Scaling Graphite at Criteo

FOSDEM 2018 - “Not yet another talk about Prometheus”
Me

Corentin Chary

Twitter: @iksaif
Mail: c.chary@criteo.com

- Working on Graphite with the Observability team at Criteo
- Worked on Bigtable/Colossus at Google
BigGraphite

Storing time series in Cassandra and querying them from Graphite
Graphite does two things:

- Store numeric time-series data
- Render graphs of this data on demand

- [https://graphiteapp.org/](https://graphiteapp.org/)
- [https://github.com/graphite-project](https://github.com/graphite-project)
graphite-web

- Django Web application

- UI to browse metrics, display graph, build dashboard
  - Mostly deprecated by Grafana

- API to list metrics and fetch points (and generate graphs)
  - /metrics/find?query=my.metrics.*
  - /render/?target=sum(my.metrics.*)&from=-10m
Carbon

- Receives metrics and relay them
  - carbon-relay: receives the metrics from the clients and relay them
  - carbon-aggregator: ‘aggregates’ metrics based on rules

- Persist metrics to disk
  - carbon-cache: writes points to the storage layer
  - Default database: whisper, one file = one metric

```
host123.cpu0.user <timestamp> 100  carbon  disk
```
Our usage

- 6 datacenters, 20k servers, 10k+ “applications”
- We ingest and query >80M metrics
  - Read: ~20K metrics/s
  - Write: ~800K points/s
- 2000+ dashboards, 1000+ alerts (evaluated every 5min)
architecture overview

Applications

<table>
<thead>
<tr>
<th>Applications</th>
<th>carbon-relay (dc local)</th>
<th>carbon-relay</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>in-memory</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(UDP)</td>
<td>(TCP)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>grafana</th>
<th>graphite API + UI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>carbon-cache</td>
</tr>
<tr>
<td></td>
<td>persisted</td>
</tr>
</tbody>
</table>
architecture overview (r=2)
current tools are improvable

- Graphite lacks true elasticity
  - ... for storage and QPS
  - One file per metric is wasteful even with sparse files
- Graphite’s clustering is naïve (slight better with 1.1.0)
  - Graphite-web clustering very fragile
  - Single query can bring down all the cluster
  - Huge fan-out of queries
- Tooling
  - Whisper manipulation tools are brittle
  - Storage ‘repair’/’reconciliation’ is slow and inefficient (multiple days)
  - Scaling up is **hard** and error prone
solved problems?

- Distributed database systems have already solved these problems
  - e.g. Cassandra, Riak, ... 

- Fault tolerance through replication
  - Quorum queries and read-repair ensure consistency 

- Elasticity through consistent hashing
  - Replacing and adding nodes is fast and non-disruptive 
  - Repairs are transparent, and usually fast 
  - Almost-linear scalability
BigGraphite
decisions, decisions

- **OpenTSDB (HBase)**
  - Too many moving parts
  - Only one HBase cluster at Criteo, hard/costly to spawn new ones

- **Cyanite (Cassandra/ES)**
  - Originally depended on ES, manually ensures cross-database consistency...
  - Doesn’t behave exactly like Carbon/Graphite

- **KairosDB (Cassandra/ES)**
  - Dependency on ES for Graphite compatibility
  - Relies on outdated libraries

- [insert hyped/favorite time-series DBs]

- Safest and easiest: build a Graphite plugin using only Cassandra
Target architecture overview

- grafana
- graphite API + UI
- Cassandra
- carbon-relay
- carbon-cache
- metrics
plug’n’play

- Graphite has support for plug-ins since v1.0

**carbon** ([carbon.py](https://example.com/carbon.py))

- create(metric)
- exists(metric)
- update(metric, points)

update(uptime.nodeA, [now(), 42])

**Graphite-Web** ([graphite.py](https://example.com/graphite.py))

- find(glob)
- fetch(metric, start, stop)

find(uptime.*) -> [uptime.nodeA]

fetch(uptime.nodeA, now()-60, now())
storing time series in Cassandra

● Store points...
  ○ (metric, timestamp, value)
  ○ Multiple resolutions and TTLs (60s:8d, 1h:30d, 1d:1y)
  ○ Write => points
  ○ Read => series of points (usually to display a graph)

● ...and metadata!
  ○ Metrics hierarchy (like a filesystem, directories and metrics — like whisper)
  ○ Metric, resolution, [owner, ...]
  ○ Write => new metrics
  ○ Read => list of metrics from globs (my.metric.*.foo.*)
<Cassandra>
storing data in Cassandra

- Sparse matrix storage
  - Map< Row, Map< Column, Value >>

- Row ⇔ Partition
  - Atomic storage unit (nodes hold full rows)
  - Data distributed according to Hash(rowKey)

- Column Key
  - Atomic data unit inside a given partition
  - Unique per partition
  - Has a value
  - Stored in lexicographical order (supports range queries)

store( row, col, val )

\[
\begin{align*}
H &= \text{hash( row )} \\
\text{Node} &= \text{get_owner( H )} \\
\text{send( Node, (row, col, val) )}
\end{align*}
\]
naïve schema

CREATE TABLE points (  
    path    text,    -- Metric name  
    time    bigint,  -- Value timestamp  
    value   double,  -- Point value  
    PRIMARY KEY ((path), time)  
) WITH CLUSTERING ORDER BY (time DESC);

- Boom! Timeseries.
  - (Boom! Your cluster explodes when you have many points on each metric.)
    (Boom! You spend your time compacting data and evicting expired points)
CREATE TABLE IF NOT EXISTS %(table)s (  
metric uuid,  -- Metric UUID (actual name stored as metadata)  
time_start_ms bigint,  -- Lower time bound for this row  
offset smallint,  -- time_start_ms + offset * precision = timestamp  
value double,  -- Value for the point.  
count int,  -- If value is sum, divide by count to get the avg  
) WITH CLUSTERING ORDER BY (offset DESC) 
AND default_time_to_live = %(default_time_to_live)d

- table = datapoints_<resolution>
- default_time_to_live = <resolution.duration>
- (Number of points per partition limited to ~25K)
cqlsh> select * from biggraphite.datapoints_2880p_60s limit 5;

<table>
<thead>
<tr>
<th>metric</th>
<th>time_start_ms</th>
<th>offset</th>
<th>count</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>7dfa0696-2d52-5d35-9cc9-114f5dccc1e4</td>
<td>1475040000000</td>
<td>1999</td>
<td>1</td>
<td>2019</td>
</tr>
<tr>
<td>7dfa0696-2d52-5d35-9cc9-114f5dccc1e4</td>
<td>1475040000000</td>
<td>1998</td>
<td>1</td>
<td>2035</td>
</tr>
<tr>
<td>7dfa0696-2d52-5d35-9cc9-114f5dccc1e4</td>
<td>1475040000000</td>
<td>1997</td>
<td>1</td>
<td>2031</td>
</tr>
<tr>
<td>7dfa0696-2d52-5d35-9cc9-114f5dccc1e4</td>
<td>1475040000000</td>
<td>1996</td>
<td>1</td>
<td>2028</td>
</tr>
<tr>
<td>7dfa0696-2d52-5d35-9cc9-114f5dccc1e4</td>
<td>1475040000000</td>
<td>1995</td>
<td>1</td>
<td>2028</td>
</tr>
</tbody>
</table>

(5 rows)
Finding nemo

- How to find metrics matching fish.*.nemo*?
- Most people use ElasticSearch, and that’s fine, but we wanted a self-contained solution
we’re feeling SASI

- Recent Cassandra builds have secondary index facilities
  - SASI (SSTable-Attached Secondary Indexes)
- Split metric path into components
  - criteo.cassandra.uptime $\Rightarrow$ part0=criteo, part1=cassandra, part2=uptime, part3=$end$
  - criteo.* $\Rightarrow$ part0=criteo, part2=$end$
- Associate metric UUID to the metric name’s components
- Add secondary indexes to path components
- Retrieve metrics matching a pattern by querying the secondary indexes

See design.md and SASI Indexes for details
do you even query?

- a.single.metric (equals)
  - Query: part0=a, part1=single, part2=metric, part3=$end$
  - Result: a.single.metric

- a.few.metrics.* (wildcards)
  - Query: part0=a, part1=few, part2=metrics, part4=$end$
  - Result: a.few.metrics.a, a.few.metrics.b, a.few.metrics.c

- match{ed_by,ing}.s[ao]me.regexp.?{0-9} (almost regexp, post-filtering)
  - ^match(ed_by|ing).s[ao]me\..{0-9}$
  - Query: similar to wildcards
  - Result: matched_by.same.regexp.b2, matched_by.some.regexp.a1, matching.same.regexp.w5, matching.some.regexp.z8
</Cassandra>
And then?
BIGGEST GRAPHITE CLUSTER IN THE MULTIVERSE! (or not)

- 800 TiB of capacity, 20 TiB in use currently (R=2)
- Writes: 1M QPS
- Reads: 10K QPS
- 24 bytes per point, 16 bytes with compression
  - But probably even better with double-delta encoding!
- 20 Cassandra nodes
- 6 Graphite Web, 10 Carbon Relays, 10 Carbon Cache
- x3! 2 Replicated Datacenters, one isolated to canary changes.
links of (potential) interest

- Github project: github.com/criteo/biggraphite
- Cassandra’s Design Doc: CASSANDRA_DESIGN.md
- Announcement: BigGraphite-Announcement

You can just `pip install biggraphite` and voilà !
Roadmap?

- Add Prometheus Read/Write support (BigGraphite as long term storage for Prometheus).
  - Already kind of works with https://github.com/criteo/graphite-remote-adapter
- Optimize writes: bottleneck on Cassandra is CPU, we could divide CPU usage by ~10 with proper batching
- Optimize reads: better parallelization, better long term caching (points usually don’t change in the past)
More Slides
Monitoring your monitoring!

- Graphite-Web availability (% of time on a moving window)
- Graphite-Web performances (number of seconds to sum 500 metrics, at the 95pctl)
- Point loss: sending 100 points per dc per seconds, checking how many arrive in less than 2 minutes
- Point delay: setting the timestamp as the value, and diffing with now (~2-3 minutes)
Read Availability (10min) 100.0%
Read Latency (90pctl - 15min) 3.77 s
Point Loss 0%
Ingestion Delay 1 min

Checkout the Graphite SLO for more details. Also look at HTTP Probe Dashboard for more HTTP metrics.
Aggregation

Downsampling/Rollups/Resolutions/Retentions/...
- Retention policy: 60s:8d, 1h:30d, 1d:1y (combination of stages)
- Stage: 60s:8d (resolution 60 seconds for 8d)
- Aggregator functions: sum, min, max, avg
- stage[N] = aggregator(stage[N-1])
- Obviously this doesn’t work for average
  - $\text{avg}(\text{avg}(1, 2), \text{avg}(3, 4)) \neq \text{avg}(1, 2, 3, 4)$
  - We have to store both ‘value’ and ‘count’!
  - (See downsampling.py for details)

- Cheaper / simpler to do directly on the write path
  - Checkpointed every 5 minutes
  - But since processes can restart!
  - We use unique write ids (and replica ids when using replicated clusters)
  - (See design.md)
- We **aggregate** in the carbon plugins, **before** writing the points
- Points for different level of resolution go to different tables for efficient compactions and TTL expiration
- Reads go to a specific table depending on the time window

What about aggregation?