Bernd Bausch

Follow 17 Followers About

How I Learned to Stop Worrying and Love Gnocchi aggregation

Bernd Bausch Aug 8, 2019 · 8 min read

Sorry for all you lovers of Italian cuisine. This is not a food blog. It's about Gnocchi, the time series database as a service. Gnocchi is software that grew out of OpenStack, the largest open-source cloud operating system, but is an independent project now. If none of this is your passion, you have an opportunity to stop reading now.

I recently tried to get the CPU utilization of OpenStack instances out of Gnocchi. Unfortunately, the latest OpenStack release, *Stein*, doesn't create CPU utilization samples anymore, and doesn't provide much (if any) documentation how to deal with this change. Gnocchi's documentation is not always easy to understand either, and so I thought I'd share my experience wrestling with the tool.

A special thankyou goes to Gordon Chung, one of Gnocchi's maintainers. Without his generous help, I would not have been able to come to terms with all this.

If you want to follow along, I recommend setting up an OpenStack cloud that contains Gnocchi. A convenient method is Devstack. Before you run the tests, set the *interval* key in */etc/ceilometer/polling.yaml* to 60, so that the cloud generates samples every minute. Restart the Ceilometer services (in Devstack: `systemctl restart devstack@ceilometer*`)and you are ready.

What is Gnocchi

Gnocchi's purpose is to receive huge amounts of datapoints, preprocess them, store them, and give you, the user, access to the data. A datapoint is nothing more than a timestamp and a value; it has no meaning by itself, but the Gnocchi database can contain additional information that determines its meaning.

Imagine you run a public cloud with several customers and need to bill those customers according to their resource usage: So many virtual servers running for so many hours, that many terabytes of storage used for so many weeks, etc. The cloud has a facility that generates raw measurement data (OpenStack calls this facility *Ceilometer*). Gnocchi's task is to efficiently process, store and provide access to that data. Your billing system then bases its invoices on the data it finds in Gnocchi's database.

Another use of Gnocchi is the automatic scaling of cloud applications. An application might determine that it can't keep up with the demand and needs more virtual servers to run smoothly. The data to determine this, such as CPU utilization or disk I/O rates, comes out of Gnocchi.

Two initial concepts: Resources and metrics

To understand what follows, you need to have a grasp of a few key concepts.

When Gnocchi receives measurements, it correlates them to **resources** and **metrics**. A resource describes the object from which measurements are sampled. In an OpenStack cloud, a resource might be an instance, an instance's disk, a network, a storage volume, etc. Assuming an instance, you can measure a number of things: The CPU time, the number of disk I/Os, the memory usage and much more. These things are named **metrics**.

A Gnocchi metric consists of a name, such as *cpu,* a unit, such as nanoseconds, and a resource ID: It belongs to a specific resource. A metric is also linked to a so-called archive policy, which is covered below.

After its installation, Gnocchi contains no resources or metrics. You have to enter them before you can enter measurements. In an OpenStack cloud, this is done automatically by Ceilometer.

Preprocessing and storing: Archive policies

Gnocchi doesn't store the raw measurements it receives. In order to reduce the data volume, it stores values at certain time intervals, for example every five or ten minutes. The duration of a storage interval is called **granularity**.

The value Gnocchi stores is an **aggregate**, for example the average value of samples during the storage interval. In fact, Gnocchi can store several aggregates. For example, it might store the average, the highest value in the interval, the number of values and the sum of values. Granularity and aggregation methods are defined in a so-called **archive policy**. Here is an example:

```
$ gnocchi archive-policy show medium
+---------------------+--------------------------------------------+
| Field               | Value                                      |
+---------------------+--------------------------------------------+
| aggregation_methods | std, count, min, max, sum, mean            |
| back_window         | 0                                          |
| definition          | - points: 10080, granularity: 0:01:00,    |
|                     | timespan: 7 days, 0:00:00                  |
|                     | - points: 8760, granularity: 1:00:00,     |
|                     | timespan: 365 days, 0:00:00                |
| name                | medium                                     |
+---------------------+--------------------------------------------+
```

The *medium* (no pun intended) archive policy stores data every minute (granularity 0:01:00) and every hour (1:00:00). It keeps 8760 hourly datapoints, which means they are kept for a year, and 10080 per-minute datapoints (one week). At each interval, it stores several aggregates: The standard deviation across the interval (aggregation method *std*), the number of measurements during the interval (*count*), their minimum, maximum, sum and average (*mean*).

How everything works together: Resources, metrics, archive policies and measurements

How does this fit together? Let's first list the metrics on an OpenStack cloud. Note that I had to drastically change the output format to make it fit Medium's page width.

```
$ gnocchi metric list
+--------------+----------+--------------------+------+-------------+
| id           | a/p      | name               | unit | resource_id |
+--------------+----------+--------------------+------+-------------+
| 03bb6e58-... | medium   | disk.device.usage  | B    | 2816573...  |
| 64183bf3-... | medium   | cpu                | ns   | 6f87dd4...  |
...
```

For example, there is a metric named *cpu*, measured in nanoseconds, using the *medium* archive policy and linked to a specific resource. What might these resources be?

```
$ gnocchi resource show 6f87dd4d-ec8b-4665-9a50-...
+-----------------------+-------------------------------------------+
| Field                 | Value                                     |
+-----------------------+-------------------------------------------+
| created_at            | 2019-08-04T00:57:10+00:00                 |
| created_by_project_id | bfd99505135846adb70ddce361400127          |
| created_by_user_id    | d17347c250ac461e86c47a6dfd03aef8          |
| deleted_at            | None                                      |
| display_name          | cpu-user2                                 |
| ended_at              | None                                      |
| metrics               | cpu: 64183bf3-6a92-4770-9864-...          |
|                       | disk.root.size: 22fcdad5-532d-4bae-...    |
|                       | memory.resident: 13e66c31-492c-45fa-...   |
                          ....(more metrics)...
| original_resource_id  | 6f87dd4d-ec8b-4665-9a50-15f60ae6ef4a      |
| project_id            | 74bd69d304544f9991fddf5c9dabb83d          |
| started_at            | 2019-08-04T00:57:23.685769+00:00          |
| type                  | instance                                  |
| user_id               | cb1e831588a5495fbe60ad159e899c12          |
+-----------------------+-------------------------------------------+
```

Again, the output is shortened and reformatted. This is a resource of type *instance.* Note that the command lists the metrics that link back to this instance.

To summarize sample processing: Whenever Ceilometer adds samples to Gnocchi, it links them to a metric. As you saw, the metric has a name, a unit, and is related to an archive policy and a resource. Gnocchi processes samples using the aggregation methods defined in the metric's archive policy and stores the results.

Getting data out of Gnocchi

To apply these concepts, launch a CPU-intensive instance and list its CPU measurements after a few minutes.

```
$ gnocchi measures show --resource-id 23fb84af-f9f3-4062-9a0f-
28cbedaeabb1 cpu
+---------------------------+-------------+-------------+
| timestamp                 | granularity |       value |
+---------------------------+-------------+-------------+
| 2019-08-04T12:00:00+09:00 |      3600.0 |  3.1741e+12 |
| 2019-08-04T12:05:00+09:00 |        60.0 | 3.15271e+12 |
| 2019-08-04T12:06:00+09:00 |        60.0 | 3.15283e+12 |
...
```

By default, this command lists the result of the *mean* aggregation. If you want *max* aggregation:

```
$ gnocchi measures show --resource-id 23fb84af-f9f3-4062-9a0f-
28cbedaeabb1 cpu --aggregation max
+---------------------------+-------------+-------------+
| timestamp                 | granularity |       value |
+---------------------------+-------------+-------------+
| 2019-08-04T12:00:00+09:00 |      3600.0 | 3.21439e+12 |
| 2019-08-04T12:05:00+09:00 |        60.0 | 3.15271e+12 |
| 2019-08-04T12:06:00+09:00 |        60.0 | 3.15283e+12 |
...
```

What do the values mean? Ceilometer measures the accumulated CPU usage of the instance, measured in nanoseconds, and sends it to Gnocchi as a measurement of the *cpu* metric. In the example, at 12:05, this instance had accumulated 3,152,710,000,000 nanoseconds of CPU time. A minute later, it was 3,152,830,000,000 nanoseconds. These figures are based on the average CPU usage of each sampling interval, which happens to be 60 seconds long. This is nice, but to determine the instance's CPU load, the CPU utilization in percent would be handy.

Aggregating data on the fly

Gnocchi stores aggregates such as *mean* and *max* values. You can post-process data when you retrieve it, by applying further aggregation. To do this, use the `gnocchi aggregates` command. For a start, generate the same data as above:

```
$ gnocchi aggregates '(metric cpu mean)' id=23fb84af-f9f3-4062-9a0f-
28cbedaeabb1
+----------------+---------------------+-------------+-------------+
| name           | timestamp           | granularity |       value |
+----------------+---------------------+-------------+-------------+
| 23.../cpu/mean | 2019-08-04T03:00... |      3600.0 |  3.1741e+12 |
| 23.../cpu/mean | 2019-08-04T03:05... |        60.0 | 3.15271e+12 |
| 23.../cpu/mean | 2019-08-04T03:06... |        60.0 | 3.15283e+12 |
| 23.../cpu/mean | 2019-08-04T03:07... |        60.0 | 3.16644e+12 |
| 23.../cpu/mean | 2019-08-04T03:08... |        60.0 | 3.18413e+12 |
| 23.../cpu/mean | 2019-08-04T03:09... |        60.0 | 3.21439e+12 |
...
```

Sharp eyes may detect that the timestamp of this command uses UTC, whereas the earlier command uses the local time zone. This is an inconsistency we have to live with.

Now, what you really want is not the accumulated CPU time, but the *change* at each interval. In other words, how much CPU time is consumed *per interval*. You can obtain this figure by **re-aggregating the data on the fly** using the *rate of change* aggregation.

```
$ gnocchi aggregates '(aggregate rate:mean (metric cpu mean))'
id=23fb84af-f9f3-4062-9a0f-28cbedaeabb1
+------------+-----------------------+-------------+---------------+
| name       | timestamp             | granularity |         value |
+------------+-----------------------+-------------+---------------+
| aggregated | 2019-08-04T03:06:00.. |        60.0 |   120000000.0 |
| aggregated | 2019-08-04T03:07:00.. |        60.0 | 13610000000.0 |
| aggregated | 2019-08-04T03:08:00.. |        60.0 | 17690000000.0 |
| aggregated | 2019-08-04T03:09:00.. |        60.0 | 30260000000.0 |
+------------+-----------------------+-------------+---------------+
```

rate aggregates are based on the difference (delta) between the current and the previous datapoint. *rate:mean* computes the average of the delta values. In this particular case, there is only one delta per interval, so that *rate:mean*, *rate:max* or *rate:min* would generate the same result. Gnocchi also allows looking at groups of resources, but let's reserve that for a later blog entry.

Now we know that our instance used 120000000 nanosecond in the 03:06 interval, 13610000000 nanoseconds at 03:07 and so on. These figures are unwieldy. Can we convert them to seconds?

```
$ gnocchi aggregates '(/ (aggregate rate:mean (metric cpu mean))
1000000000.0)' id=23fb84af-f9f3-4062-9a0f-28cbedaeabb1
+------------+---------------------------+-------------+-------+
| name       | timestamp                 | granularity | value |
+------------+---------------------------+-------------+-------+
| aggregated | 2019-08-04T03:06:00+00:00 |        60.0 |  0.12 |
| aggregated | 2019-08-04T03:07:00+00:00 |        60.0 | 13.61 |
| aggregated | 2019-08-04T03:08:00+00:00 |        60.0 | 17.69 |
| aggregated | 2019-08-04T03:09:00+00:00 |        60.0 | 30.26 |
+------------+---------------------------+-------------+-------+
```

Yes we can, because the `gnocchi aggregates` command features prefix notation arithmetic! Let's go a step further and convert the seconds to percent.

```
$ gnocchi aggregates '(* (/ (aggregate rate:mean (metric cpu mean))
60000000000.0) 100)' id=23fb84af-f9f3-4062-9a0f-28cbedaeabb1
+------------+-----------------------+-------------+---------------+
| name       | timestamp                | granularity |         value |
+------------+-----------------------+-------------+---------------+
| aggregated | 2019-08-04T03:06:00.. |        60.0 |           0.2 |
| aggregated | 2019-08-04T03:07:00.. |        60.0 | 22.6833333333 |
| aggregated | 2019-08-04T03:08:00.. |        60.0 | 29.4833333333 |
| aggregated | 2019-08-04T03:09:00.. |        60.0 | 50.4333333333 |
+------------+-----------------------+-------------+---------------+
```

I will leave the analysis of the expression `(* (/ (aggregate rate:mean (metric cpu mean)) 60000000000.0) 100)` to the reader.

With this, I have my CPU utilization back, which OpenStack Stein stole from me.

Do you need a summary?

Gnocchi stores pre-processed measurements at regular intervals. A measurement is linked to a metric and a resource. The measurement's metric determines the archive policy, which (among other things) determines the algorithms that are used to pre-process measurements.

When pulling measurements out of Gnocchi, you can either look at the data as it is stored in the database, or you can post-process it. A useful post-processing algorithm is the *rate*, which we used to derive CPU utilization from the accumulated CPU usage figure that Ceilometer stores in Gnocchi.

Cloud Computing Metering