1. ADQL and TAP

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

Agenda

- Why bother?
- A first query
- ADQL
- The finer points of TAP

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

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

Open a browser on `http://docs.g-vo.org/adql`

2. 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).

3. 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:

```
▷ 1   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.

Copying and Pasting from http://docs.g-vo.org/adql[1] is legal.

Note that in the top part of the dialog there's 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[2].

You can also use an ipython notebook, and with a bit of creativity, you can even follow this course in this kind of interface. Attached to this page or pdf is a notebook showing some of the more common features.

Most of the queries here assume you're querying against the server with the IVOA id

`ivo://org.gavo.dc/__system__/tap/run`.

To get that, typing `server ivo://org.g` and then completing with Tab should be sufficient.

You can also use TAPHandle[3], which runs entirely in your browser.

For running a TAP client in scripts there is STILTS[4] or PyVO[5]

More TAP clients can be found on the IVOA applications page[6].

See PDF attachment(s): tap-in-pyvo.ipynb

1 `http://docs.g-vo.org/adql`
2 `http://aladin.u-strasbg.fr/`
3 `http://saada.u-strasbg.fr/taphandle/`
4 `http://www.star.bris.ac.uk/~mbt/stilts/`
5 `http://pyvo.readthedocs.io/en/latest/index.html`
6 `http://www.ivoa.net/astronomers/applications.html`

4. 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

5. Relational Algebra

At the basis of relational data bases is the relational algebra, an algebra on sets of tuples ("relations") defining 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's a solid theory behind all of this.

6. 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.

▷ 2
```
SELECT TOP 5 * FROM rave.main
```

▷ 3
```
SELECT TOP 10 * FROM rave.main
```


7. 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):

▷ 4
```
SELECT TOP 5 *
FROM rave.dr2
ORDER BY rv
```

▷ 5
```
SELECT TOP 5 *
FROM rave.dr2
ORDER BY rv DESC
```

▷ 6
```
SELECT TOP 5 *
FROM rave.dr2
ORDER BY fiber_number, rv
```

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)

Problems

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

8. SELECT: what?

The select list has column names or expressions involving columns.

SQL expressions are not very different from those of other programming languages.

▷ 7
```
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“).

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's 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, 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 (that's called delimited identifiers). Don't do that if you can help it, since 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`:

▷ 8
```
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 `*`

▷ 9
```
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.

▷ 10
```
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.

Problems

(8.1) 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!). **(L)**

9. SELECT: WHERE clause

Behind the WHERE is a logical expression; these are similar to other languages as well, with operators `AND`, `OR`, and `NOT`.

▷ 11
```
SELECT name FROM rave.dr2
WHERE
  obsDate>'2005-02-02'
  AND imag<12
  AND ABS(rv)>100
```

Problems

(9.1) 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. **(L)**

10. 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.

▷ 12
```
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).

For simple `GROUP` applications, you can shortcut using `DISTINCT` (which basically computes the "domain").

▷ 13
```
SELECT DISTINCT comp, FK FROM dmubin.main
```

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[7]. Not all services support the necessary functions (in TOPCAT, you can check in the "service" tab).

While for large catalogues, such queries will have long runtimes, you can try it for smallish catalogues even in a course situation, for instance:

▷ 14
```
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, 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.

[7] http://www.star.bris.ac.uk/m̃bt/papers/adassXXVI-P1-31-poster.pdf

Problems

(10.1) 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. **(L)**

11. SELECT: JOIN USING

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

▷ 15
```
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.

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.

12. SELECT: JOIN ON

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

▷ 16
```
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 Königstuhl 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).

There are various kinds of joins, depending on what elements of the cartesian product are being retained. 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.

13. 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.

▷ 17
```
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.

▷ 18
```
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

Problems

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

14. 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:

▷ 19
```
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.

▷ 20
```
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.

15. 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:

▷ 21
```
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

▷ 22
```
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

▷ 23
```
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 to this problem before making a pattern out of this).

Problems

(15.1) Sit back for a minute and think whether the JOIN and the EXIST solution are actually equivalent. You're 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. **(L)**

16. Common table expressions

Quite a useful construct is `WITH`. This lets you name a subquery result for later use in your main query. Thus the queries are much easier to understand.

It may also let you override a catastrophic query plan:

▷ 24
```
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)
```

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.

17. 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, e.g., ex.vot from the HTML version of this page, load it into TOPCAT, go to the TAP window and there say:

▷ 25
```
SELECT mine.*, refcat.pmra, refcat.pmde FROM
  gaia.dr3lite AS refcat
  JOIN tap_upload.t1 AS mine
  ON DISTANCE (
    POINT(refcat.ra, refcat.dec),
    POINT(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.

Problems

(17.1) If you have some data of your own, try getting it into TOPCAT and try this with it (but that's really more of a TOPCAT problem).

See PDF attachment(s): ex.vot

18. 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 J2000. 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[8] 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:

▷ 26
```
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

[8] http://docs.g-vo.org/adql/html/matchme.vot

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:

▷ 27
```
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's 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.

Still... we *have* lost one object. Can you find it? And can you guess why we have lost it?

See PDF attachment(s): matchme.vot

19. TAP: Async operation

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

With TOPCAT, change the Mode selector to "Asynchronous" and run a query (any will do). In "Running Jobs", select the URL and paste it somewhere.

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's 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.

20. 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 columns are in there by querying `TAP_SCHEMA itself`:

▷ 28
```
SELECT * FROM tap_schema.tables
WHERE table_name LIKE 'tap_schema.%'
```

Of the tables you get there, you'll be most interested in `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:

▷ 29
```
SELECT * FROM tap_schema.columns
WHERE table_name='tap_schema.columns'
```

You'll see there's description, unit, and type. The indexed column says if 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 a simple semantic for physical quantities. For more information, see the UCD IVOA standard[9]. To get an idea what UCDs look like, try:

▷ 30
```
SELECT DISTINCT ucd FROM tap_schema.columns ORDER BY ucd
```

Problems

(20.1) How many tables are there on the server? How many columns? How many columns with UCDs starting with phot.mag?

21. Data Discovery 1: the Registry

The VO has a "Registry" that keeps an inventory of the services and data kept within the VO. TAP services communicate basically what's in `TAP_SCHEMA` to the registry.

There are a few ways to search the registry. In TOPCAT we already used the keyword search in the TAP service window. Another way to search the registry is WIRR[10]. With the Web Interface to the Relational Registry (WIRR) you can search the VO registry in a more elaborate way. WIRR is not limited to search TAP services only, but also services using other VO protocols like SIAP or SCS. For now our use case will be to find tables talking about quasars having a column containing redshifts:

In the query field on top select

"Text Fields" - "match" - "quasar"

then click "+" and in the new appearing row select

"Service Type" - "is" - "TAP(SQL)" ,

again click "+" and in the new row select

"Column UCD" - "like" - "redshift"

Note that WIRR offers help what the queries mean if you click on "Info" at the end of each row.

9 http://www.ivoa.net/Documents/latest/UCDlist.html
10 http://dc.zah.uni-heidelberg.de/wirr/q/ui/

Problems

(21.1) Find out the UCDs for redshifts and proper motion. **(L)**

22. Data Discovery 2: use ADQL

The relational registry[11] says how to query this data set using ADQL. All tables are in the rr schema and can be combined through NATURAL JOIN. The same use case in ADQL looks like:

▷ 31
```
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='ivo://ivoa.net/std/tap'
  AND 1=ivo_hasword(table_description, 'quasar')
  AND ucd='src.redshift'
```

As you can see, I'm using UCD to express physics. It's instructive to compare the query above with the following one:

▷ 32
```
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='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 (in late 2013, two columns containing not redshifts but references are returned) but recall is up (in late 2013, you find redshift columns from SDSS catalogues that weren't there with the UCD query).

That's fairly typical. The recommended remedy: 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.

There are a few sample queries in the standard document – with those to start with, it's unlikely you'll ever going to need to resort to graphical interfaces to the registry like WIRR[12].

11 http://www.ivoa.net/documents/RegTAP/
12 http://dc.g-vo.org/WIRR

23. Simbad

Simbad has a TAP interface at `http://simbad.u-strasbg.fr/simbad/sim-tap`.

Here's how I found that out:

▷ 33
```
SELECT ivoid, access_url
FROM rr.capability
  NATURAL JOIN rr.interface
  NATURAL JOIN rr.resource
WHERE standard_id='ivo://ivoa.net/std/tap'
  AND 1=ivo_hasword(res_title, 'simbad')
```

Change your TAP URL to there and inspect Simbad's table metadata. See what the main entries look like:

▷ 34
```
SELECT TOP 20 * FROM basic
```

The possibilities are endless.

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

▷ 35
```
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'
```

You take it from here.

For otypes, simbad has a fairly elaborate classification system[13] that you'll need to know to make useful queries against otype. Another secret they're 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.

24. 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

▷ 36
```
SELECT role_name, email, base_role
FROM rr.res_role
WHERE ivoid='ivo://org.gavo.dc/__system__/tap/run'
```

– 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...

[13] `http://simbad.u-strasbg.fr/simbad/sim-display?data=otype`