Implementing pgloader

from python to Common Lisp

Dimitri Fontaine
2ndQuadrant France
PostgreSQL Major Contributor
dimitri@2ndQuadrant.fr

ABSTRACT

In this paper, we use the example of rewriting from scratch a data loader application for PostgreSQL from python to Common Lisp to compare the offerings of those two well established *Dynamic Languages*.

1. PGLOADER

pgloader is an application to load external data into a PostgreSQL database. The external data is often organised into a *flat file* formatted in the infamous *CSV* format, or sometimes in a *fixed width specification* format.

While PostgreSQL includes the COPY command and a *streaming protocol*, it will not parse and accept any input format and the transactional behavior is not optimal in the case of *loading* an important set of data. Whether *important* means that the data is of value or that the set of data is big, the COPY command being run in a single transaction means that any error in the input will cancel the whole loading.

That's where pgloader comes into play, by dealing with errors and implementing a solution on top of the COPY command and protocol so that it is able to separate the good input from the bad one and provide both a *log file* and a *reject file* for later reprocessing.

The very first version of pgloader had been written in the TCL dynamic programming language. The second version series was a complete rewrite in Python and came with new features. After almost a decade of maintaining that code, the need to better handle huge volumes of data and some problems tied to the choice of Python led pgloader author to reconsider the programming language used to implement it, and the third version of pgloader is being written from scratch again this time in Common Lisp.

1.1 The PostgreSQL COPY protocol

PostgreSQL includes a very efficient data loading protocol which works in a streaming fashion and is called the COPY protocol. There's also a command of the same name so that you do not need to read the protocol documentation in order to benefit from the improved performance characteristics.

Of course, the COPY command happens within a single transaction. That means that in case of failure to load any row of data, the whole loading is aborted. In most cases, that's not what we want, so pgloader needs a way around that.

1.2 Handling errors in data

PostgreSQL checks that the input data you send at it conforms to the data type you've specified, in a way that some other systems are not used to. So for example the date 0000-00-00 is refused with an ERROR, because the year zero is not part of our calendar, which goes from year -1 to year 1.

The way to address data problems is simple and effective: split the input data in batches and submit them in separate commands and transactions. In case of error, split the batch in halves and submit each half independantly again. When the batch contains only 1 row and you have an error, then log that as a *bad data* and continue.

2. THE PGLOADER COMMAND LANGUAGE

The current version of pgloader has no command language yet, and this lack of an advanced *Domain Specific Language* is a limiting factor to the possible evolutions of the software.

To help with understanding what pgloader is all about, this section presents the *work in progress* language syntax. While some basic implementation has been done with *pyparsing* in Python and with *cl-yacc* in Common Lisp, another tool to test is the *smug Monadic Parser Combinators*.

Two alternatives are being studied for the pgloader command language, first a COPY variant:

```
COPY cluttered
FROM 'cluttered/cluttered.data'
      (a, c newline escaped by \, b)
  AS text
WITH field_sep = ^, field_count = 3;
```

This command would read as: please *copy* data from the cluttered/cluttered.data file which has *3 fields* separated by the ^ character, and which is not respecting the *CSV* format as the second column contains *newline* characters escaped with a *backslash*, unquoted. Also, the fields int the file are in order (a, c, b) when matching them with the names of the columns in the database table.

And also a LOAD variant:

```
LOAD foo
FROM 'path/to/file'
  AS text
CASE WHEN 1:2 = "43"
     THEN table(a, c)
     SPEC (a sep ';',
           b sep '=', -- field is not loaded
           c sep ';')

     WHEN 001:003 = "HDR"
     THEN table(a, c)
     SPEC (a, b, c)
     WITH field_sep = ','
 END
 SET maintenance_work_mem TO '128 MB';
```

That LOAD command would load data from path/to/file into the table named table, into its columns a and c. It would deal with a very strange file format indeed, such as when the line begins with "43" then the file has a column followed by the ; separator, then another column followed by the = separator, then a third column followed by a ;. When the line begins with "HDR" then there's only one field separator and this times it's a comma (,).

It's possible to find such file formats for real, generally it's easier to fix the exporting process though. Still, it's a feature set that's been asked by pgloader users used to other tools able to cope with such oddities.

The main advantage of the LOAD variant is to stay away from PostgreSQL's own COPY command and syntax, as PostgreSQL already ships with both a server and a client version of that command (on the client, it's spelled \copy).

3. IMPLEMENTING PGLOADER IN PYTHON

This implementation already was a rewrite, so we mostly had the specifications of the program and even some practical ways to attack the main problem which is all about separating away the *bad data* so that the *good data* is loaded into PostgreSQL.

pgloader had to be rewritten in order to complement a data migration from *Informix* to PostgreSQL, where the best way to attack things was to use the Informix UNLOAD command then load back the data into PostgreSQL. The then current version of pgloader (in TCL) was not able to read those files because of encoding problems, and it appeared very soon that the input format wouldn't be easy to process.

The UNLOAD command output is kind of interesting. It pretends to be somewhat like a CSV derivative, but only is able to maintain that appearance for some seconds, as it will not quote values even when they expand over multiple lines.

3.1 Main design

Even if the task at hand was restricted to parsing the UNLOAD ouput, the design was made with *custom CSV* format instead so as to be able to reuse code later. To that end, any random bits about the input format was turned into a non default setup in the way pgloader would parse the flat files, so that the default setup could remain sane and usable outside of that project.

The first version of that code then was simple enough:

- fetch the current setup to drive the run,
- init a loader object per command
- parse the file into *rows*,
- load the *rows* into PostgreSQL with a database object.

This design is quite classic and *Object Oriented*: the work to be done has been organized into sub-parts, each of them being handled by a specific object that knows nothing about the rest of the system.

While this design is good for code *modularity*, it's not good at all for *flexibility* and growing features into the code base, or even for design refactoring. Typically, adding *template* sections in the configuration file format and later *parallelism* to load the file both have included really disruptive changes in the code base.

3.2 Handling errors in data

The Python implementation of that idea uses a cStringIO buffer that is sent to COPY when considered full. That involves more data copying than strictly necessary, from the input reader buffer to the internal batch buffer then to the COPY buffer, which is implemented in a C extension that very well might include another copying of the source data.

3.3 Those infamous .INI files

In Python when you need to read a setup usually you're using some *user* format, meaning not *Python* itself. As the default distribution of Python includes the capability to parse INI files, that's the easiest way to get started.

There are only two other possibilities, either write a specialized parser or hand over Python itself to your users. Even the second one is not as easy as it sounds, which favors the INI format a lot.

The problem with the INI format is with representing complex commands rather than just simple key-value setup, and it so happens that what pgloader really needed was a *complex command* input.

That *setup* versus *command* problem is one important factor into the decision to rewrite pgloader in Common Lisp: one of the goals had been to keep it as something that is easy to install, as easy as a single file script if at all possible. While that is not been true of long, the latest Python version of pgloader still is easily *relocatable* and only depends on a bare Python installation.

Adding a command language would require a new dependency, and the best candidate that has been tried has been *pyparsing*, see above for some examples of the command language design candidates.

3.4 A try at going parallel workers

Little did I know about the Global Interpreter Lock when I convinced myself that a parallel approach is what pgloader needed to be faster at what it did.

About the *Global Interpreter Lock*, the Python documentation says that "The Python interpreter is not fully thread-safe. In order to support multi-threaded Python programs, there's a global lock, called the global interpreter lock or GIL, that must be held by the current thread before it can safely access Python objects", and then continues saying that "Therefore, the rule exists that only the thread that has acquired the GIL may operate on Python objects or call Python/C API functions".

Two concurrency approaches are possible here:

- have a reader that fills a shared queue and several consumers on that queue that will each handle separate batches

- have several input file readers each working on a chunk of the input file and doing the whole processing of what they read.

Those two approaches have been implemented in `pgloader` using `Threads` and a `BoundedSemaphore` in the main controler thread. Of course, the *GIL* makes it so that no improvement has been experienced from there.

The way to properly address the concurrency loading of data with Python would be using `multiprocessing` and find a way to communicate data in between the main reader and the workers. The problem with the communication is that we don't have the same facilities depending on the OS: no pipes on windows, where you have to use local sockets.

Switching from *threads* to *multiprocessing* has been on the todo list for a while, but didn't get addressed before the rewrite, and wouldn't have possibly solved the problems of copying too much data in memory, nor the command language problem.

3.5 Python as a platform

Meantime, the great question had to be answered: which version of Python to target in your applications, *Python2* or *Python3*? Note that any answer here is not going to help solving the three main problems of pgloader at this point:

- the command syntax;
- the concurrency capabilities;
- how to really improve performances.

4. COMMON LISP

Common Lisp has been made to ease the developer task, and offers a quite unique *dynamic programming* experience.

4.1 A unique programming experience

To program in Lisp, the first step is to start the *Lisp Image* which contains all the facilities you expect as a developer: compiler, command line interaction which acts like a interpreter, etc. From within this environment you can load your code and play with it interactively.

When using an integrated environment, you get some facilities to edit the code in some files and load whatever *form* you're currently working on in the image. So if you write a new function you can have only that function compiled and loaded, and you can test it right away at the prompt.

And when you change the function definition, you can compile the new version and load it into the running Lisp program in exactly the same way, and continue playing with your test case at the REPL only with the new definition now.

This flexibility includes also reloading whole Lisp files or *systems* live in the environement, at runtime. And it also includes data (variables) and data types: structures and classes can be reloaded in the live environement and the objects of those type will get migrated to the new type definition automatically by the system, when that makes sense.

If any error in your program or test occurs, the interactive debugger opens and you can inspect what's going on and of course evaluate expressions within the scope of the faulty frame.

While contemplating the *dynamic* nature of a Lisp system, it's important to remember that the code is actually compiled to *machine code*. You have full access to the compiler at runtime, and when you execute the function you just loaded from your editor with a single key-chord, that function is actually compiled to machine code within the running Lisp image.

There's no reason in Lisp to compromise between performance characteristics and a very *dynamic programming* experience, you have it all.

4.2 Data Structures

Lisp is known to be all about *list processing*, and that has to do with how the code is written: s-expression forms. The programmer still has access to the most common data structures, not just lists: symbols, integers, fixnums and bignums, rational numbers, hash tables, vectors, bit vectors, integers as bit vectors, multi-dimensionnal arrays, structures, classes, user defined types, characters, strings, conses and lists, sequences, filenames, files, streams.

The common Lisp standard is quite verbose about the exact behaviour of each of those types, and they are very well integrated overall.

Also, in Common Lisp the values are typed, not the bindings or the variables. Which means that the data type checked are dynamic and will only fire when the *value* is not of the expected type, independently of the binding holding the value. That's a great flexibility for *dynamic programming*.

It means, though, that the compiler will not typically be able to help you catch typing errors straight from the code it sees, as the behavior is all to be determined at run time. Common Lisp provides features to help the programmers in that area: it's possible to `declare` the expected type of any variable, parameter or binding so that the compiler knows how to produce optimized code. Then if you're lying to the compiler, the behavior is undefined. It's possible to use `check-type` to catch those erros and have a deterministic behavior or signaling a correctable error of type `type-error` instead.

4.3 Imperative, Functional, Object Oriented, all at once

As is the case with Python, Common Lisp doesn't care which tools you prefer for your code. It's possible to mix imperative, functionnal and object oriented code in the same program. The Object Oriented system is called *Common Lisp Object System* or *CLOS* for short.

Common Lisp implements multiple-dispatch with support for dispatching not only on full blown CLOS objects but also on lighter *structures*. Any standard type can also be used to dispatch on a *method* (such as `integer` or `vector`), and there's the *eql specializer* allowing to dispatch on *symbols*.

Here's a simple example of implementing the `flatten` function with the help of the *CLOS* system:

```
(defmethod flatten ((x list))
  (reduce 'append (mapcar 'flatten x)))

(defmethod flatten (x)
  (list x))

CL-USER> (flatten '(1 2 (3 4 ) 5))
(1 2 3 4 5)
```

The first *method* is only called when the given argument is a list object, otherwise the other *method* is executed.

4.4 General syntax for assignments

Common Lisp has a concept of *places* where you can assign *values*. That concept is also know as a *generalized reference* and allows the implementation of the `setf` macro, that you as a programmer can expand to handle your own objects. Let's see an example, assigning to a hash table:

```
CL-USER> (defvar ht (make-hash-table))
HT
CL-USER> (gethash 'key ht)
NIL
NIL
CL-USER> (setf (gethash 'key ht) 'value)
VALUE
CL-USER> (gethash 'key ht)
VALUE
T
```

The *setf* macro offers a way to *register* your own assignements and is provided with support for the included data types when that's relevant, such as hash tables. We can expand the macro and see what it does:

```
CL-USER> (macroexpand-1
           `(setf (gethash key ,ht) value))
(CCL::PUTHASH
  KEY
  #<HASH-TABLE :TEST EQL size 2/60 #x30200E09F46D>
  VALUE)
T
```

Now, it's also possible to directly go increment a counter stored inside the hash table, like this:

```
CL-USER> (setf (gethash 'counter ht) 0)
0
CL-USER> (incf (gethash 'counter ht))
1
CL-USER> (incf (gethash 'counter ht))
2
```

All the mechanics to add *generalized references* to your own code are of course provided, and the easiest way to benefit from them is using a *structure* as in the next example:

```
CL-USER> (defstruct person (name "007" :type string))
PERSON
CL-USER> (make-person :name "James")
#S(PERSON :NAME "James")
CL-USER> (person-name (make-person :name "James"))
"James"
```

Here when defining the `person` *structure* the `defstruct` macro will automaticall create a bunch of function for us, a constructor and accessors. The best part is that it's easy to write such a macro, so that power is all available to the programmer too.

That *general syntax for assignments* allow programmers to use a unified programming style based on forms that can look *functionnal* for the untrained. Using assignment is definitely an *imperative* style of programming though.

4.5 Scoping rules

A binding binds a variable symbol with a value. Lexical scope means that a binding is only available from within the lexical extent of code in which it appears. Dynamic binding means that the binding's new value remains visible for the whole extent of all the nested code that is run.

The API `defvar` and `defparameter` both introduce *dynamic bindings*, and you can `declare` any binding to be *special* to have them dynamic too within a specific run-time code extent.

By default, any other binding is introduced as lexically scoped.

Here's a little example including *dynamic bindings*

```
(defparameter *days*
  '(monday tuesday wednesday
    thursday friday saturday sunday)
  "List of days in the week")

(defun any-day-but-monday? (day)
  "Returns a generalized boolean,
   true unless DAY is 'monday"
  (member day
          (remove-if (lambda (day) (eq day 'monday))
                     *days*)))

CL-USER> (any-day-but-monday? 'monday)
NIL

CL-USER> (any-day-but-monday? 'tuesday)
(TUESDAY WEDNESDAY THURSDAY FRIDAY SATURDAY SUNDAY)
```

We can see in that example that the binding `day` is a different *variable* within the `member` form and within the `lambda` form, thanks to lexical binding, whereas the binding `*days*` is bound to the same value.

This example is also using the notion of a *generalized boolean* in Common Lisp where `nil` is considered to mean *false* and anything else is then *true*. Remember that `nil` and `'()` (the empty list) are the same thing in Common Lisp.

4.6 Compiler

Having a compiler allows to have compile-time checks about the program you're writing, and to avoid spending too much time on program testing. The most helpful compiler warning I came to appreciate in Common Lisp is the *Unused lexical variable* one that somehow always pair with the *Undeclared free variable* one, meaning there's yet another variable name with a typo in the code.

4.7 Advanced control flow operators

The two most important things to consider when programming certainly are the data structures and then the control flow operators. Common Lisp provides with all you need as a programmer.

4.7.1 Looping

Basic flow control operators include `dotimes` and `dolist`, and Common Lisp also includes the very general facility `loop`, which we have to see an example of as it's the kitchen sink of looping constructs:

```
CL-USER> (defun fib-list (n)
           "Fibonnacci suite from 1 to n"
```

```
          (loop
             repeat n
             for x = 0 then y
             and y = 1 then (+ x y)
             collect y))
FIB-LIST
CL-USER> (fib-list 10)
(1 1 2 3 5 8 13 21 34 55)
CL-USER> (time (last (fib-list 100)))
(LAST (FIB-LIST 100))
took 15 microseconds (0.000015 seconds) to run.
During that period, and with 4 available CPU cores,
     15 microseconds were spent in user mode
     15 microseconds were spent in system mode
 1,984 bytes of memory allocated.
(354224848179261915075)
```

Note: The *timing* has been edited to fit in the paper column size.

To understand this example you need to know that:

- the look will stop as soon as it's been done n times,
- the `for/and` construct introduces a *parallel* per-loop assignment, meaning that at the time of the assignement no binding *sees* the new values or the previous binding,
- it's possible to have *serial* assignment by just using as many `for` constructs as you need, one after the other,
- `collect` will prepare a list of its arguments and have the whole `loop` form return that when it's done.

The *loop* facility also supports initialising and final steps, and collecting elements, as we can see in a more complex real-world example:

```
      (loop
         with schema = nil
         for (table-name . coldef)
         in
           (caar (cl-mysql:query (format nil "
  select table_name, column_name,
         data_type, column_type, column_default,
         is_nullable, extra
    from information_schema.columns
   where table_schema = '~a'
order by table_name, ordinal_position" dbname)))
         do
           (let ((entry
                   (assoc table-name schema :test 'equal)))
             (if entry
               (push coldef (cdr entry))
               (push (cons table-name (list coldef))
                     schema)))
         finally (return schema))
```

4.7.2 *Lexical and dynamic flow control*

The advanced control flow operators include `block`, `return` and `return-from` that allows to jump from wherever you are in a lexical code structure back to the introduction of the *block*. Of course any function is a block.

The `tagbody` and `go` special forms act like a generalized `goto` statement limited to a lexical environement.

The `unwind-protect` form guarantees that some cleanup forms will be run, whatever happens in the *protected* forms at run time. Here's an example where we want to be sure to free our resources, even if we fail to run a query:

```
(cl-mysql:connect :host host :user user :password pass)
(unwind-protect
     (mapcan #'identity
             (caar (cl-mysql:query "show databases")))
   (cl-mysql:disconnect))
```

It's also possible to control the flow in more dynamic ways, thanks to the `throw` and `catch` constructs and with the ability to `signal` a *condition*.

4.8 Lambda lists

Each time in Common Lisp when you can construct a list of arguments to be matched against, you deal with some form of a *lambda list*. The main examples are of course `defun` with `defgeneric` and `defmethod`, but some more interesting uses are possible with `destructuring-bind`.

```
         ; helper function
CL-USER> (defun iota (n)
           (loop for i from 1 to n collect i))
IOTA
CL-USER> (iota 3)
(1 2 3)
CL-USER> (destructuring-bind (a b c)
             (iota 3)
           ;; do interesting things with a, b and c
           (+ a (* b c)))
7
```

The allowed matches for the lambda lists are `&key` for named arguments, `&optional` for arguments that you might bypass at call sites (you can provide default values for them in the lambda list of course), `&rest` to collect a bunch of parameters together inside a single list, allowing a unknown number of parameters to be passed in, and `&aux` to declare extra local bindings from the parameter list itself.

Of course `&aux` parameters default values can be expressed in terms of other parameters, that are always processed first.

And it's possible to mix and match about any of those kinds of parameters in the same *lambda list*.

4.9 It's Lisp all the way down

Having access to the Common Lisp compiler at run time means that you can actually generate code (e.g. using `defmacro`) that is compiled once then run as usual, and you can do that either at compile time or at run time. It's even more powerful than that, because the *reader* step is clearly separated away from the *compiler* step. And of course the reader itself is implemented in Lisp and accepts *reader macros*, allowing you to *dispatch* your own Lisp code so that it's used when *reading* some code.

Most other programming environements need several helper tools (such as `make`, sometime some level of `awk` or `m4`, etc) to add some levels of flexibility in preparing the code before the build process can have at it. In Lisp it's considered a part of the programming to have to prepare the code, and you can do that in Lisp too.

4.10 Multi Threading

Multi threading is not part of the Common Lisp standard document, so different implementations are free to implement it in different ways. In practice though, a portable library has been made that offers a consolidated threading API to programmers in a portable way: it uses each implementation constructs so that the common API behaves always the same. That library is called *bordeaux-threads*.

On top of *bordeaux-threads* it's possible to use *lparallel*, a library with concepts such as `queues` and `workers`, making it easy to have parallel workers sharing a work load and using communication channels in between them rather than having to share some memory and implement the right locking behavior.

The way to share memory in between threads is quite easy in Common Lisp when equiped with *bordeaux-threads*: declare a *special* variable (e.g. using `defvar`) and only bind it from within threads. Then they will all have access to the same binding and will share it. Now implement locking if you need some.

To avoid locking, you can restrict the ability to actually edit the contents of the shared memory in a single worker thread and use a communication facility for that worker to receive the data changes to apply to the shared object. *lparallel* makes implementing that rather straightforward.

5. IMPLEMENTING PGLOADER IN COMMON LISP

Now that we have a great replacement option for Python, allowing us to achieve much better performances while also adding the ability to easily embed an advanced *command syntax* and real concurrency, it's time to actually use the new tools and rewrite `pgloader`.

5.1 First attempts at writing `pgloader` in Common Lisp

The main problem to solve had been the `pgloader` command syntax. So the first code written in Common Lisp attempted to solve that problem, and was a command parser for `pgloader`, using a syntax in between the PostgreSQL `COPY` command and the *SQL Loader* `LOAD` command.

Writing a parser without having an API to target proved a rather fruitless task, so the real way to attack the problem is to get back to basics: loading data from flat file, pushing that data into PostgreSQL using the `COPY` protocol.

Meanwhile, the early history of `pgloader` stroke back in a good way, as a new customer wanted to migrate his MySQL databases to PostgreSQL. We did use the Python version of pgloader to that end, with some difficulties, and a migrating timing that wasn't appropriate. Something needed to be done now.

5.2 Main design

The main things that pgloader will have to do are:

- read data from some input, either flat files or something else
- process that data so as to have proper rows (list of strings)
- bulk load the rows using the `COPY` protocol
- handle data errors and batch retry

Having in mind the limitations of the Python version of `pgloader`, the most attractive way to address the needs here is using some concurrency already:

- a reader thread reads the data and pre-process it
- then fills a shared queue
- a copy thread reads from the queue, builds batches and copy them over

The Common Lisp package lparallel is implementing some high level facilities to hand over a task to a thread worker, and provides a *queue* to share data through. Using that, it's been easy to implement a streaming behaviour for the data loader.

5.3 lparallel queues

Thanks to our `pgloader` use case, the `lparallel` queues are now available either with auto-expand behavior or in a fixed capacity. The data streaming needs the latter so that it acts like a Unix Pipe and blocks the reader when the writer is not keeping up.

The main entry point for a new data source reader is a `map-rows` function that will know how to parse the data input and map a given function on each of those rows.

It's then possible to install a generic *map* to *queue* function to handle the specifics of our internal *streaming protocol*, as we can see here:

```lisp
(defun map-push-queue (queue map-row-fn &rest initial-args)
  "Apply MAP-ROW-FN on INITIAL-ARGS and a function of ROW
that will push the row into the queue. When MAP-ROW-FN
returns, push :end-of-data in the queue.

Returns whatever MAP-ROW-FN did return."
  (unwind-protect
       (apply map-row-fn
          (append initial-args
              (list
               :process-row-fn
               (lambda (row)
             (lq:push-queue row queue)))))
    ;; in all cases, signal the end of the producer
    (lq:push-queue :end-of-data queue)))
```

5.4 The PostgreSQL batch writer

The writer side implements a `copy-from-queue` function that knows how to *pop* data from the queue and send it over to PostgreSQL. Note that this processing only needs to keep a copy of the input data to process it again in case of failure to `commit` the batch.

Common Lisp includes some very useful control flow facilities, and it's possible to use `throw` and `catch` in order to implement reading from the queue as as single function and still stop at batch boundaries to go commit pending work, handle the retrying when necessary, etc.

In the following code excerpt, the `map-pop-queue` function is fetching data from the queue and knows nothing about splitting the data into batches and having to retry in case of errors:

```lisp
(defvar *batch* nil
  "Current batch of rows being processed.")

(defvar *batch-size* 0
  "How many rows are to be found in current *batch*.")

(defun make-copy-and-batch-fn (stream)
  "Returns a function of one argument, ROW.

   When called, the function returned reformats the
   row, adds it into the PostgreSQL COPY STREAM,
   and push it to BATCH (a list). When batch's size
   is up to *copy-batch-size*, throw the
   'next-batch tag with its current size."
  (lambda (row)
    ;; maintain the current batch
    (push reformated-row *batch*)
    (incf *batch-size*)

    ;; send the reformated row in the PostgreSQL
    ;; COPY stream
    (cl-postgres:db-write-row stream row)

    ;; return control in between batches
    (when (= *batch-size* *copy-batch-size*)
      (throw 'next-batch
             (cons :continue *batch-size*)))))
```

Then the idea is to stream the queue content directly into the PostgreSQL COPY protocol stream, but COMMIT every *copy-batch-size* rows.

That allows to have to recover from a buffer of data only rather than restart from scratch each time we have to find which row contains erroneous data. BATCH is that buffer.

So that the main copy-from-queue function is now doing the following in a loop:

```lisp
(let* ((stream
         (cl-postgres:open-db-writer conspec
                                     table-name
                                     nil))
       (*batch* nil)
       (*batch-size* 0))
  (unwind-protect
    (let ((process-row-fn
           (make-copy-and-batch-fn stream))

     (catch 'next-batch
       (pgloader.queue:map-pop-queue dataq
                                     process-row-fn)))

    ;; in case of data-exception,
    ;; split the batch and try again
    (handler-case
        (cl-postgres:close-db-writer stream)
      ((or CL-POSTGRES-ERROR:UNIQUE-VIOLATION
           CL-POSTGRES-ERROR:DATA-EXCEPTION) (e)

    (retry-batch dbname table-name
             (nreverse *batch*) *batch-size*)))))
```

We can see that we take benefits from using *dynamic bindings* and reset them in each turn, avoiding to have to pass them explictly to our helper functions.

The control flow is also non trivial:

- the pgloader.queue:map-pop-queue function shown above is the main control point, responsible for popping elements out of the queue, and knows nothing about batch and their boundaries,
- the helper function make-copy-and-batch-fn is used to produce a new function at each batch iteration, its job is to prepare then send the data over the PostgreSQL stream that it's been given, keeping a copy of the data in case we need to reprocess it later as erroneous; it will also *throw* control away so that the current *batch* can be closed,
- the main loop prepares a batch and an helper function, then arrange for the map-pop-queue to have at it; it only gets the control back when the control is transfered explicitely (that's the *catch* construct) or when the queue is empty.

The main loop also handles COMMIT, and as that's when we know about any errors in the data, it handles *unique violation* and *data exception* errors by *retrying* the *batch* from the copy we've been maintaining.

This code is responsible for controling the whole data loading in PostgreSQL. It's not only much shorter than the Python version, and is also much faster.

5.5 Handling errors in data

The retry-batch function itself is pretty well defined too, it's responsible only for retrying the batch in smaller chunks of data. The implementation of it is not doing any data copying, it's moving in the batch and sending over a limited number of rows at a time.

5.6 New sources of data

The first implementation lying around was all about importing data from a flat file, but equiped with a simple *streaming protocol* that only needs a map-rows function, it's quite easy to implement a *MySQL* reader.

```lisp
;;;
;;; Map a function to each row extracted from MySQL
;;;
(defun map-rows (dbname table-name
         &key
           process-row-fn
           (host *myconn-host*)
           (user *myconn-user*)
           (pass *myconn-pass*))
  "Extract MySQL data and call PROCESS-ROW-FN
   function with a single argument (a list of
   column values) for each row."

  (cl-mysql:connect :host host
                    :user user
                    :password pass)

  (unwind-protect
     (progn
       ;; Ensure we're talking utf-8
       ;; and connect to DBNAME in MySQL

       (cl-mysql:query "SET NAMES 'utf8'")
       (cl-mysql:query
          "SET character_set_results = utf8;")
       (cl-mysql:use dbname)
```

```

        (let* ((sql
                (format nil
                        "SELECT * FROM ~a;" table-name))
               (q   (cl-mysql:query sql :store nil))
               (rs  (cl-mysql:next-result-set q)))
          (declare (ignore rs))

          ;; Now fetch MySQL rows directly
          ;; in the stream
          (loop
             for row =
                 (cl-mysql:next-row
                     q :type-map (make-hash-table))
             while row
             counting row into count
             do (funcall process-row-fn row)
             finally (return count))))

     ;; free resources
     (cl-mysql:disconnect))

;;; Export MySQL data to our lparallel data queue.
;;; All the work is done in other basic layers,
;;; simple enough function.
(defun copy-to-queue (dbname table-name dataq)
  "Copy data from MySQL table DBNAME.TABLE-NAME into
   queue DATAQ"

(let ((read
        (pgloader.queue:map-push-queue
           dataq #'map-rows dbname table-name)))
  (pgstate-incf *state* table-name :read read)))
```

The PostgreSQL specific parts (handling the `COPY` protocol, the batch building and retrying) are not affected at all by the adding of new readers, yet there's not much infrastructure written to get there.

5.7 Migration tool

To get from the ability to load data from either flat files or a live database to an *online migration tool*, `pgloader` only missed the ability to extract the database schema (list of tables, attributes with their names and data types, indexes, sequences) from the source and cast that to a PostgreSQL compatible definition, then apply it.

That feature set has been added for *MySQL*, the API might not be stable until we add some more source variants of course. Then remain the problem of how to expose that feature in the `pgloader` command language.

5.8 Controlling pgloader

The command language still has to be designed, the idea being to have a `COPY` like command syntax that you can use. A parser will be responsible of transforming that command into a proper Lisp program, then `compile` it dynamically and finally run the compiled code.

The current UI design is to either run `pgloader` with a bunch of commands to be found in a file, or to run a `pgloader` *service* to which you could send-in new commands, either from a command-line tool or from a web interface.

5.9 Extra: Signaling conditions and PostgreSQL notifications

In Common Lisp it's possible to implement *exception* by using the *condition* system. When confronted to some unusual situation, it's possible to *signal* it to the main program, which can decide to handle the situation by itself or use a *restart* that has been provided by the lower level code.

Apart from allowing a very nice error control flow while preserving the ability to write the error handling code at the right layer in the code, this *condition system* also allows to process asynchronous messages.

In PostgreSQL, the `LISTEN` and `NOTIFY` commands implement asynchronous messaging for your application. It's no surprise then that the PostgreSQL driver for Common Lisp, `postmodern` is using *conditions* to make *notifications* available to your code:

```
(defun reset-all-sequences (dbname)
  "Reset all sequences to the max value of the column
   they are attached to."

  (let ((connection
     (apply #'cl-postgres:open-database
        (remove :port (get-connection-spec dbname)))))

    (cl-postgres:exec-query connection "listen seqs")

    (prog1
    (handler-case
        (cl-postgres:exec-query connection "
DO $$
DECLARE
  n integer := 0;
  r record;
BEGIN
  FOR r in
    SELECT 'select '
            || trim(trailing ')'
               from replace(pg_get_expr(d.adbin, d.adrelid),
                             'nextval', 'setval'))
            || ', (select max(' || a.attname || ') from only '
            || nspname || '.' || relname || '));' as sql
      FROM pg_class c
           JOIN pg_namespace n on n.oid = c.relnamespace
           JOIN pg_attribute a on a.attrelid = c.oid
           JOIN pg_attrdef d on d.adrelid = a.attrelid
                               and d.adnum = a.attnum
                               and a.atthasdef
     WHERE relkind = 'r' and a.attnum > 0
           and pg_get_expr(d.adbin, d.adrelid) ~ '^nextval'
  LOOP
    n := n + 1;
    EXECUTE r.sql;
  END LOOP;

  PERFORM pg_notify('seqs', n::text);
END;
$$; ")
      ;; now get the notification signal
      (cl-postgres:postgresql-notification (c)
        (parse-integer
          (cl-postgres:postgresql-notification-payload c))))
    (cl-postgres:close-database connection))))
```

The usual way to run that code would have been to issue the main catalog query twice here, the first time to know how many *sequences* we are going to reset and the second time to actually reset them. Using the advanced features of both PostgreSQL and Common Lisp, it's easy to run only the single query that is needed.

6. CONCLUSION

The rewrite in Common Lisp did allow to use advanced features not to be found in Python: concurrency abilities thanks to *lparallel* and its queueing support, the control flow mechanisms (`catch` and `throw`, `conditions`, `loop`, `map`) and its performances (in some testing, between 5 and 120 times better than Python: the code is actually compiled).

It's quite impressive how dynamic Common Lisp really is with its advanced *REPL* (where you can migrate objects when changing their classes definitions), interactive debugging abilities (you do not have to decide to run the debugger before hand, any unexpected error will bring you there and you can directly work on fixing the problem), and introspection abilities.

Of course a part of why the new implementation is so much better than the previous one is just because we kept the author, so all the pitfalls and knowledge about the previous implementation were not lost. Still, the tooling is now much better.