Grokking Python Event Loops and Concurrency with Apache Cassandra

by Andrew Montalenti (http://blog.parsely.com/post/author/andrew-montalenti/)

PARSE.LY TECH (HTTP://BLOG.PARSELY.COM/POST/CATEGORY/PARSE-LY-TECH/)

April 22, 2015

Python offers a number of different concurrency models, including multi-threading, process pools, and cooperative multitasking around an async event loop.

These various concurrency models were demonstrated in a fun live coding session by David Beazley at PyCon US 2015, entitled “Concurrency From the Ground Up” (https://www.youtube.com/watch?v=MCs5OvhV9S4). This post will explore why the Python driver for Apache Cassandra was designed around async event loops, and how that lets you achieve a high number of concurrent writes with a single Python process and a single CPU core.

About Cassandra

Apache Cassandra (http://cassandra.apache.org/) is a distributed database built atop a powerful Dynamo-like data model; it is a mixture of key-value and column-oriented data storage. It is written in Java (boo!), but Python support has always been good (yay!).

Traditionally communication with Cassandra happened via a Thrift interface. The module used for working with Cassandra since the 0.x and 1.x versions was pycassa (https://github.com/pycassa/pycassa).

But, in 2.x, Cassandra introduced a new query language, CQL. With it came a new native, binary protocol for communication. And thus, with that, came a new Python driver (https://github.com/datastax/python-driver).

However, the Python driver is new not just in that it supports CQL, but also in its general design. Due to Cassandra's ability to utilize many CPU cores through a highly threaded, staged event-driven architecture (SEDA) (http://en.wikipedia.org/wiki/Staged_event-driven_architecture), it was important for operations sent to a Cassandra cluster from the Python driver to support various concurrency models.

Since all the Python driver needs to do to take advantage of Cassandra's processing power is issue driver communication requests, a program written against Cassandra will likely be I/O-bound. Thus, the concurrency model chosen was to take advantage of an asynchronous event loop.

Pluggable Event Loops

The Python driver actually bundles asynchronous I/O reactors using pretty much every available Python option. These are:

- `asyncore`, Python's stdlib implementation of asynchronous sockets
- `libev`, a C library that implements async I/O
- `twisted`, the widely-renowned async networking library that first brought this style to the Python community
- `gevent`, a clever Python library that monkey patches the socket module to achieve async I/O

The design of the driver allows you to choose which reactor you want to use. The benchmark suite included with the driver also allows you to try out various concurrency models with your local or production Cassandra cluster, as described in their performance notes (http://datastax.github.io/python-driver/performance.html).

The other interesting thing is that by writing some code that works with Cassandra, we are able to see the balancing act between I/O-bound and CPU-bound work. When the processing becomes CPU-bound, we will not be able to feed enough events into our async I/O reactor, and thus throughput will decrease. We'll be able to get an additional speedup by switching from CPython to pypy (http://pypy.org/), but this will also only go so far.

You might wonder — why does Python need so many event loop implementations? You are having a good thought. Guido van Rossum, the language's creator, also had the same thought, and that's why he worked on the asyncio (https://docs.python.org/3/library/asyncio.html) standard library module, which is available in Python 3.4 and up. It offers a standard library API for event loops which these projects may one day adopt. (The popular Tornado (http://www.tornadoweb.org/en/stable/) web server already has a bridge (http://tornado.readthedocs.org/en/latest/asyncio.html).)

Benchmarking Cassandra Locally

You can create a virtualenv for the Python driver thusly:

```
mkvirtualenv cassandra
pip install cassandra-driver==2.1.4
```

If necessary, you can also install ipython. You can then test that the driver can be imported:

```
ipython
>>> from cassandra.cluster import Cluster
>>> Cluster()
<cassandra.cluster.Cluster at 0x1f70cd0>
>>> cluster = _
>>> cluster.connect()
...
NoHostAvailable: ('Unable to connect to any servers', {'127.0.0.1': error(111, 'Conne
ction refused')})
```

Of course, no host will be available because you don't have Cassandra running locally.

Getting Cassandra to run locally is probably beyond the scope of this tutorial. There are lots of options available, ranging from OS packages to Chef to Vagrant to Docker and everything in between. If you have a healthy local install of Java, it's also pretty easy to get a single-node cluster running simply by downloading the tarball and running `cassandra`.

Once `cluster.connect()` works, the benchmarking suite likely will, as well.

For this, we'll need to clone the Python driver locally:

```
git clone git@github.com:datastax/python-driver.git cassandra-driver
cd cassandra-driver
```

Then check out the tag matching our version:

```
git checkout 2.1.4
```

Finally, install the testing requirements into your virtualenv:

```
pip install -r test-requirements.txt
```

You will find the benchmarking tools in the `benchmarks/` directory. Go in there:

```
cd benchmarks
python sync.py --help

Usage: sync.py [options]

Options:
-h, --help            show help msg
-H HOSTS, --hosts=HOSTS
-t THREADS, --threads=THREADS
-n NUM_OPS, --num-ops=NUM_OPS
--asyncore-only       only benchmark asyncore
--libev-only          only benchmark libev
--twisted-only        only benchmark twisted
-m, --metrics         show metrics
-p, --profile         profile the run
```

I've elided some of the output but left the relevant bits.

The cool thing here is that you can pass your Cassandra hosts at the command-line (it will use localhost by default), and then tweak settings like number of operations (`-n`), number of threads (`-t`), and which reactor to use (`asyncore` , `twisted` , or `libev`).

The `-m` option will actually measure I/O activity and request latency using the `scales` library. The `-p` option will use Python's `cProfile` module and save the profile information in a file in the current directory, which can be analyzed using `pstats.Stats()` inside an IPython shell.

The `sync.py` is the "naive, synchronous execution" benchmark. A few others are available and offer the exact same CLI:

```
callback_full_pipeline
future_batches
future_full_pipeline
future_full_throttle
sync
```

All of these are described in the performance notes (http://datastax.github.io/python-driver/performance.html) mentioned above.

Studying the Implementations

The synchronous implementation, `sync`, shows the most naive way of sending writes to Cassandra. In a tight for loop, it just sends `INSERT` statements to Cassandra, one at a time.

This can achieve several hundred writes per second on a single core with a single thread. By tweaking the `-t` parameter, you can increase the thread pool, which will achieve greater throughput. But you will ultimately hit the limits of multi-threading for this I/O bound use case.

Three other options make use of `execute_async`, which will, in turn, make use of an async event loop that is managed for you. These are `future_batches`, `future_full_pipeline`, and `future_full_throttle`. They are named with “future” for their use of `ResponseFuture` objects, a kind of future/promise (http://en.wikipedia.org/wiki/Futures_and_promises). It is what the `execute_async` method returns.

These three techniques are similar. A number of async requests are scheduled in parallel, and the event loop can service them via I/O operations, like talking to the Cassandra cluster over the network. In the “batches” case, the async requests are scheduled in fixed-size batches; each batch is scheduled and then the next batch starts when a whole batch finishes. The “full_pipeline” case instead uses a fixed-size (throttled) queue where as requests complete, new requests are scheduled. The “full_throttle” case is the most dangerous: no batch or queue is used, and instead, all requests in the benchmark are fired and scheduled at once. These all achieve similar throughput levels (much higher than the “sync” case, often by a factor of 5-10X), but it is unwise to use “full_throttle” since you can chew up a lot of memory with pending future/promise objects.

The somewhat surprising result from these benchmarks is just how good the most complex scheduling pattern is: `callback_full_pipeline`. In this pattern, rather than individual async `INSERT` statements being sent off in batches or in a pipeline, a series of “callback chains” are established.

A first batch of writes are scheduled, but their success handlers are wired up such that when they complete, a new async `INSERT` is immediately scheduled. This achieves throughput that is **often 20X better** than the “sync” case.

On a single core and with the right data, this pattern will often saturate network and achieve pretty much the highest level of single-node throughput you can expect with Cassandra. This pattern is so good that the driver provides a helper function, `execute_concurrent` (http://datastax.github.io/python-driver/api/cassandra/concurrent.html#cassandra.concurrent.execute_concurrent), which encapsulates a reference implementation.

Upgrading to PyPy

It’s pretty shocking to realize that your write throughput can range from 300 writes per second to 7,500 writes per second on the same hardware and with the same driver/database. It makes you start to question your approach to performance tuning!

I’ve learned that simply studying — and changing — the concurrency model used for scheduling your tasks will have a big impact.

But wait, there’s more.

The last interesting benchmark here is using `pypy`. You see, if you’re not saturating network, the only thing holding you back in Python is the single-core performance of the CPython interpreter. The CPU is spinning on things like decoding the wire protocol for Cassandra and translating data types from CQL to Python native data types.

The pypy interpreter is notably faster. Could pypy execute Cassandra writes faster, too?

To do this requires the setup of another virtualenv and to build pypy locally. This is fun and harks back to the days of compiling your own Linux kernel for your local Debian or Gentoo distribution. You should try it!

To build `pypy` locally, you need to:

1. install pypy (http://pypy.readthedocs.org/en/latest/install.html) (pypy is used to build pypy)

2. install its build-time dependencies (http://pypy.readthedocs.org/en/latest/build.html#install-build-time-dependencies)
3. download the pypy 2.5.1 tarball (https://bitbucket.org/pypy/pypy/downloads/pypy-2.5.1-src.tar.bz2)
4. extract it
5. run `make` inside its directory
6. wait about 1 hour for it to compile

Once that's done (and you've had your coffee break!) you will have an executable called `pypy-c` inside that directory. You can try running it to get a pypy prompt:

```
./pypy-c

Python 2.7.9 (...)
[PyPy 2.5.1 ...] on linux2
>>>>
```

Exit out of that and you can now use that Python interpreter as the basis for a virtualenv. Simply do:

```
mkvirtualenv pypy-cassandra -p pypy-c
```

And the virtualenv `pypy-cassandra` will have its `python` interpreter supplied by `pypy`. Test this out:

```
ls -l `which python`
~/.virtualenvs/pypy-cassandra/bin/python -> pypy-c
```

Then, run it:

```
python

Python 2.7.9 (...)
[PyPy 2.5.1 ...] on linux2
>>>>
```

You’ll need to follow the same setup steps for this virtualenv as for the above one to make use of the Cassandra benchmark suite. You should try it for yourself — but, in my case, I saw some pretty dramatic speedups on even the fastest benchmarks achieved with CPython. I managed to get some benchmarks to break **20K writes per second** on a single core, and without using threads!

The Rise of Async

With the rise of `asyncio`, achieving concurrency with async event loops is a hot topic in the Python community.

At Parse.ly, we achieve concurrency for CPU-bound work using Apache Storm and our home-grown (and open source) streamparse (https://github.com/Parsely/streamparse) module, which we have also presented at PyCon this year (https://www.youtube.com/watch?v=ja4Qj9-l6WQ&t=1m30s). However, I/O-bound work — such as talking to databases and serving HTTP requests — often benefits from some async approaches. The Python community is getting more and more comfortable with these async approaches. The new Python driver for Apache Cassandra provides an excellent, real-world implementation of why async matters — and it has the benchmarks to prove it.

Meanwhile, the Python developers keep going deeper into the async rabbit hole. First, there was the `yield from` (https://www.python.org/dev/peps/pep-0380/) sugar and the “rebooted” asyncio implementation (https://www.python.org/dev/peps/pep-3156/). And now, there is a proposal for `async` and `await` keywords in PEP 492 (https://www.python.org/dev/peps/pep-0492/). It’s pretty clear that async programming and event loops are here to stay!

Are you a Pythonista who is interested in these kinds of things? We're hiring — check out our job posting for software engineers (http://www.parsely.com/jobs/#software_engineer) for more information, and be sure to mention this post!