Graphite@Scale:
How to store millions metrics per second

Booking.com

Vladimir Smirnov
System Administrator

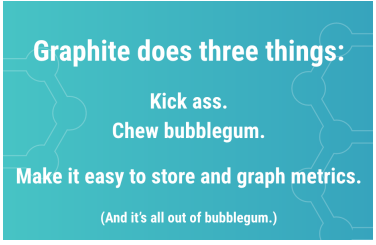
SREcon17 Asia/Australia
22 May 2017

Why you might need to store your metrics?

Most common cases:

- Capacity planning
- Troubleshooting and Postmortems
- Visualization of business data
- And more...

Graphite and its modular architecture



From the graphiteapp.org

- Allows to store time-series data
- Easy to use — text protocol and HTTP API
- You can create any data flow you want
- Modular — you can replace any part of it

Our current setup

- **Many Millions** unique metrics per second
- **Dozens Gbps** traffic on the backend
- **Hundreds** Storage servers in multiple DCs
- **Hundreds** TB of data in total
- **Dozens** of Frontend Servers
- **Hundreds** RPS on Frontend
- **Thousands** of Individual Metric Requests per second
- **Millions** points fetched from storage every second.

Our current setup

- **Many Millions** unique metrics per second
- **Dozens Gbps** traffic on the backend
- **Hundreds** Storage servers in multiple DCs
- **Hundreds** TB of data in total
- **Dozens** of Frontend Servers
- **Hundreds** RPS on Frontend
- **Thousands** of Individual Metric Requests per second
- **Millions** points fetched from storage every second.

Our current setup

- **Many Millions** unique metrics per second
- **Dozens Gbps** traffic on the backend
- **Hundreds** Storage servers in multiple DCs
- **Hundreds** TB of data in total
- **Dozens** of Frontend Servers
- **Hundreds** RPS on Frontend
- **Thousands** of Individual Metric Requests per second
- **Millions** points fetched from storage every second.

Our current setup

- **Many Millions** unique metrics per second
- **Dozens Gbps** traffic on the backend
- **Hundreds** Storage servers in multiple DCs
- **Hundreds** TB of data in total
- **Dozens** of Frontend Servers
- **Hundreds** RPS on Frontend
- **Thousands** of Individual Metric Requests per second
- **Millions** points fetched from storage every second.

Our current setup

- **Many Millions** unique metrics per second
- **Dozens Gbps** traffic on the backend
- **Hundreds** Storage servers in multiple DCs
- **Hundreds** TB of data in total
- **Dozens** of Frontend Servers
- **Hundreds** RPS on Frontend
- **Thousands** of Individual Metric Requests per second
- **Millions** points fetched from storage every second.

Our current setup

- **Many Millions** unique metrics per second
- **Dozens Gbps** traffic on the backend
- **Hundreds** Storage servers in multiple DCs
- **Hundreds** TB of data in total
- **Dozens** of Frontend Servers
- **Hundreds** RPS on Frontend
- **Thousands** of Individual Metric Requests per second
- **Millions** points fetched from storage every second.

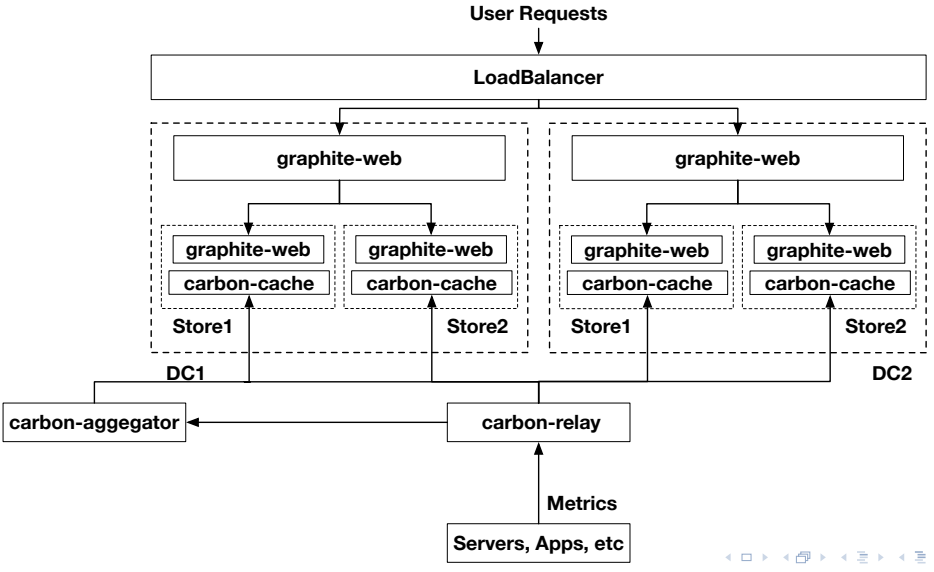
Our current setup

- **Many Millions** unique metrics per second
- **Dozens Gbps** traffic on the backend
- **Hundreds** Storage servers in multiple DCs
- **Hundreds** TB of data in total
- **Dozens** of Frontend Servers
- **Hundreds** RPS on Frontend
- **Thousands** of Individual Metric Requests per second
- **Millions** points fetched from storage every second.

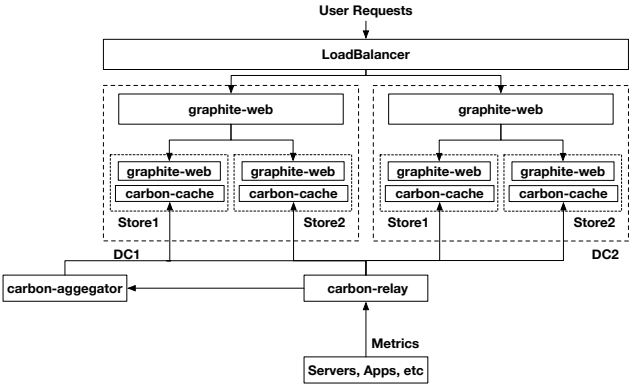
Our current setup

- **Many Millions** unique metrics per second
- **Dozens Gbps** traffic on the backend
- **Hundreds** Storage servers in multiple DCs
- **Hundreds** TB of data in total
- **Dozens** of Frontend Servers
- **Hundreds** RPS on Frontend
- **Thousands** of Individual Metric Requests per second
- **Millions** points fetched from storage every second.

Original stack



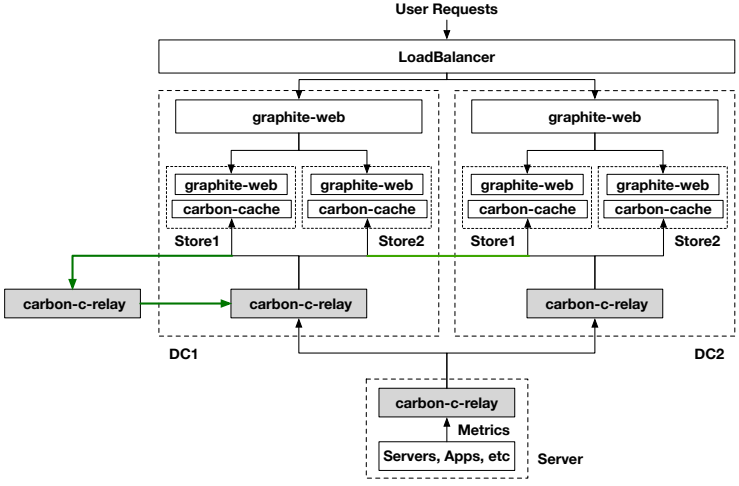
Breaking graphite: our problems at scale



What's wrong with this schema?

- carbon-relay — SPOF
- Doesn't scale well
- Stores may have different data after failures
- Render time increases with more store servers

Replacing carbon-relay



Replacing carbon-relay

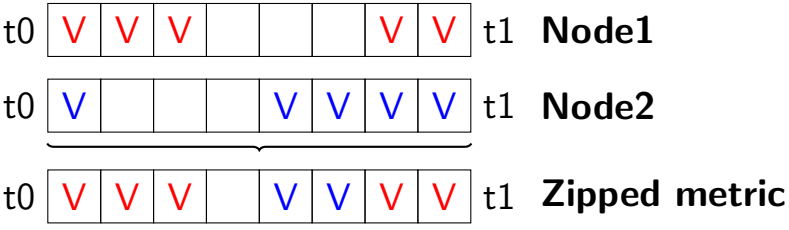
carbon-c-relay:

- Written in **C**
- Routes **1M** data points per second using only **2** cores
- L7 LB for graphite line protocol (RR with sticking)
- Can do aggregations
- Buffers the data if upstream is unavailable

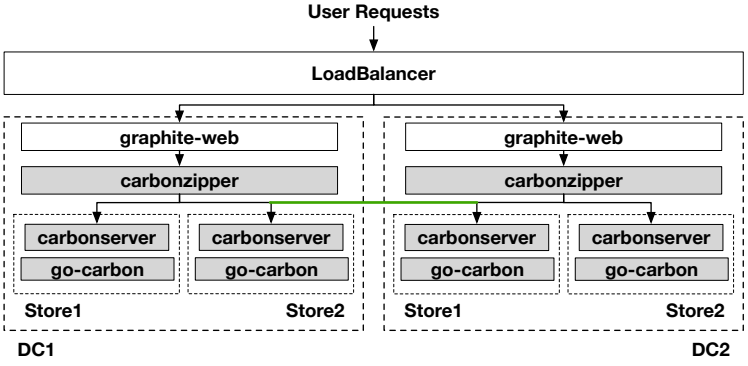
Zipper stack: Solution

Query: target=sys.server.cpu.user

Result:



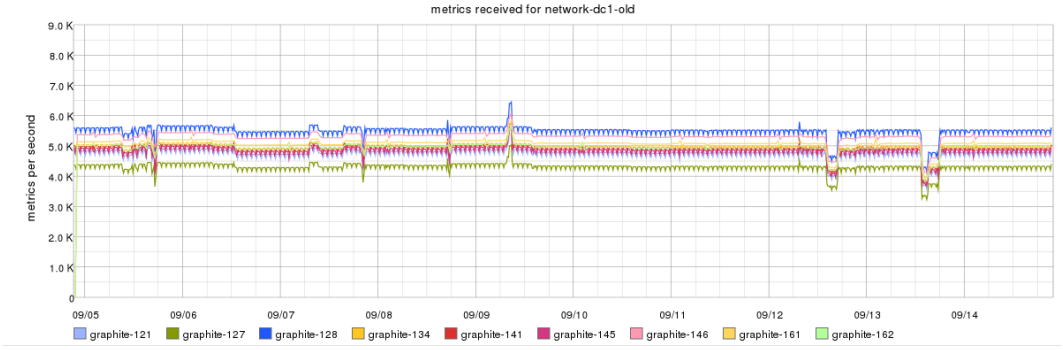
Zipper stack: architecture



Zipper stack: results

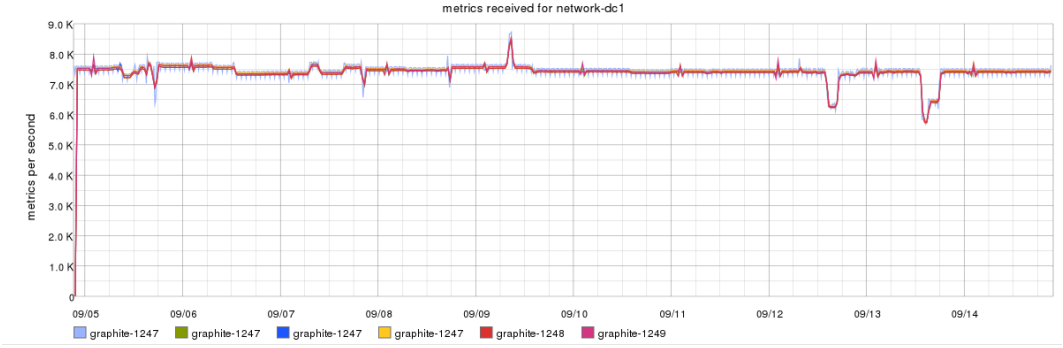
- Written in **Go**
- Can query store servers in **parallel**
- Can "Zip" the data
- carbonzipper $\Leftrightarrow$ carbonserver — **2700** RPS
  graphite-web $\Leftrightarrow$ carbon-cache — **80** RPS.
- carbonserver is now part of go-carbon (since December 2016)

Metric distribution: how it works



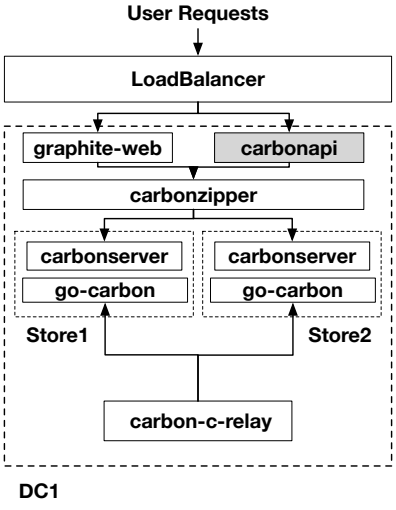
Up to **20%** difference in worst case

Metric distribution: jump hash

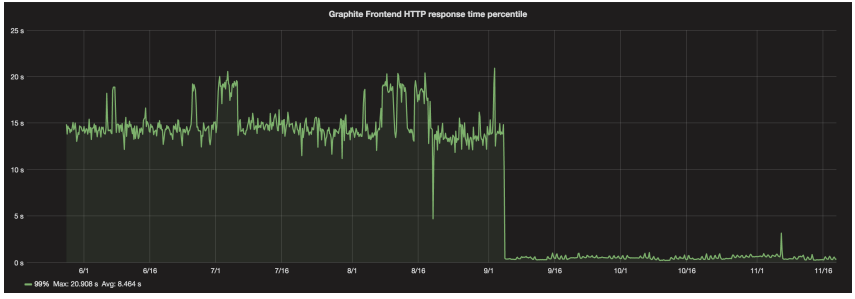


arxiv.org/pdf/1406.2294v1.pdf

Rewriting Frontend in Go: carbonapi

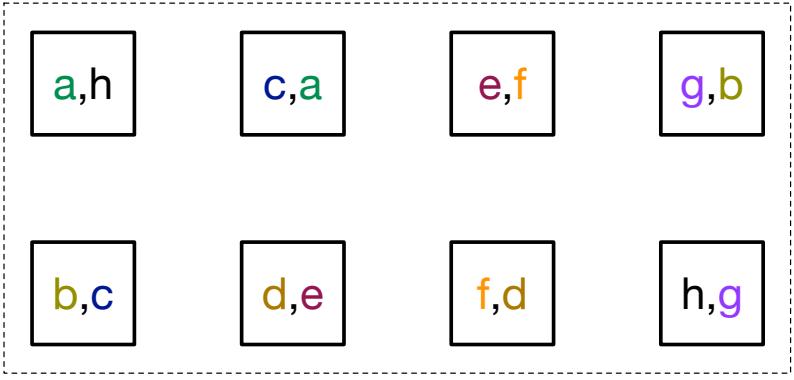


Rewriting Frontend in Go: result



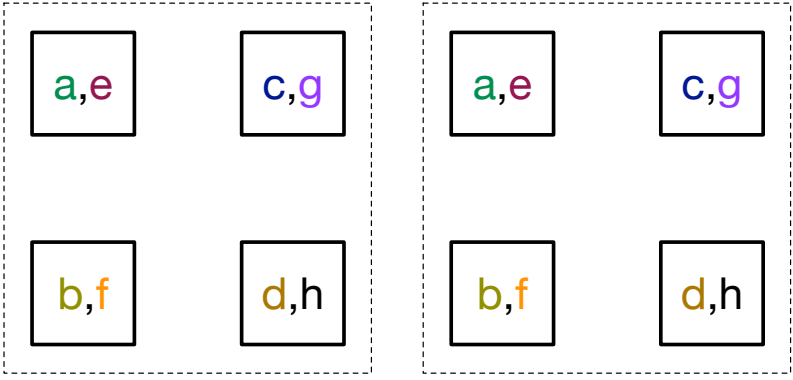
- Significantly reduced response time for users (**15s** $\Rightarrow$ **0.8s**)
- Allowes more complex queries because it's faster
- Easier to implement new heavy math functions
- Also available as Go library

Replication techniques and their pros and cons



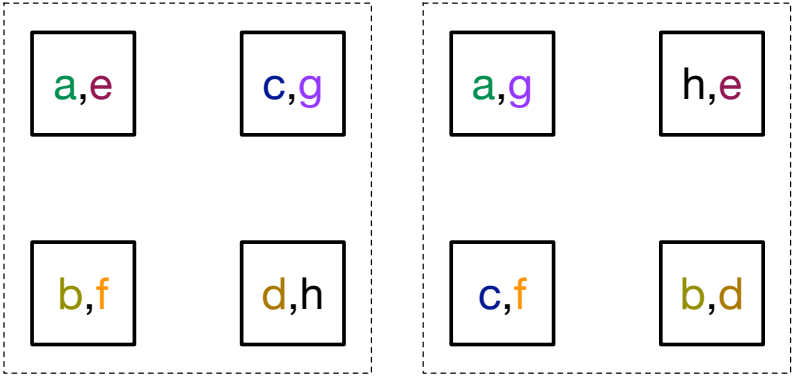
Replication Factor 2

Replication techniques and their pros and cons



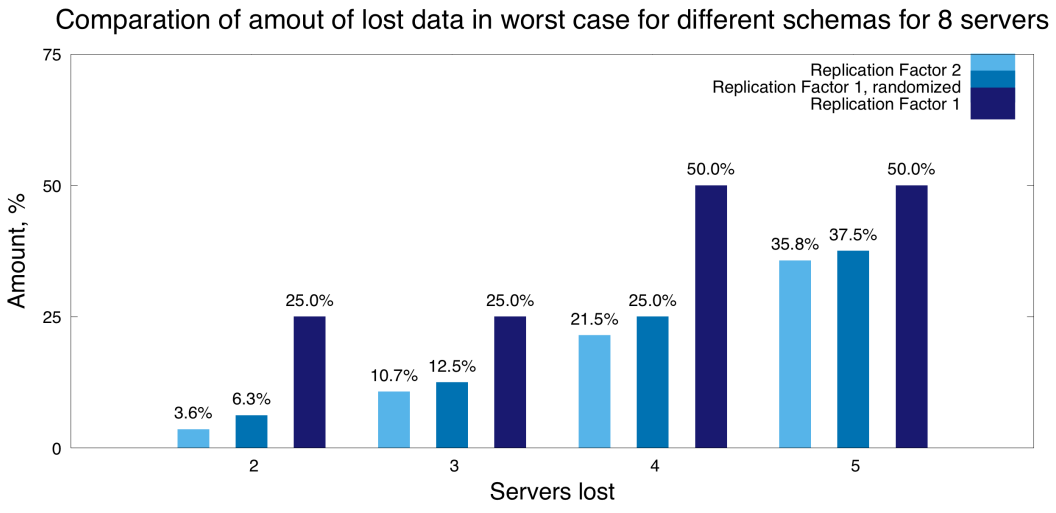
Replication Factor 1

Replication techniques and their pros and cons

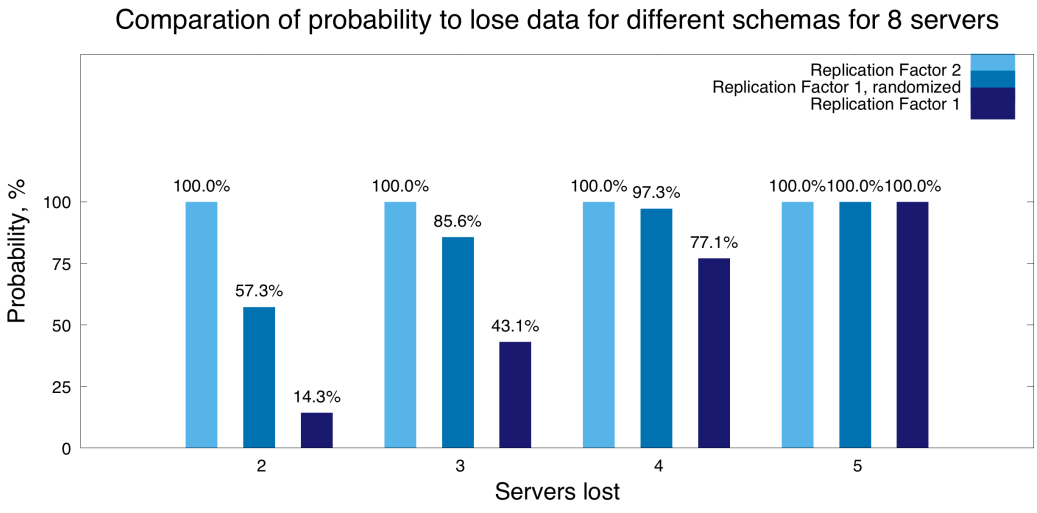


Replication Factor 1, randomized

Replication techniques and their pros and cons



Replication techniques and their pros and cons

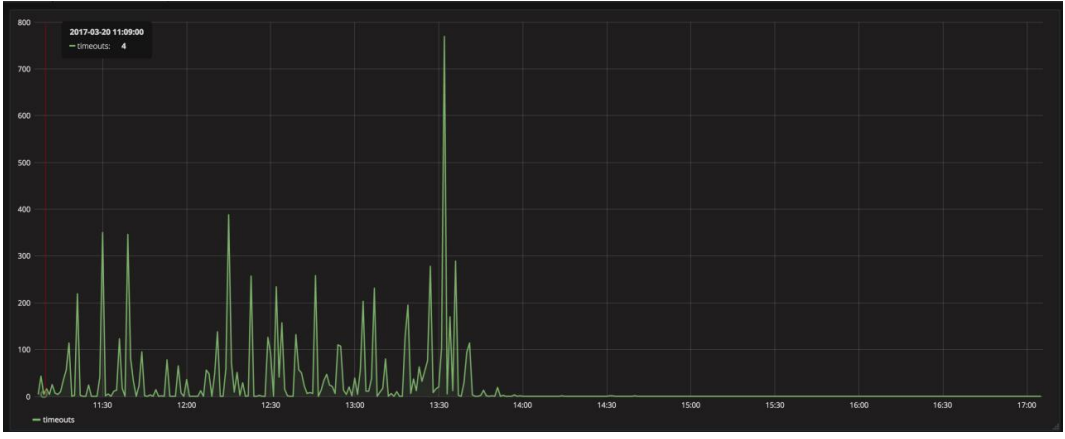


Adding simple tags

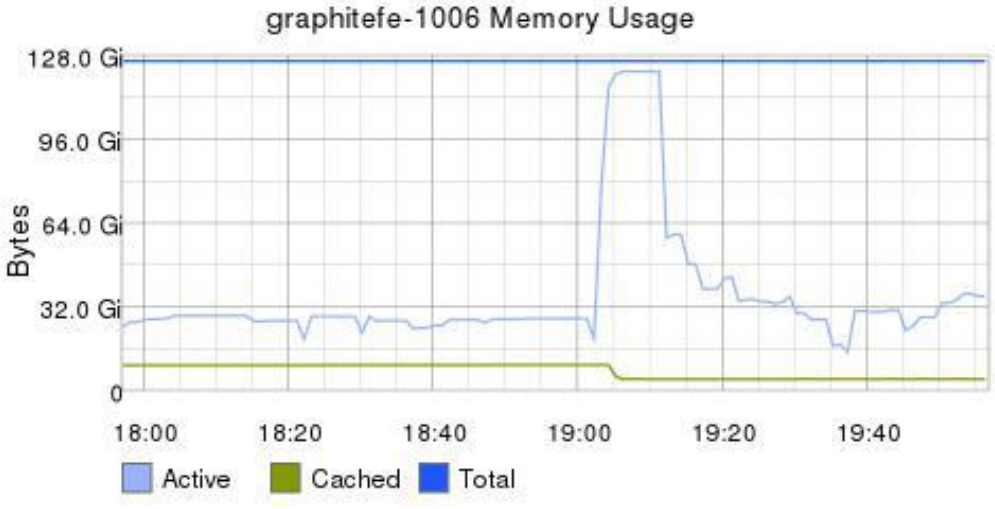
Example: target=sum(virt.v1.*.dc:datacenter1.status:live.role:graphiteStore.text-match:metricsReceived)

- Separate tags stream and storage
- No history (yet)
- No negative match support (yet)
- Only "and" syntax

Instrumenting Graphite



Instrumenting: The Problem



Instrumenting: The Problem


```
{
        "logger": "access",
        "handler": "render",
        "carbonapi_uuid": "fc5deca7-c9b2-4697-8bac-8b70c7bcc697",
        "username": "some_user",
        "targets":
          "sumSeries(exclude(transformNull(summarize(
              monitors.ticket.*.set_topic.*, '5min', 'sum', false)<...>",
        "from_raw": "-1d",
        "until_raw": "now",
        "runtime": 53.421950525,
        "zipper_requests": 16531,
        "zipper_response_size_bytes": 6531945795,
        "carbonapi_response_size_bytes": 9534
}
```

Instrumenting: know your system

- Query ran every 1 minute.
- It takes 53 seconds to Render the graph.
- It fetches **1.4 Bil** points...
- ...for each of **4** copies we have...
- ...which takes **6.5 GB** of RAM...
- ...to create **9 KB** JSON output.

Instrumenting: know your system

- Query ran every 1 minute.
- It takes 53 seconds to Render the graph.
- It fetches **1.4 Bil** points...
- ...for each of **4** copies we have...
- ...which takes **6.5 GB** of RAM...
- ...to create **9 KB** JSON output.

Instrumenting: know your system

- Query ran every 1 minute.
- It takes 53 seconds to Render the graph.
- It fetches **1.4 Bil** points...
- ...for each of **4** copies we have...
- ...which takes **6.5 GB** of RAM...
- ...to create **9 KB** JSON output.

Instrumenting: know your system

- Query ran every 1 minute.
- It takes 53 seconds to Render the graph.
- It fetches **1.4 Bil** points...
- ...for each of **4** copies we have...
- ...which takes **6.5 GB** of RAM...
- ...to create **9 KB** JSON output.

Instrumenting: know your system

- Query ran every 1 minute.
- It takes 53 seconds to Render the graph.
- It fetches **1.4 Bil** points...
- ...for each of **4** copies we have...
- ...which takes **6.5 GB** of RAM...
- ...to create **9 KB** JSON output.

Instrumenting: know your system

- Query ran every 1 minute.
- It takes 53 seconds to Render the graph.
- It fetches **1.4 Bil** points...
- ...for each of **4** copies we have...
- ...which takes **6.5 GB** of RAM...
- ...to create **9 KB** JSON output.

Instrumenting: Conclustion

- Do your logs right.
- Know your users better.
- Educate your users.
- Think of quotas.

Instrumenting: Conclustion

- Do your logs right.
- Know your users better.
- Educate your users.
- Think of quotas.

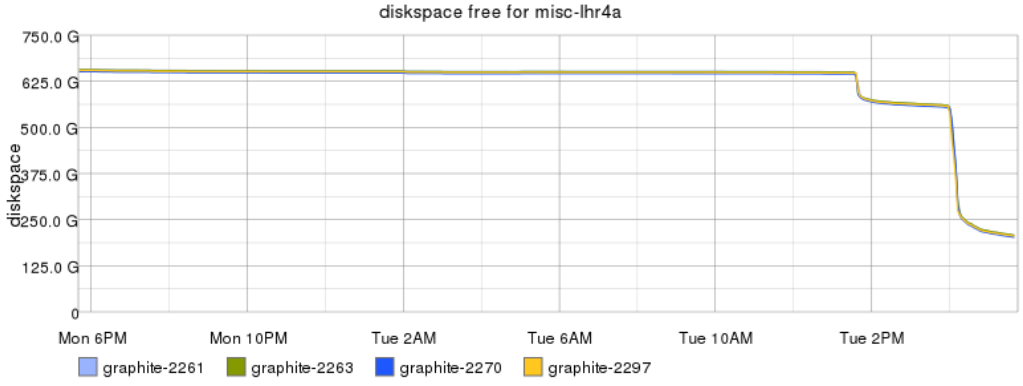
Instrumenting: Conclustion

- Do your logs right.
- Know your users better.
- Educate your users.
- Think of quotas.

Instrumenting: Conclustion

- Do your logs right.
- Know your users better.
- Educate your users.
- Think of quotas.

Instrumenting: Disk space (miss?)usage

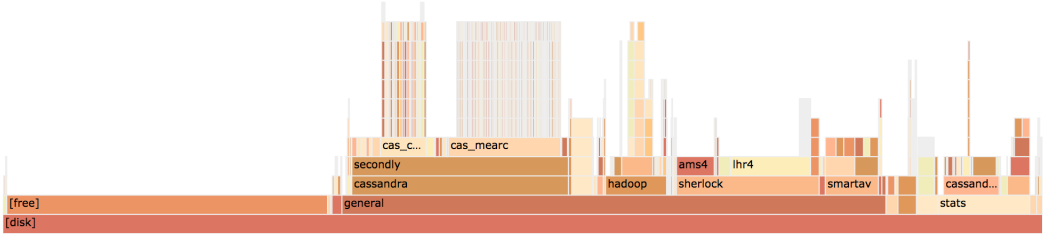


Instrumenting: Traditional way

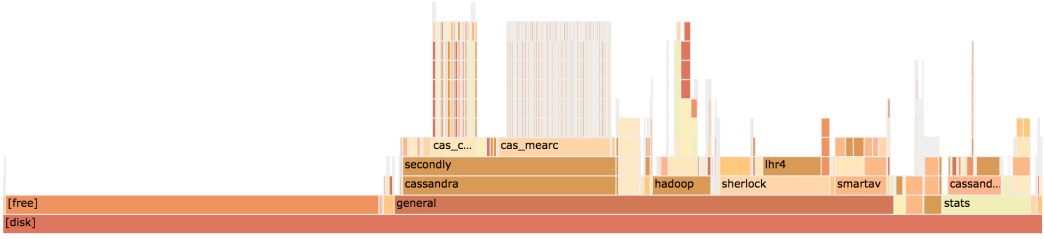
```
ssh backend
cd /var/db/whisper
du -csh *
# look at the output and remember what was there before
cd some_directory
du -csh *
# repeat as many times as you need
```

That doesn't scale!

Instrumenting: FlameGraphs: Before



Instrumenting: FlameGraphs: After



Instrumenting: Conclusion

- Collect and Store information about every metric
- Database: Clickhouse
- Stores raw data about each metric: name, size, mtime, access time, etc.

Instrumenting: Profiling stack



What's next?

- Metadata-based search (in progress)
- Find a replacement for Whisper (in progress)
- Rethink aggregators
- Replace graphite line protocol between components
- Migrate to streaming protocol between backends.
- Implement differential flamegraphs
- Continue to work on collecting traces

It's all Open Source!

- carbonapi — github.com/go-graphite/carbonapi
- carbonzipper — github.com/go-graphite/carbonzipper
- carbonsearch — github.com/kanatohodets/carbonsearch
- flamegraphs — github.com/Civil/ch-flamegraphs
- carbon-c-relay — github.com/grobian/carbon-c-relay
- go-carbon — github.com/lomik/go-carbon
- carbonmem — github.com/go-graphite/carbonmem
- replication factor test — github.com/Civil/graphite-rf-test

Questions?

vladimir.smirnov@booking.com

Thanks!

We are hiring SREs in Amsterdam!

