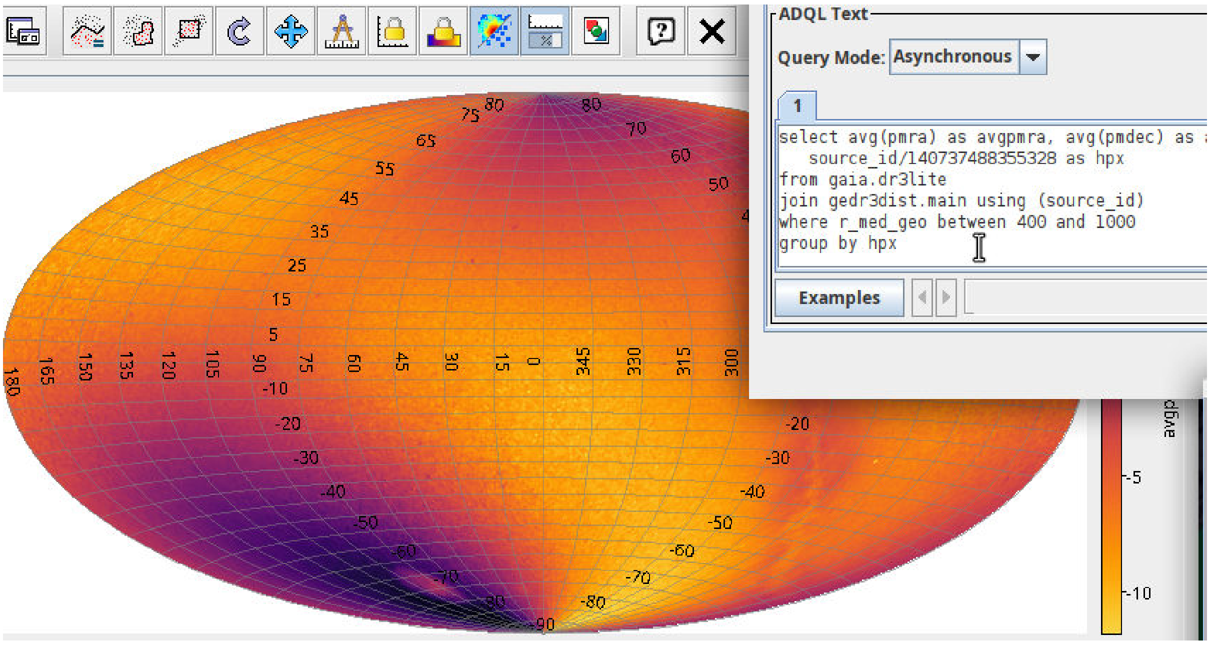
A Short Course On ADQL

Markus Demleitner Hendrik Heinl

July 25, 2024



ADQL Text

Query Mode: Asynchronous

```
select avg(pmra) as avgpmra, avg(pmdec) as
   source_id/140737488355328 as hpx
from gaia.dr3lite
join gedr3dist.main using (source_id)
where r_med_geo between 400 and 1000
group by hpx
```

Abstract

This is a course on the Virtual Observatory's main query language ADQL (short for Astronomical Data Query Language), which is a SQL dialect standardised so users do not have to learn new languages each time they want to use a new resource. We also introduce the basic aspects of the Table Access Protocol TAP, which transports ADQL queries, their results, as well as the metadata necessary to write meaningful queries.

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. Participants should have some basic knowledge of astronomy, and they should have some basic knowledge of the TOPCAT table processor.

TAP/ADQL

T(able) A(ccess) P(rotocol)

A(stronomical) D(ata) Q(uery) L(anguage)

Open a browser on http://docs.g-vo.org/adql for lecture notes.

Data Intensive Science

Data-intensive science means:

1. Using many data collections
2. Using large data collections

Point (1) requires standard formats and access protocols to the data, point (2) means moving the data to your box and operating on it with FORTRAN and grep becomes infeasible.

The Virtual Observatory (VO) in general is about solving problem (1), TAP/ADQL in particular about (2).

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 1**
>
> 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 2**
>
> 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 3**
>
> 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 4**
>
> 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 5**
>
> 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.

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.

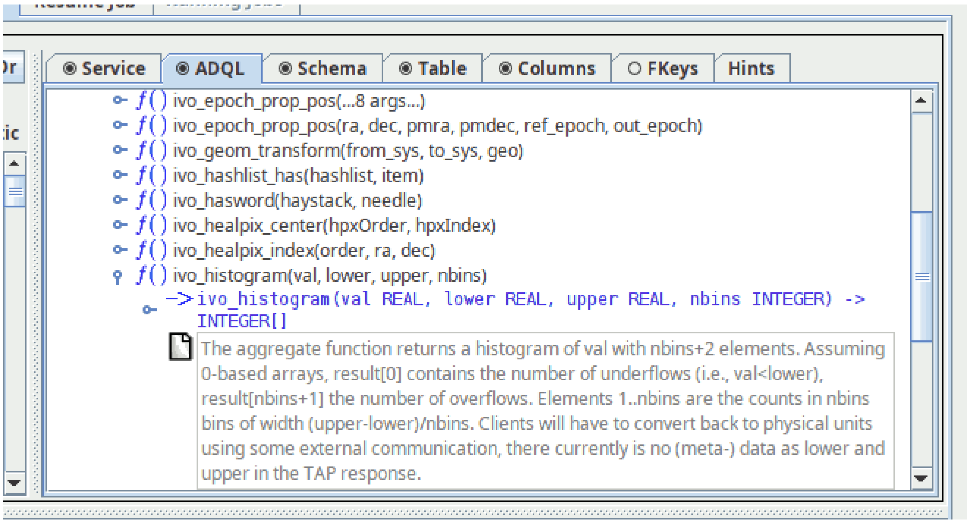
Exercise 6

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.

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 7

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 8

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 9 before making a pattern out of this).

> **Exercise 9**
>
> 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 10**
>
> 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 11

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 12

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 13

(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 14

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'
```

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. UCD stands for Unified Content Descriptor and defines simple semantics for physical quantities. For more information, see the UCD standard. To get an idea what UCDs look like, try:

```
SELECT DISTINCT ucd
FROM tap_schema.columns
ORDER BY ucd
```

Exercise 15

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?

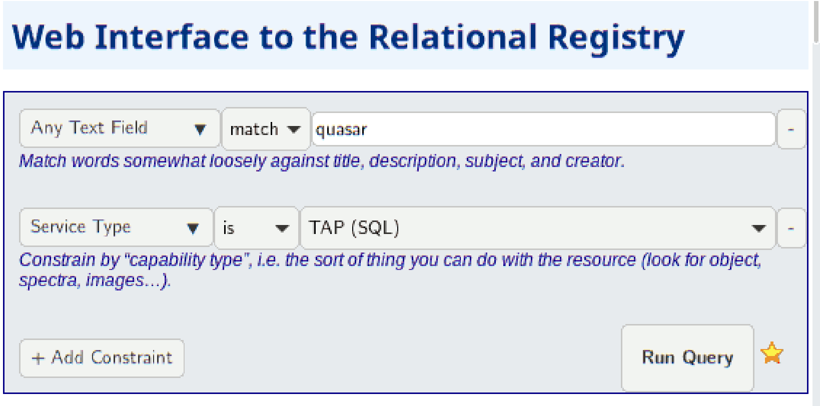
[1] https://blog.g-vo.org/a-proposed-vector-extension-for-adql.html

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…).

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 16

In exercise 14, 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…

Solution for Exercise 1

```
SELECT TOP 20 *
FROM fk6.part1
ORDER BY vmag ASC
```

Solution for Exercise 2

```
SELECT TOP 20
5+5*LOG10(pres*3600.)+vmag AS absmag, comname
FROM fk6.part1
ORDER BY vmag ASC
```

Solution for Exercise 3 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 4 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 5

```
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 6 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 7 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 8

```
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 9 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 11 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 12 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 13 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 14 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 15 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 16 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.

This document's DOI is 10.21938/uH0_xl5a6F7tKkXBSPnZxg.