Performance Profiling and Monitoring of Data Central Ingestion System

Sam Huynh (AAO, Macquarie University)

State of Surveys

- Surveys wanting to release a larger data set with a variety of files
  - From the expected:
    - Spectra
    - Catalogues
    - IFS
  - Becoming more common
    - Pipeline results and logs
    - Additional plots and analysis files

Background into Data Central's Ingestion Process

Technologies used by Data Central

- Django
- Celery
- PostgreSQL
- Docker

For each spectra file, catalogue, IFS file and imaging file, we create a record to allow users to find, access and reuse data from a defined interface.

These records are created by Celery tasks, stored in PostgreSQL and served out to users by a Django website.

Within the Celery tasks, FITS keywords are checked, catalogue tables parsed, WCS checked and so on...

In a recent addition, miscellaneous files are treated as 'blobs' so there is still a record created. Only a simple check of file size is done.

The problem?

For one data release with a large spectra data set and large set of miscellaneous files:

- Ingesting the spectra data set would take roughly 2 days
- Ingesting the miscellaneous files would take a week (if not more!)

Why would ingesting files take such a long time if the only check done is a file size check?

Diagnosing the problem

We are creating a lot of records, we should check if the database is blocked somewhere...

```
-- check for blocked queries in PostgreSQL
select pid,
       usename,
       pg_blocking_pids(pid) as blocked_by,
       query as blocked_query
from pg_stat_activity
where cardinality(pg_blocking_pids(pid)) > 0;
```

```
-- view running queries in PostgreSQL
SELECT pid, age(clock_timestamp(), query_start),
usename, query
FROM pg_stat_activity
WHERE query != '<IDLE>' AND query NOT ILIKE
'%pg_stat_activity%'
ORDER BY query_start desc;
```

Equivalent queries exist for other databases systems

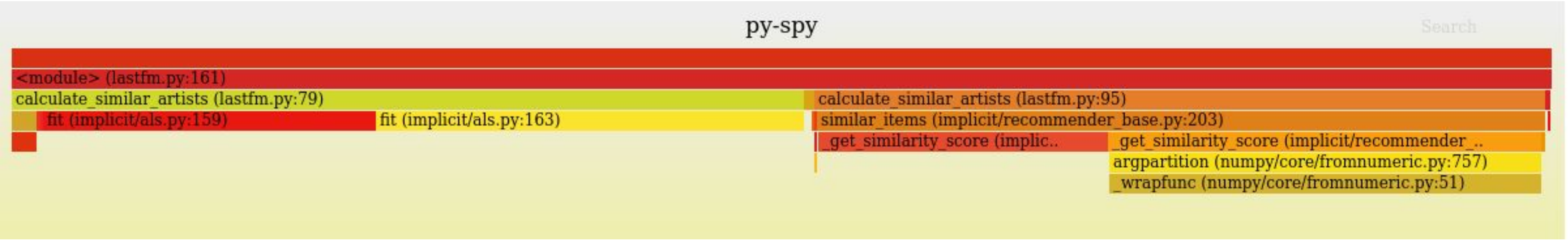
Findings: No blocked queries but queries to find IDs being repeated over and over!
But where are these queries made?

py-spy with my little eye

https://github.com/benfred/py-spy

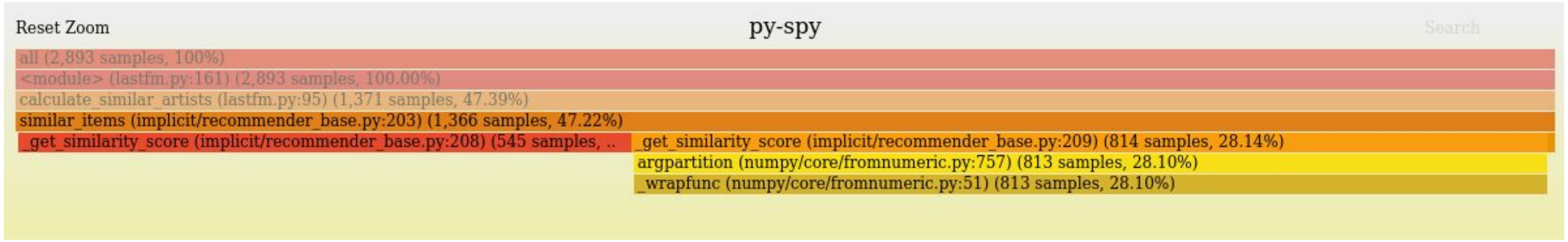
py-spy creates flame graphs of your python program visualising which lines of code take the most CPU time. This is done by sampling your python program to get the stack trace.

```
py-spy record -- python program.py
```


py-spy
Search
<module> (lastfm.py:161)
calculate_similar_artists (lastfm.py:79)
calculate_similar_artists (lastfm.py:95)
fit (implicit/als.py:159)
fit (implicit/als.py:163)
similar_items (implicit/recommender_base.py:203)
_get_similarity_score (implic..
_get_similarity_score (implicit/recommender_..
argpartition (numpy/core/fromnumeric.py:757)
_wrapfunc (numpy/core/fromnumeric.py:51)

Flame graphs: https://github.com/brendangregg/FlameGraph

The graphs are interactive!


Reset Zoom

py-spy

Search

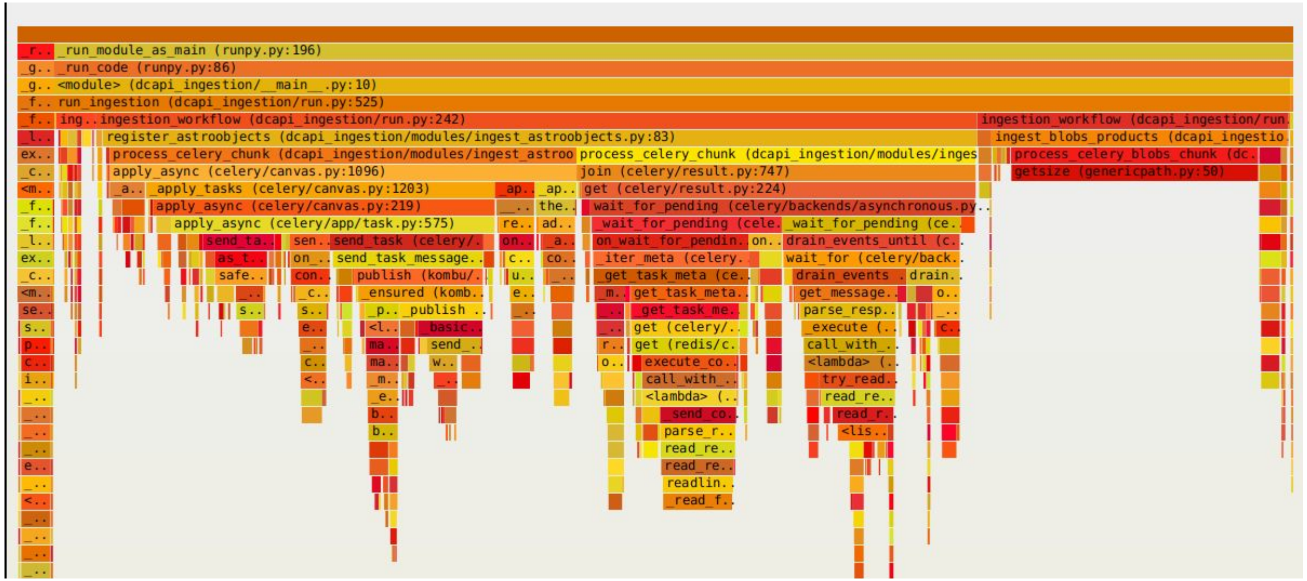
```
all (2,893 samples, 100%)
<module> (lastfm.py:161) (2,893 samples, 100.00%)
calculate_similar_artists (lastfm.py:95) (1,371 samples, 47.39%)
similar_items (implicit/recommender_base.py:203) (1,366 samples, 47.22%)
_get_similarity_score (implicit/recommender_base.py:208) (545 samples, ..   _get_similarity_score (implicit/recommender_base.py:209) (814 samples, 28.14%)
                                                                            argpartition (numpy/core/fromnumeric.py:757) (813 samples, 28.10%)
                                                                            _wrapfunc (numpy/core/fromnumeric.py:51) (813 samples, 28.10%)
```

Hovering over a rectangle displays the times a function was in the python stack trace sample

Clicking on a rectangle zooms in so you can see more information. E.g. percentage of samples, function calls

Flame graph after optimising

After removing redundant code, ingesting all miscellaneous files takes roughly 8 hours instead of a week!



Some things to know about py-spy

- Run against already running code (except in Docker and k8s environments), no need to modify your code to profile
- In Docker and k8s environments, SYS_PTRACE capability is required to run which means redeployment is necessary. See https://github.com/benfred/py-spy#how-do-i-run-py-spy-in-docker for more info
- py-spy will profile CPU. For memory profiling, use the standard tracemalloc module or other memory profiling modules on PyPI
  - Aside: even running a memory profiler for a short time can give you hints into memory leaks

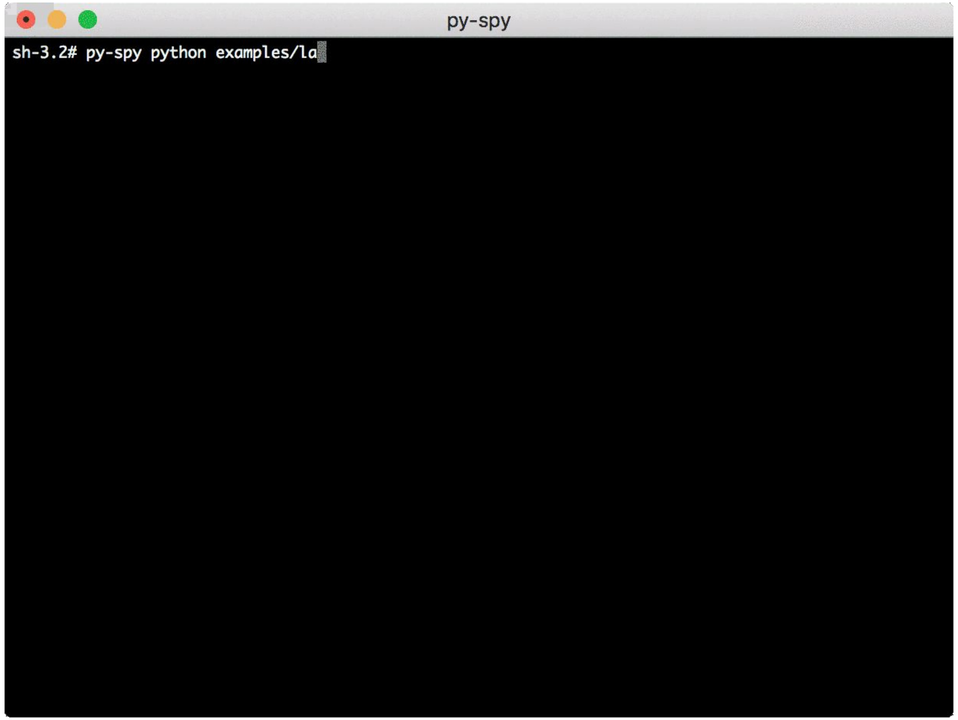
View program execution while the program is running

`py-spy top –pid pid` gives a live updating view of your python program

`py-spy dump –pid pid` outputs the current program stack

Root privileges are usually required for attaching to existing processes

`sudo py-spy top/dump ...`



```
sh-3.2# py-spy python examples/la
```

Further work

There are definitely some more optimisations to be made in our code.

With the increasing size of survey data, it is worth spending time to ensure we have measurements of our code.

We can use this to:

- Verify an optimisation results in performance increase
- Compare changes due to input size scale
- Ensure a change does not result in performance regression

Key points

- Don’t optimise in the dark
    - Profiling gives you insights into which chunks of code are causing the program to slow down
    - Some execution time might be spent for general overhead which you can’t control
    - There might be other reasons why a program is slow e.g. limited hardware resources
- Profiling can help you spot problems before they become problems
    - Don’t optimise prematurely
- Profiling is not the hammer for every nail
    - Logging and metrics can help you find other problems

Thanks!