Using the Virtual Observatory

Markus Demleitner   Hendrik Heinl   Joachim Wambsganß

July 25, 2024

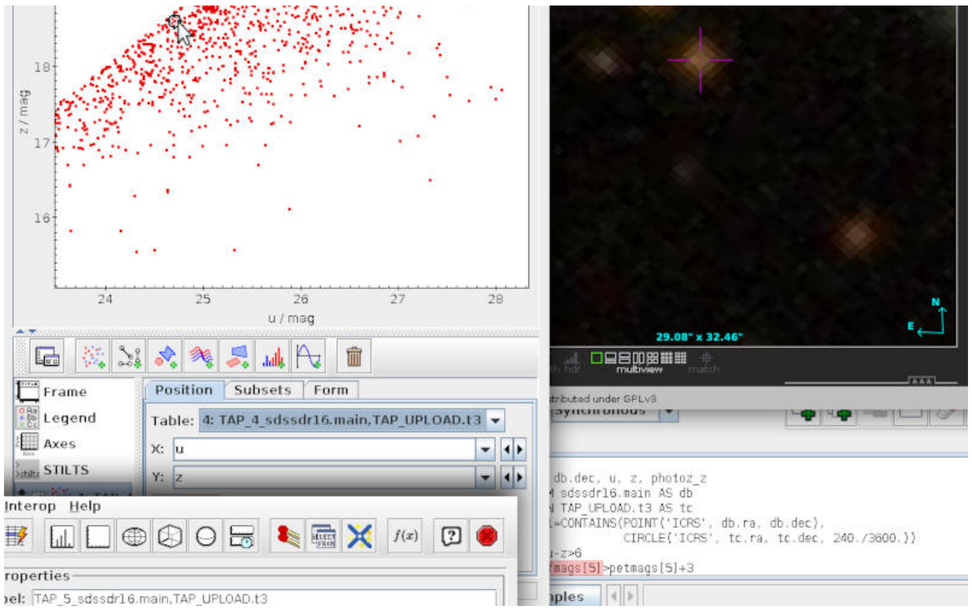


Table: 4: TAP_4_sdssdr16.main,TAP_UPLOAD.t3
X: u
Y: z

bel: TAP_5_sdssdr16.main,TAP_UPLOAD.t3

```
 db.dec, u, z, photoz_z
M sdssdr16.main AS db
N TAP_UPLOAD.t3 AS tc
1=CONTAINS(POINT('ICRS', db.ra, db.dec),
           CIRCLE('ICRS', tc.ra, tc.dec, 240./3600.))
u-z>6
fmags[5]>petmags[5]+3
```

Abstract

This is a course on using the Virtual Observatory (VO), an international research data infrastructure in Astronomy and Astrophysics. Starting with a brief discussion of some general concepts, it introduces some of the major client programs like TOPCAT and Aladin, together with some simple discovery protocols. A first focus topic is the query language ADQL, which is treated within the equivalent of three lectures. The second major focus of the course is the premier Python interface to the VO, pyVO, which is used to also more deeply investigate the topics treated before. The course is complemented by a number of side tracks, brief discussions of more fundamental or more specialised VO topics.

The course comes with many exercises, most of which also have solutions. We hope it is suitable for both self-study and as lecture notes in teacher-led situations. In the latter case, it is designed to work as a semester-long course with two hours of lectures and lab work each per week.

Participants should have some basic knowledge of astronomy, and for the later parts of the course basic skills in Python.

Contents

1 Introduction: What is the VO and why should you care?

The VO is…

1. **not** a website ("platform"),
2. **not** a bunch of websites,
3. **not** a program that does all things astronomy.

Instead…

The VO is…

Standards for finding, accessing, using, and describing data (more on which in a moment)

plus

$\sim$ 50 data centers worldwide adhering to these standards (which includes almost all the major players like ESO, NASA, ESA, etc)

plus

a few volunteers operating some $\pm$ central infrastructure (these are things like searchable Registry endpoints, our document repository, and the various bodies working on the standards)

plus

authors of client software, libraries, and web pages making these resources available to astronomers and the public (newcomers might want to look at TOPCAT and Aladin on the desktop, pyVO and STIL as libraries, or ESA Sky or Aladin lite in the browser).

Numerically…

In numbers, the VO is:

1. $\sim$ 50 data centers in $\sim$ 20 countries
2. $\sim 3 \times 10^4$ data collections
3. hundreds of millions of data sets (spectra, images,…)
4. hundreds of billions of table rows

How do I use it?

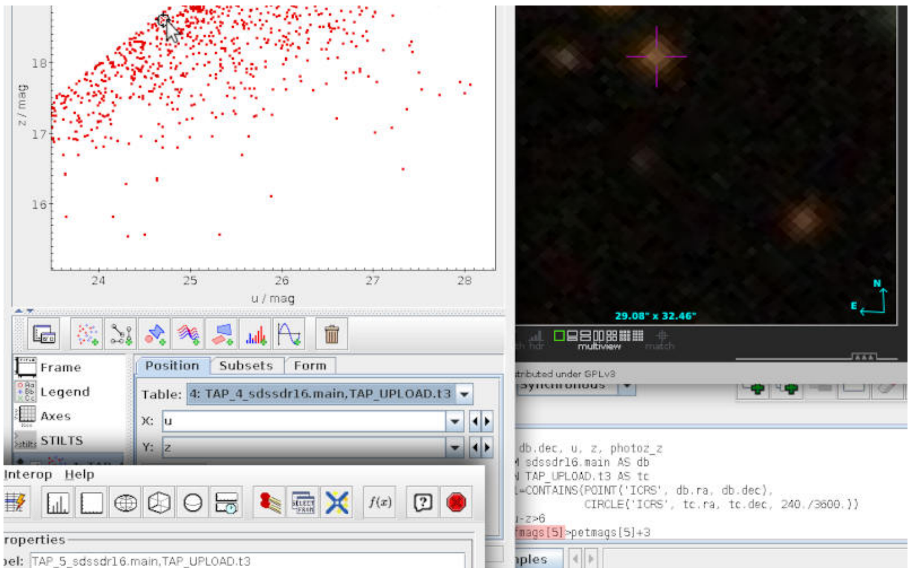
While certain parts of the VO can be consumed from web browsers, you really want client software that can talk to our APIs. Many astronomy programs can "interoperate" with the VO and in that sense are such clients. But a minimal VO toolset would usually include:

- TOPCAT – does what you want with tables
- Aladin – interactive sky atlas
- pyVO – marrying the VO and astropy

Hint: All of them can be apt install-ed on modern Debian and derivatives. A notable absence is clients for spectral analysis. Popular VO-enabled spectral clients include Splat and CASSIS, which, however, are not Debian-packaged. You will find them from the IVOA Applications page https://ivoa.net/astronomers/applications.html (but note that it is not complete).

Whetting your appetite: Demo time

Assume you want to look for candidates for gravitationally lensed compact objects. Operationally, that would be highly redshifted (and hence rather red) pointlike objects around compact clusters of galaxies (which are good candidate for lenses)



```
 db.dec, u, z, photoz_z
M sdssdr16.main AS db
N TAP_UPLOAD.t3 AS tc
1=CONTAINS(POINT('ICRS', db.ra, db.dec),
           CIRCLE('ICRS', tc.ra, tc.dec, 240./3600.))
u-z>6
fmags[5]>petmags[5]+3
```

Click it yourself: Using the VO in interactive clients You are of course not expected to understand everything I do here at this point; but the plan is that when the course is over, you can come up with something like this yourself. Let's see.

Still, you should try to reproduce this now. This will help you sort out things as the course goes on. Here are notes on how to do that:

Start TOPCAT; we are looking for a catalogue, hence open VO → TAP.

Our basic data is on clusters of galaxies; so, do a Registry search (in *Keywords*) for **galaxy cluster**.

There is a table from heasarc called *mcxc*. Double click it, and on the pane opening then find the table an inspect the metadata.

To characterise the data, use a query like:

```
select
  min(radius_500), max(radius_500),
  min(lx_500), max(lx_500),
  count(*)
from mcxc
```

Ah, there are just 1743 entries in total – we can easily pull the whole thing; control-click to select the columns we want and then use the Cols button to insert the selected columns into a query like this:

```
select dec, ra, redshift, radius_500, lx_500, name
from mcxc
```

Get reasonably luminous compact ones by plotting radius vs. luminosity and creating a blob subset.

Choose the row subset you have just created in the main window.

Change to sdssdr16 on GAVO's TAP service, click Examples → Upload Join; choose columns, widen the radius to 4 arcmin (whatever) and select only very red stuff:

```
SELECT
   name,
   db.dec, db.ra, u, z, photoz_z
   FROM sdssdr16.main AS db
   JOIN TAP_UPLOAD.t4 AS tc
   ON 1=CONTAINS(POINT('ICRS', db.ra, db.dec),
                 CIRCLE('ICRS', tc.ra, tc.dec, 240./3600.))
WHERE u-z>6
```

Look at a table, do a sky plot, plot u vs. z (or whatever suits you).

To look at things, start Aladin. Back in TOPCAT, in Views → Activation Actions check Send Sky Coordinates. Then click on objects that look interesting in the plot and inspect them in the various surveys.

If bored with objects having plain morphologies, perhaps add

```
and psfmags[5]>petmags[5]+3
```

or something like that to fetch objects for which the PSF does not quite work.

Now suppose you have better criteria to figure out promising candidates, but you can only write them in Python (rather than ADQL). Well: we're talking to APIs, so switching to an actual programming language is straightforward.

To reproduce, save your candidate table (the subset from mcxc) to **candclus.vot**.

Then, get Ψglc.py) from the lecture notes PDF. Note how TAP_URL and QUERY are directly copied from our exploratory TOPCAT operation in the TAP part:

A first taste of VO-enabled Python

```
import pyvo
from astropy import table

TAP_URL = "http://dc.zah.uni-heidelberg.de/tap"
QUERY = """
SELECT
  name, db.ra, db.dec, u, z,
  photoz_z, petrads, petmags
FROM sdssdr16.main AS db
JOIN TAP_UPLOAD.t3 AS tc
  ON 1=CONTAINS(POINT('ICRS', db.ra, db.dec),
    CIRCLE('ICRS', tc.ra, tc.dec, 240./3600.))
WHERE u-z>6
"""

svc = pyvo.dal.TAPService(TAP_URL)
objs = svc.run_sync(QUERY,
  uploads={"t3": table.Table.read("candclus.vot")}).to_table()
```

The rest of the program does some lame filtering and then broadcasts the positions one by one via SAMP:

```
with pyvo.samp.connection() as conn:
  for o in objs:
    pr = o["petrads"]
    if (sum(pr)>40 and min(pr)*6<max(pr)
        and argmax(list(o["petmags"]))==1):
      print(f"Showing a candidate around {o['name']}")
      send_position_to(conn, o["ra"], o["dec"])
      input()
```

By the way, if you open the Cone Search dialog in TOPCAT, you will see it reflects the position your program has broadcast, too. Which may come in handy one of these days.

Exercise 1

Follow the "Whetting your appetite" demo from the lecture notes until you have the radius-luminosity plot for mcxc.

Now try the same thing for the eingalclus; when constructing the new query, you will have to change the columns to select and the table name.

Do the plots look similar? Use a plausible cross match (Pair Match in TOPCAT's Joins menu) to pair plausible clusters and plot radius and luminosity from the two catalogues against each other.

Exercise 2

Continuing the last exercise, do the next steps for the mcxc example from the lecture notes until you can see the suspicious red objects. Take care that you do not lose track of which table you have selected at any time and make sure you adapt the table index in `TAP_UPLOAD.t4` to the index of your mcxc table. Can you find interesting objects? And can you think of ways to reduce the contamination with odd artefacts?

2 Simple Protocols and their clients

Simple VO protocols

The basic/simple VO-protocols are

- the **S**imple **C**one **S**earch (SCS) for table data
- the **S**imple **I**mage **A**ccess (SIA) for images
- the **S**imple **S**pectral **A**ccess (SSA) for spectra
- the **S**imple **A**pplication **M**essaging **P**rotocol (SAMP) for interoperability between software

Neither simple nor a protocol, but crucial and omnipresent:

- the VO Registry

With the exception of SAMP, the simple protocols define "typed" interfaces to give users a common way to query one collection of a specific kind of data. "Simple" means that it is fairly easy to write queries, but these are not very elaborate. Also each service just queries its own data collection; joins between different collections ("cross matches") are not possible on the server side.

More elaborate ways for quering data collections will be discussed in the next chapter where you will learn about TAP/ADQL.

Each of the simple protocols not only defines an interface for parameters, but also the technical communication protocol used to transmit the query and receive the result. In technical terms: the query must be an HTTP GET request, and the result must be a VOTable. The good news is: you do not need to know the details about how that works to use it in your client software. Hence, the learning curve here is not too steep.

In the following, we are looking into the simple protocols.

Simple Cone Search (SCS)

The SCS provides a data selection on table data based on the parameters of a position (RA, DEC) and a search radius (SR) in degrees around it.

Clients: Topcat, STILTS, pyVO, curl, the web browser of your liking, and more.

Let's get straight into an example for how to use SCS. We will take the use case of identifying members of an open star cluster by their proper motions, and will check this selection by plotting an colour-magnitude diagram of our cluster candidates.

If our idenfication method by proper motion is accurate, we can should see a narrow main sequence in the colour-magnitude diagram even if we do not account for the distance of the objects. Our steps here will be

1. Find a catalogue that provides proper motions and colours
2. Perform a query on that catalogue around a given position
3. Make a selection using a proper motion plot

4. Plot an colour-magnitude diagram

Start TOPCAT. We are looking for a specific catalogue provided by the GAVO data center. Open VO → Cone Search. In the new opening Window under *Keywords* type **hsoy**. Note: at this time, we do not provide an introduction to Data discovery, so we just assume *somebody* told you about his catalogue, or you saw it mentioned in a paper (in fact, Vizier is providing VO-services too.). Then click on **Find Services**. And yes, these days you would use the Gaia catalogue for what we are about to do; but since the discovery of HSOY is trivial, it's ideal for this example.

Thus we performed a query on the VO Registry. The Registry is where all VO services come together and register the metadata on their data, like coverages in space, time or wavelength. For now it is ok if you know that the Registry is the usual VO entry point, and that you can search for data on VO services using keywords. In that sense, you can think of it as Google of astronomical data (except for Google's snooping, which we don't do).

As a result of our search for HSOY we see three results, one of the GAVO service, and two at Vizier. At this point we ignore the result for Vizier und select the GAVO service. Just click on it in the list. You will see that below this list, a few pieces of information about the selected service are updated.

In the query window below, at *Object Name* type **pleiades** and click on **resolve**. With this we used the name resolving service Sesame which is hosted by the CDS in Strasbourg. Using this, TOPCAT can fill out the position in the form.

Now all we need to fill in is the *Radius* of **1.5** degrees. Click on **OK** to start the query.

When TOPCAT loads the table in its table plane, this has been a successful SCS query. Easy, no?

To identify the Pleiades members by their proper motion, we make use of Topcat's plotting features: go to Graphics → Plane Plot. On the *X-Axis* we select **pmRA**, and on the *Y-Axis* we select **pmDE**. You may notice two regions of overdensity. The bigger one is around (0,0) and simply shows the more distant stars in our selection (remember: the SCS did not select for distance). But there is also a smaller population visible, which we now will select as our subset of pleiades candidates.

Go to Subsets → Draw Blob Subset. This will activate the manual subset tool, which lets you draw a blob over the subset of data records you are interested in. To draw, simply hold your left mouse button. When finished, click on the same menu button as before. A small window will open, which will ask you to name the subset. Give the subset a meaningful name like "candidates", so you can easily indentify it later.

Now it's time to check if our candidates are an accurate sample. For this we look at the colour-magnitude diagram of our candidates. Luckily the HSOY catalogue provides us with colors from the 2MASS survey, and also with a magnitude from the Gaia catalogue (of course we are cheating here, not only because we knew which catalogue to select beforehand, but also because we were involved in computing and publishing it). Open another plane plot window. At *X-Axis* now type **Jmag-kmag**, and at *Y-Axis* type **phot_g_mean_mag**. Eventually we need to click on Axies and tick **Y-flip**, because we are plotting magnitudes.

The resulting plot does not look like a main sequence at all, but you may notice, that we are looking at all objects from the cone selection, so this is no surprise. To only see our candidates subset, look below the plot and go to Subsets and tick **candidates**. The subset will now be highlighted in the plot in a different color, and looks much like a main sequence.

Keep the data in TOPCAT, we will need it later.

Exercise 3

Follow the instructions in the SCS chapter (use the lecture notes) to get familiar with SCS and topcat.

1. Try to repeat the exercise using a Gaia catalogue instead of the HSOY. Caveat: the Gaia catalogue provides colours and magnitudes. Try to use the metadata of the table to figure out the names of the columns for brightness, and blue and red colours. Another good hint to not get lost in the column metadata is to search for "Gaia lite".

   In case you get stuck, you can also have a peek at:
   https://www.g-vo.org/tutorials/pleiades.pdf

2. Try to repeat the steps with the beehive cluster ("Praesepe")

Simple Image Access (SIA)

SIA services work similar to SCS services but for image access. The resulting VOTables are lists of metadata on images on a specific service matching the query parameters, thus enabling users to make decisions on which images to download.

Clients: Aladin, pyVO, curl, the web browser of your liking, and more.

Again, let's see how SIA Services work with an example: Open Aladin and wait a few seconds until it builds it data tree on the left side of the window.

What happens here in the background are two queries: the first is for the CDS Service and hosted images and catalogues on their side. These data collections will be shown in yellow in the tree. All other services will be shown in blue. Below the tree you can see the query fields, which we will use in a bit. In the middle you see the *view* field, and on the right is the *stack*.

These different fields are actually easy to understand: on the left is remote data discovery, on the right is local data handling and in the middle is where the science happens.

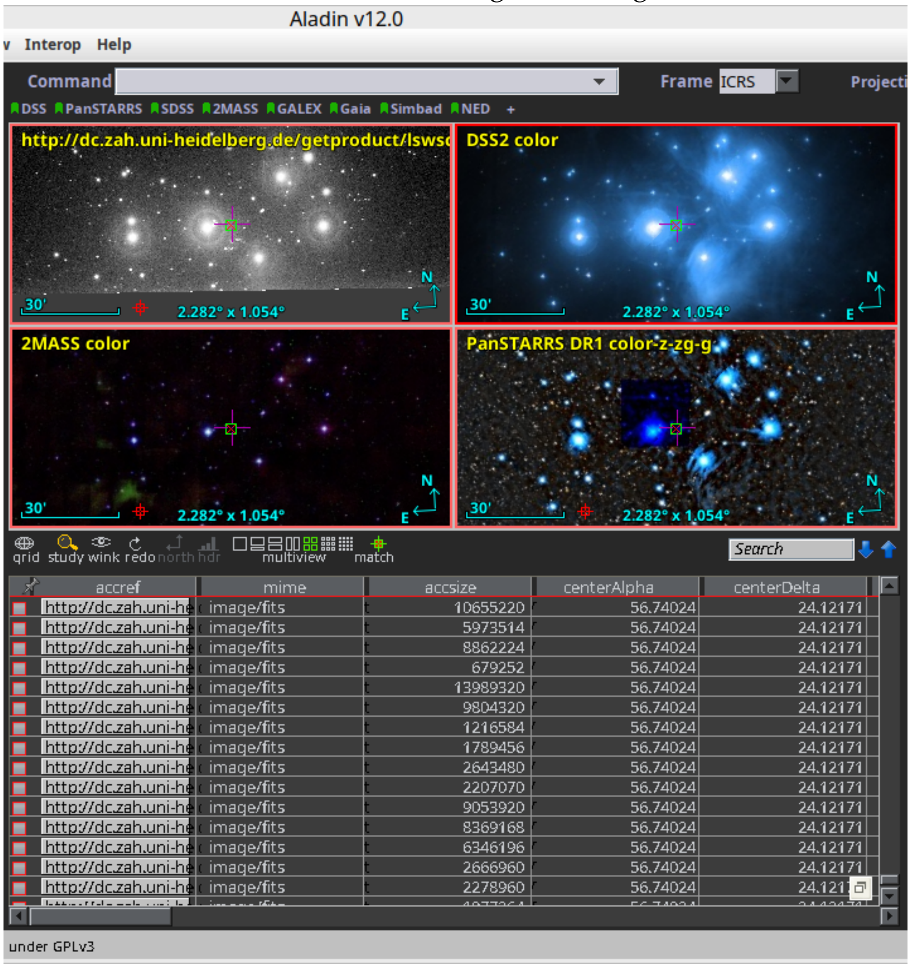
The middle seems very empty, so let's fill it with someting: simply click on DSS. Now at *command* above the view, type **pleiades** and press enter. The viewer will now jump to the according position, and you can have a look at the pleiades by zooming in and out, or paning the view around. Note that when zooming Aladin will change the resolution of the images. This happens thanks to the VO standard HiPS, which we will briefly meet again in the HEALPix interlude 4.

You may notice that some of the services in the data tree changed colors from green to orange or vice versa. This is a very helpful feature in Aladin enabled thanks to services publishing their coverage: Aladin can know whether a service provides data for the current field of view. Services which do will be shown in green. Here is another VO standard doing its magic in the background: MOCs, which will also be covered (pun intended) later on.

Let's see if the GAVO service provides any data in the proximity of the Pleiades. Below the data tree at *select*, type **HDAP**, which is the Heidelberg Digitized Astronomical Plates collections from Landessternwarte. Click on it once, and in the small window tick **in view**, and click **Load**. After a few seconds you will see a list appear below the view. This list is the VOtable containing meta data on the images of the LSW collection in the field of view.

Use the mouse to hover over the list and you will see squares pop up in the view: these are the areas covered by the corresponding image. Search the list until you find an image that is covering quite a bit of the pleiades, then simple click on that line. It may take some time for the image to download. Once the download is finished, the view will change to the image.

Aladin has a neat feature which we will now use: the multiview, which you find directly below the view. Select the view with four windows, which we will fill with different surveys. Just click on an empty window, and then in the survey selector click on the surveyes **DSS**, **2MASS**, and **PANSTARRS**. You should now see something like the figure below.


Aladin v12.0
v Interop Help
Command
Frame ICRS
Projecti
DSS PanSTARRS SDSS 2MASS GALEX Gaia Simbad NED +
http://dc.zah.uni-heidelberg.de/getproduct/lswsc
DSS2 color
2MASS color
PanSTARRS DR1 color-z-zg-g
30'
2.282° x 1.054°
N
E
grid study wink redo north hdr
multiview
match
Search
under GPLv3

accref	mime	accsize	centerAlpha	centerDelta
http://dc.zah.uni-he	image/fits	10655220	56.74024	24.12171
http://dc.zah.uni-he	image/fits	5973514	56.74024	24.12171
http://dc.zah.uni-he	image/fits	8862224	56.74024	24.12171
http://dc.zah.uni-he	image/fits	679252	56.74024	24.12171
http://dc.zah.uni-he	image/fits	13989320	56.74024	24.12171
http://dc.zah.uni-he	image/fits	9804320	56.74024	24.12171
http://dc.zah.uni-he	image/fits	1216584	56.74024	24.12171
http://dc.zah.uni-he	image/fits	1789456	56.74024	24.12171
http://dc.zah.uni-he	image/fits	2643480	56.74024	24.12171
http://dc.zah.uni-he	image/fits	2207070	56.74024	24.12171
http://dc.zah.uni-he	image/fits	9053920	56.74024	24.12171
http://dc.zah.uni-he	image/fits	8369168	56.74024	24.12171
http://dc.zah.uni-he	image/fits	6346196	56.74024	24.12171
http://dc.zah.uni-he	image/fits	2666960	56.74024	24.12171
http://dc.zah.uni-he	image/fits	2278960	56.74024	24.121

Exercise 4

Use Aladin's viewer to study the object X Persei in different wavelengths. Get familiar how to select a survey and also make use of the data tree.

Hint: try the XMM survey. Try to guess what kind of object you are looking at. How could you confirm your hypothesis? (Well, of course there is Wikipedia, but what if you wanted to keep it in the VO family?)

Simple Spectral Access Protocol (SSA)

SSA works very similar to SIA. The result of a SSA query is a VOTable with spectra matching the query parameters. Users can then select which spectra to actually download.

Clients: SPLAT-VO, CASSIS, curl, the web browser of your liking, and more.

Simple Application Messaging Protocol (SAMP)

SAMP is a bit of the magic in the VO. It is designed so that VO clients can interopate and communicate whith each other. Thus users really can select client software of their choice and make them interact with their own scripts.

Clients: almost all of them. It's fun. In the pyVO part of this course you will learn how to write your own SAMP clients.

In order to work, SAMP needs a running SAMP hub, to which each client connects and identifies which messages it will accept. The hub then organises the communication between the different clients. At the core of this interoperability of course is the VOTable standard.

Using SAMP is really simple. At this point we return to the main window of TOPCAT and tick the table with the Pleiades candidates. In the right window under *Row Subset* we select our defined subset **candidates**. Then we go to Interop → Send Table to → Aladin. Then we switch to the Aladin window and now can see a new data plane in the stack, and also the data points are plotted on the images in our four view windows.

HTTP and clients of your choice

The Hyper Text Transfer Protocol HTTP is the transport protocol of choice in the VO. It is open, robust, and well tested on 10^8 of servers and 10^{10} client devices with thousands of packages speaking it (often called web browsers).

The elegance of using a HTTP request really lies in the easy way to write such requests. For instance, for SCS, it is always

```
http://<server>/<local-path>?<base-query>,
```

where `<base-query>` must contain RA, DEC and SR. Try it in your favourite browser:

http://dc.zah.uni-heidelberg.de/ppmxl/q/cone/scs.xml?RA=56.601&DEC=24.114&SR=1.5

Of course you are not limited to a web browser. For an easy start can use curl:

```
curl -o pleiades.vot
'http://dc.zah.uni-heidelberg.de/ppmxl/q/cone/scs.xml?RA=56.601&DEC=24.114&SR=1.5'
```

Before you start writing your own simple VO clients, though, we strongly suggest to look out what is out there already.

Later in this course you will learn how to use these protocols in Python using the pyVO library, which also helps you handling the resulting VOTable. Many VO clients are published as Free software, so you can easily reuse code written in your prefered language. If in doubt, contact us or other people from within the VO community. In general people are happy (and proud) if you use their code, and maybe you even can contribute to it!

Of course there is a bit more to say about the simple protocols, and there are even more parameters you can use for a query. We skip this here, because they are mainly interesting for people writing the clients and libraries. As long as you are a user, you would probably use them through menus or function calls ("APIs").

3 TAP and ADQL

A First Query

To follow the examples, start TOPCAT and select TAP in the VO menu.

At *Keywords*, type **gavo**. Wait until the results are filtered and select the entry *GAVO DC TAP*. Then click *Use Service*.

You already made use of the VOs Google-like service: the Registry. A rough introduction of the registry how you can use it for data discovery will be explained in chapter "Data Discovery". In the query pane, enter:

```
SELECT TOP 1 1+1 AS result FROM ivoa.obscore
```

and then click "Ok". This should give you a table with a single 2 in it. If that hasn't worked complain.

Note that in the top part of the dialog there is metadata on the tables exposed by the service (in particular, the names of the tables and the descriptions, units, etc., of the columns). Use that when you construct queries later.

There are other TAP clients than TOPCAT – after all, we're talking about a standard protocol. Another TAP client widely used is Aladin.

You can also use TAPHandle, which runs entirely in your browser.

For running a TAP client in scripts there is STILTS or PyVO

More TAP clients can be found on the IVOA applications page.

You can also use TAP from Python. A lot more on this later. If you are curious now, see an Ψipython notebook explaining the basics.

Why SQL?

The SELECT statement is written in ADQL, a dialect of SQL ("sequel"). Such queries make up quite a bit of the science within the VO.

SQL has been chosen as a base because

- Solid theory behind it (relational algebra)
- Lots of high-quality engines available
- Not Turing-complete, i.e., automated reasoning on "programs" is not very hard

Relational Algebra

At the basis of relational data bases is the relational algebra, an algebra on sets of tuples ("relations") plus six operators:

- unary *select* – select tuples matching to some condition
- unary *project* – make a set of sub-tuples of all tuples (i.e., have less columns)
- unary *rename* – change the name of a relation (this is a rather technical operation)
- binary *cartesian product* – the usual cartesian product, except that the tuples are concatenated rather than just put into a pair; this, of course, is not usually actually computed but rather used as a formal step.
- binary *union* – simple union of sets. This is only defined for "compatible" relations; the technical points don't matter here
- binary *set difference* as for union; you could have used intersection and complementing as well, but complementing is harder to specify in the context of relational algebra

Good News: You don't *need* to know any of this. But it's reassuring to know that there is a solid theory behind all of this.

SELECT for real

ADQL defines only one statement, the `SELECT` statement, which lets you write down expressions of relational algebra. Roughly, it looks like this:

`SELECT` [`TOP` *setLimit*] *selectList* `FROM` *fromClause*
[`WHERE` *conditions*] [`GROUP BY` *columns*] [`ORDER BY` *columns*]

In reality, there are yet a few more things you can write, but what's shown covers most things you'll want to do. The real magic is in *selectList*, *fromClause* (in particular), and *conditions*.

TOP

setLimit: an integer giving how many rows you want returned.

```
SELECT TOP 5 * FROM rave.main
```

```
SELECT TOP 10 * FROM rave.main
```

SELECT: ORDER BY

`ORDER BY` takes *columns*: a list of column names (or expressions), and you can add `ASC` (the default) or `DESC` (descending order):

```
SELECT TOP 5 *
FROM rave.main
ORDER BY rv
```

```
SELECT TOP 5 *
FROM rave.main
ORDER BY rv DESC
```

```
SELECT TOP 5 *
FROM rave.main
ORDER BY fiber_number, rv
```

Note that `SELECT *` (pulling all columns) is usually wasteful and you should do better from the next slide on.

Also note that ordering is outside of the relational model.

That sometimes matters because it may mess up query planning (a rearrangement of relational expressions done by the database engine to make them run faster); also, of course ordering has to look at everything in a table, which is a sure way to make things slow. So: if you use ORDER, make sure it is actually necessary and that you do it at the latest possible moment (i.e., when the result set hopefully already is small).

On the other hand, looking at extreme values is a good way to find odd, presumably bad cases. I severely doubt that RVs of 1000 km/s actually correspond to any physical reality for the sort of object RAVE looked at.

Exercise 5

Select the (rows of) the 20 brightest stars in the table `fk6.part1`.

SELECT: what?

The select list has column names or expressions involving columns.
SQL expressions are not very different from those of other programming languages.

```
SELECT TOP 10
  POWER(10, phot_g_mean_mag) AS rel_flux,
  SQRT(POWER(ra_error, 2)+POWER(dec_error, 2)) AS errTot
FROM gaia.dr3lite
```

The value literals are as usual:

- Only decimal integers are supported (no hex or such)
- Floating point values are written like 4.5e-8
- Strings use single quotes (`'abc'`). Double quotes mean something completely different for ADQL (they are "delimited identifiers", which we will briefly revisit below).

The usual arithmetic, comparison, and logical operators work as expected:

- $+$, $-$, *, /; as in C, there is no power operator in ADQL. Use the `POWER` function instead.
- = (*not* ==), <, >, <=, >=
- `AND`, `OR`, `NOT`
- String concatenation is done using the `||` operator. Strings also support `LIKE` that supports patterns. `%` is "zero or more arbitrary characters", _ "exactly one arbitrary character" (like `*` and `?` in shell patterns).

Here is a list of ADQL functions:

- Trigonometric functions, arguments/results in rad: `ACOS`, `ASIN`, `ATAN`, `ATAN2`, `COS`, `SIN`, `TAN`; atan2(y, x) returns the inverse tangent in the right quadrant and thus avoids the degeneracy of atan(y/x).
- Exponentiation and logarithms: `EXP`, `LOG` (natural logarithm), `LOG10`
- Truncating and rounding: `FLOOR(x)` (largest integer smaller than x), `CEILING(x)` (smallest integer larger than x), `ROUND(x)` (commercial rounding to the next integer), `ROUND(x, n)` (like the one-argument round, but round to n decimal places), `TRUNCATE(x)`, `TRUNCATE(x,n)` (like ROUND, but discard unwanted digits).
- Angle conversion: `DEGREES(rads)`, `RADIANS(degs)` (turn radians to degrees and vice versa)
- Random numbers: `RAND()` (return a random number between 0 and 1), `RAND(seed)` (as without arguments, but seed the the random number generator with an integer)
- Operator-like functions: `MOD(x,y)` (the remainder of x/y, i.e., x%y in C), `POWER(x,y)`
- `SQRT(x)` (shortcut for `POWER(x, 0.5)`)
- Misc: `ABS(x)` (absolute value), `PI()`

Note that all names in SQL (column names, table names, reserved words, etc) are case-insensitive (i.e., *VAR* and *var* denote the same thing). You can force case-sensitivity (and use SQL reserved words as identifiers) by putting the identifiers in double quotes. These are the delimited identifiers mentioned above, and they are a constant source of trouble. Only use double quotes if the data providers force you to because they chose flamboyant names (VizieR, regrettably, did). If you publish data yourself, just use C identifiers for your column names; the full rules for how delimited identifiers interact with normal ones are difficult and confusing.

Also note how I used AS to rename a column. You can use the names assigned in this way in, e.g., `ORDER BY`:

```
SELECT TOP 10
  gaia_edr3_id,
  SQRT(POWER(pmra, 2)+POWER(pmra, 2)) AS pmTot
FROM cns5.main

ORDER BY pmTot
```

Don't do that on large catalogues without a very good reason – even with the TOP 10, the database will have to compute pmTots for *all* items in the table and then sort by that, which will take a *long* time with, for instance, Gaia DR3's 1.8 billion rows.

To select all columns, use *

```
SELECT TOP 10 * FROM rave.main
```

In general, try to only select the columns you actually need; there is no point retrieving a hundered columns when five would do, and carrying all these superfluous columns around has a very real cost in terms of ease-of-use and resources (in particular when it comes to uploads).

TOPCAT makes picking the columns really easy: Control-click the columns you want in the Columns tab, and then use the "Cols" button above the the query input to insert their names.

Use `COUNT(*)` to figure out how many items there are.

```
SELECT count(*) AS numEntries FROM rave.main
```

`COUNT` is what's called an aggregate function in SQL: A function taking a set of values and returning a single value. The other aggregate functions in ADQL are (all these take an expression as argument; count is special with its asterisk):

- `MAX`, `MIN`
- `SUM`
- `AVG` (arithmetic mean)

Note that on most services, `COUNT(*)` is an expensive operation. If you just want to get an estimate of how many rows a table has, on many services a peek into the Table pane in TOPCAT when you have selected a table will tell you.

Exercise 6

Select the absolute magnitude and the common name for the 20 stars with the greatest visual magnitude in the table fk6.part1 (in case you don't remember: The absolute magnitude is $M = 5 + 5 \log \pi + m$ with the parallax in arcsec π and the apparent magnitude m (check the units!).

SELECT: WHERE clause

Behind the WHERE is a logical expression; these are similar to other languages as well, with boolean operators `AND`, `OR`, and `NOT`. To find bright stars (apparently) moving quickly towards or from us:

```
SELECT raveid FROM rave.main
WHERE
  jmag<10
  AND ABS(rv)>100
```

> **Exercise 7**
>
> As before, select the absolute magnitude and the common name for the 20 stars with the greatest visual magnitude, but this time from the table fk6.fk6join. This will fail for reasons that should tell you something about the value of Bayesian statistics. Make the query work.

Missing Data: NULLs

SQL has an explicit concept of missing data: The magic value NULL. It has some interesting properties:

```
SELECT count(*) FROM tap_schema.tables WHERE NULL=NULL
```

returns 0. So does

```
SELECT count(*) FROM tap_schema.tables WHERE NULL!=NULL
```

All comparisons with NULLs are false, which turns out to be the least horrible thing in the presence of NULLs.

To select rows for which a given piece of data is or is not NULL use the special construct `IS (NOT) NULL`.

Explicit NULL values are an important feature, because it is *extemely* common that tables in astronomy contain unknown values. Just think of protometry near the detection limit: An object that is detectable in one band might be too faint in another.

In the FORTRAN age, people put in sentinel values like -9999 in such cases, but that is a dangerous practice: if you forget about checking for them, these might enter actual calculations. Consider an average: it will be possibly dramatically wrong, but when you notice that, it may very well be far too late.

A related concept is the NaN (not a number) from IEEE floating point numbers. In VOTables, somewhat regrettably, there is no difference between NULLs and NaNs; libraries will turn NaNs into NULLs where possible (in Python, using masked arrays or Python's own NULL value, `None`).

There are semantic differences, though, which you will notice as long as you do ADQL queries, where NULL and NaN are different (although data providers should generally avoid ingesting NaNs). As an example, when you take the average of a column, a NaN in just a single row will make the entire average NaN. Against that, rows that have NULLs will simply be ignored for computing the average. But as for NULL, NaN $\neq$ NaN holds.

> **Exercise 8**
>
> How many objects in the Fifth Catalogue of Nearby Stars (`cns5.main` on the GAVO TAP server) are missing a radial velocity?

SELECT: Grouping

For histogram-like functionality, you can compute factor sets, i.e., subsets that have identical values for one or more columns, and you can compute aggregate functions for them.

```
SELECT
  COUNT(*) AS n,
  ROUND(mv) AS bin,
  AVG(color) AS colav
FROM dmubin.main
GROUP BY bin
ORDER BY bin
```

Note how the aggregate functions interact with grouping (they compute values for each group).

Also note the renaming using AS. You can do that for columns (so your expressions are more compact) as well as for tables (this becomes handy with joins).

To just figure out the domain of columns, there is a shortcut: `DISTINCT`.

Exercise 9

Get the averages for the total proper motion from lspm.main in bins of one mag in Jmag each. Let the output table contain the number of objects in each bin, too.

SELECT: Grouping by HEALPix

If you want to characterise some property over the sky, HEALPixes are your friend.

These are mathematical miracles: a tesselation of the sky with pixels of equal area. No more headaches at the poles! ADQL as such does not know about these, but a widely implemented extension function does: `ivo_healpix_index`.

While for large catalogues, such queries will have long runtimes, because they will always scan the whole table, you can try it for smallish catalogues even in a course situation.

To find out more about HEALPix, see corresponding interlude 4.

A common operation is trying some statistical qualification over the entire sky or a significant part of it. Since healpixes have equal areas and are well-beheaved at the poles and across the stitching line of a spherical coordinate system, they are particularly well suited for work like this. An introduction to this with sample queries is given on a poster by Mark Taylor. Not all services support the necessary functions (in TOPCAT, you can check in the "service" tab).

```
SELECT ivo_healpix_index(5, raj2000, dej2000) AS bin,
  COUNT(*) AS n,
  AVG(rv) AS meanrv,
  MAX(rv)-avg(rv) AS updev,
  AVG(rv)-min(rv) AS lowdev
FROM rave.main
WHERE e_rv<20
GROUP BY bin
HAVING COUNT(*)>5
```

Plot this in TOPCAT using the sky plot, see Layers / Add Healpix Control.

Use bin as HEALPix index, set the healpix level to 5, and the select what you want to see plotted. As annotation for healpix columns improves, plotting these things should involve less manual work.

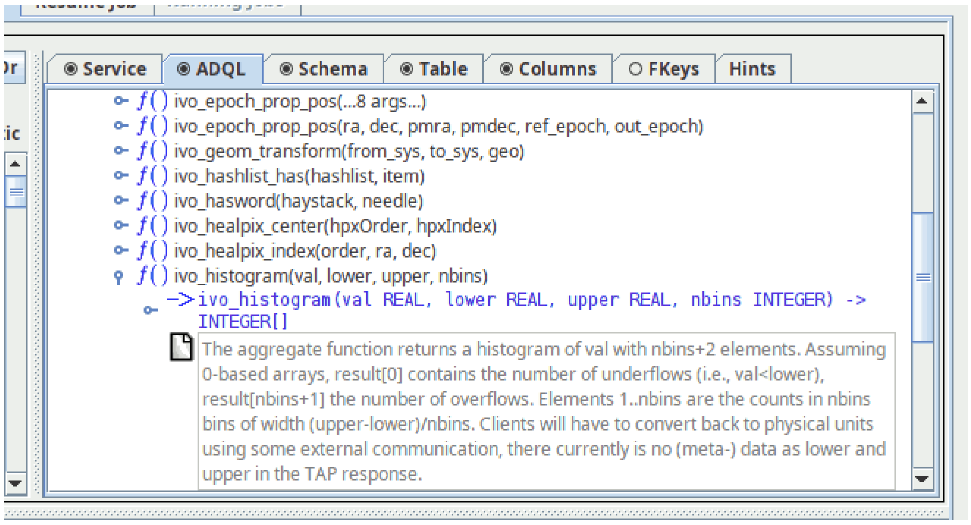
Exercise 10

Make an all-sky plot of the number of objects and their average effective temperature in HEALPixes of level 5 of the catalogue rave.main. Hint: In the server-provided Examples on the GAVO server, there is an example "Make a HEALPix Map of something" (in Local UDFs; if you don't see it, update your TOPCAT). Start from there. Can you understand the structures that you see?

ADQL User Defined Functions

`ivo_healpix_index` is an example of an ADQL extension mechanism: Operators can add *UDF*s. The purpose of this is to not overload ADQL with features that may only be relevant for a limited selection of services or even impossible with certain kinds of backends. In the example, to implement the HEALPix index computation, the database engine has to know about HEALPixes in the first place, which generally requires rather elaborate extensions. These may be entirely irrelevant for services that do not have have data in spherical coordinates.

See TOPCAT's ADQL TAP for the UDFs available on a service:



```
ƒ() ivo_epoch_prop_pos(...8 args...)
ƒ() ivo_epoch_prop_pos(ra, dec, pmra, pmdec, ref_epoch, out_epoch)
ƒ() ivo_geom_transform(from_sys, to_sys, geo)
ƒ() ivo_hashlist_has(hashlist, item)
ƒ() ivo_hasword(haystack, needle)
ƒ() ivo_healpix_center(hpxOrder, hpxIndex)
ƒ() ivo_healpix_index(order, ra, dec)
ƒ() ivo_histogram(val, lower, upper, nbins)
    ->ivo_histogram(val REAL, lower REAL, upper REAL, nbins INTEGER) ->
      INTEGER[]
```

The aggregate function returns a histogram of val with nbins+2 elements. Assuming 0-based arrays, result[0] contains the number of underflows (i.e., val<lower), result[nbins+1] the number of overflows. Elements 1..nbins are the counts in nbins bins of width (upper-lower)/nbins. Clients will have to convert back to physical units using some external communication, there currently is no (meta-) data as lower and upper in the TAP response.

In older TOPCAT's you will find a less elaborate listing of these functions in the Service tab.

UDFs prefixed with `ivo_` play a special role: These are guaranteed to have a common syntax and semantics across services – if they are available, that is. Read more about them in the (occasionally updated) Catalogue of User Defined Functions (Campillo and Demleitner, 2023).

SELECT: JOIN USING

The brainiest point in ADQL is the `FROM` clause. So far, we had a single table. Things get interesting when you add more tables: JOIN.

```
SELECT TOP 10 lat, long, flux
FROM lightmeter.measurements
JOIN lightmeter.stations
USING (stationid)
```

Check the tables in the Table Metadata shown by TOPCAT: flux is from measurements, lat and long from stations; both tables have a stationid column.

JOINing is Selecting from the Cartesian Product

`JOIN` is a combination of cartesian product and a select.

```
measurements JOIN stations USING (stationid)
```

yields the cartesian product of the measurement and stations tables but only retains the rows in which the stationid columns in both tables agree.

Note that while the stationid column we're joining on is in both tables but only occurs once in the joined table.

To understand the way joins work, consider the following simplified example, where we have two sets, A and B. Like database tables, they consist of tuples, and we will join them on the second column of A and the first column of B.

So, we first compute the cartesian product (which has six elements in this case). We will, however, only retain the rows in the result that have identical elements in the join column (highlighted here in red). That yields the rows marked in green – well, except that only one copy of the joined column is retained in a database; anything else would be a pointless waste of resources.

$A = \{(a, 1), (b, 2), (b, 3)\}$
$B = \{(1, u), (2, v)\}$
$A \times B =$

(a,	1,	1,	u)
(a,	1,	2,	v)
(b,	2,	1,	u)
(b,	2,	2,	v)
(b,	3,	1,	u)
(b,	3,	2,	v)

SELECT: JOIN ON

If your join criteria are more complex than simple equality, you can join `ON`.

```
SELECT dateobs as lswdate, t_min as appdate
FROM lsw.plates AS a
LEFT OUTER JOIN applause.main AS b
ON (dateobs BETWEEN t_min AND t_max)
WHERE dateobs BETWEEN 36050 and 36100
```

This particular query compares two archives of scanned plates, lsw.plates (from the K"onigstuhl observatories) and applause.main (from various other German observatories) and sees if lsw.plate's observation date (dateobs) is within the exposure time of the other's (which is between t_min and t_max).

The LEFT OUTER JOIN makes it so that every match on the lsw.plates side is retained. Where there is a simultaneous observation in Applause, the second column will have its MJD. Where there is no match, that second column will be NULL.

Of course, I have picked a WHERE clause for didactic reasons. If you drop it, you will get a large table with only very few matches in between (and you may need to go async; see below).

Flavours of JOIN

There are various kinds of joins, depending on what elements of the cartesian product are being retained in the presence of missing data (NULL).

First note that in a normal join, rows from either table that have no "match" in the other table get dropped. Since that's not always what you want, there are join variants that let you keep certain rows. In short (you'll probably have to read up on this):

- t1 `INNER JOIN` t2 (`INNER` is the default and is usually omitted): Keep all elements in the cartesian product that satisfy the join condition.
- t1 `LEFT OUTER JOIN` t2: as `INNER`, but in addition for all rows of *t1* that would vanish in the result (i.e., that have no match in *t2*) add a result row consisting of the row in *t1* with NULL values where the row from *t2* would be.
- t1 `RIGHT OUTER JOIN` t2: as `LEFT OUTER`, but this time all rows from *t2* are retained.
- t1 `FULL OUTER JOIN` t2: as `LEFT OUTER` and `RIGHT OUTER` performed in sequence.

Geometries

The main extension of ADQL wrt SQL is addition of geometric functions. Unfortunately, these were not particularly well designed, but if you don't expect too much, they'll do their job.

```
SELECT TOP 500 rv, e_rv, p.radial_velocity,
 p.ra, p.dec, p.pmra, p.pmdec
FROM gaia.dr3lite AS p
JOIN rave.main AS rave
ON 1=CONTAINS(
 POINT(p.ra, p.dec),
 CIRCLE(rave.raj2000, rave.dej2000, 1.5/3600.))
```

For historical reasons some geometrical functions accept an optional string value as the first argument e.g.

`POINT('ICRS',p.raj2000,p.dej2000)`

As of ADQL 2.1 this option is marked as deprecated. Many services still only support ADQL 2.0 and hence require this argument.

There are more geometry functions defined in ADQL:

`AREA, BOX, CENTROID, CIRCLE, CONTAINS, COORD1, COORD2, COORDSYS, DISTANCE, INTERSECTS, POINT, POLYGON`

Exercise 11

Look at the documentation of the `ivo_epoch_prop_pos` UDF (refer back to the UDF slide if necessary). Can you figure out how to propagate (i.e., apply the proper motions to compute positions in the future) the CNS5 to the year 2150? The positions in the CNS5 are (somewhat unusally) given for what is in the column `epoch`.

What's the RA of Sirius you determine in this way? And why will this be probably a rather poor guess?

Exercise 12

Compare the radial velocities given by the rave.main and arihip.main catalogues, together with the respective identifiers (hipno for arihip, raveid for rave). Use the POINT and CIRCLE functions to perform this positional crossmatch with, say, a couple of arcsecs.

DISTANCE

ADQL has a `DISTANCE` function to compute the spherical distance between two points:

```
DISTANCE(lon1, lat1, lon2, lat2)
```

You can also use distance with the POINT geometry, like this:

`DISTANCE(POINT (lon1, lat1), POINT (lon2, lat2) )`

– but this probably only makes sense if you have native POINT-s in a table.

The `DISTANCE` function can be used to make cone selections and is the prefered way to perform crossmatches on sky positions in ADQL 2.1.

```
SELECT TOP 1000
  raj2000, dej2000, parallax
  FROM arihip.main
  WHERE
    DISTANCE(raj2000, dej2000,
             189.2, 62.21) < 10
```

Note that there are still many TAP services out there that do not support DISTANCE or become very slow when you use it. You can always fall back to the CONTAINS/CIRCLE pattern introduced above in such cases.

Subqueries

One of the more powerful features of SQL is that you can have subqueries instead of tables within FROM. Just put them in parentheses and give them a name using AS. This is particularly convenient when you first want to try some query on a subset of a big table:

```
SELECT COUNT(*) AS n, ROUND((u-z)*2) AS bin
FROM (
  SELECT TOP 4000 * FROM sdssdr16.main) AS q
GROUP BY bin ORDER BY bin
```

Another use of subqueries is in the connection with EXISTS, which is an operator on queries that's true when a query result is not empty.

Beware – people coming from other languages have a tendency to use EXISTS when they should be using JOIN (which typically is easier to optimise for the database engine). On the other hand, EXISTS frequently is the simpler and more robust solution.

As an example, to get arihip stars that happen to be in RAVE DR5, you could write both

```
SELECT TOP 10 *
FROM arihip.main AS a
WHERE
  EXISTS (
    SELECT 1
    FROM rave.main AS r
    WHERE DISTANCE(
      r.raj2000, r.dej2000,
      a.raj2000, a.dej2000) < 1/3600.)
```

or

```
SELECT TOP 10 a.*
FROM arihip.main AS a
JOIN rave.main AS r
ON DISTANCE(
      a.raj2000, a.dej2000,
      r.raj2000, r.dej2000) < 1/3600.
```

(but see the exercise 13 before making a pattern out of this).

Exercise 13

Sit back for a minute and think whether the JOIN and the EXIST solution in the *Subqueries* chapter are actually equivalent. You are not supposed to see this from staring at the queries – but comparing the results from the two queries ought to give you a hint; retrieve a few more objects if your results happen to be identical.

Common table expressions

`WITH` lets you name a subquery result for later use in your main query.

This usually makes for more readable queries – the top-level operation is easily findable at the end of the query, and if you are curious what the individual contributions are, you can go back to the proper with clause. Consider this example where we are downloading low-resolution spectra exclusively for objects for which we have rave data:

```
WITH withrvs AS (SELECT TOP 200
  ra, dec, source_id,
  a.radial_velocity, b.rv as raverv
  FROM gaia.dr3lite AS a
  JOIN rave.main AS b
  ON (
    DISTANCE(a.ra, a.dec,
      b.raj2000, b.dej2000) < 1/3600.))
SELECT *
FROM gdr3spec.spectra
JOIN withrvs
USING (source_id)
```

This particular example also illustrates a technique WITH is being used for as well: planner barriers in case of catastrophic query plans.

Each ADQL query will be translated in a sequence of steps the database will process in order to perform the whole query. This query plan may switch the order of steps which were defined in the scripts to enhance the performance. The query planner bases this plan on estimates of table sizes and the "selectivities" of predicates (basically: how often they will be true). If they get these estimates wrong, the query plans can be wrong, too, sometimes catastrophically so. In these cases, forcing the planner using CTEs may save the day.

In our example, we crossmatch Gaia and Rave and pull radial velocities from both. Then we want to add BP/RP spectra (which here come in arrays) with a simple join on the Gaia source id; since at least in 2022, the backend database gets the estimate of the selectivity of the distance condition grossly wrong, without the CTE the database would first match the 200 million rows of of the Gaia spectra to the Gaia catalogue before turning to the half a million rave rows, turning a reasonably fast query into a matter of hours.

TAP: Uploads

TAP lets you upload your own tables into the server for the duration of the query.

Note that not all servers already support uploads. If one doesn't, politely ask the operators for it.

Example: Add proper motions to an object catalogue giving positions reasonably close to ICRS; grab some table, falling back to the attached Ψex.vot, load it into TOPCAT, go to the TAP window and there say:

```
SELECT mine.*, refcat.pmra, refcat.pmde FROM
  gaia.dr3lite AS refcat
  JOIN tap_upload.t1 AS mine
  ON DISTANCE (
    refcat.ra, refcat.dec,
    mine.raj2000, mine.dej2000) < 0.001
```

You must replace the 1 in *tap_upload.t1* with the index of the table you want to match.

You may also need to adjust the column names of RA and Dec for your table, and the match radius.

Always take into account that positions in you upload table use the same coordinate system as the remote table, and also pay attention to the epoch.

Exercise 14

If you have some data with celestial positions of your own, try reading it into TOPCAT and try the crossmatch with that. If you do not have any suitable data, try the ex.vot from the *TAP: Uploads* slide.

Almost real world

Just so you get an idea how SQL expressions can evolve to span several pages:

Suppose you have a catalogue giving alpha, delta, and an epoch of observation sufficiently far away from the Gaia epoch. To match it, you have to bring the reference catalogue on our side to the epoch of your observation. For larger reference catalogues, that would be quite an expensive endeavour. Thus, it's usually better to just transform a smaller selection of candidate stars.

To do this, you decide how far one of your stars can have moved (in the example below 0.1 degrees, the inner crossmatch), and you generate a crossmatch there. From that crossmatch, you select the rows for which the transformed coordinates match to the precision you want.

To play this through, load matchme.vot from the HTML or PDF attachment into TOPCAT. The rough crossmatch with Gaia is standard fare:

```
SELECT
  alpha, delta, epoch,
  source_id, ra, dec, pmra, pmdec
FROM tap_upload.t1
  JOIN gaia.dr3lite
  ON distance(alpha, delta, ra, dec)<0.1
```

That is returning some 10000 pairs, almost all of which are wrong (there are certainly fewer than 55 true matches, as there are just 54 rows in matchme). We will thus have to filter more strictly constraining the positions. For that, we have to apply proper motions.

There is nothing in ADQL's core that can do that. For the small distances we are talking about here, you could write something like

```
    ra+pmra/cos(radians(dec))*(epoch-2016)
      AS palpha,
    dec+pmde*(epoch-2016) AS pdelta,
```

as a workable approximation.

More and more TAP services, however, have an ADQL extension function (UDF; see TOPCAT's "Service" tab for a per-service list of those) *ivo_epoch_prop_pos* that will do a precise job. We will use it here:

```
SELECT alpha, delta, parallax, pmra, pmdec, source_id
FROM (
SELECT
  alpha, delta, parallax, pmra, pmdec, source_id,
  ivo_epoch_prop_pos(ra, dec, parallax,
    pmra, pmdec, radial_velocity, 2016, epoch) AS tpos
FROM tap_upload.t1
  JOIN gaia.dr3lite
  ON DISTANCE(alpha, delta, ra, dec)<0.1) AS q
WHERE DISTANCE(POINT(alpha, delta), tpos)<2/3600.
```

(don't forget to adapt the table name behind tap_upload!).

If you've tried it, you'll have noticed that 53 rows were returned for 54 input rows. For "real" data you'd of course not have this; there'd be objects not matching at all and probably objects matching multiple objects. The reason this worked so nicely in this case is that the sample data is artificial: I made that up using ADQL, too. The statement was:

```
SELECT coord1(tpos) alpha, coord2(tpos) AS delta, epoch FROM (
  SELECT
    ivo_epoch_prop_pos(ra, dec, parallax,
      pmra, pmdec, radial_velocity, 2016, epoch) AS tpos,
    epoch
  FROM ( SELECT d3l.*, 1900+75*rand() AS epoch
    FROM gaia.dr3lite AS d3l tablesample(1)
    WHERE
      POWER(pmra,2)+POWER(pmdec,2)>500*500) AS gs) AS transgs
```

This is rather subquery-heavy and in addition uses two features that we have not seen yet. For one, `rand()` returns a random number between 0 and 1, which we use here to generate a random source epoch.

And there is TABLESAMPLE; this is a prototype extension that may go into ADQL 2.2, perhaps somewhat modified. As used here, you pass in how many percent of the table you want to look at. Over a

`TOP 100` or so, this has the advantage that you get different rows every time you use it. It's not some statistically valid sampling, though.

The handcrafted VOTable for the example is attached as Ψmatchme.vot.

Exercise 15

Follow the example on the "Almost Real World" slide with the matchme.vot table provided there.

Despite the artificial setting, we have lost one object in the upload join. Can you find it? And can you guess why we have lost it?

Hint: Have a look at TOPCAT's Pair Match facility, paying attention to the Join Type setting.

Exercise 16

In the last exercise, we met the star with the Gaia source id 1872046574983497216 and a total proper motion of 5 arcsec/yr. In the solution I claimed this is a really extreme case. Well: how extreme is it? Can you estimate how many faster stars there are?

(Please resist the temptation to use the full Gaia catalogue for this purpose; see also the next exercise).

Exercise 17

(This is slightly advanced) In the last exercise, you were asked not to consult the Gaia source catalogue to get proper motion statistics, although to a contemporary astronomer that would be the obvious choice. That is because all-catalogue statistics are expensive on Gaia.

Can you find a way to still get the fastest stars in `gaia.dr3lite` within the time limit of sync queries on that server (i.e., a couple of seconds)?

Cheap hint: see what columns are indexed.

TAP: Async operation

TAP jobs can take hours or days. To support that, you can run your TAP jobs asynchronously. This means you do not have to keep a connection open all the time.

Most servers have relatively tight limits on the execution times of queries when they are run synchronously. For instance, on the GAVO DC TAP service, the following query will probably time out (remember that the result of this query is available in TOPCAT's table tab, too, but it's a simple query that demonstrates timeouts):

```
select count(*)
from ucac4.main
```

(if this doesn't time out, the machine has a good day; use another slow query in that case).

Async queries can also be queued (i.e., put into a waiting state until the executing machines have resources free), and hence it is much easier to be generous with execution limits in async, too.

To go async in TOPCAT, change the *Mode* selector to "*Asynchronous*". After submitting the job, you can watch your job go through "*UWS* phases":

PENDING Job created, you can configure it Configuration includes setting the query, adding uploads, setting execution limits, etc.

QUEUED Waiting for compute time

EXECUTING The job is running

COMPLETED Successful completion, fetch results

ERROR The Job has failed, fetch error message

Resuming async Jobs

You can quit your client with async and resume from somewhere else.
To do that: In *Running Jobs*, select the URL and save it. Uncheck *Delete on Exit* and leave TOPCAT.
Then restart TOPCAT, open the TAP window and paste the URL back into the URL field. If the job has finished, you can retrieve the result.
There is a bit more to async operation; for example, the server will not keep your jobs indefinitely (see "destruction time" in the resume tab). TAP lets you change these values, though TOPCAT doesn't offer an interface to that as of now. PyVO (for instance) does, and so does stilts, the command line variant of TOPCAT.

Exercise 18

In async mode, run this on the GAVO server:

```
SELECT TOP 500 source_id, flux
FROM gdr3spec.spectra
WHERE arr_max(flux)>arr_avg(flux)*5
```

This is using the experimental array extension to ADQL[1]. You can probably guess without reading the blog post that this will select spectra with something like strong lines.
Run that query in async mode on the GAVO server. In a course situation, shout out your job's phases to watch the dequeuing. Save the job URL, exit TOPCAT, resume it, and load the result when the job is COMPLETE-d.

TAP: the TAP schema

TAP services try to be self-describing about what data they contain. They provide information on what tables they contain in special tables in *TAP_SCHEMA*. Figure out what tables are in there by querying *TAP_SCHEMA* itself:

```
SELECT * FROM tap_schema.tables
WHERE table_name LIKE 'tap_schema.%'
```

Of the tables you get there, the most relevant ones are *tap_schema.tables* and *tap_schema.columns*. From the former, you can obtain names and descriptions of tables, from the latter, about the same for columns.
To see what columns there are in *tap_schema.columns*, say:

```
SELECT * FROM tap_schema.columns
WHERE table_name='tap_schema.columns'
```

[1] https://blog.g-vo.org/a-proposed-vector-extension-for-adql.html

Of course, in normal operations, clients like TOPCAT do that querying for you: it's how they fill their metadata views.

In addition to *description*, *unit*, *datatype* and *arraysize* (the latter two corresponding to what you have in VOTable), there is the *indexed* column that says whether the column is part of an index. While that information is, in general, not enough to be sure, on large tables querying against indexed columns can steer you clear of the dreaded "sequential scan", which is when the database engine has to go through all rows (which is slow and may take hours for really large tables).

The ucd column is also interesting. See the sidetrack on UCDs (appendix D) for details on these.

Exercise 19

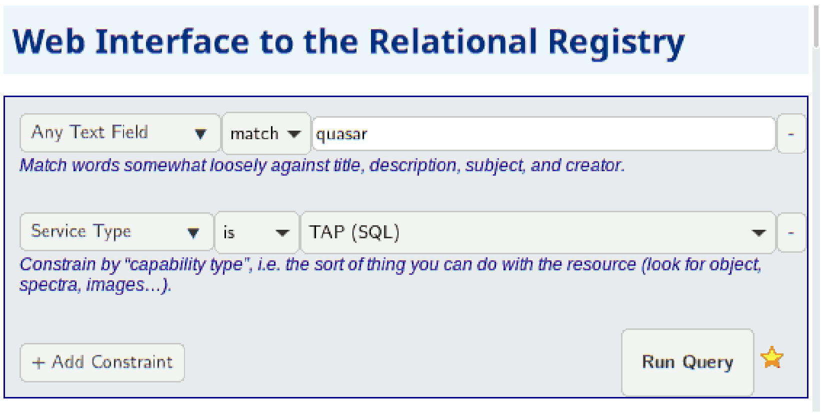
Pick a server that piques your interest from TOPCAT's server selection. How many tables are there on the server? How many columns? How many columns with UCDs starting with phot.mag?

Data Discovery 1: the Registry

The list of services in TOPCAT's TAP window comes from the VO Registry, an inventory of the services and data kept within the VO.

There are a few more ways to search the Registry, for instance in a web browser using WIRR.

Use case: Find tables talking about quasars that have redshifts.


Web Interface to the Relational Registry

Any Text Field ▾ match ▾ quasar -
Match words somewhat loosely against title, description, subject, and creator.

Service Type ▾ is ▾ TAP (SQL) ▾ -
Constrain by "capability type", i.e. the sort of thing you can do with the resource (look for object, spectra, images…).

+ Add Constraint Run Query

WIRR is not limited to search TAP services only, but also services using other VO protocols like SIAP or SCS.

In WIRR, you add and define constraints on the data collections.

Any Text Field - match - **quasar**

then click + Add Constraint and in the new row select

Service Type - is - TAP(SQL) ,

again click + Add Constraint and in the new row select Blind Discovery → Column UCD. You will then get a Pick one button. Try it to locate a redshift UCD.

What you get back is a list of data collections ("resources") that match your criteria. In principle, you could transmit these to TOPCAT using SAMP, and that works fine for SCS, SSAP, and SIAP services. For TAP services, this does not work yet (2024) for complicated reasons not easy to fix.

Data Discovery 2: use ADQL

The relational registry standard says how to query this data set using ADQL. All tables are in the rr schema and can be combined through NATURAL JOIN. Our use case looks like this in ADQL:

```
SELECT ivoid, access_url, name,
  ucd, column_description
FROM rr.capability
  NATURAL JOIN rr.interface
  NATURAL JOIN rr.table_column
  NATURAL JOIN rr.res_table
WHERE standard_id LIKE 'ivo://ivoa.net/std/tap%'
  AND 1=ivo_hasword(table_description, 'quasar')
  AND ucd='src.redshift'
```

As in WIRR, we constrain the UCD find columns with redshifts. It is instructive to compare the query above with the following one:

```
SELECT ivoid, access_url, name, ucd, column_description
FROM rr.capability
  NATURAL JOIN rr.interface
  NATURAL JOIN rr.table_column
  NATURAL JOIN rr.res_table
WHERE standard_id LIKE 'ivo://ivoa.net/std/tap%'
  AND 1=ivo_hasword(table_description, 'quasar')
  AND 1=ivo_hasword(column_description, 'redshift')
```

– the difference here is that we don't use the controlled UCD vocabulary but do a freetext query similar to the query we performed with WIRR. You notice that precision is down (you get many columns that talk *about* redshifts, for instance), but recall is up (for instance, our naive query was missing columns with UCDs *src.redshift;pos.heliocentric*, but, worse, some with empty UCDs).

To find UCDs relevant for you used "in the wild", you can use WIRR's Pick one button as above, or you can do a query like

```
SELECT ucd, MIN(column_description), MAX(column_description)
FROM rr.table_column
WHERE 1=ivo_hasword('redshift', column_description)
GROUP by ucd
```

The `MIN` and `MAX` clauses sample a few of the descriptions collected into each UCD's group. In this example, this is admittedly not very illuminating. It might be for other cases.

As to columns with missing UCDs, the recommended remedy is to complain to data providers that have lousy metadata, and make sure metadata is good on data that you publish yourself. High-quality metadata is of utmost importance for the VO – but on the other hand: Even shoddily published data is better than entirely unpublished data.

Incidentally, if you are in the business of writing RR queries yourself, be sure to look at the sample queries in the RegTAP standard.

Simbad

Simbad has a TAP interface; find it TOPCAT's server selector and inspect Simbad's table metadata. Try queries like:

```
SELECT TOP 20 * FROM basic
```

Example: Filter out boring stars. To get a sample, use your own data if you have some. Otherwise, let's use some HIPPARCOS stars. In TOPCAT, do VO/Cone Search, enter hipparcos as keyword, use the Hipparcos Main Catalogue resource and search with, say, RA 30, Dec 12, and Radius 10.

With that table open and Simbad's *public.basic* metadata in the TAP window, do Examples/Upload Join. Edit the resulting query to end up like

```
SELECT TOP 1000
   otype_txt, tc.*
   FROM basic AS db
   JOIN TAP_UPLOAD.t7 AS tc
   ON 1=CONTAINS(POINT('ICRS', db.ra, db.dec),
                 CIRCLE('ICRS', tc.ra, tc.dec, 2./3600.))
   WHERE otype_txt!='star'
```

Whatever is left either is so boring that nobody ever bothered to publish about it – or it is something *except* a boring, plain star.

For otypes, simbad has a fairly elaborate classification system that you will need to know to make useful queries against otype. Another secret they are not advertising loudly enough at the moment is that you can append two dots to an object designation to query against "thing and descendants", as in `otype='V*..'` to catch all variable stars.

Exercise 20

In exercise 18, you selected stars with odd spectra. Can you use Simbad's TAP service to find what types of star these are?

Hint: you probably need to do two upload joins, first with gaia.dr3lite (or some other Gaia DR3 table out there), then with public.basic on Simbad.

Onward

If you get stuck or a query runs forever, the operators are usually happy to help you. To find out who could be there to help you, check TOPCAT's Service tab or use – the relational registry. If you have the ivoid of the service, say

```
SELECT role_name, email, base_role
FROM rr.res_role
WHERE ivoid='ivo://org.gavo.dc/tap'
```

– if all you have is the access URL, do a natural join with interfaces.

If we have done a good job, you now know how…

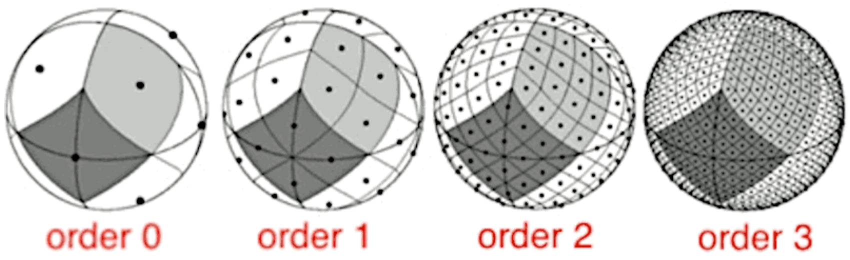
4 Interlude: HEALPix, MOC, HiPS

What are HEALPixes?

Spherical geometry is *hard*. For instance, at the poles about anything goes haywire, and there is the "stitching line" between 0 and 360 degrees.. It helps when you have numbered pixels rather polar coordinates. HEALPix is a magic scheme for that:

- Hierarchical – there are 12 pixels at level 0, and $12 \cdot 4^n$ pixels at level n where pixels at higher levels are always true subsets of pixels in lower levels: All HEALPixes make up a tree
- Equal Area – at a given level, each pixel has the same area
- isoLatitude – distinct latitudes of pixel centers go with $O(n)$ rather than $O(n^2)$ with the order
- Pixelization – mapping $(\alpha, \delta) \rightarrow [0, \ldots, N]$.

Take-away Concepts on HEALPix



The linear dimension of a HEALPix is $\sim 1^\circ$ at order 6; it changes by a factor of two on each level.
Extra trick: NEST numbering of the pixels lets you go between levels by integer division or multiplication by 4.

HEALPix in ADQL

The VO's query languge ADQL does not support HEALPix natively.
But on many TAP services there are standard extensions ("UDFs") for dealing with them:

```
ivo_healpix_center(
  hpxOrder INTEGER, hpxIndex BIGINT) -> POINT
```

and

```
ivo_healpix_index(order INTEGER,
  ra DOUBLE PRECISION, dec DOUBLE PRECISION
  ) -> BIGINT
```

To find out whether your TAP service has them, inspect the TAP capabilities; in TOPCAT, you will find the list of UDFs in the *ADQL* tab.

Application: HEALPix Maps

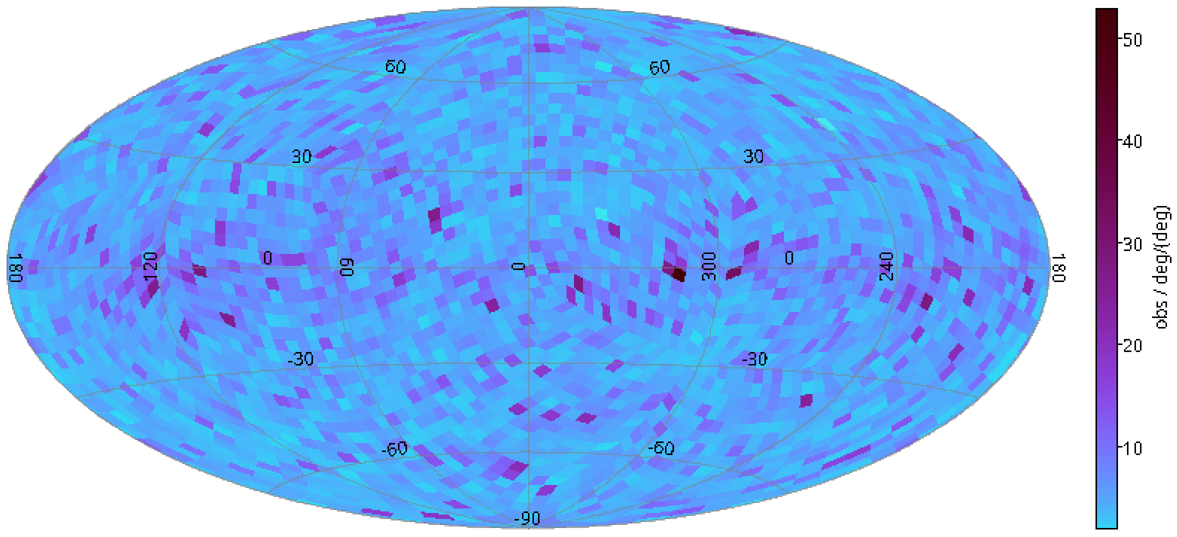
A prime application of these functions is the creation of HEALpix maps, which allow one to get a quick idea of distributions of all kinds of things in catalogues.

On the GAVO DC TAP at http://dc.g-vo.org/tap service, there is a service-provided example for producing HEALPix maps:

```
SELECT
  MAX(parallax)/AVG(parallax) AS obs,
  ivo_healpix_index(4, ra, dec) AS hpx
FROM hipparcos.main
GROUP BY hpx
```

In case you're wondering: No, I don't think there is any physical significance to the ratio of the parallax of the closest star to the average parallax in a neighbourhood. This is just to show that you can use interesting expressions here, not just `COUNT(*)`.

In TOPCAT, you can plot this by using *Sky Plot* and then *Layers* → *HEALPix control*. With a few extra adjustments, this will yield something like this:



In Gaia

HEALPixes are so nifty that Gaia uses them for its source ids. To get the HEALPix of a Gaia object at level n, compute

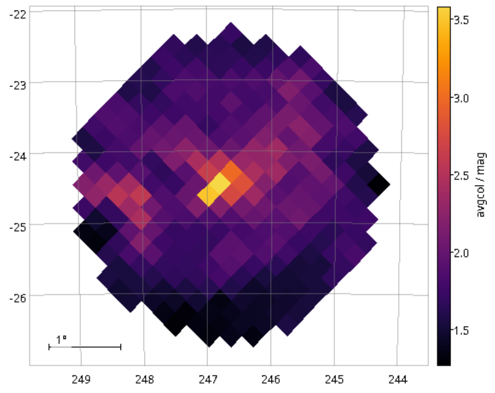
$$hpx = \frac{source\_id}{4^{12-n} \cdot 2^{35}}.$$

This only works down to level 12, which is the healpix level used to determine the Gaia source ids. According to our rule of thumb on the linear dimension at level 6, the linear dimension of that would be a degree divided by 2^6 or about an arcminute.

Use that to compute a rough dust map:

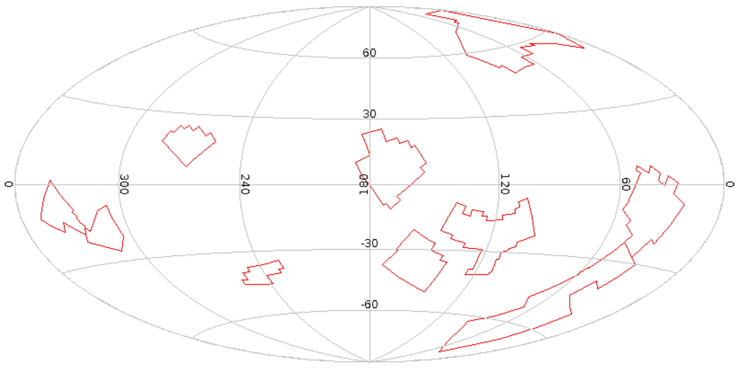
```
SELECT source_id/8796093022208 AS pix,
  AVG(phot_bp_mean_mag-phot_rp_mean_mag) AS avgcol
FROM gaia.edr3lite
WHERE DISTANCE(ra, dec, 246.7, -24.5)<2
GROUP BY pix
```

That’s level 8 two degrees around the center of a really close dust cloud in Monoceros and plots as:



Polygon union Polygon

Have you ever tried to compute the union or intersection of two spherical polygons?
It’s a nightmare. Not to mention the result is not a polygon any more:



In case you are curious: something like this is what `select top 10 * from cstl.geo` produces.
MOCs to the rescue!

MOC?

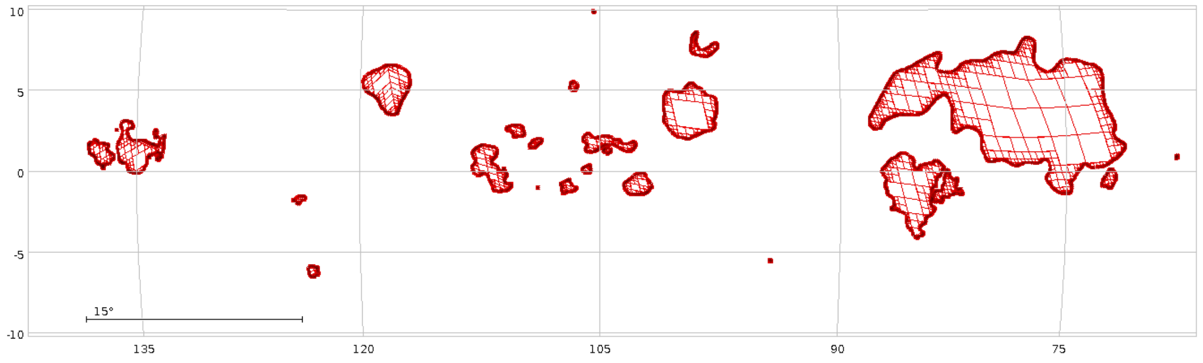
You can represent arbitrary shapes to high precision (order 29 is 0.4 mas) as lists of HEALPix indexes.
Alas, you need about 10 million such pixels for a shape of 1 deg^2.
Solution: Abbreviate ranges and use lower-order indexes when the pixels are full.

That’s a Multi Order Coverage map, or MOC in short.

MOC examples

```
select * from openngc.shapes
```

When plotting this, consider that the HEALpix lists this time are in columns. Hence, you will need an Area control.



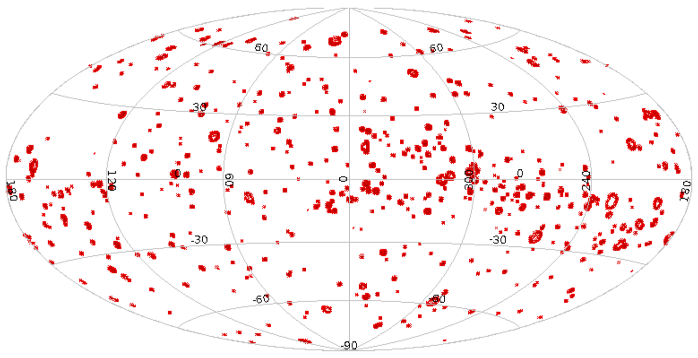
Such a shape may be written like

```
11/34094023 12/136376116-136376117
```

– all the shapes together are less than half an MB.

Math with MOCs

Most operations really become simple with MOCs. For instance, the area on the sky within magnitude-dependent circles around Hipparcos stars brighter than 4 mag:



That's the result of the following ADQL query; note, however, that MOC support in ADQL is highly experimental at this time and unavailable in most TAP servers.

```
SELECT SUM(MOC(8, CIRCLE(ra, dec, 0.5*(4-vmag)))) AS contaminated
FROM hipparcos.main
WHERE vmag<4
```

That's *one* shape you can manipulate as such.

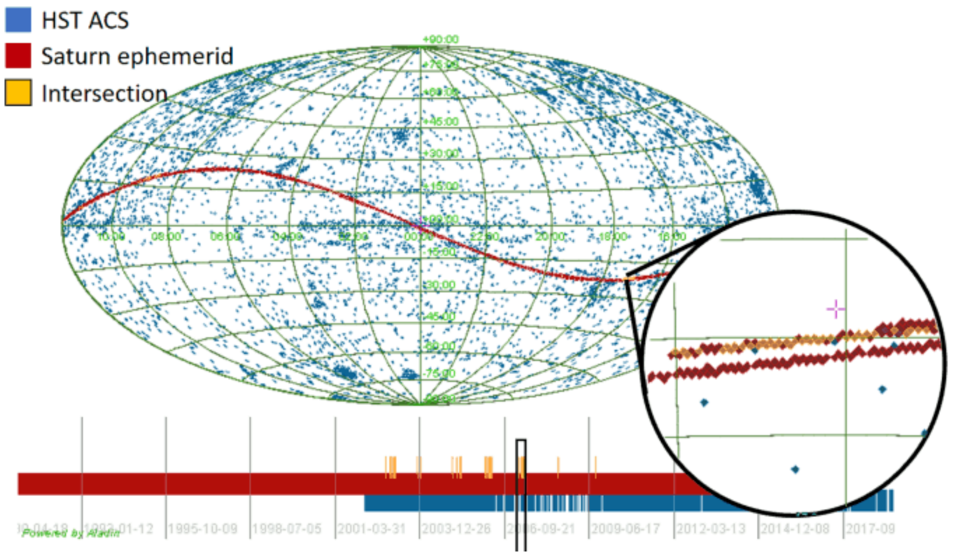
Exercise 21

Plot the total coverage of the Lockman Hole Radio Survey in the table emi.main on the GAVO TAP server as a level 8-MOC in TOPCAT. What is its area? Hint: you will probably need a subquery.

For aesthetic reasons, also try this at levels 12 and 18.

TMOCs, STMOCs

Recently, people have extended the scheme to time and correlated space-time. That's cool if you want to find data on fast-moving objects:



You can use and create these with Aladin, for instance. Outside of this, tools largely still need updates. In particular, there is no ADQL support for them at all, and one would need to think quite a bit how that would look like.

Mapping HEALPix to Anything: HiPS

HEALPixes are also behind HiPS, the Hierarchical Progressive Survey.

This is basically a set of maps

$$hpx_n \to \text{Image, Catalogue, } \ldots$$

on a number of HEALPix orders n.

This is what lets you nicely zoom in and out of image surveys and catalogues in Aladin.

You can make HiPSes yourself if you have data with high spatial dynamics.

This is not even limited to equatorial coordinates if you are willing to cut the tools a bit of slack, as exhibited by the example of a 2D classifier output for galaxy morphology[2].

5 pyVO Basics

Prerequisites

- python and astropy, of course (we assume Debian stable, at least; anaconda on proprietary systems should do, too)
- TOPCAT[3] for viewing and visualising tables
- Aladin[4] to work with images

[2]https://pretalx.com/media/adass2023/submissions/JUTWAY/resources/polsterer_ADASS2023_Lo0TZRW.pdf

[3]http://www.star.bris.ac.uk/~mbt/topcat/

[4]http://aladin.u-strasbg.fr/aladin.gml

- pyVO. Get it from
  - https://pypi.python.org/pypi/pyvo
  - or try `apt-get install python3-pyvo`
  - or try `pip install pyvo`
  - or try `conda install pyvo`

Python Matters

In this course, we will use python scripts most of the time rather than the jupyter notebooks you may be more familiar with.

This is partly personal preference, but for "production" scripts have several important advantages:

- Meaningful version control
- Can use proper editors
- Files can work as modules

However, if you prefer notebooks, you can use pyVO from Python notebooks, too. If you are unsure how this looks like, see the attached tap-obscore.ipynb (which covers several of the topics we will later discuss).

Ψ tap-obscore.ipynb

To fit things on slides, I am PEP 8-relaxed. PEP 8[5] is a set of relatively sensible rules for how you should format your Python source code so other people want to read it. I am not always following it here. In particular, on slides, I am using indents of two spaces against the PEP 8 standard of four, which you may need to fix when cutting and pasting.

What's pyVO?

pyVO provides APIs for lots of VO protocols.

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

It is a community project. You are most welcome to contribute at https://github.com/astropy/pyvo.

Running Simple Services

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

[5] https://peps.python.org/pep-0008/

```
# <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 will soon learn how to find out the access URLs.

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((340.1,3.36), size=(0.1, 0.1))
image=images[0]
image.cachedataset()
```

Ψ 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[6].

Also: `row.cachedataset` saves the image to your local disk under a name sensible for the metadata. In case the filename produced by cachedataset has an extension `.None` on your machine: that's a bug in pyVO that was fixed in 2024.

Note how you do not 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 a more interactive setting (e.g. ipython).

Getting source code: In case your PDF viewer gives you a hard time saving the attached Python code or does not support attachments at all, you can find all the files in this course's repository. Just run

```
git clone https://codeberg.org/msdemlei/pyvo-course
```

[6]http://pyvo.readthedocs.io/en/latest/api/pyvo.dal.SIAService.html#pyvo.dal.SIAService.search

Exercise 22

Get our example basicsiap.py from the notes.

Now find an image service publishing the ROSAT survey and pointed observations and see if it has an image for the position given (or try some other service and position you are actually interested in).

Use WIRR[7] to search the VO Registry for now.

What is coming back from SIAService's `search` is a sequence of SIARecords. Have a quick look at its pyvo documentation[8] and make your program print the file size and the instrument name rather than calling `cachedataset`.

This is Python

The advantage of doing this in Python is that it is easy to add your own logic. Here is how to add time constraints (SIAP version 1 unfortunately does not specify how to tell the service you are only interested in a specific time interval – we will 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 [
    (213.97, 11.50),
    (230.44, 52.92)]:
  images = svc.search(pos, size=(0.5, 0.5))
  for row in images:
    if not DATE_MIN<row.dateobs<DATE_MAX:
      continue
    row.cachedataset()
```

Ψ multisiap.py

A word on `row.dateobs`: While SIAP (as most of the VO) delivers dates as modified julian dates (MJD), pyVO turns these values into `astropy.time.Time` instances. You could turn these back into floats (my taking their `.mjd.real` attribute) and compute with MJD yourself, but it is smarter to keep your times in `Time` instances, too, as shown in the multisiap.py.

Metadata in pyVO

You can access the metadata coming with the response VOTables from pyVO, too, albeit somewhat obscurely:

```
>>> import pprint
>>> pprint.pprint(images.votable.infos)
[<INFO ID="legal" name="legal" value="The data from Maydanak observatory
>>> pprint(images.votable.resources[0].infos)
[<INFO ID="queryPars" name="queryPars" value="(%(siaarea0)s &amp;&amp; c
 <INFO ID="QUERY_STATUS" name="QUERY_STATUS" value="OK"/>,
 <INFO ID="request" name="request" value="/maidanak/res/rawframes/siap/s
 <INFO ID="standardID" name="standardID" value="ivo://ivoa.net/std/sia"/
 <INFO ID="server_software" name="server_software" value="DaCHS/2.9.3 tw
 <INFO ID="server" name="server" value="http://dc.zah.uni-heidelberg.de"
 <INFO ID="citation" name="citation" ucd="" value="http://dc.zah.uni-hei
 <INFO ID="citation" name="citation" ucd="" value="http://dc.zah.uni-hei
 <INFO ID="ivoid" name="ivoid" ucd="meta.ref.ivoid" value="ivo://org.gav
```

[7] http://dc.g-vo.org/WIRR

[8] https://pyvo.readthedocs.io/en/latest/api/pyvo.dal.SIARecord.html#pyvo.dal.SIARecord

For why the information is available in this way, you need to understand a bit of VOTable (see sidetrack F). But this pattern works for all responses you will deal with in current VOTable.

Excursion: The Python Debugger

To inspect metadata like this from within a running program (as opposed to a notebook), it is really convenient to use the python debugger. To drop into it, call `pdb.set_trace()`:

```
 for pos in [
      (150.36, 55.90)]:
   images = svc.search(pos, size=(0.5, 0.5), verbosity=2)
   import pdb;pdb.set_trace()
   for row in images:
```

You can then enter Python statements (like the info expressions) and do many other things described in the Python reference[9]. When done looking around, you can type `cont` to let your program continue of `quit` to exit it.

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="sia", waveband="optical"):
  try:
    search_one_service(svc.accessurl)
  except Exception:
    import traceback; traceback.print_exc()
```

Ψ globalsiap.py

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

The `registry.search` function we are using here interfaces to a big directory of all the services that are in the VO: The Registry, which is also what is underlying the WIRR web page employed in problem 22.

The way we are querying the Registry here is a bit simplistic. In particular, you probably do not want to use *servicetype* constraints when doing science work. Global dataset discovery (which is what we are approaching here) is a lot more involved than just querying all services of a type (although this used to somewhat work in the early days of the VO). For now, however, we when we query like this, for everything that comes back from `registry.search`, we can request an image ("SIA") service. This happens in `search_one_resource` with

```
svc = res_rec.get_service("sia", lax=True)
```

[9]https://docs.python.org/3/library/pdb.html

Accept the `lax` argument for now. We will have a closer look at pyVO's Registry API later.

The exception catcher is there since not all services claiming to be standards-compliant actually are. It does not hurt to complain to the service operators if a service you are interested in behaves weirdly – sometimes the operators simply have not noticed that it is broken, or possibly has just broken.

To find out who to complain to, you can again use the Registry; the objects that are returned from `registry.search` have a `get_contact` method:

```
>>> svcs = pyvo.registry.search(keywords="pyvo")
>>> svcs[0].get_contact()
'Hendrik Heinl (+49 6221 541849) <gavo@ari.uni-heidelberg.de>'
```

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 is not always easy. At this point it is probably too early to complain to operators about astropy's VOTable warnings. We will later turn them off.

If a service hangs, you can interrupt it by hitting Control-C. In production code, you can set timeouts. We will later see how to do that.

Exercise 23

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 (we don't need that here) and change the position to something you are interested in or expect to have pretty pictures (M1 or M51 are always good candiates). Run the thing and see what you find.

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 pyvo.samp.connection() as conn:
  ... (search) ...
  pyvo.samp.send_image_to(conn, image.acref)
```

Ψ globalsiapsamp.py

Before running this, start Aladin (or some other SAMP-enabled image client) so the images are displayed.

In general, 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 do not properly close the connection –, which is taken care of by a context manager from `pyvo.samp`.

A context manager is a python construct consisting of an opening line of the form `with cm [as name]:` and then a block, the "controlled block". It is designed to ensure what is called "external invariants", some piece of state that the system should be in outside of the controlled block. You may know this from files, where the external invariant is "the file is closed":

```
with open("test.txt", "w") as f:
  f.write("some content\n")
print("f is closed")
```

By the time the print statement is reached, Python's semantics guarantee that f is closed and the content is written, regardless of what else happened (think exceptions) happened in the controlled block. The SAMP connection similarly ensures that once the controlled block is left, the connection is closed.

Given we are doing function calls between different processes written in different languages, we would argue this kind of code actually is surprisingly compact.

Exercise 24

Get the pyVO source code and find the source of pyvo.samp. Start TOPCAT, find the implementation of the `connection` context manager, and then open a SAMP connection manually from an interactive Python prompt. And then again, and a third time. What do you observe in TOPCAT?

Hint: To get the source code, try:

`git clone https://github.com/astropy/pyvo.`

Or, on Debian-derviced boxes:

`apt source python3-pyvo`

Exercise 25

Still in `samp.py`, inspect how `send_image_to` is implemented. From reading the code, can you figure out how to only send the image to Aladin? If you can, try your solution in globalsiapsamp.py by having Aladin and ds9 (Debian package: saods9) open at the same time.

Hint: To find out Aladin's client name, check TOPCAT's SAMP status window.

6 pyVO and TAP

Enter TAP

What we have seen so far does not scale when you are interested in more regions.

Also, only fairly basic constraints are supported.

TAP is far more powerful.

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

Run Sync TAP 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"])
```

Exercise 26

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.

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

What is coming back from `run_sync` is a sequence of `dal.Record` elements (well, the truth about TAPResults[10] is a bit more complex, but that's the gist of it).

You can make a normal astropy table from the result by calling `result.to_table()`, and there often are good reasons to do that. For instance, to save the table to a disk file, you can write:

```
result.to_table().write("saved.vot", format="votable")
```

Step 1a: Multiple TAP Queries

```
# Imagine more interesting queries here.
QUERIES = [
  ("twomass", "http://dc.zah.uni-heidelberg.de/tap",
    """SELECT TOP 1000000 raj2000, dej2000, jmag, hmag, kmag
      ...CIRCLE('ICRS', {ra}, {dec}, {radius}))"""),
    ...}

with pyvo.samp.connection() as conn:
  for short_name, access_url, query in QUERIES:
    service = pyvo.dal.TAPService(access_url)
    result = service.run_sync(query.format(**locals()), maxrec=90000)
    pyvo.samp.send_table_to(
      conn,
      result.to_table(),
      client_name="topcat",
      name=short_name)
```

Ψ fetch3.py

This does several things we have not seen before:

- `QUERIES` is a sequence of tuples; for examples, check the full source. Tuples are often a good choice when you have "inhomogeneous" (e.g., each item in a sequence "means" something different) data without much behaviour. When the rows become more complex, consider using python's `dataclasses` module, and when they have non-trivial behaviour, a "normal" class. Here, we just group a service title, a service URL, and a template for the query to run, for which a tuple works nicely.

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

- `query.format(**locals())` is a trivial example of what's called *templating*; you write a string that gets filled in, in this case using python's plain `format` method. You can (and sometimes should) get a lot more fancy with templating; one reason to do that could be to automatically quote strings. But as long as you control both the template and the fillers, it is probably better to not pull in extra dependencies just for templating.

  `**locals()` is a way to say: make all local variables available as keyword arguments. In general, `**` in an argument list means: what's next is a mapping, and turn it into keyword arguments, which sometimes is convenient if you want to build up a set of arguments step by step.

- `maxrec=90000` asks the server to return up to 90'000 rows (the *match limit*). When you do not pass *maxrec*, a service-specific default kicks in; you can find that default at `service.maxrec` (but take it with a grain of salt; this *may* be something like a lower limit). PyVO will issue a warning if your result overflowed your maxrec.

- `pyvo.samp.send_table_to` does a SAMP transfer of an astropy table (hence the `.to_table()`) to a SAMP client; it does a broadcast if you do not pass a `client_name`.

Exercise 27

The following program should print URIs and titles for images in some collection for whatever names are in `OBJECTS`:

```
import pyvo

OBJECTS = ["IC 4756", "NGC 3377"]
QUERY = """select accref, imagetitle
    from maidanak.reduced
    where object={object}"""

svc = pyvo.dal.TAPService("https://dc.g-vo.org/tap")
for object in OBJECTS:
    print(svc.run_sync(QUERY.format(**locals())).to_table())
```

(Note: this is *not* the way to match against multiple objects; you would instead use SQL sets or, probably more commonly, TAP uploads outside of silly exercises).

What really happens: An error message. Can you figure out where it comes from and how to fix things?

Exercise 28

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).

Re-write fetch3.py to query the retrieve all stars between 8 and 8.2 mags from each table (don't worry about the difference between H and G magnitudes for this problem). Also, send the results to Aladin (which is known as *Aladin* (capitalised) on the SAMP bus). See if you can get a nice plot of rv, pmra, and pmdec.

Hint: Check Aladin's Catalog/Create filter for fancy plotting options.

Step 2: Go Async

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

In this case, the main advantage is that we can run our queries in parallel. If all you want is have more time for your query, see the next slide for simpler options to run async TAP jobs.

```
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))
      pyvo.samp.send_table_to(...)
      job.delete()
```

Ψ 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 cannot peek into the future, it cannot 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 is 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 is still worth waiting for a result.

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 is good style to discard jobs once you do not need them any more.

This example is primarily intended to illustrate async mode itself.

Lightweight async

If you can live without real-time monitoring, you can write more concisely:

```
job.wait()
job.raise_if_error()
result = job.fetch_result()
```

In its default configuration, `job.wait()` waits for a change in the job status or a timeout and then returns. On modern TAP services, this generally is only one request every 10 minutes or so; this saves server-side ressources.

The `raise_if_error()` method gives you more reasonable exceptions than if you blindly try to access results from jobs that failed server-side.

With only a single job at a time, it is even simpler:

```
result = svc.run_async(query, ...)
```

The interface of `run_async` is that of `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 but you want to avoid the dance with checking the phases.

Step 3a: UCDs build SEDs

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

Not simply; photometry metadata in the VO is not 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))
```

Calling our multi-band data a SED ("Spectral Energy Distribution", that is some sort of flux densities plotted as a function of the spectral coordinate) is perhaps somewhat pretentious. To make this an actual SED, we would at least have to worry about photometry systems, which is a *real* concern even in the narrower optical, not to mention when you leave the optical. But bear with us.

What we *can* do is assign rough wavebands based on the UCDs we find on the various columns we retrieve (cf. sidetrack D for more on UCDs).

The clean way, incidentally, is a proper annotation of the columns in question with full photometry metadata (e.g., central wavelength, bandwidth, the system, 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 is just not a good way to put that information into a VOTable yet. If you are looking for something to contribute to the VO: this would be a good task. Just ask on the IVOA's data models mailing list.

Step 3b: Aggregate Photometry

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

For three catalogues, 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 do not have a "master catalogue" to match against, that is 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).

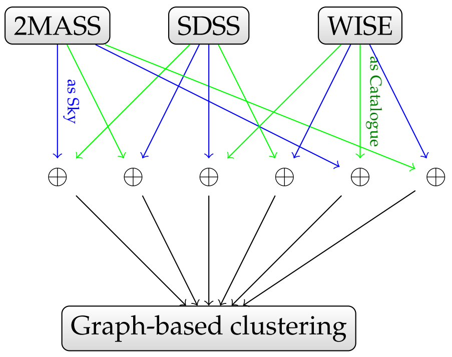
What is happening in that code? `sky_coords` are `astropy.SkyCoord` instances (in the example code, there is a function `get_coordinates_for_table` that makes these for essentially arbitrary tables as long as they are properly marked up).

The code then goes through all pairs of input `SkyCoords` and uses their catalogue match method to generate pairs of indices into these objects that are the closest pairs (that operation is not 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 is 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 will have all measurements that are crossmatched together and thus, hopefully, correspond to one object.

Sorry for this excursion. Feel free to ignore this.



For this course, but perhaps 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.

Ψ vohelper.py

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 = input(
     "s)elect SED, q)uit, enter for next? ")
   if selection=="q":
     break
   if selection=="s":
     selected.append(pos)
   plt.cla()
 return selected
```

Ψ fetch3-cluster.py

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

Exercise 29

Go through the source code of fetch3-cluster.py. You will see we have 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.

Exercise 30

Run fetch3-cluster.py and select a couple of objects. Keep the resulting file (`selected_positions.vot`) – we will want to reuse it later.

Write Tables in Style

Please furnish your tables with metadata. fetch3-cluster shows you how to do it with astropy:

```
t = table.Table()
t.add_column(table.Column(
  name='ra',
  data=selected[:, 0],
  unit=u.degree,
  description="ICRS RA of a selected object",
  meta={"ucd": "pos.eq.ra;meta.main"}))
```

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 – the S-protocols 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 do not work with simple ObsTAP (cutouts, for instance).

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

What we are doing here is another instance of the more general problem of global dataset discovery, to which I will return later in more generality.

Plan:

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

Obscore

The obscore "data model" consists of ~ 40 columns; use a TAP browser to look at them. Some highlights:

- dataproduct_type – states *image*, *timeseries*, and the like. The full list of terms is at http://www.ivoa.net/rdf/product-type.
- obs_publisher_did – a dataset identifier. By design, it should be globally unique and resolvable, but not all data providers are following this design...
- access_url – where to get the data from.
- s_ra, s_dec, s_fov – centre and FoV of the observation
- s_region – area covered by the dataset as an ADQL geometry. This column allows very concise queries, but alas, operators are free to have this NULL even when the have centre coordinates and a field of view.

Query the Registry

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

```
for svc_rec in pyvo.registry.search(datamodel="obscore"):
  print(f">>>>>> {svc_rec.short_name}...")
  try:
    svc = svc_rec.get_service("tap", lax=True)
    result = svc.run_sync("SELECT DISTINCT obs_collection"
      " FROM ivoa.obscore")
  except (Exception, KeyboardInterrupt):
    import traceback; traceback.print_exc()
    continue
  print("\n".join(r["obs_collection"] for r in result))
```

Do not run this script *just* for fun. It will hit quite a few services and make them seqscan their obscore tables.

To "use ObsTAP", just query the `ivoa.obscore` table via TAP.

To find TAP services having these tables, we once more use `pyvo.registry.search` but this time use the `datamodel` constraint. Also, we again use the `get_service` method on the RegistryResource instance that comes back from search; you should always specify what sort of service you want – `"tap"` in this case. Prefer this pattern over the explicit use of `access_url` on RegistryResource-s you may see in other places; access URLs are not a terribly well-defined concept, in particular not if one does not constrain the servicetype.

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

This script does not come attached. That's because on large services, the `SELECT DISTINCT` can actually be computationally expensive for the remote side; it is likely that you will see timeouts or very long runtimes. Hence, to try it, you will have to cut and paste, and then add the pyvo import.

More useful Obscore queries with positional constraints are usually much faster: the wonder of indexes and one of the major reasons why "just download stuff" is not a good plan with large datasets.

Query with Upload

For each ObsTAP service, we query against our object list (assumed to be in an astropy Table in `pois`):

```
if not svc.upload_methods:
  return

result = svc.run_sync(
  """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 is 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 inspect various properties of services. This, in particular, includes resource limits (maximum upload size, limit to which maxrec can be raised, etc).

Here, it is enough to know there is any upload method at all, because the standard says that inline upload must be supported if there is any upload support, and inline uploads is what we are doing.

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 will 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 – this is where the `TAP_UPLOAD.pois` above comes from. Remember that TOPCAT, which is what many upload examples are written for, has the convention of naming its uploads `t<n>`, where the *n* is the index in the table list in TOPCAT's main window.

You will almost always join the uploaded table with a table on the service, and thus it is 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, for the simple reason that other people use it, too.

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, stellar spectra come from essentially point-like objects, and thus you probably do *not* want to write something like

```
1=CONTAINS(POINT(mine.ra, mine.dec), s_region)
```

This *could* be more attractive if you are 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 does not 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).

Exercise 31

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*.

You can use URLs in a query's upload argument. To try this out, review the TGAS and RAVE exercise 28. Let the initial RAVE query be asynchronous. On the resulting job, call `wait` as above. Once it is done, upload what is job's `result_uri` attribute into the TGAS server with a normal positional upload join.

Collect Spectra finished

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

```
for ds_name, access_url in specs:
  print("Opening ...".format(access_url))
  try:
    pyvo.samp.send_spectrum_to(
      conn, access_url, client_name="splat", name=ds_name)
  except KeyError as exc:
    # regrettably, astropy raises the unspecific KeyError
    # when there it does not find the client.
    print("  ** Failed: is splat running?")
  except Exception:
    print("  *** Unexpected failure:")
    import traceback; traceback.print_exc()
```

Ψ get-spectra.py

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

What is new here is that we are catching exceptions. Somewhat suboptimally (because it is too non-specific), pyVO raises a KeyError when it cannot find SPLAT on the SAMP bus.

Giving some reminder-type message probably helpful when you run the program after a couple of months and have forgotten about SPLAT being a part of this analysis chain. Letting through the KeyError with a key of *splat* is probably a lot less helpful than the message we emit, even at the risk of catching KeyErrors of different origin. In practice, you would probably want to break out of the loop, too; the way this is written, you will get one message per spectrum, which may be slightly panic-inducing.

We catch all other exceptions; we do not want to exit the loop just because some spectrum is funny. Given what is in the try-block, the most likely origin of these exceptions is when SPLAT fails to open a spectrum for some reason and sends back an indication of that. What we are catching here, in effect, are an exceptions raised within SPLAT.

In general, there is no telling if the target client has already informed the user that something is wrong – it is 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"). But you basically never want to silence all exceptions, because that will hide all kinds of unexpected misbehaviour. So, as a relatively safe and diagnosable fallback, we just dump the traceback and trudge on.

Exercise 32

Can you change `get_spectra.py` 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 is almost always better to filter on the remote side rather than the local side. And chuck the "almost" if the constraint can be expressed as a single condition in a WHERE clause.

7 Higher SAMP Magic

Use Case: An Object Investigator

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

Plan: Write a program that other clients

- can send tables to and then
- when a table row is selected, computes a new table with data from other services
- that is then sent to Aladin for inspection.

SAMP: Listening to Messages

SAMP is based on messages; there are several message types (*MType*-s), which are documented on the IVOA wiki[11].

The SAMP client objects's `bind_receive_message` 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 call on the sender side (`params`).

SAMP has two types of messages: Notifications, which do not expect a response, and calls, which do. If you use `bind_receive_message`, you will cover both cases, which is generally a good idea, because all kinds of messages can come as either.

If a call (as opposed to a notification) comes in, it is associated with a message id, and the sending client will expect a response. If you do not give one, you will have ugly "pending" SAMP messages. Notifications have no message id, and they require not responses.

Here is a program that prints sky coordinates of "things" the user pointed to:

```
import pyvo
import vohelper

@vohelper.show_exception
def print_coord(privkey, sender_id, msg_id, mtype, params, extra):
  print("{} {}".format(params["ra"], params["dec"]))
  if msg_id is not None:
    conn.reply(msg_id, {"samp.status": "samp.ok", "samp.result": {}})

with pyvo.samp.connection(addr="localhost") as conn:
  conn.bind_receive_message("coord.pointAt.sky", print_coord)
  input()
```

The handler function has a rather complex *signature* (i.e., what parameters it takes and what it returns). Don't sweat it too much. In particular, do not be alarmed when you ignore `private_key`; for all I know no client at this point does any kind of cryptographic validation. There *are* security implications from SAMP, but very frankly: if you regularly have your browser execute Javascript from random web pages, you are in worse trouble.

The important part is `params`; this is where the parameters given on the SAMPMtypes page are in; in the case of the *coord.pointAt.sky* message we receive here, these are in the keys *ra* and *dec*. To try this, start Aladin and then the sample program. When you click on the sky, you will see the target coordinates in your terminal.

Versus the basic "Add SAMP Magic" method of getting a SAMP connection, we have now added an `addr="localhost"`. This is a workaround to make listening to messages a bit more robust on machines that have both IPv4 and IPv6 enabled (most have in 2024). If you get "connection refused" messages or the like when trying to send a message, try removing the argument.

As said above, when `msg_id` is not None (i.e., we got a call, not a notification), we have to send a reply. The sample code essentially says: "I have no results, and that is fine for this MType".

MTypes for the Vicinity Searcher

To make our program ready to receive tables via SAMP, we have to listen to *table.load.votable*. Params for that as per the MTypes wiki page:

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

url URL of the VOTable document to load

table-id local identifier for referencing

name human-readable name

To monitor whether a row in a table you received is selected, listen to *table.highlight.row*. Params:

table-id the local identifier

row the row index

Python Classes: Why?

We have to keep quite a bit of state in our program, at least:

- the SAMP connection
- the table sent to us.

There is also quite a bit of behaviour:

- receive and store the remote table
- see when rows are selected
- do searches when that happens.

When you have state and behaviour linked together, in Python think: "class".

Python Classes: How?

```
class VicinitySearcher:
  vicinity_size = 30
  client_name = "Aladin"

  def __init__(self, conn):
    self.conn = conn
    self.cur_table = self.cur_id = None

  def load_VOTable(self,
    private_key, sender_id, msg_id, mtype, params, extra):
    ...

  def handle_selection(self,
    private_key, sender_id, msg_id, mtype, params, extra):
    ...
```

Ψ vicinitysearcher.py

The trivial version of object lore in python is: All functions belonging to an object (*method-s*) have a first argument conventionally called `self` (the *instance*), 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).

You can also have attributes in the class itself; consider these constants, as assigning to these may not always do what you expect.

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

Create an instance by calling the class (here: `VicinitySearcher(conn)`). Whatever you pass into that call will be passed to the `__init__` method (the *constructor*).

Handling table.load.votable

```
class VicinitySearcher:
  def __init__(self, conn):
    [...]
    self.conn.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.conn.reply(msg_id,
      {"samp.status": "samp.ok", "samp.result": {}})
```

Since we bind the SAMP *table.load.votable* MType to **self**.load_VOTable (a *bound method*, which **VicinitySearcher**.load_VOTable would not be), we get our instance of VicinitySearcher (`self`) passed into our method for free.

When we then get notified of a table load, we set some instance variables that let us work with the table later.

To make this robust, we should catch exceptions and send replies with a status of `samp.error` in case of trouble; as said above, clients really want *some* reply when they send messages directly to clients and complain about pending SAMP calls when they receive none.

Handling table.highlight.row

```
  @vohelper.show_exception
  def handle_selection(self,
    private_key, sender_id, msg_id, mtype, params, extra):
    if params["table-id"]!=self.cur_id:
      return
    table_index = int(params["row"])
    print("Row selected:", table_index)
    response = self.make_response_table(table_index)

    if response is not None:
      vohelper.send_table_to(self.conn, self.dest_client, response)
```

The `@vohelper.show_exception` thing before the method definition is called a *decorator*. These are things (actually: functions) that operate on methods. In this particular case, all it does is make sure any exceptions raised within the SAMP handler are properly displayed. Since the SAMP handlers do not run in the main thread (and thus exceptions do not terminate the program), without this you will miss errors in the handlers.

The actual functionality (in this case, searching for matching data in a few catalogues and broadcasting any matches found) I have delegated to another method, `make_response_table`. This is an example for using Simple Cone Search; have a look at it!

Exercise 33

The action of the SAMP handler is in the `make_response_table` method; have a brief look at it to appreciate what is going on. Then, replace what is there with something that does a SIAP search on the service at http://dc.g-vo.org/lswscans/res/positions/siap/siap.xml and returns the corresponding table for sending to Aladin (hint: remember the `to_table` method of DAL results).

Exercise 34

Listening to the SAMP message *coord.pointAt.sky*, implement an "odometer" computing and printing after each step the distance travelled by the pointer.

To do this, you will need to keep the SAMP connection, the last position and the distance travelled so far as state; take the vicinitysearcher, remove the code keeping the state and behaviour used for its function, and insert our new logic.

Hints: Look at SkyCoord in Astropy and the mtypes page; when re-using SAMP bindings, make sure you handle messages, not calls.

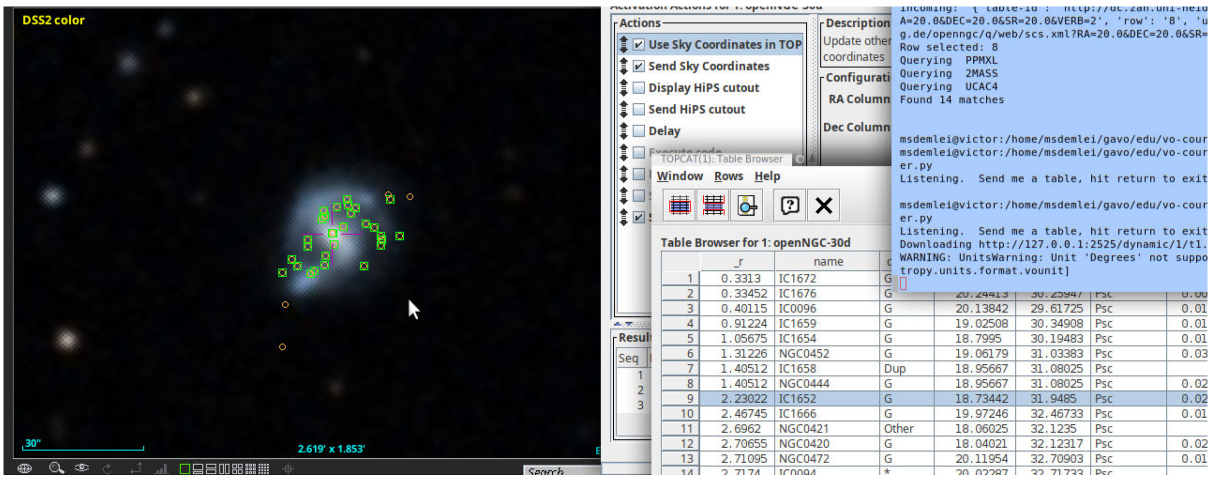
Try It Out

Start TOPCAT, Aladin, and the vicinity searcher.

Look for openngc SCS and pull some 40 degree cone.

Send the resulting table to the vicinity searcher, have *Send row index* as an activation action.

Click on table rows or plot points.



```
A=20.0&DEC=20.0&SR=20.0&VERB=2', 'row': '8', 'u
g.de/openngc/q/web/scs.xml?RA=20.0&DEC=20.0&SR=
Row selected: 8
Querying  PPMXL
Querying  2MASS
Querying  UCAC4
Found 14 matches


msdemlei@victor:/home/msdemlei/gavo/edu/vo-cour
msdemlei@victor:/home/msdemlei/gavo/edu/vo-cour
er.py
Listening.  Send me a table, hit return to exit

msdemlei@victor:/home/msdemlei/gavo/edu/vo-cour
er.py
Listening.  Send me a table, hit return to exit
Downloading http://127.0.0.1:2525/dynamic/1/t1.
WARNING: UnitsWarning: Unit 'Degrees' not suppo
tropy.units.format.vounit]
```

Table Browser for 1: openNGC-30d

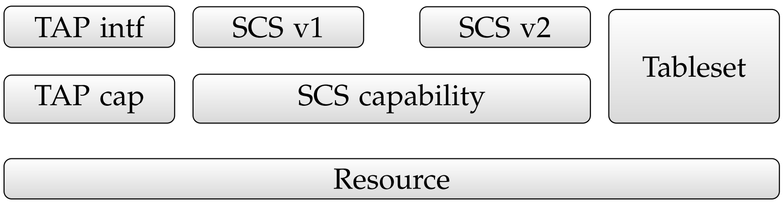
	_r	name					
1	0.3313	IC1672	G				
2	0.33452	IC1676	G				
3	0.40115	IC0096	G	20.13842	29.61725	Psc	0.01
4	0.91224	IC1659	G	19.02508	30.34908	Psc	0.01
5	1.05675	IC1654	G	18.7995	30.19483	Psc	0.01
6	1.31226	NGC0452	G	19.06179	31.03383	Psc	0.03
7	1.40512	IC1658	Dup	18.95667	31.08025	Psc	
8	1.40512	NGC0444	G	18.95667	31.08025	Psc	0.02
9	2.23022	IC1652	G	18.73442	31.9485	Psc	0.02
10	2.46745	IC1666	G	19.97246	32.46733	Psc	0.01
11	2.6962	NGC0421	Other	18.06025	32.1235	Psc	
12	2.70655	NGC0420	G	18.04021	32.12317	Psc	0.02
13	2.71095	NGC0472	G	20.11954	32.70903	Psc	0.01

8 pyVO and the Registry

A Closer Look at registry.search

We have seen `registry.search` already in some places.

To go more deeply, you need to understand a bit more of the Registry data model:



The illustration shows a resource, the thing that has common metadata like a title, description, authors, space-time coverage, and the like.

On top of that sit capabilities, which are things the resource "can do for you": typically, protocol endpoints. This particular resource has two capabilities: TAP for database queries and SCS for simple cone searches.

Each capability can have one or more interfaces, that is, things that clients can talk to. For reasons of practicality, a "good" capability should only have one interface; but this may change as future standards are defined. Interfaces for multiple versions of a protocol on one capability, as sketeched here, is not something we are planning for, though; SIA1 and SIA2, the only example where that would matter right now, are modelled as two different capabilities.

There are many other things that a resource can harbour beyond capabilities; an important example is the tableset, which lists what tables the resource contains. Be warned that VO-DataService (the standard that defines how tablesets are written) does not *require* tablesets, and so some data publishers still do not provide them. If you catch one of those, complain to them.

Principles of RegistryResource

What you get back from `registry.search` is a sequence of `RegistryResource` instances.
It has attributes for metadata (`res_title`, `res_description`...), and important methods:

- `describe()` – return a summary of what pyVO knows about the resource.
- `access_modes()` – short identifiers for the capabilities of the resource
- `get_service(type, lax, keyword)` – return a service object to query the resource
- `get_tables()` – return a sequence of table-like objects with what tables you can query

The main method for practical use really is `get_service`. Its `type` argument is something like `"tap"` – the strings that will produces something for a given resource can be obtained using `access_modes()`.

The `lax` keyword argument deserves some explanation: If there are multiple capabilities of a given type on a resource – something that is still common for VizieR, who like to keep all tables belonging to one paper together in one VO resource in this way –, pyVO does not know which one to pick unless you pass keywords (to be matched within the capabilities' descriptions). If you think you know what you are doing, you can ask pyVO to pick one of the capabilities more or less at random: That is what `lax=True` does. It is not recommended to do that in code that matters.

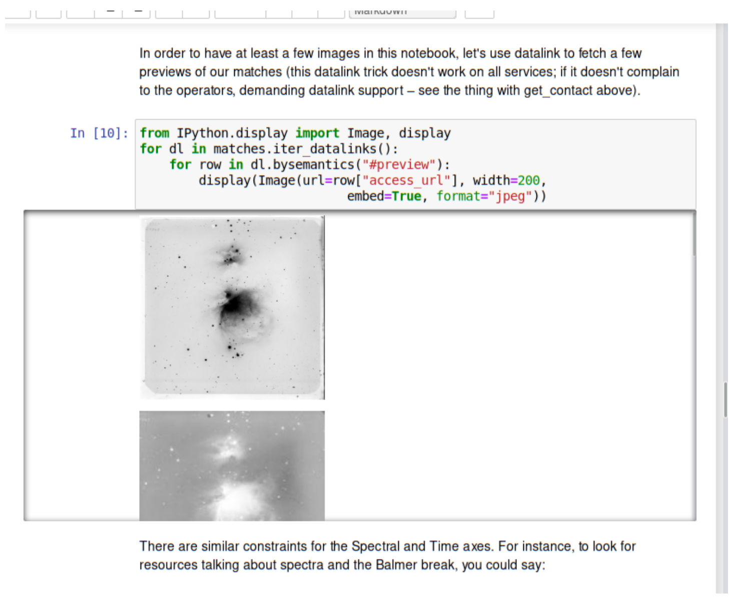
As of 2024, most of this code has recently been refurbished, and there have been bugs off and on. If you find you need to use `lax=True` when you do not expect to, it is likely you ran into a a buggy version. In these cases, don't feel bad about passing `lax=True`.

There are some legacy attributes and methods that you should no longer use: `access_url`, `service`, `search()`; all these only do something sensible when there is only one capability on a `RegistryResource`. This is not unlikely if you did constrain the `servicetype` in your call to `search`. But in general it is a much better idea to search for data and decide on access modes later in data discovery. Most resources today come with multiple capabilities, and it is good if you can choose the most appropriate for your task at hand.

Interactive Use of the PyVO Registry API

Finally: A jupyter notebook!

Ψ data-discovery-demo.ipynb


In order to have at least a few images in this notebook, let's use datalink to fetch a few previews of our matches (this datalink trick doesn't work on all services; if it doesn't complain to the operators, demanding datalink support – see the thing with get_contact above).

```
In [10]: from IPython.display import Image, display
         for dl in matches.iter_datalinks():
             for row in dl.bysemantics("#preview"):
                 display(Image(url=row["access_url"], width=200,
                                      embed=True, format="jpeg"))
```

There are similar constraints for the Spectral and Time axes. For instance, to look for resources talking about spectra and the Balmer break, you could say:

Exercise 35

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?

Exercise 36

Which IAU constellation is the least massive exoplanet in the exoplanet merged catalogue in? Try solving this using pyVO's registry API; hint: to figure out constellations, having the constellations as ADQL polygons is really handy.

Resolving Ivoids

IVOA identifiers are the primary keys in the VO Registry.

When keeping notes like "which service did I use", the ivoid (rather than a DOI) still is the better choice in the VO for the simple reason that all VO resources have an ivoid, but many have no DOI.

To resolve an ivoid:

```
svc = pyvo.registry.search(ivoid='ivo://org.gavo.dc/tap')[0]
```

You can then go on as we did above with `access_modes`, `get_service`, etc.

Write Your Own Constraint

registry.search uses constraint classes to build queries.

You can extend the set of constraint classes yourself by inheriting from `registry.SubqueriedConstraint`.

Say you want to use the experimental UAT extension to RegTAP, i.e., `rr.uat_concept`:

```
class UATConcept(pyvo.registry.SubqueriedConstraint):
    _keyword = "uat"
    _subquery_table = "rr.subject_uat"

    def __init__(self, uat_id):
        self._condition = "uat_concept={uat_id}"
        self._fillers = {"uat_id": uat_id}
```

Ψ new-constraint.py

What is going on here?

- We define a class inheriting from the base class `SubqueriedConstraint`. This is defined in pyvo/registry/rtcons.py; but the code there is rather dense, so it is probably best to look at other classes that are SubqueriedConstraint-s further down the source to get a feeling for how this is supposed to work.
- The first thing we need to define is what table we want to match in; this ends up in the `_subquery_table` class variable. Here, we are using an extension on the default RegTAP server, a table containing UAT keywords for all the services. This is more useful than what is in the standard `rr.res_subject` table, as there, you have all kinds of words and keyword schemes and all that – but the UAT table is non-standard, which may be the reason why you need to write your own constraint.
- In the constructor, we fill instance attribute `_condition`, which needs to contain ADQL suitable for WHERE. However, this is just a template with fields (here: the stuff with curly braces) to be replaced when the machinery bakes the actual query.
- For each template field, we have to give a key-value pair in the `_fillers` dictionary. Here, there is just `uat_id`. The reason this is done behind the scenes is that we want to make SQL string literals from python strings, and the logic to do that should not be repeated in each constraint class but in only one central place.
- The rest of the query generation is done by pyvo.registry. In reality, this is often a bit more complex, for instance, because you may want to have multiple terms combined with OR; when you pass multiple constraints, they are combined with AND. See, for instance, the UCD constraint for how you would go about this.
- The `_keyword` class variable gives the name of the keyword argument equivalent to passing in a `UATConcept` constraint.

Exercise 37

(You will need to have looked at the vocabularies sidetrack for this)

Take new-constraint.py and add support for query expansion: add a keyword argument `expand`. If that is true, include the narrower concepts of what was passed in, too.

Hint: You can leave (something like) this to the server with a UDF, or you can do the query expansion locally; the first way is simpler, the second perhaps more instructive.

9 Datalink

Datalink: Getting Related Artefacts

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

https://dc.g-vo.org/static/datalinks.shtml is a showcase of various applications of datalink. You can retrieve the links in a web browser and ought to get a reasonable UI if you have enabled javascript.

This is really machine-readable data; load any of these links into TOPCAT to inspect it as a VOTable:

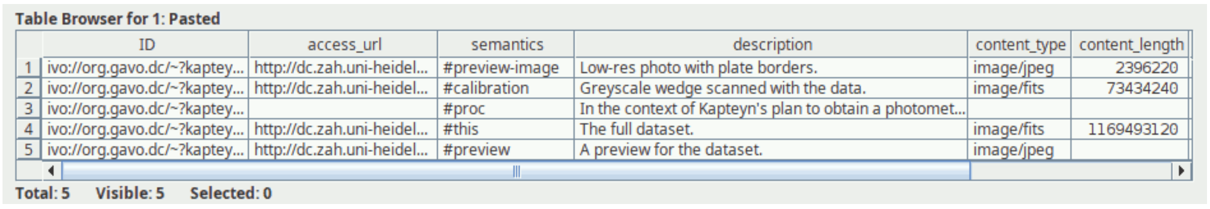


Table Browser for 1: Pasted

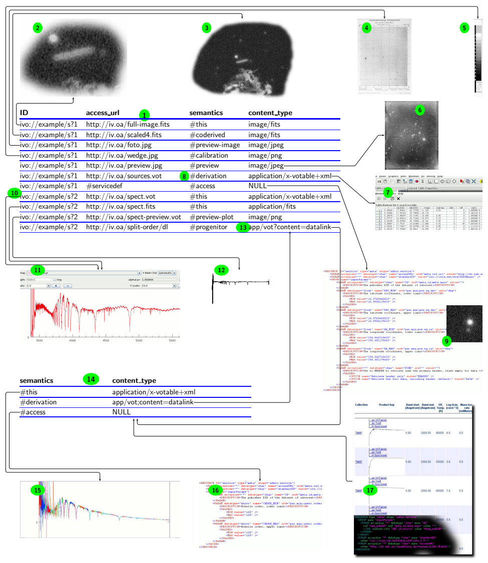
	ID	access_url	semantics	description	content_type	content_length
1	ivo://org.gavo.dc/~?kaptey...	http://dc.zah.uni-heidel...	#preview-image	Low-res photo with plate borders.	image/jpeg	2396220
2	ivo://org.gavo.dc/~?kaptey...	http://dc.zah.uni-heidel...	#calibration	Greyscale wedge scanned with the data.	image/fits	73434240
3	ivo://org.gavo.dc/~?kaptey...		#proc	In the context of Kapteyn's plan to obtain a photomet...		
4	ivo://org.gavo.dc/~?kaptey...	http://dc.zah.uni-heidel...	#this	The full dataset.	image/fits	1169493120
5	ivo://org.gavo.dc/~?kaptey...	http://dc.zah.uni-heidel...	#preview	A preview for the dataset.	image/jpeg	

Total: 5 Visible: 5 Selected: 0

The power of datalink comes from the fixed structure of these rows, which allows machines to do sensible things with them. The rows (normally) consist of

- a (theoretically globally unique) ID of the dataset the link is for
- a URL for the data linked `access_url`
- a human-readable `description`,
- `semantics`, that is, a machine-readable identification of what this link is. This comes from a controlled vocabulary, http://www.ivoa.net/rdf/datalink/core. This allows clients to sensibly group and/or select these links
- a type and length of the content that lets client figure out what to do with the file: `content_type`, `content_length`
- and a few more technical fields.

Datalink in a Cartoon



ID	access_url	semantics	content_type
ivo://example/s?1	http://iv.oa/full-image.fits	#this	image/fits
ivo://example/s?1	http://iv.oa/scaled4.fits	#coderived	image/fits
ivo://example/s?1	http://iv.oa/foto.jpg	#preview-image	image/jpeg
ivo://example/s?1	http://iv.oa/wedge.jpg	#calibration	image/png
ivo://example/s?1	http://iv.oa/preview.jpg	#preview	image/jpeg
ivo://example/s?1	http://iv.oa/sources.vot	#derivation	application/x-votable+xml
ivo://example/s?1	#servicedef	#access	NULL
ivo://example/s?2	http://iv.oa/spect.vot	#this	application/x-votable+xml
ivo://example/s?2	http://iv.oa/spect.fits	#this	application/fits
ivo://example/s?2	http://iv.oa/spect-preview.vot	#preview-plot	image/png
ivo://example/s?2	http://iv.oa/split-order/dl	#progenitor	app/vot?content=datalink

semantics	content_type
#this	application/x-votable+xml
#derivation	app/vot;content=datalink
#access	NULL

Here is what you can see on this cartoon if you zoom in sufficiently far:

The first seven rows in correspond to a scanned plate. There is a placeholder for the **original dataset** with semantics `#this`, i.e., the "main" dataset. A **rebinned version** (the figure shows a larger area) is declared as `#coderived` from the main dataset. The semantics here could be a bit more precise to indicate this link is just the resampled `#this`. If there were a clear idea what a machine would do differently if it knew that, one can define a refined term using IVOA processes (look for "IVOA VEP" if interested).

The original plate was part of an early survey which has been published in book form. A JPEG **photo of the book page** corresponding to the plate is declared as `#preview-image` in row three.

Datalink is ideal for declaring files from a dataset's provenance chain. In row four, we include a **PNG grey wedge** from the scan with `#calibration` semantics.

In the other direction, you can also declare derived data products, such as the sources.vot in row five, supposed to be a **table of extracted sources** from the image; the corresponding semantics is `#derivation`, and again there may be cases when some more refined term for extracted sources would be beneficial and should be defined.

Row six has a **thumbnail** of the image, declared as a `#preview`.

The next row defines a **cutout service**. Datalink allows a straightforward declaration of the parameters for server-side data manipulation services within the VOTables that return datalink metadata. If you decipher the XML, you will see that this is sufficient not only to operate the service but also produce attractive UIs by declaring units, UCDs, and ranges of the pertinent parameters.

The remaining three rows correspond to a spectrum (a single datalink document can contain links for more than one dataset, but in practice that is rare).

The semantics `#this` in row eight should already be familiar; it corresponds to a **spectrum** here.

The **preview** in spectrum case is a plot, which is reflected in the different semantics. A client consulting the datalink vocabulary will figure out that `#preview-plot` actually is-a `#preview`.

The last datalink shows **recursive datalink**: its file has the media type

application/x-votable+xml;content=datalink

that designates datalink documents (and can be used in protocols like ObsTAP, too). In this case, the datalink is for a `#progenitor` in the provenance chain, which here is a file with unmerged Echelle orders.

Datalink in PyVO

In pyVO, datalink is (primarily) exposed in search results.

On datalink-enabled services, you can iterate over `iter_datalinks()`, which iterates over `DatalinkResults` instances.

On these, you can pull links using `bysemantics`:

```
svc = pyvo.ssa.SSAService("http://dc.g-vo.org/feros/q/ssa/ssap.xml?")
matches = svc.search(SkyCoord.from_name("EI Eri"), 0.001)

for links in matches.iter_datalinks():
  for link in links.bysemantics("#preview"):
    print(link["access_url"])
```

Or just iterate over `links` to see all links available.

Yes, this is a bit deeply nested in the way of iteration, but that is the price of flexible protocols. The links come as dictionary-like objects with keys matching the column labels from the datalink specification. The labels are those written in typewriter in the enumeration of the datalink fields above.

Exercise 38

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

Try your program on the SSA example from the lecture.

Use Case: Overview With Previews

Let's say you want to spot bad or weird spectra without actually retrieving or plotting the spectra themselves.

Just download the previews and merge them into one image:

```
svc = pyvo.ssa.SSAService("http://dc.g-vo.org/feros/q/ssa/ssap.xml?")
matches = svc.search(SkyCoord.from_name("EI Eri"), 0.001)
previews = []
for dl in matches.iter_datalinks():
    prev_url = next(dl.bysemantics("#preview"))["access_url"]
    im = Image.open(io.BytesIO(requests.get(prev_url).content))
    previews.append(im)
```

Ψ datalink-previews.py

The perhaps slightly alarming `next(...)` construct is just "pick off the first item from an iterator"; we can do that here because we only want one preview per dataset (and actually, there is only one). This is a convenient construct when dealing with the nested iteration in datalink in many cases when you (think you) know there is only one link with a certain semantics.

The full source has some code merging all the previews into one raster image using the excellent python imaging library PIL.

Datalink: Remote Processing on Datalink Documents

Datalink also lets you declare processing services. The SODA standard defines 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 are 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))
```

Ψ datalink-soda.py

This example retrieved datasets that come as datalinks directly. This is why we included `access_format` in the obscore query: This way, pyVO knows when it is dealing with a datalink document, and it will add the `iter_datalinks` and `processed` methods when the service offers the necessary facilities.

It is more common to deliver "normal" files and offer datalink on the side. In this case, things get somewhat more complicated at the moment because with the current API you can *either* see the actual records or the datalinks.

Datalink: Remote Processing on Non-Datalink Documents

Use case: Hα maps of Sd galaxies from CALIFA.

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. We can extract Hα maps by doing spectral cutouts, supported via SODA's BAND parameter (which takes vacuum wavelengths in meters).

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. The cubes come in three different setups. To avoid duplicate data, we will only look at COMB data.

Hα is at 656.25 nm (vaccuum) in the lab. For the low redshifts we are talking about here, $\lambda_{\text{lab}} = (1+z)\lambda_0$ is just fine to compute where the galaxy's Hα is at the spectrograph.

Doing the cutouts by calling `processed` on the link for the data itself (`#this`):

```
matches = svc.run_sync(
    "SELECT califaid, obs_publisher_did, mime, em_min, em_max, redshift"
    " FROM califadr3.cubes"
    " JOIN califadr3.objects USING (califaid)"
    " WHERE setup='COMB' AND hubtyp='S d'")

for dl in matches.iter_datalinks():
    lobs = ???
    map = next(dl.bysemantics("#this")).processed(band=(lobs, lobs))
```

Trouble: How do I find the redshift (i.e., `lobs`) for my `dl`?

The (current) answer is: Use `ID` in the dl rows to match against `obs_publisher_did` in `matches`. How do you know it's that column? Well, for obscore and obscore-like tables, it will almost always be that.

If you have to dig yourself, things get messy because pyVO does not expose that information properly yet. Meanwhile, you can trudge on by inspecting the VOTable. You first get the service definition for the cutout service, most of the time the first service there is (in VOTable, that corresponds to a RESOURCE). In there, look at the PARAMs of the GROUP in there, and you will find a PARAM named `ID`. Whatever is in its `ref` attribute is what you are looking for:

```
>>> svc = next(matches.iter_adhocservices())
>>> print(list(svc.groups[0].iter_fields_and_params()))
[<PARAM ID="ID" arraysize="*" datatype="char" name="ID"
  ref="obs_publisher_did" ucd="meta.id;meta.main" value=""/>]
```

Yes. There should be a better and more robust API for this; in pyVO 1.6, there you will probably have an `original_row` attribute on what you get back from `iter_datalinks`.

Datalink: Simultaneous Links and Metadata

```
matches = svc.run_sync(
    "SELECT califaid, obs_publisher_did, mime, em_min, em_max, redshift"
    " FROM califadr3.cubes"
    " JOIN califadr3.objects USING (califaid)"
    " WHERE setup='COMB' AND hubtyp='S d'")
result_rows = matches.to_table()
result_rows.add_index("obs_publisher_did")

for dl in matches.iter_datalinks():
    rec = result_rows.loc["obs_publisher_did", dl["ID"][0]]
    califaid = rec["califaid"]
    lobs = 10*(1+rec["redshift"])
    processed = next(dl.bysemantics("#this")
      ).processed(band=(lobs, lobs))
```

Ψ soda-with-rows.py

The novelty here is that we are making a proper astropy table of the results now in order to be able to create an *index* on it. That's a technial term for "make it so we can fetch rows quickly by using values from this column". With the `add_index` call, we can use `.loc` attribute on the table to quickly pick out rows by `obs_publisher_did`. This is how we can find the table row for a datalink.

Exercise 39

Get the soda-with-rows.py script for doing cutouts on CALIFA DR3 and make a false colour image for IC 1151 by taking the slices from the COMB cube (see the setup column) at 400 nm as blue, at 550 nm as green, and at 700 nm as red. Do not download the whole cube, use SODA to just retrieve exactly what you need.

Hint: If you have no better way to combine single-channel pixels to an RGB image in Python, use the excellent Python Image Library PIL (in its modern incarnation of pillow). This is still not entirely trivial, so here is how to get three arrays red, green, and blue, made up of three frames into a colour jpeg using plain PIL and numpy:

```
def _normalize_for_image(pixels):
  pixels = numpy.flipud(pixels)
  pixMax, pixMin = numpy.max(pixels), numpy.min(pixels)
  pixels = (pixels-pixMin)/(pixMax-pixMin)*255
  return numpy.asarray(pixels, numpy.uint8)
```

```
pixels = numpy.array([
    normalize_for_image(red),
    normalize_for_image(green),
    normalize_for_image(blue)]).transpose(1,2,0)
Image.fromarray(pixels, mode="RGB"
  ).save("IC1151.jpeg", format="jpeg")
```

I have not tried looking for a less pedestrian way to do this; if you have one, please write in.

10 At the Limit: VO-Wide TAP Queries

VO-Wide TAP Queries

People often say: "I want everything in the VO on object X".

This is far too hard. There are many reasons why this is hard, beginning with what "everything" is – for instance, you would not normally want every frame containing the object ever taken.

What *is* marginally possible: "Give me all measurements of a certain sort of UCD in a certain vicinity." Actually, the constraints can be a lot more general than just a cone search, as long as you can formulate it with UCDs.

However, this is surprisingly involved, mostly for stupid reasons. Follow me along for proper motions (`pos.pm`).

Note: This is probably not something realistic for research within the next few years. But it is a nice exercise in how far you can take pyVO and TAP.

A RegTAP Query for Tables and TAP Services

For "where can I find data with UCD X?", there is `pyvo.registry.UCD`.

But we need to know *which table* has a column with our UCD.

PyVO can't do that yet; hence, use a direct RegTAP query:

```
SELECT DISTINCT access_url, table_name
FROM rr.interface
NATURAL JOIN rr.capability
NATURAL JOIN rr.res_table
NATURAL JOIN rr.table_column
NATURAL JOIN rr.stc_spatial
WHERE
    standard_id LIKE 'ivo://ivoa.net/std/tap%'
    AND ucd LIKE 'pos.pm%'
    AND 1=INTERSECTS(POINT({RA}, {DEC}, {SR}), coverage)
    AND (table_type!='output' OR table_type IS NULL)
```

How do you come up with a query like this? Well: you *can* start from what pyVO does; pyvo.registry has the `get_RegTAP_query` function that will return what pyVO would generate for a given set of constraints. For instance:

```
import pyvo

print(
  pyvo.registry.get_RegTAP_query(
    pyvo.registry.UCD('pos.pm%')))
```

outputs this horror:

```
SELECT
ivoid, res_type, short_name, res_title, content_level, res_description,
reference_url, creator_seq, created, updated, rights, content_type,
source_format, source_value, region_of_regard, waveband,
  ivo_string_agg(COALESCE(access_url, ''), ':::py VO sep:::') AS access_urls,
  ivo_string_agg(COALESCE(standard_id, ''), ':::py VO sep:::') AS standard_ids,
  ivo_string_agg(COALESCE(intf_type, ''), ':::py VO sep:::') AS intf_types,
  ivo_string_agg(COALESCE(intf_role, ''), ':::py VO sep:::') AS intf_roles,
  ivo_string_agg(COALESCE(cap_description, ''), ':::py VO sep:::') AS cap_descriptions
FROM
rr.resource
NATURAL LEFT OUTER JOIN rr.capability
NATURAL LEFT OUTER JOIN rr.interface
NATURAL LEFT OUTER JOIN rr.alt_identifier
NATURAL LEFT OUTER JOIN rr.table_column
WHERE
(ucd LIKE 'pos.pm%')
GROUP BY
ivoid, res_type, short_name, res_title, content_level, res_description,
reference_url, creator_seq, created, updated, rights, content_type,
source_format, source_value, region_of_regard, waveband
```

This is massively uglified by pyVO's need to be generic and to, in a single query, pull all kinds of information on the services available. In tailored RegTAP queries you rarely need that kind of thing. Still, you could take this query and strip it down until it does what you want, in particular as regards what tables to hit in the first place.

Alternatively, RegTAP is written such that to build a query, you only have to look for what table a piece of data you want to retrieve or constrain is in and then NATURAL JOIN with the table. The canonical source to find this kind of information is the RegTAP standard, Demleitner and Harrison et al. (2019), in particular its Figure 2; also skim over the example queries in section 10 if you need to hand-write RegTAP queries.

In this case, to be able to query TAP services, we need the access url (in rr.interface) of the service and the table name (in rr.res_table) for a table containing a column with a UCD (in rr.table_column). To be able to say "I want a TAP service", we need to constrain the standard identifier (in rr.capabilty). Finally, we want to throw out tables that do not have data for our region of interest, and hence we also need to constrain the spatial coverage (in rr.stc_spatial). That consideration almost results in the hand-tailored query shown above already.

Two details are in there on top: the DISTINCT after the SELECT is so we do not get one pair of access url and table name for every *column* in the tables that have matching UCDs; in general, there will be more than one of them, and we still only want to query the table once.

And then there is the odd

```
AND (table_type!='output' OR table_type IS NULL)
```

This is another instance of where something seemed like a good idea to the standards designers – in this case: Use the same elements to declare output tables and queriable tables – makes for something that is hard to understand in later use. What this means is: Ignore tables that are declared in the registry but that one probably cannot query.

Running the RegTAP Query

Running RegTAP queries just means picking a suitable TAP service and calling `run_sync`:

```
reg_svc = pyvo.registry.regtap.get_RegTAP_service()
result = reg_svc.run_sync(regtap_query)

svcs = {}
for row in result.to_table():
  svcs.setdefault(row["access_url"], []).append(row["table_name"])
return svcs.items()
```

There is no magic behind `get_RegTAP_service` – it is constructing a normal `TAPService`, just configured with an access URL known to lead to a RegTAP service. By the way, you can change that URL if you want to use a different registry service; use `choose_RegTAP_service` from within pyVO, or set the `IVOA_REGISTRY` environment variable to your preferred RegTAP service's access URL.

Note how we are grouping the tables belonging to a service in this code. This is exactly a GROUP BY operation in the database sense. So:

> **Exercise 40**
>
> In multitap.py, have a look at `get_services_and_tables`; in there, we are doing a grouping operation on the client (i.e., our) side. Can you move to to the server side using GROUP BY and the `ivo_string_agg` UDF?

Query Generation I: Defining the Schema

We want to build queries that let us fill a table defined like this:

```
#    col-name,  UCD,                  Unit,    type-to-cast-to
RESULT_SCHEMA = [
    ('cat_id', "meta.id;meta.main",   None,     "CHAR(*)"),
    ('ra',     "pos.eq.ra;meta.main", "deg",    None),
    ('dec',    "pos.eq.dec;meta.main","deg",    None),
    ('pmra',   "pos.pm;pos.eq.ra",    "mas/yr", None),
    ('pmde',   "pos.pm;pos.eq.dec",   "mas/yr", None),]
```

You may recognise our technique of writing "inhomogeneous" records in tuples from our fetch3 example. In this case, we give names, the UCDs from which to fill the columns, the target unit, and a type the column should have; this is important in the case of `cat_id`, the object identifier within the catalogue, which sometimes is an integer and sometimes is a string. We have to unify this if rows from different tables are supposed to end up in one result table. All other columns will be real-valued if they are somehow sane, and hence we do not need to cast.

We now need to write code that can create database queries from these specifications and table metadata.

Query Generation II: From Clause And a Template

Given a TAP service `svc`, a `table_name`, our result schema, and the region of interest in RA, DEC, and SR, make a query to produce rows for our result schema:

```
db_table, select_clause = svc.tables[table_name], []
for dest_name, ucd, unit, type in RESULT_SCHEMA:
    select_clause.append("{} AS {}".format(
        fieldname_with_ucd(ucd, db_table),
        dest_name))
select_clause.append(f"'{table}' AS table_name")
select_clause.append(f"'{svc.baseurl}' AS svc_url")

return ("SELECT {select_serialised} FROM {srctable}"
    " WHERE 1=CONTAINS(POINT('ICRS', {racol}, {deccol}),"
    "   CIRCLE('ICRS', {ra}, {dec}, {sr}))").format(
        select_serialiased=", ".join(select_clause),
        srctable=table_name,...)
```

In this snippet, we first incrementally build a select clause by looking for the UCDs we are interested in in the remote table definition (that we retrieve using `svc.tables[table_name]`) and make "their-name AS our-name" particles. We add two constant fields for the table name and the service access URL; this is so we can later still see where everything came from.

We have to define the function `fieldname_with_ucd` ourselves, because astropy tables (which is what is in `svc.tables`) do not have the convenient `fieldname_with_ucd` method that pyVO `DALResults` have. Perhaps this should change? Anyway: the implementation is trivial, except that we lowercase both the incoming UCD and the UCDs we get from the service. Curse case-insenstitive items.

These particles are then joined into the selclause in the ADQL template.

Query Generation III: Delimited Identifier Workaround

Regrettably, the code immediately fails.

```
$ python3 multitap-broken1.py
[...]
pyvo.dal.exceptions.DALQueryError:
    Incorrect ADQL query:
    Encountered "/". Was expecting one of: <EOF> "." "," ";" "AS"
        "WHERE" "GROUP" "HAVING" "ORDER" "\""
        <REGULAR_IDENTIFIER_CANDIDATE> "NATURAL" "INNER" "LEFT"
        "RIGHT" "FULL" "JOIN"
```

Ψ multitap-broken1.py

The problem: Vizier uses delimited identifiers but has them unquoted in the registry. Workaround:

```
def perhaps_quote(table_name):
    parts = table_name.split(".")
    for index, part in enumerate(parts):
        if not re.match("[A-Za-z0-9][A-Za-z0-9_]*$", part):
            parts[index] = '"{}"'.format(part.replace('"', '""'))
    return ".".join(parts)
```

This is a long-standing problem on VizieR's side; the standard has been clear on this for a long time ("when delimited identifiers are used as table names on the relational side, the quotes must be part of name's value, and the capitalisation used in the DDL must be preserved"), and actually, a function like `perhaps_quote` cannot even really work (e.g, in "USNO-B-1.0", is the dot part of a name or a schema separator?). So – this is another illustration of where sometimes one has to live with imperfections and just cope as well as possible.

Running Queries I: Feature Detection

On a service like VizieR with our *pos.pm* criterion, we will have to query a lot of tables and stack the results on the client side.

Don't take my word for "a lot of tables"; on VizieR, at the time of writing, the ADQL query

```
SELECT COUNT(*)
FROM (
    SELECT DISTINCT table_name FROM tap_schema.columns
    WHERE ucd LIKE 'pos.pm;%') AS q
```

returns a whopping 2003; in reality, due to our positional constraint, we would be firing off a lot fewer queries, but it would still be nice if we only had to run one.

Can we take a union of the results on the server side?

Perhaps. We need the ADQL UNION operator for that. Regrettably, it is optional.

Interactively, you will find information on supported features in the *ADQL* tab of modern TOPCATs. From within pyVO, there is a complex hierarchy of objects below a TAPService instance. Unless you really want to read Demleitner and Dowler et al. (2012), take the following blindly as a recipe.

Does a service support union?

```
knows_union = svc.get_tap_capability().get_adql().get_feature(
   "ivo://ivoa.net/std/TAPRegExt#features-adql-sets", "UNION")
```

The `get_feature` method takes two arguments: An ivoid identifying the sort of feature you are querying, and a key into the feature listing. To figure out these two strings, I am afraid you will need to consult the ADQL standard (Mantelet and Morris et al., 2023).

Incidentally, as of mid-2024, VizieR's ADQL engine does not yet support UNION, which is the main reason we have put it a sanity break in multitap.py

```
    if len(tables)>30:
        sys.stderr.write("  (cropping to 30 tables for handleability)\n")
        tables = tables[:30]
```

(but then we probably would have anyway, because even a union over 300 tables is a bit too much for an educational example).

Running Queries II: Adapting to Server Capabilities

Since UNION is optional, we have to have two code paths now, one for services with UNION, one for ones without. It will not get much simpler than that:

```
svc = pyvo.dal.TAPService(access_url)
knows_union = svc.get_tap_capability().get_adql().get_feature(
    "ivo://ivoa.net/std/TAPRegExt#features-adql-sets", "UNION")

queries = [get_query(svc, table_name)) for table_name in tables]

result_rows = []
def feed_rows(astropy_table):
    for row in astropy_table:
        result_rows.append(dict(zip(row.colnames, row.as_void())))

if knows_union:
    feed_rows(svc.run_sync(
        " UNION ".join(queries)).to_table())
else:
    for query in queries:
        feed_rows(svc.run_sync(query).to_table())
```

This is seriously ugly code; to smuggle shared code into the two legs of the `knows_union` selection, we first take out the generation of the queries from where they run, and then create a local function encapsulating the logic of processing result rows (this is called a *closure* in this case, because the function encloses the `result_rows` list from the parent block).

And we have two rather different pieces of code on the two sides of the selection. They will age and break differently, and all this is painful.

Take it from me: Optional features suck. In almost everything. If you ever write software or a standard, try to avoid them as much as you can.

> **Exercise 41**
>
> Can you find out the strings you need to pass to `get_feature` find find out whether a service supports the nifty `IN_UNIT` function?

Query Generation IV: Casting

Even this ends with an obscure error. Try multitap-broken2.py

Ψ multitap-broken2.py

```
pyvo.dal.exceptions.DALQueryError: Field query: UNION types integer
and text cannot be matched LINE 1: ...S(12), RADIANS(13)), RADIANS(0.1))))
UNION SELECT localid AS...
```

The reason? Idenifier columns are sometimes integers and sometimes texts.

The solution? Cast them all to string.

But: CAST is optional. Oh no!

We could probably get away with just blindly casting, because as long as a service does not support UNION, we can do the casting locally in Python, while services with UNION will probably support CAST, too. But that's just guessing, and this is more about education rather than economy of work.

Query Generation V: Still Casting

```
knows_cast = svc.get_tap_capability().get_adql().get_feature(
            "ivo://ivoa.net/std/TAPRegExt#features-adql-type", "CAST")

for dest_name, ucd, unit, type in RESULT_SCHEMA:
    if type and knows_cast:
        select_clause.append("CAST({} AS {}) AS {}".format(
            perhaps_quote(fieldname_with_ucd(ucd, db_table)),
            type,
            dest_name))

    else:
        # Don't cast and hope for the best
        select_clause.append("{} AS {}".format(
            perhaps_quote(fieldname_with_ucd(ucd, db_table)),
            dest_name))
```

The fallback is of course error-prone: If a table schema would need a CAST but the service cannot do it, we may fail that service. Sometimes best effort is all one can do.

Bringing it all together

After all this preparation, the actual program is trivial except for our usual error handling:

Ψ multitap.py

```
recs = []
svcs_and_tables = get_services_and_tables()
for svc_url, tables in svcs_and_tables:
    try:
        recs.extend(get_rows_for_svc(svc_url, tables))
    except Exception as msg:
        import traceback; traceback.print_exc()
        sys.stderr.write(f"{svc_url} broken (skipped): {msg}\n")

res_table = make_result_table(recs)
res_table.write("all-pms.vot", format="votable", overwrite=True)
with pyvo.samp.connection() as conn:
    pyvo.samp.send_table_to(conn, res_table,
        name="all-pms", client_name="topcat")
```

Exercise 42

There is one glaring hole in our multitap script: Units. Try to improve on this: If the service supports `IN_UNIT`, use it in about the way we have been using CAST.

If you actually need something like this, you can of course also compute the conversion factors locally (using astropy.units) and bake them into the queries. Feel free to try that, too.

11 Odds and Ends

11.1 EPN-TAP

EPN-TAP 1

EPN-TAP is like obscore, just for solar system data. That is: there is a pre-defined schema that you can query on many services in a uniform way. normal VO TAP plus a pre-defined table structure; the tables are always called `epn_core`. Columns of note include:

- `granule_uid` – an identifier for the dataset ("granule" is a word for something like a dataset in solar system sciences).
- `target_name` – what was observed? Regrettably, there are no strict rules for what is called what, so it requires a certain amount of domain feeling to guess how to constrain this.
- `time_min`, `time_max` – when was it observed? Most values in EPN-TAP come as pairs of min and max.
- `c<n>_min`, `c<n>_max` – where is it? Compared to core astronomy, solar system science is plagued by a plethora of coordinate systems. Hence, there is no RA and Dec, but rather three generic coordinate intervals. What they actually mean is given by `spatial_frame_type` (which could be something like "cylindrical"; in the solar system, you have a lot more than just the spherical coordinats that are fine for most of core astronomy) and some identifier for how to interpret these numbers `spatial_coordinate_description` (which would correspond to thing like ICRS or Galactic on the sky). You will need to constrain at least the latter if you expect any sensible result to come out of spatial constraints.
- `dataproduct_type` – the sort of observation. This is like the eponymous column in Obscore, except that these are hashlists of 2-letter codes at this point, defined in the standard itself (Erard and Cecconi et al., 2022) rather than in the product-type vocabulary.
- `instrument_host_name` – the probe or laboratory that produced the data. Again, at this point it is not certain what strings would match a given probe; here, however, there is hope that soon-ish a vocabulary will be produced.
- `instrument_name` – the instrument that produced the data. Again, you have to bascially guess what the instrument is called, and the column may contain a hashlist.

EPN-TAP 2: Hashlists

Many EPN-TAP fields are "hash lists": they are actually multivalued, and to still keep everything in one table, multiple values are concatenated by hashes (#), as in an instrument name like

Visible Infrared Thermal Imaging Spectrometer#VIRTIS

To match such columns, use the `ivo_hashlist_has(hashlist, item)` UDF.

EPN-TAP 3: Global Discovery

Global EPN-TAP discovery means: query all epncore tables. To find these, you have to:

- look for resources containing epncore tables at all and then
- find the tables implementing epncore in them.

To make things even more complicated, essentially all EPN-TAP tables appear twice: Once in a record dedicated to them (with author, title, description for the table itself), once in the TAP service that hosts them. We only want to match the first kind, which for technical reasons is done in pyVO by only accepting a resource record if it has an access mode `tap#aux`.

In code:

```
def iter_epncore_tables(*args, **kwargs):
  for resrec in pyvo.registry.search(datamodel="epntap", *args, **kwargs):
    if not 'tap#aux' in resrec.access_modes():
      continue

    for tab in resrec.get_tables().values():
      utype = tab.utype or ""
      if (utype=='ivo://vopdc.obspm/std/epncore#schema-2.0'
          or utype.startswith('ivo://ivoa.net/std/epntap#table-2.')):
        yield resrec, tab
```

Ψ epnquery.py

This will only work on pyVO later than 1.5, because in 1.5 the table utype was not exposed. In case you wonder what the `yield` statement does: it makes the function a *generator*. This means that you can iterate over its result without having to create a sequence in between.

The inner loop examines the tables published by the resource; tables conforming to EPN-TAP are identified by a utype, which is some characteristic string saying about as much as "something to do with data models". In this case, there are still two substantially different utypes around in the VO, one created during the development of the standard (the one with the vopdc.obspm authority), one for the final IVOA standard. Hence, we need to match against both for the time being. The ivoa.net identifier will also change as future (minor, i.e., compatible) versions of EPN-TAP come around, which is why we do a prefix match. This second constraint is what will be enough in a future when all the legacy services will be updated.

The entire extra function is necessary here because we are not only discovering full resources here (the normal "unit of discovery" in the VO Registry) but have to discover tables on top. I expect pyVO will grow a function that will isolate you from these technicalities in the future; it may be worth perusing the current documentation when you need to do something like this in practice.

Doing something with our results is a bit more complex here than in the, say, obscore case, because EPN-TAP lets people put 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. . .). Hence, in our example we restrict ourselves to simply send any non-empty result to TOPCAT.

In mid-2024, the program will also fail with a syntax error when it hits the VizieR EPN-TAP service, because the do not properly quote their table name; with a bit of luck, this problem will be gone by the time you read this.

Exercise 43

Get the epnquery.py and change it to only discover spectra (that's dataproduct type *sp* in EPN-TAP). then send the first two spectra your program finds to TOPCAT (or SPLAT, or CASSIS, if you have one of them).

11.2 Custom Parameters to Simple Services

Custom Parameters: Discovery

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 does not yet have some API that would properly hide this (not terribly pretty) implementation detail. Worse, it is 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?"
```

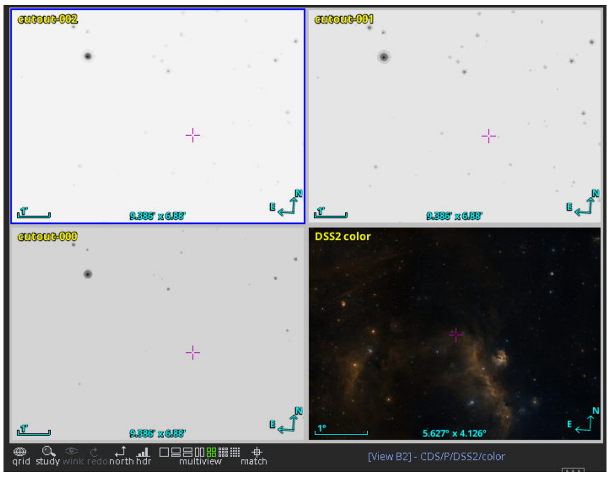
Ψ viewparams.py

Custom Parameters: Usage

Pass custom parameters as keyword arguments to search:

```
svc.search((107, -10), (0.05, 0.05),
  dateObs="57050/58050",
  bandpassId="SDSS i'")
```

Ψ siapextra.py



Custom Parameters: Syntax Trouble

We often have to pass intervals. You need some syntax to write upper/lower limits.

Old-style VO services (most of them) have intervals declared as char[*] or double) and expect min/max.

Others have two simple float parameters with _MIN and _MAX.

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

We are 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.

Exercise 44

The SSAP service at http://dc.g-vo.org/theossa/q/ssa/ssap.xml? houses theoretical spectra mostly of hot, compact stars (think central stars of planetary nebula or perhaps young white dwarfs).

See if you can retrieve three spectra for stars with `log_g` between 4.5 and 5.5, an effective temperature between 7×10^4 and 10^5 Kelvin, and a Nitrogen mass fraction larger than 0.015 dex (write `+Inf` for "no upper limit").

Send the spectra retrieved to splat.

Hints: Use `viewparams.py`, start from `siapextra.py`, remember `dal.ssa.SSAService`, and pass in `FORMAT='VOTable'` to avoid retrieving spectra in both FITS and VOTable.

Use `pyvo.samp.send_spectrum_to`; this needs a URI of the spectrum, which you will find using the `getdataurl` method or what you get back from `search`. Note that current splat-s will not start a SAMP hub themselves, so you will need to start, for instance, TOPCAT first. Feel free to try another spectral client if you want.

You cannot directly use `send_spectrum_to` to send the spectra to TOPCAT, because TOPCAT does not subscribe to spectra. You *could*, however, make an astropy table out of the spectrum using its URL and then `send_table_to` as before.

11.3 TAP Uploads: The right way

Efficient Uploads: The Problem

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.

Efficient Uploads: The Pattern

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"])})
full_result = table.join(
  first_result,
  remote_match.to_table(),
  keys="main_id")
```

Efficient Uploads: Slicing

If you still run into resource limits, you process your data in batches. Use case: retrieve quality measures for Gaia DR3 data by matching on Gaia's `source_id`.

```
def iter_slices(total_length, batch_size):
    limits = list(range(0, total_length, batch_size))+[batch_size]
    for lower, upper in zip(limits[:-1], limits[1:]):
        if lower < upper:
            yield slice(lower, upper)


def remote_match(svc, source_table, remote_table, batch_size, match_column):
    matched_records = []
    match_on = source_table[match_column]

    # only match the match_column (for a positional crossmatch, use
    # an id column (create one if necessary) and the positions).
    for slice in iter_slices(len(source_table), batch_size):
        result = svc.run_sync(
            f"""SELECT a.* FROM
                {remote_table} AS a JOIN
                TAP_UPLOAD.mine AS b
                USING ({match_column})""",
            uploads={"mine": table.Table([match_on[slice]])})
        matched_records.append(result.to_table())

    joined_match = table.vstack(matched_records)
    return table.join(source_table, joined_match, keys=match_column)
```

Ψ smart-tap-upload.py

This example is only *somewhat* contrived: For instance, in the result, you can compare the plain `ruwe` – which says how much you may trust Gaia's solution – with `fidelity_v2` – which says something similar, but *may* be a bit more meaningful, as it takes into account a source's environment –, and you can then look for systematics on, say, magnitudes or parallaxes.

Do not be alarmed by the MergeConflictWarnings; these are because the metadata of the `source_id` column different between the two TAP services participating (ESAC and GAVO) here.

Exercise 45

Add full Gaia records from ivo://esavo/gaia/tap's DR3 `gaia_source` to some records from the hdgaia.main table on GAVO's data centre. This does not need any slicing; still, only upload what you actually need for matching; for that, the smart-tap-upload.py example should be helpful.

Hint: for our simple table.join to work (which needs the same name in both tables), it is probably smart to rename `source_id3` in hdgaia at the ADQL level.

12 Troubleshooting and FAQ

This section collects a few spots of troubles including advice that came up while running the course and did not fit anywhere else.

12.1 TOPCAT and Aladin are unreadably small on HiDPI screens?

Well, that's because Java's native GUI toolkit, Swing, at its core is from way before there were small screens with giant resolutions. Fortunately, for non-antique Java environments, there is a workaround: Set the `GDK_SCALE` environment variable to 2 or so. To try things out, just type

```
GDK_SCALE=2 topcat
```

– if you now have a readable TOPCAT, consider setting the environment variable permanently (or define a shell alias).

12.2 TOPCAT TAP example stays gray?

This is particularly annoying for the Upload Join example. It is a symptom for when TOPCAT cannot work out the columns containing positions in *both* tables to be joined. The most common reason is: you did not select the *table* in the table browser. Just clicking on the *schema* (the top-level list item) is not enough, you need to open the fold and actually click on a table name. You know you're doing it right if you see the table's columns in the corresponding pane.

On the second spot of frequent reasons is that the publishers did not define proper UCDs; the RA and Dec columns need UCDs like pos.eq.ra and pos.eq.dec. If they are missing, complain to the contact address of the server operators.

A Side Track: Terminology

Terminology: Client-Server

Server A machine that runs services

Service A program listening to network requests, processing and answering them according to some standard protocol

Client A program talking to a Service using some standard protocol; perhaps a library, perhaps some polished application, perhaps a bit of cobbled-together curl

Terminology: Data

Dataset An image, spectrum, time series, etc. Ah: Is a catalogue or a catalogue row as dataset? Well: that depends on what you are doing, I'm afraid.

Data Collection A somehow coherent collection of Datasets (e.g., instrument archive, uniformly reduced data, thematic collections)

Metadata Data "about data" (who created it when, why, and how, what's inside,...). Note: one problem's data is another problem's metadata.

B Side Track: Architecture

Decentralised and Federated

The Virtual Observatory is

decentralised – there is no central node, and everyone can run any sort of service – and

federated – each client can talk to all services, and all services can be discovered uniformly.

Actually, both of these statements are whitish lies. We will introduce the one central infrastructure in a second, and whenever there are bugs, a service may work with one client but not some other. That's called *interoperability problem*, and everyone is grateful if you complain about those.

Why no Platform?

We couldn't do a platform if we wanted to.

To start with, getting the sort of international funding we would need is probably close to impossible, and then if we got that, there are so many different sub-disciplines that we'd need a large establishment that would start mainly to deal with itself.

But more importantly: With multiple

interoperable

(i.e., they can be used in a well-defined, uniform way by machines) providers the VO can

grow from the edges:

Users control

their end of processing, operators can adapt services to their needs and

evolve the standards.

No single part can dictate what happens. Not to mention it saves a lot of money if people don't have to write elaborate web pages per project, and if software written for one data collection just works (adapting for differences between instruments, that is) for another data collection.

But then, of course, people can build platforms *on top* of our standards ("APIs").

In a slogan

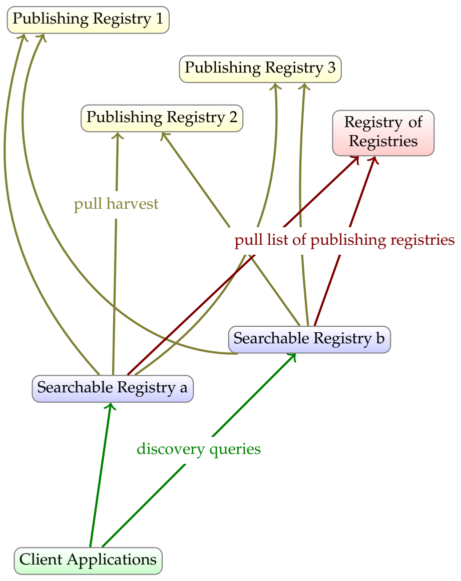
Protocols, not Platforms

. . . and you will not have to fear Elon Musk and his ilk.

Federation in practice: the VO Registry

The VO Registry is what holds everything together: It's what your client asks when you look for, say, "a TAP service with proper motions for stars fainter than 23 mag".

It is a fairly complex system; but it's also an excellent example for what "federation" means.



What's going on here? The "Publishing registries" in the yellow boxes are the different data centres. They have a machine-readable list of their "resources" (i.e., stuff they publish); sometimes just a dozen of those, sometimes, as for VizieR, several tens of thousands.

The clients (green, near the bottom) don't want to go to each of these individually when the look for services; there are currently about 50 of them, and it would suck to have to visit each in turn. Instead, clients talk to *searchable registries*. Anyone can set up one of these (in practice, currently GAVO, STScI, and ESAC run one each). These offer a common client interface, which is called RegTAP.

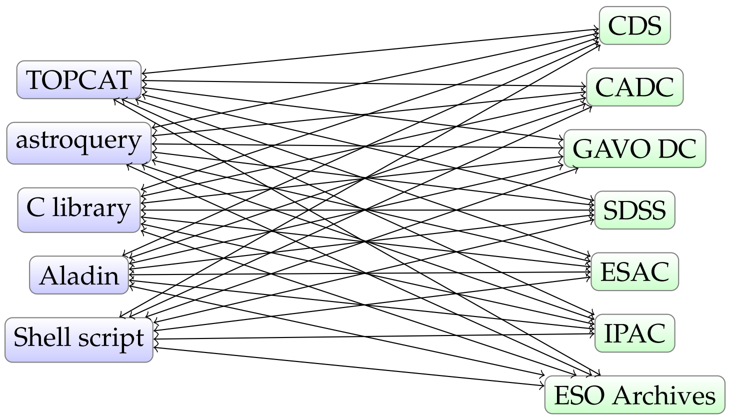
They know about the resources published in the VO because they "harvest" the publishing registries, typically once per day. The machine readable resource lists are transmitted using a non-VO protocol named OAI-PMH.

How do the searchable registries know which publishing registries to harvest? Well, *that's* the one central infrastructure in the VO: the Registry of Registries or RofR (at https://rofr.ivoa.net). You can find lists of registered publishing and searchable registries there. Physically, the RofR is at the CADC in Victoria, BC, at the moment (but it has been in Urbana-Champaign, IL and in Cambridge, MA before).

C Side Track: Standards

Data Access Without Standards

If you want to N clients (programs, say) to communicate with M servers (archives, say), there are $N \cdot M$ things that can go wrong:



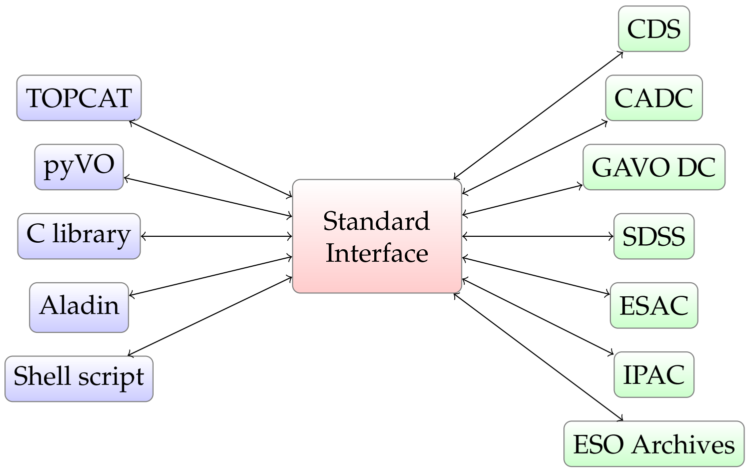
Note that M, the number of servers, is potentially pretty large. Try

```
SELECT DISTINCT gavo_getauthority(access_url)
FROM rr.interface
```

on the GAVO DC TAP server; for me, that's more than 200 different hosts running services. With a few clients in the mix, you'd quickly be up to hundreds of fragile adapter functions that would have to be maintained.

Data Access With Standards

With a standards there's just one thing to get right for each client and server (i.e., $N + M$ sources of brokenness):



IVOA Standards

Alas, one standard does not do it. Of course there's TCP/IP, HTTP, SSL, XML etc. behind the VO start with.

And there are ∼ 50 standards on https://ivoa.net.

As a consumer, you hopefully will not have to read *any* of that.

But things break or folks want to be smart. *Then* it's good to know where to look.

Also, even as a user, you are frequently confronted with names of standards, as they are often used as synonyms for things you may want to do.

Hitch-Hiker's Guide to the IVOA: DAL

The IVOA Data Access Layer Working Group talks about how to locate *data sets* and how to access them in hopefully smart ways:

Searching for data Images (SIAP), spectra (SSAP), objects (SCS), spectral lines (SLAP), generic datasets (ObsCore).

Remote manipulation SODA lets you do cutouts, rescaling, etc., to avoid pulling data you don't need.

Interacting with databases Access using TAP, common query language ADQL.

Hitch-Hiker's Guide to the IVOA: Apps

The applications working group talks about things relevant on the client side:

Formats Table exchange using VOTable, complex spherical geometries with MOC, multiscale images with HiPS.

SAMP Assembling complex environments from simple building blocks.

Hitch-Hiker's Guide to the IVOA: Others

Registry Registry Interfaces for the architecture, VOResource, VODataService, TAPRegExt, SimpleDALRegExt for the metadata format, RegTAP for how to search it.

Semantics Light semantics of physical quantities (UCD), Unit syntax, Vocabulary maintenance.

Grid and Web Services All kinds of invisible support stuff (Authentication, Authorisation, server-side metadata...).

In practice, you have clients and libraries that speak these protocols for you. Most of the time. And even where you directly see things as specified by the IVOA – as in the query language ADQL –, you normally want to learn the stuff from somewhere else than the standards that are typically hard to read. However...

Contributing

If you want to contribute, the IVOA is very open.

- Subscribe to mailing lists: https://www.ivoa.net/members/
- File bugs against standards: https://github.com/ivoa-std
- Improve our vocabularies: https://www.ivoa.net/rdf/
- Come to one of our semiannual meetings, the IVOA Interops.

D Side Track: UCDs

UCDs?

Different catalogues have different names for roughly the same thing. For instance, I found 848 column names containing V-band magnitudes:

> magc, apass_vmag, vmaglan, v74, hip_mag, v55, johnson_mag_v, vmag, mv, vmagapass, vap2, . . .

UCDs, Unified Content Descriptors, let a machine figure out that all of these correspond to *roughly* the same physical concept.

Note the *roughly*: This is not precise semantics intended to uniquely define every physical quantity there is. We would never finish if we wanted to build a vocabulary that could do this. No, these are rough indicators for use in data discovery, exploratory investigation or similar endeavours whose results will, eventually, be filtered through a human mind.

In case you are wondering *where* I found the column names for the V-band magnitudes: The Registry has UCDs for the columns in the various services. They are kept in the RegTAP table `rr.table_column`, and the query I ran was

```
SELECT DISTINCT name FROM rr.table_column
WHERE ucd='phot.mag;em.opt.v'
```

UCDs Have a Grammar

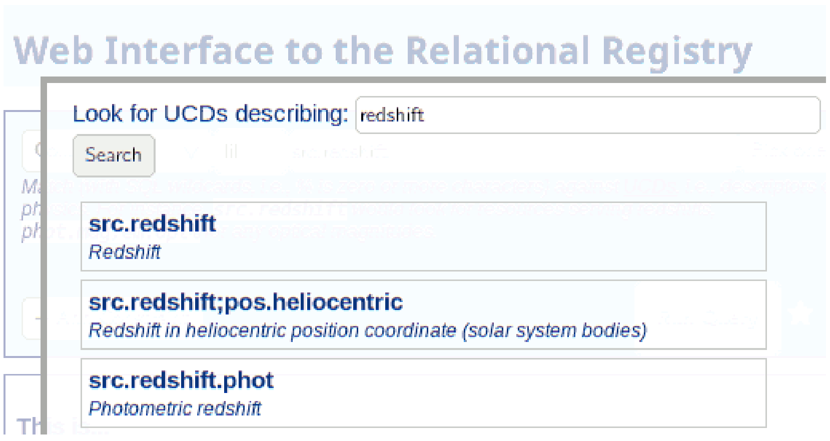
There is a large number of concepts represented in our tables. A single label hence is not enough.

The list of UCDs (Cecconi and Louys et al., 2023) therefore only defines a hierarchy of *atoms* that you can then combine according to some (simple) syntax rules. For instance:

- *phot.mag* is a “Photometric magnitude”
- *em.opt.V* is the “Optical band between 500 and 600 nm”
- *phot.mag;em.opt.V* is something like a visual magnitude

UCDs in Data Discovery

You can discover VO resources offering certain sorts of data using, for instance, WIRR, http://dc.g-vo.org/WIRR:


Web Interface to the Relational Registry

Look for UCDs describing: redshift

Search

src.redshift
Redshift

src.redshift;pos.heliocentric
Redshift in heliocentric position coordinate (solar system bodies)

src.redshift.phot
Photometric redshift

(try Blind Discovery → Column UCD)

Finding UCDs

Probably the best way to find UCDs publishers actually have used for things you are interested in is via the RegTAP table `rr.table_column`, which has a column description in which you can to free-text search:

```
SELECT DISTINCT ucd, column_description
FROM rr.table_column
WHERE 1=ivo_hasword(column_description, 'effective temperature')
```

This of course has many false positives – which is exactly why you should try to assign useful UCDs to your own columns when you publish data.

Exercise 46

Assume you are about to publish a table containing a column that gives the angular size of an object you observed. What would be a good UCD to assign to that column?

Exercise 47

Assume you are about to publish a table containing a column where you subtracted magnitudes (or the same object, of course) in the SDSS u and r bands. Can you come up with a good UCD for that?

E Side Track: Vocabularies

Why Vocabularies?

In many cases, interoperable data publication requires common labels for "things", perhaps even hierarchically organised:

- Subject keywords (as in journals)
- Reference frames (ICRS, etc), time scales, and the like
- Sorts of data products ("I need a spectrum")
- Parts of the spectrum ("Near Infrared"?)

- Object types (“AGN” or “Active Galactic Nucleus”?)
- Relations between resources (Cites, Replaces, . . .)

These must be machine-readable, and people need to be able to extend and evolve them without too much strife.

In the VO

In the VO, a standard called “Vocabularies in the VO” (Demleitner and Gray et al., 2023) says how we are doing it:

- You can get vocabularies at http://www.ivoa.net/rdf
- Full identifiers continue with `<vocname>#<concept-id>`
- e.g., http://www.ivoa.net/rdf/uat#astronomy-education, which resolves in your browser
- Vocabularies are retrievable in various RDF (Resource Description Framework, the basis of the semantic web) formats, and
- desise, dead simple semantics – a trivial JSON serialisation that allows one to use the vocabularies with minimal tooling
- Develop vocabularies in a community process using VEPs – Vocabulary Enhancement Proposals, semistructured documents arguing why some concept you propose is a good idea

In Instance Documents

While in mainstream RDF, you mostly have full URIs, in the VO, we usually only use identifiers, e.g.,

- *datalink/core:* `#progenitor` in the semantics column of datalink documents – the leading hash is a bit of cleverness where we say “it’s a URI relative to the vocabulary URI”, so you *could* use non-IVOA terms by writing full URIs. But that is not a smart thing to do in general, because datalink clients will not understand what you mean.
- *refframe:* `<COOSYS system="ICRS"/>` in VOTable
- *product-type:* `image` in Obscore’s `dataproduct_type` column
- *relationship_type:* `IsServedBy` in VOResource’s relationship
- *uat:* `abundance-ratios` in RegTAP’s `res_subject` column.

Machine Readable

IVOA vocabularies can be consumed in a trivial JSON format. Just request the vocabulary URI asking for the `application/x-desise+json` media type:

```
$ curl -LH "accept: application/x-desise+json" \
http://www.ivoa.net/rdf/timescale
{
  "uri": "http://www.ivoa.net/rdf/timescale",
  "flavour": "RDF Class",
  "terms": {
    "TAI": {
      "label": "International Atomic Time TAI",
      "description": " atomic time standard, TT-TAI = 32.184 s.",
      "wider": [],
      "narrower": []
    },
    "TT": {
...
```

In pyVO

In PyVO, use `get_vocabulary`; this will let you easily find out whether terms are in the vocabulary, their labels and descriptions, and narrower and wider terms:

```
>>> v = pyvo.utils.vocabularies.get_vocabulary("datalink/core")
>>> "preview" in v["terms"]
True
>>> "rearview" in v["terms"]
False
>>> v["terms"]["documentation"]["description"]
'Structured or unstructured metadata helping to understand, interp...
>>> v["terms"]["calibration"]["narrower"]
['bias', 'dark', 'flat']
```

In some places, code doing things like these are already built into pyVO; for instance, there is `bysemantics` on datalink results; see sect. 9 for details.

In ADQL

Some TAP services have the `gavo_vocmatch(voc, term_id, col)` UDF built in. For instance, to look for everything roughly image-like in an obscore table, you can do:

```
SELECT dataproduct_type, access_url
FROM ivoa.obscore
WHERE DISTANCE(s_ra, s_dec, 10, 10)<1
  AND 1=gavo_vocmatch('product-type',
    'spatially-resolved-dataset', dataproduct_type)
```

Exercise 48

The constraint

```
1=gavo_vocmatch('product-type',
 'spatially-resolved-dataset', dataproduct_type)
```

that we have put in on the vocabularies in ADQL slide claims to match *spatially-resolved-dataset* and all narrower concepts. Can you give an equivalent expression of the form

```
dataproduct_type IN ('...', ....)
```

based on the product-type vocabulary? And why is that less desirable than using the UDF?

F Side Track: VOTable

The VO's Native Table Format: VOTable

Most tables in the VO are transported as VOTables. These are XML files with (potentially) rich metadata, for instance:

```
<VOTABLE xmlns="http://www.ivoa.net/xml/VOTable/v1.3" version="1.4">
  <DESCRIPTION> The catalogue ARIHIP has been constructed by selecting the 'best
  [...]</DESCRIPTION>
  <RESOURCE type="results">
    <INFO name="QUERY_STATUS" value="OK"/>
    <INFO name="request" value="/arihip/q/cone/scs.xml?RA=333&amp;DEC=43&amp;SR=2"/>
    <INFO name="standardID"
      value="ivo://ivoa.net/std/ConeSearch">DaCHS 2.9.2 SCSRenderer</INFO>[...]
    <COOSYS ID="system" epoch="J2000.0" system="ICRS"/>
    <TABLE name="result">
      </FIELD>
      <FIELD ID="hipno" arraysize="*" datatype="char" name="hipno" ucd="ID_MAIN">
        <DESCRIPTION>Number of the star in the HIPPARCOS Catalogue (ESA 1997).</DESCRIPTION>
      </FIELD>
      <FIELD ID="raj2000" datatype="double" name="raj2000" ref="system"
          ucd="pos.eq.ra;meta.main" unit="deg">
        <DESCRIPTION>Right ascension from a single-star solution</DESCRIPTION>
      </FIELD>
      <DATA>
        <BINARY>
          <STREAM encoding="base64">P8ZMQ7q5V6gAAAAGMTA5[...]</STREAM>
        </BINARY></DATA>
    </TABLE>
  </RESOURCE>
</VOTABLE>
```

The example VOTable actually is an excerpt of the response of a Cone Search service; you can run it by pulling http://dc.g-vo.org/arihip/q/cone/scs.xml?RA=333&DEC=43&SR=2.

Many VO services can be run (albeit in a rather reduced fashion) using generic web tools, such as curl (available everywhere) and perhaps xmlstarlet, a nice program to make XML a bit more human-readable. On a suitably equipped POSIX system, you could try:

```
$ curl "http://dc.g-vo.org/arihip/q/cone/scs.xml?RA=333&DEC=43&SR=2" \
  | xmlstarlet fo | less
```

Admittedly, the document shown has a lot of characters looking very computerish. It still pays to have an idea of how a VOTable looks like for at least three reasons:

- When creating VOTables yourselves (and sooner or later you will), it is good to have an idea of what you may possibly want to communicate to the consumers of your data.
- When things go wrong, it is nice to be able to be able to eavesdrop into what the machines tell one another – and also to make sense of error messages.
- Abstractions leak (that's nerdspeak for: it's quite common for user interfaces to reflect the underlying technology). If you know VOTable, it is easier to understand APIs and user interfaces.

Let me hence go through the various items making up a VOTable.

VOTable: Top-Level Declarations

```
<VOTABLE xmlns="http://www.ivoa.net/xml/VOTable/v1.3" version="1.4">
  <DESCRIPTION>The catalogue ARIHIP has been constructed by selecting
  the 'best data' for a given star from combinations of HIPPARCOS data
  with Boss' GC and/or the Tycho-2 catalogue as well as the FK6. It
  provides 'best data' for 90 842 stars with a typical mean error of
  0.89 mas/year (about a factor of 1.3 better than Hipparcos for this
  sample of stars).</DESCRIPTION>
```

- Anything within `<...>` in XML is called a *tag*. A tag has a name and perhaps attributes.
- An opening tag, some content, and a closing tag make up an XML *element*.
- Elements within another element are called its *children*.

The opening tag in the snippet says it is a VOTable of version 1.4. VOTable is a standard that continually evolves. As a user, you should not notice much of that, but certain features only become available in newer versions. For instance, VOTable 1.4 introduced an element called TIMESYS to declare metadata on times.

The odd *xmlns* declaration is deeper nerdstuff that I included only in order to tell you to ignore it. The `v1.3` in there is a totally red herring. If you *really* want to know the full story, you could read Harrison and Demleitner et al. (2018) and learn something about the difficulties of writing standards for globally developed software: you can't always correct mistakes and then need ugly hacks to work around them.

The *DESCRIPTION* element contains human-readable information on what to expect further down. Since few clients show that description prominently, most data providers are not very diligent here.

VOTable: result Resources

```
<RESOURCE type="results">
  <INFO name="QUERY_STATUS" value="OK"/>
```

- A VOTable consists of *RESOURCE*-s.
- All current DAL protocols return a *RESOURCE* of *type* *results* with the main table.
- The *INFO* with the name QUERY_STATUS is a DAL-mandated machine-readable success indicator. This could also indicate an error (and then includes an error message) or an overflow, when there was more data that was not returned for one reason or another.

VOTable: Light Provenance

```
  <INFO name="request"
    value="/arihip/q/cone/scs.xml?RA=333&amp;DEC=43&amp;SR=2"/>
  <INFO name="standardID" value="ivo://ivoa.net/std/ConeSearch"
    >DaCHS 2.9.2 SCSRenderer</INFO> [...]
  <INFO name="publication_id" value="2001VeARI..40....1W"
    >A bibliographic source citable for (parts of) this data</INFO>
  <INFO name="contact" value="gavo@ari.uni-heidelberg.de"
    >Contact option</INFO>
```

Provenance is information on how some artefact came to be. It is mighty useful when debugging or trying to reproduce something one did yesterday. Not to mention last year.

In TOPCAT, see Views/Table Parameters:

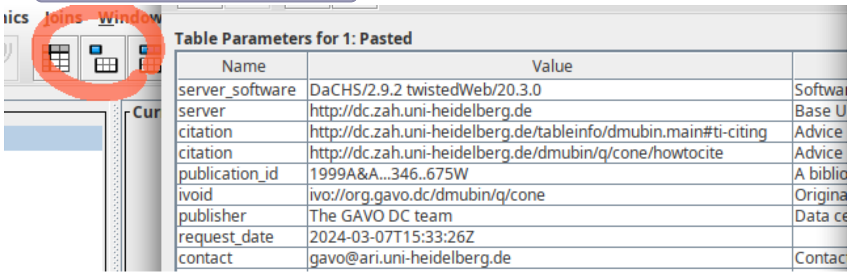


Table Parameters for 1: Pasted

Name	Value	
server_software	DaCHS/2.9.2 twistedWeb/20.3.0	Softwa
server	http://dc.zah.uni-heidelberg.de	Base U
citation	http://dc.zah.uni-heidelberg.de/tableinfo/dmubin.main#ti-citing	Advice
citation	http://dc.zah.uni-heidelberg.de/dmubin/q/cone/howtocite	Advice
publication_id	1999A&A...346..675W	A biblio
ivoid	ivo://org.gavo.dc/dmubin/q/cone	Origina
publisher	The GAVO DC team	Data ce
request_date	2024-03-07T15:33:26Z	
contact	gavo@ari.uni-heidelberg.de	Contac

Data providers are rather free to put into these *INFO* items (and the related *PARAM* elements, which work like constant *FIELD* elements) whatever they want, and most are still rather stingy.

But when they are there, they are a treasure trove where you may get advice on citing things, pointers to further information (`reference_url`), the original request and so on. The IVOA Note on "Data Origin" (Landais and Muench et al., 2024) tries to work towards a more standardised set of such pieces of information.

FIELDs of a Table

```
<FIELD ID="hipno" arraysize="*" datatype="char" name="hipno"
    ucd="meta.id;meta.main">
  <DESCRIPTION>Number of the star in the HIPPARCOS Catalogue (ESA 1997).
  </DESCRIPTION>
  <VALUES><MIN value="1"/><MAX value="120404"/></VALUES>
</FIELD>

<FIELD ID="parallax" datatype="float" name="parallax" ucd="pos.parallax"
    unit="deg">
  <DESCRIPTION>Parallax used in deriving the data of the star in the
    catalogue selected for the ARIHIP. This is either the HIPPARCOS
    parallax or a photometric/spectroscopic parallax (see
    Kp).</DESCRIPTION>
  <VALUES><MIN value="-8.216667e-06"/><MAX value="0.00015250278"/></VALUES>
</FIELD>
```

The main table metadata in VOTable is in *FIELD* elements. They give names, types, units, UCDs, value ranges.

For more on these UCDs, see the the UCD sidetrack D.

Note that VOTable at this point still does not have a string datatype but instead models strings like *hipno* here as arrays of characters, which keeps resulting in multiple headaches but is really hard to fix. In this case, for instance, the range of identifiers is not quite right: For array-valued *FIELD*s, *MIN* and *MAX* should really give the ranges of individual elements rather than the whole string.

It's often seemingly simple things that are really hard to fix in distributed systems.

When you write your units, please make sure you use the right syntax so computers can read them and convert your values as appropriate. The corresponding standard, VOUnits (Gray and Cecconi et al., 2023), is rather readable.

FIELDs in TOPCAT

In TOPCAT, use Views/Column Info to inspect the metadata from the *FIELD*s.

Note that you can sort by all the various columns, which is particularly nifty for UCDs:

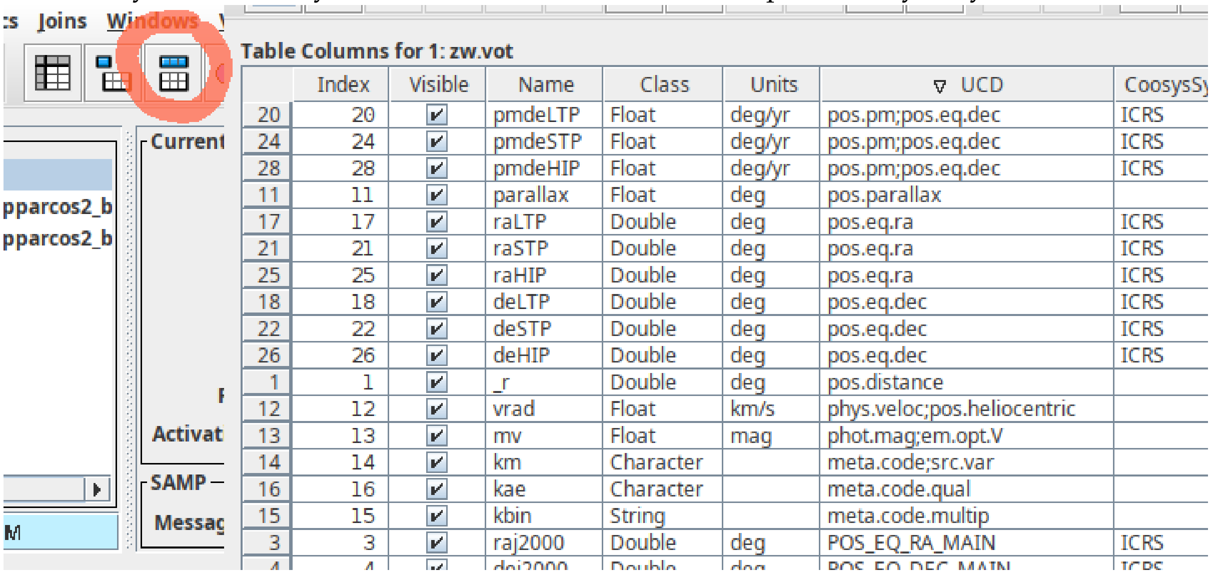


Table Columns for 1: zw.vot

	Index	Visible	Name	Class	Units	UCD	CoosysSy
20	20	✓	pmdeLTP	Float	deg/yr	pos.pm;pos.eq.dec	ICRS
24	24	✓	pmdeSTP	Float	deg/yr	pos.pm;pos.eq.dec	ICRS
28	28	✓	pmdeHIP	Float	deg/yr	pos.pm;pos.eq.dec	ICRS
11	11	✓	parallax	Float	deg	pos.parallax	
17	17	✓	raLTP	Double	deg	pos.eq.ra	ICRS
21	21	✓	raSTP	Double	deg	pos.eq.ra	ICRS
25	25	✓	raHIP	Double	deg	pos.eq.ra	ICRS
18	18	✓	deLTP	Double	deg	pos.eq.dec	ICRS
22	22	✓	deSTP	Double	deg	pos.eq.dec	ICRS
26	26	✓	deHIP	Double	deg	pos.eq.dec	ICRS
1	1	✓	_r	Double	deg	pos.distance	
12	12	✓	vrad	Float	km/s	phys.veloc;pos.heliocentric	
13	13	✓	mv	Float	mag	phot.mag;em.opt.V	
14	14	✓	km	Character		meta.code;src.var	
16	16	✓	kae	Character		meta.code.qual	
15	15	✓	kbin	String		meta.code.multip	
3	3	✓	raj2000	Double	deg	POS_EQ_RA_MAIN	ICRS
4	4	✓	dej2000	Double	deg	POS_EQ_DEC_MAIN	ICRS

VOTable: The STC Drama

Regrettably, the annotation of space-time coordinate metadata in VOTables is still woefully inadequate:

```
<COOSYS ID="system" epoch="J2000.0" system="ICRS"/>
<COOSYS ID="system-02" epoch="J2000.0" system="ICRS"/>
  <FIELD ID="raj2000" datatype="double" name="raj2000" ref="system"
      ucd="pos.eq.ra;meta.main" unit="deg">
    <DESCRIPTION>Right ascension from a single-star solution</DESCRIPTION>
  </FIELD>
  <FIELD ID="dej2000" datatype="double" name="dej2000" ref="system"
    ucd="pos.eq.ra;meta.main" unit="deg"/>
  <FIELD ID="pmra" datatype="float" name="pmra" ref="system"
    ucd="pos.pm;pos.eq.ra" unit="deg/yr"/>
  <FIELD ID="pmde" datatype="float" name="pmde" ref="system"
    ucd="pos.pm;pos.eq.dec" unit="deg/yr"/>
  <FIELD ID="raLTP" datatype="double" name="raLTP" ref="system-02"
    ucd="pos.eq.ra" unit="deg"/>
```

About the most basic thing a computer would want to do with a star catalogue is to transform it to a different epoch; perhaps to match it to some other catalogue, perhaps to precisely plot it over an image of the sky (which was taken off the catalogue epoch).

Regrettably and despite many efforts by yours truly, this still is not really possible based on standard VOTable metadata. If you are courageous, you can collect *FIELD*-s referencing the same *COOSYS* and guess their roles (position, proper motion, parallax. . .) based on the UCDs. But that's so brittle and ugly that not much software actually does this (Aladin being an exception).

It also immediately fails in cases like this, where the positions (raj2000, dej2000) belong to two coordinate sets for which only the proper motions are different.

Sorry about this. This is probably the worst failure of the IVOA's consensus-based decision model.

VOTable: The Data

```
<DATA>
  <BINARY>
    <STREAM encoding="base64">P8ZMQ7q5V6gAAAAGMTA5NTExQHT...
```

VOTable can encode tabular data in different ways. Most importantly:

- *TABLEDATA* – more or less human-readable values in *TD* and *TR* elements. Nice, for instance, to format using XSLT.
- *BINARY* – FITS-like binary data made XML-clean using base64.
- *BINARY2* – the successor to *BINARY*, mainly fixing the representation of missing values.

Here, the service has chosen to return *BINARY* data.

Many services let you influence what kind of VOTable you get back, typically by passing a `RESPONSEFORMAT` parameter. Regrettably, there is no widely accepted standard as to what different formats are called. On the particular service we have used here (and others based on DaCHS), *BINARY2* is called **votableb2** and *TABLEDATA* is called **votabletd**. So, if you want to get "readable" data here, you can say

```
$ curl "http://dc.g-vo.org/arihip/q/cone/scs.xml?RA=333&DEC=43&SR=2&RESPONSEFORMAT=votabletd"
```

Exercise 49

Get the VOTable at http://dc.g-vo.org/arihip/q/cone/scs.xml?RA=333&DEC=43&SR=2&RESPONSEFORMAT=votabletd and add to the value of the `publisher` *INFO* a (alas, hypothetical) **(note to self: they were on holiday in May 2024)**, using a text editor. If you have xmlstarlet, try re-formatting it first.

Ensure that the edit actually happened using TOPCAT. There, edit the note, too (doubleclick) and then see if you can see the change in the text editor.

Exercise 50

Again in our VOTable, use a text editor to add an *INFO* element with a name of **(yourname)-note**, a value of **(today's date): learned how to add INFO elements the hard way**, and a content of **private processing note**. Load your modified table into TOPCAT to ensure you have not damaged the file and the information is there.

(Just to be sure: This is *not* how you should add such infos in the wild. Astropy, for instance, offers ways to do this kind of thing (although it's harder than it should be). Leaving notes about your operations in such files, however, is a good idea in principle.

Exercise 51

Still in our VOTable, set the `vrad` field for the object with the Hipparcos number 109481 to the value -16.268137 (which is what Gaia DR3 gives for this object).

Again, try it once with TOPCAT and once with a text editor. For the latter, you will need the table to be in *TABLEDATA* format. At your option, use the RESPONSEFORMAT parameter or just save the table as *TABLEDATA* from TOPCAT.

G Side Track: IVOA Identifiers

Ivoids as URIs

The primary identifier for resources in the VO is the IVOA identifier or *ivoid*; it is also what you always implicitly join on in RegTAP.

They are URIs with an ivo scheme:

```
ivo://<authority>[/<local-part>][?<query-part>][#<fragment>]
```

The *authority* is a short name for who has created an ivoid; each authority must only be used by one institute and managed by one registry. The local part is a path-like thing that the authority uses to keep all their resources apart.

Ivoids regrettably must be compared case-insensitively; the best thing to do is to lowercase them as soon as you get them.

Resolving Ivoids

Ivoids can be resolved to registry records.

One way to do so is to prepend http://dc.g-vo.org/I/ to them.

Ivoids without local parts point to authorities: http://dc.g-vo.org/I/ivo://cds.vizier.

Ivoids with local parts mostly point to services: http://dc.g-vo.org/I/ivo://org.gavo.dc/bgds/l/ssa.

Actually, there several kinds of resources in the VO registry in addition to the services and authorities already mentioned. There are Registries[12] (which are, perhaps somewhat confusingly, used in keeping *the* VO Registry up to date), Organisations[13] (which are not very useful), documents[14] (which point to tutorials and the like; this is what http://dc.g-vo.org/VOTT is built from), standards[15] (which are, for instance, used to identify capabilties adhering to them), and actually several different sorts of service records. But a normal VO consumer does not need to care about resource types; the only important thing is that they agree on what the string is.

Special IVOIDs

Publisher DIDs: These are hopefully globally unique identifiers for datasets as used in datalink or obscore.

They *should* have the form

```
<ivoid-of-service-resolving-them>?<dataset-key>
```

If they are built like that, http://dc.g-vo.org/glopidir can resolve a PubDID to the dataset.

Standard IDs Fragment identifiers are supposed to be resolved into standard keys, and these, in turn, are used to define some standard features in the VO. Example:

ivo://ivoa.net/std/tapregext#upload-inline

[12] Example: ivo://gaia.aip.de/registry
[13] Example: ivo://ivoa.net/ivoa
[14] Example: ivo://edu.gavo.org/hd/arvo_dfbs
[15] Example: ivo://ivoa.net/std/sia

This ivoid, used in the right place in the capabilities document of a TAP server, informs a client that a TAP service supports table uploads.

In practice, the standard identifiers used to say "this is an SSAP service" (say) just reference the full standard rather than a standard key; that is really not a problem because no machine actually resolves the standard ids in capability elements.

H Solutions for Most of the Exercises

Solution for Exercise 1 A query you could have used against the Einstein catalogue is

```
SELECT lx, ra, dec, cluster_radius, name
FROM eingalclus
```

Again, the catalogue is so small that you can pull it in full.

The radius-luminosity plots do not have much in common with each other. My hypothesis on the reason is primarily that the Einstein catalogue is flux-limited (the instrument sensitiviy), whereas mcxc probably has tried to gather a full sample with a completely different selection function. But, as the comparison of the radii shows, different definitions of a cluster radius probably play a role, too.

Since the centres of clusters of galaxies are not terribly well-defined, I have chosen a match radius of 10 arcminutes for the crossmatch between the two catalogues (the results do not appreciatably change when you vary that within reason).

Good descriptions would let you guess details on the radius definitions. The way things are, you will have to go back to the papers to figure this out (getting to these papers is less straightforward than it should be in TOPCAT; we will get to that later). The X-ray-luminosities, on the other hand, are well-correlated; that's how this kind of thing should look like.

Solution for Exercise 2 For noise reduction, turning the condition for fancy morphologies from the lecture notes around is probably a good thing; actual quasars in SDSS will probably appear pointlike regardless of whether or not they are lensed most of the time (exception: unresolved double images). So, what about `abs(psfmags[5]-petmags[5])<1`?

Solution for Exercise 3 The column names are phot_g_mean_mag for brightness, phot_bp_mean_mag for the optical B-band and phot_rp_mean_mag for the optical R-band.

Solution for Exercise 4 X Persei is a high-mass X-ray binary system. These are usually formed from a massive star and a compact object like a neutron star or a black whole. You could figure this out by selecting the SIMBAD plane, which then will show you a "HMXB" in the images, and after clicking on that, you can use the link at the bottom of the viewer to open the selected object on the SIMBAD web page.

Solution for Exercise 5

```
SELECT TOP 20 *
FROM fk6.part1
ORDER BY vmag ASC
```

Solution for Exercise 6

```
SELECT TOP 20
5+5*LOG10(pres*3600.)+vmag AS absmag, comname
FROM fk6.part1
ORDER BY vmag ASC
```

Solution for Exercise 7 Just add a `WHERE pres>0`. In serious science, one would of course need to be more careful; there is a reason, after all, for the negative parallaxes (at least with frequentist estimators, but really with any kind of measurement).

Solution for Exercise 8 Inspecting TOPCAT's metadata browser, you will find that the radial velocity in `cns5.main` is called `rv`. With this, you can write

```
SELECT COUNT(*) FROM cns5.main WHERE rv IS NULL
```

This will yield just one row containing 4323. If you try the inverse, `rv IS NOT NULL`, you will see that a mere 1586 objects do have a radial velocity; RVs are expensive.

Solution for Exercise 9

```
SELECT
  ROUND(Jmag) AS bin,
  COUNT(*) AS n,
  AVG(SQRT(POWER(pmRA,2)+POWER(pmDE,2))) AS pmavg
FROM lspm.main
GROUP BY bin
ORDER BY bin
```

Solution for Exercise 10 The query would look something like

```
SELECT
  COUNT(*) as n,
  AVG(teff_k) AS mean_teff,
  ivo_healpix_index(5, raj2000, dej2000) AS hpx
FROM rave.main
GROUP BY hpx
```

When plotting this, remember to do Layers → Add HEALPix Control, and select your table in the Data tab. Also, you still need to manually set the HEALPix Data Level to 5, or the plot will look really odd (and not mean a thing).

As to what the structures mean: The survey largely excluded the Galactic plane, presumably to dodge blending. That there's almost no data on the northern sky is because the RAVE instrument is on the southern hemisphere.

The structures in the density plot… well, who knows what made the survey designers pick their objects – I'm sure there is a paper discussing this. On the structures in the temperature plot: I'd guess the hot patches in the galactic plane are open clusters. The "brighter" stripes along the Galactic plane I would attribute to something happening in the pipeline by gut feeling; but of course it could also represent the target selection ("thick disk sample"?).

Also, if you go to higher HEALPix indexes, remember to raise MAXREC, which on the GAVO server defaults to 20'000 – less than the number of level 6-HEALPixes on the sky.

Solution for Exercise 11 The somewhat tricky part is to pull in the CNS columns into your result, because you cannot say `SELECT whatever, *` in SQL . There is a workaround, like this:

```
SELECT
  ivo_epoch_prop_pos(ra, dec, parallax,
    pmra, pmdec, rv, epoch, 2150) AS np, cns.* from cns5.main as cns
WHERE ra IS NOT NULL
```

You did check the units of the columns going into `ivo_epoch_prop_pos`, did you?

The condition on ra is necessary because the UDF refuses to operate if only one column has a NULL position; and the CNS contains the Sun, which does not have a usable position.

There are various ways to seek out Sirius in the transformed catalogue (e.g., by looking up its position and clicking on a sky plot). A snobbish way is to use another UDF that the GAVO server has: `gavo_simbadpoint`, which returns a point from Simbad's idea of an object's position. This would look like this:

```
SELECT
  ivo_epoch_prop_pos(ra, dec, parallax,
    pmra, pmdec, rv, epoch, 2150) AS np, cns.* from cns5.main as cns
WHERE DISTANCE(POINT(ra, dec), gavo_simbadpoint('Sirius'))<0.01
```

This will give something like 101.26339444 for Sirius A's RA. And the "A" also tells you why this is going to be severely off: Sirius is a binary star, and its A component wobbles quite a bit.

I was too lazy to look for an orbit of Sirius, which one would need to make a better prediction. If you are less lazy, feel free to write in. n.

Solution for Exercise 12

```
SELECT ah.vrad, r.rv, r.raveid, ah.hipno
FROM rave.main AS r
JOIN arihip.main AS ah ON (
  1=CONTAINS(
    POINT(r.raj2000, r.dej2000),
    CIRCLE(ah.raj2000, ah.dej2000, 0.001)))
```

Solution for Exercise 13 The central difference is that the EXISTS query will have not more than one row per RAVE object; that's how SELECT works: either a row is in or it is not.

The JOIN however, my produce more than one row per RAVE object if there is more than one ARIHIP object in the close vicinity of the RAVE object – and given there are many double stars resolved already in the Hipparcos catalogues, that's a fairly common thing.

Solution for Exercise 15 To find out which object is missing, do Joins → Pair Match in TOPCAT; thanks to the UCDs, TOPCAT fills out the dialogue just fine, except that in Join Type, you have to choose 1 not 2.

This results in a single-row table for a star at 316.61589, 38.67332.

Given the way the table was produced, the only plausible explanation is that the star is fast *and* has a fairly large epoch difference; indeed, if you look at a histogram of *epoch* in matchme.vot, you will see that it is on the larger side. To see if that explanation is right, just re-run our original query, uploading the new difference table, and raising the initial match radius (i.e., adapt `tap_upload.tx` to the index of the match result, and write, perhaps 0.2 instead of 0.1).

This will return to the object with the Gaia DR3 source id 1872046574983497216, and indeed this has a massive proper motion of 5.2 arcsec/yr, which over the roughly 100 years of the epoch difference is 0.13 degrees; so, it *just* escaped our initial wide cone. That we missed *such* an extreme star is no reason to worry; there are not that many of those on the sky (and of course I have crafted the original query to contain one of them).

See also the next exercise.

Solution for Exercise 16 We have the wisdom of all astronomy at our fingertips, so go back to Select Service, type **high proper motion** into Keywords and see if you can find a good source for a statistics on fast stars.

Careful with some of the VizieR results that you get back; the table descriptions often suggest that something is a fairly comprehensive catalogue when it actually is not.

lspm.main at the GAVO data centre says something about completeness. It's just for the northern hemisphere, but for our statistical curiosity, that is good enough; high-PM stars are nearby, and thus we expect them to be roughly isotropically distributed.

By the Table pane, there are more than 60'000 objects in this table. Let's not pull them all but instead do a server-side histogram, perhaps choosing 0.1 arcsec/yr as the bin size:

```
SELECT
  COUNT(*) AS n,
  ROUND(pm*3600*10)/10 AS bin
FROM lspm.main
GROUP BY bin
```

Looking at the histogram, you will see that there are less than five stars faster than our runaway on the northern sky, and hence probably less then ten on the entire sky.

As I said: It's no accident that this one appeared in our sample. Try SAMP and Aladin's Simbad pointer if you want to find out that object's name.

Solution for Exercise 17 The table metadata tell you that both pmra and pmdec are indexed. However, you cannot use these indexes to query against total proper motion, which is a complex expression over these. Instead, you have to use the index to pick out stars with large PM *components* and then do your computations on that far smaller set. Perhaps:

```
SELECT source_id, SQRT(POWER(pmra,2)+POWER(pmdec,2)) AS pmtot
FROM gaia.dr3lite
WHERE NOT pmra BETWEEN -1000 and 1000
`AND NOT pmdec BETWEEN -1000 and 1000
```

Sorting this by pmtot, you will find that our friend 1872046574983497216 holds rank 7 among the 1.8 billion stars in Gaia DR3. What, may I quip again, are the chances for such a thing turning up in my not-so-random sample?

Solution for Exercise 18 Sorry, this exercise was really intended just to make you watch UWS phases and go through the motions of resuming. No astronomy here. But save the table, we will later be doing something interesting with it.

If you could not resume, you probably forgot to uncheck *Delete on Exit*.

Oh, but you may want to plot the spectra you selected. To do that in TOPCAT, open a Plane Plot and then do Layers → Add XYControl. In the *Position* tab, select your table; modern TOPCATs will automatically know how to plot this as a spectrum.

Solution for Exercise 19 The queries are fairly straightforward, except perhaps for the UCD thing, where you want to use an ILIKE operator that does case-insensitve matching because (stupidly) UCDs are specified to be case-insenstive. But I have not told you about this, and so you were not supposed to know that.

```
SELECT COUNT(*) FROM tap_schema.tables

SELECT COUNT(*) FROM tap_schema.columns

SELECT COUNT(*) FROM tap_schema.columns
WHERE ucd ILIKE 'phot.mag%'
```

The results change to often to include them here.

Solution for Exercise 20 Step one is to obtain positions for the stars, i.e., turn the `source_id`-s into positions. As hinted, this takes an upload join with a Gaia source table, for instance:

```
SELECT source_id, ra, dec
FROM gaia.dr3lite
JOIN tap_upload.t1 USING (source_id)
```

(as usual, modulo the TOPCAT table index).

Pro tip: before uploading, open the column metadata for the table you are going to upload and uncheck all columns you will not need in the query (in this case, flux). TOPCAT then will not upload it, so things will be faster and less fragile on top.

The second step is as in the lecture: change to Simbad's TAP server and, while having the result of the last query selected, create an Upload Join from Examples. Perhaps reduce the match radius a bit to get something like

```
SELECT TOP 1000
       *
FROM basic AS db
JOIN TAP_UPLOAD.t3 AS tc
  ON 1=CONTAINS(POINT('ICRS', db.ra, db.dec),
                CIRCLE('ICRS', tc.ra, tc.dec, 1./3600.))
```

You will see that basically all of these stars are late M stars, whether classified as long-period variables, carbon stars, Mira variables, or whatever; what you are seeing in the spectra is presumably wide molecular absorption features.

By the way, you *could* have tried to turn your `source_id`-s into identifiers that Simbad supports and that way avoid the resolution step. To do that, you first have to figure out Simbad's syntax for writing these identifiers. There is probably good documentation on that somwhere, but lazy bum that I am I made a bold guess and tried:

```
SELECT TOP 20 * FROM ident WHERE id LIKE 'Gaia%'
```

What came back contained strings like “Gaia DR2 1853339484138990848”, which suggested I ought to prepend “Gaia DR3” to my `source_id`-s. Trouble is: they are long integers, so ADQL’s concatenation operator can’t *really* be expected to work. But ADQL is fairly weakly typed, so I gave it a try:

```
SELECT * FROM
basic AS b JOIN ident AS i ON (b.oid=i.oidref)
JOIN tap_upload.t2
ON (id='Gaia DR3 ' || source_id)
```

Sure enough, Simbad’s database engine turned `source_id` into a string: Success by weak typing!

This yields (for my particular result of the `TOP 500` from the XP spectra, so this might be different for you) 436 rows versus 440 for the positional crossmatch. Simbad thus seems to be essentially complete on Gaia ids.

Doing a TOPCAT pair match on `main_id` (with “1 not 2”) gives six objects missing from the id-based match. Some of them seem genuine misses (they are long-period variables), some (like a planet candidate) are definitely false positives on the positional match.

But since six are missing from the id match, the positional match must be missing two objects, too. Matching with “2 not 1” shows these, and both of them are high-proper motion stars. We missed them due to our restrictive match radius (and raising it would have increased the false positive rate, so this is not a recommendation to make it larger) and the epoch difference between Simbad and Gaia DR3.

Hence, in *this particular* case id-based matching would probably have given the “better” result; but you can almost always do positional matching, and you don’t need to do guesses on the form of the identifiers. Consider this little excursion another reminder that you always have noise.

Solution for Exercise 21 The basic query is

```
SELECT SUM(MOC(8, s_region)) AS tot_cov FROM emi.main
```

The result is plottable with the usual Layers → Add Area Control.

Computing the area is a bit more complicated than it would need to be because at least at the time of writing the server used the official grammar of area, and that does not admit aggregate functions as arguments. Hence,

```
SELECT AREA(SUM(MOC(8, s_region))) FROM emi.main
```

is a syntax error even though it would make total sense. But you can work around it in this way:

```
SELECT AREA(tot_cov) FROM (
  SELECT SUM(MOC(8, s_region)) AS tot_cov FROM emi.main) AS q
```

You will then find something like 1.36 square degrees at level 8 and 0.288 at level 12. By the way, in this particular case you can argue that

```
select sum(area(s_region)) from emi.main
```

(which works out to 0.0028 square degrees) gives a better idea of the total exposed area, because the individual frames do not (appreciatably) overlap and the rougher MOCs are seriously misrepresenting the true coverage. On the other hand, that expression *will* count overlaps twice, and hence will be massively overestimating coverage in more typical archives.

Solution for Exercise 22 I found an access URL by typing **ROSAT survey pointed** into a freetext constraint in WIRR and adding a *Service Type* constraint of *Image Access*. This, at the moment, only leaves one service, the SIA link of which I pasted into the program.
The resulting code is:

```
import pyvo

ACCESS_URL = "http://dc.zah.uni-heidelberg.de/rosat/q/im/siap.xml?"

svc = pyvo.sia.SIAService(ACCESS_URL)
images = svc.search((340.1,3.36), size=(0.1, 0.1))
image=images[0]
print(image.filesize, image.instr)
```

If you got an exception like

```
IndexError: index 0 is out of bounds for axis 0 with size 0
```

– this is an artefact of how we blindly fetch the first result. What this really means: there was no match in the service. Depending on what part of the ROSAT results is published, it is totally conceivable that they did not cover our location.

Solution for Exercise 23 To obtain the position of M51, I am using the SkyCoord snippet from the pyVO documentation; the rest mainly is cleanup and a standard random hack:

```
import random
import sys

from astropy import coordinates
from astropy.time import Time
from pyvo.dal import sia
from pyvo import registry

POS = coordinates.SkyCoord.from_name('M51')

def search_one_resource(res_rec):
    print("\nNow querying ", res_rec.res_title)
    svc = res_rec.get_service("sia", lax=True)
    images = svc.search(POS, size=0.5)
    for match in images:
        print(f"{match.title} Get? ", end=" ")
        if input().strip().lower().startswith("y"):
            match.cachedataset()

def main():
    for res_rec in registry.search(servicetype="image"):
        if random.random()<0.9:
            continue
        try:
            search_one_service(res_rec)
        except KeyboardInterrupt:
            if input("\nQuit? ").strip().lower().startswith("y"):
                sys.exit()
        except:
            import traceback
            traceback.print_exc()

if __name__ == "__main__":
    main()
```

If you are somewhat downtrodden by how much breakage you see and how weird some of the images that you find look like: Relax, it's science. There is actually a lot of art and knowledge between the raw images and the pretty pictures you see in the newspaper.

Solution for Exercise 24 The source code in question is in `pyvo/samp.py` (at the time of writing; it might be moved at some point).

The connection manager is right at the bottom of the file, and you see there that code like this should connect you to the SAMP hub:

```
client = SAMPIntegratedClient(name="test", description="VO course problem solution")
client.connect()
```

At least in the astropy versions current while this was written, when the client object just vanishes, it will not tell the hub the client is dead, and hence zombie clients will aggregate, for instance in TOPCAT's SAMP client line. Even if astropy were to get a bit smarter here, objects are in a precarious state when the automatic garbage collection strikes. Doing explicit connection management therefore is highly preferable in any case. In particular if it is as simple as just using a context manager.

Incidentally, on Debian-derived systems an attractive alternative to feeding github behavioural data would be to say `apt-get source python3-pyvo`.

Solution for Exercise 25 You will find that `send_image_to` calls `send_product_to`, just filling in the latter's `mtype` argument. More on the MTypes later; consider them a function name.

Now, `send_product_to` basically fills a dictionary like this:

```
message = {
        "samp.mtype": mtype,
        "samp.params": {
            "url": url,
            "name": name,
        },
    }
```

This is basically like a function call with keyword arguments. That really is almost all the magic; knowing this may come in useful if you want to send out more tailored SAMP messages later.

To send the image to Aladin only, you can guess that you will want to use the `client_name` argument (don't worry about the implementation in that case). In TOPCAT's SAMP status, you can find that Aladin's client name is a capitalised "Aladin", so in sum, you would say:

```
pyvo.samp.send_image_to(
   conn,
   match.acref,
   name=match.suggest_dataset_basename(),
   client_name="Aladin")
```

Solution for Exercise 26

```
import pyvo

svc = pyvo.dal.TAPService("http://dc.g-vo.org/tap")
result = svc.run_sync("SELECT count(*) as ct FROM arihip.main")
print(result[0]["ct"])
```

Solution for Exercise 27 The error message will have looked a bit like this:

> Field query: Could not parse your query: Expected end of text, found '4' (at char 64), (line:3, col:17)

This is admittedly not terribly helpful, but it is surprisingly hard to get parsers – programs that turn sequences of characters into some structured representations – to spit out helpful error messages. When you do not understand what some error means, first look at the position the machine gave up at. In this case, this means showing the query that was actually executed; a good `print` is perfectly fine in these cases, but my advice is to drop into the debugger, which you can do with a line like this:

```
import pdb; pdb.set_trace()
```

You are then dropped into something you can interact with (try **help** at the prompt), e.g., by evaluating python expressions:

```
-> print(QUERY.format(**locals()))
(Pdb) QUERY.format(**locals())
"select accref, imagetitle\nfrom maidanak.reduced\nwhere object=IC 4756"
(Pdb) cont
```

If you are fluent in ADQL, you will notice that at the error position reported by the server, there is a naked number. And that is because of our extremely simple-minded templating: A "good" templating engine should have turned the python string into an ADQL string literal. But as I said, if you control both sides, it is fine to just adjust the template to:

```
QUERY = """select accref, imagetitle
from maidanak.reduced
where object='{object}'"""
```

Solution for Exercise 28 An adapted version of fetch3 would look like this:

```
import pyvo

QUERIES = [
  ("tgas", "http://dc.zah.uni-heidelberg.de/tap",
    """SELECT ra, dec, pmra, pmdec
        FROM tgas.main
        WHERE phot_g_mean_mag BETWEEN 8 AND 8.2"""),
  ("rave", "http://dc.zah.uni-heidelberg.de/tap",
    """SELECT raj2000, dej2000, rv, hmag
        FROM rave.main
        WHERE hmag BETWEEN 8 AND 8.2"""),]

def main():
  with pyvo.samp.connection() as conn:
    for short_name, access_url, query in QUERIES:
      service = pyvo.dal.TAPService(access_url)
      result = service.run_sync(query.format(**locals()), maxrec=90000)
      pyvo.samp.send_table_to(
        conn,
        result.to_table(),
        client_name="Aladin",
        name=short_name)


if __name__=="__main__":
  main()
```

As hinted, the secret of attractive plots in Aladin are filters. For TGAS, you can use the predefined "Draw proper motions of stars" filter and perhaps improve it a bit in the "Advanced Mode", e.g., by multiplying the two columns with 10.

For the radial velocity, perhaps a rainbow (blue and redshift) is appropriate? With a bit of experimentation I have come up with

```
{ draw rainbow(${rv}, -100, 100) fillcircle(300)}
```

Solution for Exercise 29 The two workarounds are:

- `work_around_vizast_bug` – this used to be necessary because VizieR used to put `arraysize="1"` into column declarations of their scalars, which made astropy make arrays from them. This has long been fixed.
- `work_around_sdss_ucd_bug` – this used to be necessary because the UCDs on the SDSS table were wrong and non-specific at the same time. If you drop the workaround, perhaps by writing `return ucd` at the top of the function, you will see that the optical part of our SEDs will be gone; the UCDs are still too unspecific for our purpose. Once you have understood what happens in the workaround, you can also use TOPCAT's table browser to ascertain that the UCDs are still only *phot.mag;em.opt*.

Solution for Exercise 31 Here is how to write this:

```
import pyvo

QUERIES = {
  "tgas": ("http://dc.zah.uni-heidelberg.de/tap",
    """SELECT *
```

```
        FROM
          tgas.main AS tg
          JOIN TAP_UPLOAD.rave AS mine
          ON DISTANCE(tg.ra, tg.dec, mine.raj2000, mine.dej2000)<1/3600.
    """),
  "rave": ("http://dc.zah.uni-heidelberg.de/tap",
    """SELECT raj2000, dej2000, rv, hmag
        FROM rave.main
        WHERE hmag BETWEEN 8 AND 8.1"""),}


def main():
  svc_url, query = QUERIES["rave"]
  rave_svc = pyvo.dal.TAPService(svc_url)
  job = rave_svc.submit_job(query, maxrec=90000)
  try:
    job.run()
    job.wait()
    job.raise_if_error()

    svc_url, query = QUERIES["tgas"]
    tgas_svc = pyvo.dal.TAPService(svc_url)
    result = tgas_svc.run_sync(query,
      uploads={
        "rave":  job.result_uri})

  finally:
    job.delete()

  with pyvo.samp.connection() as conn:
    pyvo.samp.send_table_to(
      conn, result.to_table(), client_name="topcat", name="rave+tgas")


if __name__=="__main__":
  main()
```

Note how this is much more logical than the first version with the individual photometric cuts, since there is just one constraint on the magnitudes now (the one on the H-band in rave) – when you send the resulting table to Aladin, you will see more matches in TGAS than you had when you were comparing the two catalogue cuts manually.

And no, I would not normally have kept queries and access URLs in a dictionary in a situation like this; the two queries are different roles, and representing them next to each other is misleading rather than helpful. I wrote it like this in order to keep the program structure as parallel to the original rave-tgas solution as possible.

If you inlined the queries, you actually showed better taste.

By the way, you could take this even further and make the tgas query async as well. You could then send a raw *table.load.votable* message to TOPCAT with the result URL as the table URL. That way, pyVO would not touch the data at all. That is a small win here, but it might be a useful technique in more demanding circumstances.

To do that, your send query would look like this:

```
svc = pyvo.dal.TAPService("http://dc.zah.uni-heidelberg.de/tap")
job2 = svc.submit_job("""
    SELECT *
    FROM tgas.main AS db
    JOIN TAP_UPLOAD.t1 AS tc
      ON DISTANCE(db.ra, db.dec, tc.raj2000, tc.dej2000) < 5./3600.
    """,
    uploads={"t1": job.result_uri})
```

```
job2.run()
job2.wait()
message = {
    "samp.mtype": "table.load.votable",
    "samp.params": {
        "url": job2.result_uri,
        "name": "tgas+rave-from-server",
    },
}
client_id = samp.find_client_id(conn, "topcat")
conn.call_and_wait(client_id, message, "10")
```

The main complication over the code above is that we `send_table_to` cannot deal with remote URIs and we have to essentially copy its implementation. But this is still relatively compact, I would say.

Solution for Exercise 32 Just add a `AND em_res_power>10000` to the query in the program.

And it is totally conceivable that you will not find anything for the objects you chose. Spectra of this sort are expensive to get and have only been obtained for relatively few stars

Solution for Exercise 33 You would probably replace the service creation in the constructor's class with

```
self.sia_service = pyvo.dal.SIAService(
  "http://dc.g-vo.org/lswscans/res/positions/siap/siap.xml")
```

With that, `make_response_table` would be as simple as:

```
ra = self.cur_table[self.ra_name][table_index]
dec = self.cur_table[self.dec_name][table_index]
return self.sia_service.search(pos=(ra, dec), size=0.05).to_table()
```

Solution for Exercise 34

```
import pyvo
from astropy.coordinates import SkyCoord
from astropy import units as u

import vohelper

class Odometer:
  def __init__(self, conn):
    self.conn = conn
    self.total_travelled = 0*u.deg
    self.last_position = None
    conn.bind_receive_message("coord.pointAt.sky", self.record_movement)

  @vohelper.show_exception
  def record_movement(self, privkey, sender_id, msg_id, mtype, params, extra):
    new_coord = SkyCoord(
      float(params["ra"])*u.deg,
      float(params["dec"])*u.deg)

    if self.last_position is not None:
      self.total_travelled += new_coord.separation(self.last_position)
    self.last_position = new_coord
    print(self.total_travelled)


def main():
  with pyvo.samp.connection(addr="localhost") as conn:
    odometer = Odometer(conn)
    input()
  print("Total travelled", odometer.total_travelled)


if __name__=="__main__":
  main()
```

Solution for Exercise 35 To answer this, you could of course read the documentation and figure out what's the name of the respective properties of the TAPService object. In fact, you should set aside an hour or two to at least browse the documentation of pyVO if you use it regularly (as you should with any other library that you regularly use). However, in this case discovery with command line completion in (i)python is legal; for instance,

```
In [3]: svc = pyvo.dal.tap.TAPService("http://dc.g-vo.org/tap")

In [4]: svc.<tab>
  svc.availability   svc.create_query   svc.run_async      svc.tables
  svc.available      svc.describe       svc.run_sync       svc.up_since
  svc.baseurl        svc.hardlimit      svc.search         svc.upload_methods
  svc.capabilities   svc.maxrec         svc.submit_job
```

It is a reasonable guess that `maxrec` and `hardlimit` are what you are looking for in this case.
A program doing what is asked for in the exercise would look somewhat like this:

```
from pyvo import registry

for res_rec in registry.search(keywords="tgas", servicetype="tap"):
  svc = res_rec.get_service("tap")
  print(svc.baseurl, svc.maxrec, svc.hardlimit)
```

Solution for Exercise 36

```
>>> from pyvo import registry
>>> rscs = registry.search(keywords="exoplanet merged catalogue")
>>> rscs.get_summary()
(we got it, short name ExoMerCat)
>>> svc = rscs["ExoMerCat"].get_service("tap")
>>> list(svc.tables.keys())
['exomercat.exomercat', ...
>>> svc.tables["exomercat.exomercat"].columns
[... <BaseParam name="ra_off"/>, <BaseParam name="dec_off"/>, <BaseParam
name="mass"/>...]
>>> res = svc.run_sync("select top 1 ra_off, dec_off from exomercat.exomercat order by mass asc")
>>> res[0]
(277.6981916666667, -10.991083333333332)
>>> cres = registry.search("constellation polygons")
>>> cres.get_summary()
<Table length=1>
```

```
index short_name ...        interfaces
int32    str9    ...          str24
----- ---------- ... ------------------------
    0  cstl cone ... conesearch, tap#aux, web
>>> csvc = cres[0].get_service("tap")
>>> csvc.tables["cstl.geo"].columns
[..., <BaseParam name="name"/>, <BaseParam name="p"/>, <BaseParam name="ra"/>, ...]
>>> csvc.run_sync("select name from cstl.geo as db join tap_upload.pt as mine"
...     " on 1=contains(point(mine.ra_off, mine.dec_off), p)",
...     uploads={"pt": res.to_table()}).to_table()
<Table length=1>
 name
object
------
Scutum
```

So: with sufficient instrumentation and a clear horizon, you could see its host star from here (as in: Heidelberg). Scutum is just a bit south of the celestial equator, visible between Atair and Antares in summer nights.

Note that, of course, this is exactly not the nice blind (i.e., without prior knowledge of concrete resources) discovery we would like to have in the VO. But bear with us: It's much easier to write problems assuming prior knowledge.

Solution for Exercise 37 For server-side-expansion, change the condition to use the `gavo_vocmatch` UDF mentioned in the side track, like this:

```
    def __init__(self, uat_id, expand=False):
        if expand:
            self._condition = "1=gavo_vocmatch('uat', {uat_id}, uat_concept)"
        else:
            self._condition = "{uat_id} = uat_concept"
        self._fillers = {"uat_id": uat_id}
```

If you try it, you will notice that you get back massively more services.

When doing things locally, there is a complication because the naive templating engine cannot cope with sets. If it could, you would be done with just pulling the vocabulary (in a class attribute so we don't parse the vocabulary each time we are called) and then using a different operator:

```
class ForSource(pyvo.registry.SubqueriedConstraint):
    _keyword = "subject"
    _subquery_table = "rr.subject_uat"

    uat_voc = pyvo.utils.vocabularies.get_vocabulary("uat")

    def __init__(self, uat_id, expand=False):
        if expand:
            uat_ids = {uat_id} | set(self.uat_voc["terms"][uat_id]["narrower"])
        else:
            uat_ids = {uat_id}

        self._condition = "uat_concept in ({uat_ids})".
        self._fillers = {"uat_ids": uat_ids}
```

As things are, this will lead to an error, because the templating engine has no idea what to do with your set. Hence, you will have to manually do your formatting. But note that this sort of hack will make you vulnerable to SQL injection, so *never* create SQL like this when processing untrusted content:

```
        uat_ids = {uat_id} | set(self.uat_voc["terms"][uat_id]["narrower"])
        self._condition = "uat_concept in ({})".format(
            ", ".join(f"'{id}'" for id in uat_ids))
```

Solution for Exercise 38

```
def get_available_semantics(dl):
  res = set()
  for link in dl:
    res.add(link["semantics"])
  return res
```

Solution for Exercise 39 The first hurdle to the solution regrettably is: how will that service spell the identifier "IC 1151"? Using TOPCAT's or pyVO's table browsers, you will find the `target_name` column, and using something plausible like

```
select target_name from califadr3.cubes where target_name like '%1151%'
```

you will find they have skipped the blank (oh, for interoperable object designations!), and hence you will have to match IC1151.

For the matter with the setup, you can guess that it's what the setup column says and you would end up constraining `setup` to COMB (which in this case says that you are using a clever combination of the results of a higher and a lower resultion setup).

With these preparations, you can do the SODA calls; because, at the time for writing, pyVO does not pick up the processing service sitting on the dataset (rather than result set) level, we need to do the extra complication handled in the `get_cutout_frame` function given in the problem statement.

Taking everything together:

```
import io
import pyvo
import numpy
from astropy import units as u
from astropy.io import fits
from PIL import Image

def _normalize_for_image(pixels):
  pixels = numpy.flipud(pixels)
  pixMax, pixMin = numpy.max(pixels), numpy.min(pixels)
  pixels = (pixels-pixMin)/(pixMax-pixMin)*255
  return numpy.asarray(pixels, numpy.uint8)


def get_cutout_frame(datalink, wavelength):
  proc = datalink.get_first_proc()
  fits_stream = proc.processed(band=(wavelength, wavelength))
  return fits.open(io.BytesIO(fits_stream.read()))[0].data[0]


svc = pyvo.dal.TAPService("http://dc.g-vo.org/tap")
res = svc.run_sync(
  """SELECT * FROM califadr3.cubes
  WHERE target_name='IC1151' AND setup='COMB'""")
datalink = next(res.iter_datalinks())
```

```
pixels = numpy.array([
  _normalize_for_image(get_cutout_frame(datalink, 700*u.nm)),
  _normalize_for_image(get_cutout_frame(datalink, 550*u.nm)),
  _normalize_for_image(get_cutout_frame(datalink, 400*u.nm))])
pixels = pixels.transpose(1,2,0)

Image.fromarray(pixels, mode="RGB"
  ).save("IC1151.jpeg", format="jpeg")
```

if you cannot get enough: It is not hard to extend this so for each band, a few spectral frames are averaged rather than just exactly one frame (which we pick out here with our relatively stupid `.data[0]`.

Solution for Exercise 44

```
import pyvo

svc = pyvo.ssa.SSAService("http://dc.g-vo.org/theossa/q/ssa/ssap.xml?")
with pyvo.samp.connection() as conn:
  for ct, result in enumerate(
      svc.search(pos=None, diameter=None, t_eff="70000/100000",
        log_g="4.5/5.5", w_N="0.015/+Inf",
        FORMAT="VOTable")):

    pyvo.samp.send_spectrum_to(conn, result.getdataurl(), client_name="splat")
    if ct==2:
      break
```

Solution for Exercise 45

```
import pyvo
from astropy import table

gavo_svc = pyvo.dal.TAPService("http://dc.g-vo.org/tap")
hd_subset = gavo_svc.run_sync("SELECT TOP 500 mv_meas, spectral,"
    " source_id3 as source_id FROM hdgaia.main").to_table()

esac_svc = pyvo.dal.TAPService("https://gea.esac.esa.int/tap-server/tap")
gaia_meta = esac_svc.run_sync(
    """SELECT *
        FROM gaiadr3.gaia_source as g
        JOIN tap_upload.mine as m
        USING (source_id)""",
    uploads={"mine": table.Table(
        [hd_subset["source_id"]])})
full_result = table.join(
    hd_subset,
    gaia_meta.to_table(),
    keys="source_id")
with pyvo.samp.connection() as conn:
    pyvo.samp.send_table_to(conn, full_result, client_name="topcat")
```

Solution for Exercise 46 This one you can easily look up in the UCD list cited above; a quick text search will give your *phys.angsize*. Or use the WIRR trick and type "angular size" in Blind Discovery → Column UCD, and then *Pick one*. And of course this would work equally well with the RegTAP ADQL.

Solution for Exercise 47 This one is tricky on two accounts; first, you will probably have to look for "color" to find anything going in the right direction. That's the astronomy part of this problem.

Once you look for UCDs with descriptions containing "color", you will find UCDs of the form *phot.color;em.opt.b;em.opt.v*. You *could* read up on that particular UCD form in the standards, but you are entirely forgiven for just guessing that it is fine replacing the two band identifiers there with those for the u and r bands.

You should probably look up the definitions in the UCD list. But failing that, you can also see what other people have used to annotate SDSS u or r using WIRR or RegTAP. Looking for **SDSS u** yields *phot.mag;em.opt.u* immediately, and similarly **SDSS r** yields *phot.mag;em.opt.r*. Note that the band identifiers from UCDs are indeed not precise identifiers for concrete filters but qualitative indicators where in the spectrum one is.

So: *phot.color;em.opt.u;em.opt.r* looks like a good guess. And it's actually correct.

Solution for Exercise 48 Using pyVO, finding the set of concept identifiers narrower than or equal to *spatially-resolved-dataset* looks like this:

```
import pyvo

voc = pyvo.utils.vocabularies.get_vocabulary("product-type")
print(set(voc["terms"]["spatially-resolved-dataset"]["narrower"])
    | {'spatially-resolved-dataset'})
```

Over writing the resulting ADQL expression

```
dataproduct IN ('cube', 'time-cube', 'polarization-cube',
  'spatially-resolved-dataset', 'image', 'slit-spectrum',
  'spectral-cube', 'spatial-profile')
```

(which is what the program above yielded in mid-2024), the advantage of using the UDF is that your query remains up to date even when more spatially resolved product types are added to the product-type vocabulary.

References

Campillo, J. J. and Demleitner, M. (2023), 'Catalogue of ADQL User Defined Functions Version 1.1', IVOA Endorsed Note 17 November 2023. https://ui.adsabs.harvard.edu/abs/2023ivoa.spec.1117C.

Cecconi, B., Louys, M., Preite Martinez, A., Derrière, S., Ochsenbein, F., Erard, S. and Demleitner, M. (2023), 'UCD1+ controlled vocabulary - Updated List of Terms Version 1.5 Version 1.5', IVOA Endorsed Note 25 January 2023. https://ui.adsabs.harvard.edu/abs/2023ivoa.spec.0125C.

Demleitner, M., Dowler, P., Plante, R., Rixon, G. and Taylor, M. (2012), 'TAPRegExt: a VOResource Schema Extension for Describing TAP Services Version 1.0', IVOA Recommendation 27 August 2012, arXiv:1402.4742. doi:10.5479/ADS/bib/2012ivoa.spec.0827D, https://ui.adsabs.harvard.edu/abs/2012ivoa.spec.0827D.

Demleitner, M., Gray, N. and Taylor, M. (2023), 'Vocabularies in the VO Version 2.1', IVOA Recommendation 06 February 2023. https://ui.adsabs.harvard.edu/abs/2023ivoa.spec.0206D.

Demleitner, M., Harrison, P., Molinaro, M., Greene, G., Dower, T. and Perdikeas, M. (2019), 'IVOA Registry Relational Schema Version 1.1', IVOA Recommendation 11 October 2019. doi:10.5479/ADS/bib/2019ivoa.spec.1011D, https://ui.adsabs.harvard.edu/abs/2019ivoa.spec.1011D.

Erard, S., Cecconi, B., Le Sidaner, P., Demleitner, M. and Taylor, M. (2022), 'EPN-TAP: Publishing Solar System Data to the Virtual Observatory Version 2.0', IVOA Recommendation 22 August 2022. https://ui.adsabs.harvard.edu/abs/2022ivoa.spec.0822E.

Gray, N., Cecconi, B., Demleitner, M., Derrière, S., Gray, N., Louys, M. and Ochsenbein, F. (2023), 'Units in the VO Version 1.1', IVOA Recommendation 15 December 2023. https://ui.adsabs.harvard.edu/abs/2023ivoa.spec.1215G.

Harrison, P., Demleitner, M., Major, B. and Dowler, P. (2018), 'XML Schema Versioning Policies Version 1.0', IVOA Endorsed Note 29 May 2018. doi:10.5479/ADS/bib/2018ivoa.spec.0529H, https://ui.adsabs.harvard.edu/abs/2018ivoa.spec.0529H.

Landais, G., Muench, A., Demleitner, M. and Savalle, R. (2024), 'Data origin in the VO version 1.1', IVOA Note 26 January 2024. http://ivoa.net/documents/DataOrigin/.

Mantelet, G., Morris, D., Demleitner, M., Dowler, P., Lusted, J., Nieto-Santisteban, M. A., Ohishi, M., O'Mullane, W., Ortiz, I., Osuna, P., Shirasaki, Y. and Szalay, A. (2023), 'Astronomical Data Query Language Version 2.1', IVOA Recommendation 15 December 2023. https://ui.adsabs.harvard.edu/abs/2023ivoa.spec.1215M.

This document's DOI is 10.21938/avVAxDlGOiu0Byv7NOZCsQ.