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. Click the pin icon in the upper right corner of the dialog to keep the query window open even while the query is executing..

In TAP URL, enter `http://dc.g-vo.org/tap` in TAP URL and click "Enter Query".

At the bottom of the form, check "Synchronous" and enter

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

in the text box, 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. For example, there's tapsh[2] that emulates a normal, command-line database shell; 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.

More TAP clients can be found on the IVOA applications page[4].

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

[1] `http://docs.g-vo.org/adql`
[2] `http://soft.g-vo.org/`
[3] `http://saada.u-strasbg.fr/taphandle/`
[4] `http://www.ivoa.net/astronomers/applications.html`

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

```
▷ 2   SELECT TOP 5 * FROM rave.dr3
```

```
▷ 3   SELECT TOP 10 * FROM rave.dr3
```

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 name, rv
      FROM rave.dr3
      ORDER BY rv
▷ 5   SELECT TOP 5 name, rv
      FROM rave.dr3
      ORDER BY rv DESC
▷ 6   SELECT TOP 5 name, fiber, rv
      FROM rave.dr3
      ORDER BY fiber, 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 common name and the visual magnitude 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, alfa_Fe) AS ppress,
        SQRT(SQUARE(e_pmde)+SQUARE(e_pmra)) AS errTot
      FROM rave.dr3
```

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 just 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)`
- Power shortcuts: `SQRT(x)` (shortcut for `POWER(x, 0.5)`), `SQUARE(x)` (shortcut for `POWER(x, 2)`)
- 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
  POWER(10, alfa_Fe) AS ppress,
  SQRT(SQUARE(e_pmde)+SQUARE(e_pmra)) AS errTot
FROM rave.dr3
ORDER BY ppress
```

To select all columns, use `*`

▷ 9
```
SELECT TOP 10 * FROM rave.dr3
```

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

▷ 10
```
SELECT count(*) AS numEntries FROM rave.dr3
```

`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)

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.dr3
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
```

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.

▷ 14
```
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, you can join ON:

▷ 15
```
SELECT TOP 20 hipno, name
FROM dmubin.main AS dmu
LEFT OUTER JOIN rave.dr3 AS rave
ON (dmu.mv BETWEEN rave.imag-0.05 AND rave.imag+0.05)
```

This particular query gives, for each hipno in dmubin, all names from rave belonging to stars having about the same I magnitude as the visual magnitude given in dmubin. This doesn't make any sense, but you may get the idea.

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.

Keep the crossmatch pattern somewhere handy (everything is in degrees):

▷ 16
```
SELECT TOP 5 rv, e_rv,
 p.raj2000, p.dej2000, p.pmRA, p.pmDE
FROM ppmxl.main AS p
JOIN rave.dr3 AS rave
ON 1=CONTAINS(
  POINT('ICRS', p.raj2000, p.dej2000),
  CIRCLE('ICRS', rave.raj2000, rave.dej2000, 1.5/3600.))
```

In theory, you could use reference systems other than ICRS (e.g., GALACTIC, FK4) and hope the server converts the positions, but I'd avoid constructions with multiple systems – even if the server implements the stuff correctly, it's most likely going to be slow.

Some sites have extra features in the REGION construct. On GAVO's site, you can, e.g., say REGION('simbad <obj>').

▷ 17
```
SELECT raj2000, dej2000, pmRA, pmDE
FROM ppmxl.main
WHERE 1=CONTAINS(
  REGION('simbad M1'),
  CIRCLE('ICRS', raj2000, dej2000, 0.05))
```

Problems

(13.1) Compare the radial velocities given by the rave.dr3 and arihip.main catalogs, together with the respective identifiers (hipno for arihip, name for rave). Use a positional crossmatch with, say, a couple of arcsecs. **(L)**

14. 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:

▷ 18
```
SELECT count(*) as n, round((u-z)*2) as bin
FROM (
  SELECT TOP 4000 * FROM sdssdr7.sources) AS q
GROUP BY bin ORDER BY bin
```

15. 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 catalog giving positions reasonably close to J2000; 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:

```
▷ 19  SELECT mine.*, ppmxl.pmra, ppmxl.pmde FROM
        ppmxl.main AS ppmxl
        JOIN tap_upload.t1 AS mine
        ON (1=CONTAINS(
          POINT('ICRS', ppmxl.raj2000, ppmxl.dej2000),
          CIRCLE('ICRS', 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.

If your positions are in galactic coordinates, things *should* work if you just write `GALACTIC` rather than `ICRS` in the `CIRCLE`.

Problems

(15.1) 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).

— Dateien zu diesem Abschnitt in der HTML-Version—

16. Almost real world

Just so you get an idea how SQL expressions can evolve to span several pages: Suppose you have a catalog giving alpha, delta, and an epoch of observation reasonable far away from J2000. To match it, you have to bring the reference catalog on our side to the epoch of your observation. For larger reference catalogs, that would be quite an expensive endeavour. Thus, it's usually better to just transform a bunch 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.

In the following, we use a rough approximation to applying proper motions. Unfortunately, ADQL does not contain builtins for applying proper motions, and the exact expressions are messy. The following query should run (for a little while) with an artificial input file[5].

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

```
▷ 20  SELECT * FROM (
        SELECT
          mine.*,
          raj2000+pmra/cos(radians(dej2000))*(epoch-2000)
            as palpha,
          dej2000+pmde*(epoch-2000) as pdelta,
          pmra,
          pmde
        FROM
          ppmxl.main AS ppmxl
          JOIN tap_upload.t1 AS mine
          ON (1=CONTAINS(
            POINT('ICRS', ppmxl.raj2000, ppmxl.dej2000),
            CIRCLE('ICRS', mine.alpha, mine.delta, 0.1)))) as q
      WHERE
        palpha BETWEEN alpha-0.5/3600 AND alpha+0.5/3600
        AND pdelta BETWEEN delta-0.5/3600 AND delta+0.5/3600
```

(don't forget to adapt the table name behind tap_upload!). Done really correctly, this would still be a bit longer, since the outer where actually is a crossmatch criterion, too. You could either write a contains clause as in the inner select or, if you insist on a box-type criterion as used in the query, you should at least divide the tolerance in alpha by $\cos\delta$.

If you've tried it, you'll have noticed that 100 rows were returned for 100 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:

```
▷ 21  select
        raj2000-epdiff*pmra/cos(radians(dej2000))+(rand()-0.5)/4000 as alpha,
        dej2000-epdiff*pmde+(rand()-0.5)/5000 as delta, 2000-epdiff as epoch
      from (
        select TOP 100 m.*, 75-RAND()*50 as epdiff
        from ppmxl.main as m
        where sqrt(SQUARE(pmra)+SQUARE(pmde)) BETWEEN 1.7/3600. and 2/3600.) as qi
```

— Dateien zu diesem Abschnitt in der HTML-Version—

17. 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`:

```
▷ 22  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:

```
▷ 23  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[6]. To get an idea what UCDs look like, try:

[6] `http://www.ivoa.net/Documents/latest/UCDlist.html`

▷ 24
```
select distinct ucd from tap_schema.columns order by ucd
```

Problems

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

18. TAP: Locating data

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.

The relational registry[7] says how to query this data set using ADQL. All tables are in the rr schema and can be combined through NATURAL JOIN.

Find tables talking about quasars having a column containing redshifts:

▷ 25
```
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'
```

The relational registry contains UCDs as well. It's instructive to compare the query above with the following one:

▷ 26
```
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. 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 catalogs that weren't there with the UCD query).

That's fairly typical. The recommended remedy: Improve metadata so people can find the data using the precise UCD vocabulary. This goes to show that high-quality metadata is of utmost importance for the VO – which is something I'd like to implant into your hearts when you put your data into the VO yourselves. But on the other hand: Even shoddily published data is better than 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[8].

[7] `http://www.ivoa.net/documents/RegTAP/`
[8] `http://dc.g-vo.org/WIRR`

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.

The tapsh does this automatically (just exit it). With TOPCAT, uncheck "Synchronous" 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. tapsh does, and it's probably the way to go if you have larger jobs to run.

20. Simbad

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

Here's how I found that out:

▷ 27
```
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:

▷ 28
```
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 Catalog 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

▷ 29
```
SELECT TOP 1000
   otype, tc.*
   FROM public.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!='star'
```

You take it from here.

21. 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, use – the relational registry. If you have the IVORN of the service, use

▷ 30
```
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.

Left to the reader as an exercise