pgSimload v.1.3.0 Documentation

Jean-Paul Argudo

April, 24th 2024

crunchy data

Contents

1 Overview

Welcome to pgSimload !

pgSimload is a tool written in Go, and accepts 3 different modes of execution:

- **SQL-Loop** mode to execute a script infintely on a given schema of a given database with a given user
- **Patroni-Watcher** mode to execute a monitoring on a given Patroni cluster. This is usefull only if… you run Patroni
- **Kube-Watcher** mode to have a minimal monitoring of a given PostgreSQL cluster in Kubernetes

Given the mode you choose, some parameters are mandatory or not. And the contexts of executions are different. Please refer to the complete documentation in docs/pgSimload.doc.md.

Alternatively, you can download the documentation in PDF format.

2 Running, Building and Installing binary

2.1 Running with Go

This is very straightforward if you have Go installed on your system. You can run the tool with Go from the main directory of the project like:

```
$ go run . <parameters...>
$ go run . -h
```

2.2 Using binaries provided

If you don't have Go installed on your system, you can also just use one of the binaries provided in bin/.

If you want to build your own binary you can build it too, as described in the next paragraph.

Feedback is welcome in any cases!

2.3 Building binaries

You can use the provided script build.sh.

```
$ sh build.sh
```

2.4 DEB and RPM packages

We've started tests to build those packages but at the moment, the work hasn't finish yet. But do you really need this, since pgSimload is a standalone binary ?

3 Usages

This tool can be used in different infrastructures:

- on the localhost, if a PostgreSQL is running on it
- on any distant stand-alone PostgreSQL or PostgreSQL cluster, in bare-metal of VMs
- on any PostgreSQL stand-alone PostgreSQL or PostgreSQL cluster running in a Kubernetes environment.

This tool as different usages, and you probably think of some that I haven't listed here:

- just initiate a plain `select 1`, a `select count(*)from...`, whatever you find usefull. But pgSimload won't get you results back from those executions
- insert dummy data (mostly randomly if you know about, mostly, `generate_series()` and `random()` PostgreSQL functions) any DB with the schema of your choice and the SQL script of your choice
  - if your database doesn't have a schema yet, you can create in a `create.json` file. Look for examples on how to do that in the `examples/SQL-Loop/` directory. It should straightforward. That file is **not** mandatory, as pgSimload need at least a `-config <file>` and a `-script <file>` to run, in SQL-Loop mode.
  - the SQL script of your choice. For that purpose you create a plain SQL file, where you put everything you want in it. It will be run in an implicit transaction, and can contain multiple statements. If you want details on how pgSimload runs those statements at once, please read chapter Multiple Statements in a Simple Query in the PostgreSQL's documentation.
  - you can set special parameters to the session like `SET synchronous_commit TO 'on'` or `SET work_mem TO '12MB'` if you want the SQL script's sessions to be tweaked depending your needs. This is usefull to compare the performances or behaviour in replication or others things. For that you'll have to use the `-session_parameters <session_parameters.json>` parameter for pgSimpload. Otherwise, without this, every DEFAULT values will of course apply.

  - if you're too lazy to gather those session parameters, you can create a template file you can letter modify and adapt to your needs. For that pgSimload will create a template file in the name you want, based on a given connection. Look for `-create_gucs_template` in this documentation.
  - this "dummy data insertion" is most often used to simulate some write work on a PostgreSQL server (standalone or the primary of a PostgreSQL cluster with a(some) replica(s).
  - the SQL-Loop mode execution can be limitated to:
    * a number of loop exections you define thanks to the `-loops <int64>` parameter and/or
    * a given execution time of your choice you can define thanks to the `-time duration` parameter, where that duration is expressed with or without simple or double-quotes like in "10s", 1m30s or '1h40m'
    * if both parameters are used at the same time, the SQL-Loop will end whenever one or the other condition is satisfied
  - the rate of the iterations can be slowed down since version 1.2.0 thanks to the `-sleep duration` parameter, where a duration is expressed the same way `-time duration` is (see upper). If this parameter is set to anything else that 0, pgSimload will sleep for that amount of time. This is usefull if you want to slow down the SQL-Loop process. It also avoid the user to manually add like a `select pg_sleep(1);` at the end of the `SQL script` used with `-script`. So it's faster to test different values of "sleeping" by recalling the command line and changing the value there instead of editing that SQL script…

- test failovers, or what happens when a DB is down: pgSimLoad handles those errors. Give it a try: simply shuting down your PostgreSQL server while it runs… You'll see it throwing errors, then restarting to load once the PostgreSQL server ("primary" if you use replication) is back.
- monitor a PostgreSQL cluster that uses Patroni, with the special `--patroni <config.json>` parameter, that has to come with a `--config <config.json>` where the later will use **mandatorily** the `postgres` user, because, on that mode, we use a special trick to get the primary's name, and this trick can only be done by a superuser in PostgreSQL (so it can be something else than `postgres`, if you set another superuser).
- so when testing a PostgreSQL cluster using Patroni, with multiple hosts (a primary and a given number of replicas, synchronous or not), usually, pgSimload is run in 2 separate terminals, one to load data, and the other, to monitor things in Patroni.
  - note the Patroni-Watcher mode can have added information thanks to the `Replication_info` set to `nogucs` or `<list of gucs separated by a comma` (e.g "synchronous_standby_names,

synchronous_commit, work_mem") in the `patroni.json` config file passed as an argument to `-patroni <patroni.json>` parameter. If set to `nogucs`, no extra GUCs are shown, only the info from `pg_stat_replication` will be

- monitor a PostgreSQL cluster that runs in Kubernetes, wheter this solution uses Patroni or not for HA: this mode only uses some `kubectl` commands to gather only the relevant information to monitor things, like who's primary, who's replica, the status of each, etc. This mode has been tested against the Postgres Operator (aka PGO), from CrunchyData, and the operator from CloudNativePG. You'll find in the `example/Kube-Watcher/` directory proper configuration JSON to use in both cases
- demo Crunchy Postgres, a fully Open Source based PostgreSQL distribution using extensively Ansible
- demo Crunchy Postgres for Kubernetes, a fully Open Source based PostgreSQL distribution to run production workloads in Kubernetes

4 Examples given

You can find several examples of usage in the main directory of the project under `examples/`.

4.1 examples/SQL-Loop/simple/

This example is the simpliest example of the SQL-Loop mode!

Prerequisites

- PostgreSQL Server, any version shoud work
- user has LOGIN capabilities

No creation of tables and others are needed, so there's no need to call for `-create <create.json>` or such. pgSimload accepts the omission of parameter `-create`.

`script.sql` contains a simple "select 1;". There's also a comment starting with `--`, like in ... plain SQL. So that means this file is just the SQL script you want it to be. Simple as that.

Once pgSimload has been compiled **and** the `config.json` adapted to suit your needs, this could be used as simple as:

```
1 $ pgSimload -config config.json -script script.sql
```

You can throtle it down asking pgSimload to wait for 1 second and a half like this:

```
$ pgSimload -config config.json -script script.sql -sleep 1s500ms
```

If you want to limit the number of loops, you can do that as simply as

```
$ pgSimload -config config.json -script script.sql -loops 10
```

Alternatively, you can limit the execution time, setting a duration:

```
$ pgSimload -config config.json -script script.sql -time 5s
```

You can do both at the same time. Whichever happens first will break the SQL-Loop:

```
$ pgSimload -config config.json -script script.sql -time 1s -loops 20
$ pgSimload -config config.json -script script.sql -time 10s -loops 20
```

4.2 examples/SQL-Loop/PG_15_Merge_command/

This example is to test the MERGE command first introduced in PostgreSQL version 15.

Prerequisites

- PostgreSQL Server version 15+ stand-alone or not
- a user called owning a schema name “test” (please adapt config.json file to match your needs here)
- user has LOGIN capabilities

Creates a schema with 3 tables in `create.json`.

`script.sql` will:

- create sample data in `test.station_data_new`
- merge that data in `test.station_data_actual`
- merge `test.station_data_actual` into `test.station_data_history`

As per jpa’s blog article on MERGE

Once pgSimload has been compiled and the binary placed in some dir your `$PATH` points to, this could be used as simple as:

```
$ pgSimload -config config.json -create create.json -script script.sql
```

The `watcher.sh` is a plain psql into watch to get some live stats on the database. You may have to adapt it to match your usage. I’ve added 2 flavours.

The first show some data, nice to have in a separate terminal (use tilix while you demo!):

```
$ sh watcher.sh query
```

The second shows a nice histogram of the data, the query is slightly more complex and heaven tho:

```
$ sh watcher.sh histogram
```

4.3 examples/SQL-Loop/testdb/

This is another example that shows one can:

- create multiple different `create.json` files to match different scenarios, adding different things like in `create.json`, `create.delete.json`, `create.delete.vacuum.json`, etc. to pass to the paramter `-create`
- create multiple different `script.sql`, `insert.sql`, etc.. to pass to the parameter `-create`

Obviously, that `delete from test.data;` is just for the example, if you really want to delete all data from a table, in the real world, you need to use truncate data!

If you have a PostgreSQL *cluster* where you want to test as an example:

- write activity to the primary and
- read activity to the secondary

Then you'll need 2 different files for credentials one to your primary, on let's say port 5432, another one to your secondary (or pool of secondaries, if you're using `pgBouncer` on a different port, or just `HAProxy` or anything else to balance to different PostgreSQL replicas, on let's say, port 5433).

You'll need also 2 different SQL script files to run read/write operations on the primary, and obviously, read/only operations to the secondary (or group of secondaries).

Finaly, you will have to run twice pgSimload, in 2 different terminals, to handle boths scenarios at the same time.

We give here a special example of the file `session_parameters.json` (you can name that like you want), as for you to use the special `-session_parameters <session_parameter.json>` if you want to modify the parameters of the session in which the script.sql queries will exectute. You can use this to set special values to a lot of configuration parameters that PostgreSQL allows to change within a session. As an example: `work_mem`, `synchronous_commit`, etc.

5 Overview of flags and parameters

Those tabulars show basic information about flags and parameters. For full documentation, please read next chapter “Reference : parameters and flags”.

5.1 All modes : flags

All flags are optional and intended to run alone.

Name	Description
`contact`	Shows author name and email
`help`	Shows some help
`license`	Shows license
`version`	Shows current version of pgSimload

5.2 SQL-Loop mode : parameters

Name	Mandatory	Optional	Value expected	Description
`config`	**X**		JSON file	Sets the PG connexion string (any user)
`create`		**X**	JSON file	Sets the SQL DDL to run once prior main loop
`script`	**X**		SQL text file	Sets the script to run inside the loop
`session_parameters`		**X**	JSON file	Sets special session configuration parameters

Name	Mandatory	Optional	Value expected	Description
`loops`		**X**	integer	Sets the number of loops to execute before exiting
`time`		**X**	duration	Sets the total execution time before exiting
`sleep`		**X**	duration	Sets a sleep duration between 2 iterations of the SQL-Loop

5.3 Kube-Watcher mode : parameters

Name	Mandatory	Optional	Value expected	Description
`kube`	**X**		JSON file	Sets parameters for this special mode

5.4 Patroni-Watcher mode : parameters

Name	Mandatory	Optional	Value expected	Description
`config`	**X** (if `Replication_info` is not empty)	**X**	JSON file	Sets the PG connexion string (superuser)
`patroni`	**X**		JSON file	Sets parameters for this special mode

5.5 Session parameters template file creation

Name	Mandatory	Optional	Value expected	Description
`config`	X		JSON file	Sets the PG connexion string
`create_gucs_template`	X		output name file	Sets the template file to create

6 Reference: parameters and flags

All flags and parameters can be listed executing `pgSimload -h`.

There are 2 different modes when executing `pgSimload`:

- **SQL-Loop mode** to execute a script infintely on a given schema of a given database
- **Patroni-Watcher mode** to execute a monitoring on a given Patroni cluster. So you need one of such for this mode to be useful to you

Given the mode you choose, some parameters are mandatory or not. And the contexts of executions are different.

Before listing each, there are common parameters that can be used. Let's see those first.

6.1 Common flags and parameters

6.1.1 config (JSON file) [MANDATORY]

In the **SQL-Loop mode** the "Username" set in the `config.json` can be any PostgreSQL user.

In the **Patroni-Watcher mode** the "Username" set in the `config.json` **has to be a superuser** in PostgreSQL, typically "postgres". Because we use special tricks to get the `hostname` of the PostgreSQL primary server.

"ApplicationName" is used to put a special "pgSimload" there, so the user can `ps aux | grep [p]gSimload` on any of the PostgreSQL server to isolate the process pgSimload uses... Or for any other SQL / bash command.

As per version 0.6 (June 2023), a valid `config.json` looks like this:

```
{
   "Hostname":         "localhost",
   "Port"    :         "5432",
   "Database":         "mydbname",
   "Username":         "myusername",
   "Password":         "123456",
   "Sslmode" :         "disable",
   "ApplicationName" : "pgSimload"
}
```

“Sslmode” has to be one among those described in Table 34.1. SSL Mode Descriptions.

Most common values would be there either `disable` for non-SSL connexion or `require` for SSL ones.

6.1.2 contact (flag) [OPTIONAL]

Executing with only `-contact` will show you where you can contact the programmer of the tool.

This flag is not supposed to be run with other parameters or flags.

6.1.3 help (flag) [OPTIONAL]

Originally, “heredocs” were used in the main program to show this help, but it became too big to do such, it’s better to have that doc in the current format you’re reading, makes the source code lighter and that’s cleaner IMHO.

So as per now, the execution of that `-help` is only to show where the current documentation is located. Actually, if you are reading this, that means wether you executed that flag…or that you find it by yourself. Kudos :-)

This flag is not supposed to be run with other parameters or flags.

6.1.4 license (flag) [OPTIONAL]

Executing with only `-license` will show you the license of this tool, currently licensed under The PostgreSQL License.

A full copy of the licence should be present aside the tool, in the main directory, in a file named `LICENCE.md`.

This flag is not supposed to be run with other parameters or flags.

6.1.5 version (flag) [OPTIONAL]

Executing with only `--version` will show you the current version of pgSimload. This is intended for general information of the users and also for any further packager of the tool in various systems.

Not supposed to be run with other parameters. No need to add a value to that flag.

6.2 SQL-Loop mode parameters

The `--config` flag is not listed down there, but is still **mandatory** to run in this mode, please read carefully informations upper in this documentation. On this mode, no particular "Username" has to be set in the `config.json` file.

6.2.1 create (JSON text file) [OPTIONAL]

If you need to create tables, or do anything prior to the execution of the main loop, you have to put your SQL commands in this JSON text file.

This script will be run only once prior the main loop on the `--script` described above.

If you're want to execute pgSimload in SQL-Loop mode on an existing database, on which you've adapted the SQL present in the `--script`, then you don't need this feature. That's why it is optional.

To have a better idea of what's expected here, please refer to `examples/SQL-Loop/PG_15_Merge_command/create.json` or `examples/SQL-Loop/testdb/create.json` files.

6.2.2 script (SQL text file) [MANDATORY]

This file is in plain text and contains SQL statements to run, in the main loop of pgSimpload in the "SQL-Loop mode".

It can be as simple as a "SELECT 1;". Or much more complex with SQL SQL statements of your choice separated by semi-colon and newlines. As an example of a more complicated example see `examples/SQL-Loop/PG_15_Merge_command/script.sql`.

It will run the querie(s) "all at once" in an implicit transaction. For more details on how it works, please read chapter Multiple Statements in a Simple Query in the PostgreSQL's documentation.

6.2.3 session_parameters (JSON text file) [OPTIONAL]

This parameter lets you tweak the PostgreSQL configuration that can be specified in a session. This can be everything your PostgreSQL version allows, and we let you define proper values for proper parameters.

Every parameter you specify here will be passed at the beginning of the session when the SQL-Loop is executed. So everything will be executed accordingly to those parameters in that session.

As an example, you can tweak `work_mem` in a session, or `synchronous_commit`, depending your PostgreSQL configuration and version.

The format of the JSON file has to be the following:

```
{
  "sessionparameters": [
    {
      "parameter" : "synchronous_commit"
     ,"value"     : "remote_apply"
    },
    { "parameter" : "work_mem"
     ,"value"     : "12MB"
    }
  ]
}
```

You can add as many parameters you want in that file, from one to many.

At the moment, we don't check if the parameter and values are OK. As an example, if you set a value for an unknown parameter, you will have this output when running pgSimload:

```
The following Session Parameters are set:
   SET synchronous_commit TO 'remote_apply';
   SET work_mem TO '12MB';
   SET connections TO 'on';

2023/06/27 14:24:38 ERROR: unrecognized configuration parameter "
   connections" (SQLSTATE 42704)
```

Or if you set a right name, but in the wrong context you could have this too:

```
The following Session Parameters are set:
   SET synchronous_commit TO 'remote_apply';
   SET work_mem TO '12MB';
   SET log_connections TO 'on';

2023/06/27 14:41:20 ERROR: parameter "log_connections" cannot be set
   after connection start (SQLSTATE 55P02)
```

And finaly, if you think you set proper values but it seems that nothing is read from the brand new `session_parameters.json` you just created, like in:

```
The following Session Parameters are set:


Now entering the main loop, executing script "./examples/SQL-Loop/
    testdb/script.sql"
Script executions succeeded :        60
```

... that's because you have probably a error in the JSON file, or maybe you changed the keyword `"sessionparameters":` : don't do that, it's expected in pgSimload to have such keyword there. It is also expected that your JSON file here is valid, like given in the example file given in `examples/SQL-Loop/testdb/session_parameters.json`

6.2.4 loops (integer64) [OPTIONAL]

This parameter was added in version 1.1.0.

It allows one to limit the number of times the SQL-Loop will be run.

As an example, passing the `-loops 10` parameter to your `pgSimload` command line will give you the following (extract of) output:

```
[...]
Now entering the main loop, executing script "light.sql"

Number of loops will be limited:
    10 executions
Script executions succeeded :         10
================================================================================

Summary
================================================================================

Script executions succeeded :         10 (9.561 scripts/second)
Total exec time             :     1.045s
================================================================================
```

This parameter can be used in conjuction with the `-time` parameter below. Whichever is satisfied first will end the SQL-Loop.

6.2.5 time (duration) [OPTIONAL]

This parameter was added in version 1.1.0.

It allows one to limit the execution time of the SQL-Loop.

The value has to be one of “duration”, that is expressed with or without simple or double quotes, so all the following values are valid:

- 10s for ten seconds
- “1m30s” for one minute and thirty seconds
- ’1h15m4s” for one hour fifteen minutes and four seconds

Note that GoLang’s Time package limits duration units to the following list, you can use to pass the duration you want:

- “ns” for nanoseconds
- “us” (or “µs”) for microseconds
- “ms” for milliseconds
- “s” for seconds
- “m” for minutes
- “h” for hours

I hardly can believe you’d ever want pgSimload to run for days, months, year.. Don’t you?

As an example passing the `-time 10s` parameter to your `pgSimload` command line will give you the following (extract of) output:

```
Now entering the main loop, executing script "light.sql"

Number of loops will be limited:
    "10s" maximum duration
Script executions succeeded :         90
================================================================================

Summary
================================================================================

Script executions succeeded :         90 (8.907 scripts/second)
Total exec time             :    10.104s
================================================================================
```

This parameter can be used in conjuction with the `-loops` parameter seen in the previous paragraph. Whichever is satisfied first will end the SQL-Loop.

6.2.6 sleep (duration) [OPTIONAL]

This parameter was added in version 1.2.0.

It allows the user to actually throttle down the execution in SQL-Loop mode.

A pause of the `sleep` duration will be added between each iteration of the execution of the SQL script.

That duration is expressed the very same way the `time` parameter is (see previous paragraph).

Note that when `sleep` and `time` are used together, it can cause side effects on the total desired execution time, as an example:

- the script is a plain "select 1;" that goes ultra fast (time to exec is close to 0 seconds)
- `time` is set to `10s`
- `sleep` is set to `4s`
- the total execution time *won't be* 10s, but rather 12s, because on the 3rd execution at 8s, the pause will last 4s, leading to 12s overall…

Just bare that in mind when creating your tests use cases, etc.

6.3 Kube-Watcher mode flag and parameters

To have pgSimload act as a Kube-Watcher tool inside a termina, all you have to do is create a `kube.json` file in the following format.

6.3.1 kube (JSON text file) [MANDATORY]

When this parameter is set (`-kube <kube.json>`), you're asking pgSimload to run in Kube-Watcher mode. This mode means that you want pgSimload to monitor your PostgreSQL cluster that runs in Kubernetes, whether this one uses *or not* Patroni or whatever for HA, or even if the cluster is *not* in HA.

Simply, this mode runs some `kubectl` commands in the background, so it is **mandatory** you have a working `kubectl` command installed in your system.

This command will show relevant info on your pods:

- which is primary
- which is(are) replica(s)
- the status of each pod
- a colored button in that description, so
  - green is a primary
  - blue is a replica

  - red means the pod is down for some reason

So that information won't show the TimeLine (aka TL), or Lag between nodes, unlike Patroni-Watcher mode would do. But still, it's probably sufficient in many cases, and is a light-weight monitoring of a PostgreSQL cluster in Kubernetes!

I've added 2 samples JSON configurations you may even not have to change:

- `examples/Kube-Watcher/kube.json.PGO` is the one to be used for any Crunchy Postgres for Kubernetes and/or Postgres Operator from Crunchy Data
- `examples/Kube-Watcher/kube.json.CloudNativePG` is the one to be used for any CloudNativePG PostgreSQL cluster

Let's present that JSON with the PGO version: only the values will differ:

```
$ cat examples/Kube-Watcher/kube.json.PGO
{
    "Namespace"        : "postgres-operator",
    "Watch_timer"      : 2,
    "Limiter_instance" : "postgres-operator.crunchydata.com/instance",
    "Pod_name"         : "NAME:.metadata.name",
    "Pod_role"         : "ROLE:.metadata.labels.postgres-operator\\.
       crunchydata\\.com/role",
    "Cluster_name"     : "CLUSTER:.metadata.labels.postgres-operator\\.
       crunchydata\\.com/cluster",
    "Node_name"        : "NODE:.spec.nodeName",
    "Pod_zone"         : "ZONE:.metadata.labels.topology\\.kubernetes
       \\.io/zone",
    "Pod_status"       : "STATUS:status.conditions[?(@.status=='True')
       ].type",
    "Master_caption"   : "leader",
    "Replica_caption"  : "... replica",
    "Down_caption"     : "<down>"
}
```

Beware those `\\.` (double-escaping) are not errors, that's the way to specify things in `kubectl` : those "dots" have to be escaped. And in pgSimload, since it's a JSON, it has to be escaped *twice*…

Namespace (string)

To specify on which kube namespace we will send the underlying `kubectl` commands.

Watch_timer (integer)

How often to refresh the output you see in this Kube-Watcher mode.

Limiter_instance (string)

From that field in the JSON, up to **Master_caption**, you won't probably have to change those, if you're using the *right* example JSON given in the directory `examples/Kube-Watcher/`.

This parameter is a way to limit the output of `kubectl get pods` commands ("-l").

Pod_name (string)

This is how we can get the pod's name.

Pod_role (string)

This is the way we get the pod's role, that can be:

- a "replica" for PGO and CloudNativePG
- a "master" (for PGO) or a "primary" (for CloudNativePG)

Cluster_name (string)

This is the way we get the PostgreSQL's cluster name.

Node_name (string)

This is the way we get node node's names. That Node name is used in the 2nd `kubectl` command that is a `kubectl get nodes` one, to get back the Zone and the Status of the given node.

Pod_zone (string)

This specifies how to gather the pod's zone, if it exists.

Pod_status (string)

This specifies how to gather the pod's status.

Master_caption (string)

This parameter allows the user to set the caption that will be displayed in the Kube-Watcher mode for the primary Postgres instance in the cluster.

It can be whatever you like. Most probably, "primary" and "leader" are good candidates here!

Replica_caption (string)

This parameter allows the user to set the caption that will be displayed in the Kube-Watcher mode for all the instances that are actually replicas of the primary.

It can be whatever you like. Most probably, "replica" is great enough… "..replica" is good too, if you want to show some hierarchy in between replica(s) and the primary.

Down_caption (string)

Finaly, this paramter allows you to specify the caption for any "down" pod.

6.4 Patroni-Watcher mode flag and parameters

To have pgSimload act as a Patroni-Watcher tool in a side terminal, all you have to do is to create a `patroni.json` file in the following format. Note that the name doesn't matter much, you can name the way you want.

6.4.1 patroni (JSON text file) [MANDATORY]

When this paramter is set (`-patroni <patroni.json>`), you're asking pgSimload to run in Patroni-Watcher mode. This parameter is used to give to the tool the relative or complete path to a JSON file formated like the following example you can find a copy in `examples/Patroni-Watcher/` subdirectories:

```
$ cat patroni.json
{
    "Cluster"          : "mycluster",
    "Remote_host"      : "u20-pg1",
    "Remote_user"      : "postgres",
    "Remote_port"      : 22,
    "Use_sudo"         : "no",
    "Ssh_private_key"  : "/home/jpargudo/.ssh/id_patroni",
    "Replication_info" : "server_version,synchronous_standby_names,
       synchronous_commit,work_mem",
    "Watch_timer"      : 5,
    "Format"           : "list",
    "K8s_selector"     : "",
    "K8s_namespace"    : ""
}
```

Cluster (string)

You must specify here the Patroni's clustername. You can generaly find it where your have Patroni installed in `/etc/patroni/<cluster_name>.yml` or inside the `postgresql.yml`.

Remote_host (string)

You have to set here the `ip` (or `hostname`) where pgSimload will `ssh` to issue the remote command `patronictl` as user `patroni_user` (see up there).

Remote_user (string)

This is an user on one of the PG boxes where Patroni is installed. That one you use to launch Patroni's `patronictl`. Depending the security configuration of your PostgreSQL box, Patroni could run with the system account PostgreSQL is running with, or another user. This one may have need to use `sudo` or not. Again,that all depends on your setup.

Remote_port (integer)

You have to set here the `port` on wich pgSimload will `ssh` to. Let the default 22 if you didn't changed the `sshd` port of your remote server.

Use_sudo (string)

If the previous user set in **Remote_user** needs to use `sudo` before issuing the `patronictl` command, then set this value to "yes".

Ssh_private_key (string)

Since `ssh`-ing to the `Remote_host` IP or (`hostname` if it's enabled in your DNS) need an SSH pair of keys to connect, we're asking where is that private key. It can be as simple as `/home/youruser/.ssh/id_patroni`. Beware not to set here the public key, because we need the private one.

Also, we assume you did the necessary thing on SSH so that user can SSH from the box where pgSimload is running to the target host, specifically, that the public key of your user is present in the `~/.ssh/authorized_keys` of the taget system and with the matching **Remote_user**.

Replication_info (string)

Thanks to this feature, pgSimload can show extra information about replication. This is usefull if Patroni doesn't do "everything in HA", like the SYNChronous replication, that can be handled by PostgreSQL itself, thanks to the `synchronous_commit` and `synchronous_standby_names` parameters. It can also adapt in other scenarios, or just to show the `server_version`, whatever you want!

If you don't need this extra information, to disable it, just set it to an empty string in the JSON like:

```
[...]
    "Replication_info" : "",
[...]
```

If disabled, the "Replication information" no extra information will be shown after the output of the `patronictl ... list` command.

If you want to activate it, like `Replication_info` is anything different to an empty string, **be sure you also provide also `-config <config.json>` parameter, pointing to a file where superuser `postgres` connection string is defined**. So that in this config file, "Username" should be set to "postgres", and the PG box name and port should be directly set.

So there's 2 ways to activate this feature described above.

If you want to activate it, but want pgSimload to only show othe output some extra information from `pg_stat_replication` system table, then you set the special value "nogucs" like:

```
[...]
    "Replication_info" : "nogucs",
```

```
[...]
```

The other way to activate it is to ask pgSimload to show also settings from the PostgreSQL Primary the whole query will be sent to. In this case, you have to set there all the GUCs you want to be shown, you just have to name those settings separated by a comma in the value of that JSON's fied.

This can be something like:

```
[...]
    "Replication_info" : "synchronous_commit,server_version,work_mem,
       synchronous_commit",
[...]
```

You can look at examples given in `examples/Patroni-Watcher/` README and subdirectories.

Watch_timer (integer)

You can ask for the output in the Patroni-Watcher mode to be like a bash "watch" command: it will run every `x` seconds you define here.

If you want the tool to issue commands each 5 seconds, then set this parameter to simply 5. Since `patronictl` command can take several seconds to run, the value you set here will be computed by the program to match your request, with timers to take into account the time of execution. So then the tool will iterate a bit before going the closest possible to your match your request.

If the value is less than 1, pgSimload will assume you only want to run it once in the Patroni-Watcher mode.

Format (string)

The `patronictl` command offers two modes to list the nodes:

- `list` will order nodes output by name while
- `topology` will show the Primary first, so the order may change if you do a *switchover* of a *failover*

K8s_selector (string)

This parameter has to be set **only if your PostgreSQL Patroni cluster is in Kubernetes**.

The value of this field must be what you'd put in the "selector" chain of that particular `kubectl` command, if you want to get the name of the pod where the current PostgreSQL primary is executing into, and you're running Crunchy Postgres for Kubernetes, that could be done with the following command:

```
$ kubectl get pods \
    -n postgres-operator \
```

```
3       --selector='postgres-operator.crunchydata.com/cluster=hippo,
            postgres-operator.crunchydata.com/role=master' \
4       -o name
```

So pgSimload knows the pod where the Primary PostgreSQL server is running.

The usage of pgSimload in Patroni-Watcher mode in Kubernetes **has requirements**, we urge you to read carrefully the documentation you can access at `examples/Patroni-Watcher/README.md` !

In short, if the Patroni-Watcher mode has to be executed on a cluster of PostgreSQL servers in Patroni, the only relevant paramters in the patroni.json file would then be:

- `Replication_info` : can be set to an empty string (`""`), if you dont need it, `nogucs` or `<list of GUCs separated by a coma>` if you want those informations to be shown. In the later case, you'll need then to run mandatorily with the `-config config.json` parameter too. In than file you'll set a superuser connection (e.g. "postgres" username)
- `Watch_timer` has to be set to a value >1 otherwise it will only runs once
- `Format` has to be set either to `list` or `topology`. In `list`, nodes will be ordered by name, while in `topology`, the Primary will be shown first
- `K8s_selector` has we already seen up there
- all others parameters won't apply, so you can leave them empty ("")

K8s_namespace (string)

This parameter has to be set **only if your PostgreSQL Patroni cluster is in Kubernetes**.

You will set this value to the namespace you're using for your current PostgreSQL Cluster.

If you're using Cruchy Postgres for Kubernetes, and you're actually following the documentation in there, that could be `postgres-operator`, or anything you're actually using.

6.5 Session parameters template file creation

6.5.1 config (value) [MANDATORY]

Same as before, you define in that `config.json` file (or whatever the name, but it has to be a valid JSON here: see previous examples) the connection that will be used to query the pg_settings system view.

You can use whatever user here (i.e. superuser or not), because we only gather the parameters in the `user` context as per pg_settings PostgreSQL documentation.

6.5.2 create_gucs_template (value) [MANDATORY]

This parameter should have been named `create_session_parameters_template_file` to understand what it does…

Here, pgSimload will connect to a given PostgreSQL server as described in the mandatory `-config <config.json>` parameter you have to use too. Then, it will query the system view pg_settings to gather the name and the value (aka `setting`) of each parameter than can be changed in a given session.

Then it will output that in file which format is expected by pgSimload to be passed to the parameter `-session_parameter`.

Beware that those parameters change from one major PostgreSQL version to another, so likely a file you previously generated, then edited to suit your needs, on a version 15 won't work on a version 12.

Also, since ALL parameters in the context `user` will be gather (see pg_settings for details), there will be likely many dozens of parameters here. As an example, as per version 15, it's more than 130 parameters…

Since you probably won't need all of these, most likely, you run that command once to have every parameter in the generated template, then you edit it to remove all uncessary parameters. You'll have then your own template you can use in different scenarios, creating as many `session_parameters.json` you need, to be tested.

7 Release notes

7.1 Version 1.3.0 (April 24th 2024)

7.1.1 Major changes

- Added a new "Kube-Watcher" mode! This one allows to have a minimal monitoring of any PostgreSQL cluster running in Kubernetes, whatever the operator is, since it *only* uses 2 `kubectl` commands, in a loop, to create the monitoring. See examples in `examples/Kube-Watcher/` and the relevant parts in the documentation on how to use it !
  - TL,DR: simple as `pgSimload -kube <kube.json>`
  - Special thanks to "Pierrick" from CloudNativePG Slack, that helped me find the right labels and selectors for that operator to be used in the `kube.json` configuration file !
- documentation update to describe the new "Kube-Watcher" mode.

7.1.2 Minor changes

- changed the way the screen is refreshed in Patroni-Watcher, for better performances and less output in the terminal
  - moved from `github.com/inancgumus/screen`, `screen.Clear()` and `screen.MoveTopLeft()`
  - to simplied ANSI's `fmt.Printf("\x1bc")`
  - **please** let me know if this change break something for you, I can revert that easily in case. I do really lack feedback from pgSimload users!
- simplier way of coding (most often, output messages) strings to be used in output: `A+=B` instead of `A=A+B`. (/me noob)
- added 3 new functions to better padding (left/right) and count length of some output strings (eg podname lenght in Kube-Watcher mode, to align the outputs when more than 1 cluster is listed in this mode)
- added `K8s_selector` field in `patroni.json` configuration file for Patroni-Watcher, so it can be monitoring any namespace other than the default set with, typically, `kubectl config set-context --current --namespace=<namespace_name>`
- renamed in doc, including examples directory and in executables'outputs everything to be consistent among the 3 modes: "SQL-Loop", "Patroni-Watcher" and the new "Kube-Watcher" in version 1.3.0
- lots of doc review, many errors find and corrected

7.2 Version 1.2.0 (April 18th 2024)

7.2.1 Major changes

- Added a new parameter to pgSimload command line to be used in SQL-Loop mode:
  - `-sleep time.Duration` adds a sleep time between 2 iterations (executions) of the `-script script.sql` (or whatever it's name).
  - The interest of this parameter is double:
    * it allows to throttle down the execution in SQL-Loop mode if this one is "going too fast" and

    * it avoids the user to add a line like `select pg_sleep(1);` at the end of the `script.sql`.
  - Actually it corrects indirectly the previous behaviour when that `select pg_sleep(n);` was used previously in `script.sql` around the count of statements executed. This one was only updated once the *whole* script was executed, including the possible `select pg_sleep(n);` at the end.
- documentation update to describe the new `-loops` and `-time` parameters to be used in SQL-Loop mode

7.2.2 Minor changes

- updated `examples/simple` examples and README file

7.2.3 Minor changes

7.3 Version 1.1.0 (January 20th 2024)

7.3.1 Major changes

- Added 2 new parameters to pgSimload command line to be used in SQL-Loop mode:
  - `-loops <int64>` will limit the SQL-Loop execution to that exact number of loops. This can be used to avoid running SQL-Loop endlessly, and/or in comparisons scenarios when one wants to compare effects of various configurations parameters, including using different values when a session parameters files is used (see `session_parameters <JSON.file>` in docs)
  - `-time time.Duration` (where Duration is a duration, without or with double or sigle quotes, like "10s" or 1m10s or '1h15m30s'...). This option will limit SQL-Loop execution to that amount of time. It can be used in various scenarios too
  - when both are used at the same time, the SQL-Loop ends when any one of those conditions is satisfied
- documentation update to describe the new `-loops` and `-time` parameters to be used in SQL-Loop mode

7.3.2 Minor changes

- updated Crunchy copyright ranges to include 2024 (patch by @youattd)
- updated `examples/simple` examples and README file
- updated `examples/patroni_monitoring/README.md` doc to mention the
- added scripts in `examples/patroni_monitoring/ha_test_tools`

7.4 Version 1.0.3 (January 15th 2024)

7.4.1 Major changes

- In SQL Loop mode, don't ping the server in between operations, but only do that **on error** to check that the server is living or not. If not, try reconnecting as before... I used the right method for 1.0.2.. but at the wrong place. Sorry, this was eating useless performances! (ping roundtrip » query exec in most scenarios..)
- don't parse `script.sql`. It's useless because Exec() handles multiple queries on a same file. And it does implicit transactions (so need to add begin/(commit|rollback) in the script file. Results in simplier code and fastest exection too!

7.4.2 Minor changes

- `ioutils` usage replaced with `os`, because `ioutils` is deprecated. So `ioutils` is removed everywhere too
- removed `Read_Config()` function in `main.go` : not used anymore
- review doc to state the major change around parsing/execution
- fix rowcount == 0 in patroni.go / Replication info
- removed the test that looks for `ssh` binary locally, because what is is used is `sshManager.RunCommand(remote_command)`
- constructing the Replication info output in a string and throw it to the screen at once, rather than Println one line by one. To reduce flickering.
- added ComputedSleep() function in `patroni.go` to compute precisely how much to wait between 2 cycles tring to match user's expectations with `Watch_timer` parameter in the `patroni.json` file

- added a warning if the system takes longer to output than the user expects it to be with then `Watch_timer` parameter in the `patroni.json` file
- added a prior check in `sqlloop.go` to check the validity of the SQL in the `script.sql` file
- `strings` and `regexp` packages no more needed in `sqlloop.go`
- `github.com/jackc/pgx/v5` package no more needed in `patroni.go`
- changed “Statements” by “Scripts” in summary and loop info, because it’s no more statements, it’s the *whole* script that it is Exec() at once !
- review error code outputs
- pgReconnectTimeout moved from 30s to 20s, and moved to pgmanager.go (was in patroni.go)
- add more precise numbers in Summary of execution (times of execution/downtime and statements per second)
- corrected paragraph ordering in `doc/05_roadmap.md`

7.5 Version 1.0.2 (January 11th 2024)

- split main.go in many other .go files for better maintenability. This will allow usage of Go Packages further more easily
- split documentation in parts for better maintenability too
- main README.md of the project is a symlink to `/00_readme.md`
- README.md of the `doc/` is the same symlink
- new PGManager for everything

Same I did in version 1.0.1 with SSHManager, now PG connections are handled by a manager. First, this bring cleaner code. Second, it allows pgSimload to function with an unique connection to the PG database, wheter it is used in SQL-Loop mode or Patroni-Watcher mode. It doesn’t change dramatically things in the SQL-Loop mode, because previously, an unique connection was used in the main loop (but others to set transactions GUCS, if used, and Exectute script if used, where still independant connections). But for the Patroni-Watcher, it changes things a lot, allowing the Replication info output to be faster, and offers less “flickering”, because we don’t pay anymore the connexion time, which has the most cost in time execution.

- more code cleaning everywhere

7.6 Version 1.0.1 (January, 8th 2024)

- new SSHManager for Patroni-Watcher mode

The way the Patroni-Watcher is handled in SSH (i.e not in Kubernetes modes) has been refactored. Previously, an SSH connection was initiated at each loop of the Patroni-Watcher. This was not very efficient, because at each `Watch_timer` an SSH connection was opened, the `patronictl` command initiated, the output shown, then the SSH connection was closed.

An SSHManager has then been added to manage this, at not only it is more efficient, and an unique SSH connection is used, but also, it will manage any disconnections of the SSH server itself, trying to reinitiate the SSH connection if the previous died.

A bit more of code refactoring has been added too, so the dependances to the `bytes` and `net` packages have been removed.

- new parameter in `patroni.json` file : `Remote_port` parameter has been added (integer), so you can specify the port of your SSH Server explicitery.
- updated Go modules
- rebuild of binaries
- tagging version 1.0.1
- updated any `patroni.json` file types in the examples to add `Remote_port`
- updated the documentation about `Remote_port` in `patroni.json` files

7.7 Version 1.0.0 (December, 8th 2023)

After 3 months of intensive tests, pgSimload v.1.0.0 is out after the beta period!

What's new? - updated Go modules - rebuild of binaries - tagging version 1.0.0 - minor fixes in documentation (links)

7.8 Version 1.0.0-beta (July, 24th 2023)

First release of pgSimload !

8 Roadmap

8.1 Short term

8.1.1 Scenario mode

With PGManager tools I can now create the Scenario mode.

It will consist of running [1..n] SQL-Loop(s) at the same time on a server to match real world usage scenarios.

A `scenario.json` will be created with: - ID of the client - associated caption to show on screen - `config.json` of that client - `script.sql` of that client - `session_parameters.json` of that client - `create.json` of that client - `execution_number` parameter to configure how many times to execute the script (int64) (eg: 100) - `execution_time` parameter to configure how much time the Loop must be run (time.Duration) (eg: “10m30s”) - `output_type` : “none” or “eta”

The client will end its work wether if the `execution_number` *or* the `execution_time` is satisfied.

The execution in Scenario mode won’t be interactive, except to be launched at start, one will still have to press the Enter key to launch it.

The `output_type` will allow (as a start?) the user to set wheter: - no output at all. The program will finish once every client is disconnected - a nice output on screen with colored progress bars, ETAs, etc (one line per client)

8.1.2 More code cleaning and simplification

Because, heh, I’m a noob Go coder. Trying to do good, but I must admit it’s a long way.

8.1.3 Study and pgmanager.go and jackc

I did this thing but I wonder if I’m using jackc properly… Probably what I’ve did here is already done…

8.1.4 Study and adapt pgmanager.go vs pgcon and pgerrcode

Same thing with pgcon I use for PG Error codes and pgerrcode… I trap some error codes to output a right message to the user, but maybe I’d rather use this project, that contains all the error codes PG has (v.14 still… hopefully PGDG won’t touch this ?)…

8.2 Longer term

8.2.1 Move to packages

Now every parts are in separated .go files I can think about building properly independant packages to manage this.