TAP query

A library to query TAP servers

Author: Markus Demleitner
Email: gavo@ari.uni-heidelberg.de
Date: 2024-04-29

Contents

Please note: gavo.votable.tapquery was a stopgap module until other libraries supported TAP properly. Today, pyvo.dal.tap is a perfectly fine TAP library for python (and has inherited much of this library's code), and hence this library isn't necessary any more. Consider it *deprecated* (in the sense of: As long as it's not broken, we'll distribute it. If it breaks, we probably will rather drop than fix it).

TAP is a relatively complex protocol to execute potentially long-running (ADQL) queries on remote servers. As a part of the GAVO VOTable library, we provide a shallow client that speaks TAP. It is the intention of this document to keep the TAP spec from your reading list. If it doesn't do that, complain, and we'll try to fix it.

Obtaining the library

See the GAVO VOTable Library documentation

For the impatient

All you need to query a server is its access URL. What we need here is the root of the hierarchy, i.e., without any `sync` or `async`.
For the typcial case of a sync query, here's the pattern to retrieve a sequence of dictionaries, one per result line, using sync querying:

```
from gavo import votable

res = votable.ADQLSyncJob(
  "http://dc.zah.uni-heidelberg.de/__system__/tap/run/tap",
  "SELECT * FROM TAP_SCHEMA.tables"
  ).run()

data, metadata = votable.load(res.openResult())
for row in metadata.iterDicts(data):
  print row
```

Here's a variation using async querying (meaning: your query can take a long time if necessary), and yielding a numpy record array:

```
from gavo import votable
from numpy import rec

accessURL = "http://dc.zah.uni-heidelberg.de/__system__/tap/run/tap"
query = "SELECT TOP 3 * FROM TAP_SCHEMA.tables"
job = votable.ADQLTAPJob(accessURL, query)
try:
  job.run()
  data, metadata = votable.load(job.openResult())
finally:
  job.delete()
ra = rec.fromrecords(data, dtype=votable.makeDtype(metadata))
```

Note, however, that with record arrays NULL values will cause the whole thing to fail. So, if you are after record arrays, our advice is to install astropy and write:

```
from cStringIO import StringIO

from astropy.table import Table
from gavo import votable

accessURL = "http://dc.zah.uni-heidelberg.de/__system__/tap/run/tap"
query = "SELECT TOP 3 * FROM TAP_SCHEMA.tables"
job = votable.ADQLTAPJob(accessURL, query)
try:
  job.run()
  table = Table.read(StringIO(job.openResult().read()), format="votable")
finally:
  job.delete()

print table
```

More on dataIterator and metadata can be found in the GAVO VOTable library documentation.

The ADQLTAPJob

The class you will usually deal with is ADQLTAPJob, which is constructed with the endpoint URL and the query. The construction will access the, which means it may very well raise network-related exceptions.
You can pass a `userParams` dictionary to the constructor. This is intended for tho TAP parameters (in particular, `FORMAT`, `MAXREC`, `RUNID` or service-defined ones). You should not use `userParams` for `UPLOAD` but instead use the `addUpload` method described below.
You can also change the parameters later using `setParameter(key, value)`. This again causes a server connection to be made, as are accesses to ADQLTAPJob's properties, viz.,

- `executionDuration` -- the number of seconds after which the server will kill your job. Simply assign some integer to change it, though of course the server might not let you.
- `destruction` -- a `datetime.datetime` (in UTC) at which the job will be completely removed (i.e., even the results) from the remote server. Again, you can assign `datetime` instances to try and change it.
- `phase` -- the current phase of the job. This is a string containing magic values; the possible values are PENDING, QUEUED, EXECUTING, COMPLETED, ERROR, ABORTED (these are also available as symbols in tapquery -- this provides some ward against typos). Most of them are pretty self-explanatory, except that PENDING means you still can change the query, whereas QUEUED jobs are, well, in the server's queue and cannot be changed any more. You should not assign to `phase` manually.
- `quote` -- returns an estimate of the number of seconds your job will execute on the remote machine. This is, of course, a guess even under the most favorable circumstances. Some servers choose to not even try to guess, in which case you'll get a None. This cannot be assigned to.
- `parameters` -- a dictionary containing your parameters. You cannot assign to parameters, and changing things here has no effect. Use setParameter(key, value) instead.

Running a job

Once you have constructed (and possibly modified) a job, call `start()` to tell the remote server to put it into its execution queue. You can then poll the job's phase (now and then):

```
job = votable.ADQLTAPJob(...)
job.executionDuration = 6000
job.start()
while job.phase not in set([tapquery.ERROR, tapquery.COMPLETED]):
  time.sleep(10)
```

You can call `abort()` to kill a running job, and you can call `delete()` when done. As a matter of fact, although server operators will eventually destroy your job anyway, it's

common courtesy to clean up behind you unless you have good reason to keep the result data. So the recommended pattern is:

```
job = votable.ADQLTAPJob(...)
try:
  ... do your thing ...
finally:
  job.delete()
```

Now, doing the polling by hand is tedious, and this there is a function `waitForPhases(phases, ...)` that takes a set of phases to wait for, and then polls at increasing intervals (that you could control though the method's further arguments; see API docs for details).
But really, the `run` method is what you'd usually use; it has the added advantage that it raises a `RemoteError` or a `RemoteAbort` exception if something went wrong on the server side. So, the standard pattern is:

```
job = votable.ADQLTAPJob(...)
try:
  ... add uploads, set parameters if you must ...
  ... change UWS parameters (executionDuration, destruction, etc) ...
  job.run()
  ... read results ...
finally:
  job.delete()
```

TAP parameters

The following parameters are defined by the TAP spec:

- `FORMAT` -- the format you want to retrieve the data in. This defaults to votable, and you should probably keep that default with this library (since, if you have it, you can parse VOTables, right?). Other possibile values include csv, tsv, fits, text, or html. Servers must support VOTables, the other formats are optional
- `MAXREC` -- a limit as to how many rows are to be returned. This basically works like the `TOP` phrase in ADQL and is rather superfluous when using ADQL.
- `RUNID` -- some identifier you can pass. It could be used for tracking and similar. The server should include it in results. If you don't know what it's for, you probably don't need it.
- `LANG` -- the query language. In this library, it defaults to `ADQL`. Let's see if the library is flexible enough to support other languages (which are not specified yet).
- `REQUEST` -- specifies the operation you want from the server. `"doQuery"` is what `ADQLTAPJob` fills in for you, and it's what you should leave it at.
- `QUERY` -- the query you are posing. `ADQLTAPJob` specifies it for you, but if you really wanted to, you could override it (e.g., using `setParameter`.
- `UPLOAD` -- table uploads and such. While you could manipulate this manually, don't. Use the `addUpload` method.

Getting results

Right now, this library heavily leans towards ADQL. When doing ADQL queries, there is only one result. You can access it using the `openResult()` method, which returns a file-like object (actually, it's what `urllib.urlopen` returns) that you can read your results from. Since successful ADQL queries return single-table VOTables, you can feed this directly to votable.load(f):

```
data, metadata = votable.load(job.openResult())
```

ADQL TAP results are simple web resources. To get their URLs, there's the `getResultURL` method. This is particularly useful with uploads (see below).

More complex query patterns could yield more results; in that case, you will have to inspect `job.results`, which is a list of triples of an access URL, an opaque string-typed id, and a UWS "result type" that you probably can safely ignore.

Errors

All exceptions originating in the library are subclasses of `tapquery.Error`. For simple applications, this is also available as `votable.TAPQueryError`.

The API docs list some exceptions you should expect; also, of course, all kinds of network-related exceptions could come out of the library. No serious attempt is being made to catch such exceptions and translate them. If something is wrong network-wise, we feel it's better to freely admit this.

TAP-level errors (like syntax errors in the query, timeouts, and the like) leave error information server-side. Rather than inspecting the exception objects, you should use the `getErrorFromServer` method on the ADQLTAPJob.

In this, proper TAP error messages (those coming back as VOTables) are parsed out, data that's not parseable as a TAP error message is returned as-is; thus the string returned may be long (e.g., some fancy 404 HTML page).

Uploads

TAP allows uploads to servers, although not all services actually support this (you can use a service's capabilities to figure that out; we may add support for parsing those if there's demand).

Services that do support it let you at least upload using URLs to VOTables or upload VOTables inline. Both cases are supported in ADQLTAPJob's `addUpload(name, data)` method. `name` is the name that the table will be visible as later. Use something that works as a regular SQL identifier here.

`data` can be either a string (which is then interpreted as the URL to take the upload from) or a file-like object (that is then turned into an inline upload). In both cases, only VOTable is (reliably) supported as the upload format.

Note that URLs of previous results work as upload URLs. Here's an example of how that might look like, where this retrieves all objects in PPMXL in the vicinity of a detection of a neutrino:

```
from gavo import votable

accessURL = "http://dc.g-vo.org/tap"
job1 = votable.ADQLTAPJob(accessURL,
  "SELECT nualpha, nudelta FROM amanda.nucand WHERE nch>100")
job1.run()

job2 = votable.ADQLTAPJob(accessURL,
  "SELECT DISTINCT TOP 10000 raj2000, dej2000, pmRA, pmDE"
  " FROM ppmxl.main"
  " JOIN TAP_UPLOAD.nupos"
  "   ON (1= CONTAINS(POINT('ICRS',raj2000,dej2000),"
  "      CIRCLE('ICRS', nualpha , nudelta, 0.25)))")
job2.addUpload("nupos", job1.getResultURL())
job2.run()

dataIterator, metadata = votable.load(job2.openResult())
job1.delete()
job2.delete()
print list(dataIterator)
```

Sync Querying

TAP also supports a synchronous querying mode; for fast-running queries, this is simpler and has less overhead. So, if you are certain that your query will run quickly and the result set is small, you can use the ADQLSyncJob. This works mostly like the ADQLTAPJob, except of course everything that deals with remote state management basically is a no-op. Instead, either `run` or `start` just query the remote server and return when the server is done. For convenience, they return the job itself, so that you can say things like:

```
print votable.ADQLSyncJob(
  "http://dc.zah.uni-heidelberg.de/__system__/tap/run/tap",
  "SELECT * FROM TAP_SCHEMA.tables"
  ).run().openResult().read()
```

All "unpredictable" exceptions on sync jobs are raised from within `run`; these will usually be `tapquery.WrongStatus` or `tapquery.NetworkError` exceptions, for when the TAP server has complained or something was wrong on the way between the client and the server. The `WrongStatus` instances have a payload attribute that contains any message body the server might have sent with the headers; frequently this contains explanations what may have gone wrong. Since no http headers are available, there's no saying what format the error message came in. Tell us if that bugs you.

The error associated with the exception object will usually not be particularly useful. Instead, obtain an error message from the server using the `getErrorFromServer` method like for ADQLTAPJobs.
