Global Dataset Discovery in the Virtual Observatory

Markus Demleitner

PUNCHLunch 2025-02-17

Talk Outline

- The problem from 30'000 ft
- pyVO and a simple API
- The role of the VO Registry
- Dupes, inadequate services, logging: Problems close up
- API considerations
- Demo?
- Only the first steps

Reminder: The Virtual Observatory

The VO is

- A set of standards you can read at http://ivoa.net/documents.
- $\sim$ 50 data centres implementing these standards and offering services to query data collections
- a bit of quasi-central infrastructure (in particular the Registry)
- an ecosystem of client software maintained by all kinds of parties

In sum: There is no centrally controlled platform, and the development model is characterised by compromises, consensus, and goodwill. Just like the Internet when it was still fun.

Glossary

Here are a few VO concepts you need at least passing familiarity with to understand what's going on here:

- The Registry – Metadata (Dublin Core, service, table, phyiscal coverage metadata); that's $\sim 30'000$ records right now
- RegTAP – a schema of ~ 20 relational tables making the Registry queriable
- TAP – the table access protocol makes remote tables SQL-queriable
- Obscore – a TAP-publishable schema for observational products
- SIAP – Simple Image Access Protocol
- pyVO – a python library wrapping up many VO protocols so they're easy to use from within Python.

The Desired API

In this distributed system, we want people to be able to globally search for datasets (for now: images) constrained only by space, time and spectrum ("blind discovery"):

```
images, log = discover.images_globally(
        space=(132, 14, 0.1),
        time=time.Time(58794.9, format="mjd"),
        spectrum=600*u.eV,
        inclusive=False)
```

Sounds easy? Well, let's try.

A Two-Step Process

In the VO, global dataset discovery has two steps:

1. Locate services that *could* have relevant datasets (from among the 30'000 in the Registry.
2. Send appropriate queries to each service discovered.

Extra trouble: In the VO, images can be published through any combination of:

- SIAPv1 – the old "simple" (i.e., atomic HTTP parameters) protocol for searching images
- SIAPv2 – a newer "simple" protocol for searching images
- Obscore – standard metadata tables queried via TAP

Challenge: Too Many Services

How many candidate services are there for images? Try:

```
import pyvo
print("#sia", len(pyvo.registry.search(servicetype="sia")))
print("#sia2", len(pyvo.registry.search(servicetype="sia2")))
print("#obscore", len(pyvo.registry.search(datamodel="obscore")))
```

This yields:

sia	320
sia2	141
obscore	43

(Note that obscore services generally publish multiple data collections, which is why there are fewer of them)

The Registry to the Rescue

An increasing number of VO resources define their coverage in space, time, and spectrum in the Registry. Using this, we can skip services that do not cover the user's region of interest.

However:

```
SELECT COUNT(*) FROM
rr.stc_spatial
NATURAL JOIN rr.capability
WHERE standard_id LIKE 'ivo://ivoa.net/std/sia%'
```

At the moment, only 84 SIAP1/2 services declare their spatial coverage (29 for spectral, 43 for temporal).

The Obscore Problem

Obscore services are currently discovered as TAP services with the obscore data model.

Hence, the coverage – if given at all – is generally not useful: there can be a lot else in the TAP service.

This is un-recoverable. We need to fix Obscore registration.

Beyond "Obscore tables need to get resource records of their own", this is VO nerd stuff. If you are a VO nerd, see the TableReg note and comment: https://ivoa.net/documents/TableReg.

Dupes, Dupes, Dupes

Another big issue: Blindly querying everything will return many datasets multiple times.

First reason for that: Services having both SIAP1 and SIAP2 interfaces. How many are there?

```
SELECT COUNT(*) FROM
rr.capability AS a
JOIN rr.capability AS b
USING (ivoid)
WHERE
  a.standard_id='ivo://ivoa.net/std/sia'
  AND b.standard_id='ivo://ivoa.net/std/sia#query-2.0'
```

That's 30 at the moment; in the service selection, one can filter these out by preferring SIAP2. But...

Dupes from Obscore

At the GAVO data centre, all of its 20 SIA services are also reflected in its Obscore table (and a sitewide SIAP2 service, too). It would be a bad waste of resources to fire off the 20 extra requests.

Not to mention we would have lots of dupes.

To enable fast global queries, SIAP(2) and SSAP records must include isServedBy relationships to Obscore, TAP, and sitewide SIAP2 services.

Running the Queries

SIAP2 and Obscore are easy: Just translate the constraints to queries and collect the rows you get back (slightly normalised).

SIAP1 is more difficult: no (generally usable) constraints on time and spectrum, so you need to filter locally if possible. Also: result rows need to be mapped to the Obscore DM.

Main problem, though: Dealing with hanging or dead services, timeouts, hanging reverse proxies...

Inclusive: Services

By default, discover will only consider resources that give coverage.

To consider services without declared coverage, pass `inclusive=True`.

At this point, this will *dramatically* increase runtime and the amount of brokenness you will see.

Inclusive: Datasets

In particular for SIA1, inclusive has a second effect: It will also return *datasets* that do not declare temporal or spectral coverage.

For SIA2, the decision whether to return datasets with unknown coverage is left to the service.

For Obscore, we *could* add clauses like `OR time_min IS NULL` for inclusive searches – but we don't at this point.

How to log

Reminder:

```
images, log = images_globally(...)
```

How machine-readable should *log* be?

It's an artefact of provenance at the very least: you need to know which services happened to be down when your discovery ran.

It is also potentially a debugging aid.

But if you log everything, it will be extremely painful to see anything in the log. Feedback on what I am doing right now is most welcome.

More API: Custom Service Lists

Perhaps you want to query a custom subset of services?

Well: Pass a `services` parameter, perhaps like this:

```
datasets, log = discover.images_globally(
  space=(274.6880, -13.7920, 1),
  services=registry.search(registry.Datamodel("obscore")))
```

More API: Watching live

When you have something like a UI, or you want to break out of discovery, you want to get notifications of what is going on.

Define a `callback(discoverer, msg)` that is called each time something happens; you can muck around in the Discoverer instance in the first argument:

```
def say(discoverer, s):
        print(s)

datasets, log = discover.images_globally(
  time=(time.Time('1995-01-01'), time.Time('1995-12-31')),
  watcher=say)
```

OMG Testing

Writing unit tests for code of this kind is a nightmare: We basically need to mock a major part of the VO.

What did not work: Record requests and responses on the http level to build the mock environment. PyVO and Python produce different requests in different versions.

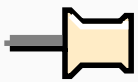
What I do: Some small tests with remote data. Yuck.

Demo Time: API

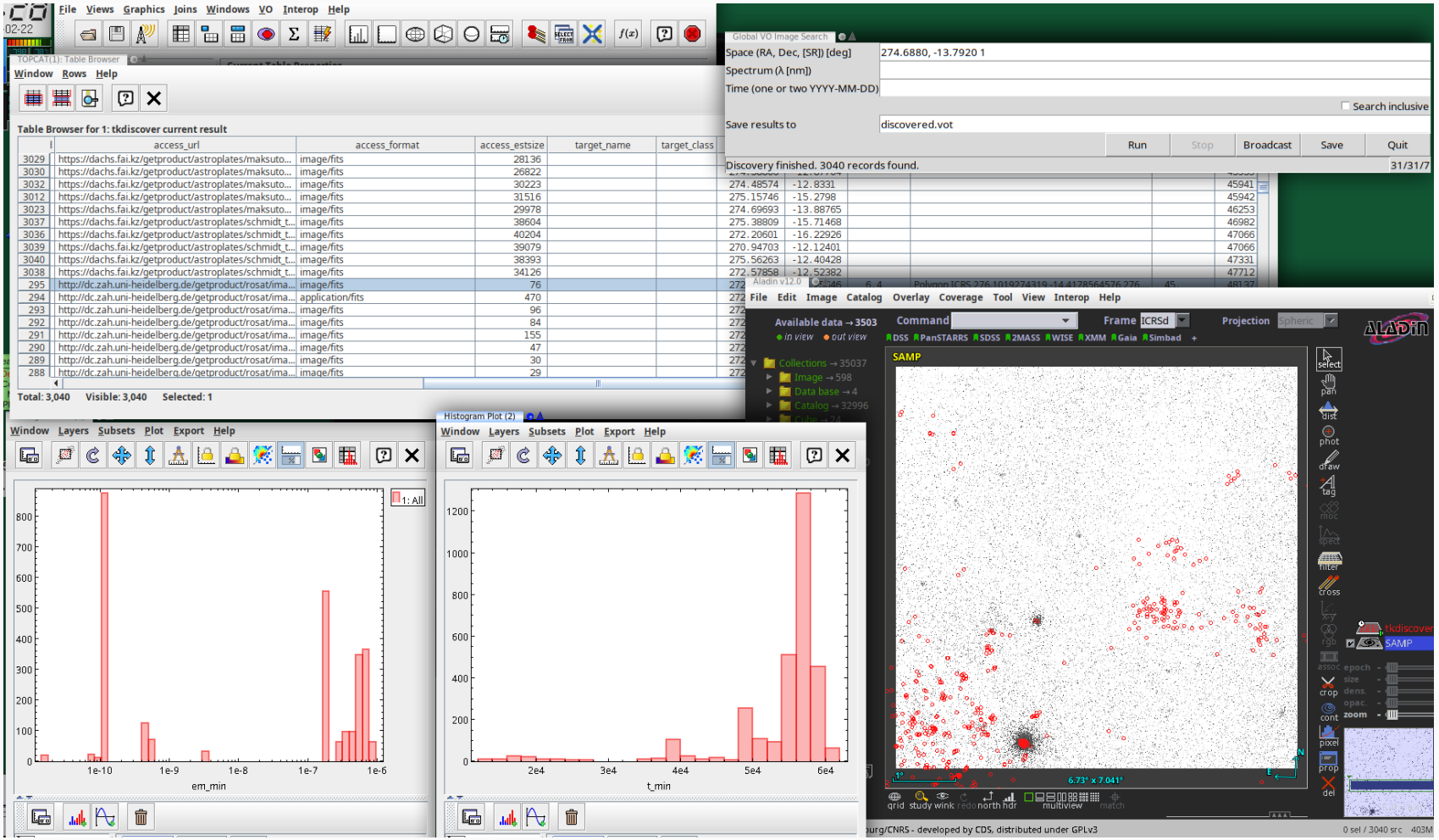
```
def say(disco, msg):
    print(process_time(), msg)

images, logs = discover.images_globally(
    space=(134, 11, 0.1),
    spectrum=600*u.eV,
    time=(time.Time('1990-01-01'), time.Time('1999-12-31')),
    watcher=say)

print("============ Result ===========")
print(logs)
for res in images:
  print(res.access_estsize, res.access_url)
```



Demo Time: A Simple GUI



https://github.com/ivoa/tkdiscover

Future Directions

Once the basics are there, here's some extensions I could see:

- Registry work with data providers to make this faster and more reliable
- Allow RoI geometries (polygons, perhaps even MOCs), intervals for scalars
- Enable object lists for upload (but: that will only work for Obscore)
- Optionally, automatic cutouts to the RoI using SODA?
- Non-remote testing by mocking and recording on the XService level?

Conclusion

Everything is hard (at least at first) in a global, distributed system.

But, as the internet itself shows, global, distributed systems are great, so with a bit of luck it is effort well spent.

See also my blog post on this:
https://blog.g-vo.org/global-dataset-discovery-in-pyvo.html.