Graphite Documentation

Release 0.10.0

Chris Davis

Jun 26, 2017

Contents

CHAPTER 1

Overview

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.

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.

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 2

FAQ

What is Graphite?

Graphite is a highly scalable real-time graphing system. As a user, you write an application that collects numeric time-series data that you are interested in graphing, and send it to Graphite's processing backend, *carbon*, which stores the data in Graphite's specialized database. The data can then be visualized through graphite's web interfaces.

Who should use Graphite?

Anybody who would want to track values of anything over time. If you have a number that could potentially change over time, and you might want to represent the value over time on a graph, then Graphite can probably meet your needs.

Specifically, Graphite is designed to handle numeric time-series data. For example, Graphite would be good at graphing stock prices because they are numbers that change over time. Whether it's a few data points, or dozens of performance metrics from thousands of servers, then Graphite is for you. As a bonus, you don't necessarily know the names of those things in advance (who wants to maintain such huge configuration?); you simply send a metric name, a timestamp, and a value, and Graphite takes care of the rest!

How scalable is Graphite?

From a CPU perspective, Graphite scales horizontally on both the frontend and the backend, meaning you can simply add more machines to the mix to get more throughput. It is also fault tolerant in the sense that losing a backend machine will cause a minimal amount of data loss (whatever that machine had cached in memory) and will not disrupt the system if you have sufficient capacity remaining to handle the load.

From an I/O perspective, under load Graphite performs lots of tiny I/O operations on lots of different files very rapidly. This is because each distinct metric sent to Graphite is stored in its own database file, similar to how many tools (drraw, Cacti, Centreon, etc) built on top of RRD work. In fact, Graphite originally did use RRD for storage until fundamental limitations arose that required a new storage engine.

High volume (a few thousand distinct metrics updating every minute) pretty much requires a good RAID array and/or SSDs. Graphite's backend caches incoming data if the disks cannot keep up with the large number of small write operations that occur (each data point is only a few bytes, but most standard disks cannot do more than a few thousand I/O operations per second, even if they are tiny). When this occurs, Graphite's database engine, whisper, allows carbon to write multiple data points at once, thus increasing overall throughput only at the cost of keeping excess data cached in memory until it can be written.

Graphite also supports *alternative storage backends* which can greatly change these characteristics.

How real-time are the graphs?

Very. Even under heavy load, where the number of metrics coming in each time interval is much greater than the rate at which your storage system can perform I/O operations and lots of data points are being cached in the storage pipeline (see previous question for explanation), Graphite still draws real-time graphs. The trick is that when the Graphite webapp receives a request to draw a graph, it simultaneously retrieves data off the disk as well as from the pre-storage cache (which may be distributed if you have multiple backend servers) and combines the two sources of data to create a real-time graph.

Who already uses Graphite?

Graphite was internally developed by Orbitz where it is used to visualize a variety of operations-critical data including application metrics, database metrics, sales, etc. At the time of this writing, the production system at Orbitz can handle approximately 160,000 distinct metrics per minute running on two niagra-2 Sun servers on a very fast SAN.

What is Graphite written in?

Python2. The Graphite webapp is built on the Django web framework and uses the ExtJS javascript GUI toolkit. The graph rendering is done using the Cairo graphics library. The backend and database are written in pure Python.

Who writes and maintains Graphite?

Graphite was initially developed by Chris Davis at Orbitz. Orbitz has long been a part of the open source community and has published several other internally developed products.

Graphite is currently developed by a team of volunteers under the Graphite-Project GitHub Organization.

What license is Graphite released under?

The Apache 2.0 License.

Does Graphite use RRDtool?

No, not since Graphite was first written in 2006 at least. Graphite has its own specialized database library called *whisper*, which is very similar in design to RRD, but has a subtle yet fundamentally important difference that Graphite requires. There are two reasons whisper was created. The first reason is that RRD is designed under the assumption that

data will always be inserted into the database on a regular basis, and this assumption causes RRD behave undesirably when given irregularly occurring data. Graphite was built to facilitate visualization of various application metrics that do not always occur regularly, like when an uncommon request is handled and the latency is measured and sent to Graphite. Using RRD, the data gets put into a temporary area inside the database where it is not accessible until the current time interval has passed *and* another value is inserted into the database for the following interval. If that does not happen within an allotted period of time, the original data point will get overwritten and is lost. Now for some metrics, the lack of a value can be correctly interpreted as a value of zero, however this is not the case for metrics like latency because a zero indicates that work was done in zero time, which is different than saying no work was done. Assuming a zero value for latency also screws up analysis like calculating the average latency, etc.

The second reason whisper was written is performance. RRDtool is very fast; in fact it is much faster than whisper. But the problem with RRD (at the time whisper was written) was that RRD only allowed you to insert a single value into a database at a time, while whisper was written to allow the insertion of multiple data points at once, compacting them into a single write operation. This improves performance drastically under high load because Graphite operates on many many files, and with such small operations being done (write a few bytes here, a few over there, etc) the bottleneck is caused by the *number of I/O operations*. Consider the scenario where Graphite is receiving 100,000 distinct metric values each minute; in order to sustain that load Graphite must be able to write that many data points to disk each minute. But assume that your underlying storage can only handle 20,000 I/O operations per minute. With RRD (at the time whisper was written), there was no chance of keeping up. But with whisper, we can keep caching the incoming data until we accumulate say 10 minutes worth of data for a given metric, then instead of doing 10 I/O operations to write those 10 data points, whisper can do it in one operation. The reason I have kept mentioning "at the time whisper was written" is that RRD now supports this behavior. However Graphite will continue to use whisper as long as the first issue still exists.

How do I report problems or request features for Graphite?

Please post any feature requests or bug reports to the GitHub Issues page.

Is this Graphite related to the SIL font rendering graphite?

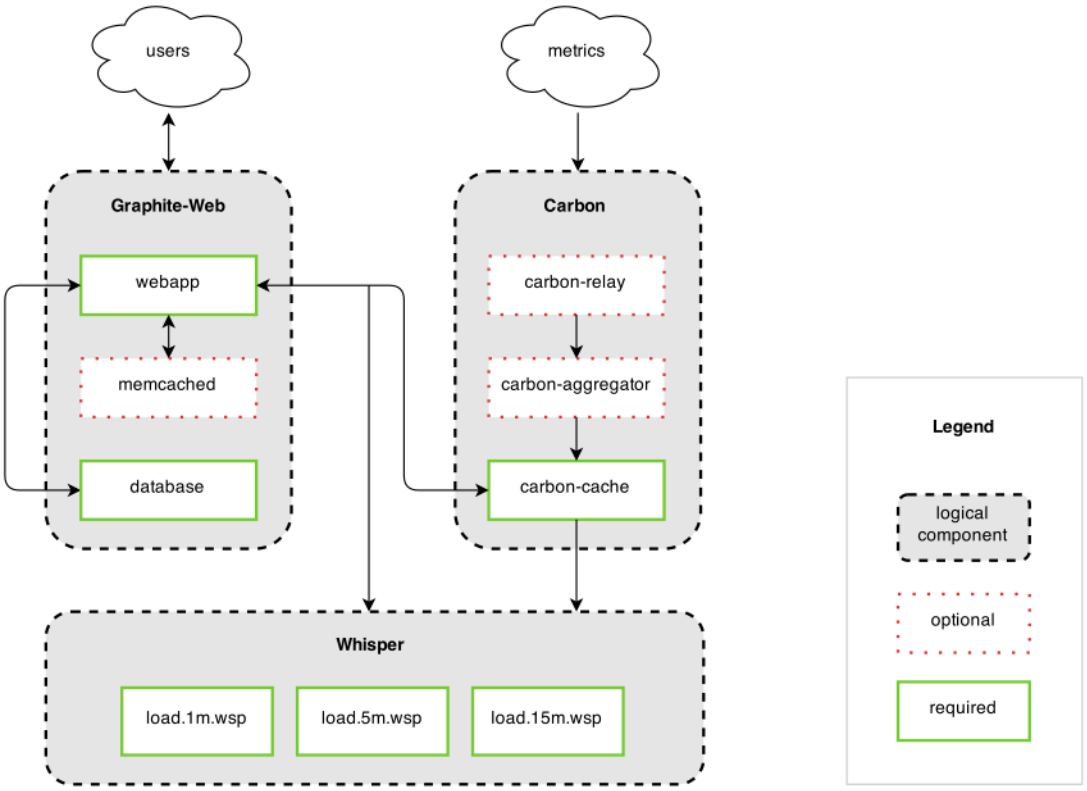
No. SIL Graphite is completely unrelated to this Graphite.

Is this Graphite related to the sourceforge project called graphite?

No. The sourceforge project called graphite is completely unrelated to this Graphite.

Is there a diagram of Graphite's architecture?

There sure is! Here it is:


users
metrics
Graphite-Web
webapp
memcached
database
Carbon
carbon-relay
carbon-aggregator
carbon-cache
Whisper
load.1m.wsp
load.5m.wsp
load.15m.wsp
Legend
logical component
optional
required

CHAPTER 3

Installing Graphite

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:

- a UNIX-like Operating System
- Python 2.7 or greater but *NOT Python 3*
- cairocffi
- Django 1.9
- django-tagging 0.4.3 (not *django-taggit* yet)
- pytz
- scandir
- fontconfig and at least one font package (a system package usually)
- A WSGI server and web server. Popular choices are:
  - Apache with mod_wsgi
  - gunicorn with nginx
  - uWSGI with nginx

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
- Dependent 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

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.

Default Installation Layout

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

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/`).

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

  - `ceres`

    Location for Ceres data files to be stored and read

- `webapp/`

  Graphite-web `PYTHONPATH`

  - `graphite/`

    Location of `local_settings.py`

  - `content/`

    Graphite-web static content directory

Installing Graphite

Several installation options exist:

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`
- Ceres: `git clone https://github.com/graphite-project/ceres.git`

Note: There currently is no tarball available for Ceres, it must be cloned from the Github project page

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, Whisper, and Ceres.

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 --install-data=/var/lib/graphite
```

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 --install-data=/var/lib/graphite
```

Installing From Pip

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

Note: In order to install Graphite-Web and Carbon, you must first install some development headers. In Debian-based distributions, this will require `apt-get install python-dev libcairo2-dev libffi-dev build-essential`, and in Red Hat-based distributions you will run `yum install python-devel cairo-devel libffi-devel`.

Installing in the Default Location

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

```
export PYTHONPATH="/opt/graphite/lib/:/opt/graphite/webapp/"
pip install --no-binary=:all: https://github.com/graphite-project/whisper/tarball/
↪master
pip install --no-binary=:all: https://github.com/graphite-project/carbon/tarball/
↪master
pip install --no-binary=:all: https://github.com/graphite-project/graphite-web/
↪tarball/master
```

Note: If your version of `pip` is < 7.0.0 then no need to use `--no-binary=:all:` parameter

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 https://github.com/graphite-project/carbon/tarball/master --install-
↪option="--prefix=/srv/graphite" --install-option="--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`:

```
pip install https://github.com/graphite-project/carbon/tarball/master --install-
↪option="--install-scripts=/usr/bin" --install-option="--install-lib=/usr/lib/
↪python2.6/site-packages" --install-option="--install-data=/var/lib/graphite"
```

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 https://github.com/graphite-project/graphite-web/tarball/master --install-
↪option="--prefix=/srv/graphite" --install-option="--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`:

```
pip install https://github.com/graphite-project/graphite-web/tarball/master --install-
↪option="--install-scripts=/usr/bin" --install-option="--install-lib=/usr/lib/
↪python2.6/site-packages" --install-option="--install-data=/var/lib/graphite"
```

Installing Ceres

Ceres is an alternative storage backend that some choose to use in place of the default Whisper backend.

```
pip install https://github.com/graphite-project/ceres/tarball/master
```

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 `django-admin.py` within a virtualenv requires using the full path of the virtualenv:

```
/path/to/env/bin/django-admin.py <command> --settings=graphite.settings
```

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.

Installing From Synthesize

Synthesize is a script dedicated to making Graphite easy to install. As of this writing, the default installation provides Graphite 0.9.15 for Ubuntu Linux 14.04 LTS with an experimental release candidate for tracking Graphite `HEAD`. Users may run the installation script manually, or they can choose to use the provided Vagrantfile.

For detailed instructions, please refer to the official project documentation on the Synthesize website.

Initial Configuration

Webapp Database Setup

You must tell Django to create the database tables used by the graphite webapp. This is very straight forward, especially if you are using the default SQLite setup.

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 instead of the old `DATABASE_*` format. Users must use the new specification to have a working database.

See the Django documentation for full documentation of the DATABASES setting.

Note: If you are using a custom database backend (other than SQLite) you must first create a $GRAPHITE_ROOT/webapp/graphite/local_settings.py file that overrides the database related settings from settings.py. Use $GRAPHITE_ROOT/webapp/graphite/local_settings.py.example as a template.

To set up a new database and create the initial schema, run:

```
PYTHONPATH=$GRAPHITE_ROOT/webapp django-admin.py migrate --settings=graphite.settings␣
↪--run-syncdb
```

If you are experiencing problems, uncomment the following line in /opt/graphite/webapp/graphite/local_settings.py:

```
# DEBUG = True
```

and review your webapp logs. If you're using the default graphite-example-vhost.conf, your logs will be found in /opt/graphite/storage/log/webapp/.

If you're using the default SQLite database, your webserver will need permissions to read and write to the database file. So, for example, if your webapp is running in Apache as the 'nobody' user, you will need to fix the permissions like this:

```
sudo chown nobody:nobody /opt/graphite/storage/graphite.db
```

Configuring The Webapp

Running the webapp with mod_wsgi as URL-prefixed application (Apache)

When using the new `URL_PREFIX` parameter in `local_settings.py` the `WSGIScriptAlias` setting must look like the following (e.g. URL_PREFIX="/graphite"):

```
WSGIScriptAlias /graphite /srv/graphite-web/conf/graphite.wsgi/graphite
```

The /graphite is needed for Django to create the correct URLs

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.

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.

General Settings

TIME_ZONE *Default: America/Chicago*

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

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

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.

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.

DEFAULT_CACHE_POLICY *Default: []*

Metric data and graphs are cached for one minute by default. If defined, DEFAULT_CACHE_POLICY is a list of tuples of minimum query time ranges mapped to the cache duration for the results. This allows for larger queries to be cached for longer periods of times. All times are in seconds. An example configuration:

```
DEFAULT_CACHE_POLICY = [(0, 60), # default is 60 seconds
                        (7200, 120), # >= 2 hour queries are cached 2 minutes
                        (21600, 180)] # >= 6 hour queries are cached 3 minutes
```

This will cache any queries between 0 seconds and 2 hours for 1 minute, any queries between 2 and 6 hours for 2 minutes, and anything greater than 6 hours for 3 minutes. If the policy is empty or undefined, everything will be cached for DEFAULT_CACHE_DURATION.

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, CERES_DIR, LOG_DIR, and INDEX_FILE default paths are referenced.

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

This directory doesn't even exist once you've installed graphite. It needs to be populated with the following command:

```
PYTHONPATH=$GRAPHITE_ROOT/webapp django-admin.py collectstatic --noinput --
↪settings=graphite.settings
```

This collects static files for graphite-web and external apps (namely, the Django admin app) and puts them in a directory that needs to be available under the `/static/` URL of your web server. To configure Apache:

```
Alias /static/ "/opt/graphite/static"
```

For Nginx:

```
location /static/ {
    alias /opt/graphite/static/;
}
```

Alternatively, static files can be served directly by the Graphite webapp if you install the `whitenoise` Python package.

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.

CERES_DIR *Default: /opt/graphite/storage/ceres* The location of Ceres 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, CERES_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 webapp.

Configure Webserver (Apache)

There is an example `example-graphite-vhost.conf` file in the examples directory of the graphite web source code. You can use this to configure apache. Different distributions have different ways of configuring Apache. Please refer to your distribution's documentation on the subject.

For example, Ubuntu uses `/etc/apache2/sites-available` and `sites-enabled/` to handle this (A symlink from `sites-enabled/` to `sites-available/` would be used after placing the file in sites-available/).

Others use an Include directive in the `httpd.conf` file like this:

```
# This goes in httpd.conf
Include /usr/local/apache2/conf/vhosts.d/*.conf
```

The configuration files must then all be added to `/usr/local/apache2/conf/vhosts.d/`. Still others may not help handle this at all and you must add the configuration to your http.conf file directly.

Graphite will be in the DocumentRoot of your webserver, and will not allow you to access plain-HTML in subdirectories without addition configuration. You may want to edit the `example-graphite-vhosts.conf` file to change port numbers or use additional `"SetHandler None"` directives to allow access to other directories.

Be sure to reload your Apache configuration by running `sudo /etc/init.d/apache2 reload` or `sudo /etc/init.d/httpd reload`.

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*

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.

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 alternately in `LDAP_URI`.

LDAP_PORT *Default: 389*

Set the LDAP server port here or alternately in `LDAP_URI`.

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).

LDAP_USER_DN_TEMPLATE: *Default: ''*

Instead of using a hardcoded username and password for the account that binds to the LDAP server you could use the credentials of the user that tries to log in to Graphite. This is the template that creates the full DN to bind with.

Other Authentications

USE_REMOTE_USER_AUTHENTICATION *Default: False*

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

REMOTE_USER_BACKEND *Default: "django.contrib.auth.middleware.RemoteUserMiddleware"*

Enables the use of an alternative remote authentication backend.

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.

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.

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 instead of the old `DATABASE_*` format. Users must use the new specification to have a working database.

See the Django documentation for full documentation of the DATABASES setting.

Note: Remember, setting up a new database requires running `PYTHONPATH=$GRAPHITE_ROOT/webapp django-admin.py migrate --settings=graphite.settings --run-syncdb` to create the initial schema.

Note: If you are using a custom database backend (other than SQLite) you must first create a $GRAPHITE_ROOT/webapp/graphite/local_settings.py file that overrides the database related settings from settings.py. Use $GRAPHITE_ROOT/webapp/graphite/local_settings.py.example as a template.

If you are experiencing problems, uncomment the following line in /opt/graphite/webapp/graphite/local_settings.py:

```
# DEBUG = True
```

and review your webapp logs. If you're using the default graphite-example-vhost.conf, your logs will be found in /opt/graphite/storage/log/webapp/.

If you're using the default SQLite database, your webserver will need permissions to read and write to the database file. So, for example, if your webapp is running in Apache as the 'nobody' user, you will need to fix the permissions like this:

```
sudo chown nobody:nobody /opt/graphite/storage/graphite.db
```

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 also be enabled and configured accordingly.

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.

CARBONLINK_HASHING_TYPE *Default: carbon_ch*

Possible values: carbon_ch, fnv1a_ch

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

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`.

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.

```
pushd /opt/graphite/conf
cp carbon.conf.example carbon.conf
cp storage-schemas.conf.example storage-schemas.conf
```

The example defaults are sane, but they may not meet your information resolution needs or storage limitations.

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.

---

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 `retentions` 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.

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.

If you want to change aggregation methods for existing data, be sure that you update the whisper files as well.

Example:

```
/opt/graphite/bin/whisper-set-aggregation-method.py /opt/graphite/storage/whisper/
↪test.wsp max
```

This example sets the aggregation for the test.wsp to max. (The location of the python script depends on your installation)

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.

aggregation-rules.conf

Aggregation rules allow you to add several metrics together as they 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'. Any metric that will arrive to `carbon-aggregator` will proceed to its output untouched unless it is overridden by some rule.

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.

Template components such as <env> will match everything up to the next dot. To match metric multiple components including the dots, use <<metric>> in the input template:

```
<env>.applications.<app>.all.<app_metric> (60) = sum <env>.applications.<app>.*.<<app_
↪metric>>
```

It is also possible to use regular expressions. Following the example above when using:

```
<env>.applications.<app>.<domain>.requests (60) = sum <env>.applications.<app>.
↪<domain>\d{2}.requests
```

You will end up with `prod.applications.apache.www.requests` instead of `prod.applications.apache.all.requests`.

Another common use pattern of `carbon-aggregator` is to aggregate several data points of the *same metric*. This could come in handy when you have got the same metric coming from several hosts, or when you are bound to send data more frequently than your shortest retention.

rewrite-rules.conf

Rewrite rules allow you to rewrite metric names using Python regular expressions. 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:

```
regex-pattern = replacement-text
```

This will capture any received metrics that match 'regex-pattern' and rewrite the matched portion of the text with 'replacement-text'. The 'regex-pattern' must be a valid Python regular expression, and the 'replacement-text' can be any value. You may also use capture groups:

```
^collectd\.([a-z0-9]+)\. = \1.system.
```

Which would result in:

```
collectd.prod.cpu-0.idle-time => prod.system.cpu-0.idle-item
```

rewrite-rules.conf consists of two sections, [pre] and [post]. The rules in the pre section are applied to metric names as soon as they are received. The post rules are applied after aggregation has taken place.

For example:

```
[post]
_sum$ =
_avg$ =
```

These rules would strip off a suffix of _sum or _avg from any metric names after aggregation.

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.

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.

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.

Windows Users

Despair Not! Even though running Graphite on Windows is completely unsupported (we fear that handling the escaping of paths in the regexes would result only in jibbering madness, and life is just too short; pull requests happily considered!), you are not completely out of luck. There are some solutions that make it easier for you to run a UNIX VM within your Windows box. The *Installing via Synthesize* article will help you set up a Vagrant VM that will run Graphite. In order to leverage this, you will need to install Vagrant.

CHAPTER 4

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`. As an installation grows, the `carbon-relay.py` and `carbon-aggregator.py` daemons can be introduced to distribute metrics load and perform custom aggregations, respectively.

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. This document gives a brief overview of what each daemon does and how you can use them to build a more sophisticated storage backend.

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. It also provides a query service for in-memory metric datapoints, used by the Graphite webapp to retrieve "hot data".

`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/somehost/something.wsp'
```

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.

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 `DESTINATIONS` 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 With `RELAY_METHOD = rules` set, pattern/servers tuples in this file define which metrics matching certain regex rules are forwarded to which hosts.

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 5

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:

- There are some tools and APIs which can help you get your data into Carbon.
- 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.

Existing tools and APIs

- *client daemons and tools*
- *client APIs*

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

If you use the OpenBSD implementation of `netcat`, please follow this example:

```
PORT=2003
SERVER=graphite.your.org
echo "local.random.diceroll 4 `date +%s`" | nc -q0 ${SERVER} ${PORT}
```

The `-q0` parameter instructs `nc` to close socket once data is sent. Without this option, some `nc` versions would keep the connection open.

If you use the GNU implementation of `netcat`, please follow this example:

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

The `-c` parameter instructs `nc` to close socket once data is sent. Without this option, `nc` will keep the connection open and won't end.

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!), and pickled it (if your client is running a more recent version of python than your server, you may need to specify the protocol) 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, protocol=2)
header = struct.pack("!L", len(payload))
message = header + payload
```

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

Using AMQP

When AMQP_METRIC_NAME_IN_BODY is set to True in your carbon.conf file, the data should be of the same format as the plaintext protocol, e.g. echo "local.random.diceroll 4 *date +%s*". When AMQP_METRIC_NAME_IN_BODY is set to False, you should omit 'local.random.diceroll'.

CHAPTER 6

Getting Your Data Into Graphite

The Basic Idea

Graphite is useful if you have some numeric values that change over time and you want to graph them. Basically you write a program to collect these numeric values which then sends them to graphite's backend, Carbon.

Step 1 - Plan a Naming Hierarchy

Everything stored in graphite has a path with components delimited by dots. So for example, website.orbitz.bookings.air or something like that would represent the number of air bookings on orbitz. Before producing your data you need to decide what your naming scheme will be. In a path such as "foo.bar.baz", each thing surrounded by dots is called a path component. So "foo" is a path component, as well as "bar", etc.

Each path component should have a clear and well-defined purpose. Volatile path components should be kept as deep into the hierarchy as possible.

Matt _Aimonetti has a reasonably sane post describing the organization of your namespace.

Step 2 - Configure your Data Retention

Graphite is built on fixed-size databases (see *Whisper.*) so we have to configure in advance how much data we intend to store and at what level of precision. For instance you could store your data with 1-minute precision (meaning you will have one data point for each minute) for say 2 hours. Additionally you could store your data with 10-minute precision for 2 weeks, etc. The idea is that the storage cost is determined by the number of data points you want to store, the less fine your precision, the more time you can cover with fewer points. To determine the best retention configuration, you must answer all of the following questions.

1. How often can you produce your data?
2. What is the finest precision you will require?

3. How far back will you need to look at that level of precision?
4. What is the coarsest precision you can use?
5. How far back would you ever need to see data? (yes it has to be finite, and determined ahead of time)

Once you have picked your naming scheme and answered all of the retention questions, you need to create a schema by creating/editing the `/opt/graphite/conf/storage-schemas.conf` file.

The format of the schemas file is easiest to demonstrate with an example. Let's say we've written a script to collect system load data from various servers, the naming scheme will be like so:

```
servers.HOSTNAME.METRIC
```

Where HOSTNAME will be the server's hostname and METRIC will be something like cpu_load, mem_usage, open_files, etc. Also let's say we want to store this data with minutely precision for 30 days, then at 15 minute precision for 10 years.

For details of implementing your schema, see the *Configuring Carbon* document.

Basically, when carbon receives a metric, it determines where on the filesystem the whisper data file should be for that metric. If the data file does not exist, carbon knows it has to create it, but since whisper is a fixed size database, some parameters must be determined at the time of file creation (this is the reason we're making a schema). Carbon looks at the schemas file, and in order of priority (highest to lowest) looks for the first schema whose pattern matches the metric name. If no schema matches the default schema (2 hours of minutely data) is used. Once the appropriate schema is determined, carbon uses the retention configuration for the schema to create the whisper data file appropriately.

Step 3 - Understanding the Graphite Message Format

Graphite understands messages with this format:

```
metric_path value timestamp\n
```

`metric_path` is the metric namespace that you want to populate.

`value` is the value that you want to assign to the metric at this time.

`timestamp` is the unix epoch time.

A simple example of doing this from the unix terminal would look like this:

```
echo "test.bash.stats 42 `date +%s`" | nc graphite.example.com 2003
```

There are many tools that interact with Graphite. See the *Tools* page for some choices of tools that may be used to feed Graphite.

CHAPTER 7

Administering Carbon

Starting Carbon

Carbon can be started with the `carbon-cache.py` script:

```
/opt/graphite/bin/carbon-cache.py start
```

This starts the main Carbon daemon in the background. Now is a good time to check the logs, located in `/opt/graphite/storage/log/carbon-cache/` for any errors.

CHAPTER 8

Administering The Webapp

CHAPTER 9

Using The Composer

...

CHAPTER 10

The Render URL API

The graphite webapp provides a `/render` endpoint for generating graphs and retrieving 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`

Graphing Metrics

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

target

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

Paths and Wildcards

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

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

Asterisk The asterisk (`*`) matches zero or more characters. It is non-greedy, so you can have more than one within a single path element.

Example: `servers.ix*ehssvc*v.cpu.total.*` will return all total CPU metrics for all servers matching the given name pattern.

Character list or range Characters in square brackets (`[...]`) specify a single character position in the path string, and match if the character in that position matches one of the characters in the list or range.

A character range is indicated by 2 characters separated by a dash (`-`), and means that any character between those 2 characters (inclusive) will match. More than one range can be included within the square brackets, e.g. `foo[a-z0-9]bar` will match `foopbar`, `foo7bar` etc..

If the characters cannot be read as a range, they are treated as a list - any character in the list will match, e.g. `foo[bc]ar` will match `foobar` and `foocar`. If you want to include a dash (`-`) in your list, put it at the beginning or end, so it's not interpreted as a range.

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

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

Examples

This will draw one or more metrics

Example:

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

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

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

Now let's say you have 10 servers.

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

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

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

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

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

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

from / until

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

There are multiple formats for these functions:

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

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

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

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

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

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

Examples:

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

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

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

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

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

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

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

template

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

Examples

Example:

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

Default values for the template variables can also be provided:

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

Positional arguments can be used instead of named ones:

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

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

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

Data Display Formats

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

format

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

Examples:

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

png

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

raw

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

```
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, null],
          "options": {},
          "valuesPerPoint": 1
        }
      ],
      "y": {
        "labelValues": [0, 0.25, 0.5, 0.75, 1.0],
        "top": 1.0,
        "labels": ["0 ", "0.25 ", "0.50 ", "0.75 ", "1.00  "],
        "step": 0.25,
        "bottom": 0
      },
      "x": {
        "start": 1335398400,
        "end": 1335423600
      },
      "font": {
        "bold": false,
        "name": "Sans",
        "italic": false,
        "size": 10
      },
      "options": {
        "lineWidth": 1.2
      }
    }
  ]]>
</script>
```


pdf

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

dygraph

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

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

rickshaw

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

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

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],
  }
]
```

rawData

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

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

Example:

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

Returns the following text:

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

Graph Parameters

areaAlpha

Default: 1.0

Takes a floating point number between 0.0 and 1.0

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

areaMode

Default: none

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

`none` Disables areaMode

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

`all` Fills the areas under each target

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

bgcolor

Default: 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[AA] where RR, GG, and BB are 2-digit hex vaules for red, green and blue, respectively. AA is an optional addition describing the opacity ("alpha"). Where FF is fully opaque, 00 fully transparent.

Examples:

```
&bgcolor=blue
&bgcolor=2222FF
&bgcolor=5522FF60
```

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)

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. Any RGB value can also have an optional transparency (00 being fully transparent, FF being opaque), as shown in the second example.

Example:

```
&colorList=green,yellow,orange,red,purple,DECAFF
&colorList=FF000055,00FF00AA,DECAFFEF
```

drawNullAsZero

Default: false

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

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.

fontBold

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

If set to true, makes the font bold.

Example:

```
&fontBold=true
```

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
```

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
```

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
```

format

See: *Data Display Formats*

from

See: *from / until*

graphOnly

Default: False

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

graphType

Default: line

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

line A line graph displaying metrics as lines over time

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

hideLegend

Default: <unset>

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

hideNullFromLegend

Default: False

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

Example:

```
&hideNullFromLegend=true
```

hideAxes

Default: False

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

Example:

```
&hideAxes=true
```

hideXAxis

Default: False

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

hideYAxis

Default: False

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

hideGrid

Default: False

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

Example:

```
&hideGrid=true
```

height

Default: 250

Sets the height of the generated graph image in pixels.

See also: *width*

Example:

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

jsonp

Default: <unset>

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

leftColor

Default: color chosen from colorList

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

leftDashed

Default: False

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

leftWidth

Default: value of the parameter lineWidth

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

lineMode

Default: slope

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

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

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

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

Example:

```
&lineMode=staircase
```

lineWidth

Default: 1.2

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

Example:

```
&lineWidth=2
```

logBase

Default: <unset>

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

localOnly

Default: False

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

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
```

margin

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

Example:

```
&margin=20
```

max

Deprecated since version 0.9.0: See *yMax*

maxDataPoints

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

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

minorGridLineColor

Default: 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
```

minorY

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

Example:

```
&minorY=3
```

min

Deprecated since version 0.9.0: See *yMin*

minXStep

Default: 1

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

noCache

Default: False

Set to disable caching of rendered images

noNullPoints

Default: False

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

pickle

Deprecated since version 0.9.10: See *Data Display Formats*

pieLabels

Default: horizontal

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

`horizontal` Labels are oriented horizontally within each slice

`rotated` Labels are oriented radially within each slice

pieMode

Default: average

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

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

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

`minimum` THe minimum of non-null points in the series

rightColor

Default: color chosen from colorList

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

rightDashed

Default: False

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

rightWidth

Default: value of the parameter lineWidth

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

template

Default: default

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

Example:

```
&template=plain
```

thickness

Deprecated since version 0.9.0: See: *lineWidth*

title

Default: <unset>

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

Example:

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

tz

Default: The timezone specified in 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`.

uniqueLegend

Default: False

Display only unique legend items, removing any duplicates

until

See: *from / until*

valueLabels

Default: percent

Determines how slice labels are rendered within a pie chart.

`none` Slice labels are not shown

`numbers` Slice labels are reported with the original values

`percent` Slice labels are reported as a percent of the whole

valueLabelsColor

Default: black

Color used to draw slice labels within a pie chart.

valueLabelsMin

Default: 5

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

vtitle

Default: <unset>

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

Example:

```
&vtitle=Threads
```

vtitleRight

Default: <unset>

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

width

Default: 330

Sets the width of the generated graph image in pixels.

See also: *height*

Example:

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

xFormat

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

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

yAxisSide

Default: left

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

yDivisors

Default: 4,5,6

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

yLimit

Reserved for future use See: *yMax*

yLimitLeft

Reserved for future use See: *yMaxLeft*

yLimitRight

Reserved for future use See: *yMaxRight*

yMin

Default: The lowest value of any of the series displayed

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

Example:

```
&yMin=0
```

yMax

Default: The highest value of any of the series displayed

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

Example:

```
&yMax=0.2345
```

yMaxLeft

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

yMaxRight

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

yMinLeft

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

yMinRight

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

yStep

Default: Calculated automatically

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

yStepLeft

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

yStepRight

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

yUnitSystem

Default: si

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

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

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

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

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

`none` Dont compact values, display the raw number

CHAPTER 11

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*.

Usage

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

List of functions

`absolute`(*seriesList*)

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

Example:

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

`aggregateLine`(*seriesList*, *func='avg'*)

Takes a metric or wildcard seriesList and draws a horizontal line based on the function applied to each series.

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

Example:

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

alias (*seriesList*, *newName*)

Takes one metric or a wildcard seriesList and a string in quotes. Prints the string instead of the metric name in the legend.

```
&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.

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

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

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

Example:

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

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

Additional Examples:

Given keys of

- *stats.counts.haproxy.web.2XX*
- *stats.counts.haproxy.web.3XX*
- *stats.counts.haproxy.web.5XX*
- *stats.counts.haproxy.microservice.2XX*
- *stats.counts.haproxy.microservice.3XX*
- *stats.counts.haproxy.microservice.5XX*

The following will return the rate of 5XX's per service:

```
applyByNode(stats.counts.haproxy.*.*XX, 3, "asPercent(%.2XX, sumSeries(%.*XX))", "
↪%.pct_5XX")
```

The output series would have keys *stats.counts.haproxy.web.pct_5XX* and *stats.counts.haproxy.microservice.pct_5XX*.

areaBetween(*seriesList*)

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

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

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

Example:

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

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

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

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

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

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

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

Example:

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

averageAbove(*seriesList*, *n*)

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

Example:

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

Draws the servers with average values above 25.

averageBelow(*seriesList*, *n*)

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

Example:

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

Draws the servers with average values below 25.

averageOutsidePercentile(*seriesList*, *n*)
Removes functions lying inside an average percentile interval

averageSeries(**seriesLists*)
Short Alias: avg()

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

Example:

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

averageSeriesWithWildcards(*seriesList*, **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(host.*.cpu-system.value)`

cactiStyle(*seriesList*, *system=None*, *units=None*)
Takes a series list and modifies the aliases to provide column aligned output with Current, Max, and Min values in the style of cacti. Optionally takes a "system" value to apply unit formatting in the same style as the Y-axis, or a "unit" string to append an arbitrary unit suffix.

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

A possible value for `system` is `si`, which would express your values in multiples of a thousand. A second option is to use `binary` which will instead express your values in multiples of 1024 (useful for network devices).

Column alignment of the Current, Max, Min values works under two conditions: you use a monospace font such as terminus and use a single cactiStyle call, as separate cactiStyle calls are not aware of each other. In case you have different targets for which you would like to have cactiStyle to line up, you can use `group()` to combine them before applying cactiStyle, such as:

```
&target=cactiStyle(group(metricA,metricB))
```

changed(*seriesList*)
Takes one metric or a wildcard seriesList. Output 1 when the value changed, 0 when null or the same

Example:

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

color(*seriesList*, *theColor*)
Assigns the given color to the seriesList

Example:

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

consolidateBy(*seriesList*, *consolidationFunc*)
Takes one metric or a wildcard seriesList and a consolidation function name.

Valid function names are 'sum', 'average', 'min', and 'max'.

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

```
&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*)
Takes one metric or a wildcard seriesList.

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

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

```
&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)
```

delay(*seriesList*, *steps*)

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

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

Example:

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

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

derivative(*seriesList*)

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

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

Example:

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

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

diffSeries(**seriesLists*)

Subtracts series 2 through n from series 1.

Example:

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

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

Example:

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

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

divideSeries(*dividendSeriesList*, *divisorSeries*)

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")
```

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

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

The Constant is calculated as:

constant = 2 / (windowSize + 1)

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

Example:

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

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

Example:

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

Draws a 0 line when server metric does not exist.

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.load5),...
```

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

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

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

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

highestAverage (*seriesList*, *n*)
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=').

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

Example:

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

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

interpolate (*seriesList*, *limit=inf*)
Takes one metric or a wildcard seriesList, and optionally a limit to the number of 'None' values to skip over.

Continues the line with the last received value when gaps ('None' values) appear in your data, rather than breaking your line.

Example:

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

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

Example:

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

isNonNull (*seriesList*)
Takes a metric or wildcard seriesList and counts up the number of non-null values. This is useful for understanding the number of metrics that have data at a given point in time (i.e. to count which servers are alive).

Example:

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

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

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

Example:

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

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

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

limit (*seriesList*, *n*)
Takes one metric or a wildcard seriesList followed by an integer N.

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

Example:

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

Draws only the first 5 instance's memory free.

lineWidth (*seriesList*, *width*)
Takes one metric or a wildcard seriesList, followed by a float F.

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

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

Example:

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

linearRegression (*seriesList*, *startSourceAt=None*, *endSourceAt=None*)
Graphs the liner regression function by least squares method.

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

Example:

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

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

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

Example:

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

lowestAverage (*seriesList*, *n*)
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.

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

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

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

```
mapSeries(servers.*.cpu.*,1) =>

  [
```

```
    servers.server1.cpu.*,
    servers.server2.cpu.*,
    ...
    servers.serverN.cpu.*
  ]
```

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

Example:

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

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

Example:

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

This would only display interfaces which 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.

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

Example:

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

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

mostDeviant (*seriesList*, *n*)

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

Example:

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

Draws the 5 instances furthest from the average memory free.

movingAverage (*seriesList*, *windowSize*)

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

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

Example:

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

movingMax (*seriesList*, *windowSize*)

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

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

Example:

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

movingMedian (*seriesList*, *windowSize*)

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

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

Example:

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

movingMin (*seriesList*, *windowSize*)

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

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

Example:

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

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

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

Example:

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

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

Example:

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

multiplySeriesWithWildcards (*seriesList*, **position*)
Call multiplySeries after inserting wildcards at the given position(s).

Example:

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

This would be the equivalent of

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

nPercentile (*seriesList*, *n*)
Returns n-percent of each series in the seriesList.

nonNegativeDerivative (*seriesList*, *maxValue=None*)
Same as the derivative function above, but ignores datapoints that trend down. Useful for counters that increase for a long time, then wrap or reset. (Such as if a network interface is destroyed and recreated by unloading and re-loading a kernel module, common with USB / WiFi cards.

Example:

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

offset (*seriesList*, *factor*)
Takes one metric or a wildcard seriesList followed by a constant, and adds the constant to each datapoint.

Example:

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

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

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

An example use case is for comparing different round trip time results. When measuring RTT (like pinging a server), different devices may come back with consistently different results due to network latency which will

be different depending on how many network hops between the probe and the device. To compare different devices in the same graph, the network latency to each has to be factored out of the results. This is a shortcut that takes the fastest response (lowest number in the series) and sets that to zero and then offsets all of the other datapoints in that series by that amount. This makes the assumption that the lowest response is the fastest the device can respond, of course the more datapoints that are in the series the more accurate this assumption is.

Example:

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

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

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

Example:

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

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

percentileOfSeries(*seriesList*, *n*, *interpolate=False*)

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

pow(*seriesList*, *factor*)

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

Example:

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

powSeries(**seriesLists*)

Takes two or more series and pows their points. A constant line may be used.

Example:

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

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

Short Alias: randomWalk()

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

Example:

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

This would create a series named "The.time.series" that contains points where x(t) == x(t-1)+random()-0.5, and x(0) == 0. Accepts optional second argument as 'step' parameter (default step is 60 sec)

rangeOfSeries(**seriesLists*)

Takes a wildcard seriesList. Distills down a set of inputs into the range of the series

Example:

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

reduceSeries(*seriesLists*, *reduceFunction*, *reduceNode*, **reduceMatchers*)

Short form: `reduce()`

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

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

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

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

Assume that metrics in the form below exist:

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

To get the percentage of disk used for each server:

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

  alias(asPercent(servers.server1.disk.bytes_used,servers.server1.disk.total_
↪bytes),"servers.server1.disk.reduce.asPercent"),
  alias(asPercent(servers.server2.disk.bytes_used,servers.server2.disk.total_
↪bytes),"servers.server2.disk.reduce.asPercent"),
  alias(asPercent(servers.server3.disk.bytes_used,servers.server3.disk.total_
↪bytes),"servers.server3.disk.reduce.asPercent"),
  ...
  alias(asPercent(servers.serverN.disk.bytes_used,servers.serverN.disk.total_
↪bytes),"servers.serverN.disk.reduce.asPercent")
```

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

```
servers.server1.disk.reduce.asPercent
servers.server2.disk.reduce.asPercent
servers.server3.disk.reduce.asPercent
...
servers.serverN.disk.reduce.asPercent
```

See also:

mapSeries()

removeAbovePercentile(*seriesList*, *n*)

Removes data above the nth percentile from the series or list of series provided. Values above this percentile are assigned a value of None.

removeAboveValue(*seriesList*, *n*)

Removes data above the given threshold from the series or list of series provided. Values above this threshold

are assigned a value of None.

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

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

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

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

Example:

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

Draws only live servers with not empty data.

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

Example:

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

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

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

secondYAxis (*seriesList*)
Graph the series on the secondary Y axis.

sinFunction (*name*, *amplitude=1*, *step=60*)
Short Alias: sin()

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

Example:

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

This would create a series named "The.time.series" that contains sin(x)*2. Accepts optional second argument as 'amplitude' parameter (default amplitude is 1) Accepts optional third argument as 'step' parameter (default step is 60 sec)

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)
```

sortByName(*seriesList*, *natural=False*)
Takes one metric or a wildcard seriesList. Sorts the list of metrics by the metric name using either alphabetical order or natural sorting. Natural sorting allows names containing numbers to be sorted more naturally, e.g: - Alphabetical sorting: server1, server11, server12, server2 - Natural sorting: server1, server2, server11, server12

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

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

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

Example:

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

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

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

Example:

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

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

Example:

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

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

Example:

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

substr (*seriesList*, *start=0*, *stop=0*)

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

Example:

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

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

sumSeries (**seriesLists*)

Short form: sum()

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

Example:

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

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

sumSeriesWithWildcards (*seriesList*, **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-[0-7].cpu-user.value)&target=sumSeries(host.cpu-[0-7].cpu-system.value)`

summarize (*seriesList*, *intervalString*, *func='sum'*, *alignToFrom=False*)

Summarize the data into interval buckets of a certain size.

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

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

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

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

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

Example:

```
&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*, *step=60*)

Short Alias: time()

Just returns the timestamp for each X value. T

Example:

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

This would create a series named "The.time.series" that contains in Y the same value (in seconds) as X. Accepts optional second argument as 'step' parameter (default step is 60 sec)

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

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

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

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

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

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

Example:

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

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

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

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

Example:

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

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

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

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

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

Example:

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

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

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

Example:

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

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

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

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

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

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

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

Takes a timestamp string ts.

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

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

Example:

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

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

Takes a series of average values and a series of weights and produces a weighted average for all values. The corresponding values should share one or more zero-indexed nodes.

Example:

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

CHAPTER 12

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.

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.

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.

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*.

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.

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.

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.

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*.

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*.

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.

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.

Opening a Dashboard

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

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").

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.

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.

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.

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.

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 *yDivisors*)
- 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.

Other Operations on the Graph

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

Breakout Creates new graphs for each of the metrics in the graph, adds them to the dashboard, and removes the original.

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.

Other Global Menu Options

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.

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...

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).

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.

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.

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.

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.

Changing Default Graph Parameters

By default, graphs are generated with a standard render template. If you find yourself applying *Render Options* to each and every graph you create, then you can select *Edit Default Parameters* in the *Graphs* menu to automatically handle that. These parameters are saved with the dashboard and persisted in a cookie.

The format is as a set of key-value pairs separated by ampersands, like a query string. The keys and values come from *The Render URL API* and they're all available. For example:

```
drawNullAsZero=true&graphOnly=true
```

Any new graphs created after saving that as the default graph parameters would have unreported metrics graphed as zeroes and omit the grid lines.

CHAPTER 13

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.

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'
```

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 its longest archive.

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

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 highest 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. If you are in need for aggregation of the highest resolution points, please consider using *carbon-aggregator* for that purpose.

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.

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).

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.

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.

Database Format

Whisper-File	*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 14

The Ceres Database

Ceres is a time-series database format intended to replace Whisper as the default storage format for Graphite. In contrast with Whisper, Ceres is not a fixed-size database and is designed to better support sparse data of arbitrary fixed-size resolutions. This allows Graphite to distribute individual time-series across multiple servers or mounts.

Ceres is not actively developped at the moment. For alternatives to whisper look at *alternative storage backends*.

Storage Overview

Ceres databases are comprised of a single tree contained within a single path on disk that stores all metrics in nesting directories as nodes.

A Ceres node represents a single time-series metric, and is composed of at least two data files. A slice to store all data points, and an arbitrary key-value metadata file. The minimum required metadata a node needs is a `'timeStep'`. This setting is the finest resolution that can be used for writing. A Ceres node however can contain and read data with other, less-precise values in its underlying slice data.

Other metadata keys that may be set for compatibility with Graphite are `'retentions'`, `'xFilesFacter'`, and `'aggregationMethod'`.

A Ceres slice contains the actual data points in a file. The only other information a slice holds is the timestamp of the oldest data point, and the resolution. Both of which are encoded as part of its filename in the format `timestamp@resolution`.

Data points in Ceres are stored on-disk as a contiguous list of big-endian double-precision floats. The timestamp of a datapoint is not stored with the value, rather it is calculated by using the timestamp of the slice plus the index offset of the value multiplied by the resolution.

The timestamp is the number of 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'
```

Slices: Precision and Fragmentation

Ceres databases contain one or more slices, each with a specific data resolution and a timestamp to mark the beginning of the slice. Slices are ordered from the most recent timestamp to the oldest timestamp. Resolution of data is not considered when reading from a slice, only that when writing a slice with the finest precision configured for the node exists.

Gaps in data are handled in Ceres by padding slices with null datapoints. If the slice gap however is too big, then a new slice is instead created. If a Ceres node accumulates too many slices, read performance can suffer. This can be caused by intermittently reported data. To mitigate slice fragmentation there is a tolerance for how much space can be wasted within a slice file to avoid creating a new one. That tolerance level is determined by `'MAX_SLICE_GAP'`, which is the number of consecutive null datapoints allowed in a slice file.

If set very low, Ceres will waste less of the tiny bit disk space that this feature wastes, but then will be prone to performance problems caused by slice fragmentation, which can be pretty severe.

If set really high, Ceres will waste a bit more disk space. Although each null datapoint wastes 8 bytes, you must keep in mind your filesystem's block size. If you suffer slice fragmentation issues, you should increase this or defrag the data more often. However you should not set it to be huge because then if a large but allowed gap occurs it has to get filled in, which means instead of a simple 8-byte write to a new file we could end up doing an `(8 * MAX_SLICE_GAP)`-byte write to the latest slice.

Rollup Aggregation

Expected features such as roll-up aggregation and data expiration are not provided by Ceres itself, but instead are implemented as maintenance plugins.

Such a rollup plugin exists in Ceres that aggregates data points in a way that is similar behavior of Whisper archives. Where multiple data points are collapsed and written to a lower precision slice, and data points outside of the set slice retentions are trimmed. By default, an average function is used, however alternative methods can be chosen by changing the metadata.

Retrieval Behavior

When data is retrieved (scoped by a time range), the first slice which has data within the requested interval is used. If the time period overlaps a slice boundary, then both slices are read, with their values joined together. Any missing data between them are filled with null data points.

There is currently no support in Ceres for handling slices with mixed resolutions in the same way that is done with Whisper archives.

Database Format

CeresSlice	*Data*	
	Data	*Point+*

Data types in Python's struct format:

Point	`!d`

Metadata for Ceres is stored in JSON format:

{"retentions": [[30, 1440]], "timeStep": 30, "xFilesFactor": 0.5, "aggregationMethod": "average"}

CHAPTER 15

Alternative storage finders

Built-in finders

The default graphite setup consists of:

- A Whisper database
- A carbon daemon writing data to the database
- Graphite-web reading and graphing data from the database

It is possible to use an alternate storage layer than the default, Whisper, in order to accommodate specific needs. The setup above would become:

- An alternative database
- A carbon daemon or alternative daemon for writing to the database
- A custom *storage finder* for reading the data in graphite-web

This section aims at documenting the last item: configuring graphite-web to read data from a custom storage layer.

This can be done via the `STORAGE_FINDERS` setting. This setting is a list of paths to finder implementations. Its default value is:

```
STORAGE_FINDERS = (
    'graphite.finders.standard.StandardFinder',
)
```

The default finder reads data from a Whisper database.

An alternative finder for the experimental Ceres database is available:

```
STORAGE_FINDERS = (
    'graphite.finders.ceres.CeresFinder',
)
```

The setting supports multiple values, meaning you can read data from both a Whisper database and a Ceres database:

```
STORAGE_FINDERS = (
    'graphite.finders.standard.StandardFinder',
    'graphite.finders.ceres.CeresFinder',
)
```

Custom finders

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

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

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

```
from graphite.node import LeafNode, BranchNode

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

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

```
from graphite.intervals import IntervalSet, Interval

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

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

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

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

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

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

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

Installing custom finders

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

- Cyanite finder
- BigGraphite finder
- KairosDB finder

CHAPTER 16

Graphite Events

Graphite is well known for storing simple key/value metrics using the Whisper time-series database on-disk format. What is not well known about Graphite is that it also ships with a feature known as **Events** that supports a richer form of metrics storage suitable for irregular events often associated with metadata.

Examples of data appropriate for this storage format include releases, commits, application exceptions, and state changes where you may wish to track or correlate the event with traditional time-series activity.

Database Storage

As Whisper was designed to hold simple time-series data (metric key, value, and timestamp), it's altogether unsuitable for storing rich metric data such as events. Many users continue to store simple event-type data (e.g. releases, state changes, etc) in Whisper by encoding its meaning within the metric namespace and rendering them as a vertical line with Graphite's drawAsInfinite function.

However, taking advantage of this pattern typically requires the use of wildcards across a significant number of these singleton metric files and directories, which can cause a significant performance hit on the server and result in a poor experience for users. To accommodate this more sophisticated use case, Graphite's webapp database was extended to support this new metric type.

Note: Events require Graphite webapp version 0.9.9 or newer.

Adding Events

Events can be submitted via HTTP POST using command-line tools such as `curl` or with a variety of HTTP programming libraries. The JSON format is simple and predictable.

```
$ curl -X POST "http://graphite/events/"
    -d '{ "what": "Event - deploy", "tags": ["deploy"], "when": 1467844481,
    "data": "deploy of master branch happened at Wed Jul  6 22:34:41 UTC 2016" }'
```

`when` is an optional key which is set to the current Unix timestamp if `when` is not set.

Note: Prior to 0.10.0, the value of `tags` is a string, with multiple tags being separated by a space.

Querying Events

Graphite allows you to query for tags associated with events. You can search for a single tag string, a combination of space-delimited tags, or a simple `*` wildcard using the events function.

```
$ curl -s "http://graphite/render/?target=events('exception')&format=json" | json_pp

[
   {
      "target" : "events(exception)",
      "datapoints" : [
         [
            1,
            1388966651
         ],
         [
            3,
            1388966652
         ]
      ]
   }
]
```

It's also possible to dump the raw events using the API.

```
$ curl -s "http://graphite/events/get_data?tags=deploy&from=-3hours&until=now" | json_
↪pp

[
   {
      "when" : 1392046352,
      "tags" : ["deploy"],
      "data" : "deploy of master branch happened at Fri Jan 3 22:34:41 UTC 2014",
      "id" : 2,
      "what" : "Event - deploy"
   },
   {
      "id" : 3,
      "what" : "Event - deploy",
      "when" : 1392046661,
      "tags" : ["deploy"],
      "data" : "deploy of master branch happened at Fri Jan 3 22:34:41 UTC 2014"
   }
]
```

The `set` parameter accepts an optional `union` or `intersection` argument to determine the behavior for filtering sets of tags (i.e. inclusive or exclusive). By default, Graphite uses a "lazy union" that will return any matching events for a given tag in a list of tags. This behavior is not intuitive and will therefore be deprecated in a future release.

Managing Events in the Admin UI

Events can be managed using the Graphite administration module. This is particularly handy for deleting a large number of events at once, although it also supports adding and editing individual events.

CHAPTER 17

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 hierarchical 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 dependent and can be found by executing (with Python 2.6 or later):: python -c 'import sys; print sys.float_info'

CHAPTER 18

Tools That Work With Graphite

Collection

Brubeck A statsd-compatible stats aggregator written in C.

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

Carbonator Windows Service Simple lightweight Windows Service that collects Performance Counter metrics and sends them over to the Graphite server. Configured via .NET xml application configuration.

collectd 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`.

Collectl 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.

Diamond 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.

Ganglia 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.

graphite-pollers A collection of scripts that shovel data into Graphite including a multi-threaded SNMP poller for network interface IF-MIB statistics and another which pulls linux network stack data from files in /proc/net. Add to cron and go.

Graphite PowerShell Functions A group of functions that can be used to collect Windows Performance Counters and send them over to the Graphite server. The main function can be run as a Windows service, and everything is configurable via an XML file.

HoardD 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.

Host sFlow 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.

jmx2graphite The easiest way to poll JMX metrics and write them to Graphite. This tool runs as a Docker container, polling your JMX every X seconds and writing the metrics to Graphite. Requires a minimum of configuration to get started.

jmxtrans 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.

Logster 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.

metrics-sampler 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.

Sensu A monitoring framework that can route metrics to Graphite. Servers subscribe to sets of checks, so getting metrics from a new server to Graphite is as simple as installing the Sensu client and subscribing.

snort2graphite Snort IDS/IPS can be configured to generate a rich set of metrics about network traffic. Presently there are more than 130 metrics available. Snort2graphite will pick up the most recent data from your snort.stats file and send all the metrics into Graphite.

SqlToGraphite An agent for windows written in .net to collect metrics using plugins (WMI, SQL Server, Oracle) by polling an endpoint with a SQL query and pushing the results into graphite. It uses either a local or a centralised configuration over HTTP.

SSC Serv A Windows service (agent) which periodically publishes system metrics, for example CPU, memory and disk usage. It can store data in Graphite using a naming schema that's identical to that used by collectd.

Forwarding

Backstop 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.

carbon-c-relay Enhanced C implementation of Carbon relay, aggregator and rewriter.

carbon-relay-ng Fast carbon relay+aggregator with admin interfaces for making changes online - production ready.

Evenflow 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.

Grafsy Very light caching proxy for graphite metrics with additional features:

- Caching metrics in case of outage and sending them later
- Validation of metrics
- Aggregating of metrics, including SUM and AVG functions
- Much more

Graphite-Newrelic Get your graphite data into New Relic via a New Relic Platform plugin.

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

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

Graphout A Node.js application that lets you forward Graphite based queries (using the render API) out to different external services. There are built in modules for Zabbix and CloudWatch. Custom modules are very easy to write.

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

Gruffalo An asynchronous Netty based graphite proxy, for large scale installations. It protects Graphite from the herds of clients by minimizing context switches and interrupts; by batching and aggregating metrics. Gruffalo also allows you to replicate metrics between Graphite installations for DR scenarios, for example.

Ledbetter 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.

pipe-to-graphite A small shell script that makes it easy to report the output of any other cli program to Graphite.

Polymur A fast relay and HTTPS forwarder toolset written in Go.

statsd 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/>

Visualization

Charcoal 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.

Descartes 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.

Dusk 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.

Firefly A web application aimed at powerful, flexible time series graphing for web developers.

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

Giraffe 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.

Grafana A general purpose graphite dashboard replacement with feature rich graph editing and dashboard creation interface. It contains a unique Graphite target parser that enables easy metric and function editing. Fast client side rendering (even over large time ranges) using Flot with a multitude of display options (Multiple Y-axis,

Bars, Lines, Points, smart Y-axis formats and much more). Click and drag selection rectangle to zoom in on any graph.

Graphene 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.

graphite-dashboardcli A tool for manage graphite dashboards from command line:

- importExport dashboards fromto Graphite servers
- synchronize dashboards between multiple Graphite servers
- keep dashboards in YAML format

Graphite-Tattle A self-service dashboard frontend for Graphite and Ganglia.

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

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

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

Graphsky A 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.

Graph-Explorer 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.

Graph-Index An index of graphs for Diamond.

Hubot 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.

Leonardo A Graphite dashboard inspired by Gdash. It's written in Python using the Flask framework. The interface is built with Bootstrap. The graphs and dashboards are configured through the YAML files.

Orion A powerful tool to create, view and manage dashboards for your Graphite data. It allows easy implementation of custom authentication to manage access to the dashboard.

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

Targets-io A dashboard application for organizing, analyzing, benchmarking and reporting of performance test results. All performance test metrics are stored in Graphite and can be benchmarked between test runs, providing automated feedback on the performance of an application.

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

Terphite Terminal tool for displaying Graphite metrics.

Tessera A flexible front-end for creating dashboards with a wide variety of data presentations.

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.

Monitoring

Cabot A self-hosted monitoring and alerting server that watches Graphite metrics and can alert on them by phone, SMS, Hipchat or email. It is designed to be deployed to cloud or physical hardware in minutes and configured via web interface.

graphite-beacon A simple alerting application for Graphite. It asynchronous and sends notification alerts based on Graphite metrics. It hasn't any dependencies except *Tornado* package. Very light and really very easy deployed.

graphite-to-zabbix A tool to make zabbix alerts based on Graphite data.

Icinga Icinga 2 will directly write metrics to the defined Graphite Carbon daemon tcp socket if the graphite feature is enabled. This feature is a more simple integration compared to Icinga 1.x and Graphios.

Moira An alerting system based on Graphite data. Moira is a real-time alerting tool, independent from graphite storage, custom expressions and extendable notification channels.

rearview A real-time monitoring framework that sits on top of Graphite's time series data. This allows users to create monitors that both visualize and alert on data as it streams from Graphite. The monitors themselves are simple Ruby scripts which run in a sandbox to provide additional security. Monitors are also configured with a crontab compatible time specification used by the scheduler. Alerts can be sent via email, pagerduty, or campfire.

Rocksteady 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).

Seyren An alerting dashboard for Graphite.

Shinken 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.

Storage Backend Alternates

If you wish to use a backend to graphite other than Whisper, there are some options available to you.

BigGraphite A time-series database written in Python on top of Cassandra.

Ceres An alternate storage backend provided by the Graphite Project. It it intended to be a distributable time-series database. It is currently in a pre-release status.

Cyanite A highly available, elastic, and low-latency time-series storage wirtten on top of Cassandra

InfluxDB A distributed time series database.

KairosDB A distributed time-series database written on top of Cassandra.

OpenTSDB A distributed time-series database written on top of HBase.

Other

bosun Time Series Alerting Framework. Can use Graphite as time series source.

Bryans-Graphite-Tools A collection of miscellaneous scripts for pulling data from various devices, F5, Infoblox, Nutanix, etc.

buckytools Go implementation of useful tools for dealing with Graphite's Whisper DBs and Carbon hashing.

carbonate Utilities for managing graphite clusters.

go-carbon Golang implementation of Graphite/Carbon server with classic architecture: Agent -> Cache -> Persister.

riemann A network event stream processing system, in Clojure. Can use Graphite as source of event stream.

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

CHAPTER 19

Working on Graphite-web

Graphite-web accepts contributions on GitHub, in the form of issues or pull requests. If you're comfortable with Python, here is how to get started.

First, keep in mind that Graphite-web supports Python versions **2.6 to 2.7** and Django versions **1.4 and above**.

Setting up a development environment

The recommended workflow is to use virtualenv / virtualenvwrapper to isolate projects between each other. This document uses virtualenv as the lowest common denominator.

Create a virtualenv at the root of your graphite-web repository:

```
virtualenv env
source env/bin/activate
```

Install the required dependencies:

```
pip install -r requirements.txt
```

Create the default storage directories:

```
mkdir -p storage/{ceres,whisper,log/webapp}
```

Then you should be able to run the graphite development server:

```
cd webapp
./manage.py runserver
```

Running the tests

To run the tests for the Python and Django versions of your virtualenv:

```
cd webapp
./manage.py test --settings=tests.settings
```

If you want to run the tests for all combinations of Python and Django versions, you can use the tox tool.

```
pip install tox
tox
```

This will run the tests for all configurations declared in the `tox.ini` file at the root of the repository.

You can see all the configurations available by running:

```
tox -l
```

You can run a single configuration with:

```
tox -e <configuration>
```

Note that you need the corresponding python version on your system. Most systems only provide one or two different python versions, it is up to you to install other versions.

Writing tests

Pull requests for new features or bugfixes should come with tests to demonstrate that your feature or fix actually works. Tests are located in the `webapp/tests` directory.

When writing a new test, look at the existing files to see if your test would fit in one. Otherwise simply create a new file named `test_<whatever>.py` with the following content:

```
from django.test import TestCase

class WhateverTest(TestCase):
    def test_something(self):
        self.assertEqual(1, 2 / 2)
```

You can read Django's testing docs for more information on `django.test.TestCase` and how tests work with Django.

CHAPTER 20

Client APIs

Cubism.js

Cubism.js is a D3 plugin for visualizing time series data in real time, and can pull data from Graphite.

Graphitejs

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

Scales

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

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.

txCarbonClient

txCarbonClient is a simple Twisted API for reporting metrics to Carbon.

CHAPTER 21

Who is using Graphite?

Here are some organizations that use Graphite:

- Brightcove (see http://opensource.brightcove.com/project/Diamond/)
- Canonical
- Datacratic
- Douban
- Dyn
- Etsy (see http://codeascraft.etsy.com/2010/12/08/track-every-release/)
- GapLabs, a division of Gap Inc.
- GitHub
- Google (opensource Rocksteady project)
- GOV.UK
- ImmobilienScout24
- InMobi
- Instagram
- ITV
- Maaii
- Mackerel
- Media Temple
- Oracle
- Orbitz
- Outbrain (see https://github.com/outbrain/gruffalo)
- Pandora

- Rubicon Project
- Sears Holdings
- SocialTwist
- Uber
- Vimeo
- Wikimedia Foundation

And many more

CHAPTER 22

Release Notes

1.0.0

04/12/2017

Graphite 1.0.0 is now available for usage. This marks the first release of Graphite's main line in many years. Also, there's a new *optional* component available: Carbonate, a suite of tools for managing and rebalancing Whisper files.

Most users will only need to install the Graphite-Web, Carbon, and Whisper components.

Source bundles are available from GitHub:

- https://github.com/graphite-project/graphite-web/archive/1.0.0.tar.gz
- https://github.com/graphite-project/carbon/archive/1.0.0.tar.gz
- https://github.com/graphite-project/whisper/archive/1.0.0.tar.gz
- https://github.com/graphite-project/carbonate/archive/1.0.0.tar.gz

Graphite can also be installed from PyPI via pip. PyPI bundles are here:

- http://pypi.python.org/pypi/graphite-web/
- http://pypi.python.org/pypi/carbon/
- http://pypi.python.org/pypi/whisper/
- http://pypi.python.org/pypi/carbonate/

Upgrading

Graphite-Web requires Python 2.7 or newer and Django version 1.9. Carbon requires Twisted version 13.2 or newer. There are a number of new features in this release, but we've been careful to avoid any behaviorial regressions in the default settings files. You'll want to review the new Graphite-Web and Carbon service options in the `local_settings.py.example` and `carbon.conf.example` files, respectively, before merging those into your production configurations.

If you're not already running from the *master* branch, Graphite-Web's application database will need to be upgraded . It's a good idea to backup the database before proceeding with the migration. The following steps will upgrade a SQLite database:

```
sudo cp /opt/graphite/storage/graphite.db \
      /opt/graphite/storage/graphite.db.backup-`date +%Y%m%d_%H%M%S`
sudo PYTHONPATH=/opt/graphite/webapp django-admin.py migrate \
      --noinput --settings=graphite.settings --run-syncdb
```

A new document is available for *developers and contributors* that covers how to setup your own development environment, and running & writing tests.

We're also happy to publish a new document for *Graphite Events*. It covers the storage, creation, retrieval, and management of events used for annotation or tracking release-style events.

Security Notes

- XSS issue affecting saved graph definitions. Targets are now properly sanitized before being handed back to the user.

New Render Functions

See the *functions documentation* for more information.

- `aggregateLine`
- `applyByNode`
- `averageOutsidePercentile`
- `delay`
- `exponentialMovingAverage`
- `fallbackSeries`
- `grep`
- `groupByNodes`
- `integralByInterval`
- `interpolate`
- `invert`
- `isNonNull`
- `linearRegression`
- `linearRegressionAnalysis`
- `mapSeries`
- `movingMin`
- `movingMax`
- `movingSum`
- `multiplySeriesWithWildcards`
- `offsetToZero`

- pow
- powSeries
- reduceSeries
- removeBetweenPercentile
- removeEmptySeries
- sortByTotal
- squareRoot
- timeSlice
- verticalLine
- weightedAverage

New Display Formats

- pdf
- dygraph
- rickshaw

New Graph Parameters

- hideNullFromLegend
- hideXAxis
- noNullPoints
- pieLabels
- valueLabels
- valueLabelsColor
- valueLabelsMin

Bug Fixes

Graphite-Web

- Render infinite values correctly for compatibility with JSON consumers (e.g. Grafana).
- Fix for aliasByMetric to handle trailing parentheses properly.
- Some functions would not handle event tags formatting. The format for these strings has been fixed.
- Improved data extraction from CarbonLink cache when crossing archive boundaries.
- Follow symlinks for RRD files.
- Unicode fixes for RRD paths.
- Support for the FNV1a_ch hashing algorithm used by the carbon-c-relay project.

- Fix for `smartSummarize` where it would discard timezone information, sometimes resulting in an exception due to broken intervals.
- Better handling for missing data in the divisor series for `divideSeries`.
- Fix function name reported in path expression for `stdev`.
- The `countSeries` function will now return zeroes instead of an empty series.
- The `constantLine` function can now be rendered without any other series.
- Incorrect float format across a variety of functions.
- Fix "thousands" unit for y-axis.
- Average/current/max/min Above/Below functions not fails if there is no data.
- Fix blank space below the legend
- Update the command to setup a new database
- aliasByNode(): support all chars allowed by the grammar

Carbon

- Avoid duplication of aggregator statistics by using a dedicated pipeline.
- Log incorrect schemas missing the `retentions` attribute.
- Improved logging for writer create or update failures.
- Fix long-standing issue with negative cache size statistic.
- Use the correct `AGGREGATION_RULES` configuration file.
- Fix race condition possible when queue is full and destinations reconnect.
- Fix `--profile` option for recording performance data.
- Improved help documentation in the Composer.
- Compute seconds_left rightly in util.
- Better handling for failed creates.
- Import manhole and amqp correctly
- Fix CacheManagementHandler() and associated tests

Whisper

- remove xFilesFactor float comparison, use whisper-resize if whisper-resize.py is not available
- (fix) [FreeBSD-i386] correct size of off_t in posix_fallocate call
- whisper-auto-resize.py: error fix
- double-fix bad commit to whisper-resize
- whisper-auto-resize: fix default values for xFilesFactor and aggregationMethod
- Fix indentation on CACHE_HEADERS, not related to AUTOFLUSH
- rrd2whisper: fix relative destinationpaths

Other Changes

Graphite-Web

- Brand new clustering implementation using a pool of worker threads and persistent connections to backends
- New Graphite logo in the Composer banner.
- Pluggable storage finders have been added. This allows graphite to fetch data from other datastores than Whisper or Ceres. See the *storage finder docs* for more information.
- The search index file is now generated with the `build-index` command that has been rewritten in Python. `build-index.sh` is still available but is just an alias to `build-index`.
- The `CONTENT_DIR` setting has been replaced with `STATIC_ROOT` and now allows to easily serve non-graphite static files such as Django admin's. See the *configuration docs* for usage instructions.
- Tox is now used for running the tests locally across the supported Django and Python version combinations. A *section about working on graphite-web* has been added to the documentation.
- Python's own log rotation can be disabled using the *LOG_ROTATION* setting. This is useful when running multiple WSGI workers.
- The events API now requires `tags` to be an array when creating tagged events. Previous versions only accepted string attributes. Tags are also serialized as arrays.
- Enhancements and optimizations to brace expansion for wildcards.
- Graphite Dashboards support absolute time ranges passed in the URL.
- Dumping the known metrics list with `/metrics/index.json` now includes RRD metrics.
- Improved support for special characters in metric names.
- Support for jsonp requests in the metrics view.
- New "refresh" button in the metrics tree navigation panel.
- Refresh all visible nodes in the metrics tree navigation view, not just the current node level.
- Support for globstar matching in target paths.
- Introduce the `MAX_TAG_LENGTH` setting for overriding the maximum tag length for events.
- Ability to retrieve a single event via the API. Previously you would have to dump the entire events database to inspect any events.
- Configurable `DATE_FORMAT` setting for overriding the short date format.
- New `nodelist` format for the metrics find view. This mode makes it easier for clients to query metric node information from the API.
- Ability to pass units suffix string (e.g. "Kb") to `cactiStyle`.
- Interpolate across null values in `perSecond` function.
- Dashboards are now sorted alphabetically in the finder.
- Support for unicode in rendered graph text.
- Improved sorting of saved graphs.
- Event times are now converted to local time to align with query times.
- Faster calculation algorithm for `movingAverage`.

- Automatically close the Dashboard's upper navigation panel if the dashboard was loaded by a parameterized URL.
- Cluster servers can now communicate over HTTPS when `INTRACLUSTER_HTTPS` is enabled.
- Readers are more resilient to the loss of a single backend
- Support whisper aggregation method "last"
- Improve json rendering performance
- Allow to override Memcache options easily
- Make readers.py more easilly importable by moving away models
- Support 0.9.x backends in 1.0.0 cluster
- hange deprecated request.REQUEST
- Decreasing number of stat syscalls dramatically using *scandir* module
- Add json format option for find api
- Forward HTTP request headers to CLUSTER_HOSTS
- Pre-load each graph image before updating the dashboard UI
- Fix incorrect display of 'title' URI parameter
- prevent repeated series evaluations for hitcount and smartSummarize

Carbon

- Support for pluggable protocols and clients. Support for protobuf was added, existing protocols have been ported over to use the new design.
- Support for pluggable routers, including new `fast-hashing` and `fast-aggregated-hashing` relay methods based on MurmurHash3.
- Introduced `CERES_NODE_CACHING_BEHAVIOR` for tuning Ceres' caching behavior.
- Aggregators now report `destinations` statistics.
- Remove unused `list` and `match-all` schema options.
- Introduced `WHISPER_FADVISE_RANDOM` as an option to avoid disk thrashing in certain scenarios.
- Support for `MAX_RECEIVER_CONNECTIONS` to limit the number of TCP connections to the intended Carbon service.
- Listeners will include metric path details when logging due to invalid line submission.
- Support logging to syslog with the `--syslog` runtime option.
- Allow Manhole to operate with no passphrase.
- New `--profiler` runtime option for specifying the profiler.
- Improved `HUP` signal handling.
- Add support for IPv6 addresses enclosed in square brackets in the destination parser.
- Add LOG_CREATES to disable creation logs
- Enforce better syslog tag
- Update carbon.amqp.conf.example

- Add protobuf support
- Add MIN_TIMESTAMP_RESOLUTION
- Fix CACHE_WRITE_STRATEGY and add TimeSortedStrategy

Whisper

- Add fallocate and sparse support in whisper-create
- use whisper-resize if whisper-resize.py is not available
- Support FADVISE_RANDOM for create/update/update_many
- rrd2whisper.py: Only suffix wsp files if required
- whisper-fill: move to optparse, add whisper file locking
- whisper-auto-resize.py: allow subdir to be a single file
- Faster evaluation for 'last' aggregation method
- Add update-storage-times.py - a tool to change storage schemas for whisper files and update the data. Threaded to vastly improve speed over whisper-auto-resize.py
- whisper: disable buffering for update operations
- whisper-diff improvement: new until param
- Add optional destinationPath for rrd2whisper
- added json output to whisper-diff.py script
- setup: install contrib scripts
- Make indentation consistent in files with mixed indentation
- Add whisper-set-xfilesfactor.py utility
- parseRetentionDef: raise more descriptive exception on wrong retention
- whisper resize: don't attempt to merge empty files
- whisper-fetch whisper-auto-update: validate if whisper.fetch() returned data

0.9.16

04/12/2017

Graphite 0.9.16 is now available for usage. Please note that this is a bugfix release for Graphite 0.9.x versions and not recommended for production use. Please use Graphite 1.0.0 for production if possible and use 0.9.16 only if you can't upgrade to 1.0.0 because of dependencies. Users upgrading from older releases are advised to review the 0.9.16 release notes first.

Source bundles are available from GitHub:

- https://github.com/graphite-project/graphite-web/archive/0.9.16.tar.gz
- https://github.com/graphite-project/carbon/archive/0.9.16.tar.gz
- https://github.com/graphite-project/whisper/archive/0.9.16.tar.gz

Graphite can also be installed from PyPI via pip. PyPI bundles are here:

- http://pypi.python.org/pypi/graphite-web/0.9.16

- http://pypi.python.org/pypi/carbon/0.9.16
- http://pypi.python.org/pypi/whisper/0.9.16

Upgrading

Graphite 0.9.16 requires a Django version of at least 1.4. Ensure this dependency is satisfied before updating *graphite-web*.

Graphite 0.9.16 was not tested against python 2.6.

Graphite 0.9.16 require Twisted version 13.2.0 or newer.

As always, comparing the example config files with existing ones is recommended to ensure awareness of any new features.

If you're not already running 0.9.15, Graphite-web's application database will need to be upgraded for a new Django fixture. It's a good idea to backup the database before proceeding with the migration. The following will upgrade a SQLite database:

```
sudo cp /opt/graphite/storage/graphite.db \
      /opt/graphite/storage/graphite.db.backup-`date +%Y%m%d_%H%M%S`
sudo PYTHONPATH=/opt/graphite/webapp django-admin.py syncdb \
      --noinput --no-initial-data --settings=graphite.settings
```

Security Notes

No known security issues.

New Features

Graphite-web

- Stop testing python2.6 (obfuscurity)
- Adding support for rendering PDF (squarebracket)
- Backport of *sortbytotal* function (mattus, jbergler)
- Gracefully handle offline cluster backends (nyerup)
- Make the rrdtool CF configurable (DazWorrall)
- Forward HTTP request headers to CLUSTER_HOSTS (benburry)
- Support for fnv1a_ch hashing for 0.9.x (deniszh)

Carbon

- Stop testing python2.6 (obfuscurity)

Whisper

- Stop testing python2.6 (obfuscurity)
- Update whisper.py : backport __archive_fetch and good version of file_fetch (Starlight42)

Bug fixes

Graphite-web

- Fixed automatic computing of yMin and yMax when drawNullAsZero=true (Crypto89)
- replace _fetchWithBootstrap (arielnh56)
- fill in missing bits from wikidot carbon page (obfuscurity)
- add events page (obfuscurity)
- Applying #1560 fix to python 2.6 as well (roeezab)
- Don't suppress exceptions in remote_storage.FindRequest (drawks, obfuscurity)
- Allow setColor to recognize unquoted integers as hex values (0.9.x) (liyichao, obfuscurity)
- port timeShift fix 7fc03ae to 0.9.x (obfuscurity)
- Fix datalib mergeResults function to properly handle time frames that are in the future compared to the cached results (iliapolo)
- Fix brace expansion (Crypto89)
- Optimizing brace expansion (iain-buclaw-sociomantic, deniszh)

Carbon

- fixing handler import for cache overflow (rubbish)
- carbon: export state.instrumentation (iksaif)

Whisper

- Fixing #163 - failing merge for multiply retentions archive (sw0x2A, deniszh)

0.9.15

11/27/2015

Graphite 0.9.15 is now available for usage. This is primarily a bugfix release for some regressions introduced in 0.9.14. Users upgrading from older releases are advised to review the 0.9.14 release notes first.

Source bundles are available from GitHub:

- https://github.com/graphite-project/graphite-web/archive/0.9.15.tar.gz
- https://github.com/graphite-project/carbon/archive/0.9.15.tar.gz
- https://github.com/graphite-project/whisper/archive/0.9.15.tar.gz

Graphite can also be installed from PyPI via pip. PyPI bundles are here:

- http://pypi.python.org/pypi/graphite-web/
- http://pypi.python.org/pypi/carbon/
- http://pypi.python.org/pypi/whisper/

Upgrading

Graphite 0.9.15 requires a Django version of at least 1.4. Ensure this dependency is satisfied before updating *graphite-web*.

As always, comparing the example config files with existing ones is recommended to ensure awareness of any new features.

If you're not already running 0.9.14, Graphite-web's application database will need to be upgraded for a new Django fixture. It's a good idea to backup the database before proceeding with the migration. The following will upgrade a SQLite database:

```
sudo cp /opt/graphite/storage/graphite.db \
      /opt/graphite/storage/graphite.db.backup-`date +%Y%m%d_%H%M%S`
sudo PYTHONPATH=/opt/graphite/webapp django-admin.py syncdb \
      --noinput --no-initial-data --settings=graphite.settings
```

Security Notes

No known security issues.

New Features

Graphite-web

- "Natural" sorting functionality added to *sortByName* (sylr, obfuscurity)

Carbon

- PyPy support (robert-zaremba, deniszh)

Bug fixes

Graphite-web

- Fix sample WSGI configuration (deniszh)
- Unnecessary call to log handler when logging disabled (ShalomCohen)
- Fix index exception for *removeAbovePercentile* and *removeBelowPercentile* (toote, obfuscurity)
- Fix premature break when fetching series (bmhatfield)
- Timezone fixes (bmhatfield)

Carbon

- Fix aggregator instrumentation (deniszh)
- Fix support for twistd syslog (rppala90, obfuscurity)
- Fix event tracking via state import (bmhatfield)

Whisper

- Revert change affecting Whisper boundaries (obfuscurity)

0.9.14

11/7/2015

Graphite 0.9.14 is now available for usage. Source bundles are available from GitHub:

- https://github.com/graphite-project/graphite-web/archive/0.9.14.tar.gz
- https://github.com/graphite-project/carbon/archive/0.9.14.tar.gz
- https://github.com/graphite-project/whisper/archive/0.9.14.tar.gz

Graphite can also be installed from PyPI via pip. PyPI bundles are here:

- http://pypi.python.org/pypi/graphite-web/
- http://pypi.python.org/pypi/carbon/
- http://pypi.python.org/pypi/whisper/

Upgrading

Graphite 0.9.14 now requires a Django version of at least 1.4. Ensure this dependency is satisfied before updating *graphite-web*.

As always, comparing the example config files with existing ones is recommended to ensure awareness of any new features.

Graphite-web's application database will need to be upgraded for a new Django fixture. It's a good idea to backup the database before proceeding with the migration. The following will upgrade a SQLite database:

```
sudo cp /opt/graphite/storage/graphite.db \
      /opt/graphite/storage/graphite.db.backup-`date +%Y%m%d_%H%M%S`
sudo PYTHONPATH=/opt/graphite/webapp django-admin.py syncdb \
      --noinput --no-initial-data --settings=graphite.settings
```

Security Notes

No known security issues.

New Features

Graphite-web

- New `minimumBelow()` function (ojilles)
- Add `margin()` function to Composer menu (obfuscurity)
- Sort "User Graphs" by creator's username (ciranor)
- New `changed()` function (kamaradclimber, obfuscurity)

- Use `query_bulk` for improved cache query performance (huy, deniszh)
- A URL shortener has been added to the composer toolbar (seveas, cbowman0, deniszh)
- Allow metric finder to return jsonp objects (kali-hernandez-sociomantic)
- Document use of alpha color values in `colorList` and `bgcolor` (ojilles)
- Improved documentation for `areaBetween()` (ojilles, gwaldo)
- Improved documentation for `cactiStyle()` (ojilles)
- Document installation via Synthesize (gwaldo)
- Mention the Synthensize installation method for Windows users (gwaldo)
- Massive cleanup across all of our documentation pages (gwaldo)
- Support for custom remote authentication backends (steve-dave)
- Improvements to the cache hash function (rmca, esc)
- Improved logging on CarbonLink failed queries (jamesjuran)
- Add navigation elements to reorder metrics in Composer (justino)
- Use Django's native TZ method (jcsp)
- New test suite and TravisCI integration (brutasse, esc)
- Rename the metrics node from "Graphite" to "Metrics" (obfuscurity)
- New `perSecond()` function (cbowman0, pcn)
- Add Composer button to load existing graph url (justino)
- Add `maxDataPoints` to limit number of returned datapoints for json requests (philiphoy, gingerlime)
- Refactor json responses for clarity (whilp)
- New *grafana* color scheme (lainSR)
- Add optional 'now' parameter to remote node fetch (Kixeye)
- Graphlot was completely removed (obfuscurity)
- Use cairocffi instead of pycairo (brutasse, esc)
- New icons converted from Font Awesome project (obfuscurity)
- Adding tests for Django 1.7 (brutasse, syepes)
- Bulk-fetch metrics from remote hosts (bmhatfield, deniszh, favoretti)
- Perform all remote fetches in parallel (bmhatfield)
- Perform `/metrics/find` queries in parallel (jraby)
- New prefetch cache for remote queries (jraby)
- New `REMOTE_STORE_USE_POST` setting for remote requests (steve-dave)
- New single-hue color templates (obfuscurity)
- New `sortByName()` function (jssjr, Krylon360)
- Namespace request-cache data with the local node (jssjr)
- Support for retrieving static assets with whitenoise (brutasse, deniszh)
- Enable buffering in HTTPResponse objects (jjneely, deniszh)

- Merge TimeSeries when more than one returned per key, making this feature optional (jjneely, deniszh)

Carbon

- Record the blacklist and whitelist metrics (jssjr, deniszh)
- Ability to run in the foreground without debug (jib, deniszh)
- Introduce `QUEUE_LOW_WATERMARK_PCT` and `TIME_TO_DEFER_SENDING` to improve batching (pcn, steve-dave)
- Configurable files for aggregation and rewrite rules (drawks)
- New bulk cache query type (huy)
- Bulk cache query instrumentation (mleinart)
- Require Twisted >= 13.2.0, adds IPv6 support (obfuscurity)
- Document how to disable individual listeners (steve-dave)
- Support backlogs for TCP listeners (jssjr)
- Improved cache tests (jssjr)
- New `--profiler` option (tail, deniszh)
- Support for `DIVERSE_REPLICAS` (deniszh, dctrwatson)

Whisper

- New whisper-fill utility imported from Carbonate project (jssjr, grobian, deniszh)
- Add `--estimate` option to whisper-create (steve-dave)
- New whisper-diff utility (bheilman, deniszh)

Bug fixes

Graphite-web

- Missing `pathExpression` in `constantLine()` function (markolson)
- Fix for `CLUSTER_SERVERS` when `ip_nonlocal_bind` is enabled (PrFalken)
- Missing ExtJS gif in Dashboard when viewed in tree configuration (obfuscurity)
- Fix docs builds for the render function list (obfuscurity, gwaldo)
- `aliasByMetric()` was including trailing arguments (obfuscurity)
- Fix `initialState` for Dashboard (cbownman0, jamesjuran)
- Broken `series.name` in `percentileOfSeries` (simm42)
- Refresh "My Graphs" *after* graph is saved or deleted (obfuscurity)
- Remove superfluous grid line with log scale (ralphm)
- Fix `holtWintersAberration()` when bands have `None` values (aaronfc)
- Number of results from cache query was incorrectly logged (steve-dave)

- Dashboard should only refresh on positive values (linkslice)
- Fix `logBase()` when value between 0 and 1 (wellle)
- Fix `PICKLE_SAFE` for remote rendering (deniszh)
- Fix `yMaxValue` when `areaMode=stacked` (bitprophet)
- Convert `time.mktime` to int to fix `identity()` function (dpkp)
- Compatibility fix in Graphlot (steve-dave)
- Off-by-one bug that broke JSON output for `constantLine` (steve-dave)
- Minor documentation fix for `sumSeriesWithWildcards()` (steve-dave)
- Fix TypeError with `sum()` function (macolu)
- Remote storage should return `None` when `seriesList` is empty (steve-dave)
- Fix project url in `setup.py` (esc)
- Fix condition where missing `until` parameter caused TypeError (steve-dave)
- Remove old jQuery workaround in Graphlot (steve-dave)
- Fix `now` handling in render queries (jcsp)
- Fix `PICKLE_SAFE` for CarbonLink queries (Dieterbe)
- Decimals not printed for `cactiStyle()` (SuminAndrew, drawks)
- Typo in exception name (also)
- Fix assumption that RemoteNode inherits from Node (mleinart)
- Updating the copyright notice (gwaldo)
- CACHE_* settings are deprecated in Django 1.3, so, was replaced with CACHES setting (brutasse, deniszh)
- Fix data cache invalidation (esc, deniszh)
- Fix documentation for divideSeries (gwaldo)
- Make HTTP clients only cache graphs as long as we keep them in memcached (aroben, deniszh)
- DST fixes, backport from graphite-api (brutasse, deniszh)
- HttpRequest.raw_post_data was deprecated in Django 1.4 (obfuscurity)
- XSS fixes for browser and composer (illicium, piotr1212, deniszh)
- Docs: Python Dev Headers needed for custom install location (gwaldo)
- Fix pytz install dependency (deniszh)
- Javascript compatibility fixes for Internet Explorer (piotr1212)
- Timezone fixes and tests (brutasse, MFAnderson, deniszh)
- Fix for remote fetch threads (deniszh)
- Fixes for `normalize()` (g76r, jstangroome)
- Avoid exceptions when `CARBONLINK_HOSTS` is an empty list (jstangroome)
- Lock django-tagging to fix Travis CI (jstangroome)
- Set default timezone (jjneely)
- Never attempt to write empty data to request-cache (apg-pk)

- Never merge CarbonLink results with Whisper rollups (penpen, obfuscurity)
- Fix for SVG graphs (grobian, obfuscurity)
- Skip empty target parameters (obfuscurity)
- Remove unnecessary dependencies (obfuscurity)

Carbon

- Restore recursive mkdir on `LOG_DIR` (jamesjuran)
- More accurate queue length reporting (pcn, bitprophet)
- Set ownership on log subdirectories if `USER` is defined (jamesjuran)
- Improved documentation for `FORWARD_ALL` (hdoshi)
- Fix whisper directory umask (alexandreboisvert, steve-dave)
- Unable to load `AGGREGATION_RULES` (drawks)
- Compatibility with Twisted 13.2.0 (esc, drawks)
- Incorrect log rotation documentation (mleinart)
- Fix carbon-cache cpu usage 100% when sent metric with too big name (jssjr, deniszh)
- Fix aggregator replication factor setting (jssjr, deniszh)
- Change the max update on shutdown (f80)
- Document the fact that one can use regexps in the aggregation-rules (ctavan)
- Move tests to tox (jssjr)
- Add hup signal handler (jssjr)
- Fix instrumentation (avishai-ish-shalom, jssjr)
- Fix exception handling (steve-dave)
- Fix CACHE_WRITE_STRATEGY (jssjr)
- Fix aggregated metrics (pgul, ctavan)
- Logging fixes (obfuscurity, piotr1212)
- Fix race condition for full queues (mleinart)
- Default value for `MAX_UPDATES_PER_SECOND_ON_SHUTDOWN` (jssjr)
- Never cache empty aggregation results (mleinart)
- Fixes for MetricsCache size leak (jssjr, deniszh)
- Documentation fix for relay-rules (obfuscurity)
- Fix test assertions (obfuscurity)
- Fix for `--profile` arg (tail, deniszh)
- Move Red Hat initscripts to examples (deniszh, bmhatfield)

Whisper

- Write optimization in `update_many` (timob, deniszh)
- Add optional `now` parameter to fetch for graphite-web compatibility (jcsp, steve-dave)
- Remove unused Tox configuration (steve-dave)
- TravisCI no longer supports Python 2.5 (steve-dave)
- Unlink Whisper file if empty/corrupted (jraby)
- Enforce closing of Whisper files (AstromechZA, jjneely)
- Handle zero length time ranges by returning the next valid point (jjneely)
- Used wrong `until` boundary for selecting archive (obfuscurity)

0.9.12

8/22/2013

This is a patch release to fix a couple of critical regressions that made it into *0.9.11*:

- Usage of django.utils.timezone breaks Django 1.3 compatibility
- Missing import in graphite.util breaks cache-queries

Source bundles are available from GitHub:

- https://github.com/graphite-project/graphite-web/archive/0.9.12.tar.gz
- https://github.com/graphite-project/carbon/archive/0.9.12.tar.gz
- https://github.com/graphite-project/whisper/archive/0.9.12.tar.gz

Graphite can also be installed from Pypi via pip. Pypi bundles are here:

- http://pypi.python.org/pypi/graphite-web/
- http://pypi.python.org/pypi/carbon/
- http://pypi.python.org/pypi/whisper/

0.9.11

8/20/2013

NOTE: Graphite 0.9.11 has a regression which breaks cache queries. Please use *0.9.12* instead

Graphite 0.9.11 is now available for usage. Source bundles are available from GitHub:

- https://github.com/graphite-project/graphite-web/archive/0.9.11.tar.gz
- https://github.com/graphite-project/carbon/archive/0.9.11.tar.gz
- https://github.com/graphite-project/whisper/archive/0.9.11.tar.gz

Graphite can also be installed from Pypi via pip. Pypi bundles are here:

- http://pypi.python.org/pypi/graphite-web/
- http://pypi.python.org/pypi/carbon/

- http://pypi.python.org/pypi/whisper/

Upgrading

It's recommended to install all three 0.9.11 packages together for the most success, however in this case *graphite-web* can be installed separately from carbon if necessary. *Carbon* and *Whisper* must be updated together due to the coupling of certain changes.

Graphite 0.9.11 now requires a Django version of at least 1.3. Ensure this dependency is satisfied before updating *graphite-web*

As always, comparing the example config files with existing ones is recommended to ensure awareness of any new features.

Security Notes

This release contains several security fixes for cross-site scripting (XSS) as well as a fix for a remote-execution exploit in graphite-web (CVE-2013-5093). Patches for the past three prior releases are available in these gists:

- 0.9.10
- 0.9.9
- 0.9.8

In a pinch, the following url mapping can be removed by hand if the remote-rendering feature is not being used:

```
diff --git a/webapp/graphite/render/urls.py b/webapp/graphite/render/urls.py
index a94a5d1..f934b43 100644
--- a/webapp/graphite/render/urls.py
+++ b/webapp/graphite/render/urls.py
@@ -15,7 +15,6 @@ limitations under the License."""
 from django.conf.urls.defaults import *

 urlpatterns = patterns('graphite.render.views',
-  ('local/?$','renderLocalView'),
   ('~(?P<username>[^/]+)/(?P<graphName>[^/]+)/?','renderMyGraphView'),
   ('', 'renderView'),
 )
```

Finally, The setting of Django's SECRET_KEY setting is now encouraged and exposed in local_settings.py as well.

New Features

Graphite-web

- Properly return an HTTP 400 on missing query parameter in metrics/search endpoint (dieterbe)
- cumulative() is now superceded by consolidateBy() which supports min/max/avg/sum (nleskiw)
- Make graphlot target host configurable for easier embedding (dieterbe)
- Allow graphlot graphs to be embedded for use in dashboard apps (dieterbe)
- When wildcarding, prefer matching metric files to directories with the same name (tmm1)
- New header design and css cleanups (obfuscurity)

- New composer button to open the target in graphlot (magec)
- timeshift() can now shift beyond current time, allowing better current-over-week charts (mgb)
- Unit scaling added to cactiStyle (drawks)
- Support RRD files in index.json view (obfuscurity)
- Support for alternate target[] url syntax (luxflux)
- New countSeries() function which returns the cardinality of a wildcard (obfuscurity)
- Bootstrap data for movingAverage and movingMedian (seveas)
- movingAverage and movingMedian now optionally take time periods to specify window size (danielbeardsley)
- jsonp support in events/get_data (gingerlime)
- Ace editor for manually editing dashboard json (jordanlewis)
- New stddevSeries(), timeStack() functions (windbender)
- Remove ugly graph image background in dashboard (frejsoya)
- y-axis divisors for determining y-axis scale are now configurable (wfarr)
- Allow any characters in axis labels
- Target grammar now supports scientific notation for numbers
- New identity() function (dieterbe)
- Update default color scheme (obfuscurity)
- Dont blow up on permissions errors while walking directories (log instead)
- Encourage users to set SECRET_KEY uniquely with a warning

Carbon

- Improvements to setup.py rpm generation and basic init scripts (bmhatfield)
- Allow alternate update rate at shutdown (Daniel314)
- Add support for new fallocate() allocation method in Whisper (slackhappy)
- Improvements to noisy logging (nleskiw, drawks)
- Protect against writes outside the storage tree
- Performance fixes to rate limiting, removal of unnecessary locks (drawks)
- Alternate write strategies for carbon-cache (max size, random) (drawks)
- carbon-aggregator aware consistent-hashing for carbon-relay (slackhappy)
- Allow custom umask to be passsed to twisted at startup (egnyte)
- New options WRITE_BACK_FREQUENCY to control frequency of partially-aggregated output (jdanbrown)
- Improve consistent-hashing performance when replication factor is 1 (slackhappy)
- Various code cleanups (sejeff)
- Allow a timestamp of -1 to be sent to aggregator to set to current time (gwillem)
- Allow log rotation to be handled by an external process (justinvenus)
- min/max aggregation methods are now supported (ishiro)

Whisper

- Better commandline sanity checking and messaging (sejeff)
- Handle SIGPIPE correctly in commandline utils (sejeff)
- Option to intelligently aggregate values on whisper-resize (jens-rantil)
- Use more efficient max() instead of sorted()[-1] (ryepup)
- Add fallocate() support (slackhappy)
- Improve handling of exceptional fetch cases (dieterbe)
- Improve rrd2whisper's handling of rrd files
- Improve error messaging on retention errors at create time (lambdafu)

Bug fixes

Graphite-web

- broken nPercentile() and related functions
- Python 2.4 compatibility in browser endpoint (dcarley)
- Missing URL parameters in composer load
- Fix to multiplySeries to return the expected type (nleskiw)
- Don't blow up when empty series passed to cactiStyle (mattus)
- Trailing commas in js breaking ie (nleskiw, davecoutts)
- Remove extra and unnecessary rendering while loading saved graphs (hostedgraphite)
- Broken entry of timezone in composer menu (hcchu)
- constantLine() not drawing across the entire graph (mattsn0w)
- SVG rendering broken when using secondYAxis (obfuscurity)
- Expect url-encoded octothorpes in colorList (magec)
- Display relative times properly in dashboard (daveconcannon)
- cactiStyle() blows up with empty series (eranrund)
- Remove problemmatic and unnecessary url encoding
- Several pathExpressions missing which caused trouble in certain function combinations (dieterbe,colby,kovyrin)
- Use non-linux-specific datetime formatter %I instead of %l (richg)
- Use os.sep properly for path separation (justinc)
- Negative numbers not allowed in yAxis input box
- scale() misreports itself in legend when using small decimals
- colorList incorrectly cast to an int in some cases (rckclmbr)
- removeBelow* menu items adding the wrong functions to target list (harveyzh)
- nPercentile renders it's name incorrectly (TimZehta)
- CSV rendering does not respect tz parameter

- Missing max interval in xAxisConfigs causes long-term graphs with few points to render with a 12hr axis config
- Stacked graphs not filling completely in staircase mode
- Stacked graphs and many drawAsInfinite() lines do not draw cleanly
- Graphlot does not handle event timestamps properly (matthew keller)
- sin() time() and randomWalk() incorrectly using float times (jbrucenet)
- legend height is incorrect when secondYAxis used (obfuscurity)
- Expanded wildcards in legends are misordered (dieterbe)
- Regression in formatPathExpression (jeblair)
- index.json returns leading periods when WHISPER_DIR does not endin a trailing slash (bitprophet)
- Regression in areaMode=all causes only the last series to be filled (piotr1212)
- Default to settings.TIMEZONE if timezone unknown (gingerlime)
- Negative filled graphs render from bottom rather than 0 (piotr1212)
- Composer and Dashboard XSS fixes (jwheare, sejeff)
- Fix persistence of tz aware datetime in non-postgres databases
- Fix insecure deserialization of pickled objects (CVE-2013-5093)
- Lots of documentation improvement (jeblair,bclermont,lensen,cbliard,hvnsweeting)

Carbon

- Empty lines match everything in whitelist (gographs)
- Storage-schemas dont auto reload when they should
- Carbon-relay per-destination metrics are broken
- Regression in MAX_CREATES_PER_MINUTE where values >60 were set to 0 (jeblair)
- Memory leak in carbon-aggregator in certain cases (lbosson)
- Python2.4 compatibility in AMQP send/receive (justinvenus)
- Cache/queue sizes are misreported (bitprophet)
- NaN values shouldn't be passed through from amqp (llaurent)

Whisper

- Python2.4 compatibility for whisper-dump.py (snore)
- Correct filtering of duplicate values to ensure last-write-wins

0.9.10

5/31/12

Graphite 0.9.10 has been released and is now available. The packages for Whisper, Carbon, and Graphite-web are available via several sources:

- Pypi (and by extension, pip)
- http://pypi.python.org/pypi/graphite-web/
- http://pypi.python.org/pypi/carbon/
- http://pypi.python.org/pypi/whisper/
- Github
- https://github.com/graphite-project/graphite-web/downloads
- https://github.com/graphite-project/carbon/downloads
- https://github.com/graphite-project/whisper/downloads
- Launchpad
- https://launchpad.net/graphite/0.9/0.9.10

This release contains a fabulous amount of incremental improvement over 0.9.9. Some highlights include: * Fixes to several annoying Composer and Dashboard UI bugs * Import of Saved Graphs into Dashboards * Fixes to cache-full behavior for carbon-cache and carbon senders (relay and aggregator) * Many new useful render functions and graph options * Improvements to the rendering engine and fixes to many rendering bugs * Support for rendering graphs as annotated SVG * Better organized and more flexible Graphite-web config layout (local_settings.py)

Upgrading from 0.9.9 should be as simple as updating the packages. It is recommended but not necessary that local_settings.py be recreated based on the newly shipped local_settings.py.example as it includes many newly exposed settings and an improved organization and comments. Carbon's config files also have a few new settings as well to check out.

The Graphite project is also in the midst of some project changes. For those who have not yet noticed, the Graphite codebase has been moved to Github (http://github.com/graphite-project) and split into individual components (Graphite-web, Carbon, Whisper, and soon Ceres). The Launchpad project remains active in supporting the project with its Answers (http://answers.launchpad.net/graphite/) and Bugs (http://bugs.launchpad.net/graphite/) functionality.

Development going forward will focus on preparing what will become Graphite 0.10.0 which will include support for the Ceres database format as well as a major refactor of the Carbon daemon (nicknamed "Megacarbon"). The master branches of the project should be considered to be in an 'alpha' state for the time being and subject to backwards-incompatible changes. Fixes to the current version will be maintained in the 0.9.x project branches but no 0.9.11 version is planned for the time being.

A big thanks goes out to all those who have helped the project in contributions of time and energy in the form of code contributions, testing, discussion, and helping each other out with support questions. Additional thanks are due to Aman Gupta (tmm1) for all of his great work on the rendering engine and other fixes, Sidnei Da Silva for his work migrating the project to Github and his fixes, and everyone who's taken the time to answer questions on the Answers site and on IRC.

As always, if you need any assistance please ask a question or join us on IRC in #graphite on Freenode.

The following is a summary of changes since the last release:

New features

Whisper

- Allocate Whisper files in chunks by default (jordansissel)
- Allow Whisper files to be allocated sparsely (jordansissel)
- Add whisper-merge command to copy data from one file to another (sidnei)

- Add whisper-dump utility (amosshapira)

Graphite Dashboard

- New button to retrieve Graph URL (octplane)
- Add button to send email of rendered graph as attachment (bkjones)
- Allow relative 'until' time to be set in dashboard (daniellawrence)
- Add ability to import Graphs into dashboards from URL or Saved Graphs

Rendering Engine

- New minorY option to configure minor gridlines (whd)
- New alpha() function to set individual color alpha values (tmm1)
- Allow areaAlpha to set alpha values for all styles of stacked graphs (tmm1)
- New minimumAbove() function: draw only series whose min is above n (tmm1)
- New areaBetween() function: draw the area between two graph lines (tmm1)
- New holtWintersConfidenceArea() function: display area between Holt-Winters confidence bands (tmm1)
- New SVG output format with embedded graph metadata (tmm1)
- New metric whitelist/blacklist functionality using pattern files
- New filterBelowPercentile() function: remove data below n percentile from a series (tmm1)
- New removeAbovePercentile() and removeAboveValue() functions to remove outliers (tmm1)
- New removeBelowPercentile() and removeBelowValue() functions to match above counterparts
- New aliasSub() function: perform a regex search/replace on metric names (tmm1)
- New rangeOfSeries() function: reduces multiple series into the value range of each point (saysjonathan)
- New movingMedian() function: moving median, similar to movingAverage (recursify)
- New multiplySeries() function: combine series by multiplying them
- New hideYAxis option (mdeeks)
- New percentileOfSeries() function: Combines series into the value at n percentile for each point
- New tranformNull() function: transforms None values to specified (cbrinley)
- New scaleToSeconds() function: scales values based on series step (redbaron)
- New aliasByMetric() function: trims all but the last element of metric name in legend (obfuscurity)
- New uniqueLegend option to filter duplicate metric names in legend (mdeeks)
- New vtitleRight option to label 2nd Y-axis

Carbon

- Allow flock() mode to be configured for Whisper
- Allow flushing of rrdcached before rrd data fetches (shufgy)
- Add ability to configure carbon metric prefix (jblaine)

Bug fixes

Whisper

- Record only the last value when duplicate timestamps are sent (knyar)
- Fix rrd2whisper.py script to work with newer python-rrdtool api

Carbon

- Fix full drain of queue after cache-full event when flow-control is enabled in both client and carbon-cache
- Fix unnecessary drop of a single metric point when cache is full
- Fix instrumentation of carbon-relay (darrellb)

Webapp

- Fix reading of Gzip'd whisper files and remote reading of RRDs
- Fix registration of Event model in admin site
- Fix events() to work with timezone aware dates
- Fix Event model to use tagging properly and fix compatibility with MySQL (hellvinz)
- Fix compatibility of built-in json module in events and graphlot
- Fix loading of saved graphs where a target has a '%' in the name

Rendering Engine

- Fix removal of whitespace above stacked graphs with yMax setting (tmm1)
- Use powers of 2 when calculating yStep and yUnitSystem=binary (tmm1)
- Force 100% usage of vertical space when yMax=max
- Compact memcached keys to keep size under 250 after Django processing (Kevin Clark)
- Fix alignFromTrue functionality in summarize() (tmm1)
- Fix cases of mismatched units in holt-winters bootstraps (lapsu,tmm1)
- Force integer in moving average window parameter (lapsu)
- Fix incorrect cache fetch when storage dir is symlinked (mk-fraggod)
- Fix infinite loop in Y-axis render when series range is very-very small
- Fix "Undo Function" button when braces expressions are present in the target
- Fix legend column calculation (darrellb)
- Fix broken aliasByNode() (darrellb)
- Fix rendering failures when infinite values are present in series
- Fix legend text overlap with dual Y-axis mode (nleskiw)
- Fix missing hunk of graph on right side with Dual Y-axis
- Fix cactiStyle() handling of None values

- Fix rendering breakage during DST time switch
- Allow multiple named stacks of metrics (aleh)
- Fix incorrect/misaligned graphs when series with unaligned steps are mixed in a graph
- Properly shift over series that have a later start time than the graph start

Composer

- Fix JS error on IE due to tailing list commas (reed-r-lance)
- Fix usage of + instead of %20 for spaces in URL encoding in composer view
- Fix display of a broken image rather than "No Data" when last target is removed
- Fix the loss of multiple targets when loading a saved graph with new params (vilkaspilkas)
- Fix unremovable duplicate metrics

Dashboard

- Fix automatic edit field selection on click (octplane)
- Fix usage of browser cache-busting uniq parameter to be filtered from memcache key (sidnei)
- Fix inability to remove Graphs with duplicate target lists

Other improvements

Carbon

- Match time units used in storage-schemas.conf with those in the webapp (ohlol)
- Only log Carbon queue fullness once (sidnei)
- Only log Carbon queue space free if it was once full (sidnei)
- Log a message with the affected filename when a Whisper update fails (bmhatfield)
- Move carbon instance logs to their own own directory to prevent clobbering
- Prevent carbon-aggregator from clobbering aggregated values when aggregating to same-name
- Add SSL option to amqp publisher (sidnei)
- Remove duplicate dot metric path filtering for performance (drawks)
- Refactor of schema validation to give more informative errors
- Add reloading of rewrite-rules and aggregation-schemas for consistency

Webapp

- Refactor settings.py to allow more complete configuration in local_settings.py
- Make Graphite compatible with Django 1.4
- Add jsonp support for /browser endpoint
- Make it harder to break metric browsing with a bad DATA_DIRS entry

Rendering Engine

- Make asPercent() much more flexible and useful
- stddev() function made more robust
- Allow metrics to begin with a braces-wildcard
- Prevent drawAsInfinite() lines from affecting Y axis height (bmhatfield)
- Pass through time with secondly rather than minutely resolution to rrdfetch (tmm1)
- Tree branches should display above all leaves (mdeeks)
- Add alignToInterval to hitcount() function similar to summarize() (jwoschitz)
- Fix PieGraph missing function
- Allow timeShift() to shift forward as well as backward

Composer

- Don't reorder targets when applying functions
- Refactor of Graph Options menu

Dashboard

- Explicitly size img tags to keep scroll position intact during reloads
- Default the navBar as collapsed when loading an existing dashboard view
- Show wildcards in top nav browsing view
- Allow dashboards to have any character in title (octplane)
- Make "Remove All Graphs" and "Change Size" dialogs modal (dannyla)
- Make the new "north" navbar the default

0.9.9

10/6/11

Graphite 0.9.9 is now out and available for download. It available through PyPI (http://pypi.python.org/pypi) and the Launchpad project page (https://launchpad.net/graphite).

This is a very substantial release. To give you an idea, the 0.9.8 release was cut from trunk around revision 380 while 0.9.9 was cut from revision 589, so that's almost as many commits as Graphite has ever had just since 0.9.8. The full changelog is too big for me to assemble nicely unfortunately, but I will try to cover all the important bits and if you're really curious you can see all the changes at http://bazaar.launchpad.net/~graphite-dev/graphite/main/changes

There are some really important things you need to know if you're upgrading from an earlier release (even trunk). Read all the change summaries below please!

API Changes

There have been API changes in whisper, carbon, and the webapp. If you are upgrading to 0.9.9 YOU MUST UPGRADE ALL 3 PACKAGES, if you mix 0.9.8 whisper with 0.9.9 carbon for example, it won't work. Upgrade all 3, and don't forget to use the –force.

The webapp has a new dependency on django.tagging (you should be able to simply 'pip install django-tagging')

New Default Behavior

We've addressed a security vulnerability with receiving pickled datapoints, see Bug #817247. This affects you in that the new default behavior is to use a more secure unpickler, which is slightly slower than the standard insecure unpickler. To revert to the less secure but faster approach previously used, you have to set USE_INSECURE_UNPICKLER=True in your carbon.conf.

Revamped Dashboard UI

- You can now use all the composer functionality by clicking on a dashboard graph
- You can drag and drop to move graphs around (and hover-drop to combine them!)
- There is an awesome new auto-completer interface available by going to the Dashboard menu, Configure UI, Completer. This may become the default in the future because its so awesome. (pro tip: try using the dashboard completer with * instead of * for some really powerful 'group by' functionality)

Other Stuff

- Tons of readthedocs.org improvements, also the example config files now have some great comment documentation
- Whisper now supports rollup aggregation methods other than averaging. The default is still to average but there a new aggregation-schemas.conf (see Bug #853955)
- To learn about the new metric metadata API that can be used to configure custom rollup aggregation methods read my answer to https://answers.launchpad.net/graphite/+question/173304 (you can skip the question part if you just care about the new API)

As for the current development focus, I can now finally work on the long-awaited merge of the 1.1 branch into trunk. The Ceres database will be in the next release, I'm going to try and merge it in (including the new refactored storage API) in the next week or so. I'll announce on graphite-dev when its available for testing. My aim is to get it fully documented for 1.0, which I'm targeting for end of this year. There might be an 0.9.10 first, depending on how many bugs are found in 0.9.9.

As always, thanks to everyone who has contributed, especially the following rockstar crew that made some major contributions in the past few months:

- Aman Gupta (tmm1)
- Nick Leskiw (nleskiw)
- Sidnei da Silva

- ChrisMD

0.9.8

4/3/11

Graphite 0.9.8 is now out and available for download. It available through PyPI (http://pypi.python.org/pypi) and the Launchpad project page (https://launchpad.net/graphite).

This release is a major step forward for Graphite, with a long list of substantive enhancements only 3 months after the last release. One of the highlights is the move of our documentation to readthedocs.org, the docs are now built using Sphinx and they live in trunk under the 'docs' folder. Just commit any changes and readthedocs.org will automatically update by pulling changes from launchpad nightly.

A special thanks goes out to AppNexus (http://appnexus.com/), who sponsored the development of two awesome new features. First is the new carbon-aggregator daemon. This new daemon lets you configure the calculation of aggregate metrics at storage time instead of using a heavy-weight sumSeries or averageSeries at rendering time. This daemon can also rewrite metric names. You manage it like the other two carbon daemons, via carbon.conf. Documentation on configuring carbon-aggregator will be coming soon.

AppNexus also sponsored the development of the new Dashboard UI. This new interface allows you to put together dashboards containing many graphs quickly and easily. You can save a dashboard and view it later. Note that this is a basic implementation for now

Beyond that, there are many other new features so please read through the changelog carefully.

Changes

- New carbon-aggregator daemon can compute your aggregate metrics
- New Dashboard UI
- Upgraded to ExtJS 3.3
- All Documentation is moving to Sphinx in our bzr branch, HTML builds of it are hosted by readthedocs.org (http://graphite.readthedocs.org/)
- The recommended Apache setup is now officially mod_wsgi and not mod_python.
- New metric pattern syntax, eg. {{example.{foo,bar}.metric}}, matches both {{example.foo.metric}} and {{example.bar.metric}}
- Y-axis now draws much more useful labels for values much less 1
- The YAxis=left|right parameter has been renamed to yAxisSide=left|right
- Rewrote webapp/render/grammar.py to be much more readable
- Added new json api call /metrics/expand/?query=foo.* -> ["foo.bar", "foo.baz", ...]
- Added debugging manhole in carbon-cache.py (ssh-accessible python intepreter interface into carbon at runtime)
- Added new hitcount function (thanks to Shane Hathaway)
- The "User Graphs" tree now works properly for usernames that contain dots
- Fixed data roll-up bug in whisper
- Added AUTOFLUSH option in whisper/carbon for synchronous I/O
- and as always, many more smaller bug fixes

- ChrisMD

0.9.7

1/8/11

A little late but better than never, Graphite 0.9.7 is now out and available for download. It available through PyPI (http://pypi.python.org/pypi) and the Launchpad project page (https://launchpad.net/graphite). Here is a quick-rundown of the new features and some nice bug fixes:

Features

- Composer UI menus have been updated to reflect all currently available functions and options
- New threshold() function allows you to draw a horizontal line with a custom color and legend name (though color and legend name are not available through composer UI yet)
- New summarize() function allows you to draw data at a lower precision than it is stored at (ie. draw hourly datapoints for minutely data)
- New group() function allows you to specify a collection of metrics for passing to other functions that require a single arg, without using wildcards.
- Retention configurations support a new more convenient syntax (see [https://bugs.launchpad.net/graphite/+bug/697896 Bug #697896])
- Carbon's logging of every whisper update can be disabled now (set LOG_UPDATES = False in carbon.conf)
- Carbon-relay can now specify ports for remote carbon-caches
- Timezones can now be specified at render-time using Olson timezone names (see [http://pytz.sourceforge.net/pytz])
- Saved MyGraphs now support a hierarchical structure when dots are used in the saved graph names
- By popular request, carbon now ignores improperly formatted datapoint lines rather than disconnecting the client ([https://bugs.launchpad.net/graphite/+bug/589476 Bug #589476])
- X-axis labeling has been revamped to avoid overlapping and confusing labels
- RPM and source RPM packages are available for download. Note that they currently **do not check dependencies** and **do not perform post-install tasks**. This means they are suitable for upgrades but the usual install doc will need to be followed for new installations. Please contribute feedback regarding these packages so we can make them work out of the box on Fedora and CentOS.

Bugs Fixed (woefully incomplete)

- [https://bugs.launchpad.net/graphite/+bug/528228 Bug #528228] - fixed 'tz' parameter for specifying custom timezone at render-time
- [https://bugs.launchpad.net/graphite/+bug/676395 Bug #676395] - fixed timeShift() function
- [https://bugs.launchpad.net/graphite/+bug/690586 Bug #690586] - fixed log() function to work with negative data
- [https://bugs.launchpad.net/graphite/+bug/684563 Bug #684563] - fixed carbon-cache.py –config parameter
- [https://bugs.launchpad.net/graphite/+bug/660861 Bug #660861] - fixed nonNegativeDerivative() math for wrapping counters
- [https://bugs.launchpad.net/graphite/+bug/591948 Bug #591948] - fixed X-axis labeling that was inaccurate in some situations

- [https://bugs.launchpad.net/graphite/+bug/595652 Bug #595652] - fixed bug preventing clustering from working with default settings.py
- [https://bugs.launchpad.net/graphite/+bug/542090 Bug #542090] - fixed Y-axis labeling issue for large values with small variance
- Dozens more...

The best part is, the work in this release has continued to be largely a community effort. Almost all bugs that got fixed were reported from users and the vast majority have been fixed because of contributed patches and highly detailed bug reports. In other words, this ain't a one-man show! Thanks to everyone who has contributed code, bug reports, documentation, questions, and answers.

In the interest of not being incredibly wrong again, I will refrain from putting a date on when the next Graphite release will be out. But it will not be another year, that's for sure... Several projects I have to do for work in the coming months are going to involve major enhancements to Graphite, unlike this past year during which I've really only worked on it in my spare time. Thanks again to everyone and happy new year!

- ChrisMD

0.9.6

2/26/10

This has probably been the most active month of Graphite development since the project was open sourced. Lots of community members have contributed code and ideas to help move Graphite forward. I'm really excited about this, the project is gaining momentum and I hope we can keep that up by continuing with the new monthly release cycle. To give credit where it is due, here is a list of this month's most active users and what they've been working on (in no particular order):

- Lucio Torre - AMQP support
- jdugan - beautification of the Y-axis labels via the yUnitSystem option
- Nick Leskiw - the YAxis=right rendering option
- Kraig Amador - tons of rendering options/functions such as yLimit, timeShift(), log(), sumSeriesWithWildcard(), new filtering functions, and much more! (Kraig you're the man!)
- Arthur Gautier - debian packaging
- Elliot Murphy - packaging, inclusion in Ubuntu
- fp - RHEL / CentOS RPM packaging
- and many more...

Thanks to everyone who has gotten involved with Graphite, your support helps motivate others (especially me).

Many of these new features are really great but unfortunately undocumented, but the good news is that my focus for March is going to be 100% on *documentation*. There may not be an actual code release in March but I hope to get a substantial amount of documentation written right here on this wiki. Stay tuned.

- ChrisMD

0.9.5

1/4/10

It's hard to believe it's been an entire year since the last release of Graphite. This just goes to show how good I am at procrastination. After taking a look at the old 0.9.4 release I can safely say that the new 0.9.5 release is very significant. Here are the biggest changes:

Graphite now supports [[[federated storage]]] (for better scalability) # Carbon was completely rewritten using Twisted (much cleaner, more configurable, and more fault tolerant) # The installation process now uses distutils (finally!) # Graphite, Carbon, and Whisper are now three separate packages (for more flexible deployment) # Graphite's browser UI fully migrated to pure ExtJS 3.0 (cleaner code, less bugs) # Many many bug fixes as always

I'd like to thank everyone in the community who has been using graphite and contributing to the project. I don't usually do new year's resolutions, but I've got a good idea for one this year. I really want to get away from infrequent huge releases like this one and get back to very frequent small releases in the true spirit of open source. So my resolution is to release *something* once a month. It will probably usually just be bug fixes, or perhaps some much needed documentation. So look forward to something new in February!

- *ChrisMD*

0.9.4

1/30/09

It's been a good 6 months since the last release. Not much has changed aside from a few minor enhancements and some good bug fixes, unfortunately I've not had nearly as much time as I'd like to dedicate to working on Graphite. Regardless, it is getting more mature slowly but surely. In the next few months I may be in a better position to get more real work done on it, but we shall see. For now I'd just like to thank everyone who has given me great questions and bug reports, your feedback is what keeps this project moving. Thanks.

- ChrisMD

0.9.3

7/16/08

This release is an incremental improvement over 0.9.2, including lots of bug fixes, major enhancements to the installer, and several new handy scripts. Thanks to everyone who submitted bug reports and questions. The next few Graphite releases will continue to focus on quality rather than new features. In particular, 0.9.4 will include a re-write of the carbon backend, which will be much simpler and easier to administer and troubleshoot. I am also working on porting lots of internal documentation to this wiki. My goal is to have a 1.0 release by the end of the year, which must be well-documented, easy to deploy, easy to troubleshoot, and of course as bug-free as possible. If there is time a new feature or two might make it in, but this is not the primary focus.

- ChrisMD

0.9.2

I've received a bunch of great bug reports and feedback on the 0.9 release and have resolved lots of minor issues this week as a result. So please try out the new 0.9.2 release on the downloads page and keep the bug reports coming!

- ChrisMD

CHAPTER 23

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

Index

T

U

V

W