PgLoader, the parallel ETL for PostgreSQL

Dimitri Fontaine

October 17, 2008

Table of contents

ETL

Definition

An ETL process data to load into the database from a flat file.

1. Extract
2. Transform
3. Load

pgloader's features

PGLoader will:

- Load CSV data

pgloader's features

PGLoader will:

- Load CSV data
- Load pretend-to-be CSV data

pgloader's features

PGLoader will:

- Load CSV data
- Load pretend-to-be CSV data
- Continue loading when confronted to errors

pgloader's features

PGLoader will:

- Load CSV data
- Load pretend-to-be CSV data
- Continue loading when confronted to errors
- Apply user define transformation to data, on the fly

pgloader's features

PGLoader will:

- Load CSV data
- Load pretend-to-be CSV data
- Continue loading when confronted to errors
- Apply user define transformation to data, on the fly
- Optionaly have all your cores participate into processing

Configuration

We first parse the configuration, with templating system

Example

```
[simple]
use_template = simple_tmpl
table        = simple
filename     = simple/simple.data
columns      = a:1, b:3, c:2
```

Loading: file reading

PGLoader supports many input formats, even if they all look like `CSV`, the rough time is parsing data:

- Read files one line at a time

Loading: file reading

PGLoader supports many input formats, even if they all look like `CSV`, the rough time is parsing data:

- Read files one line at a time
- Parse physical lines into logical lines

Loading: file reading

PGLoader supports many input formats, even if they all look like `CSV`, the rough time is parsing data:

- Read files one line at a time
- Parse physical lines into logical lines
- Supports several readers
    - csvreader

Loading: file reading

PGLoader supports many input formats, even if they all look like `CSV`, the rough time is parsing data:

- Read files one line at a time
- Parse physical lines into logical lines
- Supports several readers
  - textreader
  - csvreader

Loading: file reading

PGLoader supports many input formats, even if they all look like `CSV`, the rough time is parsing data:

- Read files one line at a time
- Parse physical lines into logical lines
- Supports several readers
    - textreader
    - csvreader
    - fixedreader

Processing lines

Parsing data is the `CPU` intensive part of the job. You could even have to guess where lines begin and end.

Processing lines

Parsing data is the CPU intensive part of the job. You could even have to guess where lines begin and end. Then you add:

- columns restrictions

Processing lines

Parsing data is the CPU intensive part of the job. You could even have to guess where lines begin and end. Then you add:

- columns restrictions
- columns reordering

Processing lines

Parsing data is the `CPU` intensive part of the job. You could even have to guess where lines begin and end. Then you add:

- columns restrictions
- columns reordering
- user defined columns (constants)

Processing lines

Parsing data is the `CPU` intensive part of the job. You could even have to guess where lines begin and end. Then you add:

- columns restrictions
- columns reordering
- user defined columns (constants)
- user defined reformating modules

COPYing to PostgreSQL

This is how we do it:

- python `cStringIO` buffers

COPYing to PostgreSQL

This is how we do it:

- python `cStringIO` buffers
- configurable size (`copy_every`)

COPYing to PostgreSQL

This is how we do it:

- python `cStringIO` buffers
- configurable size (`copy_every`)
- using `copy_expert()` when available (CVS)

COPYing to PostgreSQL

This is how we do it:

- python `cStringIO` buffers
- configurable size (`copy_every`)
- using `copy_expert()` when available (CVS)
- dichotomic error search

Handling of erroneous data input

PGLoader will continue processing your input when it contains erroneous data.

- reject data file

Handling of erroneous data input

PGLoader will continue processing your input when it contains erroneous data.

- reject data file
- reject log file, containing error messages

Handling of erroneous data input

PGLoader will continue processing your input when it contains erroneous data.

- reject data file
- reject log file, containing error messages
- errors count in `summary`

Error logging

PGLoader will continue processing your input when it contains erroneous data, and will make it so that you know about the failures.

- log file

Error logging

PGLoader will continue processing your input when it contains erroneous data, and will make it so that you know about the failures.

- log file
- console log level: `client_min_messages`

Error logging

PGLoader will continue processing your input when it contains erroneous data, and will make it so that you know about the failures.

- log file
- console log level: `client_min_messages`
- logfile log level: `log_min_messages`

Why going parallel?

Loading is IO bound, not CPU bound, right?

Why going parallel?

Loading is IO bound, not CPU bound, right?

- for large disks array, *not so much*

Why going parallel?

Loading is IO bound, not CPU bound, right?

- for large disks array, *not so much*
- with complex parsing, *not so much*

Why going parallel?

Loading is IO bound, not CPU bound, right?

- for large disks array, *not so much*
- with complex parsing, *not so much*
- with heavy user rewritting, *not so much*

Ok... How?

- mutli-threading is easy to start with in python

Example

```
    class PGLoader(threading.Thread):
```

Ok... How?

- mutli-threading is easy to start with in python
- then you add in dequeues and semaphores (critical sections) and signals

Example

```
class PGLoader(threading.Thread):
```

Ok... How?

- mutli-threading is easy to start with in python
- then you add in dequeues and semaphores (critical sections) and signals
- Giant Interpreter Lock

Example

```
class PGLoader(threading.Thread):
```

Ok... How?

- mutli-threading is easy to start with in python
- then you add in dequeues and semaphores (critical sections) and signals
- Giant Interpreter Lock
- fork() based reimplementation could be of interrest

Example

```
class PGLoader(threading.Thread):
```

Parallelism choices

Has beed asked by some hackers, their use cases dictated two different modes.

Parallelism choices

Has beed asked by some hackers, their use cases dictated two different modes.

The idea is to have a parallel pg_restore testbed, interresting with large input files (100GB to several TB). PGLoader's can't compete to plain `COPY`, due to clientserver roundtrips compared to local file reading, but with some more CPUs feeding the disk array, should show up nice improvements.

Parallelism choices

Has beed asked by some hackers, their use cases dictated two different modes.

The idea is to have a parallel pg_restore testbed, interresting with large input files (100GB to several TB). PGLoader's can't compete to plain `COPY`, due to clientserver roundtrips compared to local file reading, but with some more CPUs feeding the disk array, should show up nice improvements.

Testing and feeback more than welcome!

Round robin reader

Parsing is all done by a single thread for all the content.

Round robin reader

Parsing is all done by a single thread for all the content.

`N` readers are started and get each a queue where to fill this round data, and issue `COPY` while main reader continue parsing.

Round robin reader

Parsing is all done by a single thread for all the content.

N readers are started and get each a queue where to fill this round data, and issue COPY while main reader continue parsing.

Example

```
[rrr]
section_threads    = 3
split_file_reading = False
```

Split file reader

The file is split into `N` blocks and there's as much pgloader doing the same job in parallel as there are blocks.

Split file reader

The file is split into N blocks and there's as much pgloader doing the same job in parallel as there are blocks.

Example

```
[rrr]
section_threads    = 3
split_file_reading = True
```

Examples

PGLoader distribution comes with diverse examples, don't forget to see about them.

simple

That simple:

simple

That simple:

Example

```
[simple]
table         = simple
filename      = simple/simple.data
format        = text
datestyle     = dmy
field_sep     = |
trailing_sep  = True
columns       = *
```

User defined columns

Constant columns added at parsing time.

User defined columns

Constant columns added at parsing time.

Use case: adding an origin_server_id field depending on the file to get loaded, for data aggregation.

User defined columns

Constant columns added at parsing time.

Use case: adding an origin_server_id field depending on the file to get loaded, for data aggregation.

Example

```
 [server_A]
file            = imports/A.csv
columns         = b:2, d:1, x:3, y:4
udc_c           = A
copy_columns    = b, c, d
```

User defined Reformating modules

The basic idea is to avoid any pre-processing done with another tool (`sed`, `awk`, you name it).

User defined Reformating modules

The basic idea is to avoid any pre-processing done with another tool (sed, awk, you name it).
file has '12131415'

Example

```
[fixed]
table          = fixed
format         = fixed
filename       = fixed/fixed.data
columns        = *
fixed_specs    = a:0:10, b:10:8, c:18:8, d:26:17
reformat       = c:pgtime:time
```

User defined Reformating modules

The basic idea is to avoid any pre-processing done with another tool (`sed`, `awk`, you name it).

file has `'12131415'` we want `'12:13:14.15'`

Example

```
def time(reject, input):
    """ Reformat str as a PostgreSQL time """
    if len(input) != 8:
        reject.log(mesg, input)

    hour        = input[0:2]
    ...
    return '%s:%s:%s.%s' % (hour, min, secs, cents)
```

The fine manual says it all

At `http://pgloader.projects.postgresql.org/` or `man pgloader`

Example

```
> pgloader --help
> pgloader --version
> pgloader -DTsc pgloader.conf
```

TODO

http://pgloader.projects.postgresql.org/dev/TODO.html

TODO

`http://pgloader.projects.postgresql.org/dev/TODO.html`

- Constraint Exclusion support

TODO

`http://pgloader.projects.postgresql.org/dev/TODO.html`

- Constraint Exclusion support
- Reject Behaviour

TODO

`http://pgloader.projects.postgresql.org/dev/TODO.html`

- Constraint Exclusion support
- Reject Behaviour
- XML support with user defined XSLT StyleSheet

TODO

`http://pgloader.projects.postgresql.org/dev/TODO.html`

- Constraint Exclusion support
- Reject Behaviour
- XML support with user defined XSLT StyleSheet
- Facilities

TODO

`http://pgloader.projects.postgresql.org/dev/TODO.html`

- Constraint Exclusion support
- Reject Behaviour
- XML support with user defined XSLT StyleSheet
- Facilities

Don't be shy and just ask for new features!

Resources and Users

pgfoundry, 1 developper, some users, no mailing list yet (no one asking for one), some mails sometime, seldom bug reports (fixed)

Resources and Users

pgfoundry, 1 developper, some users, no mailing list yet (no one asking for one), some mails sometime, seldom bug reports (fixed)

Support ongoing at `#postgresql` and `#postgresqlfr`

Resources and Users

pgfoundry, 1 developper, some users, no mailing list yet (no one asking for one), some mails sometime, seldom bug reports (fixed)

Support ongoing at `#postgresql` and `#postgresqlfr`

packages for `debian`, `FreeBSD`, `OpenBSD`, `CentOS`, `RHEL` and `Fedora`.