PgQuartz

Release 0.2

Sebastiaan Mannem

Mar 24, 2023

CONTENTS:

PgQuartz is a framework for scheduling PostgreSQL workload across PostgreSQL clusters with an etcd consensus mechanism. PgQuartz jobs can be defined in yaml format as a series of steps consisting of a series of commands and the checks to verify results.

CHAPTER

ONE

INTRODUCTION

The goal of PgQuartz is to *schedule* jobs against a Primary/Standby cluster with minimal toil. We can best describe 'minimal toil' with the alternative: cron and bash scripts.

Although we consider *systemd/timer* and *Kubernetes cron* more mature *scheduling* solutions, PgQuartz is not trying to replace *cron* as a scheduling solution. Which means you can just as easily schedule PgQuartz using *cron* as you could schedule your original bash scripts.

PgQuartz is trying to replace the bash scripts to some extent, mainly for al the generic things like:

- Orchestration of jobs across a Postgres cluster
  - Run on a master, a standby, or on any node
  - Run on only one node, run on every node one at the time, or run on all in parallel
- Reuse of database connections
- Separation of code and configuration
- Separation of run and check

PgQuartz is not trying to replace the actual (application specific) definitions of your (bash/sql) scripts. PgQuartz just brings a framework where they can be defined with minimal toil without being required to define all stuff the framework does for you.

By leveraging PgQuartz, basically your jobs get a clearer definition, less error-prone on generic tasks and more efficient on runtime.

For more information, please see our documentation at readthedocs.io.

CHAPTER

TWO

SCHEDULING

2.1 Systemd/Timer

When scheduling jobs on VM Deployed clusters, the PgQuartz team advices using the Systemd/Timer units over cron entries. With the Systemd/Timer implementation, every job (defined by a job config file) is defined as a separate service unit, and is scheduled with a systemd timer unit. This means that every job consists of:

- A [job].yml file which defines the job
  - Including any required scripts that are part of this job, but they can be shared across jobs
- A [job].service unit file (`oneoff`) in /etc/systemd/system/ which runs PgQuartz as the `pgquartz` user specifying the path to the job.yml file
- A [job].timer unit file in /etc/systemd/system/ which triggers the service to run the job at specified moments in time

The beauty of the setup is that every job gets to be a uniquely identifiable unit, and all metadata on that job (last execution result, logging) gets to be a part of that unit. Moreover, generic stuff, like log space recycling, runs of all jobs, etc. can be managed by the generic solution that systemd has to offer.

2.1.1 Defining a systemd service and timer

Defining a PgQuartz job can be easily achieved by creating a service and timer using below templates (modify as required for your use case):

Service: `/etc/systemd/system/pgquartz_myjob.service`:

```
# This service unit is for running a pgQuartz job 'as a service'.
# PgQuartz job services can be triggered by systemd timers for scheduling

[Unit]
Description=Run a pgQuartz job

[Service]
Type=oneshot
Environment=PGQUARTZ_CONFIG=/etc/pgquartz/jobs/myjob.yml
ExecStart=/usr/local/bin/pgquartz
User=pgquartz

[Install]
WantedBy=multi-user.target
```

Timer: /etc/systemd/system/pgquartz_myjob.timer:

```
# This timer unit is for scheduling a pgQuartz job 'as a service'.

[Unit]
Description=Run the pgQuartz myjob Daily at quarter past 11 pm.
Requires=pgquartz_myjob.service

[Timer]
Unit=pgquartz_myjob.service
OnCalendar=*-*-* 23:15:00

[Install]
WantedBy=timers.target
```

2.2 Cron

The de facto solution for scheduling scripts has always been cron, and of course cron is also supported by PgQuartz. Using cron has some minimal cons against systemd, but they are negligible in most cases:

- you need to manage log files.
  - Easiest is to specify a log folder as logFile, and issue a logrotate file for cleaning
- you need to set up mail for alerting
  - PgQuartz logs errors to stderr, so redirecting stdout to /dev/null allows for only being mailed on errors
  - setting up a logFile location allows for seeing a combined stderr and stdout error trail when needed

Scheduling a pgQuartz job with cron gets to be as easy as:

Cron: /etc/cron.d/pgquartz:

```
MAILTO = me@example.com
15 23 * * * pgquartz /usr/local/bin/pgquartz -c /etc/pgquartz/jobs/myjob.yml >/dev/null
```

LogRotate: /etc/logrotate.d/pgquartz:

```
/var/log/pgquartz/pgquartz.log {
    missingok
    ifempty
    daily
    compress
    rotate 7
}
```

2.3 Kubernetes cron

PgQuartz is designed to be run as a Kubernetes cron job, which could be as easy as:

- Create an image with PgQuartz
- Supply the job definition in a ConfigMap
- Schedule a kubernetes cron job to run the Job Pod at required times

We are working on building an image and defining a more verbose runtime example in Issue #40.

CHAPTER

THREE

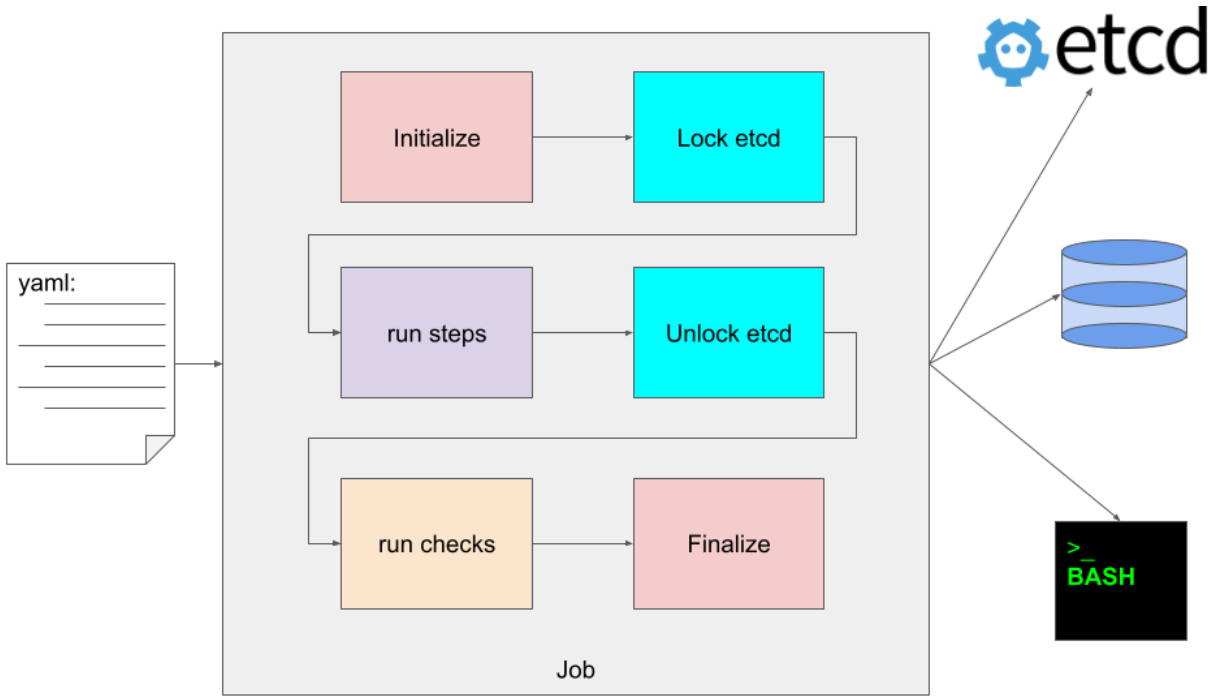
PGQUARTZ JOBS

Jobs are the highest level of configuration within PgQuartz. When you run PgQuartz, you pass it a configuration file and all configuration in that file might be defined as 'the job'.

Jobs basically consist of:

- *Steps*, *instances* and *commands*
- *checks*
- connection config
  - *PostgreSQL connections*
  - *etcd*
- Generic config (laid out below)

3.1 Graphical representation



3.2 Generic job config chapters

The following configuration can be set at the top level:

3.2.1 debug

Be more verbose. Debug mode can also be enabled at commandline with the -d argument

3.2.2 git

If PgQuartz detects that the job is defined in a git repository, PgQuartz will pull the latest version and reload the config before running the job. In the git chapter some config can be configured to control this git pull behaviour. The following options can be defined:

- remote: The remote to pull from. Defaults to `origin`.
- rsaPath: The rsa private key to use when pulling from an ssh remote
- httpUser / httpPassword: The user / password to use when pulling from a http(s) remote. Can also be set as part of the remote url.
- disable: Disable the pull feature

3.2.3 logFile

PgQuartz logs errors to stderr and other messages to stdout. By setting a logFile, PgQuartz additionally writes logging to the destination file. If logFile points to a directory, PgQuartz creates a file in that directory with a predefined filename consisting of the current date, and the job name. ***note*** that the job name is derived from the yaml that defines the job (e.a. `/etc/pgquartz/jobs/job1.yaml` would result in a job name `job1`)

3.2.4 parallel

PgQuartz has implemented parallelism with regard to:

- runs multiple instances of a step in parallel (see *instances* for more info).
- runs multiple steps in parallel when it can (see *steps configuration* for more info) The parallel setting configures the number of runners which defines the number of parallel tasks run by PgQuartz

3.2.5 runOnRoleError

Connections can be defined with a role. When the configured (expected) role does not match the actual role, PgQuartz exits with an error. By setting `runOnRoleError=true`, PgQuartz continues processing, and skips commands against a connection with unexpected role.

3.2.6 timeout

Connection operations, like locking in etcd and running PostgreSQL queries run within a context. The timeout parameter times out this context and as such acts as a generic timeout for the entire job. When the timeout exceeds all running operations are cancelled and PgQuartz quits with an error message and error exit code.

3.2.7 workdir

The workdir from where all scripts are loaded. This parameter defaults to the location of the job definition file and can usually be left out.

3.3 Example config

```
debug: true
git:
  remote: origin
  rsaPath: ~/.ssh/id_rsa
  httpUser: memyselfandi
  httpPassword: secret
  disable: false
logFile: /var/log/pgquartz/pgquartz.log
parallel: 2
runOnRoleError: true
timeout: 1h
workdir: /etc/pgquartz/jobs/job1/

# steps need to be configured. Please see https://github.com/MannemSolutions/PgQuartz/
↪docs/STEPS.md for more information on the definition
steps: {}
# checks should be configured. Please see https://github.com/MannemSolutions/PgQuartz/
↪docs/CHECKS.md for more information on the definition
checks: []
# connections should be configured. Please see https://github.com/MannemSolutions/
↪PgQuartz/docs/CONNECTIONS.md) for more information on the definition
connections: {}
```

CHAPTER

FOUR

STEPS

Steps are the basic building blocks out of which *Jobs* are defined.

4.1 Some facts about Steps

- Steps are the building basic building blocks at the center of the PgQuartz solution.
- When processing a *job* PgQuartz will first run all Steps (the actual work at hand), and then run *Checks* to verify results.
- Steps are built out of *Commands* which are run in order until one of them fails.
  - When a Command fails, that Step stops processing *Commands* and enters a failed state
  - Steps entering a failed state does not impact other Steps and/or *Checks* unless you configure it so.
- Steps are all run in parallel unless dependencies between steps are configured
- Steps can be run multiple times with different arguments by specifying a *matrix* of all arguments
  - Every combination of arguments is labelled a Step Instance
  - All instances of a step are run in parallel as well

This boils down to a PgQuartz *job* definition where everything that can be run in parallel is defined as separate steps, or (*matrix*) arguments on a step.

4.2 Configuration options

4.2.1 Dependencies

Although steps by default are independent units scheduled to be run in parallel, dependencies between steps can be configured with a dependency setting. When dependencies are configured, the step will not be scheduled before dependencies have been run and finished first.

> **Note** that dependencies are not automatically added for When resolution. All steps referenced in When rules should be added as manual dependencies as well. See #42 for more information.

4.2.2 When

Next to dependencies (wait for other step to finish) we can also configure checks on results of other jobs. We can them when-rules, and they should be configured in the `when:` argument of the step whom should check before running. If one or more rules don't check out to be successful, the step is not scheduled, but moves to `Done` state directly. For more information, please refer to *when*

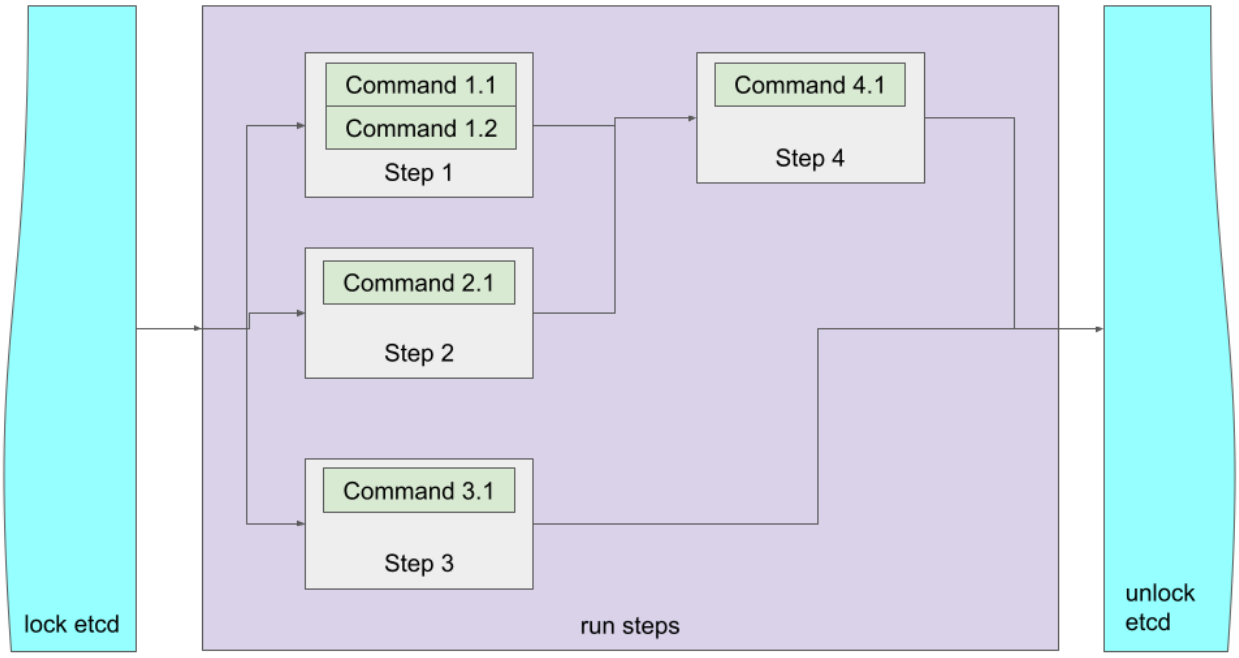
4.3 Example

We make the 'Steps concept' more tangible with an example:

4.3.1 Example config

```
steps:
  step 1:
    commands:
      - name: Run command 1.1
        type: pg
        file: ./sql/step_1.1.sql
      - name: Run command 1.2
        type: shell
        file: ./bash/step_1.2.sh
  step 2:
    commands:
      - name: Run command 2.1
        type: pg
        file: ./sql/step_2.1.sql
  step 3:
    commands:
      - name: Run command 3.1
        type: shell
        file: ./bash/step_3.1.sh
  step 4:
    commands:
      - name: Run command 4.1
        type: pg
        file: ./sql/step_4.1.sql
    depends:
      - step 1
      - step 2
    when:
      - "eq .Steps["step 1"].Instances.Rc 0"
      - '.Steps.["step 2"].Instances.StdOut.Contains "{oid}={13694}"'
parallel: 2
```

4.4 Graphical representation



4.4.1 What does it do?

When running a job with a specification as shown in the *example*, PgQuartz will do the following:

1. PgQuartz will create a work queue and add `step 1`, `step 2` and `step 3` to that queue.
   - `step 4` needs to wait for its dependencies to resolve.
2. PgQuartz will create 2 Runners due to `parallel: 2`.
   - Both Runners start processing a step (either `step 1`, `step 2` or `step 3`).
   - the third step (which could be any of the three) needs to wait until a runner is finished processing its current step
3. Should command 1.1 fail (see *commands* on what can make a command enter failed state), command 1.2 is never run.
4. Once `step 1` and `step 2` are done, `step 4` will be added to the queue, but only if the `when` part works out
   - `step 1` must finish with a return code 0
   - `step 2` requires output that contains a string "{oid}={13694}"
   - If either (or both) is not the case, then the job enters the done state without being queued.
   - See *when* for more details

CHAPTER

FIVE

WHEN

When is an option to *steps* which defines rules for jobs to run. The rules are checked against output and result codes of previous *job steps*.

5.1 Some facts about the `when` statement

`When` statements are:

- the most complex parts of job definitions.
- configured at the *step* level
- checks against results of other steps
- decisions that define if a *step* should be run
- defined as go templates

5.2 Special methods

PgQuartz has implementations of special methods to be used as part of the `when` statements:

5.2.1 Rc

PgQuartz keeps track of all Return Codes of all commands and has a special method to collect the return codes of all commands. the method is called `Rc` and is implemented on Steps, Instances, and Commands, where:

- Step.RC returns the sum of all Rc values of all instances
- Instance.Rc returns the sum of all Rc values of all Commands
- Command.Rc:
  - shell: Is set to the return code of the execution
  - postgresql: Is set to 0 on success, is set to 1 of failure

  Note that Rc is implemented as an integer, which means that it could overflow but only with more than 16843009 commands ending in exit code 255, or even more with lower exit codes, which probably is not a realistic use case.

5.2.2 StdOut and StdErr

PgQuartz keeps track of all output of all Commands, and has a special method to collect the return codes of all commands. the method is called Rc and is implemented on Steps, Instances, and Commands, where:

- Step.StdOut returns a concatination all StdOut values of all instances ; similar for StdErr
- Instance.StdOut concatination all StdOut values of all Commands ; similar for StdErr
- for shell scripts, StdOut contains the stdOut of the execution and StdErr (as expected) contains the stdErr of the execution
- for postgresql queries, StdOut contains a special formatted output of the table where
  - every row ends on a new line
  - every line is compiled version of {name}={value} where name is the columns name and value is the column value (with ‘ replaced by ‘’ for values)
- for postgresql queries, StdErr is empty

  Note Internally PgQuartz works with a special type Result, which are meant for efficient handling of concatination of stdout and stdErr. But the less output (rows or lines) the better, so make sure as a developer that Commands produce as little output as required.

5.2.3 StdOut and StdErr

5.3 Example

We make the ‘When concept’ more tangible with an example:

5.3.1 Example config

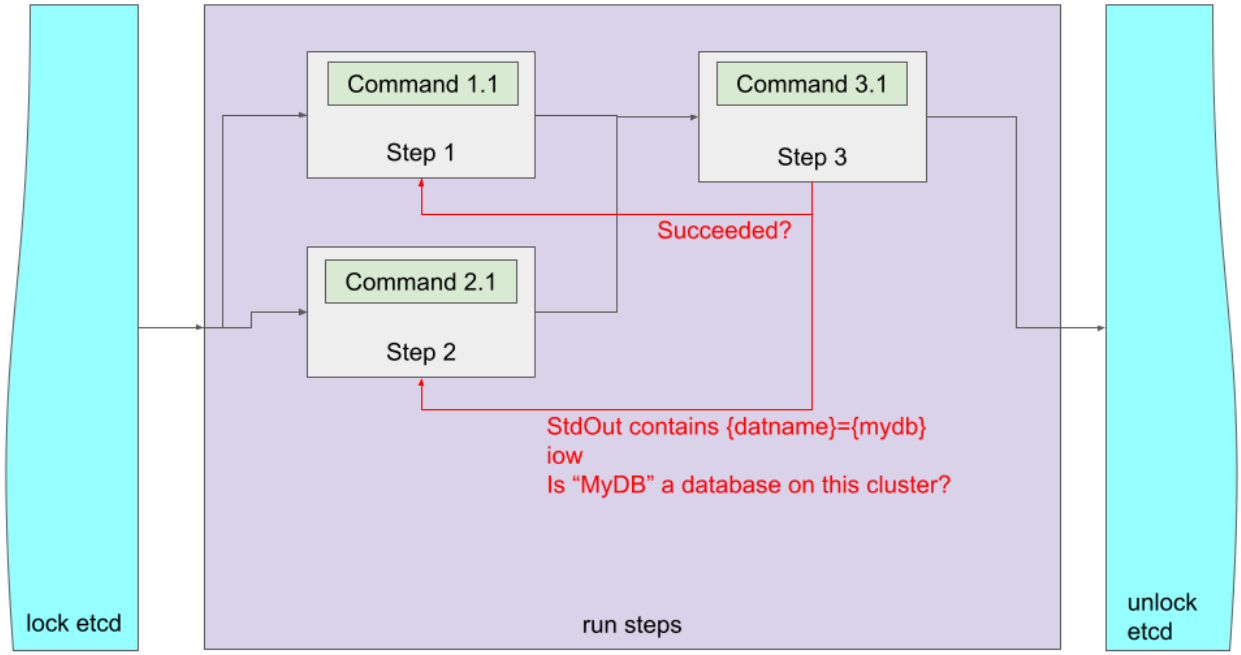
```
steps:
  step 1:
    commands:
      - name: Check if /etc/fstab exists
        type: shell
        inline: test -x /etc/fstab
  step 2:
    commands:
      - name: Run command 2.1
        type: pg
        inline: select datname from pg_database
  step 3:
    commands:
      - name: Run command 3.1
        type: pg
        file: ./sql/step_3.1.sql
    depends:
      - step 1
      - step 2
    when:
      - "eq .Steps["step 1"].Instances.Rc 0"
```

(continues on next page)

(continued from previous page)

```
      - '.Steps.["step 2"].Instances.StdOut.Contains "{datname}={mydb}"'
parallel: 2
```

5.4 Graphical representation



5.4.1 What does it do?

When running a job with a specification as shown in the *example*, PgQuartz will do the following:

1. PgQuartz will create a work queue and add `step 1`, and `step 2` to that queue.
   - `step 3` needs to wait for its dependencies to resolve.
2. PgQuartz will create 2 Runners (due to `parallel: 2`).
   - Both Runners start processing a step (both `step 1` and `step 2` are processed).
3. Should command 1.1 fail (see *commands* on what can make a command enter failed state):
   - RC of `step 1` would not be 0
   - the first `when` statement on `step 3` will prevent `step 3` from running
4. Should command 2.1 fail, or not find any database `mydb`:
   - the stdout will not contain `{datname}={mydb}`
   - the second `when` statement on `step 3` will prevent `step 3` from running

note that `step 3` also has dependencies on `step 1` and `step 2`. These are currently required because without them:

- `step 3` would be scheduled before `step 1` and `step 2` would have been running

- at schedule time, `step 1` would not have an RC set. RC defaults to 0, so `step 3` would still be scheduled regardless of failure of step 1.
- at schedule time `step3` would not have a stdout set. StdOut defaults to empty string, which does not contain `{datname}={mydb}` so step 3 would not be run, even if `mydb` would exist
- `step 3` would not be run (prevented by second when statement 'always failing' without dependencies)

CHAPTER

SIX

MATRIX AND JOB INSTANCES

Steps can be defined once, and run multiple times with different arguments. This is done by specifying all values of arguments in a matrix. Internally, PgQuartz will convert the matrix of arguments into a list of combinations.

Example:

```
matrix:
  arg1: ["1", "2"]
  arg2: ["A", "B"]
```

Would be converted into the following instances:

```
{"arg1": "1", "arg2": "A"}
{"arg1": "1", "arg2": "B"}
{"arg1": "2", "arg2": "A"}
{"arg1": "3", "arg2": "B"}
```

PgQuartz then schedules and runs every instance separately. As such, (with enough runners) the step would be run 4 times, with the arguments set accordingly.

> **Note** that without specifying a matrix, the step would be run only once, without any arguments being set.

6.1 Passing arguments

6.1.1 Shell scripts

When running commands as shell scrips, the arguments are passed as environment variables. As an example:

```
{"arg1": "1", "arg2": "A"}
```

would be run as

```
PGQ_INSTANCE_ARG1=1 PGQ_INSTANCE_ARG2=A /path/to/step/command.sh
```

Inside the script the arguments can be accessed through the names of the environment variables, like:

```
echo "arg1: ${PGQ_INSTANCE_ARG1}, arg2: ${PGQ_INSTANCE_ARG2}"
```

Which would create a stdout value containing:

```
arg1: 1, arg2: A
```

6.1.2 PostgreSQL scripts

When running a PostgreSQL command, PgQuartz does the following:

- When in batch mode, PgQuartz splits the command by ';' characters into multiple queries, and does all of this for every query.
  - When not in batch mode, PgQuartz expects the query to be one query and does all of this for the one query.
- PgQuartz scans the query for named arguments (e.a. `:argname`) and replaces them with positional arguments (e.a. `$1`) while maintaining a list of the arguments values
- PgQuartz runs the query with positional arguments while passing the arguments as a list of positional arguments

This does mean that:

- arguments can (only) be passed by name specifying `:argname` placeholders in your query as required
- PgQuartz runs them as positional arguments, so your queries ed up in PostgreSQL logs with `$n` placeholders instead
- But, at least the interface to both PostgreSQL scripts and bash scripts is the same (named arguments)

6.2 Example config

An example of running just one step, but with 6 different combinations of arguments, 6 times in parallel

```
steps:
  step 1:
    commands:
      - name: Run step 1.1
        type: shell
        inline: 'touch "/tmp/${PGQ_INSTANCE_ARG1}_${PGQ_INSTANCE_ARG2}"'
      - name: Run step 1.2
        type: pg
        inline: insert into t1 (id, txt) values(:arg1::integer, :arg2)
    matrix:
      arg1: ["1", "2"]
      arg2: ["A", "B", "C"]
parallel: 6
```

6.2.1 What does it do?

This job will create 6 runners, and each runner will run the step, but with different arguments. All instances will run in parallel (**note that even with queuing, order is not enforced between instances).

Assuming that we have

- a /tmp location which is writable, and does not have the files already existing
- a t1 table which is empty and can hold the values (id integer, txt text)

We end up with:

1. The following (empty) files:
   - /tmp/1_A
   - /tmp/1_B

- /tmp/1_C
- /tmp/2_A
- /tmp/2_B
- /tmp/2_C

2. A table with six rows:
   - id=1, txt='A'
   - id=1, txt='B'
   - id=1, txt='C'
   - id=2, txt='A'
   - id=2, txt='B'
   - id=2, txt='C'

CHAPTER

SEVEN

COMMANDS

Commands are the basic building blocks out of which *Steps* are defined. Every *Step* is defined as a series of Commands, and every *Instance* runs this series of Commands until one fails.

7.1 Limit your output

> ***Note*** that PgQuartz basically keeps track of all StdOut and StdErr, and that can require a lot of memory. PgQuartz has no special handling, flushing to disk capabilities, or other options. Therefore it is crucial for developers of PgQuartz jobs to make sure output is limited options like tail and LIMIT!!!

7.2 Configuration options

7.2.1 Name

Every Command has a name, which is used in output and logging. **Note** that emptystring (default) is accepted as Name as well.

7.2.2 Role

Roles can be set per *Connection*, but can be overruled per Command. For more information, please refer to *Roles on connections*.

7.2.3 BatchMode

As a convenience option SQL Queries (Command bodies run against PostgreSQL connections) can be run in `batchMode`, which means they are split by semicolons and then run one at the time. This splitting is done in a very crude manner, and as such below examples also act as separator characters

- such semicolons within `SQL strings` (like "My text with ;")
- SQL Names ('my name with ;')
- semicolons used inside PL/PgSQL and other (SQL) code blocks

Note that specifying `batchMode: true` is not encouraged because:

- As mentioned before, the implementation is very crude and can easily mess up commands
- there is no technical downfall to specifying every Query as a separate Command

- there is upside to specifying every Query as a separate Command, because all other configuration option like `Name` and `Role` can be set differently for every separate Commands

7.2.4 Command types

The `type` field of a command can have 2 types of values:

1. `shell` (default), which means 'execute this command in a terminal shell'
2. Any name of a *Connection*, which means run this command as a SQL Query on the specified connection.

Note that PgQuartz verifies the Job definition before running the job, and errors out if anything else is specified as a Command type.

7.2.5 Inline or file

Command bodies can be either specified inline, and the effect depends on the type of command:

- Inline type bodies against PostgreSQL connections are directly run against the connection
- File type bodies against PostgreSQL connections are read from file into memory and then run against the connection
- Inline type bodies of type shell are written to a file from memory and the file is executed in a shell
- File type bodies of type shell are directly run in a shell

7.3 Example

We make the 'Commands concept' more tangible with an example:

7.3.1 Example config

```
steps:
  step 1:
    commands:
      - name: Run command 1.1
        type: pg
        inline: |
          CREATE TABLE IF NOT EXISTS t1 (id int, txt text);
          CREATE TABLE IF NOT EXISTS t2 (id int, id2 int);
      - name: Run command 1.2
        type: shell
        inline: "echo 'Done that' > /tmp/beenhere.txt"
      - name: Run command 1.3
        type: pg
        file: ./sql/step_1.3.sql
      - name: Run command 1.4
        type: shell
        file: ./bash/step_1.4.sh
connections:
  pg:
```

(continues on next page)

(continued from previous page)

```
  type: postgresql
  conn_params:
    host: /tmp
```

7.3.2 What does it do?

When running a job with a specification as shown in the *example*, PgQuartz will do the following:

1. PgQuartz will create a work queue and add only one *Instance* of `step 1` to that queue.
2. PgQuartz will create at least one Runner.
3. the Runner starts processing the *Instance*:
    - Command 1.1 is run against the PostgreSQL connection and
        - (unless it already exists) the table t1 is created
        - (unless it already exists) the table t2 is created
    - If Command 1.1 succeeded, the runner will run Command 1.2 in a shell and add a line to the file
    - If Command 1.2 succeeded, the runner will run Command 1.3, read the actual command from the specified file and run it against the PostgreSQL connection
    - If Command 1.3 succeeded, the runner will run Command 1.3, read the actual command from the specified file and run it in a shell

CHAPTER

EIGHT

CHECKS

PgQuartz is meant as a framework for easy job definition without the toil of technical implementation of underlying infrastructure, such as cluster wide coordination and parallelization. This framework makes a clear distinction between the actual work (*Steps*), and quality assurance (*Checks*).

8.1 Why this distinction

This approach adds the following benefits to teh PgQuartz framework:

- Results of *Steps* in basis don't have effect on each other; e.a. one stop failing does not block other steps
  - End result is done after running the *Steps*
  - During the *Steps* phase PgQuartz by default tries to finalize as much as possible without caring about failures in other steps ; This behaviour can be tuned by defining *when definitions* as required.
  - ***note*** that *Commands* are serialized and failures do effect future *Commands*
- the Job developer is forced to make clear distinction between the work and QA which helps in proper definition and readability
- *Steps* can run in parallel without taking other steps into account
  - The *Steps* phase is about getting the work done as fast as possible
  - This even goes for Instances of Steps, everything can run in parallel as good as possible
- At QA time, we can look at result of steps without requiring to look at results of separate instances,
  - We can still also look at results of a single instance if required, but the framework allows for inspection of combined results of all instances of a step
- This simplifies failure analysis
  - The failed Check is a clear direction to the actual failure
  - Failed Checks can easily be found (always logged in the log tail)
- With this separation
  - PgQuartz enables functional testing (is the end state as expected) over technical testing (did the command raise an error)
  - PgQuartz test frequency is brought down significantly ; e.a. don't parse every output of every command for every instance of every step with grep, but simply do one check in the end of the job

8.2 Configuration options

8.2.1 Overlapping with Commands

The following options for *Commands* can also be set for Checks.

- Name
- BatchMode
- Inline / file
- Role
- Check type (See [Command type](./COMMANDS.md#Command types) for info on how it works)

8.2.2 Specifying arguments on checks

Similar by defining *Steps*, a Check can also be defined with a matrix of arguments. The implementation is very similar to *Step instances*, with a few distinctions:

- We don't call them separate Instances, since they are run in series, and one failure halts the rest of the current Check and all future Checks.
- For *Steps and Instances*, a matrix of arguments is applicable to all *Commands* in a *Step*, but Checks don't have that extra dimension. A Matrix of arguments is only applicable to one Check and every Check can have its own definition.

8.2.3 Rc

Rc configures the expected Return Code for a check. This defaults to 0 (success return code). If a check is run and does not return the expected return code, the command is expected to have failed.

8.2.4 Expected

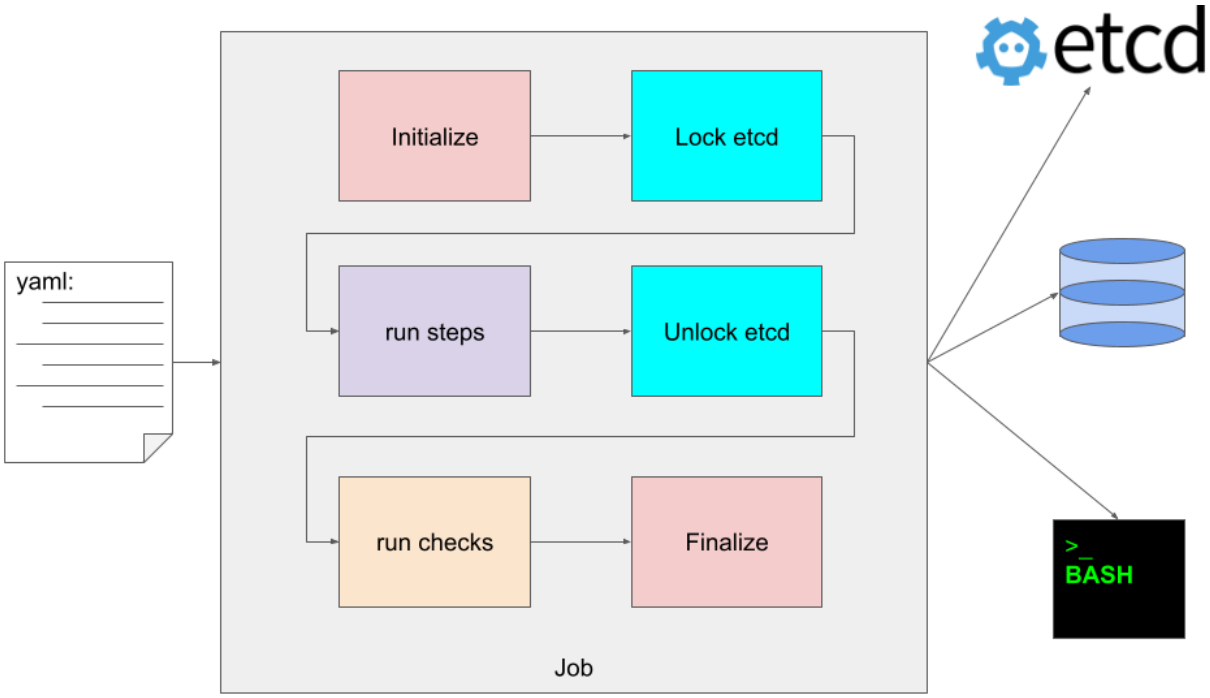
An `expected string` can be configured. When set, PgQuartz searches stdout of the check for this `expected string` and when not found, the command is expected to have failed. Consider this as an alternate to running a command and then grepping its output for `SUCCESS` to see if it succeeded.

8.2.5 Unexpected

An `unexpected string` can be configured. When set, PgQuartz searches stdout of the check for this `unexpected string` and when it is found, the command is expected to have failed. Consider this as an alternate to running a command and then grepping its output for `FAIL` to see if it failed.

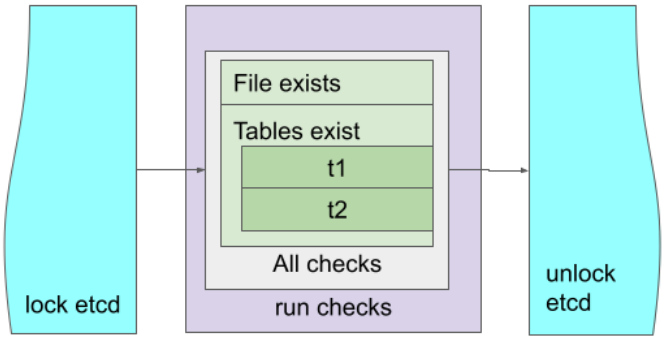
8.3 How and when are Checks run

8.3.1 When is the check block run



As can be seen in the above diagram, PgQuartz first runs *Steps* and then runs Checks. Furthermore, *Steps* are run inside an *etcd* context (blocking same jobs on other servers), where Checks are not.

8.3.2 When are checks run



As can be seen in the above diagram, PgQuartz basically runs all Checks as one block comparable to how it runs all *Commands* for one *Step*:

- all Checks are run as one big block
- all Checks run in series (one runner, no parallelization)
- if a Check fails, it is reported, and PgQuartz exits with an error exit code

8.4 Example

We make the 'Checks concept' a bit more tangible with an example:

8.4.1 Example config

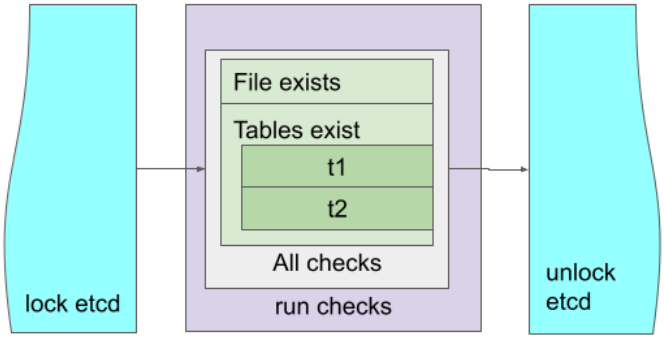
```
steps:
  step 1:
    commands:
      - name: Run command 1.1
        type: pg
        inline: |
          CREATE TABLE IF NOT EXISTS t1 (id int, txt text);
          CREATE TABLE IF NOT EXISTS t2 (id int, id2 int);
      - name: Run command 1.2
        type: shell
        inline: "echo 'Done that' > /tmp/beenhere.txt"
checks:
```

(continues on next page)

(continued from previous page)

```
- name: File exists
  type: shell
  inline: test -f /tmp/beenhere.txt
  rc: 1
- name: Tables exist
  type: pg
  inline: "select * from pg_tables where schemaname='public' and tablename = :tblname"
  matrix:
    tblname:
      - t1
      - t2
```

8.5 Graphical representation



8.5.1 What does it do?

When running a job with a specification as shown in the *example*, PgQuartz will do the following:

1. PgQuartz will lock etcd, run the steps (in parallel), and unlock etcd
   - ***note*** that PgQuartz does not exit on Step errors
2. After that, PgQuartz will run the checks in the following order:
   - File exists
     - if it does not result as expected, log error end exit with error exit code (1)
   - Tables exist (t1)

    - if it does not result as expected, log error end exit with error exit code (1)
  - Tables exist (t2)
    - if it does not result as expected, log error end exit with error exit code (1)
3. If all checks resulted as expected:
  - Report 'Job finished successfully'
  - Exit with success exit code

CHAPTER

NINE

PGQUARTZ CONNECTION ENDPOINT CONFIGURATION

The `Connections` block can be used to define one or more PostgreSQL connection endpoints for processing the SQL queries.

9.1 Configuration options

9.1.1 Connection type

Currently, a type can be set, and `postgresql` is usually set as the type, but setting this field has no effect. In future releases we might add other connection endpoint types, such as MySQL, MongoDB, etc. if we receive requests from the community. Please specify an Issue to request other endpoint types.

9.1.2 Role

PgQuartz has the option to only run Queries on Connection endpoints if they have a specific role. One of the following roles can be specified:

- standby: Only run if the endpoint is a standby (`SELECT pg_is_in_recovery()` returns `true`)
- primary: Only run if the endpoint is a standby (`SELECT pg_is_in_recovery()` returns `false`)
- all: Don't worry, all roles are fine

PgQuartz has 2 types of behavior on Connection Roles:

- When the job configuration option `runOnRoleError=false` (default)
  - PgQuartz errors out if the endpoint does not meet expectations (e.a. configured and actual role differ for a connection)
  - *role configuration on Commands* can be set to skip a certain Command depending on the role of the endpoint
- When the job configuration option `runOnRoleError=true`
  - PgQuartz continues if the endpoint does not meet expectations (e.a. configured and actual role differ)
  - *role configuration on Commands* can still be set to skip a certain Command depending on the role of the endpoint
  - Role configuration on the Connection acts as a default for *roles on Commands* See *runOnRoleError* for more info.

  Note that PgQuartz uses jackc/pgx/v4, which also supports setting `target_session_attrs` to target a specific role for the connection.

9.1.3 Connection parameters

PgQuartz uses jackc/pgx/v4, which also supports setting many libpq Parameter Key Words including:

- host: Set host name(s) to connect to
- port: Set port to connect to
- user: Set User to connect as
- password: Specify a password; ***NOTE:*** there are far more secure options than specifying a clear text password in a config file, probably maintained in a git repo!!!
- `target_session_attrs` to target a specific role for the connection. But more parameters can be applied.

9.2 Example

We make the 'Connections concept' more tangible with an example:

9.2.1 Example config

```
connections:
  pg:
    type: postgresql
    role: all
    conn_params:
      host: /tmp
      port: 5432
      user: postgres
      password: supassword
```

CHAPTER

TEN

ETCD INTEGRATION

PgQuartz can integrate with etcd in which case it will lock a key while it is running. This allows for scheduling jobs across clustered nodes one at a time, where

- any node will lock the key and run the steps
- all other nodes will wait until the node has finished its operations and released the key
- when the key is released the next node will lock the key, run the steps and release the key
- and on and on it goes until either
  - all nodes have successfully finished running the steps, or
  - the *Jobs timeout* has expired after which all nodes that have not run yet will exit with an error code

10.1 Configuration options

10.1.1 endpoints

a list of endpoints can be set to point to all instances of etcd. Every node should be formatted as `hostname:port` (e.a. `server1:2379`). When not set, this defaults to only one node `localhost:2379`.

10.1.2 LockKey

The key that will be locked for this job can be configured. When not set, it defaults to the job name. ***note*** that the job name is derived from the yaml that defines the job (e.a. `/etc/pgquartz/jobs/job1.yaml` would result in a job name `job1`)

10.1.3 Lock timeout

While running, PgQuartz automatically refreshes the lock, and when finished, PgQuartz automatically releases the lock. But there are circumstances where etcd would retain the lock until a preset lock timeout, after which it would release the lock.

Examples of these circumstances are:

- PgQuartz dies a horrible death (e.a. `killall -KILL pgquartz` would be horrible)
- a network partition would occur between PgQuartz and etcd

And when the lock is released:

- the current PgQuartz job fails

- PgQuartz running on another node would be released to run the job

10.2 Example

To explain the 'etcd integration' consider the following config:

10.2.1 Example config

```
steps:
  step 1:
    commands:
      - name: Run command 1.1
        type: shell
        inline: "sleep 10"
      - name: Run command 1.2
        type: shell
        inline: "date +%s > /tmp/beenhere.txt"
etcdConfig:
  endpoints:
    - localhost:2379
  lockKey: awesomeJob1
  lockTimeout: 1m
timeout: 75s
```

10.2.2 What does it do under normal circumstances?

As an example the example config would be run as a job on a cluster of 3 nodes in an etcd cluster.

- each node is running etcd on port 2379, and together they form a cluster
- each node has this exact job definition, and PgQuartz is scheduled to run this job at the same time

The following would happen:

1. any node would be first to run PgQuartz and lock the key `awesomeJob1`
   - that node would wait 10 seconds and then add the current epoch (seconds elapsed since January 1, 1970) into a file /tmp/beenhere.txt
   - other nodes wait until the job on the locking node is finished and has released the node
2. When the job on the first node is finished
   - on this initial node, PgQuartz releases the lock, and wraps up (run checks and exit)
   - one of the other 2 nodes would notice the release and lock the key `awesomeJob1` after which it would
     - wait 10 seconds
     - add the current epoch (seconds elapsed since January 1, 1970) into a file /tmp/beenhere.txt
   - the last node waits until the job on the second node is released
3. When the job on the second node is finished
   - on this second node, PgQuartz releases the lock, and wraps up (run checks and exit)

- the last node would notice the release and lock the key `awesomeJob1` after which it would
  - wait 10 seconds
  - add the current epoch (seconds elapsed since January 1, 1970) into a file /tmp/beenhere.txt

note that:

- The job has been scheduled to run at the same time
- The job has actually run after each other one node at a time
- There is no predetermined order. Probably, ntp drift would decide run order
- the last line in the /tmp/beenhere.txt files would all differ by 10 seconds (or 11 in some rare cases)

10.2.3 What happens on issues

If a network partition would occur between the node currently running the job, and the other 2 nodes, the following could happen:

1. If the network partition resolves within 60 seconds, all works out fine
2. If the network partition takes longer:
   - etcd releases the lock
   - the job that was running exits with an error
   - the next node in line would run the job
3. Maybe the second job, and the 3rd job for sure will not succeed due to the job timeout (81 seconds):
   - first node runs (0 - 10) seconds
   - partition hangs for 60 seconds
   - second node runs for 10 seconds (starts 60-70 seconds after job start)
   - if it is started after waiting for 65 seconds or more, it would not finish, but be timed out after 75 seconds
   - If not, it will finish, only to leave 0-5 seconds for node 3 before timing out as well

If any issue occurs (hang, process kill, etc.) either PgQuartz would cleanly release the lock, or the above described behavior would be perceived.