Graphite-API Documentation

Release 1.1.3

Bruno Renié

Oct 25, 2017

Contents

Graphite-API is an alternative to Graphite-web, without any built-in dashboard. Its role is solely to fetch metrics from a time-series database (whisper, cyanite, etc.) and rendering graphs or JSON data out of these time series. It is meant to be consumed by any of the numerous Graphite dashboard applications.

Graphite-API is a fork of Graphite-web and couldn't have existed without the fantastic prior work done by the Graphite team.

CHAPTER 1

Why should I use it?

Graphite-API offers a number of improvements over Graphite-web that you might find useful. Namely:

- The Graphite-API application is completely stateless and doesn't need a SQL database. It only needs to talk to a time series database.
- Python 2 and 3 are both supported.
- The HTTP API accepts JSON data additionnaly to form data and querystring parameters.
- The application is extremely simple to *install* and *configure*.
- The architecture has been drastically simplified and there are many fewer moving parts than in graphite-web:
  - No memcache integration – rendering is live.
  - No support for the Pickle format when rendering.
  - Plugin architecture for *integrating with time series databases* or *adding more analysis functions*.
- The codebase has been thoroughly updated with a focus on test coverage and code quality.

Note: Graphite-API does **not** provide any web/graphical interface. If you currently rely on the built-in Graphite composer, Graphite-API might not be for you. However, if you're using a third-party dashboard interface, Graphite-API will do just fine.

CHAPTER 2

Contents

Installation

Debian / Ubuntu: native package

If you run Debian 8 or Ubuntu 14.04 LTS, you can use one of the available packages which provides a self-contained build of graphite-api. Builds are available on the releases page.

Once installed, Graphite-api should be running as a service and available on port 8888. The package contains all the *optional dependencies*.

Python package

Prerequisites

Installing Graphite-API requires:

- Python 2 (2.6 and above) or 3 (3.3 and above), with development files. On debian/ubuntu, you'll want to install `python-dev`.
- `gcc`. On debian/ubuntu, install `build-essential`.
- Cairo, including development files. On debian/ubuntu, install the `libcairo2-dev` package.
- `libffi` with development files, `libffi-dev` on debian/ubuntu.
- Pip, the Python package manager. On debian/ubuntu, install `python-pip`.

Global installation

To install Graphite-API globally on your system, run as root:

```
$ pip install graphite-api
```

Isolated installation (virtualenv)

If you want to isolate Graphite-API from the system-wide python environment, you can install it in a virtualenv.

```
$ virtualenv /usr/share/python/graphite
$ /usr/share/python/graphite/bin/pip install graphite-api
```

Extra dependencies

When you install `graphite-api`, all the dependencies for running a Graphite server that uses Whisper as a storage backend are installed. You can specify extra dependencies:

- For Sentry integration: `pip install graphite-api[sentry]`.
- For Cyanite integration: `pip install graphite-api[cyanite]`.
- For Cache support: `pip install graphite-api[cache]`. You'll also need the driver for the type of caching you want to use (Redis, Memcache, etc.). See the Flask-Cache docs for supported cache types.

You can also combine several extra dependencies:

```
$ pip install graphite-api[sentry,cyanite]
```

Configuration

/etc/graphite-api.yaml

The configuration file for Graphite-API lives at `/etc/graphite-api.yaml` and uses the YAML format. Creating the configuration file is optional: if Graphite-API doesn't find the file, sane defaults are used. They are described below.

Default values

```
search_index: /srv/graphite/index
finders:
  - graphite_api.finders.whisper.WhisperFinder
functions:
  - graphite_api.functions.SeriesFunctions
  - graphite_api.functions.PieFunctions
whisper:
  directories:
    - /srv/graphite/whisper
time_zone: <system timezone> or UTC
```

Config sections

Default sections

search_index

The location of the search index used for searching metrics. Note that it needs to be a file that is writable by the Graphite-API process.

finders

A list of python paths to the storage finders you want to use when fetching metrics.

functions

A list of python paths to function definitions for transforming / analyzing time series data.

whisper

The configuration information for whisper. Only relevant when using WhisperFinder. Simply holds a `directories` key listing all directories containing whisper data.

time_zone

The time zone to use when generating graphs. By default, Graphite-API tries to detect your system timezone. If detection fails it falls back to UTC. You can also manually override it if you want another value than your system's timezone.

Extra sections

carbon

Configuration information for reading data from carbon's cache. Items:

hosts List of carbon-cache hosts, in the format `hostname:port[:instance]`.

timeout Socket timeout for carbon connections, in seconds.

retry_delay Time to wait before trying to re-establish a failed carbon connection, in seconds.

hashing_keyfunc Python path to a hashing function for metrics. If you use Carbon with consistent hashing and a custom function, you need to point to the same hashing function.

hashing_type Type of metric hashing function. The default `carbon_ch` is Graphite's traditional consistent-hashing implementation. Alternatively, you can use `fnv1a_ch`, which supports the Fowler-Noll-Vo hash function (FNV-1a) hash implementation offered by the carbon-c-relay project. Default: `carbon_ch`

carbon_prefix Prefix for carbon's internal metrics. When querying metrics starting with this prefix, requests are made to all carbon-cache instances instead of one instance selected by the key function. Default: `carbon`.

replication_factor The replication factor of your carbon setup. Default: `1`.

Example:

```
carbon:
  hosts:
    - 127.0.0.1:7002
  timeout: 1
  retry_delay: 15
  carbon_prefix: carbon
  replication_factor: 1
```

sentry_dsn

This is useful if you want to send Graphite-API's exceptions to a Sentry instance for easier debugging.

Example:

```
sentry_dsn: https://key:secret@app.getsentry.com/12345
```

Note: Sentry integration requires Graphite-API to be installed with the corresponding extra dependency:

```
$ pip install graphite-api[sentry]
```

allowed_origins

Allows you to do cross-domain (CORS) requests to the Graphite API. Say you have a dashboard at `dashboard.example.com` that makes AJAX requests to `graphite.example.com`, just set the value accordingly:

```
allowed_origins:
  - dashboard.example.com
```

You can specify as many origins as you want. A wildcard can be used to allow all origins:

```
allowed_origins:
  - *
```

cache

Lets you configure a cache for graph rendering. This is done via Flask-Cache which supports a number of backends including memcache, Redis, filesystem or in-memory caching.

Cache configuration maps directly to Flask-Cache's config values. For each `CACHE_*` config value, set the lowercased name in the `cache` section, without the prefix. Example:

```
cache:
  type: redis
  redis_host: localhost
```

This would configure Flask-Cache with `CACHE_TYPE = 'redis'` and `CACHE_REDIS_HOST = 'localhost'`.

Some cache options have default values defined by Graphite-API:

- `default_timeout`: 60
- `key_prefix`: `'graphite-api:`.

Note: Caching functionality requires you to install the cache extra dependency but also the underlying driver. E.g. for redis, you'll need:

```
$ pip install graphite-api[cache] redis
```

statsd

Attaches a statsd object to the application, which can be used for instrumentation. Currently Graphite-API itself doesn't use this, but some backends do, like Graphite-Influxdb.

Example:

```
statsd:
    host: 'statsd_host'
    port: 8125  # not needed if default
```

Note: This requires the statsd module:

```
$ pip install statsd
```

render_errors

If `True` (default), full tracebacks are returned in the HTTP response in case of application errors.

Custom location

If you need the Graphite-API config file to be stored in another place than `/etc/graphite-api.yaml`, you can set a custom location using the `GRAPHITE_API_CONFIG` environment variable:

```
export GRAPHITE_API_CONFIG=/var/lib/graphite/config.yaml
```

Deployment

There are several options available, depending on your setup.

Gunicorn + nginx

First, you need to install Gunicorn. The easiest way is to use `pip`:

```
$ pip install gunicorn
```

If you have installed Graphite-API in a virtualenv, install Gunicorn in the same virtualenv:

```
$ /usr/share/python/graphite/bin/pip install gunicorn
```

Next, create the script that will run Graphite-API using your process watcher of choice.

Upstart

```
description "Graphite-API server"
start on runlevel [2345]
stop on runlevel [!2345]

respawn

exec gunicorn -w2 graphite_api.app:app -b 127.0.0.1:8888
```

Supervisor

```
[program:graphite-api]
command = gunicorn -w2 graphite_api.app:app -b 127.0.0.1:8888
autostart = true
autorestart = true
```

systemd

```
# This is /etc/systemd/system/graphite-api.socket
[Unit]
Description=graphite-api socket

[Socket]
ListenStream=/run/graphite-api.sock
ListenStream=127.0.0.1:8888

[Install]
WantedBy=sockets.target
```

```
# This is /etc/systemd/system/graphite-api.service
[Unit]
Description=Graphite-API service
Requires=graphite-api.socket

[Service]
ExecStart=/usr/bin/gunicorn -w2 graphite_api.app:app
Restart=on-failure
#User=graphite
#Group=graphite
ExecReload=/bin/kill -s HUP $MAINPID
ExecStop=/bin/kill -s TERM $MAINPID
PrivateTmp=true

[Install]
WantedBy=multi-user.target
```

Note: If you have installed Graphite-API and Gunicorn in a virtualenv, you need to use the full path to Gunicorn. Instead of `gunicorn`, use `/usr/share/python/graphite/bin/gunicorn` (assuming your virtualenv is at `/usr/share/python/graphite`).

See the Gunicorn docs for configuration options and command-line flags.

Finally, configure the nginx vhost:

```
# /etc/nginx/sites-available/graphite.conf

upstream graphite {
    server 127.0.0.1:8888 fail_timeout=0;
}

server {
    server_name graph;
    listen 80 default;
    root /srv/www/graphite;

    location / {
        try_files $uri @graphite;
    }

    location @graphite {
        proxy_pass http://graphite;
    }
}
```

Enable the vhost and restart nginx:

```
$ ln -s /etc/nginx/sites-available/graphite.conf /etc/nginx/sites-enabled
$ service nginx restart
```

Apache + mod_wsgi

First, you need to install mod_wsgi.

See the mod_wsgi InstallationInstructions for installation instructions.

Then create the graphite-api.wsgi:

```
# /var/www/wsgi-scripts/graphite-api.wsgi

from graphite_api.app import app as application
```

Finally, configure the apache vhost:

```
# /etc/httpd/conf.d/graphite.conf

LoadModule wsgi_module modules/mod_wsgi.so

WSGISocketPrefix /var/run/wsgi

Listen 8013
<VirtualHost *:8013>

    WSGIDaemonProcess graphite-api processes=5 threads=5 display-name='%{GROUP}'␣
↪inactivity-timeout=120
    WSGIProcessGroup graphite-api
    WSGIApplicationGroup %{GLOBAL}
    WSGIImportScript /var/www/wsgi-scripts/graphite-api.wsgi process-group=graphite-
↪api application-group=%{GLOBAL}

    WSGIScriptAlias / /var/www/wsgi-scripts/graphite-api.wsgi

    <Directory /var/www/wsgi-scripts/>
        Order deny,allow
        Allow from all
    </Directory>
    </VirtualHost>
```

Adapt the mod_wsgi configuration to your requirements.

See the mod_wsgi QuickConfigurationGuide for an overview of configurations and mod_wsgi ConfigurationDirectives to see all configuration directives

Restart apache:

```
$ service httpd restart
```

Docker

Create a `graphite-api.yaml` configuration file with your desired config.

Create a `Dockerfile`:

```
FROM brutasse/graphite-api
```

Build your container:

```
docker build -t graphite-api .
```

Run it:

```
docker run -t -i -p 8888:8888 graphite-api
```

`/srv/graphite` is a docker `VOLUME`. You can use that to provide whisper data from the host (or from another docker container) to the graphite-api container:

```
docker run -t -i -v /path/to/graphite:/srv/graphite -p 8888:8888 graphite-api
```

This container has all the *extra packages* included. Cyanite backend and Sentry integration are available.

Nginx + uWSGI

First, you need to install uWSGI with Python support. On Debian, install `uwsgi-plugin-python`.

Then create the uWSGI file for Graphite-API in `/etc/uwsgi/apps-available/graphite-api.ini`:

```
[uwsgi]
processes = 2
socket = localhost:8080
plugins = python27
module = graphite_api.app:app
```

If you installed Graphite-API in a virtualenv, specify the virtualenv path:

```
home = /var/www/wsgi-scripts/env
```

If you need a custom location for Graphite-API's config file, set the environment variable like this:

```
env = GRAPHITE_API_CONFIG=/var/www/wsgi-scripts/config.yml
```

Enable `graphite-api.ini` and restart uWSGI:

```
$ ln -s /etc/uwsgi/apps-available/graphite-api.ini /etc/uwsgi/apps-enabled
$ service uwsgi restart
```

Finally, configure the nginx vhost:

```
# /etc/nginx/sites-available/graphite.conf

server {
    listen 80;

    location / {
        include uwsgi_params;
        uwsgi_pass localhost:8080;
    }
}
```

Enable the vhost and restart nginx:

```
$ ln -s /etc/nginx/sites-available/graphite.conf /etc/nginx/sites-enabled
$ service nginx restart
```

Other deployment methods

They currently aren't described here but there are several other ways to serve Graphite-API:

- nginx + circus + chaussette

If you feel like contributing some documentation, feel free to open pull a request on the Graphite-API repository.

HTTP API

Here is the general behavior of the API:

- When parameters are missing or wrong, an HTTP 400 response is returned with the detailed errors in the response body.
- Request parameters can be passed via:
  - JSON data in the request body (`application/json` content-type).
  - Form data in the request body (`application/www-form-urlencoded` content-type).
  - Querystring parameters.

  You can pass some parameters by querystring and others by json/form data if you want to. Parameters are looked up in the order above, meaning that if a parameter is present in both the form data and the querystring, only the one from the querystring is taken into account.
- URLs are given without a trailing slash but adding a trailing slash is fine for all API calls.
- Parameters are case-sensitive.

The Metrics API

These API endpoints are useful for finding and listing metrics available in the system.

`/metrics/find`

Finds metrics under a given path. Other alias: `/metrics`.

Example:

```
GET /metrics/find?query=collectd.*

{"metrics": [{
    "is_leaf": 0,
    "name": "db01",
    "path": "collectd.db01."
}, {
    "is_leaf": 1,
    "name": "foo",
    "path": "collectd.foo"
}]}
```

Parameters:

***query* (mandatory)** The query to search for.

format The output format to use. Can be `completer` or `treejson` (default).

***wildcards* (0 or 1)** Whether to add a wildcard result at the end or no. Default: 0.

from Epoch timestamp from which to consider metrics.

until Epoch timestamp until which to consider metrics.

***jsonp* (optional)** Wraps the response in a JSONP callback.

`/metrics/expand`

Expands the given query with matching paths.

Parameters:

***query* (mandatory)** The metrics query. Can be specified multiple times.

***groupByExpr* (0 or 1)** Whether to return a flat list of results or group them by query. Default: 0.

***leavesOnly* (0 or 1)** Whether to only return leaves or both branches and leaves. Default: 0

***jsonp* (optional)** Wraps the response in a JSONP callback.

`/metrics/index.json`

Walks the metrics tree and returns every metric found as a sorted JSON array.

Parameters:

***jsonp* (optional)** Wraps the response in a jsonp callback.

Example:

```
GET /metrics/index.json

[
    "collectd.host1.load.longterm",
    "collectd.host1.load.midterm",
    "collectd.host1.load.shortterm"
]
```

The Render API – `/render`

Graphite-API provides a `/render` endpoint for generating graphs and retrieving raw data. This endpoint accepts various arguments via query string parameters, form data or JSON data.

To verify that the api is running and able to generate images, open `http://<api-host>:<port>/render?target=test` in a browser. The api should return a simple 600x300 image with the text "No Data".

Once the api is running and you've begun feeding data into the storage backend, use the parameters below to customize your graphs and pull out raw data. For example:

```
# single server load on large graph
http://graphite/render?target=server.web1.load&height=800&width=600

# average load across web machines over last 12 hours
http://graphite/render?target=averageSeries(server.web*.load)&from=-12hours

# number of registered users over past day as raw json data
http://graphite/render?target=app.numUsers&format=json

# rate of new signups per minute
http://graphite/render?target=summarize(derivative(app.numUsers),"1min")&title=New_
↪Users_Per_Minute
```

Note: Most of the functions and parameters are case sensitive. For example `&linewidth=2` will fail silently. The correct parameter in this case is `&lineWidth=2`

Graphing Metrics

To begin graphing specific metrics, pass one or more *target* parameters and specify a time window for the graph via *from / until*.

target

The `target` parameter specifies a path identifying one or several metrics, optionally with functions acting on those metrics. Paths are documented below, while functions are listed on the *functions* page.

Paths and Wildcards

Metric paths show the "." separated path from the root of the metrics tree (often starting with `servers`) to a metric, for example `servers.ix02ehssvc04v.cpu.total.user`.

Paths also support the following wildcards, which allows you to identify more than one metric in a single path.

Asterisk

> The asterisk (`*`) matches zero or more characters. It is non-greedy, so you can have more than one within a single path element.
>
> Example: `servers.ix*ehssvc*v.cpu.total.*` will return all total CPU metrics for all servers matching the given name pattern.

Character list or range

> Characters in square brackets (`[...]`) specify a single character position in the path string, and match if the character in that position matches one of the characters in the list or range.
>
> A character range is indicated by 2 characters separated by a dash (`-`), and means that any character between those 2 characters (inclusive) will match. More than one range can be included within the square brackets, e.g. `foo[a-z0-9]bar` will match `foopbar`, `foo7bar` etc..
>
> If the characters cannot be read as a range, they are treated as a list – any character in the list will match, e.g. `foo[bc]ar` will match `foobar` and `foocar`. If you want to include a dash (`-`) in your list, put it at the beginning or end, so it's not interpreted as a range.

Value list

> Comma-separated values within curly braces (`{foo,bar,...}`) are treated as value lists, and match if any of the values matches the current point in the path. For example, `servers.ix01ehssvc04v.cpu.total.{user,system,iowait}` will match the user, system and I/O wait total CPU metrics for the specified server.

Note: All wildcards apply only within a single path element. In other words, they do not include or cross dots (`.`) Therefore, `servers.*` will not match `servers.ix02ehssvc04v.cpu.total.user`, while `servers.*.*.*.*` will.

Examples

This will draw one or more metrics

Example:

```
&target=company.server05.applicationInstance04.requestsHandled
(draws one metric)
```

Let's say there are 4 identical application instances running on each server:

```
&target=company.server05.applicationInstance*.requestsHandled
(draws 4 metrics / lines)
```

Now let's say you have 10 servers:

```
&target=company.server*.applicationInstance*.requestsHandled
(draws 40 metrics / lines)
```

You can also run any number of *functions* on the various metrics before graphing:

```
&target=averageSeries(company.server*.applicationInstance.requestsHandled)
(draws 1 aggregate line)
```

The target param can also be repeated to graph multiple related metrics:

```
&target=company.server1.loadAvg&target=company.server1.memUsage
```

Note: If more than 10 metrics are drawn the legend is no longer displayed. See the *hideLegend* parameter for details.

from / until

These are optional parameters that specify the relative or absolute time period to graph `from` specifies the beginning, `until` specifies the end. If `from` is omitted, it defaults to 24 hours ago If `until` is omitted, it defaults to the current time (now).

There are multiple possible formats for these functions:

```
&from=-RELATIVE_TIME
&from=ABSOLUTE_TIME
```

RELATIVE_TIME is a length of time since the current time. It is always preceded by a minus sign (-) and followed by a unit of time. Valid units of time:

Abbreviation	Unit
s	Seconds
min	Minutes
h	Hours
d	Days
w	Weeks
mon	30 Days (month)
y	365 Days (year)

ABSOLUTE_TIME is in the format HH:MM_YYMMDD, YYYYMMDD, MM/DD/YY, or any other `at(1)`-compatible time format.

Abbreviation	Meaning
HH	Hours, in 24h clock format. Times before 12PM must include leading zeroes.
MM	Minutes
YYYY	4 Digit Year.
MM	Numeric month representation with leading zero
DD	Day of month with leading zero

`&from` and `&until` can mix absolute and relative time if desired.

Examples:

```
&from=-8d&until=-7d
(shows same day last week)

&from=04:00_20110501&until=16:00_20110501
(shows 4AM-4PM on May 1st, 2011)

&from=20091201&until=20091231
(shows December 2009)

&from=noon+yesterday
(shows data since 12:00pm on the previous day)

&from=6pm+today
(shows data since 6:00pm on the same day)

&from=january+1
(shows data since the beginning of the current year)

&from=monday
(show data since the previous monday)
```

template

The `target` metrics can use a special `template` function which allows the metric paths to contain variables. Values for these variables can be provided via the `template` query parameter.

Example:

```
&target=template(hosts.$hostname.cpu)&template[hostname]=worker1
```

Default values for the template variables can also be provided:

```
&target=template(hosts.$hostname.cpu, hostname="worker1")
```

Positional arguments can be used instead of named ones:

```
&target=template(hosts.$1.cpu, "worker1")
&target=template(hosts.$1.cpu, "worker1")&template[1]=worker*
```

In addition to path substitution, variables can be used for numeric and string literals:

```
&target=template(constantLine($number))&template[number]=123
&target=template(sinFunction($name))&template[name]=nameOfMySineWaveMetric
```

Data Display Formats

Along with rendering an image, the api can also generate SVG with embedded metadata, PDF, or return the raw data in various formats for external graphing, analysis or monitoring.

format

Controls the format of data returned Affects all `&targets` passed in the URL.

Examples:

```
&format=png
&format=raw
&format=csv
&format=json
&format=svg
&format=pdf
&format=dygraph
&format=rickshaw
```

png

Renders the graph as a PNG image of size determined by *width* and *height*

raw

Renders the data in a custom line-delimited format. Targets are output one per line and are of the format `<target name>,<start timestamp>,<end timestamp>,<series step>|[data]*`.

Example:

```
entries,1311836008,1311836013,1|1.0,2.0,3.0,5.0,6.0
```

csv

Renders the data in a CSV format suitable for import into a spreadsheet or for processing in a script.

Example:

```
entries,2011-07-28 01:53:28,1.0
entries,2011-07-28 01:53:29,2.0
entries,2011-07-28 01:53:30,3.0
entries,2011-07-28 01:53:31,5.0
entries,2011-07-28 01:53:32,6.0
```

json

Renders the data as a json object. The *jsonp* option can be used to wrap this data in a named call for cross-domain access.

```
[{
  "target": "entries",
  "datapoints": [
    [1.0, 1311836008],
    [2.0, 1311836009],
    [3.0, 1311836010],
    [5.0, 1311836011],
    [6.0, 1311836012]
  ]
}]
```

svg

Renders the graph as SVG markup of size determined by *width* and *height*. Metadata about the drawn graph is saved as an embedded script with the variable `metadata` being set to an object describing the graph.

```
<script>
  <![CDATA[
    metadata = {
      "area": {
        "xmin": 39.195507812499997,
        "ymin": 33.96875,
        "ymax": 623.794921875,
        "xmax": 1122
      },
      "series": [
        {
          "start": 1335398400,
          "step": 1800,
          "end": 1335425400,
          "name": "summarize(test.data, \"30min\", \"sum\")",
          "color": "#859900",
          "data": [null, null, 1.0, null, 1.0, null, 1.0, null, 1.0, null, 1.0, null,␣
↪null, null, null],
          "options": {},
          "valuesPerPoint": 1
        }
      ],
      "y": {
        "labelValues": [0, 0.25, 0.5, 0.75, 1.0],
        "top": 1.0,
        "labels": ["0 ", "0.25 ", "0.50 ", "0.75 ", "1.00  "],
        "step": 0.25,
        "bottom": 0
      },
      "x": {
        "start": 1335398400,
        "end": 1335423600
      },
      "font": {
        "bold": false,
        "name": "Sans",
        "italic": false,
        "size": 10
      },
      "options": {
        "lineWidth": 1.2
      }
    }
  ]]>
</script>
```

pdf

Renders the graph as a PDF of size determined by *width* and *height*.

dygraph

Renders the data as a json object suitable for passing to a Dygraph object.

```
{
  "labels" : [
    "Time",
    "entries"
  ],
  "data" : [
    [1468791890000, 0.0],
    [1468791900000, 0.0]
  ]
}
```

rickshaw

Renders the data as a json object suitable for passing to a Rickshaw object.

```
[{
  "target": "entries",
  "datapoints": [{
    "y": 0.0,
    "x": 1468791890
  }, {
    "y": 0.0,
    "x": 1468791900
  }]
}]
```

rawData

Deprecated since version 0.9.9: This option is deprecated in favor of format

Used to get numerical data out of the webapp instead of an image Can be set to true, false, csv. Affects all `&targets` passed in the URL.

Example:

```
&target=carbon.agents.graphiteServer01.cpuUsage&from=-5min&rawData=true
```

Returns the following text:

```
carbon.agents.graphiteServer01.cpuUsage,1306217160,1306217460,60|0.0,0.00666666520965,
↪0.00666666624282,0.0,0.0133345399694
```

Graph Parameters

areaAlpha

Default: 1.0

Takes a floating point number between 0.0 and 1.0.

Sets the alpha (transparency) value of filled areas when using an *areaMode*.

areaMode

Default: none

Enables filling of the area below the graphed lines. Fill area is the same color as the line color associated with it. See *areaAlpha* to make this area transparent. Takes one of the following parameters which determines the fill mode to use:

`none` Disables areaMode

`first` Fills the area under the first target and no other

`all` Fills the areas under each target

`stacked` Creates a graph where the filled area of each target is stacked on one another. Each target line is displayed as the sum of all previous lines plus the value of the current line.

bgcolor

Default: white

Sets the background color of the graph.

Color Names	RGB Value
black	0,0,0
white	255,255,255
blue	100,100,255
green	0,200,0
red	200,0,50
yellow	255,255,0
orange	255, 165, 0
purple	200,100,255
brown	150,100,50
aqua	0,150,150
gray	175,175,175
grey	175,175,175
magenta	255,0,255
pink	255,100,100
gold	200,200,0
rose	200,150,200
darkblue	0,0,255
darkgreen	0,255,0
darkred	255,0,0
darkgray	111,111,111
darkgrey	111,111,111

RGB can be passed directly in the format #RRGGBB where RR, GG, and BB are 2-digit hex vaules for red, green and blue, respectively.

Examples:

```
&bgcolor=blue
&bgcolor=#2222FF
```

cacheTimeout

Default: the value of `cache.default_timeout` in your configuration file. By default, 60 seconds.

colorList

Default: blue,green,red,purple,brown,yellow,aqua,grey,magenta,pink,gold,rose

Takes one or more comma-separated color names or RGB values (see *bgcolor* for a list of color names) and uses that list in order as the colors of the lines. If more lines / metrics are drawn than colors passed, the list is reused in order.

Example:

```
&colorList=green,yellow,orange,red,purple,#DECAFF
```

drawNullAsZero

Default: false

Converts any None (null) values in the displayed metrics to zero at render time.

fgcolor

Default: black

Sets the foreground color This only affects the title, legend text, and axis labels.

See *majorGridLineColor*, and *minorGridLineColor* for further control of colors.

See *bgcolor* for a list of color names and details on formatting this parameter.

fontBold

Default: false

If set to true, makes the font bold.

Example:

```
&fontBold=true
```

fontItalic

Default: false

If set to true, makes the font italic / oblique.

Example:

```
&fontItalic=true
```

fontName

Default: 'Sans'

Change the font used to render text on the graph The font must be installed on the Graphite-API server.

Example:

```
&fontName=FreeMono
```

fontSize

Default: 10

Changes the font size Must be passed a positive floating point number or integer equal to or greater than 1.

Example:

```
&fontSize=8
```

format

See: *Data Display Formats*

from

See: *from / until*

graphOnly

Default: false

Display only the graph area with no grid lines, axes, or legend.

graphType

Default: line

Sets the type of graph to be rendered. Currently there are only two graph types:

line A line graph displaying metrics as lines over time.

pie A pie graph with each slice displaying an aggregate of each metric calculated using the function specified by *pieMode*.

hideLegend

Default: <unset>

If set to `true`, the legend is not drawn.

If set to `false`, the legend is drawn.

If unset, the legend is displayed if there are less than 10 items.

Hint: If set to `false` the `&height` parameter may need to be increased to accommodate the additional text.

Example:

```
&hideLegend=false
```

hideNullFromLegend

Default: False

If set to `true`, series with all null values will not be reported in the legend.

Example:

```
&hideNullFromLegend=true
```

hideAxes

Default: false

If set to `true` the X and Y axes will not be rendered.

Example:

```
&hideAxes=true
```

hideXAxis

Default: false

If set to `true` the X Axis will not be rendered.

hideYAxis

Default: false

If set to `true` the Y Axis will not be rendered.

hideGrid

Default: false

If set to `true` the grid lines will not be rendered.

Example:

```
&hideGrid=true
```

height

Default: 300

Sets the height of the generated graph image in pixels.

See also: *width*

Example:

```
&width=650&height=250
```

jsonp

Default: <unset>

If set and combined with `format=json`, wraps the JSON response in a function call named by the parameter specified.

leftColor

Default: color chosen from colorList.

In dual Y-axis mode, sets the color of all metrics associated with the left Y-axis.

leftDashed

Default: false

In dual Y-axis mode, draws all metrics associated with the left Y-axis using dashed lines.

leftWidth

Default: value of the parameter lineWidth

In dual Y-axis mode, sets the line width of all metrics associated with the left Y-axis.

lineMode

Default: slope

Sets the line drawing behavior. Takes one of the following parameters:

`slope` Slope line mode draws a line from each point to the next. Periods with Null values will not be drawn.

`staircase` Staircase draws a flat line for the duration of a time period and then a vertical line up or down to the next value.

`connected` Like a slope line, but values are always connected with a slope line, regardless of whether or not there are Null values between them.

Example:

```
&lineMode=staircase
```

lineWidth

Default: 1.2

Takes any floating point or integer (negative numbers do not error but will cause no line to be drawn). Changes the width of the line in pixels.

Example:

```
&lineWidth=2
```

logBase

Default: <unset>

If set, draws the graph with a logarithmic scale of the specified base (e.g. 10 for common logarithm).

majorGridLineColor

Default: rose

Sets the color of the major grid lines.

See *bgcolor* for valid color names and formats.

Example:

```
&majorGridLineColor=#FF22FF
```

margin

Default: 10

Sets the margin around a graph image in pixels on all sides.

Example:

```
&margin=20
```

max

Deprecated since version 0.9.0: See *yMax*

maxDataPoints

Set the maximum numbers of datapoints returned when using json content.

If the number of datapoints in a selected range exceeds the maxDataPoints value then the datapoints over the whole period are consolidated.

minorGridLineColor

Default: grey

Sets the color of the minor grid lines.

See *bgcolor* for valid color names and formats.

Example:

```
&minorGridLineColor=darkgrey
```

minorY

Default: 1

Sets the number of minor grid lines per major line on the y-axis.

Example:

```
&minorY=3
```

min

Deprecated since version 0.9.0: See *yMin*

minXStep

Default: 1

Sets the minimum pixel-step to use between datapoints drawn. Any value below this will trigger a point consolidation of the series at render time. The default value of `1` combined with the default lineWidth of `1.2` will cause a minimal amount of line overlap between close-together points. To disable render-time point consolidation entirely, set this to `0` though note that series with more points than there are pixels in the graph area (e.g. a few month's worth of per-minute data) will look very 'smooshed' as there will be a good deal of line overlap. In response, one may use *lineWidth* to compensate for this.

noCache

Default: False

Set it to disable caching in rendered graphs.

noNullPoints

Default: False

If set and combined with `format=json`, removes all null datapoints from the series returned.

pieLabels

Default: horizontal

Orientation to use for slice labels inside of a pie chart.

`horizontal` Labels are oriented horizontally within each slice

`rotated` Labels are oriented radially within each slice

pieMode

Default: average

The type of aggregation to use to calculate slices of a pie when `graphType=pie`. One of:

`average` The average of non-null points in the series.

`maximum` The maximum of non-null points in the series.

`minimum` The minimum of non-null points in the series.

rightColor

Default: color chosen from colorList

In dual Y-axis mode, sets the color of all metrics associated with the right Y-axis.

rightDashed

Default: false

In dual Y-axis mode, draws all metrics associated with the right Y-axis using dashed lines.

rightWidth

Default: value of the parameter lineWidth

In dual Y-axis mode, sets the line width of all metrics associated with the right Y-axis.

template

Default: default

Used to specify a template from `graphTemplates.conf` to use for default colors and graph styles.

Example:

```
&template=plain
```

thickness

Deprecated since version 0.9.0: See: *lineWidth*

title

Default: <unset>

Puts a title at the top of the graph, center aligned. If unset, no title is displayed.

Example:

```
&title=Apache Busy Threads, All Servers, Past 24h
```

tz

Default: The timezone specified in the graphite-api configuration

Time zone to convert all times into.

Examples:

```
&tz=America/Los_Angeles
&tz=UTC
```

uniqueLegend

Default: false

Display only unique legend items, removing any duplicates.

until

See: *from / until*

valueLabels

Default: percent

Determines how slice labels are rendered within a pie chart.

none Slice labels are not shown

numbers Slice labels are reported with the original values

percent Slice labels are reported as a percent of the whole

valueLabelsColor

Default: black

Color used to draw slice labels within a pie chart.

valueLabelsMin

Default: 5

Slice values below this minimum will not have their labels rendered.

vtitle

Default: <unset>

Labels the y-axis with vertical text. If unset, no y-axis label is displayed.

Example:

```
&vtitle=Threads
```

vtitleRight

Default: <unset>

In dual Y-axis mode, sets the title of the right Y-Axis (see: *vtitle*).

width

Default: 330

Sets the width of the generated graph image in pixels.

See also: *height*

Example:

```
&width=650&height=250
```

xFormat

Default: Determined automatically based on the time-width of the X axis

Sets the time format used when displaying the X-axis. See datetime.date.strftime() for format specification details.

yAxisSide

Default: left

Sets the side of the graph on which to render the Y-axis. Accepts values of `left` or `right`.

yDivisors

Default: 4,5,6

Sets the preferred number of intermediate values to display on the Y-axis (Y values between the minimum and maximum). Note that Graphite will ultimately choose what values (and how many) to display based on a 'pretty' factor, which tries to maintain a sensible scale (e.g. preferring intermediary values like 25%,50%,75% over 33.3%,66.6%). To explicitly set the Y-axis values, see *yStep*.

yLimit

Reserved for future use

See: *yMax*

yLimitLeft

Reserved for future use

See: *yMaxLeft*

yLimitRight

Reserved for future use

See: *yMaxRight*

yMin

Default: The lowest value of any of the series displayed

Manually sets the lower bound of the graph. Can be passed any integer or floating point number.

Example:

```
&yMin=0
```

yMax

Default: The highest value of any of the series displayed

Manually sets the upper bound of the graph. Can be passed any integer or floating point number.

Example:

```
&yMax=0.2345
```

yMaxLeft

In dual Y-axis mode, sets the upper bound of the left Y-Axis (see: *yMax*).

yMaxRight

In dual Y-axis mode, sets the upper bound of the right Y-Axis (see: *yMax*).

yMinLeft

In dual Y-axis mode, sets the lower bound of the left Y-Axis (see: *yMin*).

yMinRight

In dual Y-axis mode, sets the lower bound of the right Y-Axis (see: *yMin*).

yStep

Default: Calculated automatically

Manually set the value step between Y-axis labels and grid lines.

yStepLeft

In dual Y-axis mode, Manually set the value step between the left Y-axis labels and grid lines (see: *yStep*).

yStepRight

In dual Y-axis mode, Manually set the value step between the right Y-axis labels and grid lines (see: *yStep*).

yUnitSystem

Default: si

Set the unit system for compacting Y-axis values (e.g. 23,000,000 becomes 23M). Value can be one of:

si Use si units (powers of 1000) - K, M, G, T, P.

binary Use binary units (powers of 1024) - Ki, Mi, Gi, Ti, Pi.

sec Use time units (seconds) - m, H, D, M, Y.

msec Use time units (milliseconds) - s, m, H, D, M, Y.

none Dont compact values, display the raw number.

Built-in functions

Functions are used to transform, combine, and perform computations on series data. They are applied by manipulating the `target` parameters in the *Render API*.

Usage

Most functions are applied to one series list. Functions with the parameter `*seriesLists` can take an arbitrary number of series lists. To pass multiple series lists to a function which only takes one, use the `group()` function.

List of functions

absolute (*seriesList*)
Takes one metric or a wildcard seriesList and applies the mathematical abs function to each datapoint transforming it to its absolute value.

Example:

```
&target=absolute(Server.instance01.threads.busy)
&target=absolute(Server.instance*.threads.busy)
```

aggregateLine (*seriesList*, *func='avg'*)
Takes a metric or wildcard seriesList and draws a horizontal line based on the function applied to each series.

Note: By default, the graphite renderer consolidates data points by averaging data points over time. If you are using the 'min' or 'max' function for aggregateLine, this can cause an unusual gap in the line drawn by this function and the data itself. To fix this, you should use the consolidateBy() function with the same function argument you are using for aggregateLine. This will ensure that the proper data points are retained and the graph should line up correctly.

Example:

```
&target=aggregateLine(server01.connections.total, 'avg')
&target=aggregateLine(server*.connections.total, 'avg')
```

alias (*seriesList*, *newName*)
Takes one metric or a wildcard seriesList and a string in quotes. Prints the string instead of the metric name in the legend.

Example:

```
&target=alias(Sales.widgets.largeBlue,"Large Blue Widgets")
```

aliasByMetric(*seriesList*)

Takes a seriesList and applies an alias derived from the base metric name.

Example:

```
&target=aliasByMetric(carbon.agents.graphite.creates)
```

aliasByNode(*seriesList*, **nodes*)

Takes a seriesList and applies an alias derived from one or more "node" portion/s of the target name. Node indices are 0 indexed.

Example:

```
&target=aliasByNode(ganglia.*.cpu.load5,1)
```

aliasSub(*seriesList*, *search*, *replace*)

Runs series names through a regex search/replace.

Example:

```
&target=aliasSub(ip.*TCP*,"^.*TCP(\d+)","\1")
```

alpha(*seriesList*, *alpha*)

Assigns the given alpha transparency setting to the series. Takes a float value between 0 and 1.

applyByNode(*seriesList*, *nodeNum*, *templateFunction*, *newName=None*)

Takes a seriesList and applies some complicated function (described by a string), replacing templates with unique prefixes of keys from the seriesList (the key is all nodes up to the index given as *nodeNum*).

If the *newName* parameter is provided, the name of the resulting series will be given by that parameter, with any "%" characters replaced by the unique prefix.

Example:

```
&target=applyByNode(servers.*.disk.bytes_free,1,
        "divideSeries(%.disk.bytes_free,sumSeries(%.disk.bytes_*))")
```

Would find all series which match *servers.*.disk.bytes_free*, then trim them down to unique series up to the node given by nodeNum, then fill them into the template function provided (replacing % by the prefixes).

areaBetween(**seriesLists*)

Draws the vertical area in between the two series in seriesList. Useful for visualizing a range such as the minimum and maximum latency for a service.

areaBetween expects **exactly one argument** that results in exactly two series (see example below). The order of the lower and higher values series does not matter. The visualization only works when used in conjunction with `areaMode=stacked`.

Most likely use case is to provide a band within which another metric should move. In such case applying an `alpha()`, as in the second example, gives best visual results.

Example:

```
&target=areaBetween(service.latency.{min,max})&areaMode=stacked

&target=alpha(areaBetween(service.latency.{min,max}),0.3)&areaMode=stacked
```

If for instance, you need to build a seriesList, you should use the `group` function, like so:

```
&target=areaBetween(group(minSeries(a.*.min),maxSeries(a.*.max)))
```

asPercent (*seriesList*, *total=None*)

Calculates a percentage of the total of a wildcard series. If *total* is specified, each series will be calculated as a percentage of that total. If *total* is not specified, the sum of all points in the wildcard series will be used instead.

The *total* parameter may be a single series, reference the same number of series as *seriesList* or a numeric value.

Example:

```
&target=asPercent(Server01.connections.{failed,succeeded},
                  Server01.connections.attempted)
&target=asPercent(Server*.connections.{failed,succeeded},
                  Server*.connections.attempted)
&target=asPercent(apache01.threads.busy,1500)
&target=asPercent(Server01.cpu.*.jiffies)
```

averageAbove (*seriesList*, *n*)

Takes one metric or a wildcard seriesList followed by an integer N. Out of all metrics passed, draws only the metrics with an average value above N for the time period specified.

Example:

```
&target=averageAbove(server*.instance*.threads.busy,25)
```

Draws the servers with average values above 25.

averageBelow (*seriesList*, *n*)

Takes one metric or a wildcard seriesList followed by an integer N. Out of all metrics passed, draws only the metrics with an average value below N for the time period specified.

Example:

```
&target=averageBelow(server*.instance*.threads.busy,25)
```

Draws the servers with average values below 25.

averageOutsidePercentile (*seriesList*, *n*)

Removes functions lying inside an average percentile interval

averageSeries (**seriesLists*)

Short Alias: avg()

Takes one metric or a wildcard seriesList. Draws the average value of all metrics passed at each time.

Example:

```
&target=averageSeries(company.server.*.threads.busy)
```

averageSeriesWithWildcards (*seriesList*, **positions*)

Call averageSeries after inserting wildcards at the given position(s).

Example:

```
&target=averageSeriesWithWildcards(
    host.cpu-[0-7].cpu-{user,system}.value, 1)
```

This would be the equivalent of:

```
&target=averageSeries(host.*.cpu-user.value)&target=averageSeries(
    host.*.cpu-system.value)
```

cactiStyle (*seriesList*, *system=None*, *units=None*)

Takes a series list and modifies the aliases to provide column aligned output with Current, Max, and Min values in the style of cacti. Optionally takes a “system” value to apply unit formatting in the same style as the Y-axis, or a “unit” string to append an arbitrary unit suffix. NOTE: column alignment only works with monospace fonts such as terminus.

Example:

```
&target=cactiStyle(ganglia.*.net.bytes_out,"si")
&target=cactiStyle(ganglia.*.net.bytes_out,"si","b")
```

changed (*seriesList*)

Takes one metric or a wildcard seriesList. Output 1 when the value changed, 0 when null or the same Example:

```
&target=changed(Server01.connections.handled)
```

color (*seriesList*, *theColor*)

Assigns the given color to the seriesList

Example:

```
&target=color(collectd.hostname.cpu.0.user, 'green')
&target=color(collectd.hostname.cpu.0.system, 'ff0000')
&target=color(collectd.hostname.cpu.0.idle, 'gray')
&target=color(collectd.hostname.cpu.0.idle, '6464ffaa')
```

consolidateBy (*seriesList*, *consolidationFunc*)

Takes one metric or a wildcard seriesList and a consolidation function name.

Valid function names are ‘sum’, ‘average’, ‘min’, and ‘max’.

When a graph is drawn where width of the graph size in pixels is smaller than the number of datapoints to be graphed, Graphite consolidates the values to to prevent line overlap. The consolidateBy() function changes the consolidation function from the default of ‘average’ to one of ‘sum’, ‘max’, or ‘min’. This is especially useful in sales graphs, where fractional values make no sense and a ‘sum’ of consolidated values is appropriate.

Example:

```
&target=consolidateBy(Sales.widgets.largeBlue, 'sum')
&target=consolidateBy(Servers.web01.sda1.free_space, 'max')
```

constantLine (*value*)

Takes a float F.

Draws a horizontal line at value F across the graph.

Example:

```
&target=constantLine(123.456)
```

countSeries (**seriesLists*)

Draws a horizontal line representing the number of nodes found in the seriesList.

Example:

```
&target=countSeries(carbon.agents.*.*)
```

`cumulative`(*seriesList*)

Takes one metric or a wildcard seriesList.

When a graph is drawn where width of the graph size in pixels is smaller than the number of datapoints to be graphed, Graphite consolidates the values to prevent line overlap. The cumulative() function changes the consolidation function from the default of 'average' to 'sum'. This is especially useful in sales graphs, where fractional values make no sense and a 'sum' of consolidated values is appropriate.

Alias for `consolidateBy(series, 'sum')`

Example:

```
&target=cumulative(Sales.widgets.largeBlue)
```

`currentAbove`(*seriesList*, *n*)

Takes one metric or a wildcard seriesList followed by an integer N. Out of all metrics passed, draws only the metrics whose value is above N at the end of the time period specified.

Example:

```
&target=currentAbove(server*.instance*.threads.busy,50)
```

Draws the servers with more than 50 busy threads.

`currentBelow`(*seriesList*, *n*)

Takes one metric or a wildcard seriesList followed by an integer N. Out of all metrics passed, draws only the metrics whose value is below N at the end of the time period specified.

Example:

```
&target=currentBelow(server*.instance*.threads.busy,3)
```

Draws the servers with less than 3 busy threads.

`dashed`(*seriesList*, *dashLength=5*)

Takes one metric or a wildcard seriesList, followed by a float F.

Draw the selected metrics with a dotted line with segments of length F If omitted, the default length of the segments is 5.0

Example:

```
&target=dashed(server01.instance01.memory.free,2.5)
```

`delay`(*seriesList*, *steps*)

This shifts all samples later by an integer number of steps. This can be used for custom derivative calculations, among other things. Note: this will pad the early end of the data with None for every step shifted.

This complements other time-displacement functions such as timeShift and timeSlice, in that this function is indifferent about the step intervals being shifted.

Example:

```
&target=divideSeries(server.FreeSpace,delay(server.FreeSpace,1))
```

This computes the change in server free space as a percentage of the previous free space.

derivative(*seriesList*)

This is the opposite of the integral function. This is useful for taking a running total metric and calculating the delta between subsequent data points.

This function does not normalize for periods of time, as a true derivative would. Instead see the perSecond() function to calculate a rate of change over time.

Example:

```
&target=derivative(company.server.application01.ifconfig.TXPackets)
```

Each time you run ifconfig, the RX and TXPackets are higher (assuming there is network traffic.) By applying the derivative function, you can get an idea of the packets per minute sent or received, even though you're only recording the total.

diffSeries(**seriesLists*)

Subtracts series 2 through n from series 1.

Example:

```
&target=diffSeries(service.connections.total,
                   service.connections.failed)
```

To diff a series and a constant, one should use offset instead of (or in addition to) diffSeries.

Example:

```
&target=offset(service.connections.total, -5)

&target=offset(diffSeries(service.connections.total,
                          service.connections.failed), -4)
```

divideSeries(*dividendSeriesList*, *divisorSeriesList*)

Takes a dividend metric and a divisor metric and draws the division result. A constant may *not* be passed. To divide by a constant, use the scale() function (which is essentially a multiplication operation) and use the inverse of the dividend. (Division by 8 = multiplication by 1/8 or 0.125)

Example:

```
&target=divideSeries(Series.dividends,Series.divisors)
```

divideSeriesLists(*dividendSeriesList*, *divisorSeriesList*)

Iterates over a two lists and divides list1[0] by list2[0], list1[1] by list2[1] and so on. The lists need to be the same length

drawAsInfinite(*seriesList*)

Takes one metric or a wildcard seriesList. If the value is zero, draw the line at 0. If the value is above zero, draw the line at infinity. If the value is null or less than zero, do not draw the line.

Useful for displaying on/off metrics, such as exit codes. (0 = success, anything else = failure.)

Example:

```
drawAsInfinite(Testing.script.exitCode)
```

exclude(*seriesList*, *pattern*)

Takes a metric or a wildcard seriesList, followed by a regular expression in double quotes. Excludes metrics that match the regular expression.

Example:

```
&target=exclude(servers*.instance*.threads.busy,"server02")
```

exponentialMovingAverage (*seriesList*, *windowSize*)
Takes a series of values and a window size and produces an exponential moving average utilizing the following formula:

ema(current) = constant * (Current Value) + (1 - constant) * ema(previous)

The Constant is calculated as:

constant = 2 / (windowSize + 1)

The first period EMA uses a simple moving average for its value.

Example:

```
&target=exponentialMovingAverage(*.transactions.count, 10)
&target=exponentialMovingAverage(*.transactions.count, '-10s')
```

fallbackSeries (*seriesList*, *fallback*)
Takes a wildcard seriesList, and a second fallback metric. If the wildcard does not match any series, draws the fallback metric.

Example:

```
&target=fallbackSeries(server*.requests_per_second, constantLine(0))
```

Draws a 0 line when server metric does not exist.

formatPathExpressions (*seriesList*)
Returns a comma-separated list of unique path expressions.

grep (*seriesList*, *pattern*)
Takes a metric or a wildcard seriesList, followed by a regular expression in double quotes. Excludes metrics that don't match the regular expression.

Example:

```
&target=grep(servers*.instance*.threads.busy,"server02")
```

group (**seriesLists*)
Takes an arbitrary number of seriesLists and adds them to a single seriesList. This is used to pass multiple seriesLists to a function which only takes one.

groupByNode (*seriesList*, *nodeNum*, *callback*)
Takes a serieslist and maps a callback to subgroups within as defined by a common node.

Example:

```
&target=groupByNode(ganglia.by-function.*.*.cpu.load5,2,"sumSeries")
```

Would return multiple series which are each the result of applying the "sumSeries" function to groups joined on the second node (0 indexed) resulting in a list of targets like:

```
sumSeries(ganglia.by-function.server1.*.cpu.load5),
sumSeries(ganglia.by-function.server2.*.cpu.load5),...
```

groupByNodes (*seriesList*, *callback*, **nodes*)
Takes a serieslist and maps a callback to subgroups within as defined by multiple nodes.

Example:

```
&target=groupByNodes(ganglia.server*.*.cpu.load*,"sumSeries",1,4)
```

Would return multiple series which are each the result of applying the “sumSeries” function to groups joined on the nodes’ list (0 indexed) resulting in a list of targets like:

```
sumSeries(ganglia.server1.*.cpu.load5),
sumSeries(ganglia.server1.*.cpu.load10),
sumSeries(ganglia.server1.*.cpu.load15),
sumSeries(ganglia.server2.*.cpu.load5),
sumSeries(ganglia.server2.*.cpu.load10),
sumSeries(ganglia.server2.*.cpu.load15), ...
```

`highestAverage`(*seriesList*, *n=1*)
Takes one metric or a wildcard seriesList followed by an integer N. Out of all metrics passed, draws only the top N metrics with the highest average value for the time period specified.

Example:

```
&target=highestAverage(server*.instance*.threads.busy,5)
```

Draws the top 5 servers with the highest average value.

`highestCurrent`(*seriesList*, *n=1*)
Takes one metric or a wildcard seriesList followed by an integer N. Out of all metrics passed, draws only the N metrics with the highest value at the end of the time period specified.

Example:

```
&target=highestCurrent(server*.instance*.threads.busy,5)
```

Draws the 5 servers with the highest busy threads.

`highestMax`(*seriesList*, *n=1*)
Takes one metric or a wildcard seriesList followed by an integer N.

Out of all metrics passed, draws only the N metrics with the highest maximum value in the time period specified.

Example:

```
&target=highestMax(server*.instance*.threads.busy,5)
```

Draws the top 5 servers who have had the most busy threads during the time period specified.

`hitcount`(*seriesList*, *intervalString*, *alignToInterval=False*)
Estimate hit counts from a list of time series.

This function assumes the values in each time series represent hits per second. It calculates hits per some larger interval such as per day or per hour. This function is like summarize(), except that it compensates automatically for different time scales (so that a similar graph results from using either fine-grained or coarse-grained records) and handles rarely-occurring events gracefully.

`holtWintersAberration`(*seriesList*, *delta=3*)
Performs a Holt-Winters forecast using the series as input data and plots the positive or negative deviation of the series data from the forecast.

`holtWintersConfidenceArea`(*seriesList*, *delta=3*)
Performs a Holt-Winters forecast using the series as input data and plots the area between the upper and lower bands of the predicted forecast deviations.

holtWintersConfidenceBands(*seriesList*, *delta=3*)
Performs a Holt-Winters forecast using the series as input data and plots upper and lower bands with the predicted forecast deviations.

holtWintersForecast(*seriesList*)
Performs a Holt-Winters forecast using the series as input data. Data from one week previous to the series is used to bootstrap the initial forecast.

identity(*name*, *step=60*)
Identity function: Returns datapoints where the value equals the timestamp of the datapoint. Useful when you have another series where the value is a timestamp, and you want to compare it to the time of the datapoint, to render an age

Example:

```
&target=identity("The.time.series")
```

This would create a series named "The.time.series" that contains points where x(t) == t.

Accepts optional second argument as 'step' parameter (default step is 60 sec)

integral(*seriesList*)
This will show the sum over time, sort of like a continuous addition function. Useful for finding totals or trends in metrics that are collected per minute.

Example:

```
&target=integral(company.sales.perMinute)
```

This would start at zero on the left side of the graph, adding the sales each minute, and show the total sales for the time period selected at the right side, (time now, or the time specified by '&until=').

integralByInterval(*seriesList*, *intervalUnit*)
This will do the same as integral() funcion, except resetting the total to 0 at the given time in the parameter "from" Useful for finding totals per hour/day/week/..

Example:

```
&target=integralByInterval(company.sales.perMinute,
                           "1d")&from=midnight-10days
```

This would start at zero on the left side of the graph, adding the sales each minute, and show the evolution of sales per day during the last 10 days.

interpolate(*seriesList*, *limit=inf*)
Takes one metric or a wildcard seriesList, and optionally a limit to the number of 'None' values to skip over. Continues the line with the last received value when gaps ('None' values) appear in your data, rather than breaking your line.

Example:

```
&target=interpolate(Server01.connections.handled)
&target=interpolate(Server01.connections.handled, 10)
```

invert(*seriesList*)
Takes one metric or a wildcard seriesList, and inverts each datapoint (i.e. 1/x).

Example:

```
&target=invert(Server.instance01.threads.busy)
```

isNonNull (*seriesList*)

Takes a metric or wild card seriesList and counts up how many non-null values are specified. This is useful for understanding which metrics have data at a given point in time (ie, to count which servers are alive).

Example:

```
&target=isNonNull(webapp.pages.*.views)
```

Returns a seriesList where 1 is specified for non-null values, and 0 is specified for null values.

keepLastValue (*seriesList*, *limit=inf*)

Takes one metric or a wildcard seriesList, and optionally a limit to the number of 'None' values to skip over. Continues the line with the last received value when gaps ('None' values) appear in your data, rather than breaking your line.

Example:

```
&target=keepLastValue(Server01.connections.handled)
&target=keepLastValue(Server01.connections.handled, 10)
```

legendValue (*seriesList*, **valueTypes*)

Takes one metric or a wildcard seriesList and a string in quotes. Appends a value to the metric name in the legend. Currently one or several of: *last*, *avg*, *total*, *min*, *max*. The last argument can be *si* (default) or *binary*, in that case values will be formatted in the corresponding system.

Example:

```
&target=legendValue(Sales.widgets.largeBlue, 'avg', 'max', 'si')
```

limit (*seriesList*, *n*)

Takes one metric or a wildcard seriesList followed by an integer N.

Only draw the first N metrics. Useful when testing a wildcard in a metric.

Example:

```
&target=limit(server*.instance*.memory.free,5)
```

Draws only the first 5 instance's memory free.

lineWidth (*seriesList*, *width*)

Takes one metric or a wildcard seriesList, followed by a float F.

Draw the selected metrics with a line width of F, overriding the default value of 1, or the &lineWidth=X.X parameter.

Useful for highlighting a single metric out of many, or having multiple line widths in one graph.

Example:

```
&target=lineWidth(server01.instance01.memory.free,5)
```

linearRegression (*seriesList*, *startSourceAt=None*, *endSourceAt=None*)

Graphs the liner regression function by least squares method.

Takes one metric or a wildcard seriesList, followed by a quoted string with the time to start the line and another quoted string with the time to end the line. The start and end times are inclusive (default range is from to until). See `from / until` in the render_api_ for examples of time formats. Datapoints in the range is used to regression.

Example:

```
&target=linearRegression(Server.instance01.threads.busy,'-1d')
&target=linearRegression(Server.instance*.threads.busy,
                         "00:00 20140101","11:59 20140630")
```

linearRegressionAnalysis(*series*)
Returns factor and offset of linear regression function by least squares method.

logarithm(*seriesList*, *base=10*)
Takes one metric or a wildcard seriesList, a base, and draws the y-axis in logarithmic format. If base is omitted, the function defaults to base 10.

Example:

```
&target=log(carbon.agents.hostname.avgUpdateTime,2)
```

lowestAverage(*seriesList*, *n=1*)
Takes one metric or a wildcard seriesList followed by an integer N. Out of all metrics passed, draws only the bottom N metrics with the lowest average value for the time period specified.

Example:

```
&target=lowestAverage(server*.instance*.threads.busy,5)
```

Draws the bottom 5 servers with the lowest average value.

lowestCurrent(*seriesList*, *n=1*)
Takes one metric or a wildcard seriesList followed by an integer N. Out of all metrics passed, draws only the N metrics with the lowest value at the end of the time period specified.

Example:

```
&target=lowestCurrent(server*.instance*.threads.busy,5)
```

Draws the 5 servers with the least busy threads right now.

mapSeries(*seriesList*, *mapNode*)
Short form: `map()`.

Takes a seriesList and maps it to a list of sub-seriesList. Each sub-seriesList has the given mapNode in common.

Example (note: This function is not very useful alone. It should be used with *reduceSeries()*):

```
mapSeries(servers.*.cpu.*,1) =>
    [
        servers.server1.cpu.*,
        servers.server2.cpu.*,
        ...
        servers.serverN.cpu.*
    ]
```

maxSeries(**seriesLists*)
Takes one metric or a wildcard seriesList. For each datapoint from each metric passed in, pick the maximum value and graph it.

Example:

```
&target=maxSeries(Server*.connections.total)
```

maximumAbove (*seriesList*, *n*)

Takes one metric or a wildcard seriesList followed by a constant n. Draws only the metrics with a maximum value above n.

Example:

```
&target=maximumAbove(system.interface.eth*.packetsSent,1000)
```

This would only display interfaces which at one point sent more than 1000 packets/min.

maximumBelow (*seriesList*, *n*)

Takes one metric or a wildcard seriesList followed by a constant n. Draws only the metrics with a maximum value below n.

Example:

```
&target=maximumBelow(system.interface.eth*.packetsSent,1000)
```

This would only display interfaces which always sent less than 1000 packets/min.

minSeries (**seriesLists*)

Takes one metric or a wildcard seriesList. For each datapoint from each metric passed in, pick the minimum value and graph it.

Example:

```
&target=minSeries(Server*.connections.total)
```

minimumAbove (*seriesList*, *n*)

Takes one metric or a wildcard seriesList followed by a constant n. Draws only the metrics with a minimum value above n.

Example:

```
&target=minimumAbove(system.interface.eth*.packetsSent,1000)
```

This would only display interfaces which always sent more than 1000 packets/min.

minimumBelow (*seriesList*, *n*)

Takes one metric or a wildcard seriesList followed by a constant n. Draws only the metrics with a minimum value below n.

Example:

```
&target=minimumBelow(system.interface.eth*.packetsSent,1000)
```

This would only display interfaces which sent at one point less than 1000 packets/min.

mostDeviant (*seriesList*, *n*)

Takes one metric or a wildcard seriesList followed by an integer N. Draws the N most deviant metrics. To find the deviants, the standard deviation (sigma) of each series is taken and ranked. The top N standard deviations are returned.

Example:

```
&target=mostDeviant(server*.instance*.memory.free, 5)
```

Draws the 5 instances furthest from the average memory free.

movingAverage (*seriesList*, *windowSize*)

Graphs the moving average of a metric (or metrics) over a fixed number of past points, or a time interval.

Takes one metric or a wildcard seriesList followed by a number N of datapoints or a quoted string with a length of time like '1hour' or '5min' (See `from / until` in the render_api_ for examples of time formats). Graphs the average of the preceding datapoints for each point on the graph.

Example:

```
&target=movingAverage(Server.instance01.threads.busy,10)
&target=movingAverage(Server.instance*.threads.idle,'5min')
```

`movingMax`(*seriesList*, *windowSize*)
Graphs the moving maximum of a metric (or metrics) over a fixed number of past points, or a time interval.

Takes one metric or a wildcard seriesList followed by a number N of datapoints or a quoted string with a length of time like '1hour' or '5min' (See `from / until` in the render_api_ for examples of time formats). Graphs the maximum of the preceeding datapoints for each point on the graph.

Example:

```
&target=movingMax(Server.instance01.requests,10)
&target=movingMax(Server.instance*.errors,'5min')
```

`movingMedian`(*seriesList*, *windowSize*)
Graphs the moving median of a metric (or metrics) over a fixed number of past points, or a time interval.

Takes one metric or a wildcard seriesList followed by a number N of datapoints or a quoted string with a length of time like '1hour' or '5min' (See `from / until` in the render_api_ for examples of time formats). Graphs the median of the preceding datapoints for each point on the graph.

Example:

```
&target=movingMedian(Server.instance01.threads.busy,10)
&target=movingMedian(Server.instance*.threads.idle,'5min')
```

`movingMin`(*seriesList*, *windowSize*)
Graphs the moving minimum of a metric (or metrics) over a fixed number of past points, or a time interval.

Takes one metric or a wildcard seriesList followed by a number N of datapoints or a quoted string with a length of time like '1hour' or '5min' (See `from / until` in the render_api_ for examples of time formats). Graphs the minimum of the preceeding datapoints for each point on the graph.

Example:

```
&target=movingMin(Server.instance01.requests,10)
&target=movingMin(Server.instance*.errors,'5min')
```

`movingSum`(*seriesList*, *windowSize*)
Graphs the moving sum of a metric (or metrics) over a fixed number of past points, or a time interval.

Takes one metric or a wildcard seriesList followed by a number N of datapoints or a quoted string with a length of time like '1hour' or '5min' (See `from / until` in the render_api_ for examples of time formats). Graphs the sum of the preceeding datapoints for each point on the graph.

Example:

```
&target=movingSum(Server.instance01.requests,10)
&target=movingSum(Server.instance*.errors,'5min')
```

`multiplySeries`(**seriesLists*)
Takes two or more series and multiplies their points. A constant may not be used. To multiply by a constant, use the scale() function.

Example:

```
&target=multiplySeries(Series.dividends,Series.divisors)
```

`multiplySeriesWithWildcards` (*seriesList*, **position*)

Call multiplySeries after inserting wildcards at the given position(s).

Example:

```
&target=multiplySeriesWithWildcards(
    web.host-[0-7].{avg-response,total-request}.value, 2)
```

This would be the equivalent of:

```
&target=multiplySeries(web.host-0.{avg-response,total-request}.value)
&target=multiplySeries(web.host-1.{avg-response,total-request}.value)
...
```

`nPercentile` (*seriesList*, *n*)

Returns n-percent of each series in the seriesList.

`nonNegativeDerivative` (*seriesList*, *maxValue=None*)

Same as the derivative function above, but ignores datapoints that trend down. Useful for counters that increase for a long time, then wrap or reset. (Such as if a network interface is destroyed and recreated by unloading and re-loading a kernel module, common with USB / WiFi cards.

Example:

```
&target=nonNegativederivative(
    company.server.application01.ifconfig.TXPackets)
```

`offset` (*seriesList*, *factor*)

Takes one metric or a wildcard seriesList followed by a constant, and adds the constant to each datapoint.

Example:

```
&target=offset(Server.instance01.threads.busy,10)
```

`offsetToZero` (*seriesList*)

Offsets a metric or wildcard seriesList by subtracting the minimum value in the series from each datapoint.

Useful to compare different series where the values in each series may be higher or lower on average but you're only interested in the relative difference.

An example use case is for comparing different round trip time results. When measuring RTT (like pinging a server), different devices may come back with consistently different results due to network latency which will be different depending on how many network hops between the probe and the device. To compare different devices in the same graph, the network latency to each has to be factored out of the results. This is a shortcut that takes the fastest response (lowest number in the series) and sets that to zero and then offsets all of the other datapoints in that series by that amount. This makes the assumption that the lowest response is the fastest the device can respond, of course the more datapoints that are in the series the more accurate this assumption is.

Example:

```
&target=offsetToZero(Server.instance01.responseTime)
&target=offsetToZero(Server.instance*.responseTime)
```

`perSecond` (*seriesList*, *maxValue=None*)

NonNegativeDerivative adjusted for the series time interval This is useful for taking a running total metric and showing how many requests per second were handled.

Example:

```
&target=perSecond(company.server.application01.ifconfig.TXPackets)
```

Each time you run ifconfig, the RX and TXPackets are higher (assuming there is network traffic.) By applying the nonNegativeDerivative function, you can get an idea of the packets per minute sent or received, even though you're only recording the total.

percentileOfSeries (*seriesList*, *n*, *interpolate=False*)
percentileOfSeries returns a single series which is composed of the n-percentile values taken across a wildcard series at each point. Unless *interpolate* is set to True, percentile values are actual values contained in one of the supplied series.

pow (*seriesList*, *factor*)
Takes one metric or a wildcard seriesList followed by a constant, and raises the datapoint by the power of the constant provided at each point.

Example:

```
&target=pow(Server.instance01.threads.busy,10)
&target=pow(Server.instance*.threads.busy,10)
```

powSeries (**seriesLists*)
Takes two or more series and pows their points. A constant line may be used.

Example:

```
&target=powSeries(Server.instance01.app.requests,
                  Server.instance01.app.replies)
```

randomWalkFunction (*name*, *step=60*)
Short Alias: randomWalk()

Returns a random walk starting at 0. This is great for testing when there is no real data in whisper.

Example:

```
&target=randomWalk("The.time.series")
```

This would create a series named "The.time.series" that contains points where x(t) == x(t-1)+random()-0.5, and x(0) == 0.

Accepts an optional second argument as step parameter (default step is 60 sec).

rangeOfSeries (**seriesLists*)
Takes a wildcard seriesList. Distills down a set of inputs into the range of the series

Example:

```
&target=rangeOfSeries(Server*.connections.total)
```

reduceSeries (*seriesLists*, *reduceFunction*, *reduceNode*, **reduceMatchers*)
Short form: `reduce()`.

Takes a list of seriesLists and reduces it to a list of series by means of the reduceFunction.

Reduction is performed by matching the reduceNode in each series against the list of reduceMatchers. The each series is then passed to the reduceFunction as arguments in the order given by reduceMatchers. The reduceFunction should yield a single series.

The resulting list of series are aliased so that they can easily be nested in other functions.

Example: Map/Reduce asPercent(bytes_used,total_bytes) for each server.

Assume that metrics in the form below exist:

```
servers.server1.disk.bytes_used
servers.server1.disk.total_bytes
servers.server2.disk.bytes_used
servers.server2.disk.total_bytes
servers.server3.disk.bytes_used
servers.server3.disk.total_bytes
...
servers.serverN.disk.bytes_used
servers.serverN.disk.total_bytes
```

To get the percentage of disk used for each server:

```
reduceSeries(mapSeries(servers.*.disk.*,1),
             "asPercent",3,"bytes_used","total_bytes") =>

    alias(asPercent(servers.server1.disk.bytes_used,
                    servers.server1.disk.total_bytes),
          "servers.server1.disk.reduce.asPercent"),
    alias(asPercent(servers.server2.disk.bytes_used,
                    servers.server2.disk.total_bytes),
          "servers.server2.disk.reduce.asPercent"),
    ...
    alias(asPercent(servers.serverN.disk.bytes_used,
                    servers.serverN.disk.total_bytes),
          "servers.serverN.disk.reduce.asPercent")
```

In other words, we will get back the following metrics:

```
servers.server1.disk.reduce.asPercent,
servers.server2.disk.reduce.asPercent,
...
servers.serverN.disk.reduce.asPercent
```

See also:

mapSeries()

removeAbovePercentile(*seriesList*, *n*)
Removes data above the nth percentile from the series or list of series provided. Values above this percentile are assigned a value of None.

removeAboveValue(*seriesList*, *n*)
Removes data above the given threshold from the series or list of series provided. Values above this threshold are assigned a value of None.

removeBelowPercentile(*seriesList*, *n*)
Removes data below the nth percentile from the series or list of series provided. Values below this percentile are assigned a value of None.

removeBelowValue(*seriesList*, *n*)
Removes data below the given threshold from the series or list of series provided. Values below this threshold are assigned a value of None.

removeBetweenPercentile(*seriesList*, *n*)
Removes lines who do not have an value lying in the x-percentile of all the values at a moment

removeEmptySeries (*seriesList*)

Takes one metric or a wildcard seriesList. Out of all metrics passed, draws only the metrics with not empty data.

Example:

```
&target=removeEmptySeries(server*.instance*.threads.busy)
```

Draws only live servers with not empty data.

scale (*seriesList*, *factor*)

Takes one metric or a wildcard seriesList followed by a constant, and multiplies the datapoint by the constant provided at each point.

Example:

```
&target=scale(Server.instance01.threads.busy,10)
&target=scale(Server.instance*.threads.busy,10)
```

scaleToSeconds (*seriesList*, *seconds*)

Takes one metric or a wildcard seriesList and returns "value per seconds" where seconds is a last argument to this functions.

Useful in conjunction with derivative or integral function if you want to normalize its result to a known resolution for arbitrary retentions

secondYAxis (*seriesList*)

Graph the series on the secondary Y axis.

sinFunction (*name*, *amplitude=1*, *step=60*)

Short Alias: sin()

Just returns the sine of the current time. The optional amplitude parameter changes the amplitude of the wave.

Example:

```
&target=sin("The.time.series", 2)
```

This would create a series named "The.time.series" that contains sin(x)*2.

A third argument can be provided as a step parameter (default is 60 secs).

smartSummarize (*seriesList*, *intervalString*, *func='sum'*)

Smarter experimental version of summarize.

sortByMaxima (*seriesList*)

Takes one metric or a wildcard seriesList.

Sorts the list of metrics by the maximum value across the time period specified. Useful with the &areaMode=all parameter, to keep the lowest value lines visible.

Example:

```
&target=sortByMaxima(server*.instance*.memory.free)
```

sortByMinima (*seriesList*)

Takes one metric or a wildcard seriesList.

Sorts the list of metrics by the lowest value across the time period specified.

Example:

```
&target=sortByMinima(server*.instance*.memory.free)
```

sortByName(*seriesList*, *natural=False*)
Takes one metric or a wildcard seriesList.

Sorts the list of metrics by the metric name using either alphabetical order or natural sorting. Natural sorting allows names containing numbers to be sorted more naturally, e.g:

- Alphabetical sorting: server1, server11, server12, server2
- Natural sorting: server1, server2, server11, server12

sortByTotal(*seriesList*)
Takes one metric or a wildcard seriesList.

Sorts the list of metrics by the sum of values across the time period specified.

squareRoot(*seriesList*)
Takes one metric or a wildcard seriesList, and computes the square root of each datapoint.

Example:

```
&target=squareRoot(Server.instance01.threads.busy)
```

stacked(*seriesLists*, *stackName='__DEFAULT__'*)
Takes one metric or a wildcard seriesList and change them so they are stacked. This is a way of stacking just a couple of metrics without having to use the stacked area mode (that stacks everything). By means of this a mixed stacked and non stacked graph can be made

It can also take an optional argument with a name of the stack, in case there is more than one, e.g. for input and output metrics.

Example:

```
&target=stacked(company.server.application01.ifconfig.TXPackets, 'tx')
```

stddevSeries(**seriesLists*)
Takes one metric or a wildcard seriesList. Draws the standard deviation of all metrics passed at each time.

Example:

```
&target=stddevSeries(company.server.*.threads.busy)
```

stdev(*seriesList*, *points*, *windowTolerance=0.1*)
Takes one metric or a wildcard seriesList followed by an integer N. Draw the Standard Deviation of all metrics passed for the past N datapoints. If the ratio of null points in the window is greater than windowTolerance, skip the calculation. The default for windowTolerance is 0.1 (up to 10% of points in the window can be missing). Note that if this is set to 0.0, it will cause large gaps in the output anywhere a single point is missing.

Example:

```
&target=stdev(server*.instance*.threads.busy,30)
&target=stdev(server*.instance*.cpu.system,30,0.0)
```

substr(*seriesList*, *start=0*, *stop=0*)
Takes one metric or a wildcard seriesList followed by 1 or 2 integers. Assume that the metric name is a list or array, with each element separated by dots. Prints n - length elements of the array (if only one integer n is passed) or n - m elements of the array (if two integers n and m are passed). The list starts with element 0 and ends with element (length - 1).

Example:

```
&target=substr(carbon.agents.hostname.avgUpdateTime,2,4)
```

The label would be printed as "hostname.avgUpdateTime".

sumSeries(**seriesLists*)
Short form: sum()

This will add metrics together and return the sum at each datapoint. (See integral for a sum over time)

Example:

```
&target=sum(company.server.application*.requestsHandled)
```

This would show the sum of all requests handled per minute (provided requestsHandled are collected once a minute). If metrics with different retention rates are combined, the coarsest metric is graphed, and the sum of the other metrics is averaged for the metrics with finer retention rates.

sumSeriesWithWildcards(*seriesList*, **positions*)
Call sumSeries after inserting wildcards at the given position(s).

Example:

```
&target=sumSeriesWithWildcards(host.cpu-[0-7].cpu-{user,system}.value,
                                   1)
```

This would be the equivalent of:

```
&target=sumSeries(host.*.cpu-user.value)&target=sumSeries(
    host.*.cpu-system.value)
```

summarize(*seriesList*, *intervalString*, *func='sum'*, *alignToFrom=False*)
Summarize the data into interval buckets of a certain size.

By default, the contents of each interval bucket are summed together. This is useful for counters where each increment represents a discrete event and retrieving a "per X" value requires summing all the events in that interval.

Specifying 'avg' instead will return the mean for each bucket, which can be more useful when the value is a gauge that represents a certain value in time.

'max', 'min' or 'last' can also be specified.

By default, buckets are calculated by rounding to the nearest interval. This works well for intervals smaller than a day. For example, 22:32 will end up in the bucket 22:00-23:00 when the interval=1hour.

Passing alignToFrom=true will instead create buckets starting at the from time. In this case, the bucket for 22:32 depends on the from time. If from=6:30 then the 1hour bucket for 22:32 is 22:30-23:30.

Example:

```
# total errors per hour
&target=summarize(counter.errors, "1hour")

# new users per week
&target=summarize(nonNegativeDerivative(gauge.num_users), "1week")

# average queue size per hour
&target=summarize(queue.size, "1hour", "avg")

# maximum queue size during each hour
&target=summarize(queue.size, "1hour", "max")
```

```
# 2010 Q1-4
&target=summarize(metric, "13week", "avg", true)&from=midnight+20100101
```

threshold(*value*, *label=None*, *color=None*)

Takes a float F, followed by a label (in double quotes) and a color. (See `bgcolor` in the render_api_ for valid color names & formats.)

Draws a horizontal line at value F across the graph.

Example:

```
&target=threshold(123.456, "omgwtfbbq", "red")
```

timeFunction(*name*, *step=60*)

Short Alias: time()

Just returns the timestamp for each X value. T

Example:

```
&target=time("The.time.series")
```

This would create a series named "The.time.series" that contains in Y the same value (in seconds) as X.

A second argument can be provided as a step parameter (default is 60 secs)

timeShift(*seriesList*, *timeShift*, *resetEnd=True*, *alignDST=False*)

Takes one metric or a wildcard seriesList, followed by a quoted string with the length of time (See `from / until` in the render_api_ for examples of time formats).

Draws the selected metrics shifted in time. If no sign is given, a minus sign (-) is implied which will shift the metric back in time. If a plus sign (+) is given, the metric will be shifted forward in time.

Will reset the end date range automatically to the end of the base stat unless resetEnd is False. Example case is when you timeshift to last week and have the graph date range set to include a time in the future, will limit this timeshift to pretend ending at the current time. If resetEnd is False, will instead draw full range including future time.

Because time is shifted by a fixed number of seconds, comparing a time period with DST to a time period without DST, and vice-versa, will result in an apparent misalignment. For example, 8am might be overlaid with 7am. To compensate for this, use the alignDST option.

Useful for comparing a metric against itself at a past periods or correcting data stored at an offset.

Example:

```
&target=timeShift(Sales.widgets.largeBlue,"7d")
&target=timeShift(Sales.widgets.largeBlue,"-7d")
&target=timeShift(Sales.widgets.largeBlue,"+1h")
```

timeSlice(*seriesList*, *startSliceAt*, *endSliceAt='now'*)

Takes one metric or a wildcard metric, followed by a quoted string with the time to start the line and another quoted string with the time to end the line. The start and end times are inclusive. See `from / until` in the render api for examples of time formats.

Useful for filtering out a part of a series of data from a wider range of data.

Example:

```
&target=timeSlice(network.core.port1,"00:00 20140101","11:59 20140630")
&target=timeSlice(network.core.port1,"12:00 20140630","now")
```

timeStack(*seriesList*, *timeShiftUnit*, *timeShiftStart*, *timeShiftEnd*)

Takes one metric or a wildcard seriesList, followed by a quoted string with the length of time (See `from / until` in the render_api_ for examples of time formats). Also takes a start multiplier and end multiplier for the length of time-

Create a seriesList which is composed the original metric series stacked with time shifts starting time shifts from the start multiplier through the end multiplier.

Useful for looking at history, or feeding into averageSeries or stddevSeries.

Example:

```
# create a series for today and each of the previous 7 days
&target=timeStack(Sales.widgets.largeBlue,"1d",0,7)
```

transformNull(*seriesList*, *default=0*, *referenceSeries=None*)

Takes a metric or wildcard seriesList and replaces null values with the value specified by *default*. The value 0 used if not specified. The optional referenceSeries, if specified, is a metric or wildcard series list that governs which time intervals nulls should be replaced. If specified, nulls are replaced only in intervals where a non-null is found for the same interval in any of referenceSeries. This method compliments the drawNullAsZero function in graphical mode, but also works in text-only mode.

Example:

```
&target=transformNull(webapp.pages.*.views,-1)
```

This would take any page that didn't have values and supply negative 1 as a default. Any other numeric value may be used as well.

useSeriesAbove(*seriesList*, *value*, *search*, *replace*)

Compares the maximum of each series against the given *value*. If the series maximum is greater than *value*, the regular expression search and replace is applied against the series name to plot a related metric.

e.g. given useSeriesAbove(ganglia.metric1.reqs,10,'reqs','time'), the response time metric will be plotted only when the maximum value of the corresponding request/s metric is > 10

Example:

```
&target=useSeriesAbove(ganglia.metric1.reqs,10,"reqs","time")
```

verticalLine(*ts*, *label=None*, *color=None*)

Takes a timestamp string ts.

Draws a vertical line at the designated timestamp with optional 'label' and 'color'. Supported timestamp formats include both relative (e.g. -3h) and absolute (e.g. 16:00_20110501) strings, such as those used with `from` and `until` parameters. When set, the 'label' will appear in the graph legend.

Note: Any timestamps defined outside the requested range will raise a 'ValueError' exception.

Example:

```
&target=verticalLine("12:3420131108","event","blue")
&target=verticalLine("16:00_20110501","event")
&target=verticalLine("-5mins")
```

weightedAverage(*seriesListAvg*, *seriesListWeight*, **nodes*)

Takes a series of average values and a series of weights and produces a weighted average for all values.

The corresponding values should share one or more zero-indexed nodes.

Example:

```
&target=weightedAverage(*.transactions.mean,*.transactions.count,0)
&target=weightedAverage(*.transactions.mean,*.transactions.count,1,3,4)
```

Storage finders

Graphite-API searches and fetches metrics from time series databases using an interface called *finders*. The default finder provided with Graphite-API is the one that integrates with Whisper databases.

Customizing finders can be done in the `finders` section of the Graphite-API configuration file:

```
finders:
  - graphite_api.finders.whisper.WhisperFinder
```

Several values are allowed, to let you store different kinds of metrics at different places or smoothly handle transitions from one time series database to another.

The default finder reads data from a Whisper database.

Custom finders

`finders` being a list of arbitrary python paths, it is relatively easy to write a custom finder if you want to read data from other places than Whisper. A finder is a python class with a `find_nodes()` method:

```python
class CustomFinder(object):
    def find_nodes(self, query):
        # ...
```

`query` is a `FindQuery` object. `find_nodes()` is the entry point when browsing the metrics tree. It must yield leaf or branch nodes matching the query:

```python
from graphite_api.node import LeafNode, BranchNode

class CustomFinder(object):
    def find_nodes(self, query):
        # find some paths matching the query, then yield them
        # is_branch or is_leaf are predicates you need to implement
        for path in matches:
            if is_branch(path):
                yield BranchNode(path)
            if is_leaf(path):
                yield LeafNode(path, CustomReader(path))
```

`LeafNode` is created with a *reader*, which is the class responsible for fetching the datapoints for the given path. It is a simple class with 2 methods: `fetch()` and `get_intervals()`:

```python
from graphite_api.intervals import IntervalSet, Interval

class CustomReader(object):
    __slots__ = ('path',)  # __slots__ is recommended to save memory on readers

    def __init__(self, path):
        self.path = path

    def fetch(self, start_time, end_time):
        # fetch data
        time_info = _from_, _to_, _step_
        return time_info, series

    def get_intervals(self):
        return IntervalSet([Interval(start, end)])
```

`fetch()` must return a list of 2 elements: the time info for the data and the datapoints themselves. The time info is a list of 3 items: the start time of the datapoints (in unix time), the end time and the time step (in seconds) between the datapoints.

The datapoints is a list of points found in the database for the required interval. There must be `(end - start) / step` points in the dataset even if the database has gaps: gaps can be filled with `None` values.

`get_intervals()` is a method that hints graphite-web about the time range available for this given metric in the database. It must return an `IntervalSet` of one or more `Interval` objects.

Fetching multiple paths at once

If your storage backend allows it, fetching multiple paths at once is useful to avoid sequential fetches and save time and resources. This can be achieved in three steps:

- Subclass `LeafNode` and add a `__fetch_multi__` class attribute to your subclass:

  ```
  class CustomLeafNode(LeafNode):
      __fetch_multi__ = 'custom'
  ```

  The string `'custom'` is used to identify backends and needs to be unique per-backend.

- Add the `__fetch_multi__` attribute to your finder class:

  ```
  class CustomFinder(objects):
      __fetch_multi__ = 'custom'
  ```

- Implement a `fetch_multi()` method on your finder:

  ```
  class CustomFinder(objects):
      def fetch_multi(self, nodes, start_time, end_time):
          paths = [node.path for node in nodes]
          # fetch paths
          return time_info, series
  ```

  `time_info` is the same structure as the one returned by `fetch()`. `series` is a dictionnary with paths as keys and datapoints as values.

Installing custom finders

In order for your custom finder to be importable, you need to package it under a namespace of your choice. Python packaging won't be covered here but you can look at third-party finders to get some inspiration:

- Cyanite finder

Configuration

Graphite-API instantiates finders and passes it its whole parsed configuration file, as a Python data structure. External finders can require extra sections in the configuration file to setup access to the time series database they communicate with. For instance, let's say your `CustomFinder` needs two configuration parameters, a host and a user:

```
class CustomFinder(object):
    def __init__(self, config):
        config.setdefault('custom', {})
        self.user = config['custom'].get('user', 'default')
        self.host = config['custom'].get('host', 'localhost')
```

The configuration file would look like:

```
finders:
  - custom.CustomFinder
custom:
  user: myuser
  host: example.com
```

When possible, try to use sane defaults that would "just work" for most common setups. Here if the `custom` section isn't provided, the finder uses `default` as user and `localhost` as host.

Custom functions

Just like with storage finders, it is possible to extend Graphite-API to add custom processing functions.

To give an example, let's implement a function that reverses the time series, placing old values at the end and recent values at the beginning.

```
# reverse.py

def reverseSeries(requestContex, seriesList):
    reverse = []
    for series in seriesList:
        reverse.append(TimeSeries(series.name, series.start, series.end,
                                  series.step, series[::-1]))
    return reverse
```

The first argument, `requestContext`, holds some information about the request parameters. `seriesList` is the list of paths found for the request target.

Once you've created your function, declare it in a dictionnary:

```
ReverseFunctions = {
    'reverseSeries': reverseSeries,
}
```

Add your module to the Graphite-API Python path and add it to the configuration:

```
functions:
  - graphite_api.functions.SeriesFunctions
  - graphite_api.functions.PieFunctions
  - reverse.ReverseFunctions
```

Graphite-API releases

1.1.3 – 2016-05-23

- Remove extra parenthesis from `aliasByMetric()`.
- Fix leap year handling in `graphite_api.render.attime`.
- Allow colon and hash in node names in `aliasByNode()`
- Fix calling `reduceFunction` in `reduceSeries`
- Revert a whisper patch which broke multiple retentions handling.
- Specify which function is invalid when providing an invalid consolidation function.

1.1.2 – 2015-11-19

- Fix regression in multi fetch handling: paths were queried multiple times, leading to erroneous behaviour and slowdown.
- Continue on IndexError in `remove{Above,Below}Percentile` functions.

1.1.1 – 2015-10-23

- Fix `areaMode=stacked`.
- Fix error when calling functions that use `fetchWithBootstrap` and the bootstrap range isn't available (fill with nulls instead).

1.1 – 2015-10-05

- Add CarbonLink support.
- Add support for configuring a cache backend and the `noCache` and `cacheTimeout` API options.
- When no timezone is provided in the configuration file, try to guess from the system's timezone with a fallback to UTC.
- Now supporting Flask >= 0.8 and Pyparsing >= 1.5.7.
- Add support for `fetch_multi()` in storage finders. This is useful for database-backed finders such as Cyanite because it allows fetching all time series at once instead of sequentially.
- Add `multiplySeriesWithWildcards`, `minimumBelow`, `changed`, `timeSlice` and `removeEmptySeries` functions.
- Add optional `step` argument to `time`, `sin` and `randomWalk` functions.
- Add `/metrics` API call as an alias to `/metrics/find`.
- Add missing `/metrics/index.json` API call.
- Allow wildcards origins (`*`) in CORS configuration.
- Whisper finder now logs debug information.
- Fix parsing dates such as "feb27" during month days > 28.

- Change `sum()` to return `null` instead of 0 when all series' datapoints are null at the same time. This is graphite-web's behavior.
- Extract paths of all targets before fetching data. This is a significant optimization for storage backends such as Cyanite that allow bulk-fetching metrics.
- Add JSONP support to all API endpoints that can return JSON.
- Fix 500 error when generating a SVG graph without any data.
- Return tracebacks in the HTTP response when app errors occur. This behavior can be disabled in the configuration.
- Fixes for the following graphite-web issues:
  - #639 – proper timezone handling of `from` and `until` with client-supplied timezones.
  - #540 – provide the last data point when rendering to JSON format.
  - #381 – make `areaBetween()` work either when passed 2 arguments or a single wildcard series of length 2.
  - #702 – handle backslash as path separator on windows.
  - #410 – SVG output sometimes had an extra `</g>` tag.

1.0.1 – 2014-03-21

- `time_zone` set to UTC by default instead of Europe/Berlin.
- Properly log app exceptions.
- Fix constantLine for python 3.
- Create whisper directories if they don't exist.
- Fixes for the following graphite-web issues:
  - #645, #625 – allow `constantLine` to work even if there are no other targets in the graph.

1.0.0 – 2014-03-20

Version 1.0 is based on the master branch of Graphite-web, mid-March 2014, with the following modifications:

- New `/index` API endpoint for re-building the index (replaces the build-index command-line script from graphite-web).
- Removal of memcache integration.
- Removal of Pickle integration.
- Removal of remote rendering.
- Support for Python 3.
- A lot more tests and test coverage.
- Fixes for the following graphite-web issues:
  - (meta) #647 – strip out the API from graphite-web.
  - #665 – address some DeprecationWarnings.
  - #658 – accept a float value in `maxDataPoints`.

- #654 – ignore invalid `logBase` values (<=1).
- #591 – accept JSON data additionaly to querystring params or form data.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index