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

Graphite does three things:
Kick ass.
Chew bubblegum.
Make it easy to store and graph metrics.
(And it’s all out of bubblegum.)

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

User Requests

LoadBalancer

graphite-web

carbon-cache

Store1

DC1

Servers, Apps, etc

carbon-relay

Metrics

carbon-aggegator

graphite-web
carbon-cache

Store2

DC2

graphite-web
carbon-cache

Store1

graphite-web
carbon-cache

Store2
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

User Requests

LoadBalancer

graphite-web

graphite-web
carbon-cache
carbon-cache

Store1

Store2

DC1

DC2

carbon-c-relay
carbon-c-relay
carbon-c-relay

carbon-c-relay

carbon-c-relay

Metrics

User Requests

Servers, Apps, etc

Server
Replacing carbon-relay

carbon-c-relay:

- Written in C
- Routes $1\text{M}$ 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:

```
<table>
<thead>
<tr>
<th>t0</th>
<th>V</th>
<th>V</th>
<th>V</th>
<th></th>
<th>V</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>Node1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>t0</th>
<th>V</th>
<th>V</th>
<th>V</th>
<th>V</th>
<th>V</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>Node2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>t0</th>
<th>V</th>
<th>V</th>
<th>V</th>
<th>V</th>
<th>V</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>Zipped metric</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Zipper stack: architecture

User Requests

LoadBalancer

graphite-web

carbonzipper

carbonserver

go-carbon

Store1

DC1

graphite-web

carbonzipper

carbonserver

go-carbon

Store2

DC2
Zipper stack: results

- Written in **Go**
- Can query store servers in **parallel**
- Can ”Zip” the data
- carbonzipper ⇔ carbonserver — **2700** RPS
  graphite-web ⇔ 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

metrics received for network-dc1

Rewriting Frontend in Go: result

- Significantly reduced response time for users (15s ⇒ 0.8s)
- Allows 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

- a, e
- c, g
- b, f
- d, h
Replication techniques and their pros and cons

Replication Factor 1, randomized

- (a, e)
- (b, f)
- (c, g)
- (d, h)
- (a, g)
- (c, f)
- (b, d)
- (h, e)
Replication techniques and their pros and cons

Comparison of amount of lost data in worst case for different schemas for 8 servers
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: The Problem

![graphitefe-1006 Memory Usage](image)

- **Bytes**
  - 128.0 Gi
  - 96.0 Gi
  - 64.0 Gi
  - 32.0 Gi
  - 0

- **Active**
- **Cached**
- **Total**

- **Time:**
  - 18:00 to 19:40
{ "logger": "access", "handler": "render", "carbonapi_uuid": "fc5deca7-c9b2-4697-8bac-8b70c7b0c697", "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 }
- 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.
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.
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.
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.
Query ran every 1 minute.
It takes 53 seconds to Render the graph.
It fetches $1.4 \text{ Bil}$ points...
...for each of 4 copies we have...
...which takes $6.5 \text{ GB}$ of RAM...
...to create $9 \text{ KB}$ JSON output.
Instrumenting: Conclusion

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

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

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

- Do your logs right.
- Know your users better.
- Educate your users.
- Think of quotas.
Instrumenting: Disk space (miss?) usage
Instrumenting: Traditional way

```bash
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

Application: go-carbon

Application Traces
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!