1. Gaia Data Queries with ADQL

Markus Demleitner *(msdemlei@ari.uni-heidelberg.de)*
Hendrik Heinl *(heinl@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-gaia/html`

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.

In TAP URL: at the bottom of the window, enter `http://gaia.ari.uni-heidelberg.de/tap` and click "Use Service".

At the bottom of the form, at Mode: check "Synchronous" and enter

```
▷ 1   SELECT TOP 1 1+1 AS result FROM gaiadr1.tgas_source
```

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-gaia[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.

[1] `http://docs.g-vo.org/adql-gaia`

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 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 gaiadr1.tgas_source
▷ 3   SELECT TOP 10 * FROM gaiadr1.tgas_source
```

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 source_id, parallax
      FROM gaiadr1.tgas_source
      ORDER BY parallax
▷ 5   SELECT TOP 5 source_id, parallax
      FROM gaiadr1.tgas_source
      ORDER BY parallax DESC
▷ 6   SELECT TOP 5 source_id, phot_g_mean_mag , parallax
      FROM gaiadr1.tgas_source
      ORDER BY phot_g_mean_mag, parallax
```

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 source id and visual magnitude of the 20 brightest stars in the table tgas_source. **(L)**

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
        source_id,
        SQRT(POWER(pmdec_error,2)+POWER(pmra_error,2)) AS pm_errTot
      FROM gaiadr1.tgas_source
```

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
        source_id,
        SQRT(POWER(pmdec_error,2)+POWER(pmra_error,2)) AS pm_errTot
      FROM gaiadr1.tgas_source
      ORDER BY pm_errTot
```

To select all columns, use `*`

```
▷ 9   SELECT TOP 10 * FROM gaiadr1.tgas_source
```

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

```
▷ 10  SELECT count(*) AS numEntries
      FROM gaiadr1.tgas_source
```

`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 source id, position, proper motion in arcsec/yr and mag g for the 20 fastest stars in tgas_source. **(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 source_id, ra, dec
  FROM gaiadr1.tgas_source
  WHERE
  phot_g_mean_flux > 13
  AND parallax < 0.2
```

Problems

(9.1) Select the absolute magnitude, position and the source id for the 20 stars with the largest observed apparent magnitude in the g band from the table tgas_source (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!). This will fail. Try to fix it. **(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(phot_g_mean_mag) AS bin,
  AVG(parallax) AS parallax_mean
FROM gaiadr1.tgas_source
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
ROUND(phot_g_mean_mag), ROUND(parallax)
FROM gaiadr1.tgas_source
```

Problems

(10.1) Get the averages for the total proper motion from tgas_source in bins of one mag in mag g mean 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 h1.ra, h1.dec, h1.hip, t1.hip
FROM hipparcos AS h1
JOIN tycho2 AS t1
USING (hip)
```

Check the tables in the Table Metadata shown by TOPCAT: astroref is from hipparcos, hp_mag from hipparcos_newreduction; both tables have a hip column.

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

```
FROM hipparcos AS h1
JOIN tycho2 AS t1
USING (hip)
```

yields the cartesian product of the hipparcos and tycho2 tables but only retains the rows in which the hip columns in both tables agree.

Note that while the hip 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 source_id, h.hip
FROM gaiadr1.tgas_source AS tgas
LEFT OUTER JOIN hipparcos as h ON (tgas.phot_g_mean_mag BETWEEN
h.hpmag -0.05 AND h.hpmag+0.05)
```

This particular query gives, for each source_id in tgas_source, all ids from ucas4 belonging to stars having about the same aperture magnitude as the mean g magnitude given in tgas_source. 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
        source_id, tgas.ra, tgas.dec, tm.raj2000,
              tm.dej2000, hmag, e_hmag
      FROM gaiadr1.tgas_source as tgas
      JOIN twomass AS tm
      ON 1=CONTAINS (
           POINT('ICRS', tm.raj2000, tm.dej2000),
           CIRCLE('ICRS', tgas.ra, tgas.dec, 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.

When catalogs are on different epochs, you may need to account for proper motions to match faster stars. You should, however, *not* apply the proper motions in the primary selection. If you do that, the index cannot be used, and your query will waste a lot of CPU and disk bandwidth. Instead, decide about the maximum proper motion your objects might have (to get an idea, of the statistics, try selecting the fastest stars from ppmx.data – apart from the fact that the catalog got the fastest stars pretty wrong with two copies of some fast stars, there's only a handful stars moving faster than four arcsecs per year).

Then multiply this with your epoch difference and make that your initial crossmatch radius. Then filter out the spurious matches with an extra where clause taking into account the proper motions. For moderate epoch differences, don't worry about going into the tangential plane to apply proper motions and, for now, say something like

```
▷ 17  SELECT
         TOP 30
         *
         FROM ppmxl AS m
         JOIN gaiadr2.gaia_source AS g
         ON 1=CONTAINS(POINT('ICRS', m.raj2000, m.dej2000),
                       CIRCLE('ICRS', g.ra, g.dec, 30./3600.))
      WHERE 1=CONTAINS(POINT('ICRS',
                      m.raj2000+m.pmra*COS(RADIANS(m.dej2000))*15,
                      m.dej2000+m.pmde*15),
        CIRCLE('ICRS', g.ra, g.dec, 0.5/3600.))
```

The 15 is because Gaia DR1 is on J2015, whereas PPMXL is on J2000. Also, be careful with the units – in many catalogs, positions and proper motions are given in different units.

Also note how the outer PM-based filter is just a WHERE-clause. Since JOIN is a combination of operators of the relational algebra, the result of a join is a relation again and thus can be treated like any other table.

Problems

(13.1) Compare the total proper motions of the top 5000 stars in hipparcos and tycho2, together with the respective identifiers (hip for hipparcos, id for tycho2). 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((hmag-jmag)*2) as bin
      FROM (
        SELECT TOP 4000 * FROM twomass) 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: Take a subset of tgas_source with positions and proper motions and crossmatch it with sdss to get colors. First we make the subset with:

```
▷ 19  SELECT TOP 200
      source_id, ra, dec, pmra, pmdec
      FROM gaiadr1.tgas_source
      WHERE 1=CONTAINS(POINT('ICRS', raj2000, dej2000),
            CIRCLE('ICRS', 18.02, 9.281, 4.0 ))
```

16. TAP: Uploads 2

Then we change the TAP Service to `http://dc.zah.uni-heidelberg.de/tap` and perform the following query:

```
▷ 20  SELECT TOP 100
         tgas.*, sdss.u, sdss.i, sdss.r, sdss.g
         FROM sdssdr7.sources AS sdss
         JOIN TAP_UPLOAD.t1 AS tgas
         ON 1=CONTAINS(
               POINT('ICRS', sdss.ra, sdss.dec),
               CIRCLE('ICRS', tgas.ra, tgas.dec, 3./3600.))
```

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.