Graphite Documentation

Release 0.9.10

Chris Davis

August 14, 2013
# CONTENTS

1 Overview
  1.1 About the project ........................................... 1
  1.2 The architecture in a nutshell ................................. 1

2 Installing Graphite
  2.1 Dependencies .................................................. 3
  2.2 Fulfilling Dependencies ....................................... 4
  2.3 Default Installation Layout .................................. 4
  2.4 Installing Graphite ............................................ 5
  2.5 Initial Configuration .......................................... 8
  2.6 Help! It didn’t work! ........................................... 8
  2.7 Post-Install Tasks ............................................. 9

3 The Carbon Daemons
  3.1 carbon-cache.py ............................................... 11
  3.2 carbon-relay.py ............................................... 11
  3.3 carbon-aggregator.py ......................................... 12

4 Configuring Carbon
  4.1 carbon.conf .................................................... 13
  4.2 storage-schemas.conf ......................................... 13
  4.3 storage-aggregation.conf ..................................... 15
  4.4 relay-rules.conf ............................................... 15
  4.5 aggregation-rules.conf ....................................... 16
  4.6 whitelist and blacklist ....................................... 16

5 Feeding In Your Data
  5.1 The plaintext protocol ....................................... 17
  5.2 The pickle protocol .......................................... 17
  5.3 Using AMQP ..................................................... 18

6 Configuring The Webapp ........................................... 19

7 Administering The Webapp ......................................... 21

8 Using The Composer ................................................ 23

9 The Render URL API
  9.1 Graphing Metrics ............................................... 25
  9.2 Data Display Formats ......................................... 27
9.3 Graph Parameters .................................................. 29

10 Functions .......................................................... 41
  10.1 Usage ........................................................... 41
  10.2 List of functions .................................................. 41

11 The Dashboard UI ................................................... 53

12 The Whisper Database .............................................. 55
  12.1 Data Points ....................................................... 55
  12.2 Archives: Retention and Precision ............................ 55
  12.3 Rollup Aggregation .............................................. 55
  12.4 Multi-Archive Storage and Retrieval Behavior .............. 56
  12.5 Disk Space Efficiency .......................................... 56
  12.6 Differences Between Whisper and RRD ....................... 56
  12.7 Performance ..................................................... 56
  12.8 Database Format ................................................. 57

13 Graphite Terminology ............................................... 59

14 Tools That Work With Graphite .................................. 61
  14.1 Bucky ............................................................ 61
  14.2 collectd .......................................................... 61
  14.3 Collectl .......................................................... 61
  14.4 Charcoal ........................................................ 62
  14.5 Diamond ........................................................ 62
  14.6 Ganglia .......................................................... 62
  14.7 GDash ........................................................... 62
  14.8 Graphene ......................................................... 62
  14.9 Graphite-relay ................................................... 62
  14.10 Graphite-Tattle .................................................. 62
  14.11 Graphiti ........................................................ 62
  14.12 Graphitoid ....................................................... 62
  14.13 Graphpios ....................................................... 63
  14.14 Graphitejs ....................................................... 63
  14.15 Grocket ........................................................ 63
  14.16 HoardD .......................................................... 63
  14.17 Host sFlow ....................................................... 63
  14.18 hubot-scripts ................................................... 63
  14.19 jmxtrans ........................................................ 63
  14.20 Logster .......................................................... 63
  14.21 Pencil ............................................................ 64
  14.22 Rocksteady ....................................................... 64
  14.23 Scales ............................................................ 64
  14.24 Shinken .......................................................... 64
  14.25 statsd ............................................................ 64
  14.26 Tasseo .......................................................... 64

15 Who is using Graphite? ............................................... 65

16 Indices and tables ................................................... 67

Python Module Index .................................................. 69
Graphite does two things:

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

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

1.1 About the project

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

1.2 The architecture in a nutshell

Graphite consists of 3 software components:

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

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

INSTALLING GRAPHITE

2.1 Dependencies

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

Basic Graphite requirements:

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

Python 2.4 and 2.5 have extra requirements:

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

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

There are also several other dependencies required for additional features:

- Render caching: `memcached` and `python-memcache`
- LDAP authentication: `python-ldap` (for LDAP authentication support in the webapp)
- AMQP support: `txamqp`
- RRD support: `python-rrdtool`
- Dependant modules for additional database support (MySQL, PostgreSQL, etc). See Django database install instructions and the Django database documentation for details
See Also:
On some systems it is necessary to install fonts for Cairo to use. If the webapp is running but all graphs return as broken images, this may be why.

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

2.2 Fulfilling Dependencies

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

2.3 Default Installation Layout

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

2.3.1 Whisper

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

2.3.2 Carbon and Graphite-web

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

- bin/
- conf/
- lib/
  
  - Carbon PYTHONPATH
- storage/
  
    - log
      
      Log directory for Carbon and Graphite-web
    - rrd
      
      Location for RRD files to be read
    - whisper
      
      Location for Whisper data files to be stored and read
- webapp/

  - Graphite-web PYTHONPATH
2.4 Installing Graphite

Several installation options exist:

2.4.1 Installing From Source

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

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

Installing in the Default Location

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

Installing Carbon in a Custom Location

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

The following parameters influence the install location:

- --prefix
  Location to place the bin/ and storage/ and conf/ directories (defaults to /opt/graphite/)
- --install-lib
  Location to install Python modules (default: /opt/graphite/lib)
- --install-data
  Location to place the storage and conf directories (default: value of prefix)
- --install-scripts
  Location to place the scripts (default: bin/ inside of prefix)

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

python setup.py install --prefix=/srv/graphite --install-lib=/srv/graphite/lib
To install Carbon into the system-wide site-packages directory with scripts in /usr/bin and storage and configuration in /usr/share/graphite:

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

### 2.4.2 Installing From Pip

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

#### Installing in the Default Location

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

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

**Note:** On RedHat-based systems using the `python-pip` package, the `pip` executable is named `pip-python`
Installing Carbon in a Custom Location

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

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

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

```
pip install carbon --install-option="--prefix=/srv/graphite" --install-option="--install-lib=/srv/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 carbon --install-option="--install-scripts=/usr/bin" --install-option="--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 graphite-web --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 graphite-web --install-option="--install-scripts=/usr/bin" --install-option="--install-option="--install-lib=/usr/lib/python2.6/site-packages" --install-option="--install-data=/var/lib/graphite"
```

2.4.3 Installing in Virtualenv

*Virtualenv* provides an isolated Python environment to run Graphite in.

Installing in the Default Location

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

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

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

Installing in a Custom Location

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

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

Running Graphite-web Within Virtualenv

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

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

Apache mod_wsgi

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

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

```
WSGIPythonHome /opt/graphite
```

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

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

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

Gunicorn

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

uWSGI

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

2.5 Initial Configuration

2.6 Help! It didn’t work!

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

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

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

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

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

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

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

THE CARBON DAEMONS

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

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

3.1 carbon-cache.py

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

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

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

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

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

3.2 carbon-relay.py

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

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

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

carbon-relay.py is configured via:

- **carbon.conf** The [relay] section defines listener host/ports and a RELAY_METHOD
relay-rules.conf  In RELAY_METHOD = rules, pattern/servers tuples define what servers metrics matching certain
regex rules are forwarded to.

3.3 carbon-aggregator.py

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

carbon-aggregator.py is configured via:

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

aggregation-rules.conf  Defines a time interval (in seconds) and aggregation function (sum or average) for incoming
metrics matching a certain pattern. At the end of each interval, the values received are aggregated and published
to carbon-cache.py as a single metric.
Carbon’s config files all live in /opt/graphite/conf/. If you’ve just installed Graphite, none of the .conf files will exist yet, but there will be a .conf.example file for each one. Simply copy the example files, removing the .example extension, and customize your settings.

4.1 carbon.conf

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

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

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

4.2 storage-schemas.conf

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

Important notes before continuing:

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

A given rule is made up of 3 lines:

• A name, specified inside square brackets.
• A regex, specified after “pattern=“
• A retention rate line, specified after “retentions=“

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

Frequencies and histories are specified using the following suffixes:

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

Here’s a simple, single retention example:

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

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

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

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

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

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

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

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

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

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

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

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

### 4.3 storage-aggregation.conf

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

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

Here’s an example:

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

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

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

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

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

### 4.4 relay-rules.conf

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

Example:

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

You must define at least one section as the default.
4.5 aggregation-rules.conf

Aggregation rules allow you to add several metrics together as 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’.

For example, if your metric naming scheme is:

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

You could configure some aggregations like so:

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

As an example, if the following metrics are received:

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

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

4.6 whitelist and blacklist

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

GRAPHITE_CONF_DIR is searched for whitelist.conf and blacklist.conf. Each file contains one regular expressions per line to match against metric values. If the whitelist configuration is missing or empty, all metrics will be passed through by default.
Getting your data into Graphite is very flexible. There are three main methods for sending data to Graphite: Plaintext, Pickle, and AMQP.

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

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

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

### 5.1 The plaintext protocol

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

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

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

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

### 5.2 The pickle protocol

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

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

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

Once you’ve formed a list of sufficient size (don’t go too big!), send the data over a socket to Carbon’s pickle receiver (by default, port 2004). You’ll need to pack your pickled data into a packet containing a simple header:
payload = pickle.dumps(listOfMetricTuples)
header = struct.pack("!L", len(payload))
message = header + payload

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

### 5.3 Using AMQP

...
CONFIGURING THE WEBAPP
ADMINISTERING THE WEBAPP
USING THE COMPOSER
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

### 9.1 Graphing Metrics

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

#### 9.1.1 target

This will draw one or more metrics

Example:
Let’s say there are 4 identical application instances running on each server.

Now let’s say you have 10 servers.

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

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

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

### 9.1.2 from / until

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

There are multiple formats for these functions:

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

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

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>Seconds</td>
</tr>
<tr>
<td>min</td>
<td>Minutes</td>
</tr>
<tr>
<td>h</td>
<td>Hours</td>
</tr>
<tr>
<td>d</td>
<td>Days</td>
</tr>
<tr>
<td>w</td>
<td>Weeks</td>
</tr>
<tr>
<td>mon</td>
<td>30 Days (month)</td>
</tr>
<tr>
<td>y</td>
<td>365 Days (year)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>HH</td>
<td>Hours, in 24h clock format. Times before 12PM must include leading zeroes.</td>
</tr>
<tr>
<td>MM</td>
<td>Minutes</td>
</tr>
<tr>
<td>YYYY</td>
<td>4 Digit Year.</td>
</tr>
<tr>
<td>MM</td>
<td>Numeric month representation with leading zero</td>
</tr>
<tr>
<td>DD</td>
<td>Day of month with leading zero</td>
</tr>
</tbody>
</table>
&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+l
(shows data since the beginning of the current year)

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

9.2 Data Display Formats

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

9.2.1 format

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

Examples:

&format=png
&format=raw
&format=csv
&format=json
&format=svg

png

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

raw

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

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

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

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

json

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

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

svg

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

```
<svg width="1000" height="500">
<rect width="1000" height="500"/>
<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, 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 "]
        }
    }
    ]>
</script>
</svg>
```
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]
  }

]

9.2.2 rawData

Deprecated since version 0.9.9. Used to get numerical data out of the webapp instead of an image. Can be set to true, false, csv. Affects all &targets passed in the URL.

Example:

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

Returns the following text:

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

9.3 Graph Parameters

9.3.1 areaAlpha

Default: 1.0
Takes a floating point number between 0.0 and 1.0 Sets the alpha (transparency) value of filled areas when using an areaMode.

### 9.3.2 areaMode

*Default: none*

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

- **none** Disables areaMode
- **first** Fills the area under the first target and no other
- **all** Fills the areas under each target
- **stacked** Creates a graph where the filled area of each target is stacked on one another. Each target line is displayed as the sum of all previous lines plus the value of the current line.

### 9.3.3 bgcolor

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

Sets the background color of the graph.

<table>
<thead>
<tr>
<th>Color Names</th>
<th>RGB Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>black</td>
<td>0,0,0</td>
</tr>
<tr>
<td>white</td>
<td>255,255,255</td>
</tr>
<tr>
<td>blue</td>
<td>100,100,255</td>
</tr>
<tr>
<td>green</td>
<td>0,200,0</td>
</tr>
<tr>
<td>red</td>
<td>200,0,50</td>
</tr>
<tr>
<td>yellow</td>
<td>255,255,0</td>
</tr>
<tr>
<td>orange</td>
<td>255,165,0</td>
</tr>
<tr>
<td>purple</td>
<td>200,100,255</td>
</tr>
<tr>
<td>brown</td>
<td>150,100,50</td>
</tr>
<tr>
<td>aqua</td>
<td>0,150,150</td>
</tr>
<tr>
<td>gray</td>
<td>175,175,175</td>
</tr>
<tr>
<td>grey</td>
<td>175,175,175</td>
</tr>
<tr>
<td>magenta</td>
<td>255,0,255</td>
</tr>
<tr>
<td>pink</td>
<td>255,100,100</td>
</tr>
<tr>
<td>gold</td>
<td>200,200,0</td>
</tr>
<tr>
<td>rose</td>
<td>200,150,200</td>
</tr>
<tr>
<td>darkblue</td>
<td>0,0,255</td>
</tr>
<tr>
<td>darkgreen</td>
<td>0,255,0</td>
</tr>
<tr>
<td>darkred</td>
<td>255,0,0</td>
</tr>
<tr>
<td>darkgray</td>
<td>111,111,111</td>
</tr>
<tr>
<td>darkgrey</td>
<td>111,111,111</td>
</tr>
</tbody>
</table>

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

Examples:

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

*Default:* The value of `DEFAULT_CACHE_DURATION` from `local_settings.py`  
The time in seconds for the rendered graph to be cached (only relevant if memcached is configured)

9.3.5 colorList

*Default:* value from the `[default]` template in `graphTemplates.conf`  
Takes one or more comma-separated color names or RGB values (see `bgcolor` for a list of color names) and uses that list in order as the colors of the lines. If more lines / metrics are drawn than colors passed, the list is reused in order.  
Example:

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

9.3.6 drawNullAsZero

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

9.3.7 fgcolor

*Default:* value from the `[default]` template in `graphTemplates.conf`  
Sets the foreground color. This only affects the title, legend text, and axis labels.  
See `majorGridLineColor`, and `minorGridLineColor` for further control of colors.  
See `bgcolor` for a list of color names and details on formatting this parameter.

9.3.8 fontBold

*Default:* value from the `[default]` template in `graphTemplates.conf`  
If set to true, makes the font bold.  
Example:

```
&fontBold=true
```

9.3.9 fontItalic

*Default:* value from the `[default]` template in `graphTemplates.conf`  
If set to true, makes the font italic / oblique. Default is false.  
Example:

```
&fontItalic=true
```
9.3.10 **fontName**

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

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

Example:

```
&fontName=FreeMono
```

9.3.11 **fontSize**

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

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

Example:

```
&fontSize=8
```

9.3.12 **format**

See: Data Display Formats

9.3.13 **from**

See: from / until

9.3.14 **graphOnly**

*Default: False*

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

9.3.15 **graphTypes**

*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**

9.3.16 **hideLegend**

*Default: <unset>*

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

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

\&hideLegend=false

### 9.3.17 hideAxes

*Default: False*

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

Example:

\&hideAxes=true

### 9.3.18 hideYAxis

*Default: False*

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

### 9.3.19 hideGrid

*Default: False*

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

Example:

\&hideGrid=true

### 9.3.20 height

*Default: 250*

Sets the height of the generated graph image in pixels.

See also: `width`

Example:

\&width=650&height=250

### 9.3.21 jsonp

*Default: <unset>*

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

### 9.3.22 leftColor

*Default: color chosen from colorList*

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

*Default: False*

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

### 9.3.24 leftWidth

*Default: value of the parameter lineWidth*

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

### 9.3.25 lineMode

*Default: slope*

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

- **slope**  Slope line mode draws a line from each point to the next. Periods will Null values will not be drawn.
- **staircase**  Staircase draws a flat line for the duration of a time period and then a vertical line up or down to the next value.
- **connected**  Like a slope line, but values are always connected with a slope line, regardless of whether or not there are Null values between them.

Example:

```
&lineMode=staircase
```

### 9.3.26 lineWidth

*Default: 1.2*

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

Example:

```
&lineWidth=2
```

### 9.3.27 logBase

*Default: <unset>*

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

### 9.3.28 localOnly

*Default: False*

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

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

Sets the color of the major grid lines.
See bgcolor for valid color names and formats.
Example:
&majorGridLineColor=#FF22FF

9.3.30 margin

_default: 10 Sets the margin around a graph image in pixels on all sides.
Example:
&margin=20

9.3.31 max

Deprecated since version 0.9.0: See yMax

9.3.32 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

9.3.33 minorY

Sets the number of minor grid lines per major line on the y-axis.
Example:
&minorY=3

9.3.34 min

Deprecated since version 0.9.0: See yMin
9.3.35 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.

9.3.36 noCache

Default: False

Set to disable caching of rendered images

9.3.37 pickle

Deprecated since version 0.9.10: See Data Display Formats

9.3.38 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

9.3.39 rightColor

Default: color chosen from colorList

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

9.3.40 rightDashed

Default: False

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

9.3.41 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
9.3.42 template

*Default: default*

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

Example:

```
&template=plain
```

9.3.43 thickness

Deprecated since version 0.9.0: See: lineWidth

9.3.44 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
```

9.3.45 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`.

---

9.3.46 uniqueLegend

*Default: False*

Display only unique legend items, removing any duplicates

9.3.47 until

See: from / until
9.3.48 vtitle

Default: <unset>
Labels the y-axis with vertical text. If unset, no y-axis label is displayed.
Example:
&vtitle=Threads

9.3.49 vtitleRight

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

9.3.50 width

Default: 330
Sets the width of the generated graph image in pixels.
See also: height
Example:
&width=650&height=250

9.3.51 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.

9.3.52 yAxisSide

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

9.3.53 yLimit

Reserved for future use See: yMax

9.3.54 yLimitLeft

Reserved for future use See: yMaxLeft

9.3.55 yLimitRight

Reserved for future use See: yMaxRight
9.3.56 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

9.3.57 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

9.3.58 yMaxLeft

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

9.3.59 yMaxRight

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

9.3.60 yMinLeft

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

9.3.61 yMinRight

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

9.3.62 yStep

*Default: Calculated automatically*
Manually set the value step between Y-axis labels and grid lines

9.3.63 yStepLeft

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

9.3.64 yStepRight

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

_Default: si_

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

- **si** Use si units (powers of 1000) - K, M, G, T, P
- **binary** Use binary units (powers of 1024) - Ki, Mi, Gi, Ti, Pi
- **none** Dont compact values, display the raw number
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.

10.1 Usage

Most functions are applied to one series list. Functions with the parameter $\ast$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.

10.2 List of functions

**absolute** *(seriesList)*

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

Example:

```target=absolute(Server.instance01.threads.busy)```

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

**alias** *(seriesList, newName)*

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

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

**aliasByMetric** *(seriesList)*

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

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

**aliasByNode** *(seriesList, *nodes)*

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

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

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

Runs series names through a regex search/replace.
&target=aliasSub(ip.*TCP\*,"^.*TCP\(\d+\)\","\1")

**alpha** *(seriesList, alpha)*

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

**areaBetween** *(seriesList)*

Draws the area in between the two series in seriesList.

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

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

The `total` parameter may be a single series or a numeric value.

Example:

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

**averageAbove** *(seriesList, n)*

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

Example:

```plaintext
&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:

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

Draws the servers with average values below 25.

**averageSeries** *(seriesLists)*

Short Alias: `avg()`

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

Example:

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

**averageSeriesWithWildcards** *(seriesList, *position)*

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

Example:

```plaintext
&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)*

Takes a series list and modifies the aliases to provide column aligned output with Current, Max, and Min values in the style of cacti. NOTE: column alignment only works with monospace fonts such as terminus.
&target=cactiStyle(ganglia.*.net.bytes_out)

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’)

constantLine (value)
Takes a float F.
Draws a horizontal line at value F across the graph.

Example:
&target=constantLine(123.456)

cumulative (seriesList)
Takes one metric or a wildcard seriesList.
By default, when a graph is drawn, and the width of the graph in pixels is smaller than the number of datapoints to be graphed, Graphite averages the value at each pixel. The cumulative() function changes the consolidation function to sum from average. This is especially useful in sales graphs, where fractional values make no sense (How can you have half of a sale?)

&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=highestAbove(server*.instance*.threads.busy,50)

Draws the servers with more than 50 busy threads.

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

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

Draws the servers with less than 3 busy threads.

dashed (*seriesList)
Takes one metric or a wildcard seriesList, followed by a float F.
Draw the selected metrics with a dotted line with segments of length F If omitted, the default length of the segments is 5.0

Example:
&target=dashed(server01.instance01.memory.free,2.5)
**derivative** (*seriesList*)
This is the opposite of the integral function. This is useful for taking a running total metric and showing how many requests per minute were handled.

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

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

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

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

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

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

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

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

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

**events** (*tags*)
Returns the number of events at this point in time. Usable with drawAsInfinite.

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

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

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

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

**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),...

**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=highestCurrent(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.
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=').

keepLastValue (seriesList)
Takes one metric or a wildcard seriesList. 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)

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

&target=legendValue(Sales.widgets.largeBlue, ‘avg’, ‘max’, ‘si’)

limit (seriesList, n)
Takes one metric or a wildcard seriesList followed by an integer N.
Only draw the first N metrics. Useful when testing a wildcard in a metric.

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

Draws only the first 5 instance’s memory free.

lineWidth (seriesList, width)
Takes one metric or a wildcard seriesList, followed by a float F.
Draw the selected metrics with a line width of F, overriding the default value of 1, or the &lineWidth=X.X parameter.
Useful for highlighting a single metric out of many, or having multiple line widths in one graph.

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

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

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

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

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

Draws the bottom 5 servers with the lowest average value.

**lowestCurrent**(seriesList, n)

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

Example:

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

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

**maxSeries**(seriesLists)

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

Example:

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

**maximumAbove**(seriesList, n)

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

Example:

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

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

**maximumBelow**(seriesList, n)

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

Example:

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

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

**minSeries**(seriesLists)

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

Example:

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

**minimumAbove**(seriesList, n)

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

Example:

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

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

**mostDeviant**(n, seriesList)

Takes an integer N followed by one metric or a wildcard seriesList. Draws the N most deviant metrics. To find
the deviant, the average across all metrics passed is determined, and then the average of each metric is compared to the overall average.

Example:

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

Draws the 5 instances furthest from the average memory free.

**movingAverage** *(seriesList, windowSize)*

Takes one metric or a wildcard seriesList followed by a number N of datapoints and graphs the average of N previous datapoints. N-1 datapoints are set to None at the beginning of the graph.

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

**movingMedian** *(seriesList, windowSize)*

Takes one metric or a wildcard seriesList followed by a number N of datapoints and graphs the median of N previous datapoints. N-1 datapoints are set to None at the beginning of the graph.

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

**multiplySeries** *(seriesLists)*

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

Example:

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

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

Returns n-percent of each series in the seriesList.

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

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

Example:

```
&target=derivative(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)
```

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

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

**randomWalkFunction** *(name)*

Short Alias: randomWalk()

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

Example:

```
&target=randomWalk("The.time.series")
```
This would create a series named “The.time.series” that contains points where \( x(t) = x(t-1) + \text{random()} - 0.5 \), and \( x(0) = 0 \).

**rangeOfSeries** (*seriesLists*)

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

Example:

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

**removeAbovePercentile** (seriesList, n)

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

**removeBelowPercentile** (seriesList, n)

Removes data above 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 above the given threshold from the series or list of series provided. Values below this threshold are assigned a value of None.

**scale** (seriesList, factor)

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

Example:

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

**scaleToSeconds** (seriesList, seconds)

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

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

**secondYAxis** (seriesList)

Graph the series on the secondary Y axis.

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

Short Alias: sin()

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

Example:

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

This would create a series named “The.time.series” that contains \( \sin(x) \times 2 \).

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

Smarter experimental version of summarize.

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

10.2. List of functions
sortByMaxima (seriesList)
   Takes one metric or a wildcard seriesList.
   Sorts the list of metrics by the maximum value across the time period specified. Useful with the &areaMode=all parameter, to keep the lowest value lines visible.
   Example:
   &target=sortByMaxima(server*.instance*.memory.free)

sortByMinima (seriesList)
   Takes one metric or a wildcard seriesList.
   Sorts the list of metrics by the lowest value across the time period specified.
   Example:
   &target=sortByMinima(server*.instance*.memory.free)

stacked (seriesLists, stackName='__DEFAULT__')
   Takes one metric or a wildcard seriesList and change them so they are stacked. This is a way of stacking just a couple of metrics without having to use the stacked area mode (that stacks everything). By means of this a mixed stacked and non stacked graph can be made
   It can also take an optional argument with a name of the stack, in case there is more than one, e.g. for input and output metrics.
   Example:
   &target=stacked(company.server.application01.ifconfig.TXPackets, 'tx')

stdev (seriesList, points, windowTolerance=0.1)
   Takes one metric or a wildcard seriesList followed by an integer N. Draw the Standard Deviation of all metrics passed for the past N datapoints. If the ratio of null points in the window is greater than windowTolerance, skip the calculation. The default for windowTolerance is 0.1 (up to 10% of points in the window can be missing). Note that if this is set to 0.0, it will cause large gaps in the output anywhere a single point is missing.
   Example:
   &target=stdev(server*.instance*.threads.busy,30)
   &target=stdev(server*.instance*.cpu.system,30,0.0)

substr (seriesList, start=0, stop=0)
   Takes one metric or a wildcard seriesList followed by 1 or 2 integers. Assume that the metric name is a list or array, with each element separated by dots. Prints n - length elements of the array (if only one integer n is passed) or n - m elements of the array (if two integers n and m are passed). The list starts with element 0 and ends with element (length - 1).
   Example:
   &target=substr(carbon.agents.hostname.avgUpdateTime,2,4)
   The label would be printed as “hostname.avgUpdateTime”.

sumSeries (*seriesLists)
   Short form: sum()
   This will add metrics together and return the sum at each datapoint. (See integral for a sum over time)
   Example:
   &target=sum(company.server.application*.requestsHandled)
This would show the sum of all requests handled per minute (provided requestsHandled are collected once a minute). If metrics with different retention rates are combined, the coarsest metric is graphed, and the sum of the other metrics is averaged for the metrics with finer retention rates.

**sumSeriesWithWildcards** (*seriesList*, *position*)

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

Example:

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

This would be the equivalent of `target=sumSeries(host.*.cpu-user.value)&target=sumSeries(host.*.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 calcuated 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, "omgwtffbbq", red)
```

**timeFunction** (*name*)

Short Alias: `time()`

Just returns the timestamp for each X value. T

Example:

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

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

**timeShift** (*seriesList*, *timeShift*)

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.

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

Example:

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

**transformNull** *(seriesList, default=0)*

Takes a metric or wild card seriesList and an optional value to transform Nulls to. Default is 0. This method compliments drawNullAsZero flag in graphical mode but also works in text only mode. Example:

```plaintext
&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

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

THE DASHBOARD UI

...
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.

12.1 Data Points

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

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

12.2 Archives: Retention and Precision

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

To support accurate aggregation from higher to lower resolution archives, the number of points in a longer retention archive must be divisible by its next lower retention archive. For example, an archive with 1 data points every 60 seconds and retention of 120 points (2 hours worth of data) can have a lower-resolution archive following it with a resolution of 1 data point every 300 seconds for 1200 points, while the same resolution but for only 1000 points would be invalid since 1000 is not evenly divisible by 120.

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. Instead, it will provide 1 year of storage.

12.3 Rollup Aggregation

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

- average
- sum
- last
- max
- min
12.4 Multi-Archive Storage and Retrieval Behavior

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

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

12.5 Disk Space Efficiency

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

- **Each data point is stored with its timestamp** Rather than a timestamp being inferred from its position in the archive, timestamps are stored with each point. The timestamps are 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).

12.6 Differences Between Whisper and RRD

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

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

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

12.7 Performance

Whisper is fast enough for most purposes. It is slower than RRDtool primarily as a consequence of Whisper being written in Python, while RRDtool is written in C. The speed difference between the two in practice is quite small as much effort was spent to optimize Whisper to be as close to RRDtool’s speed as possible. Testing has shown that update operations take anywhere from 2 to 3 times as long as RRDtool, and fetch operations take anywhere from 2 to
5 times as long. In practice the actual difference is measured in hundreds of microseconds (10^-4) which means less than a millisecond difference for simple cases.

### 12.8 Database Format

<table>
<thead>
<tr>
<th>Whisper-File</th>
<th>Header,Data</th>
<th>Metadata,ArchiveInfo+</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td>Metadata</td>
<td>aggregation-Type, maxRetention.xFilesFactor,ArchiveCount</td>
</tr>
<tr>
<td></td>
<td>ArchiveInfo</td>
<td>Offset, SecondsPerPoint, Points</td>
</tr>
<tr>
<td>Data</td>
<td>Archive+</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Point+</td>
<td>times-tamp, value</td>
</tr>
</tbody>
</table>

Data types in Python’s struct format:

| Metadata     | !2LfL      |
| ArchiveInfo  | !3L        |
| Point        | !Ld        |
CHAPTER
THIRTEEN

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 dependant and can be found by executing (with Python 2.6 or later): python -c 'import sys; print sys.float_info'
TOOLS THAT WORK WITH GRAPHITE

14.1 Bucky

Bucky is 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.

14.2 collectd

collectd is a daemon which collects system performance statistics periodically and provides mechanisms to store the values in a variety of ways, including RRD. To send collectd metrics into carbon/graphite, use:

- Jordan Sissel’s node collectd-to-graphite proxy
- Joe Miller’s perl collectd-graphite plugin
- Gregory Szorc’s python collectd-carbon plugin
- Scott Sanders’s C collectd-write_graphite 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.

14.3 Collectl

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

14.4 Charcoal

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

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

14.6 Ganglia

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

14.7 GDash

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

14.8 Graphene

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

14.9 Graphite-relay

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

14.10 Graphite-Tattle

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

14.11 Graphiti

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

14.12 Graphitoid

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

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

14.14 Graphitejs

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

14.15 Grockets

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

14.16 HoardD

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

14.17 Host sFlow

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

14.18 hubot-scripts

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

14.19 jmxtrans

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

14.20 Logster

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

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

14.22 Rocksteady

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

14.23 Scales

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

14.24 Shinken

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

14.25 statsd

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

14.26 Tasseo

Tasseo is a lightweight, easily configurable, real-time dashboard for Graphite metrics.
WHO IS USING GRAPHITE?

Here are some organizations that use Graphite:

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

And many more
INDICES AND TABLES

- genindex
- modindex
- search
g

graphite.render.functions, 41