Using the Virtual Observatory

Markus Demleitner Hendrik Heinl Joachim Wambsganß

July 25, 2024

German Astrophysical Virtual Observatory

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

plus

$\sim$ 50 data centers worldwide adhering to these standards

plus

a few volunteers operating some $\pm$ central infrastructure

plus

authors of client software, libraries, and web pages making these resources available to astronomers and the public.

Numerically. . .

In numbers, the VO is:

1. ~ 50 data centers in ~ 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.

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

Whetting your appetite: Demo time

Assume you want to look for candidates for gravitationally lensed compact objects.

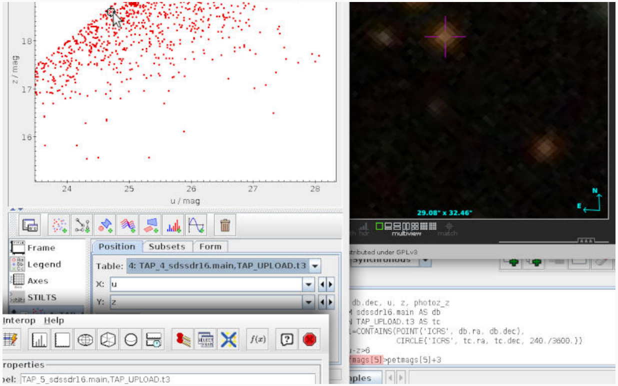


Table: 4: TAP_4_sdssdr16.main,TAP_UPLOAD.t3

X: u

Y: z

roperties

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
```

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()
```

Simple Protocols and their clients

Simple VO protocols

The basic/simple VO-protocols are

- the **S**imple **C**one **S**earch (SCS)
- the **S**imple **I**mage **A**ccess (SIA)
- the **S**imple **S**pectral **A**ccess (SSA)
- the **S**imple **A**pplication **M**essaging **P**rotocol (SAMP)

Neither simple nor a protocol, but crucial and omnipresent:

- the VO Registry

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, and more.

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, and more.

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, 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.

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*.

In the query pane, enter:

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

and then click “Ok”.

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*
- unary *project*
- unary *rename*
- binary *cartesian product*
- binary *union*
- binary *set difference*

Good News: You don't *need* to know any 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*]

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.

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
```

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

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

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
```

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.

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

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
```

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

SELECT: Grouping by HEALPix

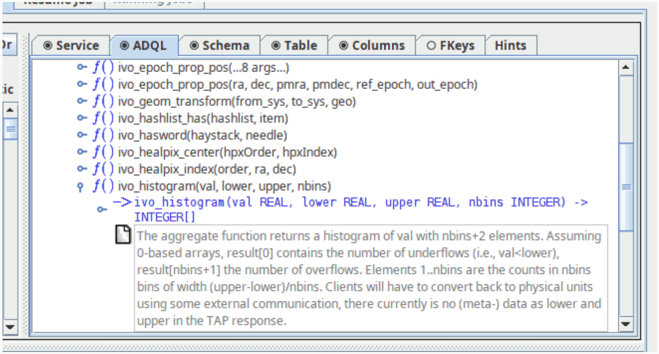
If you want to characterise some property over the sky, HEALPixes are your friend.

```
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
```

ADQL User Defined Functions

`ivo_healpix_index` is an example of an ADQL extension mechanism: Operators can add *UDF*s.

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



```
f() ivo_epoch_prop_pos(...8 args...)
f() ivo_epoch_prop_pos(ra, dec, pmra, pmdec, ref_epoch, out_epoch)
f() ivo_geom_transform(from_sys, to_sys, geo)
f() ivo_hashlist_has(hashlist, item)
f() ivo_hasword(haystack, needle)
f() ivo_healpix_center(hpxOrder, hpxIndex)
f() ivo_healpix_index(order, ra, dec)
f() 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.

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)
```

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.

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

(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
```

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).

- t1 `INNER JOIN` t2
- t1 `LEFT OUTER JOIN` t2
- t1 `RIGHT OUTER JOIN` t2
- t1 `FULL OUTER JOIN` t2

Geometries

The main extension of ADQL wrt SQL is addition of geometric functions.

```
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.))
```

There are more geometry functions defined in ADQL:

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

DISTANCE

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

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

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
```

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
```

Common table expressions

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

```
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)
```

TAP: Uploads

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

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
```

Almost real world

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.

```
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.
```

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.

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
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.

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.%'
```

To see what columns there are in *tap_schema.columns*, say:

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

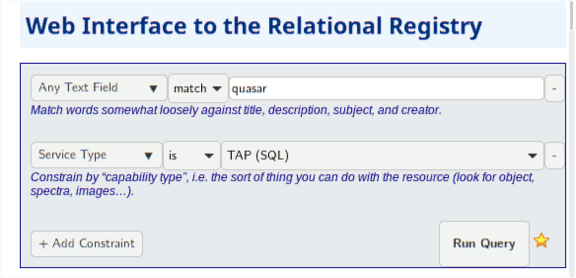
Of course, in normal operations, clients like TOPCAT do that querying for you: it's how they fill their metadata views.

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…).

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'
```

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
```

```
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'
```

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. . .

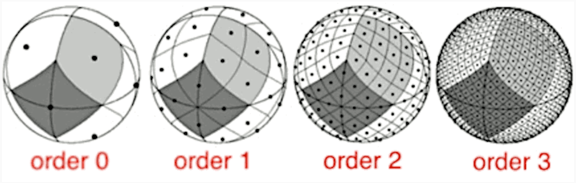
Interlude: HEALPix, MOC, HiPS

What are HEALPixes?

Spherical geometry is *hard*. . 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
- 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 $(\alpha, \delta) \to [0, \ldots, N]$.

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:

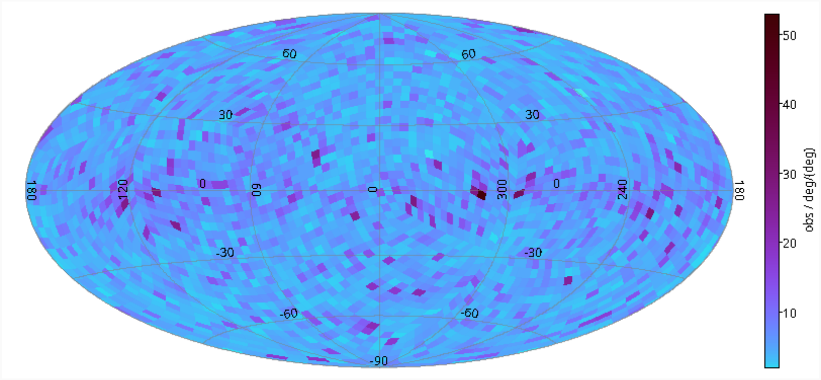
```
ivo_healpix_center(
  hpxOrder INTEGER, hpxIndex BIGINT) -> POINT
```

and

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

Application: HEALPix Maps

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

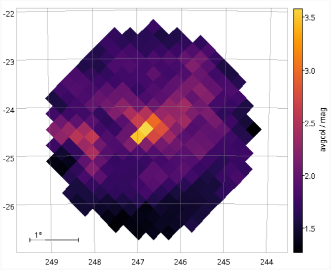


In Gaia

To get the HEALPix of a Gaia object at level n, compute

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

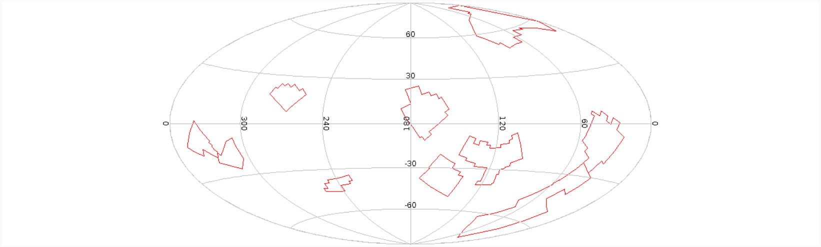
```
SELECT source_id/8796093022208 AS pix,
  AVG(phot_bp_mean_mag-phot_rp_mean_mag) AS avgcol
FROM gaia.edr3lite
WHERE DISTANCE(ra, dec, 246.7, -24.5)<2
GROUP BY pix
```



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:



MOCs to the rescue!

MOC?

You can represent arbitrary shapes to high precision (order 29 is 0.4 mas) as lists of HEALPix indexes.

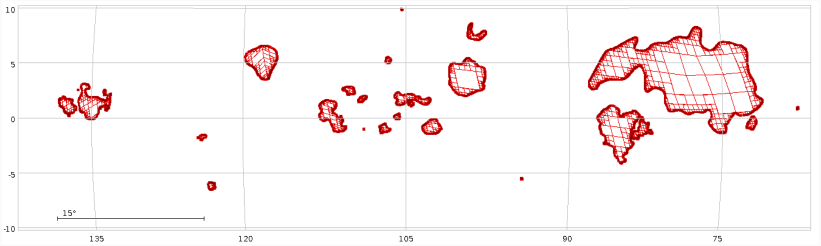
Alas, you need about 10 million such pixels for a shape of 1 deg^2.

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
```



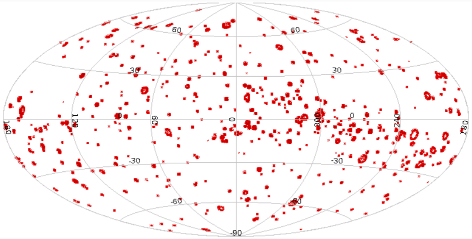
Such a shape may be written like

```
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:

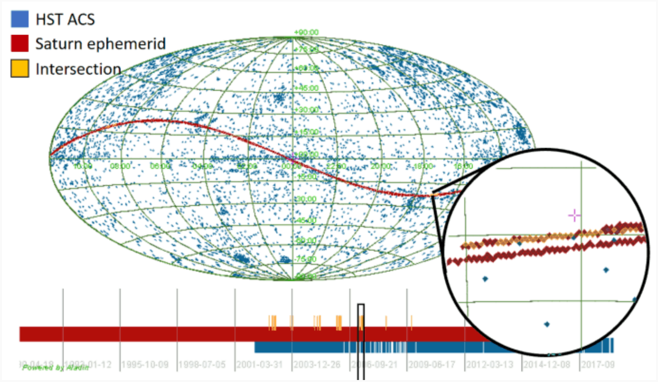


```
SELECT SUM(MOC(8, CIRCLE(ra, dec, 0.5*(4-vmag)))) AS contaminated
FROM hipparcos.main
WHERE vmag<4
```

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

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:



Mapping HEALPix to Anything: HiPS

HEALPixes are also behind HiPS, the Hierarchical Progressive Survey.

This is basically a set of maps

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

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.

pyVO Basics

Prerequisites

- python and astropy, of course
- TOPCAT for viewing and visualising tables
- Aladin to work with images
- 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.

Ψ tap-obscore.ipynb

To fit things on slides, I am PEP 8-relaxed.

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:

```
# <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...
```

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.

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

This is Python

The advantage of doing this in Python is that it is easy to add your own logic:

```
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

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
```

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:
```

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.

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.

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"])
```

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

Step 2: Go Async

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

```
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

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()
```

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

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

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:

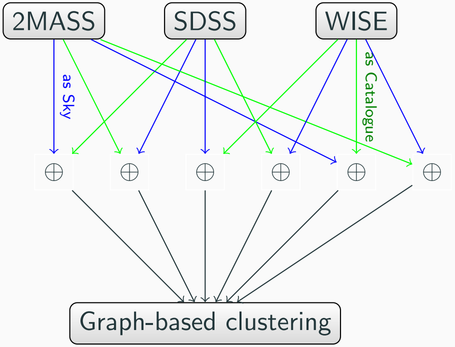
```
UCD_TO_WL = {
  "phot.mag;em.opt.u": 3.5e-7,
  "phot.mag;em.opt.b": 4.5e-7,
  "phot.mag;em.opt.v": 5.5e-7,
  "phot.mag;em.opt.r": 6.75e-7, ...}

  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))
```

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.



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

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?

Plan:

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

Obscore

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

- dataproduct_type – states *image*, *timeseries*, and the like.
- obs_publisher_did – a dataset identifier.
- 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.

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.

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})
```

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

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.

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()
```

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:

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):
    ...
```

Class name

Class variables

Constructor

Instance variables

Conventional self

Method definition

Ψ vicinitysearcher.py

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": {}})
```

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)
```

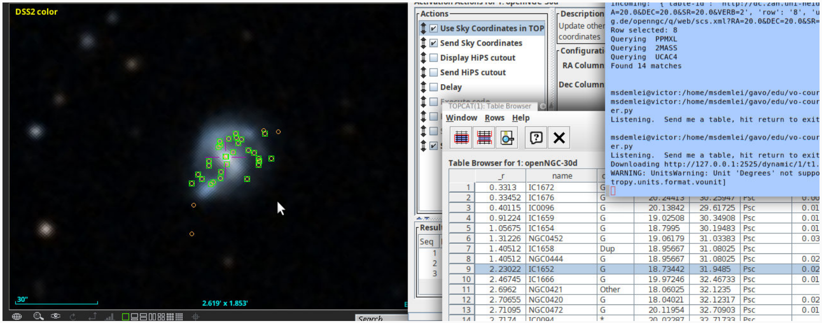
Try It Out

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

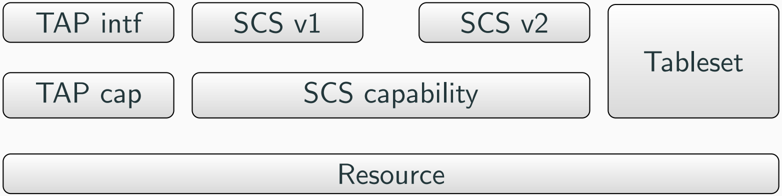
	_r	name					
1	0.3313	IC1672	G				
2	0.33452	IC1676	G	20.24413	30.25947	Psc	0.00
3	0.40115	IC0096	G	20.13842	29.61725	Psc	0.01
4	0.91224	IC1659	G	19.02508	30.34908	Psc	0.01
5	1.05675	IC1654	G	18.7995	30.19483	Psc	0.01
6	1.31226	NGC0452	G	19.06179	31.03383	Psc	0.03
7	1.40512	IC1658	Dup	18.95667	31.08025	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
14	2.7174	IC0094	*	20.02287	32.71733	Psc	

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:



Principles of RegistryResource

What you get back from `registry.search` is a sequence of `RegistryResource` instances.

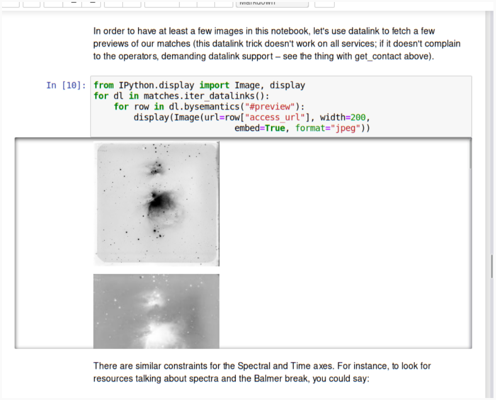
It has attributes for metadata (`res_title`, `res_description`. . .), and important methods:

- `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

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:

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.

To resolve an ivoid:

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

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

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.

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

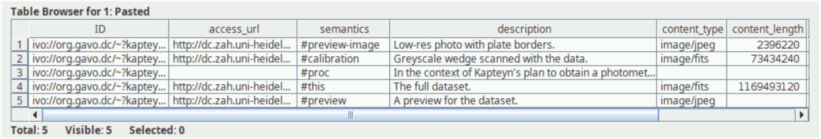
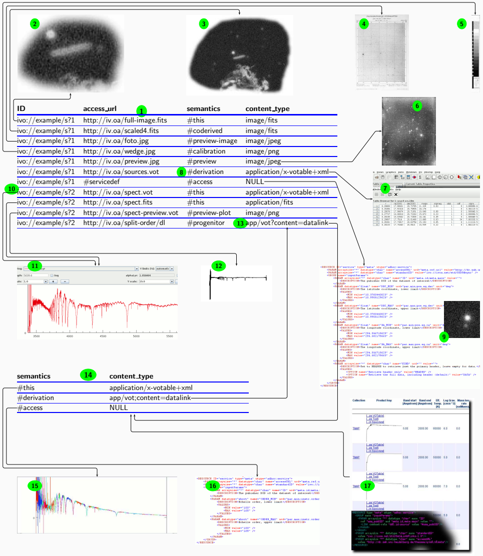


Table Browser for 1: Pasted

	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**

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

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.

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

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

Datalink: Remote Processing on Non-Datalink Documents

Use case: Hα maps of Sd galaxies from CALIFA.

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`?

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

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.

What *is* marginally possible: “Give me all measurements of a certain sort of UCD in a certain vicinity.”

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)
```

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()
```

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),]
```

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,...)
```

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

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.

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.

Does a service support union?

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

Running Queries II: Adapting to Server Capabilities

```
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())
```

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!

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))
```

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")
```

Odds and Ends

EPN-TAP 1

EPN-TAP is like obscore, just for solar system data. Columns of note include:

- `granule_uid` – an identifier for the dataset
- `target_name` – what was observed?
- `time_min`, `time_max` – when was it observed?
- `c<n>_min`, `c<n>_max` – where is it?
- `dataproduct_type` – the sort of observation.
- `instrument_host_name` – the probe or laboratory that produced the data.
- `instrument_name` – the instrument that produced the data.

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.

```
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

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.

pyVO does not yet have some API that would properly hide this (not terribly pretty) implementation detail.

```
python viewparams.py "http://dc.g-vo.org/bgds/q/sia/siap.xml?"
```

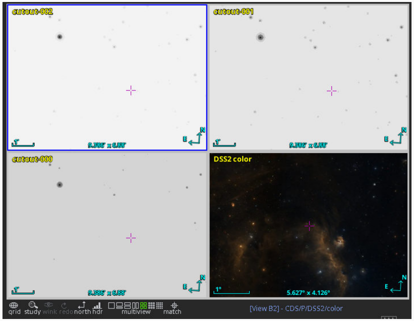
Ψ viewparams.py

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


cutout-002
cutout-001
cutout-000
DSS2 color
9.386' x 6.88'
5.627° x 4.126°
N
E
grid study wink redo north hdr multiview match
[View B2] - CDS/P/DSS2/color

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.

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
```

Troubleshooting and FAQ

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?

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.

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.

Why no Platform?

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

But more importantly: With multiple

interoperable

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.

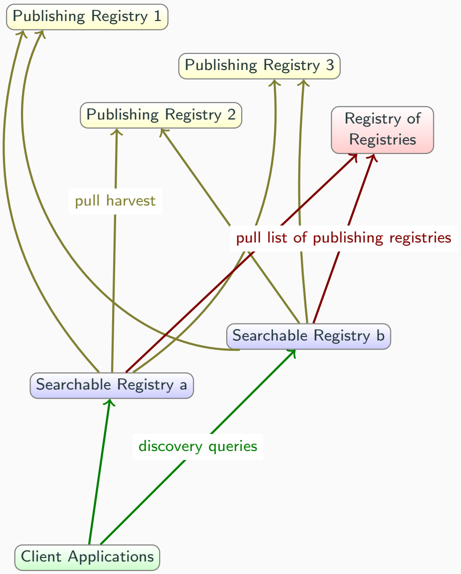
In a slogan

Protocols, not Platforms

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

Federation in practice: the VO Registry

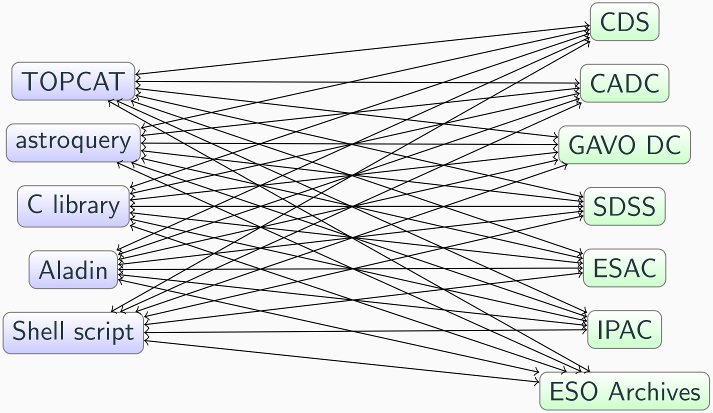
The VO Registry is what holds everything together: It's what your client asks when you look for, say, "a TAP service with proper motions for stars fainter than 23 mag".
It is a fairly complex system; but it's also an excellent example for what "federation" means.



Side Track: Standards

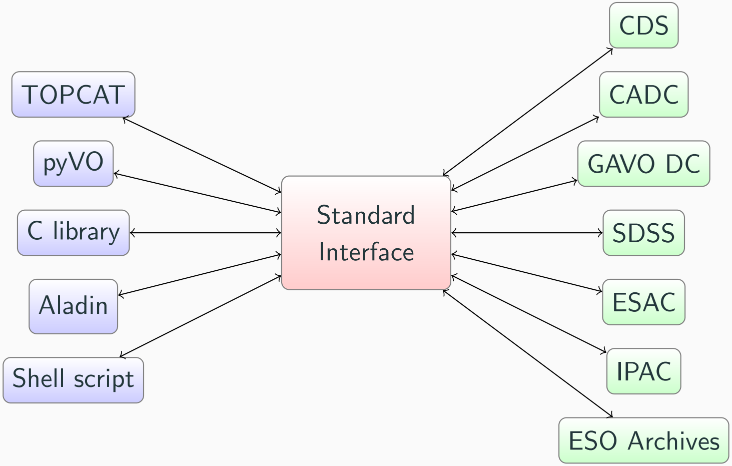
Data Access Without Standards

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:



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 $\sim$ 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.

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. . .).

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.

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.

UCDs Have a Grammar

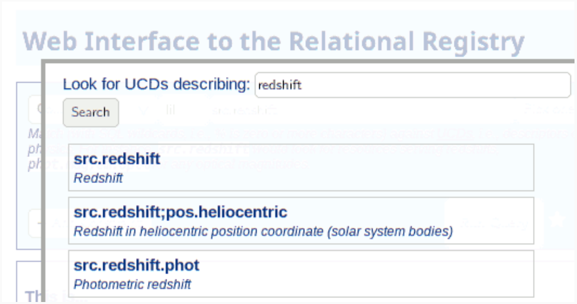
There is a large number of concepts represented in our tables. A single label hence is not enough.

The list of UCDs 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

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.

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” 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 formats, and
- desise, dead simple semantics
- Develop vocabularies in a community process using VEPs

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
- *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 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)
```

Side Track: VOTable

The VO’s Native Table Format: VOTable

```
<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>
```

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*.

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.

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.

In TOPCAT, see Views/Table Parameters:

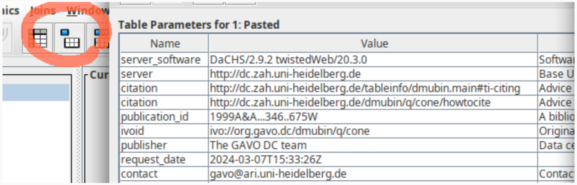


Table Parameters for 1: Pasted

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

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.

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:

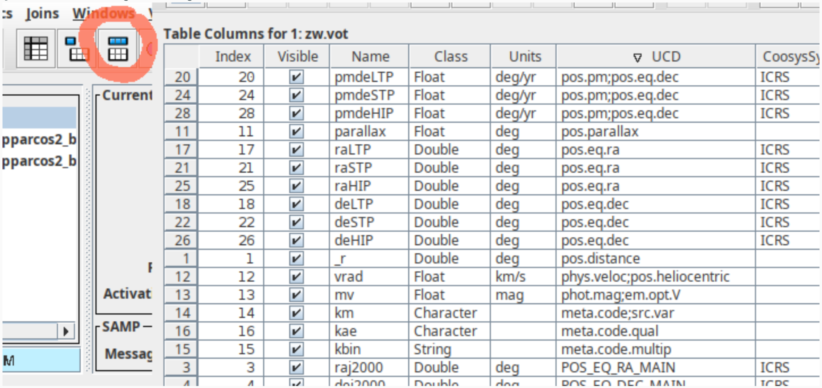


Table Columns for 1: zw.vot

	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"/>
```

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.

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>]
```

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.

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