How Do I?

Recipes and tricks for solving problems using GAVO DaCHS

Author: Markus Demleitner
Email: gavo@ari.uni-heidelberg.de
Date: 2024-02-14

Contents

In case you are missing something from here that you remember was here before: Some items were moved to tutorial.html.

We will try to cover everything that is more or less routine there, reserving this document to less common tasks and material that will be of historical value only rather quickly (such as instructions for particular upgrade steps).

Ingestion

...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. Realistically: DaCHS depends on astropy, and so just use whatever astropy gives you.

On the other hand, there's an interface in the gavo.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
```

...map NaNs to NULLs?

(which is, by the way, something you should do in case your upstream gives you NaNs – databases to a lot more sensible things with NULLs than with NaNs in general).

I've come to use a pattern like:

```
<LOOP listItems="mag_dered_u mag_dered_g mag_dered_r mag_dered_i
    mag_dered_z zspec zphot odds_phot">
  <events>
    <map key="\item">None if math.isnan(@\item) else @\item</map>
  </events>
</LOOP>
```

in rowmakers for tables that have a few columns involving NaNs. Bug me and I'll add some functionality to fitsTableGrammars (where this kind of thing is most common) that does this internally.

...configure my Indexes?

Suppose you want to have all (or the majority of) your indexes on a fast disk. The way to do that is to create a Postgres tablespace as usual (DaCHS does not know about them) and then add:

```
<option>TABLESPACE fasttb</option>
```

to the `index` elements in question (added in DaCHS 2.5.4). However, when you do that declaration in many indexes, moving the tablespace will be very hard, as you would then have to edit many files.

To make this somewhat more declarative, create a userconfig RD and in there – probably near the end –, add an element like:

```
<STREAM id="fastindex">
  <option>TABLESPACE fasttb</option>
</STREAM>
```

With this, an index on the fast disk would be declared by:

```
<index columns="gaia_id">
  <FEED source="%#fastindex"/>
</index>
```

In this way, operators could give some default index configurations. For instance, DaCHS tables often are static, i.e., populated once and then left alone. In that case, you may save 10% storage without any ill effect if you declare the index's fill factor as 100. You could create a second userconfig stream for that; it is probably more reasonable, though, to combine something like a sane site default into:

```
<STREAM id="fast-static-index">
  <option>TABLESPACE fasttb</option>
  <option>WITH (fillfactor=100)</option>
</STREAM>
```

Note that this will *not* influence indexes made through mixins and perhaps STREAMs. We believe that for most of them, that is reasonable, as they are typically created on rather modest tables.

This is different for pgs-pos-index and q3cindex. These can be challenging indexes, and if you would like to move and configure them, please contact us with your use case. Meanwhile, you can of course simply look at the definitions (`dachs adm dumpDF //scs` is your friend) and adapt them within your tables rather than use the mixin.

Services and RDs 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.
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. You can, however, protect the entire TAP service (with HTTP basic auth, so this isn't for the gold of Fort Knox unless you restrict connections to HTTPS and change to hashed passwords in dc.users).
First, create a group of authorized users and add a user to it:

```
dachs admin adduser tapusers master_password
dachs admin adduser smart_pi their_password
dachs admin addtogroup smart_pi tapusers
```

Then, limit access to the TAP service itself; as usual (see tutorial.html#restricting-access), you will have to add a `limitTo` clause to the service element. However, this service element is built-in, so you cannot easily change it.
For TAP, there's a trick because the service element includes the tapdescription STREAM from the userconfig RD. So, get userconfig unless you already have it (cf. tutorial.html#the-userconfig-rd) and simply add:

```
<limitTo>tapusers</limitTo>
```

after the `<STREAM id="tapdescription">`. To make DaCHS see the change, you need to restart the server or run `dachs serve exp //tap`.

That's not quite enough, though, because there is the public form-based ADQL service. For that, you can override the `//adql` built-in RD as per tutorial.html#overridden-system-rds – but as pointed out there, if you do that, you need to manually track changes we do to the ADQL RD, so if that's something you want to do permanently, shout and we'll think of something easier in maintenance. Having said that, what you'd do is something like:

```
mkdir -p /var/gavo/inputs/__system__
cd /var/gavo/inputs/__system__
dachs adm dumpDF //adql > adql.rd
```

and then change that new adql.rd file such that:

```
<service id="query" core="qcore">
```

becomes, say:

```
<service id="query" core="qcore" limitTo="tapusers">
```

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>
```

...pre-set values for buildFrom condDescs?

The normal way to build service parameters in DaCHS is to say:

```
<condDesc buildFrom="some_column"/>
```

in a dbCore. This will automatically understand VizieR-like expressions in web forms, DALI intervals in modern protocols, "PQL" in SSAP, and so on. This is trouble when you want to set a default for this parameter. For instance, to say "between 2 and 5", the default would have to be "2 .. 5" with VizieR, "2 5" in DALI and "2/5" in PQL. Yes, it *is* a mad world.

DaCHS does not yet have a way to do such pre-sets in a sane way. Where the `defaultForForm` property (see Element inputKey) is not enough, you will have to use code in a phrase maker and in it dispatch on the type DaCHS has chosen when adapting the condDesc to the renderer. The following example might help you writing such a thing:

```
<condDesc inputOptional="True" buildFrom="some_column">
  <phraseMaker>
    <code>
      if inPars[inputKeys[0].name] is None:
        # None means: no input
        if inputKeys[0].xtype=="interval":
          # That's DALI (e.g., SCS): an interval is a parsed 2-list
          inPars[inputKeys[0].name] = [2, 5]

        elif inputKeys[0].xtype=="vexpr-float":
          # That's VizieR on a float-valued column, which just hands
          # through the input trings to getSQLForField
          inPars[inputKeys[0].name] = "2 .. 5"

      yield base.getSQLForField(inputKeys[0], inPars, outPars)
    </code>
  </phraseMaker>
</condDesc>
```

If you find yourself resorting to horrors like that more often, let us know and we will think a bit harder about whether there really is not a better way to do that.
(starting version 2.8)

...pre-set a value for a datalink parameter?

Sometimes you would like to pre-fill a parameter that you generate in a datalink metaMaker.
Regrettably, as of Datalink 1.0, this is not generally possible, because there is no way to interoperably communicate such an editable default to a client (if you set the InputKey's value, it will become a constant input that the user can no longer edit).
However, in DaCHS and with DaCHS' datalink stylesheet, you can put in a default on most widgets by setting the `defaultForForm` property on the InputKey, for instance:

```
<metaMaker>
    <code>
    ik = MS(InputKey, name="format", type="text",
      ucd="meta.format",
      description="Output format desired",
      values=MS(Values,
        options=[MS(Option, content_=descriptor.mime),
          MS(Option, content_="application/fits")]))
    ik.setProperty("defaultForForm", "application/fits")
    yield ik
    </code>
</metaMaker>
```

By a custom (but non-interfering) extension to Datalink, this produces:

```
<LINK action="rdf" content-role="#pre-set" value="application/fits"/>
```

as a child to the VOTable PARAM produced. DaCHS' own datalink stylesheet picks this up and pre-fills the HTML widget it spits out. It is virtually certain that that will not be what a future standard will say about pre-setting parameters; but I can almost promise that whatever the standard will say, DaCHS will keep the `setProperty` technique, so your RD still ought to work once a proper standard is there.
This may not work for all widgets that the style sheet produces (for instance, intervals are only covered starting version 2.7.1); if it doesn't work for you, complain on dachs-support.

...run code when and RD is loaded?

There are no *onload* scripts in DaCHS. If you really need to do some magic when an RD is initialised (and you should probably ask on dachs-support if you think you need to do that), you can abuse the Element LOOP with `codeItems`, perhaps like this:

```
<LOOP>
  <!-- initialise the foobar module at RD load time.  This will not
  generate any RD elements. -->
  <codeItems>
    import foobar
    foobar.initialise()
    if False:
      yield
  </codeItems>
  <events/>
</LOOP>
```

...attach a param to a column?

Sometimes it is useful to associate a param with a column; the classic case is when you have flux values in columns and want to push out the spectral points in a single VOTable PARAM to save space (provided they're all the same).
To do that, (ab)use the dm element. You can freely invent “data models“ for that purpose until there are some that work for your case and clients actually interpret use dm annotation; to prevent later clashes with prefixes claimed by IVOA model, use some private tag on the prefix (`gavo-` in the example below). In our example, you could write in the table definition, with made-up class and attribute names:

```
<table id="spectra">
  <dm>
    (gavo-spec:spectrum) {
      spectralAxis: @spectral
      fluxAxis: @flux
    }
  </dm>
  <param name="spectral" type="real[5]" unit="J"
    >1e-19 2e-19 3e-19 4e-19 5e-19</param>
  <column name="flux" type="real[5]" unit="Jy"/>
</table>
```

Incidentally: Yes, you need the explicit length here if you want TOPCAT to notice that spectral and flux belong together (again, until there's proper DM annotation).

Form-based interfaces

...add a field with the distance from the search center in a image service?

Sometimes researchers want a column with the distance from the search center in image services (very typically through a form service). In contrast to SCS, where we artificially add such a column, DaCHS doesn't add such a column by default, mainly because we're unconvinced that that is a good idea.
But you can retrofit such a column. The main challenge is that the search area can be specified in varied ways (or not at all, in particular when going through a web interface), and so you need a bit of logic (in the SIAP case) or perhaps even a lot of it.

You could use a custom core for that, but it turns out there is a less intrusive way to do it: Stuff the logic you need to pull parameters out of the user input into values accessible to database queries into a silent `condDesc`, and then add your distance column to the *core*'s output table (rather than the service's, which you can still vary). Using the core has the advantage that people can then sort by `_r`.
A live example for the technique is in kapteyn/q. Specifically, have a look at the `web` service's dbCore. There, we override the outputTable with to include all of the queried table (which is what dbCore normally produces), plus our `_r` column:

```
<outputTable original="plates">
  <outputField name="_r"
    unit="deg" ucd="pos.distance"
    tablehead="Dist."
    description="Distance to the search center"
    select="degrees(spoint(radians(centerAlpha), radians(centerDelta))
      &lt;-> %(X_search_center)s)"
    displayHint="displayUnit=arcmin,sf=0"/>
</outputTable>
```

The `select` term for the `_r` field refers to a key `X_search_center`; this then needs to reflect the remote input and hence has to get into the `outPars` dictionary somehow (cf. Element phraseMaker to learn about these dicts) somehow. Filling `outPars` is the job of `condDesc`-s – even though in this case we do not want to produce any constraints.
In this particular case the additional trouble is that parsing the `hPOS` parameter is non-trivial, and so we'd like to piggyback on what's already parsed by `//siap#humanInput`. Here's a secret: that `condDesc` leave keys `_ra` and `_dec` in `outPars` (and that won't change because it's used in the cutout core) if there's a positional constraint in the request.
Hence, you can have the following `condDesc` *after* the `//siap#humanInput` one:

```
<condDesc combining="True">
  <phraseMaker>
    <code>
      if "_ra" in outPars:
        outPars["X_search_center"] = pgsphere.SPoint.fromDegrees(
          outPars["_ra"], outPars["_dec"])
      else:
        outPars["X_search_center"] = None
      if False:
        yield
    </code>
  </phraseMaker>
</condDesc>
```

Things you might wonder about: The `combining="True"` simply makes sure the `condDesc` will be considered even if no input key for it is given (you could give it the hPOS input key and perhaps save an `if`, but it's more robust this way).
The `if "_ra" in outPars` wards against missing positional constraints; without an input position, the best we can do is have NULLs in `_r`, which is what the select expression above will do when one argument is NULL.
Finally, the odd `if False: yield` is to tell Python that this is a generator, although it doesn't generate anything (usually, `phraseMakers` will yield query fragments, which we don't do here).

Finally, in kapteyn/q, we have an output table in the service, so our new column coming from the core would be filtered out by the service. To have it in the final output table, in this case we've added an `<outputField original="_r"/>` (but that, of course, can be quite different for your case).

...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 use
    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. All you need to do is set the new display hint.

See the how to ...override metadata for a column coming from a mixin? for the general idea behind this; what it boils down to in this case is:

```
<table id="data" onDisk="True" mixin="//products#table">
  <column original="accref"
    displayHint="type=product,nopreview=True"/>
  ...
```

Having said this: People love previews, so just try to make some; see Product Previews for how DaCHS helps you in that.

...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.

...get back the mouseover previews we had in DaCHS 1?

In DaCHS 1, when you had a product URI, the machinery would pull the previews only when the user moused over the link. We changed this in DaCHS 2 to pulling the preview automatically when a row becomes visible, because many operators had asked for that.
If you would like to have the previous behaviour back, you have to override the formatter of the field containing the product URL, which in most cases is accref. As usual, you override a column by overwriting it, in this case in the service's output table (if it doesn't have one yet, use `<outputTable autoCols="*"/>` as a start).
The actual formatter needs a bit of care, as you will be returning HTML, and that obviously must not bleed into, say, VOTable or CSV outputs. Hence, you need a format switch.
Also, the accrefs DaCHS stores internally are keys into the product table. To obtain a URL from that, you have to go through `<serverurl>/getproduct/<productkey>`.
That's almost all you need to know to write a formatter in which you can attach essentially arbitrary javascript (you could define extra functions in a custom template). To get back DaCHS 1 behaviour, the following should do:

```
<column original="accref">
  <formatter>
    fullURL = base.makeAbsoluteURL("/getproduct/"+data)
    if queryMeta["format"]=="HTML":
      return T.a(href=fullURL, onmouseover="insertPreview(this)")[
        fullURL.split("/")[-1]]
    else:
      return fullURL
  </formatter>
</column>
```

You can observe this in its natural habitat in apo/res/apo.

...customise the ADQL query form?

DaCHS comes with a basic form that lets people without a proper TAP client enter ADQL queries at `/__system__/adql/query`. This is rather basic at this point, and if you'd like to add a bit more glitz for your browser-only users, it may be worth doing some customisation here. The built-in service cannot really be customised by users (at least for until we add some hooks), except by overriding the ADQL RD (cf. tutorial.html#overridden-system-rds; this is really not recommended).

You can, however, create a custom ADQL-entering service yourself by putting an RD like the following into, say, `/var/gavo/inputs/adqlform/q`:

```
<resource schema="dc" resdir=".">

  <meta name="_intro" format="rst"><![CDATA[
    From this form, you can run ADQL queries on my data center in case
    you don't have TOPCAT or some other TAP client at hand.  Let me
    particularly recommend running queries on the **mauve.results** and
    **violet.images** tables.
]]>
  </meta>
  <meta name="_bottominfo" format="rst">
    There is a fixed limit to 100000 rows on this service.  If this
    bugs you, use \RSTservicelink{tap}{TAP} (you should anyway).

    Oh, and here's a few queries you could try:

    * Trivial

      ::

        SELECT foo FROM bar

    * Less trivial

      ::

        SELECT bar FROM foo
  </meta>

  <service id="q">
    <adqlCore>
      <inputTable id="adqlInput">
        <inputKey name="query" tablehead="ADQL query" type="text"
          description="A query in the Astronomical Data Query Language"
          widgetFactory="widgetFactory(ScalingTextArea, rows=6)"
          required="True"/>
        <inputKey name="_TIMEOUT" type="integer" unit="s"
          tablehead="Timeout after"
          description="Seconds until the query is aborted.  If you find
            yourself having to raise this beyond 200 or so, please contact
            the site operators for hints on how to optimize your query">
          <values default="5"/>
        </inputKey>
      </inputTable>
    </adqlCore>
    <meta name="shortName">myadql</meta>
    <meta name="title">My Local ADQL Query</meta>
    <publish render="form" sets="local"/>
  </service>

</resource>
```

That way, you'll get a customised form at `http://localhost:8080/adqlform/q/q`; if you `dachs pub` that RD, it will also appear on your root page. Come to think of it, you could even put:

```
[web]
root: adqlform/q/q/form
```

into your `/etc/gavo.rc` and have that as your root page.

If this still does not provide sufficient flexibility for your use case, look into overriding the `form` template (probably based on what you get back from `dachs adm dumpDF templates/defaultresponse.html`). See templating.html for details.

...have math in my descriptions?

If you have non-trivial math in your `description`, `_longdoc`, or `note` meta items (or, indeed, elsewhere in your meta elements), use the `rst` format. ReStructuredText has an interpreted text role and a diretive both called `math`. With these, you can write things like:

```
Newton figured out that :math:‘F=ma‘, and that specifically for
gavitation:

.. math::

  F = G \frac{M \cdot m}{r^2}
```

Much of LaTeX math is supported in these constructs.

DaCHS tells the ReStructuredText machinery (docutils) to produce HTML from this. This sometimes is good enough in common browsers. Most of the time, however, you will want to include some extra CSS to improve the appearance of the formalae. DaCHS has that on board; the challenge is to smuggle this CSS into the formatted documents, because as you do not know where your meta items will be rendered, and hence the `extraCSS` property of any single service will usually not work.

The current workaround is to include the following material in your meta item:

```
.. role:: raw-html(raw)
  :format: html

:raw-html:‘<style type="text/css">@import url(/static/css/math.css);</style>‘
```

Yeah, that's not pretty, and we'll try to think of something better.

...query against Array-valued columns?

First, if you have arrays in your tables' columns, that's a de-normalisation: in the pure relational creed, values in columns are atomic. Many interesting things become a lot simpler when you unfold these arrays to another table – for ingestion, you would look into dispatching grammars then.
But there are many cases when it actually makes sense to have arrays for convenience, speed, or simplicity. Consider a list of target objects, declared like this:

```
<column name="objects" type="char(15)[]"
   ucd="meta.id;src"
   tablehead="Objs."
   description="Names objects from the observation log."/>
```

– sorry, VOTable can't do arrays of variable-length strings and hence DaCHS refuses that, too; so, you can't say `text[]` – and filled, perhaps, like this:

```
<map key="objects">@mapped_names.split("|")</map>
```

The trouble starts when you try to write a constraint against this; the desired behaviour would be "let people choose from a list of array elements and match when they give one or more of what's in a row's array". This is a bit subtle, for one because you can't rely on DaCHS' column statistics (which can't deal with arrays) and need to do a custom collection, and then because you can't compare your input to the column for equality.
Here's a condDesc that fits the bill:

```
<condDesc>
  <inputKey name="object" type="text" multiplicity="multiple"
      tablehead="Target Object"
      description="Object being observed, Simbad-resolvable form"
      ucd="meta.name">
      <values fromdb="unnest(objects) FROM fai50mak.main"/>
  </inputKey>
  <phraseMaker>
    <setup imports="numpy"/>
    <code><![CDATA[
      yield "%({})s && objects".format(
        base.getSQLKey("object",
        numpy.array(inPars["object"]), outPars))
    ]]></code>
  </phraseMaker>
</condDesc>
```

Note the `numpy.array` – this is necessary here because normal Python lists get converted to tuples rather than arrays by DaCHS.
Note that this does not support indexes, and the list of objects will be collected on each RD load. This means that as your tables grow beyond, perhaps, 10'000 rows or so, this will become impractible and you should go for a proper relational solution.

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(td, 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 original="//scs#humanInput">
    <phraseMaker original="//scs#humanSCSPhrase">
      <setup original="//scs#scsSetup" id="proxQuery">
        <code><![CDATA[
          def genQuery(td, 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>
```

Incidentally, the first argument to genQuery is the table definition of the table queried, which sometimes comes in handy. DaCHS 1 didn't have that argument, so old RDs may need an update if they used this recipe.

...figure out why an obscore query is slow?

In DaCHS, ivoa.obscore is a view over potentially many underlying tables. Some of these tables may be large, and then when an obscore query comes in and a single large table can't use an index, the whole obscore query is slow. The trouble is: You don't know which table is the culprit.
To figure that out, write a minimal slow query and write an `EXPLAIN ANALYZE` in front of it; this will make postgres report actual execution times in a query plan. However, since the view typically is rather large, so is the query plan. Hence, save it to a file for closer inspection:

```
psql gavo -c "EXPLAIN ANALYZE SELECT obs_id \
  FROM ivoa.obscore WHERE q3c_join(s_ra, s_dec, 10, 10, 0.001)" \
  > plan.txt
```

(this is for q3c-based positional searches) or:

```
psql gavo -c "EXPLAIN ANALYZE SELECT obs_id \
  FROM ivoa.obscore WHERE spoint(0.1, 0.1) <@ s_region" > plan.txt
```

(for pgsphere-based positional searches) – other conditions that may be slow translate essentially 1:1 to SQL; you can also pull translated queries from /var/gavo/logs/dcInfos.
Inspect the resulting file and watch out for "Seq Scan" – if these happen on large tables, things will we slow. For small tables (a couple of thousand records, perhaps), on the other hand, it doesn't really matter.
You'll know where your query time went by looking at the "actual time" part of these lines; the numbers are in microseconds, so you don't need to worry about anything if there's something like 100 there. Hunt down the ones having a few thousand, and then strategically add indexes.

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
```

This probably will not become of the purge routine of the Debian package; we are reluctant to have a script blindly initiate a devastation of this scope.

...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.
- 2.0: This is a move to python3 and is a longer thing. See how to ...go from DaCHS 1 to DaCHS 2?
- 2.3: This updates jquery; if you've used a recent Debian package, you already used that. If you have javascript in any templates, that may needs updates, among others:
  - change `.unload(` to `.on("unload",` (this happens in some SAMP code, for instance).
  - element names are now returned in lower case, so add a toUpperCase when you've compared them.
  - in the SAMP code, change the icon URL to `completeURL("/logo_tiny.png")` (or whatever) to avoid trouble with https-only installations.
- 2.3: the .query methods on table or querier objects are now strongly deprecated, and they'll disappear because their semantics sucked. Use .query (for queries returning rows) or .execute (for everything else) on the connection attributes of table or querier instead. Note that those won't expand macros; use table.expand(query) if you need that.
- 2.4: In form handlers, you saw empty uploads as (None, None) so far; they're simply None now.

...switch between git and packaged DaCHS?

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

```
$ sudo apt remove python-gavodachs gavodachs-server
$ git clone https://gitlab-p4n.aip.de/gavo/dachs.git dachs
$ cd dachs
$ sudo python3 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/python*/dist-packages/gavodachs.egg-link
$ sudo apt 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 a2enmod proxy
$ sudo a2enmod proxy_http
$ 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 ALL
```

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.

...do IPv6?

If all you want is to listen on your IPv6 interface, you can give is address literal (e.g., `::1`) in [web]bindAddress. Use `::` to listen on all configured addresses. I've not dug into this any deeper, but it *seems* this will also cover IPv4 by some magic.
Just using an empty `bindAddress` will not automatically listen on IPv6 interfaces in 2021 twisted and linux.

...go from DaCHS 1 to DaCHS 2?

Starting mid-2020, DaCHS operators ought to upgrade to DaCHS 2; don't be scared by the major version jump – unless you're doing higher magic, you will not need to change much (if anything) in your RDs. See also https://blog.g-vo.org/dachs-2-1-say-hello-to-python-3/.
We have dropped some exotic legacy, though, and in particular DaCHS 2 is Python 3 only (whereas DaCHS 1 in Python 2 only). It's quite likely that if you need to change your RDs at all, it'll be about Python 3 rather than DaCHS 2 (see below).

Upgrading the Software

Important: DaCHS wants at least python 3.7; in Debian terms, that's buster and above.
If you run from package, upgrading should be as simple as adding the beta repository and then running:

```
apt install gavodachs2-server
```

The two packages conflict, and thus this will remove the old dachs package(s).

If you use some other means to install DaCHS, please note the the dependencies have changed; in Debian terms, we're now requiring python3-astropy, python3-setuptools, python3-all, python3-docutils, python3-twisted, python3-psycopg2, python3-lxml, python3-pkg-resources, python3-matplotlib, and it'd be great if you had (some less important parts of DaCHS won't work if you don't have them) python3-healpy, python3-pil, python3-testresources[1] .

Also, if you install via setup.py, note that you must run setup.py with python3, not with python (which is Python 2 on all systems I've seen so far).

Then restart your server and run:

```
dachs test ALL
```

If you have defined regression tests, this should show you troublesome spots. If you don't have regression tests, gnash your teeth and perhaps see if `dachs val` points you to problems.

Updates to RDs and Similar

Most code within RDs should be ok. An important exception may be your userconfig RD (cf. tutorial.html#the-userconfig-rd). In DaCHS 1, the template for a while had:

```
<script id="_test-script" lang="python" name="test instrumentation"
        type="preIndex">
        # (this space left blank intentionally)
</script>
```

which is no longer allowed in DaCHS 2.2. This will, on every operation, give you an error "Invalid script type preIndex for resource elements". Just remove the element in /var/gavo/etc/userconfig.rd.

Of course, you will have port any python code in the vicinity of the RDs to Python 3. The following points turned up while porting GAVO DC RDs:

- strings vs. bytes throughout – you can't sensibly compare them or mix them, and you can only write strings to encoding-aware files and bytes to binary files, so every open is suspicous. Our advice from Python 2 days has been: Whenever any text comes in, turn it into strings as soon as possible, and encode just before writing (that of course doesn't apply when you're actually dealing with binary data).

- encoding-aware files (this was actually our number one problem not automatically caught by 2to3) – when you just `open("name")`, you get back a file that returns true strings and hence supposes an encoding. That may actually do what you expect by accident, but mostly it doesn't. You'll have to decide in each case whether you want bytes (in which case you `open("name", "rb")` or strings (in which case you should specify the encoding explicitly – `open("name", "r", enc="utf-8")` to avoid run-time surprises).

- `urllib.quote` is now `urllib.parse.quote` (the latter is in the rowmaker namespace)

- int/int now is a float. Well, DaCHS should always have future-imported division, but alas it hasn't. Whenever you want integer division, write //.

- `dict.has_key(k)` no longer exists. Write `k in dict` instead.

- `except Ex, x:` is verboten. Use `except Ex as x:` instead.
- Well, and then there's the thing with print no longer being a statement. But then you probably don't have a lot of print in your RD code.

If you use text-base grammars (columnGrammar, reGrammar, ...) and your inputs contain non-ASCII chars, you could previously get away with it under some circumstances. You will not any longer (but since that only breaks on import, I figured it's not so bad). To fix things, add `enc="e"` attributes to the grammar elements (where `e` will typically be one of iso-8859-1 or utf-8).

That's about it as long as you didn't venture into the land of `service/@customPage`. If you don't know it: it lets you include code that directly uses the web framework. DaCHS was previously built on newow, which has not been ported to Python 3, and, having tried it, I don't expect it ever will. Instead, DaCHS is now built directly on twisted web, but since several aspects of nevow were part of DaCHS' UI even for non-adventurous users, we've created a thin emulation layer that lets you get away with your old templates, render functions, and such. If you don't, shout and we'll probably fix it.

Well, except for customPage and the custom renderer. If you went there, you'll have some nasty work ahead of you, as our emulation layer doesn't reach there. You will, at the very least, now have to inherit from gavo.nevowc.TemplatedPage (rather than rend.Page), and renderHTTP(self, ctx) now needs to be render(self, request), with all usage of the context somehow replaced. Similarly, render and data functions need to accept a request rather than a context. There is a very rough porting guide in gavo.formal.README. Let us know if you need it, and we'll try to improve it.

[1] If you can get your hands on python3-soappy and python3-zsi *and* you'd like to run SOAP services (which I doubt), you can install these as well. They're not currently in Debian, though.