Graphite Documentation

Release 0.9.10

Chris Davis

June 12, 2013

CONTENTS

CHAPTER

ONE

OVERVIEW

1.1 What Graphite is and is not

Graphite does two things:

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

What Graphite does not do is collect data for you, however there are some *tools* out there that know how to send data to graphite. Even though it often requires a little code, *sending data* to Graphite is very simple.

1.2 About the project

Graphite is an enterprise-scale monitoring tool that runs well on cheap hardware. It was originally designed and written by Chris Davis at Orbitz in 2006 as side project that ultimately grew to be a foundational monitoring tool. In 2008, Orbitz allowed Graphite to be released under the open source Apache 2.0 license. Since then Chris has continued to work on Graphite and has deployed it at other companies including Sears, where it serves as a pillar of the e-commerce monitoring system. Today many large *companies* use it.

1.3 The architecture in a nutshell

Graphite consists of 3 software components:

1. **carbon** - a Twisted daemon that listens for time-series data
2. **whisper** - a simple database library for storing time-series data (similar in design to RRD)
3. **graphite webapp** - A Django webapp that renders graphs on-demand using Cairo

Feeding in your data is pretty easy, typically most of the effort is in collecting the data to begin with. As you send datapoints to Carbon, they become immediately available for graphing in the webapp. The webapp offers several ways to create and display graphs including a simple *URL API* for rendering that makes it easy to embed graphs in other webpages.

CHAPTER

TWO

INSTALLING GRAPHITE

2.1 Dependencies

Graphite renders graphs using the Cairo graphics library. This adds dependencies on several graphics-related libraries not typically found on a server. If you're installing from source you can use the `check-dependencies.py` script to see if the dependencies have been met or not.

Basic Graphite requirements:

- Python 2.4 or greater (2.6+ recommended)
- Pycairo
- Django 1.0 or greater
- django-tagging 0.3.1 or greater
- Twisted 8.0 or greater (10.0+ recommended)
- zope-interface (often included in Twisted package dependency)
- fontconfig and at least one font package (a system package usually)
- A WSGI server and web server. Popular choices are: - Apache with mod_wsgi and mod_python - gunicorn with nginx - uWSGI with nginx

Python 2.4 and 2.5 have extra requirements:

- simplejson
- python-sqlite2 or another Django-supported database module

Additionally, the Graphite webapp and Carbon require the whisper database library which is part of the Graphite project.

There are also several other dependencies required for additional features:

- Render caching: memcached and python-memcache
- LDAP authentication: python-ldap (for LDAP authentication support in the webapp)
- AMQP support: txamqp
- RRD support: python-rrdtool
- Dependant modules for additional database support (MySQL, PostgreSQL, etc). See Django database install instructions and the Django database documentation for details

See Also:

On some systems it is necessary to install fonts for Cairo to use. If the webapp is running but all graphs return as broken images, this may be why.

- https://answers.launchpad.net/graphite/+question/38833
- https://answers.launchpad.net/graphite/+question/133390
- https://answers.launchpad.net/graphite/+question/127623

2.2 Fulfilling Dependencies

Most current Linux distributions have all of the requirements available in the base packages. RHEL based distributions may require the EPEL repository for requirements. Python module dependencies can be install with pip rather than system packages if desired or if using a Python version that differs from the system default. Some modules (such as Cairo) may require library development headers to be available.

2.3 Default Installation Layout

Graphite defaults to an installation layout that puts the entire install in its own directory: `/opt/graphite`

2.3.1 Whisper

Whisper is installed Python's system-wide site-packages directory with Whisper's utilities installed in the bin dir of the system's default prefix (generally `/usr/bin/`).

2.3.2 Carbon and Graphite-web

Carbon and Graphite-web are installed in `/opt/graphite/` with the following layout:

- `bin/`
- `conf/`
- `lib/`

  Carbon `PYTHONPATH`
- `storage/`
  - `log`

    Log directory for Carbon and Graphite-web
  - `rrd`

    Location for RRD files to be read
  - `whisper`

    Location for Whisper data files to be stored and read
- `webapp/`

  Graphite-web `PYTHONPATH`

  - `graphite/`

    Location of `manage.py` and `local_settings.py`

  - `content/`

    Graphite-web static content directory

2.4 Installing Graphite

Several installation options exist:

2.4.1 Installing From Source

The latest source tarballs for Graphite-web, Carbon, and Whisper may be fetched from the Graphite project download page or the latest development branches may be cloned from the Github project page:

- Graphite-web: `git clone https://github.com/graphite-project/graphite-web.git`
- Carbon: `git clone https://github.com/graphite-project/carbon.git`
- Whisper: `git clone https://github.com/graphite-project/whisper.git`

Installing in the Default Location

To install Graphite in the *default location*, `/opt/graphite/`, simply execute `python setup.py install` as root in each of the project directories for Graphite-web, Carbon, and Whisper.

Installing Carbon in a Custom Location

Carbon's `setup.py` installer is configured to use a `prefix` of `/opt/graphite` and an `install-lib` of `/opt/graphite/lib`. Carbon's lifecycle wrapper scripts and utilities are installed in `bin`, configuration within `conf`, and stored data in `storage` all within `prefix`. These may be overridden by passing parameters to the `setup.py install` command.

The following parameters influence the install location:

- `--prefix`

  Location to place the `bin/` and `storage/` and `conf/` directories (defaults to `/opt/graphite/`)

- `--install-lib`

  Location to install Python modules (default: `/opt/graphite/lib`)

- `--install-data`

  Location to place the `storage` and `conf` directories (default: value of `prefix`)

- `--install-scripts`

  Location to place the scripts (default: `bin/` inside of `prefix`)

For example, to install everything in `/srv/graphite/`:

```
python setup.py install --prefix=/srv/graphite --install-lib=/srv/graphite/lib
```

To install Carbon into the system-wide site-packages directory with scripts in /usr/bin and storage and configuration in /usr/share/graphite:

```
python setup.py install --install-scripts=/usr/bin --install-lib=/usr/lib/python2.6/site-packages --i
```

Installing Graphite-web in a Custom Location

Graphite-web's setup.py installer is configured to use a prefix of /opt/graphite and an install-lib of /opt/graphite/webapp. Utilities are installed in bin, and configuration in conf within the prefix. These may be overridden by passing parameters to setup.py install

The following parameters influence the install location:

- --prefix

  Location to place the bin/ and conf/ directories (defaults to /opt/graphite/)
- --install-lib

  Location to install Python modules (default: /opt/graphite/webapp)
- --install-data

  Location to place the webapp/content and conf directories (default: value of prefix)
- --install-scripts

  Location to place scripts (default: bin/ inside of prefix)

For example, to install everything in /srv/graphite/:

```
python setup.py install --prefix=/srv/graphite --install-lib=/srv/graphite/webapp
```

To install the Graphite-web code into the system-wide site-packages directory with scripts in /usr/bin and storage configuration, and content in /usr/share/graphite:

```
python setup.py install --install-scripts=/usr/bin --install-lib=/usr/lib/python2.6/site-packages --i
```

2.4.2 Installing From Pip

Versioned Graphite releases can be installed via pip. When installing with pip, Installation of dependencies will automatically be attempted.

Installing in the Default Location

To install Graphite in the *default location*, /opt/graphite/, simply execute as root:

```
pip install whisper
pip install carbon
pip install graphite-web
```

Note: On RedHat-based systems using the python-pip package, the pip executable is named pip-python

Installing Carbon in a Custom Location

Installation of Carbon in a custom location with *pip* is similar to doing so from a source install. Arguments to the underlying `setup.py` controlling installation location can be passed through *pip* with the `--install-option` option.

See *Installing Carbon in a Custom Location* for details of locations and available arguments

For example, to install everything in `/srv/graphite/`:

```
pip install carbon --install-option="--prefix=/srv/graphite" --install-option="--install-lib=/srv/gra
```

To install Carbon into the system-wide site-packages directory with scripts in `/usr/bin` and storage and configuration in `/usr/share/graphite`:

```
pip install carbon --install-option="--install-scripts=/usr/bin" --install-option="--install-lib=/usr
```

Installing Graphite-web in a Custom Location

Installation of Graphite-web in a custom location with *pip* is similar to doing so from a source install. Arguments to the underlying `setup.py` controlling installation location can be passed through *pip* with the `--install-option` option.

See *Installing Graphite-web in a Custom Location* for details on default locations and available arguments

For example, to install everything in `/srv/graphite/`:

```
pip install graphite-web --install-option="--prefix=/srv/graphite" --install-option="--install-lib=/s
```

To install the Graphite-web code into the system-wide site-packages directory with scripts in `/usr/bin` and storage configuration, and content in `/usr/share/graphite`:

```
pip install graphite-web --install-option="--install-scripts=/usr/bin" install-option="--install-lib=
```

2.4.3 Installing in Virtualenv

Virtualenv provides an isolated Python environment to run Graphite in.

Installing in the Default Location

To install Graphite in the *default location*, `/opt/graphite/`, create a virtualenv in `/opt/graphite` and activate it:

```
virtualenv /opt/graphite
source /opt/graphite/bin/activate
```

Once the virtualenv is activated, Graphite and Carbon can be installed *from source* or *via pip*. Note that dependencies will need to be installed while the virtualenv is activated unless –system-site-packages is specified at virtualenv creation time.

Installing in a Custom Location

To install from source activate the virtualenv and see the instructions for *graphite-web* and *carbon*

Running Carbon Within Virtualenv

Carbon may be run within Virtualenv by activating virtualenv before Carbon is started

Running Graphite-web Within Virtualenv

Running Django's manage.py within virtualenv requires activating virtualenv before executing as normal.

The method of running Graphite-web within Virtualenv depends on the WSGI server used:

Apache mod_wsgi

Note: The version Python used to compile mod_wsgi must match the Python installed in the virtualenv (generally the system Python)

To the Apache mod_wsgi config, add the root of the virtualenv as `WSGIPythonHome`, `/opt/graphite` in this example:

```
WSGIPythonHome /opt/graphite
```

and add the virtualenv's python site-packages to the `graphite.wsgi` file, python 2.6 in `/opt/graphite` in this example:

```
site.addsitedir('/opt/graphite/lib/python2.6/site-packages')
```

See the *mod_wsgi documentation on Virtual Environments <http://code.google.com/p/modwsgi/wiki/VirtualEnvironments>* for more details.

Gunicorn

Ensure Gunicorn is installed in the activated virtualenv and execute as normal. If gunicorn is installed system-wide, it may be necessary to execute it from the virtualenv's bin path

uWSGI

Execute uWSGI using the `-H` option to specify the virtualenv root. See the uWSGI documentation on virtualenv for more details.

2.5 Initial Configuration

2.6 Help! It didn't work!

If you run into any issues with Graphite, please to post a question to our Questions forum on Launchpad or join us on IRC in #graphite on FreeNode

2.7 Post-Install Tasks

Configuring Carbon Once you've installed everything you will need to create some basic configuration. Initially none of the config files are created by the installer but example files are provided. Simply copy the `.example` files and customize.

`Administering Carbon` Once Carbon is configured, you need to start it up.

Feeding In Your Data Once it's up and running, you need to feed it some data.

Configuring The Webapp With data getting into carbon, you probably want to look at graphs of it. So now we turn our attention to the webapp.

Administering The Webapp Once its configured you'll need to get it running.

Using the Composer Now that the webapp is running, you probably want to learn how to use it.

CHAPTER

THREE

THE CARBON DAEMONS

When we talk about "Carbon" we mean one or more of various daemons that make up the storage backend of a Graphite installation. In simple installations, there is typically only one daemon, `carbon-cache.py`. This document gives a brief overview of what each daemon does and how you can use them to build a more sophisticated storage backend.

All of the carbon daemons listen for time-series data and can accept it over a common set of *protocols*. However, they differ in what they do with the data once they receive it.

3.1 carbon-cache.py

`carbon-cache.py` accepts metrics over various protocols and writes them to disk as efficiently as possible. This requires caching metric values in RAM as they are received, and flushing them to disk on an interval using the underlying *whisper* library.

`carbon-cache.py` requires some basic configuration files to run:

carbon.conf The `[cache]` section tells `carbon-cache.py` what ports (2003/2004/7002), protocols (newline delimited, pickle) and transports (TCP/UDP) to listen on.

storage-schemas.conf Defines a retention policy for incoming metrics based on regex patterns. This policy is passed to *whisper* when the `.wsp` file is pre-allocated, and dictates how long data is stored for.

As the number of incoming metrics increases, one `carbon-cache.py` instance may not be enough to handle the I/O load. To scale out, simply run multiple `carbon-cache.py` instances (on one or more machines) behind a `carbon-aggregator.py` or `carbon-relay.py`.

Warning: If clients connecting to the `carbon-cache.py` are experiencing errors such as *connection refused* by the daemon, a common reason is a shortage of file descriptors.
In the `console.log` file, if you find presence of:
`Could not accept new connection (EMFILE)`
or
`exceptions.IOError: [Errno 24] Too many open files: '/var/lib/graphite/whisper/systems/`
the number of files `carbon-cache.py` can open will need to be increased. Many systems default to a max of 1024 file descriptors. A value of 8192 or more may be necessary depending on how many clients are simultaneously connecting to the `carbon-cache.py` daemon.
In Linux, the system-global file descriptor max can be set via sysctl. Per-process limits are set via ulimit. See documentation for your operating system distribution for details on how to set these values.

3.2 carbon-relay.py

`carbon-relay.py` serves two distinct purposes: replication and sharding.

When running with `RELAY_METHOD = rules`, a `carbon-relay.py` instance can run in place of a `carbon-cache.py` server and relay all incoming metrics to multiple backend `carbon-cache.py`'s running on different ports or hosts.

In `RELAY_METHOD = consistent-hashing` mode, a `CH_HOST_LIST` setting defines a sharding strategy across multiple `carbon-cache.py` backends. The same consistent hashing list can be provided to the graphite webapp via `CARBONLINK_HOSTS` to spread reads across the multiple backends.

`carbon-relay.py` is configured via:

carbon.conf The `[relay]` section defines listener host/ports and a `RELAY_METHOD`

relay-rules.conf In `RELAY_METHOD = rules`, pattern/servers tuples define what servers metrics matching certain regex rules are forwarded to.

3.3 carbon-aggregator.py

`carbon-aggregator.py` can be run in front of `carbon-cache.py` to buffer metrics over time before reporting them into *whisper*. This is useful when granular reporting is not required, and can help reduce I/O load and whisper file sizes due to lower retention policies.

`carbon-aggregator.py` is configured via:

carbon.conf The `[aggregator]` section defines listener and destination host/ports.

aggregation-rules.conf Defines a time interval (in seconds) and aggregation function (sum or average) for incoming metrics matching a certain pattern. At the end of each interval, the values received are aggregated and published to `carbon-cache.py` as a single metric.

CHAPTER

FOUR

CONFIGURING CARBON

Carbon's config files all live in `/opt/graphite/conf/`. If you've just installed Graphite, none of the `.conf` files will exist yet, but there will be a `.conf.example` file for each one. Simply copy the example files, removing the .example extension, and customize your settings.

4.1 carbon.conf

This is the main config file, and defines the settings for each Carbon daemon.

Each setting within this file is documented via comments in the config file itself. The settings are broken down into sections for each daemon - carbon-cache is controlled by the `[cache]` section, carbon-relay is controlled by `[relay]` and carbon-aggregator by `[aggregator]`. However, if this is your first time using Graphite, don't worry about anything but the `[cache]` section for now.

Tip: Carbon-cache and carbon-relay can run on the same host! Try swapping the default ports listed for `LINE_RECEIVER_PORT` and `PICKLE_RECEIVER_PORT` between the `[cache]` and `[relay]` sections to prevent having to reconfigure your deployed metric senders. When setting `DESTINATIONS` in the `[relay]` section, keep in mind your newly-set `PICKLE_RECEIVER_PORT` in the `[cache]` section.

4.2 storage-schemas.conf

This configuration file details retention rates for storing metrics. It matches metric paths to patterns, and tells whisper what frequency and history of datapoints to store.

Important notes before continuing:

- There can be many sections in this file.
- The sections are applied in order from the top (first) and bottom (last).
- The patterns are regular expressions, as opposed to the wildcards used in the URL API.
- The first pattern that matches the metric name is used.
- This retention is set at the time the first metric is sent.
- Changing this file will not affect already-created .wsp files. Use whisper-resize.py to change those.

A given rule is made up of 3 lines:

- A name, specified inside square brackets.

- A regex, specified after "pattern="
- A retention rate line, specified after "retentions="

The retentions line can specify multiple retentions. Each retention of `frequency:history` is separated by a comma.

Frequencies and histories are specified using the following suffixes:

- s - second
- m - minute
- h - hour
- d - day
- y - year

Here's a simple, single retention example:

```
[garbage_collection]
pattern = garbageCollections$
retentions = 10s:14d
```

The name `[garbage_collection]` is mainly for documentation purposes, and will show up in `creates.log` when metrics matching this section are created.

The regular expression pattern will match any metric that ends with `garbageCollections`. For example, `com.acmeCorp.instance01.jvm.memory.garbageCollections` would match, but `com.acmeCorp.instance01.jvm.memory.garbageCollections.full` would not.

The retention line is saying that each datapoint represents 10 seconds, and we want to keep enough datapoints so that they add up to 14 days of data.

Here's a more complicated example with multiple retention rates:

```
[apache_busyWorkers]
pattern = ^servers\.www.*\.workers\.busyWorkers$
retentions = 15s:7d,1m:21d,15m:5y
```

In this example, imagine that your metric scheme is `servers.<servername>.<metrics>`. The pattern would match server names that start with 'www', followed by anything, that are sending metrics that end in '.workers.busyWorkers' (note the escaped '.' characters).

Additionally, this example uses multiple retentions. The general rule is to specify retentions from most-precise:least-history to least-precise:most-history – whisper will properly downsample metrics (averaging by default) as thresholds for retention are crossed.

By using multiple retentions, you can store long histories of metrics while saving on disk space and I/O. Because whisper averages (by default) as it downsamples, one is able to determine totals of metrics by reversing the averaging process later on down the road.

Example: You store the number of sales per minute for 1 year, and the sales per hour for 5 years after that. You need to know the total sales for January 1st of the year before. You can query whisper for the raw data, and you'll get 24 datapoints, one for each hour. They will most likely be floating point numbers. You can take each datapoint, multiply by 60 (the ratio of high-precision to low-precision datapoints) and still get the total sales per hour.

Additionally, whisper supports a legacy retention specification for backwards compatibility reasons - `seconds-per-datapoint:count-of-datapoints`

```
retentions = 60:1440
```

60 represents the number of seconds per datapoint, and 1440 represents the number of datapoints to store. This required some unnecessarily complicated math, so although it's valid, it's not recommended.

4.3 storage-aggregation.conf

This file defines how to aggregate data to lower-precision retentions. The format is similar to `storage-schemas.conf`. Important notes before continuing:

- This file is optional. If it is not present, defaults will be used.
- There is no `retentions` line. Instead, there are `xFilesFactor` and/or `aggregationMethod` lines.
- `xFilesFactor` should be a floating point number between 0 and 1, and specifies what fraction of the previous retention level's slots must have non-null values in order to aggregate to a non-null value. The default is 0.5.
- `aggregationMethod` specifies the function used to aggregate values for the next retention level. Legal methods are `average`, `sum`, `min`, `max`, and `last`. The default is `average`.
- These are set at the time the first metric is sent.
- Changing this file will not affect .wsp files already created on disk. Use whisper-set-aggregation-method.py to change those.

Here's an example:

```
[all_min]
pattern = \.min$
xFilesFactor = 0.1
aggregationMethod = min
```

The pattern above will match any metric that ends with `.min`.

The `xFilesFactor` line is saying that a minimum of 10% of the slots in the previous retention level must have values for next retention level to contain an aggregate. The `aggregationMethod` line is saying that the aggregate function to use is `min`.

If either `xFilesFactor` or `aggregationMethod` is left out, the default value will be used.

The aggregation parameters are kept separate from the retention parameters because the former depends on the type of data being collected and the latter depends on volume and importance.

4.4 relay-rules.conf

Relay rules are used to send certain metrics to a certain backend. This is handled by the carbon-relay system. It must be running for relaying to work. You can use a regular expression to select the metrics and define the servers to which they should go with the servers line.

Example:

```
[example]
pattern = ^mydata\.foo\..+
servers = 10.1.2.3, 10.1.2.4:2004, myserver.mydomain.com
```

You must define at least one section as the default.

4.5 aggregation-rules.conf

Aggregation rules allow you to add several metrics together as the come in, reducing the need to sum() many metrics in every URL. Note that unlike some other config files, any time this file is modified it will take effect automatically. This requires the carbon-aggregator service to be running.

The form of each line in this file should be as follows:

```
output_template (frequency) = method input_pattern
```

This will capture any received metrics that match 'input_pattern' for calculating an aggregate metric. The calculation will occur every 'frequency' seconds and the 'method' can specify 'sum' or 'avg'. The name of the aggregate metric will be derived from 'output_template' filling in any captured fields from 'input_pattern'.

For example, if your metric naming scheme is:

```
<env>.applications.<app>.<server>.<metric>
```

You could configure some aggregations like so:

```
<env>.applications.<app>.all.requests (60) = sum <env>.applications.<app>.*.requests
<env>.applications.<app>.all.latency (60) = avg <env>.applications.<app>.*.latency
```

As an example, if the following metrics are received:

```
prod.applications.apache.www01.requests
prod.applications.apache.www02.requests
prod.applications.apache.www03.requests
prod.applications.apache.www04.requests
prod.applications.apache.www05.requests
```

They would all go into the same aggregation buffer and after 60 seconds the aggregate metric 'prod.applications.apache.all.requests' would be calculated by summing their values.

4.6 whitelist and blacklist

The whitelist functionality allows any of the carbon daemons to only accept metrics that are explicitly whitelisted and/or to reject blacklisted metrics. The functionality can be enabled in carbon.conf with the `USE_WHITELIST` flag. This can be useful when too many metrics are being sent to a Graphite instance or when there are metric senders sending useless or invalid metrics.

`GRAPHITE_CONF_DIR` is searched for `whitelist.conf` and `blacklist.conf`. Each file contains one regular expressions per line to match against metric values. If the whitelist configuration is missing or empty, all metrics will be passed through by default.

CHAPTER

FIVE

FEEDING IN YOUR DATA

Getting your data into Graphite is very flexible. There are three main methods for sending data to Graphite: Plaintext, Pickle, and AMQP.

It's worth noting that data sent to Graphite is actually sent to the *Carbon and Carbon-Relay*, which then manage the data. The Graphite web interface reads this data back out, either from cache or straight off disk.

Choosing the right transfer method for you is dependent on how you want to build your application or script to send data:

- For a singular script, or for test data, the plaintext protocol is the most straightforward method.
- For sending large amounts of data, you'll want to batch this data up and send it to Carbon's pickle receiver.
- Finally, Carbon can listen to a message bus, via AMQP.

5.1 The plaintext protocol

The plaintext protocol is the most straightforward protocol supported by Carbon.

The data sent must be in the following format: `<metric path> <metric value> <metric timestamp>`. Carbon will then help translate this line of text into a metric that the web interface and Whisper understand.

On Unix, the `nc` program can be used to create a socket and send data to Carbon (by default, 'plaintext' runs on port 2003):

```
PORT=2003
SERVER=graphite.your.org
echo "local.random.diceroll 4 `date +%s`" | nc ${SERVER} ${PORT};
```

5.2 The pickle protocol

The pickle protocol is a much more efficient take on the plaintext protocol, and supports sending batches of metrics to Carbon in one go.

The general idea is that the pickled data forms a list of multi-level tuples:

```
[(path, (timestamp, value)), ...]
```

Once you've formed a list of sufficient size (don't go too big!), send the data over a socket to Carbon's pickle receiver (by default, port 2004). You'll need to pack your pickled data into a packet containing a simple header:

```
payload = pickle.dumps(listOfMetricTuples)
header = struct.pack("!L", len(payload))
message = header + payload
```

You would then send the `message` object through a network socket.

5.3 Using AMQP

...

CHAPTER

SIX

GRAPHITE-WEB'S LOCAL_SETTINGS.PY

Graphite-web uses the convention of importing a `local_settings.py` file from the webapp `settings.py` module. This is where Graphite-web's runtime configuration is loaded from.

6.1 Config File Location

`local_settings.py` is generally located within the main `graphite` module where the webapp's code lives. In the *default installation layout* this is `/opt/graphite/webapp/graphite/local_settings.py`. Alternative locations can be used by symlinking to this path or by ensuring the module can be found within the Python module search path.

6.2 General Settings

TIME_ZONE *Default: America/Chicago*

Set your local timezone. Timezone is specifed using zoneinfo names.

DOCUMENTATION_URL *Default: http://graphite.readthedocs.org/*

Overrides the *Documentation* link used in the header of the Graphite Composer

LOG_RENDERING_PERFORMANCE *Default: False*

Triggers the creation of `rendering.log` which logs timings for calls to the *The Render URL API*

LOG_CACHE_PERFORMANCE *Default: False*

Triggers the creation of `cache.log` which logs timings for remote calls to *carbon-cache* as well as Request Cache (memcached) hits and misses.

LOG_METRIC_ACCESS *Default: False*

Trigges the creation of `metricaccess.log` which logs access to *Whisper* and *RRD* data files

DEBUG = True *Default: False*

Enables generation of detailed Django error pages. See Django's documentation for details

FLUSHRRDCACHED *Default: <unset>*

If set, executes `rrdtool flushcached` before fetching data from RRD files. Set to the address or socket of the rrdcached daemon. Ex: `unix:/var/run/rrdcached.sock`

MEMCACHE_HOSTS *Default: []*

If set, enables the caching of calculated targets (including applied functions) and rendered images. If running a cluster of Graphite webapps, each webapp should have the exact same values for this setting to prevent unneeded cache misses.

Set this to the list of memcached hosts. Ex: `['10.10.10.10:11211', '10.10.10.11:11211', '10.10.10.12:11211']`

DEFAULT_CACHE_DURATION *Default: 60*

Default expiration of cached data and images.

6.3 Filesystem Paths

These settings configure the location of Graphite-web's additional configuration files, static content, and data. These need to be adjusted if Graphite-web is installed outside of the *default installation layout*.

GRAPHITE_ROOT *Default: /opt/graphite* The base directory for the Graphite install. This setting is used to shift the Graphite install from the default base directory which keeping the *default layout*. The paths derived from this setting can be individually overridden as well

CONF_DIR *Default: GRAPHITE_ROOT/conf* The location of additional Graphite-web configuration files

STORAGE_DIR *Default: GRAPHITE_ROOT/storage* The base directory from which WHISPER_DIR, RRD_DIR, LOG_DIR, and INDEX_FILE default paths are referenced

CONTENT_DIR *Default: See below* The location of Graphite-web's static content. This defaults to `content/` two parent directories up from `settings.py`. In the *default layout* this is `/opt/graphite/webapp/content`

DASHBOARD_CONF *Default: CONF_DIR/dashboard.conf* The location of the Graphite-web Dashboard configuration

GRAPHTEMPLATES_CONF *Default: CONF_DIR/graphTemplates.conf* The location of the Graphite-web Graph Template configuration

WHISPER_DIR *Default: /opt/graphite/storage/whisper* The location of Whisper data files

RRD_DIR *Default: /opt/graphite/storage/rrd* The location of RRD data files

STANDARD_DIRS *Default: [WHISPER_DIR, RRD_DIR]* The list of directories searched for data files. By default, this is the value of WHISPER_DIR and RRD_DIR (if rrd support is detected). If this setting is defined, the WHISPER_DIR and RRD_DIR settings have no effect.

LOG_DIR *Default: STORAGE_DIR/log/webapp* The directory to write Graphite-web's log files. This directory must be writable by the user running the Graphite-web webapp

INDEX_FILE *Default: /opt/graphite/storage/index* The location of the search index file. This file is generated by the *build-index.sh* script and must be writable by the user running the Graphite-web webap

6.4 Email Configuration

These settings configure Django's email functionality which is used for emailing rendered graphs. See the Django documentation for further detail on these settings

EMAIL_BACKEND *Default: django.core.mail.backends.smtp.EmailBackend* Set to `django.core.mail.backends.dummy.EmailBackend` to drop emails on the floor and effectively disable email features.

EMAIL_HOST *Default: localhost*

EMAIL_PORT *Default: 25*

EMAIL_HOST_USER *Default: ‘’*

EMAIL_HOST_PASSWORD *Default: ‘’*

EMAIL_USE_TLS *Default: False*

6.5 Authentication Configuration

These settings insert additional backends to the AUTHENTICATION_BACKENDS and MIDDLEWARE_CLASSES settings. Additional authentication schemes are possible by manipulating these lists directly.

6.5.1 LDAP

These settings configure a custom LDAP authentication backend provided by Graphite. Additional settings to the ones below are configurable setting the LDAP module’s global options using `ldap.set_option`. See the module documentation for more details.

```
# SSL Example
import ldap
ldap.set_option(ldap.OPT_X_TLS_REQUIRE_CERT, ldap.OPT_X_TLS_ALLOW)
ldap.set_option(ldap.OPT_X_TLS_CACERTDIR, "/etc/ssl/ca")
ldap.set_option(ldap.OPT_X_TLS_CERTFILE, "/etc/ssl/mycert.pem")
ldap.set_option(ldap.OPT_X_TLS_KEYFILE, "/etc/ssl/mykey.pem")
```

USE_LDAP_AUTH *Default: False*

LDAP_SERVER *Default: ‘’*

Set the LDAP server here or alternativly in `LDAP_URL`

LDAP_PORT *Default: 389*

Set the LDAP server port here or alternativly in `LDAP_URL`

LDAP_URI *Default: None*

Sets the LDAP server URI. E.g. `ldaps://ldap.mycompany.com:636`

LDAP_SEARCH_BASE *Default: ‘’*

Sets the LDAP search base. E.g. `OU=users,DC=mycompany,DC=com`

LDAP_BASE_USER *Default: ‘’*

Sets the base LDAP user to bind to the server with. E.g. `CN=some_readonly_account,DC=mycompany,DC=com`

LDAP_BASE_PASS *Default: ‘’*

Sets the password of the base LDAP user to bind to the server with.

LDAP_USER_QUERY *Default: ‘’*

Sets the LDAP query to return a user object where `%s` substituted with the user id. E.g. `(username=%s)` or `(sAMAccountName=%s)` (Active Directory)

6.5.2 Other Authentications

REMOTE_USER_AUTHENTICATION *Default: False*

Enables the use of the Django *RemoteUserBackend* authentication backend. See the Django documentation for further details

LOGIN_URL *Default: /account/login*

Modifies the URL linked in the *Login* link in the Composer interface. This is useful for directing users to an external authentication link such as for Remote User authentication or a backend such as django_openid_auth

6.6 Dashboard Authorization Configuration

These settings control who is allowed to save and delete dashboards. By default anyone can perform these actions, but by setting DASHBOARD_REQUIRE_AUTHENTICATION, users must at least be logged in to do so. The other two settings allow further restriction of who is able to perform these actions. Users who are not suitably authorized will still be able to use and change dashboards, but will not be able to save changes or delete dashboards.

DASHBOARD_REQUIRE_AUTHENTICATION *Default: False*

If set to True, dashboards can only be saved and deleted by logged in users.

DASHBOARD_REQUIRE_EDIT_GROUP *Default: None*

If set to the name of a user group, dashboards can only be saved and deleted by logged-in users who are members of this group. Groups can be set in the Django Admin app, or in LDAP.

Note that DASHBOARD_REQUIRE_AUTHENTICATION must be set to true - if not, this setting is ignored.

DASHBOARD_REQUIRE_PERMISSIONS *Default: False*

If set to True, dashboards can only be saved or deleted by users having the appropriate (change or delete) permission (as set in the Django Admin app). These permissions can be set at the user or group level. Note that Django's 'add' permission is not used.

Note that DASHBOARD_REQUIRE_AUTHENTICATION must be set to true - if not, this setting is ignored.

6.7 Database Configuration

The following configures the Django database settings. Graphite uses the database for storing user profiles, dashboards, and for the Events functionality. Graphite uses an Sqlite database file located at `STORAGE_DIR/graphite.db` by default. If running multiple Graphite-web instances, a database such as PostgreSQL or MySQL is required so that all instances may share the same data source.

Note: As of Django 1.2, the database configuration is specified by the DATABASES dictionary. For compatibility with Django 1.1, Graphite's default Sqlite database configuration still uses the old method. This means that users running under Django 1.4 will not have a working default. In any case, it is recommended that all users on Django 1.2 or above explicitly specify a database configuration using the new format

See the Django documentation for full documentation of the DATABASE setting. Users on Django 1.1 will require setting the deprecated `DATABASE_*` settings outlined in the Django 1.1 documentation

Note: Remember, setting up a new database requires running `manage.py syncdb` to create the initial schema

6.8 Cluster Configuration

These settings configure the Graphite webapp for clustered use. When `CLUSTER_SERVERS` is set, metric browse and render requests will cause the webapp to query other webapps in CLUSTER_SERVERS for matching metrics. Graphite will use only one successfully matching response to render data. This means that metrics may only live on a single server in the cluster unless the data is consistent on both sources (e.g. with shared SAN storage). Duplicate metric data existing in multiple locations will *not* be combined.

CLUSTER_SERVERS *Default: []*

The list of IP addresses and ports of remote Graphite webapps in a cluster. Each of these servers should have local access to metric data to serve. The first server to return a match for a query will be used to serve that data. Ex: ["10.0.2.2:80", "10.0.2.3:80"]

REMOTE_STORE_FETCH_TIMEOUT *Default: 6*

Timeout for remote data fetches in seconds

REMOTE_STORE_FIND_TIMEOUT *Default: 2.5*

Timeout for remote find requests (metric browsing) in seconds

REMOTE_STORE_RETRY_DELAY *Default: 60*

Time in seconds to blacklist a webapp after a timed-out request

REMOTE_FIND_CACHE_DURATION *Default: 300*

Time to cache remote metric find results in seconds

REMOTE_RENDERING *Default: False*

Enable remote rendering of images and data (JSON, et al.) on remote Graphite webapps. If this is enabled, `RENDERING_HOSTS` must be configured below

RENDERING_HOSTS *Default: []*

List of IP addresses and ports of remote Graphite webapps used to perform rendering. Each webapp must have access to the same data as the Graphite webapp which uses this setting either through shared local storage or via `CLUSTER_SERVERS`. Ex: ["10.0.2.4:80", "10.0.2.5:80"]

REMOTE_RENDER_CONNECT_TIMEOUT *Default: 1.0*

Connection timeout for remote rendering requests in seconds

CARBONLINK_HOSTS *Default: [127.0.0.1:7002]*

If multiple carbon-caches are running on this machine, each should be listed here so that the Graphite webapp may query the caches for data that has not yet been persisted. Remote carbon-cache instances in a multi-host clustered setup should *not* be listed here. Instance names should be listed as applicable. Ex: ['127.0.0.1:7002:a','127.0.0.1:7102:b', '127.0.0.1:7202:c']

CARBONLINK_TIMEOUT *Default: 1.0*

Timeout for carbon-cache cache queries in seconds

6.9 Additional Django Settings

The `local_settings.py.example` shipped with Graphite-web imports `app_settings.py` into the namespace to allow further customization of Django. This allows the setting or customization of standard Django settings and the installation and configuration of additional middleware. To manipulate these settings, ensure `app_settings.py` is imported as such:

```
from graphite.app_settings import *
```

The most common settings to manipulate are `INSTALLED_APPS`, `MIDDLEWARE_CLASSES`, and `AUTHENTICATION_BACKENDS`

CHAPTER

SEVEN

CONFIGURING THE WEBAPP

CHAPTER

EIGHT

ADMINISTERING THE WEBAPP

CHAPTER

NINE

USING THE COMPOSER

...

CHAPTER

TEN

THE RENDER URL API

The graphite webapp provides a `/render` endpoint for generating graphs and retreiving raw data. This endpoint accepts various arguments via query string parameters. These parameters are separated by an ampersand (`&`) and are supplied in the format:

`&name=value`

To verify that the api is running and able to generate images, open `http://GRAPHITE_HOST:GRAPHITE_PORT/render` in a browser. The api should return a simple 330x250 image with the text "No Data".

Once the api is running and you've begun *feeding data into carbon*, 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`

10.1 Graphing Metrics

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

10.1.1 target

The `target` parameter specifies a path identifying one or 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.

10.1.2 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 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 my a minus sign (-) and follow 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)
```

10.2 Data Display Formats

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

10.2.1 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
```

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]*`

```
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

```
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, nul
          "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>
```

pickle

Returns a Python pickle (serialized Python object). The response will have the MIME type 'application/pickle'. The pickled object is a list of dictionaries with the keys: `name`, `start`, `end`, `step`, and `values` as below:

```
[
  {
    'name'  : 'summarize(test.data, "30min", "sum")',
    'start': 1335398400,
    'end'   : 1335425400,
    'step' : 1800,
    'values' : [None, None, 1.0, None, 1.0, None, 1.0, None, 1.0, None, 1.0, None, None, None, None],
  }
]
```

10.2.2 rawData

Deprecated since version 0.9.9. 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|0.0,0.00666666520965,0.006666666242
```

10.3 Graph Parameters

10.3.1 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

10.3.2 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.

10.3.3 bgcolor

Default: value from the [default] template in graphTemplates.conf

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
```

10.3.4 cacheTimeout

Default: The value of DEFAULT_CACHE_DURATION from local_settings.py

The time in seconds for the rendered graph to be cached (only relevant if memcached is configured)

10.3.5 colorList

Default: value from the [default] template in graphTemplates.conf

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
```

10.3.6 drawNullAsZero

Default: false

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

10.3.7 fgcolor

Default: value from the [default] template in graphTemplates.conf

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.

10.3.8 fontBold

Default: value from the [default] template in graphTemplates.conf

If set to true, makes the font bold.

Example:

```
&fontBold=true
```

10.3.9 fontItalic

Default: value from the [default] template in graphTemplates.conf

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

Example:

```
&fontItalic=true
```

10.3.10 fontName

Default: value from the [default] template in graphTemplates.conf

Change the font used to render text on the graph. The font must be installed on the Graphite Server.

Example:

```
&fontName=FreeMono
```

10.3.11 fontSize

Default: value from the [default] template in graphTemplates.conf

Changes the font size. Must be passed a positive floating point number or integer equal to or greater than 1. Default is 10

Example:

```
&fontSize=8
```

10.3.12 format

See: Data Display Formats

10.3.13 from

See: from / until

10.3.14 graphOnly

Default: False

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

10.3.15 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

10.3.16 hideLegend

Default: <unset>

If set to `true`, the legend is not drawn. If set to `false`, the legend is drawn. If unset, the `LEGEND_MAX_ITEMS` settings in `local_settings.py` is used to determine whether or not to display the legend.

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

Example:

```
&hideLegend=false
```

10.3.17 hideAxes

Default: False

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

```
&hideAxes=true
```

10.3.18 hideYAxis

Default: False

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

10.3.19 hideGrid

Default: False

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

Example:

```
&hideGrid=true
```

10.3.20 height

Default: 250

Sets the height of the generated graph image in pixels.

See also: width

Example:

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

10.3.21 jsonp

Default: <unset>

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

10.3.22 leftColor

Default: color chosen from colorList

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

10.3.23 leftDashed

Default: False

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

10.3.24 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

10.3.25 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 will 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
```

10.3.26 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
```

10.3.27 logBase

Default: <unset>

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

10.3.28 localOnly

Default: False

Set to prevent fetching from remote Graphite servers, only returning metrics which are accessible locally

10.3.29 majorGridLineColor

Default: value from the [default] template in graphTemplates.conf

Sets the color of the major grid lines.

See bgcolor for valid color names and formats.

Example:

```
&majorGridLineColor=#FF22FF
```

10.3.30 margin

Default: 10 Sets the margin around a graph image in pixels on all sides.

Example:

```
&margin=20
```

10.3.31 max

Deprecated since version 0.9.0: See yMax

10.3.32 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.

10.3.33 minorGridLineColor

Default: value from the [default] template in graphTemplates.conf

Sets the color of the minor grid lines.

See bgcolor for valid color names and formats.

Example:

```
&minorGridLineColor=darkgrey
```

10.3.34 minorY

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

Example:

```
&minorY=3
```

10.3.35 min

Deprecated since version 0.9.0: See yMin

10.3.36 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.

10.3.37 noCache

Default: False

Set to disable caching of rendered images

10.3.38 pickle

Deprecated since version 0.9.10: See Data Display Formats

10.3.39 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

10.3.40 rightColor

Default: color chosen from colorList

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

10.3.41 rightDashed

Default: False

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

10.3.42 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

10.3.43 template

Default: default

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

Example:

```
&template=plain
```

10.3.44 thickness

Deprecated since version 0.9.0: See: lineWidth

10.3.45 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
```

10.3.46 tz

Default: The timezone specified in local_settings.py

Time zone to convert all times into.

Examples:

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

Note: To change the default timezone, edit `webapp/graphite/local_settings.py`.

10.3.47 uniqueLegend

Default: False

Display only unique legend items, removing any duplicates

10.3.48 until

See: from / until

10.3.49 vtitle

Default: <unset>

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

Example:

```
&vtitle=Threads
```

10.3.50 vtitleRight

Default: <unset>

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

10.3.51 width

Default: 330

Sets the width of the generated graph image in pixels.

See also: height

Example:

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

10.3.52 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.

10.3.53 yAxisSide

Default: left

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

10.3.54 yDivisor

Default: 4,5,6

Supplies the preferred number of intermediate values for the Y-axis to display (Y values between the min and max). Note that Graphite will ultimately choose what values (and how many) to display based on a set of 'pretty' values. To explicitly set the Y-axis values, see yStep

10.3.55 yLimit

Reserved for future use See: yMax

10.3.56 yLimitLeft

Reserved for future use See: yMaxLeft

10.3.57 yLimitRight

Reserved for future use See: yMaxRight

10.3.58 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
```

10.3.59 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
```

10.3.60 yMaxLeft

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

10.3.61 yMaxRight

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

10.3.62 yMinLeft

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

10.3.63 yMinRight

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

10.3.64 yStep

Default: Calculated automatically

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

10.3.65 yStepLeft

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

10.3.66 yStepRight

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

10.3.67 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

`none` Dont compact values, display the raw number

CHAPTER

ELEVEN

FUNCTIONS

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

11.1 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.

11.2 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)
```

`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.

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

`aliasByMetric`(*seriesList*)

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

```
&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.

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

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

Runs series names through a regex search/replace.

```
&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.

areaBetween(*seriesList*)
Draws the area in between the two series in seriesList

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 or a numeric value.

Example:

```
&target=asPercent(Server01.connections.{failed,succeeded}, Server01.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.

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*, **position*)
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(`

cactiStyle(*seriesList*, *system=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. Optonally takes a “system” value to apply unit formatting in the same style as the Y-axis. NOTE: column alignment only works with monospace fonts such as terminus.

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

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.

```
&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.

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

cumulative (*seriesList*, *consolidationFunc='sum'*)
Takes one metric or a wildcard seriesList, and an optional function.

Valid functions are 'sum', 'average', 'min', and 'max'

Sets the consolidation function to 'sum' for the given metric seriesList.

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

```
&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*)

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)
```

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*)

Can take two or more metrics, or a single metric and a constant. Subtracts parameters 2 through n from parameter 1.

Example:

```
&target=diffSeries(service.connections.total,service.connections.failed)
&target=diffSeries(service.connections.total,5)
```

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)
```

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)
```

events(**tags*)

Returns the number of events at this point in time. Usable with drawAsInfinite.

Example:

```
&target=events("tag-one", "tag-two")
&target=events("*")
```

Returns all events tagged as "tag-one" and "tag-two" and the second one returns all events.

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")
```

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

```
&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.l
```

highestAverage (*seriesList*, *n*)

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*)

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*)

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*)

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.

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=’).

invert (*seriesList*)

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

Example:

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

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.

&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)
```

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*)
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*)
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.

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 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 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 sent more 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(5, server*.instance*.memory.free)
```

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 preceeding datapoints for each point on the graph. All previous datapoints are set to None at the beginning of the graph.

Example:

```
&target=movingAverage(Server.instance01.threads.busy,10)
&target=movingAverage(Server.instance*.threads.idle,'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 preceeding datapoints for each point on the graph. All previous datapoints are set to None at the beginning of the graph.

Example:

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

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)
```

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

Derivative 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 derivative 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.

randomWalkFunction(*name*)
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.

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

Example:

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

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 threshole 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 threshole are assigned a value of None

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*)
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.

smartSummarize (*seriesList*, *intervalString*, *func='sum'*, *alignToFrom=False*)
Smarter experimental version of summarize.

The alignToFrom parameter has been deprecated, it no longer has any effect. Alignment happens automatically for days, hours, and minutes.

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)
```

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*, **position*)

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.*.c`

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 caculated 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:

```
&target=summarize(counter.errors, "1hour") # total errors per hour
&target=summarize(nonNegativeDerivative(gauge.num_users), "1week") # new users per week
&target=summarize(queue.size, "1hour", "avg") # average queue size per hour
&target=summarize(queue.size, "1hour", "max") # maximum queue size during each hour
&target=summarize(metric, "13week", "avg", true)&from=midnight+20100101 # 2010 Q1-4
```

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*)
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.

timeShift(*seriesList*, *timeShift*, *resetEnd=True*)
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.

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")
```

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 orginal 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 seriesAverage or seriesStdDev

Example:

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

transformNull(*seriesList*, *default=0*)
Takes a metric or wild card seriesList and an optional value to transform Nulls to. Default is 0. This method compliments drawNullAsZero flag 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

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

CHAPTER

TWELVE

THE DASHBOARD USER INTERFACE

The Dashboard interface is the tool of choice for displaying more than one graph at a time, with all graphs showing the same time range. Unless you're using the HTTP interface to embed graphs in your own applications or web pages, this is the Graphite interface you'll use most often. It's certainly the interface that will be of most use to operations staff.

12.1 Getting Started with the Dashboard Interface

You can access the Dashboard interface directly at `http://my.graphite.host/dashboard`, or via the link at the top of the Composer interface.

12.1.1 Completer or browser tree?

When you open the Dashboard interface, you'll see the top of the page taken up by a completer. This allows you to select a metric series to show on a graph in the dashboard.

If you're only viewing a dashboard rather than modifying one, the completer just gets in the way. You can either resize it by dragging the splitter bar (between the completer and graph panels), or hide it by clicking on the little triangular icon in the splitter bar. Once hidden, the same triangular icon serves to display the panel again.

An alternative to the completer is a browser tree, which shows to the left of the graph panel. To change to this mode, use the *Dashboard | Configure UI* menu item, and choose *Tree (left nav)*. You'll have to refresh the page to get this to show. The completer and browser tree do the same job, so the choice is down to your personal preference. Your choice is recorded in a persistent browser cookie, so it should be preserved across sessions.

12.2 Creating or Modifying a Dashboard

When you open the Dashboard interface, no dashboard is open. You can either start building a new dashboard, or you can open an existing one (see Opening a Dashboard) and modify that. If you're working on a previously-saved dashboard, its name will show at the top of the completer and browser tree panels.

Note for Power Users: Any action that can be performed via the UI, as explained in this section, can also be performed using the Edit Dashboard function (as JSON text). See Editing, Importing and Exporting via JSON.

12.2.1 Adding a Graph

To add a new graph directly, you select a metric series in the completer or browser tree, and a graph for that value is added to the end of the dashboard. Alternatively, if a graph for that metric series already exists on the dashboard, it will be removed.

See later for ways of customizing the graph, including adding multiple metric series, changing axes, adding titles and legends etc.

12.2.2 Importing a Graph

Existing graphs can be imported into your dashboard, either from URLs or from saved graphs.

Import a graph from a URL when you already have the graph you want displaying elsewhere (maybe you built it in the Completer, or you want to copy it from another dashboard). Use the *Graphs | New Graph | From URL* menu item and enter the URL, which you probably copied from another browser window.

Alternatively, if you've saved a graph in the Composer, you can import it. Use the *Graphs | New Graph | From Saved Graph* menu item, and select the graph to import.

12.2.3 Deleting a Graph

When you hover the mouse over a graph, a red cross icon appears at the top right. Click this to delete the graph from the dashboard.

12.2.4 Multiple Metrics - Combining Graphs

The simplest way to show more than one metric on a graph is to add each as a separate graph, and then combine the graphs. To combine 2 graphs, drag one over the other and then wait until the target graph shows "Drop to Merge". Drop the graph, and the target graph will now show all metrics from both graphs. Repeat for as many metrics as required.

Note, however, that if you have multiple *related* metrics, it may be easier to use a single path containing wildcards - see Paths and wildcards.

12.2.5 Re-ordering Graphs

Drag a graph to the position you want, and drop it *before the "Drop to Merge" message shows.*

For power users wanting to perform a large scale re-ordering of graphs in a dashboard, consider using Editing, Importing and Exporting via JSON.

12.2.6 Saving the Dashboard

If the dashboard has previously been saved, and assuming you have any required permissions (see later), you can use the *Dashboard | Save* menu item to save your changes. Note that your dashboard will be visible to all users, whether logged in or not, and can be edited and/or deleted by any user with the required permissions.

You can use the *Dashboard | Save As* menu item to save your dashboard for the first time, or to save it with a different name.

12.3 Viewing a Dashboard

This section explains the options available when viewing an existing dashboard. Once you've defined the dashboards you need, you'll spend most of your time in this mode.

Note that you'll most likely want to hide the completer when working in this mode - see earlier.

12.3.1 Opening a Dashboard

Use the *Dashboard | Finder* menu item to select the dashboard to open.

12.3.2 Setting the Time Range

Graphite allows you to set a time range as relative or absolute. Relative time ranges are most commonly used. The same time range is applied to every graph on the dashboard, and the current time range is shown in the center of the menu bar.

To set a relative time range, click the *Relative Time Range* menu button, and enter the time range to display (value and units, e.g. "6 hours"). By default, this time range ends at the current time, as shown by "Now" in the "Until" units field. However, you can move the time range back by entering your own value and units in the "Until" fields.

To set an absolute time range, click the *Absolute Time range* menu button, and set the start and end dates and times (all are required). Dates can be selected using the calendar picker or entered directly in US format (mm/dd/yyyy), while times can be selected from the dropdown or entered in 12 or 24 hour format (e.g. "5:00 PM", "17:00").

12.3.3 Manual and Auto Refresh

By default, dashboards are set to manually refresh. Click the green refresh menu button to the left of the *Auto-Refresh* button to refresh the dashboard. The time of the last refresh is shown at the right of the menu bar.

Alternatively, set the dashboard to auto-refresh by ensuring that the *Auto-Refresh* menu button is pressed in. The refresh defaults to 60 seconds, but you can change this in the edit field to the right of the *Auto-Refresh* button.

Note that refresh options are saved with the dashboard.

12.4 Customizing Graphs

To change a graph on the dashboard, click on it. This will display a pop-up containing the following sections:

- A list of all metric elements, i.e. the path and functions for each of the data elements displayed on the graph
- An *Apply Function* menu button, which allows functions to be applied to the currently-selected item in the metrics list
- A *Render Operations* menu button, which allows customization of the graph as a whole
- A *Graph Operations* menu button, providing menu items for miscellaneous actions to take on the graph.

Note: The items in the list of metrics can be edited in place. Double-click the item, edit as required, then hit Enter to complete.

12.4.1 Paths and Wildcards

In any reasonably-sized environment, you'll have the same or similar metrics being collected from a number of points. Rather than requiring you to add each one to the graph individually, Graphite provides a powerful wildcard mechanism - for example, the metric path `servers.*ehssvc*.cpu.total.{user,system,iowait}` will include a line on the graph for the user, system and I/O wait CPU usage for every server whose name contains `ehssvc`. Each of these is referred to as a metric series. Graphite also provides a large number of functions for working on groups of metric series, e.g. showing only the top 5 metric series from a group.

See *Paths and Wildcards* for further information.

12.4.2 Customizing a Single Metric Element

To customize a single metric element, you select the element in the metric list, then use the menu items on the *Apply Function* menu button to apply functions to the metric element. Note that each metric element in the list may include multiple metric series, e.g. if the path includes wildcards.

Note: All these actions use functions documented on *the functions page*. For further information, read the documentation for the appropriate function on that page. Function names are included in brackets in the list below.

The functions are grouped in the menu, as follows:

Combine Functions that combine a group of metric series (returned by a path containing wildcards) into a single series (and therefore a single line). Includes sum, average, product, minimum, maximum.

Transform Functions that transform the values in a metric series, against either the Y-axis or (less commonly) the X-axis. Includes scale, scale to seconds, offset, derivative, integral, time-shift, log.

Calculate Functions that calculate a new metric series based on an existing metric series. Includes moving average, percentage, Holt-Winters forecast, ratio and difference (of 2 metrics)

Filter Functions that filter metric series from a group. Includes highest current value, current value above limit, most deviant, remove below percentile.

Special Functions that control how the metric series are drawn on the graph. Includes line colors/widths/styles, drawing stacked, drawing on the second Y-axis, and setting the legend name either directly or from the path.

The last menu item is *Remove Outer Call*, which removes the outer-most function on the current metric.

12.4.3 Customizing the Whole Graph

The *Render Options* menu button is used to set options that apply to the whole graph, rather than just the selected metric.

Note: Each of the items in this menu matches a graph parameter in the *The Render URL API*. For further information, read the documentation for the appropriate parameter on that page.

The functions are grouped as follows:

Graph Title Unsurprisingly, this sets the title for the graph. See *title*.

Display Provides options for:

- fonts (see *fontName*, *fontBold*, *fontItalic*, *fontSize* and *fgcolor*)
- colors (see *colorList*, *bgcolor*, *majorGridLineColor*, *minorGridLineColor* and *areaAlpha*)

- legends (see *hideLegend* and *uniqueLegend*)
- line thickness (see *lineWidth*)
- hiding of graph elements (see *graphOnly*, *hideAxes*, *hideYAxis* and *hideGrid*)
- apply a template (see *template*).

Line Mode Sets the way lines are rendered, e.g. sloped, staircase, connected, and how the value `None` is rendered. See *lineMode* and *drawNullAsZero*.

Area Mode Determines whether the area below lines is filled, and whether the lines are stacked. See *areaMode*.

X-Axis Allows setting the time format for dates/times on the axis (see *xFormat*), the timezone for interpretation of timestamps (see *tz*), and the threshold for point consolidation (the closest number of pixels between points before they are consolidated, see *minXStep*).

Y-Axis Determines how the Y-axis or axes are rendered. This includes:

- label (see *vtitle*)
- minimum/maximum values on the axis (see *yMin* and *yMax*)
- the number of minor lines to draw (see *minorY*)
- drawing on a logarithmic scale of the specified base (see *logBase*)
- step between the Y-axis labels and gridlines (see *yStep*)
- divisor for the axis (see *yDivisor*)
- unit system (SI, binary, or none - see *yUnitSystem*)
- side the axis appears (see *yAxisSide*).

When you have more than one Y-axis (because you selected *Apply Function | Special | Draw in second Y axis* for at least one metric series), use the *Dual Y-Axis Options* item on this menu. This provides individual control of both the left and right Y-axes, with the same settings as listed above.

12.4.4 Other Operations on the Graph

The *Graph Operations* menu button is used to perform miscellaneous actions on the graph.

Breakout *TODO: What does this do?*

Clone Creates a copy of the graph, and adds it to the dashboard.

Email Allows you to send a copy of the graph to someone via email.

Direct URL Provides the URL for rendering this graph, suitable for copying and pasting. Note that changing this URL does not affect the chart it came from, i.e. this is not a mechanism for editing the chart.

12.5 Other Global Menu Options

12.5.1 Editing, Importing and Exporting via JSON

The *Dashboard | Edit Dashboard* menu item shows a JSON (JavaScript Object Notation) representation of the current dashboard and all its graphs in an editor dialog.

If you're a power user, you can edit the dashboard configuration directly. When you click the *Update* button, the changes are applied to the dashboard on screen only. This function also provides a convenient mechanism for importing and exporting dashboards, for instance to promote dashboards from development to production systems.

Note: The Update button does not save your changes - you'll need to use *Save* or *Save As* to do this.

12.5.2 Sharing a Dashboard

The *Share* menu button shows a URL for the dashboard, allowing others to access it directly. This first warns you that your dashboard must be saved, then presents the URL.

Note: If you haven't yet saved your dashboard (ever), it will be given a name like "temporary-0", so you probably want to save it first. It's important to note that temporary dashboards are never shown in the Finder, and so the only way to delete them is via the Admin webapp or the database. You probably don't want that...

12.5.3 Changing Graph Sizes

The *Graphs | Resize* menu item and the Gear menu button allow all graphs on the dashboard to be set to a specified size. You can either choose one of the preset sizes, or select *Custom* and enter your own width and height (in pixels).

12.5.4 New Dashboard

Selecting the *Dashboard | New* menu item removes the association between the current dashboard on the screen and its saved version (if any), which means that you'll need to use *Dashboard | Save As* to save it again. Note that it doesn't clear the contents of the dashboard, i.e. the graphs - use *Remove All* to achieve this.

12.5.5 Removing All Graphs

To remove all graphs on the current dashboard, use the *Graphs | Remove All* menu item or the red cross menu button. This asks for confirmation, and also gives you the option to skip confirmation in future.

12.5.6 Deleting a Dashboard

To delete a dashboard, open the Finder (using the *Dashboard | Finder* menu item), select the dashboard to delete in the list, and click *Delete*. Note that you may need to be logged in as a user with appropriate permissions to do this, depending on the configuration of Graphite.

12.5.7 Login/logout

By default, it's not necessary to be logged in to use or change dashboards. However, your system may be configured to require users to be logged in to change or delete dashboards, and may also require appropriate permissions to do so.

Log into Graphite using the *Dashboard | Log in* menu item, which shows a standard login dialog. Once you're logged in, the menu item changes to *Log out from "username"* - click this to log out again. Note that logins are recorded by a persistent browser cookie, so you don't have to log in again each time you connect to Graphite.

12.5.8 Changing Default Graph Parameters

TODO: What does this do and how?

CHAPTER

THIRTEEN

THE WHISPER DATABASE

Whisper is a fixed-size database, similar in design and purpose to RRD (round-robin-database). It provides fast, reliable storage of numeric data over time. Whisper allows for higher resolution (seconds per point) of recent data to degrade into lower resolutions for long-term retention of historical data.

13.1 Data Points

Data points in Whisper are stored on-disk as big-endian double-precision floats. Each value is paired with a timestamp in seconds since the UNIX Epoch (01-01-1970). The data value is parsed by the Python float() function and as such behaves in the same way for special strings such as `'inf'`. Maximum and minimum values are determined by the Python interpreter's allowable range for float values which can be found by executing:

```
python -c 'import sys; print sys.float_info'
```

13.2 Archives: Retention and Precision

Whisper databases contain one or more archives, each with a specific data resolution and retention (defined in number of points or max timestamp age). Archives are ordered from the highest-resolution and shortest retention archive to the lowest-resolution and longest retention period archive.

To support accurate aggregation from higher to lower resolution archives, the precision of a longer retention archive must be divisible by precision of next lower retention archive. For example, an archive with 1 data point every 60 seconds can have a lower-resolution archive following it with a resolution of 1 data point every 300 seconds because 60 cleanly divides 300. In contrast, a 180 second precision (3 minutes) could not be followed by a 600 second precision (10 minutes) because the ratio of points to be propagated from the first archive to the next would be 3 1/3 and Whisper will not do partial point interpolation.

The total retention time of the database is determined by the archive with the highest retention as the time period covered by each archive is overlapping (see Multi-Archive Storage and Retrieval Behavior). That is, a pair of archives with retentions of 1 month and 1 year will not provide 13 months of data storage as may be guessed. Instead, it will provide 1 year of storage - the length of it's longest archive.

13.3 Rollup Aggregation

Whisper databases with more than a single archive need a strategy to collapse multiple data points for when the data rolls up a lower precision archive. By default, an average function is used. Available aggregation methods are: * average * sum * last * max * min

13.4 Multi-Archive Storage and Retrieval Behavior

When Whisper writes to a database with multiple archives, the incoming data point is written to all archives at once. The data point will be written to the lowest resolution archive as-is, and will be aggregated by the configured aggregation method (see Rollup Aggregation) and placed into each of the higher-retention archives.

When data is retrieved (scoped by a time range), the first archive which can satisfy the entire time period is used. If the time period overlaps an archive boundary, the lower-resolution archive will be used. This allows for a simpler behavior while retrieving data as the data's resolution is consistent through an entire returned series.

13.5 Disk Space Efficiency

Whisper is somewhat inefficient in its usage of disk space because of certain design choices:

Each data point is stored with its timestamp Rather than a timestamp being inferred from its position in the archive, timestamps are stored with each point. The timestamps are used during data retrieval to check the validity of the data point. If a timestamp does not match the expected value for its position relative to the beginning of the requested series, it is known to be out of date and a null value is returned

Archives overlap time periods During the write of a data point, Whisper stores the same data in all archives at once (see Multi-Archive Storage and Retrieval Behavior). Implied by this behavior is that all archives store from now until each of their retention times. Because of this, lower-resolution archives should be configured to significantly lower resolution and higher retentions than their higher-resolution counterparts so as to reduce the overlap.

All time-slots within an archive take up space whether or not a value is stored While Whisper allows for reliable storage of irregular updates, it is most space efficient when data points are stored at every update interval. This behavior is a consequence of the fixed-size design of the database and allows the reading and writing of series data to be performed in a single contiguous disk operation (for each archive in a database).

13.6 Differences Between Whisper and RRD

RRD can not take updates to a time-slot prior to its most recent update This means that there is no way to back-fill data in an RRD series. Whisper does not have this limitation, and this makes importing historical data into Graphite much more simple and easy

RRD was not designed with irregular updates in mind In many cases (depending on configuration) if an update is made to an RRD series but is not followed up by another update soon, the original update will be lost. This makes it less suitable for recording data such as operational metrics (e.g. code pushes)

Whisper requires that metric updates occur at the same interval as the finest resolution storage archive This pushes the onus of aggregating values to fit into the finest precision archive to the user rather than the database. It also means that updates are written immediately into the finest precision archive rather than being staged first for aggregation and written later (during a subsequent write operation) as they are in RRD.

13.7 Performance

Whisper is fast enough for most purposes. It is slower than RRDtool primarily as a consequence of Whisper being written in Python, while RRDtool is written in C. The speed difference between the two in practice is quite small as much effort was spent to optimize Whisper to be as close to RRDtool's speed as possible. Testing has shown that update operations take anywhere from 2 to 3 times as long as RRDtool, and fetch operations take anywhere from 2 to

5 times as long. In practice the actual difference is measured in hundreds of microseconds (10^-4) which means less than a millisecond difference for simple cases.

13.8 Database Format

WhisperFile	*Header,Data*			
	Header	*Metadata,ArchiveInfo+*		
		Metadata	aggregationType,maxRetention,xFilesFactor,archiveCount	
		ArchiveInfo	Offset,SecondsPerPoint,Points	
	Data	*Archive+*		
		Archive	*Point+*	
			Point	timestamp,value

Data types in Python's struct format:

Metadata	`!2LfL`
ArchiveInfo	`!3L`
Point	`!Ld`

CHAPTER

FOURTEEN

GRAPHITE TERMINOLOGY

Graphite uses many terms that can have ambiguous meaning. The following definitions are what these terms mean in the context of Graphite.

datapoint A *value* stored at a *timestamp bucket*. If no value is recorded at a particular timestamp bucket in a *series*, the value will be None (null).

function A time-series function which transforms, combines, or performs computations on one or more *series*. See *Functions*

metric See *series*

metric series See *series*

precision See *resolution*

resolution The number of seconds per datapoint in a *series*. Series are created with a resolution which determines how often a *datapoint* may be stored. This resolution is represented as the number of seconds in time that each datapoint covers. A series which stores one datapoint per minute has a resolution of 60 seconds. Similarly, a series which stores one datapoint per second has a resolution of 1 second.

retention The number of datapoints retained in a *series*. Alternatively: The length of time datapoints are stored in a series.

series A named set of datapoints. A series is identified by a unique name, which is composed of elements separated by periods `(.)` which are used to display the collection of series into a heirarchical tree. A series storing system load average on a server called `apache02` in datacenter `metro_east` might be named as `metro_east.servers.apache02.system.load_average`

series list A series name or wildcard which matches one or more *series*. Series lists are received by *functions* as a list of matching series. From a user perspective, a series list is merely the name of a metric. For example, each of these would be considered a single series list:

- `metro_east.servers.apache02.system.load_average.1_min`,
- `metro_east.servers.apache0{1,2,3}.system.load_average.1_min`
- `metro_east.servers.apache01.system.load_average.*`

target A source of data used as input for a Graph. A target can be a single metric name, a metric wildcard, or either of these enclosed within one or more *functions*

timestamp A point in time in which *values* can be associated. Time in Graphite is represented as epoch time with a maximum resolution of 1-second.

timestamp bucket A *timestamp* after rounding down to the nearest multiple of a *series's resolution*.

value A numeric or null value. Values are stored as double-precision floats. Values are parsed using the python `float()` constructor and can also be None (null). The range and precision of values is system dependant and can be found by executing (with Python 2.6 or later):: python -c 'import sys; print sys.float_info'

CHAPTER

FIFTEEN

TOOLS THAT WORK WITH GRAPHITE

15.1 Backstop

Backstop is a simple endpoint for submitting metrics to Graphite. It accepts JSON data via HTTP POST and proxies the data to one or more Carbon/Graphite listeners.

15.2 Bucky

Bucky is a small service implemented in Python for collecting and translating metrics for Graphite. It can current collect metric data from CollectD daemons and from StatsD clients.

15.3 collectd

collectd is a daemon which collects system performance statistics periodically and provides mechanisms to store the values in a variety of ways, including RRD. To send collectd metrics into carbon/graphite, use collectd's write-graphite plugin (available as of 5.1). Other options include:

- Jordan Sissel's node collectd-to-graphite proxy
- Joe Miller's perl collectd-graphite plugin
- Gregory Szorc's python collectd-carbon plugin
- Paul J. Davis's Bucky service

Graphite can also read directly from collectd's RRD files. RRD files can simply be added to `STORAGE_DIR/rrd` (as long as directory names and files do not contain any `.` characters). For example, collectd's `host.name/load/load.rrd` can be symlinked to `rrd/collectd/host_name/load/load.rrd` to graph `collectd.host_name.load.load.{short,mid,long}term`.

15.4 Collectl

Collectl is a collection tool for system metrics that can be run both interactively and as a daemon and has support for collecting from a broad set of subsystems. Collectl includes a Graphite interface which allows data to easily be fed to Graphite for storage.

15.5 Charcoal

Charcoal is a simple Sinatra dashboarding frontend for Graphite or any other system status service which can generate images directly from a URL. Charcoal configuration is driven by a YAML config file.

15.6 Descartes

Descartes is a Sinatra-based dashboard that allows users to correlate multiple metrics in a single chart, review long-term trends across one or more charts, and to collaborate with other users through a combination of shared dashboards and rich layouts.

15.7 Diamond

Diamond is a Python daemon that collects system metrics and publishes them to Graphite. It is capable of collecting cpu, memory, network, I/O, load and disk metrics. Additionally, it features an API for implementing custom collectors for gathering metrics from almost any source.

15.8 Dusk

Dusk is a simple dashboard for isolating “hotspots” across a fleet of systems. It incorporates horizon charts using Cubism.js to maximize data visualization in a constrained space.

15.9 Evenflow

Evenflow is a simple service for submitting sFlow datagrams to Graphite. It accepts sFlow datagrams from multiple network devices and proxies the data to a Carbon listener. Currently only Generic Interface Counters are supported. All other message types are discarded.

15.10 Ganglia

Ganglia is a scalable distributed monitoring system for high-performance computing systems such as clusters and Grids. It collects system performance metrics and stores them in RRD, but now there is an add-on that allows Ganglia to send metrics directly to Graphite. Further integration work is underway.

15.11 GDash

Gdash is a simple Graphite dashboard built using Twitters Bootstrap driven by a small DSL.

15.12 Giraffe

Giraffe is a Graphite real-time dashboard based on Rickshaw and requires no server backend. Inspired by Gdash, Tasseo and Graphene it mixes features from all three into a slightly different animal.

15.13 Graphitus

graphitus is a client side dashboard for graphite built using bootstrap and underscore.js.

15.14 Graph-Explorer

Graph-Explorer is a graphite dashboard which uses plugins to add tags and metadata to metrics and a query language with lets you filter through them and compose/manipulate graphs on the fly. Also aims for high interactivity using TimeseriesWidget and minimal hassle to set up and get running.

15.15 Graphene

Graphene is a Graphite dashboard toolkit based on D3.js and Backbone.js which was made to offer a very aesthetic realtime dashboard. Graphene provides a solution capable of displaying thousands upon thousands of datapoints all updated in realtime.

15.16 Graphite-relay

Graphite-relay is a fast Graphite relay written in Scala with the Netty framework.

15.17 Graphite-Tattle

Graphite-Tattle is a self-service dashboard frontend for Graphite and Ganglia.

15.18 Graphiti

Graphiti is a powerful dashboard front end with a focus on ease of access, ease of recovery and ease of tweaking and manipulation.

15.19 Graphitoid

Graphitoid is an Android app which allows one to browse and display Graphite graphs on an Android device.

15.20 Graphios

Graphios is a small Python daemon to send Nagios performance data (perfdata) to Graphite.

15.21 Graphitejs

Graphitejs is a jQuery plugin for easily making and displaying graphs and updating them on the fly using the Graphite URL api.

15.22 Graphsky

Graphsky is flexible and easy to configure PHP based dashboard. It uses JSON template files to build graphs and specify which graphs need to be displayed when, similar to Ganglia-web. Just like Ganglia, it uses a hierarchial structure: Environment/Cluster/Host/Metric to be able to display overview graphs and host-specific metrics. It communicates directly to the Graphite API to determine which Environments, Clusters, Hosts and Metrics are currently stored in Graphite.

15.23 Grockets

Grockets is a node.js application which provides streaming JSON data over HTTP from Graphite.

15.24 HoardD

HoardD is a Node.js app written in CoffeeScript to send data from servers to Graphite, much like collectd does, but aimed at being easier to expand and with less footprint. It comes by default with basic collectors plus Redis and MySQL metrics, and can be expanded with Javascript or CoffeeScript.

15.25 Host sFlow

Host sFlow is an open source implementation of the sFlow protocol (http://www.sflow.org), exporting a standard set of host cpu, memory, disk and network I/O metrics. The sflow2graphite utility converts sFlow to Graphite's plaintext protocol, allowing Graphite to receive sFlow metrics.

15.26 hubot-scripts

Hubot is a Campfire bot written in Node.js and CoffeeScript. The related hubot-scripts project includes a Graphite script which supports searching and displaying saved graphs from the Composer directory in your Campfire rooms.

15.27 jmxtrans

jmxtrans is a powerful tool that performs JMX queries to collect metrics from Java applications. It is requires very little configuration and is capable of sending metric data to several backend applications, including Graphite.

15.28 Ledbetter

Ledbetter is a simple script for gathering Nagios problem statistics and submitting them to Graphite. It focuses on summary (overall, servicegroup and hostgroup) statistics and writes them to the nagios.problems metrics namespace within Graphite.

15.29 Logster

Logster is a utility for reading log files and generating metrics in Graphite or Ganglia. It is ideal for visualizing trends of events that are occurring in your application/system/error logs. For example, you might use logster to graph the number of occurrences of HTTP response code that appears in your web server logs.

15.30 metrics-sampler

metrics-sampler is a java program which regularly queries metrics from a configured set of inputs, selects and renames them using regular expressions and sends them to a configured set of outputs. It supports JMX and JDBC as inputs and Graphite as output out of the box.

15.31 Pencil

Pencil is a monitoring frontend for graphite. It runs a webserver that dishes out pretty Graphite URLs in interesting and intuitive layouts.

15.32 Rocksteady

Rocksteady is a system that ties together Graphite, RabbitMQ, and Esper. Developed by AdMob (who was then bought by Google), this was released by Google as open source (http://google-opensource.blogspot.com/2010/09/get-ready-to-rocksteady.html).

15.33 Scales

Scales is a Python server state and statistics library that can output its data to Graphite.

15.34 Seyren

Seyren is an alerting dashboard for Graphite.

15.35 Shinken

Shinken is a system monitoring solution compatible with Nagios which emphasizes scalability, flexibility, and ease of setup. Shinken provides complete integration with Graphite for processing and display of performance data.

15.36 statsd

statsd is a simple daemon for easy stats aggregation, developed by the folks at Etsy. A list of forks and alternative implementations can be found at <http://joemiller.me/2011/09/21/list-of-statsd-server-implementations/>

15.37 Structured Metrics

structured_metrics is a lightweight python library that uses plugins to read in Graphite's list of metric names and convert it into a multi-dimensional tag space of clear, sanitized targets.

15.38 Tasseo

Tasseo is a lightweight, easily configurable, real-time dashboard for Graphite metrics.

15.39 Therry

Therry ia s simple web service that caches Graphite metrics and exposes an endpoint for dumping or searching against them by substring.

15.40 TimeseriesWidget

TimeseriesWidget adds timeseries graphs to your webpages/dashboards using a simple api, focuses on high interactivity and modern features (realtime zooming, datapoint inspection, annotated events, etc). Supports Graphite, flot, rickshaw and anthracite.

CHAPTER

SIXTEEN

WHO IS USING GRAPHITE?

Here are some organizations that use Graphite:

- Orbitz
- Sears Holdings
- Etsy (see http://codeascraft.etsy.com/2010/12/08/track-every-release/)
- Google (opensource Rocksteady project)
- Media Temple
- Canonical
- Brightcove (see http://opensource.brightcove.com/project/Diamond/)
- Vimeo
- SocialTwist
- Douban

And many more

CHAPTER

SEVENTEEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX