HOW TO MIGRATE FROM POSTGRESQL TO HDF5 AND LIVE HAPPILY EVER AFTER

Michele Simionato @ GEM Foundation

legacy code warrior

The story of a successful software rewriting
WHERE I COME FROM

- Ph. D. in Theoretical Physics (yep, I understand gravitational waves)
- contributed to Python with article/docs, author of the decorator module
- worked a couple of years as consultant (Zope/Plone + other stuff)
- worked 7 years in a Finance firm (mostly database and web programming)
- arrived at GEM in October 2012
- in charge of the earthquake simulation engine from September 2014
• I spent 15 years doing Physics and 14 years programming
• I am still more interested in solving the scientific problems than the technological problems
• lots of experience with programming in the large
• know all about code maintenance, dependencies management, product documentation, automatic tests, continuous integration, code reviews, ...
• SQL lover and generally old school boy
• still using Emacs in the 21th century
• *I don't throw away old code just because it is old*
WHAT I AM GOING TO TALK ABOUT

- "big" numerical simulations for earthquakes
- lots of floating point outputs with geospatial data
- why storing them in Postgres/PostGIS did not work
- what to do when you have a huge architectural problem
- technical and political issues faced
- how the migration PostGIS -> HDF5 was done
- lessons learned
NUMBER CHRUNCHING WITH A RELATIONAL DATABASE??

- yep, seriously
- after a few weeks on the new job I had already realized that the architecture was completely wrong
- everything was structured like a Django application without being a web application
- there was an insane mix of Django objects and arrays
- large numpy arrays were stored in the database as pickled objects
there were absurd things like doing the aggregation on the database with locks

there were hundreds of workers writing concurrently on the same table

the database logic was *hopelessly* coupled with the scientific logic

strangely enough, the architecture was totally wrong but the code base pretty good

*Then rewrite everything?*
THE FIRST LAW OF SOFTWARE REWRITING

Rewriting a project takes *always* longer than writing it in the first place (as we seasoned developers know)
I did not want to embark myself in such an adventure
but sometimes there is no choice :-(
Hazard Event Based Calculation for Germany, 1000 SES x 1 year

Cumulative seconds

- save gmfs
- compute ruptures
- compute gmfs
- save ruptures

engine-1.5
engine-2.0
WHAT FIVE ORDERS OF MAGNITUDE MEAN

- 1 day -> less than 1 second
- 1 year -> 5 minutes

I measured a speedup exceeding 200,000x

(+ memory gain of 1-2 orders of magnitude)
BLOCKED BY POLITICAL ISSUES

- the architecture of the application had been just rewritten (more than an year of effort, it was ported from Redis to PostgreSQL)
- a younger colleague of mine had already written a criticizing mail went in the wrong hands
- the official release of version 1.0 had to be ready in six months
- there was a team friction between Zurich and Pavia
- there was an artificial division between hazard code and risk code making it impossible to fix the risk code
DOING NOTHING FOR 8 MONTHS

• sometimes doing nothing is the best choice
• study the codebase, maintain the old code
• let the frustration grow
• *(a good thing, if limited in time)*
• getting new case studies and adding new tests
• improve where you can (monitoring, XML parsing, concurrency, ...)*
WAITING PAYS OFF

- the Zurich team evaporated
- we took change of both hazard and risk in Pavia
- we started removing old cruft (10,000+ lines of code)
- we decided on a conservative strategy: keep Postgres, rewrite the relational schema and the slow queries
13 MORE MONTHS FIGHTING POSTGRESQL

- implemented a migration mechanism
- changed most of the tables
- changed the critical queries several times
- improvements by several orders of magnitude
- still, it was not enough, as realized in the summer of 2014
- but we kept the users and scientists happy
THE DECISION TO DROP POSTGRESQL

- started the oq-lite project with the *excuse* of the Windows porting, September 2014
- ported the simplest calculators to the new architecture
- removed gradually the geospatial queries
- kept in parallel both versions, with the same functional tests
- removed a lot of annoying unittests
- built expertise with HDF5
OTHER ARCHITECTURAL CHANGES

- all the concurrency is managed by a pure map-reduce
- made the concurrency layer independent from the low level parallelization technology
- changed from everybody read/write on the database to only the controller node can read/write
- all scientific data are now in the datastore (one .hdf5 file per calculation)
- all the metadata (i.e. start/stop time of the calculation, description, logs, performance information, output information) are in SQLite
- the workers do not write anything on the filesystem and they do not communicate at all with the master, except via the map-reduce (shared nothing)
- totally decoupled the database from the calculation logic
- added a DbServer in Python serializing the access to the database (needed for the WebUI)
- removed the ORM layer (thanks to Martin Blais dbapiext)
EVEN MORE CHANGES

- removed completely the need for XMLSchema, doing the validation entirely at the Python level
- implemented a generic serializer Python <-> XML instead of dozens of different serializers, one per class
- added CSV exports with the final goals of removing XML exports
- implemented a serialization protocol Python <-> hdf5 with methods __toh5__ and __fromh5__
- supported both single user mode and multiple user mode
- ported the engine to Windows and Mac OS X
- modernized the code to Python 3
ALL WENT SURPRISINGLY WELL!?
THINGS THAT I ALREADY KNEW, REINFORCED

- monitoring the running system for speed and memory allocation is *essential*
- if testing is difficult, the architecture is bad
- unittests are bad, functional tests are good
- if you want performance, replace Python objects and dictionaries with numpy.arrays
- no ORM please
MY TECHNICAL ADVICE

- follow a principle of simplicity/cleanness: 95% of the speedups and memory saving came for free after *removing* code
- invest your time in solving the real problem, not in complicating your technological stack (so I did not spend time on numbas, GPUs, Intel compiler, etc etc)
- always challenge the underlying assumptions
- take the most difficult problem *that you can solve* and solve it first
- most of all, be patient
POLITICAL ADVICE

- don't be confrontational with your boss
- it is his job to be conservative
- take the slow way and make sure that at every little step you have a measureable improvement to show off
- performance is a good excuse for change
- you can raise your voice once or two in four years
- it takes time to build trust
THINGS I DISCOVERED ALONG

- using a database + ORM requires a *LOT* more memory than you think
- it is sometimes best to use all of the available memory
- sometimes it is better to run out of memory early
- the data transfer is really important
- a story about parsing XML: lxml -> ElementTree -> expat
- the migration to Python 3 had several surprises
LESSONS ABOUT TECHNOLOGIES

- concurrent.futures is just fine
- Travis is good
- wheels are great
- h5py is ultra-fast but can bite you
REGRETS

- nearly two years "lost" (from October 2012 to August 2014)
- I was too conservative and I should have cut more stuff/tests
- I should have investigated better what features were really important

*(but removing 50,000+ lines of code feels really great!)*
THE END: SECOND LAW OF SOFTWARE REWRITING

Software rewriting takes *always* longer than you think

But sometimes it is worth it :-)

[https://github.com/gem/oq-engine](https://github.com/gem/oq-engine)