collectd and Graphite: Monitoring PostgreSQL With Style

Author: Shaun M. Thomas
Date: September 19th, 2014
Venue: Postgres Open 2014

Your Presenter

Shaun M. Thomas
Exclusive PostgreSQL DBA since 2005
Author of PostgreSQL 9 High Availability Cookbook

Why Graphical Monitors

- At a glance overview
- Trend detection
- Historical activity

This slide could really have a lot more in terms of justification, but that isn't the point. These three items tend to define the major role a graphical monitor will play in many organizations. Depending on our chosen zoom level, we may even be able to visualize the last two items simultaneously.

Our Stack

- collectd
- Graphite
- Grafana?

This stack isn't exactly simple thanks to the rather large list of prerequisites for Graphite. But without a PostgreSQL-focused visualization tool out there, it's one of the best and most configurable. Let's explore why.

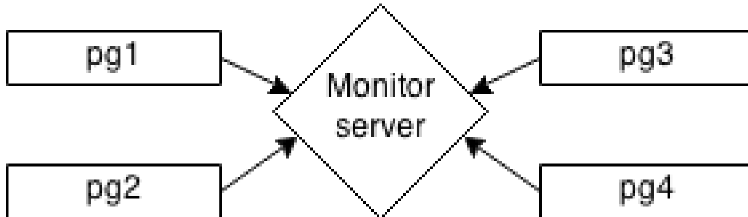
Why collectd?

- Distributed
- Customized
- Also, lots of plugins

We'll take a deeper look into collectd's distributed nature and how it can be customized in further slides. But what about the plugins? The way collectd works, is by acting as an invocation pipeline. Plugins can either be input or output based, while collectd defines various callbacks that it periodically activates. It works well as a data aggregation mechanism, and the plugins tend to leverage it well.

Why collectd: Distributed

Follows a client/server push model



Preventing overloaded monitor systems

The reason we're using collectd instead of statsd or some other well-known summary API, is due to the tiered aggregation collectd perpetuates. It strongly encourages separating data collection from final aggregation and presentation. As such, data collected locally on any or all of our PostgreSQL nodes can be independently transmitted to collectd for final tabulation or distribution. It's all in the name.

Since the data collection nodes operate asynchronously from the listener, the monitor statistics increase with every source we add. Provided the monitor server has sufficient reserve operational capacity, monitoring large clusters of servers is greatly simplified.

Why collectd: Customized

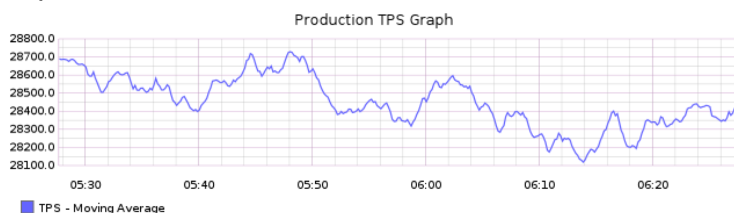
- Plugin control
- Arbitrary SQL statements
- Up to database granularity

Because of its plugin-driven nature, collectd can run with no plugins at all. If so, it probably just ticks away and periodically checks its callback function lists for new entries before sleeping. Or, we can activate every plugin and see if collectd can keep up with the demand. This flexibility is perfect for graphing important details of our PostgreSQL server nodes.

More pertinent to this discussion however, is the PostgreSQL plugin. With it, we can define several arbitrary queries to output whatever we want, so long as it is a quantifiable measurement. Each query can target a specific database on the server, or every database on every instance of the server. It's all up to us.

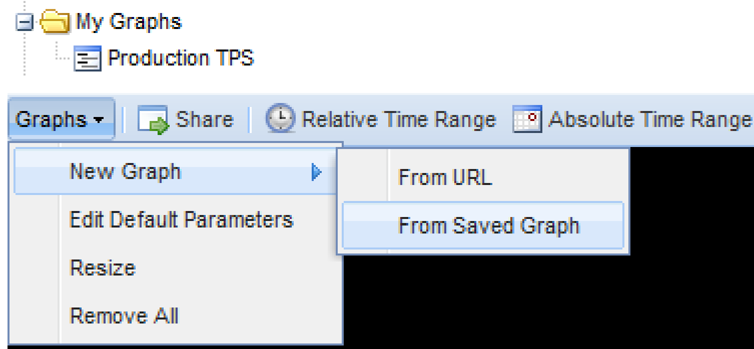
Why Graphite?

Graphs!



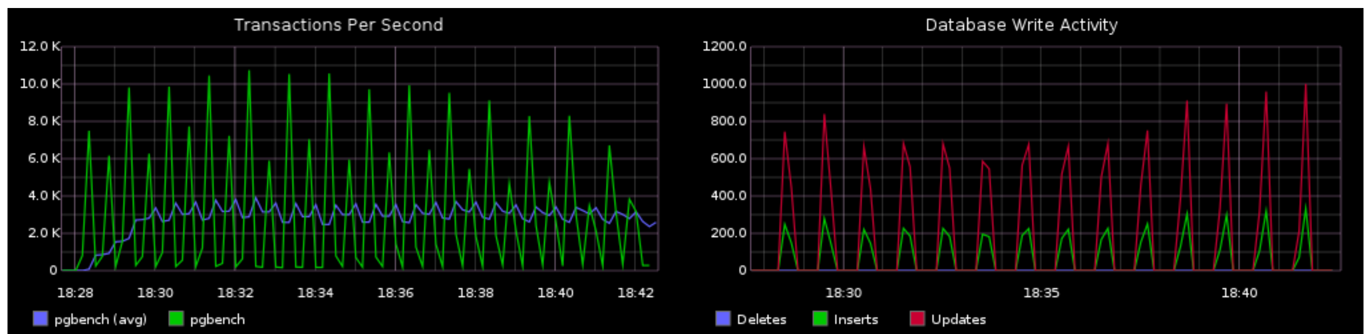
Really, Why Graphite?

Saved graphs!



C'mon, Why Graphite?!

Dashboards!



JUST TELL ME ABOUT GRAPHITE!

Ok, fine.

- Easily combine arbitrary measurements
- Apply data transformations
- API for receiving data points
- API for *sending* data points

The important thing to understand about Graphite is that it doesn't care about the data it's graphing. We could combine system load, measurements regarding the phase of the moon, and the count of queries currently executing in the QA environment, all within the same graph. This is both incredibly flexible, and monumentally dangerous if the scales are not comparable.

In addition, nearly every aspect of the graph and the data points can be customized. Line type, colors, widths; we can even apply data transformations such as moving standard deviations or logarithmic scaling. There is a lot of built-in functionality here.

The last two points are complementary. Like collectd, Graphite is *functional middleware*. The primary difference is that Graphite does not need direct control of its inputs. As such, collectd also has a Graphite plugin we can leverage to send data to our Graphite install from our collectd aggregator node. This means there are also several *further* pieces of software we could add to our stack, that are capable of consuming data from Graphite itself.

So, let's start building the stack and make use of all of this potential.

First Things First: Monitor Safely

- Define a separate user:

```
CREATE USER perf_user
  WITH PASSWORD 'testpw';
```

- Grant where necessary:

```
GRANT SELECT ON some_table
   TO perf_user;
```

While monitoring is done locally by our collectd daemon, we can further reduce exposure risk by locking down the user we leverage for gathering monitor data. Indeed, there are very few reasons *not* to do this. Try to err on the side of caution, especially since monitor users tend to be overly permissive.

Install collectd

Download the source, then:

```
tar -xzf collectd-5.4.1.tar.gz
cd collectd-5.4.1/
./configure --sysconfdir=/etc/collectd
make
make install
```

Why did we use the source? In the case of collectd, some Linux distributions are still using an old version that was packaged with a broken PostgreSQL plugin. The important thing is that we need 5.4.0 or above to be completely insulated from that possibility.

Splitting collectd Configuration

- collectd.conf
- network.conf
- local.conf
- postgresql.conf

We only really *need* the *collectd.conf* configuration file. But it's good practice to modularize. The above four files makes it very easy to control collectd with any number of configuration management suites like Puppet or Ansible.

The *network.conf* file basically acts as a toggle. We can configure the network plugin to either transmit data to an aggregator node, or listen for pushed data. Making it a separate file simplifies this choice.

The *local.conf* file serves a couple purposes we'll cover in more depth later. On the local PostgreSQL nodes, its only job is to include the *postgresql.conf* file that defines the PostgreSQL plugin actions. On the aggregator node, we configure the Graphite

plugin. This means many PostgreSQL collectd nodes will feed into a single collectd aggregate before finally being stored long term in Graphite's round-robin summary files.

Such is the chosen architecture for this demonstration. Other models are possible, of course, but we like this one, and use it in our own monitoring infrastructure.

collectd.conf For All Nodes

Start with a minimal configuration:

```
PIDFile "/var/run/collectd.pid"

LoadPlugin load
LoadPlugin syslog

Include "/etc/collectd/network.conf"
Include "/etc/collectd/local.conf"
```

The only really notable thing about this configuration is that we log to syslog instead of our own collectd log file. We don't have to do this, but it keeps our examples cleaner. To log to a separate file, replace the syslog LoadPlugin entry with this:

```
LoadPlugin "logfile"
<Plugin "logfile">
  LogLevel "info"
  File "/var/log/colletd/collectd.log"
  Timestamp true
</Plugin>
```

PostgreSQL Node network.conf

If the monitor server is *pgmon*:

```
LoadPlugin network
<Plugin network>
    Server "pgmon" "25826"
</Plugin>
```

There are other directives valid for the network plugin, but for now, we only need to worry about the *Server* entry. This directs the local collectd to transmit all data-points to the *pgmon* server on port *25826*.

Monitor Node network.conf

Use this on the aggregating monitor server:

```
LoadPlugin network
<Plugin network>
  Listen "*" "25826"
</Plugin>
```

As with the PostgreSQL nodes, we are being very terse. This time, we need the *Listen* directive. By using *, we want to monitor all network interfaces, and again, we're using port *25826*. Any collectd node can connect to this interface and transmit data-points.

PostgreSQL Node local.conf

PostgreSQL nodes should use this:

```
Include "/etc/collectd/postgresql.conf"
```

We can easily set up a server configuration management system like Puppet or Ansible to control which servers receive which configuration files. By keeping *local.conf* simple, we always know how each local server is operating in comparison to others.

Aggregate Node local.conf

Using *pgmon* for name of the server where Graphite will run:

```
LoadPlugin write_graphite
<Plugin write_graphite>
  <Node "pgmon">
    LogSendErrors true
    Prefix "collectd."
    StoreRates true
    SeparateInstances true
  </Node>
</Plugin>
```

These particular settings give us breadcrumb drill-down. This makes it much easier to find PostgreSQL-related collectd data entries in Graphite. The prefix also allows us to make collectd-specific configuration statements within Graphite as well.

Defining postgresql.conf

Anatomy of a PostgreSQL plugin entry:

```
LoadPlugin postgresql
<Plugin postgresql>
  <Query query-name>
    ... Stuff in here ...
  </Query>

  <Database postgres>
    ... Stuff in here ...
  </Database>
</Plugin>
```

After we load the *postgresql* plugin, we need to use pseudo-XML to define a few elements. We can define as many custom queries and database targets as we want between the *Plugin* tags.

postgresql.conf Query Sections

To register a query named TPS:

```
<Query tps>
  Statement "SELECT ...;"
  <Result>
    Type derive
    InstancePrefix "TPS"
    InstancesFrom "datname"
    ValuesFrom "tps"
  </Result>
</Query>
```

Several <Query> sections can be defined with this syntax. This is why we put everything in a single separate configuration file. We don't want to have one giant configuration file for collectd, as it's likely we'll have dozens of <Query> sections defined within the *postgresql* plugin.

The full query we're executing looks like this:

```
Statement "SELECT datname, \
                  xact_commit + xact_rollback AS tps \
             FROM pg_stat_database"
```

We need the backslashes to escape each newline, since those are not otherwise allowed in collectd configuration entries. This helps keep our custom queries human-readable. Note that **datname** corresponds to the database name for each database on the local PostgreSQL instance. This means we have a separate TPS data-point for each database. This will be reflected in the collectd drill-down later.

postgresql.conf Database Sections

To run the TPS query on the *postgres* database:

```
<Database postgres>
    Host "localhost"
    User "perf_user"
    Password "testpw"
    Instance "Production"
    Query tps
</Database>
```

Why the **Instance** line? If we have a large PostgreSQL cluster, there may be several hosts that each have databases with identical names. For instance, if our application uses *appdb* as the name of the database, we wouldn't be able to easily differentiate between our development and QA instances.

For each **Query** line, we can assign any of the previously defined *Query* definitions to this database entry. Again, this makes it much easier to control which queries are valid for which hosts as defined by configuration management tools.

Starting collectd

Use one of the scripts from contrib…

- Upstart: upstart.collectd.conf
- init-d: redhat/init.d-collectd
- systemd: collectd.service

The High Availability Cookbook has one too.

This is a choice that is most likely dependent on the Linux distribution we have on our servers, but most of the major players are represented. No matter which init system is in place, we need to use whichever is actually functional when given a *start* command. The *redhat* contrib script for example, will not work on a Debian system because it depends on functions defined only on Redhat or CentoS systems.

For what it's worth, the collectd init script supplied in the PostgreSQL 9 High Availability Cookbook is self-contained, so it should work on nearly any Linux system.

About Graphite

- Django project
- Lots of dependencies
- Aggregator node, or elsewhere

Graphite is a Django project. As such, its installation and activation can be somewhat fragile. Of particular note, is that Django revised its function exports after version 1.6. This means the version of Django packaged with our Linux distribution can adversely affect our Graphite installation. To be safe, try to use Django 1.5 unless you install directly from the Graphite sources.

Unfortunately, Graphite also has *numerous* prerequisite packages that make it somewhat annoying to install manually. Even more unfortunately, the version of Graphite available in the Python Package Index is not compatible with Django 1.6 and above, meaning its generally not suitable for newer Linux distributions. This makes resolving the required software list rather onerous.

Due to the architecture we advocate, collectd will be transmitting collected data to Graphite. This can either be done locally on the same monitoring server, or the collectd aggregate node can be physically separate from wherever Graphite is running. In either case, make note of server names and check to ensure they're set properly on each link in the chain.

Basicaly, if Graphite doesn't work, download it from the primary download site instead of pip. If the necessary preliminary software is installed, that version should work normally.

Install Graphite Vitals

- RHEL: python-pip, django, cairo, python-devel
- Deb: python-pip, python-django, python-cairo, python-dev
- May need EPEL for RHEL/CentOS systems.

Django, and Cairo, and PIP, oh my! Basically it boils down to these packages. We will also need some of the python development headers for the pip command-line tool to work properly.

Install Graphite Parts

Install all of this stuff:

```
sudo pip install graphite-web
sudo pip install carbon Twisted==11.1
sudo pip install whisper
```

In the Python Package Index, graphite-web is the web GUI portion of Graphite. Whisper is the storage format we will use by default. So what is carbon? It's the network API portion that collects data sent to Graphite for retention in whatever storage format is configured. Again, this is whisper by default.

The pip tool allows us to include dependencies by version. In this case, we specifically called for Twisted version 11.1. There's an unfortunate incompatibility in newer versions of Twisted, so we have to revert to an older one for Graphite to work for now. We'd also like to point out that this slide fixes a typo in the PostgreSQL 9 High Availability Cookbook; it only used one equal sign, which is a syntax error.

Once everything is installed, it's time to configure.

Configure the Graphite Web App

- Copy the default Graphite Django settings file and activate:

```
cd /opt/graphite/webapp/graphite
mv local_settings.py.example local_settings.py
```

- Then change *SECRET_KEY* in *local_settings.py*
- Finally, activate the web-app configuration:

```
python manage.py syncdb
```

The web application is based on Django. As such, we need to have a *local_settings.py* so Django can instantiate the database, secret key, or any other elements we can configure for such an application. For now, we'll let Django use the default sqlite storage for its own metadata. The only setting we actually need to change to prevent Django from complaining is the *SECRET_KEY* value, which it uses internally to secure the platform.

Then we use the Django manage.py to actually define the metadata database, and handle any other behind-the-scenes elements necessary to bootstrap our Graphite setup. This command will ask us to define a root Graphite user as well. We can log into the GUI with this user, and also use the Django administrative console to add further users.

It's possible to use LDAP or Active Directory for these users as well, but that's beyond the scope of this talk. Feel free to find this information in the Django or Graphite documentation.

Configure Graphite Backend

Define the default Graphite config files:

```
cd /opt/graphite/conf
mv carbon.conf.example carbon.conf
mv storage-schemas.conf.example storage-schemas.conf
```

We're obviously glossing over quite a bit of configuration, here. For now, the defaults are quite serviceable. This is the minimum to get Graphite up and running in a way that fits this demonstration. From here, it should not be excessively difficult to customize the installation.

Tweak Graphite Storage Settings

Add this to the *beginning* of storage-schemas.conf:

```
[collectd]
pattern = ^collectd\.
retentions = 10s:1d,1m:7d,5m:30d,10m:90d,1h:1y
```

The default data retention periods for Graphite are actually pretty bad for our use case. The setting we used here will retain data points for much longer than the default values. For instance, we keep ten-second data granularity for one full day. For weekly views, we can only see metrics at the minute level. Then for a month, our resolution falls further to a five minute interval. At three months, we see a data point every ten minutes. Finally at the year level, we only see hourly measurements.

Feel free to tweak these, but keep in mind the Whisper storage system Graphite uses is very similar to RRD. By that, we mean the data collection files are defined upon the first measurement saved based on the retention period we defined. The file size for each retention file will grow as the number of data-points increases. You may find our suggested retention periods are actually too granular if you are receiving data from hundreds of hosts. Just make sure to monitor the storage of the monitor server and take action if it can't keep up with the write load from saving all the incoming measurements.

As for the reason we add this section to the end of the storage-schemas.conf file, Graphite defines the data retention in the order defined in this file. As such, it would use the default of one reading per minute for a single day before it encountered our override. That's definitely not what we want!

Start Graphite

Start the Graphite storage engine and GUI:

```
cd /opt/graphite/bin
sudo ./carbon-cache.py start
sudo su -c "./run-graphite-devel-server.py \
     /opt/graphite &> /var/log/graphite.log &"
```

Graphite is actually comprised of two parts. One receives data and directs it to the round-robin storage files. The second reads the storage files and translates them to graphs depending on which data sources are requested by drilling down in the GUI.

Other Queries: Write Activity

Let's see our database writes:

```
Statement "SELECT sum(n_tup_ins) AS inserts, \
                  sum(n_tup_upd) AS updates, \
                  sum(n_tup_del) AS deletes \
             FROM pg_stat_user_tables"
```

This is a statement line we could use for displaying database write activity. In order for this to work, we actually need to do some decoding with collectd structures. The full **Query** block would look more like this:

```
<Query row_stats>
  Statement "SELECT sum(n_tup_ins) AS inserts, \
                    sum(n_tup_upd) AS updates, \
                    sum(n_tup_del) AS deletes \
               FROM pg_stat_user_tables"

  <Result>
    Type derive
    InstancePrefix "rows_inserted"
    ValuesFrom "inserts"
  </Result>
  <Result>
    Type derive
    InstancePrefix "rows_updated"
    ValuesFrom "updates"
  </Result>
  <Result>
    Type derive
    InstancePrefix "rows_deleted"
    ValuesFrom "deletes"
  </Result>
</Query>
```

Other Queries: Replication

How about replication lag:

```
Statement "SELECT coalesce(round(
               extract(epoch from (now() - \
               pg_last_xact_replay_timestamp()))),
               0) AS time_lag"
```

This is a statement line we could use for displaying the number of seconds a replica is behind its assigned source. Here's the full block:

```
<Query stream_lag>
  Statement "SELECT coalesce(round(extract(epoch from (now() - \
           pg_last_xact_replay_timestamp()))),0) AS time_lag"

  <Result>
    Type gauge
    InstancePrefix "replication_stream_lag"
    ValuesFrom "time_lag"
  </Result>
</Query>
```

We use a *gauge* type because this measurement represents an absolute value, and not one that slowly increments like most of the pg_stat_* tables. As always, we change the prefix to disambiguate, and values sent to collectd are supplied by the *time_lag* column alias we made. Keep in mind that this will only work on PostgreSQL instances currently undergoing replication.

Other Queries: Client State

Even a connection summary:

```
Statement "SELECT count(1) AS total, \
           sum(CASE WHEN state LIKE 'idle in%' \
               THEN 1 ELSE 0 END) AS trans_idle, \
           sum(CASE WHEN state = 'active' \
               THEN 1 ELSE 0 END) AS active, \
           sum(waiting::INT) AS waiting, \
           sum(CASE WHEN now() - query_start > \
               INTERVAL '1s' AND state = 'active' \
               THEN 1 ELSE 0 END) AS slow \
           FROM public.pg_stat_activity()"
```

When we said arbitrary, we meant it. Imagine wanting to see totals of how many clients are connected, and the basic state of their connection. We can watch for spikes in waiting connections, or an excessive amount sitting around in idle transactions, or even just set thresholds for slow queries. The full **Query** block looks like this:

```
<Query client_activity>
  Statement "SELECT count(1) AS total, \
             sum(CASE WHEN state = 'idle' \
                      THEN 1 ELSE 0 END) AS idle, \
             sum(CASE WHEN state LIKE 'idle in%' \
                      THEN 1 ELSE 0 END) AS trans_idle, \
             sum(CASE WHEN state = 'active' \
                      THEN 1 ELSE 0 END) AS active, \
             sum(waiting::INT) AS waiting, \
             sum(CASE WHEN now()-query_start > INTERVAL '1s' \
               AND state = 'active' THEN 1 ELSE 0 END) AS slow \
             FROM public.pg_stat_activity();"

  <Result>
    Type gauge
    InstancePrefix "clients_total"
    ValuesFrom "total"
  </Result>
  <Result>
    Type gauge
    InstancePrefix "clients_idle"
    ValuesFrom "idle"
  </Result>
  <Result>
    Type gauge
    InstancePrefix "clients_idle_transaction"
    ValuesFrom "trans_idle"
  </Result>
  <Result>
```

```
    Type gauge
    InstancePrefix "clients_active"
    ValuesFrom "active"
  </Result>
  <Result>
    Type gauge
    InstancePrefix "clients_waiting"
    ValuesFrom "waiting"
  </Result>
  <Result>
    Type gauge
    InstancePrefix "clients_slow"
    ValuesFrom "slow"
  </Result>
</Query>
```

We could add several more data points here if we really wanted, but this is enough for now. This query also makes use of a *pg_stat_activity()* function that may seem somewhat odd at first glance, since it's not part of the PostgreSQL core set of functions. The pg_stat_activity view does not allow non superusers to see all query activity, and we don't want our monitor user to have that kind of power. So we can circumvent that restriction like this:

```
CREATE OR REPLACE FUNCTION pg_stat_activity()
RETURNS SETOF pg_stat_activity AS $$
  SELECT * FROM pg_stat_activity;
$$ LANGUAGE sql SECURITY DEFINER;

GRANT EXECUTE ON FUNCTION pg_stat_activity() TO perf_user;
```

So long as the function is created by a superuser, our monitor user can utilize it. If allowing all columns is too permissive, we can easily pare down the list as well, or add a where clause. Why not a view instead? The *pg_stat_activity* structure *is* a view, and it uses a function that checks the currently authorized user. This is the only way to break out of the system-imposed shell.

Demo

- Periodic pgbench running in background.
- Play around with Graphite, saved graphs, dashboard.

More Information

- Download collectd
- collectd Documentation
- collectd Data source
- collectd PostgreSQL Plugin
- Graphite Wiki
- Updated Graphite Documentation

Questions