# GAVO DaCHS Tutorial

**Author**: Markus Demleitner
**Email**: gavo@ari.uni-heidelberg.de
**Date**: 2016-03-16

## Contents

This tutorial intends to guide you through ingestion of data and setting up of services. Even if you plan to only publish images or spectra, you should work through the first part; it explains a lot about DaCHS' central concept, the resource descriptor (RD), recommended directory layouts, the basics of metadata and services, and debugging.

## Invoking DaCHS

For historical reasons, all DaCHS functionality is invoked through a program called `gavo`. Multiple functions are integrated and selected through the first argument; run `gavo help` to see what's available; realistically, the functions most operators will be confronted with are `import`, `serve`, `publish`, and `test`.

There is a man page on `gavo`, but it's less well maintained then we would like, so the best way to discover subcommands and options available is to use the built-in help, as in:

```
$ gavo --help
Usage: gavo {<global option>} <func> {<func option>} {<func argument>}
<func> is a unique prefix into {admin, adql, config, dlrun, drop,
  gendoc, import, info, init, limits, mkboost, mkrd, publish, purge,
  raise, serve, show, stc, taprun, test, totesturl, upgrade, uwsrun,
  validate}

Try gavo <func> --help for function-specific help

Options:
  -h, --help            show this help message and exit
  --traceback           print a traceback on all errors.
  --hints               if there are hints on an error, display them
  --enable-pdb          run pdb on all errors.
  --disable-spew        Ignored.
  --profile-to=PROFILEPATH
                        enable profiling and write a profile to PROFILEPATH
  --suppress-log        Do not log exceptions and such to the gavo-specific
                        log files
  --debug               Produce debug info as appropirate.
  --version             Write software version to stdout and exit
  -U UI, --ui=UI        use UI to show what is going on; try --ui=help to see
                        available interfaces.
```

Most of DaCHS's global options have to do with debugging; it is sometimes useful go say:

```
gavo --ui stingy ...
```

do reduce the program's chattiness.

For a brief overview what the individual functions are, see the man page or use the built-in help, as in:

```
$ gavo limits --help
usage: gavo limits [-h] itemId

Updates existing values min/max items in a referenced table or RD.

positional arguments:
  itemId      Cross-RD reference of a table or RD to update, as in ds/q or
              ds/q#mytable; only RDs in inputsDir can be updated.

optional arguments:
  -h, --help  show this help message and exit
```

# Building a Catalog Service

## Quick start

To do anything useful with DaCHS, you will have to write a resource descriptor (RD), and you'll probably have to have some data. Both must reside within some subdirectory of DaCHS' input directory (unless you configured otherwise, that's /var/gavo/inputs; we assume that in the following).

This tutorial will use the ARIHIP catalog as an example – this is a re-reduction of the Hipparcos result catalog with particularly careful solutions for proper motion. It has a column-based input and probably a few more columns than your average catalog these days. It hence is simple in principle but let us demonstrate a few advanced concepts, too.

You can, in principle, run the following under any user id, as long as you wisely manage the permissions. For testing, however, we recommend doing it as the data center administrative account (for the Debian package, that's gavoadmin, if you did the setup manually, it's whatever user did the `gavo init`).

To get a quick start, just pull in the RD in use by GAVO's data center:

```
cd /var/gavo/inputs
mkdir arihip
cd arihip
curl -O http://svn.ari.uni-heidelberg.de/svn/gavo/hdinputs/arihip/q.rd
```

The directory name ("arihip" in this case) will (normally) appear in URLs, so it's a good idea to choose something descriptive and short. The directory created here is ususally called the resource directory in DaCHS lingo.

The resource descriptor name also appears in URLs. At GAVO's data center, we usually call it `q.rd` as that looks nicely query-ish (to our tastes).

Next, get the raw data. We recommend keeping data in a subdirectory of resource directory (and suggest to call that subdirectory `data`). At the GAVO DC we usually keep everything in the resource directory under version control except for that data directory (which tends to be large, full of binary files, and either versioned by upstream or not at all):

```
mkdir data
cd data
curl -O http://dc.g-vo.org/arihip/q/cone/static/data.txt.gz
```

At this point, we're ready for ingestion. All commands to DaCHS go through the `gavo` program that has several sub-commands; in this case, we need the `import` sub-command. The sub-commands can be abbreviated as long as the abbreviation is unambiguous:

```
cd ..
gavo imp q
```

This should run for a while, reporting the number of ingested rows now and then, and finally say something like "Rows affected: XY". With this, the data is in the database and is ready for querying.

Let us mention in passing that `gavo imp` tries to interpret its first argument first as a file system path. If that fails, it tries to interpret it as an RD identifier, i.e., the inputsDir-relative path of the RD with the extension stripped. Our example RD thus has the RD id arihip/q, and you could have said:

```
gavo imp arihip/q
```

from anywhere in the file system.

After you have imported a table, it is a good idea to run `gavo info` with the DaCHS identifier of the freshly imported table, e.g.,:

```
gavo info arihip/q#main
```

The DaCHS identifier of the table consists of the RD id as introduced above, the hash (inspired by the URL fragment identifier) and the id of the table.

This will output several properties (min, max, avg) of numeric columns that may help spot import errors. Also note that for each column, the presence of NULLs is given. When you import data, it is a good idea to check whether these correspond to your expectations, and to consider declaring columns as required when they do not indeed contain NULLs.

Now start the server; if you installed from the Debian package, it is already running; stop it first for this tutorial:

```
sudo /etc/init.d/dachs stop  # only if installed from package
gavo serve debug
```

(if the last command fails with permission problems, add yourself to the gavo group, say `newgrp gavo` and try again).

The RD sets up a form-based service you can operate from a web browser; open the URL http://localhost:8080/arihip/q/cone/form[1] and play around a bit. Note the small links behind some query fields – DaCHS supports VizieR-like expressions in those fields.

Briefly have a look at the URL; apart from the host name and port (see the operator's guide on how to change those), there is the path to the RD (without the file extension), then the id of the service element (see below) and a "renderer name". That essentially defines the physical interface of the service, i.e., which protocol it is accessed through. In this case, it's `form` for an HTML form.

Another renderer supported by this service is `scs.xml`, which implements the IVOA Simple Cone Search (SCS) protocol. A client that supports this is TOPCAT; to try it out, in TOPCAT select VO/Cone Search and fill out the Cone URL field in the lower part of the window to be `http://localhost:8080/arihip/q/cone/scs.xml`. Enter some object name and a sufficiently large search radius (e.g., Aldebaran and 0.5 degrees), and you'll see the results coming in.

Incidentally, cone search does not (yet) have a usable interface to discover additional parameters, and hence TOPCAT restricts you to those mandatory for every SCS service. For instance, as delivered, arihip admits an `mv` parameter. DaCHS supports a special syntax for "free" parameters of cone searches as defined by the spectral access protocol SSAP; to say "everything brighter than 6th magnitude", the parameter setting would be `/6`; to use this

constraint within TOPCAT, the access URL needs to be amended like this: `http://localhost:8080/arihip/q/cone/scs.xml?mv=/6`.

Finally, the RD opens the arihip table for the IVOA Table Access Protocol TAP, which allows queries in a dialect of SQL. Again, TOPCAT has a nice client for TAP built in. To try it, select VO/TAP Query, enter `http://localhost:8080/tap` in the TAP URL field near the bottom of the window, and hit "Enter Query". In the resulting dialog, you can browse the table's metadata and then enter queries like:

```
SELECT * from arihip.main where sqrt(pmde*pmde+pmra*pmra)>2/3600.
```

For more information on what fancy things you can do here, see GAVO's ADQL short course

### The anatomy of the RD

Now have a look at the RD by bringing up q.rd in your favourite editor. It starts out with:

```
<resource schema="arihip">
```

RDs are normal XML files (meaning that you could, e.g., add an XML declaration if you want an encoding other than utf-8), and thus they need a root element. Hopefully unsurprisingly, this is called resource for RDs. DaCHS typically does not distinguish attributes and elements with atomic content, which means that you could also have written the fragment above as:

```
<resource>
  <schema>arihip</schema>
```

This notational freedom sometimes allows clearer notation, and it helps with defining active tags. Multiple specifications of the same property make up multiple values where the property is sequence-like (in the reference documentation this is indicated by phrases like "zero or more" or "list of" in the properties descriptions). For atomic properties, later specifications overwrite earlier ones.

The `schema` attribute on resource gives the (database) schema that tables for this resource will turn up in. You should, in general, use the name of the resource directory here. If you don't, you have to give the subdirectory name in the resource element's `resdir` attribute – either way, this is then used to build absolute paths within the RD, e.g., for the sources element discussed below.

In general, you should have exactly one RD per database schema. This is not enforced, but sharing schemata between RDs will cause many undesirable behaviours. An example is permissions: When importing a table, the schema access rights are adapted. If you have one RD A defining an ADQL-queriable table in schema X and another RD B that has no ADQL-queriable table, importing A will make schema X readable to untrusted queries, whereas importing B will make it unreadable again; this would lead to query failures (which could, in this case, fixed by adding untrusted to B's readRoles manually, but you get the idea).

Another hint: There's a fairly large body of RDs at http://svn.ari.uni-heidelberg.de/svn/gavo/hdinputs, and most of them are free for inspection and blatant stealing (if you need a license on any of this, let us know). These RDs can be seen live on http://dc.g-vo.org. To locate examples for concrete elements, meta items, and such, have a look at our RD element reference for these.

The RD goes on to give metadata applying to everything within the RD:

```
<meta name="title">ARIHIP astrometric catalogue</meta>
<meta name="creationDate">2010-11-03T10:13:00</meta>
<meta name="description">
  The catalogue ARIHIP has been constructed by
  selecting the 'best data' for a given star from combinations of HIPPARCOS
  data with Boss' GC and/or the Tycho-2 catalogue as well as the FK6.  It
  provides 'best data' for 90 842 stars with a typical mean error of
  0.89 mas/year (about a factor of 1.3 better than Hipparcos for this
  sample of stars).
</meta>
<meta name="creator">Wielen, R.; Schwan, H.; Dettbarn, C.; et al</meta>

<meta name="subject">Catalogs</meta>
<meta name="subject">Astrometry</meta>
<meta name="subject">Stars: Proper Motions</meta>
<meta name="type">Catalog</meta>

<meta name="coverage">
  <meta name="profile">AllSky ICRS</meta>
  <meta name="waveband">Optical</meta>
</meta>

<meta name="_longdoc" format="rst">
  The ARIHIP Catalogue is a suitable combination of the results of the
  HIPPARCOS astrometry satellite with ground-based data.

  (abridged)
</meta>

<meta name="source">
  Veröff. Astron. Rechen-Inst. No. 40 (2001); http://www/datenbanken/arihip/catalog.html
</meta>
```

```
<meta name="_intro" format="rst"> <![CDATA[
  For advanced queries on this catalogue use ADQL_
  possibly via TAP_

  .. _ADQL: /adql
  .. _TAP: /tap
]]> </meta>
```

This metadata is cruicial for later registration of the service, and some of it turns up in service responses. If you have a look at the HTML form you opened above, you will find quite a bit of it in the sidebar.

Metadata elements have a `name` attribute that gives the "kind" of metadata contained, and sometimes also determine a specific type. Metadata can be hierarchical, where hierarchy elements are separated by dots, and metadata can come in various formats as determined by the `format` attribute. If you give nothing here, DaCHS will apply some whitespace normalization, and it will interpret empty lines as paragraphs if the target format supports it. With `format="rst"`, the content will be interpreted as reStructuredText. Be careful to use consistent indenation in this case. There are some other, more obscure, formats, too, that you do not need to worry about right now.

See More on Metadata for more information on what is what here.

### Defining Tables

A major part of the metadata DaCHS deals with is the table structure. It is defined in table elements, which usually are direct children of the resoource element. A resource element may contain multiple table definitions. See http://dc.g-vo.org/arihip/q/cone/static for what upstream documentation we had when we made the service.

Skip over the `macDef` elements for now to where the `table` element begins. What you see is something like:

```
<table id="main" onDisk="True" adql="True" mixin="//scs#q3cindex"
  primary="hipno">
```

The `id` attribute of the table doubles as the name of the database table; make sure you use something that works as a valid simple SQL identifier (i.e., `[A-Za-z_][A-Za-z0-9_]*`) – DaCHS does not support for delimited identifiers as table names.

Be sure to always specify `onDisk="True"` unless you're going for special effects – without it, the table will end up only in memory. The `adql` attribute says that TAP queries should be allowed on the table; leave it out for tables not suitable for "raw" consumption by your clients.

For the `mixin` attribute, see Indices and Mixins.

Finally, using the `primary` attribute you can specify an explicit primary key of the table (if it is made up of several columns, concatenate their names with commas). This is made into a primary key for postgres straightforwardly, which means that the database makes sure there are no two rows with the same value for the primary key. Also, the database creates an index for efficient queries using the primary key.

What follows is a definition of the structure of the space-time coordinates:

```
<stc>
  Position ICRS Epoch J2000.0 "raj2000" "dej2000" Error "err_ra" "err_de"
    Velocity "pmra" "pmde" Error "err_pmra" "err_pmde"
</stc>
<stc>
  Position ICRS Epoch J2000.0 "raHIP" "deHIP" Error "err_raHIP" "err_deHIP"
    Velocity "pmraHIP" "pmdeHIP" Error "err_pmraHIP" "err_pmdeHIP"
</stc>
<stc>
  Position ICRS Epoch J2000.0 "raSTP" "deSTP" Error "err_raSTP" "err_deSTP"
    Velocity "pmraSTP" "pmdeSTP" Error "err_pmraSTP" "err_pmdeSTP"
</stc>
<stc>
  Position ICRS Epoch J2000.0 "raLTP" "deLTP" Error "err_raLTP" "err_deLTP"
    Velocity "pmraLTP" "pmdeLTP" Error "err_pmraLTP" "err_pmdeLTP"
</stc>
```

What this says is that there's a number of coordinate structures in the table, grouping together positions, errors, and proper motions in a specific reference frame. This is again a topic of its own, discussed in STC

Finally, we're at the column definitions:

```
<column name="hipno" type="integer" ucd="meta.id;meta.main"
  tablehead="HIP id" verbLevel="1"
  description="Number of the star in the HIPPARCOS Catalogue (ESA 1997)."
  required="True"/>
<column name="srcSel" type="text" ucd="meta.flag"
  tablehead="Source" verbLevel="25"
  description="Source of the astrometric solution"
  note="src"/>
```

For every column ending up in the table, there is one column element with a host of attributes. The `name` attribute is central in that it will be the column name in the database (incidentally, it's not `id` as with tables as it is quite common for different tables in one RD to have columns with the same name, and that would violate the id attribute's uniqueness constraint), the key for the column's value in record dictionaries that the software uses internally, and it is usually used to reference the column from the outside. Column names must be legal identifiers for both python and SQL in DaCHS. SQL delimited identifiers thus are not allowed (this is not the whole truth, but it's true enough, and you're saving yourself a lot of headache if you simply believe it).

The type attribute defaults to real, and can otherwise take values in valid SQL datatypes. The DC software knows how to handle

- `text` – a string. You could also use types with explicit length like char(7), but this is highly discouraged; it does not help postgres (or much anything else within the DC), but it hurts insofar as DaCHS cannot produce NULLs for such constructs in most VOTable serialisations.
- `real` – a real number.
- `double precision` – a floating point number. You should use in doubles if you need to keep more than about 7 digits of mantissa.
- `integer` – typically a 32-bit integer
- `bigint` – typically a 64-bit integer
- `smallint` – typically a 16-bit integer
- `timestamp` – a combination of date and time. While postgres can process a very large range of dates, the DC stores timestamps in datetime.datetime objects, which means that for "astronomical" times (like 10000 B.C. or 10000 A.D. you may need to use custom representations. Also, the DC assumes all times to be without time zones. Further time metadata (like distinguishing TT from UT) is given through STC specifications.
- `date` – a date. See timestamp.
- `time` – a time. See timestamp
- `box` – a rectangle.
- `spoint`, `scircle`, `sbox`, `spoly` – objects of spherical geometry, taken from pgSphere (http://pgsphere.projects.pgfoundry.org/). Ask for documentation...

Some more types (like `raw` and `file`) are available to tables in service definitions, but they should, in general, not appear in database tables.

Futher metadata on columns includes:

- `unit` – the unit the column values are in. The syntax is that of VOUnits. Unit is left out for unitless values.
- `tablehead` – a very short string designating the content. This string is typically used for display purposes, e.g., as table headings or labels on input fields and defaults to the capitalized column name.
- `description` – a longer string characterizing the content. This may be in bubble help or VOTable descriptions. Since these could be longer, you may want to put them in a child element rather than an attribute; in both cases, whitespace is normalized, so you can enter line breaks and similar for readability in the source, and they will always be rendered as a single blank. For even longer, note-like material, see Notes. An example for a long descripton:

```
<column name="aperture">
  <description>The aperture is the full-width-half-mean of the
    response function of our sage 3000 hyper-detector.</description>
</column>
```

- `ucd` – a Unified Content Descriptor as defined by IVOA. To figure out "good" UCDs, the GAVO UCD resolver can help. An easy way to come up with them is also to leave them out initially and then run `gavo admin suggestucds` (it's what we do these days).
- `required` – True if value must be set in order for the record to be valid. By default, NULL (which in python is None) is a valid value for any column and will silently be inserted if you don't assign a value in the rowmaker (see below). For required columns, an error will be raised when a value is missing. In HTTP service interfaces, missing required parameters will lead to a 400 invalid parameters HTTP response.
- `verbLevel` – A measure for the "importance" of the column. Various protocols have the notion of "verbosity", where higher verbosity means you get to see more columns with more esoteric content. Within DaCHS, verbLevel is a number between (usefully) 1 and 30, with columns with verbLevel 1 always given and those with verbLevel 30 only given if someone really wants to see all columns. Technically, in SCS, a column is part of the output table if its verbLevel is smaller or equal to ten times the query's VERB parameter.

Column elements may have a child element values. This lets you specify metadata like maximum or minimum, or enumerate possible values. The most common use is the definition of null literals, though. This is not necessary for floats, and usually not even strings, because these have useful (and actually non-overridable) null values in the VOTable representation (where this sort of thing counts most). It is, however, highly recommended to give null literals when defining integral types (including chars) that may have null values. DaCHS will try to pick useful null values for those automatically when possible, but when streaming tables, this is impossible, and errors will be raised during VOTable rendering when NULLs are encountered in such a situation.

So, just define null values whenever you define a non-required integral column, like this:

```
<column name="n_obs" type="integer"
    description="Number of
    observations, NULL if interpolated data">
  <values nullLiteral="-1"/>
</column>
```

The output of `gavo info` (see above) can help you to choose suitable NULL values. To help people spot them when metadata is missing, it's usually wise to choose "conspicuous" null values (like -1, 9999, or similar).

Table elements may contain metadata. You do not need to repeat metadata given for the resource, because (in most cases) the DC performs metadata inheritance. This means that if a table is asked for a piece of metadata it does not have, it forwards that request to the embedding resource. For multi-table resources, you should usually give `title` and `description` metas.

Scrolling a bit further down in the arihip RD, you'll notice some LOOP constructs. These are discussed below under active tags.

At the end of the table element, there are meta elements called "note"; for those, see Notes.

### Parsing Input Data

Going further down in the RD, you will find a data element. Their main purpose is to describe how certain input files fill the table(s) defined above. It starts like this:

```
<data id="import">
  <sources>data/data.txt.gz</sources>
```

Data elements can have ids which can be used to indiviually reference them from a `gavo imp` command line; this is useful if you just want to import one part of a multi-table data collection. The default of `gavo imp` default is to build all `data` elements except those having an `auto="False"` attribute.

It is recommended that the id is a short verb phrase, as `data` basically contains instructions for an action. You might rightly argue that we have not chosen the element name too aptly (`recipe` might have been more appropriate), but we feel it's too late to change it now.

The sources element lets you specify the names of the input files to be processed. There are several ways to do that; in this case, there's just one input file, which is given as element content, with the path interpreted relative to the resource directory. If the data was distributed into several files in two directories, something like the following specification would do the trick:

```
<sources>
  <pattern>inp2/*.txt</pattern>
  <pattern>inp1/*.txt</pattern>
</sources>
```

The `sources` element also has a recurse (boolean) attribute that makes DaCHS search for the pattern in the subdirectories of the path part of the pattern.

Now have a look at the input file:

```
zless data/data.txt.gz
```

You'll see that we have a plain ASCII file with aligned columns, header lines, and even a cell separator ("|"). That's still a fairly common format for raw data, but by no means the only one. To give DaCHS the flexibility to deal with anything upstream cares to throw at you, DaCHS has the concept of a grammar.

There are many grammars defined, e.g., for getting values from FITS files, VOTables, or using column-based formats; you can also write specialized grammars in python. All grammars read "something" and emit a mapping from names to (mostly) string values; those unprocessed string-to-string mappings are called "rawdicts" in DaCHS jargon (distinguished from "rowdicts" ready for database ingestion, which sport processed and typed data).

It is often useful to inspect what a grammar emits. You can do that using import's `--dump` flag. During development, it is frequently convenient to just import a few rows and watch what they produce; this would look like this:

```
gavo imp -M 100 --dump q.rd | less
```

(if you interrrupt the above command, your table should be unscathed – check with `gavo info` –; otherwise just re-run the full import).

With a source that has both a separator character and aligned columns, there are several valid choice for which grammar to use on this particular file. In the RD, it next says:

```
<columnGrammar topIgnoredLines="9" preFilter="zcat">
  <colDefs>
    hipno:      3-8
    srcSel:     47-49
    alphaHMS:   59-73
    deltaDMS:   77-91
    pmra_mas:   95-103
    pmde_mas:   107-115
    t_ra_mod:   119-123
    err_ra_mas:127-131
    err_pmra_mas:135-139
    t_de_mod:   143-147
    err_de_mas:151-155
    err_pmde_mas:159-163
    parallax_mas:167-172
    e_parallax_mas:176-180
    kp:         184
    vrad:       188-195
    mv:         199-203
    km:         207
    kbin:       211-212
    kdmu:       216
    kae:        220
    flags:      882-901
  </colDefs>
```

– so we went for a columnGrammar. Those cut up every input line along character indices that are here, following the display in most editors, are counted from 1 upwards. Note that in ranges, the last column is included in the string – these are no python slices but basically a representation of the character ranges in VizieR-style "byte-by-byte"-descriptions.

The assignment of names to column ranges can happen both in a `colDefs` element as shown above – one specification per line, label mapped to a column or a range of columns.

Alternatively, have several `col` elements, each of which has a `key` attribute that gives a name. This could be the `name` of a target column in the simplest case, or it can be an auxillary identifier that is later processed in a rowmaker. These individual specifications are interesting when combined with RD macros, and that's where they come in in the arihip RD (again, using LOOPs).

Grammars also have various attributes; the ones parsing from text files support, for example, `topIgnoredLines`, which allows you to skip header lines, and `preFilter` that allows running the input through a shell command before it is processed using DaCHS (if you find yourself doing more than just decompression in such a preFilter, you should probably look for a different solution).

### Mapping data

The arihip RD then goes on with:

```
<make table="main">
  <rowmaker idmaps="*">
    <var name="raj2000">hmsToDeg(@alphaHMS, None)</var>
    <var name="dej2000">dmsToDeg(@deltaDMS, None)</var>
    ...
    <map dest="kbin">parseWithNull(@kbin, str, "9")</map>
    <map dest="vrad">parseWithNull(
      @vrad, lambda a:float(killBlanks(a)), "")</map>
```

The make element brings together a table (in the `table` attribute) with a recipe how to fill it from the output of the grammar (the row maker).

Incidentally, there can be multiple `make` elements in a single `data` element if multiple tables (using different row makers) are generated from the same grammar output. This is particularly useful in combination with dispatching grammars that let, in effect, the grammar choose which make to use.

Makes can also carry scripts in SQL or python, at various points of the building process. These let you perform all kinds of higher magic. For details, see the chapter on scripting in the reference.

As explained above, output of grammars and hence the input to a make is a sequence of mappings from names to strings (the "rawdicts"). The database, on the other hand, wants typed values, i.e., integers, time stamps, etc. Also, data in input tables is frequently given in inconvenient formats (e.g., sexagesimal angles), deprecated or inconsistent units, or values may be distributed over multiple columns (e.g., date and time of an observation when we want a single timestamp). To cover these and more tasks, DaCHS has row makers, the results of which are then called rowdicts (note the subtle difference from the rawdicts coming in from the grammars: "row makers turn rawdicts into rowdicts").

Basically a row maker consists of

- var elements -- assignments of expression values names in the rawdict.

- map elements -- simple mappings of (python) expressions to values in the destination rowdict
- procedure applications (see apply) -- manipulations of both rawdicts and rowdicts in python code

The fragment above shows one of several ways to use both `var` and `map` (which work exactly the same way, except that vars end up in the rawdict, and map in the rowdict): generating values from python expressions, where there is the special syntax `@identifier` which expands to whatever value the rawdict has for that key (or raises a KeyError if the key is not present in the rawdict).

The rawdict manipulations that `var` does are useful if you want to re-use whatever you compute. The `map` element, on the other hand, writes directly into the results dictionary, the keys of which directly correspond to the column names.

When building a rowdict for ingestion into the database, a row maker first binds var names, then applies procedures and finally performs the mappings. In the bodies of the mappings, you can use all built-in python functions plus a set of useful rowmaker functions documented in the reference documentation, as well as everything from the python standard library modules `datetime`, `math`, `os`, `re`, `sys`, `time`, and `urllib` (you need to give the module name when referring to names from these modules as in, e.g., `re.sub`). Furthermore, the gavo modules `base`, `stc`, and `utils` are in the namespace of the mapping code, as well as the submodule `utils.pgsphere`. TODO: link to useful documentation for them here.

For simple cases, maps will suffice; frequently, you can do without python expressions by giving a `src` attribute specifying a rawdict key instead of element content (which is preferable if possible – as elsewhere, less code is better in RDs, too). The rawdict string will in this case be converted to a typed value using "sane" defaults (e.g., integers will be converted by python's int constructor, where empty strings are mapped to None, datetimes are parsed as ISO strings, etc)

If you match the keys in the rawdicts with the names of the database columns their content is supposed to end up with and the content needs no further manipulations, a row maker like:

```
<rowmaker>
  <map dest="evi" src="evi"/>
  <map dest="av" src="av"/>
  <map dest="ai" src="ai"/>
</rowmaker>
```

would to the trick. Since this is a bit unwieldy, DaCHS provides a shortcut:

```
<rowmaker> <simplemaps>evi:evi,av:av,ai:ai</simplemaps> </rowmaker>
```

which expands to exactly what is written above. The keys in each pair do not need to be identical; the first item of each pair is the table column name, the second the rawdict key.

The case where the names of rawdict and rowdict keys are identical is so common (since the RD author in general controls both) that there is yet another shortcut for this:

```
<rowmaker>
  <idmaps>evi,av,ai</idmaps>
</rowmaker>
```

Idmaps sets up one map element each with both dest and src set to the value for every name in the comma separated list idmaps.

You can abbreviate this further to:

```
<rowmaker idmaps="*"/>
```

– so, idmaps values can contain shell patterns. They will be matched to the column names in the target table. For every column for which there is no explicit mapping, an identity mapping (with type conversion) will be set up with this specification.

Of course, you can have values that do not even depend on grammar output:

```
<map dest="dateIngested">datetime.datetime.now()</map>
```

Null values are always troublesome. Within DaCHS, the null value (almost) always is python's None. There is the row maker function `parseWithNull` to help you come up with those; if your upstream was devious enough to use 99.99 as a null value for a magnitude, you could say:

```
<map dest="Vmag">parseWithNull(@VmagSrc, float, "99.99")</map>
```

Note that the null value here is a literal matched against the string coming from the grammar; due to the rounding errors when converting from decimal to binary floating points, you can only safely compare against relatively few floating point numbers (99.99 is not among them), so you shouldn't do that if you can avoid it.

If you need to scale this (or if null values are chosen that they are invalid literals to begin with), a feature that lets you null out a value when an specific type of exception is raised comes in handy. This is map's `nullExcs` attribute, which is just a comma separated list of exceptions that should be caught and interpreted as "this is null". If, in the example above, the source would give the magnitude in millimags to save a comma, you could use:

```
<map dest="Vmag" nullExcs="TypeError"
  >parseWithNull(@VmagSrc, float, "99999")/1000.</map>
```

If `parseWithNull22 here returns None, a ‘‘TypeError` will be raised and caught, and `Vmag` will be None.

You can turn more than one exception into None. For example example, if `magicOffset` has been parsed before and could be None, while `magicLit` is to be parsed and has the empty string as a NULL literal, you could write:

```
<map dest="magic" nullExcs="ValueError,TypeError"
  >@magicOffset+float(@magicLit)</map>
```

If `magicOffset` is None, magic will be None via the `TypeError`, whereas empty `magicLits` will result in Nones via a `ValueError`.

We defer the discussion of `apply` elements to the discussion of how to build SIAP services.

`rowmaker` elements may also be direct children of `resource`; this is for when they are used in more than one `data`. You would then give the rowmaker an `id` attribute and say something like `<make rowmaker="id-of-rowmaker" table=.../>` However, for the standard case it's best to keep everything related to a given import together in the make element.

### Indices and Mixins

We have so far deferred the discussion of the `mixin` attribute in arihip's opening table element:

```
<table id="main" onDisk="True" adql="True" mixin="//scs#q3cindex"
  primary="hipno">
```

Mixins are DaCHS' primary tool to endow tables with "everything needed to serve a standard" (e.g., a minimal set of columns, certain indices, or metadata). For instance, an image table must have a certain structure determined by the SIA

protocol. Either of the `//siap#pgs` and `//siap#bbox` mixins make sure that tables have this structure, and they make sure that the table containing information on all the file-like datasets in the data center (which is called `dc.products`) is updated when the table is filled.

The content of the mixin element (or the attribute value when you give the mixin property as an attribute) is a reference to a mixin definition. These references typically go into some system descriptor (though you could define your own mixins), and the double slash in a DaCHS reference means "system descriptor" (in actual truth, it's just an abbreviation for `__system__/`). The reference documentation contains a chapter on DaCHS' public mixins. For the curious: you can have a look at the actual definitions by admin's `dumpDF` subcommand, e.g., like this:

```
gavo admin dumpDF //scs
```

The `//scs#q3cindex` mixin referenced here arranges for spatial indexing of tables having some sort of spherical coordinates. To identify which columns to index, DaCHS inspects the UCDs of the columns; what it looks for here are columns with UCDs of `pos.eq.(ra|dec);meta.main‘` as index columns[2]. Contrary to mixins for other standard protocols, it does not automatically insert these columns (and neither the only other required column in SCS, the main row identifier with the UCD `meta.id;meta.main`).

Since in addition to spatial queries, we also expect a lot of queries constraining the `mv` column, we ask for an index on it using DaCHS' index element, which is a child of `table`:

```
<index columns="mv"/>
```

This is the simplest, but mostly sufficient, form of defining an index; for advanced usage, please refer to the reference documentation. If you decide to add an index to a table later on, or to initiate re-indexing after a lot of data changed, see the `-I` option of `gavo imp`.

### Cores and Services

The last but one part of the RD deals with how to get the data out of the database again, i.e., the services exposing the data. This part is fairly simple for arihip:

```
<service id="cone" allowed="scs.xml,form">
  <meta name="shortName">arihip cone</meta>
```

```
  <meta name="testQuery">
    <meta name="ra">9.4076</meta>
    <meta name="dec">9.6414</meta>
    <meta name="sr">1.0</meta>
  </meta>

  <dbCore queriedTable="main">
    <FEED source="//scs#coreDescs"/>
    <condDesc buildFrom="mv"/>
    <condDesc>
      <inputKey original="hipno" required="False"/>
    </condDesc>
  </dbCore>

  <publish render="scs.xml" sets="ivo_managed"/>
  <publish render="form" sets="ivo_managed,local"/>
  <outputTable verbLevel="20"/>
</service>
```

To understand what's going on here, some basic understanding of DaCHS' service architecture is required; it consists of:

- cores; they actually do the computation or database query
- renderers; these digest the data coming in from the service and (in general) format the result in some way requested by the user. There are renderers for web forms, VO protocols, imges, etc. Frequently – as in the example –, you can use the same core for both a VO protocol and a form-based service by just allowing different renderers.
- The service; it holds together the core and the renderer, can reformat core results, controls the metadata, etc.

The renderers are referenced by name in the service's `allowed` attribute. What can be given there (concatenated by commas) is listed in the reference documentation's renderer chapter. As you have seen above, the renderer is selected via the URL. If a client tries to retrieve a URL with a renderer that is not in the service's `allowed` list, DaCHS will respond with a 403 forbidden HTTP code (excepting certain "unchecked" renderers like `info` that typically expose service metadata). In addition, not all cores can be combined with all renderers even if you list them in `allowed`. For example, the `ssap.xml` renderer will not (usefully) work on anything but an `ssapCore`.

The most common core for catalog services (and the one you'll typically use for SCS services) is the dbCore, as used here. See cores available in the reference documentation for more predefined cores -- e.g., to run ADQL queries or to upload files. For special functionality, you can even write your own core.

The dbCore generates a (single-table) query from condition descriptors and returns a table that you describe through an output table. Cores are defined as direct children of the resource (as with grammars, you can also have them in `resource` and then write `core="id-of-element"`, which makes sense when a single core is shared by several services).

dbCores need a `queriedTable` attribute, the value of which must be a table reference. This is the table the query will run against.

The condition descriptors (or condDescs for short) define input fields (for the form renderer, these will be rendered as form items people can fill in). Most commonly, you will either define them using the `original` attribute (when inheriting from predefined condDescs) or using `buildFrom`. The first case is typically used in connection with protocols and on tables having mixins; such condDescs result in zero or more input fields, and they typically inspect the queried table. For example, the `//scs#humanScs` condDesc locates the "main" positions as identified by UCDs and generates queries against them using two input fields, one it tries to guess a position from, and another for the search radius.

When you define a `condDesc` using `buildFrom`, the result is usually one or more input field(s) constraining values in the column named in the `buildFrom` attribute. The software tries to make some useful input definition from that column, depending on the renderer. Renderers with a parameter style (this is given in each renderer's description the reference documentation) of "form", for example, let users query string-like columns using Vizier-like string expressions, real and double precision columns using Vizier-like float expressions, and so on. The `pql` style allows a specification like for SSAP (e.g., "range_min/range_max").

The `service` element must have an `id` attribute that is used to select the service run in the access URL. Furthermore, there should be certain pieces of metadata useful in later registration. First, there's `shortName`, which is typically used by clients in space-restricted displays. It must not be longer than 16 characters, so something like an acronym and a very terse role identifier is the best you can do (hence the "arihip cone" here). Frequently, a `title` meta is also useful, in particular when an RD contains multiple services, in which case one could be "Cone search for ARI's HIPPARCOS re-reduction" and the other, say, "Autocorrelation on ARI's HIPPARCOS re-reduction".

See the data checklist for more information on useful generic metadata and remember that services inherit whatever is defined within `resource` when not specified within the element.

Many standard VO protocols require additional, protocol-specific metadata. In the case of arihip, we have a Simple Cone Search service, which, as laid down in the section on the scs.xml renderer in the reference documentation, requires the parameters of a test query returning a nonempty result.

## Starting from Scratch

When you start to write your own RD, it might be a good idea to start from our RD template. To do this, create your resource directory, go in there and say:

```
gavo admin dumpDF src/template.rd_ > q.rd
```

(dumpDF stands for "dump distribution file" and is the canonical way to get at DaCHS "built-in" files).

Right now, this only contains the skeleton for the metadata. We may expand it as we get good ideas on how to keep things generic. Or should we have different templates for various major service types?

There's also `gavo mkrd`, which can generate RD templates from some types of inputs. We're not convinced this kind of thing actually is useful, but you're welcome to try it and encourage us if you like it, in particular if you have ideas on how to improve things.

## Debugging

Writing RDs is like programming, and sometimes it involves actual programming in python.

Hence, you'll get things wrong, and DaCHS probably will annoy you. Giving good error messages – neither drowning you in a deluge of mostly-useless information nor swallowing important pieces of data, neither claiming too much nor to little, not misleading you about what is actually going on – is a high art, and we are aware we should be doing better. Your feedback will help us improve.

Still, with a few hints and techniques figuring out what's wrong isn't much harder in DaCHS than with your average programming system. In the following, we collect some hints on what to do if things don't work.

### General Hints

- If you get error messages, be sure to check our hints on common problems – may commonly encountered problems are explained there with suggestions for how to fix them.
- The gavo command takes a `--hints` switch. With it, error messages are frequently accompanied with – you guessed it – hints on what might cause the problem and possible solutions. Note that `--hints`, like the other debugging switches, goes between `gavo` and the subcommand name, as in `gavo --hints serve`.

- Validate your RD. This is, in general, a good idea before doing anything with the RD, since it will allow you to more easily catch errors than the in all likelihood even more byzantine error messages that may arise when something goes wrong later. The `gavo val` subcommand takes one or more RDs. If you don't understand its output, complain to gavo@ari.uni-heidelberg.de -- the command is really intended to help you catch errors, and if it doesn't do so, it's a bug.

- Check the logs. They are in /var/gavo/logs by default, and there's dcErrors and dcInfos (you'll want to look at both).

- When hunting bugs, it's usually a good idea to enable the logging of (many) tracebacks by passing the `--debug` flag to `gavo`.

- If you're trying to figure out server behaviour, don't run the server daemonized but use `gavo serve debug` instead; this won't detach and log to stdout.

- With this (or the problem is in normally-running DaCHS code in the first place), the python debugger is your friend. The `gavo` command has an option `--enable-pdb` that will dump you into the debugger where an uncaught exception happened (as the server should never let an uncaught exception through, that's not useful with `gavo serve`). If that doesn't help you, you can set breakpoints (e.g., in your own in-RD procedure defininitions by writing `import pdb; pdb.set_trace`. If you install the python-ipdb package, and write ipdb instead of pdb, you'll get a nicer debugger commandline with tab completion and similar frills.

- To see what SQL is actually sent to the database, set the GAVO_SQL_DEBUG environment variable to any value. This could look like this:

  ```
  env GAVO_SQL_DEBUG=1 gavo imp q create
  ```

  The first couple of requests are for internal use (like checking that some meta tables are present).

- If `gavo serve start` doesn't actually cause the server to run, something went wrong after detaching from the controlling terminal. The messages are in the logs directory, in the file `serverStderr`.

### Debugging Services

Debugging services is of course an extra challenge since code runs deep in the bowels of a complex system, with timeouts, threads, and all kinds of nastyness involved. The first adivce is: Pull whatever is mysterious into your development system and run `gavo serve debug`. You can even do the `import`

`pdb;pdb.set_trace()` trick then. It's a bit tricky to communicate with the debugger in between the log messages of `gavo serve debug`, and there's no readline support since pdb doesn't think it's running within a terminal there, but it's definitely doable.

Another challenge is that sometimes problems manifest themselves in a running server. In that case it's sometimes useful to open a manhole into the server. One reasonably convenient way to do this is to but a special RD somewhere (e.g., `__tests/manhole.rd`) and use some RD-embedded code to introspect the server. We ususally use a datalink service for this since it keeps things nicely self-contained – an obvious alternative with less bending could be a custom renderer.

Here's how something that fiddles out a column property would look like:

```
<resource schema="test">
  <service id="look" allowed="dlget">
    <datalinkCore>
      <descriptorGenerator>
        <code>
          return ProductDescriptor(None, None, None, 'text/plain', )
        </code>
      </descriptorGenerator>
      <dataFunction>
        <code>
          descriptor.data = "debugging"
        </code>
      </dataFunction>
      <dataFormatter>
        <code>
          class DebugResult(Page):
            def renderHTTP(self, ctx):
              request = IRequest(ctx)
              request.setHeader("content-type", "text/plain")
              return "%s"%base.caches.getRD("maidanak/res/rawframes"
                ).getById("rawframes").getColumnByName("accref").displayHint
          return DebugResult()
        </code>
      </dataFormatter>
    </datalinkCore>
  </service>
</resource>
```

All but the data formatter is just blind code for hiding our true intentions from the datalink machinery. In the data formatter, you can return arbitrary text. You can now access `http://localhost:8080/__tests/manhole/look?ID=0` and see whatever gets returned from `renderHTTP` (the value of ID obviously is arbitrary here, although you could use it to transmit information into your debugging code; not that we think that's a good idea).

Note that you can edit manhole.rd, save it, and reload the page; the server will notice your changes and display the new result without a restart.

### Case Studies

In this section we will damage the arihip RD in various ways, show you how the errors manifest themselves, and how you could try and figure them out. It's highly recommended to play the scenarios; it's very well possible that the actual messages will look differently for you if we've changed and hopefully improved the software. We're thankful if you point us to outdated samples.

**XML syntax errors** These are mostly easy to diagnose (and fix), except that sometimes the errors show up too late. For an example of a dramatic failure, delete the closing tag of the `source` meta. The result then is:

```
arihip > gavo imp q
** Error: In /home/msdemlei/gavo/inputs/arihip/q.rd: mismatched tag:
line 674, column 2
```

In this particular case, a dedicated XML validator gives more helpful diagnostics:

```
/arihip > xmlstarlet val -e q.rd
q.rd:674.12: Opening and ending tag mismatch: meta line 72 and resource
</resource>
           ^
q.rd - invalid
```

The reason DaCHS does so bad here is that `meta` elements are allowed to have all kinds of children (for typed meta, which we've not covered so far). DaCHS does better when you add some random XML fragment, e.g., the wonder element in the next example:

```
<table id="main" onDisk="True" adql="True" mixin="//scs#q3cindex"
     primary="hipno">
   <wonder>foo</wonder>
```

This time, DaCHS gets it pretty much right:

```
msdemlei@victor:/home/msdemlei/gavo/inputs/arihip > gavo imp q
** Error: At /home/msdemlei/gavo/inputs/arihip/q.rd, (112, 4): table
elements have no wonder attributes or children.
```

Well-formedness problems frequently turn up with embedded python code or reStructuredText markup. To see what happens then, delete the opening CDATA sequence in:

```
<meta name="note" tag="tabflags"><![CDATA[
```

This results in:

```
arihip > gavo imp q
** Error: In /home/msdemlei/gavo/inputs/arihip/q.rd: not well-formed
(invalid token): line 411, column 24
```

If you check out the indicated line, you'll see some table markup. Now that you're warned, you'll probably see immediately that there's a less-than sign there that's not allowed in XML parsed character data, but while writing up such material, it's easy to forget that $<$ and & are magic to XML. CDATA is your fried for embedded formal languages.

Now for a particularly nasty syntax error that's not XML-related at all. To trigger it, go to the note meta with the "src" tag and remove whitespace from the second column line like this:

```
<meta name="note" tag="src">
  The srcSel field indicates which catalogue the astrometric solution
was taken from using the following codes:
```

That results in:

```
arihip > gavo --debug imp q
** Error: At /home/msdemlei/gavo/inputs/arihip/q.rd, (317, 4): Bad
text in meta value (Bad indent in line u'    was taken from using the
following codes:')
```

If you go to the position given, you'll notice it's the end of the meta element. What bugs DaCHS there is somewhat pythonesque. Both python and reStructuredText are indentation-sensitive. In particular, even if you, say, indent all lines in a python program by two blanks, the result will be invalid source code.

Hence, DaCHS removes leading indentation from the content of all elements containing such material, and the amount of indentation is goverened by the second line, where the line with the opening tag counts; in the example above, DaCHS tries to subtract from every subsequent line the indentation of the line starting with "The srcSel field" (the first line is the one containing the tag).

This kind of problem is particularly insidious if you're mixing blanks with tabs for indentation. Don't do this in python, don't do this in RDs.

All this means that RDs can break of XML processing tools normalize whitespace in certain elements. This is a bit unfortunate since the way RDs are written, they're not even completely wrong in doing this. So, the bottom line is: you need careful instructions to standard XML processors if you actually want to *write* RDs. However, if you feel the need to write RDs programmatically, you're probably doing it wrong: RDs already generate things, and if another generation layer seems necessary, that would indicate a design problem *somewhere* – possibly in DaCHS.

**Rowmaker trouble** Since rowmakers are a fairly thin layer on top of python, it's easy to elicit fairly confusing messages here. To see how this looks in an easy case, change the kbin mapping to:

```
<map dest="kbin">parsWithNull(@kbin, str, "9")</map>
```

(note the missing e). The result is an error message that looks a bit frightening with rawdicts as large as in this case:

```
arihip > gavo imp q
Making data import
Starting /home/msdemlei/gavo/inputs/arihip/data/data.txt.gz
Failed /home/msdemlei/gavo/inputs/arihip/data/data.txt.gz
** Error: Row {u'pmdeLTP': None, u'srcSel': 'T2H', u'err_raHIP':
[abridged]
.. .... ... ...', u'ddeLTP': '-     2.37', u'pmra_mas': '-    4.85',
u'raHIP': None} Field kbin: While building kbin in None: name
'parsWithNull' is not defined
```

The dump of the rawdict, however, is often helpful enough to warrant the scary appearance, even at the risk of obscuring the actual message at the very end. Note that the "while building kbin" tells you where in the rowmaker something went wrong: At the mapping of kbin. The `in None` part may be a bit less fortunate – the "None" here is the id of the rowmaker, which you didn't give. If you have multiple rowmakers in an RD, it's a good idea to name them. So, add an `id` attribute as in:

```
<make table="main">
  <rowmaker idmaps="*" id="make_main_row">
```

and to the improvement in the error message:

```
Field kbin: While building kbin in make_main_row: name
'parsWithNull' is not defined
```

Now undo the changes and try another frequent problem by deleting the vrad mapping, i.e., removing the entire element:

```
<map dest="vrad">parseWithNull(
   @vrad, lambda a:float(killBlanks(a)), "")</map>
```

This gives:

```
/arihip > gavo imp -M 12 q
[...]
Field vrad: While building vrad in None: could not
convert string to float: +   8.3
```

What's going on? Something is going on with vrad; specifically, the string '+ 8.3' cannot be turned into a float (which is because it's not a valid float literal). But since we're no longer mapping vrad, why is the machinery even trying that? Well, this is a consequence of the `idmaps="*"` of this rowmaker. When there is a key vrad in the rawdict and a column vrad is required, the default transformation from string to float is tried (and this fails in this case).

What happens if no such input key is present? To find out, additionally remove the line:

```
vrad:      188-195
```

from the column grammar's `colDefs` element. The result is:

```
arihip > gavo imp -M 12 q
Making data import
Starting /home/msdemlei/gavo/inputs/arihip/data/data.txt.gz
  Source hit import limit, import aborted.
Done /home/msdemlei/gavo/inputs/arihip/data/data.txt.gz, read 13
Shipped 13/13
Create index Primary key on arihip.main
Create index main_mv
Create index main_q3c_main
Rows affected: 13
```

– a clean import. However, all vrads in the table are now, of course, NULL:

```
arihip > gavo info arihip/q#main
 col           min         avg     max       hasnulls
 [...]
 vrad          None        None    None      True
 [...]
```

– watch out for those. For DaCHS, it's normally no problem if a key is missing in the input (which is in many situations desirable); the missing value is just replaced with None. You can forbid NULLs using the `required` attribute on the column vrad by editing it like this:

```
<column name="vrad" ucd="phys.veloc;pos.heliocentric"
  required="True"
  tablehead="v_rad" verbLevel="20" unit="km/s"
  description="Radial velocity as used in calculating the foreshortening
  effect."/>
```

This now yields an error; note that the message is fairly generic, but if you consider that rawdicts are mappings, you will at least see the logic in it:

```
arihip > gavo imp -M 12 q
Making data import
Starting /home/msdemlei/gavo/inputs/arihip/data/data.txt.gz
  Source hit import limit, import aborted.
Done /home/msdemlei/gavo/inputs/arihip/data/data.txt.gz, read 1
** Error: Row {u'pmdeLTP': None, u'srcSel': 'T2H', u'err_raHIP':
[...]
u'ddeLTP': '-     2.37', u'pmra_mas': '-    4.85', u'raHIP': None}
Field vrad: While building vrad in None: Key 'vrad' not found in a
mapping.
```

When debugging stuff like this, it is sometimes useful to cut-and-paste the rawdict dumped into a file, join all the lines, remove the `parser_` key-value pair (the content of which cannot be represented as a string), assign it to a name and then manipulate it in the rest of the file using python statements. You can have a similar effect by giving `--enable-pdb` as a `gavo` main option; this will dump you in a debugger at the place of the problem.

Undo all changes to the arihip RD to continue.

If you embed code yourself, the potential for challenging bugs is yet larger. To see how basic problems are reported, add:

```
 <apply>
  <code>
    ddt
  </code>
</apply>
```

somewhere within the RD. This yields:

```
arihip > gavo imp q
[...]
u'raHIP': None} Field proc98577cc: While executing proc98577cc in
None: global name 'ddt' is not defined
```

The message coming from the bowels of python is clear enough, but the alphabet soup giving the error is not. This name was invented by DaCHS since no `name` (sorry, not id this time) attribute was given to `apply`. Try it, the error message will be much more palatable with it.

The error message still could be more helpful, however; consider code like:

```
<apply>
   <code>
     a = 1
     b = 2/a
     c = 3/(a+b)
     d = 4/(a+b+c+1)
     e = 5/d
   </code>
 </apply>
```

The error message here is:

```
While executing proc985706c in None: integer division or modulo by zero
```

So – where does this happen? To get an idea, you can pass the `-debug` flag to gavo, which in the `dcInfo` log file at least yields:

```
2013-09-23 15:57:37,304 [INFO 24430] Swallowed the exception below, re-raising Field proc985706c
Traceback (most recent call last):
  File ".../gavo/rscdef/rmkdef.py", line 583, in __call__
    exec self.code in self.globals, locals
  File "generated mapper code", line 13, in <module>
  File "<string>", line 9, in proc985706c
ZeroDivisionError: integer division or modulo by zero
```

The trouble is that the source code line isn't given, and the source that refers to isn't yet visible to you in this case. We promise to improve the management of the source code. Until then, frankly, the nicest way to debug stuff like this is to write:

```
<apply>
   <code>
     import pdb; pdb.set_trace()
     ...
```

and then use the python debugger to figure things out.

## More on Grammars

In addition to the `columnGrammar` mentioned above, there are several other grammars you should know about; the full list of grammars available is found in the reference documentation.

### reGrammars

The reGrammar is another grammar suitable for parsing text files. The idea here is that you give two regular expressions to separate the file into records and the records into fields, and that you simply enumerate the names used in the mapping.

In the simplest case – whitespace separated columns in lines containing no whitespace, with one comment line at the top –, such a grammar could be specified like this:

```
<reGrammar topIgnoredLines="1">
  <names>raMin, raMax, decMin, decMax, EVI, AV, AI</names>
</reGrammar>
```

reGrammars have a few tricks built-in to make them a bit more versatile. The following lets you simulate properly parsing some bzipped database dump by making some strong assumptions:

```
<reGrammar topIgnoredLines="15" preFilter="bzcat">
   <fieldSep>"?,(\s*")?</fieldSep>
   <recordCleaner>\((.*)\)</recordCleaner>
   <names>primflag, measure_no, paper_id, source_no, source_name</names>
</reGrammar>
```

If things get noticeably more complex than this, an reGrammar may no longer be a terribly good solution. Indeed, we would welcome a contributed grammar that would do a somewhat more robust parsing of common SQL text dumps.

### fitsProdGrammar

This grammar exposes FITS headers as rawdicts. Since both data structures represent essentially the same data structure – sequences of key-value pairs, you can get away with just:

```
<fitsProdGrammar/>
```

– and that would cover a lot of use cases that read FITS files.

DaCHS uses pyfits to parse the headers and thus supports all the major conventions (CONTINUE cards are transparent, as are HIERARCH-style long keywords). Neither HISTORY nor COMMENT cards are present in the rawdicts.

There are some snags, anyway. For example, DaCHS by default reads the header bytes using specialized code that is somewhat more robust and has more desirable behaviour with odd files than the pyfits one (you can deselect it by setting `qnd="False"` if necessary). This gives up collecting header blocks if it has not found the end card after `maxHeaderBlocks` blocks. The default `maxHeaderBlocks` is chosen to be 40, which is reasonable for reasonable FITSes. However, we have seen in the wild massive abuse of comment cards to hold entire tables. If you're unlucky enough to have to handle such files, you may have to raise this.

It is fairly common for FITS keywords to contain a dash (-). Since rawdict keys are supposed to be python identifier (e.g., for @-referencing), fitsProdGrammars translate these to underscores. If further cleanup is necessary, there is the mapKeys element that lets you write things like:

```
<mapKeys>
  <map key="properName">PROP NAM</map>
</mapKeys>
```

The element should only be used to fix crazy names. Actual mapping of names should be performed in rowmakers.

FITS files are somewhat more complex, and fitsProdGrammars expose some of this. If you need to parse from a header other than the primary one, give the 0-base extension number in hdu. For the full power of pyfits (but also all incompatibilities that are introduced by using that power), there is the `hdusField` attribute. This gives the key under which the pyfits HDU is visible in the rawdict. Be warned that using this might make your grammar *dramatically* slower, in particular when it operates on gzipped FITS files (which are, incientally, supported if their file names end in ".gz").

### csvGrammar

csvGrammar parses from files containing comma separated values. It actually is a thin wrapper around python's csv module, and thus it is fairly forgiving about the idiosyncrasies of CSV (e.g., quotes around all, some, or no values). You can configure the delimiter character via the same-named attribute (it defaults to, expectably, comma). It is currently *required*, though, that the first row gives the column headings. If you need the capability to name fields as in, say, the reGrammar, let us know – it's just a few lines of code.

## Source Fields

All grammars can have a `sourceFields` element. It contains a standard procedure definition (i.e., you could predefine those and bind parameters), but usually you will just fill in the code.

This code is called once for each source processed, and receives the sourceToken as argument. It must return a dictionary, the key/value pairs of which will be added to all rows returned by the row iterator.

The purpose of sourceFields is to precompute values that depend on the source ("file") and are constant for all rows within it. An example for where you need this is when you want to create backlinks to the file a piece of data came from:

```
<xygrammar>
  <sourceFields>
    <code>
      srcKey = utils.getRelativePath(sourceToken,
          base.getConfig("inputsDir"))
      return locals()
    </code>
  </sourceFields>
</xygrammar>
```

You can then retrieve the path to the source file via srcKey key in rawdicts (and then, using render functions and static renderers, turn this into links).

In addition to the sourceToken, you also have access to the data that will be fed from the grammar. This can be used to, e.g., retrieve the resource directory (`data.dd.rd.resdir`) or data descriptor properties (`data.dd.getProperty("whatever")`).

Sometimes you want to do database queries from within sourceFields. This is tricky when you access the table being written or otherwise being accessed. This is because sourceTokens run in the midst of a transaction updating the table. So, something like:

```
<code>
  <!-- will deadlock, don't do it like this -->
  base.SimpleQuerier().query(...)
</code>
```

will wait for the transaction to finish. But the transaction is waiting for data that will only come when the query finishes -- this is a deadlock, and gavo imp will just sit there and wait (see also Deadlocks).

To get around this, you need to query using the data's connection. So, instead write:

```
<code>
  base.SimpleQuerier(connection=data.connection).query(...)
</code>
```

# More on Tables

## Notes

Frequently, you need to say more about a column than is appropriate in the few-phrase description. In catalog descriptions and VizieR, such situations are handled using notes, and DaCHS follows suit.

The notes themselves are kept in meta elements belonging to tables. Since the notes tend to be markup-heavy, their default format is reStructuredText. When entering notes in RDs, there is an attribute `tag` on these meta items:

```
<table id="demo">
  ...
  <meta name="note" tag="1">
    The meaning of the flag is as follows:

    =====  ==========
    value  meaning
    =====  ==========
    1      value is 2
    2      value is 1
    =====  ==========
 </meta>

 <meta name="note" tag="2">
 ...
</table>
```

To assoicate a column with a note, use the column's note attribute:

```
<colum name="crazyflag" type="smallint" ... note="1"/>
```

As tag, you may use basically any string, but it's a good idea to keep it to numbers or at least characters not requiring URL encoding.

The notes will exposed in HTML table heads, table and service descriptions, etc. If you need to link to one, there is the built-in tablenote renderer that takes the table and the note from its query path. The most convenient way to is it is through the built-in vanity name tablenote, where you would access the note above using a URL like `http://your.server/tablenote/demoschema.demo/1`.

## STC

As soon as you have coordinates, you will want to declare coordinate metadata on them, i.e., reference frames, roles played by tables (x is the derivative of y, and x1 is a galactic latitude, etc). In VO lingo, this is known as declaring "space-time coordinates" or STC for short.

DaCHS uses a language called STC-S to do this. The STC-S definition currently only exists as a note and is both a bit terse and not quite as rigorous as one would wish, but the good news is that you will get by with but a few features most of the time.

STC is defined in children of table elements, with references to table columns in quoted strings:

```
<table id="withcoo">
  <stc>
    Position ICRS "ra" "dec" Error "e_ra" "e_dec"
  </stc>
  <stc>
    Position FK4 J1950.0 "ra_orig" "dec_orig"
  </stc>

  <column name="ra" unit=...
  <column name="dec" ...
  ...
</table>
```

You do not need to change anything in the column definitions themselves, since the machinery will resolve your column references. If you refer to non-existing columns, RD parse errors will be thrown.

# More on Services

## Custom Templates

Within the data center, most pages are generated from templates; these are written in XHTML (well-formed XML is important, DaCHS itself does not care avout valid XHTML, though) with stan/nevow markup. Please bug us to provide more documentation on this.

The pages the form renderer on services displays are generated from such templates, too. To effect special effects, you may want to override them (though in general, it is a much better idea to work within the standard template since that will give your service all kind of automatic updates and would make, e.g., changes much easier if your institution undergoes the yearly reorganization).

You can retrieve the default response template as something to start from by saying:

```
gavo admin dumpDF templates/defaultsresponse.html
```

To obtain the plainest output conceivable, try:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html mlns="http://www.w3.org/1999/xhtml"
    xmlns:n="http://nevow.com/ns/nevow/0.1">
<head>
  <title>No title</title>
</head>
<body>
   <div class="result" n:render="ifdata" n:data="result">
     <div class="result">
      <n:invisible n:render="resulttable"/>
    </div>
  </div>
  <n:invisible n:render="form genForm"/>
</body>
</html>
```

Save this to a file within the resource directory, let's say "res/plain.html". Then, say:

```
<template key="form">res/plain.html</template>
```

in your service; this should do give you a minimally decorated page.

Of course, this will display a severely degraded page. To get at least the standard style sheet and the standard javascript, say:

```
<head n:render="commonhead">
```

instead of the plain head.

### Values Metadata

For input parameters, it's usually a good idea to indicate to users what the valid range for them might be. When you give `values` elements in your tables' columns, DaCHS will have placeholders in floating point and integer fields in web forms and appropriate VOTable values elements in the metadata responses for DAL protocols. So, essentially:

```
<column ...>
  <values min="-1.0" max="2.0/>
</column>
```

would be enough. However, maintaining these min and max values is a bit of a chore. On the other hand, obtaining them from the database can be costly, and hence it shouldn't be done at each server start. Hence, DaCHS has a subcommand `limits` that takes an RD or table id as the command line argument and replaces the *existing* min/max values in the referenced thing with data obtained from the database. So, the recommended way to do these things is to stereotypically add `<values min="0" max="0"/>` in columns that generate query parameters and then run either of:

```
gavo limits arihip/q       # update all tables present
gavo limits arihip/q#main  # only update the single table
```

after an import giving new data.

## More on Cores

### CondDescs

dbCores and cores derived from them take most of their power from condition descriptors or CondDescs. These combine inputKeys, which are basically column objects with some additional presentation-related information, with code generating SQL conditions.

A condDesc can contain zero or more input keys (though having zero input keys makes no sense for user-defined condDescs since they would never "fire"). Having more than one input key is useful when input quantities can only be interpreted when present as a group. An example is the standard cone search, where you need both a position and a search radius.

### Automatic and manual control

However, most condDescs correspond to one input key, and the input key is mostly derived from a table column. This is effected by the standard idiom:

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

where somecol is a column in the table queried by the core. This construct will cause the an input key to be built from somecol. While doing this, the type will be mapped automatically. The primary rules are:

- Numeric types will get mapped to numeric vizier-like expressions
- Datetimes will get mapped to date vizier-like expressions
- text and chars will get mapped to string vizier-like expressions
- enumerated values (i.e., columns with value elements giving options) will not become vizier-like expressions but input keys that yield selection widgets.

To have more control (e.g., if you do not want to allow vizier-like expressions, give the input key yourself):

```
<condDesc>
  <inputKey original="primaryId" required="False"/>
</condDesc>
```

(which would make a column required in the table optional in the query), or:

```
<condDesc>
  <inputKey name="specType" tablehead="Spectral Type"
    type="text" description="Spectral type of the target object">
</condDesc>
```

(which creates an input key matching everything literally), or even:

```
<condDesc>
  <inputKey name="color" type="text" required="True">
    <values multiOk="True">
      <option title="Red">R</option>
      <option title="Green">G</option>
      <option title="Blue">B</option>
    </values>
  </inputKey>
</condDesc>
```

-- if the input key is required, queries not giving it will be rejected. The title attribute on option gives the label of an option in the HTML input widget; if it's missing, a string representation of the value will be used.

In all those cases, the SQL generated from the condDesc is a conjunction of the input key's individual SQL expressions. Those, in turn, are simply comparisons for equality for plain types and more or less arbitrary expressions for vizier expression types.

Incidentally, two properties on inputKeys are defined to only show inputs for certain renderers, viz., `onlyForRenderer` and `notForRenderer`. Both have single strings as values. This is intended mainly for cases like SIAP and SCS where there are "human-oriented" versions of the input fields available. The built-in SCS and SIAP conditions already to that, so you can give both scs and humanSCS conditions in a core. Here is how you would define an input key that is only used for the form renderer:

```
<inputKey original="color">
  <property name="onlyForRenderer">form</property>
</inputKey>
```

### Phrase makers

For complete control over what SQL is generated, condDescs may contain code called a phrase maker. This, again, is a procedure application, quite like with rowmaker procs, except that the signature of condDesc code is different.

Phrase maker code has the following names available:

- inputKeys -- the list of input keys for the parent CondDesc
- inPars -- a dictionary mapping inputKey names to the values provided by the user
- outPars -- a dictionary that is later used as the parameter dictionary to the query.

The code should amend the outPars dictionary with the keys mentioned in the the conditions. The conditions themselves are yielded. So, a very simple condDesc with generated SQL could look like this:

```
<condDesc> <!-- don't do it like this, see below -->
  <inputKey name="val"/>
  <phraseMaker>
    <code>
      outPars["xxyy"] = "x"*inPars.get("val", 20)
      yield "someColumn=%(xxyy)s"
    </code>
  </phraseMaker>
</condDesc>
```

However, using fixed names in outPars is not recommended, if only because condDescs could be used multiple times. The recommended way uses the vizierexprs.getSQLKey function. It takes a name, a value, and the outPars dictionary. It will return a key unique to the query in question and enter the value into the outPars dictionary under that key. While that sounds complicated, it is actually rather harmless, as shown in the following real-world example that lets users input date, time and an interval in split-up form (e.g., when you cannot hope anyone will try to write the equivalent vizier-like expressions):

```
<condDesc>
  <inputKey name="date" tablehead="Date" type="date"
    multiplicity="single"
    required="True"/>
  <inputKey name="time" tablehead="Time (UTC)" type="time"
    multiplicity="single"
    required="True"/>
  <inputKey name="within" required="True" type="integer"
    multiplicity="single"
    tablehead="plus/minus" unit="minutes"
    description="Give measurements within this many minutes
     of your chosen date and time.  The sampling rate is 20 minutes">
    <values default="11"/>
  </inputKey>
  <phraseMaker>
    <code>
      baseTS = datetime.datetime.combine(inPars["date"], inPars["time"])
      dt = datetime.timedelta(minutes=inPars["within"])
      yield "date BETWEEN %%(%s)s AND %%(%s)s"%(
        vizierexprs.getSQLKey("date", baseTS-dt, outPars),
        vizierexprs.getSQLKey("date", baseTS+dt, outPars))
    </code>
  </phraseMaker>
</condDesc>
```

## More on Metadata

In general, most metadata for services and resources rather closely follows what's defined in Resource Metadata for the Virtual Observatory; see also the Reference Manual on RMI-style metadata.

### Authors

VOResource wants split-up author specifications, and for a good reason. However, for longer author lists, these are a pain to write down (see also RMI-Style Metadata in the reference).

As a shortcut, DaCHS lets you specify authors in the simple `creator` metadata as semicolon-separated names. Unless you want to set logos or similar, the recommended way to declare authors is:

```
<meta name="creator">Author1, S.; Author-Two, J.C.</meta>
```

### Coverage

One tricky spot is coverage, i.e., the parts of the STC space covered by what's in the resource. In general, you will define coverage more or less like this:

```
<meta name="coverage">
  <meta name="profile">AllSky ICRS</meta>
  <meta name="waveband">Optical</meta>
</meta>
```

The easy part is the waveband. Values here are from a fixed set of strings, viz., Radio, Millimeter, Infrared, Optical, UV, EUV, X-ray, Gamma-ray; capitalization is important, and you may give multiple elements (the software doesn't enforce this selection, but your registry documents will become invalid if you use anything else).

The coverage.profile meta item has STC-S strings as values. See the STC-S Note as well as the STC library documentation for more information on the STC-S understood by DaCHS. In principle, you can get fancy here; for example, you could write:

```
<meta name="coverage.profile">
  TimeInterval TT BARYCENTER 1999-10-01T20:30:00 1999-10-02T20:30:10
    unit s Error 10 Resolution 1 2
  Circle FK5 J1980.0 GEOCENTER 0.13 0.45 0.03 unit rad
    PixSize 0.0001 0.0001
  SpectralInterval HELIOCENTER 2000 6000 unit Angstrom Error 1
  RedshiftInterval TOPOCENTER VELOCITY RELATIVISTIC -10 10 unit km/s
</meta>
```

However, the registries probably evaluate not very much of this information as yet, and you most certainly should try to give positions in ICRS.

### (Content) Type

Values for `type` come from a controlled vocabulary that includes Other, Archive, Bibliography, Catalog, Journal, Library, Simulation, Survey, Transformation, Education, Outreach, EPOResource, Animation, Artwork, Background, BasicData, Historical, Photographic, Press, Organisation, Project, Registry.

Specifying the content type is optional, and you can repeat the meta element as often as you need to.

If you are unsure what these mean, see Resource Metadata for the Virtual Observatory, section 3.3.

### Copyright

Within the astronomical community, licensing issues have traditionally played a minor role – if you referenced properly, using data from other people was not only ok, it was encouraged. We should keep it that way, even in the days of easy reproducability. Still, formal statements about how your data may be used may be useful. These statements are called licenses.

RMI has the copyright meta for this purpose. Right now, DaCHS doesn't do much with this information; it includes it in VOResource records, and the default response template shows it below the query form. We recommend either specifying something like "The data is in the public domain" or, if you want to use something that's more in line with scientific habits, the Creative Commons Attribution ("CC-BY"). To support this, DaCHS includes a macro that can be used in meta elements that are direct children of the resource element. Use it like this:

```
<resource...
  <meta name="copyright" format="rst">
    \RSTccby{Image metadata}

    Usage conditions for individual images could differ.  See the
    COPYING FITS header.
  </meta>
</resource>
```

The advantage of using the macro is that you get a nice image, and in the future we may expand this to a formal, machine-readable declaration.

## Active Tags

Active "tags" delemit elements within resource descriptor XML that do not directly contribute to result tree. Their typical use is to "record" event sequences and replay them later. Much of this is used internally. However, some applications of active tags are interesting for RD writers, too. Active tags always have names in all upper-case.

### LOOP

Loop lets you create multiple elements by rules. The simplest way to use it is by giving a space-separated list of "items":

```
<LOOP listItems="a b c">
  <events>
    <column name="\item"/>
  </events>
</LOOP>
```

The `events` child of the `LOOP` element creates a list of events (think "begin column element", "value for name attribute", "end column element"). These events are then replayed to the parser for each item in the LOOP's `listItems` attribute. Each occurrence of the `\\item` macro is replaced with the current item. So, in the resulting RD tree, the fragment above will have the same result as:

```
<column name="a"/>
<column name="b"/>
<column name="c"/>
```

Sometimes the list items are used in multiple places in the same document. To avoid having to maintain multiple lists, you can define macros using RD's `macDef` element; this could look like this:

```
<resource schema="foo">
  <macDef name="bands">U B V R</macDef>
  <table id="mags">
    <LOOP listItems="\bands">
      <events>
        <column name="mag\item"/>
      </events>
    </LOOP>
  </table>
  <rowmaker id="build_mags">
    <LOOP listItems="\bands"/>
      <events>
        <map dest="mag\item">parseFromString(MAG_\item)</map>
```

```
      </events>
    </LOOP>
  </rowmaker>
....
</resource>
```

Note that macro names must be at least two characters long.

Frequently, the loop variable should not just take on a single string. For such cases you can feed in tuples. The most convenient way to do this is `csvItems`. The content of this element is a string literal containing comma separated values *with labels*, i.e., parsable with python's csv.DictReader. In your events, you can then refer to the labeled items using macros. For example:

```
<resource schema="foo">
  <macDef name="bands">
    band,source
    U 10-12
    V 13-16
  </macDef>
  <table id="mags">
    <LOOP csvItems="\bands">
      <column name="mag\band"/>
    </LOOP>
  </table>
  <data id="magscontent">
    <columnGrammar>
      <LOOP csvItems="\bands">
        <col key="mag\band">\source</col>
      </LOOP>
    </columnGrammar>
    <make table="mags"/>
  </data>
</resource>
```

TODO: EDIT actives?

## Some Words on Times

Among the messier data types in astronomical databases are dates and times – they come in lots of crazy input formats, they can be represented in lots of different ways in the database, they are expected in lots of crazy output formats, plus there's a host of exciting metadata on them, including time scales and reference positions.

With DaCHS, we recommend one of the following ways of storing dates and times (written as attributes of column):

- `type="double precision" xtype="mjd" unit="d"` – a modified julian date
- `type="double precision" unit="d"` – a julian date
- `type="double precision" unit="s"` – a unix timestamp
- `type="double precision" unit="yr"` – a Julian year with fractions
- `type="timestamp"` – a postgresql timestamp

All other things being equal, we recommend using mjds; most VO data models and protocols employ them, and they are fairly easy to query against. In HTML forms, they are easily displayed as human-readable datetimes by using an `displayHint="type=humanDate"` (which you can do for the others, too, of course).

The Julian years are a good choice, too, and they are immediately human-readable to some extent. They are certainly the representation of choice for epochs and equinoxes. Note that the storage of Bessel years is strongly discouraged. Use the `bYearToDateTime` function to transform them to datetime instances which you can then map to any recommended representation.

While timestamps might sound like a good idea in that they are the proper native type to manipulate dates and times with, they usually are a bad choice. The main reason is that in ADQL there is basically no support of timestamps at all, which makes any manipulation of them in ADQL queries virtually impossible. If you're sure your table will never turn up on a TAP service, that doesn't hurt much, but can you be sure?

All this didn't mention any UCDs or utypes that may apply. UCDs should not, in general, depend on the time format chosen; all of the above could be used for quantities like `time.creation`, `time.end`, `time.epoch`, `time.equinox`, `time.processing`, `time.release`, `time.start`, and more. The SIAP version 1 protocol made a funky exception there, defining an `VOX:Image_MJDateObs` UCD; everything about that UCD is horrible, and it is generally accepted in the VO that this was an error.

Finally, there is advanced metadata, in particular time zones, time scales (i.e., how does the the time pass) and reference positions (i.e., where is the clock positioned).

Time zones are not supported at all in the VO. All times are for the Greenwhich meridian (i.e., they should be close to UTC).

The time scales are important on the level of seconds; they include TAI (the time scale defined by a bunch of atomic clocks, UTC (TAI with leap seconds, basically our everyday time), UT, UT0, UT1, UT2 (several sorts of true times in Greenwich), and TT (Terrestial Time, a time scale linked to the TAI and

used quite a bit in Astronomy). More of that on the fairly readable http://stjarnhimlen.se/comp/time.html.

The reference positions are currently relevant on a level of milliseconds or below; they need to be declared for high precision work since a clock in the barycenter of the solar system will (evaporate but before that) run slower than one on Pluto due to relativistic effects of various sorts. Common reference positions would be TOPOCENTER (the observatory), GEOCENTER (the center of the Earth), BARYCENTER (the barycenter of the solar system) and UNKNOWN (the default, which you should keep unless you are sure; it doesn't matter anyhow for most applications).

To declare those, you must include a time phrase in your STC declaration in your table. Typically, this could look like this:

```
<table id="foo">
  <stc>TimeInterval TT "timeStart" "timeEnd" Time "dateObs"</stc>

  <column name="timeStart" ucd="time.start" unit="d"/>
  <column name="timeEnd" ucd="time.end" unit="d"/>
  <column name="dateObs" ucd="time.epoch;obs" unit="yr"/>
  ...
```

(descriptions and everything else left out for clarity; in particular, for times using double precision almost always is a good idea).

## Publishing DAL Services

DAL is VO-speak for "Data Access Layer", the standard protocols the VO uses to allow remote querying of data. To support such a protocol, you usually need to arrange things in three places:

- The table queried needs a certain set of columns
- The core must support certain input and output fields
- The renderer must exhibit specified behaviour as regards, e.g., the formatting of error messages, and it may require protocol-specific metadata

This section discusses the individual protocols in turn.

## SCS

SCS, the simple cone search, is the simplest IVOA DAL protocol -- it is just HTTP with RA, DEC, and SR parameters, a slightly constrained VOTable response, plus a special way to encode errors (in a way somewhat different from what has been specified for later DAL protocols).

The service discussed in Building a Catalog Service is a combined SCS/form service. This section just briefly recapitulates what was discussed there. For a quick start, just follow the tutorial above.

### Tables

In principle, SCS can expose any table that has a exactly one column each with the UCDs `pos.eq.ra;meta.main`, `pos.eq.dec;meta.main`, and `meta.id;meta.main`, where the coordinates must be real or double precision, and the id must be either some integral type or text; the standard requires the id to be text, but the renderer will automatically convert integral types. The main query is then ran against the position specified in this way.

You almost always want to have a spatial index on these columns. To do that, use the `//scs#q3cindex` mixin on the tables, like this:

```
<table id="forSCS" onDisk="true" mixin="//scs#q3cindex"> ...
```

Finally, note that to have a valid SCS service, you must make sure the output table always contains the three required columns (as defined by the UCDs given above. To ensure that, these columns' `verbLevel` attribute must be 10 or less (we advise to have it at 1).

### Cores

The SCS core simply is a dbCore. You must include the SCS condDesc, like this:

```
<dbCore queriedTable="main">
  <condDesc original="//scs#protoInput"/>
</dbCore>
```

There is an alternative condDesc more suitable for humans. They can be used in parallel. The form renderer will then use the human-oriented one, the DAL renderer the protocol one. You'll get this by writing:

```
<dbCore id="xlcore" queriedTable="main">
  <condDesc original="//scs#humanInput"/>
  <condDesc original="//scs#protoInput"/>
</dbCore>
```

Although not required by SCS, we recommend to also include a MAXREC argument that lets people change the match limit in the SCS service (for the web service, the database widget already provides this functionality). A usable definition for it is given in the SCS RD in a STREAM with the id coreDescs, together with the two condDescs above. So, here's the recommended way to build a bare-bone SCS service:

```
<dbCore id="xlcore" queriedTable="main">
  <FEED source="//scs#coreDescs"/>
</dbCore>
```

SCS allows more query parameters; you can usually use condDesc's buildFrom attribute to directly make one from an input column. If you want to add a larger number of them, you would use an active tag:

```
<dbCore id="xlcore" queriedTable="main">
  <condDesc original="//scs#humanInput"/>
  <condDesc original="//scs#protoInput"/>
  <LOOP listItems="ipix bmag rmag jmag pmra pmde">
    <events>
      <condDesc buildFrom="\item"/>
    </events>
  </LOOP>
</dbCore>
```

Note that most current SCS clients are not good at discovering them, since for SCS this requires going through the registry. In TOPCAT, for example, users would have to manually edit the cone search URL.

**Service**

To expose that core through a service, just allow the scs.xml renderer on it. As the core is built, you can have a web-based form interface for free:

```
<service id="cone" allowed="scs.xml,form">
  <meta name="title">Nice Catalog Cone Search</meta>
  <meta name="shortName">NC Cone</meta>
  <meta name="testQuery.ra">10</meta>
  <meta name="testQuery.dec">10</meta>
  <meta name="testQuery.sr">0.01</meta>
  <dbCore id="xlcore" queriedTable="main">
```

```
    <FEED source="//scs#coreDescs"/>
    <LOOP listItems="ipix bmag rmag jmag pmra pmde">
      <events>
        <condDesc buildFrom="\item"/>
      </events>
    </LOOP>
  </dbCore>
</service>
```

The meta information given is used when generating registration records. In particular, you should make sure that a query with the given ra, dec, and sr actually returns some data.

## SIAP

DaCHS' SIAP implemention right now assumes you are publishing FITS files with WCS headers. Other arrangements are of course possible, but you'd have to write an equivalent of the `//siap#computePGS` procDef yourself.

### Quick Start

Check out a sample resource directory:

```
svn co http://svn.ari.uni-heidelberg.de/svn/gavo/hdinputs/emi
cd emi
mkdir data
```

Now fetch some files to populate the data directory so you have something to import:

```
cd data
SRCURL=http://dc.g-vo.org/emi/q/s/siap.xml
curl -s $SRCURL"?POS=163.3,57.8&SIZE=20,20&MAXREC=5&weighting=uniform" \
  | tr '<TD>' '\n' \
  | grep "^http://" \
  | sed -e 's/<[^>]*>//g' \
  | xargs -n1 curl -sO
```

(no, this is not in general the way to operate SIAP services; use a proper client for real work, and we didn't show you this).

Run the import:

```
cd ..
gavo imp q
```

Now start the server as necessary (see above), and start TOPCAT and Aladin. In TOPCAT, open VO/SIA Query, enter your new service's access URL (it's `http://localhost:8080/emi/q/s/siap.xml` unless you did something cunning and should know better yourself) under "SIA URL" pretty far down in the dialog.

Then have "Lockman Hole" as Object Name and resolve it, or manually enter 161.25 and 58.0 as RA and Dec, respectively, and have 2 as Angular Size. Send off the request. You'll get back a table that you can send to Aladin (Interop/Send to/Aladin), which will respond by presenting a load dialog. Doubleclick and load as you like (yes, the images look a bit like static noise; that's all right here; but do combine these images with, say, DSS colored optical imagery and marvel at the wonders of modern VLBI interferometry).

Indicentally, we made the detour through TOPCAT since there's no nice UI to query non-registred SIAP services in Aladin.

### Tables

SIAP-capable tables should mix in `//siap#pgs` (the older `//siap#bbox` is deprecated; you could still use it if for some reason you have no pgSphere). This mixin provides all the columns necessary for valid SIAP responses.

So, in the simplest case, a table that's going to be published through SIAP would look like this:

```
<table id="images" onDisk="True" mixin="//siap#pgs"/>
```

(of course, you can add more columns if you need them, and you might need metadata and all that, but this is all it really takes).

In the case of the Lockman Hole RD, things are a bit more verbose:

```
<table id="main" onDisk="True">
  <mixin>//siap#pgs</mixin>
  <mixin
    calibLevel="3"
    collectionName="'VLBA LH sources'"
    facilityName="'VLBA'"
    oUCD="'phot.flux.density;em:radio.750-1500MHz;phys.polarisation.Stokes.I'"
    polStates="'/I/'"
    targetName="object"
    tResolution="5000000"
    targetClass="'AGN'"
  >//obscore#publishSIAP</mixin>

  <column name="object" type="text"
    ucd="meta.id"
```

```
    tablehead="Object"
    description="Source name as in
      2009MNRAS.397..281I (VizieR J/MNRAS/397/281)"
    verbLevel="1"/>
  <column name="obsra"
    ucd="pos.eq.ra" unit="deg"
    description="Antenna pointing, RA"
    verbLevel="18"/>
  <column name="obsdec"
    ucd="pos.eq.dec" unit="deg"
    description="Antenna pointing, Dec"
    verbLevel="18"/>
  <column name="weighting" type="text"
    ucd="meta.code"
    description="Natural or uniform, according to weighting method."
    verbLevel="18">
    <values><option>natural</option><option>uniform</option></values>
  </column>
</table>
```

More on the obscore mixin below. otherwise, you see that additional, non-siap columns are simply added as usual. Anything with a verbLevel lower or equal 20 will be included in standard SIAP replies.

To fill such tables, the normal `//products#define` rowfilter is necessary, and there are two procDefs from `//siap` that come in handy:

```
<data id="import_main">
  <sources recurse="True">
    <pattern>data/*.fits</pattern>
  </sources>
  <fitsProdGrammar qnd="True">
    <maxHeaderBlocks>80</maxHeaderBlocks>
    <mapKeys>
      <map key="object">OBJECT</map>
      <map key="obsdec">OBSDEC</map>
      <map key="obsra">OBSRA</map>
    </mapKeys>
    <rowfilter procDef="//products#define">
      <bind key="table">"emi.main"</bind>
    </rowfilter>
  </fitsProdGrammar>

  <make table="main" >
    <rowmaker id="gen_rmk" idmaps="object, obsra, obsdec">
      <apply procDef="//siap#computePGS"/>
      <apply procDef="//siap#setMeta">
        <bind name="bandpassLo">0.207</bind>
        <bind name="bandpassHi">0.228</bind>
        <bind name="bandpassId">"1.4 GHz"</bind>
        <bind name="bandpassRefval">0.214</bind>
        <bind name="bandpassUnit">"m"</bind>
```

```
        <!-- since the images are fairly complex mosaics, there's no way we
        can have sensible dates; this one here "plays a special role
        in the calibration" (Middelberg) -->
        <bind name="dateObs"
          >dateTimeToMJD(datetime.datetime(2010, 7, 4))</bind>
        <bind name="instrument">@INSTRUME</bind>
        <bind name="title">"VLBA 1.4 GHz "+@object</bind>
      </apply>

      <apply name="fixObjectName">
        <setup>
          <code>
            import csv
            with open(rd.getAbsPath(
                "res/namemap.csv")) as f:
              nameMap = dict(csv.reader(f))
          </code>
        </setup>
        <code>
          @object = nameMap[@object]
        </code>
      </apply>

      <map key="weighting">\inputRelativePath.split("_")[-1][:-5]</map>
    </rowmaker>
  </make>
</data>
```

This does, step by step:

- The `sources` element is as always – with image collections, the `recurse` attribute usually comes in particularly handy.
- When ingesting images, you will (at least for a while, still) almost always read from FITS images, i.e., FITS primary headers. A `fitsProdGrammar` delivers the key-value-pairs from a header as a rawdict.
- The `qnd` attribute of the grammar is recommended. It makes some (weak) assumptions to yield significant speedups with large images, just so long as you can make do with the primary header.
- The `fitsProdGrammar` will map keys with hyphens to names with underscores, which allows for smoother action with them in rowmakers. The `mapKey`‘ element can produce additional mappings; in this case, we abuse it a bit to let us have idmaps (rather than simplemaps) in the rowmaker. And, actually, to illustrate the feature, as this data does not need that facility, really.
- The grammar further needs a rowfilter. The products#define rowfilter lets you add keys on owners and embargo in case you want password protection

for images, but most importantly it defines what table the data is destined for. This is crucial information, and if you ever get it wrong, you need to manually connect to the database and issue a command like `DELETE FROM products WHERE sourcetable='<your wrong table>'`. So, always bind table. Make sure to include the quotes, this is supposed to be a valid python expression yielding a string.

- You then need to define a rowmaker that must apply two procs. For one, you need //siap#computePGS (if you mixed in `//siap#pgsSIAP`). No bindings are required here.
- The second proc application required is //siap#setMeta . Try to give all its keys somewhat sensible values, you will make your users' lives much easier.
- Typically, many values coming in the FITS headers will be messy and fouled up. You'll spend some quality time fixing these values in the typical case. Here, we translate somewhat broken object names using a simple mapping file that was provided by the author. In other circumstances there's the procs#mapValue procApp helping you.
- As is usual in procApps, you can access the embedding RD as `rd`. Here, we use that to let DaCHS find the input file independently of where the program was started.

Warning: Do *not* use idmaps="*" with SIAP, since the auto-generated mappings will clobber the work of the xSIAP procs.

### Cores

There are two cores you may want for SIAP services:

- dbCore, to which you add the necessary condDescs manually as below, for "normal" SIAP services.
- siapCutoutCore, which speaks SIAP but returns cutouts rather than full images; the size of these cutouts is determined by the SIZE argument (i.e., the region of interest).

To furnish these cores with the parameters required by the standard, use the //siap#protoInput condDesc. If you want to re-use the core for a form-based service, use the //siap#humanInput condDesc as well. Both are written in a way that they'll sense if they run under a SIAP renderer or not.

So, a basic core with a couple of additional fields would look like this:

```
<dbCore id="query_images" queriedTable="main">
  <condDesc original="//siap#protoInput"/>
  <condDesc original="//siap#humanInput"/>
  <condDesc buildFrom="dateObs"/>
  <condDesc buildFrom="bandpassId" />
  <condDesc>
    <inputKey name="object" type="text"
        tablehead="Target Object"
        description="Object being observed, Simbad-resolvable form"
        ucd="meta.name" verbLevel="5" required="True">
        <values fromdb="object FROM lensunion.main"/>
    </inputKey>
  </condDesc>
</dbCore>
```

### Service

If you wrote the core to work for both SIAP and form as described above, there's little more to say except you'll want to use the `siap.xml` renderer, and you need some additional metadata for VO registration. The latter is described in the siap.xml reference.

With this, the service definition would look like this:

```
<service id="im" allowed="form,siap.xml" core="query_image">
  <meta name="shortName">sample images</meta>
  <meta name="title">Sample Image Archive</meta>
  <meta name="sia.type">Pointed</meta>

  <meta name="testQuery.pos.ra">230.444</meta>
  <meta name="testQuery.pos.dec">52.929</meta>
  <meta name="testQuery.size.ra">0.1</meta>
  <meta name="testQuery.size.dec">0.1</meta>

  <publish render="siap.xml" sets="ivo_managed"/>
  <publish render="form" sets="local,ivo_managed"/>
</service>
```

(where again you can just write the above core inline rather than referencing it; that's the style we usually recommend).

## SSAP

### Tables

SSAP tables come in essentially two flavours: they can be "homogeneous" data collections, i.e., tables for which every data set was generated by the

same instrument, code, or similar. Those mix in `//ssap#hcd`. Alternatively, the datasets can come from different sources, in which case the table would have to mix in `//ssap#mixc`. The following text mainly talks about hcd, but mixc isn't much different from an operator point of view.

This mixin has lots of parameters that define the instrument; see the SSAP HCD mixin in the ref doc.

For example, you could say:

```
<table id="data" onDisk="true">
  <mixin
    instrument="HLT Coude"
    fluxCalibration="RELATIVE"
  >//ssap#hcd</mixin>
</table>
```

Note that despite their name, the timeSI, fluxSI, and spectralSI items in the mixin do *not* contain actual unit strings. Instead, they are intended to contain conversion factors in a custom syntax called "Osuna-Salgado convention". We recommend to not set these fields and instead provide useful VOUnit-compliant units where appropriate.

To fill such a table, it is recommended to use the products#define rowfilter and the ssap#setMeta rowmaker apply. This could look like this:

```
<data id="content">

  <fitsProdGrammar>
    <rowfilter procDef="//products#define">
      <bind name="table">"\schema.data"</bind>
    </rowfilter>
  </fitsProdGrammar>

  <make table="data">
    <rowmaker idmaps="ssa_*">
      <apply procDef="//ssap#setMeta">
        <bind name="dstitle">@FILENAME</bind>
        <bind name="pubDID">"ivo://org.gavo.dc/ccd700/q#"+@FILENAME</bind>
       </apply>
    </rowmaker>
  </make>
</data>
```

Caution: In the ssa table, we force the spectral axis to be a wavelength in meters. You must convert all values manually if necessary. For the spectra themselves you could use different units, but in our experience that's more confusing than helpful.

In contrast to images where delivering FITS is likely all you need, there's a plethora of formats spectra are delivered in. To help a bit, you should make sure one of the formats you offer are VOTables conforming to the spectral data model (see Making SDM Tables in the reference documentation). If you want to deliver the "native" format as well, you'll have to have two rows for each spectrum. The standard way to achieve that is through a rowmaker in the grammar importing the spectra, like this:

```
<rowfilter name="generateFormatLinks">
  <code>
    baseAccref = os.path.splitext(row["prodtblPath"])[0]
    row["prodtblAccref"] = baseAccref+".txt"
    row["prodtblMime"] = "text/plain"
    # this is the file as delivered from upstream
    yield row
    row["prodtblAccref"] = baseAccref+".vot"
    row["prodtblPath"] = "dcc://\rdIdDotted/mksdm?"+baseAccref+".txt"
    row["prodtblMime"] = "application/x-votable+xml"
    # this is our processed SDM VOTable
    yield row
  </code>
</rowfilter>
```

SSAP's FORMAT parameter lets clients select what they want. The way the default FORMAT argument works, only application/x-votable+xml records are considered compliant.

FITS files with spectra are a nasty chapter. Most of the FITS spectra out there currently are basically 1D images. Use an image/fits MIME type for those; application/fits is reserved for FITS binary tables conforming to the spectral data model; chances are you'll have to build those yourself.

### Cores

Use the ssapCore for SSAP services. You must manually feed in the condition descriptors for the SSAP parameters. For homogeneous data collections, this is:

```
<ssapCore queriedTable="newdata">
  <FEED source="//ssap#hcd_condDescs"/>
</ssapCore>
```

The `hcd_condDescs` includes condition descriptors for all mandatory and optional parameters meaningful in the case of homogeneous data collections (i.e., excluding those that match against constant values).

Some of them may not be relevant to your service because your table never has values for them. For example, theoretical spectra will typically not give information on positions. The SSAP spec says that such a service should ignore POS rather than returning the empty set.

If you think you must ignore certain conditions, you can use the PRUNE active tag. This looks like this:

```
<ssapCore queriedTable="newdata">
  <FEED source="//ssap#hcd_condDescs">
    <PRUNE id="coneCond"/>
    <PRUNE id="bandCond"/>
  </FEED>
</ssapCore>
```

Do not do this just because you don't have position information -- this would mean that you would dump your complete archive for (typical) queries with a position, and that is neither required by the spec (even if you might think so at first reading) nor desirable.

Here is a table of parameter names and ids; you can always check them in `$gavo_installed/resources/inputs/__system__/ssap.rd`.

| Parameter name | condDesc id |
|---|---|
| POS, SIZE | coneCond |
| BAND | bandCond |
| TIME | timeCond |

For APERTURE, SNR, REDSHIFT, TARGETNAME, TARGETCLASS, PUBDID, CREATORDID, and MTIME, the condDesc id simply is `<keyname>_cond`, e.g., `APERTURE_cond`.

To have custom parameters, simply add condDesc elements as usual:

```
<ssapCore queriedTable="newdata">
  <FEED source="//ssap#hcd_condDescs"/>
  <condDesc buildFrom="t_eff"/>
</ssapCore>
```

For SSAP cores, `buildFrom` will enable "PQL"-like query syntax such that users can post arguments like `20000/30000,35000` to `t_eff`.

### Service

To expose SSAP services, use the ssap.xml renderer. The metadata keys required for registration of these are documented in the reference manual. A complete declaration of a published SSAP service would then look like this:

```
<service id="ssa" allowed="form,ssap.xml">
  <meta name="shortName">mydata SSAP</meta>
  <meta name="ssap.dataSource">theory</meta>
  <meta name="ssap.creationType">archival</meta>
  <meta name="ssap.testQuery">MAXREC=1</meta>

  <publish render="ssap.xml" sets="ivo_managed"/>

  <ssapCore queriedTable="data">
    <FEED source="//ssap#hcd_condDescs"/>
    <condDesc buildFrom="t_eff"/>
    <condDesc buildFrom="log_g"/>
  </ssapCore>
</service>
```

This service will expose all standard SSAP query parameters, and additionally condDescs built from the `t_eff` and `log_g` columns in the source table (see above).

Incidentally, in web versions of such services, you may want to have specview-based "quick-view" links based on the `run` system rd that exposes the specview template. Here's an example of an `outputTable` (that would reside in the service element):

```
<outputTable>
  <outputField original="accref">
    <formatter><![CDATA[
      res[T.a(href=makeProductLink(data))[
        "[Spectrum as VOTable]"]]
      res[" ", T.a(href=base.makeAbsoluteURL(
        "__system__/run/specview/fixed?source=%s%%3fdm=sed"%
        urllib.quote(makeProductLink(data))))["[in VOSpec]"]]
      return res
    ]]></formatter>
  </outputField>
  <outputField original="ssa_specstart" displayHint="displayUnit=Å"/>
  <outputField original="ssa_specend" displayHint="displayUnit=Å"/>
</outputTable>
```

Some less cody approach would be welcome, but we'd need to collect some experience what people expect there. Also note that specview is (or possibly was, when you're reading this) very picky in what it accepts as VOTables; in the example, the `dm=sed` parameter is used to instruct DaCHS' SDM-making machinery to come up with a table palatable by current specviews.

### ObsTAP

ObsTAP is basically a single table, ivoa.ObsCore. In DaCHS, this is a view generated from input tables. To include the products within a table, you must use one of the mixins from the //obscore RD and fill out some of the mixin's parameters. There is some documentation on what to put where in the mixin documentation, but frankly, as a publisher, you should have at least passing knowledge of the obscore data model as laid down in Tody et al (2011).

In the simplest case, a SIAP table, you could get by simply adding:

```
mixin="//obscore#publishSIAP"
```

to the table definition's start tag. You do not have to re-import a table to publish it to obscore after the fact – `gavo imp -m <rd id> && gavo imp //obscore create` will include an existing table to the obscore view.

Even for SIAP, you will usually want to add metadata not contained in DaCHS' SIAP meta. To do this, add a mixin element to the table definition's body:

```
<mixin
  sResolution="0.5"
  calibLevel="2"
  >//obscore#publishSIAP</mixin>
```

To find out what parameters the mixin takes, see //obscore#publishSIAP in the reference documentation.

On a table import, the obscore table will automatically be recreated to include the data. If you retrofit ObsCore support to large tables, you can avoid having to re-import everything by adding the mixin clause and then updating the metadata. In that case, you must manually remake the obscore table:

```
gavo imp -m path/to/my/rd
gavo imp //obscore create
```

For SSAP tables, there is an `//obscore#publishSSAPHCD` mixin that works like its SIAP cousin (see the reference documentation of details).

You can also have "pure" Obscore tables which do not build on protocol mixins. A live example is the cubes table in the califa/q2 RD within the GAVO data center. Here's a brief explanation of how this works.

For the non-constant parts of your data, re-use the metadata given in the global obscore table – to make that convenient, tell DaCHS to resolve `original` references in there:

```
<table id="cubes" onDisk="True" namePath="//obscore#ObsCore">
```

`adql="True"` is absent here as the obscore mixin set it. It wouldn't hurt, though.

You will almost always want to have DaCHS manage your products. This works even when all your files are external (i.e., you're entering http URLs in accessURL), so it's a good idea to always mix in products:

```
<mixin>//products#table</mixin>
```

Then, you mix in `//obscore#publish`, which is like the protocol-specific mixins except it doesn't pre-set parameters based on what's already in protocol-specific tables:

```
<mixin
    accessURL="dlurl"
    size="10"
    mime="'application/x-votable+xml;content=datalink'"
    calibLevel="3"
    collectionName="'CALIFA'"
    coverage="s_region"
    dec="s_dec"
    emMax="7e-7"
    emMin="3.7e-7"
    emResPower="4000/red_disp_mean"
    expTime="t_exptime"
    facilityName="'Calar Alto'"
    fov="0.01"
    instrumentName="'PMAS/PPAK at 3.5m Calar Alto'"
    oUCD="'phot.flux;em.opt'"
    productType="'cube'"
    ra="s_ra"
    sResolution="0.0002778"
    title="obs_title"
    tMax="t_min"
    tMin="t_max"
    targetClass="'Galaxy'"
    targetName="target_name"
  >//obscore#publish</mixin>
```

Essentially, what's constant is given in literals, what's variable is given as a column reference. It is a bit unfortunate that you have to enter quite a few identity mappings in here, so we might provide a mixin presetting those if this turns out to be a common use case. Tell us if you're annoyed.

You can then add your custom columns (which might be useful if people directly query your table). The central part is copying over obscore columns that are not constant for your data collection. For califa, this looks like this:

```
<LOOP listItems="obs_id obs_title obs_publisher_did
    target_name t_exptime t_min t_max s_region
    t_exptime">
    <events>
      <column original="\item"/>
    </events>
  </LOOP>
```

– you'll obviously have to adapt the `listItems`. To see what column names are available, see the obscore table description.

That's about it for defining the table. To fill the table, just have a normal rowmaker; since the table contains products, don't forget the `//products#define` rowfilter in the grammar.

## Publishing DaCHS-managed tables via TAP

In the simplest form, all you need to do to publish a table through the TAP endpoint is to add an `adql="True"` attribute to the table definition and update the metadata (by saying `gavo imp -m <rd>`).

You should, however, take particular care that there's a useful description of the table, usually as a direct meta on the table. Keep in mind that people will stumble across the table in some sort of registry and need to be able to figure out whether the table contains useful data by that description and the column metadata alone.

The TAP endpoint only exposes rather limited metadata. At least when there is no published service on the table, you may want to just publish the data to the registry, too. This leads to a much richer set of metadata, increasing people's chances to able to locate the data.

To publish a nonservice (usually a table definition, but you can register data descriptors containing multiple tables, too), use the register Element . For a simple table, just wringing `<register/>` is enough, since the set name defaults to `ivo_managed` and ADQL-accessible tables are automatically related to the TAP services.

When `register` is the child of a data item, you need to manually declare that child tables are TAP-accessible, like this:

```
<data id="collection" auto="false">
  <register services="__system__/tap#run"/>
  <make table="part1"/>
  <make table="part3"/>
</data>
```

When publishing "non-obvious" tables to TAP, it's a good idea to add one or more TAP examples for it. See Writing Examples

### Publishing existing tables via TAP

If you already have a database table and now want to use DaCHS to publish it via TAP, just write an RD as described above, except that the data element is trivial. Here's an example of how that could look like:

```
<resource schema="mydata">
  <meta name="title">My great table</meta>
  <meta name="creationDate">... (more metadata)

  <table id="values" onDisk="True" adql="True">
     <column name="id" type="bigint" unit="" ucd="meta.id;meta.main">
       <description>id of object covered here</description></column>
  </table>

  <data id="import">
    <make table="values"/>
  </data>
</resource>
```

Within the data element you need one make each for each table you want in ADQL; it would cause the tables to be created on a plain `gavo imp`, in the present context, it just says something like "put the table metadata into DaCHS' internal catalogs".

After that, say `gavo imp -m <rd-id>`; make sure you don't forget the `-m`, because without it, `gavo imp` will drop the existing tables if it can, i.e., if gavoadmin has write access to the schema in question, and it should have that for reasons explained in the next paragraph.

This adds the metadata you've given to all kinds of administrative tables DaCHS keeps but does not touch the data. It will also try to fix the permissions of the table such that DaCHS's untrusted user can read it. To let DaCHS manage the permissions, in psql say (assuming standard profiles):

```
GRANT ALL PRIVILEGES ON SCHEMA <your schema> TO gavoadmin
  WITH GRANT OPTION;
GRANT SELECT ON <your schema>.<your table> TO gavoadmin
  WITH GRANT OPTION;
```

If you have local users accessing the table, you should declare them in either the allRoles or readRoles attributes to the table definiton. Maybe even adapting the

profiles in GAVOROOT/etc to match your existing infrastructure could make sense.

Also do not forget that people should have some way to locate your data collection (i.e., the table(s) that you are exposing). If you have sufficient metadata defined – basically as for services –, you can register your data collection. To do this, just add an empty `<register/>` element to either a table definition or, more convenient in multi-table setups, a data element for your data collection. The defaults for register are publication to the VO and, for ADQL-exposed tables, serviced by the TAP service, which is about what you want in this situation.

Here's an example for the case of a multi-table publication:

```
<data id="collection" auto="false">
  <register services="__system__/tap#run"/>
  <make table="part1"/>
  <make table="part3"/>
</data>
```

Don't forget that you need to execute:

```
gavo pub the/rdid
```

to make DaCHS actually publish the table.

## EPN-TAP

EPN-TAP will be a standard for publishing planetary data via TAP; it is still somewhat in development, but we will try to keep DaCHS' interface as stable as reasonable while EPN-TAP evolves. If you want to publish data via EPN-TAP, you will probably want to have a quick look at the EPN-TAP proposed specification (check if there are newer ones).

Data in planetary sciences often comes in "PDS format", which superficially resembles FITS but is quite a bit more sophisticated. Unfortunately, python support for PDS is underwhelming. At least there is PyPDS, which needs to be installed for DaCHS' pdsGrammar to work.

### Quick Start

Install PyPDS if you don't have it anyway:

```
curl -LO https://github.com/RyanBalfanz/PyPDS/archive/master.zip
unzip master.zip
cd PyPDS
python setup.py build
sudo python setup.py install
```

Get the sample data:

```
cd ‘gavo config inputsDir‘
curl -O http://docs.g-vo.org/epntap-example.tar.gz
tar -xvzf epntap-example.tar.gz
cd lutetia
```

Import it and build the previews from the PDS images:

```
gavo imp q
python bin/makePreview.py
```

Start the server as necessary. If you go to your local ADQL endpoint (something like http://localhost:8080/__system__/adql/query/form) and execute queries like:

```
SELECT * FROM lutetia.epn_core
```

there. Note how you get previews when hovering over the links in "Product Key". This is, incidentally, a local extension to EPN-TAP, which is why futher right there's still the EPN-TAP access_reference column that also lets users retrieve the data.

For access through a standard protocol, start TOPCAT, select "VO/TAP Query", and at the bottom of the dialog enter http://localhost:8080/tap (or whatever you configured) in "TAP URL". Hit "Enter Query", wait until the table metadata is in and then again query something like:

```
SELECT * FROM lutetia.epn_core
```

Open the table and play with it. As a little visual treat, in TOPCAT's main window hit "Activation Action", and configure the `preview_url` column under "View URL as Image". Then click on the table rows.

### Tables

In essence, EPN-TAP is just a set of columns. In DaCHS, these are in the //epntap#table mixin. See the reference documentation of details. If your data are images calibrated in some sort of spherical coordinates, you'll likely not have to change anything here. The mixin will already open the table for ADQL querying, and to make that work, it is also declared onDisk. What's not declared is the table name, although it's fixed at epn_core. Hence, the minimal definition of an EPN-TAP table is:

```
<table id="epn_core" mixin="//epntap#table"//>
```

You can of course add further columns as necessary.

The real work is populating the table. Let's first assume you have the data products and use DaCHS to publish them. You will then use a grammar, and as EPN-TAP deals with data products, the grammar must use the rowfilter products#define rowfilter as discussed above in SIAP; the example shows how this can be done while also providing for DaCHS-generated previews.

Filling the table from what comes from the grammar happens through and `idmaps="*"` (which you want as the `apply` doesn't actually map anything into the results and there's too many columns to make enumerating fun) and an application of //epntap#populate . There's little to add to the description of the parameters. You will probably need to refer to the EPN-TAP proposed specification now and then while filling things out. Note again that as always when binding apply parameters, what you're entering are python expressions, so don't forget the quotes where appropriate.

More complex operations should probably go to `var` declarations rather than the `bind` bodies; again, see the example to see how.

If, on the other hand, you already have a database table containing the data as well as a network service pushing out the data products, you can still use the mixin and just create a view as described in Services Over Views . To see what columns you can map, you should still consult the documentation of `epntap#populate` (the parameter names are simply the column names). In addition there are the columns from the product mixin (accref, mime, embargo, and owner, which in this case probably should all be NULL). [TODO: example; if you have data like this, talk to us so we can get an example here]

### Service

EPN-TAP tables are queried through the data center's TAP service. If you have registred that, there is nothing else you need to do to access your data.

For registration, just add:

```
<publish/>
```

to your table body and run `gavo pub <rd-id>`.

## Writing Examples

In the VO, there is a (as of this writing, fledgling) standard for giving examples for service usage; the idea is to produce HTML that's useful for human consumption with additional, RDFa-based, markup to let clients figure out how to fill their interface forms.

DaCHS lets you write such examples in ReStructuredText with some extra markup that is turned into the machine-readable semantics.

### TAP examples

The most prominent kind of examples are the one for TAP/ADQL. These reside `$GAVO_ROOT/etc/userconfig.rd`. If you don't have that file yet, create it with a contents of:

```
<resource schema="__system">
</resource>
```

There can be quite a bit of configuraiton in there (`gavo admin dumpDF //userconfig` shows you what DaCHS uses when you don't override the items). TAP examples are taken from a STREAM with id `tapexamples`. One such example is already given in the userconfig.rd of the distribution. It will be ignored if you define your own `tapexamples`, which is probably not a major loss.

In the stream, there are `_example` meta elements with a mandatory `title` attribute and ReStructuredText contents. The built-in example looks like this:

```
<meta name="_example" title="tap_schema example">
  To locate columns "by physics", as it were, use UCD in
  :taptable:‘tap_schema.columns‘.  For instance,
  to find everything talking about the mid-infrared about 10µm, you
  could write:

  .. tapquery::

    SELECT * FROM tap_schema.columns
      WHERE description LIKE '%em.IR.8-15um%'
</meta>
```

You must give a query in a block marked `..tapquery::`. A typical client would fill this into whatever its UI provides to write queries.

Optionally, you can give the client a hint what table the example pertains to using the `:taptable:` interpreted text role. A client would typically use that to restrict the display of the example to states in which it assumes the user wishes to runs queries against that particular table.

DaCHS does not (yet) pick up changes to `userconfig` automatically. Hence, after adding or changing examples, you have to run:

```
gavo serve exp % //tap
```

as the gavo user on the server. The somewhat funky-looking command line consists of `exp` as the unique abbreviation of serve's exipireRDs subcommand, `%` as the identifier of userconfig (which needs manual reloading), and `//tap` (which needs reloading as the rendered examples text is cached on it).

This will only work if you've set `[web]adminpasswd` in your `/etc/gavo.rc`; of course, you could also restart the server.

To see your shiny new example(s), point your browser to `<server url>/tap/examples`.

### Datalink examples

DaCHS also contains provisional support for examples associated with datalink services. Since client support for it is not in the pipeline and it's not planned for the standard either, it's of questionable utility so far, but hopefully that's going to change.

To add an example to a datalink service, add an `_example` meta with a title attribute directly to the service definition; for instance:

```
<service id="sdl" allowed="dlget,dlmeta">
  <meta name="title">FEROS Datalink Service</meta>
  <meta name="_example" title="Usage Example">
    On published datasets like
    :dl-id:‘ivo://org.gavo.dc/~?feros/q/f04031.bdf‘,
    this service lets you to cutouts, translations into FITS binary
    tables, ASCII, and possibly more, as well as simple recalibration.
  </meta>
```

There is only one interpreted text role in there so far, `dl-id`. That's a PubDID the service will generate a datalink document for.

To see the example, point your browser to the service URL with the examples renderer. If the above fragment were in the RD `flash/q`, the URL would thus work out to be `<server url>/flash/q/sdl/examples`. No manual reloading is necessary here, changes will be picked up automatically.

### Generic examples

The DALI standard defines a term *generic-parameter* which can be used to annotate all kinds of services; this may come in particularly handy with the api renderer.

To use it, you can write something like:

```
<meta name="_example" title="Dataset identifier">
        Publisher dataset identifiers have a query part, but the
        IVORN part still has to resolve:
        :genparam:‘uri(ivo://org.gavo.dc/~?feros/data/f89411.vot)‘
</meta>

<meta name="_example" title="Standard identifier">
        New-style standardIds use fragments to refer to standard keys within
        vstd:Standard records, as in
        :genparam:‘uri(ivo://org.gavo.dc/std/glots#tables-1.0)‘
</meta>
```

Note that such examples best sit in the service rather than top-level in the RD; if the are direct children of the RD, they would appear in all services defined in the RD.

DaCHS does not yet have support for the *capability* and *continuation* properties defined by DALI. Ask if you need them.

## Services Over Views

Sometimes a service should execute queries spanning several tables. One way to go about this would be to use a fancyQueryCore.

However, since metadata generation is much more straightforward if a service sits on top of something that's actually pretty much a table, the better way usually is to define a view executing the query. There are some subtleties with this, though, so here's a few words on how we recommend you go about that.

First, you'll obviously define the tables involved:

```
<table onDisk="True" id="master" mixin="//scs#q3cindex"
    primary="catno" adql="True">
  <column name="catno" type="integer" required="True"
```

```
    ucd="meta.id;meta.main"
    tablehead="id#"
    description="Identification number in the ARIGFH master catalog"
    verbLevel="1"/>
  <column name="raj2000" type="double precision"
  ...

<table onDisk="True" id="identified" adql="True">
  <column name="dist" type="double precision"
    ucd="pos.angDistance" unit="deg"
    tablehead="Offset"
    description="Offset between master catalog position at catalog
      epoch and equinox and the catalog position"
    verbLevel="1" displayHint="displayUnit=mas,sf=1"/>
  <column name="masterNo" type="integer" required="True"
    ...
```

A good approach might be to stuff the columns that will later show up in the view into a STREAM and then replay that stream in both the table definitions and the view definition, but since the RD we're using as an example here worked with `original`, this we use in this introduction; with STREAMs, sharing the columns looks differently, the rest remains the same.

To save typing and make things a bit clearer, we use LOOPs for copying the source columns. The view definition then looks somewhat like this:

```
<table onDisk="true" id="id" adql="True">
  <meta name="description">
    The stars from the gfh table having counterparts in the master
    catalog, together with those counterparts.
  </meta>
  <column original="master.catno" name="masterNo"/>
  <column original="master.component" name="compMaster"/>
  <column original="master.raj2000"/>
  <column original="master.dej2000"/>
  <column original="master.pmra" name="pmraMaster"/>
  <column original="master.pmde" name="pmdeMaster"/>
  <column original="master.mv" name="mvMaster"/>
  <column original="master.mb" name="mbMaster"/>

  <LOOP listItems="catid catan dist iq">
    <events>
      <column original="identified.\item"/>
    </events>
  </LOOP>

  <LOOP>
    <codeItems>
      for col in context.getById("gfh"):
        yield {'item': col.name}
    </codeItems>
```

```
    <events>
      <column original="gfh.\item"/>
    </events>
  </LOOP>

  <viewStatement>
    CREATE VIEW \curtable AS (
      SELECT \colNames FROM
        (SELECT catno, raj2000, dej2000,
          pmra AS pmraMaster,
          pmde AS pmdeMaster,
          mv AS mvMaster,
          mb AS mbMaster,
          component AS compMaster FROM \schema.master) AS m
      JOIN
        \schema.identified AS idf
      ON (masterNo=catno)
      JOIN \schema.gfh
      USING (catid, catan))
  </viewStatement>
</table>
```

As you can see, you can rename columns, and the second loop actually gets column names from some table obtained via its id programmatically (only do that if you're sure you actually want to follow changes in source table structure; the reason this is no one of the joined tables in the view has to do with the specific dataset).

The viewStatement essentially is more or less arbitrary SQL. For robustness against changes of table structure or schema or table name changes, however, you should probably always start with:

```
CREATE VIEW \curtable AS (
  SELECT \colNames FROM
```

– `\\curtable` will automatically adjust to whatever schema and table name is given in the RD, and `\\colNames` gives all the names of the columns; using it, you'll get errors instead of silent failures if you add or remove columns in the table definition but fail to adjust the view statement.

When making these tables, it pays to be a bit careful with the `data` elements, as of course the view depends on the source tables. In particular, when you re-import one of the source tables, the view will get dropped, and it is a good idea to tell DaCHS to automatically re-make it. The recommended setup for this looks like this:

```
<data id="import_master" recreateAfter="gfhtables">
```

```
  ...
  <make table="master"...

<data id="import_identified" recreateAfter="gfhtables">
  ...
  <make table="identified"...


<data id="gfhtables" auto="False">
  <make table="id"/>
</data>
```

– essentially, you make the view in a non-auto data which gets imported every time the source tables are re-made. Note that making the `data` for one source table when the other doesn't exist and will not be made in the same import will lead to an error in the view creation. This is harmless for the import of the table you imported, as `data` creation on `recreateAfter` happens in a separate transaction.

## The Registry Interface

Conceptually, the VO's Registry is a set of resource records (i.e., descriptions of services, data, or other entities) to let users locate resources relevant to them (e.g., look for a service giving surface temperatures for OB stars). Whatever as a resource record is called *VO resource* in the following to keep them apart from whatever DaCHS resource descriptors describe; DaCHS RDs may descibe zero, one, or multiple VO resources. We apologize for the confused nomenclature.

Physically, there are several services that keep and update this set and let people query them (a "full registry"), e.g., the VAO registry, the ESAVO registry, or the Astrogrid registry. All these should harvest each other and thus have identical content (this is currently not always true).

To be part of the VO, you have to register your services. DaCHS makes this fairly easy since it contains a publishing registry. This is again a service that exposes a standard interface defined by the Open Archives Initiatives. There is a renderer for the OAI harvesting protocol (OAI-PMH) called `pubreg.xml` that goes together with `registryCore`. The service `//services#registry` with this renderer has a vanity name of `/oai.xml`, which is you data center's publishing registry "endpoint". Full registries obtain the resource records present on your data center for there.

Each VO resource has a unique identifier of the form:

```
ivo://<authority>/<stuff>
```

-- <stuff> is defined by the DaCHS software (to be `<RD id>/<XML id of registred object`), whereas <authority> is a globally unique string. It is recommended that you use your DNS name (or some appropriate part of it), which will provide some uniqueness. The authority is declared in your gavorc (see below). Details on VO identifiers can be found in IVOA Identifiers.

To claim an authority, you have to define who you -- as an organization -- are. For this, DaCHS will create a resource record for your organization, too, where "your organization" for DaCHS means whatever you give as creator.name in defaultmeta (see below), which in general should be something like "My Institute Data Center" rather than "My Institute". You can register "My Institute" as well, if you want, but, the way things are written now, not as the entity running managing the authority.

To make the VO aware of the existence of your data center, you will need to tell the RofR (Registry of Registries) about your data center. Before you can do this, you need to fill in quite a bit of information in your gavorc and `etc/defaultmeta.txt`. The registry section in the operator's guide has information on what to do.

## Restricting Access

Unfortunately, many data providers believe they want to have their data proprietary for a while. Although they are almost certainly misguided in this, it is hard to enlighten them, and so it's preferable to have the data in a data center with encumbered access rather than just on the providers' machines.

Therefore, DaCHS has features to restrict access. Right now, this is very basic and only provides what is known as "mild security", since it is based on HTTP basic authentication over (usually) unencrypted lines. Given that we are not dealing with sensitive information and snooping attacks on our connections would be too much honor, we consider this enough. However, if you were interested in contributing support for OAuth (say), we'd gladly help.

Note, however, that even HTTP basic authentication needs client support. Recent versions of TOPCAT and Splat have that, others do not, and they simply will not work with password-encumbered services. The situation with OAuth is much worse.

### User/Group management

The DaCHS user administration is fashioned a bit after the Unix user/group model, except that there always is a group corresponding to a user. To create a user and its group, use `gavo admin adduser`, like this:

```
gavo admin adduser kroisos notsecret "Remove when xy is public"
```

This command adds the user kroisos with the password notsecret and an optional comment reminding future operators what to do with the identity. Note that the password is stored in clear text in the database – which allows you to handle "I forgot my password" requests gracefully; as long as we only do HTTP Basic authentication, this doesn't matter much since with it, the passwords traverse the net in basically cleartext anyway. Again: all this is mild deterrence rather than hard security.

To add existing users to groups, use `gavo admin addtogroup`, like this:

```
gavo admin addtogroup kroisos happy
```

– this adds kroisos to the happy group, and whoever can authenticate as kroisos will be allowed access to any products or services retstricted to happy.

To discover further commands manipulating the user table, try:

```
gavo admin --help
```

*Important*: When you use authentication, please set the `[web]realm` configuration item to some string reasonably characteristic for your site. Many systems will store credentials by realm, and if different sites use the same realm, their credentials will clobber each other. For details see the customization info in the operators' guide

### Protecting Services

To password-protect entire services, use the `limitTo` child of the `service` element, for instance:

```
<service id="scs" core="scsCore" allowed="form,scs.xml"
  limitTo="happy">
  ...
</service>
```

Any access to the service will then require a client to authenticate as a user belonging to the group given in `limitTo`. Only one such group can be given. If you forsee the need for complex authorization schemes (rather than "there's one user on my system, and whoever's authorized get its credentials"), it is probably a good idea to create one user per service and add "real" users to the corresponding group as necessary.

### Embargoing Products

DaCHS' products subsystem has the notion of owners and embargo periods, which allows public services to deliver metadata on products during their proprietarity period, while handing out the data itself only to authorized clients. The embargo will automatically be lifted once the proprietarity period is over.

To make a "product" (e.g., spectrum or image) proprietary, in the products#define application building the rowdict, set the `owner` and `embargo` keys. Owner is the name of a user created as described above, embargo must eventually become a timestamp, so you'll in general come up with an ISO datetime string or a python `datetime.datetime` instance. Here's an example that says images become public a year after the observation:

```
<fitsProdGrammar qnd="True">
  <rowfilter procDef="//products#define">
    <bind key="embargo">parseTimestamp(row["DATE_OBS"])+datetime.timedelta(
      days=365)</bind>
    <bind key="owner">"danish"</bind>
    <bind key="table">"danish.data"</bind>
  </rowfilter>
</fitsProdGrammar>
```

This is, in our view, an acceptable policy, but many observers want weird policies (try to talk them out of it, since such behaviour is not nice, and it leads to a bad user experience in the VO as a whole). You can get as fancy (or antisocial) as you like using custom rowfilters, as in the following example that sets a default embargo for the end of 2008, except for calibration frames and the observations of two objects made in 2003:

```
<fitsProdGrammar qnd="True">
  <rowfilter procDef="//products#define">
    <setup><code>
      <![CDATA[
      def getEmbargo(row):
        res = '2008-12-31'
        if (row.get("ARI_TYPE")!="SCIENCE" or
            row["ARI_OBJC"]=='Q2237+0305'
            or row["ARI_OBJC"]=='SBSS 1520+530'):
          if '2003-01-01'<=row['DATE_OBS']<='2003-12-31':
            res = '2005-12-31'
        return res
      ]]>
    </code></setup>
    <bind key="owner">"maidanak"</bind>
    <bind key="embargo">getEmbargo(row)</bind>
    <bind key="table">"maidanak.rawframes"</bind>
  </rowfilter>
```

An embargoed product can only be retrieved by the owner until the embargo period is over. What you give as `owner` is a group name; if someone can authenticate as the member of a group, she can access the data – see above for details on how to create users and groups.

[1] This assumes you're doing this on your local machine (which were recommend to get started. If you do this on a remote machine, you will obviously have to replace the localhost in the url with the machine's host name. However, that still will not work as, by default, DaCHS only binds to the loopback address. To change this, edit or create the file `/etc/gavo.rc` to include at least:

```
[web]
bindAddress:
```

(yes, there's nothing behind "bindAddress:"). Restart the server and you should see the output.

[2] The `pos.eq` UCDs are hardcoded; this is because the standard simple cone search protocol specifies that the coordinates passed in are ICRS, and hence other systems – galactic, say – make little sense here. Also, the web form-variants of the protocols let users enter Simbad-resolvable identifiers rather than positions. In sum, making these things generic for other systems would be an unreasonable implementation effort.

If you want to allow queryies in other systems, just write the index statement inline, e.g., for galactic coordinates in the columns `lambda`, `beta`:

```
<index name="q3c_gal" cluster="True" columns="lambda,beta"
  >q3c_ang2ipix(lambda,beta)</index>
```

A simple condDesc that searches using this index could be:

```
<condDesc>
  <inputKey original="lambda" required="True"/>
  <inputKey original="beta" required="True"/>
  <inputKey name="sr" tablehead="Search Radius">
    <values default="0.001"/>
  </inputKey>
  <phraseMaker>
    yield q3c_radial_query(lambda, beta, %%(%s)s, "
      "%%(%s)s, %%(%s)s)")%(
      base.getSQLKey("lambda", inPars["lambda"], outPars),
      base.getSQLKey("beta", inPars["beta"], outPars),
      base.getSQLKey("sr", inPars["sr"], outPars))
  </phraseMaker>
</condDesc>
```