Graphite@Scale:
How to store millions of metrics per second

Booking.com

Vladimir Smirnov
System Administrator

GrafanaCon EU 2018
1 March 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

- Allows to store time-series data
- Easy to use — text protocol and HTTP API
  ```bash
  echo "metric.name 1.234 $(date +%s)" | nc host 2003
  ```
- Modular — you can replace any part of it

From the graphiteapp.org

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

https://host/render?target=aliasByNode(carbon.*.metricsReceived,1)
Our current setup

- **$O(100)$** Storage servers in multiple DCs
- **$O(10)$** of Frontend Servers
- **$O(100)$** TB of data in total
- **$O(100 \, M)$** unique metrics
- **$O(10 \, M)$** unique points per second
- **$O(10 \, k)$** RPS on Frontend
- **$O(10 \, k)$** of Individual Metric Requests per second
- **$O(10 \, M)$** points fetched from storage every second.
Our current setup

- **O(100)** Storage servers in multiple DCs
- **O(10)** of Frontend Servers
  - **O(100)** TB of data in total
  - **O(100 M)** unique metrics
  - **O(10 M)** unique points per second
  - **O(10 k)** RPS on Frontend
  - **O(10 k)** of Individual Metric Requests per second
  - **O(10 M)** points fetched from storage every second.
Our current setup

- \(O(100)\) Storage servers in multiple DCs
- \(O(10)\) of Frontend Servers
- \(O(100)\) TB of data in total
  - \(O(100\ M)\) unique metrics
  - \(O(10\ M)\) unique points per second
  - \(O(10\ k)\) RPS on Frontend
  - \(O(10\ k)\) of Individual Metric Requests per second
  - \(O(10\ M)\) points fetched from storage every second.
Our current setup

- $O(100)$ Storage servers in multiple DCs
- $O(10)$ of Frontend Servers
- $O(100)$ TB of data in total
- $O(100 \text{ M})$ unique metrics
- $O(10 \text{ M})$ unique points per second
- $O(10 \text{ k})$ RPS on Frontend
- $O(10 \text{ k})$ of Individual Metric Requests per second
- $O(10 \text{ M})$ points fetched from storage every second.
Our current setup

- $O(100)$ Storage servers in multiple DCs
- $O(10)$ of Frontend Servers
- $O(100)$ TB of data in total
- $O(100 \text{ M})$ unique metrics
- $O(10 \text{ M})$ unique points per second
- $O(10 \text{ k})$ RPS on Frontend
- $O(10 \text{ k})$ of Individual Metric Requests per second
- $O(10 \text{ M})$ points fetched from storage every second.
Our current setup

- $O(100)$ Storage servers in multiple DCs
- $O(10)$ of Frontend Servers
- $O(100)$ TB of data in total
- $O(100\ M)$ unique metrics
- $O(10\ M)$ unique points per second
- $O(10\ k)$ RPS on Frontend
- $O(10\ k)$ of Individual Metric Requests per second
- $O(10\ M)$ points fetched from storage every second.
Our current setup

- $O(100)$ Storage servers in multiple DCs
- $O(10)$ of Frontend Servers
- $O(100)$ TB of data in total
- $O(100 \text{ M})$ unique metrics
- $O(10 \text{ M})$ unique points per second
- $O(10 \text{ k})$ RPS on Frontend
- $O(10 \text{ k})$ of Individual Metric Requests per second
- $O(10 \text{ M})$ points fetched from storage every second.
Our current setup

- $O(100)$ Storage servers in multiple DCs
- $O(10)$ of Frontend Servers
- $O(100)$ TB of data in total
- $O(100 \text{ M})$ unique metrics
- $O(10 \text{ M})$ unique points per second
- $O(10 \text{ k})$ RPS on Frontend
- $O(10 \text{ k})$ of Individual Metric Requests per second
- $O(10 \text{ M})$ points fetched from storage every second.
Problems: Scalability

User Requests

LoadBalancer

graphite-web

graphite-web

carbon-cache

Store1

DC1

Servers, Apps, etc

carbon-relay

Metrics

User Requests

graphite-web

graphite-web

carbon-cache

Store2

DC2

carbon-aggregator

carbon-relay

SPOF

Doesn’t scale well

Servers, Apps, etc

Carbon-aggregator

Carbon-relay

Doesn’t scale well
Problems: Consistency

Need resync after failures

Doesn’t scale well

SPOF

Doesn’t scale well

User Requests

LoadBalancer

graphite-web

carbon-cache

Store1

DC1

graphite-web

carbon-cache

Store2

DC2

carbon-aggegator

carbon-relay

Servers, Apps, etc

Metrics
Problems: Render time

- Need resync after failures
- Slow renders
- Doesn't scale well
- SPOF
- Need resync after failures
- Slow renders
- Doesn't scale well
- SPOF
Replacing carbon-relay

User Requests

LoadBalancer

graphite-web

graphite-web

carbon-cache

Store1

DC1

carbon-c-relay

carbon-c-relay

graphite-web

graphite-web

carbon-cache

Store2

DC2

carbon-c-relay

carbon-c-relay

graphite-web

graphite-web

carbon-cache

Store1

carbon-c-relay

carbon-c-relay

graphite-web

graphite-web

carbon-cache

Store2

Server

Metrics

Servers, Apps, etc
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
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></td>
<td></td>
<td></td>
<td>Node1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>t0</th>
<th>V</th>
<th></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>
<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>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Zipper stack: architecture

User Requests

LoadBalancer

graphite-web

carbonzipper

carbonserver
go-carbon

Store1
DC1

User Requests

LoadBalancer

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

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
- Parsing and functions are available as separate libraries.
Replication techniques and their pros and cons

Replication Factor 2

\begin{align*}
&\{a,h\} & &\{c,a\} & &\{e,f\} & &\{g,b\} \\
&\{b,c\} & &\{d,e\} & &\{f,d\} & &\{h,g\}
\end{align*}
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, h,e
- c,f, b,d
Replication techniques and their pros and cons

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

Bars represent the amount of lost data for different numbers of servers lost, with bars for Replication Factor 1, randomized (light blue) and Replication Factor 2 (dark blue). The amount of lost data increases as the number of servers lost increases, with the highest amount of lost data being 50.0% for both Replication Factor 1 and Replication Factor 2 when all servers are lost.
Replication techniques and their pros and cons

Comparison of probability to lose data for different schemas for 8 servers

<table>
<thead>
<tr>
<th>Servers lost</th>
<th>Replication Factor 2</th>
<th>Replication Factor 1, randomized</th>
<th>Replication Factor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>100.0%</td>
<td>100.0%</td>
<td>100.0%</td>
</tr>
<tr>
<td>3</td>
<td>57.3%</td>
<td>85.6%</td>
<td>43.1%</td>
</tr>
<tr>
<td>4</td>
<td>100.0%</td>
<td>97.3%</td>
<td>77.1%</td>
</tr>
<tr>
<td>5</td>
<td>100.0% 100.0% 100.0%</td>
<td>100.0%</td>
<td>100.0%</td>
</tr>
</tbody>
</table>
Adding simple tags

Example:

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

- Separated tags stream and storage
- No history
- No negative match support (yet)
- Only "and" syntax
What's next?

- Find a replacement for Whisper (in progress)
- Replace graphite line protocol between components (in progress)
- Migrate to streaming protocol between backends (in progress)
- Implement differential flamegraphs
- Continue to work on collecting traces
It’s all Open Source!

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

Several major users: Booking.com, eBay Classifieds Group and Slack
Questions?

vladimir.smirnov@booking.com

civil.over@gmail.com

Twitter: @Civilus

Facebook: civilus

Telegram: Civiloid

LinkedIn: vladsmirnov
Thanks!

We are hiring SREs in Amsterdam!
https://workingatbooking.com
Bonus: Instrumenting: FlameGraphs: Before
Bonus: Instrumenting: FlameGraphs: After
Bonus: Instrumenting: Conclusion

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