International

Virtual

Observatory

Alliance

Table Access Protocol

Version 0.02

IVOA Working Draft 2008 May 11

This version:
ThisVersion-YYYYMMDD
Latest version:
http://www.ivoa.net/Documents/latest/latest-version-name
Previous version(s):

Author(s):
Author1
Author2
…

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 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 parameterized queries (TAP/Param) but also supports ADQL-based queries within an integrated interface, and includes support for asynchronous queries and VOSpace.

Status of This Document

This is a working draft. This is not yet a complete specification and may contain discussion and comments intended for use within the design team and working group to advance the TAP interface design. Some aspects of the interface are not yet fully detailed and may be discussed here only in general terms.

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) 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 actual table data. Both simple filtering operations on individual tables as well as more general multi-table operations such as relational joins are supported. A parameter-based query operation is provided for simple table data and metadata queries of single tables, as well as a more general SQL-based query operation for general database queries. SQL queries are based upon the Astronomical Data Query Language (ADQL), to provide a uniform query interface regardless of the database management system (DBMS) used to implement the service, while providing additional astronomy-specific functionality such as for spatial region based queries. Simple Web-friendly synchronous queries as well as Grid-enabled capabilities for large asynchronous queries are both supported.

1.1 Basic Usage

Provide some examples of simple synchronous GET-based table data and metadata queries, using both *ParamQuery* as well as a simple SELECT using *AdqlQuery*.

1.2 Requirements for Compliance

Define the minimum requirements for a TAP service, e.g., provide *ParamQuery* supporting both table data and metadata queries, with metadata queries supporting the core TAP schema. Define usage of **should**, **must**, **may**.

2 Interface Overview

2.1 Architecture

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

- 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 can be accessed via the same query interface.

- The ***ADQL*** query language [ref], provides an advanced query language capability based upon SQL, enhanced with astronomy specific extensions, e.g., for applying spatial query constraints.

- 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 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, less TAP-specific 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 ***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 however 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 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. AdqlQuery is an optional advanced capability, required for a fully compliant service but optional for a minimal TAP service.

- **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).

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 Basic Service Elements

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 5. In the current section we merely summarize the elements of the basic service interface. [*This section is adapted from the SSA specification with minor changes*.]

2.3.1 Request Format

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

The TAP interface described in this document is based on a distributed computing platform (DCP) comprising Internet hosts that support the Hypertext Transfer Protocol (HTTP). Thus, the online representation of each operation supported by a service is composed as a HTTP Uniform Resource Locator (URL).

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.

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 operations define the following standard parameters:

REQUEST The request or operation name, e.g., "ParamQuery" (mandatory).

VERSION The version number of the interface (optional).

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 5) 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 (2K or so).

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. More information on error responses is given in section 5.

3 TAP Service Operations

3.1 Common Query Elements

The following concepts and notations are common to all TAP queries. In particular, both AdqlQuery and ParamQuery share these common elements.

3.1.1 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 (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 not significant in table names.

TAP defines several special case schemas, each of which may contain any number of tables. "$VOSPACE" refers to the user's VOSpace storage area co-resident with the TAP service. "$UPLOAD" refers to a space containing any tables uploaded inline in the current query. "$TAP_SCHEMA" refers to the TAP information schema, used to describe database and table metadata. All of these are discussed further in the sections which follow.

3.1.2 Table Field Names

In general table field names are defined by the table and its representation within the underlying DBMS. Table field names must comply with any additional limitations as defined by ADQL [*possibly we should be more specific about this*]. Table field names originate in the TAP service in a metadata query and should not be modified by the client, which should pass them back to the service unchanged when queries are submitted.

TAP further introduces the concept of *name spaces* when specifying table names. By default an unqualified field name (no name space specified) refers explicitly to a physical table field name, such as would be returned in a table metadata query.

A field name name space is indicated by a "*name_space*:" prefix in the table name, with the referenced name space being the name of a data model or other well defined name space supported by the service [*support for data models is an optional advanced capability and would be indicated by the service capability metadata*]. For example, "`src:ra`", might refer to the field "ra" of the *source* catalog data model. The entire field reference constitutes a UTYPE referencing a field or other element of the referenced data model. The physical field name could be anything, so long as it does not conflict with the use of name spaces.

A special case of a name space is "`ucd:`", which refers to the UCD name space consisting of all defined unified content descriptors (UCDs). In other words, UCD is a special case of a data model, which in this case attempts to describe the individual physical attributes of all astronomical data.

In the most general case a table might support a data model providing UTYPE references for some subset of the table fields, plus UCDs for most or all physical table fields, plus define the physical field names of the table fields. In such a case any of these names could be used to refer to a given table field. In all cases the field name reference resolves to a single physical field of the table. This is called ***field name resolution***. Multiple name spaces may be mixed within a single query. A direct reference to a table field by its physical table field name is always permitted, and is the most unambiguous form of reference.

3.1.3 Inline Table Uploads

TAP supports two methods by which a client application can upload table or other data for use in a query. The simplest is an ***inline*** table upload, where the data is uploaded inline as part of the query, used within the query like any other table, then discarded once the query completes (the second is via ***vospace***, which is discussed in section 3.1.4). Inline uploads are simplest for small tables, but do not provide persistence and are not practical for very large tables. An example of an inline table upload is a multi-position query, where a table containing multiple spatial positions is uploaded by the client to perform a "multi cone search" query.

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. Within the query the uploaded table can be referred to as "$UPLOAD.*name*" (the name is case insensitive). Tables must be uploaded in VOTable format, but can be used within TAP queries like any other database table.

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 (TAP per se specifies handling only for table data).

See Appendix XX for a more detailed example of the use of the table upload capability.

3.1.4 VOSpace Usage

VOSpace [ref XX] provides network data storage on a per-user basis, as well as facilities for transfer of data between VOSpaces or between a VOSpace and the client application. TAP services which implement the (optional) VOSpace capability integrate the VOSpace directly into the TAP service, allowing tables stored in the VOSpace to be used efficiently in TAP queries. Tables may be uploaded by the client to the VOSpace and subsequently used in a query, may be output to the VOSpace and used in a subsequent query without having to transport the table to and from the client, may be output to the VOSpace by an asynchronous query and later retrieved by the client, or used as input to a remote service for further processing.

Assuming that a client application or user-specific VOSpace storage area has been identified, tables stored within the VOSpace may be used within any TAP query by referencing them as "$VOSPACE.*name*", where *name* is the table name within the user's TAP-resident VOSpace (any input tables must be somehow written to the TAP-resident VOSpace before they are used in a query).

There are three main scenarios involving user (client) data tables used within or produced by TAP queries:

- The simplest case is an inline table upload, as discussed in the previous section (3.1.3). VOSpace is not required or used, and there are no issues with data transport or authentication.

- The second simplest case is an asynchronous query which writes its output table to a VOSpace on behalf of the user or client application. In this case it is possible for the TAP service to assign an anonymous VOSpace storage area on behalf of the "user" (client application instance), and return a data access URL pointing to where this table will be available once the query completes. Storage allocated in this fashion can be purged after a time interval defined by the service, e.g., several days. A table name can also be assigned to these temporary tables to allow their use in subsequent queries.

- In the most general case the client application transfers tables directly to and from their TAP-resident VOSpace, using the VOSpace provided service interface [*SOAP currently; a GET interface is planned for VOSpace 2.0*]. Since in this case the client interacts directly with the VOSpace for data management and transfer, there needs to be some way to identify the VOSpace to be used to the TAP service when a query is performed. Either the user can unambiguously authenticate with both the VOSpace and the TAP service, or an anonymous VOSpace storage area can be created by the VOSpace on behalf of the client application or user. In the latter case a container can be created within the TAP VOSpace for client data storage, and the container ID can be used to uniquely identify the storage area in a subsequent TAP query [*exactly how to do all this is still TBD*].

In summary, in the most general case the user will simply sign on with a login and password, using SSO authentication, transfer any tables to or from their TAP-resident VOSpace, and execute one or more queries via TAP which reference these tables.

If a general persistent VOSpace is used, it is up to the client application to transfer tables to and from the VOSpace, or delete such tables when they are no longer needed. A TAP service which supports the VOSpace capability deals only with its local VOSpace. Tables are transferred over the Internet in VOTable format, but may be referenced within TAP queries as ordinary DBMS tables.

Since VOSpace is a defined (logical) schema within the TAP service, all TAP semantics defined apply to VOSpace resident tables just as they do to ordinary data tables. In particular, table queries may read and write tables to the local

VOSpace, and table metadata queries may be used to list the tables in the local user VOSpace, or list the columns of any such tables.

Although TAP refers to the local use VOSpace as a DBMS schema, how it is implemented is up to the service. Typically the local TAP VOSpace would be implemented as either a DBMS catalog or schema, and would store table data for all users of the TAP service. The TAP interface however always makes the VOSpace appear as a special schema ($VOSPACE), and allows the user to see only their tables within the VOSpace. [*An advanced issue is how to reference tables from another user's VOSpace within a TAP query*].

3.1.5 Asynchronous Execution

All TAP table queries (both AdqlQuery and ParamQuery) can execute either synchronously or asynchronously. Synchronous execution is the default; asynchrnous execution is an optional advanced capability for a TAP service. Synchronous execution is simpler and is adequate for many queries, in particular "small" queries, or filter-type queries of a single table where the output is not sorted or ordered and can be streamed to the client (as is typically the case for astronomical catalogs).

Asynchronous execution is requested by the client by specifying an output table name, with the new table to be placed in their per-user VOSpace storage area (be it authenticated or anonymous). The query response for an asynchronous query is synchronous, and consists of a status VOTable indicating either a successful or unsuccessful response. If the asynchronous query is successfully initiated a job ID is returned, along with an access reference URL indicating where the output table will be available once the query completes (this points to location within the user's VOSpace).

Monitoring of job execution is performed using the UWS pattern [*this needs to be detailed but can probably be common to any asynchronous operation and not specific to TAP. Either simple polling or messaging could be used to monitor job execution*.]. Once the asynchronous query completes, either the access reference URL or the VOSpace client API can be used to retrieve the output table, or the table can be used as input to a subsequent query.

[*Query estimation, i.e., estimated job run time, is a difficult matter and is not currently addressed. A simple technique is to specify a per-job limit on the run time and leave it to the user to avoid running into this limit.*]

3.1.6 Output Formats

In the case of TAP, all regular (non tableset) table data or metadata queries produce a single table as a result. In the case of synchronous queries, 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. In the case of asynchronous queries, output is always written to the local per-user VOSpace storage area, in a format which is internal to the implementation of the TAP service and VOSpace (normally a native DBMS table). When a remote client subsequently retrieves data from the VOSpace, the default transport format is VOTable.

The suggested supported output formats for synchronous table queries include VOTable (the default), CSV or TSV, pretty-printed text, and HTML (possibly Javascript-enhanced). Other options might include FITS binary table or XML, however these are generally less useful for general table queries.
Aside from the mandatory VOTable, the supported output formats for table data or metadata queries are an optional service capability, and are specified in the service capability metadata. A TAP service **must** support VOTable as the default output format, and **should** support at least CSV/TSV and text as well.

Tableset metadata (see section 3.3.4) does not constitute normal table output and can be returned only in either dataless VOTable or registry compliant XML format.

3.2 AdqlQuery Operation

The TAP service **should** implement the *AdqlQuery* Operation, used to execute a TAP query composed as an ADQL statement. Alternative query languages, e.g., native SQL pass-through, may optionally be supported by the service as an advanced capability.

In the case of AdqlQuery, much of the power of the query is provided by the richness of the ADQL language and the capability of the TAP service to process general ADQL statements input by the client. Specification of the ADQL language is beyond the scope of this document, and is addressed separately in the ADQL specification (ref XX).

As noted earlier, many of the AdqlQuery input parameters, and everything having to do with query execution and the query response, are common to both AdqlQuery and ParamQuery.

3.2.1 Input Parameters

[*A number of these parameters are based on SSA parameters of the same name and are intended to be common elements of the standard DAL service profile*.] [*This part of the specification needs to be made more precise and rigorous once a first round of discussions are completed.*]

3.2.1.1 QUERY

A service which implements the AdqlQuery Operation **must** support this parameter, used to input the ADQL (or other QL) 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 5). The TAP service must be prepared to decode the URL encoded string (any modern service framework will normally do this transparently to the service code).

3.2.1.2 QUERYTYPE

A string specifying the language and optionally 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 queryType. The service **may** support other query language encodings as well, e.g., other ADQL versions, or pass-through of native SQL. The service should return an "unknown queryType" error if an unsupported queryType is specified.

3.2.1.3 FORMAT

The service **must** 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", "tsv", "text", or "html" (case independent). The service should return an "unsupported output format" error if an unsupported output format is requested.

3.2.1.4 MAXREC

The service **should** implement a MAXREC parameter indicating 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).

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. A sufficiently large MAXREC may permit streaming of arbitrarily large tables. Output tables larger than the maximum permitted value of MAXREC must use VOSpace (if supported by the service) for data storage and transfer. [*Support for paging through the output of large queries is another option which could be considered*]

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 (overflow should still be indicated if table data rows were discarded; a

status of OK should be indicated if no table data rows were produced). This is an example of a ***null query***, that is, a query which produces an empty table.

3.2.1.5 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 but should be at least several days.

3.2.1.6 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.

3.2.1.7 OUTPUT

The service **may** support an OUTPUT parameter, used to specify the name of the output table to be generated. If no output table is generated the query is synchronous and the output table is returned as the query response, with a status of OK, OVERFLOW, or ERROR (e.g., if the query times out or otherwise fails). If an output table name is given an asynchronous job is initiated which upon successful completion will save the query result to the named table in the user's VOSpace storage area (see section 3.1.4 and 3.1.5).

If the named output table would overwrite a valid table of the same name the service should return an ERROR status indicating "would overwrite existing table". If for any reason an asynchronous query cannot be initiated, an ERROR response should be returned.

3.2.2 Query Response

If the query is unsuccessful an ERROR response VOTable is returned as described in section 2.3.6. An error condition always results in a VOTable

response regardless of the output format requested by the client (except for low level errors where a HTTP error could occur instead).

If the query is successful the type of response depends upon whether it was a synchronous or asynchronous query. In the case of a ***synchronous query***, the output table is returned as the query response, in the format requested by the client. In the case of a VOTable, the VOTable **must** contain a RESOURCE element, identified with the tag `type="results"`, containing a single TABLE element with the results of the query. The RESOURCE element **must** contain an INFO with `name="QUERY_STATUS"`, with the value attribute set to "OK" if the query executed successfully, regardless of whether any data rows were returned. If the query executed successfully but resulted in overflow, a value of "OVERFLOW" should be returned.

In the case of an ***asynchronous query***, an asynchronous job is initiated to execute the query, and the request returns immediately. A standard status VOTable response **must** be returned containing an INFO element with `name = "QUERY_STATUS"`, with the value attribute set to either "OK" or "ERROR". If the asynchronous query job is successfully initiated an additional INFO is included with `name="JOBID"` with the value set to the job ID of the asynchronous job, which may subsequently be used to monitor job execution (see 3.1.5).

Example:

```
<VOTABLE … version="1.1">
  <RESOURCE type="results">
    <INFO name="QUERY_STATUS" value="OK"/>
    <INFO name="JOBID" value="4316"/>
    <INFO name="TABLE_NAME" value="highz"/>
    <INFO name="ACCESS_URL" value="http://..."/>
  </RESOURCE>
</VOTABLE>
```

In addition, an INFO is included with `name="TABLE_NAME"`, with the value set to the name of the table to be output, plus another INFO with `name = "ACCESS_URL"`, with the value set to the URL to be used to retrieve the output table from the local VOSpace once job execution is completed. In simple cases this allows the client to retrieve data following job completion without having to interact directly with the VOSpace.

3.3 ParamQuery Operation

The service **must** implement the *ParamQuery* operation, used to provide basic access to both table data and metadata. ParamQuery is the standard way to query table metadata with TAP: an advanced service might also be able to query table metadata with AdqlQuery but this is overkill for basic table metadata queries. ParamQuery allows simple filter-type queries of individual tables to be

implemented without requiring ADQL support; most simple queries of astronomical catalogs are of this type. Explicit support is provided for common query cases such as cone search and multi-position queries.

3.3.1 Input Parameters

While the ParamQuery input parameters are defined in detail in the following sections, a few simple examples should help illustrate how such queries are formed. Only the ParamQuery parameters are shown in these examples; the full query URL would include the prefix "*baseURL?*`REQUEST=ParamQuery&`".

The following example would read the 2MASS point source catalog, finding all sources within six arcminutes of the indicated position, where the J band SNR is greater than or equal to 2.5:

```
FROM=fp_psc&POS=180.0,0&SIZE=0.2&WHERE=j_snr,2.5/
```

The following would list all tables in the user's VOSpace at the TAP service:

```
FROM=$TAP_SCHEMA.tables&WHERE=tablename,$vospace.*
```

By default these are synchronous queries, with output returned in the default ouput format (VOTable).

3.3.1.1 POS, SIZE

POS and SIZE define a circular search region in the indicated coordinate system (default ICRS). The service **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 are applied to a table which does not tag records with spatial positions an ERROR should be returned (the client should omit these parameters to query such a table).

The coordinate values for POS are specified in list format (comma separated) with no embedded white space, as defined in section 5.

Example: `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).

Example: `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 would use other coordinate systems, or omit POS entirely). 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 3.3.6.

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

Example: `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.

3.3.1.2 REGION

The service **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 region specifier [*a future capability to consider would be uploading a more general STC/X region mask and referencing this via indirection using REGION*.]

Example: `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 in an actual URL they would have to be URL encoded as "`%20`".

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 3.3.6), where a large table of possible search positions may include positions outside the desired search region.

3.3.1.3 SELECT

The table fields to be returned by the query, specified either as a comma delimited list of field names, or by specifying one of the special values "`$STD`" (to return only the "primary" fields), or "`$ALL`" (to return all table fields). These values are case insensitive.

Example: `SELECT=ra,dec,flux`

By default only the standard or "primary" fields are returned. The "primary" fields are specified on a per-table basis, and form a subset of the table fields including only those fields thought to be most important. This is used to provide a more

readable view of very wide tables. Field name resolution (3.1.2) is performed, hence UTYPE or UCD references may optionally be used to reference table fields, if supported by the service. [*It has been suggested that use of SELECT FROM WHERE as parameter names could be confusing and that other names should be used – this needs further consideration*.]

3.3.1.4 FROM

The table to be queried, specified as defined in section 3.1.1. Only a single table reference is allowed. There is no default, hence the query is not valid unless a table is specified.

Example: `FROM=$VOSPACE.highz`

In addition to the data tables managed by the service, tables in the query upload or user VOSpace area may be referenced, as well as the table metadata tables defined by the TAP information schema.

3.3.1.5 WHERE

An optional filtering constraint to be applied to the table to determine which table rows are returned. By default all table rows are returned.

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 colon would be more readable for this but conflicts with field name resolution*].

A simple example should help illustrate the syntax:

Example: `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 (“/” indicates a numerical range).

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. Field name resolution is applied, hence table field names can be referred to indirectly by UTYPE or UCD if this capability is supported by the service.

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, and consistent with similar usage in other data access services, e.g., in the use of “/”

for numerical ranges. An effort has been made to select 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.

Some details of the syntax:

- 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 exactly, except for case.

- For numeric fields the constraint is either a single numeric value or a range, using "/" as the range delimiter. 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". [*this is a DAL range list as in SSA etc.*].

- For both string and numeric valued fields the constraint can also be a comma delimited list of allowable values with an OR relationship.

- A value expression prefixed with "!" specifies a NOT of whatever follows. Only the entire field value expression can be prefixed in this fashion.

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

- 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 (semicolon, comma, slash, asterisk). A string can also be quoted to force case-sensitive comparison. The single quote is used to avoid conflict with double quote which is often used to quote the entire URL string. Hence, "`snr,2.5/;'ucd:meta.id;src',ir`" demonstrates the use of quotation to avoid misinterpretation of the ";" embedded in the UCD formatted field name.

3.3.1.6 TOP

A service **should** support the TOP parameter, used to order the query result by a score heuristic and return the specified number of top ranked records from the output table. For example, TOP=20 would return the top 20 ranked records from the output table. The details of the scoring heuristic used to rank the query response are server-specific, but the intent is to order the response by the degree of match to the query parameters. An example of a simple scoring heuristic for a spatial query is the distance from the center of the search region.

```
FROM=fp_psc&POS=180.0,0&SIZE=0.2&WHERE=j_snr,2.5/&TOP=20
```

The above example is identical to what we presented earlier except that only the top 20 rows of the response are returned (this differs from MAXREC in that overflow cannot occur, and an order-by-score heuristic is implied).

[*TOP comes from SSA etc., but is very similar to the SQL TOP except for the suggestion that a scoring heuristic be used to order the output table; if the ordering heuristic is omitted they are the same*.]

3.3.1.7 Other Query Parameters

The FORMAT, MAXREC, MTIME, RUNID, and OUTPUT parameters are identical for both ParamQuery and AdqlQuery. Refer to section 3.2.1 for a description of these parameters.

3.3.2 Query Response

The ParamQuery query response is identical to that for AdqLQuery. Refer to section 3.2.2 for a description of the query response.

3.3.3 Table Data Queries

Although ParamQuery supports only a restricted range of query expressions compared to AdqlQuery, it is equally efficient for table data queries even for very large tables. In the case of both ParamQuery and AdqlQuery the input query resolves to the same database operations, and so long as indexing is used carefully, both types of queries can be used to access very large tables, e.g., tables containing hundreds of millions of records or more. For some specialized applications where it is desirable to put a TAP interface in front of some form of data storage which is not SQL-based (e.g., to directly query a set of VOTables or FITS files), ParamQuery has the advantage of making it possible to provide a basic query capability without requiring either ADQL or SQL.

3.3.4 Table Metadata Queries

Rather than provide access to database and table metadata via a custom interface, querying of database and table metadata is provided by representing such metadata in a set of tables within a special database schema called the TAP schema (`$TAP_SCHEMA`). 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 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. A good example of dynamically changing table metadata is using the metadata query interface to query the

contents of the user's VOSpace storage area, which changes constantly as tables are added or deleted.

The core TAP schema defines the following tables (more detailed information on the TAP schema is given in section 4.1):

- **$TAP_SCHEMA.tables**. Lists all tables known to the TAP service and visible to the current client or user. This includes any views, and the contents of the user's VOSpace storage area at the service.

- **$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 table name will return only the columns of the given table.

- **$TAP_SCHEMA.tableset**. This is not a true table but is provided to enable the standard interface to be used to query *tableset* metadata, providing a convenient means to get a description of all data tables managed by the service.

Some examples should help illustrate how table metadata queries are used. The following simple query will 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
```

To list only the tables in the user's VOSpace the following query would be used instead:

```
FROM=$TAP_SCHEMA.tables&WHERE=tablename,$vospace.*
```

To list only data tables, ignoring the VOSpace:

```
FROM=$TAP_SCHEMA.tables&WHERE=tablename,!$vospace.*
```

To list all columns of table "fp_psc":

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

Table metadata queries use the regular table query interface, so the query response is returned as a VOTable by default, with other output format options available as for a data table, if supported by the service.

Simple TAP services are required only to be able to list all tables, or all the fields of a single table (a minimal TAP service does not implement VOSpace so that is not an issue). A fully compliant TAP service will support general table metadata queries using the full query interfaces.

3.3.5 Cone Search Query

The POS and SIZE parameters provide a spatial position query capability equivalent to the legacy cone search interface. Going back to our original example, the following would execute a cone search of table "fp_psc" using the specified position and search region diameter:

```
FROM=fp_psc&POS=180.0,0&SIZE=0.2
```

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 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 used for searches, 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.

3.3.6 Multi-Position Queries

A *multi-position* query generalizes POS, SIZE to a table of positions, allowing an arbitrarily large number of spatial position-based 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. The multi-position query provides this simple spatial cross match capability.

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 which the service's POS implementation understands [*we need to specify this more carefully in terms of UCDs and possibly UTYPEs*].

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 "positions" when executing the multi-position query, in which case we would have "`POS=@$upload.positions`". If a persistent table of positions is preferred this could be uploaded to the user's VOSpace instead, and referenced as "`POS=@$vospace.positions`".

In the most general case any table containing position information can be used. For example we could use the 2MASS point source catalog from our earlier examples. This table contains nearly half a billion sources, so the REGION parameter is 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 may optionally be added to further refine the query.

The output from a multi-position query is a single table, containing a sequence of zero or more table rows corresponding to each input position. A unique position ID is added to the table to indicate the position from the input position table to which the output table row corresponds. The other output table fields are taken 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 position table, and is allowed to vary for each position [*as with POS we need to specify more precisely how this is done.*]

3.4 GetCapabilities Operation

[*To be added*]

3.5 GetAvailability Operation

[*To be added*]

4 Table Metadata

4.1 TAP Core Schema

The TAP core schema is equivalent to that defined by the registry for a *VODataService* with minor additions. VODataService is in turn modeled after VOTable.

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

Tablename	table name including catalog and schema if used
Tabletype	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:

Name	column name
Tablename	table name, e.g., <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 *tablename* should include any catalog or schema names if these are used to reference tables by the server. This should include TAP_SCHEMA, plus the "vospace" schema if supported by the service (a VOSpace could be implemented as either a catalog or schema depending upon the DBMS, but should be treated as a schema in queries to avoid having the client know how the vospace is implemented). 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 element naming convention used here follows 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=true` indicates that the column is visible in the default (narrow) view of a table; `SELECT=$all` would display all columns. `Indexed=true` 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.

Other possible metadata fields under consideration but not yet included in the core schema include "dbtype" (the SQL database type used on the server), "width" and "precision", and "is_nullable".

The schema defined by a TAP service could add additional fields to extend the core schema shown here.

4.2 Table Sets

The schema also defines `$TAP_SCHEMA.tableset`, however this is not an actual table but rather a structured view of the two 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”.

5 Basic Service Elements

[*To be added. This is very similar to section 8 of the SSA specification*.]

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