GAVO DaCHS Tutorial

Author: Markus Demleitner
Email: gavo@ari.uni-heidelberg.de
Date: 2024-03-05

Contents

Chapter 1

Introduction

DaCHS is an integrated suite for publishing astronomical data to the Virtual Observatory, with support for services operated through web browsers, too. If you are not sure what the Virtual Observatory is: Technically, it is a set of APIs and a Registry in which the URIs for services compliant with these APIs are associated with a bunch of relatively formalised metadata. What that actually means is probably best illustrated by trying some of the tutorials involving TOPCAT or Aladin from VOTT.

This tutorial is part of the DaCHS documentation, in particular the Installation Guide and the Reference Documentation. The goal is that after skimming the tutorial DaCHS operators have a good idea of what to do when and to provide recipes for the most common problems. It will also guide operators in setting up a site well integrated into the larger VO.

In order to have some basic understanding of what all this is about, our advice is to linearly work through DaCHS Basics in order to have an idea of DaCHS' basic concepts (yes, it's a bit more than 20 pages, but we'd like to believe that grokking that is a half hour well spent) and then try the quick start with DaCHS.

Depending on the data you intend to publish, then skim the corresponding section from Publishing Data Using VO Protocols and build a service, presumably using `dachs start`. Once the service is ready to go public, look at Configuring DaCHS and finally Interacting with the VO Registry; they tell you how to become a good citizen of the VO.

For continued operation, and if you have inherited a running DaCHS installation, read Server Operations.

The concluding chapter on Topics in DaCHS Publishing collects more detailed explanations, recipes, and other material helping in common publishing situations. We believe it is worth skimming through this part in order to have an idea what is there and return to it for closer study when necessary.

All this assumes that you have a basic understanding of how to run a simple Debian server (or some other distribution, which then you should understand a bit better). If this is your first server, you probably should skim (at least) sections 5 through 10 of the Debian Administrator's Handbook. It is not important that you understand and memorise everything that is written there, but you should get an idea of where to look up what and how, roughly, things fit together.

Installation

Unless you have truly overriding reasons to try something else, to try out and learn DaCHS, get a machine running Debian stable and type:

```
apt install gavodachs2-server
```

This will give you the DaCHS software that was current when the Debian release was frozen, which ought to be just fine for "normal" publishing activity. If you need a bleeding-edge DaCHS (or run into a fixed bug), read up on manually add our repository.
DaCHS can be installed and run in many other ways, as detailed in the Installation Guide. But since DaCHS involves communication with a database server and the network, it is certainly a good idea to start out with a simple, one-server setup on a platform that many others use and that therefore has a rather low likelihood of odd surprises.

Notation

DaCHS is the name of the software package (it's an acronym for Data Center Helper Suite and happens to mean "badger" in German). The unix command is called `dachs`. The default paths, the python package, and the default database are called `gavo` (though you could configure it differently if you wanted) – that's the name of the project within which DaCHS was born, and changing these names to dachs afterwards would have been an challenge for backwards compatiblity. Since it didn't seem a big deal to us, we didn't do it. Still, apologies for the confusion.
In the following, we call people running DaCHS **operators** (so, if you're reading this, that's you), people consuming your services **users**, programs communicating with your services **clients**.
Something like "Element table" is a link into the reference documentation, whereas "install.html" references other pieces of DaCHS documentation.
Specifications like `[web]serverURL` refer to configuration items in `/etc/gavo.rc` (or its alternatives; see Basic Gavo.rc Settings and Configuration Reference for more on this).

Chapter 2

DaCHS Basics

Invoking DaCHS

All DaCHS functionality is invoked through a program called `dachs`. Multiple functions are integrated and selected through the first argument; run `dachs help` (or, for perhaps less cryptic output, `man dachs`) to see what's available. Realistically, the functions most operators will be confronted with are `start`, `import`, `serve`, `publish`, `test`, and perhaps `admin`. DaCHS will be happy with unique prefixes into these command names; for instance, we tend to just type `dachs adm xsd` rather than `dachs admin xsdValidate`.

DaCHS has some global options (that go in front of the subcommand name), which mostly are useful for Debugging. Most subcommands take options and/or arguments, which then have to be after the subcommand name. If you've ever used git: Same deal, except we're aiming for less cryptic man pages.

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

```
$ dachs imp --help
Usage: dachs imp [options] <rd-name> {<data-id>}

imports all (or just the selected) data from an RD into the database.

Options:
  -h, --help            show this help message and exit
  -n, --updateRows      Use UPDATE on primary key rather than INSERT with rows
[and so on]
```

or look at the man page.

DaCHS on Disk

In the default configuration, almost everything DaCHS is concerned with is below its **root directory**, which by default is `/var/gavo`. There are two exceptions to this rule:

- Configuration in `/etc/gavo.rc` (or, in a pinch, `~/.gavorc`) – see Basic gavo.rc Settings for details; and also on how to have these in other locations.
- anything DaCHS has put into the database.

The **inputs directory**, defaulting to `/var/gavo/inputs`, is where most of a DaCHS operator's work is done. It is where the services are configured, where metadata and tabular data is fed from into the database, and where bulk data is served from.
Three of the other directories in there are relevant for the casual user:

- `/var/gavo/etc` (the **config directory**) contains various extra configuration like some Basic Metadata, the database access profiles (which is nontrivial in DaCHS because it runs queries as differently privileged users), and The Userconfig RD.
- `/var/gavo/logs` contains access, error, and info logs. In particular, dcErrors and dcInfos deserve a look whenever DaCHS spits out errors; in particular when you called `dachs` with the `--debug` global flag, it will dump tracebacks there that are not visible in the main output.
- `/var/gavo/web` lets you override many of the browser-facing things of DaCHS, and there is `nv_static`, in which you can dump files that then become visible at `<serverurl>/static`. Templates are in a `templates` child, and one template you will probably want to override is The Root Page of your data centre.

The Resource Descriptor

When publishing data, what you do in DaCHS is write a **resource descriptor** (RD). That's a piece of XML (the serious ones at the GAVO data center are between 100 and 2000 lines) containing almost all you and DaCHS need to know about a service and its underlying data. DaCHS tries to avoid spreading information on a single resource over multiple files in order to facilitate understanding what is going on later or after a handover.
There are exceptions to the one-stop-definition principle (e.g., C-language boosters, custom grammars and pages, etc.), but for publishing of well-behaved data operators should not need those: Essentially, a service is the **upstream data** (i.e., what you got from the scientist or project) data plus the RD. Put another way, the RD is the formalisation of what you can learn about the data from your upstream *about* the data.
This section will introduce an example RD for a simple service (we will use it in the Quick Start with DaCHS) publishing a simple astrometric catalogue. To follow the exposition below, have a look at the XML by fetching http://svn.ari.uni-heidelberg.de/svn/gavo/hdinputs/arihip/q.rd (use HTTP, not HTTPS, as the latter requires authentication) and opening the resulting file q.rd in a text editor. The upstream data for this is available at http://dc.g-vo.org/arihip/q/cone/static.
This file comes from the collection of RDs active at GAVO's Heidelberg Data Centre (https://dc.g-vo.org) which you can inspect at http://svn.ari.uni-heidelberg.de/svn/gavo/hdinputs – in particular, you're welcome to copy as much as you want. To locate examples for concrete elements, meta items, and such, have a look at elemref.html for these.
While the XML elements within an RD could come in any sequence convenient (except that most references need to be lexically after the elements they reference), we recommend to adhere to first declare global metadata, then define the tables and the ingestion rules, continuing with the service definitions and concluding with the regression tests. The following exposition follows this recommendation.

Global Metadata

The root element of an RD always is `resource`, and you should immediately define the database schema anything you import will end up in:

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

Attributes and Elements in DaCHS

DaCHS almost never distinguishes between attributes and elements with atomic content, which means that you could also have written the opening tag of the root element as:

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

This notational freedom sometimes allows clearer statements, 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 description). For atomic properties, later specifications overwrite earlier ones.

RDs are normal XML files (meaning that you could, e.g., add an XML declaration if you want an encoding other than utf-8).

DaCHS assumes the schema name to be identical to the **resource directory**, which again must be a child of the inputs directory. If you have a strong reason to deviate from that convention, you must 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. From DaCHS 2.6 on, it is recommended to always set `resdir="."` explicitly. This way, assuming the RD is a direct child of the resdir (and you should always build your resources this way), the RDs will still work even if the directory is mored. This is an advantage in the context of mirroring resources.

In general, you should have exactly one RD per database schema. This is not enforced, but sharing schema names between RDs will cause some undesirable behaviour. 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.

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.waveband">Optical</meta>

<coverage>
  <spatial>0/0-11</spatial>
  <spectral>2.721e-19 4.138e-19</spectral>
  <temporal>1989-09-01 1993-08-15</temporal>
</coverage>

<FEED source="//procs#license-cc-by" what="ARIHIP"/>

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

  [...]
</meta>

<meta name="source">2001VeARI..40....1W</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 crucial for later registration of the service, some of it turns up in service responses, and quite a bit is used in making web pages, in the metadata sidebars and elsewhere.

Metadata elements have a `name` attribute that gives the "kind" of metadata contained, and sometimes also determine a specific type. The names should usually give a good indication of what information is given. We will later revisit individual items and what should and should not be put there.

If there is more than one meta element for a name (like for subject in the example), the key is **multi-valued**. For quite a few meta keys, DaCHS will at some point raise errors you if you have multiple values where you can't have them, as for instance, with titles, descriptions, or creation dates. Because DaCHS itself tries to treat all metadata as equally as possible, the failures will not happen during metadata definition but when generating things like registry records, which sometimes makes it a bit hard to figure out the error's origin actually lies in the metadata definition. The `dachs val` subcommand should catch such problems, though (and if it doesn't, please report that as a bug).

Metadata can come in various **formats** as determined by the `format` attribute. If you give nothing there (which makes it `plain`), DaCHS will apply some whitespace normalisation, and it will interpret empty lines as paragraphs if the target format supports it.

With `format="rst"` (as used in `_longdoc` and `_intro` in the example), the content will be interpreted as reStructuredText, which lets you do rather fancy things when the

metadata is being rendered into HTML. Be careful to use consistent indendation in this case. There are also `raw` and `literal` as formats; don't worry about them now.

Metadata with names starting with underscores – like `_intro` in the example – is for DaCHS-internal use, very typically when building informational pages; where there's no underscore, there is generally some external standard, and in all likelihood the information will end up in the VO Registry in some way. Many of those we discuss in data_checklist.html. This checklist was actually written to hand out to people wanting to publish in the Heidelberg deata center, but you are welcome to re-use it.

In the above RD snippet, there are two elements that are not simple meta elements. One is `coverage`; this gives the extent of the data contained in the service in space, time, and spectrum. You will usually only define this manually if you know things the computer can't easily find out itself. This is typically "data consists of several well-defined campaigns" (so you have clear time intervals) or "data is taken in a few of rather small bands" (so you have clear spectral intervals).

The other non-meta element is `FEED`. In DaCHS uppercase elements signify active tags, elements that are replaced by something else (or nothing) in the document tree. `FEED`, as used here, simply inserts a stream (essentially, a sequence of elements) defined elsewhere and puts it in. The reference manual has list of predefined streams with an indication of what is inserted. Here, this is used to insert a few more formalised meta items declaring the licence under which your data can be used and distributed. Please be explicit on your licences, see Licensing below for why we ask that (and why not).

The last major concept you should now in connection with DaCHS metadata is **metadata inheritance**. For instance, if you define a service within the RD, and there is no title meta within the service, DaCHS will use the title meta on the RD. The root of the metadata inheritance hierarchy is the file `etc/defaultmeta.txt`. If you defined a meta item `title` there, this would be the default title for everything in your data centre (which is something you probably don't want). This default metadata, however, is handy for items that plausibly are essentially constant across a data centre, like a publisher or a contact. By the inheritance rules you can still override it in individual RDs or services if necessary.

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.

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 (but over standard SQL it will let you have leading underscores).

Be sure to always specify `onDisk="True"` unless you're going for special effects – without it, the table will end up only in memory and be gone after the import. 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 is one of the deeper concepts in DaCHS. We defer a closer look at that to the Mixin Introduction.

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.

While this table doesn't have metadata (because this is a one-table resource, and the global resource metadata is the one of the table itself), table elements may override RD metadata, and in multi-table RDs it's usually a good idea to override title and description in the table element itself.

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. See Space-Time Metadata for more information.

Column definitions

The RD continues with the definitions of the columns that make up the table:

```
<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 in a 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, 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 (see Referencing in DaCHS).

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. Among the more common types are:

- `text` – a string. You could also use types with explicit length like char(7), but this is discouraged. It does not help postgres (or much anything else within the DC). The only advantage would be if you later want to handle FITS Binary serialisations, because when all strings and arrays in a record have fixed lengths, libraries can directly seek to records. Also note that text-valued columns can only keep ASCII. Use the `unicode` type if you need non-ASCII.
- `real` – a floating point number, pretty much like `float` in C.
- `double precision` – a floating point number. You should use such 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. See Some Words on Times below for why you probably do not want timestamps.
- `date` – a date. See timestamp.
- `time` – a time. See timestamp
- `spoint`, `scircle`, `sbox`, `spoly` – objects of spherical geometry, taken from pgSphere. See Creating pgSphere Geometries below on how to ingest them.
- `jsonb`, `json` – structured data in a column. If you think this solves one of your problems, you are most probably wrong (in the long run). Still, if you want to use these, feed them with lists, dicts, or JSON literals in strings.

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. Because it's hard to rigorously define, DaCHS does not distinguish between dimensionless (as, for instance, width/height, which in principle has a unit that just happens to come out as 1), things that can't have units to begin with (e.g., an observer name), and things that should have units but are written in a way that they can't (e.g., sexagesimal coordinates or ISO datetime strings). In case you must have units not in VOUnits (e.g., mCrab or jupiterMass), enclose them in single quotes as forseen by VOUnits. DaCHS will then not complain about them.

- `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 end up in help bubbles 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 Table Notes. An example for a descripton:

```
<column name="aperture">
  <description>The aperture is the full-width-half-max 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 `dachs 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) in columns that may have NULLs. 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 `dachs info` 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).
A child element of `table` we have not mentioned above is this:

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

This is the simplest, but mostly sufficient, form of telling DaCHS to index a column, where the columns attribute just lists the names of the columns to index – and most of the time, that will be just one. More complicated index patterns are possible – see element index in the reference.
Indices are usually generated after a full import. If you add or change indices later, run `dachs imp -I`, which will re-make all indices. For large tables, you may want to just create one particular index; in that case, see the `indexStatements` subcommand to `dachs admin`.
Scrolling a bit further down in the arihip RD, you will notice some LOOP constructs. These are discussed below under active tags.
After the column definitions, there are meta elements called "note"; for those, see Table 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 must have ids which can be used to individually reference them from a `dachs imp` command line; this is useful if you just want to import one part of a multi-table data collection. The default of `dachs 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.

Source Specifications

The Element sources 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 if true.

Grammars

In the RD, it next says:

```
<columnGrammar topIgnoredLines="9" preFilter="zcat">
  <colDefs>
    hipno:     3-8
    srcSel:    47-49
    alphaHMS:  59-73
    [...]
  </colDefs>
```

This is the definition of a **grammar**, which in DaCHS is something that turns some (specific) sort of input into a sequence of dictionaries. Stuff coming out of a grammar we sometimes call **rawdict**, with a bit of a pun, because after some processing in the rowmaker (discussed below) they will turn into **rowdicts**. While a rawdict almost always (exceptions are for higher-level source formats like FITS) maps strings to strings, the rowdicts map strings (column names) to something directly ingestable into the database (e.g., integers, or perhaps polygons).

There are many grammars built into DaCHS, e.g., for getting values from FITS headers or tables, from VOTables, via regular expressions, or using column-based formats; you can also write specialised grammars in python, and there are direct grammars that you will want to learn about if you have data larger than, say 10e7 rows (see Grammars in C for these).

It is often useful to inspect what a grammar emits. You can do that using import's `--dumpRows` flag, usually telling DaCHS to stop before you drown in screen output, perhaps like this:

```
dachs imp -M 100 --dumpRows q.rd | less
```

The source file of arihip has both a separator character and aligned columns, so we could use either Element columnGrammar (which, as shown above, has ranges of character positions with a line as shown by most editors to define what will be in the rawdicts' values) or Element reGrammar (which we could have used to split along vertical bars). The choice was mainly a matter of taste, but experience also shows that data providers are not always careful to somehow escape their separator characters when the appear in values, so using a column grammar is perhaps the somewhat more defensive choice.

By the way, 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.

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 lets you run the input through a shell command before it is processed using DaCHS (if you find yourself doing a lot more than just decompression in such a preFilter, you should probably look for a different solution).

Somewhat further on in the grammar, you will find a LOOP element with lots of backslashes. It is not necessary to understand that to write good RDs. But it is handy and helps avoiding repetition, which is why we are showing it in the example RD, and we have a chapter devoted to it later in in the introduction: Metaprogramming: Macros and LOOPs.

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

As explained above, the 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). All that can be fixed in row makers, recipes for producing rowdicts.

Rowmakers are where it's rather common to have embedded python code in RDs; however, in simple cases you can get by with purely declarative XML, which, of course, is preferable, as it is much less likely we will break that while DaCHS, python, and DaCHS' dependencies evolve.

A row maker has three kinds of children:

- Element var – assignments of expression values names in the rawdict.
- Element map – simple mappings of (python) expressions to values in the destination rowdict
- procedure applications (Element 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 var generates additional values in the rawdict, whereas map directly manipulates the rowdict): with simple mappings between names, using pre-defined procedure applications, or generating values from python expressions. In the latter case, 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).

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 functions available for row makers, as well as everything from the python standard library modules `datetime`, `math`, `os`, `re`, `sys`, `time`, and `urllib.parse` (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` as `pgsphere`. These latter may not be quite as stable, so it's probably better if you ask for having something in the official rowmaker functions than

pull them from these internal-ish modules. Having said that, you can figure out what is in there in DaCHS' apidoc.
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 (as said above, less code is better in RDs). This will perform some default conversion (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"/>
```

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 often controls both) that there is an even more compact shortcut for this:

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

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.
In the authors' experience, building the rowdict in the rawdict (i.e., using `var`) and then mapping it all with `idmaps="*"` has proven to be the preferable approach in many situations. This pattern is used, for instance, in the built-in mapping procedures for SSAP. On the other hand, the built-in mapping procedures for SIAP were written before we had this insight, and thus they are, regrettably, incompatible with this approach.
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, i.e., before conversion to the target type. This is good because otherwise you could only safely compare against relatively few floating point numbers (99.99 is not among them). In cases where you can compare against the converted value, it is preferable to use the `nullExpr` attribute on vars and maps.

A complementary way to produce NULLs (in particular when `parseWithNull` is used in expressions) the `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 decimal point, you could use:

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

If `parseWithNull` 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, 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`.

Rowmaker procedures are in Element apply. There are a number of predefined ones (see Procedures available for rowmaker/parmaker apply in the reference documentation). Quite commonly, mixins for tables underlying VO standard protocols come with one or more applys to populate the tables in a controlled fashion. More on these when we discuss the protocols served by these mixins.

You can also define your own applys, but that's for higher-level DaCHS magicians and won't be discussed here.

Service Definitions

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 do the actual computation or database query.
- renderers; these digest the bits coming in from the client and (in general) format the result in some way requested by the client again. There are renderers for web forms and several VO protocols, ones producing HTML documentation for RDs, and so on. In the ideal case – 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.
- services; they hold together the core and the renderer, can reformat core results, and they are what is registered, which means that they hold the metadata that ends up in the registry (most of it is typically inherited from their RDs).

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 section on Renderers Available.

Which renderer is used when a service is run is selected by the client: it's (usually) the last segment of the path part of the access 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). Note that some of the more exotic renderers may pose special requirements on cores. Don't be too disappointed of a given renderer won't work with your core.

The most common core for catalogue services (and the one you'll typically use for SCS services) is the Element scsCore, as used here; you might also use a generic Element dbCore. Other cores you should know about include the Element nullCore (for services that don't do any server-side computation at all), Element datalinkCore (which can define all sorts of manipulations on and linking of datasets), the python and custom cores (cf. writing-custom-cores), and several protocol-specific cores discussed below.

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 and rowmakers, 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 the id of a `table` element. This is the table the query will run against.

The condition descriptors within dbCores (defined in Element condDesc) define, in essence, input fields: in 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 (which is part of the //scs#coreDescs stream) locates the "main" positions as identified by UCDs and generates queries against them using two input fields. One somehow specifies center position of the region of interest, the other gives 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 current renderer's parameter style (which is given in renderers available).

For instance, if you build a condDesc from a real- or double precision-typed column, and the service is accessed with a form-styled renderer, users can constrain the column's values with Vizier-like float expressions, where, for instance, an interval would be written as `range_min .. range_max`. When the request comes in through renderers with a parameter style of `pql` (e.g., SSAP), the condDesc would expect SSAP-style intervals (e.g., `range_min/range_max`). In the parameter style `dali`, that interval would be written `range_min range_max`. Yes, it is deplorable that something like the parameter style exists, but that particular uglyness is not DaCHS' fault; in practice, it's a lot less horrifying than it sounds, because clients very typically just use one renderer, and they hopefully agree with it on the parameter syntax.

The `service` element must have an `id` attribute that is used to select the service run in the access URL's last but one segment. Furthermore, there should be certain pieces of metadata useful for the Registry. 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.

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 reference for the the scs.xml renderer, requires the parameters of a test query returning a non-empty result.

Regression Tests

While it's admittedly boring, writing regression tests while developing your RD will save you a lot of fear, uncertainty, and doubt later. In the arihip RD, there are two regression tests:

```
<regSuite title="ARIHIP regression">
  <regTest title="ARIHIP form service appears to work.">
    <url parSet="form" hscs_sr="1.0" hscs_pos="0.00 1.08"
      >cone/form</url>
    <code>
      self.assertHasStrings("PM RA (STP)",
        "1.0890086361", "-5.00", "Note bin")
    </code>
  </regTest>

  <regTest title="ARIHIP SCS response looks reasonable">
    <url RA="0.0273384667" DEC="25.8864593500" SR="0.001"
      VERB="3">cone/scs.xml</url>
    <code>
      row = self.getFirstVOTableRow()
```

```
        self.assertEqual(row["flags"], 's 00 .. .... ... ...')
        self.assertAlmostEqual(row["err_parallaxSI"], 5.25e-07)
        self.assertEqual(row["hipno"], '8')
      </code>
    </regTest>
  </regSuite>
```

As you can see, the tests are grouped into suites, where for most RDs you will just have one suite.
Each test has a title, which is used in diagnostics. The body of a test is a pair of an element url – which, really, is just a convenient way to write a URL with potentially complex parameters – and of an element code, which mostly gives a bunch of assertions about what a running server should be returning for the query encoded by `url`.
We have tried to write the reference documentation's section on regression testing in a tutorial-like fashion and therefore point you there for further details.

Referencing in DaCHS

You will often have to reference items within DaCHS RDs, for instance in command line arguments, in `original` attributes, when pulling in STREAMs, or when deriving from procedures. All these follow (essentially) the same logic:

(a) An RD has an identifier that is the path to the RD file relative to inputsDir (corollary: DaCHS doesn't let you have active RDs outside of inputsDir), with the `.rd` extension cut off.

(b) When referencing an RD from the command line, you can normally use relative references. For instance, if you are in `inputs/myres`, saying `dachs imp q` and `dachs imp myres/q` is the same thing (exceptions apply).

(c) When referencing elements within RDs, use an XML-like fragment identifier. For instance, `original="main"` would use the element with the id `main` in the current RD.

(d) With tables or other items that have children with names, you can reference them by their names after a dot. For instance, `myres/q#main.ra` references the `ra` column in the `main` table in `myres/q`. If referencing within an RD, you can leave the RD id out; in the example, within `myres/q` you can write `main.ra` as well (and you can abbreviate that even more when the parent has a `namePath` attribute; this is what's behind the compact `buildFrom` notations in output tables).

(e) If an RD id starts with a double slash (`//`), it refers to a built-in RD; `//tap` is the built-in TAP RD. To inspect these built-in RD, use a command like `dachs adm dumpDF //tap`. For instance, the TAP service in this way has the id `//tap#run`.

Mixin Introduction

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

```
<table [...] mixin="//scs#q3cindex">
```

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. The `//siap#pgs` mixin makes sure that tables have this structure, and it makes sure that The Products Table 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. If you have read Referencing in DaCHS, you will see that the mixin here sits in a built-in RD (because it starts with a double slash). That is where you will usually get them from, and what you can choose from you can figure out from the reference documentation's mixins section.

For the curious: If you want to see the definition of a mixin (it's in an RD, after all), use admin's `dumpDF` subcommand like this:

```
dachs admin dumpDF //scs
```

The //scs#pgs-pos-index mixin referenced in arihip/q arranges for spatial indexing of tables having some sort of spherical coordinates. To identify which columns to index, this particular mixin inspects the UCDs of the columns, which happens to almost be enough to make the table suitable for publication through the IVOA's simple cone search protocol (SCS). What it looks for here are columns with UCDs of `pos.eq.(ra|dec);meta.main`, and it will index them. 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`).

Metaprogramming: Macros and LOOPs

In the paragraph on Grammars, we skipped over something that admittedly is a bit cryptic (you can skip this subsection if you get scared; you can use all of DaCHS without knowing about it):

```
<LOOP csvItems="\modes">
    <events>
      <col key="dra\mname">\racols</col>
      [...]
      <col key="err_ra\mname\+_mas">\errracols</col>
      [...]
    </events>
<LOOP>
```

As with the `colDefs` discussed above, this part still assigns column ranges in the input file to keys in the rawdict. Column grammars accept these in the relatively compact form in an Element colDefs as seen above.

Alternatively, you can have several of Element col, each of which has a `key` attribute that gives the key in the rawdict and the range of columns in the body. These are useful here because they let us use another piece of RD metaprogramming: The active Element LOOP. Essentially, this lets you define some sort of fillers, and for each set of fillers, whatever is in the LOOP's `events` child will end up in the document tree once with the appropriate fillers.

In this particular case, there were three modes of data reduction in arihip, and these modes turn up in several places in the RD; you may have wondered what a similar thing

in the column definition was. To make this enumeration re-usable, I've defined it in a macro near the top of the RD:

```
<macDef name="modes">
  mname, racols, decols, pmracols, pmdecols, tracols, errracol...
  LTP, 224-233, 276-285, 328-335,  383-390,  438-444, 478-483 ...
  STP, 237-246, 289-298, 339-346,  394-401,  448-454, 487-492 ...
  HIP, 250-259, 302-311, 350-357,  405-412,  458-464, 496-501 ...
</macDef>
```

These macros can now be used in many places, in particular in LOOP's `csvItems` attribute, where LOOP will make one set of fillers from each CSV line. And these fillers then become macros in the LOOP's event body. In that way, the example above spits out something like:

```
<col key="draLTP">224-233</col>
<col key="draSTP">237-246</col>
<col key="draHIP">250-259</col>
```

from the first `col` in `events` and the first and second column in the CSV.
The second element I show in the example above has the somewhat weird `\mname\+_mas` macro call; the `\+` here just makes up for the lack of whitespace after `\mname`. If it read `\mname_mas`, DaCHS would look for a filler called `mname_mas`, which doesn't exist. With that cleared, the expansions of that second `col` example would be:

```
<col key="err_raLTP_mas">478-483</col>
<col key="err_raSTP_mas">487-492</col>
<col key="err_raHIP_mas">496-501</col>
```

The good news: This is essentially how far RD wizardry goes (well, you could argue multiple expansion is worse, but I'd dispute that). Once you write your first RD with LOOPs and macros, you have ascended into the ranks of DaCHS grand mastery.
Most operators lead happy lives without such aspirations. The others can go on to a gentler introduction to active tags below.

Chapter 3

Quick start with DaCHS

This brief section will guide you through the process of importing data into DaCHS, running an associated service, and publishing it, using the arihip RD discussed in The Resource Descriptor above. It is probably a good idea to have skimmed the DaCHS Basics before experimenting here.

Preparing the Resource Directory

In general, when you start publishing data in DaCHS, you will do something like:

```
cd /var/gavo/inputs
mkdir <collection-name>
cd <collection-name>
mkdir data
(put your upstream data into the data subdirectory)
dachs start <data-type-tag>
```

For the sake of getting something running quickly, we will skip the `dachs start` (which necessarily is followed by a lot of work on mapping the data) and just use an existing RD.
You can run the following under any user id, as long as you wisely manage the permissions (short version: `sudo adduser ‘id -nu‘ gavo` to add you to the gavo group, then log out and in again to make it effective). For testing, however, we recommend doing it as the DaCHS' administrative account (for the Debian package, that's `dachsroot`; if you did the setup manually, it's whatever user did the `dachs init`). If you want to stay yourself, see becoming a DaCHS operator.
To get a quick start, just pull in the RD in use by GAVO's data centre:

```
sudo -u dachsroot bash
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, short, and without characters that need URL escaping. The directory you have just created is the resource directory introduced in the overview.

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. In particular because sources assumes that much, the raw data should reside in a subdirectory of the resource directory, and unless you have a good reason to break with that convention (e.g., separating several data releases would be such a good reason), just call it `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
```

Starting from Scratch

(You can skip this when you are just following the quick start).
When you do not have the luxury of having a ready-made RD (i.e., almost always), use `dachs start`. Again, begin by creating your resource directory, change into it and build a q.rd template by running `dachs start <protocol>`. For instance, when you have a catalogue called smartdata, you would write:

```
cd /var/gavo/inputs
mkdir smartdata
cd smartdata
dachs start scs
```

Use `dachs start list` to see what templates there are (scs, siap, and ssap are the most likely candidates for VO beginners).
When your data does not fit anything offered by `dachs start`, you probably want to send mail to dachs-support describing what you are trying to do; perhaps we want to extend `dachs start`. Also, have a look at the GAVO data centre's service roster if there's anything similar there and then see if starting from the corresponding RD in our RD repository looks like it will save you work.
Either way, you will end up with an RD in your resource directory, and that would be the perfect time to check it into your version control system. Don't go on without one.
After bringing in your data, the actual work begins: Filling in the global metadata, the table structure, the ingestion rules, service metadata, and regression tests. Where possible, the templates are written such that when you have filled out everything `%between percent signs%`, you should have a very basic (possibly useless) service. See also the section in *Publishing Data Using VO Protocols* corresponding to your data type (if it exists) for more material that may help you figure out what is going on in the template and what additional declarations you may want to put in.
The envisioned workflow is that you go through the file percent pattern by percent pattern, ideally with support from your editor. As an example, if you're using the vim editor, you can drop the following into your ~/.vimrc:

```
augroup rd
  au!
  au BufRead,BufNewFile *.rd imap <F8> <ESC>/%[^%]*%<CR>a
  au BufRead,BufNewFile *.rd imap <F9> <ESC>cf%
augroup END
```

Then, when you open an RD file, you can, in insert mode, alternately hit F8 to go to the next thing to fill in, and F9 to replace the template text (that usually gives you a hint on what to put there – but also read the comments).

Another hint on working with the templates: Since the patterns are not valid values for what they stand in for in some cases, you may want to comment out what you haven't yet filled out (e.g., while doing a test import). Since there are comments in the material, that's not straightforward, but you can fairly simply comment a large part of RD by inserting `<macDef name="todo"><![CDATA[` at the start of the ignored area and `]]></macDef>` at its end. Of course, this needs to respect XML and RD rules, so the resulting `macDef` element must be a direct child of `resource` (which mostly is natural).

If you have skimmed the DaCHS Basics above, not much of what you'll be seeing should surprise you. If you're wondering about a particular thing, you can always refer to the elemref.html to see how it's used in our data centre, or go to the reference documentation. Also, please give us hints if you see a way to make the material more helpful.

If your input data already comes with metadata (FITS binary table, VOTable, or perhaps you have a VizieR-style byte-by-byte description), you may want to look into using `dachs gencol`; this emits column definitions that can be pasted into a table element (in the case of VizieR, also material for Element columnGrammar). Don't be content with what the machine gives you: in all cases we have seen so far, the metadata extracted from data was sub-standard.

Also note that the the gencol output for FITS binary tables assumes you use fitsTableGrammar with `lowerKeys="True"`.

Running the Ingestion, Getting Stats

Back with arihip, we're ready for ingestion, which is covered by the `import` subcommand (cf. Invoking DaCHS). So, return to the resource directory and run the default import:

```
cd ..
dachs 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.

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

```
dachs info arihip/q#main
```

If this command's argument looks confusing to you, please review Referencing in DaCHS.

This will output several properties (right now, minimum, maximum, average) of numeric columns, which may help spot problems in the data or the row maker (or both). 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.

Once everything looks right, tell DaCHS to inspect your table and collect column statistics, space-time coverage, and similar metadata on it[4]:

```
dachs limits q
```

Note: if installing from Debian 11 Bullseye, the current dachs version is 2.3 and `dachs limits q` will not work. See the error `module 'gavo.utils' has no attribute 'pyparsingWhitechars'` here https://docs.g-vo.org/DaCHS/commonproblems.html for a solution.

For large tables, this can run for a long time; to make good for it, DaCHS samples increasingly less of the tables as they get larger (which might still take hours). To override DaCHS' wisdom here, both `limits` and `info` take a `--sample-percent` option, a number between 0 and 100.

DaCHS will only do statistics on views if you pass the table id to limits or give the view's descriptor a `forceStats` property.

In some cases, you may want to skip statistics gathering on certain columns. If so, you can set the relevant columns' `statistics` property to `no`.

To inspect the limits DaCHS has figured out, say:

```
dachs limits -d q
```

Trying the services

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

```
sudo service dachs stop  # only if installed from package
dachs serve debug
```

If the last command fails with permission problems, add yourself to the gavo group, say `newgrp gavo` (or log out and in again) and try again[1].

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[2] 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. A promised in Service Definitions, apart from the host name and port (see the Configuring DaCHS on how to change those), there is the path to the RD (without the file extension), then the id of the service element, and finally the renderer name. In this case, that's `form` for an HTML form.

This systematics makes for longish URLs, which is inconvenient when you would like to include them on paper publications. To get around that, DaCHS supports **vanity names**, short aliases that let you absorb all these extra segments. In the present case, you could, for instance, add:

```
arihip/q/cone/form ARIHIP
```

to `/var/gavo/etc/vanitynames.txt` and then reach your form as http://localhost:8080/ARIHIP. For details, see The Vanity Map in the reference documentation.

By the way, a convenient backdoor to find such links without looking up the individual parts is the RD browser. To enter that, just append your RD id to the `browse` child of your server; in this case, that would be http://localhost:8080/browse/arihip/q. This will give you a convenient overview over the tables and services defined by the RD.

Another renderer supported by this service is `scs.xml`, which implements the IVOA Simple Cone Search (SCS) protocol. A client speaking SCS 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 2 degrees), and you'll see the results coming in.

In case you're missing the mv parameter you had in the form interface: cone search does not (yet) have a usable interface to discover custom parameters, and hence TOPCAT restricts you to those mandatory for every SCS service. DaCHS supports a special syntax for such "free" parameters of cone searches (see the discussion of parameter styles in Service Definitions; SCS is pql in DaCHS). To say "everything brighter than 6th magnitude", the parameter setting would be `-Inf 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=-Inf%206`.

Finally, the RD opens the arihip table for the IVOA Table Access Protocol TAP, which lets you write 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.

This is also the moment to write regression tests (yeah, it would be cool write the tests before writing tables and services, but frankly, that's something you'll only pull off if you're really familiar with both the data and the protocols; don't feel bad about writing the tests after the fact, but do feel bad if you don't write them at all) and then run:

```
dachs test -v q
```

Publishing a Service

Once you are satisified the service is in presentable shape, you should publish it.

In DaCHS, publication is an extra action. The publish element, if you will, earmarks a thing for publication. To actually perform it, you have to run:

```
dachs pub q
```

This will enter all the published services and tables in the RD q.rd into the table of published resources. For `local` publications, this means they will appear on the default front page. However, DaCHS caches the front page, and to see changes, you will have to invalidate the cache. If you have set the admin password, and it fits what the `serverURL` says, `dachs pub` will do that itself. If in doubt, just restart the DaCHS server.

Note again that before running dachs pub, it is a good idea[4] to run `dachs limits q`, which will create (or update) all kinds of statistal estimates which are being communicated to the registry and are important for discovery (e.g., coverage and some statistical measures for the columns). As of 2021, even some DaCHS-internal hacks working around deficiencies in the database server's query planner in connection with common

spherical indexing schemes rely on these statistics. Also, your publication will be more useful if you have endowed your resource with STC coverage.

For `ivo_published` services, publication means that their metadata will be served whenever DaCHS is harvested by VO registries using a protocol called OAI-PMH. DaCHS' implementation of that protocol is done such that XSLT-aware user agents (which, at least in 2020, still means: all major web browsers) can be used to (somewhat clumsily) operate it.

In practice, you can point your browser to the `oai.xml` child of your server root – by default http://localhost:8080/oai.xml –, and you will see an error message (because you've violated the protocol in this way), but you will also see a link to a list of links to the metadata of all the resources you publish. Follow this and look around; inspecting what DaCHS tells the Registry sometimes lets you spot metadata errors.

Note that for your services to show up in common VO clients, you will have to do a bit more (once). See Interacting with the VO Registry.

By the way, once you are in the Registry, there are two kinds of things that may become publicly visible without a `dachs pub` because their typical discovery bypasses the Registry:

(a) TAP tables. They will be entered into TAP_SCHEMA immediately, and TAP clients will find it there at once. Also, many clients still use GloTS, which, once it has re-harvested your server, will make your TAP table globally discoverable. If you don't want this, write `adql="Hidden"` in the table head and change it to `adql="True"` when you publish your data.

(b) Obscore datasets. When you put products into the obscore table, they will immediately appear there, and, for instance, all-VO obscore searches will find them. The only way to avoid that at this point is to comment out obscore mixins pre-publication and then comment them in when you publish.

The recommended way to deal with both problems is to have a development or test server that is non-public (and perhaps even has a global access restriction) and only take the data and RDs to the production server when you go for publication.

Chapter 4

Publishing Data Using VO Protocols

This chapter gives somewhat more in-depth information on what to do to create services compliant to the various DAL protocols by data product type. While `dachs start` should give you a fair chance of getting a service running without reading any of this, it is still a good idea to read the section for the sort of data you want to publish before setting out.

DAL is VO-speak for "Data Access Layer", the standard protocols the VO defines for data discovery and access. 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

Publishing Catalogues via SCS (Cone Search)

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 the DaCHS Basics is a combined SCS/form service. This section just briefly recapitulates what was discussed there. For a quick start, just follow Quick start with DaCHS.

SCS Tables

SCS can expose any table that has 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#pgs-pos-index mixin on the tables, like this:

```
<table id="forSCS" onDisk="true" mixin="//scs#pgs-pos-index"> ...
```

The "pgs" in pgs-pos-index refers to pgSphere, a postgres database extension for spherical geometry. You may see RDs around that use the //scs#q3cindex mixin instead here. It does the same thing (dramatically speed up spatial queries) but uses a different scheme. It's faster and takes up less space, but it's also less general, which is why we are trying to phase it out. Only use it when you are sure you cannot afford the (reasonable, i.e., mostly within a factor of two) cost of pgSphere.

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) discussed above. To ensure that, these columns' `verbLevel` attribute must be 10 or less (we advise to have it at 1).

SCS Cores

SCS could work with a dbCore, but friendly cone search services include a field with the distance between the object found and the position passed in; this is added by the special element scsCore.

You (in effect) must include the some pre-defined condDescs that make up the SCS protocol, like this:

```
<scsCore queriedTable="main">
  <FEED source="//scs#coreDescs"/>
</scsCore>
```

This will provide the RA, DEC, and SR parameters for most renderers. The form renderer, however will show a nice input box that lets humans enter object names or, if they cannot live without them, sexagesimal positions (if you are curious: this works by setting the `onlyForRenderer` and `notForRenderer` attributes on Element inputKey).

In addition, //scs#coreDescs gives you a paramter MAXREC to limit or raise the number of matches returned. This parameter is not required by SCS, but it is useful if people with sufficient technical skills (they'll need those because common SCS clients don't support MAXREC yet) want to raise or lower DaCHS' default match limit (which is configured in `[ivoa]dalDefaultLimit` and can be raised up to `[ivoa]dalHardLimit`).

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 might want to use active tags:

```
<dbCore id="xlcore" queriedTable="main">
  <FEED source="//scs#coreDescs"/>
  <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 such additional parameters, since for SCS this requires going through the Registry. In TOPCAT, for example, users would have to manually edit the cone search URL.

Also note that SCS does not really define the syntax of these parameters, which is relevant because most of the time they will be float-valued, and hence you will generally

need to use intervals as constraints. The interval syntax used by the SCS renderer is DALI, so a bounded interval would be `22.3 30e5`, and you'd build half-bounded intervals with IEEE infinity literals, like `-Inf -1`. Of course, when accessed through a form, the usual VizieR parameter syntax applies.

SCS Services

To expose that core through a service, just allow the scs.xml renderer on it. With the extra human-oriented positional constraint and mainly builtFrom condDescs, you can usually have a web-based form interface for free:

```
<service id="cone" allowed="scs.xml,form">
  <meta name="title">Nice Catalogue 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>
  <scsCore queriedTable="main">
    <FEED source="//scs#coreDescs"/>
    <LOOP listItems="ipix bmag rmag jmag pmra pmde">
      <events>
        <condDesc buildFrom="\item"/>
      </events>
    </LOOP>
  </scsCore>
</service>
```

The meta information given is used when generating registry records; the idea is that a query with the ra, dec, and sr you give actually returns some data.

Publishing Images via SIAP

SIAPv2 as described here is only available in DaCHS 2.7.3 and later. If you want to create new image services, please make sure you update to that.

In the VO, there are currently two versions of the Simple Image Access Protocol SIAP. While DaCHS still supports SIAP version 1, you should only use it for new services if you know exactly what you are doing. Hence, this tutorial (and, actually, the reference documentation, too) only talks about SIAP2.

To generate a template RD for an image collection published through SIAP, run:

```
dachs start siap
```

See Starting from Scratch for a discussion on how to fill out this template.

While you can shoehorn DaCHS to pull the necessary information from many different types of images, anything but FITS files with halfway sane WCS headers is going to be fiddly – and of course, FITS+modern WCS is about the only thing that will work nicely on all relevant clients.

If you have to have images of a different sort, it is probably a good idea to inquire on the dachs-support mailing list before spending a major effort on local development.

Quick Sample Image Service

Check out a sample resource directory:

```
cd ‘dachs config inputsDir‘
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
curl -FLANG=ADQL -FFORMAT=votable/td \
  -FQUERY="SELECT TOP 5 access_url FROM emi.main WHERE weighting='uniform'" \
  http://dc.g-vo.org/tap/sync \
| tr '<>' '\n' \
| grep getproduct \
| xargs -n1 curl -sO
```

(no, this is no way to operate TAP services; use a proper client for real work, and we didn't show you this).
This RD also publishes to obscore, so make sure you have the obscore table:

```
dachs imp //obscore
```

If you do not plan to publish via obscore yourself (which is reasonably unlikely) and you try this on a box that the Registry will see later (you shouldn't), be sure to `dachs drop` obscore again when done.
Run the import:

```
cd ..
dachs 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.

Defining SIAP Tables

SIAP-capable tables should mix in `//siap2#pgs`. This mixin provides all the columns necessary for valid SIAP responses, and it will prepare the table so that spatial queries (which are the most common) will use a pg_sphere index.
So, in the simplest case, a table published through a SIAP service would be declared like this:

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

This only has the minimal SIAP metadata. The net result of this is that your new `images` table will have all the mandatory columns of Obscore. You can add your own, custom columns as usual.
The //siap#pgs mixin also takes care that anything added to the table also ends up in the products table. This means that the grammar filling the table needs a //products#define rowfilter.

Filling SIAP Tables

When filling SIAP tables, you will almost always use:

- Element fitsProdGrammar as the grammar; see also fitsProdGrammars further down.
- the //siap2#computePGS apply to interpret the FITS WCS to fill the spatial columns in the table.
- the //siap2#setMeta apply to fill the non-spatial SIAP metadata.

In practice, this might look like this (taken from the emi/q RD from the quick start):

```
<data id="import_main">
  <sources recurse="True">
    <pattern>*.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="__system__/products#define">
      <bind key="table">"emi.main"</bind>
    </rowfilter>
  </fitsProdGrammar>

  <make table="main" >
    <rowmaker id="gen_rmk" idmaps="obsra, obsdec">
      <apply name="fixObjectName">
        <setup imports="csv">
          <code>
            with open(rd.getAbsPath("res/namemap.csv")) as f:
              nameMap = dict(csv.reader(f))
          </code>
        </setup>
        <code>
          @target_name = nameMap[@object]
        </code>
      </apply>

      <apply procDef="//siap2#computePGS"/>
      <apply procDef="//siap2#setMeta">
        <bind name="em_min">0.207</bind>
        <bind name="em_max">0.228</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="t_min"
          >dateTimeToMJD(datetime.datetime(2010, 7, 4))</bind>
        <bind name="t_max"
          >dateTimeToMJD(datetime.datetime(2010, 7, 4))</bind>
        <bind name="obs_title">"VLBA 1.4 GHz "+@target_name</bind>
        <bind name="calib_level">3</bind>
        <bind name="obs_collection">"VLBA LH sources"</bind>
        <bind name="o_ucd">"phot.flux.density;em:radio.750-1500MHz;phys.polarisation.Stokes.I"</bind
        <bind name="pol_states">"/I/"</bind>
        <bind name="pol_xel">1</bind>
        <bind name="target_name">@target_name</bind>
        <bind name="t_resolution">5000000</bind>
        <bind name="target_class">"agn"</bind>
      </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 often comes in particularly handy.
- When ingesting images, you will very typically read from FITS primary headers. That is what element fitsProdGrammar does unless told otherwise: Its rawdicts simply are the (astropy.io.fits) headers turned into plain python dictionaries.
- The `qnd` attribute of the grammar is recommended as long as you get away with it. It makes some (weak) assumptions to yield significant speedups, but it limits you to the primary header. You cannot use `qnd` with compressed FITS images. Also, note the `hdusField` attribute when you have more complex FITSes to process.
- The `fitsProdGrammar` will map keys with hyphens to names with underscores, which allows for smoother action with them in rowmakers. The `mapKeys` 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 key mapping, really.
- Since we are defining a table serving data products here, the grammar needs the //products#define rowfilter discussed in the products table.
- We have mentioned the //siap2#computePGS apply above; as long as astropy can deal with your WCS, it's all automatic (though you may want to pass parameters when you have cubes with suboptimal WCS or want to keep products without WCS in your table). And if you don't have proper WCS: see above on checking with dachs-supoort.
- The second apply you want when feeding SIAP tables is //siap2#setMeta. This has a lot of parameters, because depending on what you are publishing – SIAP2

explicitly was designed to push out cubes, too – you may want to give metadata like the length of a wavelength axis. `dachs start siap` puts in default mappings for most of them; feel free to delete them if they don't make sense for your kind of data.

- There are two non-obscore parameters in setMeta. For one, instead of `em_min` and `em_max`, you can set bandpassId and have the //siap#getBandFromFilter look them up. Whether a band is known you can find out by running `dachs admin dumpDF data/filters.txt` – and we are grateful for further contributions to that file.

- You can also set `dateObs` to the central time of the observation and give `t_exposure` a sensible value in seconds. DaCHS will then compute `t_min` and `t_max` for you, assuming a single, contiguous observation.

- Typically, many values you find 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 many other situations, the //procs#mapValue or //procs#dictMap applys let you do fixes with less code.

- As is usual in DaCHS procdures, you can access the embedding RD as `rd`. In our object name fixer, we use that to let DaCHS find the input file independently of where the program was started.

Somewhat regrettably, //siap2#setMeta cannot be used with `idmaps="*"`; as settings from setMeta would be overwritten then. That's a compromise for backwards compatibility.

Cores

For SIAP1, there was the Element siapCutoutCore that returned cutouts rather than full images. Nothing of this sort is possible for SIAP2, because spatial constraints are optional for the latter. If you really need that functionality, you'll either have to go back to SIAP1 or (probably preferably, although client support isn't all that great yet) return datalinks rather than full images (see SIAP and datalink).

That said, the core to use for SIAP2 is the Element dbCore. Add the condDescs necessary for SIAP manually (and add any custom condDescs as you see fit). See the next section for an example.

Service

The service uses the siap2.xml renderer, for which you need some additional metadata for VO registration as described in the siap2.xml renderer.

With this, a service definition would look like this (again taken from emi/q):

```
<service id="s" allowed="form,siap2.xml">
  <meta name="shortName">VLBI-Lockman</meta>
  <meta name="sia.type">Pointed</meta>
  <meta name="testQuery.pos.ra">163.3</meta>
  <meta name="testQuery.pos.dec">57.8</meta>
  <meta name="testQuery.size.ra">0.1</meta>
  <meta name="testQuery.size.dec">0.1</meta>
```

```
  <publish render="siap2.xml" sets="ivo_managed"/>

  <dbCore queriedTable="main">
    <FEED source="//siap2#parameters"/>
    <condDesc buildFrom="weighting"/>
  </dbCore>
</service>
```

You probably do not want to point real users at the from rendering of this service. If you want to have image services working in the browser, write an extra service; the output of `dachs start siap` has a starting point for that.

SIAP and Obscore

The SIAP2 metadata schema is exactly the one of Obscore. Hence, to publish a SIAP table to Obscore, all you need to say is:

```
<mixin>//obscore#publishObscoreLike</mixin>
```

in your table element.
Note that for tables of non-trivial size, you really should have a spatial index on s_ra and s_dec (our advice: q3c) and on s_region (that will have to be a pg_sphere one) and otherwise index at least obs_publisher_did, em_min, em_max, t_min, and t_max.

SIAP and Datalink

If you have larger images or cubes and serve them through SIAP, consider offering datalinks or perhaps even have Datalinks as Products. The latter case is particularly attractive if your images are so large that people just clicking on some row in Aladin might not expect a download of that size (in 2020, I'd set that limit at, perhaps, 100 MB). In both cases, people can select and download only parts of the image or request scaled versions of it.
Defining a datalink service for normal FITS images is not hard. In the simplest case, you just give a bit of metadata, use the //soda#fits_genDesc descriptor generator (you don't need to understand exactly what that is; if you are curious: Datalink and SODA has the full story) and FEED //soda#fits_standardDLFuncs. Done.
The following example, taken from lswscans/res/positions, adds a fixed link to a scaled version, which might work a bit smoother with unsophisticated Datalink clients, using a meta maker:

```
<service id="dl" allowed="dlget,dlmeta">
  <meta name="title">HDAP Datalink</meta>
  <meta name="description">This service lets you access cutouts
    from HDAP plates and retrieve scaled versions.</meta>
  <datalinkCore>
    <descriptorGenerator procDef="//soda#fits_genDesc">
      <bind key="accrefPrefix">lswscans</bind>
    </descriptorGenerator>
     <FEED source="//soda#fits_standardDLFuncs"/>

    <metaMaker semantics="#science">
      <code>
        yield descriptor.makeLink(
          makeProductLink(descriptor.accref+"?scale=4"),
          contentType="image/fits",
```

```
            description="FITS, scaled by 1/4",
            contentLength=descriptor.estimateSize()/16.)
        </code>
      </metaMaker>
    </datalinkCore>
  </service>
```

See Meta Makers for more information on what is going on inside the meta maker. The remaining material is either stereotypical or pure metadata: title and description are as for any other serivce, and the `accrefPrefix` should in general reflect your resource directory name. DaCHS' datalink machinery will reject any publisher DID asking for an accref not starting with that string. The idea here is to avoid applying code to datasets it is not written for.

When you attach the datalink functionality (rather than having datalink links as access URLs), declare your table as having datalink support; both SIAP and TAP will pick that up and add the necessary declarations so datalink-aware clients will know they can run datalink queries against your service:

```
<meta name="_associatedDatalinkService">
  <meta name="serviceId">dl</meta>
  <meta name="idColumn">pub_did</meta>
</meta>
```

This block needs to sit in the table element. The serviceId meta contains the id of the datalink service.

If you also produce HTML forms and tables, see datalinks in columns.

Publishing Spectra or Time Series via SSAP

Publishing spectra is harder than publishing catalogues or images; for one, the Simple Spectral Access Protocol comes with a large bunch of metadata, quite a bit of which regrettably repeats VOResource. And there is no common format for spectra, just a few contradicting loose conventions.

That is why `dachs start` produces a template that contains an embedded datalink service. This lets you push out halfway rational VOTables that most interesting clients can reliably deal with, while still giving access to whatever upstream data you got.

In the past, we have tried to cope with the large and often constant metadata set of SSAP using various mixins that have a certain part of the metadata in PARAMs (which is ok by the standard). These were, specifically, the mixins `//ssap#hcd` and `//ssap#mixc`. *Do not use them any more* in new data and ignore any references to them in the documentation.

The modern way to deal with SSAP – both for spectra and for time series – is to use the //ssap#view mixin. In essence, this is a relatively shallow way to map your own metadata to SSA metadata using a SQL view. This is also what the `dachs start` template does.

Quick Sample SSAP Service

Check out the feros resource directory into your inputs directory:

```
cd ‘dachs config inputsDir‘
svn co http://svn.ari.uni-heidelberg.de/svn/gavo/hdinputs/feros
cd feros
mkdir data
```

As recommended, the checkout does not contain actual data, so let's fetch a file:

```
cd data
curl -O http://dc.g-vo.org/getproduct/feros/data/f04031.fits
cd ..
```

This RD also publishes to obscore, so make sure you have the obscore table:

```
dachs imp //obscore
```

If you do not plan to publish via obscore yourself (which is reasonably unlikely) and you try this on a box that the Registry will see later (you shouldn't), be sure to `dachs drop` obscore again when done.
Run the input and the regression tests:

```
dachs imp q
dachs test q
```

One regression test should fail since you've not yet pre-generated the previews (which are optional but recommended for your datasets, too):

```
python3 bin/makepreviews.py
dachs test q
```

If the regressions tests don't pass now, please complain to the authors of this tutorial.
From here on, you can point your favourite spectral client (fallback: TOPCAT's SSA client; note that TOPCAT itself cannot open this service's native format and you'll have to go through datalink, which TOPCAT knows how to do since about version 4.6) to http://localhost:8080/feros/q/ssa/ssap.xml and do your queries (if you don't know anything else, do a positional query for 149.734, 2.28216).
Please drop the dataset again when you're done playing with it:

```
dachs drop feros/q
```

Since the dataset is in the obscore table, it would otherwise be globally discoverable, and that'd be bad.

SSAP Tables

Contrary to what DaCHS does with the relatively small SCS, SIAP, and SLAP models, due to the size of the SSAP model, spectral services are always based on a database view on top of a table the structure of which is controlled by you; you're saving quite a bit of work if you keep your own table's columns as close to their SSA form as possible, though.
Another special situation is that most spectra are delivered in fairly crazy formats, which means that it's usually a good idea to have a datalink service that serves halfway standard files – in DaCHS, these comply to the VO's spectral data model, which is a VOTable with a bit of standard metadata. It's certainly not beautiful or very sensible, but it sure beats the IRAF-style 1D images people still routinely push around.
So, to start a spectral service, use the `ssap+datalink` template:

```
$ mkdir myspectra; cd myspectra
$ dachs start ssap+datalink
```

This will result in the usual `q.rd` file as per starting from scratch; see there for how to efficiently edit this and for explanations on the common metadata items.

SSAP-specific material starts at the meta definitions for `ssap.dataSource` and `ssap.creationType`. These are actually used in service discovery, so you should be careful to select the right words. Realistically, using `survey`/`archival` for observational data and `theory`/`archival` for theoretical spectra should be the right thing most of the time.

Next, you define the `raw_data` table; this should contain all metadata "unpredictably" varying between datasets (or, for large data collections, anything that needs to be indexed for good performance). For instance, for observational data, the observed location is going to change from row to row. The start and the end of the spectrum is probably going to be fixed for a given instrument, and so if you have a homogeneous data collection you probably will not have columns for them and rather provide constant values when defining the view.

To conveniently define the table, it is recommended to pull the SSA columns for `raw_data` by name from DaCHS' SSAP definitions and use SSAP conventions (e.g., units). The generated RD is set up for this by giving `namePath="//ssap#instance"`, which essentially means "if someone requests an element by id, first look in the `instance` table of the RD". This is then used in the following `LOOP` (cf. Active Tags). As generated, this will look somewhat like:

```
<LOOP listItems="ssa_dateObs ssa_dstitle ssa_targname ssa_length
  ssa_specres ssa_timeExt">
```

– this will pull in the enumerated columns as if you had defined them literally in the table. Depending on the nature of your data, you may want to pull in more columns if they vary for your datasets (or throw out ones you don't need, as `ssa_dateObs` for theoretical data).

To see what is available, refer to the reference documentation of the the //ssap#view mixin. Any parameter that starts with `ssa_` can be pulled in as a column.

The template RD then mixes in `//products#table` (which you pretty certainly want; see The Products Table for an explanation), `//ssap#plainlocation` (which at this point you must have to make a valid SSA service) and `//ssap#simpleCoverage` (which you want if you want to publish your observational spectra through obscore). The template then defines:

```
<FEED source="//scs#splitPosIndex"
  long="degrees(long(ssa_location))"
  lat="degrees(lat(ssa_location))"/>
```

This again is mainly useful for obscore as long as DaCHS' ADQL engine may turn queries into q3c statements; just leave it if you have positions, and remove it if you don't.

You can, of course, define further columns here, both for later use in the view and for local management. SSAP lets you return arbitrary local columns, and in particular for theory services, you will have to (to define the physics of your model). As a DaCHS convention, please don't use the `ssa_` prefix on your custom columns. See theossa/q for an example of a table with many extra columns.

The SSA template then goes on with a data item filling the `raw_data` table. The template assumes you're parsing from IRAF-style 1D images. You will have to use a different grammar if that is not what you have, and in that case you in particular you cannot use the `specAx` var defined in the rowmaker.

The data item has `<recreateAfter>make_view</recreateAfter>` quite early on; this simply makes sure that the SSA view will be regenerated after you import the table itself.

The rowfilter in the grammar is fairly complex here because we will completely hide the original; if you simply want to serve your upstream format, just cut it down to just giving `table`, `mime`, `preview` and `preview_mime`. If you do that, use the following strings in `mime`:

- `image/fits` for IRAF-style 1D image spectra
- `application/fits` for spectra in FITS tables
- `application/x-votable+xml` for spectra in VOTables

Please do not put anything else into SSA tables, because you will most certainly overstrain most SSA clients; if you have a different upstream format and you want to make it available, turn it into SDM VOTables and use datalink to link to the original source.

Hence, for most cases (including also ASCII spectra), here's what we recommend as the product definition rowfilter (it's roughly what's in the template) to isolate your clients from the odd upstream formats:

```
<rowfilter procDef="//products#define">
  <bind name="table">"\\schema.main"</bind>
  <bind name="path">\\fullDLURL{"sdl"}</bind>
  <bind name="fsize">%typical size of an SDM VOTable%</bind>
  <bind name="datalink">"\\rdId#sdl"</bind>
  <bind name="mime">"application/x-votable+xml"</bind>
  <bind name="preview">\\standardPreviewPath</bind>
  <bind name="preview_mime">"image/png"</bind>
</rowfilter>
```

This is pointing the path accessed to a datalink URL using the `fullDLURL` macro, which expands to a URL retrieving the full dataset; the "sdl" argument to the macro references the datalink service defined further down. Since the data returned is generated on the fly, you will have to give an estimate of how large the VOTable will be (overriding DaCHS' default of the size of the source file). Don't sweat this too much, just don't claim something is 1e9 bytes when you're really just returning a few kilobytes. The rowfilter expects the size in bytes.

The bindings already prepare for making and serving previews, which is discussed in more detail in Product Previews in the DaCHS reference; see there for everything mentioning "preview".

SSAP has a feature that lets users request certain formats, and for clients that don't know Datalink, this *may* be a good idea. In that scheme, you use a rowfilter to return a description of your native data *and* the processed SDM-compliant dataset as used here. See theossa/q for an example how that would look like. Our recommendation: don't bother, it's a misfeature that will most likely just confuse your users.

The rowmaker is fairly standard; we should perhaps mention the elements:

```
<var name="specAx">%use getWCSAxis(@header_, 1) for IRAF FITSes
  (delete otherwise)%</var>
<map key="ssa_specstart">%typically, @specAx.pixToPhys(1)*1e-10%</map>
<map name="ssa_length">%typically, @specAx.axisLength%</map>
```

`getWCSAxis` is a function that looks at a FITS image's WCS information to let you transform pixel to physical coordinates. This currently uses a simplified DaCHS implementation that only does a small part of WCS (but we may change that, keeping the interface stable). The `var`, anyway, binds the resulting object to `specAx`. You can use that later to find out the limits of the spectrum. The way it is written here, you will still have convert the value to metres manually. But as said above, if you're publishing a homogeneous collection of spectra, both values are probably constant, and you'll want to remove both maps from the template.

The SSA View

The template goes on defining the `data` table that will serve as the basis of the service. It starts with the declaration:

```
<meta name="_associatedDatalinkService">
  <meta name="serviceId">sdl</meta>
  <meta name="idColumn">ssa_pubDID</meta>
</meta>
```

This is lets DaCHS add a link to the datalink service to results generated from this (both via SSAP and TAP). There's nothing you need to change here (unless you chuck datalink); see SSAP and Datalink and Datalink for details.

The main body of the table definition is the //ssap#view mixin. In it, you need to write the SSA parameters as SQL literals (i.e., strings need single quotes around them) or expressions involving column references. To keep things fast, you should have SSA-ready columns in the source table, so you will usually have column references only here. Most of these items default to NULL, so if you do not have a piece of metadata, it is reasonably safe to just remove the attribute.

A few of the mixin's parameters deserve extra discussion:

- `sourcetable` – this is a *reference*, i.e., this must resolve to the `id` of some table element. It can be cross-RD if really necessary. It is *not* the SQL table reference (that would include a schema reference).
- `copiedcolumns` – this lets you copy over columns from the source table, i.e., the one you just defined using (comma-separated) shell patterns of their names (yes, that's just like `idmaps` in rowmakers). The `*` given in the template should work in most cases, but if you have private columns in the source table, you can suppress them in the view; a useful convention might be to start all private columns with a p; you'd then say `copiedcolumns="[^p]*"`. Note that copied columns are automatically added in the view as 1:1 maps, and you cannot use view arguments to override them. Use different column names in the source table and the view if you (think you) have to do view-level processing of your values.
- `customcode` – use this if you have extra material to enter in the view definition generated by the mixin. We hope you won't need that and would be interested in your use case if you find yourself using this.

- `ssa_spectralunit`, `ssa_fluxunit` – these are the only mandatory parameters starting with `ssa_` (but their values are still overwritten if they are in copiedcolumns). There really is no point in having them vary from row to row because their values are metadata for the corresponding error columns (which is one of the many spec bugs in SSAP).
- `ssa_spectralucd`, `ssa_fluxucd` – these are like the unit parameters in that they contain data collection-level metadata. The only reason they are not mandatory is that there are defaults that seem sensible for a large number of cases. Check them, and again, you cannot really let them vary from row to row.
- `ssa_fluxSI`, `ssa_spectralSI`, `ssa_timeSI` – these were an attempt to work around a missing specification for unit strings in the VO. Since we now have VOUnit, just ignore them.

The data item making this table is trivial. You should set it to `auto="False"` (i.e., don't build this on an unadorned `dachs imp`). The building of this data will normally be triggered by the `recreateAfter` of the source table import.

SSAP Services

Use the element ssapCore for SSAP services. You must feed in the condition descriptors for the SSAP parameters you want to support (some are mandatory). The simplest way to do that is to `FEED` the //ssap#hcd_condDescs stream. It includes condition descriptors for all mandatory and optional parameters that we can remotely see in use.
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. We consider that an unfortunate recommendation that you should ignore; if someone queries your theoretical service with a position, it is highly likely they do not want to see all your spectra.
If you nevertheless 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>
```

Again, do not do this just because you don't have, say position information.
Here is a table of parameter names and ids; you can always check them by inspecting the output of `dachs adm dumpDF //ssap`:

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`. This is in keeping with the general SSAP parameter style, while more modern VO services use 2-arrays for intervals ("DALI style"). To expose SSAP cores, use the ssap.xml renderer.

Form-based Spectral services

Using the form renderer on SSAP cores is not terribly useful, because the core returns XML directly, and there are far too many parameters no human will ever be interested in anyway. Hence, you will typically define extra browser-based services. The example RD shows a compact way to do that:

```
<service id="web" defaultRenderer="form">
  <meta name="shortName">\\schema Web</meta>

  <dbCore queriedTable="main">
    <condDesc buildFrom="ssa_location"/>
    <condDesc buildFrom="ssa_dateObs"/>
    <condDesc>
      <inputKey original="data.ssa_targname" tablehead="Star">
        <values fromdb="ssa_targname from theossa.data
          order by ssa_targname"/>
      </inputKey>
    </condDesc>
  </dbCore>

  <outputTable>
    <autoCols>accref, mime, ssa_targname,
      ssa_aperture, ssa_dateObs, datalink</autoCols>
    <FEED source="//ssap#atomicCoords"/>
    <outputField original="ssa_specstart" displayHint="displayUnit=Angstrom"/>
    <outputField original="ssa_specend" displayHint="displayUnit=Angstrom"/>
  </outputTable>
</service>
```

Essentially, we only select a few fields people might want to query against, and we directly build them out of the query fields; the SSA condDescs are bound to the funny and insufficiently defined SSA input syntax and probably not very useful in interactive applications.

The extra selector for object names with the names actually present in the database is a nice service as long as you only have a few hundred objects or so. Since the query over `ssa_targname` is executed at each load of the RD, it should be fast, which means that even for medium-sized tables, you should have an index on the object names in the `raw_data` table, probably like this:

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

On DaCHS newer than 2.5.1, instead of `fromDb`, it is usually better to say:

```
<column original="ssa_targname">
  <property key="statistics">enumerate</property>
</column>
```

in the table element. This will make `dachs limits` create the necessary metadata once and not on every RD reload. Note that DaCHS will only create statistics for views if the `forceStats` property on the table definition is set. However, if you create statistics on the metadata table, these will be inherited by columns copied over into the view, which means that you will in general not have to do this.
In the output table, we only give a few of the many dozen SSAP output fields, and we change the units of the spectral limits to Angstroms, which will look nicer for optical spectra. For educational reasons you might want to change this to nm (nanometer).
In the template, this form-based service is published as a capability of the SSA service. This is done using the service attribute in the Element publish *in the SSAP service element*:

```
<publish render="form" sets="ivo_managed,local" service="web"/>
```

See Registering Web Interfaces to DAL Services for more background.

SSAP and Obscore

The SSA metadata is not far from the Obscore metadata (cf. publishing anything through obscore), and so an Obscore publication of SSAP data almost comes for free: Minimally, just mix in //obscore#publishSSAPMIXC and set `calibLevel`. The template does a bit more:

```
<mixin
  calibLevel="%likely one of 1 for uncalibrated or 2 for uncalibrated data%"
  coverage="%ssa_region -- or remove this if you have no ssa_region%"
  sResolution="ssa_spaceres"
  oUCD="ssa_fluxucd"
  emUCD="ssa_spectralucd"
  >//obscore#publishSSAPMIXC</mixin>
```

– if you use one of the old hcd or mixc mixins, you do not want oUCD and emUCD.
In particular for larger spectral collections, it is highly recommended to also have the //ssap#simpleCoverage mixin in an obscore-published spectral table; only then will you get indexed queries when there are constraints on s_region, and having these non-indexed will lead to really slow obscore queries.
Similarly, you should probably add:

```
<FEED source="//ssap#obscore-time-index"/>
```

to large SSAP tables, which creates an index useful for obscore queries over `t_min` and `t_max` (cf. //ssap#obscore-time-index).
Whenever the spectral coverage is not constant on large-ish SSAP tables, you should also have:

```
<index columns="ssa_specstart"/>
<index columns="ssa_specen"/>
```

SSAP and Datalink

Given that you will usually get fairly bizarre inputs and will probably want to publish "repaired" spectra, using Datalink to provide both native and SDM ("Spectral Data Model") compliant spectra without having to resort to SSAP's ill-thought-out FORMAT feature is a fairly natural thing to do. That is why the SSAP+datalink template comes with almost all that you need to do that; what is left is mainly to write an embedded grammar to parse the spectra (if the parsing is complex, you might to go for an Element customGrammar, which lets you keep the source outside of the RD). Other than that, it is just a few formalities.

So, you first define the table that will later hold your spectrum. Use the //ssap#sdm-instance mixin for that (this continues material from the template):

```
<table id="instance" onDisk="False">
  <mixin ssaTable="main"
    spectralDescription="%something like 'Wavelength' or so%"
    fluxDescription="%something like 'Flux density' or so%"
    >//ssap#sdm-instance</mixin>
  <meta name="description">%a few words what a spectrum represents%</meta>
</table>
```

The descriptions you need to enter here typically end up being labels on axes of spectral plots, so it is particularly important to be concise and precise here.

If your spectrum has additional columns (e.g., errors, noise estimates, bin widths), just put more columns in here. The mixin pulls in all the various params that SDM wants to see from the row of the spectrum in the SSAP table.

Note that the table does not have `onDisk="True"`; these tables are only made for the brief moment it takes to serialise them into what the user receives.

As usual, to fill tables, you want a data element. The template just gives a few hints on how that might work. As a working example, zcosmos/q parses from 1D FITS images like this:

```
<data id="build_sdm_data" auto="False">
  <embeddedGrammar>
    <iterator>
      <setup imports="gavo.protocols.products, gavo.utils.pyfits"/>
      <code>

      fitsPath = products.RAccref.fromString(
          self.sourceToken["accref"]).localpath
      hdus = pyfits.open(fitsPath)
      ax = utils.getWCSAxis(hdus[0].header, 1)

      for spec, flux in enumerate(hdus[0].data[0]):
        yield {"spectral": ax.pix0ToPhys(spec), "flux": flux}
      hdus.close()
      </code>
    </iterator>
  </embeddedGrammar>
  <make table="spectrum">
    <parmaker>
      <apply procDef="//ssap#feedSSAToSDM"/>
    </parmaker>
  </make>
</data>
```

The way this figures out the file from which to parse will work if you have the actual file path in the product table. When you hide the upstream format as recommended, you have to follow some custom convention. The SSAP+datalink template has:

```
sourcePath = urllib.decode(
  self.sourceToken["ssa_pubDID"].split('?', 1)[-1])
```

This works if you use the macro standardPubDID as in the template.
But you can do arbitrary things here; see the califa/q3 RD for an example for how you can decode the accref to database rows. A more complex scenario with ASCII data stored externally and cached is in theossa/q, a case where multiple orders of echelle spectra are being processed in flashheros/q.
You will notice that the rowmaker in both the example and the template is missing. DaCHS will then fill in the default one, which essentially is `idmaps="*"`. Since you are writing the grammar from scratch, just use the names of the columns defined in the instance table and be done with it. The predefined column names are `spectral` and `flux`, so make sure you always have keys for them in the dictionaries you yield from your grammars.
While there is no rowmaker, the `make` does have an Element parmaker; this is stereotypical, just always use the `procDef` as here. It copies values from the SSA input row to the params in the instance table.
Finally, you would need to write the service. For SSAP and SODA, what's in the template ought to just work. Add Element metaMaker-s to provide links to, e.g., your raw input files if you want. In that case, please skim the endless chapter on Datalink and SODA in the reference documentation to get an idea of how descriptor generators, data functions, and meta makers play together.
For reasons discussed in Datalinks in Columns, it *may* be a good idea to include a custom column with a datalink URL in the SSAP table. The ssap+datalink template already has such a column in its source table and fills it in import's rowmaker.
You will usually want to touch up the instance table's metadata a bit. For example, the VOTable name of these tables by default will be constant (`instance`, usually), which will then show up, e.g., in the spectrum browser of SPLAT, where it is not terribly useful. It is hence usually a good idea to have an apply in then parmaker of your sdm instance-making data item.
Note that the SSAP row with your metadata is not in `vars` here as usual elsewhere. Instead, you will find it as the `sourceToken` on the parser. The following example, lifted from gaia/s3, ought to help you figure things out:

```
<make table="instance">
  <parmaker>
    <apply procDef="//ssap#feedSSAToSDM"/>
    <apply name="update_metadata">
      <code>
        sourceId = vars["parser_"].sourceToken["source_id"]
        targetTable.setMeta("description",
          base.getMetaText(targetTable, "description")
          +" for Gaia DR3 object {}".format(sourceId))
        targetTable.setMeta("name", str(sourceId))
      </code>
    </apply>
  </parmaker>
  <rowmaker idmaps="*"/>
</make>
```

SSAP for Time Series

As long as there is no anointed successor to SSAP explicitly catering to time series, you can use SSAP to publish time series. It is a bit of a hack, but clients like SPLAT do about the right thing with them.

As to examples, check out k2c9vst/q (which parses the data from ASCII files) and gaia/q2 (which stores the actual time series in the database, a technique we believe is a very good idea).

To notify the Registry (and possibly clients, too), that you are producing time series, do two things:

(1) Globally declare that you serve time series by setting:

```
<meta name="productType">timeseries</meta>
```

near the top of your RD. The ssap+datalink template has more information on the `productType` meta.

(2) have `'timeseries'` as ssa_dstype in your view definition.

Building Time Series VOTables

This section is under construction

Since there is no actual agreed-upon standard for the serialisation of time series, you will probably have to produce time series on the fly; DaCHS helps you to produce something that will hopefully follow standardisation efforts in the VO without *much* additional work later on, using, you guessed it, a mixin. For now, the only mixin available is for photometric timeseries: See the //timeseries#phot-0 mixin to build things. If you have other time series, please write mail to dachs-support.

To see things in action, refer to k2c9vst/q, the instance table and the corresponding makes.

However, there is an additional complication, showcased in gaia/q2 and bgds/l: it is quite common to have time series in multiple bands in one resource. For DaCHS, this is a bit of a problem, because the band influences quite a bit of the table metadata in DaCHS – this is in the mixin, and what you set there is fixed once the table instance is made.

To get around this, look at the technique shown in bgds/l. This first defines a STREAM `time-series-template` with a macros where the items very between bands:

```
<STREAM id="time-series-template">
  <table id="instance-\band_short">
    <mixin
      effectiveWavelength="\effective_wavelength"
      filterIdentifier="\band_human"
      longitude="@ra"
      latitude="@dec"
```

It then uses a LOOP to fill these slots and create one table definition per band:

```
 <LOOP>
  <csvItems>
    band_short, band_human, band_ucd, effective_wavelength
    i,          SDSS/i,     em.opt.I, 7.44e-7
    r,          SDSS/r,     em.opt.R, 6.12e-7
  </csvItems>
  <events source="time-series-template"/>
</LOOP>
```

The dispatch between the different table templates then happens in the data function of the tsdl service, using a somewhat obscure feature of `rsc.Data`: when using the `createWithTable` class function, you can pass in the table that the data item should make. This obviously only works in specialised circumstances like the one here, but then it's really convenient. So, while the make in `make_instance` claim to build `instance-i`, this is really overridden in the datalink service's data function to select the actual table definition:

```
<dataFunction>
  <setup imports="gavo.rsc"/>
  <code>
    dd = rd.getById("make_instance")
    descriptor.data = rsc.Data.createWithTable(dd,
      rd.getById("instance_"+descriptor.band))
    descriptor.data = rsc.makeData(
      dd,
      data=descriptor.data,
      forceSource=descriptor)
  </code>
</dataFunction>
```

(where `descriptor.band` has been pulled from the dataset identifier in the custom descriptor generator of that datalink service; that pattern is probably a good idea when you are in a similar situation).

This will not scale well to many dozens of bands – if you have that, you probably want somewhat more hardcore means –, but for the usual handful of bands this is a relatively reasonable way to produce time series with nice metadata.

Publishing Anything Through Obscore

"Obscore", in VO jargon, refers to a publication of datasets by putting their metadata into a TAP-queriable database table with a bespoke set of columns. It lets people pose very complex constraints, even using uploaded tables, and it is flexible enough to support almost any sort of data the typed services (SIAP, SSAP) serve and a lot more.

You may ask: Why have the S*APs in the first place? The answer is, mainly, history. Had we had TAP from the start, it is likely we had not bothered with defining standards for typed services. But that's not how things worked out, and thus client support of Obscore still is inferior to that of the typed services.

However, with a view to a future migrating towards obscore, it is certainly a good idea to publish data through obscore, too. The good news is that in DaCHS, that is generally close to trivial.

You will sometimes see something called ObsTAP mentioned. This was meant to refer to "Obscore queried through TAP", but since, really, everyone uses Obscore through

TAP, people do not say ObsTAP much any more. If you see it somewhere, pretend it is really saying Obscore.
Before you can do anything with obscore, you have to run:

```
dachs imp //obscore
```

This will also declare support for the obscore data model in your TAP service's registry record, which will make all-VO obscore queries use your service. Avoid that if you do not really publish anything through Obscore.
To drop Obscore if you have accidentally imported it, run:

```
dachs drop --system //obscore
```

Internally, the `ivoa.obscore` table is implemented as a view. If this view contains bad SQL or tables that have been dropped, working with obscore can result in rather confusing messages. If that happens, try:

```
dachs imp //obscore recover
```

This should remove the bad content from the view statement.

Obscore Derived from Typed Service Tables

Obscore defines a single table called `ivoa.obscore`. In DaCHS, that table typically contains data from a multitude of resources with different metadata structures. To keep that manageable, DaCHS implements the table as a view, where the individual tables are mapped onto that common schema. These mappings are almost always created using a mixin from the //obscore RD. Filling out its parameters will result in SQL DDL fragments that are eventually combined to the view definition. In case you are curious: The fragments are kept in the `ivoa._obscoresources` table.
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 (2017ivoa.spec.0509L).
When you start with a table underlying a typed service, you can get away with just saying something like (using SIAP as an example):

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

to the table definition's start tag. You do not have to re-import a table to publish it to Obscore when you have already imported it – `dachs imp -m <rd id> && dachs imp //obscore` will include an existing table in the obscore view.
When you import data without the `-m` flag, the mixins arrange for everything, so you do not need the extra step of importing `//obscore`.
Since the Obscore data model is quite a bit richer than SIAP's and just a bit richer than SSAP's, you will usually want to add extra metadata through the mixin, for instance:

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

Again, a `dachs imp -m` followed by an import of `//obscore` would be enough to make these changes visible in `ivoa.obscore`.
See SIAP and Obscore and SSAP and Obscore for more information on how to Obscore-publish typed data.

Obscore Notes

Dataset Identifiers

Obscore uses the concept of dataset identifiers rather extensively, and it is not unlikely that queries against the `obs_publisher_did` column will be run – not the least in connection with datalink, in which the DID has the role of something like a primary key. DaCHS' obscore-associated datalink service, for instance, will do such queries, and will be slow if postgres has to seqscan large tables for pubDIDs.

While DaCHS probably does a good job with creating usable (and globally unique) publisher DIDs, it will not index them by default. Use the `createDIDIndex` parameter of the various mixins to make one if your data collection contains more than just a few hundered entries and there is no index on it anyway.

On the other hand, the creator DID would be assigned by whoever wrote the data file, and you should not change or invent it. It was intended to let people track who republishes a given data set, weed out duplicates, and the like. Regrettably, only very few data provides assign creator DIDs, so it's probably not worth bothering.

If you are in a position in which you could make your data provider generate creator DIDs, you could make them set a good precendent. DaCHS helps you by letting you claim an authority for them (which would be the first step). See tutreg/gavo_edu_auth for an example RD that, when `dachs pub`-ed, will claim an authority for your publishing registry, and see Claiming an Authority for the background on authorities.

target_class

The obscore model has the notion of a target class for pointed observations; this is intended to cover use cases like "get me spectra of Galaxies" or so. Of course, this only works with a common vocabulary of object types, which does not actually exist in the VO at this time. The next best thing is SIMBAD's types, which are to be used until then.

s_region

Obscore has two ways to do spatial queries: using s_ra, s_dec, and perhaps s_fov on the one hand, and using s_region on the other. That is a bit unfortunate because in practice you have to have two indices over at least three columns. Also, DaCHS really likes it if columns are type-clean, and thus the mixins take quite a bit of pain to make sure only polygons are in `s_region`. Given `s_region` in our obscore has an xtype of `adql:REGION` and is thus polymorphic, you *might* get away with having other types in there. No proimises on the long term, though.

Having said all that: please make sure that whenever there is a position of some kind, you also fill s_region; this is not a problem in SIAP, but where you only have a position and aperture, in a pinch fill in something like:

```
<map key="s_region">pgsphere.SCircle.fromDALI(
   [alpha, delta, aperture]).asPoly(6)</map>
```

(the //ssap#setMeta mixin already does that when both a position and an aperture are available).

See also Creating pgSphere Geometries for more information on how to fill geometry-valued columns.

Pure Obscore Tables

You can also have "pure" Obscore tables which do not build on protocol mixins. A live example is the `cubes` table in the califa/q3 RD within the GAVO data centre. Here is a brief explanation of how this works.

Somewhat like with the SSA view, you define a table for the obscore columns varying for your particular data collection. In that tables' definition re-use the metadata given in the global obscore table. A compact way to do that is through a LOOP (see Active Tags) and `original` references, exploiting the `namePath` on Element Table:

```
<table id="cubes" onDisk="True" namePath="//obscore#ObsCore">
  <LOOP listItems="obs_id obs_title obs_publisher_did
    target_name t_exptime t_min t_max s_region
    t_exptime em_min em_max em_res_power">
    <events>
      <column original="\item"/>
    </events>
  </LOOP>
```

`adql="True"` is absent here as the obscore mixin will set it later. To just have all obscore columns in your table, you can write:

```
<FEED source="//obscore#obscore-columns"/>
```

instead of the LOOP.

If you do not have any additional columns (which you can of course have) and just want to have your datasets in the obscore table, consider having `<adql>hidden</adql>` after the obscore mixin. This will make your table invisible to but still readable by TAP. This is desirable in such a situation because the entire information of the table would already be contained in the obscore table, and thus there is no real reason to query the extra table. In the Califa example cited above, that is not the case; there is a wealth of additional columns in the custom, non-obscore table. We believe this will be the rule rather than the exception.

For a quick overview over what column names you can have in the `listItems` above, see the obscore table description.

Even with a custom obscore-like table, 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 `//products#define`'s path), and so use the //products#table mixin (which you don't see with SIAP and SSAP as their mixins pull it in for you):

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

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

```
<mixin
    access_estsize="10"
    access_format="'application/x-votable+xml;content=datalink'"
    access_url="dlurl"
    calib_level="3"
    dataproduct_type="'cube'"
    em_max="7e-7"
    em_min="3.7e-7"
```

```
    em_res_power="4000/red_disp_mean"
    facility_name="'Calar Alto'"
    instrument_name="'PMAS/PPAK at 3.5m Calar Alto'"
    o_ucd="'phot.flux;em.opt'"
    obs_collection="'CALIFA'"
    obs_title="obs_title"
    s_dec="s_dec"
    s_fov="0.01"
    s_ra="s_ra"
    s_region="s_region"
    s_resolution="0.0002778"
    t_exptime="t_exptime"
    t_max="t_max"
    t_min="t_min"
    target_class="'Galaxy'"
    target_name="target_name"
  >//obscore#publish</mixin>
```

Essentially, what is constant is given in literals, what is variable is given as a column reference. It is a bit unfortunate that you have to enter quite a few identity mappings in here, but pre-setting them won't help in most cases.
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.

Registering Obscore-Published Datasets

Most of the time, you do not need to worry about telling the Registry anything about what you do with obscore. As long as you have the obscore table, your TAP registry record will tell the Registry about it, and as long as that is published, clients looking for obscore-published data will find your service and thus your datasets (if they match the constraints in the obscore query, that is).
In the other direction, when you register a service for a data collection published via a typed protocol, DaCHS will add a reference such that clients can see that the data is available through obscore, too.
But when you do not register a typed service for your data collection for some reason, you should also register the standalone table as described in publishing DaCHS-managed tables via TAP

SIAP version 2

SIAP version 2 is just a a thin layer of parameters on top of obscore. To publish with SIAP version 2, simply ingest your data as described in publishing images via SIAP and add the //obscore#publishSIAP mixin.
In contrast to SIAP version 1, you do not define or register a service for a SIAPv2-published data collection. Instead, there is a sitewide SIAPv2 service at `<root URL>/__system__/siap2/sitewide/siap.xml`. It is always there, but it is unpublished by default. To publish it, you should furnish some extra metadata in the userconfig RD and then run:

```
dachs pub //siap2
```

Specifically, get the `sitewidesiap2-extras` stream and follow the instructions there to update the meta items as appropriate; at this point, they are exactly analogous to the ones for SIAP version 1.

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 `dachs imp -m <rd>`). This will include basic table metadata including its columns in both the TAP schema and the tableset of the TAP service, which is marginally enough to make it discoverable once your TAP service is registered.

You should, however, take particular care to give a useful description of the table, usually as a direct meta on the table. Keep in mind that people will typcially discover the table through some sort or Registry query. Try to make sure they can figure out whether the table contains data useful to them by that description and the column metadata.

However, both the TAP schema and the tableset only contain rather limited metadata. Hence, when there is no published service on the table (which will take care of producing a proper registry record), you should have a publication for the table itself, which substantially increases people's chances to locate the data in typical Registry queries.

To publish such a non-service (usually a table definition, but you can register data descriptors containing multiple tables, too), use the Element publish on the table. For a simple table, just adding `<publish/>` to the table body and a subsequent `dachs pub` is enough.

To have such a table turn up on the root page of your site – this then links to a page describing the table and giving some information on how to query it using TAP –, as for services tell `publish` to include the local set, too:

```
<publish sets="ivo_managed,local"/>
```

In more complex scenarios involving multiple tables that should be all be mentioned in the registration, you can define an "inactive" data element. An example is found in califa/q3, where five tables make up the data collection's table set:

```
<data id="tables" auto="false">
  <meta name="title">CALIFA DR3 tables</meta>
  <make table="spectra"/>
  <make table="fluxposv500"/>
  <make table="fluxposv1200"/>
  <make table="cubes"/>
  <make table="objects"/>
  <publish/>
</data>
```

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

Publishing Externally Managed Tables via TAP

While we generally discourage not letting DaCHS do table creation and ingestion, sometimes there already is substantial infrastructure building a postgres table, in particular when data "lives". If you now want to use DaCHS to publish it via TAP, just write an RD to describe the table, but make the data element trivial and updating (which means that DaCHS will not tear down the table if one accidentally runs `dachs imp` without an `-m` flag). 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="d" updating="True">
    <publish/>
    <make table="values"/>
  </data>
</resource>
```

Within the data element you need one `make` each for each table you want to publish. After that, say `dachs imp -m <rd-id>`. This adds the metadata you've given to all kinds of administrative tables DaCHS keeps, but DaCHS will not touch the rows in the tables. It will also try to fix the permissions of the table such that DaCHS' 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 may need to declare them in either the allRoles or readRoles attributes to the table definition (see Element table).
See publishing DaCHS-managed tables via TAP on the Registry aspects, and in particular the use of a `sets` attribute to `publish` if you want the table(s) to be visible from your site's front page.

EPN-TAP

EPN-TAP is a standard for publishing planetary data via TAP[5]. The EPN-TAP recommendation is now in version 2.0, and support for that is provided by //epntap2. There's an official web-based client for EPN-TAP at http://vespa.obspm.fr. A hands-on guide on how to do EPN-TAP publications from scratch with a view to this custom client is available on the VO Paris wiki. If you have not at least skimmed this document's Introduction and the Dachs Basics, by all means start there.
You can use EPN-TAP to publish data without any associated datasets; this happens, for instance, in the catalogue of minor planets, mpc/q. More commonly, however, there are data files ("products") associated to each row. In this case, have at least:

```
optional_columns="access_url access_format access_estsize"
```

– these are required to manage such products.
When publishing datasets, there are two basic scenarios:

- local files; you let DaCHS find the sources, parse them, and infer metadata from this; DaCHS will then serve them. That's what's shown in the quick start example below. We believe that is the more robust model overall.

- ingest from pre-destilled metadata; this is when you don't have the files locally (or at least DaCHS should not serve them). Instead, you read metadata from dumps from other databases, metadata stores, or whatever. The titan/q RD shows an example for that.

To start an EPN-TAP service, do as per Starting from Scratch and use the epntap template:

```
dachs start epntap
```

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' Element pdsGrammar to work.

Quick Sample EPN-TAP Service

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 `dachs 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:

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

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

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

there.
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 "Use Service", wait until the table metadata is in and then again query something like:

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

Hit "Run Query",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.
To get into Vespa's query interface, you will have to register your table. **Do not** do this with the sample data.

Tables

In essence, EPNcore is just a set of columns, some mandatory, some optional. The mandatory ones are pulled into a table by using the //epntap2#table-2_0 mixin, which needs the `spatial_frame_type` parameter (see the reference for what's supported for it) since that determines the metadata on the spatial columns. Optional columns can be pulled in through the `optional_columns` mixin parameter, and, as said above, a few of these optional columns are actually required if you want to publish data products through EPN-TAP. The reference documentation lists what is available. You can, of course, define further, non-standard columns as usual.

So, an EPN-TAP-publishable table might be defined like this:

```
<table id="epn_core">
  <mixin spatial_frame_type="body"
    optional_columns= "access_url access_format access_estsize
    access_md5 alt_target_name publisher
    bib_reference" >//epntap2#table-2_0</mixin>

    <column name="acquisition_id" type="text"
        tablehead="Acquisition_id"
          description="Extra: ID of the data file in the original archive"
          ucd="meta.id"
        verbLevel="2"/>
</table>
```

Filling EPNcore Tables

To populate EPNcore tables, use the //epntap2#populate-2_0, apply identifying the parameters applying to your data collection and setting them as usual (cf. Mapping Data). You may need to refer to the EPN-TAP proposed specification now and then while doing that. Note again that parameter values are python expressions, and so you have to use quotes when specifying literal strings.

If you have to evaluate complex expressions, it is recommended to do the computations in Element var-s and then use the variables set there in the `bind‘‘s (as ‘‘@myvar`). This also lets you re-use values once computed. Even more complex, multi-statement computations can be done in Element apply with custom `code`.

Serving Local Products

When DaCHS is intended to serve local files itself (which is preferable),

- use the //products#define rowfilter in the grammar as usual (cf. The Products Table). Note that this assumes by default that you are serving FITS files, which in EPN-TAP most likely is not the case. Hence, you will usually have to set the `mime` parameter as in, perhaps:

  ```
  <bind name="mime">"image/x-pds"</bind>
  ```

- in your row maker, the use the epntap2#populate-localfile-2_0 apply (if this gives you errors, make sure you have the optional columns for products as described above).

Incidentally, you could still use that even for external products, which is useful if you have DaCHS-generated previews or want to attach a datalink service. In that case, however, you have to invent some accref for DaCHS (probably the remote file path) and set that in products#define's `accref` parameter. The remote URI then needs to go into the `path` parameter.

Serving External Products

When all you have are external URLs, you do not need to go through the products table (though you still can, as described in Serving Local Products). It is simpler, however, to just directly fill the `access_url`, `access_format` and `access_estsize` columns using plain Element map-s.

On `s_region` in EPN-TAP

The `s_region` parameter (see //epntap2#populate-2_0) is essentially a footprint describing the area covered by 2D spatially extended data products. It uses pgshpere types such as `spoly`, `scircle`, `smoc`, or `spoint` (we advise against the use of `spoint` as a `s_region` type: only spatially extended types should be used). The default type is `spoly`, the others must be specified using the `regiontype` mixin parameter (see //epntap2#table-2_0).

For more information on how to create values for these regions, see Creating pgSphere Geometries.

Service

EPN-TAP tables are queried through the DaCHS' 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 `dachs pub <rd-id>`.

Datalink

Datalink is not a discovery protocol like the others discussed so far; rather, it is a file format and a simple access protocol for representing relationships between parts of complex datasets. Essentially, datalink is for you if you have parts of a dataset's provenance chain, refined products like source lists and cutouts, masks, or whatever else. Together with its companion standard SODA, it also lets clients do server-side manipulations like cutouts, scaling, format conversion, and the like.

Datalink this is particularly attractive when you have large datasets and you don't want to push out the whole thing in one go by default. Instead, clients can then query their users for what part of the dataset they would like to get – or to alert them of the fact that a large amount of data is to be expected.

Since Datalink is very flexible, defining datalink services is a bit involved. The reference documentation has a large section on it. Here, we discuss some extra usage patterns. The concrete application to spectra and images is discussed in SSAP and Datalink and SIAP and Datalink. See also the Datalink showcase in the GAVO data centre for live examples of datalink documents.

Associating Datalink Services

In DaCHS, Datalink services are associated with tables. This association is declared using the `_associatedDatalinkService` meta item, which consists of a `serviceId` (a service reference as per referencing in DaCHS) and an `idColumn` (stating from which column the ID parameter to the datalink service is to be taken from). So, within the table, you add something like:

```
<meta name="_associatedDatalinkService">
   <meta name="serviceId">dl</meta>
   <meta name="idColumn">pub_did</meta>
</meta>
```

This implies that the service `dl` within the current RD will produce a datalink document if passed a string from `idColumn`. The example implies that this column ought to contain publisher DIDs (see Dataset Identifiers), which is what the standard descriptor generators that come with DaCHS like to see. Since publisher DIDs tend to be a bit unwieldy (they are supposed to be globally unique, after all), the standard descriptor generators will also let you pass in plain accrefs.

If you write your own descriptor generator, you are free to stick whatever you like into the `idColumn`, just so long the table and the descriptor generator agree on its interpretation.

Datalinks in Columns

The `_associatedDatalinkService` declaration discussed in the previous section is all it takes when you serve data to datalink-aware clients. If, however, you also want to cater to clients without native datalink support, you may want to add links to the datalink documents in your responses; this is particularly advisable when you have services working through forms in web browsers.

One way to effect that is by defining a column like this:

```
<column name="datalink" type="text"
    ucd="meta.ref.url"
    tablehead="DL"
    description="URL of a datalink document for this dataset"
    verbLevel="1" displayHint="type=url">
  <property name="targetType"
    >application/x-votable+xml;content=datalink</property>
  <property name="targetTitle">Datalink</property>
</column>
```

The property declarations add some elements to response VOTables that inform clients like Aladin what to expect when following that link. At this point, this is a nonstandard convention.

You will then have to fill that column in the rowmaker. As long as the product is being managed through the products table and you thus used the //products#define rowfilter in the grammar, all that takes is a macro:

```
<map key="datalink">\dlMetaURI{dl}</map>
```

Here, the "dl" in the macro argument must be the id of the datalink service.

This method will retain the datalink columns even in protocol responses. While at this point there is something to be said for that, because users immediately discover that

datalink is available, datalink-aware clients will then have both the datalink through `_associatedDatalinkService` and the in-table column, which, since they cannot know that the two are really the same, will degrade user experience: Why should the same datalink be present twice?
With increasing availability of datalink-aware protocol clients, we therefore prefer a second alternative: produce the extra datalinks only when rendering `form` responses. To do that, furnish web-facing services with an Element outputTable. In there, do not include the column with your publisher DID but instead produce a link directly to the links response, somewhat like this:

```
<service id="web" core="siacore">
  ...
  <outputTable>
     <outputField name="dlurl" select="accref"
      tablehead="Datalink Access"
      description="URL of a datalink document for the dataset
        (cutouts, different formats, etc)">
      <formatter>
        yield T.a(href=getDatalinkMetaLink(
          rd.getById("dl"), data)
          )["Datalink"]
      </formatter>
      <property name="targetType"
        >application/x-votable+xml;content=datalink</property>
      <property name="targetTitle">Datalink</property>
    </outputField>
```

Datalinks as Products

In particular for large datasets, it is usually a good idea to keep people from blindly pulling the data without first having been made aware that what they're accessing is not just a few megabyte of FITS. For that, datalink is a good mechanism by pointing to a links response as the primary document retrieved.
Of course, without a datalink-enabled client people might be locked out from the dataset entirely. On the other hand, DaCHS comes with a stylesheet formatting links responses to be usable in a common web brower, so that might still be preferable to overwhelming unsuspecting clients with large amounts of data.
To have datalinks rather than the plain dataset as what the accref points to, you need to change what DaCHS thinks of your dataset; this is what the `//products#define` rowfilter in your grammar is for:

```
<fitsProdGrammar qnd="True">
  <rowfilter procDef="//products#define">
    <bind key="path">\dlMetaURI{dl}</bind>
    <bind key="mime">'application/x-votable+xml;content=datalink'</bind>
    <bind key="fsize">10000</bind>
    [...]
  </rowfilter>
  [...]
</fitsProdGrammar>
```

The `fsize` here reflects an estimation of the size of the links response.
When you do this, you must use a descriptor generator that does not fetch the actual file location from the path in the products table, since that column now contains the URI of the links response.

For FITS images, you can use the `DLFITSProductDescriptor` class as //soda#fits_genDesc's `descClass` parameter. The base functionality of a FITS cutout service with datalink products would then be:

```
<service id="dl" allowed="dlget,dlmeta">
 <meta name="title">My Cutout Service</meta>
 <datalinkCore>
   <descriptorGenerator procDef="//soda#fits_genDesc"
     name="genFITSDesc">
     <bind key="accrefPrefix">'mysvcs/data'</bind>
     <bind key="descClass">DLFITSProductDescriptor</bind>
   </descriptorGenerator>
   <FEED source="//soda#fits_standardDLFuncs"/>
 </datalinkCore>
</service>
```

If you have something else, you will have to write the resolution code yourself – `DLFITSProductDescriptor`'s sources (in gavo.protocols.datalink) should give you a head start on how to do that); see also the tsdl service bgds/l for how to integrate that into your RD.

Note that DaCHS will not produce automatic previews in this situation. Have a look at Product Previews for what to do instead.

HiPS

The Hierarchical Progressive Survey is a nifty way do publish "zoomable" data, in particular for images (but catalogues work as well). Conceptually, HiPSes are static data, which you will first have to generate. We will cover publishing an image HiPS from FITS files here.

Building the HiPS

First, get Hipsgen, a piece of java that will do the necessary math. Move the downloaded file `Hipsgen.jar` to a convenient place; the following assumes it's in your home directory. If you want to publish HiPSes, at least skim the hipsgen user manual before proceeding.

To generate your HiPS, you next need to build a parameter file. In DaCHS, you will conventionally have that in `res/hips.params` (do put it under version control). DaCHS will generate a template for you. Use something like (mkdiring `res` if necessary):

```
dachs adm genhips q#import 4 > res/hips.params
```

Here, `q#import` points to the `data` element importing the images you want to turn into a HiPS. The `4` in the example gives that smallest order to generate a HiPS for. The value here depends on the coverage of your data collections. Use 0 when you have full sky coverage, 1 for half the sky, and so on. To generate a HiPS for a "survey" of a one-degree patch, you would use 6.

This creates a control file for hipsgen that will guess parameters to turn the FITS files the grammar will presumably read into a HiPS in the hips subdirectory of your resource directory.

It is likely that you will want to edit the hips.params. For one, `adm hipsgen` contains a few rather naive heuristics on how to come up with the inputs directories and identifiers.

But mainly, the hipsgen manual describes numerous options to influence the hipsgen works for use on the command line (e.g., sky background subtraction, cutting off borders, and much more that DaCHS simply cannot guess). Put these options into your `hips.params` rather than on the command line, and you will have a much easier time re-generating the HiPS some later time.
In simple cases, you may get away with what DaCHS has generated. Either way, one hips.params is ready, run:

```
java -Xmx4g -jar ~/Hipsgen.jar -param=res/hipsgen.param
```

inside the resource directory (`-Xmx4g` means "give it 4 Gig RAM"; increase as sensible). In particular if experimenting on a large data collection, consider using `-pilot=10` (for just processing 100 input images) first and proceed to inspect the visual appearance of your new HiPS before wasting more resources on a computation that could possibly be improved quite substantially with a modicum of tweaking.
This produces (with quite a bit of computation) the hierarchy of files that then serves as a HiPS.

Publishing the HiPS

In principle, you could use a static renderer to publish this HiPS; once computed, DaCHS only needs to hand out files. However, for proper registration and some minor smoothing, there is a custom renderer for HiPS, the hips renderer. With this, a service for handing out your HiPSes looks like this:

```
<service id="hips" allowed="hips">
  <meta name="title">Fornax Cluster Core in HiPS</meta>
  <meta name="description">A HiPS generated from the high-resolution
    image.</meta>
  <property name="staticData">hips</property>
  <nullCore/>
</service>
```

The `id` is of course up to you. With our choice here, the URIs into the service look a bit silly (`...q/hips/hips`). Shortening the description over the RD's description is probably a good idea; HiPS descriptions tend to be one-liners. The staticData property needs to point to the subdirectory into which your built your HiPS (where what is here is the default of `adm hipsgen`). Finally, the nullCore says that this service will never do any computation.
With this service, DaCHS has enough information to complete the `hips/properties` file that keeps HiPS metadata in a proprietary format. Hence, run:

```
dachs adm hipsfill q#hips
```

(the argument is the DaCHS id of the service you just defined). Have a look at `hips/properties` after that and fix things as necessary. Note that hipsfill will not touch lines that are not commented out. If you want it to re-compute a line, prefix it with a hash.
If your HiPS is relatively small, consider pointing the Aladin lite in the `hips/index.html` file generated by Hipsgen to some position that actually has imagery. To do that, locate the instantiation of aladin in that file and edit it to read somewhat like this:

```
var aladin = $.aladin("#aladin-lite-div", {showSimbadPointerControl: true,
  target: "54.625 -35.455", fov: 1});
```

(for starting up at α=54.625, δ=-35.455 with a field of view of a degree).
You could publish this service to the registry in the usual way. However, it is rather likely that you already have a service (e.g., SIAP or TAP) on the data collection in question. In that case (and `dachs hipsgen` in effect assumes it), it would be wasteful to create a second resource record for the same data collection. Instead, *add* a hips capability to the existing record using:

```
<publish render="hips" sets="ivo_managed" service="hips"/>
```

in the service element of the *other* service. Run `dachs pub` after doing this.

biblink-harvest

These are facilities for communicating metadata to VO-external metadata services like the ADS. At this point, we have little to add to the material in the reference documentation: Bibliographic Links and biblink-harvest.

HTML form-based services

In principle, you can often just add `form` to the list of allowed renderers in a service and have a service display an HTML form and return an HTML table.
However, such a service will typically not be very pretty or usable – DaCHS services are normally intended to be consumed by machines. These do not care about a plethora of parameters they will never use, and they can easily fix the display of units or a large number of columns for their users.
Web browsers and humans are not good at that, and so it often pays to build extra services targeted at browser users next to typed services – or perhaps have stand-alone forms, for instance on top of a TAP-published table.
Soap box: don't waste too much time on form-based services when there are standard protocols to access the data – they're a pain to use, even if, and I admit that, most of your initial users may not have realised that yet. Consider every minute spent on form service as a compromise with an imperfect world. Of course, there are exceptions. Me, for instance, I kind of like our sasmirala service, which doesn't work without a good dash of HTML.

Basics

The template for a form-based service to a database table is:

```
<service id="web" allowed="form">
  <meta name="shortName">____ web</meta>
  <dbCore queriedTable="____"/>
</service>
```

As usual, the short name must be shorter than 17 characters, and queriedTable contains the id of the table you want to query. That's an XML id or a DaCHS cross-RD reference, *not* the name the table has in the database. Of course, giving more of the usual service metadata usually is a good idea.

Query Parameters

The query parameters are generated from the condDescs of the dbCore; see Service Definitions for a bit more on them. So, the classical pattern for adding such a parameter is:

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

This will enable all kinds of VizieR-like expressions on col_to_query, where the operators available depend on whether col_to_query is a string, a number, or a timestamp. If you've run `dachs limits`[4] on the RD, you will also get placeholders giving the ranges available for numeric columns.

However, there are cases when you want to fight the horror vacui (sitting in front an empty form without an idea what to put where) more carefully, in particular with enumerated columns. In that case, you can tell DaCHS to produce selection boxes. This is based on building `condDescs` from `inputKeys` with options:

```
<condDesc>
  <inputKey original="calib_level">
    <values>
      <option title="raw/custom">0</option>
      <option title="raw/standard">1</option>
      <option title="calibrated">2</option>
      <option title="derived">3</option>
    </values>
  </inputKey>
</condDesc>
```

where `calib_level` references a column that carries the remaining metadata. When the column already carries such a `values` element, you can skip the values on the input key. In that case, even a simple `buildFrom` will produce a selection box. Note, however, that giving the values on the column itself will reject values not mentioned when importing.

Manually constructing the input key also lets you control how many items the browser will show in the selection box by giving a `showItems` attribute to the input key (see below for an example). Making `showItems` -1 will use checkboxes (or radiobuttons) instead of a selection list.

Explicitly enumerating the options in inconvenient when there is an open-ended list of terms, as for instance with object names, `obs_collection` in obscore, or perhaps band or emulsion names. For such cases, DaCHS lets you pick the values from the database itself using the values element's `fromdb` attribute. This must be a SQL query fragment without the SELECT returning exactly one column. For instance,

```
<condDesc buildFrom="calib_level"/>
 <condDesc>
  <inputKey original="obs_collection" showItems="10">
    <values fromdb="obs_collection from ivoa.obscore"/>
  </inputKey>
</condDesc>
```

Note, however, that DaCHS will execute this query when loading the RD. Hence, if this is a long-running query, the RD will take a long while to load, which usually is unwelcome. Simply make sure that the query can use an index if your table has a non-trivial size.

On DaCHS newer than 2.5.1, instead of fromdb simply give the source column a `statistics` property `enumerate`. Then, `dachs limits` will gather the necessary statistics itself (on the obscore, this is already done).

DaCHS will also evaluate the `multiplicity` attribute of such input keys. If it is `multiple` (which is the default in this situation), widgets are chosen that let users select multiple items; if it is `single`, it the widgets will only admit a single selected item.

In all these cases, the input keys will generate constraints using DaCHS' default rules. This includes the various input syntaxes like the VizieR expressions mentioned above; use `vexpr-float`, `vexpr-string`, or `vexpr-date` as `type` when building explicit input keys rather than using `buildFrom` *and* you want these extra syntax things. With explicit input keys, you could also tell DaCHS to understand PQL-like syntax (as in SSAP) with the types `pql-int`, `pql-float`, `pql-string`, and `pql-date`, but, frankly, I don't think that's a good idea.

When this is not enough and you need to generate custom query fragments, see More on CondDescs.

Output Table

In browser-based services, you are directly talking to the user, and therefore you probably want to return not too many columns. I can also confidently predict that even in the 2020ies you will still be asked for horrible sexagesimal coordinates and wavelengths in whatever twisted units your data providers prefer (though I have to say that moving away from the VO's preferred wavelengths *is* a good thing).

For column selection, DaCHS by default picks the columns with a `verbLevel` up to and including 20. Columns excluded in this way can be added back using the "more output fields" selector in the query form. However, in particular for services built on top of standard services, that selection that is often... improvable.

In such cases, you can define an output table on the service. The idiomatic way to do that is to copy over the columns you want to retain. The element outputTable has a legacy `autoCols` attribute for that, but now that DaCHS has LOOPs, my advice is to use the following pattern, because it makes it much easier to sprinkle in modified columns now and then:

```
<outputTable namePath="//obscore#ObsCore">
  <LOOP listItems="obs_collection obs_title t_min t_max"><events>
    <outputField original="\item"/>
  </events></LOOP>

  <outputField original="s_ra" displayHint="type=hms"/>
  <outputField original="s_dec" displayHint="type=dms"/>
  <outputField original="s_fov" displayHint="displayUnit=arcsec,sf=0"/>

  <LOOP listItems="em_min em_max"><events>
    <outputField original="\item" displayHint="displayUnit=Angstrom"/>
  </events></LOOP>

  <LOOP listItems="access_estsize access_url s_xel1 s_xel2">
    <events><outputField original="\item"/></events>
  </LOOP>
</outputTable>
```

The first thing to note here is the `namePath`, which tells DaCHS where to resolve the

columns in by default (this will default to the table the core queries, so you can usually leave it out).
Then, we use Active Tags to copy over columns we want to treat uniformly. In this case, we first copy over a few columns literally, but then want to furnish other columns with display hints: here, we mogrify the positions to sexagesimal (definitely not recommended) and do some unit conversion and numeric formatting for the field of view. Then we give both spectral coordinates in Angstrom (this assumes optical astronomers, obviously; a propos of which I should also call out the spectralUnit display hint DaCHS has since version 2.6). The element concludes with another loop copying literally.
Note that you can usually give the display hints in the column definition, as they will not hurt the protocol access. This would simplify the element above because you can then copy things without further modification. An extra (questionable) benefit of giving the display hints in the column definitions in the table is that these will even be picked up when users to ADQL query through the web interface.
In case DaCHS' built-in display hints are not enough, you can use the `select` and `formatter` attributes discussed in the reference (Element outputField). For instance:

```
<outputField name="dimensions"
    tablehead="Dim"
    description="Pixel Dimensions, space x space x time x spectrum x
      polarization"
    select="array[s_xel1, s_xel2, t_xel, em_xel, pol_xel]">
  <formatter>
    return "x".join(str(s) if s else "1" for s in data)
  </formatter>
</outputField>
```

which also shows how to select multiple values at a time.
You can also produce arbitrary HTML using stan syntax (cf. templating.rstx, the section on in-python stan):

```
<outputField original="ivoid">
  <formatter>
    return T.a(href="/glots/q/showtables/qp/%s"%urllib.parse.quote(data))[
      data[6:]]
  </formatter>
</outputField>
```

Example: Form-based obscore

Several of the examples above were from obsform/q, and RD that is active in the GAVO data centre. If you have an obscore service of your own, you can probably just pull the resdir into your inputs and have the contents of that accessible to browser users, too.
It showcases a few tricks that may come in handy in your own form creations. First, there is a cone search condition choosing custom columns; this is necessary here because the obscore s_ra and s_dec columns do not have the UCDs expected by the SCS condDesc (requires DaCHS 2.6+):

```
<condDesc original="//scs#humanInput">
  <phraseMaker original="//scs#humanSCSPhrase">
    <bind key="raColName">"s_ra"</bind>
    <bind key="decColName">"s_dec"</bind>
  </phraseMaker>
</condDesc>
```

The next condition descriptor, `<condDesc buildFrom="calib_level"/>`, showcases how DaCHS turns values metadata attached to the table column into a selection box. To inspect the respective column definition, see the output of `dachs adm dump //obscore`.

Then there is a condition descriptor for the collection column:

```
<condDesc>
  <inputKey original="obs_collection"
    tablehead="Collection" showItems="10">
    <values fromdb="obs_collection from ivoa.obscore"/>
  </inputKey>
</condDesc>
```

This is a bit of a sore spot, as this will be slow at this point once your obscore table has grown to a certain size. This is because of a nasty conspiracy between how you cannot have indexes on views and that postgres does not optimise for constants in DISTINCT queries. We will think about this a bit more deeply if and when this becomes a problem for other deployers.

The real exciting pars are the last next two conditions, those for spectrum and time. These are tricky because they are effectively interval-valued in the database, with a minimum and maximum column each. We still want to let people enter our VizieR-like expressions, which means we have to do what DaCHS does behind the scenes form `buildFrom` manually.

This at the moment requires a certain about of manual code (that will also only work on DaCHS newer than 2.5). Here's the code for the spectral condition:

```
<condDesc>
  <inputKey name="BAND" type="vexpr-float"
    unit="Angstrom" ucd="em.wl"
    tablehead="Wavelength"
    description="Wavelength covered by the dataset"/>
  <phraseMaker>
    <setup imports="gavo.svcs.vizierexprs">
      <code>
        obscore = parent.parent.queriedTable
        minCol = svcs.InputKey.fromColumn(
          obscore.getColumnByName("em_min"),
          inputUnit="Angstrom")
        maxCol = svcs.InputKey.fromColumn(
          obscore.getColumnByName("em_max"),
          inputUnit="Angstrom")
      </code>
    </setup>
    <code>
      try:
        tree = vizierexprs.parseNumericExpr(inPars[inputKeys[0].name])
      except utils.ParseException as msg:
        raise base.ValidationError(
          f"Bad VizieR syntax: {msg}", "BAND")
      res = vizierexprs.NumericIntervalFlattener(
        ).getSQLFor(tree, (minCol, maxCol), outPars)
      yield res
    </code>
  </phraseMaker>
</condDesc>
```

So, we first define a fully synthetic input key – none of the metadata of em_min or em_max is terribly helpful here.

Then we declare a phrase maker. Once we have a good idea how to write this, we will probably give a procDef that hides quite a bit of this uglyness; then again, we as a community probably should just use more interval-typed columns.

Phrase makers are procDef-s as, say apply, and hence they consist of a setup part executed when the RD is imported and a code part executed once per (in this case) query.

Here, we use the setup to find in the min and max columns. We make input keys from them, as that is what the flattener we use later expects. The fromColumn constructor also lets us smuggle in code for unit adaptation (the inputUnit attribute that is turned into a scaling factor by the input key machinery).

Whatever is defined in a procedure definition's setup code is available in its code; we will use minCol and maxCol in a moment.

First, however, we have to deal with the VizieR-like expressions we are expecting. To parse them, we are using parseNumericExpr from the svcs.vizierexprs module. This returns a parse tree that can, in turn, be translated into SQL. Before we do that, we catch parse errors (which DaCHS would return as 500 internal server errors) and turn them into ValidationError-s for the BAND parameter. This lets DaCHS' form renderer mark up errors inline rather than just spit out some ugly error page.

The actual SQL generation happens using a NumericIntervalFlattener, which encapsulates how to translate the various VizieR constructs to SQL. If you think they should be translated differently, you could derive your own flattener – see gavo.svcs.vizierexprs on how to do that. Its getSQLFor method takes the parsed expression, the input columns, and the dictionary of SQL parameters that gets passed into condDescs implicitly. It is the use of two column objects rather than just one that makes these interval flatteners so special that you currently cannot currently get by without custom code.

If you have understood this code, the condition descriptor for TIME will not be very surprising: You just use parseDateExprToMJD to get the parse tree (and would use parseDateExprToDateTime if you had timestamp-valued columns in the database). Once that's done, the remaining code is essentially the same, as date and numeric intervals are rather parallel in the VizieR grammar.

By the way, these two are obvious candidates for writing a common procDef. Don't be too surprised if the actual SVN code already has that by the time you read this.

Services for Editing the Database

Users can enter data into the database in many ways. The most common would be through uploads, either directly into the database (as, e.g., in theossa/q), by doing file uploads with then get periodically ingested (as, e.g., in lightmeter/q), or by being harvested (the classical example would be the relational registry, rr/q).

Sometimes, however, it is convenient to let people interactively edit database content. We don't have a service that does this in the GAVO data centre; there is a somewhat contrived example for that at http://docs.g-vo.org/editsample_q.rd; to play with it, put it into /var/gavo/inputs/editsample/q.rd.

If you have a look at the RD's content, you will first see the definition of the table to be edited, objlist, and a data item that fills it with more or less random data. If you have never used an embedded grammar before, a brief glance at this might be inspiring. To continue with the example, run dachs imp q, as you will need the table to edit it.

Linking to Edit Services

The RD then has a `view` service, which lets you query the table using the object id. In addition to normal DaCHS fare, it has this:

```
<outputTable original="objlist">
  <outputField name="edit"
      select="array[id, remarks]">
    <formatter>
      return T.a(class_="buttonlike", href="/\rdId/edit/form?"
        +urllib.parse.urlencode({
          "id": data[0],
          "remarks": data[1] or ""}))["Edit this"]
    </formatter>
  </outputField>
</outputTable>
```

By saying `original="objlist"`, you tell DaCHS to base the output table on the full table in the database (see also Output Table for alternative ways of getting output fields).
What's new is the `edit` output field; see http://localhost:8080/editsample/q/view and send off the empty form to see its effect. It has a `select` attribute that directly gives material to put into the select clause. We use an array because we want both the id (to know what we will be editing) and the remarks (in order to make it easy not to lose old remarks). If you want to make more or all fields editable, it is probably preferable to use the `wantsRow` attribute of Element outputField; in that case `data` within formatter is the entire row as a dictionary.
The formatter then produces stan, which is essentially HTML written in a slightly cooler way (see templating.html#in-python-stan). Here, I am setting a `class` attribute (the underscore is to dodge python's `class` keyword) so you could style the link with Operator CSS, and I am computing the URL in a halfway convenient way: using the `\rdId` macro makes this thing independent of what RD it lives in, and I am using `urllib.parse.urlencode` (which is part of the functions available in rowmakers, which you have in formatters, too) to robustly produce a query string.
In this particular case, we only want to edit the `remarks` column. If you wanted to edit more columns, you would add the respective columns in a similar way. Note, however, that urlencode will encode `None` as literal `"None"`, which is rarely what you want; if you may have null values, make sure you map them to empty strings manually.

Writing an Edit Service

The actual edit service is protected to prevent accidental overwrites by rampaging robots:

```
<service id="edit" limitTo="ari">
```

– see Restricting Access for how to manage users, groups, and credentials in DaCHS.
I am also seting a link back to the view service:

```
<meta name="_related" title="Query Service"
  >\internallink{\rdId/view}</meta>
```

This helps people to quickly go from the edit service's sidebar back to where they can query the table (but see below for a plausible alternative).

Since I cannot see major common use cases in this problem's vicinity, DaCHS has no built-in cores for editing things. Instead, write a Element pythonCore. This starts with declaring the input and output structure, where I am requiring both inputs, and I set a manual `widgetFactory` to give `remarks` a nice text input box:

```
<pythonCore>
  <inputTable>
    <inputKey original="objlist.id" required="True"/>
    <inputKey original="objlist.remarks" required="True"
      widgetFactory="widgetFactory(TextArea, rows=5, cols=40)"/>
  </inputTable>
  <outputTable original="objlist" primary=""/>
```

Don't question too much what that `widgetFactory` thing is. It is ancient and venerable, and has been in need of fixing for 15 years. Just take it as a scheme for producing text boxes rather than single-line input fields as it stands. Ahem.

You might consider giving the input key for the id a `widgetFactory` of `Hidden`; that would not produce a user-editable widget, which in this scenario *might* be preferable; but then I almost always prefer to assume I am dealing with sensible people, and for them, editing the id might one day be useful, and they would otherwise leave it alone.

The output table as defined here is just the object list again; the plan here is to return the edited line. Alternatives are discussed below.

The action of python cores is defined in an Element coreProc, which is a regular procedure application (just as the Element apply you may know from rowmakers). It could look like this:

```
<coreProc>
  <code>
    with base.getWritableAdminConn() as conn:
      if 1!=conn.execute("UPDATE \schema.objlist"
          " SET remarks=%(remarks)s"
          " WHERE id=%(id)s", inputTable.args):
        raise base.ValidationError("No row for id '{}'".format(
          inputTable.args["id"]), colName="id")

      return rsc.TableForDef(
        self.outputTable,
        rows=list(conn.queryToDicts("SELECT * FROM \schema.objlist"
          " WHERE id=%(id)s", inputTable.args)))
  </code>
</coreProc>
```

So, we basically rely on DaCHS' built-in input validation and just turn the items from `inputTable.args` – which is sufficiently dictionary-like for our database interface – into an update query. I am using a writeble admin connection here, as normal DaCHS tables (the ones originating from a plain `dachs imp`) are non-writable by table connections, and the normal table connections cannot write at all (cf. Database Queries). You *might* consider making such editable tables writable by the normal web user, too, but for now I have no plans to make admin connections somehow inaccessible to the web server any more, so that is probably not terribly useful.

`conn.execute` returns the number of rows that were touched by an operation. I am making sure that is one here, and if that is not the case, I am raising a ValidationError

with `colName="id"` (there cannot be more than one row touched because id is a primary key). Giving the `colName` lets DaCHS mark the location of the problem in its browser interface – try it.

If everything has gone well, I am building an output table out of the modified row. This is what DaCHS displays in response of a successful request.

While that behaviour makes sense – it lets people verify that their edits did what they expected they would –, it is unlikely that users will like it very much. It is more likely that they would like to get back to the original table display. To effect that, make DaCHS redirect there, perhaps with a restriction to the edited id:

```
raise svcs.Found("/\rdId/view?"+urllib.parse.urlencode({
  "id": inputTable.args["id"],
  "__nevow_form__": "genForm"}))
```

To make this work, you have to acquiant your python core with the gavo.svcs module that defines the `Found` exception (this is just an API to produce a 302 Found HTTP status code); the most succinct way to do that is to add:

```
<setup imports="gavo.svcs"/>
```

to the coreProc's content.

It somewhat cryptic `"__nevow_form__": "genForm"` is a weird implementation detail. Without it, DaCHS will give the user a form filled out with the id. With it, it will actually execute the query.

Looking Further

I give you the proposed interaction feels somewhat clunky in today's world of animated widgets, whether or not it makes a lot of sense. I'd expect most scientists will put up with it eventually, though.

But you *could* do in-place editing by inserting a solid helping of Javascript into a defaultresponse template (cf. templating.html). This would have a text box open on some user interaction, and once things are typed in retrieve the URL of the service call we produce in the formatter using Javascript's `fetch`.

In such a scenario, it is certainly simpler if the service just returns, say, `YES` or `NO` depending on whether the update has succeeded. You would do that by returning a pair of media type and payload:

```
return "text/plain", "YES"
```

If you made whole rows editable, your should probably return the entire row, too, presumably in JSON. To do that, you could write something like:

```
return "application/json", json.dumps(
  next(conn.queryToDicts("SELECT * FROM \schema.objlist"
    " WHERE id=%(id)s", inputTable.args)))
```

Chapter 5

Configuring DaCHS

While you can run services on DaCHS without further configuration, as soon as other people on the network get involved, you must or should configure your server in various ways. This section starts with a walkthrough though the various items you *must* configure in order to let your services work within the wider network and then covers a few more advanced topics.

As delivered, the web interface of DaCHS will make it seem you are running a copy of the GAVO data centre, with some metadata defused such that you are not actually disturbing our operation if you accidentally activate your registry interface. To change this and make the site "yours", you need to:

- Set up a few basic items in gavorc
- Set the fallback metadata
- Adapt a few things facing the web

In particular if running a busier site, you should also look into configuring your Postgres. There are many guides on this on the net; before delving into any of these, you may want to have a brief look at this blog post discussing relatively DaCHS-specific configuration issues. When you have larger tables, you should also have a look at a post on parallel queries.

Becoming a DaCHS operator

When you write RDs, run `dachs serve debug`, write in `/var/gavo/etc` or do similar DaCHS administrative work, you need to be in the `gavo` group, and things are simpler when you are a postgres superuser. The default Debian package creates the `dachsroot` system user for that purpose but disables logins for that; you could run `sudo -u dachsroot bash` before doing DaCHS work or enable logins for dachsroot by running `passwd dachsroot`.

If, on the other hand, you want to be yourself when doing DaCHS work, add userself to the `gavo` group:

```
sudo adduser `id -nu` gavo
```

and make yourself a postgres superuser:

```
sudo -u postgres createuser -s `id -nu`
```

If multiple people work in `/var/gavo/inputs`, you will either want to play with POSIX ACLs or at least make things group-writable and make sure everything is group-owned by gavo. Also, set the setgid bit of all directories and subdirectories you create there, as in:

```
sudo chmod g+s /var/gavo/inputs
```

That way, you minimise permission problems (see also `help umask` in bash or the equivalent in your shell if you don't use bash).

Basic Gavo.rc Settings

DaCHS keeps basic settings in `/etc/gavo.rc`; a few of them you must set before you can go public. Here is a reference for what you minimally should give on a standalone server:

```
[general]
maintainerAddress: you@mx.example.org

[web]
bindAddress:
serverURL: http://mydc.example.org:8080
sitename: Example Observatory Archive
```

The next few subsections explain the items here (and a few more that you will need in certain common deployment scenarios). More information on syntax and contents of gavo.rc is in the reference documentation's Configuration Reference.

The Server URL, Ports, and Addresses

In today's internet with its reverse proxies and firewalls, a service cannot usually figure out by itself what it can be reached as. That is why you will almost certainly have to tell DaCHS in `[web]serverURL`.

For the basic setup, we recommend you try things with HTTP first. HTTPS adds quite a bit that can go wrong. See Enabling HTTPS on how to enable HTTPS once the rest runs as you would like it to run.

Also note that serverURL is reflected in several internal tables of DaCHS. If you change it later, remember to run `dachs pub -a` to update them; the side effect is that (at least if you use DaCHS' built-in publishing registry) this will update the links in the VO Registry as well.

There are a few scenarios:

Standalone DaCHS

Connecting the DaCHS server directly to the internet is simpler and preferable if you can convince your network folks. You will still have to tell DaCHS what its machine is called from the outside.

Also, by default DaCHS only listens on the loopback address, which in practice means you can only reach it from the host it runs on. This is mainly a matter of precaution: We feel a program should have informed consent before running a public service. To make it listen on all addresses configured for your machine, set `[web]bindAddress` to the

empty string (you could also use the public IP here, but then DaCHS won't listen on localhost any more, and you will have to adapt some examples).

Hence, the minimal configuration required for a public site is:

```
[web]
bindAddress:
serverURL: http://mydc.example.org:8080
```

In practice, it looks a lot less provisional if you run on port 80; on the other hand, DaCHS needs to be started as root when ordered to listen on a privileged port (it drops privileges soon, though). That would then look like this:

```
[web]
bindAddress:
serverPort: 80
serverURL: http://mydc.example.org
```

Reverse Proxy

If there is an nginx or apache that serves as a reverse proxy and DaCHS does not talk with the net itself, you need to use the server name of the virtual host of that reverse proxy in serverURL, perhaps like:

```
[web]
serverURL: https://the.proxy.url
adaptProtocol: False
```

If the reverse proxy runs on a different machine, you will additionally need an empty `bindAddress` as above. The `adaptProtocol` in there means that DaCHS will not try to change http vs. https in links towards itself depending on whether you speak with in encrypted; you'll almost always want this in proxy operation, in particular if your proxy is of the ugly sort that indisciminately "upgrades" to https.

If the reverse proxy runs on the same machine, it may proxy other servers as well, and then it is likely that something else is already listening on port 8080. In that case, change serverPort without changing the serverURL, like this:

```
[web]
serverURL: http://the.proxy.url
serverPort: 8081
adaptProtocol: False
```

– where serverPort obviously needs to be adapted in the proxy configuration, too.

The Site Name

The machinery needs some short name for the site in so many places that this is in the basic configuration (rather than the basic metadata, where it should reasonably be). This should be a very terse phrase saying who you are; for Heidelberg, that's "GAVO Data Centre". That phrase goes into the `[web]sitename` item.

The Maintainer Address

There are several situations when the server wants to contact you (e.g., failed cron jobs). To let it do that, teach the server it is on to send mail (we like the nullmailer package for things like that) and set `[general]maintainerAddress` to a mail address you regularly read.

Enabling e-mail for a server may be a challenge these day; theoretically, all it would take is install the nullmailer package and configure it with some SMTP smarthost. In practice, this may now be a lot more difficult. Ask your institution's mail people.

If that does not go anywhere, you could install the exim4 package and configure a local address (for instance, dachsroot) as the maintainer address. You would then read the mail locally on your server.

If all else fails, you can configure:

```
sendmail: cat >> /var/gavo/logs/dead.letters
```

and then have a look at that dead.letters file now and then.

But letting your computer send mail is still the preferred thing by far.

Path Configuration

If you would like to (perhaps temporarily, for instance in a `~/.gavorc`) override where DaCHS looks for its tree (the `/var/gavo` we have been assuming here; this is often called `$GAVO_ROOT` in the reference documentation), set `[general]rootDir`, perhaps like this:

```
[general]
rootDir: /home/user/gavo
```

You can also override other paths (see Section [general]), which are then are interpreted relative to rootDir unless they start with a slash.

Two items you may want to change if your rootDir is on a network file system are `[general]tempDir` and `[general]cacheDir` – things probably run more smoothly if they are local.

The Admin Password

You can influence a running DaCHS server through its web interface after authenticating as `gavoadmin`. The password to use for that is set in the `[web]adminpasswd` configuration item (and has no default, which means no remote administration is possible). You normally want this on a site that has publications now and then, as having it lets `dachs imp` and `dachs pub` tell the web component to release some caches, and it also makes `dachs serve expire` work (which is sometimes useful to expunge cached RDs).

Authenticated gavoadmin can also access all restricted resources.

The password is normally stored in clear text in gavo.rc. This lets a running DaCHS job (e.g., dachs imp or dachs pub) use it to clear caches on a running server. If you don't want or need that – you can always reload or restart manually –, you can store the admin password in hashed form. Run `dachs adm hashPassword` to do that

Otherwise, as in other places, DaCHS assumes the machine it runs on is trusted. You can restrict gavo.rc access to the gavo group, but of course that only helps against people who do not need to run the `dachs` command (which needs access to `gavo.rc`).

Basic Metadata

The next configuration file you must edit before going public is `/var/gavo/etc/defaultmeta.txt`. This is the root of metadata inheritance (see near the end of Global Metadata); some items (like who to contact) will be the same for almost all your services. Also, some metadata (like your site's description) does not sit on any service to begin with. All this information is given in `defaultmeta.txt` in, essentially, lines of `<key>: <value>` (in fact, the file is written in the meta stream format).

In principle, you could set any meta item here that you want to be satisfied if metadata just "falls through". Items you must override in what DaCHS gives you after installation include:

- `publisher` – A short, human-readable name for you (for us, somewhat lamely, "The GAVO DC Team").
- `contact.name` – A human-readable name for some entity people should write to. This is not necessarily different from publisher, but ideally people can write "Dear <contact.name>" in their mails.
- `contact.address` – A contact address for surface mail (we've never received mail as far as I remember, but it sure looks nice, and perhaps one day grateful astronomers will send nice presents...).
- `contact.email` – An email address. It will be published on web pages, so there probably should be some kind of spam filter in front of it.
- `contact.telephone` – A telephone number people can call if things really look bad. This has happened, and I don't think we have ever had spam calls on the number we've been publishing for more than a decade, so I'd say it's reasonably safe.
- `creator.name` – A name to use when you give no creator in your resource descriptors. This is used in the creation of some technical records, so put something sensible there; contact.name probably is a good fallback if you are unsure.
- `creator.logo` – A URL for a logo to use when none is given in the resource metadata. Use a small PNG here; if you've put in a logo_medium.png (see Customising the Web Interface), just use `<your-DaCHS-url>/favicon.png`, which is a suitably scaled version of that provided by DaCHS.
- `authority.creationDate` – This is just a characteristic date for your site; dachs init has filled in the date it was run on, which probably is good enough.
- `authority.shortName` – A very short (<13 characters) identifier for your site (we're using "GAVO DC"). This is used in a few short names of built-in resources which are limited to 16 characters total. Make this string too long and your registry records will not validate.
- `site.description` – A description of your site that gives humans some idea what they are looking at. Example: `The GAVO data centre provides VO publication services to all interested parties on behalf of the German Astrophysical`

`Virtual Observatory`. Use backslashes an the end of the lines to break long lines. This is used when registering your publishing registry, and it is the introductory paragraph on the default root page.

- `_noresultwarning` – You can probably leave it like it is; this is a text that is displayed on empty results on forms; since individual services may want to override it, it is configured through the meta system.

Since favicon.png was mentioned in there: You can give DaCHS a URL to a Favicon in the `[web]favicon` configuration item. This could be something in static (in which case you would write `/static/my-fav-icon.png`) or a full URL. DaCHS provides a scaled copy of your logo that might be suitable on `<your-DaCHS-url>/favicon.png`, which you might use as well.

The Userconfig RD

In addition to gavo.rc and defaultmeta.conf, there is a third source for configuration information in DaCHS: the userconfig RD. As the name says, it is a DaCHS resource descriptor, but it mainly contains STREAMs or similar material for inclusion somewhere else.
Doing this in a way that doesn't break regularly on updates is not simple, and therefore there is a bit procedure involved.

Creating a Userconfig RD

DaCHS has a *builtin* RD `//userconfig` that is updated as you update DaCHS. It always contains fallbacks for everything that can be in userconfig used by the core code. When DaCHS attempts to resolve a reference into the Userconfig RD (i.e., looks for an element with a given id), it first looks it up in your local copy (which resides in `/var/gavo/etc`) and then in the distributed userconfig. That way, DaCHS can add new things in the userconfig without forcing you to merge on every update.
To create your local, editable copy, pull the distributed RD into `/var/gavo/etc/userconfig.rd` (unless, of course, that file already exists):

```
cd ‘dachs config configDir‘
dachs admin dumpDF //userconfig > userconfig.rd
```

Maintaining and Changing the Userconfig RD

If you already have a local userconfig.rd and cannot find something referenced in a userconfig in there, instead run:

```
dachs admin dumpDF //userconfig | less
```

then pick out the element(s) for whatever you want to change and copy them into your own `etc/userconfig.rd`.
Changes to userconfig.rd are picked up by DaCHS but will usually not be visible in the RDs they end up in. This is because DaCHS does not track which RDs make use of userconfig, so these will typically need to be reloaded manually. For instance, if you changed TAP examples, you'd need to run:

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

to make your change show up in the web interface. Although usually not necessary, you can reload userconfig itself using:

```
gavo serve exp %
```

Note that for `dachs serve exp` to work, you need to set The Admin Password, and if all that confuses you you can always just restart the server.

Referencing into the Userconfig RD

While Referencing in DaCHS in principle applies to the userconfig RD, normal RDs cannot do the fallback between the local and built-in RD. For this, there is a special syntax that denotes this "magic" RD `%`. This is what the `exp %` above was about.
Within an RD, you reference items from userconfig as `%#id`, as in `%#obscore-extracolumns`.
You can also define your own material in the userconfig RD. As an example, if you have a list of authors that publish a lot of data collections between them and you want to re-use that declaration in multiple RDs, you would put something like:

```
<STREAM id="thegang">
  <meta>
    creator.name: Author, F.
    creator.logo: http://example.org/logos/author_f.png
  </meta>
  <meta name="creator">Author, S.; Co-Author, N.; Ontributor, C.</meta>
</STREAM>
```

into your userconfig.rd and then, in each RD that describes work from them, just have:

```
<FEED source="%#thegang"/>
```

Claiming an Authority

If you want to run your own publishing registry (which you should if you will publish more than a handful resources), you need to claim and describe an authority.
You can skip this section if you intend to Register Through purx or to Register Through Web Forms.

Choosing your Authority

In the VO, every publisher has something like a "namespace", within which they are free to name things; that namespace is called your **authority**, it it needs to be gobally unique so VO identifiers are globally unique.
To pick one, think of a resonably short string defining your site; use dots to structure this as necessary. In Heidelberg, we are using `org.gavo.dc`, somewhat reflecting our DNS name. Today, I'd probably choose `gavo.dc`. Since astronomy is small enough, there is no need for deeply-nested hierarchy.
If you have picked a string, make sure it is not taken yet. A convenient way to do that is through WIRR. There, constrain *Resource Type* to *Authority*, then *Add Constraint* and constrain IVOID to the authority you have chosen (preconfigured WIRR query).
If nothing comes back, the authority is still free and you can enter it in your gavo.rc in the `[ivoa]authority` item, like this:

```
[ivoa]
authority: my-auth
```

Registering your Authority

The authority truly only belongs to you once you have added an authority record to the VO Registry. The chances that someone else will beat you to a non-trivial authority are slim, though, so do not worry if you have to delay a bit before going live with the following. It doesn't hurt to re-run the WIRR check before you actually go live, though.

You will first have do define your authority record. To do that, you have to edit your userconfig RD. See The Userconfig RD if you have not set one up yet.

Once you have your `etc/userconfig.rd`, look for the `registry-interfacerecords` stream. This contains two (relevant) elements: a resource record for claiming your authority, and the registry service, which is what the VO registry will later talk to.

While the latter usually does not manual intervention, in the authority, you must replace the the metaString macros defaulting to UNCONFIGURED (you could set the corresponding items in defaultmeta instead, but the additional indirection will not buy you anything), specifically:

- authority.title – A human-readable descriptor of what the authority corresponds to.
- authority.description – A sentence or two on what the authority you are using means. This could be the same as site.description if all you are claiming authority for is that; if you are claiming authority for your institute or organisation, this obviously should be different.
- referenceURL – A URL at which people can learn more about your data centre.

As an example, the following could work:

```
<resRec id="authority">
  <meta>
    resType: authority
    creationDate: \metaString{authority.creationDate}
    title: The Utopia Observatory Data Center Authority
    shortName: \\metaString{authority.shortName}
    subject: Authority
    managingOrg: \\metaString{publisher}
    referenceURL: http://utopia.example.org
    identifier: ivo://\getConfig{ivoa}{authority}
    sets: ivo_managed
  </meta>
  <meta name="description">
    The Data Center at the Observatory of Utopia manages lots of
    substantial data sets created by the bright scientists all over
    the world.  All data collections published there are
    registered under this authority.
  </meta>
</resRec>
```

You can, of course, override any of what is filled through the remaining macros, too, but you should have a good reason to do so.

Once you have done that, you can proceed to Preparing your Publishing Registry to make the VO Registry see all this. Or you can keep customising your DaCHS installation first.

Customising the Web Interface

DaCHS talks to web browsers, too. To customise the appearance, you can override almost everything DaCHS delivers. This section discusses the various hooks, roughly going from things everyone will want to do to things interesting if you want your site to look more exciting than our rather plain default look.

The Logo

The least you should do is replace the DaCHS logo in `/var/gavo/web/nv_static/img/logo_medium.png` with something pertaining to you. Scale the bitmap to about 250 pixels width or so (it will typically be used at 100 pt in the CSS).

In general, anything you put into `/var/gavo/web/nv_static` will be visible underneath `/static/` on the web. See also Customising the Web Interface.

The Root Page

When people dereference your data centre URI, they will see the root.html template rendered. The default is a simple list of services published in the local set (see Registering Services and Data), together with site.description from your defaultmeta.

This may do when you plan to publish more than a few but less than many services.

If you only have some very few services that furthermore will rarely, if ever, change, it is probably perferable to just write a static root page in plain XHTML in `/var/gavo/web/templates/root.html` (use `nv_static` from ancillary material as mentioned in The Logo).

If you expect to have many services, you should probably derive your root page from the root page that we use in Heidelberg; more on all this in templating.html#the-root-template.

Note that for DaCHS to notice you have overridden a built-in file, you must restart it once.

If you have special needs, there is also the `[web]root` configuration item. Set that to a relative path to replace the root service (which defines custom data functions such as titleList, chunkedServicesList and such used in the root templates) with a completely different service. Typical uses for this include pointing people directly to the service on single-service installations, or pointing to the ADQL form when what you are really doing is run a TAP-only system; in that latter case, you would say:

```
[web]
root: __system__/adql/query/form
```

Extra Sidebar Items

It is rather common to want to stick extra material into the sidebar. Overriding the sidebar template is a possibility, and it will probably survive several upgrades without breaking big time.

However, if it is just a few links or lines of text you want in there, there is the `_sidebarlocal` meta item that the default sidebar templates includes near its foot. There are the `sidebarnote` and `sidebarmicro` classes you can use make things fit; and, in order to pull raw HTML all the way through DaCHS' guts, you have to use the raw meta-data format. Hence, in `etc/defaultmeta.txt`, you would put something like (see Stream Metadata for details):

```
_sidebarlocal:  raw:<div class="sidebarnote">\
    <a href="/adql">Try ADQL</a> to query our data.</div>
```

Note that you can easily ruin the well-formedness of your document in this way; the material is really included in the HTML without any adaptation (except UTF-8 encoding). A recommended use for this is to link to a privacy policy. DaCHS tries hard not to store any data that could fall under the GDPR or similar legislation, but it's still nice to tell that to people; if you do process personal data, you must have some declaration like that in most juristictions anyway.

To have a privacy policy, go to `/var/gavo/web/nv_static` and put the following data into privacy.shtml:

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:n="http://nevow.com/ns/nevow/0.1">
<head n:render="commonhead">
<title>Privacy Policy</title>
</head>
<body n:render="withsidebar">
  <h1>Privacy Policy</h1>
  <p>It's nothing personal.  Really.  In particular, while we do
  store our logs for a year, we never keep the source IPs.</p>
</body>
```

You will see the result at http://localhost:8080/static/privacy.shtml.

To have this linked from your sidebar, just add the following to your default meta:

```
_sidebarlocal: raw:<div class="sidebarmicro">\
    <p><a href="/static/doc/privpol.shtml">Privacy</a></p>
  </div>
```

The somewhat odd extension `.shtml` for DaCHS means that the file will be interpreted as a template over the service `//services#root`, which means that you can use the usual render functions and data items as per templating.html. This is be an easy way to have your page fit into the look of the remaining site.

You could also use plain `.html`, in which case you could even escape the necessity to provide well-formed XHTML.

Operator CSS

To override css rules we distribute or add new rules, avoid changing or overriding gavo_dc.css, as that will be a liability when upgrading. Instead, drop a CSS file somewhere (recommended location: `/var/gavo/web/nv_static/user.css`) and add a configuration item in `[web]operatorCSS`. With the recommended location, this would work out to be:

```
[web]
operatorCSS: /static/user.css
```

in /etc/gavo.rc.

This can also be an external URL, but we recommend against that, as that would force a browser to open one external connection per web page delivered.

By far the most common complaint is that we are limiting the width of p and li elements to 40em. We believe that text lines much longer than that are hard to read and should

be avoided. On pages with tables where users might actually want their browsers to fill the entire screen, this choice cannot be made through a sensible choice of the width of the user agent window but requires CSS intervention.
Having said that, if you really think you want window-filling text lines, just put:

```
p, li {
  max-width: none;
}
```

into your operator CSS.

Overiding Built-in Web Resources

As mentioned in The Logo, you can put files and directories into `/var/gavo/web/nv_static` and have them show up on the web under `/static`. These files will override built-in resources of DaCHS if they overwrite their names. This might be intersting if you would like to fix a bug or try something out.
The originals sit in DaCHS' resource tree under web. Hence, if you want a local help file, in `nv_static` say:

```
dachs adm dump web/help.shtml > help.shtml
```

DaCHS decides between built-in and overridden when a resource is first accessed. Hence, when you first override a given resource, you have to restart the server. After that, DaCHS will pick up updates by itself.
Here are some built-in resources that you might want to override:

- `help.shtml` – the help file. Unfortunately, we blurb quite a lot about GAVO in there right now. We'll think of something more parametrisable, but meanwhile you may want to have your own version. This changes rarely, so it probably won't hurt to just fork it.
- `web/xsl/dachs-xsl-config.xsl` – this lets you customise (somewhat) the responses of various pages that are just XML with XSLT when viewed through the browser. This includes the OAI-PMH pages, UWS management, and VOSI responses.
- `web/css/gavo_dc.css` – the central CSS; only override this during bug fixing or similar. For anything more permanent, use Operator CSS.
- `web/js/gavo.js` – this is most of the custom javascript that DaCHS uses here and there. Again, only override this for debugging (in which case you will always want to set `[web]jsSource` to `True` to keep DaCHS from minifying your code). If you need custom javascript globally on a permanent basis, let us know and we'll do something similar to the operator CSS.

On the meaning of the `.shtml` extension, see the end of Extra Sidebar Items.

Overridden System RDs

You can also dump system RDs into `/var/gavo/inputs/__system__` (**not** `__system` -- that's something else) edit them there; after a restart, DaCHS will use your copy then. Doing this is even more of a liability for upgrades than overriding the CSS or the Javascript. Only do that when debugging or to fix a transient bug until a fixed package is released.

The local namespace

Here's a feature you should only use sparingly if at all: if you have code you need from multiple different RDs, you can put it into `/var/gavo/etc/local.py`. Whatever is defined in there is then visible to DaCHS code in the `api.local` namespace.
DaCHS will not automatically reload this module when it is updated. However, you can call `api.reloadLocal()` to force such a reload. Local-using server-side code might want to call that when you develop local.py itself.

Configuring dependencies

DaCHS' behaviour is partly determined by what dependent libraries do, and we try to ensure that the respective configuration is in your configDir (by default, `/var/gavo/etc/`). Right now, this is true for matplotlib, which, once you have imported `gavo.base`, is in `<configDir>/matplotlibrc` and astropy, which is configured in `<configDir>/matplotlibrc/astropy/astropy.cfg`.

Chapter 6

Interacting with the VO Registry

If you have tried any of the VOTT examples mentioned in the Introduction, you will have realised that you are not in the VO unless your service is registered. There are several options for how to do that. In this chapter, we will first discuss three options for doing the registration. The remainer of the chapter is then devoted to the preferred option, i.e., using DaCHS' built-in publishing registry.
For an introduction into the VO Registry, see 2014A&C.....7..101D.

Getting into the Registry

To make your services visible to the VO, you have to get things called **searchable registries** to index your metadata. The first condition for that is that this metadata is organised in a standard structure called VOResource. DaCHS does the organisation for you: on each info page (they're linked from the sidebar, but you can also get them by replacing the renderer name in the access URL with `info`) there is a link to "VOResource XML", and the fact that you are now interested in this proves that you start becoming a VO nerd according to what it says there.
This piece of XML needs to get "harvested" by the searchable registries. Usually, this happens through a protocol called OAI-PMH. DaCHS speaks this protocol, and most of the rest of this chapter deals with the wonderful things you can do if you let it do that.
However, for very small sites (one or two services, we'd say), preparing and running a **publishing registry** (that's what the OAI-PMH endpoints are called), including Claiming an Authority may be overdoing it. Instead you could either:

- use purx, or
- register through web forms.

The remainder of this section is devoted to these two options. If want to run a publishing registry, you can skip it.

Register Through purx

purx is a little service run by GAVO that lets people put arbitrary (but valid) VOResource records on some web server and have this service talk to the VO Registry to get them in there. Purx will pick up changes to the records, polling them roughtly daily, and propagate their contents to the VO Registry as necessary.

To use purx, get the URL of your service's VOResource XML as described above. Paste this URL into purx' enrollment field, wait for a mail to arrive, confirm the publication, and you are done. Your service will have a service identifier starting with `ivo://purx`, but the rest should contain material from your RD id.

To make purx stop publishing your record, you can take down the service and wait for purx to give up after a couple of months. A faster way would be to have the VOResource link return an HTTP 403 status code, which DaCHS so far cannot do (selectively). Let us know if you need it.

The good thing about purx is that once enrolled updates are automatic. Still, the extra HTTP-based polling increases latency and adds fragility. And you don't get to have your own authority in the identifiers, which sometimes is inconvenient (e.g., when Validating your Services).

Register through Web Forms

Both the EuroVO registry at ESAC and the NAVO registry at STScI let you register resources using a web browser and filling out web forms. When you are running DaCHS, this is not terribly attractive, because if you have followed Global Metadata for your RD and the first two sections of Configuring DaCHS, DaCHS already has essentially all information you would enter there.

Going through one of the web-based services still lets you offload authority management to them. And, at least on the EuroVO service, you can simply upload the XML DaCHS generates for you – see the "Create new Resource from File" (or, later, the "Edit Resource" tab in the details for a resource). The material to upload is again what the "VOResource XML" link on your service's info page points to. You will, however, probably have to edit the `identifier`.

Preparing your Publishing Registry

The VO Registry needs to know a few things about you before it accepts you as a full member; in particular, you must register your authority and your publishing registry. These are defined in the `//services` RD (well, in truth, the definition is the userconfig RD, cf. Registering your Authority), so the first records you have to add to your registry come from there; use `dachs pub` to do that:

```
dachs pub //services
```

Depending on other services you are running, you may want to publish the following additional built-in resources at this (or any later) point:

```
# If you're publishing spectra, images, and the like, in particular
# if you produce pubDIDs
dachs pub //products
# If you run a TAP service
dachs pub //tap
# If you run a TAP service and want to point people to the built-in
# web interface
dachs pub //adql
# If you have an obscore service with images and want to also
# offer the material via SIAv2
dachs pub //siap2
```

Registering Services and Data

As you have already seen in Service Definitions, DaCHS will only tell the Registry about a service if there is an Element publish in it; in Publishing DaCHS-Managed Tables via TAP, you have seen that this element can also be used in Element table or Element data to tell the Registry about TAP-queriable tables.

The `publish` element requires a specification of the OAI set the resources shall be published to. Unless you have specific applications, only two sets are relevant: `ivo_managed` for publishing to the VO through DaCHS' built-in publishing registry, and `local` for publishing to your data centre's service roster on its front page. Other sets could be introduced and used for, e.g., specific sub-rosters, but you'd have to tell DaCHS about them in `[ivoa]validOAISets`.

In services, the `publish` element needs, in addition, a `render` attribute, giving the renderer the publication is for. With `ivo_managed`, one publication essentially always corresponds to one capability of the VOResource record. For example, a typical pattern could be:

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

This generates one capability each for the simple cone search and a browser-based interface; the browser-based interface is, in addition, listed in the local service roster, typically on the root template.

When you publish tables (or collection of tables via a `data` element), the notion of renderers makes no sense. Instead, you can use a `service` attribute to say which service serves that data, except that when you publish tables that have `adql="True"`, the local TAP service is automatically considered to be a service for that data.

So, to publish an ADQL-queriable table to the VO for querying via TAP, just write:

```
<publish/>
```

within the table element. A table containing, e.g., data that's queried in a SIAP service in a different RD, would require something like:

```
<publish service="other/rd#siapsvc/>
```

(use multiple publish elements if there are multiple services).

As we have already discussed in the quick start's Publishing a Service section, just writing publish into the RD still is not enough, you also need to run `dachs pub <rdid>` to make DaCHS update its tables of published resources according to what is in the RD.

Looking a bit closer, `dachs pub` first un-publishes everything that was published through the RD before and is no longer published and then publishes everything declared in the in the current RD. The net effect is that services that vanished will still be recorded as **deleted resources**, which are necessary in a world of incremental harvesting (where Registries only ask "what's new?" most of the time and wouldn't catch vanishing resources without a means to say "X is gone").

To unpublish everything that is defined in an RD, run `dachs pub -u <rdid>`. You definitely want to do this before renaming an RD.

If the `[web]serverURL` config item on the machine running gavo pub is pointing to the actual running server and `[web]adminpasswd` machtes, the server will automatically be made aware of these changes. Otherwise, you need to prod the server as discussed in Dealing with RD Changes.

Making the VO see Your Registry

The searchable registries in the VO need to know which publishing registries to harvest, and the place they learn that at is the Registry of Registries or RofR. So, this where you need to enter your site to become part of the VO. The RofR only admits publishing registries with no technical defects, and hence to get in, you first have to go to http://rofr.ivoa.net/regvalidate/ and enter your registry endpoint (i.e., your installation's root URL with /oai.xml appended).

GAVO DaCHS is lenient with missing metadata and will deliver invalid VOResource for records missing some. It is not unlikely that your registry will not validate on the first attempt. Reading the error messages should give you a hint what's wrong. You can also use the `dachs val` command on the RDs that generate invalid records to figure out what's wrong. Also, see Simple OAI Operation for hints.

Once your registry passes the validation test, you can add it to the RofR, and the searchable registries will start to harvest your publishing registry. They do that perhaps once a day; do not become nervous if nothing happens for a couple of hours.

For the GAVO RegTAP registry, which is used by default in clients like TOPCAT and pyVO, you can speed up the process by using our Harvest Trigger Service. When you have just registered with the RofR, first paste `ivo://ivoa.net/rofr` into the ivoid field to make us notice your registry.

Then (and on any later update), paste your publishing registry's id, which in DaCHS will always be `ivo://<your authority>/__system__/services/registry`, and hit Go. Note that right now this will not update anything in GloTS (which is where TOPCAT still reads its TAP information from).

Simple OAI operation

If you want to check what you have published, see the `/oai.xml` on your server, e.g., http://localhost:8080/oai.xml. This is a plain OAI-PMH interface with some style sheets, and since your web browser does not speak OAI-PMH, you first get an error message. However, the default style sheets add a link to "All identifiers defined here". Just follow it to a list of all records you currently publish.

The OAI endpoint can also be used to help you in debugging validity problems with your registry content. To XSD-validate your registry without bothering the RofR (see Making the VO See your Registry), you can do the following:

```
curl <your oai.xml url>?verb=ListRecords&metadataPrefix=ivo_vor |\
  xmlstarlet fo > toval.xml
gavo admin xsdValidate toval.xml
```

This may result in a few error messages; if you don't understand them, it's a good idea to just go to the respective line in toval.xml and give it a long, hard look. You might also find hints on how to fix things in commonproblems.html.

Backing Up the Registry State

From your file system, there's no way to figure out what you have published when. To ensure that your publishing registry stays consistent even when your database disappears, you should make sure that either you have a safe backup of your entire database (and

no, backing up the files doesn't count, as that probably won't let you restore the thing) or, simpler, just add:

```
[db]
  dumpSystemTables: True
```

to your `/etc/gavo.rc`. With that, DaCHS, every midnight, will dump the contents of the tables in //services and //users (which contain everything permanently operator-changable that doesn't go through imp) to `$stateDir/systemtables.dump`. You can restore this file using `dachs dump load` in case of a disaster (in particular if you back it up off-site once more; backing up files is a lot simpler than backing up database tables).

Registering DaCHS-external Services

The registry interface of DaCHS can be used to register entities external to DaCHS; actually, you are already doing this when you're claiming an authority.

To register such a non-service resource, use Element resRec. You could reserve an RD (say, `/var/gavo/inputs/ext.rd` to collect such external registrations, or you could user one RD per external resource; we started with option one but increasingly prefer option two. Without further intervetion, your authority, the containing RD's rd id, and the resRec's `id` attribute will determine the ivoid of resource record, resulting in `ivo://<your authority>/<rd id>/<id of resRec>`.

If you do not like the autogenerated identifier, you can always override the `identifier` meta. It must still be from an authority you claim (see below on claiming additional authorities).

When what you want to register actually is a service, use a normal DaCHS service element with with a nullCore. You probably need to manually give an `accessURL`, which you can do (in this case) by setting a meta item.

The most common case is that of a service with a `WebBrowser` capability. These result from `external` or `static` renderers. Thus, the pattern here usually is:

```
<service id="myservice" allowed="external">
  <nullCore/>
  <meta>
    shortName: My external service
    description: This service does wonderful things, even though\
      it's not based on GAVO's DaCHS software.
  </meta>
  <publish render="external" sets="ivo_managed">
    <meta name="accessURL">http://wherever.else/svc</meta>
  </publish>
</service>
```

The rest of the RD would then give all the normal metadata as per Global Metadata `dachs pub` should complain if there's metadata missing.

In case you want register services in authorities other than `[ivoa]authority`, DaCHS lets you do that, too. Edit the userconfig RD, and locate your authority definition in the `registry-interfacerecords` STREAM. Copy the element and edit what is different in your new authority.

A bit further down in the userconfig, there is the service with the id `registry`; it says what authorities are being managed by your registry, and since you are registering a new one, you have to add it there, too, by adding a `managedAuthority` meta, like this:

```
<meta name="managedAuthority">edu.euro-vo.org</meta>
```

When done, run `dachs pub //services` to update DaCHS' resource tables.

Registering Web Interfaces to DAL Services

A typical situation is that you have a standard service (SSA, SCS, SIAP, etc) and a form-based custom service on the same data. Since the form-based service caters to humans, it can require quite different input parameters (and thus usually cores) and output tables, and so you will usually have a different service for it.

If you want to publish both services to the VO, you could add `publish` elements with `sets="ivo_managed"` to both `service` elements – but that would yield two resource records (which you then should link via relatedTo metas). At least when the form interface does not add significant functionality, this would usually seem undesirable – e.g., your service would show up twice in sufficiently unconstrained responses from the VO Registry.

Therefore, it is typically preferable to add the web interface as a capability to the resource record of the standard service. To let you do that, the `publish` element takes an optional `service` attribute containing the id of a service that should be used to fill the capability's metadata.

Here's an example:

```
<service id="web" defaultRenderer="form">
  <meta name="title">Form-based service</meta>
  <!-- add this service to the local roster -->
  <publish render="form" sets="local"/>
  ...
</service>

<service id="ssa" allowed="form,ssap.xml">
  <publish render="ssap.xml" sets="ivo_managed"/>
  <!-- now make a WebBrowser capability on this service in the IVOA
  published resource record, based on the service with the id web -->
  <publish render="form" sets="ivo_managed" service="web"/>
  ...
</service>
```

Note that since both services are published through the same resource, they share a title – the one of the parent of publish. You hence need to make sure that that title fits the browser service, too. For instance, don't say "Parrot Survey SSAP" but rather "Parrot Survey Spectra" (which, I would propose, is preferable anyway).

Creating an Organisation Record

If you want to fill out the publisherID meta item – and there's no strong reason to do so at this point –, you have to create a registry record for you, i.e., the organisation that runs the data centre and hence the registry.

If you still want to have a publisherId, put something like:

```
<resRec id="manager">
 <meta>
   resType: organization
```

```
    creationDate: 2007-12-19T12:00:00
    title: GAVO Heidelberg Data Centre
    subject: Organization
    referenceURL: http://www.ari.uni-heidelberg.de
    identifier: ivo://org.gavo.dc/org
    sets: ivo_managed
  </meta>
  <meta name="description">
    Operating at the Astronomisches Rechen-Institut (ARI) (part of
    Centre for Astronomy of Heidelberg University) on behalf of
    the German VO organisation GAVO, the GAVO Heidelberg Data Centre
    is a one-stop shop for astronomical data publication projects
    of (almost) any description.  We are also active within the
    IVOA in standards development and Registry operations.
  </meta>
</resRec>
```

into your userconfig.rd's `registry-interfacerecords` and then say `dachs pub //services`. Of course, you'll have to change things to match your situation; in particular, make sure identifier simply is `ivo://<your-authority>/org` – and that is what you'll use as publisherID.

Chapter 7

Server Operations

This section covers a few topics that concern the daily, monthly, or yearly maintenance of your DaCHS server.

Starting and stopping the server

The `dachs serve` subcommand is used to control the server. `dachs serve start` starts the server, changes the user to what is specified in the `[web]user` config item if it has the privileges to do so and detaches from the terminal.

Analoguously, `dachs serve stop` stops the server. To reload some of the server configuration (e.g., the resource descriptors, the vanity map, and the /etc/gavo.rc and ~/.gavorc files), run `dachs serve reload`. This does not reload database profiles, and not all configuration items are applied (e.g., changes to the bind address and port only take effect after a restart). If you remove a configuration item entriely, their built-in defaults do not get restored on reload either.

Finally, `dachs serve restart` restarts the server. The start, stop, reload, and restart operations generally should be run as root; you can run them as the server user (by default, gavo), too, as long as the server doesn't try to bind to a privileged port (lower than 1025).

All this can and should be packed into a startup script or the equivalent entity for the init system of your choice. Our Debian package provides both a System V-style init script and a systemd unit. They would typically be installed to /etc/init.d/dachs and /etc/systemd/system/dachs, respectively (but this might, of course, be different if you're running non-Debian systems).

With this, on your typical system, you can control DaCHS with:

```
sudo service dachs start
sudo service dachs stop
sudo service dachs restart
```

For development work or to see what is going on, you can run `dachs serve debug`; this does not detach and does not change users, and it also gives a lot more tracebacks in the logs. See Debugging for how to use this to figure out misbehaviour.

Access Logs

While I claim that access numbers are meaningless until you know enough of a domain that you don't need the access numbers any more to figure out whether or not something is a good idea, most data providers at some point find they need access logs.
Hence, DaCHS does write logs, albeit without IP addresses by default. We are doing this to keep you quite certainly GDPR-clean unless you configure DaCHS otherwise.
However, that is quite clearly not enough to compute the "number of distinct hosts" statistic that is part of the industry standard of Bad Metrics. To be able to compute something like that, you need to enable the logging of IPs and then install something that somehow evaluates the files in /var/gavo/logs.
Doing the first simply means to add:

```
logFormat: combined
```

to your `/etc/gavo.rc`'s `web` section. After that, you need to restart the server for the new setting to take effect.
While you can probably use whatever log analysis program you want for these files (I'd hope it's close enough to the apache combined log format), you can simply use DaCHS to do some basic accounting. That way, you do not need extra tools. You will also in all likelihood treat your users' data a bit more carefully.
For this, check out our accesslog resource:

```
cd `dachs config inputsDir`
svn checkout http://svn.ari.uni-heidelberg.de/svn/gavo/hdinputs/accesslogs
```

(this needs the subversion package; your could also use `wget -r`, but this way you can simply call `svn update` in accesslogs to pull upgrades).
The `accesslogs/q` RD contains everything that's necessary to regularly (daily at 2:30 UTC, right now) pull logs files into the database and then remove them. It tries a few tricks to weed out robots (which of course has limits), and it will pseudonymise IP addresses so that they cannot be correlated across months.
When the first logs are ingested, go to http://localhost:8080/accesslogs/q/stats to look at your server's stats. To figure out access statistics for some specific service, constrain the URI using VizieR-like character patterns. For instance, `= /__system__/tap*` will compute the statistics for TAP accesses (the = requests case-sensitive pattern matches). For certain protocols, constraining to POST requests lets you better guess what was going on, but the IVOA S-protocols treat GET and POST analogous.
If you install the `python3-geoip` package, accesslogs/q will also fill the `origin` column with the ISO country code of what geoip thinks is the country of origin of the request. Authorities misguided enough to want to see logs in the first place tend to be so thoroughly misguided as to be interested in "data" of that kind, too. Since origin is even more speculative than access counting, the current public interface does not show it in any way. But if you need some figures for some tedious report, you could execute on your local postgres prompt something like:

```
select
  origin,
  count(*) as ct
from accesslogs.accs
where acc_year in (2020, 2021)
group by origin
```

and perhaps make a beancounter or two happy.

Validating your Services

While DaCHS tries to follow the standards in order to ensure maximum interoperability, there are always things that can go wrong; in particular, there are many ways write RDs such that the resulting services are invalid.
Some places operate validators that tell you if there is something wrong with your site. If you are logged in as gavoadmin, you can run some of these interactively from the service info pages. We recommend, however, to peruse your results on http://heasarc.gsfc.nasa.gov/vo/validation/vresults.pl (the interface lets you select just your services).
Also, if you run a TAP service, you should run something like:

```
stilts taplint tapurl=<root-url>/tap | grep "^E-"
```

– stilts you can get from where you get TOPCAT from. Errors here could be due to DaCHS mis-implementing TAP or because DaCHS implements some TAP proposals stilts doesn't expect; in case of doubt, as on dachs-support. However, it is far more common that your metadata got our of whack for some reason or other. For instance, you may have changed RD metadata (which controls the output on the tables endpoint) without running `dachs imp -m` which updates TAP_SCHEMA). An easy way to fix all such problems is to just run:

```
dachs imp //tap refresh-schema
```

(since 2.9.3; use `dachs adm updateTAP` before).

Dealing with RD Changes

A DaCHS server will in general try to reload an RD if it finds it is changed on disk. If the updated RD cannot be loaded (for instance, because it is invalid XML), it will keep the old RD and leave error messages in the logs (dcErrors and dcInfos) – so, keep these in view when editing RDs under live services.
DaCHS will *not* pick up changed RDs under some circumstances:

- Built-in `__system__` RDs are not controlled. The main reason here is that these may actually have archive members rather than disk files behind them.
- Only the file the RD was loaded from is checked. This means that if you override a built-in `__system__` RD with your own version in `inputs/__system__`, DaCHS will not automatically pick that up.
- Of course, if you edit a file only used by an RD (e.g., a custom core), DaCHS will also not notice a change; it only looks at the RD.
- If you access an RD with no corresponding file and create that file afterwards, that change will also not be picked up automatically.

When DaCHS doesn't notice a change to the RD, you can still expunge it from the server using `dachs serve exp` if you have set The Admin Password).

Admin Interfaces

Admin Web Interfaces

Some administrative operations on DaCHS can (or have to) be done from its web interface. To use these features, you have to set The Admin Password. You can then use the "Log in" link in the side bar using `gavoadmin` as the user name.

If you are logged in as gavoadmin, you should see an "Admin me"-link in the side bar of services. The page behind that link lets you block all services on the respective RD – where blocking means all requests are rejected until the RD is reloaded – and reload the RD (this is equivalent to running `dachs serve exp`).

In the form, you can also set scheduled down times. This is for VOSI, an interface clients could use to figure out whether a service can reasonable be expected to work. Since there don't seem to be clients exploiting the VOSI endpoints for such purposes so far, you probably don't need to bother.

You can directly access the administration panel for an RD by accessing `/seffe/<rdId>`, e.g , `seffe/__system__/services`.

There are several more or less introspective resources within DaCHS that do not need authentication. Among those, there you should know `/browse`. This gives a list of all RDs that have (ivo or local) published services or data in them. Links on the RD ids lead to info pages on the RDs, in particular giving tables and services within the RD.

Admin CLI Interfaces

You can also perform various housekeeping operations using `dachs admin`. Try `dachs admin --help`. This includes User/Group Management, precomputing automatic previews, and creating registry records for deleted services that got lost.

Sometimes people execute stupid TAP queries on your machine (e.g., constraining ra and dec in separate intervals, which is both slow and physically at least doubtful) and you would like to give them a friendly nudge on how to do it better. This is when `dachs admin tapabort` is your friend.

Call it with a TAP job id (which you can obtain from `/tap/async` unless people authenticated; if they did authenticate, you will have to check the `tap_schema.tapjobs` table directly in postgres) and a helpful (!) text to abort a TAP job and set it to an error state giving a short explanation what happened and the helpful text.

robots.txt

DaCHS answers to requests for robots.txt with a built-in resource that forbids to index URLs starting with `/seffe` and `/login`. You may want to keep other pages out of indices. To exclude those, add a file `robots.txt` in your `webDir` (`/var/gavo/web` by default) and add lines like, say:

```
Disallow: /browse
```

The built-in rules will be prepended to whatever you specify in your user robots.txt. For more information on what you can put into robots.txt, see Robot exclusion standard.

Two-Server Operation

When you want to run the database server on a different machine than the DaCHS server, the setup becomes a bit more involved. First, you only install python-gavodachs2 (rather than gavodachs2-server) on the application server.
Then, you need to configure the two servers to talk to each other. Details depend on your setup, and you may want to apply extra hardening over the following. This assumes the machine DaCHS runs on is called appserver.local, the database server is dbserver.local
To enable remote queries on the database server, on dbserver.local do something like:

```
$ cd /etc/postgres/<version>/main
$ sudo vi posgresql.conf
  # edit so that listen_addresses = '*'
  # -- without that, the server only listens on the loopback address
$ sudo vi pg_hba config
  # add a line like
  # host    all             all             appserver.local/32      md5
  # below the access line enabling access for 127.0.0.1
  # you can be more restrictive (e.g., "gavo" instead of the first "all",
  # but that's left as an exercise to the reader
$ sudo service postgresql restart
  # the following assumes you have the same username on appserver and on
  # dbserver, otherwise use the username on *appserver*.
$ sudo -u postgres createuser -sP `id -nu`
$ createdb -Ttemplate0 --encoding=UTF-8 --locale=C gavo
```

After that, you have a database DaCHS can configure from your account on appserver, using the password you gave in the createuser step.
To do that, on appserver.local say:

```
$ dachs init -d "host=dbserver.local user=<yourname> password=<pwd> dbname=gavo"
```

After that, everything should work as in the one-machine case.
Hint: do something like:

```
export PGHOST=dbserver.local
export PGUSER=<yourname>  (or gavoadmin, if you prefer)
```

in your appserver's `.bashrc` (or equivalent), and `psql gavo` will give you a database shell on dbserver. If you trust your appserver, you can even give your password in .pgpass. But don't complain if this bites you later...

Enabling HTTPS

DaCHS can natively speak HTTPS; you have to let it claim port 443 on all the network interfaces you bind it to, though.
If, on the other hand, you use an external HTTPS termination (usually a reverse proxy), your best shot is probably to configure the reverse proxy's HTTPS URL as serverURL. You cannot currently have parallel HTTP in such a configuration, which is a bit unfortunate (see HTTP preferred). With external HTTPS termination, also set `adaptProtocol` to `False` in `/etc/gavo.rc`:

```
[web]
serverURL: https://terminating.pro.xy
adaptProtocol: False
```

When, on the other hand, having DaCHS speak HTTPS natively, all you need to do is give it a secret key and a certificate (which really is a public key with a CA's signature); these need to come concatenated in PEM format in a file called `/var/gavo/hazmat/bundle.pem`. Just so the whole crypto thing doesn't become a total carnival, you should have the hazmat directory owned by root and with permissions 700. DaCHS will only read it when starting up using `dachs serve` and before dropping privileges. That way, if there's a minor security issue in DaCHS, you at least don't necessarily lose your private key.
Note that DaCHS only supports https on port 443 because that's all you are likely to get certificates for. Exception: `dachs serve debug` is intended to be run with user privileges and hence will listen to 40433 if it can read bundle.pem (which it shouldn't) – of course, you will get certificate errors when accessing that. But that's for debugging only anyway.
If you really don't want to use letsencrypt (see the next section) and get your own key-certificate pairs, here's what will likely work:

```
  (become root)
mv my-key.key my-key.crt /var/gavo/hazmat
  (that's a move because you don't want secret keys linger around
  all over the place)
cd /var/gavo/hazmat
cat my-key.key my-key.crt > bundle.pem
  (exit root shell, restart DaCHS)
```

In case you got a bundle of intermediate certificates on top, just append that to your /var/gavo/hazmat/bundle.pem.

Letsencrypt

To get a certificate that is (hopefully) widely accepted, we strongly recommend you use letsencrypt. DaCHS has built-in support to update these certificates in time.
Here's how to go about it:

```
  # DaCHS internally calls a tool called acme-tiny; we much rather trust
  # it than any crypto code we'd hack together
$ sudo apt install acme-tiny
  # now create the directory with the key material; we'll do all of this
  # as root; if you don't like this, at least make it a user distinct
  # from gavo and gavoadmin
$ sudo bash
# cd `dachs config rootDir`
# mkdir hazmat
# chmod 700 hazmat
# cd hazmat
  # generate an account key; this is what identifies you against
  # letsencrypt
# openssl genrsa 4096 > account.key
  # Generate your secret key; these are the holy bits if you believe
  # in https.
# openssl genrsa 4096 > server.key
  # now tell dachs to have the certificate signed; warning: this will
  # try to restart the server
# dachs serve updateCertificate
  # Done -- hit ^D to exit the root shell.
```

Letsencrypt certificates are only good for three months. It's a good idea to renew them every two months. And that won't fly if you don't automate it. Automating it means your server will restart without manual intervention. That's bad because it might be down for a significant time (namely if someone is doing a large download at that time). As said below: HTTPS is an operational liability.

Anyway, add the following to root's crontab (if you installed DaCHS from source, you'll probably have to use the full path to /usr/local/bin/dachs here):

```
21 4 1 1,3,5,7,9,11 * dachs serve updateCertificate
```

-- adapt the time (here, 4:21 local time) to avoid times when it's likely you'll have many users. Also, make sure you receive mails from cron from the machine that does this, because there's a lot that can go wrong here. When updating keys, DaCHS will just write error messages to stderr for cron to transport them. That's probably not a big additional liability, as you should teach the DaCHS server to send mails also for several other DaCHS subsystems.

Finally, to register your https endpoints, add:

```
registerAlternative: True
```

to the `ivoa` section in your `/etc/gavo.rc`. To propagate the information, restart the server and then say `dachs pub -a` to make the registries re-harvest your site.

If your machine is reachable through different host names (our box in Heidelberg, for instance, is both dc.g-vo.org and dc.zah.uni-heidelberg.de), tell DaCHS about it by listing all the non-primary names (i.e., anything that's not in serverURL) in a comma-separated list in `[web]alternateHostnames`.

HTTP preferred

We strongly recommend to have an unencrypted HTTP endpoint, and have that as the primary interface even when you support HTTPS *unless* you actually have authenticated services.

In particular, please to not forcibly redirect to HTTPS versions of pages; it breaks POST requests (which are quite common in the VO) and quite a few other things, too.

DaCHS will give you the same effect in a sane way because (since version 2.2.3) it honours the upgrade-insecure-requests header. That way, clients that are confident they can deal with HTTPS can inform the server, and if the server thinks it's safe to upgrade, it can redirect them. Anything not prepared to deal with or interested in HTTPS will be unconcerned, and everything is under user control, as it should be.

Here's the reasoning for preferring HTTP in the VO: HTTPS is largely ineffective against most privacy breaches: Almost all nation-states can issue certificates that almost all user systems will accept; companies or institutions wanting to eavesdrop on their employees can force http proxies and install their own CA certificates on their employee's machines; and of course most privacy violations are side effects of platforms executing loads of Javascript sailing into the browsers with valid certificates.

On the other hand, networks become a lot more brittle with HTTPS: On the server side, the certificates need to be managed and regularly refreshed. And that's even before operators employ secure key management (which would probably require extra hardware or at least the use of passphrases).

Worse, the client side needs to keep their CA bundles (that's essentially lists of private keys they trust) up to date. With browsers that often carry these along and are updated quite regularly on most boxes, that's marginally manageable. For the VO, clients as a rule are not browsers but tools like TOPCAT or Aladin that may use a CA bundle coming with the Java VM, which often is not as well maintained. And once people start using curl or pyVO, the operating system's CA bundle is used, which on many systems is a mess. The net effect is that a given service may appear to work in the browser and in a script, but not from TOPCAT. Oh my.

Given the miniscule benefits and the serious operational implications, you should provide HTTP endpoints and register your those. If HTTPS is available, DaCHS will tell clients via the Registry's mirrorURL feature. If clients think HTTPS is worth it, they in this way can learn about support for it and use that as they see fit; and for browsers, see above on upgrade-insecure-requests.

Upgrading

Upgrading DaCHS

In general, we try to make upgrades painless, but with a system allowing people to play tricks with intestines like DaCHS guarantees are hard.

If you are running DaCHS for the longer term (i.e., more than 2 year so or), be sure to subscribe to DaCHS-users. We announce new releases there, together with brief release notes pointing to possible spots of trouble.

Ideally, you will have a development system and regression tests in place that let you diagnose problems before going to production.

Upgrading Installations from Debian Packages

(1) Make sure you have enabled the intended distribution (*release* or *beta*) in your `/etc/apt.sources` (or equivalent)

(2) Make sure all RDs DaCHS sees are in order:

```
dachs val -vc ALL_RECURSE
```

(make that `ALL` before DaCHS 2.2).

Warnings you can usually ignore, but try to the understand messages you get. You can hide known-broken RDs from DaCHS by dropping a file named DACHS_PRUNE into their directories.

Broken RDs are very likely to break upgrades. Fix them or remove them.

(3) Do the actual upgrade:

```
apt update
apt upgrade
```

(4) Run:

```
dachs val -tc ALL
```

This will complain if any of our changes break your services. If that is true and the Changelog did not alert you to the issue, please complain immediately. We may still be able to fix things for other people.

`dachs val` with a `-c` option might complain about mismatches between RD and on-disk metadata; there are several reasons why that may happen, including dumb or clever things we've done in the software. In any case, you should fix the problem, most easily by re-importing the respective table.

Upgrading Installations from SVN

If you run from a checkout of our version control system (or, for that matter, from our tarballs), the most important thing to remember is: After every update, do, as a user with ingestion privileges, run the restart sequence:

```
$ dachs val -c ALL
$ sudo service dachs stop # (or whatever you use to stop the server)
$ dachs upgrade
$ sudo service dachs start # (or whatever you use to start the server)
$ dachs test ALL
```

In principle, `dachs upgrade` can run while the server is active, and with most updates, users won't even see errors, but since you need to restart anyway, why bother. On possible failures of the `dachs val` command, see the respective text for the packaged upgrade case.

The steps to update depend on what you did to install out of the subversion checkout. If you initially said `setup.py develop` (which we recommend), all it takes to upgrade is:

```
$ cd <checkout dir>
$ svn update
<run the restart sequence given above>
```

If you instead initially said `setup.py install`, do:

```
$ cd <checkout dir>
$ svn update
$ sudo python setup.py install
<run the restart sequence given above>
```

Upgrading Postgres

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

The text assumes you are using the default cluster, which is called main. If – and that's likely if you installed DaCHS in the olden days – yours has a different name (we used to suggest "pgdata"), you will have to replace "main" with that name throughout. In case you do not remember, at least in Debian you can just `ls /etc/postgresql/<pgversion>` – what you see there is the name of your cluster.

As it is easy to destroy all your tables at once if you mistype things here, it is wise to dump your database somewhere before starting with this:

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

If you are using the Debian package, do that as `dachsroot`; that user has the necessary privileges.
The remainder of this is assumed to be executed as root.
Let's say you are upgrading from 11 to 12. Since we will need these numbers quite a few times, set them in variables – but be *very* careful, if you get this wrong, the commands further down will nuke your database:

```
export OLDVER=13
export NEWVER=15
```

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

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

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

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

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

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

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

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

If these give errors to the effect that the "specified cluster does not exist", that's fine.
During the database update, postgres will be down, and hence DaCHS will not work. For a large database and a complicated upgrade, the update can take several hours (though, really, the last few upgrades for us were minute jobs although we have about 5 TB of tables). You should therefore let people know what is going on. As long as nobody really does anything with VOSI availability, the next best thing is to put your DaCHS installation in panic mode.
To do this, place a file called MAINT into DaCHS' stateDir, maybe somewhat like this:

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

We expect to be back online late afternoon UTC, 2021-08-11.
EOF
```

Then say:

```
pg_upgradecluster -k -m upgrade -v $NEWVER $OLDVER main
```

If you want to keep your database in a non-standard place, add another argument here (e.g., `/big_partition/postgres/$NEWVER`). If you forgot how you did it, you can find the old location using:

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

Do *not* use that old directory for the new cluster, but do keep the postgres version in whatever path you use.

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

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

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

You can now probably run your services again, so, remove MAINT in the stateDir, restart the DaCHS server and run your regression tests (you wrote some, didn't you?).

However, quite typically the table stats are wrong or ununderstandable to the new version, so for large tables you may see timeouts and the like. In pre-15 versions of pg_upgradecluster, it created a shell script to update the statistics. For these versions, see the command's output for how to call that script (just be sure to execute it as the postgres user).

In newer postgreses, simply run:

```
sudo -u postgres /usr/lib/postgresql/$NEWVER/bin/vacuumdb --all --analyze
```

Pg_upgrade has also printed a command that will remove the old cluster; execute it, and finally deinstall the old postgres packages to avoid confusion – locate them somehow like:

```
dpkg -l | grep "postgres.*$OLDVER"
```

What is left is one last step, that is, informing postgres about any upgrade extensions that DaCHS uses. This needs to be run as a database superuser, and so DaCHS itself cannot do it. It can, however, produce the necessary commands. So, become `dachsroot` again and as that user, execute:

```
dachs upgrade -e | psql gavo
```

When that's all done, take a deep breath and relax.

Possible Issues during upgrade

Frankly, there's lots. We'll update this section based your feedback.
(1) Some versions of pgsphere had bad declarations of negators and commutators. These manifest themselves as errors like:

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

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

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

and:

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

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

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

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

Chapter 8

Topics in DaCHS Publishing

This section discusses various DaCHS aspects in somewhat more detail than in what is in DaCHS Basics. The idea is that you can skim the material and return later when a given problem turns up.

Metadata Specials

The reference manual has a fairly extensive section on defining metadata; much of it is tutorial-style, so we encourage you to skim that section at some point.
This section takes a look at some metadata-related points that merit discussion on a wider scope.

Ranking Schemas and Tables

(all this since version 2.9.3)
You can influence the sequence in which certain clients (like TOPCATs after 2024) display your schemas or tables by ranking them. In typcial DaCHS deployments, where there may be numerous schemas with not terribly many tables, this is mainly interesting for schemas, i.e., RDs. To set an RD's "rank", set its `schema-rank` meta:

```
<resource ...>
  <meta name="schema-rank">500</meta>
```

There is nothing wrong with having multiple RDs that have the same schema rank. Clients will show these next to each other.
We recommend choosing relatively large schema ranks; that way, when a data collection comes in that you would like to fit right between resource A and resource B, there will likely still be an integer between them.
At the GAVO data centre, our scheme roughly is:

- 10 is the built-in //tap and //adql RDs (but that's subject to change if it turns out TAP_SCHEMA near the top gets on peoples' nerves).
- 20 for high-value (in the sense of: output of non-trivial funded projects), original (in the sense of: we are/were the first publishers of this data) data collections
- 50 for well-known datasets that people will likely want for cross-matches and similar

- 100 for lower-value original data collections that are likely still of high scientific relevance
- 500 for rather technical, VO-infrastructure resources
- 1000 for data we mainly keep for archiving

Anything else is NULL, which means clients will sort them towards the end.
Against that, in typical deployments the analogous `table-rank` meta probably is less interesting, and you will probably get by with just a few small integers even for fairly complex data.
Not setting ranks will sort the respective schemas and tables to the end of this lists.
If you play around with the ranks, use `dachs imp -m` to update the ranks in the TAP_SCHEMA. If you manipulate a lot of ranks at one time, you can sync the entire TAP_SCHEMA with your RDs by saying `dachs imp //tap refresh`.

Authors, or: Nested Sequential Meta

If you read RMI-Style Metadata in the reference, you will notice that the real meta key for the authors of the data is `creator.name` (please pay particular attention to what we are saying there about the format of that string). However, in the templates and our examples, you will see we are using `creator` throughout. The reason for this is that `creator` is complex metadata (there's name, logo, and possibly more), and setting multiple pieces of complex metadata according to the rules discussed in Defining Metadata has a nasty pitfall with these.
Since this pitfall probably will not bite you outside of `creator`, you might decide to skip this and just never write `creator.name` in your RDs. But then it is safer to understand the issue and read on.
So, suppose you had more than one creator. What you then want is a metadata structure like this:

```
+-- creator -- name (Arthur)
|
+-- creator -- name (Berta)
```

However, if you write:

```
creator.name: Arthur
creator.name: Berta
```

or, equivalently:

```
<meta name="creator.name">Arthur</meta>
<meta name="creator.name">Berta</meta>
```

by rules of Defining Metadata, you will get this:

```
+-- creator -- name (Arthur)
       |
       +------ name (Berta)
```

i.e., one creator with two names, which is both invalid and wrong.
To avoid this, you have to insert a new creator node in between, and write:

```
creator.name: Arthur
creator:
creator.name: Berta
```

This yields the upper (correct) meta structure.

Since the creator.name case is so common, DaCHS provides a shortcut, which you should simply always use to define the creators: if you set `creator` directly, DaCHS will expect a string of the form:

```
<author1>, <inits1> {; <authorn>, <initsn>}
```

(i.e., Last, I.-form separated by semicolons, as in "Foo, X.; Bar, Q.; et al") and split it up into the proper structure. You can mix the two notations, for instance if you want to set a logo on the first creator:

```
<meta name="creator">
  <meta name="name">Chandrasekhar, S.</meta>
  <meta name="logo">http://sit.in/chandra.png</meta>
</meta>
<meta name="creator">Copernicus, N.; Gallilei, G.</meta>
```

The Products Table

DaCHS has a central place in which it keeps metadata on datasets it serves: The product table dc.products. It is used to assign media types (is it a FITS or a text file?), access control (who owns the file and should it be handed out to everybody?), associated thumbnails, and the table through which the product is actually published (i.e., the table underlying the SIAP, SSAP, or whatever service).

This last part, the source table name, is crucial information, because if that table is torn down, the corresponding entries in the product table need to be deleted. Hence, 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>'`.

This source table name is almost always declared in a //products#define rowfilter in the grammar feeding the table, which minimally looks like this:

```
<rowfilter procDef="//products#define">
    <bind key="table">"emi.main"</bind>
</rowfilter>
```

Make sure you do not forget the quotes around the value here – as usual in DaCHS procs, what you bind here are python expressions. You can use macros, so if you're not sure what schema you will eventually choose, you could have written `"\schema.main"`, too.

Unless you are serving FITS files, you also have to give the media type here, for instance:

```
<rowfilter procDef="//products#define">
    <bind key="table">"emi.main"</bind>
    <bind key="mime">"image/jpeg"</bind>
</rowfilter>
```

More on Tables

Table Notes

Frequently, you need to say more about a column than is appropriate in the few-phrase description. In catalogue 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 be present 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`.

Space-Time Metadata

As soon as you have coordinates, you will want to declare coordinate metadata on them, i.e., reference frames, roles played by columns (say, x is the time derivative of y, and x1 is a galactic latitude), the position for which times and positions are given for, and so on. Collectively, this is known as declaring space-time coordinates or STC for short.

The problem here is that this got a bad start in the VO and went downhill from there. And that is why there is, to this day, no real model for how to express these things.

DaCHS has been using a language called STC-S to let operators declare such metadata. It is clear that we will have to change this at some point, and for time metadata, we are already using something else that will probably be roughly stable. Apologies for this state of affairs. We've tried to fix it, but so far without much success.

Legacy STC-S definitions

STC-S was an attempt to define a compact notation for describing STC structures. While the STC-S note still exists, it is clear that neither the underlying model nor the mechanism as such will be retained. Also, few clients interpret the declarations DaCHS makes from this (Aladin does), so it may be not worth the effort to put these declarations in. On the other hand, if you do it now, it will be a lot simpler to add good declarations when there finally is a usable STC model in the VO.

The basic idea is that STC-S strings are defined in children of table elements, with Note-style STC extended to allow references to atomic table columns in quoted strings (and to geometry-valued columns in square brackets), for instance:

```
<table id="withcoo">
  <stc>
    Position ICRS "ra" "dec" Error "e_ra" "e_dec"
  </stc>
  <stc>
    Position FK4 Epoch 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; this is pure annotation. If you refer to non-existing columns, RD parse errors will be thrown.

The full STC-S grammar is humonguos and you don't want to bother. Until a proper STC model comes around, we hope the following examples, taken from various RDs used in Heidelberg, will give you enough material to see what to do:

```
<stc>
  Position ICRS Epoch J2015.0 "ra" "dec"
      Error "ra_error" "dec_error"
    Velocity "pmra" "pmde" Error "sigpmr" "sigpmd"
</stc>
<stc>
  Position ICRS Epoch "epu" "ra_ucac" "de_ucac"
    Velocity "pmra" "pmde" Error "sigpmr" "sigpmd"
</stc>

<stc>
  Position Galactic "glon" "glat"
</stc>

<stc>
  Polygon ICRS [s_region]
</stc>

<stc>
  PositionInterval ICRS [bbox]
</stc>

<stc>
  Position J2000 "raj2000" "dej2000"
  Redshift OPTICAL "redshift"
```

```
</stc>

<stc>
  Time TT 2000-01-01T00:00:00
  Position ICRS Epoch J2000 "alphaFloat" "deltaFloat"
    Error "RAErr" "DecErr"
    Velocity "pmRA" "pmDE"
      Error "PMRAErr" "PMDEErr"
</stc>

<stc>
  Time TT "detection_time"
  Position ICRS "raj2000" "dej2000"
  Spectral "energy_cor" unit keV
</stc>

<stc>
  TimeInterval UTC TOPOCENTER "time_min" "time_max"
</stc>


<stc>
  Position ICRS SPHER3 Epoch J2010 "alpha" "delta" "distance"
  Velocity "mualpha" "mudelta" "radialvelocity"
</stc>

<stc>
  Position ICRS SPHER3 "raj2000" "dej2000" "dist"
    Size "rcluster" "rcluster" -1
    Velocity "pmra" "pmde" "rv" Error "e_pm" "e_pm" "e_rv"
</stc>
```

New-style SIL definitions

Regrettably, the Coords data model is not really usable for annotating database tables. However, you can (and should) use an ad-hoc, DaCHS-specific DM to provide frames and such. You can do that using a little language called Simple Instance Language or SIL that you can use in dm children of table elements. The reference manual has an introduction to the syntax in Annotation Using SIL.

Since version 2.7.3, DaCHS in interprets time and space children from votable:Coords instances. They are defined in dm children in tables in this way:

```
<dm id="obs_pos">
  (votable:Coords) {
    time: (votable:TimeCoordinate) {
      frame: (votable:TimeFrame) {
          timescale: TCB
          refPosition: BARYCENTER
          time0: 0 }
      location: @obs_time
    }
    space:
      (votable:SphericalCoordinate) {
        frame: (votable:SpaceFrame) {
          orientation: ICRS
          refPosition: TOPOCENTER
          epoch: "J2000.0"
        }
```

```
        longitude: @ra
        latitude: @dec
        distance: @dist
        pm_longitude: @pmra
        pm_latitude: @pmdec
        rv: @rv
      }
  }
</dm>
```

To adjust this to your tables, pick a timescale from http://www.g-vo.org/rdf/timescale, a reference position from http://www.g-vo.org/rdf/refposition, and change time0 to the JD of the epoch of your calendar; 0 is for JD itself or 2400000.5 for MJD. If you have some other time origin else, DaCHS can help you. For instance:

```
>>> from gavo import api
>>> import datetime
>>> api.dateTimeToJdn(datetime.datetime(1970, 1, 1))
2440587.5
```

computes the JD of the Unix epoch.

Finally, to set the location, reference the column with a leading @. If you use a literal here, the implicit unit is day. See also STC annotation.

At this point, DaCHS produces VOTable COOSYS and TIMESYS elements from this data. We hope that if and when there is a suitable annotation based on an IVOA data model, we can translate this to whatever it is.

Defining Views

It is quite regularly preferable to represent data in multiple tables (for instance, for normal form reasons); however, outside of TAP you cannot expose such table structures in VO protocols unless you re-join the various tables to some de-normalised table or, in SQL terms, a view. Other reasons to define views include the mapping of nonstandard data to standard table structures, which is what DaCHS does internally for SSAP tables and when publishing anything through obscore.

If you want to manually define a view, declare a table as usual and then give it a `viewStatement` – done. In practice, there are a few things you should keep in mind to make such a view definition maintainable.

First, you will obviously define the tables involved, for instance:

```
<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 catalogue"
    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 catalogue position at catalogue
      epoch and equinox and the catalogue position"
    verbLevel="1" displayHint="displayUnit=mas,sf=1"/>
  <column name="masterNo" type="integer" required="True"
    ...
```

Note, in particular, that postgres does not let you define indices over views (exception: materialised views). This means that anything you want to index must be indexed on the table level (hence the `//scs#q3cindex` mixin on the `master` table). You still should *declare* the underlying indexes; you could write index statements as usual (add `metaOnly="True"`), but it is generally faster and safer to use //procs#declare-indexes-from. Insert it right above your viewDefinition.

A good approach is to keep 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. This is used in, for instance gedr3mock/q. An alternative that lets you keep the table definition as-is is to go through `original`; this is what we do here, inspired by arigfh/q.

One reason we are going for original rather than STREAM is that with original, we can straightforwardly rename some columns, which here we want because in the view, we want to mark columns coming from the master table, like this:

```
<table onDisk="true" id="id" adql="True">
  <meta name="description">
    The stars from the gfh table having counterparts in the master
    catalogue, together with those counterparts.
  </meta>
  <column original="master.catno" name="master_no"/>
  <column original="master.component" name="comp_master"/>
  <column original="master.raj2000"/>
  <column original="master.dej2000"/>
  <column original="master.pmra" name="pmra_master"/>
  <column original="master.pmde" name="pmde_master"/>
  <column original="master.mv" name="mv_master"/>
  <column original="master.mb" name="mb_master"/>
```

When just copying columns, active tags come in handy to copy over column definitions from the two tables; note the use of `<tablename>.<columnname>` in referencing:

```
<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>
```

As shown in the second loop, you can go as far as to programmatically obtain the column names for your source tables. Only do that if you are sure you actually want to follow changes in source table structure in the view.

After declaring the columns, what is left is actually defining the view (see above for the FEED):

```
  <FEED source="//procs#declare-indexes-for"
    sourceTables="master gfh"/>

  <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>
```

The viewStatement essentially is more or less arbitrary SQL. For robustness against changes of table structure or schema or table name changes (as in: you should get errors quickly if things are broken), however, you should probably always start with:

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

The macro `\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 will 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>

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


<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.

Also note that `dachs limits`, when run on an RD, will ignore views, partly because their columns will typcially already have statistics in the originating tables, partly because Postgres cannot TABLESAMPLE views, which may make column statistics really expensive. If your views contain, for instance, unions, that will yield table metadata that is seriously wrong. While you can update the statistics by passing the identifier of the table descriptor into a dachs limits call, it is preferable to not this in the RD. For that, add:

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

to the table element describing the view.

Materialised Views

Materialised views work like regular views, but instead of re-writing the query every time, an actual table is created. This *sometimes* is faster, and you can create indices over the results. The disadvantage is that as the underlying tables change, the materialised view will not change automatically. And of course materialised views take up disk space, and there are quite a few situations in which they actually is slower, sometimes by large factors.

There still are situations in which materialised view are useful. To define materialised views in DaCHS you do as for defining views, except that instead of `CREATE VIEW` you write `CREATE MATERIALIZED VIEW` in the view statement. Of course, any indices you may have on the source tables will not help queries against the materialised view. Hence, you will have to add `index` elements (again) as necessary.

`dachs limits` will skip materialised views, too, by default. For them, that has a lot less sense, so you will almost always want to add:

```
<property name="forceStats">1</property>
```

in materialised views.

You can update materialised views from the postgresql command line, using something like:

```
REFRESH MATERIALIZED VIEW myschema.matview
```

This will lock the relation to concurrent queries, which will wait until the view is updated. There is a CONCURRENTLY version of this avoiding that lock-out, but it will need to be used with care. See the postgres docs if you think that matters a lot for your use of materialised views.

You could tell DaCHS to regularly update a materialised view if necessary, for instance with something like:

```
<execute at="2:45">
  <job>
    <code>
      with base.getWritableAdminConn() as conn:
        conn.execute("REFRESH MATERIALIZED VIEW \schema.matview")
    </code>
  </job>
</exceute>
```

See Element execute for how to specify more complex cadences, and note that DaCHS lives on UTC if you want to keep such updates outside of your local rush hours.

More on Importing Recipes

Doing Incremental Imports

When running archives of ongoing projects, it is usually desirable to import often, but ignore items that are already in the database to make the imports quick and cheap.

The key ingredient to accomplish that is the Element ignoreSources, which lets you tell DaCHS which files of those it found in `sources` it should not ingest (again). What this deals with are paths relative to DaCHS' inputsDir. Unless you did something special, the accref column in tables having data products contains exactly these inputsDir-relative paths.

There are various ways to communicate accrefs to ignore: from lines in a file, by a pattern, or from the database. This last case is what most commonly is used in incremental imports, in the form of the `fromdb` attribute of `ignoreSources`.

DaCHS, however, when you import something, will always tear down all tables being made, so just ignoring some sources will not be sufficient. To make DaCHS keep tables mentioned in `make`, give the Element data an `update` attribute.

In sum, the pattern is:

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

Of course, you have to adapt the source pattern and the table to get the accrefs from.

For non-data product sources (e.g., files you feed into catalogs), you will have to use some other technique, because these do not have accrefs and you will have to manage the sources already processed manually. An example for how this can be done is the `rr.imported` table in rr/q.

Even more on this is found in the reference documentation's Updating Data Descriptors.

Creating pgSphere Geometries

DaCHS lets you store spherical geometries (polygons, circles, and MOCs) in database columns. Obscore and EPN-TAP have the `s_region` column holding such geometries, and you can have custom columns with geometry types that facilitate writing interesting TAP queries. Also, DaCHS is moving towards using pgSphere's spoint for spatial indices (instead of q3c, which has been the preferred spatial index until DaCHS2).

To feed such columns, you first have to choose a type for the column; DaCHS' geometry columns are type-clean, i.e., if a column has a polygon in one row, it all other rows will have to have a polygon (or NULL) there, too – rather than, say, a circle or a point.
The choice of types is:

- `spoint` – a simple point. You should not have spoints in `s_region`, as common queries will run contains or intersects against them, and that rarely does what users expect with points.
- `scircle` – a spherical circle. pgSphere limits those to radii smaller than pi/2.
- `spoly` – a spherical polygon, defined through a number of edges. pgSphere currently has rather severe limitations on what these can be; in particular, you cannot have intersecting edges, and their maximum size has limits, too. For complex geometries, prefer smoc.
- `smoc` – a Multi-Order Coverage map (MOC). MOCs are, conceptually, lists of HEALPixes and thus can represent arbitrarily areas on the sky. If you have possibly large, possibly non-convex, possibly non-simply connected areas, they are what you are looking for.

Creating values of these types in DaCHS would usually use classes from the `gavo.utils.pgsphere` module, which is available as `pgsphere` in rowmakers. These are usually constructed with arguments in rad; for this and other reasons, we recommend using their `fromDALI` class methods. These take the canonical representations forseen by DALI for VOTables, which are usually just arrays for floating point values in degrees.
So, to fill an spoint column, you would say something like:

```
<var key="center"
  >pgsphere.SPoint.fromDALI([
    float(@centerLong),
    float(@centerLat)])</var>
```

A circle with a radius given in arcminutes would be:

```
<var key="s_region"
  >pgsphere.SPoint.fromDALI([
    float(@ra), float(@dec),
    float(@radius)/60])</var>
```

A polygon is constructed with longitudes and latitudes interleaved:

```
<var key="s_region"
  >pgsphere.SPoly.fromDALI([@x1, @y1,
    @x2, @y2, @x3, @y3])</var>
```

The DALI representation of a MOC is a string of whitespace-separated order/pixel lists. The details are in section 2.3.2 of the MOC specification. There are various ways in which you can come up with such strings.
If you have a list of points, decide on the order you want your MOC to give the coverage on (6 is about one degree, and each order gives you a factor of two in linear resolution). Then, make circles of about that resolution around your points, turn these into circles and join the MOCs. For instance:

```
<apply name="compute_coverage">
  <setup>
    <bind name="order">10</bind>
  </setup>
  <code>
    circles = []
    for x, y in @points:
      circles.append(
        pgsphere.SCircle.fromDALI([x, y, 6/order]))

    m = circles.pop().asSMoc(order=order)
    for c in circles:
      m.moc += c.asSMoc(order=order)

    @s_region = m
  </code>
</apply>
```

This is based on the fact that a `pgsphere.SMoc`'s `moc` attribute (currently) contains a pymoc.MOC instance; even if we change the backend implementation (which is likely), the pattern should keep working.

Similarly, for a bunch of polygons, you could do this:

```
<apply name = "compute_region">
  <setup>
    <bind name="order">10</bind>
  </setup>
  <code>
    @s_region = pgsphere.SPoly.fromDALI(polys[0]
      ).asSMoc(order)
    for vertices in polys[1:]:
      @s_region.moc += pgsphere.SPoly.fromDALI(vertices
        ).asSMoc(order)
   </code>
</apply>
```

Note that the HEALPix library DaCHS uses currently (2020) cannot deal with non-convex polygons and may slightly over- or underestimate the coverage of circles and polygons.

If either limitation bothers you: The CDS HEALPix library provides a nice alternative for which there is a convenient wrapper in CDS-HEALPix-python. If you produce an ASCII MOC from whatever you compute using these tools, you can pass them to `pgsphere.SMoc.fromASCII`. See also openngc/q for sample usage of the relatively naked rust HEALPix library.

Skipping Things

Sometimes there are a few "broken" things in material that is otherwise just fine: A couple of broken FITSes, or some lines of a CSV that should not make it into a database table.

While the `-c` flag to `dachs import` will let you conveniently ignore individual sources (note that this is rather certainly not what you want with the bad rows), it will also hide genuine errors and thus should only be used in development. In production, you should skip the bad sources or rows.

Skipping Sources

You can skip known-bad sources by accref using Element ignoreSources, using the `fromfile` attribute.
More commonly, however, you can tell sources to skip by criteria like certain items in the file names, the presence or absence of certain FITS headers, or similar. The preferred method to do that these days is raising `SkipThis` in a row filter. For instance, to skip all files that have `_failed.` in their name:

```
<fitsProdGrammar>
  <rowfilter>
    <code>
      if "_failed" in rowIter.sourceToken:
        raise SkipThis("Ignoring failed observation")
      yield row
    </code>
  </rowfilter>
  ...
```

While you could just as well make the `yield` conditional on the negation of your condition, raising `SkipThis` makes the intention clearer and may (some day) give clearer indications of what is going on in the UI.
If you want to skip whenever a certain header is missing, do something like:

```
<fitsProdGrammar>
  <rowfilter>
    <code>
      if not "CD1_1" in row:
        raise SkipThis("Ignoring uncalibrated observation")
      yield row
    ...
```

In case you need to ward off totally broken FITS (or other) files that crash the FITS parser, you cannot use a row filter because that will only run once the parser has run. However, you can abuse the Source Fields computer to skip them before the FITS parser sees them:

```
<fitsProdGrammar>
   <sourceFields>
      <code>
        try:
          _ = pyfits.open(
            os.path.join(
              base.getConfig("inputsDir"),
              sourceToken))
        except Exception as ex:
          raise SkipThis("Skipping utterly broken {}: {}".format(
            sourceToken, ex))
        return {}
      </code>
   </sourceFields>
```

– though my advice in such a situation would be to fix the broken files using an external processor (see processors.html).
DaCHS still contains a declarative facility for skipping sources in the form of Element ignoreOn and Triggers. Although experience has shown these to be too unflexible to be worth it, still feel free to use them – we will not remove them any time soon.

Skipping Rows

To skip individual rows of a source, raise `IgnoreThisRow` in a procedure application, like this:

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

Note that if you raised `SkipThis` here, you would skip the entire rest of the source, which is probably not what you want.
Row makers also admit the Element ignoreOn mentioned in Skipping Sources; as for them, we believe the declarative means rather typically are are not flexible enough, and skipping using Python code is usually about as clear and evolvable.

More on Grammars

As discussed in Parsing Input Data, grammars are what turn just about any kind of input into neat sequences of Python dictionaries that are then processed (and possibly multiplied) in row filters; whatever comes out of this is then handed over to row makers and turned into database rows.
There are many grammars available in DaCHS. This section discusses some of them in a more leisurely fashion than the terse descriptions in the reference documentation, and it covers a few additional aspects of what can go on in any grammar.

Source Fields

It is a fairly common situation that you want to make available certain properties of a source file to the row makers; the most common case is that the data providers have encoded information like object names, times, or whatever, in file names, or that there is some auxiliary information in a different files.
In these situations, it is often efficient to use Element sourceFields. 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.
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")`).
As a trivial example, here is how to only parse a field identifier from a source name once per source (rather than have this expression in a row maker and to that once per row):

```
<!-- this sits in a grammar element -->
<sourceFields>
  <code>
    return {"field":
```

```
      int(sourceToken.split("bul_sc")[-1].split(".pm.gz")[0])}
  </code>
</sourceFields>
```

A more complex example, taken from lightmeter/q, looks up some calibration values from a different table based on an identifier again pulled from the source file – clearly, you wouldn't want to do one query per row when there's thousands of rows coming from one source:

```
<sourceFields>
  <code>
    stationId = os.path.split(os.path.split(sourceToken)[0])[1]
    if re.match("[0-9]+$", stationId):
      raise base.SkipThis("Archive directory")
    try:
      with base.getTableConn() as conn:
        stationPars = list(conn.query(
          "SELECT calibA, calibB, calibC, calibD, timeCorrection"
          " FROM lightmeter.stations"
          " WHERE stationId=%(stationId)s", locals()))[0]
      calibA, calibB, calibC, calibD, timeCorrection = stationPars
    except IndexError:
      raise base.ValidationError("No station data for %s.  Import"
        " stationsdata first."
        %stationId, "stationId", stationId)
    if calibA is None:
      raise base.SkipThis("No calibration for %s."%stationId)
    srcKey = "/".join(utils.getRelativePath(sourceToken,
      base.getConfig("inputsDir"), liberalChars=True).split("/")[2:])
    return locals()
  </code>
</sourceFields>
```

On getTableConn, see Database Queries.

Note, however, that you need a special pattern when you want to query the table that you are filling. This is because `sourceFields` run in the midst of a transaction updating the table. So, something like:

```
<code>
  <!-- will deadlock, don't do it like this -->
  with base.getTableConn() as conn:
    conn.queryToDicts("select ... from <table mentioned in make>")
</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 dachs imp will just sit there and wait (see also commonproblems.html#deadlocks).

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

```
<code>
  yield data.connection.queryToDicts(...)
</code>
```

preFilters

DaCHS Grammars parsing from files have a `preFilter` attribute. This can contain a pipeline that has to read from stdin and produce its output on stdout.
The typical use case is to decompress input files without bothering DaCHS, e.g., using `preFilter="zcat"` (for which there's the historical `gunzip="True"` alias – don't use that any more), `preFilter="bzcat"`, or `preFilter="xzcat"`.
However, if you have some weird format for which there is a dumping tool that produces something that can be turned into CSV from a stream coming in from stdin something like the following provides an elegant way to get the data in with minimal code:

```
<csvGrammar preFilter="dump-weird-format-to-stuff | stuff-to-csv"/>
```

reGrammars

The Element reGrammar is the swiss army knife of text-parsing grammars when neither element columnGrammar nor csvGrammars will do. The idea here is that you give one regular expression each to separate the file into records and to split the 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 other 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>
```

The `recordCleaner` used here is an RE matched against every record and that contains exactly one capturing group. If it matches, the group will be used to apply the separators; in this case, it simply discards the parentheses you have in your typical database dump. Records not matching at all are discarded (which, admittedly, makes it easy to miss errors in ways that will lose data silently).
An alternative, and somewhat safer, way to get rid of junk in the input is to use the `commentPat` attribute. That is an RE that is removed from record before trying to parse it. The reference documentation mentions the classic case: `(?m)^#.*$` will throw away lines starting with a hash.
Another gung-ho feature of reGrammars is the `lax` attribute. Normally, reGrammars will fail on records that do not have exactly one field per name you put in. Some ugly input may skip "optional" fields at the end – with `lax="True"`, DaCHS will map those to empty strings.
Finally, sometimes files have junk at the end. This is when `stopPat` comes in handy: When a record matches that RE, the entire remainder of the input is ignored.

fitsProdGrammars

This grammar exposes FITS headers as rawdicts. Since both 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 images.

If you have FITS tables, this grammar is not for you; use Element fitsTableGrammar instead, and if you have large FITS tables, have a look at Grammars in C (even if you don't know C).

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 specialised 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.

You also have to use `qnd="False"` if you parse from compressed FITS images.

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 Element mapKeys 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, because that's where you will look for them in two months.

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* attribute. 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, incidentally, transparently supported if their file names end in ".gz").

csvGrammars

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).

CsvGrammars by default expect input files to follow the convention that the first row contains the column names, and it will use whatever is there as rawdict keys. If these

names are not usable as Python identifiers, use Element mapKeys to map them to other keys so you can have nice @-references in your row maker.
If the first CSV line does not contain the names, you must use the `names` child to give these names, like this:

```
<csvGrammar names="ra, dec, mag"/>
```

The csvGrammar will not complain if there are more fields in the file than you give names for; any extra values are reachable through `@NOTASSIGNED` (which should probably only be used to catch exceptional circumstances).
If your input files follow the (silly) convention of having the colum names in a comment in the first line, like this:

```
# ra, dec, mag
13,2, 14.45, -0.33
```

you have to give `names` as above and make the csvGrammar ignore the first line with `topIgnoredLines="1"`.
There is currently no way to ignore comments in any other way; if you need that, you should probably retrofit that functionality in Python's csv module.

Python-defined grammars

Sometimes inputs are so nasty that DaCHS' grammars really can't be used to parse them. For these situations, DaCHS has Element customGrammar and Element embeddedGrammar that let you just slap in pieces of Python yielding dictionaries as grammars.
First off: before you go there, think hard if one of DaCHS' built-in grammars really can't do the job (including Element binaryGrammar). Long experience has shown that custom code is much more likely to break in the evolution of your system, Python, and DaCHS itself.
Custom grammars come in modules of their own, which is a good thing because you can test them outside of DaCHS. But they break the principle of "all you need is in the RD", so don't use them unless you must. An introduction to them is in Writing Custom Grammars in the reference documentation.
Embedded grammars are somewhat more light-weight. They in particular come in handy when you need to generate a few rows, like this:

```
<embeddedGrammar>
  <iterator>
    <setup>
      <par name="testData">"a"*4</par>
    </setup>
    <code>
      for i in self.sourceToken:
        yield {'index': i, 'data': testData}
    </code>
  </iterator>
</embeddedGrammar>
```

Essentially, they are usual DaCHS procedures that `yield` rawdicts in whatever way you like. They see the source to be parsed in `self.sourceToken`. While this usually is a file name (that you woujld then pass to `open`, you can also use the string for something else, as is done here. Combined with:

```
<sources><item>ab</item><item>cd5</item></sources>
```

the grammar above would produce the rawdicts:

```
{'index': 'a', 'data': 'aaaa'}
{'index': 'b', 'data': 'aaaa'}
{'index': 'c', 'data': 'aaaa'}
{'index': 'd', 'data': 'aaaa'}
{'index': '5', 'data': 'aaaa'}
```

Embedded grammars are intended to be used for small and simple tasks, and if anything goes wrong, the error messages produced by default are not very useful. So, when things get a bit more complicated, program defensively and keep a note of where in the source you are, somehwhat like this:

```
with open(self.sourceToken) as f:
  for line_number, line in f:
    try:
      ...
    except Exception as ex:
      raise utils.SourceParseError(
        "Bad line: {}".format(ex),
        offending=line,
        location=str(line_number+1),
        source=self.sourceToken)
```

Grammars in C

If you have large data collections (millions to billions of rows), going through the all the steps of parsing, row making, and postgres ingestion with lots of Python in between can become prohibitively slow. In such cases, you can use Element directGrammar (or "a booster" for short). They let you bypass everything DaCHS and almost everything Postgres and produce binary dump material that is just slurped into the database at maximum speed.

This normally requires some knowledge of C, although DaCHS tries to help you if you're not terribly C-savvy. The one exception are FITS binary tables; DaCHS can generate C code for pulling them in without further intervention.

Boosters are a longer story, which we tell in :dachsdoc:booster.html.

More on CondDescs

CondDescs were introduced in Service Definitions above. Since they are powerful tools when building service interfaces, we cast a second look at them here.

Automatic and manual control

The standard idiom introduced above:

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

adapts to protocols; for instance, if somecol is a floating point variable, the condDesc will accept VizieR-like expressions in HTML forms, PQL-like intervals ("min/max") in legacy DAL protocols and DALI-like intervals ("min max") in modern DAL protocols.

Sometimes you need more control, for instance, when there should always be just one single string regardless of what renderer the service is used through. In that case, define the input key directly:

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

The extra attribute is there to make sure that even a column that is required in the table remains optional in the query (you can skip that if your column is not required in the first place). You can also define the input key from scratch:

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

If you add an enumeration of possible values, like this:

```
<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>
```

DaCHS will try to communicate the choices to clients; in HTML forms, for instance, this will translate into a pop-up menu.

Limiting condDescs by renderer

In order to let you emulate `buildFrom` behaviour of making different input key types depending on the renderer even when you cannot use `buildFrom`, there are 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

In all the condDescs we have seen so far, the SQL generated becomes conjunction (i.e., OR expression) of the condDesc's input key's individual SQL expressions – unless you set the `joiner` attribute to AND, that is (incidentally, constraints from different condDescs are always combined using AND in DaCHS).

The SQL expression derived from an input key is a simple comparison for equality for plain types, a test for overlap for interval-valued parameters, and even more complex expressions for VizieR-like expressions in the form renderer.
For complete control over what SQL is generated, condDescs may contain code in Element phraseMaker. 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 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 `base.getSQLKey(name, value, outPars)` function, where name is more or less arbitrary but should usually be the input key name, value is what you would like to see in the SQL, and outPars is a dictionary that you get passed into your phrase maker. All that you should do with outPars is pass it to this function.
`getSQLKey` 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"%(
          base.getSQLKey("date", baseTS-dt, outPars),
          base.getSQLKey("date", baseTS+dt, outPars))
      </code>
    </phraseMaker>
  </condDesc>
```

STC coverage

Most resources you will publish have a certain coverage in space, time and spectrum. The Registry can represent this coverage, and this is obviously really helpful in many discovery operations. So, if you can, declare your resources' coverages (or "footprints").

In DaCHS, you can do that manually using the children of Element coverage. These are:

- `spatial`: This contains something called an ASCII MOC, which is explained in the MOC Recommendation . Coming up with those essentially always requires some tooling, so unless you really want to say "all sky" (which is `0/0-11`) you should probably let DaCHS work it out (see below).

- `temporal`: You can have multiple of those; each needs to be a space-separated pair of either floats (interpreted as MJD) or DALI-conforming ISO strings (YYYY-MM-DD). For many tables, DaCHS can automatically work this out, but it is not very diligent and will just use the minimal and maximal time within the resource. In particular when you have resources with multiple observation runs, it is preferable to manually enter them.

- `spectral`: Again, DaCHS lets you have multiple of those, and they are pairs of floats giving spectral coordinates in Joule (in case you are wondering: it's energy so it properly works for non-EM messengers as well, and it's Joules because that sucks for everyone except ultra-high-energy cosmic ray people who deserve a break because they have so few messengers). Giving these manually is a particularly good idea if you have several narrow bands, because DaCHS will not notice that when automatically determining the coverage.

For many common table types (SSAP, Obscore, SIAP, SCS), DaCHS can automatically work out the coverages (with the caveats mentioned above). To make it do that, use Element updater, which lets you define the table DaCHS should derive the coverage from (or, for odd cases, several tables for the various axes). The results of these coverage estimations are stored in the database table `dc.rdmeta`.

Note there can only be one coverage per RD. This means that if to things have different coverage, they must be in two different RDs.

You can mix and match manual and automatic coverages; DaCHS will not overwrite manually given coverage information. Hence, if you say:

```
<coverage>
  <updater sourceTable="main"/>
  <spectral>3.02e-19 3.03e-19</spectral>
  <spectral>1.63e-18 1.64e-18</spectral>
</coverage>
```

you would let DaCHS try to determine the temporal and spatial coverage from the `main` table, but you would say that your spectral coverage is around Lyman α and Balmer α.

Writing Examples

As part of the DALI standard, there is a convention for giving examples for service usage; the idea is to produce HTML that is useful for human consumption with additional, RDFa-based, markup to let clients figure out how to fill their interface forms; the prime example of where that works nicely is in TOPCAT's TAP client (check the examples button, the service-provided entry).

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

TAP examples

The most immediately useful examples are those for TAP/ADQL. While examples normally reside in the service elements they illustrate, that doesn't work for TAP, because the service is defined in the built-in RD `//tap`.

Rather than change that (with all the trouble that would give when upgrading DaCHS), instead add the TAP examples in userconfig.rd; to prepare for that, see The userconfig RD.

Once you have your userconfig.rd, locate the STREAM with the id `tapexamples`. One example is already given in the userconfig.rd of the distribution; take that as a template for your own examples.

Each example resides in an `_example` meta element. This has a mandatory `title` attribute, and its content format is fixed to ReStructuredText. 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>
```

In addition to title, the only thing mandatory is a block marked `.. tapquery::` that contains the ADQL query you want to show. A typical client would fill this into whatever its UI provides to write queries.

Of course, you should explain what the query does and how it does this. ReStructuredText lets you have basic markup, including, if necessary, section headings if your example gets long enough.

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.

Note again that DaCHS does not pick up changes to `userconfig` automatically. Hence, after adding or changing examples, you have to run:

```
dachs serve exp % //tap
```

on the server (if you have set The Admin Password).

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. We cannot recommend a client to try them with at this point, but of course having services with useful examples will make client support more attractive.

To add an example to a datalink service, add an `_example` meta with a title attribute 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`. Since DaCHS picks up changes to "normal" RDs automatically, you will immediately see changes to your example text.

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>
```

As you can see, generic parameters are defined using a `genparam` interpreted text role; the content of that role is the parameter name and then, in parentheses, the parameter value. Don't worry if your parameter value has a closing parenthesis in it; DaCHS only recognises the parenthesis that ends the genparam if it is at the end of the interpreted text.

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

DaCHS does not yet have support for the *capability* property defined by DALI. Ask if you need it.

Hierarchical Examples

In particular for TAP, you may collect so many examples over the time that a flat list becomes unwieldy. In that case, you can collect examples in stand-alone DALI examples documents (which are XHTML) or other services and reference these extra examples collections from, say, the TAP examples. You can co-locate these within other services or create an extra examples-only service, for instance:

```
<service id="ex" allowed="examples">
  <meta name="title">RegTAP examples</meta>
  <meta name="description">This service has the examples from the
    RegTAP spec in the form of a DALI examples document.
  </meta>
  <meta name="_example" title="Find all TAP services; return their
    accessURLs.">
    As the capability type is in :taptable:‘rr.capability‘, whereas
    the access URL can be found from :taptable:‘rr.interface‘, this
    requires a (natural) join.

    .. tapquery::
      SELECT ivoid, access_url
      FROM rr.capability
      NATURAL JOIN rr.interface
      WHERE standard_id like 'ivo://ivoa.net/std/tap%'
         AND intf_role='std'
         AND authenticated_only=0
  </meta>


  <meta name="_example" title="Find all SIA services that might have spiral
    galaxies">
    Translating the intention into "find all SIA services that mention
    spiral in either the subject (from :taptable:‘rr.res_subject‘),
    the description, or the title (which are in
    :taptable:‘rr.resource‘)", the query would become:

    .. tapquery::
      SELECT ivoid, access_url
      FROM rr.capability
        NATURAL JOIN rr.resource
        NATURAL JOIN rr.interface
        NATURAL JOIN rr.res_subject
      WHERE standard_id like 'ivo://ivoa.net/std/sia%'
        AND intf_role='std'
        AND (
          1=ivo_nocasematch(res_subject, '%spiral%')
          OR 1=ivo_hasword(res_description, 'spiral')
```

```
        OR 1=ivo_hasword(res_title, 'spiral'))
  </meta>
  <nullCore/>
</service>
```

If this were within the RD rr/q, you could then amend your `tapexamples` STREAM in the userconfig with:

```
<meta>
  moreExamples: rr/q#ex
  moreExamples.title: RegTAP queries
</meta>
```

Suffiently advanced clients will pick this up and display it in something like a submenu. You can repeat moreExamples, and you can use URLs rather than service references in there (provided there are indeed DALI examples documents at those URLs). For instance, to include the Heidelberg obscore examples (which may work for you to some extent), you could add:

```
moreExamples: https://dc.g-vo.org/static/obscore-examples.shtml
moreExamples.title: ObsCore queries
```

Inspecting the Triples

The best way to make sure your RDFa actually says what you intend it to say is to transform it to the Turtle (alias N3) format, which is reasonably human-readable when you have grokked the basics of RDFa (recommended reading).
There used to be a simple web service for this kind of thing, but it seems it has fallen into disrepair, and in general RDFa tooling at this point seems somewhat marginal. In early 2023, what I did was create a virtual python environment (see virtualenv) and then running:

```
pip install pyrdfa requests
```

(don't do that with sudo: only use pip with virtual environments). Then you can run a program like this:

```
from rdflib import Graph
g = Graph()
g.parse("http://dc.g-vo.org/tap/examples", format='rdfa')
print(g.serialize(format="turtle"))
```

(changing the URI as appropriate).

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 – many commonly encountered problems, in particular the ones that give particularly baffling messages, are explained there with suggestions for how to fix them.
- The dachs 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 `dachs` and the subcommand name, as in `dachs --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 `dachs 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 `dachs`.
- If you're trying to figure out server behaviour, don't run the server daemonized but use `dachs serve debug` instead; this won't detach and log to stdout. The user executing the serve command needs to be in the `gavo` group, or you'll get permission problems.
- With this (or if the problem is in non-server DaCHS code in the first place), the Python debugger is your friend. The `dachs` 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 `dachs 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()`.
- 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 dachs imp q create
  ```

  Don't be too alarmed by the deluge of queries you'll see while DaCHS starts up; this is just making sure that the tables DaCHS cannot live without are present.
- If `dachs 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 Rowmakers

When debugging rowmakers, it helps keeping in mind that there are two dictionaries that play a role here: The incoming rawdict and the rowdict that is being built. Having a clear image which key is where at which point is helpful. Also, it is important to

remember that first all `var` elements are processed, then all `apply` elements, and finally all `map` elements. Within each class, elements are processed in the order of definition, with `simpleMaps` and `idmaps` after all explicit maps.

Since only applys contain larger amounts of code, debugging will mostly focus on them. If you want to look at one of your own applys, just set breakpoints as discussed above (`import pdb; pdb.set_trace()`). If you'd just like to have a look at what `vars` and `results` look like at some point, just add a `<apply procDef="//procs#debug"/>`. This will, again, drop you into a debugger.

One thing that may baffle you when you write row makers are unexpected primary key clashes (or indeed, violations of other constraints enforced by the database) or just failures where invalid data is passed to the database. Since DaCHS ships out rather large batches of rows (5000 by default) to the database if you let it, the errors will typically occur far from the lines that cause them.

In such a situation, pass `-b 1` to `dachs imp` – this will slow down the import quite dramatically, but the import will fail exactly at the source or row that causes the problem.

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 `dachs serve debug`. You can even drop into the python debugger while the server processes a request (the server will of course block while you are in the debugger). It's a bit tricky to communicate with the debugger in between the log messages of `dachs 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.

Note that in python3, the debugger by default changes SIGINT processing so pressing Control-C will drop you back into the debugger. With `dachs serve debug`, that behaviour is unwelcome, because it makes terminating the server a bit of a steeplechase. Hence, when setting breakpoints in server code, use:

```
import pdb;pdb.Pdb(nosigint=True).set_trace()
```

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 put a special RD somewhere (e.g., `__tests/manhole.rd`) and use some RD-embedded code to introspect the server. We usually use a datalink service for this since it keeps things nicely self-contained – an obvious alternative with less bending could be a python core returning a pair of `text/plain` and a string.

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

```
<service id="cq" allowed="qp">
  <property name="queryField">prop</property>
  <pythonCore>
    <inputTable>
      <inputKey name="prop" type="text"/>
    </inputTable>
    <coreProc>
      <code>
        from gavo import base
        col = base.resolveCrossId("arihip/q#main.hipno")
```

```
        payload = "Result: %s"%(
          getattr(col, inputTable.getParam("prop"), "(err)"))
        return ("text/plain", payload.encode("ascii"))
      </code>
    </coreProc>
  </pythonCore>
</service>
```

See the qp renderer for how the `queryField` magic is to be understood. The net effect is that looking at http://localhost:8080/manhole/cq/qp/type you can find out what DaCHS things the type of arihip.main's hipno column is – and you can try description, ucd, or whatever in that last URI segment, too.

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 examples.

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

If you look up the indicated line, you will see some table markup. Now that you are warned, you will probably see immediately that there is a less-than sign there, and that is not allowed in XML parsed character data. While writing reStructuredText (or Python, for that matter), it is easy to forget that $<$ and & are magic to XML. CDATA is your friend for embedded formal languages.

Now for a particularly nasty syntax error that is 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 > dachs --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 are 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 if XML processing tools normalize whitespace in certain elements. This is a bit unfortunate since the way RDs are written, they are 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 is 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 > dachs imp q
Making data arihip/q#import
Starting /home/msdemlei/gavo/inputs/arihip/data/data.txt.gz
Done /home/msdemlei/gavo/inputs/arihip/data/data.txt.gz, read 1
*** Error:   Row
  alphaHMS -> '0  0  0.216002'
  ddeHIP -> '+     0.93'
  [...lots of lines...]
  t_ra_mod -> '91.45'
  vrad -> ''

Field kbin: While building kbin in <rowmaker without id>: name
'parsWithNull' is not defined
```

The many lines are a dump of the rawdict that is being processed; it is a bit bulky, but it frequently helps figuring out problems that only occur when there is an odd value in the incoming data.

The "while building kbin" tells you where in the rowmaker something went wrong: At the mapping of kbin. The `<rowmaker without id>` part would be more informative if we had given the rowmaker element an id. Doing that obviously is advisable when you have multiple rowmakers in one RD, as for instance:

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

That woud improve the message to:

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

– watch out for those. DaCHS does not consider it an error if a key is missing in the rowdict (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 > dachs imp -M 12 q
Making data arihip/q#import
Starting /home/msdemlei/gavo/inputs/arihip/data/data.txt.gz
Done /home/msdemlei/gavo/inputs/arihip/data/data.txt.gz, read 1
*** Error:   Row
alphaHMS -> '0  0  0.216002'
[...]
t_ra_mod -> '91.45'

Field vrad: While building vrad in <rowmaker without id>: Key 'vrad'
not found in a mapping.
```

Incidentally, the `--enable-pdb` main option to `dachs` sometimes is helpful in such situations, as it will dump you in the Python 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 rowmaker. This yields:

```
arihip > dachs imp q
[...]
Field procf11fed4c: While executing procf11fed4c in <rowmaker without
id>: name 'ddt' is not defined
```

The message coming from the bowels of python is clear enough (`'ddt' is not defined`), but the alphabet soup surrounding it is not. This name was invented by DaCHS since no ``name` (it's not id as the same name might be used in two different rowmakers) 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 `dachs`, 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: Wh
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.

Deleting Resources

Sometimes data collections become outdated, typically because new data has come in and an improved reduction even of existing data is made available (a "data release"). Our recommendation is to in most cases overwrite the previous data in order to avoid making people use known-bad data.

Not infrequently, however, one wants to keep old data releases around, perhaps for easy comparisons, perhaps because software that is still in use uses the outdated resources. As an aside, the commonly advocated reason for keeping outdated data around that some archived workflow still produce identical results is, I claim, a weak reason to keep around terabytes of data.

Be that as it may, at some point you should make stale data less discoverable, and perhaps at some point drop it altogether.

As a general policy, we recommend to take data from old data releases out of obscore and similar globally discoveable resources, with the argument that whoever whats old data at least should explitiely look for it. Keeping SSAP or SIAP services one or two releases back published may be a nice service at this point, but an attractive alternative that puts outdated data out of casual users' view is to drop the old services and keep the data available through datalink on the successors.

When dropping services and other published resources for one reason or other, you should give people a chance to find the updates.

From a registry point of view, this is not trivial, because deleted services (which may be kept in the registry forever) do not have metadata. You can, however, stretch the rules a bit and give, in the updated service, a *Continues* relationship to the old ivoid.

To set this up, look for all `publish` elements in the old RD, and look for their counterparts in the new RD. Then figure out the ivoids of the publihed resources in the old RD (the `/browse` trick introduced in Trying the services may help you do that), and in each counterpart add a *Continues* meta with the old ivoid in the `ivoId` attribute and the old title as a body, for instance:

```
<meta name="continues" ivoId="ivo://org.gavo.dc/gaia/q/dr1">
  Gaia Data Release 1 (DR1) gaia_source</meta>
```

Then delete all `publish` elements in the old resource and `dachs pub` both the old and the new RD.

After that, you can `dachs drop` the old RD; note that this will only drop data items automatically made. If you have DDs with `auto="False"` that are building tables (cf. `dachs show dds`), you will have to explicitly name them in the drop. Do *not* `dachs drop -f` such RDs; that will remove the RD from DaCHS' memory and prevent the creation of deleted records, which in turn will lead to zombie services hanging around in the Registry.

In case you're wondering why this is bending the rules: Well, you're not supposed to hand out ivoids of things that are not in the Registry, and the ivoid of the deleted service by definition isn't in the Registry any more. We will have to think about it a bit more in the VO.
Now that you've done the right thing I can tell you that regrettably, very few people will actually notice it, because few people look for ivoids in the Registry to figure out updates. Instead, they'll just point their user agents at your old URIs, and after you've dropped the underlying database tables, they will simply 404 on your users without further comment.
To fix this, give the old RD a `superseded` meta that ought to point to whereever the updated material is, for instance:

```
<meta name="superseded" format="rst">
 We do not publish Gaia DR1 data here any more.  If you actually
 need DR1 data, refer to the full Gaia mirrors, for instance
 ‘the one at ARI‘_.  Otherwise, please use more recent data releases,
 for instance ‘eDR3‘_.

 .. _the one at ARI: http://gaia.ari.uni-heidelberg.de
 .. eDR3: /browse/gaia/q3
</meta>
```

Whatever you write there will be shown alongside the 404s when people try to access the services, and it will be prominently shown in the resource info.
Regrettably, the message will not be shown when people follow links to tableinfos – for a dropped table, DaCHS does not remember where it might once have been defined, and thus it will not see the RD and just return an unspecific 404. Changing this would require quite a bit of bookkeeping which seems disproportionate for a relatively exotic case.
If you just supersede a single service in a larger RD, you can also attach the `superseded` meta to a single service (which means you would keep its `service` element around, but you can give it a `nullCore`).

Restricting Access

For one reason or another, you may need to restrict access to services or data products for a while.
To support this, DaCHS has a simple user and group management and lets you declare individual services or datasets as requiring authorisation.
Right now, this is rather basic and only provides what is known as "mild security", since it is based on HTTP basic authentication, which, unless you run HTTPS only, means that eavesdroppers can trivially find out passwords.
Given that we are not dealing with sensitive information and snooping attacks on our connections don't sound likely, we consider that marginally ok. Still, admonish your users to not use valuable passwords in DaCHS. Or restrict your site to HTTPS access.
Given enough client support, however, we will certainly add support for fancier authentication schemes (OAuth2?); the authorisation mechanisms discussed here should not be (materially) affected.
At this point, recent versions of TOPCAT, Aladin, pyVO, and Splat support HTTP basic authentication, so if you can live without federation and accept that the various tools

will request the credentials separately, your users can probably work with proprietary data reasonably well.

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 `dachs admin adduser`, like this:

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

```
dachs 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 restricted to happy.
To discover further commands manipulating the user table, try:

```
dachs admin --help
```

Important: When you use authentication, please set the `[web]realm` configuration item to some string reasonably characteristic for your site. DaCHS sends that realm with the authentication challenge, and even if these days, it's not terribly common to show it any more since it's been abused in various ways, it may still be shown to users.

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 challenge the 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 rowfilter, 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 will in general come up with an ISO datetime string or a Python `datetime.datetime` instance. Here is 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, they can access the data – see above for details on how to create users and groups.

Active Tags

Sometimes RDs need to contain material in several places, or there are tags that basically are copies of each other with minimal variation; we have already introduced the case in the DaCHS Basics as Metaprogramming: Macros and LOOPs.

In the spirit of DRY, it is desirable to avoid the repetition, if only because when the RD develops, you may edit some repeated item in one place but not the other, at which point you'll have a bug.

To avoid repetition (and some other minor things), DaCHS has active tags, which mark up 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.

In DaCHS, you can tell active tags from others by their all-uppercase names. Normal elements in DaCHS always have names starting with lower case letters.

STREAM and FEED

When you have a sequence of elements that you need in several places, you can put these elements into an Element STREAM and replay them with Element FEED in as many places as you want. You have seen several examples of that in this tutorial, for instance, in:

```
<FEED source="//scs#coreDescs"/>
```

To see what this plays back, look for an element with id `coreDescs` in the RD `scs`, which in turn you can most easily access with:

```
dachs adm dumpDF //scs | less
```

You will see that this element looks like this:

```
<STREAM id="coreDescs">
        <doc>[...]</doc>
        <condDesc original="//scs#humanInput"/>
        <condDesc original="//scs#protoInput"/>
</STREAM>
```

The `doc` element is local to the STREAM; make sure that you use it to drop a few words on what the stream is supposed to do – your future self will be grateful.
All other elements will end up exactly like typed there whereever there is a corresponding FEED.
Another example where a simple duplication of elements is useful is in fk6/q, where a catalogue has been split into two parts. Both parts share a lot of the columns, but there are some extra columns specific to each part. To deal with this, the RD defines a STREAM:

```
<STREAM id="commonFrontFields" doc="Field 1..23 are (about) the same
  for parts I and III.">

  <column name="hipno" type="integer" required="True"
    ucd="meta.id"
    tablehead="HIP#"
    description="Number of the star in the HIPPARCOS Catalogue (ESA 1997)"
    verbLevel="25"/>
  <column name="comname" type="text"
  ...
```

and then, further down, replays it into each table:

```
<table onDisk="True" id="part1" adql="True" primary="localid"
    mixin="//scs#q3cindex">
  <mixin>//scs#pgs-pos-index</mixin>
  <meta name="description">Part I of the FK6 (successor to Basic FK5)</meta>
  <nrows>878</nrows>
  <FEED source="commonFrontFields"/>
  ...

<table onDisk="True" id="part3" adql="True">
  ...
  <FEED source="commonFrontFields"/>
```

However, more commonly there will be slight variations between the replayed elements. Consider, for example, the license declaration we have seen above:

```
<FEED source="//procs#license-cc-0" what="ARIHIP"/>
```

Since CC-0 recommends that you explitictly state what you distribute, we have to put in something different every time we replay the stream. If you look at the stream (`adm dump` is your friend again), you will see how this works:

```
<STREAM id="license-cc0">
  <doc>[...]</doc>
  <meta name="rights" format="rst">\\RSTcc0{\what}</meta>
  <meta name="rights.rightsURI"
    >http://creativecommons.org/publicdomain/zero/1.0/</meta>
</STREAM>
```

The basis here is that all attributes to `FEED` are available for macro expansion. On the other hand, `FEED` does one level of macro expansion, which explains the double backslash in front of `RSTCcc0`: after the first macro expansion, this becomes a single backslash. Hence with `what="ARIHIP"` as above, the `FEED`'s net effect is the same as if someone had typed:

```
<meta name="rights" format="rst">\RSTcc0{ARIHIP}</meta>
<meta name="rights.rightsURI"
  >http://creativecommons.org/publicdomain/zero/1.0/</meta>
```

Another example for a parameterised `STREAM` is in sp_ace/q, where multiple output fields of a code have possibly asymmetric confidence intervals:

```
<STREAM id="valWithConfidence">
  <doc>
    A param with a confidence interval (give name, ucd, unit, tablehead
    and description and get three params including asymmetric error
    bars.
  </doc>
  <param name="\name"
    ucd="\ucd" unit="\unit"
    tablehead="\tablehead"
    description="\description of the best-fit modelled spectrum."/>
  <param name="\name\+_low"
    ucd="stat.error;stat.min;\ucd" unit="\unit"
    tablehead="Lower \tablehead"
    description="Lower limit of 68% confidence interval of \description"/>
  <param name="\name\+_up"
    ucd="stat.error;stat.max;\ucd" unit="\unit"
    tablehead="Upper \tablehead"
    description="Upper limit of 68% confidence interval of \description"/>
</STREAM>
```

This is used later on like this:

```
<FEED source="valWithConfidence"
  ucd="phys.temperature.effective" unit="K"
  name="teff" tablehead="T_eff"
  description="Effective Temperature"/>
<FEED source="valWithConfidence"
  ucd="phys.gravity" unit="cm/s**2"
  name="logg" tablehead="Log g"
  description="Log of gravity"/>
```

LOOP

Element LOOP takes the idea of having multiple FEED elements further and makes it easy to generate lots of fillers, possibly even programmatically.

The simplest way to use LOOP 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 is something like an immediate STREAM; you could use a stream instead and reference it in a source attribute, but that's rarely easier to read.

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 LOOP's csvItems attribute. The content of this 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>
```

You can take this even further and generate the replacements with Python code. For instance, in califa/q3, I needed to pull dozens of columns through a `parseWithNull` in a row maker. To maintain the list of such columns was already a challenge, and so I had the computer do it by pulling the table out of the parse context (which is a complex object you usually don't see; `codeItems` runs while the RD is being parsed, and so there are additional limits to what you can see) and retrieving all "flag" columns. Those are the ones that need a uniform parsing:

```
<LOOP>
  <codeItems>
    for col in context.getById("cubes"):
      if col.name.startswith("flag_"):
        yield {"item": col.name}
  </codeItems>
  <events>
    <map dest="\item">parseWithNull(@\item, int, -1)</map>
  </events>
</LOOP>
```

Programmatically Filling RD Elements

The LOOP active tag just introduced can be (ab-) used when you need to move around element content in ways not supported by DaCHS otherwise. The basic idea is to use `codeItems` to fetch information and hand it over through macros (which are defined by yielding dictionaries from the code items as details above). This section collects a few examples of how this technique can be applied.

Copying Coverage

Say you have a resource A that really is just some sort of "value-added" service on top of another resource B. It stands to reason that A should declare the same coverage as B. If the coverage in B were given literal, you could say:

```
<coverage original="B#cov"/>
```

in A if you had furnished B's coverage element with the id `cov`. But usually, the coverage will have an updater rather than actual, copyable elements. The updater will not work in A, and even if it were, computing the coverage usually is expensive, and it's wise to avoid computing it twice. Fortunately, using `LOOP`, you can get B's computed coverage into A.

The first thing you need to figure out is that the computed coverage is kept in the dc.rdmeta table (it's per-resource metadata, after all). Inspecting this table, you see that spatial is just text (a MOC, ready for inclusion into an RD), whereas temporal and spectral are arrays of double, which you will have to serialise before putting them into the RD.

In DaCHS' current implementation these are always 2-arrays, which makes this a good deal simpler. This way, I only need to produce one `temporal` and `spectral` child each in the Element coverage. If there were multiple intervals, I would need separate loops for those (and that may happen in the future). As long as that is not the case, you can re-use a coverage like this:

```
<LOOP>
<codeItems>
  with base.getTableConn() as conn:
    for row in conn.query("SELECT spatial, temporal, spectral"
      " FROM dc.rdmeta"
      " WHERE sourcerd='__system__/obscore'"):
      yield {
        "spatial": row[0],
        "temporal": " ".join(str(v) for v in row[1]),
        "spectral": " ".join(str(v) for v in row[2])}
</codeItems>
<events>
  <coverage>
    <spatial>\spatial</spatial>
    <temporal>\temporal</temporal>
    <spectral>\spectral</spectral>
  </coverage>
</events>
</LOOP>
```

This pattern assumes you are sure coverage information is available on all axes. In case you want to be robust against NULLs, you could add clauses like `"" if row[1] is None else` or similar.

Note how this naturally becomes a no-op if there is no obscore coverage: The code items will not yield anything then, and no coverage element will be produced in A. In case there are multiple result rows – this is impossible in this case as sourcerd is a primary key on rdmeta –, the last row would win due to DaCHS' overwriting rules; this is about as good as any other choice (except *perhaps* raising an error).

Creating Placeholders

When you build condDescs from multiple database columns as in Example: Form-based obscore, DaCHS has no idea of the ranges on the resulting parameter(s). It's still not nice to have no placeholders in Web forms (say). You can, however, compute them based on the table statistics. DaCHS manages these in the system table dc.simple_col_stats. When building a parameter out of, say, `t_min` and `t_max`, you could find the limits using a query like:

```
SELECT column_name, min_value, max_value
FROM dc.simple_col_stats
WHERE tablename='ivoa.ObsCore'
  AND column_name in ('t_min', 't_max')
```

However, making a useful placeholder takes some extra manipulation. If, as is likely in such a case, the input is through VizieR-like expressions, you would need to format the MJDs that we get here to ISO time stamps. Also note that the values in simple_col_stats are not floats but VOTable literals in strings (this is so it can also represent, say, datetimes); you hence need to parse values you pull from that table.
To set the actual placeholder, you use a property. The total input key would then be:

```
<inputKey name="TIME" type="vexpr-date"
  ucd="time.epoch"
  tablehead="Obs. time"
  description="Time covered by the observation.">
  <LOOP>
    <codeItems>
      from gavo import base, stc
      with base.getTableConn() as conn:
        limits = dict((r[0], r[1:]) for r in
          conn.query("SELECT column_name, min_value, max_value"
            " FROM dc.simple_col_stats"
            " WHERE tablename='ivoa.ObsCore'"
            "   AND column_name in ('t_min', 't_max')"))

        if "t_min" in limits:
          yield {"obscore_tmin":
              stc.mjdToDateTime(float(limits["t_min"][0])
                ).date().isoformat(),
            "obscore_tmax":
              stc.mjdToDateTime(float(limits["t_max"][1])
                ).date().isoformat()}
    </codeItems>
    <events>
      <property name="placeholder"
        >\obscore_tmin ..  \obscore_tmax</property>
    </events>
  </LOOP>
</inputKey>
```

Contrary to our loop pattern in Copying Coverage, here we need to explicitly handle the case of missing statistics. This is why we are checking for the presence of `t_min` in the results from the database.

Licensing

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 transmission and copying. Still, formal statements about how your data may be used are required if, for instance, the data or parts thereof are re-distributed with software, or are aggregated and re-published. Such statements are called licences, and to facilitate re-use you should choose and declare a licence.
In DaCHS, you can do that manually using the following two meta items:

- rights – this is free text transmitted to the registry that should state the conditions under which the data can be used, shared, etc. This can include requests for acknowledgement, but don't be overly verbose here; if, for instance, you'd like to acknowledge all your contributors, rather put that into a `_longdoc` meta and reference that from rights.

- rightsURI – a URI of a common licence; this is so machines can figure out conditions if necessary. For now, only include it if your licence actually gives such a URI.

In practice, you should stick to a well-known licence and probably don't bother with these two items by hand. DaCHS comes with support for CC-0 (essentially, no restrictions; that's not precisely a licence, but it stands in well for one), CC-BY (state where you got the data from if re-publishing), and CC-BY-SA (as for CC-BY, but also require anyone who re-uses the data to share what they got from you at least as liberally as you shared it). Let us know if you want other licences supported, and we'll do something about it.

To apply them, replay one of the streams

- //procs#license-cc0
- //procs#license-cc-by
- //procs#license-cc-by-sa

into the RD and give a `what` attribute saying what is licensed (usually, the data). And yes, we know that CC-0 isn't a licence; still, this arrangement seemed logical, and it stands in for a licence, anyway. For instance:

```
<FEED source="//procs#license-cc-by" what="ARIHIP"/>
```

If you get to choose the licence but are not sure which one to pick: Our advice is to go for CC-0. This is although we do believe in the GPL for software, which has strong traits of CC-BY-SA in that it prevents others from making your code proprietary. But data is not code. While it is probably unreasonable to combine more than a dozen pieces of source code from different parties into a single reasonable piece of software (excepting distributions, but these have special rules, and modern web browsers, which are not reasonable), it is totally reasonable to include data from thousands of sources. Even something as permissive as CC-BY then becomes a liability – if you published the result of such an effort, in your `rights` meta you would have to reproduce a thousand attributions, and that is absolutely not worth it.

In case your data providers are nervous about putting their work into, in effect, the public domain: Licensing has **nothing** whatsoever to do with requiring people to properly cite and/or acknowledge. That is part of good scientific practice and is unrelated to legal issues. Licensing is necessary for hassle-free re-use of the data in situations where it needs to be re-distributed; the classic example would be a star catalogue within a planetarium software.

Sometimes you want an additional acknowleding clause. In that case, just add another rights meta item:

```
<meta name="rights" format="rst">If you use this data, please
  acknowledge: "This work made use of the HDAP which was produced at
  Landessternwarte Heidelberg-Königstuhl under grant No. 00.071.2005
  of the Klaus-Tschira-Foundation."

  The data in this service was calibrated using Astrometry.net,
  :bibcode:`2010AJ....139.1782L`.

</meta>
<FEED source="//procs#license-cc0" what="the HDAP scans"/>
```

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):

- `name="x_mjd" type="double precision" 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).
And yes, historically DaCHS did use the column name to tell MJDs from JDs. We used to use the xtype for that (and that still works), but that is incompatible with the SODA/SIAv2 use of xtype. As of DaCHS version 2.8, you should add a full time annotation for MJD times (cf. New-style SIL definitions). This could look like this:

```
<dm>
  (votable:Coords) {
    time: {
      frame: {
        timescale: UTC
        refPosition: TOPOCENTER
        time0: MJD-origin
      }
      location: @my_time
    }
</dm>
<column name="my_time"
  unit="d"
  .../>
```

Admittedly, that is a bit verbose, and perhaps we will experiment with shortcuts later.

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 did not 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 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 Greenwich meridian (i.e., they should be close to UTC).

The time scales are important on the level of seconds; time scales known to the VO are listed on http://www.g-vo.org/rdf/timescale. To understand a bit better where all this comes from, we recommend a look at the fairly readable explanation at http://stjarnhimlen.se/comp/time.html.

The reference positions impact arrival times by simple geometry; common reference positions include "roughly earth" and "roughly sun", which leads to differences on the order of ten minutes. Additionally, there are relativistic effects at 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.

Obviously, any number of reference positions might be necessary at some point ("a clock on the Voyager II space probe"). In the VO, we try to stay on top of this by collecting "well-known" reference positions on http://www.g-vo.org/rdf/refposition. This is list will grow over time, but it is unlikely that many clients will be able to do much with anything except GEOCENTER, EMBARYCENTER, BARYCENTER, and HELIOCENTER. That does not mean that it is not a good idea to declare something else. Good clients might at least adjust estimates of systematic (or even random) errors based on such metadata.

To declare them, use the procedure described in New-style SIL definitions for now.

Apologies that this is so obscure, but for once that's really not our fault.

[1] Another reason this may fail is if you have something else listening on port 8080 already, in with case you can create a file `.gavorc` in your home directory and add:

```
[web]
port: 4040
```

there. Of course, you have to adjust the URLs given in the tutorial then.

[2] This assumes you are doing this on your local machine (which we 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.

[3] If you are running DaCHS on a reasonably stable platform, you are most welcome to join it. You would have to check out the resdir at http://svn.ari.uni-heidelberg.de/svn/gavo/hdinputs/rr/, follow the instructions in the README (after which you have a fully functionaly RegTAP registry). To be part of the machines that provide fallbacks in case of failures, please contact the Heidelberg maintainers. Incidentally, you can also run a mirror of the web interface to the registry then; you would just have to check out http://svn.ari.uni-heidelberg.de/svn/gavo/hdinputs/wirr/.

[4] `dachs limits` on DaCHS 2.3 and earlier does something somwhat different, and you should ignore everything we say here if you're still on DaCHS 2.3. Additionally in the DaCHS version that is in Debian bullseye (which is also a 2.3) the limits subcommand is broken. Bottom line: If you want dachs limits, use 2.4 or newer.

[5] DaCHS has had support for version 0.37, which is in the //epntap RD. Do not use that for new projects.