pg_jobmon

pg_jobmon is a PostgreSQL extension designed to add autonomous logging capabilities to transactions and functions. The logging is done in a NON-TRANSACTIONAL method, so that if your function/transaction fails for any reason, any information written to that point will be kept in the log tables rather than rolled back.

For more information on how to use pg_jobmon, please see pg_jobmon/doc/pg_jobmon.md

INSTALLATION

Requirements: PostgreSQL 9.2+, dblink extension

In the directory where you downloaded pg_jobmon run

```
make
make install
```

Log into PostgreSQL and run the following commands: (Note: You can change the schema name to be whatever you wish, but it cannot be changed after installation.)

```
CREATE SCHEMA jobmon;
CREATE EXTENSION pg_jobmon SCHEMA jobmon;
```

This extension uses dblink to connect back to the same database that pg_jobmon is running in (this is how the non-transactional magic is done). To allow non-superusers to use dblink, you'll need to enter role credentials into the dblink_mapping_jobmon table that pg_jobmon installs.

```
INSERT INTO jobmon.dblink_mapping_jobmon (username, pwd) VALUES ('rolename', 'rolepassword')
```

Ensure you add the relevant line to the pg_hba.conf file for this role. It will be connecting back to the same postgres database locally.

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD
local   dbname          rolename                                md5
```

The following permissions should be given to the above role (substitute relevant schema names as appropriate):

```
grant usage on schema jobmon to rolename;
grant usage on schema dblink to rolename;
grant select, insert, update, delete on all tables in schema jobmon to rolename;
grant execute on all functions in schema jobmon to rolename;
grant all on all sequences in schema jobmon to rolename;
```

If you're running PostgreSQL on a port other than the default (5432), you can also use the dblink_mapping_jobmon table to change the port that dblink will use.

```
INSERT INTO jobmon.dblink_mapping_jobmon (port) VALUES ('5999');
```

Be aware that the dblink_mapping_jobmon table can only have a single row, so if you're using a custom host, role or different port, you will need to enter those values within a single row. None of the columns are required, so just use the ones you need for your setup.

```
INSERT INTO jobmon.dblink_mapping_jobmon (host,username, pwd, port) VALUES ('host','rolename
```

UPGRADE

Make sure all the upgrade scripts for the version you have installed up to the most recent version are in the $BASEDIR/share/extension folder.

```
ALTER EXTENSION pg_jobmon UPDATE TO '<latest version>';
```

For detailed change logs of each version, please see the top of each update script.

LICENSE AND COPYRIGHT

 pg_jobmon
==========

LOGGING

add_job(p_job_name text) RETURNS bigint
Create a new entry in the job_log table. p_job_name is automatically capitalized.
Returns the job_id of the new job.

add_step(p_job_id bigint, p_action text) RETURNS bigint
Add a new step in the job_detail table to an existing job. Pass it the job_id created by add_job() and a description of the step.
Returns the step_id of the new step.

update_step(p_step_id bigint, p_status text, p_message text) RETURNS void
Update the status of the step_id passed to it.
p_message is for further information on the status of the step.

sql_step(p_job_id bigint, p_action text, p_sql text) RETURNS boolean
Logs a full step for a given sql query. Logs the number of rows affected. Simple function for when you don't need to log extensive details of a single query.
p_job_id is the job id to log this step with.
p_action is for the action column in job_detail.
p_sql is the full sql query to be run Returns true/false so you can check how the step ran and continue handling the rest of your job appropriately.

close_job(p_job_id bigint) RETURNS void
Used to successfully close the given job_id.

fail_job(p_job_id bigint, p_fail_level int DEFAULT 3) RETURNS void
Used to unsuccessfully close the given job_id.
Will default to failing with a level 3 alert in the log. Optional second argument allows you to fail a job with a different error level
(Ex. close job w/ level 2 (WARNING) to avoid a critical page but still notify that something's wrong).

cancel_job(v_job_id bigint) RETURNS boolean
Used to unsuccessfully terminate the given job_id from outside the running job.
Calls pg_cancel_backend() on the pid stored for the job in job_log.
Sets the final step to note that it was manually cancelled in the job_detail table.

sql_job(p_job_name text, p_sql text) RETURNS text
Log a complete job for a given query. Records the number of rows affected. Also ensures that the job cannot run concurrently with itself.
p_job_name is the job name that will be recorded in job_log.
p_sql is the full sql query to be run.
Returns the job id that was created for this job and whether the job was successful.

job_log_clear(p_interval interval) RETURNS void
Deletes all jobs in job_log and job_detail tables older than the given interval. Also logs this task as another job itself.

The below functions all return full rows of the format for the given SETOF table, which means you can treat them as tables as far as filtering the result. For all functions that have a default integer parameter at the end, this signifies a default limit on the number of rows returned. You can change this as desired, or just leave out that parameter to get the default limit.

All show functions also automatically uppercase the job name parameter to be consistent with the add_job function.

show_job(p_name text, int default 10) RETURNS SETOF job_log
Return all jobs from job_log that match the given job name. Automatically sorts to get the most recent jobs first, so by default gets the first 10 matching jobs

show_job_like(p_name text, int default 10) RETURNS SETOF job_log
Return all jobs from job_log that contain the given text somewhere in the job name (does a ~ match). Automatically sorts to get the most recent jobs first, so by default gets the first 10 matching jobs

show_job_status(p_status text, int default 10) RETURNS SETOF job_log
Return all jobs from job_log matching the given status. Automatically sorts to get the most recent jobs first, so by default gets the first 10 matching jobs

show_job_status(p_name text, p_status text, int default 10) RETURNS SETOF job_log
Return all jobs from job_log that match both the given job name and given status. Automatically sorts to get the most recent jobs first, so by default gets the first 10 matching jobs

show_detail(p_id bigint) RETURNS SETOF job_detail
Return the full log from job_detail for the given job id.

show_detail(p_name text, int default 1) RETURNS SETOF job_detail
Return the full log from job_detail matching the given job name. By default returns only the most recent job details.
Given a higher limit, it will return all individual job details in descending job id order.

show_running(int default 10) RETURNS SETOF job_log
Returns data from job_log for any currently running jobs that use pg_jobmon. Note that if there are any jobs with a NULL "status" with the same pid as anything currently running in pg_stat_activity, this function may return false results. Fix unfinished jobs to prevent this from happening.

Log Tables:

IMPORTANT NOTE
The job_log and job_detail table data WILL NOT be exported by a pg_dump. The extension system still has some issues to be worked out, one of them being that if you do a full database dump specifying –schema-only, any extension table data that has been set to be dumped will be (see catalog.pg_extension_config_dump() function). As you can imagine, over time these two tables can get quite large and that could cause a schema dump of the database to be needlessly large. Until that is fixed, these tables have had their dump setting specifically left off. If you need to dump these tables' data, please DROP the table from the extension, do your dump, then ADD the table back

(http://www.postgresql.org/docs/9.1/static/sql-alterextension.html). The other tables have minimal, and more critical data, so their data has been set to be dumped.

job_log
Logs the overall job details associated with a job_id. Recommended to make partitioned on start_time if you see high logging traffic or don't need to keep the data indefinitely

job_detail Logs the detailed steps of each job_id associated with jobs in job_log. Recommended to make partitioned on start_time if you see high logging traffic or don't need to keep the data indefinitely

dblink_mapping_jobmon Configuration table for storing dblink connection info. Allows non-superusers to use this extension and changing the port of the cluster running pg_jobmon. This table enforces there only being a single row in it. See README.md file for usage during installation.

MONITORING

check_job_status(OUT alert_code int, OUT alert_status text, OUT job_name text, OUT alert_text text) RETURNS SETOF record

Function to run to see if any jobs are having problems. No argument needs to be passed and it will automatically use the longest configured threshold interval from the job_check_config table as the period of time to check for bad jobs (see below). If nothing is configured in that table, it just checks for 3 consecutive job failures.

The **alert_code** output indicates one of the following 3 statuses. Additionally configured, custom alert codes are not currently supported by this function:
* Return code 1 means a successful job run
* Return code 2 is for use with jobs that support a warning indicator. Not critical, but someone should look into it * Return code 3 is for use with a critical job failure

This monitoring function was originally created with nagios in mind, hence these text alert values. By default, the job_status_text table contains the following:

```
1 = OK
2 = WARNING
3 = CRITICAL
```

If you'd like different status text values, just update the job_status_text table with the error_text values you'd like. DO NOT change the alert_code values, though, if you want to use the check_job_status() function! And while you're free to add additional code values for different statuses you may need, the core monitoring code always assumes that the alert_code for a job completing successfully is 1. If you change that, the extension's monitoring capabilities may not act in a 100% predictable manner.

The **alert_status** output is a simple, small string indicator as to what is wrong. Example outputs are: FAILED_RUN, MISSING, BLOCKED, RUNNING. Note that a job will only report the MISSING, BLOCKED or RUNNING status if it has been added to the job_check_config table for additional monitoring outside the default 3 consecutive failures.

The **job_name** output is the job_name recorded in the job_log table.

The **alert_text** output is a more detailed message about what the problem may be.

You can filter the output however is easiest for your monitoring solution. A simple example for nagios (with good and bad output) to get the highest alert code status along with some additional info on one of the jobs currently having issues is:

```
select t.alert_text ||'('||COALESCE(c.job_name||': ', '')||c.alert_text||')' as alert_status
from jobmon.check_job_status() c
join jobmon.job_status_text t on c.alert_code = t.alert_code limit 1;

         alert_status
------------------------------
 OK(All jobs run successfully)


                      alert_status
-------------------------------------------------------------------
 CRITICAL(PG_JOBMON TEST BAD JOB: 1 consecutive job failure(s))
```

Here's a more advanced query that will still give you the highest alert_code text at the beginning of the result, but also give all the job details for every job that currently has an issue, or provide the same All Clear message as above.

```
select t.alert_text||'('||l.job_list||')' as alert_status from (select max(alert_code) as al
```

The output of check_job_status() hopefully gives you enough flexibility to allow you to monitor results as you need them.

Monitoring Tables:

job_check_config
Table of jobs that require special job monitoring other than 3 consecutive failures (this is done by default). * job_name - This is the EXACT job name as it appears in the job_log table. It is case sensitive, so jobs entered here should be in all caps. Easiest thing to do is just copy the job name right from the job_log table. * warn_threshold - This is the interval of time that can pass without the job running before alert_code 2 is returned by check_job_status() * error_threshold - This is the interval of time that can pass without the job running before alert_code 3 is returned by check_job_status() * active - Set this to TRUE if you want check_job_status() to actively monitor this job.

Set to FALSE to disable checking without removing the data from the config table * sensitivity - This is the number of times the job can fail (status column in job_log is the text value of alert_code 3, CRITICAL by default) before alert_code 3 is returned by check_job_status(). Note that if you want this to return immediately on the first failure, set it to zero, not one. * escalate - How many times a job can fail at each alert level before it is escaleted to the next highest available alert level. See ALERT ESCALATION section below for more.

job_check_log
This table is used to record the job_id, job_name & alert_code automatically whenever the status column of job_log contains the text value for alert levels other than 1. You should hopefully never have to insert or delete from this table manually. A trigger on job_log handles this. However, if a job fails and can never be run successfully again, you may have to manually clean the table to clear out the bad job from returning in check_job_status(). It's recommended NOT to truncate to clean it up as you could inadvertently delete a job that had just failed. It's best to delete the job number specifically or delete all jobs <= a specific job_id. If you're not going to use the check_job_status() function, this is the table you want to watch for anything causing problems and raising alerts other than 1.

job_status_text
Table containing the text values for the alert levels. Defaults are listed above. Change the alert_text column for each code to have custom statuses used for the status column in job_log. The default alert_codes (1,2,3) are used by check_job_status() and if changed, that monitoring function can no longer be used. Even if custom alert codes are used, alert_code 1 should always be used for a successfully completed job. All other alerts can be customized as needed.

ALERT ESCALATION

PG Jobmon supports alert escalation where if a job fails a given number of times at a certain alert level, it will automatically be escalated to the next highest alert_code found in the job_status_text table. Set the **escalate** column found in the **job_check_config** table for the job you want escalation on to the number of times a job should cause an alert on each level before being raised to the next. For example, if you want the alert level raised to 3 (CRITICAL by default) after issuing the level 2 (WARNING by default) alert three times, set the escalate value to 3. If you have added any custom alert codes higher than this, it will do the same escalation at each level.

Escalation is not turned on by default and is still slightly experimental at this time. Looking for feedback on this, good or bad. Even just an "It works!" would be appreciated.

FURTHER READING

The pg_jobmon extension was created based on a series of ad-hoc implementations used on various OmniTI clients. Once PostgreSQL added the extensions format, this work was standardized into the tool set you see now. The principle developer during that time was Keith Fiske, who blogged about his work, including several articles showing more detailed use of the extension, at (http://www.keithf4.com/tag/pg_jobmon/).