*Fig. 1*

# 1. A short course in PyVO

Markus Demleitner *(msdemlei@ari.uni-heidelberg.de)*

This course will introduce you to the primary concepts of PyVO, an astropy-affiliated package for accessing Virtual Observatory services from Python. It's too much for a day in an interactive situation, so if you're reading this at the beginning of a course day: Say what you're interested in – we'll have to select material anyway.

It assumes familiarity with VO concepts (services, protocol types, the registry) as well as astropy, but you can probably gather missing parts as you go (or ask, if you're reading this in an interactive course situation). You should now enough of ADQL to be able to understand and edit queries.

The course is structured into an general introduction that you should at least cursorily read. It introduces a few general patterns for pyVO usage, and it discusses the basics of the Table Access Protocol TAP, which arguably is the most versatile protocol in the VO. It does this along a few more or less contrived use cases designed to touch the central topics.

In a second part, we have a number of largely unconnected special topics, for instance, receiving SAMP messages, using EPN-TAP, or Datalink. Look at these as need arises.

The full source code for the programs discussed here is also available as an attachment if you read this in pdf. One way to retrieve them is to get pdftk (there's packages for it for Debian-derived systems), run `pdftk pyvo.pdf unpack_files`. Other PDF tools may also support attachments. For instance, in KDE's Okular its at File/Embedded Files, and in Adobe's proprietary Acrobat Reader 9, attachments can be retrieved through the paperclip icon in the lower left corner.

**Please open a browser** and point it to

**http://docs.g-vo.org/pyvo**

It's also a good idea to have a page open on PyVO's documentation[1] as well as astropy's documentation[2].

Coming up (possibly): VO redux, operating simple data access services, multi-service queries, sending results around on the desktop, parameter discovery, TAP queries, async services, UCDs, ObsTAP, Registry, VO for the solar system, datalink, remote manipulation...



(cf. Fig. 1)

1 `http://pyvo.readthedocs.io/en/latest/`
2 `http://docs.astropy.org/en/stable/`
3 `https://creativecommons.org/licenses/by/4.0/legalcode`



# 2. Prerequisites

- python and astropy, of course (we assume Debian stable, at least; anaconda on proprietary systems should do, too)
- TOPCAT[4] for viewing and visualising tables
- Aladin[5] to work with images
- PyVO. Get it from
  - https://pypi.python.org/pypi/pyvo
  - or try `apt-get install python-pyvo`
  - or try `pip install pyvo`
  - or try `conda install pyvo`

# 3. What's the VO

The VO is a set of standards that let clients discover and interrogate astronomical data services in a uniform manner. Standards include:

- Registry – describing and finding services
- VOTable, UCD – writing tables with rich metadata
- SAMP – connecting software components
- SCS, SIAP, SSAP – querying catalog, image, and spectral services
- TAP – running remote database queries
- Datalink – bundling up complex data and services
- MOC, HiPS – sky coverage and hierarchical imaging

The purpose of all this is so machines instead of humans can operate the services. With an average web page, that's hard to impossible.

Machines operating services, in turn, are important to save manual work. This is part convenience, but mainly it's so you can use more and diverse data for your research.

See the IVOA home page[6] for more information.

4 `http://www.star.bris.ac.uk/~mbt/topcat/`
5 `http://aladin.u-strasbg.fr/aladin.gml`
6 `http://ivoa.net`

# 4. What's PyVO?

PyVO provides APIs for lots of VO protocols.

It's glue between astropy and python in general and the astronomical data services in the VO.

PyVO works for both python2 and python3. We hope the examples here do, too (but they're mostly only run under python2, so complain if you're seeing odd errors with python3).

It's a community project. You're welcome to contribute at
PyVO on github[7]

We will speak almost all of the protocols mentioned above within this course, but there's no need to dig into what all of them do here – they'll come in quite naturally when we want to solve problems.

# 5. Running Simple Services

When querying "simple" remote services (image, spectral, cone search; *not* directly TAP), PyVO has a consistent pattern:

```
# <prot> is SIA, SSA, SCS, SLA...
import pyvo

# construct a service object with a service's endpoint URL
service = pyvo.dal.<prot>Service(access_url)

#call the search method with the protocol's parameters
for result in service.search(<parameters>):
  ...work on dict-like object result...
```

The "dal" in here means "Data Access Layer", which essentially means: the VO protocols dealing with how to query services and how the services are supposed to respond.

You'll soon learn who to find out the access URLs.

7 https://github.com/pyvirtobs/pyvo



# 6. Query a Single Image Service

Example: SIAP, the VO's protocol to access image servers.

Query a VO service for a list of images covering a small field on the sky, and download one of these images:

```
svc = pyvo.sia.SIAService(ACCESS_URL)
images = svc.search((11,35), (0.1, 0.1), verbosity=2)
image=iamges[0]
image.cachedataset()
```

[See basicsiap.py]

For SIAP, `pos` (as a tuple of ra and dec) and `size` (in degrees, either one radius or extent in ra and dec) are mandatory. More parameters: in the pyvo docs[8].

Note how you don't actually have to know anything about the service except its access URL. Since PyVO uses a standard protocol, it knows enough to be able to, in this case, retrieve the file and (mostly) give it a reasonable name.

This is a very basic example though. PyVO provides you with more functionality that helps analysing the results before selecting the images. We will see some of these functions by using PyVO in an interpreter (e.g. ipython).

**Aufgaben**

**(6.1)** Get the full program from the attachment trivialsiap.py and inspect it to see how things work.

Find some other image service – use WIRR[9] to access the VO Registry for now – and see if they have some images for the positions given (or positions you're actually interested in).

What's coming back from SIAService's `search` is a sequence of SIARecords. Have a quick look a its pyvo documentation[10] and make your program print the file size, too. If you find some frames of reasonable size, download them into your favourite FITS viewer.

**Datei(en) im PDF-Anhang:** trivialsiap.py

8 http://pyvo.readthedocs.io/en/latest/api/pyvo.dal.SIAService.html#pyvo.dal.SIAService.search

9 http://dc.g-vo.org/WIRR

10 http://pyvo.readthedocs.io/en/latest/api/pyvo.dal.SIARecord.html#pyvo.dal.SIARecord

# 7. This is Python

The advantage of doing this in Python is that it's easy to add your own logic. Here's how to add time constraints (SIAP version 1 unfortunately doesn't specify how to tell the service you're only interested in a specific time interval – we'll later see how more modern standards let you push time constraints to the server) and search multiple positions:

```
svc = pyvo.sia.SIAService(ACCESS_URL)
for pos in [
    (10, 20),
    (45, 85)]:
  images = svc.search(pos, (0.5, 0.5), verbosity=2)
  for row in images:
    if not DATE_MIN<row.dateobs<DATE_MAX:
      continue
    row.cachedataset()
```

[See multisiap.py]

Also: `row.cachdataset` saves the image to your local disk under a name sensible for the metadata.

**Datei(en) im PDF-Anhang:** multisiap.py

# 8. And now all-VO

The nice thing about standard services: Handle one, and you get them all. So, let's add a query to the Registry and run our query all over the VO –

```
for svc in registry.search(servicetype="image", waveband="optical"):
  try:
    search_one_service(svc.accessurl, image_sender)
  except Exception:
    import traceback; traceback.print_exc()
```

[See globalsiap.py]

The Registry that's being used here is a big directory of all the services that are in the VO. If you have tried the problem above and tried WIRR: it's the same underlying data.

The Registry interface of PyVO is a bit simplistic. For a few basic tasks, it works ok, though. You will usually want to avoid doing "blind" all-VO queries, as they will take a long, and you'll be hitting a lot of services that will return data you can't use and just have to week out later. For cone searches, just don't ever do it, since there are more than $10^4$ of them

We will later see how to directly use RegTAP, the powerful and flexible Registry query technology; we recommend you use that for all but the most trivial discovery tasks.

Having said that, you can also search for `catalog`, `spectrum`, `line`, and `database` services, and `waveband` takes quite a few other terms (see docs for which these are). You can also pass `keyword` (for queries against titles and descriptions), and datamodel (which we'll look at later) as further keyword arguments.

The exception catcher is there since not all services claiming to be standards-compliant actually are. It doesn't hurt to complain to the service operators if a service you're interested in behaves weirdly – sometimes the operators haven't noticed it's broken or just broke.

To find out who to complain to, you can again use the Registry. In the most common case, you would use WIRR with an access URL constraint. In the query result, you should see a mail address when clicking on the person icon.



You will probably also see lots of warnings from astropy's VOTable parser. This is partly because astropy is overly paranoid, rejecting UCDs actually required by the SIAP standard, partly because operators botch things. Interoperability isn't always easy. I'd say at this point it's too early to complain to operators about your average VOTable warning, which is why we'll later shut them off.

If a service hangs, you can interrupt it by hitting Control-C. In production code, you can use python's `socket.settimeout` is your friend to fail broken services after a while and just go on.

Rule: In multi-service queries, expect at least one service to be broken. Write your scripts to cope.

#### Aufgaben

**(8.1)** Get the globalsiap.py script from the attachment and change it so it skips 90% of the services discovered randomly (use `random.random()`). Also, remove the constraint on the date (few services will have stuff this old) and change the position to something you're interested in or expect to have pretty pictures (M1's or M51's are always good candiates). Run the thing and see what you find.

**Datei(en) im PDF-Anhang:** globalsiap.py

# 9. Add SAMP Magic

SAMP lets you exchange data between VO clients. Your script is a VO client, too. Let's make it broadcast some of the found images:

```
with vohelper.SAMP_conn() as conn:
  ... (search) ...
  vohelper.send_image_to(conn, None, image.acref)
```

[See globalsiapsamp.py]

(also, vohelper.py abstracts SAMP here).

For this course, but probably also for convenience in wider usage, we have gathered some helper functions in a module vohelper that you can find on the web page and attached to the PDF. Have a glance at the source code if you want. Otherwise, just dump it next to your scripts so you can import it.

Before running this, start Aladin so the images are displayed.

SAMP-enabling programs may not come quite natural to people who so far have mainly written fairly linear science code, because when doing SAMP you usually want to react to external events. In linear code this is rather uncommon.

In this example we are just sending data, which does not require much reacting to external signals. We still have to manage the connection to the SAMP hub – things get ugly if you don't properly close the connection –, which is taken care of by a context manager from vohelper.

If you inspect how send_image_to actually is implemented in vohelper, the way arguments are passed between SAMP services may seem a bit funky: We build message dictionaries with odd keys and then use methods on the "conn" object. But think of the mtype as the function name, and passing arguments in dictionaries instead of tuples isn't that far-fetched, either.

Given we're doing function calls between different processes written in different languages, I'd argue this kind of code actually is surprisingly compact.

**Datei(en) im PDF-Anhang:** globalsiapsamp.py

**Datei(en) im PDF-Anhang:** vohelper.py

**Aufgaben**

**(9.1)** Have a look at the implementation of SAMP_conn in vohelper. This is done as a context manager, which is a python construct ensuring what's called "external invariants" (e.g., the status of a file is closed before and after a piece of code that needs it). They're used together with the `with` keyword that you may already know from file handling.

Can you imagine why such a context manager is a good idea here? Try the code creating a connection and run it without a disconnect several times. Look at the SAMP info in TOPCAT meanwhile. **(L)**

# 10. Enter TAP

What we've seen so far doesn't scale when you're interested in more regions.

Also, only fairly basic constraints are supported.

TAP is far more powerful.

Sample use case: Integrate photometry from different source catalogs, do some local work on results, try to obtain spectra for interesting candidates.

# 11. Step 1a: Synchronous Queries

Run queries via TAP:

```
access_url = "http://dc.g-vo.org/tap"

service = pyvo.dal.TAPService(access_url)
result = service.run_sync(
  """SELECT raj2000, dej2000, jmag, hmag, kmag
       FROM twomass.data
       WHERE jmag<3""")
for row in result:
  print(row["raj2000"], row["jmag"])
```

[See fetch3.py]

This is another instance of the PyVO pattern "create a service object, then call a method". In this case, we're calling `run_sync` – this is not called `query` as for the other services because TAP has two modes of operation; we'll get to the other one (unsurprisingly called async) in a moment.

What's coming back from `run_sync` is a sequence of `dal.Record` elements (well, the truth about TAPResults[11] is a bit more complex, but that's the gist of it).

`result.to_table()` is an astropy.table instance – here, we take a column from it. To save it, say:

```
with open("result.vot") as f:
  result.to_table().write(output=f, format="votable")
```

**Aufgaben**

**(11.1)** Write a program that prints the number of rows in the table `arihip.main` in the TAP service at http://dc.g-vo.org/tap (do *not* pull all the rows and use python's len).

Hint: With ADQL's `AS` construct you can control the names of table columns. **(L)**

[11] `http://pyvo.readthedocs.io/en/latest/api/pyvo.dal.TAPResults.html`



# 12. Step 1b: Three Queries, TOPCAT

Separate "science" from "code" as much as possible:

```
QUERIES = [
  ("twomass", "http://dc.zah.uni-heidelberg.de/tap",
    """SELECT TOP 1000000 raj2000, dej2000, jmag, hmag, kmag
    ...
  ("allwise", "http://tapvizier.u-strasbg.fr/TAPVizieR/tap",
    """SELECT raj2000, dej2000, w1mag, w2mag, w3mag, w4mag
    ...

with vohelper.SAMP_conn() as conn:
  topcat_id = vohelper.find_client(conn, "topcat")
  for short_name, access_url, query in QUERIES:
    service = pyvo.dal.TAPService(access_url)
    result = service.run_sync(query.format(**locals()), maxrec=90000)
    vohelper.send_table_to(conn, topcat_id, result.table, short_name)
```

[See fetch3.py]

Also new: send retrieved tables directly to TOPCAT.

We specify services and actions in a list `QUERIES`. Each list item is a tuple consisting of short name (which later is used as a label in TOPCAT), an access URL, and a query, which we've hand-crafted here. It is conceivable to automate this a lot by generating the queries based on metadata you get from the services, but that's for later. Obviously, new services can simply be added by appending another triple to this list.

For a detailed introduction into TAP and ADQL please look at `http://docs.g-vo.org/adql`

Note the `maxrec` parameter – when you expect large result sets, always pass an explicit maxrec to the service, or it may truncate your results. Most services have rather moderate default maxrecs (our server software assumes 2000 unless the operators override it).

**Datei(en) im PDF-Anhang:** fetch3.py

**Aufgaben**

**(12.1)** Use TOPCAT's TAP data browser to locate services and table names for TGAS and RAVE (or just use the GAVO DC TAP service with tables tgas.main and rave.main). Also figure out where the positions and some usable magnitude are, plus the proper motions from TGAS and the radial velocities from RAVE (or just blindly use ra, dec, pmra, pmdec, phot_g_mean_mag for TGAS and raj2000, dej2000, rv, and hmag for RAVE).

Write queries to retrieve proper motions from TGAS and radial velocities from RAVE for all stars between 8 and 8.5 mags of some magnitude (don't worry about the difference between H and G for this problem).

Then, re-write fetch3.py to query the two services, and change it to send the results to Aladin (which is known as *Aladin* in upper case on the SAMP bus). See if you can get a nice plot of rv, pmra, and pmdec (which, of course, would be particularly interesting for the stars that happen to be in both sets). **(L)**

# 13. Step 2: Go Async

When doing a lot of queries or long-running queries, run them asynchronously and in parallel.

Asynchronous means that you go to a service, submit your query there and immediately receive some sort of token. With this token, you can come back later and retrieve your result. In the meantime, you're free to do whatever else you have to do – which includes turning off and/or moving your machine, for instance.

```
jobs = set()
for short_name, access_url, query in QUERIES:
  job = pyvo.dal.TAPService(access_url).submit_job(
    query.format(**locals()), maxrec=9000000)
  job.run()
  jobs.add((short_name, job))

 while jobs:
  time.sleep(5)
  for short_name, job in list(jobs):
    if job.phase not in ('QUEUED', 'EXECUTING'):
      jobs.remove((short_name, job))
      vohelper.send_table_to(conn, topcat_id,
          job.fetch_result().table, short_name)
      job.delete()
```

[See fetch3_async.py]

We told you sync is easier to program with. But on the other hand: With this program, all three queries run in parallel, which is nice, in particular if they take a while. Additionally, you have a little more control about when to receive the data.

What's happening here? First, we submit all jobs. Rather than `run_sync` we now use TAPService's `submit_job` method. While taking the same arguments as `run_sync`, it immediately returns. Since it can't peek into the future, it can't return the finished result. Instead, you get an object that one can use to manipulate the remote job. That remote job is *not* started by `submit_job`. It is instead waiting for further configuration (e.g., increasing its maximal runtime) or a request to put it into the processing queue.

For our task, it's enough to just start the job using the `run` method. We then add it to a watch set of running jobs..

The rest of the code above is all about managing this set. In a polling loop – be sure to introduce sleeps or your code will hit the remote services all the time – we iterate through the jobs. Actually, we iterate over a copy of the job set since we want to delete completed from it, and we couldn't do that if there was an iterator over it active.

In the loop body, we check the `phase` attribute of the job. Although this looks like an attribute access, in each iteration PyVO goes to the remote service and asks it what our job is doing. While it is in either QUEUING or EXECUTING states, it's still worth waiting for a result. Other states include PENDING (not yet started), COMPLETED (done, result available), ERROR (done, some kind of failure happened; call the `raise_if_error` method to turn it into a python exception), and ABORTED (interrupted by client or operator intervention).

Once we find a job is done, we remove it from the job list and send the result over to TOPCAT as before.

Finally, we delete the remote job. That's a nice thing to do. Services will eventually delete your job anyway (you can figure out when and even change that date in the job's `destruction` attribute), but it's good style to discard jobs once you don't need them any more.

Note that PyVO also gives you a `run_async` method on `TAPServices` – this works exactly like `run_sync`, i.e., it will block until the results are in. Use it if you have to go async because your job runs too long for sync (in general, sync jobs have to finish in seconds to minutes, while async jobs can run for hours) but you want to avoid the dance with checking the phases.

**Datei(en) im PDF-Anhang:** fetch3˙async.py



# 14. Step 3a: UCDs build SEDs

Can we build SEDs from the results of the three services?

Not simply; photometry metadata in the VO isn't quite sufficient for that yet. However, UCDs let us do a workaround:

```
UCD_TO_WL = {
  "phot.mag;em.opt.u": 3.5e-7,
  "phot.mag;em.opt.b": 4.5e-7,
  "phot.mag;em.opt.v": 5.5e-7,
  "phot.mag;em.opt.r": 6.75e-7, ...

  for row in rows:
    for index, col in enumerate(row):
      ucd = row.columns[index].meta.get("ucd", "").lower())
      if ucd.startswith("phot.mag"):
        if ucd in UCD_TO_WL:
          phots.append((UCD_TO_WL[ucd], col))
```

UCDs ("Unified Content Descriptors") are VO-standardised strings defining the physics contained in columns. They even have a bit of syntax. In our example, we can see that first, we have magnitudes ("phot.mag") and then that they were taken in a certain band.

Similarly, "pos.eq.ra" would tell you that something is a right ascension as part of an equatorial position; since tables sometimes have multiple positions in a single row (e.g., different reduction, position in some reference catalog, or position of a sub-feature), you may want to single out a particular column as your preferred, primary, default, or whatever RA. For that, use "pos.eq.ra;meta.main".

UCDs are particularly nifty in data discovery when you're looking for tables that have a certain kind of physics. Of course, that only works when people properly mark up their tables with UCDs – be sure to do that on your data whenever you let a VOTable leave your disk. The full list of UCD atoms is available from the IVOA document respository[12].

The clean way, incidentally, is a proper annotation of the columns in question with full photometry metadata (e.g., central wavelength, bandwidth, perhaps a URL of the detector's response curve, etc). The details are hellish, but there actually is a photometry DM in the VO. There's just not a good way to put that info into a VOTable yet. If you're looking for something to contribute to the VO: this would be a good task. Just ask on the IVOA's data models mailing list.

**Aufgaben**

**(14.1)** Can you figure out what the UCD for a B-V colour would be? **(L)**

[12] `http://www.ivoa.net/documents/latest/UCDlist.html`

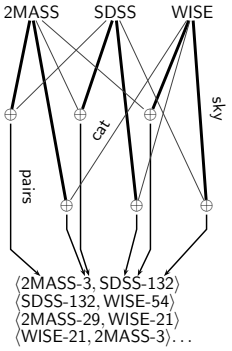


*Fig. 2*

# 15. Step 3b: Aggregate Photometry

Construction of "clusters" is in vohelper.py and uses astropy's `SkyCoords` and `match_catalog_to_sky` (asymmetric!).

For three catalogs, we must perform six sky matches to get pairs, then walk the graph to gather the clusters.

This actually is pure astropy and has nothing to do with PyVO as such. As a matter of fact, it is usually smarter to have the remote sides do the cross matches if at all possible.

In this case, since we don't have a "master catalogue" to match against, that's actually hard. For smallish crossmatches, the code in vohelper works reasonably well (but it scales horribly when then number of tables increases; use specialised packages when your problem takes that direction).

What's happening in that code? `sky_coords` are `astropy.SkyCoord` instances (in the example code, there's a function `get_coordinates_for_table` that makes these for essentially arbitrary tables as long as they're properly marked up).

The code then goes through all pairs of input `SkyCoords` and uses their `match_to_catalog_sky` method to generate pairs of indices into these objects that are the closest pairs (that operation isn't symmetrical, which is why we compute the matches with all permutations).

The remaining code filters out those pairs that are closer than a limit that's passed in and adds a new pair of rows to be matched to a set. Each row is designated as a pair of table index and row index within that table.

The rest is a graph problem: If you compute the connected subsets of the graph formed in this way, you'll have all measurements that are crossmatched together and thus, hopefully, correspond to one object.

Sorry for this excursion. Feel free to ignore this.

(cf. Fig. 2)



# 16. Combine with "your" Code

This is python: Add your own logic!

Here: Let's display the approximate SEDs and let the user interactively select "interesting" cases.

```
for pos, phots in seds:
   to_plot = np.array(phots)
   plt.semilogx(to_plot[:,0], to_plot[:,1], '-')
   plt.show(block=False)
   selection = raw_input(
     "s)elect SED, q)uit, enter for next? ")
   if selection=="q":
     break
   if selection=="s":
     selected.append(pos)
   plt.cla()
 return selected
```

[See fetch3_cluster.py]

This is now fairly standard matplotlib. We're interacting through `raw_input` in the shell here for simplicity. It's not actually hard to interact through the matlotlib window, but that requires a bit object magic that we wanted to avoid here.

**Datei(en) im PDF-Anhang:** fetch3˙cluster.py

**Aufgaben**

**(16.1)** Go through the source code of `fetch3_cluster`. You'll see we've put in two workarounds for where the data providers messed up. Can you see in each case what might have gone wrong? Have the service operators fixed their software or do things still fail when you remove a workaround? In a course setting, coordinate with your neighbours and split up the work so each only looks at one workaround.

**(16.2)** Run the program and select a couple of objects. Keep the resulting file (`selected_positions.vot`) – we'll want to reuse it later.

# 17. Looking for Spectra

Suppose you have a couple of positions for "interesting" objects. Can we find spectra for them?

SSAP is the traditional VO protocol to access spectra, quite like SIAP, and we could query SSAP services just like we queried SIAP services. However, SSAP only lets you access one object at a time, which is kind of tedious.

Let's use

**ObsTAP** = TAP with table `ivoa.obscore`

`ivoa.obscore` has lots of metadata on observational data products (spectra, cubes, timeseries).

Having what people generally call a "data model" – here, rather a set of pre-defined columns – enables a lot of powerful data discovery scenarious when coupled with TAP. So, why do we bother with SCS, SIAP, and SSAP?

Good question. It mainly has historical reasons – S*AP where easier to define than TAP and Obscore. And until datalink was there, there were a few tricks you could play with them that just don't work with simple ObsTAP (cutouts, for instance).

Even now, there's still much less data in ObsCore services than in SSAP; hence, if your problem easily admits querying through SSAP, it's certainly no mistake to do so, perhaps in addition to Obscore (beware: there is some data that's in Obscore but not in SSAP).

Plan:

- Search for `obscore` services
- Use TAP upload to search to collect spectra
- Send spectra to SPLAT

# 18. Query the Registry

Iterate over all obscore services (here: see what data collections they house):

```
for svc_rec in pyvo.registry.search(datamodel="obscore"):
  svc = pyvo.dal.TAPService(svc_rec.access_url)
  result = svc.run_sync("SELECT DISTINCT obs_collection"
    " FROM ivoa.obscore")
  print("\n>>>>{}\n{}\n".format(
    svc_rec.short_name,
    "\n".join(
      r["obs_collection"] for r in result))
```

To "use ObsTAP", just query the `ivoa.obscore` table; all (registred) ObsTAP services can be found by passing the data model name "obscore" to `registry.search`.

The selling point here is: we're running *the same* database query on all the ObsTAP service, and we're processing their results in the same way. That's the power of uniform data models.

This script doesn't come attached. That's because on large services, the `SELECT DISTINCT` can actually be computationally expensive for the services. Do not run this script *just* for fun.



# 19. Query with Upload

For each ObsTAP service, we query against our object list:

```
if not svc.upload_methods:
  return

result = vohelper.run_sync_resilient(svc,
  """SELECT TOP 2000 oc.obs_publisher_did, oc.access_url
      FROM ivoa.obscore AS oc
      JOIN TAP_UPLOAD.pois AS mine
      ON 1=CONTAINS(
        POINT('ICRS', oc.s_ra, oc.s_dec),
        CIRCLE('ICRS', mine.ra, mine.dec, 0.01))
      WHERE oc.dataproduct_type='spectrum'
      """),
      uploads = {"pois": pois})
```

What's going on here? Right after constructing the service, we check whether it supports table uploads – not all TAP services do. TAPService objects have a few other attributes that let you figure things out about services. This, in particular, includes resource limits (maximum upload size, limit to which maxrec can be raised, etc).

Here, it's enough to know there's any upload method at all, because the standard says that inline upload must be supported if there's any upload support.

To actually perform the upload, pass a dictionary to the `uploads` keyword argument of run_sync and friends. The keys there are simple names (starting with a letter and letters or numbers after that), the values can be various things, but you'll probably get by passing either a string (which is interpreted as a URL to fetch a VOTable from) or an astropy table.

You can upload multiple tables using different keys; for each key, a table `TAP_UPLOAD.`*key* becomes available – in the example above, that's `TAP_UPLOAD.pois`. You will almost always join the uploaded table with a table on the service, and thus it's almost always a good idea to use ADQL's `AS` construct to give abbreviated names to tables. The name `mine` is typically a good choice if you only have one upload.

Note that even if you don't *have* to qualify column names in a query with a join with the source table names, you will regret not doing so in queries you will likely reuse (trap: you'll always reuse the one you least expect to reuse) – just because there's no column `s_ra` in the table uploaded here doesn't mean the table you have in your next program doesn't, and if it has and you've not used the `oc.` prefix (here), your function will fail.

Instead of the common `run_sync`, this uses `vohelper.run_sync_resilient`, which catches all kinds of exceptions and other trouble. As said above, when you do all-VO queries, expect at least one service to fail completely and another to give results that look like they come from a fuzzer.

The actual obscore query does a classical, ADQL 2.0 crossmatch, because we are querying lots of services, many of which will not be updated to more recent standards even by the time your read this. Also, spectra are essentially point-like objects, so you probably do *not* want to write the at the first glance more attractive alternative
`1=CONTAINS(POINT('', up.ra, up.dec), s_region)`

This *could* be more attractive if you're looking for images or other artefacts with a reasonable coverage. Note, however, that proper s_region support is not mandatory, whereas all data providers get the center RA and Dec for their datasets roughly right. The bottom line is: If you can get by with just positions (rather than `s_region`) in your obscore queries, do it.

The code in get_spectra.py is actually a bit more general in that it doesn't hardcode the column names in the uploaded table but instead discovers them using UCDs. So, as long as your tables are properly annotated, the function there will just work for global spectra discovery (or, if you change the query, really any other global ObsCore discovery on sets of positions).

**Aufgaben**

**(19.1)** Can you figure out the default output limit (i.e., in effect an implied TOP) for the TAP service at http://dc.g-vo.org/tap? How far can you raise it?

Can you write a program that figures it out for all TAP services out there that talk about tgas? **(L)**

**(19.2)** (This is more of an async excercise, but you need the upload stuff to solve it)

One particularly cool part about async is that you can keep your results publicly available on the remote server for a while. That, in turn, you can use to do cross-service joins *without having to download intermediate tables*.

As said above, you can use URLs in a query's upload argument. To try this out, review the TGAS and RAVE example above. Make the RAVE query asynchronous. Watch the resulting job, and when it is done, get the URI of the result table from the job's `result_uri` attribute. Push that into an upload such that the following query does a join between the two datasets:

```
SELECT *
FROM
  tgas.main as tg
  JOIN TAP_UPLOAD.rave as mine
  ON (1=CONTAINS(
    POINT('', tg.ra, tg.dec),
    CIRCLE('', mine.raj2000, mine.dej2000, 1/3600.)))
```

Obviously, this is much more logical than the first version, since there's just one constraint on the magnitudes now (the one on the H-band in rave) – when you send the resulting table to Aladin, you'll see more matches in TGAS than you had when you were comparing the two catalog cuts manually. **(L)**

# 20. Collect Spectra finished

The rest is almost standard SAMP fare to get the spectra retrieved to SPLAT as they come in:

```
try:
  target_id = vohelper.find_client(conn, "splat")
except KeyError:
  sys.exit("Start Splat and try again")
...
for ds_name, access_url in specs:
  try:
    vohelper.send_spectrum_to(
      conn, target_id, access_url, ds_name)
  except vohelper.SAMPProxyError:
    print("  (Failed)")
```

[See get_spectra.py]

As for images, spectra are usually passed around by their URLs in SAMP.

What's new here is that we're catching exceptions. Catching the KeyError when trying to find Splat is probably helpful when you run the program after a couple of months, have forgotten about splat being a part of this analysis chain, and would wonder otherwise why you see a `KeyError` of all things.

The second exception catching, around `send_spectrum_to`, is a lot more subtle. Here, and exception is raised if Splat fails to open a spectrum and sends back a notice to this effect. So, what we're catching here, in effect, is an exception raised within Splat. In general, there's no telling if the target client has already informed the user that something is wrong – it's probably better to assume it has in generic code most of the time, and so sending code should avoid modal error messages ("Click here to continue"). Our very terse notification might not be optimal either, though.

**Datei(en) im PDF-Anhang:** get˙spectra.py



**Aufgaben**

**(20.1)** Can you change our program such that only spectra of resolving power 10000 or greater are retrieved?

Hint: Use TOPCAT or the `tables` property of your `TAPService` to inspect the metadata of the `ivoa.obscore` table to figure out which column to query against. Just in case: It's almost always better to filter on the remote side rather than the local side, and that's always true if the constraint can be expressed as a single condition in a WHERE clause.

# 21. End of Part 1

We believe you now know enough to further explore PyVO and the VO on your own.

If you're into jupyter notebooks, this page has one attached that revisits TAP uploads, discovery, and SAMP.

However, we've prepared a couple of extra slides on special topics. Here's some titles – let me know after the break what you'd like to do.

- Reacting to SAMP messages
- Solar System science with EPN-TAP
- Using datalink
- Doing TAP uploads the right way
- Multi-service TAP (Improvised)
- Walking a spectral grid (Improvised)
- Discovering and using custom parameters in DAL services

**Datei(en) im PDF-Anhang:** tap˙obscore.ipynb

# 22. Higher SAMP Magic

Let's say you're debugging your pipeline and want to manually inspect "weird" objects by querying a set of other catalogs have on them.

Plan: Write a program that other clients

- can send tables to (`table.load.votable`) and then
- when a table row is selected, computes a new table
- that's then broadcast.

**Pattern for listening:**

```
conn.bind_receive_notification(
  "table.highlight.row",
  self.handle_selection)
```

SAMP is based on messages; there are several message types (MTypes), which are documented on the IVOA wiki[13].

To make our program ready to receive tables via SAMP, we have to listen to `table.load.votable`.

To react to row selections, we have to react to `table.highlight.row`. An alternative would be `coord.pointAt.sky`, which communicates where people are looking at; but in this case we're looking for odd rows, not odd positions.

The SAMP client objects's `bind_receive_notification` method arranges for the hub to call a function when a message of a certain MType comes in. The calling pattern is a bit complicated, but what really counts is a dictionary of the parameters passed to the originating call; according to what's said on the wiki, you'll be passed a table-id, a URL, and a row index.

#### Aufgaben

**(22.1)** The somewhat verbose argument list for a handler of a SAMP message is `handler(privkey, sender_id, msg_id, mtype, params, extra)`. You can usually ignore all of these except the params, which are a dicitionary.

Write a little program that listens for `coord.pointAt.sky` messages and just prints the sky coordinate looked at. Test it by starting the program when TOPCAT or Aladin are already running. In Aladin, you can just pan around. In TOPCAT, you must configure an activation action to see something.

Hint 1: The basic code to obtain a client object as discussed on the "Add SAMP Magic" slide.

Hint 2: At least some versions of astropy don't show exceptions raised within a handler function. To save yourself grief in such cases, decorate your handler function with `vohelper.show_exception`, that is, defined it like this:

```
@vohelper.show_exception
def print_coord(privkey, sender_id, msg_id, mtype, params, extra):
```

Hint 3: This program just waits for events from the outside, which is common for server programs but perhaps scary to you if you've mainly written "user code" so far. Astropy makes it easy for you, though – just have your program wait with `raw_input` and you're fine. **(L)**

[13] `http://wiki.ivoa.net/twiki/bin/view/IVOA/SampMTypes`



# 23. Doin' It With Class

Our program needs to manage quite a bit of state. At least:

- A table sent to us
- The SAMP connection

Whenever your problems gather state (and that's quite usual when you handle SAMP messages), think **object**.

Don't be scared: An object is essentially like a dictionary with an odd syntax and some keys giving slightly magic functions ("methods"). For instance, a `table.load.votable` handler:

```
class VicinitySearcher(object):
  def __init__(self, client):
    self.client = client
    self.cur_table = self.cur_id = None
    self.client.bind_receive_call(
      "table.load.votable", self.load_VOTable)

  def load_VOTable(self, private_key, sender_id, msg_id, mtype,
      params, extra):
    self.cur_table = Table.read(params['url'])
    self.cur_id = params["table-id"]
    self.client.reply(msg_id,
      {"samp.status": "samp.ok", "samp.result": {}})
```

[See vicinitysearcher.py]

The trivial version of object lore in python is: All functions belonging to an object (*methods*) have a first argument called `self`, and whenever you put an attribute on `self`, you can find it again in other methods' `self`, provided these other methods are called on the same *instance* (i.e., object)..

To call other methods of the same object, use `self.methodname`.

Create an object (you can now call that an instance and sound a lot cleverer) by calling the class (here: `VicinitySearcher(conn)`). Whatever you pass into the constructor will be passed to the `__init__` method.

**Datei(en) im PDF-Anhang:** vicinitysearcher.py

#### Aufgaben

**(23.1)** The action of the SAMP handler is in the make_response_table method; have a look at it. The UCDs used are those of SCS, an ancient standard made before modern UCDs were invented.

Use UCDs to add another column, `mag`, which, for this exercise, can be just the first column you find with a UCD starting with `phot.mag`.

Hint: You can iterate over result.table.columns, and you'll find the UCD in `col.meta['ucd']`.

# 24. EPN-TAP 1: Discovery

EPN-TAP is a protocol for distributing solar system data; essentially, it's normal VO TAP plus a pre-defined table structure; the tables are always called `epn_core`.

Let's try an all-VO query for data on Mars. For discovery, we use GloTS:

```
glots_svc = pyvo.dal.TAPService("http://dc.g-vo.org/tap")
epn_services = glots_svc.run_sync(
  "SELECT accessurl, ivo_string_agg(table_name, '#') as tables"
  "  FROM glots.services NATURAL JOIN glots.tables"
  "  WHERE table_name LIKE '%epn_core'"
  "  GROUP BY accessurl")
```

GloTS, the global TAP schema, is a service that iterates over all TAP services out there and harvests their TAP schemas. The union of those, annotated with the source service, are then published as `glots.services`, `glots.tables`, an `glots.columns`. This is a private initiative and should be supplanted by the registry soon-ish. While it exists, it's pretty nifty, though.

The official way to discover EPN-TAP services is a normal registry query looking like this:

```
SELECT ivoid, access_url
FROM rr.capability
NATURAL JOIN rr.interface
NATURAL JOIN rr.res_table
WHERE
  standard_id LIKE 'ivo://ivoa.net/std/tap%'
  AND table_utype LIKE 'ivo://vopdc.obspm/std/epncore#schema-2.%'
```

– however, this assumes that all EPN-TAP services properly publish table metadata, and, unfortunately, not all do, yet. Which is why we recommend using GloTS for now.

# 25. EPN-TAP 2: Querying EPN-TAP

EPN-TAP services are queried like any other TAP service. Use a table browser to see what columns are available or check the standard[14].

```
for svcrow in epn_services.table():
  service = pyvo.dal.TAPService(svcrow["accessurl"])
  for table_name in svcrow["tables"].split("#"):
    print("\nQuerying {} on {}".format(
      table_name, svcrow["accessurl"]))
    for row in vohelper.run_sync_resilient(service,
        "SELECT TOP 2 * FROM {} WHERE target_name='Mars'".format(
        table_name).table():
      print(row)
```

[See epnquery.py]

This looks a bit more complex than before because a single service can have multiple EPN-TAP tables, so we have another iteration to go, and we need to fiddle in the table names into the queries.

Things *are* a bit more complex than in anything we've seen so far because EPN-TAP lets people stick in almost any kind of data into such tables, and what your `access_url` points to – spectra, profiles of elemental abundances, odd magnetospheric data, or nothing at all – is impossible to tell before at least inspecting the `dataproduct_type` column (and even then your average non-solar-system astronomer may be stumped...)

Of course, you want to do smarter things than print a row.

**Datei(en) im PDF-Anhang:** epnquery.py

[14] https://voparis-confluence.obspm.fr/display/VES/EPN-TAP+V2.0+parameters



**Aufgaben**

**(25.1)** Get the little epnquery program and change it to only discover spectra. then send the first two spectra your program finds to TOPCAT (or SPLAT, if you have it).

# 26. Datalink: Related Infos

Datalink is a standard for "linking" files to datasets. Think previews, extracted objects, etc.

After a data discovery query *on a datalink-enabled service*, you can use the result's `iter_datalinks` method:

```
for dl in result.iter_datalinks():
  for link in dl: # multiple links per dataset
    print link
```

Each link has a URL, a description, and machine-readable semantics[15]. E.g., to load previews:

```
for dl in matches.iter_datalinks():
    prev_url = dl.bysemantics("#preview").next()["access_url"]
    im = Image.open(io.BytesIO(requests.get(prev_url).content))
    ...
```

[See datalink-previews.py]

**Datei(en) im PDF-Anhang:** datalink-previews.py

**Aufgaben**

**(26.1)** Write a function `get_available_semantics(dl) -> set` returning a set of the semantics available for a given datalink.

What semantics do you get for the links coming from the datalink-previews program? **(L)**

[15] http://www.ivoa.net/rdf/datalink/core

# 27. Datalink: Remote processing

Datalink also lets you declare processing services. SODA is a special set of parameters applicable to astronomical images (CIRCLE, POLYGON, TIME, BAND,...).

Save *a lot* of time by only downloading cutouts of the object you're interested in:

```
roi = SkyCoord.from_name('Mira')
for rec in svc.run_sync(
    "SELECT access_url, access_format FROM ivoa.obscore"
    " WHERE obs_collection='HDAP'"
    "AND 1=CONTAINS(CIRCLE('ICRS', {}, {}, 0.05),"
    "s_region)".format(roi.ra.deg, roi.dec.deg)):
    processed = rec.processed(
        circle=(roi.ra.deg, roi.dec.deg, 0.05))
```

[See datalink-soda.py]

**Aufgaben**

**(27.1) Warning: Doesn't work with pyVO as of Feb 2018**

CALIFA is a collection of spectral cubes (i.e., an array of small-band images) of galaxies; there is a datalink-enabled TAP table (califadr3.cubes) listing the cubes on the TAP service http://dc.g-vo.org/tap.

Use TOPCAT to inspect the tables belonging to califadr3; in particular note the objects table that you can join with cubes via the califaid column. Now write a program that gets images of Sd galaxies in their H$\alpha$ light.

Hints:

1. Use the Hubble type in the `hubtyp` column of califadr3's objects table. Use a TOPCAT query to see how Hubble types are written.
2. The cubes come in three different setups. Only look at `COMB` for this exercise (this avoids duplicate data).
3. H$\alpha$ is at 656.25 nm. For the low redshifts we're talking about here, use $\lambda_{\mathrm{lab}} = (1+z)\lambda_0$ to compute the wavelength in the lab frame (and don't worry about vacuum vs. air wavelengths for this exercise).
4. You'll need to select the `obs_publisher_did` and `mime` columns for datalink to work. Use `califaid` to generate file names.
5. To cut out by wavelength, use SODA's BAND parameter (future versions of pyVO will have better facilities to inspect the parameters the services support and the ranges applicable to a given dataset). It's in meters of (vaccuum) wavelength.
6. To cut out just a single pixel in wavelength, just use the same upper and lower bound.

**(L)**



# 28. Scaling TAP Queries

TBD (Take from https://blog.g-vo.org/adql-tricks-at-mpia/)

For many interesting discovery problems, a simple parameter-based interface as in `registry.search` just isn't powerful enough. Fortunately, if you know TAP and the (admittedly somewhat complicated) relational mapping of the Registry data model, you can get almost arbitrarily fancy. This is called RegTAP and is available from reg.g-vo.org (and a few other TAP services). There, you can query roughly a dozen tables that contain the service metadata. To learn more, inspect the metadata of the tables in the rr schema (they are designed to be joined using NATURAL JOINs) if you want to know more, and if you're still not satisfied, there's more explanations and examples in the underlying standard[16].

Here, we combine the tables of interfaces (things a client can talk to), capabilities (ways of using a services, in this case: TAP), and details (various "minor" properties of resources; in this case: implementation of a data model).

This may look a bit complex, but it's fairly stereotypic. If you have somewhat more advanced data discovery problems and want to come up with queries of this sort yourself, you may want to read 2015A&C....10...88D.

**Datei(en) im PDF-Anhang:** query˙lots.py

# 29. Operating Over Spectral Grids

TBD (let's have some spectral arithmetic here – anyone in for a nice python lib for rebinning spectra and computing RMSes?)

**Datei(en) im PDF-Anhang:** download˙a˙spectral˙grid.py

16 `http://ivoa.net/documents/RegTAP`

# 30. Splitting Up Queries

It usually pays to try and optimize ADQL queries (and we'll finally write a guide on this one of these days). But sometimes you just need to partition queries; for instance, your result set otherwise becomes too large, or your query really takes that long. In the latter case, you can play with execution_duration on async jobs:

```
job = svc.run_async("...")
job.execution_duration=10000
```

This will not help you when you hit the hard match limit. In such cases, the recommended way is to use the table's primary key to partition the data; usually, that should be the column with the UCD `meta.id;meta.main`. For a rough partition, where the partition sizes may be grossly different, just figure out the maximum value of the identifier. For our light version of Gaia DR2, you could query:

```
SELECT max(source_id) FROM gaia.dr2light
\endverbatim

(if that's slow, you probably haven't chosen a good primary key); in
this case, that yields 6917528997577384320.

With that number, you can enter a program like this:

\startverbatim
import pyvo

MAX_ID, N_PART = 6917528997577384320+1, 10
partition_limits = [(MAX_ID/N_PART)*i
  for i in range(N_PART+1)]

svc = pyvo.dal.TAPService("http://dc.g-vo.org/tap")
main_query = "SELECT count(*) FROM cur_part"

for lower, upper in zip(partition_limits[:-1], partition_limits[1:]):
  result = svc.run_sync("WITH sample AS "
    "(SELECT * FROM gaia.dr2light"
    "  WHERE source_id BETWEEN {} and {}) ".format(lower, upper-1)
    +main_query)
    print(result)
```

(Exercise: Can you see why the +1 is necessary in the MAX_ID assignment)

You'll obviously have to adapt that a bit when the primary key is a string, but that's rare these days.

Since most astronomical objects are distributed highly unevenly on the sky, this will yield chunks of very different sizes for commen schemes, where the idenfier somehow encodes the sky position.

If you have a use case where you need a guaranteed maximum result size per partition, you will have to use two passes, first figuing out the distribution of objects and then computing the desired partition from that.

Here's an example for how one might go about this:

```
from astropy import table
import pyvo

MAX_ID, ROW_TARGET = 6917528997577384320+1, 10000000

ENDPOINT = "http://dc.g-vo.org/tap"

# the 20000 is just the number of bins to use; make it too small, and
```



```
# your inital bins may already overflow ROW_TARGET
ID_DIVISOR = MAX_ID/10000

QUERY = """
select round(source_id/%d) as bin, count(*) as ct
from gaia.dr2light
group by bin
"""%ID_DIVISOR


def get_bin_sizes():
        """returns a ordered sequence of (bin_center, num_objects) rows.
        """
        # since the partitioning query already is expensive, cache it,
        # and use the cache if it's there.
        try:
                with open("partitions.vot", "rb") as f:
                        tbl = table.Table.read(f)
        except IOError:
                # Fetch from source; takes about 1 hour
                print("Fetching partitions from source; this will take a while"
                        " (provide partitions.vot to avoid re-querying)")
                svc = pyvo.dal.TAPService(ENDPOINT)
                res = svc.run_async(QUERY, maxrec=1000000)
                tbl = res.table
                with open("partitions.vot", "wb") as f:
                        tbl.write(output=f, format="votable")

        res = [(row["bin"], row["ct"]) for row in tbl]
        res.sort()
        return res


def get_partition_limits(bin_sizes):
        """returns a list of limits of source_id ranges exhausting the whole
        catalog.

        bin_sizes is what get_bin_sizes returns (and it must be sorted by
        bin center).
        """
        limits, cur_count = [0], 0
        for bin_center, bin_count in bin_sizes:
                if cur_count+bin_count>MAX_ROWS:
                        limits.append(int(bin_center*ID_DIVISOR-ID_DIVISOR/2))
                        cur_count = 0
                cur_count += bin_count
        limits.append(MAX_ID)
        return limits


def get_data_for(svc, query, low, high):
        """returns a TAP result for the (simple) query in the partition
        between low and high.

        query needs to query the ``sample`` table.
        """
```

```
        job = svc.submit_job("WITH sample AS "
    "(SELECT * FROM gaia.dr2light"
    "  WHERE source_id BETWEEN {} and {}) ".format(lower, upper-1)
    +query, maxrec=ROW_TARGET)
  try:
          job.run()
          job.wait()
          return job.fetch_result()
        finally:
                job.delete()


def main():
  svc = pyvo.dal.TAPService(ENDPOINT)
        for ct, (low, high) in enumerate(zip(limits[:-1], limits[1:])):
                print("{}/{}".format(ct, len(limits)))
                res = get_data_for(svc, low, high-1)
                # do your thing here
```

But, most importantly: If you need any of this, you're probably doing it wrong.

# 31. Custom Parameters

SIAP only has very few standard parameters (e.g., no time constraints), and even SSAP's rich parameter set is insufficient for, e.g., theoretical spectra.

SIAP and SSAP services can define custom parameters. Discover them using a `FORMAT=METADATA` URL parameter.

The input parameters are given as VOTable params in the root VOTable RESOURCE, where their names are prefixed with `INPUT:`. You can figure out names, units, descriptions, and, if the service operators do a good job, even hints as to what you should pass in when you want to get data back.

PyVO doesn't yet have some API that would properly hide this (not terribly pretty) implementation detail. Worse, it's not totally trivial to get these PARAMs with astronomer-level PyVO.

To make amends, this course comes with a script `viewparams.py` that has a function and a UI to retrieve metadata. To see how an example works, try
`python viewparams.py "http://dc.g-vo.org/bgds/q/sia/siap.xml?"`

Pass custom parameters as keyword arguments to search:

```
  svc.search((107, -10), (0.1, 0.1),
    dateObs="57050/58050",
    bandpassId="SDSS i'")
```

[See siapextra.py]

The attached script `siapextra.py` that lets you look at this in practice uses extra trickery we've not seen yet to avoid downloading entire datasets. This is using a standard called datalink that we'll look at later.

**Syntax trouble:** Old-style VO services (parameters usually declared as `char[*]` or `double`) write intervals with slashes.

New-style (SIAv2, datalink...) have *interval* xtypes and type `double[2]`. Their intervals are written with a blank.

We're sorry about this, but not all standards work out well on the first attempt. In defence of the early standards authors that came up with the wretched slash syntax: There was prior un-art for this from the geospatial community.



**Datei(en) im PDF-Anhang:** viewparams.py siapextra.py

**Aufgaben**

**(31.1)** There is a spectral service with the access URL

`http://dc.g-vo.org/theossa/q/ssa/ssap.xml?`

It houses theoretical spectra mostly of hot, compact stars (think central stars of planetary nebula or perhaps young white dwarfs).

Start from trivialsiap.py to write a little script querying the service.

Inspect its custom parameters using the attached `viewparams.py` script and see if you can retrieve three spectra for stars with log_g (if you don't know what that is: viewparams.py will tell you) between 4.5 and 5.5, an effective temperature between $7\times10^4$ and $10^5$ Kelvin, and a Nitrogen mass fraction (that that's in "dex" – decadic logarithm of ratio to the solar value – goes without saying for people in the field; the metadata could explain that better, yes) larger than 0.015 (write `+Inf` for "no upper limit").

Use TOPCAT to convince yourself that you actually retrieved spectra.

Hint: Remember `dal.ssa.SSAService`.

Hint: To stop at three spectra, `enumerate` is a nifty thing.

Hint: Due to SSA breakage, this service will return each spectrum as both text and VOTable. To retrieve each spectrum just once (and in VOTable), pass `FORMAT='VOTable'` to search. **(L)**

# 32. TAP Uploads: The right way

TAP uploads are powerful, but they do have limits. In general, you cannot upload billion-row tables and expecte services to go along.

To make things fast and save the server's resources, you should only upload enough to select the relevant data. So, avoid:

```
first_result = svc1.run_sync(...).to_table()
second_result = svc2.run_sync(
  "SELECT * FROM local.t JOIN TAP_UPLOAD.up as b USING (foo, bar)",
  uploads={"up": first_result})
```

– this will upload all of first_result and download it right again; transferring data you already have, ingesting it into the remote database in between is just a waste of resources.

Instead, if you want to join on first_result's columns foo and bar, make a new local table containing just those plus a unique local identifier (add a record number if no such identifier exists), somewhat like this:

```
first_result = svc1.run_sync(...).to_table()
remote_match = svc2.run_sync(
  "SELECT * FROM local.t JOIN TAP_UPLOAD.up as b USING (foo, bar)",
  uploads={"up": table.Table([
    first_result["main_id"],
    first_result["foo"],
    first_result["bar"])})
second_result = table.join(
  first_result,
  remote_match,
  keys="main_id")
```

In practice, you might still run into resource limits when doing the upload join. In that case, you should first re-think what you're doing. If there's really no way around it, you can split up the remote action into bunches. The attached file shows that technique together with the upload column selection.

**Datei(en) im PDF-Anhang:** smart-tap-upload.py