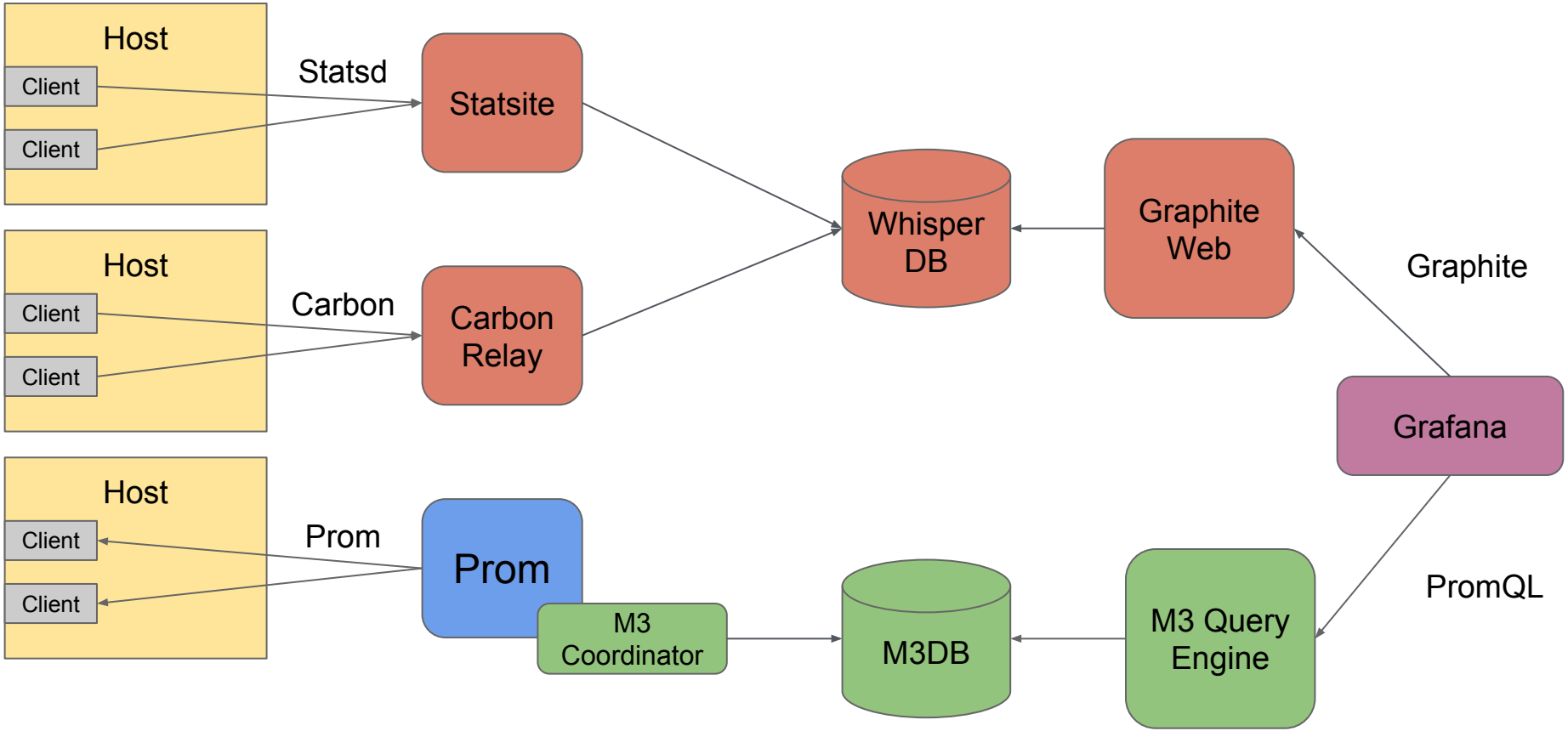
- Choose containers based on cardinality / presence of continuous sequences.
  - Can represent both sparse and dense sets efficiently.
  - Grows to very large integer spaces (uint64) due to the chunking mechanism.
- When union or intersection, stack the containers on top of each other and perform as efficient as possible set operation based on container types.

Key Prefix	0	1	2	3	4	5	6	7
Cardinality	1	4095	4096	8000	65,536	0 -> 9,500	0	0
Container	Array	Array	Bitmap	Bitmap	Run	Run	null	null

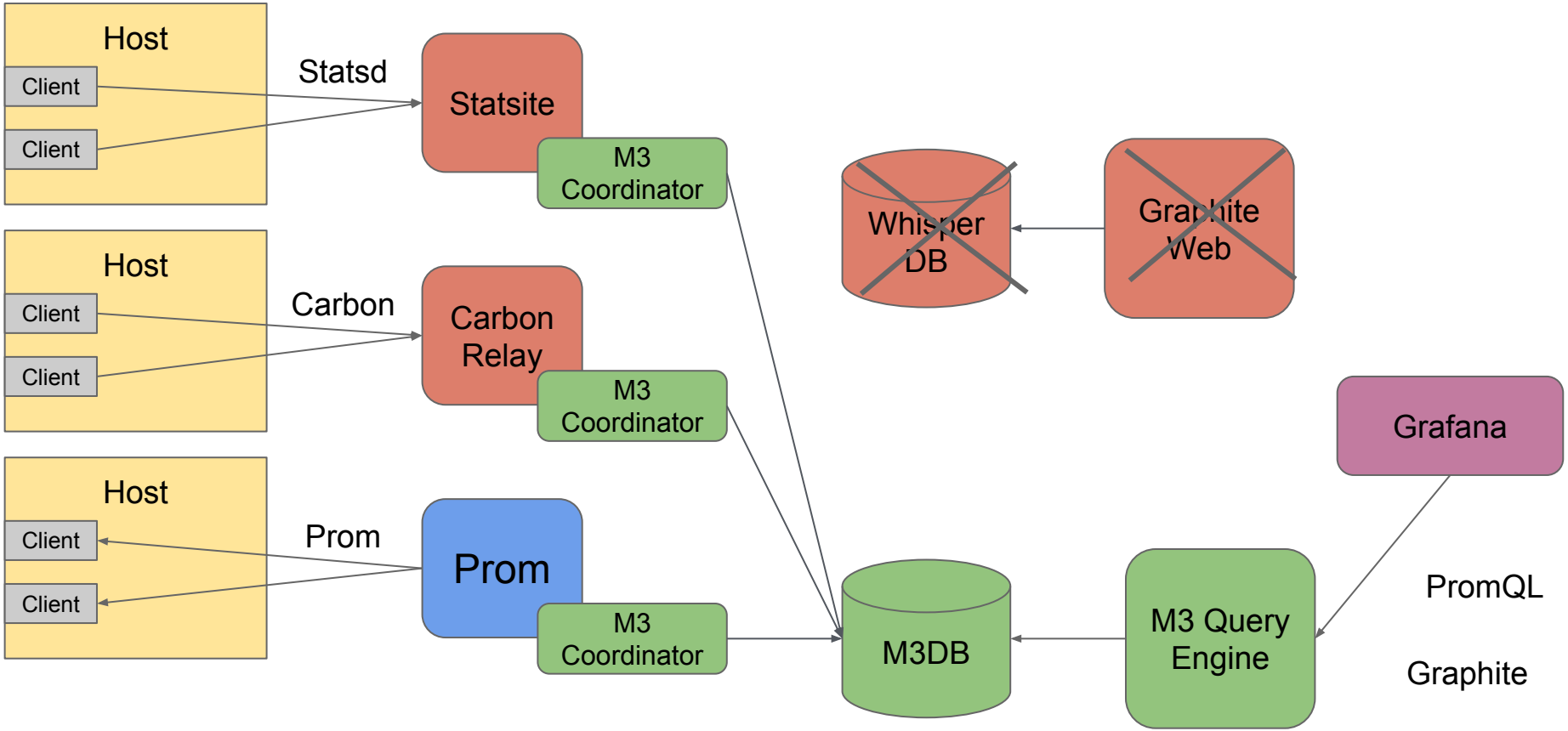
Powerful but also *kinda* easy to use

- M3DB can be deployed on premise without any dependencies.
- M3DB also can run on Kubernetes and the M3DB k8s operator can manage your cluster.
    - See more at https://github.com/m3db/m3db-operator
- Just requires two roles M3DB and M3 Coordinator to get started.
- Clustered version open sourced and can scale to billions of time series.
- Read more at https://eng.uber.com/m3

Graphite, Statsd, Carbon?



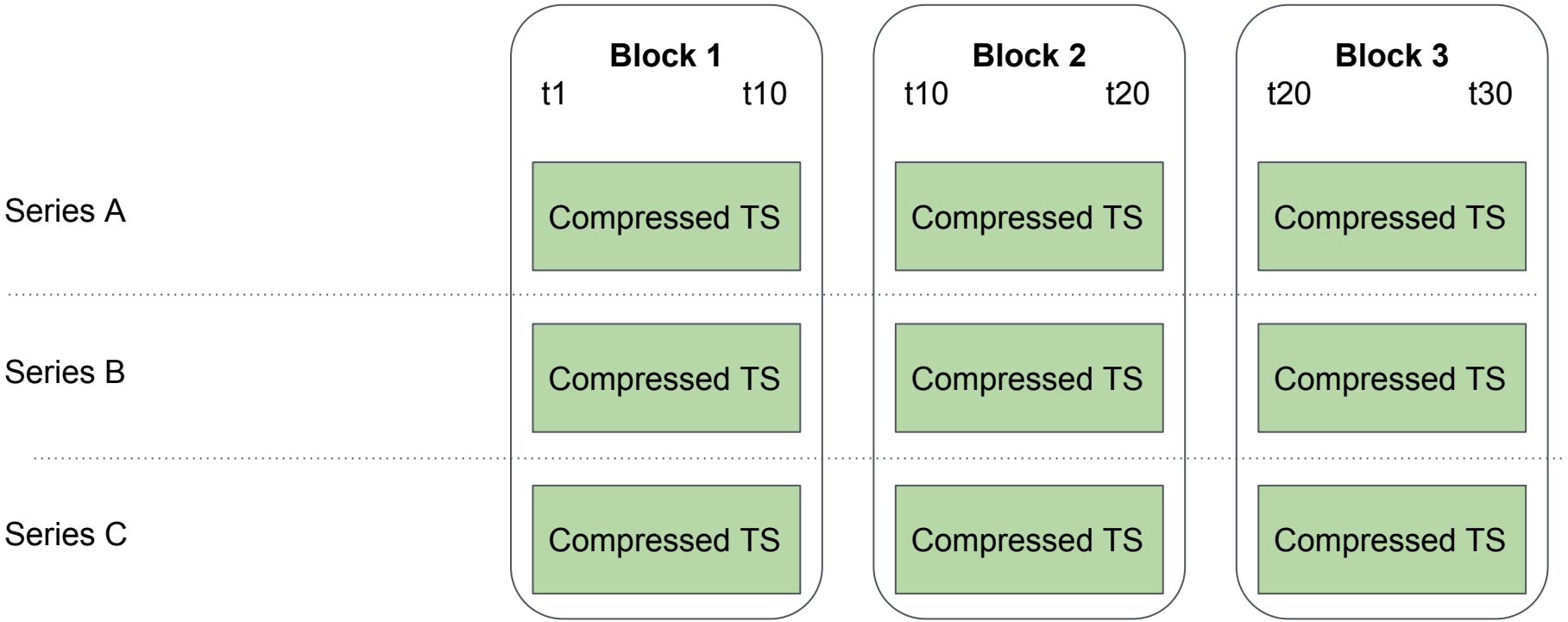
Graphite, Statsd, Carbon?



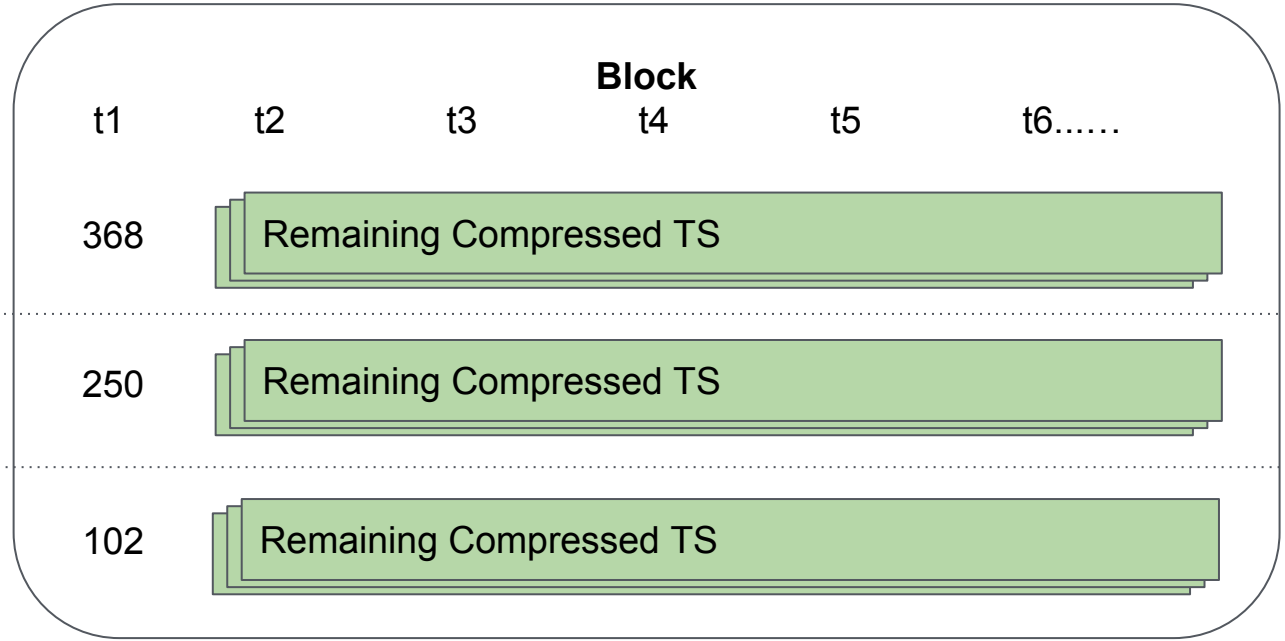
M3 Query Engine

- Multi-language with support for PromQL and Graphite
- Written in Go and can sustain much higher QPS than graphite-web
- More here https://eng.uber.com/billion-data-point-challenge

Parallelized Compressed Blocks



JIT Decompression + Lazy Function Evaluation



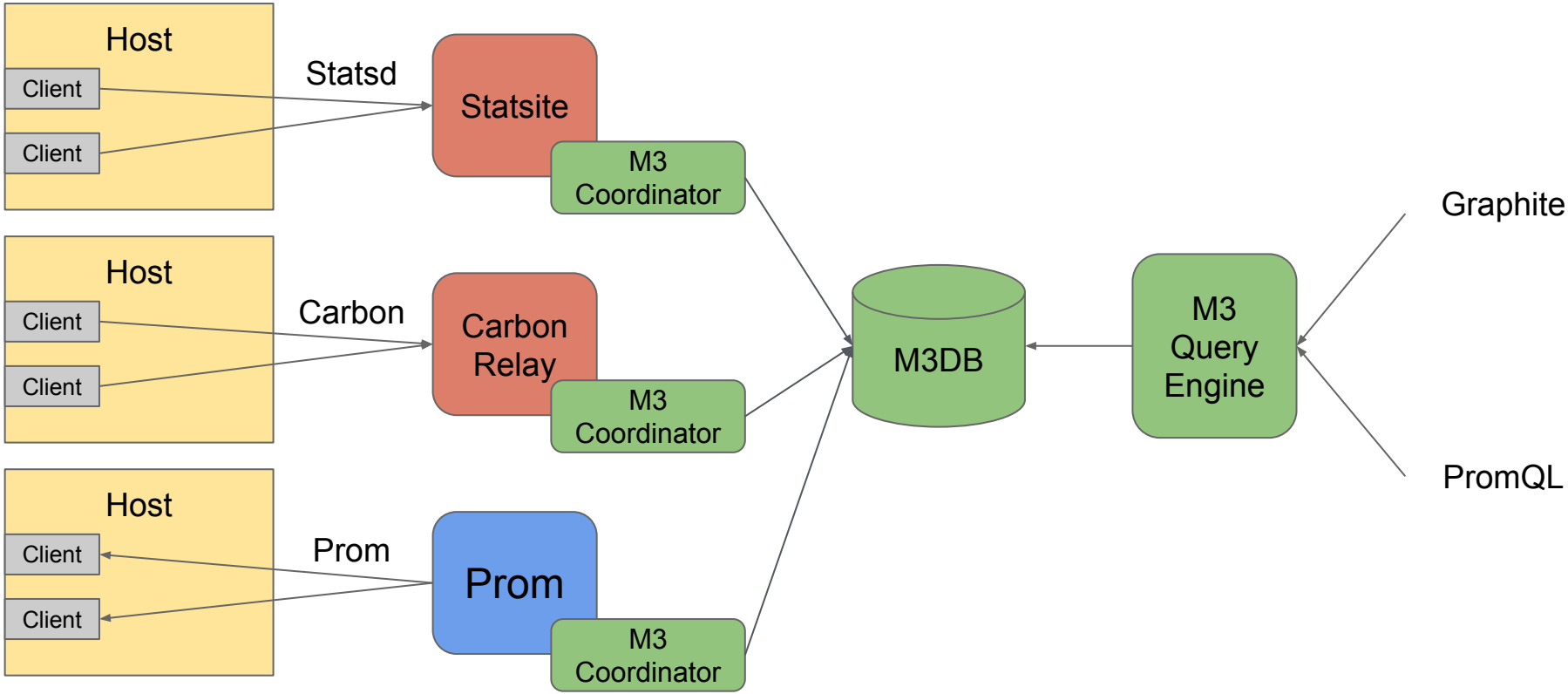
Result:

72 = (368 + 250 + 102) / 10 - scaleToSeconds function lazily evaluated after sum even though it precedes sum in the query.

Demo

- How to configure m3coordinator for ingesting Prometheus and Graphite metrics.
- Adding m3coordinator as a Prometheus and Graphite data source.
- Sending metrics and graphing them in Grafana.
- The setup for the demo is a gist on GitHub at http://bit.ly/m3fosdem.

Today


Host
Client
Client
Statsd
Statsite
M3 Coordinator
Host
Client
Client
Carbon
Carbon Relay
M3 Coordinator
Host
Client
Client
Prom
Prom
M3 Coordinator
M3DB
M3 Query Engine
Graphite
PromQL

Roadmap

H1 2019

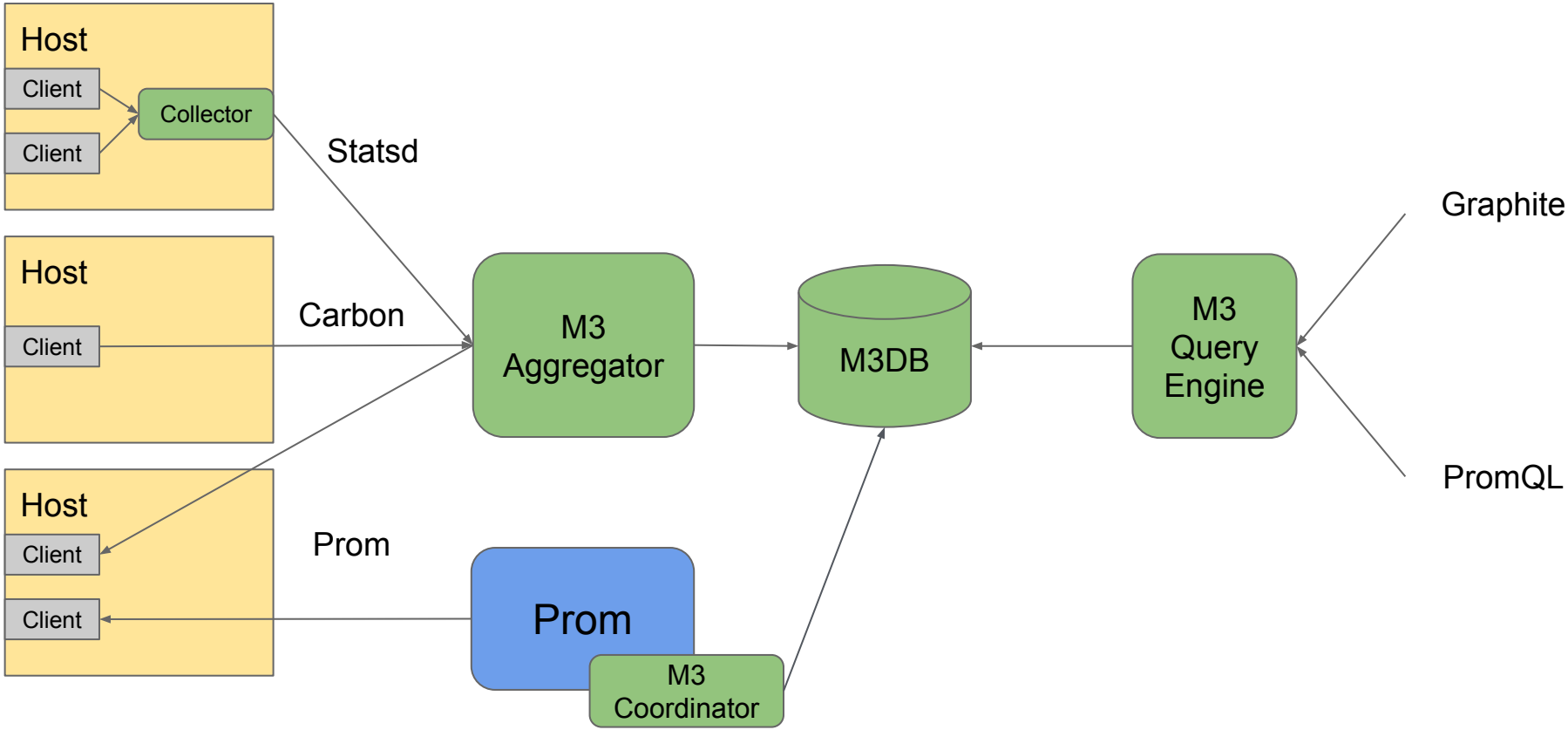
1. Better guides, simpler to configure and use out of the box.
2. Horizontally scalable metrics collection with OpenMetrics scraping from M3Collector and possibly M3Aggregator.
3. Horizontally Carbon and Statsd aggregation with M3Aggregator instead of manual HA and sharding configuration of Graphite metrics with M3Coordinator.

H2 2019

Contribute and help discuss:

- Mail (m3db@googlegroups.com)
- Gitter (gitter.im/m3db)
- GitHub issues (github.com/m3db/proposal/issues)

Future


Host
Client
Client
Collector
Statsd
Host
Client
Carbon
Host
Client
Client
Prom
M3
Aggregator
M3DB
M3
Query
Engine
Graphite
PromQL
Prom
M3
Coordinator

Questions?

GitHub and Web
https://github.com/m3db/m3, https://m3db.io

Mailing List
https://groups.google.com/forum/#!topic/m3db

Gitter (like IRC, except.. it's not IRC)
https://gitter.im/m3db

M3 Eng Blog Post
https://eng.uber.com/m3

(we're hiring)

Thank you

Questions: email ospo@uber.com

Follow our Facebook page:
www.facebook.com/uberopensource