Psycopg: A New DA between Python and PostgreSQL

Federico Di Gregorio

Dec 24, 2001

1 A little bit of history

Almost all projects have their roots in other, older projects. Only very few are completely original and psycopg makes no difference. It predates, if not code, at least ideas and principles from its ancestors and it would not be the solid program it is without them. The fact that its ancestors were (and are) free software made the whole process much easier, but we all know that free is better, right? Lets go on.

It was more than two years ago. We were developing some big web sites using Zope (very few people used Zope at that time) and Oracle^{TM}. The database adapter (DCOracle) had some problems but was just great. Then we moved to PostgreSQL and the database adapter (PygreSQL) was just ... er ... terrible. No support for the standard Python DBAPI, NULL values returned as empty strings, Zope hanging on database operations. Terrible. At the time, PygreSQL development seemed pretty stalled, so we decided to write a new driver from scratch. And no, it was not psycopg. It was PoPy.

PoPy was written mainly by Thierry Michel and Eric Bianchi (two French students doing a stage at MIX-AD LIVE), with some minor help from me (packaging, some very little bug fixing) and Michele Comitini. PoPy supported the Python DBAPI-2.0 and had some advantages over PygreSQL but was far from perfect, having been written in a very short time. We started to talk about a complete rewrite of PoPy and how a well-designed database adapter should look like. Before writing even a single line of code, Thierry and Eric left and PoPy went with them. We continued to use it and sent some bug reports but we were still talking about something with a better design. Months passed until Michele decided it was time to write the driver we talked about ...

The first version of psycopg was a tarball named something like `pgpyDBA.tgz` sent to me by Michele after a couple of unsleepy nights. The code did almost nothing at that time and the name was terrible but I started playing with it with the idea of letting Michele do almost all the work and just help him with packaging and bug fixing. 15 days after i ported the build system to the GNU autotools and imported the package into our cvs. Another couple of weeks and I found myself adding the connection cache code and the type system. I was definitely sucked into psycopg development ...

2 Techniqalities ...

... or the technical qualities we wanted in psycopg. Even if the code started rough we had a clear mental picture of how we wanted psycopg to work. We wanted it to:

- support heavy multi-threading and release the Python interpreter while connecting or exchanging data with the DB;
- have an aggressive caching policy for the physical database connections (opening a new connection has quite an high overhead); and

- have a nice, non hard-coded way, of mapping PostgreSQL types to Python ones.

Technical digression: Python (for some strange reasons tied to cross-platform compatibility) maintains a global lock to the interpreter and does not allow a thread to run unless its time has come or the thread kindly releases the lock by calling the `Py_BEGIN_ALLOW_THREADS` macro. From then on the thread runs in parallel with the interpreter, until it wants to call one of the Python internals, and has to reacquire the lock by calling `Py_END_ALLOW_THREADS`. If it does not re-acquire the lock, a segfault is the minimum you can expect (nothing if compared to data corruption.)

After releasing 0.1 to the public (I don't remember if somebody ever downloaded it) we begun wrapping almost every network operation in `Py_*_ALLOW_THREADS` macros introducing a big speedup and lots of bugs (a clear example of the porto-effect, see end of this article.) Anyway, multithreading was there.

At the same time we introduced connection caching allowing cursors to re-use the connection previously open for another cursor and not closed at cursor disposal. Implementing it was quite easy but to understand how it works we need to grab the difference of physical network connections to the database from the connection objects create by calling the `connect()` function in the psycopg module. The basic idea was to let a connection object manage more than one network connection assigning them to the newly created cursors as required. When a cursor (created by invoking the `.cursor()` method on the connection object) is disposed, the network connection is not closed, just cleaned up and stored in a queue, ready for the next `.cursor()` call. This simple trick gives a speed boost to applications that create and destroy lots of cursors, like servers that spawn a new thread for every incoming connection.

Around 0.3 we introduced the type system, built on dictionaries of type-casting objects. The basic idea was to have a flexible system where the user can add at runtime mappings from PostgreSQL types to Python ones. (After all, PostgreSQL has a rich set of types and even user-defined types!) When the psycopg module is imported in Python, it just setup a default set of type-casting objects (for numbers, time stamps and intervals, dates, strings and binary data) but the user can define new ones or override the defaults. The fact that PostgreSQL returns everything as a string make really easy to write customized type-casting objects. Here is an example of what the integer caster does:
`def IntTypeCaster(data): return int(data)`
(Except the fact that the predefined type-casting objects are written in C, for increased speed.)

The 0.3 release was the first usable one, but the first real downloads begun at 0.4. At that point we begun to rewrite big chunks of code, both to fix bugs reported by the users and to implement required functionality. During the 0.5 cycle, for example, the entire threading code was redone to comply with the DBAPI-2.0 document, requiring cursors not to be 'isolated', i.e., any modification done to the database via a cursor should immediately be visible to all other cursors derived from the same connection. Confused? We surely were ...

We wanted a completely stable one-dot-oh release, so, after six months, we stopped adding new features and started the 0.99 series. Michele rewrote three times the binary data management code and we fixed the last few bugs in a couple of weeks. We felt confident. We released 1.0pre1.

3 Time to code

1.0pre1 seems pretty stable, so it is time to abandon psycopg's 1.history and take a look at Python DBAPI programming. The DBAPI-2.0 document defines a minimal set of functionality that all the Python database adapter should provide. The DBAPI does not make porting an application from a

database to another automatic. Different databases support different dialects of SQL and implement different extensions, after all. But it helps, because you don't need to re-learn everything from the start every time you switch database. Assuming some SQL and Python knowledge, extracting data from a database is as easy as (lets suppose we have a database with name, surname, telephone number and a PNG image saved as binary data):

```
import psycopg

o = psycopg.connect('dbname=mydb user=fog')
c = o.cursor()
c.execute('SELECT * FROM addressbook WHERE\
 name = %s', ['Michele'])
data = c.fetchone()
print "Saving image of %s %s" % (data[0], \
data[1])
open(data[0]+".png", 'w').write(data[3])
```

The first two lines simply create a connection to the database and then a cursor to execute a query. The third line executes a very simple query using a 'bound variable', whose value is evaluated at execution time. A nice plus is that bound variables are automatically quoted, e.g., psycopg will automatically quote the string `"rl'ye"` into `"ry"lie"` before passing it to the database backend. The fourth line just fetch a row of data, used by the following two lines to print a message and save the PNG image into a file. Easy.

But, how does psycopg decide if and how to quote? Strings are rather easy; more complicated data types will need a little help from you, like in the following example that saves Michele's photo into the database (lets suppose we already have an opened cursor, c):

```
photo = open("Michele-new.png").read()
data = {'photo':psycopg.BINARY(photo),'nam\
e':'Michele'}
c.execute("""UPDATE addressbook SET photo \
= %(photo)s
             WHERE name = %(name)s""", dat\
a)
```

The psycopg module (and all other DBAPI compliant modules) have type singletons (BINARY, DATE, TIME, ROWID, etc.) that can be used to convert any kind of data from string to a representation better suited for the database in use. This example also show how to bind variables using a dictionary instead of a list.

As a final example lets see how the connection pooling of psycopg can help when designing a multi-threaded server with database access. To make it easy, we abstract the client-server protocol into a function `get_next_request()` that returns the next pending request to be served. A very minimalist server, without any error checking would then be:

```
import threading, psycopg

## setup a *single* connection for
## all threads, make it not serialized
o = psycopg.connect('dbname=mydb user=fog', \
serialize=0)

## main loop
i = 0 ; r = get_next_request()
while r:
    t = threading.Thread(None, manage_req, '\
Requst-'+str(i), o)
    t.start()
    i += 1 ; r = get_next_request()

## a function to manage requests
def manage_req(o):
    c = o.cursor()
    # do something with the cursor ...
    c.close()
```

Note how we created a single database connection and how we pass it to the new threads. When a thread finishes serving the request, it closes the cursor, but the physical connection associated with that cursor is not closed. It is put apart in a pool and reused when

the next thread calls `.cursor()`. This allows for a great speed gain and if you don't believe it, just try to remove the `serialize=0` argument to `.connect()`, disabling the connection cache.

Enough code. As I said at the start of this section psycopg 1.0pre1 seemed quite stable. Seemed. Let's return to our history ...

4 The -ility stuff

software is made of **Stability. Availability. Usability.** We had the last two but, unfortunately, not the first one. There is that strange effect: when you put a zero as the first digit of the version of some software, bug reports flow in at a steady rate. Then, you decide to name a release 1.0something and, abruptly, the bug reports increase in number and severity. After the 1.0pre1 release, the mailing list started to fill with messages from people experiencing grave bugs and even trying to port psycopg to other archs! Now I understand why some software skips the 1.0 stuff entirely an moves directly to 1.x or even 2.0. It is to avoid the dreaded one-dot-oh effect ...

Anyway, about two months later, with a lot of help from psycopg users, we fixed the last (reported) bug (a strange segfault due to a single line of code dating back to 0.1) and released 1.0, the current release. A patch release, fixing a little memory leak, will probably be released for Christmas and after then we are planning documentation (both about psycopg itself and DBAPI programming) and a new development branch including features as asynchronous queries and (absolutely the most requested) NOTIFYs.

In conclusion, I do not know if psycopg is really better than PoPy or PygreSQL. Better how? Better for what? It is too difficult to make comparisons. What I can say is that psycopg was well-designed from the start and gained a lot of supporters using it in real-world cases. And this the most important thing. Looking back I discover that I spent more time in 'user management' (reading and writing to the ML, investigating problems, helping newbies, etc.) than in coding.

There is a big lesson here. Never, never, never postpone an answer to a bug report to 'just code that little feature I dreamed tonight'. Most of the bugs where found after some users submitted a snipped of code happily crashing psycopg. Some beautiful and elegant code was written after an user said (publicly, on the mailing list) that 'there should be a better way to do it.' Users are the base on which write better code. Among them there are testers, contributors and maybe even the future maintainer of your project. Even the better software, written by a demigod hacker is doomed if nobody uses it.

```
c.execute("SELECT byebye FROM article_parts \
WHERE time = %s", ('late',))
```

Ah! Yes, the porto effect. To put it shortly, never drink a bottle of porto wine with friends and then go programming. You will surely introduce a bug so terrible it will still bite you one year after. This term was introduced by tersite after an apt-get upgrade that completely destroyed its X11 installation on a Debian box. His exact words were something like : "...the X11 maintainer should have drunk a lot of porto yesterday night, just before uploading the debs." (Hi, Branden.)

About Author Federico Di Gregorio was born in 1971-10-19, in Pinerolo, ITALY, He met the Free Software the first time on the mailing lists of ADE (Amiga Development Environment), a project to port the GNU software to the AmigaOS. Fisrt installed GNU/Linux on a laptop received as graduation (Physics) gift. Debian developer from the 1996 and founder of ASSOLI (FSF Europe affiliate assso-ciation.) Partner of MIXAD LIVE (now closing) for two years, now works as 'free software consultant and programmer'. He even managed to not starvate. He can be reached by email: `fog@initd.org`