International

Virtual

Observatory

Alliance

Table Access Protocol

Version 0.3

IVOA Internal Working Draft 2008 October 20

This version:
TAP-V0.3-20081020
Latest version:
Not yet issued
Previous version(s):
http://www.ivoa.net/internal/IVOA/TableAccess/tap-v0.2.pdf
http://www.ivoa.net/internal/IVOA/TableAccess/TAP-QL-0.1.pdf

Editors:
P. Dowler, G. Rixon, D. Tody
Author(s):
K. Andrews, P. Dowler, J. Good, R. Hanisch, T. McGlynn, K. Noddle, F. Ochsenbein, I. Ortiz, P. Osuna, R. Plante, G. Rixon, J. Salgado, A. Stebe, D. Tody

---

Abstract

The table access protocol (TAP) defines a service protocol for accessing general table data, including astronomical catalogs as well as general database tables. Access is provided for both database and table metadata as well as for actual table data. Both simple filtering operations on individual tables as well as more general multi-table operations such as relational joins are supported. This version

of the protocol includes support for both ADQL-based queries and parameterized queries within an integrated interface, and includes support for both synchronous and asynchronous queries. Special support is provided for spatially indexed queries using astronomical coordinate systems. A multi-position query capability permits queries against an arbitrarily large list of astronomical targets, providing a simple spatial cross-matching capability. More sophisticated distributed cross-matching capabilities are possible by orchestrating a distributed query across multiple TAP services.

Status of This Document

This is a working draft internal to the DAL-WG.

This is an IVOA Working Draft for review by IVOA members and other interested parties. It is a draft document and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use IVOA Working Drafts as reference materials or to cite them as other than "work in progress".

A list of current IVOA Recommendations and other technical documents *can be found at http://www.ivoa.net/Documents/.*

Acknowledgements

"Ack here, if any"

Contents

1 Introduction

The *Table Access Protocol* (TAP) is a Web-service protocol that gives access to collections of tabular data. TAP services accept queries posed against the *tableset* (set of tables) available via the service and return the query response as another table, in accord with the relational model. Queries may be parameter based or may be composed as expressions in some query language such as the Astronomical Data Query Language (ADQL [*ref*]), or possibly native SQL, and may execute synchronously or asynchronously.

The result of a TAP query is another table, returned as a VOTable or optionally in some other format. This table contains directly the requested table data; it is not a table containing links to data objects to be downloaded separately (c.f. SIAP and SSAP).

The table collections made accessible via TAP are typically stored in relational database management systems (RDBMS), but TAP may also be implemented for data stored in other ways, such as in flat-file systems. This aspect of the implementation is abstracted by the protocol and is not visible to users. A TAP service exposes the database schema to client applications so that queries can be posed directly against arbitrary data tables available via the service.

TAP is a member of the IVOA Data Access Layer (DAL) family of data access protocols, conformant to the second generation (DAL2) interface standards [*ref*]. These standards provide uniformity among all the DAL2 protocols and include conformance to relevant query language, data model, VOTable, registry, and Grid and Web services standards (VOSI, UWS, etc.), where appropriate. Except where explicit reference is made to other standards documents an attempt is made to make the current specification self-contained, while maintaining conformance with more general IVOA standards. In case of conflict or ambiguity with the more general standards the TAP standard documented herein has precedence.

1.1 Parametric and ADQL-based Queries

This request, to an IVOA cone-search service, is an example of a parametric query:

```
http://some.where/some/thing?RA=180&DEC=42&SR=0.5
```

This query could be translated as “from the table dataset represented by the service at the URL `http://some.where/some/thing`, find all records for which the recorded position is within 0.5 degrees of the search position (180,42), where the coordinates are right ascension and declination in the ICRS coordinate system, measured in degrees”. The query is the boolean combination of the constraints expressed in the RA, DEC and SR (search-radius) parameters.

Parametric queries such as this are simple to express and to implement for cases where the data model is sufficiently well defined and adequate for the data to be queried, hiding many of the details required to pose and evaluate the query (both the simple spatial cone search as well as queries of the TAP metadata schema are examples of such simple well defined queries). When we query arbitrary *data tables* however, there often is no well defined data model, and the data table itself must be queried directly. The Astronomical Data Query Language (ADQL), a standardized sub-set of SQL92, was defined to deal with this more general use case.

Because ADQL has a formally-defined grammar it is feasible to build a complete parser for ADQL. Where an ADQL-consuming service uses a standard SQL-based DBMS as the back-end, it is possible to use an off-the-shelf ADQL parser to do most of the work required to generate SQL queries for the back-end DBMS. TAP includes provisions for ADQL queries for the general case as well as simplified parametric queries for the most common use cases.

1.2 Synchronous and Asynchronous Queries

We say that a TAP query is synchronous if the results of the query are delivered in the HTTP response to the request that originally posed the query. In this case, the service delays the response until the query completes or fails. Conversely, if the service returns an immediate HTTP-response upon accepting a query and the client later obtains the results of the query in response to a separate HTTP request, then we say the request is asynchronous.

In the synchronous case, the client must wait for the query to finish. If it times out or otherwise breaks communication before receiving the response, then the query fails. Synchronous queries are analogous to blocking I/O in a file-system; asynchronous queries correspond to non-blocking I/O.

Asynchronous queries require that client and server share knowledge of the state of the query during its execution and between HTTP exchanges. They are an

example of stateful interactions. In TAP, the mechanism by which the clients and services share the state of transactions is the Universal Worker Service (UWS) pattern. Synchronous queries are stateless between HTTP exchanges and need no such mechanism.

Synchronous queries are easier to implement, both for the client and the service; they are easier for scientists to use, and are adequate for most simple queries. However, there are many more advanced use-cases where synchronous queries are not sufficient. Therefore, TAP supports both synchronous and asynchronous queries.

1.3 Requirements for Compliance

The keywords "**must**", "**required**", "**should**", and "**may**" as used in this document are to be interpreted as described in the W3C specifications (IETF RFC 2119). Mandatory interface elements are indicated as **must**, recommended interface elements as **should**, and optional interface elements as **may** or simply "may" without the bold face font.

A fully compliant TAP implementation **must** provide the following capabilities:

- ADQL query with synchronous execution
- ADQL query with asynchronous execution
- Parametric query with synchronous execution
- Parametric query with asynchronous execution
- Table metadata query (synchronous, VOSI compliant)
- Service metadata query (synchronous, VOSI compliant)
- Service availability query (synchronous, VOSI compliant)

A minimally compliant service **must** provide synchronous and asynchronous ADQL queries and **should** provide the other features.

All capabilities are provided as service operations as specified in section 2.3. The VOSI compliant operations resolve to fixed URIs described in the service capability metadata. The TAP service including its service metadata **must** be registered with the IVOA resource registry.

2 Interface Overview

2.1 Architecture

A TAP service provides access to one or more tables, referred to as a ***tableset***, normally co-located at a single site. Multi-table operations such as joins or cross matches are possible provided the tables are all managed by the local TAP

service, and provided the service supports these capabilities. Larger scale operations such as a distributed cross match are also possible, but require combining the results of multiple TAP services. In the most general case table operations might make use of any of the following components:

- A top level application, for example a ***cross-match portal*** capable of multi-parameter statistical cross-matching of distributed tables. Access to remote tables is via TAP services running locally where the table data is stored. The remote TAP service might perform a first order spatial cross match or apply other constraints to filter the data at access time, thereby reducing the volume of data which has to be passed back to the remote portal or application.

- One or more ***TAP services***, each providing access to one or more tables via a range of query capabilities, similar to what is typically provided by an individual DBMS. Both table data and metadata may be accessed via the same query interface.

- The ***ADQL*** query language, provides an advanced query language capability based upon SQL, enhanced with astronomy specific extensions, e.g., for applying spatial query constraints. Pass-through of native SQL may also be possible if permitted by a specific service.

- A ***table data model***, if supported by a service, can optionally be used to access a table without having to understand the details of how information is stored in the table (this is a planned future capability not yet supported by TAP, but is part of the architectural design). This capability is especially important when a client might access many different tables from a variety of origins. For example, a source catalog data model might define a number of standard attributes for a “source” object (position, extent, morphology, brightness, etc.), which the TAP service would map to a native data table on behalf of the client.

While we highlight the cross match portal here as a primary example of a TAP client application, any variety of other client analysis applications are possible, including custom applications written directly by end users, or incorporated directly into analysis environments. All such applications share the same underlying data access facilities, which provide a common interface profile and semantics for access to tables or other types of astronomical data.

TAP also makes use of additional VO technology, in particular *VOTable* provides a standard model and format for table interchange, *VOSpace* is used for network table storage and transport, and *UWS* provides a standard interface pattern for interacting with asynchronous services. Certain TAP operations for determining the capabilities and runtime status of a service instance are based upon the VO Standard Interfaces (*VOSI)* standard. VO standards for single sign-on

authentication are used to manage resources on behalf of a user and provide secure access to data where necessary.

2.2 Service Operations

A TAP service implements multiple service operations, each of which performs some well defined function when invoked by a client application. The service operations described here use HTTP GET and POST as the low level communications protocol. The functionality of each operation is defined independently of the low level communications protocol, and semantically equivalent operations could be implemented in the future via other protocols.

TAP defines the following standard service operations:

- **AdqlQuery.** Execute an ADQL query (or a query in some other query language if supported by the service). The query is passed as string, which may be URL encoded if required by the low level protocol used. General relational operations upon multiple tables are supported. Both synchronous and asynchronous versions are provided. Data tables may be uploaded or may optionally be staged to a VOSpace.

- **ParamQuery.** Execute a parameterized query. The query is defined by a set of parameters rather than by a free form language as for AdqlQuery. Both table data and metadata can be queried with the same interface, and ParamQuery provides the standard mechanism used to query table metadata. Except for some well defined special cases (multi-position queries, tableset queries), queries are limited to a single table. Both synchronous and asynchronous versions are provided. Data tables may be uploaded or may optionally be staged to a VOSpace.

- **GetCapabilities.** Return a standardized XML description of the capabilities of the service instance, describing what the service is capable of doing (VOSI compliant, registry cacheable and searchable).

- **GetAvailability.** Return a standardized XML description of the runtime status of the service, describing the state and availability of the service (VOSI compliant).

- **GetTableMetadata.** Return a standardized XML description of the tableset metadata for all tables available via the service (VOSI compliant). In the case of TAP this is not an actual service operation, but rather a URI which resolves into a specific table metadata query via ParamQuery.

The AdqlQuery and ParamQuery operations provide two alternative ways to pose queries against the service. These queries differ only in the way they are posed. Once the query inputs (ADQL statement in the case of AdqlQuery, or parameter

set in the case of ParamQuery) are translated by the service into whatever form the back-end DBMS requires, execution is the same for both types of query. Hence table uploads, VOSpace integration, output formatting, asynchronous execution facilities, table metadata, and so forth are identical for both forms of query. In addition, the same query interface is used for both table data and metadata, simplifying the service interface and providing uniform, fully featured facilities for querying both types of data.

2.3 Service Profile

The basic form of a TAP service (or any other second generation data service, all of which share the same basic service interface) is specified in detail in section 7. In the current section we merely summarize the elements of the basic service interface.

2.3.1 Request Format

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

The TAP interface described in this document is based upon 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).

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.

Example:
`$<service-baseURL>/sync?REQUEST=AdqlQuery&QUERY=…`

TAP defines two versions of the baseURL, one for synchronous operations and another for asynchronous operations. These are formed by contentating the *service-baseURL* with either "`/sync?`" or "`/async?`". Hence for synchronous operations we have a full baseURL of

`<service-baseURL>"/sync?"`

and for asynchronous operations the full baseURL is

`<service-baseURL>"/async?"`

In general the service operation is much the same whether or not it executes synchronously or asynchronously. Minor differences in service operation

function or input parameters are possible and are defined in the description of the individual service operation below.

Note that since a URI pathname segment is appended to the service baseURL the service baseURL may not contain any HTTP GET parameters, and must be a fixed URI.

2.3.2 Parameters

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 service operations define the following standard parameters, which are part of the basic service profile:

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

The `REQUEST` parameter specifies the service operation to be executed. `VERSION` allows a specific version of the interface to be requested. The values of both the `REQUEST` and `VERSION` parameters are case-insensitive.

A given service instance may support multiple versions of the TAP interface, and by default the service assumes the highest *standard* version which is implemented (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 necessary 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.

2.3.3 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 formal output metadata, other than in ISO 8601 formatted time strings (sexagesimal format is permitted in any informal output intended for a human, e.g., text or HTML formatted tables).

TAP defines a special ***range-list*** format for specifying numerical ranges or lists of ranges as parameter values. For example, "`1E-7/3E-6`" specifies a closed range from 1E-7 to 3E-6 inclusive. The syntax supports both open and closed ranges. Ranges or range lists are permitted only when explicitly indicated in the definition of an individual parameter. A variant of the range list is the value of the

WHERE parameter, used to specify the query constraint for a ParamQuery operation. For a full description of range list syntax refer to section 3.3.1.

2.3.4 Use of GET and POST

Where specified, individual service operations may provide both HTTP GET and POST forms for issuing the service request. Both forms share the same input parameters and operation semantics, being merely two different ways of invoking the same service operation. In general, the GET form is used for synchronous operations which are idempotent (have no side effects, the result is cacheable, multiple instances may be simultaneously active and will return the same result). POST is used for any request which has a side effect, e.g., initiation of an asynchronous job, or which needs to pass a large amount of data to the service, e.g., uploading a table or region mask to be used within a query.

2.3.5 URL Encoding

URL encoding (see section 7.3.2) is a standard technique used to encode characters appearing in HTTP requests, such as a GET URL, to pass characters which are not otherwise legal and could interfere with the HTTP protocol. By using URL encoding it is possible to pass arbitrary character data to a service in a HTTP request, for example an arbitrary ADQL statement could be passed in a simple GET request so long as it is not too large for a GET URL (2K or so characters).

2.3.6 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`". More fundamental service or protocol errors may result in an HTTP level protocol error, hence a client program should be prepared to handle either response. A null query, that is a queryData which does not find any data, is not considered an error; likewise an overflow condition is distinguised from error. More information on error responses is given in section 7.

2.4 Request Examples

Some examples of simple TAP requests follow. These are intended only to help introduce and illustrate basic usage of the TAP service interface; the details are specified in the following sections of this document.

Synchronous parametric query performing a simple cone search of table "foo" (*baseURL* would be replaced with the actual service base URL):

```
$baseURL/sync?REQUEST=paramquery&POS=12,34&SIZE=0.5&FROM=foo
```

Synchronous ADQL query returning all data from table "foo":

```
$baseURL/sync?REQUEST=adqlquery&QUERY=select+*+FROM+foo
```

Simple cone search query executed asynchronously:

```
curl -d 'REQUEST=paramquery&POS=12,34&SIZE=0.5&FROM=foo' \
   $baseURL/async
curl -d 'PHASE=RUN' $baseURL/$jobID
    [wait]
curl $baseURL/$jobID/results
```

In this example the commonly available *curl* application (*wget* or a browser could also be used) is used to issue HTTP `GET` and `POST` requests to the remote UWS-based job manager. The query may run for an arbitrarily long time. When the job completes the output can be retrieved.

Asynchronous version of our simple ADQL-based query:

```
curl -d 'REQUEST=adqlquery&QUERY=select+*+FROM+foo' $baseURL/async
curl -d 'PHASE=RUN' $baseURL/$jobID
    [wait]
curl $baseURL/$jobID/results
```

Simple synchronous table metadata request to list the columns of table “foo”:

```
$baseURL/sync/REQUEST=paramquery&FROM=TAP_SCHEMA.COLUMNS& \
    WHERE=tablename,foo
```

3 TAP Service Operations

3.1 Synchronous ADQL Query

The *AdqlQuery* operation is used to pose a query expressed using a query language, normally the Astronomical Data Query Language (ADQL). The query is posed against the *tableset* (set of tables) exposed by a TAP service. ADQL is a variant of the Structured Query Language (SQL) providing a uniform query language which can be translated and executed against any modern RDBMS. Extensions, e.g., for spatial queries in astronomical coordinate systems, are provided to enhance the query language for astronomical queries.

An ADQL query may be used to query a single table, or may query multiple tables in a single operation. The input tables to be queried may be any table exposed by the TAP service, or any external table accessible via a URI or uploaded inline as part of the query. The query response for a successful query is the table produced as the result of the query. Output tables are returned in VOTable format by default, although other output formats may optionally be provided by the service (see sections 3.1.1.3 and 4.5). Queries may execute either synchronously or asynchronously.

Our purpose here is merely to define the use of the TAP interface to execute an ADQL query. Specification and usage of the ADQL language is covered elsewhere ([*ref*]).

The parameters used to control the AdqlQuery operation are defined in the remainder of this section.

Synchronous AdqlQuery requests may be submitted with either a GET or a POST (so long as POST is supported by the service, e.g., if it supports inline table uploads).

3.1.1 Input Parameters

3.1.1.1 QUERY

A service which implements the AdqlQuery operation **must** support the QUERY parameter, used to input the ADQL (or other query language) statement to be executed. The query string should be URL-encoded by the client if it contains any characters not legal in a URL (see section 7.3.2). The query string is case sensitive. In particular, the case of table and column names should be preserved between a metadata query and a subsequent query of a data table.

3.1.1.2 LANG

The service **should** implement the LANG parameter. The value is a string specifying the language and optionally the language version used for the QUERY parameter, as defined by the service capabilities. A service which implements the AdqlQuery operation **must** support "`ADQL`" (case insensitive) as the default query language. The service **may** support other query language encodings as well, e.g., other ADQL versions, or pass-through of native SQL, as specified by the service capabilties (3.5). The version of the query language may optionally be specified, e.g., "`ADQL-1.0`" (the syntax should be as shown). The service should return an "unknown query language" error if an unsupported and incompatible value of LANG is specified.

3.1.1.3 FORMAT

The service **should** implement a FORMAT parameter specifying the output format requested by the client, specified either as a MIME type or as one of the shorthand forms "`votable`", "`csv`" (comma separated values), "`fits`" (FITS binary table), "`text`" (pretty-printed text), or "`html`" (pretty-printed Web page), all such values being case independent. The service should return an "unsupported output format" error if an unsupported output format is requested. The default table output format if no format is specified is VOTable. All services **must** support at least VOTable as an output format, and **must** permit a query with `FORMAT` indicating VOTable, without error.

3.1.1.4 UPLOAD

The service **should** implement an UPLOAD parameter, used to reference read-only external tables via their URL, to be uploaded for use as input tables to the query. Tables uploaded in this fashion are assumed to be encoded in VOTable format. The value of the UPLOAD parameter is a list of table name-URL tuples, delimited by semicolon, using comma to delimit each table name-URL tuple (that is, a list-structured parameter as specified in section 7.7.2). For example:

`UPLOAD=table_a,http://host_a/path;table_b,http://host_b/path`

would define two input tables “table_a” and “table_b”, located at the given URLs (URL-encoding is mandatory in this case since we embedding a URL within a URL). The specified table names are arbitrary but must be legal ADQL table names and must be unique within the upload table namespace for the lifetime of the query (4.2). The table name should be a simple table name; uploaded tables will automatically be placed in the upload table storage area (schema) hence no schema name prefix is required. The upload table storage area is shared with any tables uploaded in-line with the query. The specified table names are used to refer to the uploaded tables in the query, prefixed with an upload namespace (DBMS schema) as specified in section 4.2.

3.1.1.5 MAXREC

The service **should** implement a MAXREC parameter specifying the maximum number of table records (rows) to be returned. If the result set for a query exceeds this value a valid data table should be returned with a status of “OVERFLOW” indicating that overflow occurred (this may not be supported for output formats other than VOTable; see also section 7.10.2).

The default MAXREC value defined by a service should be large enough to avoid overflow for most small queries, but small enough to provide a response to the user reasonably quickly. The client may override the default MAXREC, increasing the value up to the maximum value permitted by the service, as defined in the service capabilites. A sufficiently large MAXREC may permit streaming of arbitrarily large output tables. Output tables larger than the maximum permitted value of MAXREC must use some other technique such as asynchronous computation of the output table followed by retrieval using a streaming synchronous GET (VOSpace output may also be supported in a later version of TAP).

In the case of a large output table which is streamed back to the client as it is being computed it may not be possible to know in advance whether overflow will occur (for a fully streamed response the VOTable header may be output before the table data has been computed). In this case the output should be returned with a query status of “OK”, indicating a valid query; if overflow occurs, MAXREC plus one rows should be returned to indicate that overflow occurred.

A value of MAXREC=0 indicates that, in the event of an otherwise successful query, a valid output table should be returned containing metadata but no table data rows. It is up to the service whether or not to actually execute the query and generate table rows which will be discarded; the query status should be returned as "OK" so long as the query is otherwise valid. This is an example of a ***null query***, that is, a query which produces an empty table.

3.1.1.6 MTIME

The service **may** support an MTIME parameter, used to query a table for only rows which were modified within a given range of times, specified as an ISO8601 open or closed range list in the UTC time system. A "modified" row is a table row which was inserted, updated, or deleted during the indicated time interval (hence MTIME may be used to see deleted rows which are not visible in any other fashion). This feature may be used by a remote client to maintain a replica of a large table, or to periodically poll a table for changes. The period of time for which deletions are preserved is server dependent (depending upon how often deleted rows are purged) but should be at least one week.

3.1.1.7 RUNID

The service **should** implement the RUNID parameter, used to tag service requests with the job ID of a larger job which the request may be part of. For example, if a cross match portal issues multiple requests to remote TAP services to carry out a cross-match operation, all would receive the same RUNID, and the service logs could later be analyzed to reconstruct the service operations initiated in response to the job. The service need not do anything with RUNID other than pass the parameter on to any other services which it in turn calls, e.g., a VOSpace. The service should also ensure that RUNID is preserved in any service logs.

3.1.2 Query Response

The response of a successful synchronous AdqlQuery request is the table resulting from the query (4.4). By default the query response is returned as a VOTable. The response of an unsuccessful AdqlQuery request is an error response VOTable (7.10).

3.2 Asynchronous ADQL Query

The asynchronous form of the ADQL query is semantically the same as the synchronous form, the main difference being that the query executes asynchronously (it keeps running after an initial synchronous request to start the job), and hence may run an arbitrarily long time. All request parameters are otherwise the same for both execution modes of the service operation.

The Universal Worker Service (UWS) mechanism is used to manage the request. The asynchronous ADQL query is specified as for the synchronous version except that it is always submitted as a HTTP POST request to the

$*baseURL*/async service endpoint. All subsequent interaction conforms to the UWS specification.

```
POST to $baseURL/async:
    REQUEST=AdqlQuery&
    QUERY=[…] (etc.)
```

Upon receiving the request the service will execute the query, serializing the output table in the encoding specified by the FORMAT parameter. MAXREC applies to the asynchronous request the same as for the synchronous request, hence may limit the size of the output table generated.

During and subsequent to execution of the query, the client may monitor and control the job using the controls specified by UWS. The output table is stored as the job output and may be retrieved via the UWS mechanism once the job completes.

If the query fails, the service shall report this using the mechanisms specified in UWS.

3.3 Synchronous Parametric Query

The *ParamQuery* operation provides basic access to both table data and metadata without requiring the full generality of ADQL. ParamQuery is the primary mechanism provided to query table metadata with TAP; one can easily obtain a list of the tables available via a TAP service, or a list of the columns of an individual table, in any supported output format, using the standard table query mechanism. ParamQuery also allows simple filter-type queries of individual data tables. Most queries of astronomical catalogs are of this type. Simplified and optimized support is provided for common spatially-indexed astronomical query use cases such as cone search and multi-position queries. There is no inherent limit on the size of input or output tables.

ParamQuery differs from AdqlQuery only in the way the query is posed. In both cases query execution and output processing may be the same. Many of the input parameters are common to both operations.

As for AdqlQuery, parameter values for ParamQuery are case sensitive except where otherwise specified. In particular table and column names are case sensitive, and case should be preserved between a table metadata query and the subsequent use of a table or table column name in a query.

3.3.1 Input Parameters

3.3.1.1 POS, SIZE

The POS and SIZE parameters provide an easy to use, optimized facility for performing spatial queries of astronomical catalogs, similar to the legacy cone search protocol. Spatial queries are supported only for tables which contain positional information (e.g., RA and DEC for each table record), however many astronomical catalogs are of this type.

POS and SIZE define a circular search region in the specified coordinate system (default ICRS). A service which implements ParamQuery **must** support the POS and SIZE parameters, and implement them as a query constraint for tables containing records tagged with spatial positions. If POS and SIZE cannot be applied to the referenced table an error should be returned.

The coordinate values for POS are specified in list format (comma separated) with no embedded white space, as defined in section 7.7.2 and as implemented in other second generation DAL interfaces.

```
POS=52,-27.8
```

POS defaults to right-ascension and declination in decimal degrees in the ICRS coordinate system. A coordinate system reference frame **may** optionally be specified to indicate a spatial coordinate system other than ICRS. The reference frame is specified as a list format modifier, with the acceptable values as defined by Table 3 (standard reference frames) in STC (Rots 2007).

```
POS=52,-27.8;GALACTIC
```

Whether or not a service supports coordinate systems other than ICRS for POS is an optional service-defined capability (solar and planetary data for example might use other coordinate systems). It is an error if a coordinate reference frame is specified which the service does not support.

POS also defines a special syntax which is used to reference a table of positions for ***multi-position queries***. This is discussed separately in section 3.3.5.

SIZE specifies the diameter of the search region input in decimal degrees.

```
SIZE=0.05
```

A valid query does not have to specify a SIZE parameter. If SIZE is omitted in a positional query, the service should supply a default value intended to find nearby objects which are candidates for a match to the given object position, taking into account the spatial resolution of the data.

3.3.1.2 REGION

The ParamQuery operation **may** implement a REGION parameter, used to define more general spatial search regions than can be defined using POS, SIZE. The value is a STC/S (string encoded) region specifier.

```
REGION=Ellipse ICRS 148.9 69.1 2.0 4.0 32.7
```

In the example above the embedded spaces are shown for clarity, but if used in an URI they should be URL encoded.

If both POS,SIZE and REGION are specified in the same query, REGION acts as a mask to further qualify the circular region specified by POS,SIZE. This is most useful for multi-position queries (see section 3.3.5), where a large table of possible search positions may include positions outside the desired search region. In this case REGION specifies the sub-region of the referenced table to be used. This allows large tables to be used in a multi-position query. In particular it permits a cross match of two data tables (e.g., two large astronomical catalogs) to be performed in a single operation, restricting the spatial portion of the cross match to the mask region.

3.3.1.3 SELECT

The ParamQuery operation **must** implement a SELECT parameter, used to specify the table fields to be returned by the query, specified either as a comma delimited list of field names, or optionally by specifying one of the reserved values "`$STD`" (to return only the standard or "primary" fields), or "`$ALL`" (to return all table fields).

```
SELECT=ra,dec,flux
```

By default only the "primary" fields are returned. The "primary" fields are specified on a per-table basis, and define a subset of the most important table fields. This is used to provide a more readable view of very wide tables. The service must permit `$STD` and `$ALL` to be input without error, but is not required to actually use them to adjust the view of the table. If no "narrow" view is defined for a table the service should ignore `$STD` and merely return all table fields.

3.3.1.4 FROM

The ParamQuery operation **must** implement a FROM parameter, indicating the name of the table to be queried, specified as defined in section 4.1. Only a single table reference is allowed. There is no default, hence it is an error if no table name is specified, or if the specified table name is invalid.

```
FROM=hdfv2
```

In addition to the data tables managed by the service, tables in the query upload area (4.2) may be referenced, as well as the (real or virtual) metadata tables defined by the TAP information schema (3.3.3, 5).

In a client query FROM must be specified to identify the table to be queried. SELECT and WHERE are optional.

3.3.1.5 WHERE

The ParamQuery operation **must** implement the WHERE parameter, used to specify an optional filtering constraint to be applied to the table to determine which table rows are returned. By default all table rows are returned.

In a client query, WHERE may be combined with other query constraints such as POS and REGION to further refine the query.

The syntax of the ParamQuery WHERE parameter value (not to be confused with the SQL WHERE clause of the same name) is a simple sequence of equality or range constraints delimited by semicolons, with the field name and value elements of an individual constraint separated by a comma.

A simple example should help illustrate the syntax:

```
WHERE=observer,*smith*;z,1.5/2.2
```

This specifies two table field constraints: the field “observer” must contain the case-insensitive substring “smith” (hence the wildcards), and the field “z” must be in the range 1.5 to 2.2 inclusive. This syntax is explained in more detail below.

The ParamQuery WHERE syntax has deliberately been kept simple as TAP already has ADQL to provide a fully general expression evaluation capabillity, which should be used for advanced data queries. Each constraint applies to a single table field; multiple constraints on the same table field are allowed. The constraints have an AND relationship, hence all must evaluate to true for a table row to satisfy the WHERE. Individual constraints may be negated to construct more complex expressions.

The syntax chosen is intended to easy to compose, easy and unambiguous for a service to parse and map to a SQL back end or otherwise evaluate (a conventional rule-based parser is not required). It was also chosen to be consistent with similar usage in other data access services, e.g., in the use of range-list syntax (7.7.2) for the WHERE expression (the BAND, TIME, etc. parameters in other DAL services use the same range-list syntax). An effort has been made to define a minimal set of metacharacters so as to minimize the need for URL encoding – most simple expressions should not require URL encoding,

e.g., if typed interactively into a Web browser, allowing the simplest Web tools to be easily used for basic queries.

A partial BNF for the WHERE expression is as follows:

```
<where-expr>    ::= <field-list>
<field-list>    ::= <field-expr> [ ';' <field-list> ]
<field-expr>    ::= <field> ',' ['!'](<list> | "null")
<list>          ::= <numeric-list> | <string-list> | <date-list>
<numeric-list>  ::= <number> [ ',' <numeric-list> ]
<string-list>   ::= <string> [ ',' <string-list> ]
<date-list>     ::= <date> [ ',' <date-list> ]
```

Where we have not attempted to detail the BNF for the numeric, string, and date tokens. Some additional notes follow.

- Each field expression defines a constraint on the named table field (column).

- Field expressions are of the form <*field-name*>','<*value*> (meaning field-name=value), where <*value*> is a range list (a single value, a range, or a list of single values or ranges all of the same type). Constraint expressions have an "and" relationship within the overall WHERE expression. Values within a range-list have an "or" relationship, i.e., the range-list for a specific field reference is a list of valid values.

- A parameter value may optionally be prefixed with '!' (exclamation) to negate the sense of the entire clause.

- The special value "null" indicates a null-valued field. For example "`flux,!null`" is true only if field "flux" has a non-null value.

- A `<date>` conforms to ISO8601 date syntax, e.g., "2007-04-05T14:30".

- A `<number>` token is any legal integer or floating point number optionally preceded by '+' or '-'.

- A `<string>` token is any token which is not a number or date, or any sequence of characters which is quoted using single quotes.

- While accumulating a string token, anything quoted in single quotes is literally included in the string, otherwise (where case-insensitive context applies), characters are converted to lower case for use in case-insensitive comparisons. Quoted characters are treated in a case sensitive fashion. Any metacharacter other than the quote character may be quoted to include it within a token. A single quote may be included within a string by quoting it (that is, three single quotes in sequence). Quotes used within a string token do not delimit the token.

- For string-valued fields the constraint is a case-insensitive simple pattern, with "*" matching zero or more characters. Absent any use of "*", the entire string must match. Hence "obj,m31" specifies that the value of field "obj" must match "m31" exactly, except for case. To force a case sensitive match the case sensitive characters must be quoted.

- For numeric or date values the constraint is either a single value or a range, using "/" as the range delimiter (range syntax is not supported for strings). Both open and closed ranges can be specified, e.g., "5/" specifies an open range equivalent to "greater than or equal to 5", whereas "5/9" means "5 to 9 inclusive".

- Spaces may be embedded to improve readability, but if so they must be URL encoded as "%20".

Field names or value expressions must be quoted if they contain any special characters (e.g., semicolon, comma, forward slash, asterisk). The single quote is used to avoid conflict with double quote which is often used to quote the entire URL string.

As a more complex example of WHERE usage consider the following somewhat contrived expression:

```
vmag,4.5/5.5; imag,4.5/; bmag,/5.5; flag,4,5,6;
jmag,4.5/5.5,/3.0,9.0/; name,*Lon*; kmag,4.5/5.5; flux,null;
last,1
```

The equivalent SQL WHERE clause would be the following:

```
vmag between 4.5 and 5.5 and imag >= 4.5 and bmag <= 5.5
  and (flag = 4 or flag = 5 or flag = 6)
  and (jmag between 4.5 and 5.5 or jmag <= 3.0 or jmag >= 9.0)
  and name like '%Lon%' and kmag between 4.5 and 5.5
  and flux is null and last = 1
```

The following is a complete example of a typical ParamQuery of a data catalog. This returns the selected fields from the "fp_psc" catalog for sources within 0.2 degrees of the given position, where the J magnitude is less than or equal to 10.

```
$baseURL/sync?REQUEST=ParamQuery&
   SELECT=ra,dec,j_m,h_m,k_m &
   FROM=fp_psc &
   POS=10.68469,41.26904 &
   SIZE=0.2 &
   WHERE=j_m,/10.0
```

3.3.1.6 Other Query Parameters

The FORMAT, UPLOAD, MAXREC, MTIME, and RUNID parameters are identical for both ParamQuery and AdqlQuery. Refer to section 3.1.1 for a description of these parameters.

3.3.2 Query Response

The ParamQuery query response for a simple data or metadata query is identical to that for AdqLQuery, and is detailed in section 4.4. Multi-position queries and *tableset* queries (a form of table metadata query) can produce somewhat different responses as detailed in the following sections.

3.3.3 Table Metadata Queries

Rather than provide access to database and table metadata via a separate interface, querying of database and table metadata is provided by representing such metadata as just another set of set of tables which can be queried like any data table, using the same query interface, output formatting, and so forth. This is done by defining a special database schema called the *TAP schema*, referred to as "`TAP_SCHEMA`" in queries (upper case is required).

This approach has the advantage of allowing the standard TAP table query interface to be used to query table metadata as well as ordinary table data. In addition, since table metadata is represented as data which is queried at runtime, table metadata can vary dynamically, and is easily extended without any changes to the query interface.

The core TAP schema defines the following tables (more detailed information on the TAP schema is given in section 5.1):

- **TAP_SCHEMA.tables**. Lists all tables (including views) accessible via the TAP service and visible to the current client or user.

- **TAP_SCHEMA.columns**. Lists all columns (fields) of all tables known to the TAP service and visible to the current user. A query specifying a specific table name will return only the columns of that table.

- **TAP_SCHEMA.schemas**. Lists all database schemas (named table storage areas) accessible via the TAP service and visible to the current client or user. Knowledge of the schemas supported by a service is not required for basic table access (the schema name is included in the table names returned in `TAP_SCHEMA.tables`), but is useful to understand what data is available.

Some examples should help illustrate how table metadata queries are used. All of these queries should include the request name, i.e., `REQUEST=ParamQuery`. If no output format is specified data will be returned in VOTable format, however output may be requested in any table output format supported by the service.

List all the tables known to the service and visible to the client (no WHERE is needed since all tables are to be listed):

```
FROM=TAP_SCHEMA.tables
```

List all columns of table “fp_psc”:

```
FROM=TAP_SCHEMA.columns&WHERE=tablename,fp_psc
```

List the full *tableset* supported by the service, in registry compliant XML format:

```
FROM=TAP_SCHEMA.tableset&FORMAT=xml
```

The tableset query is eqivalent to a query of `TAP_SCHEMA.tables` except that *all* tableset metadata is returned, i.e., all tables matching the query, and all columns of each table. Only two output formats are supported for a tableset query, “`xml`” (registry compliant XML) and “`votable`” (a dataless VOTable containing only TABLE and FIELD metadata describing the tables in the tableset). The metadata returned is the same as for a `TAP_SCHEMA.tables` or `TAP_SCHEMA.columns` query, the only difference being that all metadata is returned and output is available in only two specialized output formats aggregating data from multiple tables (hence the output is not relational in the normal sense). Tableset output is required for VOSI compliance as specified in section 3.5.

A TAP service **must** support the above table metadata queries. A TAP service **may** support more advanced table metadata queries using other ParamQuery parameters. Some examples of advanced table metadata queries follow.

List all database schemas queryable by the service, in pretty-printed text format:

```
FROM=TAP_SCHEMA.schemas&FORMAT=text
```

List only the tables in the database schema “hdf”:

```
FROM=TAP_SCHEMA.tables&WHERE=tablename,hdf.*
```

List only tables containing data within the specified region on the sky:

```
FROM=TAP_SCHEMA.tables&POS=180.0,1.0&SIZE=1.0
```

As above, but return tableset metadata in VOTable format:

```
FROM=TAP_SCHEMA.tableset&POS=180.0,1.0&SIZE=1.0
```

List only the table name and description fields for all tables:

```
SELECT=tablename,description&FROM=TAP_SCHEMA.tables
```

List only tables modifed since July 4 2005:

```
FROM=TAP_SCHEMA.tables&MTIME=2005-07-04/
```

An advanced TAP service **may** also support querying of table metadata via ADQL.

TAP services are required only to be able to list all tables or all the fields of a single table, as well as provide full tableset metadata in both XML and VOTable format. A more advanced TAP service will support general table metadata queries using the full ParamQuery interface.

3.3.4 Cone Search Query

The POS and SIZE parameters provide an optimized spatial query capability comparable to the legacy cone search interface. For example, the following would execute a cone search of table “fp_psc” using the specified position and search region diameter, selecting only objects for which the J magnitude is less than or equal to 10:

```
$baseURL/sync?REQUEST=ParamQuery&
  FROM=fp_psc&POS=180.0,0&SIZE=0.2&
  WHERE=j_m,/10.0
```

Legacy cone search also provided a VERB parameter to control which fields are returned in a query. This is equivalent to a SELECT, with “$STD” providing the default “narrow” view, and “$ALL” returning all table fields.

ParamQuery thus duplicates all the functionality of the legacy cone search and is equally easy to use, and nearly as simple to implement. It is much more powerful however, since a single service can support multiple tables, the table metadata can be queried as easily as the table itself, additional query constraints can be specified to refine the query, spatial coordinate system frames other than ICRS can be specified, non-circular regions can optionally be specified using REGION, multiple output formats can be specified, optional Grid capabilities are available to permit large queries, and as we will see in the next section, multi-position queries can be used to provide a “multi-cone search” type of capability.

3.3.5 Multi-Position Queries

A *multi-position* query generalizes POS, SIZE to a table of positions, allowing an arbitrarily large number of spatial queries to be executed simultaneously. In a typical scenario the user uploads a list of the positions of their favorite objects, and executes a spatial cross match against some data table. Additional query contraints may be specified with the WHERE parameter to further refine the query. If desired the query may be further refined on the client side, providing a more sophisticated multi-parametric distributed cross match as a two step operation.

A multi-position query is indicated by using POS to point to a table containing positions, instead of inputing a single position directly. Any table can be used so long as it contains position information for each table record.

The POS syntax used to point to a table of positions is "`POS=@`*`tablename`*", where *`tablename`* can be any valid table known to the TAP service. For example the client might upload a table named "positions" when executing the multi-position query, in which case the query might be:

```
$baseURL/sync?REQUEST=ParamQuery&
   POS=@TAP_UPLOAD.positions&SIZE=0.2
```

Alternatively the asynchronous form of the query could be performed, using POST to submit the same query parameters and upload table (4.2) to the $*`baseURL/async`* service endpoint.

UTYPE	*UCD*	*Description*
src:Position.ID	meta.id;meta.main	Position identifier
src:Position.Coord1	pos.eq.ra;meta.main	Right Ascension, degrees
src:Position.Coord2	pos.eq.dec;meta.main	Declination, degrees
src:Position.Size	instr.fov [??]	Diameter of search region

In the most general case any table containing position information may be used. For example we could use the 2MASS point source catalog from our earlier examples, assuming a copy is available to the TAP service. This table contains nearly half a billion sources, so the REGION parameter would be used to apply a spatial mask to restrict POS to only the positions within the specified region. In this case we might have "`POS=@fp_psc`", with REGION specifying whatever spatial region the user requires. Additional query constraints could optionally be added to further refine the query.

While any table could be used for POS, some mechanism is needed to identify the table fields to be used to define positions. The table fields to be used from the input positions table can be identified within the table using the UTYPE and UCD tags given in the table above. Specification of UTYPE is the preferred technique, otherwise a fallback to UCD should be attempted (UTYPE is preferred

over UCD as UCD is difficult to generalize to multiple coordinate systems). Although the table above refers to RA and DEC, any coordinate system can be used for positions if supported for spatial queries by the service.

When an existing data table managed by the TAP service is used for position input (such as `fp_psc` in our example above) it is the responsibility of the TAP service to know what table metadata to use for input positions, but this is already the case to be able to do a simple single-position cone search on such a table.

The output from a multi-position query is a single table, containing output data for all query positions, with a sequence of zero or more table rows corresponding to each input position. A unique position ID column **must** be added to the output table to indicate the position from the *input* position table to which the *output* table row corresponds. If positions in the input table have been assigned a unique position identifier (i.e., `src:Position.ID`) then this is what should be used, otherwise a 1-indexed integer position ID specifying the row of the input table should be used as the unique position ID in the output table. The other output table fields are copied from the data table being queried.

If a SIZE parameter is specified the value given applies to all positions. Otherwise the region size is taken from the input position table, and is allowed to vary for each position. It is an error if SIZE is not specified in either the input position table or as a parameter.

3.4 Asynchronous Parametric Query

The asynchronous form of ParamQuery is semantically the same as the synchronous form, the main difference being that the query executes asynchronously (it keeps running after an initial synchronous request to start the job), and hence may run an arbitrarily long time. All request parameters are otherwise the same for both execution modes of the service operation.

The Universal Worker Service (UWS) mechanism is used to manage the request. The asynchronous parametric query is specified as for the synchronous version except that it is always submitted as a HTTP POST request to the $*baseURL*`/async` service endpoint. All subsequent interaction conforms to the UWS specification.

```
POST to $baseURL/async:
    REQUEST=ParamQuery&
    FROM=[…] (etc.)
```

Upon receiving the request the service will execute the query, serializing the output table in the encoding specified by the FORMAT parameter. MAXREC applies to the asynchronous request the same as for the synchronous request, hence may limit the size of the output table generated.

During and subsequent to execution of the query, the client may monitor and control the job using the controls specified by UWS. The output table is stored as the job output and may be retrieved via the UWS mechanism once the job completes.

If the query fails, the service shall report this using the mechanisms specified in UWS.

3.5 VOSI Operations

The Virtual Observatory Standard Interfaces (VOSI) standard [*ref*] specifies certain elements of the service interface which all VO services must adhere to. The second generation data access layer (DAL2) interfaces [*ref*] extend this standard to define the specific form which the VOSI interfaces take within a DAL2 service interface. The DAL2 VOSI operations are essentially the same for all DAL2 interfaces.

All the VOSI service operations are available only as synchronous operations. The content of the data returned is determined by the VOSI standard, however the service URL used to invoke each service operation is defined by the individual service specification, in this case TAP.

A compliant TAP implementation **must** implement all VOSI operations as described below.

3.5.1 Get Service Capabilities

The *getCapabilities* operation is used to query the capabilities of an individual TAP service implementation, including:

- Which service interface versions are supported by the service.
- The optional capabilities implemented by the service.
- Any extensions to the service interface.
- Any limitations imposed by the service implementation.

The service Capabilities description describes only the service capabilities and does not include general registry Resource metadata, which is described elsewhere, e.g., when the service is registered.

The following URL may be used to invoke the *getCapabilities* operation:

```
$baseURL/sync/REQUEST=getCapabilities
```

The returned output is a XML document containing a registry compliant *Capabilities* element describing the service capabilities. The content of the TAP service capabilties are TBD.

3.5.2 Get Service Availability

The *getAvailability* operation is used to query and monitor the availability and status of the service, including:

- Service uptime.
- Any scheduled outages or planned downtime.
- The last time any aspect of the service (such as its capabilities) was modified.

The following URL may be used to invoke the *getAvailability* operation:

```
$baseURL/sync/REQUEST=getAvailability
```

The returned output is a XML document containing a VOSI compliant description of the service status and availability. The content of the TAP service availability description are TBD.

3.5.3 Get Table Metadata

The *getTableMetadata* operation is used to query the *tableset* metadata for the TAP service, i.e., the tables supported by the service, and the columns of each table. For TAP this is not an explicit service operation, but rather is a special case of a table metadata request.

The following URL may be used to invoke the *getTableMetadata* operation:

```
$baseURL/sync/REQUEST=ParamQuery&
   FROM=TAP_SCHEMA.tableset&FORMAT=xml
```

The returned output is a XML document containing a registry compliant description of the tables available via the TAP service. The content of the TAP tableset metadata document are TBD.

4 Common Query Elements

4.1 Table Names

A fully qualified table name has the form

[[*catalog_name*"."[*schema_name*"."]*table_name*]]

where *catalog_name* is the “catalog” name (often the “database” name) in SQL DBMS terminology, *schema_name* is the “schema” name in DBMS terminology (often also called a “database”; a DBMS schema is a type of data model where the top level data model elements are tables), and *table_name* is the actual table name. All elements of the table name are optional except *table_name*. Depending upon the DBMS, “catalog” or “schema” may or may not be implemented; some DBMS implement both, others one or the other, and the simplest database systems might not implement either.

Table names originate in the TAP service in a metadata query and should be passed back to the TAP service unchanged by the client. It is up to the service whether or not catalog and schema names need to be included to fully qualify a given table. Case is significant in table names.

The full table name may have any of these combinations:

- Just the base table name
- Catalog name and base table name (with “..” between the two)
- Schema name and base table name (with “.” between the two)
- Catalog name, schema name, and base table name

The names are revealed to clients and users through the TAP metadata query and the VOSI *getTableMetadata* interface. Where a service provides both of these interfaces, it must expose the same names in each.

4.2 Table Uploads

TAP currently supports two methods by which a client application can upload table or other data for use in a query. The simplest approach for tables which are Web-accessible is use of the UPLOAD parameter (3.1.1.4) to reference an external table by URI. More flexible for dynamic client queries is the inline table upload where the table is uploaded inline as part of the query.

In both cases uploaded tables share the TAP_UPLOAD schema, and should be referred to in queries as “TAP_UPLOAD.<*tablename*>”, where the *tablename* is specified by the client at upload time, and must be a legal ADQL table name. Tables are uploaded in VOTable format. Tables in the TAP_UPLOAD schema persist only for the lifetime of the query (although caching might be used behind the scenes).

Uploading a table at query time using the UPLOAD parameter is straightforward so long as the table has already been made Web-accessible. For example, a table could be placed in a publically readable VOSpace, and the VOSpace URI of the table could be used with UPLOAD to reference the table in a query.

In the case of the inline table upload a table is uploaded inline as part of the query, used within the query like any other table, then discarded once the query completes. A typical example would be a multi-position query where the user uploads a list of source positions.

To upload a table inline the POST form of the query must be used. The content type used is "`multipart/form-data`", using a "`file`" type input element, with the "`name`" attribute specifying the table name.

So for example in the POST data (following the header and input parameters) we might have:

```
Content-Type: multipart/form-data; boundary=AaB03
    [...]

--AaB03x
Content-disposition: form-data; name="table1"; filename="table1.xml"
Content-type: application/x-votable+xml
    [...]

--AaB03x
Content-disposition: form-data; name="region"; filename="region.xml"
Content-type: application/x-stc+xml
```

The upload table would automatically propagate and could be referenced in either ADQL or param queries as "`TAP_UPLOAD.table1`". In the above example a STC region mask is also being uploaded.

Inline table uploads may be used both with standard Web forms in a browser, as well as for programmatic input.

Any number of tables can be uploaded using this technique, so long as they are assigned unique table names within the query. Although our discussion here concerns uploading tables, any type of file can be uploaded in this fashion provided the service can do something useful with the file.

4.3 VOSpace Integration

This version of TAP provides limited VOSpace integration, although better support for VOSpace is planned for a later version following prototyping. Ultimately one would like to have per-user VOSpace storage co-located with the TAP service, allowing user queries to save output tables to the local VOSpace as well as use them for input in subequent queries, without having to serialize to and from VOSpace and transfer tables over the network. Frequently used tables such as source lists for multi-position queries could persist between queries, and could be arbitrarily large.

The current version of TAP does provide limited VOSpace integration via the table UPLOAD parameter, using the upload URI to point to a table stored in either a local or remote VOSpace.

4.4 Query Response

The response to a successful table query (using either AdqlQuery or ParamQuery) is a table. By default tables are returned to the client in VOTable format, although other output formats are possible, as described in the next section.

The output VOTable should be compliant with VOTable V1.1 or greater [*ref*] and for a synchronous query should be returned with a MIME type of:

```
text/xml;content=x-votable
```

A base MIME-type of `text/xml` is used for synchronous queries to enable display of query results in browsers using direct rendering of the XML or an optional style sheet. VOTables which are manipulated as file data should instead use the MIME type `application/x-votable+xml`.

The VOTable **must** contain a RESOURCE element, identified with the tag type = "results", containing a single TABLE element with the results of the query. Additional RESOURCE elements may be present, but the usage of any such elements is not defined here.

The RESOURCE element **must** contain an INFO with name="QUERY_STATUS". Its value attribute should be set to ”OK” if the query executed successfully, regardless of whether any matching data were found. All other possible values for the value attribute are described in section 7.10.

Examples:

```
<INFO name="QUERY_STATUS" value="OK"/>
<INFO name="QUERY_STATUS" value="OK">Successful query</INFO>
```

Additional INFOs may be provided, e.g., to echo the input parameters back to the client in the query response (a useful feature for debugging or to self-document the query response), however this is not required.

Where possible output table columns **should** be assigned UCDs (uniform content descriptors) to indicate the type of quantity stored in the column. If the table contains a data model columns **may** also be assigned UTYPEs, and may be aggregated with the VOTable GROUP construct to identify a subset of table columns as a data model instance.

4.5 Output Formats

In the case of TAP, all regular (non tableset) table data or metadata queries produce a single table as a result. Output table data may be rendered and returned to the client in any format supported by the service, VOTable being the default and the only output format for which support is mandatory.

A TAP service **must** support VOTable as the default output format, and **should** support at least comma separated values format (CSV) as well. The service **may** also provide pretty printed text, HTML, or FITS binary table as output formats, or any additional custom format defined by the service. The output formats supported by the service should be defined in the service metadata (3.5.1), along with their MIME types and FORMAT parameter short names, if defined.

CSV formatted data should represent the output table with one row of text per table row, with the table column values rendered as text and separated by commas. If a column value contains a comma the entire column value should be enclosed in double quotes. Text lines may be arbitrarily long. The first data row should give the column name as the data value. Header lines may optionally be included in the first few lines of output, prior to the first data row, and should be indicated by placing the character '#' in the first character of the line.

Tableset metadata (see section 3.3.3) does not constitute normal table output and can be returned only in either dataless VOTable or registry compliant XML format.

5 TAP Schema

5.1 TAP Core Schema

The TAP core schema is intended to define the minimal metadata required to describe and use the tables exposed by a TAP service. The TAP core schema is equivalent to that defined by the registry for a *VODataService* *[this is the goal but both are still being revised]*. VODataService is in turn modeled after VOTable.

The table "`TAP_SCHEMA.schemas`" contains the following columns:

schema_name	fully qualified schema name (*catalog.schema*)
description	brief description of schema
utype	UTYPE if schema corresponds to a data model

The table "`TAP_SCHEMA.tables`" contains the following columns:

schema_name	fully qualified schema name (*catalog.schema*)
table_name	fully qualified table name (*catalog.schema.table*)

table_type	one of: base_table, view, output
description	brief description of table
utype	UTYPE if table corresponds to a data model

The table "`TAP_SCHEMA.columns`" contains the following columns:

column_name	column name
table_name	fully qualified table name (*catalog.schema.table)*
description	brief description of column
unit	unit in VO standard format
ucd	UCD of column if any
utype	UTYPE of column if any
datatype	datatype as in VOTable/Registry
arraysize	array dimensions as in VOTable/Registry
primary	column is visible in default selection
indexed	column is indexed on the server
std	standard column (as opposed to custom)

The TAP schema is extensible, and additional schema columns or tables may be defined by an individual service.

The *table_name* should include any catalog or schema names if these are used to reference tables by the server. This should include TAP_SCHEMA for any tables part of the TAP schema itself. The schema TAP_UPLOAD should be included in the table name for any tables located in the table upload area. The TAP_SCHEMA may be queried for tables named "`TAP_SCHEMA.*`" to get information about the schema itself, e.g., to determine if any extended schema metadata is defined by the service.

The schema naming conventions used here follow that of the registry. Data types are expressed as in VOTable and the registry, e.g., boolean, unsignedByte, short, int, float, double, and so forth. "Arraysize" specifies the dimensions of an array, e.g., "*", "5", "5x20" etc. "Primary" indicates that the column should be visible in the default (narrow) view of a table. "Indexed" indicates that the column is indexed, potentially making queries run much faster if this column is used as a constraint. "Std" is included for compatibility with the registry, which uses this value to indicate that a given column is defined by some standard, as opposed to a custom column defined by a particular service.

5.2 Table Sets

The TAP schema also defines `TAP_SCHEMA.tableset`, however this is not an actual table but rather a structured view of the core schema tables above. A simple query will return the entire tableset, but advanced services may permit selection with a WHERE clause, e.g., to find only tables within a given region or

for which the tablename matches some pattern. The tableset contains a sequence of table entries with each entry listing the columns of that table. XML and VOTable output formats are defined. For XML table metadata is formatted as defined for a VODataService. For VOTable a VOTable is returned which contains a sequence of "empty" TABLE entries containing only metadata (FIELD and PARAM definitions) and no table data.

To return the full tableset supported by the service in VOTable format:

```
FROM=TAP_SCHEMA.tableset&FORMAT=votable
```

To return the same metadata in registry compliant XML format the same command would be used, with FORMAT specified as “xml”.

6 Service Registration

Publication of a service to the VO requires that it be *registered* with the VO registry, including describing the identity and capabilities of the service.

The registration metadata for a TAP service instance is structured according to *VOResource* 1.0 [*ref*] and is assigned the sub-type *CatalogService* as defined in *VODataService* 1.1 [*ref*].

Each of the primary *capabilities* (service interfaces or versions) provided by the service is expressed as a *capability* element in the service registration. These elements are distinguished by their *standardID* attributes. The *standardID* attributes for TAP are TBD.

Where the database-schema of the archive is fixed or slowly-changing, the service registration should detail this schema using elements drawn from VODataService 1.1. Where the schema changes frequently, these metadata should not be included in the registration.

7 Basic Service Elements

7.1 Introduction

This section specifies the basic form of the service interface, including the form of a service request and its response. These characteristics are common to all service operations, and are largely common to all the second generation DAL protocols.

7.2 Version numbering and negotiation

7.2.1 Version number form and value

The TAP protocol defines a protocol version number. The version number applies to all aspects of the protocol as defined in this document, including any associated XML schema and the request encodings. The TAP version refers only to the TAP protocol; ADQL is versioned separately and TAP and ADQL versions may differ.

Version numbers follow IVOA document conventions and contains two non-negative integers, separated by decimal points, in the form *"x.y",* for example, "1.0", or "1.13". This is actually a *three* level version number encoded as two digits, e.g., "1.23" is logically the same as "1.2.3". One result of this syntax is that second level version numbers cannot be greater than 9, for example "1.9" is a *higher* version number than "1.10" (logically "1.9.0 vs. "1.1.0"). Hence IVOA version numbers cannot be numerically compared without first being parsed.

7.2.2 Version number changes

The protocol version number will change with each published revision of this document. The number will increase monotonically and will comprise no more than two integers separated by decimal points, with the first integer being the most significant. There may be gaps in the numerical sequence. Some numbers may denote draft versions. Servers and their clients need not support all defined versions, but shall obey the negotiation rules below.

A version number change at the first level (e.g., 1.0 – 2.0) indicates a major change. A version number change at the second level indicates a minor change which is not necessarily backwards compatible. A version number change at the third level is considered backwards compatible, and should not affect the pre-existing functionality of the interface.

7.2.3 Appearance in requests and in service metadata

The version number may appear in at least three places: in the service metadata, as a parameter in client requests to a server, and in the query response. The version number used in a client's request of a particular server must be equal to a version number which that server has declared it supports (except during negotiation, as described below). A server may support several versions, whose values clients may discover according to the negotiation rules.

7.2.4 Version number negotiation

If a TAP client does not specify the version number in a request, the server assumes the highest *standard* version supported by the service, and no explicit version checking takes place. If the client specifies an explicit version number, and this does not match a version available from the service at level two, the service returns a version number mismatch error. The client can determine what

versions of the protocol the service supports by a prior call to getCapabilities or via a registry query.

7.3 General HTTP request rules

7.3.1 Introduction

This document defines the implementation of the TAP service on a distributed computing platform (DCP) comprising Internet hosts that support the Hypertext Transfer Protocol (HTTP) (see IETF RFC 2616). Thus, the Online Resource of each operation supported by a server is an HTTP Uniform Resource Locator (URL). The URL may be different for each operation, or the same, at the discretion of the service provider. Each URL shall conform to the description in IETF RFC 2616 (section **Error! Reference source not found.** "HTTP URL") but is otherwise implementation-dependent; only the query portion comprising the service request itself is defined by this document.

While the TAP protocol currently only supports HTTP as the DCP for general parameterized operations, data access references are more general and may use other internet protocols, e.g., FTP, or potentially grid protocols.

HTTP supports two primary request methods: GET and POST. One or both of these methods may be offered by a server, and the use of the Online Resource URL differs in each case. Support for the GET method is mandatory; support for the POST method is optional except where required for a service operation to function, e.g., uploading a large quantity of data inline in a query, or when issuing a request to the service which changes the server state.

7.3.2 Reserved characters in HTTP GET URLs

The URL specification (IETF RFC 2396) reserves particular characters as significant and requires that these be escaped when they might conflict with their defined usage. This document explicitly reserves several of those characters for use in the query portion of TAP requests. When the characters "?", "&", "=", "," (comma), "/", and ";" appear in one of the roles defined in Table 1, they shall appear literally in the URL. When those characters appear elsewhere (for example, in the value of a parameter), they should be encoded as defined in IETF RFC 2396. The server shall be prepared to decode any character escaped in this manner.

Table 1 — Reserved characters in TAP query string

Character	Reserved usage
?	Separator indicating start of query string.
&	Separator between parameters in query string.
=	Separator between name and value of parameter.

,/;	Separator between individual values in list-oriented parameters (such as POS, BAND, TIME, etc.).

In particular, if any parameter value contains the character “#” (for example in a dataset identifier) it must be URL encoded to be legally included in a URL.

7.3.3 HTTP GET

A TAP service shall support the “GET” method of the HTTP protocol (IETF RFC 2616).

An Online Resource URL intended for HTTP GET requests is in fact only a URL prefix to which additional parameters are appended in order to construct a valid Operation request. A URL prefix is defined in accordance with IETF RFC 2396 as a string including, in order, the scheme (“http” or “https”), Internet Protocol hostname or numeric address, optional port number, path, mandatory question mark “?”, and optional string comprising one or more server-specific parameters ending in an ampersand “&”. The prefix defines the network address to which request messages are to be sent for a particular operation on a particular server. Each operation may have a different prefix. Each prefix is entirely at the discretion of the service provider.

This document defines how to construct a query part that is appended to the URL prefix in order to form a complete request message. Every TAP operation has several mandatory or optional request parameters. Each parameter has a defined name . Each parameter may have one or more legal values, which are either defined by this document or are selected by the client based on service metadata. To formulate the query part of the URL, a client shall append the mandatory request parameters, and any desired optional parameters, as name/value pairs in the form “name=value&” (parameter name, equals sign, parameter value, ampersand). The “&” is a separator between name/value pairs, and is therefore optional after the last pair in the request string.

When the HTTP GET method is used, the client-constructed query part is appended to the URL prefix defined by the server, and the resulting complete URL is invoked as defined by HTTP (IETF RFC 2616).

Table 2 summarizes the components of an operation request URL when HTTP GET is used.

Table 2 — Structure of TAP request using HTTP GET

URL component	Description
http://host:port]/path[?[name[=value]{&name=[value]}]]	Base-URL (prefix) of service operation. [] denotes 0 or 1 occurrence of an optional part; {} denotes 0 or more occurences.

name=value&	One or more standard request parameter name/value pairs as defined for each operation by this document.

7.3.4 HTTP POST

TAP uses he “POST” method of the HTTP protocol (IETF RFC 2616) whenever a large amount of data needs to be uploaded inline in the query, e.g., when uploading an inline table, or whenever the request may change the server state, e.g., when requesting asynchronous execution of a query. Semantically POST and GET are largely the same, permitting the same parameters to be transmitted to the server to define the request. Parameters should be URL encoded in a POST whenever they would need to be URL encoded for a GET.

7.4 General HTTP response rules

Upon receiving a valid request, the server shall send a response corresponding exactly to the request as detailed in section 3 of this document, or send a service exception if unable to respond correctly. Only in the case of Version Negotiation (see 7.2.4) may the server offer a differing result. Upon receiving an invalid request, the server shall issue a service exception as described in 7.10.

A server may send an HTTP Redirect message (using HTTP response codes as defined in IETF RFC 2616) to an absolute URL that is different from the valid request URL that was sent by the client. HTTP Redirect causes the client to issue a new HTTP request for the new URL. Several redirects could in theory occur. Practically speaking, the redirect sequence ends when the server responds with a valid TAP response. The final response shall be a TAP response that corresponds exactly to the original request (or a service exception).

Response objects shall be accompanied by the appropriate Multipurpose Internet Mail Extensions (MIME) type (IETF RFC 2045) for that object. A list of MIME types in common use on the internet is maintained by the Internet Assigned Numbers Authority (IANA) . Allowable types for operation responses and service exceptions are discussed below. The basic structure of a MIME type is a string of the form “type/subtype”. MIME allows additional parameters in a string of the form “type/subtype; param1=value1; param2=value2”. A server may include parameterized MIME types in its list of supported output formats. In addition to any parameterized variants, the server should offer the basic unparameterized version of the format.

Response objects should be accompanied by other HTTP entity headers as appropriate and to the extent possible. In particular, the Expires and Last-Modified headers provide important information for caching; Content-Length may be used by clients to know when data transmission is complete and to efficiently

allocate space for results, and Content-Encoding or Content-Transfer-Encoding may be necessary for proper interpretation of the results.

7.5 Numeric and boolean values

Integer numbers shall be represented in a manner consistent with the specification for integers in XML Schema Datatypes. This document shall explicitly indicate where an integer value is mandatory. Real numbers shall be represented in a manner consistent with the specification for double-precision numbers in XML Schema Datatypes. This representation allows for integer, decimal and exponential notations. A real value is allowed in all numeric fields defined by this document unless the value is explicitly restricted to integer.

Sexagesimal formatting is generally not permitted other than in ISO 8601 formatted time strings unless otherwise specified in this document. For TAP an exception is made for queries of data tables where the native table formatting is normally preserved.

Positive, negative and zero values are allowed unless explicitly restricted.

Boolean values shall be represented in a manner consistent with the specification for Boolean in XML Schema Datatypes. The values “0” and “false” are equivalent. The values “1” and “true” are equivalent. Absence of an optional value is equivalent to logical false. This document shall explicitly indicate where a Boolean value is mandatory.

7.6 Output formats

The response to a TAP request is always a computer file that is transferred over the Internet from the server to the client. The file may contain text, or the file may be a graphics or FITS-formatted file . As stated in 4.5, the type of the returned file shall be indicated by a MIME type string.

Text output formats are usually formatted as Extensible Markup Language (XML; MIME type text/xml). Text formats are used to convey service metadata, descriptions of error conditions, or responses to data queries. In particular, the response to a data query is always returned as an XML file in VOTable format.

7.7 Request parameter rules

7.7.1 Parameter ordering and case

Parameter names shall not be case sensitive, but parameter values shall be. In this document, parameter names are typically shown in uppercase for typographical clarity, not as a requirement.

Parameters in a request may be specified in any order.

When request parameters are duplicated with conflicting values, the response from the server may be undefined. This document does not mandate which of the duplicated values sent by the client are to be used by the server. It is recommended that neither the client nor the service should repeat parameter values in a query URL.

A TAP service shall be prepared to encounter additional request parameters that are not part of this document without reporting an error. In terms of producing results per this document, a TAP service shall not require such parameters, but may define additional service-defined parameters.

7.7.2 Range-list parameters

Parameters which are *list-valued* (for example, UPLOAD and POS) use the comma (",") as the separator between successive items in the list. Embedded white space is not permitted. If a parameter value includes a space or comma, it must be escaped using the URL encoding rules (see 7.3.2 and IETF RFC 2396).

In some lists, individual entries may be empty, and should be represented by the empty string. Thus, two successive commas indicate an empty item, as does a leading comma or a trailing comma. An empty list should be interpreted either as a list containing no items, or as a list containing a single empty item, depending upon the context.

Some parameters (for example MTIME and WHERE) may allow a parameter value to be specified as a numeric range. Such *range-valued* parameters use the forward slash ("/") character as the separator between elements of the range specification (as in the ISO 8601 date specification after which this convention is patterned). For example, "5E-7/8E-7" would specify a range consisting of all values from 5E-7 to 8E-7, inclusive. If a third field is specified it is a step size for traversing the indicated range. If a parameter permits a step size the semantics of the step size are defined by the specific parameter.

An open range may be specified by omitting either range value. If the first value is omitted the range is open toward *lower* values. If the second value is omitted the range is open toward *higher* values. Omitting both values indicates an infinite range which accepts all values. For example, "/5" is an open range which accepts all values less than or equal to 5. To specify all values less than 5, "/4" would be used (for an integer valued range). Range values are limited to numeric values or ISO dates.

If specified by the definition of a particular parameter a list may be qualified by appending the character ";" (semicolon) followed by a qualifier string. For example "180.0,1.0;galactic" would specify a position in galactic coordinates. In

some cases (e.g., UPLOAD; the ParamQuery WHERE), multiple semicolons may be used to delimit separate sub-lists or clauses within the parameter value.

List and range syntax may be combined, e.g., to indicate a list of scalar or range-valued parameter values. Such a *range list* may be ordered or unordered, and may contain either numeric or string data. An *ordered* list is one which requires values to be processed in a specified order, and to ensure this the range list is sorted or ordered *by the service* as necessary before being used. It is the responsibility of the service to sort an ordered range list, hence the client can input ranges or range values in any order for an ordered range list and the result will be the same. The sequence in which items in an *unordered* list occur on the other hand is significant, as since there is no intrinsic ordering for the list which can be enforced by the service, items will be processed by the service in the order they are input by the client.

The ParamQuery SELECT parameter is an example of an *unordered* list, that is, a list which does not have a specified order mandated by the service (hence in this case the client determines the order in which table fields will be output).

7.7.3 Missing or null-valued parameters

If a parameter is not included in a query its value is *unset*; no value has been specified. If a parameter is given a null value, e.g., “POS=”, the parameter value has been set and the value is the null string. The interpretation of such an input is defined separately for each parameter, and may or may not be an error condition.

7.8 Common request parameters

7.8.1 VERSION

The VERSION parameter specifies the protocol version number. The format of the version number, and version negotiation, are described in 7.2.

7.8.2 REQUEST

The REQUEST parameter indicates which service operation is being invoked. The value shall be the name of one of the operations offered by the server. It is an error to reference an unknown service operation. The service operation to be executed must be explicitly specified in every request or it is an error.

7.8.3 Extended capabilities and operations

The TAP service allows for optional extended capabilities and operations. Extensions may be defined within an information community when needed for additional functionality or specialization. A generic client shall not be required or expected to make use of such extensions. Extended capabilities or operations shall be defined by the service metadata. Extended capabilities provide additional metadata about the service, and may or may not enable optional new

parameters to be included in operation requests. Extended operations may allow additional operations to be defined.

A server shall produce a valid response to the operations defined in this document, even if parameters used by extended capabilities are missing or malformed (*i*.e. the server shall supply a default value for any extended capabilities it defines), or if parameters are supplied that are not known to the server.

Service providers shall choose extension names with care to avoid conflicting with standard metadata fields, parameters and operations.

7.9 Service result

The return value of a valid Service request shall correspond to the output type specified for the operation, or requested in the FORMAT parameter in the case of an operation which can return data in a choice of output formats. In an HTTP environment, the Content-type header of the response shall be exactly the MIME type associated with the valid request.

7.10 Error Response and Other Exceptional Results

Upon receiving a request that is invalid according to this document, the server shall issue a service exception report. The service exception report is meant to describe to the client application or its human user the reason(s) that the request is invalid. The allowed service exception formats are defined below.

If a service operation throws an error response and exits, the default action of the service should be to return a VOTable noting that an error has occurred, and describing the error. An `INFO` element within the "results" `RESOURCE` element of the VOTable is used to indicate success or failure of the operation. As described in the previous section, the `INFO` element must have `name="QUERY_STATUS"`; if the operation is successful (regardless of whether any data is returned) the value attribute is set to `"OK"`. The remainder of this section defines other possible values to indicate that the query was unsuccessful in some way. When the query is unsuccessful, the contents of `INFO` element (i.e. its `PCDATA` child node) **should** contain an error message suitable for display.

If an error response VOTable is returned the MIME type of the response **must** be "`text/xml;content=x-votable;status=error`" to indicate (without having to read the VOTable content) that an error occurred.

When the query is unsuccessful (in any of the senses described below), the resulting VOTable is not required to contain any other elements as specified for a successful operation; however, it is not an error to do so. For example,

additional INFO elements may be returned to echo back the input parameters of the operation which failed, as in the following example.

Example:

```
<VOTABLE … version=”1.1”>
  <RESOURCE type="results">
    <INFO name="QUERY_STATUS" value="ERROR">unrecognized operation</INFO>
    <INFO name="SERVICE_PROTOCOL" value="1.0">TAP</INFO>
    <INFO name="REQUEST" value="queryData"/>
    <INFO name="baseUrl" value="http://webtest.aoc.nrao.edu/ivoa-dal"/>
    <INFO name="serviceVersion" value="1.0"/>
    <INFO name="serviceName" value="tap"/>
    <INFO name="ServiceEngine" value="tap: TAP 1.0 DALServer version 0.4"/>
  </RESOURCE>
</VOTABLE>
```

IThe other allowed values for the value attribute besides `"OK"` are as specified below.

7.10.1 Service Error

The server failed to process the operation. Typical reasons include:

- The input query contained a syntax error.
- The way the query was posed was invalid for some reason, e.g., due to an invalid query specification.
- A constraint parameter value was given an illegal value; e.g. `DEC=91`.
- The server trapped an internal error (e.g., failed to connect to its database) preventing further processing.

In this case a descriptive error message **should** be included in the query status `INFO`.

Example:

```
    <INFO name="QUERY_STATUS" value="ERROR">DEC out of range: DEC=91</INFO>
```

7.10.2 Output Overflow

The operation produced results that exceeded the maximum output in effect for the query. For instance, a data query exceeded the maximum number of output records defined for the session or operation. In this case, the operation was successful (overflow is not an error condition), and the service **must** return a valid response with valid data records, but with `QUERY_STATUS` set to “`OVERFLOW`” instead of “`OK`” (see also section 3.1.1.5).

Example:

```
    <INFO name="QUERY_STATUS" value="OVERFLOW">Number of table rows exceeds
```

```
    default limit of 5000</INFO>
```

If overflow occurs valid data is returned and the client may either choose to ignore the overflow and use the data returned, or may repeat the query, requesting a higher MAXREC value than the default, up to the hard limit defined by the service capabilities. Alternatively the query parameters may be adjusted to more carefully constrain the query. Currently these are the only ways to avoid overflow when performing a query.

Since an output overflow is not an error condition, the MIME type of the output VOTable should be the same as for any successful query.

7.10.3 Other Errors

Although the intention is that service should catch all errors and return a uniform error response in the prescribed VOTable format, informing the client of the nature of the error which occurred in service-specific terms, this is not always possible. More fundamental errors may result in a HTTP level error. The client should be prepared to handle either form of error. Which is returned in a given case, may depend upon the operation performed, the nature of the error which occurred, and the details of how a given service is implemented.

Appendix A: “Appendix Title”

Insert appendix here

References

[1] R. Hanisch, *Resource Metadata for the Virtual Observatory* , http://www.ivoa.net/Documents/latest/RM.html

[2] R. Hanisch, M. Dolensky, M. Leoni, *Document Standards Management: Guidelines and Procedure* , http://www.ivoa.net/Documents/latest/DocStdProc.html