How Do I?

Recipes and tricks for solving problems using GAVO DaCHS

Author: Markus Demleitner
Email: gavo@ari.uni-heidelberg.de
Date: 2018-06-20

Contents

Ingestion

...do incremental importing?

...that is, only import files that were added since the last import?

The key ingredient here is the Element ignoreSources, which lets you tell DaCHS which files or those it found in `sources` it should not ingest (again). What this deals with are paths relative to DaCHS' inputsDir. There are various ways to communicate them: from lines in a file, by a pattern, or from the database, and it's the last case we're really interested in here.

Unless you did something special, the accref column in tables having data products will be such an this inputsDir-relative path. With the `fromdb` attribute of the `ignoreSource` element, you can pull in these accrefs as the files to ignore.

DaCHS, however, when you `imp` something, will always tear down all tables being made. To suppress that behaviour, just give the data element an `update` attribute. In sum, the pattern is:

```
<data id="import" updating="True">
  <sources pattern="data/*.fits">
    <ignoreSources fromdb="select accref from myschema.mytable"/>
  </sources>...
```

Of course, you have to adapt the source pattern and the table to get the accrefs for (to where your data products are managed).

For non-data product sources (e.g., files you feed into catalogs), you'll have to use some other technique. In case you need such a thing, you may want to look at rr/q (see the `rr.imported` table).

...skip a row from a rowmaker?

Raise IgnoreThisRow in a procedure application, like this:

```
<rowmaker id="build_mytable" idmaps="*">
  <apply>
    <code>
      if 2+colX>22:
        raise IgnoreThisRow()
    </code>
  </apply>
</rowmaker>
```

However, it's probably more desirable to use the rowmakers' built-in `ignoreOn` feature, possibly in connection with a procedure, since it is more declarative.

Still, the following is the recommended way to selectively ignore broken records defined via certain identifiers:

```
<apply name="ignoreBrokenRecords">
  <doc>This proc filters out records too broken to ingest.
  for the set of ids in the toIgnorePar
  </doc>
  <setup>
    <par name="toIgnore">set([
      202, 405])
    </par>
  </setup>
  <code>
    if @catid in toIgnore:
      raise IgnoreThisRow("Manually ignored from RD")
  </code>
</apply>
```

...skip a single source?

If you want to skip processing of a source, you can raise SkipThis from an appropriate place. Usually, this will be a sourceFields element, like this:

```
<reGrammar>
  <sourceFields>
    <code>
      if len(sourceToken)>22:
        raise base.SkipThis("%s skipped since I didn't like the name"%
          sourceToken)
     </code>
  </sourceFields>
</sourceFields>
```

...fix duplicate values?

There are many reasons why you could violate the uniqueness constraints on primary keys, but let's say you just got a message saying:

```
Primary key <whatever> could not be added ('could not create unique
index "data_pkey" DETAIL:  Table contains duplicated values.)'
```

The question at this point is: What are the duplicated values? For a variety of reasons, DaCHS only applies constraints only after inserting all the data, so the error will occur at the end of the input. Not even the `-b1` trick will help you here.

Instead, temporarily remove the primary key condition from the RD and import your data.

Then, exececute a query like:

```
select *
from (
  select <primary-col>, count(*) as ct
  from <your-table>
  group by <primary-col>) as q
where ct>1;
```

...cope with "Undefined"s in FITS headers?

Pyfits returns some null values from FITS headers as instances of "Undefined" (Note that this is unrelated to DaCHS' base.Undefined). If you want to parse from values that *sometimes* are Undefined, use code like:

```
<bind name="alpha">parseWithNull(@RA,
  lambda l: hmsToDeg(l, ":"),
  checker=lambda l: isinstance(l, utils.pyfits.Undefined))</bind>
```

...force the software to accept weird file names?

When importing products (using //products#define or derived procDefs), you may see a message like:

```
File path
'rauchspectra/spec/0050000_5.00_H+He_0.000_1.meta'
contains characters known to the GAVO staff to be hazardous in URLs.
Please defuse the name before using it for published names.
```

The machinery warns you against using characters that need escaping in URLs. While the software itself should be fine with them (it's a bug if it isn't), such characters make other software's lives much harder – the plus above, for example, may be turned to a space if used in a URL.

Thus, we discourage the use of such names, and if at all possible, you should try and use simpler names. If, however, you insist on such names, you can simply write something like:

```
<rowfilter procDef="//products#define">
  <bind key="table">"\schema.newdata"</bind>
  <bind key="accref">\inputRelativePath{True}</bind>
</rowfilter>
```

(plus whatever else you want to define for that rowfilter, of course) in the respective data element.

Note that, at least as of DaCHS 1.0, due to postgres limitations this will generate a bad publisher DID in obscore. To fix that, explicitly have a publisher DID column in your table and fill it with `getStandardPubDID(@accref)`. Then pass that column to te obscore mixin's publisherDID parameter. A non-annoying fix would require a urlencode function in postgres – which DaCHS could define, but we'd rather not.

...handle formats in which the first line is metadata?

Consider a format like the metadata for stacked spectra:

```
C 2.3 5.9 0
1 0 0 1
2 1 0 1
...
```

– here, the first line gives a center position in degrees, the following lines offsets to that.

For this situation, there's the grammar's sourceFields element. This is a python code fragment returning a dictionary. That dictionary's key-value pairs are added to each record the grammar returns.

For this example, you could use the following grammar:

```
<reGrammar names="obsNo, dra, dde" topIgnoredLines="1">
  <sourceFields name="getBasePos">
    <code>
      with open(sourceToken) as f:
```

```
        _, cra, cde, _ = f.readline().split()
        return {
          "cra": float(cra),
          "cde": float(cde)}
    </code>
  </sourceFields>
</reGrammar>
```

In the rowmaker, you could then do something like this:

```
<map dest="raj2000">@cra+float(@dra)*DEG_ARCSEC</map>
<map dest="dej2000">@cde+float(@dde)*DEG_ARCSEC</map>
```

...use binaries when the gavo directory is mounted from multiple hosts?

If your GAVO_ROOT is accessible from more than one machine and the machines have different architectures (e.g., i386 and amd64 or ppc, corresponding to test machine and production server), compiled binaries (e.g., getfits, or the preview generating code) will only work on one of the machines.

To fix this, set platform in the [general] section of your config file. You can then rename any platform-dependent executable base-<platform>, and if on the respective platform, that binary will be used. This also works for computed resources using binaries, and those parts of the DC software that build binaries (e.g., the booster machinery) will automatically add the platform postfix.

If you build your own software, a make file like the following could be helpful:

```
PLATFORM=$(shell dachs config platform)
TARGET=@@@binName@@@-$(PLATFORM)
OBJECTS=@@@your object files@@@

$(REG_TARGET): buildstamp-$(PLATFORM) $(OBJECTS)
        $(CC) -o $@ $(OBJECTS)

buildstamp-$(PLATFORM):
        make clean
        rm -f buildstamp-*
        touch buildstamp-$(PLATFORM)
```

You'll have to fill in the @@@.*@@@ parts and probably write rules for building the files in $OBJECT, but otherwise it's a simple hack to make sure a make on the respective machine builds a suitable binary.

...transform coordinates from one coordinate system to another?

In general, that's fairly complicated, involving proper motions, radial velocities, and other complications. There's an interface in the stc module based on conformTo, but you'd not want to do that when filling tables since it's not fast and usually not what you need either.

For simple things (i.e., change plain positions between Galatic and ICRS or Equinox B1965 and J2010) and bulk transforms, you can use the following pattern in a rowmaker apply (transforming from Galactic to ICRS):

```
<apply name="convertPosition">
  <setup>
    <code>
      gal2ICRS = stc.getSimple2Converter(
        stc.parseSTCS("Position GALACTIC"),
        stc.parseSTCS("Position ICRS"))
    </code>
  </setup>
  <code>
    @raj2000, @dej2000 = gal2ICRS(@galLon, @galLat)
  </code>
</apply>
```

...access the creation date of the file from which I'm parsing?

The row iterator that feeds the current table is in rawdicts under the key `parser_`. For grammars actually reading from files, the `sourceToken` attribute of these gives the absolute path of the input file. Hence, you'd say:

```
<map dest="modstamp">os.path.getmtime(@parser_.sourceToken)</map>
```

...update ivoa.obscore when I've changed something in an obscore mixin?

When you change a specification in an obscore mixin, it would be a waste to re-import the table with the mixin, since in general nothing has changed there. However, the script updating the obscore view will only be run on a full `dachs imp`, not on a `dachs imp -m`.

Since version 0.9.7, `dachs imp -m` now at least updates the view fragment when the table metadata of a table with obscore is updated. It does not, however, re-make the view. You can easily do that, manually, however:

```
dachs imp //obscore
```

Services in General

...set a constant input to a core?

Use a service input key with a Hidden widget factory and a default:

```
<service id="cutout" ...>
  ...
  <inputKey name="cutoutSize" widgetFactory="Hidden">
    <values default="0.1"/>
  </inputKey>
</service>
```

...add computed columns to a dbCore output?

Easy: define an output field with a select attribute, e.g.:

```
<outputField name="a_U" tablehead="A_U" ucd="phys.absorption;em.opt.U"
  description="Extinction in U, assuming R_V=3.1"
  select="ev_i*5.434"/>
```

This will add an output field that looks to the service like it comes from the DB proper but contains the value of the `ev_i` column multiplied with 5.434.

The expression must be valid SQL.

There is a fairly common case for this that's somewhat tricky: Compute the distance of the position of a match to the position of a cone search (or endless variants of that). Tasks like that are hard for DaCHS, as the select clause (where the distance is computed) needs information on something only avaialable to the condition descriptors (the input position).

There's a slightly hackish way around this. It builds on the predictability of the names chosen by `base.getSQLKey` and the way parameters are passed through to PostgreSQL. Hence, this is a portability liability. We'd like to hear about your applications for this, as that may help us figure out a better solution. Meanwhile, to add a spherical distance to a cone search service over a table with positions in raj2000, dej2000, say something like:

```
<outputTable verbLevel="20">
  <outputField name="_r"
    unit="deg" ucd="pos.andDistance"
    tablehead="Dist"
    description="Distance to the queried position"
    select="DEGREES(spoint(RADIANS(raj2000), RADIANS(dej2000))
      &lt;->spoint(RADIANS(%(RA0)s), RADIANS(%(DEC0)s)))"/>
</outputTable>
```

As you can see, this unconditionally assumes that the parameter names for the cone condition are in RA0, DEC0; it is likely that that is true, but this is not API-guaranteed. You could write your condDesc with fixed field names to be safe, but even then this builds on the specific API of psycopg, so it's certainly not terribly portable.

Another issue is that not all queries necessarily give RA and Dec; as pulled in by the ususual `//scs#coreDescs` STREAM, cone conditions in forms are not mandatory. The above construct will yield fairly ugly errors when they are left out. To fix this, make humanInput mandatory, saying something like:

```
<condDesc original="//scs#protoInput"/>
<condDesc original="//scs#humanInput" required="True"/>
```

instead of the coreDescs (maybe adding the built-in MAXREC, too).

...import data coming in to a service?

In a custom renderer or core, you can use code like:

```
from gavo import api

...

def import(self, srcName, srcFile):
  dd = self.service.rd.getById("myDataId")
  with api.getWritableAdminConn() as conn:
    self.nAffected = api.makeData(dd, forceSource=srcFile,
      connection=conn).nAffected
```

You want to use a separate connection since the default connections obtained by cores and friends are unprivileged and typically cannot write to table.

The nAffected should contain the total number of records imported and could be used in a custom render function.

srcName and srcFile come from a formal File form item. In submitActions, you obtain them like:

```
srcName, srcFile = data["inFile"]
```

Note that you can get really fancy and manipulate `data` in some way up front. That could look like this:

```
from gavo import rsc
...
data = rsc.Data.create(dd, parseOptions=api.parseValidating,
  connection=conn)
data.addMeta("_stationId", self.stationRecord["stationId"])
self.nAffected = api.makeData(dd, forceSource=srcFile, data=data,
  connection=conn).nAffected
```

...define an input field doing google-type full text searches?

Since version 8.3 (or so), postgres supports query modes inspired by information retrieval on text columns -- basically, you enter a couple of terms, and postgres matches all strings containing them.

Within DaCHS, this is currently only supported using custom phrase makers. This would look like this:

```
<condDesc>
  <inputKey name="columnwords"
    description="Terms in column description (case does not
      matter, patterns not allowed)"
    type="text" tablehead="Column words"/>
  <phraseMaker>
    <code>
      yield ("to_tsvector('english', description)"
        " @@ plainto_tsquery('english', %%(%s)s)"%(
        base.getSQLKey("columnwords", inPars["columnwords"], outPars))
    </code>
  </phraseMaker>
</condDesc>
```

-- here, `description` is the column containing the strings, and the `'english'` in the function arguments gives the language according to which the strings should be interpreted.

You may want to create an index supporting this type of query on the respective columns, too. To do that, say:

```
<index columns="bibref" method="gin">
  to_tsvector('english', bibref)</index>
```

Relatedly, if what you want is matches *within* strings (as in `foo LIKE '%bar%'`), this will not help you. The way to get these indexed is to use the *pg_trgm* module. You'll have to create the extension manually (it's shipped with postgres since 9.1, though):

```
CREATE EXTENSION pg_trgm
```

After that, you can say:

```
<index columns="table_name" method="gin"
  >LOWER(table_name) gin_trgm_ops</index>
```

(this is to speed up searching with the ivo_nocasematch ADQL user defined function; drop the LOWER as you see fit.

...protect my TAP service against usage?

Sometime there's a proprietarity time on some data, and you may want to password protect your data. DaCHS doesn't support protection of individual tables yet (we'll accept patches, but since this kind of thing is considered fairly antisocial by most of us, it looks bleak for an implementation of this from our side). You can, however, protect the entire TAP service (with HTTP basic auth, so this isn't for the gold of Fort Knox).

First, create a group of authorized users and add a user to it:

```
dachs admin adduser tapusers master_password dachs admin ad-
duser smart_pi his_password dachs admin addtogroup smart_pi
tapusers
```

Then, limit access to the TAP service itself. To make it easy to redo your changes and return to mainline development, we recommend to use a custom TAP RD. To do that, create a `__system__` directory in your inputs directory; with the default config:

```
cd /var/gavo/inputs
mkdir __system__
cd __system__
curl -O http://svn.ari.uni-heidelberg.de/svn/gavo/python/trunk/gavo/resources/inputs/__system__/
```

(or find your local tap.rd and use that, which may be a bit safer).

In tap.rd, look for the actual service element:

```
<service id="run" core="null" allowed="tap">
```

In that copy, write `limitTo="tapusers"`. You need to restart the server to pick up the new RD, after that, it should pick up changes by itself. To get back standard behaviour, just remove that copy.

That's not enough, though; you'll need to do the same with the service in adql.rd (or remove that service altogether).

This is not sufficient for top secret things -- people can still inspect your column metadata, descriptions and so on. But they shouldn't get to the data.

Not all TAP clients support HTTP basic authentication, but several do. For instance, since v4.0-1 TOPCAT just pops up a box requesting the credentials. On older releases you can still set the star.basicauth.user and star.basicauth.password java system properties. See http://www.star.bristol.ac.uk/~mbt/topcat/sun253/jvmProperties.html on how to do this.

...have output columns with URLs in them?

You might be tempted to use `outputField/formatter` for this, and indeed this lets you generate arbitrarily odd things, including javascript, in whatever you make of the value coming in. But nothing but a web browser can do anything with this kind of thing. Therefore, DaCHS does not execute formatters for non-HTML output, which means for something like:

```
<outputField name="obsid" type="text"
  description="Link to retrieve data">
  <formatter>
    return T.a(href="http://my.data.serv.er/customservice?"+data)[
      "Download"]
  </formatter>
</outputField>
```

VOTable and friends will only have the value of whatever obsid is in their corresponding column, which in this case is probably *not* what you want.

Instead, use the `select` attribute of `outputField` to generate the URL within the database, like this:

```
<outputField name="obsURL" type="text"
  description="Link to retrieve data"
  select="'http://my.data.serv.er/customservice?'||obsid"
  displayHint="type=url"/>
```

The `type=url` display hint will generate links in HTML (and possibly some other appropriate thing for URLs in particular in other output formats). The anchor text is the last item from the path component of the URL, which is not always what you want. To override it, you can use a formatter, or, in simple cases when a constant text will do, the anchorText property, like this:

```
<outputField name="obsURL" type="text"
  description="Link to retrieve data"
  select="'http://my.data.serv.er/customservice?'||obsid"
  displayHint="type=url">
  <property key="anchorText">Retrieve Data</property>
</outputField>
```

...wrap an external service with datalink?

Say you have an existing ephemeris computing service that conforms to no standard and you'd like to make it usable through a datalink client. The trick is to declare the parameters in DaCHS, compute a target URL and then redirect. Note that this will not be as easy if the wrapped services require POSTing or file uploads, because these will, in general, not survive POSTing.

In the following example, we first define a custom descriptor generator; we'll need that here as DaCHS' built-in descriptor generators all assume they can find the referenced data sets in the `dc.products` table. In this case, we assume the data is remote and all we have is what is in a database table. In EPN-TAP style, the dataset is identified with the `granule_uid` column.

Then, we define our parameter range from a database column – note how the descriptor is used to communicate the database row.

Finally, in the data function we construct the destination URL and then redirect to the process that actually serves the data:

```
<service id="dl-cutout" allowed="dlget,dlmeta">
  <datalinkCore>
    <descriptorGenerator>
      <setup>
        <code>
          class CustomDescriptor(object):
            def __init__(self, granule_uid):
              self.pubDID = granule_uid
              with base.getTableConn() as conn:
                self.metadata = list(
                  conn.queryToDicts("SELECT * FROM \schema.epn_core"
                    " WHERE granule_uid=%(guid)s",
                    {"guid": self.pubDID}))[0]
        </code>
      </setup>
      <code>
        return CustomDescriptor(pubDID)
      </code>
    </descriptorGenerator>

    <metaMaker>
      <code>
```

```
          yield MS(InputKey, name="TIME_MJD",
            type="double precision[2]",
            xtype="interval",
            ucd="meta.epoch",
            description="Range of data to return.",
            multiplicity="forced-single",
            required=True,
            values=MS(Values,
              min=jdnToMJD(descriptor.metadata["time_min"]),
              max=jdnToMJD(descriptor.metadata["time_max"]),))
        </code>
      </metaMaker>

      <dataFunction>
        <setup>
          <par name="server_url"
            >'http://custom-upstream.example.edu/cutout?'</par>
          <code>
            import urllib
            from gavo.svcs import WebRedirect
          </code>
        </setup>
        <code>
          time_interval = args["TIME_MJD"]
          parameters = {
            "magic_value": "any_madness",
            "dataset": descriptor.pubDID,
            "start_time": mjdToDateTime(time_interval[0]).isoformat(),
            "end_time": mjdToDateTime(time_interval[1]).isoformat(),
          }
          raise WebRedirect(server_url+urllib.urlencode(parameters))
        </code>
      </dataFunction>
    </datalinkCore>
  </service>
```

Form-based interfaces

...get a multi-line text input for an input key?

Use a widgetFactory, like this:

```
<inputKey original="adqlInput.query"
    widgetFactory="widgetFactory(ScalingTextArea, rows=15)"/>
```

...make an input widget to select which columns appear in the output table?

In general, selecting fancy output options currently requires custom cores or custom renderers. Ideas on how to change this are welcome.

For this specific purpose, however, you can simply define an service key named _ADDITEM. This would look like this:

```
<service ....>
  ....
   <inputKey name="_ADDITEM" tablehead="Extinctions for band"
    description="Output extinctions for the respective band (if you want multiple bands, you can
    type="text" required="True" mul>
    <values multiOk="True" showItems="-1">
      <option title="Landolt U">a_U</option>
      <option title="Landolt V">a_V</option>
      <option title="Landolt I">a_I</option>
    </values>
  </inputKey>
  ...
</service>
```

Setting showItems to -1 gives you checkboxes rather than a select list, which is mostly what you want. Try with and without and see what you like better.

If you do that, you *probably* do not want the standard "additional fields" widget at the bottom of the form. To suppress it, add a line

```
<property name="noAdditionals">True</property>
```

to the service definition. The "True" in there actually is a bit of a red herring, the widget is suppressed for any value.

...add an image to query forms?

There are various variations to that theme -- you could go for a custom template if you want to get fancy, but usually putting an image into an _intro or _bottominfo meta section should do.

In both cases, you need a place to get your image from. While you could put it somewhere into rootDir/web/nv_static, it's probably nicer to have it within a resource's input directory. So, add a static renderer to your service, like this:

```
<service allowed="form,static">
  <property name="staticData">static</property>
```

This lets you put service-local static data into resdir/static/ and access it as <serviceURL>/static/<filename>

Usually, your _intro or _bottominfo will be in reStructured text. Plain images work in there using substitution references or simply the naked image directive:

```
<meta name="_bottominfo" format="rst">
  The current data set comprises these fields:

  .. image:: \servicelink{cars/q/cat/static/fields-inline.png}
</meta>
```

The servicelink macro would ensure that the image would still be found if the server ran off-root.

This is the recommended way of doing things. If, however, you insist on fancy layouts or need complete control over the appearance of your image (or whatever), you can use the evil "raw" meta format:

```
<meta name="_bottominfo" format="raw"><![CDATA[
  <a href="\servicelink{cars/q/cat/static/fields.png}">
  <img src="\servicelink{cars/q/cat/static/fields-inline.png}"/>
  </a>]]>
</meta>
```

Make sure you enter valid HTML here, no checks are done by the DC software.

...put more than one widget into a line in web forms?

Use input table groups with a <property name="style">compact</property>. In DB cores, however, you probably do not want to give inputTables explicitely since it's much less hassle to have them computed from the condDescs. In this case, the widgets you want to group probably come from a single condDesc. To have them in a group, define a group within the condDesc without any paramRefs (or colRefs) -- they cannot be resolved anyway. Give the group style and label properties, and it will be added to the input table for all fields of the condDesc:

```
<condDesc name="example">
  <inputKey original="col1"/>
  <inputKey original="col2"/>
  <group description="An example group that will end up in one line">
    <property name="style">compact</property>
    <property name="label">Example vals</property>
  </group>
</condDesc>
```

If you are doing this, you probably want to use the `cssClass` property of input keys and the `customCSS` property of services. The latter can contain css specifications. They are added into form pages by the defaultresponse template (in your custom templates, you should have `<style type="text/css" n:render="servicestyle"/>`

in the head if you want this ufnctionality). These can be used to style form elements by their css class, which in turn is given by specifying `cssClass` properties on inputKeys.

Here's an example that uses CSS to insert material, which currently is the only way to input something between the fields (short of redefining the widgets). This may be overdoing it since the usability of this widget without CSS is questionable (whereas it would be fine if the group were non-compact and no CSS tricks were played); first a condDesc template:

```
<STREAM id="massfrac_template">
  <doc>A condDesc for a mass fraction.  These consist of an element label,
  a center value and a weird way of specifying the error.

  There can be a few of them for a given service, and thus you need to
  define the macro item. It is used to disambiguate the names.
  </doc>

  <condDesc original="massfrac_cond">
    <inputKey name="el\item" type="text" tablehead="Element \item"
        description="Element for mass fraction \item">
      <values>
        <LOOP csvItems="\elements">
          <events>
            <option>\item</option>
          </events>
        </LOOP>
      </values>
    </inputKey>
    <inputKey name="mfval\item" tablehead="Mass fraction \item"
        description="Mass fraction c\item">
      <property name="cssClass">a_val</property>
    </inputKey>
    <inputKey name="mffuzz\item" tablehead="Fuzz \item"
      description="Range width r for mass fraction \item.  A mass fraction
      matches if it is between c/10^r and c*10^r">0.3<property
        name="cssClass">a_fuzz</property></inputKey>
    <group name="mf\item">
      <description>Mass fraction of an element.  The condition expands
      to c/10^r &#8804; mass fraction(Element) &#8804; c*10^r</description>
      <property name="label">Mass Fraction \item</property>
      <property name="style">compact</property>
    </group>
  </condDesc>
</STREAM>
```

And here's a service with that `condDesc`, including the custom css:

```
<service id="web">
  <meta name="title">Theoretical spectra of hot compact stars</meta>
```

```
  <dbCore queriedTable="newdata">
    <LOOP listItems="1 2" source="massfrac_template"/>
  </dbCore>
  <property name="customCSS">
    input.a_val {width:3em}
    input.a_fuzz {width: 3em}
    span.a_val:before { content:" in "; }
    span.a_fuzz:before { content:" &#x00b1; "; }
    span.a_fuzz:after { content:" dex"; }
  </property>
</service>
```

Note that we are styling both `input` and `span` elements with the css class we set. Before and after can only be used on the span since input elements do not support before and after. For that reason, DaCHS wraps each element within a compact group with a span of the same css class.

Also see the next question.

...get a range input widget?

Well, VizieR expressions let your users specify intervals and more, but ok, they would need to read docs to know that, so there's a case to be made for widgets like:

```
Phlogistics      between __________  and __________
```

These basically work as discussed in grouping widgets above, but since stuff like this is fairly common, there's built-in support for this in //procs#rangeCond. This is a stream requiring three definitions:

- name -- the column name in the core's queried table
- groupdesc -- a terse phrase describing the range. This will be used in the description of both the input keys and the group
- grouplabel -- a label (include the unit, it is not taken from InputKey) written in front of the form group

groupdesc has to work after "Range of", "Lower bound of", and "Upper bound of". Do not include a concluding period in groupdesc.

Here's an example:

```
<dbCore queriedTable="data">
  <FEED source="//procs#rangeCond"
    name="t_eff"
    groupdesc="the atmosphere's effective temperatures to include"
    grouplabel="Effective Temperature [K]"/>
```

...use a custom input widget for a field?

Right now, you cannot really; we're still waiting for an enlightenment on how to sensibly do that from RDs. Pester us if you really want this.

Meanwhile, to have little tags like the explanations of the vizier-like-expressions, you can use a custom widget with fields. This could look like this:

```
<condDesc>
  <inputKey original="object" type="vexpr-string">
    <description>Designated object on the plate (i.e.,
    the object the observers entered into their journal).
    You can use wildcards if your prefix your expression with "~".
    </description>
    <widgetFactory
      >widgetFactory(StringFieldWithBlurb, additionalMaterial=
      T.a(href="/objectlist")[
      "[List of known objects]"])</widgetFactory>
  </inputKey>
</condDesc>
```

Here, instead of the String in `StringFieldWithBlurb`, you could have used Numeric or Date, and then used vexpr-float or vexpr-date, respectively, for the inputKey's date.

The value of the `additionalMaterial` argument is some nevow stan. Info on what you can write there can be found elsewhere.

...disable automatic previews in "product" columns?

By default, whenever a table mixes in `//products#table`, in HTML displays the cell values will contain a link that becomes a preview when the user mouses over it. That behaviour is a nuisance if the files cannot be previewed for one reason or another (remote file, unknown file type, etc).

The previews code is inserted when columns with `displayHint="type=product"` are rendered into HTML tables. The code that does this recognizes a second display hint, `nopreview`, which, when set to any value, inhibits the generation of previews.

Regrettably, at this point there is no way to change the display hint in the table itself (we're thinking of a general way for ad-hoc overriding of mixin columns, but unfortunately, that's a bit subtle).

Therefore, you need to fix the display hint when building the output table. The quickest way to do this is by using the output table's `verbLevel` attribute, and then overriding the accref column. This could look like this:

```
<service...>
  ...
  <outputTable verbLevel="20">
    <outputField original="accref" displayHint="type=product,nopreview=1"/>
  </outputTable>
  ...
</service>
```

`verbLevel="20"` selects all columns in the queried table with a `verbLevel` less or equal 20 (which is the default when you don't give an `outputTable`); you can adapt that number to whatever fits your needs.

A downside of this approach is that the image column ends up a the very beginning of each row. For full control over which columns are displayed and which go where, use the following pattern:

```
 <service...>
  ...
  <outputTable>
    <LOOP listItems="source srcId confirmed lensalpha"
      <events>
        <outputField original="\item"/>
      </events>
    </LOOP>

    <outputField
      original="accref" displayHint="type=product,nopreview=1"/>

    <LOOP listItems="raj2000 dej2000">
      <events>
        <outputField original="\item"/>
      </events>
    </LOOP>
  </outputTable>
  ...
</service>
```

An added benefit of this is that the first approach only works starting DaCHS SVN rev. 3351, whereas this has been possible at least since about version 0.7.

But then, the main point is: Don't do this – just provide previews. They're making a world of a difference, and they are in general not hard to make even if DaCHS doesn't support your formats out of the box. See Product Previews for details.

...override metadata for a column coming from a mixin?

Sometimes you want to change a column coming from a mixin, for instance, to be more specific in the description, or to change a display hint. Most mixins add

their columns at table instanciation, which means that the columns provided by them are already available when you define your columns. Also, columns with the same name overwrite existing columns. Thus, if you say:

```
<column original="bandpassLo" displayHint="displayUnit=Angstrom"/>
```

you'll have the metadata from bandpassLo with a new displayHint.

This is also necessary if you want non-ASCII characters in string-valued columns. For efficiency in VOTable serialisation, DaCHS maps the `text` type to VOTable chars, which only allow ASCII. To make DaCHS use VOTable unicodeChars, you need to declare the corresponding fields as `unicode`. To do that, drop something like:

```
<column original="instrument_host_name" type="unicode"/>
```

into the body of the table having the mixin.

...have a custom upload parameter?

Astronomers often want to upload object lists. Typically, they don't know what they're talking about and would really like to use TAP and perhaps a good client or scripting environment. However, there's no arguing with some people, so here's a way to receive an upload and make a query out of it from a web form. Making this use a VOTable and proper metadata is left as an exercise to the operator (but then the browser-upload faction typically isn't hot about that either).

This concrete example matches against the SSA `ssa_location` column using pgsphere; if you were to match against a split-coordinate thing in a table mixing in q3cindex, you'd rather use `q3cjoin(ra, dec, ra, dec, radius)`, again all in degrees:

```
<condDesc>
  <inputKey type="file" name="upload"
    description="Upload a list of target positions; this expects
       a text file with three floats per line (ICRS RA, Dec, radius,
       all in decimal degrees)."/>
  <phraseMaker>
    <code>
      circles = []
      for ln in inPars[inputKeys[0].name][1].readlines():
        ra, dec, sr = map(float, ln.split()[:3])
        circles.append(pgsphere.SCircle(
          pgsphere.SPoint.fromDegrees(ra, dec),
          sr*utils.DEG))
```

```
      yield " OR ".join(
        "ssa_location @ %%(%s)s"%base.getSQLKey("roi", c, outPars)
        for c in circles)
    </code>
  </phraseMaker>
</condDesc>
```

Note that the much-asked-for pattern "do a query, then retrieve results as a tar" here has the snag that it's hard to transport file uploads from one HTTP request to the next and hence your users either have to use the back button and switch the output format on the orginal form or repeat the upload file selection on the form at the foot of the result table.

VO Protocols

...change the query issued on SCS queries?

You may want to do that because for some reason there is no q3c index on the queried table, or the semantics aren't quite a point vs. point cone search but close enough.

Since we've not yet migrated cone search querying to STREAMs, this is a bit clumsy, using the `original` attribute. That said, the current way looks like this (the example checks for an intersection between the search cone and a pgsphere geometry called `origin_est`:

```
<condDesc original="//scs#humanInput">
  <phraseMaker original="//scs#humanSCSPhrase">
    <setup original="//scs#scsSetup" id="proxQuery">
      <code><![CDATA[
        def genQuery(inPars, outPars):
          return ("scircle(spoint(radians(%%(%s)s), radians(%%(%s)s)),"
            " radians(%%(%s)s)) && origin_est")%(
            base.getSQLKey("RA", inPars["RA"], outPars),
            base.getSQLKey("DEC", inPars["DEC"], outPars),
            base.getSQLKey("SR", inPars["SR"], outPars))
      ]]></code>
    </setup>
  </phraseMaker>
</condDesc>
```

-- so, you are inheriting from the SCS condition on three levels and then override the `genQuery` function defined in the common setup code. The way the condDescs are written, you must return rather than yield the actual query string. See the tutorial on how condDesc code works in general.

This has only changed the condDesc used in web forms. To change the condition used in the cone search protocol, you can re-use the match function, though. Finally, you should include the normal DALI parameters, so the total template for an overridden search condition looks like this:

```
<scsCore queriedTable="data">
  <condDesc silent="True">
    <FEED source="//pql#DALIPars"/>
  </condDesc>
  <condDesc original="//scs#humanInput">
    <phraseMaker original="//scs#humanSCSPhrase">
      <setup original="//scs#scsSetup" id="proxQuery">
        <code><![CDATA[
          def genQuery(inPars, outPars):
            return ("scircle(spoint(radians(%%(%s)s), radians(%%(%s)s)),"
              " radians(%%(%s)s)) && origin_est")%(
              base.getSQLKey("RA", inPars["RA"], outPars),
              base.getSQLKey("DEC", inPars["DEC"], outPars),
              base.getSQLKey("SR", inPars["SR"], outPars))
        ]]></code>
      </setup>
    </phraseMaker>
  </condDesc>
  <condDesc original="//scs#protoInput">
    <phraseMaker original="//scs#scsPhrase">
      <setup original="proxQuery"/>
    </phraseMaker>
  </condDesc>
</scsCore>
```

...change a single param of ssap#hcd to being a column?

Sometimes spectral pipelines emit inhomogeneous data even for a single instrument; for instance, "most" spectra could be normalised to the continuum, but for some whatever heuristics the pipeline used didn't work out and the spectra ended up as uncalibrated (as far as the spectral axis is concerned).

The officially correct way to deal with this is to use `//ssap#mixc`. However, moving all the params to table columns is a bit of an overkill when all that's varying is a single column. Hence, we recommend the following workaround in such a case:

(1) Set the param to null (hoping to help clients ignore it); in this case it could be:

```
<table id="data" onDisk="true" adql="True">
  <mixin
    ...
    fluxCalibration="__NULL__"
  >//ssap#hcd</mixin>
```

(you only need to do this when the parameter overridden is required by the mixin).

(2) Add a column with the right utype to the table:

```
<column name="ssa_fluxcalib" type="text"
 utype="ssa:Char.FluxAxis.Calibration"
 tablehead="Calib Flux" verbLevel="20"
 description="Type of flux calibration"/>
```

(3) In the rowmaker, map whatever you need manually:

```
<make table="data">
  <rowmaker id="make_data" idmaps="ssa_*, localKey">
    <apply procDef="//ssap#setMeta" name="setSSAPMeta">
     ...
    </apply>

    <map key="ssa_fluxcalib">@FLUXCAL</map>
```

The disadvantage is that the (uninformative) PARAM with the utype in question will still be in the VOTable, so the using `mixc` would be cleaner. On the other hand, we could add PRUNE to mixins should this really become a bother.

Operations

...get rid of stuff I imported or published?

First, to avoid trouble you should always drop things before renaming them – DaCHS tries hard to not touch things that it does not know about (i.e., things no longer mentioned in RDs).

So, it's best to run:

```
dachs drop res/q
```

while the RD is still there. But of course, you usually only remember to do that when it's hard to recover the previous situation. To save the day then, DaCHS offers the `-f` flag to `drop`:

```
dachs drop -f res/q
```

With `-f`, `dachs drop` doesn't even bother to look at what might have been in the RD. It just visits all tables in which rows record which RD they came from, and when the RD id you give matches the source of the row, it does down the drain.

This is somewhat dangerous when the RD contained published services. If you absolutely have to use `drop -f` on such an RD, please be sure to recreate deleted services after dropping the RD (see `dachs admin makeDeletedRecord` for how to do that).

If you merely see a table that should have been long gone and you can't even remember what RD it came from, there's the `purge` subcommand. In contrast to most other DaCHS commands, it accepts a database table name, and again removes traces for that table where it can find it (in particular in TAP_SCHEMA):

```
dachs purge myschema.tablename.
```

In short: To drop stuff from an RD, use `drop -f`. To drop a table with a qualified name, use `purge`.

By the way, DaCHS will never remove a schema right now. To get rid of empty schemas, use psql ("drop schema...").

...get rid of DaCHS entirely?

If you installed from package, first do:

```
apt-get --purge remove gavodachs-server
```

This will *not* remove tables in the database or your gavo file system hierarchy. These you can get rid of with something like this:

```
rm -r <your-gavo-root-path>
sudo -u postgres dropdb gavo
```

We may be swayed into doing this by users; for now, we are reluctant to have a script blindly initiate a devastation of this scope.

...upgrade DaCHS?

See opguide.html#upgrading.

...update my code?

Code fragments are more or less inevitable when publishing non-trivial data. Unfortunately, sometimes APIs changes, those of DaCHS not excluded. This section tries to collect cases with simple recipes, starting shortly before the 1.0 release.

- 1.0: pyfits header `.update` is now `.set`
- 1.0: pyfits header `.ascardlist` is now `.cards`
- 1.0: pyfits card `.ascard` is now `.ascardimage()`
- 1.0: onDisk tables no longer do automatic connections. So, instead of `t = rsc.TableForDef(onDiskTD)`, you'll have to say:

  ```
  with base.getTableConn() as conn:
    t = rsc.TableForDef(onDiskTD, connection=conn)
  ```

  (use `WritableAdminConn` if you want to write data)
- 1.0: The processing of input parameters has been sanitised. In particular, there is no `InputDD` any more. If you did non-trivial things with it, use a `customCore` now (see apfs/q rev. 5559 for an example for how to port). Also, if you want sequences, you now always have to use multiplicity="multiple" in the input keys. Finally, just use inPars["parName"] throughout in your `condDescs` to retrieve values, and you should be fine.
- 1.0: If you previously used import pyfits, you *might* get away with saying `from astropy import wcs` and replacing every occurrence of pywcs with wcs. On the other hand, the astropy API isn't quite the same any more, so be sure to test things.

...upgrade the database engine?

Postgres' on-disk formats change from version to version. This makes upgrages a bit of an adventure, in particular if you have large databases that take days to export and ingest. However, Debian helps a bit; even if you don't run Debian, the following might be helpful.

The text assumes you're using the default cluster, which is called main. If – and that's likely if you installed DaCHS in the olden days – yours has a different name (we used to suggest "pgdata"), you'll have to replace "main" with that name throughout.

As it's easy to destroy all of your tables at once if you mistype things here, it's highly recommended to dump your database somewhere before starting with this:

```
pg_dump -f (some descriptive name).pgdump -F c gavo
```

Let's say you're upgrading from 9.1 to 9.4. Get a root shell and set these values. Be *very* careful, if you get this wrong, the commands further down will nuke your database:

```
export OLDVER=9.1
export NEWVER=9.4
```

To upgrade, first install the new engine and extensions, for instance:

```
apt-get install postgresql-${NEWVER}-pgsphere postgresql-${NEWVER}-q3c postgresql-${NEWVER}
```

If you use other extensions in your database (e.g., postgis), also install them now. To see what extensions you have installed, a command like:

```
dpkg-query -W "postgresql-${OLDVER}-*"
```

is probably helpful. Other versioned packages you may want to install include:

- `postgresql-server-dev-$NEWVER` if you use locally-built extensions. Rebuild and install those extensions *before* proceeding with the upgrade.
- `postgresql-plpython-$NEWVER` if you use python DB extensions
- `postgresql-contrib-$NEWVER` for several interesting index schemes and the like.

You will probably first have to drop the new cluster the Debian maintainer scripts may already have created, as pg_upgradecluster wants a clean slate:

```
pg_ctlcluster $NEWVER main stop
# this is a good time to see if your data center still works; if not,
# you stopped the currently active cluster and should *not* drop it
pg_dropcluster $NEWVER main
```

If these give errors to the effect that the "specified cluster does not exist", that's fine.

During the database update, postgres will be down, and hence your data center will not work. For a large database, the update can take several hours. You should let people know that there's something wrong as good as you can. As long as nobody really does anything with VOSI availability, the next best thing is to put your DaCHS installation in panic mode.

To do this, place a file called MAINT into DaCHS's stateDir, maybe somewhat like this:

```
cat <<EOF > `dachs config stateDir`/MAINT
We're currently upgrading our database and cannot properly respond to
requests.

We expect to be back online late afternoon UTC, 2015-08-11.
EOF
```

Then say:

```
pg_upgradecluster -k -m upgrade -v $NEWVER $OLDVER main /var/lib/postgresql/$NEWVER
```

You may need to change the last argument, depending on where your database files actually are. You can figure out where the old ones were using:

```
grep data_directory /etc/postgresql/$OLDVER/main/postgresql.conf
```

Do *not* use that old directory for the new cluster.

Note the `-k` option to the above command: this makes pg_upgrade use hard links instead of copies. This saves a lot of space and time. The downside is that once the upgrade has succeeded, the old server may go haywire because its files have been changed to fit the new version. In our experience, that's a price worth paying.

If this fails, **do not panic**. Most of the time it's some extension you forgot to install, and once you've done this, everything will be fine. The upgrade script points you to the logs of the operation, which should let you figure out what to do.

On jessie and before, you'll get unhelpful error messages ("cannot determine cluster locales") unless you restart the old cluster manually (`service postgresql restart` should work) before re-starting pg_upgradecluster.

When all is done, pg_upgradecluster configures the old cluster to no longer start automatically and the new cluster to listen where the old one was.

You can now probably run your services again, so, remove MAINT in the stateDir, restart the DaCHS server and run your regression tests (you wrote some, didn't you?). However, quite typically the table stats are wrong or ununderstandable to the new version. It is therefore highly recommended to run:

```
psql gavo -c "VACUUM ANALYZE"
```

(which can run in parallel to normal operations; you could use the opportunity of a downtime to run a VACUUM FULL ANALYZE, too, but that may take ages).

If all looks well, drop the old cluster. pg_upgradecluster prints the respective command last thing it does. And deinstall the old binaries to avoid confusion.

Possible Issues during upgrade

Frankly, there's lots. We'll update this section based your feedback.

(1) Some versions of pgsphere had bad declarations of negators and commutators. These manifest themselves as errors like:

```
pg_restore: [archiver (db)] could not execute query: ERROR:  argument of negator must be a name
    Command was: CREATE OPERATOR @ (
    PROCEDURE = "sline_contains_point_com",
    LEFTARG = "spoint",
    RIGHTARG = "sline",
    COMMUTAT...
```

during upgrades. If you see something like this, you need to remove dangling references from the the `pg_operator` system catalog before commencing. So, in the *old* cluster, run:

```
UPDATE pg_operator AS o
SET oprcom=0
WHERE
  NOT EXISTS (
    SELECT 1 FROM pg_operator AS i
      WHERE i.oid=o.oprcom)
  AND oprcom!=0
```

and:

```
UPDATE pg_operator AS o
SET oprnegate=0
WHERE
  NOT EXISTS (
    SELECT 1 FROM pg_operator AS i
      WHERE i.oid=o.oprnegate)
  AND oprnegate!=0
```

(2) Remove versioned tablespaces after failures. pg_upgrade in principle does the right thing with tablespaces: It has per-version names for the actual data directories. However, when an upgrade has failed, that extra data directory is not removed. Another attempt at upgrading then usually fails because that directory is already there. For instance, you'll see in the tablespace /media/db_overflow:

```
$ ls
PG_9.4_201409291/  PG_9.6_201608131/
```

In that case, you'll need to remove the 9.6 directory before proceeding.

...switch between SVN and packaged DaCHS?

Sometimes you need to be at the bleeding edge with your DaCHS and run from SVN for a while. This is even the recommended way to do an SVN installation, as dependency installation is taken care of for you.

To go from package to SVN, do something like:

```
$ sudo apt-get remove python-gavodachs gavodachs-server
$ svn co http://svn.ari.uni-heidelberg.de/svn/gavo/python/trunk/ dachs
$ cd dachs
$ sudo python setup.py develop
```

Note, however, that all dependencies of DaCHS that are not used by anything else will be removed on, e.g., apt-get autoremove or by aptitude, so you may want to mark them manually installed, perhaps even before package removal (this is fairly convenient in aptitude).

To go back to package, you will need to remove the path entry that tells python to look in that local directory for `gavo` packages. The concrete location of that file is a bit variable. Use, for instance, locate (mlocate package) for find the file gavodachs.egg-link; when you have the path, do:

```
$ sudo rm /usr/local/lib/python2.7/dist-packages/gavodachs.egg-link
$ sudo apt-get install python-gavodachs gavodachs-server
```

...run DaCHS behind a proxy?

The recommended way to run DaCHS when it has to share a public IP with other services is to use a dedicated virtual server as a reverse proxy. Here's how to do this when your main server is an apache.

Suppose your server's primary name is www.obs.example.org. Then, add a DNS entry for vo.obs.example.org; depending on taste, make it a CNAME of www.obs.example.org or have it point to its IP directly.

To create a virtual server that will serve as reverse proxy, adapt the following snippet and drop it into `/etc/apache/sites-available/voproxy`:

```
<VirtualHost *:80>
  ServerName vo.obs.example.org
  ProxyRequests off
  <Location />
    ProxyPass http://localhost:8080/
  </Location>
</VirtualHost>
```

This assumes you're using a Debian-derived apache package and that DaCHS is running on the same machine as the apache; it shouldn't be hard to adapt the example to other situations.

Still assuming you're on a Debian-derived apache, run:

```
$ sudo a2ensite voproxy
$ sudo service apache reload
```

For many purposes, DaCHS has to know which external Name it is visible under. You need to configure that in `/etc/gavo.rc`, [web]serverURL. In the example, you'd write:

```
[web]
serverURL: http://vo.obs.example.org
```

This is enough when DaCHS runs on the same machine as the reverse proxy.

When the machines are different (which includes DaCHS running within a Docker), note that DaCHS only binds to the loopback interface. Thus, in such cases you'd have to write:

```
[web]
bindAddress:
serverURL: http://vo.obs.example.org
```

to make it bind to all interfaces the machine has (in particular the one that the reverse proxy can see).

...change the host name of my server?

If your services are not registered, all you need to fix is the [web]serverURL setting in your `/etc/gavo.rc`.

If (as they should) your services are registered, you have to re-publish them, as the URLs are baked into the registry records. To do that, run:

```
dachs pub -a
```

However, changing the host name will also change the access URL of your publishing registry endpoint. If that is what you use to distribute the resource records (i.e., if you're registered with the Registry of Registries), you must therefore inform them of the change, and there is no automated way to do that yet. For now, just send a mail to ivoa-rofr (at) cfa.harvard.edu saying something like:

```
Dear RofR maintainer,

We've had to change the host name of our data centre.  Hence, the
publishing registry for the authority ivo://my.authority is no
longer at http://old.domain.name/oai.xml but instead at
http://new.domain.name/oai.xml.

Would you change the RofR files accordingly?

Thanks,

        X
```

If instead you're using purx to get your records out, send mail to gavo@ari.uni-heidelberg; there's no automatic way to change a purx-harvested URL right now, either.

If you're using one of the browser-based publishing registries (at ESAC or STScI), you'll have to re-upload your registry records there.