



International

Virtual

Observatory

Alliance

IVOA Data Access Layer Service Architecture and Standard Profile

Version 1.0

IVOA Note 2010-05-21

This version:

<http://www.ivoa.net/Documents/Notes/DAL2Arch/NOTE-DAL2Arch-1.0.pdf>

Latest version:

<http://www.ivoa.net/Documents/latest/latest-version-name>

Previous version(s):

none

Authors:

D. Tody (ed.), F. Bonnarel, M. Louys, J. Salgado

Abstract

The IVOA Data Access Layer (DAL) protocols define a family of service interfaces providing access to all astronomical data available via the VO. These interfaces are structured as a class hierarchy, rooted at the *generic dataset*, with a subclass for each type of astronomical data, i.e., *Table*, *Image*, *Spectrum*, etc. Much of the service functionality, interface, and metadata is common to all classes of data and hence to all data access interfaces. A major goal of the

second generation DAL interfaces (DAL2) design was to define this architecture including the basic service profile and common metadata to permit a uniform set of second generation interfaces to be defined, unlike the first generation prototypes where the interfaces evolved significantly with each new interface. In this document we examine the overall architecture for the DAL interfaces, and note what is common to all interfaces as well as where they differ.

Status of This Document

This is an IVOA Note. The first release of this document was 2008 October 5.

This is an IVOA Note expressing suggestions from and opinions of the authors. It is intended to share best practices, possible approaches, or other perspectives on interoperability with the Virtual Observatory. It should not be referenced or

A list of [current IVOA Recommendations and other technical documents](http://www.ivoa.net/Documents/) can be found at <http://www.ivoa.net/Documents/>.

Acknowledgements

“Ack here, if any”

Contents

1	Introduction	3
2	Data Service Architecture	4
2.1	Class Hierarchy	4
2.2	Generic and Typed Interfaces	6
2.3	Data Discovery vs. Data Access	6
2.4	Data Models	7
2.4.1	Dataset Metadata and Data Queries	8
2.4.2	Data Representation and UTYPE	9
2.5	Virtual Data	9
3	Common Service Elements	10
3.1	Form of Interface	11
3.1.1	Request Format	11
3.1.2	Parameters	11
3.1.3	Parameter Values	12
3.1.4	Error Response	12
3.1.5	WMS Comparison	13
3.2	Query Input	13
3.3	Query Response	15

Data Access Layer Architecture

3.4	Standard Service Profile	16
3.5	Grid Capabilities	17
	References	18

1 Introduction

The DAL architecture and standard service profile described here (also known as the second generation DAL service architecture, or DAL2) has been under development since at least May 2003 when the first IVOA interop was held in Cambridge, UK. It was largely complete by the May 2006 interop in Victoria, with the SSA interface (the first of the second generation DAL interfaces) completed by the time of the Beijing interop in May 2007. In the fall of 2007 SSA became an IVOA Recommendation, and by this time has been widely implemented. The SIAV2 and TAP protocols currently under development, along with SSA and its related SED and time series interfaces and the planned generic dataset interface, will provide the core of the second generation IVOA DAL protocols.

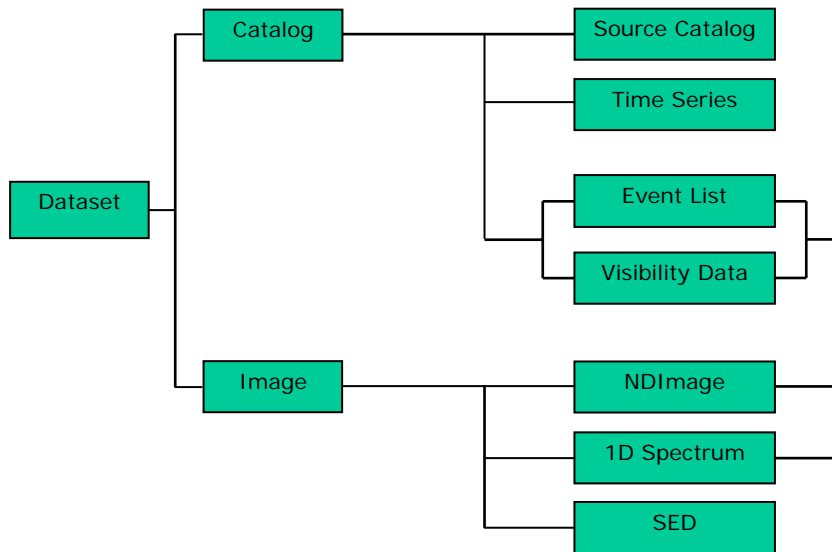


Figure 1. Historical DAL Service Architecture (Cambridge UK, May 2003)

The DAL2 protocols form a family of data access interfaces, with a great deal which is common to all the interfaces. This becomes evident if we examine the class hierarchy of related classes of data and their associated interfaces, with the inheritance of common query parameters, metadata, service methods, and service interface which this class structure implies.

Exploiting this commonality of the DAL interfaces is important to provide uniformity; users expect to see similar functionality implemented consistently in a

Data Access Layer Architecture

family of closely related interfaces. It is also important to promote code sharing, both on the server side when using a service framework to implement the entire suite of interfaces, and on the client side, where the service interface is much the same for all classes of data which can therefore share a common implementation, with only the data-specific details differing from one class of data to the next.

Our intention here is only to provide a brief overview of the DAL service architecture. Further information can be found in the SSA specification [1], and other documents; in particular, most of the SSA query parameters and query response metadata belong to the generic dataset and are not specific to SSA. Likewise section 8 of the SSA specification, titled *Basic Service Elements*, describes the standard service profile for a second generation DAL service and is not specific to spectral access. Ultimately this may want to be moved out to a separate document (possibly the generic dataset specification) since it will be common to all the DAL2 services, but since SSA was the first DAL2 interface it was simpler to just include it directly in the specification.

The recent document *Scope and Concepts of SIAV2* [2] discusses the concept of the generic dataset and the relationship of this to the typed interfaces such as SSA and SIA. The basic service profile adopted for DAL2 is based upon the OpenGIS specification for the *Web Mapping Service* (WMS) [3] which is the basis for applications such as NASA Worldwind and which helped motivate later applications such as Google Earth. The second generation DAL protocols are based in part on the original VO data access protocols, Simple Cone Search [4], and Simple Image Access V1.0 [5].

2 Data Service Architecture

2.1 Class Hierarchy

DAL has always had the concept of a unified service architecture and family of data access protocols, although it has evolved quite a bit over the years, and will probably continue to evolve. The current DAL class hierarchy is shown in Figure 2 below. This diagram is not intended to be complete. There may be additional classes at each level, but the important thing is the overall structure and the relationships between levels.

The *generic dataset* is at the root. Each of the top level “typed” data classes is a subclass of the generic dataset, representing the major classes of astronomical data. These can be subclassed further to create more specialized classes of data, e.g., a source catalog is a subclass of table with the addition of a source catalog data model, and a class of spectra created from a theoretical model (as in TSAP) is a special class of Spectrum.

Data Access Layer Architecture

At the lowest level we have instrumental data collections which may be subclassed from any of the higher level classes, inheriting the basic service interface and data model while adding metadata and functionality specific to the data collection. In this example a client application may view the data as a generic Dataset, as a top level typed object (Image, Spectrum, etc.), or as an instrumental dataset.

A given physical data collection may be interfaced in more than one way simultaneously, e.g., an event list could be viewed as both a table and as an image, or a spectral data cube could be viewed as both an image cube and as a source for extracted spectra. More complex datasets could be modeled as logical associations of simple data objects.

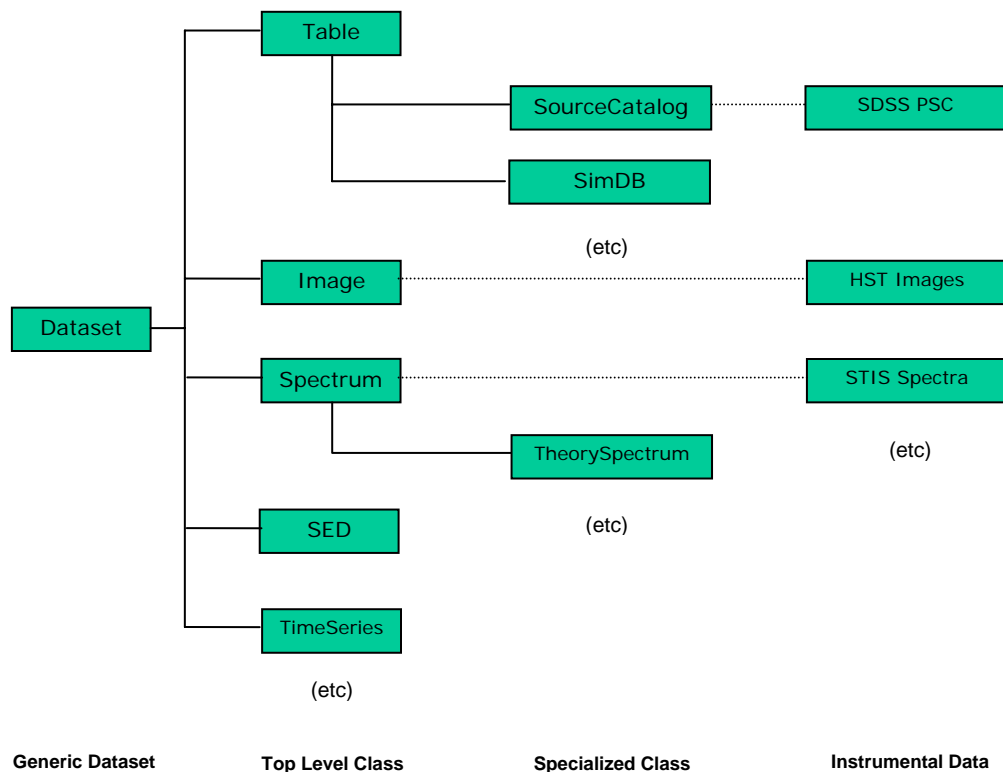


Figure 2. Current DAL class hierarchy of data and their associated services.

Some classes of data are not shown, in part because we are still considering where they fit into this structure. Spectral line data (SLAP), while not shown, is probably a top level class of data and interface. Solar and planetary data might require a top level interface, or might be subclassed from top level classes such as Image and Spectrum if these are generic enough to apply to solar and planetary data. Data from general theoretical models might be represented either via a table-oriented SimDB, or via a data access oriented SNAP interface.

2.2 Generic and Typed Interfaces

The main difference between the generic dataset and the typed interfaces (*Table*, *Image*, *Spectrum*, etc.) is that the generic dataset can describe any type of astronomical data, whereas the typed interfaces can describe only a single type of data, but can do so in greater detail, with a more refined data model specific to the data. A second key difference is that the generic dataset can describe only entire *archival* datasets (entire data objects or files as stored in some archive), whereas the typed interfaces can describe, generate, and provide access to both static archival datasets as well as *virtual data* (datasets which may not actually exist when described, but which can be computed on demand if accessed). Finally, since the generic dataset can describe any type of data it can also describe relationships between different types of data, for example to associate multiple discrete data products as elements of *complex data* of some sort.

This approach provides a great deal of flexibility for both describing and accessing data. A complex observation consisting of several related data products can be described via the generic dataset query mechanism. For example we might have a survey field consisting of a spectral data cube, some 2-D projections of the cube, a source catalog for the field computed from the 2-D continuum, and possibly some extracted spectra of objects in the field. Client applications which do not understand the complex data association could still be used to access and analyze the individual primary datasets. Data objects such as extracted spectra or projections could be either precomputed or computed on demand as virtual data. If the client application is sufficiently knowledgeable of the data it could use the instrumental or collection-specific metadata provided to perform a more detailed analysis of the data.

2.3 Data Discovery vs. Data Access

The DAL protocols provide capabilities for both data discovery and data access, generally within the same interface. While one might imagine separating discovery from access in the case of access to static archival data, this would be difficult or impossible for access to virtual data. “Discovery” in the case of virtual data includes negotiation with the service to determine the details of what the service can actually deliver, given the parameters of the ideal dataset as requested by the client.

The data query present in the typed interfaces combines both pure data discovery and negotiation of the details of virtual data into a single query interface. In many simple cases a single query is sufficient to discover data of interest. In more complex cases the process is iterative, e.g.:

- Perform an initial discovery query to find data of interest.
- Select a dataset to be accessed.
- Use the dataset metadata to refine the query, specifying in detail the virtual dataset to be retrieved.

Data Access Layer Architecture

- Download (and hence compute) the final fully specified dataset.

This process can be repeated starting from the second step for any number of individual datasets, or to access different portions or views of a single dataset.

The planned generic dataset service will provide an additional dimension to this process, allowing discovery of more than one type of data in a single query as well as describing any complex data associations. A generic dataset query would find all static archival datasets of any type matching the search criteria. Access to specific datasets would then be performed using the appropriate typed interface, iterating in the usual fashion upon the parameters of the virtual data to be generated.

2.4 Data Models

Data models affect DAL primarily in two areas: the query response describing available data, and the dataset to be accessed, which implements a data model of some form which will affect access to and analysis of the data.

For all DAL interfaces with the exception of TAP (which has only a very limited data discovery capability) the query data operation implements a data model based upon the generic dataset metadata. In the case of SSA for example this includes the following types of metadata:

Service Metadata	
Query	Describes the query itself
Association	Logical associations
Access	Dataset access-related metadata
Data Model Metadata	
Dataset	General dataset metadata
DataID	Dataset identification (creation)
Curation	Publisher metadata
Target	Observed target, if any
Derived	Derived quantities
CoordSys	Coordinate system frames
Char	Dataset characterization
Characterization Metadata	
Char.FluxAxis	Observable, normally a flux measurement
Char.SpectralAxis	Spectral measurement axis, e.g., wavelength
Char.TimeAxis	Temporal measurement axis
Char.SpatialAxis	Spatial measurement axis
Char.*.Coverage	Coverage in any axis
Char.*.Accuracy	Resolution and error in any axis

Essentially all of this is generic dataset metadata and would be present for any of the DAL2 interfaces. Dataset identification, curation, target information, and dataset characterization for example are applicable to any type of data. The only class of metadata specific to spectra is the “Dataset” metadata, which provides a place to put metadata specific to the class of Dataset being described (Spectrum in the case of SSA).

2.4.1 Dataset Metadata and Data Queries

Dataset metadata can be tricky to use for queries as much of the standard dataset metadata is optional and will often not be provided, and what is provided will vary from one service to the next (although some basic metadata is mandatory). One of the reasons we have parameter-based queries in DAL is because this provides a mechanism to permit data discovery even in the presence of missing metadata. An ADQL query for example, being a formal expression, would be prone to failure when posed against dataset metadata due to missing metadata, especially when posing the same query against multiple services. The parameter-based query avoids this by identifying the most important bits of metadata needed for queries, and by defining query semantics which permit queries to be well-formed and computable even in the presence of missing metadata.

For the planned generic dataset service the goal is to provide both parameter-based and ADQL-based discovery queries. While the missing metadata problem will affect generic dataset queries much as for typed dataset queries, it is more tractable as there is no virtual data to deal with (the query is more like a traditional database query), and the effort to support ADQL-based queries is more easily justified since the whole point of the generic dataset query will be data discovery and description.

As mentioned earlier, TAP (table access) is different as it is more of a pure data access interface with only a limited data discovery capability (this is an optional capability using POS, SIZE or REGION in a table metadata query). TAP queries still have a data model; it is just that the model (the table schema) belongs to the data table being queried rather than being defined by the TAP interface (future UTYPE-based catalog queries being the exception to this rule). Although the data model is different, a TAP query is really little different than any other DAL data query such as a SSA or SIA query. Data operation: parameters are input, query response data model elements are set for each output table row, and an output table is returned in any of the supported output formats. Execution can be either synchronous or asynchronous. The form of the service interface can be the same in both cases, with the only significant difference being the data model or schema assumed by the query.

2.4.2 Data Representation and UTYPE

As we saw in section 2.4, the query response data model for a DAL protocol is structured as a number of reusable *component data models*: DataID, Curation, Target, Characterization, and so forth. These little models are intended to be components which can be reused in all the DAL interfaces. These standard component models are *aggregated*, possibly combined with other data specific or custom component models, to describe some particular type of data.

In order to support a variety of output data formats as well as flexibility of representation within client software, data models in DAL are defined abstractly, independent of representation. The UTYPE construct allows the elements of a data model to be tagged with unique, fixed identifiers. This makes it possible to reduce a hierarchical data model (up to some level of complexity) to a simple set of keyword-value pairs. This greatly simplifies client software by allowing standard containers of many different types to be used to store and manipulate the “content” of a data model, while allowing the exact same content to be represented in many different forms. For example, a data model composed in this way may be represented as a VOTable, as a DBMS table, as a FITS header, as a parameter set, as a hash table or map, and so forth.

Metadata extension is an important capability of data models as represented in the DAL interfaces. Metadata extension is what allows a standard model to be subclassed to represent a more specialized type of data, or to represent instrumental data where instrument-specific metadata is to be passed through. Since data model components are aggregated within a container, the standard model components are unaffected if additional custom components are included in the container.

The DAL query response mechanism also defines a metadata extension mechanism which allows arbitrary data to be *associated* with the main query response table. For example, if we have a VOTable container, the main query response will be a table, using the REF-ID mechanism to associate query response records with extension metadata stored in additional RESOURCE elements. This mechanism allows more complex objects to be included in the query response than can easily fit into custom components within the main query response table. This is done without complicating the main query response table, which remains a simple flat table. Client applications which don't need to access the extension metadata are unaffected if such optional metadata is included.

2.5 Virtual Data

Most DAL services decompose into two major elements: the data query and actual data access. Data access may be as simple as providing pass-through access to entire archival datasets in some native collection-specific format; this mode is comparable to most current archive data interfaces, and is fully supported by the DAL service architecture. Limiting access to pass-through of

Data Access Layer Architecture

entire native-format datasets is very restrictive though, as the datasets can be very large even though only a small portion of the data may be required, and it is hard for client applications to accommodate the great variety of native data formats present in the VO. *Virtual data* solves this problem, by providing optional data subsetting, filtering, and transformation on the fly at access time. The capability to dynamically provide virtual data tailored to what the client application requires for analysis is extremely important to enabling efficient, sophisticated astronomical data analysis via the VO.

Full support for virtual data is inherent in the DAL interfaces due to the separation of the query from actual data access, and the iterative query-response nature of the data query. Describing a virtual data product is very much the same thing as describing a static archival data product, allowing the same query interface to be used for both, simplifying the interface overall and abstracting away the complexities of virtual data generation.

Computation of virtual data products can be either trivial or quite complex and computationally expensive depending upon the data characteristics, the service capabilities, and the client request. The latter case however, is a good example of “moving the computation to the data”, and provides a powerful while still well-defined mechanism for distributing computation and maximizing data bandwidth within the VO. In the most challenging cases, computation of a virtual data product may require a significant astronomical algorithmic data processing capability as well as scalability, asynchronous execution, authentication, and data staging to network data storage. All of the latter are addressed by related VO technologies while the former is the domain of traditional astronomical data processing.

3 Common Service Elements

As noted earlier the DAL interfaces form a closely related family of interfaces. While each interface is tailored for the type of data to be accessed, there is much that is common to all the interfaces (the biggest exception is virtual data generation which can be very specific to the type of data being manipulated). A major goal for the second generation DAL (DAL2) interfaces is to provide a uniform look, feel, and function to all the interfaces; this is important for consistency and rigor as well as to promote code sharing at all levels, from service to client application. This is feasible now that the necessary VO technology has largely been developed and is becoming mature.

In this section we summarize those things which are largely common to all the DAL2 interfaces, and which we have tried to standardize beginning with SSA.

3.1 Form of Interface

The basic interface for a second generation DAL service is specified in detail in section 8 (*Basic Service Elements*) of the SSA specification [1]. In this section we merely summarize the basic elements of a DAL2 service interface. Much of what is presented here is adapted from the OpenGIS WMS standard [3].

3.1.1 Request Format

In general a service may implement multiple **operations**, such as `queryData`; altogether these define the **interface** to the service. Interfaces may change with time hence are versioned. It is possible for a given service instance to simultaneously expose multiple interfaces or versions of interfaces.

The SSA interface described in this document is based on a distributed computing platform (DCP) comprising Internet hosts that support the Hypertext Transfer Protocol (HTTP). Thus, the online representation of each operation supported by a service is composed as a HTTP Uniform Resource Locator (URL). Service functionality is defined independently of the DCP and the same service functionality could be implemented in the future for other distributed computing platforms.

A request URL is formed by concatenating a **baseURL** with zero or more operation-defined **request parameters**. The baseURL defines the network address to which request messages are to be sent for a particular operation of a particular service instance on a particular server. Service operations generally share the same baseURL but this is not required.

3.1.2 Parameters

A given service operation may define zero or more parameters which are used to control the function of the operation. In general parameters are specific to each operation, although it is a good practice when a parameter of the same name appears in multiple operations that the function be the same.

Parameters may appear in any order. If the same parameter appears multiple times in a request the operation is *undefined* (if alternate values for a parameter are desired the *range-list* syntax may be used instead). Parameter *names* are case-insensitive. Parameter *values* are case-sensitive unless defined otherwise in the description of an individual parameter.

All operations define the following standard parameters (these are part of the service specification, not the individual operations):

- REQUEST The request or operation name (mandatory).
- VERSION The version number of the interface (optional).

The values of both the REQUEST and VERSION parameters are case-insensitive. Use of REQUEST is mandatory even if a service interface defines only a single operation.

Example:

```
<baseUrl>?REQUEST=queryData&POS=22.438,-17.2&SIZE=0.02
```

A given service instance may support multiple versions of the service interface, which includes all service operations, their input parameters, the query response with all of its complex metadata, and the service capabilities. By default if the interface version is not specified the service assumes the highest *standard* version which is implemented by the service (access to any experimental versions supported by a service requires explicit specification of the version by the client). Explicit specification of the interface version assumed by the client is desirable to ensure against a runtime version mismatch, e.g., if the client caches the service endpoint but a newer version of the service is subsequently deployed. If desired the client can omit the `VERSION` parameter to disable runtime version checking, and default to the highest version standard interface implemented by the service.

All other request parameters are defined separately for each operation.

3.1.3 Parameter Values

Integer numbers are represented as defined in the specification of integers in XML Schema Datatypes. Real numbers are represented as specified for double precision numbers in XML Schema Datatypes. Sexagesimal formatting is not permitted, either for parameter input or in output metadata, other than in ISO 8601 formatted time strings (sexagesimal format is fine for a user interface but inappropriate for a lower level machine interface, where it only complicates things).

The DAL interfaces define a special *range-list* format for specifying numerical ranges, simple lists of values, or lists of ranges as parameter values. For example, “1E-7/3E-6;source” could specify a spectral bandpass defined in the rest frame of the source. Time ranges are permissible using ISO8601 syntax (this is where the use of “/” as the range delimiter comes from). String values may be included in lists but the range syntax is not supported for string values. The range syntax supports both open and closed ranges. Ranges or range lists are permitted only when indicated in the definition of an individual parameter.

3.1.4 Error Response

In the case of an error, service operations should return a VOTable containing an `INFO` element with name `QUERY_STATUS` and the value set to “ERROR”. Explanatory text may be included as the value of the `INFO` to describe the error which occurred. More fundamental service or protocol errors may however result in an HTTP level error; hence a client program should be prepared to handle either response. A null query, that is a data query which does not find any data, is not considered an error.

3.1.5 WMS Comparison

As a comparison to the request formatting specified herein, the following example illustrates a typical OpenGIS WMS request, used to get a GIS image (map), with the requested graphics overlays (interestingly, WMS also defines a *getCapabilities* operation returning service metadata in XML, much like that used in DAL and VOSI):

Example:
`<baseURL>?REQUEST=GetMap&VERSION=1.3.0
 &CRS=CRS:84&BBOX=-97.105,24.913,78.794,36.358
 &WIDTH=560&HEIGHT=350&LAYERS=AVHRR-0927&FORMAT=image/png`

Aside from adopting a proven HTTP interface style, the similarity to OpenGIS standards such as WMS might prove useful someday, if for example we should want to adopt (probably with some modifications or extensions) OpenGIS standards such as WMS for VO.

3.2 Query Input

The parameters defined by a service operation are specific to the operation and in principle can be anything. In practice services and service operations are often similar in many respects, and may have similar parameters. When common patterns and functionality are identified, input parameters should be consistent between services or sometimes service operations.

The query parameters summarized below are either generic (inherited from the generic dataset and common to all data queries based upon the generic dataset metadata), or specific to the basic table-oriented query mechanism common to all DAL queries. These are taken from the SSA specification, but are applicable to most or all classes of data. We only summarize these parameters here; no attempt is made to fully define each parameter.

The following are basic to the DAL2 discovery query:

<i>Parameter</i>	<i>Sample value</i>	<i>Physical unit</i>	<i>Datatype</i>
POS	52,-27.8	degrees; defaults to ICRS	string
SIZE	0.05	degrees	double
BAND	2.7E-7/0.13	meters	string
TIME	1998-05-21/1999	ISO 8601 UTC	string
FORMAT	votable	-	string

The following include more advanced query parameters based upon the generic dataset metadata, as well as parameters specific to the query mechanism (e.g., MAXREC, RUNID):

Parameter	Sample value	Unit	Req	Datatype
-----------	--------------	------	-----	----------

Data Access Layer Architecture

SPECRP	2000	$\lambda/d\lambda$	REC	double
SPATRES	0.05	degrees	REC	double
TIMERES	31536000 (=1yr)	seconds	OPT	double
TARGETNAME	mars		OPT	string
TARGETCLASS	star		OPT	string
ASTCALIB	absolute		OPT	string
WAVECALIB	absolute		OPT	string
FLUXCALIB	relative		OPT	string
PUBDID	ADS/col#R5983		REC	string
CREATORID	ivo://auth/col#R1234		REC	string
COLLECTION	SDSS-DR5		REC	string
MAXREC	5000		REC	string
MTIME	2005-01-01/2006-01-01	ISO 8601	REC	string
COMPRESS	true		REC	boolean
RUNID			REC	string

The SSA interface defines additional parameters (e.g., SNR, REDSHIFT) which we do not include here as they are not applicable to all types of data, although they would be appropriate for any other single object aperture-based observation, such as a SED or time series.

PUBDID and CREATORID allow the use of dataset identifiers to specify a particular dataset to be accessed. COLLECTION allows a single data collection to be queried or accessed.

MAXREC is part of the mechanism used to return table output data synchronously to the client. By default a service will return at most a certain number of table records, as defined in the service capability metadata. If overflow occurs this is indicated in the table output returned to the client. If desired the client can reissue the query with a large value of MAXREC (up to the maximum defined by the service capabilities), to attempt to stream a large query response back to the client. While not fully general, this simple mechanism is sufficient for many queries, and Grid capabilities (async, VOSpace) can be used if simply increasing MAXREC is not sufficient.

MTIME is used to detect any modifications (including additions and deletions) to a data resource. The value is a time range in ISO8601 format. This may be used to maintain replicas of a data resource such as a catalog of images, or a table.

COMPRESS is used to allow a service to return compressed datasets, e.g., a gzip-compressed version of the dataset file (this is not the same as HTTP-level stream compression, which would still deliver an uncompressed dataset to the client).

Data Access Layer Architecture

RUNID is a pass-through parameter used to tag all operations which are part of some larger distributed job. If a service operation should call other services, it passes on the value of RUNID, and so on down the line.

Other standard parameters will be defined for DAL2 as specification of additional services such as TAP and SIAV2 goes forward. For example a REGION parameter has been proposed for both TAP and SIAV2, which would allow use of a STC-S (or possibly STC-X) region specification in spatial queries.

3.3 Query Response

The form and semantic content of a DAL query was already covered in section 2.4. To summarize, the following are common to most DAL queries:

- **Generic dataset metadata.** Any query used for dataset discovery returns generic dataset metadata, augmented by metadata specific to the typed dataset, and possibly collection-specific metadata.
- **Form of response.** The query response is a table. Each row corresponds to a data model or schema of some sort. For dataset queries the table is returned as a VOTable, using UTYPE to identify data model elements. For more general queries the same semantic content may optionally be returned in other formats, e.g., FITS, CSV, text, HTML, etc.
- **Metadata Extension.** Metadata extension using standard mechanisms (additional component models or table fields; custom extension records) may be used to extend a standard model to add additional metadata specific to the type of data being described, without changing the basic query mechanism (generic clients are unaffected by the inclusion of the extra information).

Much of implementing a DAL service involves a query of some sort (the other major functionality being virtual data generation). While details such as exactly what query parameters are defined and the specific data model implemented by the query response will vary somewhat from one DAL query to another, the basic query mechanism can be the same for all such queries.

Mechanisms such as separation of data model from representation, the use of VOTable and UTYPE, the use of MAXREC etc. to manage the query response, can be common to all queries. In the case of dataset queries (SIA, SSA, etc.), much of the data model and metadata can be common to all such queries. Even in the case of services such as TAP, the same basic query mechanism can be reused, substituting a table schema for a DAL data model. All such reuse carries over to the client side as well, affecting client VO interfaces as well as client applications.

3.4 Standard Service Profile

As we saw in section 3.1.1, the DAL2 service model is a service, normally located by a single service endpoint (baseURL in the case of HTTP), which defines one or more service operations, identified by the `REQUEST` and `VERSION` parameters.

In general the operations defined by a service can be anything, depending upon the service functionality required. Most DAL services follow a similar pattern however, as the functionality is much the same for each service, the main difference being the type of data being queried or accessed.

For dataset-oriented services the standard service profile is as follows:

- **QueryData.** Query for data satisfying the given parameters. Used both for data discovery and to refine the parameters of a virtual dataset to be generated. Depending upon the service this may be a single operation or multiple operations; advanced query techniques such as inclusion of an ADQL expression could be used. Both GET and POST versions are possible in the case of an HTTP-based interface. Execution may be either synchronous or asynchronous.
- **GetData.** Get a single dataset as referenced in a prior queryData. This is not normally an explicit service operation at the protocol level; normally an access reference URL is used instead, to allow greater flexibility in how the actual data access is performed. GetData is always synchronous, always gets a single dataset, and is always provided as a GET in the case of HTTP. Transports other than HTTP are possible so long as they can be expressed as a URL.
- **StageData** (optional). Initiate an asynchronous operation to compute one or more datasets and stage them for later retrieval. As with getData, stageData refers to actual or virtual datasets referenced in a prior queryData; however multiple datasets may be referenced in a single stageData request. StageData may or may not be an explicit service operation. StageData is always synchronous, returning a result status and a JobID if the request is successful. The UWS pattern is used to monitor and control subsequent job execution. Data is normally staged either to service local JobID-specific temporary storage, or to a VOSpace.
- **GetCapabilities.** Get the service capabilities. This describes what the service can do, including the service interface (input parameters), what optional service capabilities defined by the service specification are implemented by the service instance, and any limits on maximum search region size, maximum table output size, and so forth. Output is returned as a registry-compatible resource Capability element, the contents of which are defined by the service specification. Execution is synchronous.

Data Access Layer Architecture

Since `getCapabilities` can be called by directly by a client application it is implemented as an explicit service operation (`REQUEST = getCapabilities`). VOSI compatibility is met since this reduces to a single fixed URL which can be set when the service is registered. Note that `getCapabilities` may take a `VERSION` parameter; this is required if a service instance implements multiple versions of a service interface.

Other VOSI operations are possible, for example **`getAvailability`**, to monitor that a service is available and functional, and **`getTableMetadata`**, to get registry-compliant table metadata for a service. As with `getCapabilities`, these are implemented as normal service operations which reduce to a single fixed URL which can be set when the service is registered. Both are synchronous.

In the case of non-dataset oriented services such as TAP (or the legacy cone search) the usual `queryData-getData` interaction pattern may not apply. In such a case we still have something very close to `queryData`, but it directly queries a data table with a table-specific schema, instead of querying a DAL data model. `GetData` is not required. `StageData` is still required, but as noted earlier may or may not be a separate explicit service operation (it may instead be a variation on the data query, implemented as a POST, which merely initiates an asynchronous job, since no access reference is required). All of the VOSI operations are the same for all service classes.

3.5 Grid Capabilities

By “Grid” capabilities we refer to service capabilities which go beyond what can be done with the usual synchronous HTTP GET or POST. This includes asynchronous execution, authentication including single sign-on authentication, scalable execution, coordinated distributed execution, and network storage of data on a per-user basis.

DAL relies upon the IVOA GWS standards (UWS, SSO, VOSpace, etc.) for all Grid capabilities. Aside from integration issues such as how an asynchronous job is created or how VOSpace storage is referenced in a DAL operation, these capabilities are independent of the type of data being accessed, and function the same for all DAL services, and indeed for all VO services. Hence they are an important class of capability which are common to all the DAL services, and which can probably share a common interface and implementation.

The details of how Grid capabilities are to be integrated into the DAL2 protocols are still being worked out. The initial DAL protocols, including SSA, do not yet support Grid capabilities so the details of how integration of Grid functionality into the DAL interfaces will be provided has not yet been specified. This is expected to be addressed in TAP and SIAV2, both of which will require Grid capabilities for high-end use-cases.

Data Access Layer Architecture

In general, to serve a broad set of use-cases, both simple synchronous, unauthenticated execution, and Grid-oriented execution (auth, async, etc.) are required. If the DAL services were to be used only within datacenter applications or large project software then it might be simplest to only implement the full-up Grid-enabled capabilities. If however we expect the user community to write software which directly accesses VO data then a simple open-access, synchronous mode of execution is also required. Experience over the past decade or more has proven that this is adequate for many applications, and important to encourage user take-up of VO-type middleware. High end applications however do require Grid capabilities.

References

[1] D. Tody, M. Dolensky, J. McDowell, F. Bonnarel, et.al, *Simple Spectral Access Protocol* , <http://www.ivoa.net/Documents/latest/SSA.html>

[2] D. Tody, F. Bonnarel, M. Dolensky, *Scope and Concepts of SIAV2* , <http://www.ivoa.net/internal/IVOA/SiaInterface/SIA-V2-Analysis.pdf>

[3] Jeff de la Beaujardiere ed., *OpenGIS® Web Map Server Implementation Specification*, <http://www.opengeospatial.org/standards/wms>

[4] R. Williams, R. Hanisch, A. Szalay, R. Plante, *Simple Cone Search*, <http://www.ivoa.net/Documents/latest/ConeSearch.html>

[5] D. Tody, R. Plante, *Simple Image Access Specification*, <http://www.ivoa.net/Documents/latest/SIA.html>