pgSimload v.1.0.2 Documentation

Jean-Paul Argudo

January, 11th 2023

crunchy data

Contents

1 Overview

Welcome to pgSimload !

The actual version of the program is:

pgSimload version 1.0.2 - January, 11th 2024

pgSimload is a tool written in Go, and accepts 2 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-monitoring 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. 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 main.go <parameters...>
$ go run main.go -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.

Note that Mac and Windows versions aren't fully tested at the moment.

Feedback is welcome in any cases!

2.3 Building binaries

You can use the provided script build.sh.

```
1 $ 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. Those packages will be available soon.

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 can be used in different scenarios:

- 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/` directory. It should straightforward. That file is **not** mandatory, as pgSimload need at least a `-config <file>` and a `-script <file>` to run.
  - the SQL script of your choice. For that purpose you create a plain SQL file, where you put everything you want in it. Beware the parsing is really simple, it would probably fail when creating complex things like functions in this script.
  - 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).

- 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-monitoring 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
- 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 # Examples given

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

3.1 examples/simple/

This example is the simpliest one:

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`.

Here's something for the user to understand:

- `script.one_liner.sql` contains a simple "select 1;" with a sleep(1) **on the same line**, where
- `script.two_liner.sql` contains the same, but each command on one line.

When you pass one or the other in the `-script <script.sql>` parameter, you'll see the difference in counting "statements": actually, pgSimload counts statements as one per line, because the parsing method for the `-script <script.sql>` thing is very basic.

We did tests with a JSON version of it like `-script <script.json>` : it adds an overhead and makes everything unreadable in the end. And pgSimload is not designed to have "exact" parameters and results, the thing is having "some data", and watch differences in between different configurations of PG infrastructure, or parameters (GUCS), etc.

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

```
$ pgSimload -config config.json -script script.one_liner.sql
$ pgSimload -config config.json -script script.two_liner.sql
```

3.2 examples/PG_15_Merge_command/

This example is to test new MERGE command in PG 15.x

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:

```
1 $ 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. We've added 2 flavours.

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

```
1 $ sh watcher.sh query
```

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

```
1 $ sh watcher.sh histogram
```

3.3 examples/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`

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. # 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”.

3.4 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

3.5 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

Name	Mandatory	Optional	Value expected	Description
session_parameters		X	JSON file	Sets special session configuration parameters

3.6 Patroni-monitoring 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

3.7 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

4 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-monitoring 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.

4.1 Common flags and parameters

4.1.1 config (JSON file) [MANDATORY]

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

In the **Patroni-monitoring 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.

4.1.2 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.

4.1.3 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.

4.1.4 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.

4.1.5 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.

4.2 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.

4.2.1 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/PG_15_Merge_command/create.json` or `examples/testdb/create.json` files.

4.2.2 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 newlines. As an example of a more complicated example see `examples/PG_15_Merge_command/script.sql`.

Warning, as per version 0.6, the parsing is very basic for this script : each SQL statement is separated with `;\n`, so, this doesn't fit complex usages.

So consider limiting the content of those files with simple SQL commands, and not creating functions or other more complex things. If you need to create prior functions, do that with `psql ... < create.sql` prior to run pgSimload.

4.2.3 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/testdb/script.
   sql"
Script statements succeeded   : |00000060|
```

… 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/testdb/session_parameters.json`

4.3 Patroni-monitoring mode flag and parameters

To use have pgSimload act as a small Patroni-monitoring 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.

4.3.1 patroni (value) [MANDATORY]

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

```
$ 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"     : ""
}
```

Cluster

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

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_port

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.

Remote_user

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.

Use_sudo

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

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

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** a `-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 at `examples/patroni_monitoring/`.

Watch_timer

You can ask for the output in the Patroni-monitoring 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

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

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:

```
$ kubectl get pods --selector='postgres-operator.crunchydata.com/
    cluster=hippo,postgres-operator.crunchydata.com/role=master' -o name
pod/hippo-instance1-mr6g-0
```

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

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

In short, if the Patroni monitoring 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 ("")

4.4 Session parameters template file creation

4.4.1 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.

4.4.2 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.

5 Release notes

5.1 Version 1.0.2 (January 11th 2023)

- 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

5.2 Version 1.0.1 (January, 8th 2024)

- new SSHManager for Patroni Watched 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

5.3 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)

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

First release of pgSimload !

6 Roadmap

6.1 Short term

6.1.1 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` associated file - `script.sql` to be used in the loop - `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)

6.2 More code cleaning

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

6.3 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…

6.4 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 ?)…

6.5 Longer term

6.5.1 Move to packages

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