The GAVO ADQL Library

Author: Markus Demleitner
Email: gavo@ari.uni-heidelberg.de
Date: 2019-01-11

A library to deal with ADQL and have it executed by postgres

Author: Markus Demleitner

Email: gavo@ari.uni-heidelberg.de

This library tries to provide glue between a service eating ADQL and delivering VOTables on the one side and concrete DBMSes (currently only postgres) on the other.

To do this, it parses ADQL, tries to infer types, units, systems, and UCDs for the result columns ("annotation"), and rewrites ("morphs") queries so they can be executed on postgres.

Basic Usage

The standard way to access the package is:

```
from gavo import adql
```

The adql namespace contains the functions documented below. To see if things work at least to some degree, try:

```
In [1]:from gavo import adql

In [2]:from pprint import pprint

In [3]:t = adql.parseToTree("SELECT * FROM t WHERE 1=CONTAINS("
   ...:                 "CIRCLE('ICRS', 4, 4, 2), POINT('', ra, dec))")

In [4]:pprint t.asTree()
```

```
------>pprint(t.asTree())
('querySpecification',
 ('fromClause', ('possiblyAliasedTable', ('tableName',))),
 ('selectList',),
 ('whereClause',
  ('comparisonPredicate',
   ('factor',),
   ('predicateGeometryFunction',
    ('circle', ('factor',), ('factor',), ('factor',)),
    ('point', ('columnReference', 'ra'), ('columnReference', 'dec'))))))
```

Annotations

One central task of the ADQL library is the inference of column metadata from queries. To receive an annotated tree, use code like:

```
adql.parseAnnotated(query, getFieldInfo)
```

`query` is the ADQL input, `getFieldInfo` a function described below. This is the main entry point into the library.

FieldInfo objects have the following attributes:

- type -- an SQL type name (see below for the type system). Use lower case.
- ucd, unit
- tainted -- a boolean specifying whether the library had to guess anything (a simple scalar multiplication is enough; see below)
- userData -- a sequence of opaque data passed in by the host application (see below)
- stc -- None or an AST from DaCHS STC.

The source for these annotations is the metadata of the input columns. These are communicated to the library through the `getFieldInfo` callback passed to `annotate`. It has the signature:

```
getFieldInfo(tableName) -> list of info pairs,
```

where an info pair consists of the column's name and quintuple of:

```
(type, unit, ucd, userdata, stc)
```

`userdata` is a tuple of application specific column descriptions; on input, these should always be tuples of length 1. The library will collect those, and all userdata objects that went into a particular `FieldInfo` can be retrieved under its `userdata` attribute. *stc* is a DaCHS STC AST.

A `tableName` passed to getFieldInfo is an object with schema and name attributes. Tables from TAP_UPLOADS appear with an empty schema here.

The Type System

TBD. See the tree in adql.fieldinfo for the names currently understood.

User Functions

TBD.

Morphing

TBD

Note that `parseAnnotated` applies a standard morphing, mapping INTERSECTS calls having a POINT as one argument to a corresponding CONTAINS call as per 2.4.11 of the ADQL spec. We hope this makes mapping the function calls to efficient database operations easier.

Custom Region Specifications

The ADQL library always accepts STC-S for regions. It will raise an error if the STC specifies no or more than one region.

To enable more region specifications, define region makers. Region makers are functions taking the argument to REGION and trying to do something with it. They should return either some kind of FieldInfoedNode that will then replace the REGION or None, in which case the next function will be tried. Anything not derived from a FieldInfoedNode is not suitable as a return value in general since Regions might be annotated.

As a convention, region specifiers here should always start with an identifier (like simbad, siapBbox, etc, basically [A-Za-z]+). The rest is up to the region maker, but whitespace should separate this rest from the identifier.

Here's an example of a region looking up a Simbad identifier (using some DaCHS code for the actual Simbad interface):

```
def _getRegionId(regionSpec, pat=re.compile("[A-Za-z_]+")):
  mat = pat.match(regionSpec)
  if mat:
    return mat.group()

def _makeSimbadRegion(regionSpec):
  if not _getRegionId(regionSpec)=="simbad":
    return
  object = "".join(regionSpec.split()[1:])
  resolver = base.caches.getSesame("web")
  try:
    alpha, delta = resolver.getPositionFor(object)
  except KeyError:
    raise adql.RegionError("No simbad position for '%s'"%object)
  return adql.getSymbols()["point"].parseString("POINT('ICRS',"
    "%.10f, %.10f)"%(alpha, delta))
adql.registerRegionMaker(_makeSimbadRegion)
```