*International*

*Virtual*

*Observatory*

*Alliance*

# Table Access Protocol

## Version 0.31

## IVOA Internal Working Draft 2008 November 24

**This version:**

TAP-V0.31-20081124

**Latest version:**

Not yet issued

**Previous version(s):**

http://www.ivoa.net/internal/IVOA/TableAccess/TAP-v0.3.pdf

http://www.ivoa.net/internal/IVOA/TableAccess/tap-v0.2.pdf

http://www.ivoa.net/internal/IVOA/TableAccess/TAP-QL-0.1.pdf

**Lead authors:**

P. Dowler, G. Rixon (editor), D. Tody

**Contributors:**

K. Andrews, J. Good, R. Hanisch, T. McGlynn, K. Noddle,

F. Ochsenbein, I. Ortiz, P. Osuna, R. Plante, G. Rixon, J. Salgado,

A. Stebe, A. Szalay

## 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....................4
1.1 Types of query....................5
1.1.1 Data, metadata, tableset and VOSI queries....................5
1.1.2 Parametric and ADQL queries....................5
1.1.3 Synchronous and asynchronous queries....................6
1.2 Interface Overview (informative)....................7
2 Requirements for a TAP service (normative)....................9

2.1 Principal functions....9
2.2 Web resources....9
2.3 TAP operations....10
2.4 Parameters for HTTP requests....11
2.4.1 REQUEST....11
2.4.2 QUERY....12
2.4.3 Parameters for parametric query....12
2.4.4 LANG....13
2.4.5 FORMAT....13
2.4.6 UPLOAD....13
2.4.7 MAXREC....14
2.4.8 MTIME....15
2.4.9 RUNID....15
2.4.10 VERSION....15
2.4.11 Range-list parameters....16
2.4.12 Missing or null-valued parameters....17
2.4.13 Case of parameters....17
2.4.14 Order and cardinality of parameters....17
2.5 Table names....17
2.6 Metadata tables and TAP schema....18
2.7 Table Uploads....19
2.8 Representations of results....20
2.8.1 Data and metadata queries....20
2.8.2 Tableset queries....21
2.8.3 VOSI....21
2.8.4 Error documents....22
2.8.5 Overflows....22
2.9 Versioning of the TAP protocol....23
2.9.1 Version number form and value....24
2.9.2 Version number changes....24
2.9.3 Appearance in requests and in service metadata....24
2.9.4 Version number negotiation....24
2.10 Parametric query-language....25

2.10.1 POS, SIZE....25
2.10.2 REGION....27
2.10.3 SELECT....27
2.10.4 FROM....27
2.10.5 WHERE....28
2.11 Numeric and boolean values....30
2.12 Use of VOTable....31
3 Service Registration (normative)....33
4 Extended capabilities (normative)....34
5 Use of UWS (informative)....34
6 VOSpace Integration (informative)....36
7 Use of HTTP (informative)....36
7.1 General HTTP request rules....36
7.1.1 Introduction....36
7.1.2 Reserved characters in HTTP GET URLs....37
7.1.3 HTTP GET....37
7.1.4 HTTP POST....38
7.2 General HTTP response rules....39
8 References....39

# 1 Introduction

The *Table Access Protocol* (TAP) is a Web-service protocol that gives access to collections of tabular data referred to collectively as a *tableset*. 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 [1]), 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.

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.

TAP is a member of the IVOA Data Access Layer (DAL) family of data access protocols, conforming 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 Types of query*

### 1.1.1 Data, metadata, tableset and VOSI queries

TAP services distinguish four different kinds of query by the information returned.

Data queries apply to the science archive served by a TAP installation. They are the reason for providing a TAP service. All the other kinds of query support the ability to make data queries. Data queries are phrased in a query language, either ADQL [1] or TAP's parametric query-language (see Section ).

Metadata queries work like data queries, using the same query languages, but they are applied to standardized tables which explain the data model of a particular TAP installation. Metadata queries allow a client to discover the names of tables and columns to be used in data queries.

Tableset queries are a special case metadata queries that reveal the entire data model in one response. They use a different output format to metadata queries.

VOSI 'queries' supply metadata concerning the availability of a TAP service, its main interfaces ('VOSI-capabilities'), and its data model ('VOSI-tables'). VOSI-capabilities and VOSI-tables outputs use the same XML schema as the IVOA registry and can be incorporated in service registrations.

### 1.1.2 Parametric and ADQL 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.

### 1.1.3 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.

## 1.2 Interface Overview (informative)

TAP defines a RESTful web service with two primary URLs: one for synchronous queries, including metadata requests, and one for asynchronous queries.[1] The query language and query constraints for a particular request are determined by the HTTP query-string.

This is an example of the URL for a synchronous ADQL query on *r* magnitude:

```
http://some.where/tap/sync?REQUEST=AQDLQuery \
&QUERY='SELECT * FROM magnitudes as m where m.r>10 and m.r<16'
```

The URL for an equivalent parametric query would be

```
http://some.where/tap/sync?REQUEST=ParamQuery
&FROM=magnitudes&WHERE=r,10/16
```

The REQUEST parameter indicates the kind of query required.

Synchronous queries return the table of results in the HTTP response to the initial request. In the examples above, the output format defaults to VOTable; the FORMAT parameter could be added to select a different format.

Synchronous queries can be requested using HTTP GET or HTTP POST; the examples show the GET form. Using GET allows the query result to come from a cache, either inside the TAP service or between the service and client.

Asynchronous queries are started in the same way as the synchronous kind, using the other URL:

```
HTTP POST to http://some.where/tap/async
REQUEST=AQDLQuery
QUERY='SELECT * FROM magnitudes as m where m.r>10 and m.r<16'
```

or

```
HTTP POST to http://some.where/tap/async
REQUEST=ParamQuery
FROM=magnitudes
WHERE=r,10/16
```

1 The original plan was to have just one, primary URL for all kinds of query. After investigation, we find that a single URL cannot satisfy fully the requirements of both the DAL generation-2 conventions and the UWS standard.

The service's response to these requests is an HTTP redirection (code 303 'see other') to a URL representing the query's state and progress. The progress may be tracked by polling a URL for the state: it will eventually go to *COMPLETED* or *ERROR*. The results, or an error document can then by retrieved from a URL associated with the job. This is an application of the UWS pattern. Details are in sections  and 5.

Positional queries have special support in the parametric query-language. This is a cone search on a specified table:

```
http://some.where/TAP/sync? \
REQUEST=paramquery&POS=12,34&SIZE=0.5&FROM=foo
```

A 'multi-position' cone-search may be done by uploading a table of search positions. This query

```
http://some.where/tap/sync?REQUEST=ParamQuery \
&UPLOAD=http://some.where.else/data/foo.vot,positions \
&POS=@TAP_UPLOAD.positions&SIZE=0.2
```

uploads the VOTable from *http://some.where.else/data/foo.vot* (i.e. from a server separate from the TAP service) and searches a 0.2-degree cone around each position. Section  gives more detail.

Each TAP service has its own 'tableset': a particular collection of tables and columns with locally-defined names. Those local names are the operands in the queries and so a client needs to know the tableset for a particular service to form a query. There are two ways of exploring the tableset.

First, a description of the entire tableset may be obtained in XML via the VOSI-tables URL, e.g.

```
http://some.where/TAP/sync?REQUEST=GetTableMetadata
```

These metadata are in the format defined for the IVOA resource-registry and the client may find a cached copy in the registry. Details are specified in section .

Secondly, the structure of the tableset is described by a set of tables with fixed names beginning with *TAP_SCHEMA*. These can be queried using either ADQL or the parametric-query languge.

This lists the columns of table 'foo':

```
http://some.where/tap/sync\
?REQUEST=paramquery&FROM=TAP_SCHEMA.COLUMNS
&WHERE=tablename,foo
```

The service's availability can be read using VOSI:

```
http://some.where/tap/sync?REQUEST=getAvailability
```

Finally, the service's capabilities can be read using VOSI:

```
http://some.where/tap/sync?REQUEST=getCapabilities
```

This output lists any extra interfaces such as legacy cone-search or built-in VOSpace support.

## 2 Requirements for a TAP service (normative)

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 [2]). 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.

### 2.1 Principal functions

An implementation of a TAP service must provide these capabilities:

- ADQL query with synchronous execution
- ADQL query with asynchronous execution

An implementation of a TAP service should provide these capabillities:

- Table metadata query (synchronous, VOSI compliant)
- Service metadata query (synchronous, VOSI compliant)
- Service availability query (synchronous, VOSI compliant)

An implementation of a TAP service may provide these capabilities:

- Parametric query with synchronous execution
- Parametric query with asynchronous execution

A TAP service **must** be registered in the IVOA resource-registry in the form specified in section .

*[Arguably, the requirements above come into force* because *a service is registered as TAP. This opens the question as to inheritance of requirements when a service derived from TAP is registered in a different form. E.g., a service searching a catalogue image cubes could be defined as a TAP service with a specific data-model and a different form of registration. In this case it is not clear that ADQL query would still be mandatory. This point was discussed briefly at the October 2008 Interop but no conclusion was reached. -Ed.]*

### 2.2 Web resources

A TAP service **must** be represented as a tree structure of web resources each addressable via a URL in the http scheme, or the https scheme, or both.

The web resource at the root of the tree **must** represent the service as a whole. This specification defines no standard representation for this root resource. Implementations **may** provide a representation, or may return a '404 not found' response to requests for the root web-resource. One possible representation is an HTML page describing the scientific usage of the service. TAP clients **must not** depend on a specific representation of the root web-resource.

A TAP service **must** provide a web resource with relative URL */sync* that is a direct child of the root web-source. This web resource **must** represent the results of synchronous queries, including metadata outputs defined by VOSI. The exact form of the query, and hence the representation of the resource, is defined by the query parameters as listed in section 2.4. Representations of results of data queries, metadata queries and VOSI outputs are defined in sections 2.8.1, 2.8.2 and 2.8.3 respectively.

*[During the writing of TAP 0.3, there was a divergence of opinion between the editors as to whether the VOSI metadata should be representations of the /sync web-resource or whether they should be made available on separate endpoints. The requirements in the preceding paragraph are a compromise between REST principle and the “DAL-2 architectural style”. This decision might be reviewed in later versions of TAP. - Ed.]*

An HTTP-GET request to the */sync* web-resource **may** return a cached copy of the representation. This cached copy might come from an HTTP cache between the client and the service, and the service **may** also maintain its own cache. Clients which require an up-to-date representation of volatile data or metadata **must** use HTTP POST.

A TAP service **must** provide a web resource with relative URL */async* that is a direct child of the root web-resource. This web resource **must** represent controls for asynchronous queries. Specifically, the web resource **must** represent the job-list as specified in the UWS standard [3].

A TAP service **must** provide web resources as specified by UWS. These are descendants of the */async* web-resource, and they include a web resource that represents the eventual result of an asynchronous query. A client making an asynchronous request **must** use the UWS facilities to monitor or control the job and to retrieve the result. If an asynchronous query succeeds, then the table of results **must** be made available as a UWS result with the formal name *result*.[2]

Requests to run queries on the */async* web-resource are always sent as HTTP-POST requests.

## 2.3 TAP operations

The operations supported by a TAP service are listed in the following table.

| Operation | Web resource | Requirement |
|---|---|---|

2 A 'UWS result' is a web resource for which the position in the tree of web resources is defined by the UWS standard. See section  for an explanation in the context of  TAP.

| | | |
|---|---|---|
| ADQLquery | /sync | “must” |
| ADQLquery | /async | “must” |
| ParamQuery | /sync | “may” |
| ParamQuery | /sync | “may” |
| GetCapabilities | /sync | “should” |
| GetAvailability | /sync | “should” |
| GetTableMetadata | /sync | “should” |

Operation names match allowed values of the *REQUEST* parameter, as specified in section . The web resources are specified in section .

## 2.4 Parameters for HTTP requests

The */sync* and */async* web-resources **must** accept the parameters listed in the following sub-sections. In a synchronous request, the parameters select the representation returned in the response message. In an asynchronous request, the parameters select the representation of the eventual query-result rather than the response to the initial request.

Not all combinations of the parameters are meaningful. E.g., if a request carries *REQUEST=ADQLquery* then the *SELECT* parameter (from the parametric-query language) is spurious. If a service receives a spurious parameter in an otherwise-correct request, then the service **must** ignore the spurious parameter, must respond to the request normally and **must not** report errors concerning the spurious parameter.

### 2.4.1 REQUEST

This parameter distinguishes data-query requests from VOSI requests and distinguishes ADQL queries from parametric queries. A TAP client **must** set this parameter correctly in every request. If a TAP service receives a request without this parameter or with an incorrect value for this parameter, then the service **must** reject the request and return an error document as the result.

These are the allowed values of the parameter.

- paramquery: execute a parametric query.
- ADQLquery: execute an ADQL query.
- getCapabilities: return VOSI-capabilities metadata.

- getAvailability: return VOSI-availability metadata.
- getTableMetadata: return VOSI-tables metadata.

*[The various examples of the REQUEST parameter in TAP 0.3 are inconsistent in case. Given that parameter values are stated below to be case-sensitive, TAP 0.4 should pick one capitalization of the REQUEST values and stick to it – Ed.]*

### 2.4.2 QUERY

A service **must** support the *QUERY* parameter, used to input the ADQL [1] (or other query language) statement to be executed. The query string is case sensitive. In particular, the case of table and column names **must** be preserved between a metadata query and a subsequent query of a data table.

Within the ADQL query, the service **must** support the use of datetime/timestamp values in ISO8601 format.

If the tables that are queried through a service contain columns with spatial coordinates and the services wants to enable the caller to perform spatial queries, the service **must** support the *INTERSECTS* function and it **must** support the following geometry functions: *REGION*, *POINT*, *BOX*, *CIRCLE*, *COORD1*, *COORD2*, *COORDSYS*. Support for the *AREA*, *CONTAINS*, and *POLYGON* functions are optional. If the service supports the *REGION* function, it **must** support region encoding in STC-S format [4]; the extent of STC-S support within the *REGION* function is left up to the implementation. Coordinate system specification for *POINT*, *BOX*, *CIRCLE*, and *POLYGON* **must** use values from Table 3 (standard reference frames) in STC [4].

Although it is allowed by the ADQL syntax, services **should** return an error if use of *POINT, BOX, CIRCLE*, or *POLYGON* mix constants and column references for coordinate system and coordinate values. For example, *POINT('ICRS', t.ra, t.dec)* **should** be an error. The only place where mixing constants and column references is acceptable is when using a constant for the radius in a *CIRCLE*, e.g. *CIRCLE(t.coordsys, t.ra, t.dec, 0.1)*, as this allows the caller to control search radii for multi-position searches or source cross-matching.

The *QUERY* parameter is not used to input a parametric query.

### 2.4.3 Parameters for parametric query

A number of parameters are used specifically for the parametric queries, both synchronous and asynchronous. This query language is described in section .

### 2.4.4 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. The version of the query language **may** 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.

### 2.4.5 FORMAT

The *FORMAT* parameter indicates the client's desired format for the table of results of a query. Its value **should** be a MIME type for tabular data or one of the following shorthand forms:

- votable
- csv (comma separated values)
- fits (FITS binary table)
- text (pretty-printed text)
- html (pretty-printed Web page)

All the shorthand forms are insensitive to case.

If the parameter is omitted, the default format is VOTable.

A TAP service **must** support VOTable as an output format, **should** support CSV output and **may** support other formats. A TAP service **must** accept a *FORMAT* parameter indicating a format that the service supports and **should** reject queries where the *FORMAT* parameter demands an unsupported format.

### 2.4.6 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 2.4.11). 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 (see section 2.7). The given name for the table name **should** be an unqualified table-name; uploaded tables will automatically be qualified with the schema name T*AP_SCHEMA*. The upload table storage area is shared with any tables uploaded in-line with the query.

### 2.4.7 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 an overflow indicator as specified in section .

If *MAXREC* is not specified in a query, the service **may** apply a default value or **may** set no limit. 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 capabilities. 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.

*[This paragraph seems to contradict the detailed rules for handling overflow stated elsewhere in this document. Both this and the alternate provisions are taken from TAP 0.3. Since the rules for handling overflow in section  are more complete, I suggest that the paragraph above be dropped in TAP 0.4 - Ed.]*

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.

### 2.4.8 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.

When the *MTIME* parameter is specified, the service **must** add extra columns to the output table (in addition to that specified in the select statement of the query), with utypes *Record.Modified* and *Record.Deleted*. The values in this column are the time-stamp when the last insert/update occurred or when the row was deleted respectively. One value must be set and the other null for every row in the table; values are in ISO8601 format. Rows that are thus marked as deleted must include values for one or more column(s) that uniquely identify the row, but other column values may be null.

The *MTIME* parameter **must not** be used with queries that select from multiple tables (joins in ADQL). If *MTIME* is used in a such a query the service **must** reject the request and return an error document.

### 2.4.9 RUNID

The service **should** implement the *RUNID* parameter, used to tag service requests with the job ID of a larger job of which the request may be part. 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 **should** ensure that RUNID is preserved in any service logs.

The service **should** pass on the *RUNID* value in any calls to other services, e.g. VOSpace.

### 2.4.10 VERSION

The *VERSION* parameter specifies the protocol version number. The format of the version number, and version negotiation, are described in section 2.9.

A TAP service **must** support the *VERSION* parameter.

### 2.4.11 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 section  and IETF RFC 2396 [5]).

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 **may** input ranges or range values in any order for an ordered range list and the result **must** 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 *SELECT* parameter in the parametric query-language 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).

#### 2.4.12 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.

#### 2.4.13 Case of parameters

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

#### 2.4.14 Order and cardinality of parameters

Parameters in a request **may** be specified in any order.

When request parameters are duplicated with conflicting values, the response from the service is undefined. The service **may** reject the request or it **may** pick one value for for the parameter. Clients **should not** repeat parameters in a request.

### *2.5 Table names*

A fully qualified table name has the form

```
[[catalog_name"."[schema_name"."]table_name]]
```

where *catalog_name* is the the name of the DB catalogue (often the "database" name) in SQL DBMS terminology, *schema_name* is the name of the "schema" 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.

The implementation of a TAP service **must** define the table names acceptable in queries and **must** reveal these to clients through metadata queries or through VOSI-tables output, and the names **must** be identical in each of these sources. .

A TAP client **must** determine the acceptable names from one of these sources or from the cached form of the VOSI-tables output included in the service's registration. A client **must** use the names in the exact form given by the service, reserving the case of letters and the embedded punctuation.

## 2.6 Metadata tables and TAP 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 information in the TAP core-schema is equivalent to that defined by VOSI-tables and allowed by the registry for a *VODataService*.

The qualified names in the tables of the TAP schema **must** follow the rules defined in section 2.5. The names **must** be stated in a form that is acceptable as an operand of a query.

The table TAP_SCHEMA.schemas **must** contain 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 **must** contain 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 must contain 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) |

A TAP service **must** provide the tables listed above and **may** provide other tables in the *TAP_SCHEMA* namespace.

The schema name *TAP_UPLOAD* **should** be included in the table name for any tables uploaded to the service by a client.

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.

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. Special output formats are defined for queries against this view; see section 2.8.2. A simple tableset-query **must** 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 table name matches some pattern.

## *2.7 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 (section 2.4.6) 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 publicly-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 uploaded table would automatically propagate and could be referenced in either ADQL or parametric 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.

## *2.8 Representations of results*

### 2.8.1 Data and metadata queries

The result of a data query or a metadata query **must** be a single table.

This table **must** be encoded in the output format specified by the FORMAT parameter of the query. See section  for required, optional and default formats. VOTable is the default format and VOTable support is mandatory.

VOTables **must** follow the rules in section . These VOTables **should** be returned with a MIME type of *text/xml;content=x-votable*.

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.

### 2.8.2 Tableset queries

If the target of the query is the special table *TAP_SCHEMA.tableset*, then the service **must** support an XML serialization of the tableset and **must** support a special use of VOTable to express the structure of the tableset.

The special, XML serialization **must** conform to the registry standard expressed in *VODataService* v1.1 [7] and the corresponding XML-schema. This serialization format is identical to that used for VOSI tables [6].[3] This format is selected by the parameter setting *FORMAT=xml* in the query.

The special use of VOTable **must** be a data-less VOTable in which the header elements denote the structure of the tableset. There **must** be one *VOTABLE* element per table in the tableset. This is an exception to the rule that query results contain single tables. This format is selected by the parameter setting *FORMAT=votable* in the query.

*[In v0.3 of the TAP standard the intent w.r.t metadata queries is clear but the implementation details are not. I have inferred some of the detailed rules as best I can, but may have diverged from the original intent. These details should be cleared up in TAP 0.4 - Ed]*

### 2.8.3 VOSI

Representations of VOSI outputs (service capabilities, availability, table metadata) **must** be as defined in the VOSI standard [6].

The representation of table metadata **must** include all tables in the service's tableset.

VOSI's representation of table metadata is that mandated for the registry in *VODataService* [7].

3 The registry-compliant-XML serialization of the tableset structure is almost the same thing as the VOSI-tables output of the service but is not strictly identical. The format is the same, but while the VOSI output is required to cover all the tables in the tableset (implicitly '*SELECT * FROM TAP_SCHEMA.tableset*), the result of a tableset *query* can be restricted by the *WHERE* clause of that query.

*[In TAP v0.3, it was written that 'The content of the TAP service availability description are TBD.' My understanding of VOSI is that there are no details left to determine. -Ed]*

### *2.8.4 Error documents*

If the service detects an exceptional condition, it **must** return an error document with an appropriate HTTP-status code. TAP distinguishes three classes of exceptions.

- Errors in the use of the HTTP protocol.
- Errors in the use of the TAP protocol, including failure of the service to complete valid requests.
- Overflow conditions where the number of rows returned from a query would exceed a pre-set limit (set either by the client or by the service).

Error documents for HTTP-level errors are not specified in the TAP protocol. Responses to these errors are typically generated by service containers an cannot be controlled by TAP implementations.

Error documents for TAP errors **must** be VOTable documents; in exceptional conditions, any result-format specified in the query is ignored. When returning such a document, the service **must** set HTTP status-code 200 'OK' (because the HTTP operation is correct, even though the request cannot be fulfilled). The exception condition **must** be signaled to the client using a status code in the VOTable header and a qualifier in the MIME type reported in the HTTP header. Section  specifies the exact use of these error documents.

Overflow conditions are not strictly errors and results from overflowed queries are not strictly error-documents. Therefore, a response to an overflowed query **must** contain the results table truncated at the row limit and **must** be in the format requested by the client. A response to an overflowed query **should** contain an indication of the overflow if the output format allows this. Section  specifies the means of reporting overflows in VOTables. No reporting mechanism is specified for other formats.

Example:

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

### 2.8.5 Overflows

If a query is executed by a TAP service, the number of rows in the table of results may exceed a limit set by the user (using the MAXREC parameter or the TOP keyword in ADQL) or a limit set by the service implementation. In these cases, the query is said to have 'overflowed'. Typically, a TAP service will not detect an overflow until some part of the table of results has been sent to the client.

On detecting an overflow, a TAP service **must** produce a table of results that valid in the required output format and which contains all the results up to the point of overflow. Since an output overflow is not an error condition, the MIME type of the output VOTable **must** be the same as for any successful query and the HTTP status-code **must** be as for a successful, complete query.

If the service detects the overflow before sending the query response to the client, and if the output format is VOTable, then the service **must** include in the table of results an *INFO* element with name attribute set to *QUERY_STATUS* and value element set to *OVERFLOW*. The service should set the value of this element to an error message explaining the overflow. The error message should state the number of rows at which the output was truncated.

If the output format is VOTable, and if the service detects the overflow after the header for the table of results has been sent to the client, then the service **must**, after closing the *TABLE* element for the table of results, write another *TABLE* element to indicate the overflow. This latter table **must** not include data but **must** an *INFO* element announcing the overflow. This element **must** have attributes and content as specified for the case where the overflow was detected before starting to write the VOTable.

No method of reporting an overflow is defined for formats other than VOTable.

*[The discussion leading to the TAP 0.3 specification produced the two-table. The TAP 0.3 document does not mention this scheme. Since the two-table approach was agreed to be necessary by the editors of TAP 0.3, I have included it this draft -Ed.]*

```
Example:
<INFO name="QUERY_STATUS" value="OVERFLOW">
Number of table rows exceeds default limit of 5000
</INFO>
```

## 2.9 Versioning of the TAP protocol

The TAP protocol provides explicitly for versioning of the interface, using the features provided by the VOA registry and the conventions of the DAL-2 architecture.

### 2.9.1 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.

### 2.9.2 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 **must** 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.

### 2.9.3 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.

### 2.9.4 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 VOSI-capabilities or via a registry query.

## 2.10 Parametric query-language

A TAP service which implements parametric queries on data must do so using the parameters defined in this section. None of these parameters are mandatory unless the service supports parametric queries; however, many of the parameters become mandatory once a service registers its support for such queries. These parameters have no meaning in ADQL queries or VOSI requests.

### 2.10.1 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 2.4.11 and as implemented in other second generation DAL interfaces.

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 [4].

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 later in this section.

*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, taking into account the spatial resolution of the data.

The service **must** accept a value for *POS* in the form *POS=@name* where *name* identifies a table known to the TAP service and conforms to the rules for table names specified in section . When *POS* is given in this form, it identifies a table of search positions. The service **shall** then perform a positional search for each position in the positions table in the manner specified above for the single-position form of *POS*. The service **must** concatenate the results from all these searches and return them in a single table.

When interpreting a table of positions, the service **must** identify the relevant columns, by utypes if these are known, or else **must** identify the columns by UCDs. These values of utype and UCD indicate appropriate columns:

| 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 |

The table of results from a multi-position query **must** contain all the columns of the data table being queried plus an extra column identifying the row in the positions table that selected each row of the results. If the positions table contains a position-identifier column, then this **must** be transcribed to the output. Otherwise, the service **must** add a position-identifier column containing a 1-indexed integer position ID specifying the row of the input table.

If the positions table is not annotated with either utypes or UCDs, then the result of a multi-position query is undefined. If the position table is part of the TAP service's tableset, as opposed to a table uploaded by the client, then the service **may** use an implementation-dependent technique to identify the positional and search-radius columns. (This form of inference is required for the single-position form of position query.)

If a *SIZE* parameter is specified the service **must** apply this value at all search positions, overriding any column of search radii. If there is neither a *SIZE* parameter nor a column or radii then the service **must** use a default search-radius.

Positions tables for multi-position searches **may** be arbitrarily large (consider using the 2MASS catalogue as a positions table for a search of SDSS inside one tableset), but a given TAP service **may** refuse to process such a query. Clients **should** the REGION parameter to constrain searches on large lists of positions.

### 2.10.2 REGION

The service **may** accept a *REGION* parameter, used to define more general spatial search regions than can be defined using *POS* and *SIZE*. The value of *REGION* **must** be a STC/S (string encoded) region specifier, e.g.

```
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 *POS*,*SIZE* and *REGION* are all specified in the same query, *REGION* **must** be used as a mask to further qualify the circular region specified by *POS* and *SIZE*. This is most useful for multi-position queries, 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.

### 2.10.3 SELECT

The **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.

### 2.10.4 FROM

The *ParamQuery* operation must implement a FROM parameter, indicating the name of the table to be queried, specified as defined in section . 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. E.g.

FROM=hdfv2

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

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

### 2.10.5 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 illustrates 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 be 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 (2.4.11) 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 meta-characters 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
```

## 2.11 Numeric and boolean values

Integer numbers **must** be represented in a manner consistent with the specification for integers in *XML Schema Datatypes* [10]. This document indicates explicitly where an integer value is mandatory. Real numbers **must** 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 must 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 indicates explicitly where a Boolean value is mandatory.

## 2.12 Use of VOTable

VOTable is a general format. TAP requires that it be used in a particular way.

VOTables **should** comply with VOTable v1.1 or greater [9].

VOTables resulting from successful queries, including overflowed queries (see section  for a definition of overflow) **must** be returned with MIME type *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 and TAP clients **should not** depend upon them.

The *RESOURCE* element **must** contain, before the *TABLE* element, an *INFO* element with attribute *name = "QUERY_STATUS"*. The *value* attribute **must** contain one of the following values:

- “OK”, meaning that the query completed successfully and did not overflow;
- “ERROR”, meaning that an error was detected at the level of the TAP protocol;
- “OVERFLOW”, meaning that the query completed without error but overflowed;
- “STREAM”, meaning that neither error nor overflow had been detected when the service started to write the results to the client, but that either condition could still arise before the response is completed.

The STREAM status covers the case where the service streams a long table of results to the client rather than buffering it. In this situation, the data typically come from an SQL cursor and the service does not know the number of rows in the response when starting to write the *TABLE* element; overflow cannot be reported in the initial *INFO* element. When the initial status is set to STREAM, the service **must** write a second *INFO* element, with *name="QUERY_STATUS"*, after the end of the *TABLE* element. This element **must** have its *value* attribute set to *"OK", "ERROR"* or *"OVERFLOW"*.

The value of the *INFO* element conveying the status **should** be a message suitable for display to the user describing the status.

**Examples:**

```
<INFO name="QUERY_STATUS" value="OK"/>
<INFO name="QUERY_STATUS" value="OK">Successful query</INFO>
<INFO name="QUERY_STATUS" value="ERROR">DEC out of range: DEC=91</INFO>
<INFO name="QUERY_STATUS" value="OVERFLOW">
Number of table rows exceeds default limit of 5000
</INFO>
```

Additional *INFO* elements **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), but clients **should not** depend on these.

**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>
```

*[TAP 0.3 also says this concerning status reporting in streamed responses: 'Alternatively, the initial query status could be OK or ERROR and a failure later on would require just the additional INFO with OVERFLOW or ERROR - then we do not have to add STREAM... that might be a more general solution' – Ed.]*

If the output of a query includes column(s) of type datetime/timestamp (?), the values **must** be specified in ISO8601 format.

If the output of a query includes columns of type region (e.g. a column of type *POINT, CIRCLE, POLYGON*, or *REGION* as defined by ADQL), the value **must** be output in a single column and encoded in STC-S format (Rots 2007). If the underlying tables (as described by the table metadata) store spatial information in multiple columns (e.g. RA and DEC in separate columns), then the output **may** also use multiple columns (and, in the case of VOTable, the coordinate system can usually be specified by a *PARAM* rather than a *FIELD*).

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.

## 3 Service Registration (normative)

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 resource document for a TAP service instance **must** be structured according to *VOResource* 1.0 [8] using the sub-type *CatalogService* as defined in *VODataService* 1.1 [7].

The resource document **must** include a *capability* element denoting the TAP interface and functions. The content of this element, including the value of its *standardID* attribute is TBD.

*[In the debate leading to TAP 0.3, it was suggested that the capability might list as interface the URL for the root web-resource of the service (as defined in section ). Clients would add to this URL /sync or /async as appropriate. This arrangement was not confirmed in the text of v0.3 and should be confirmed or replaced in TAP 0.4 – Ed.]*

The resource document **must** contain capability elements for the VOSI-capabilities, VOSI-availability and VOSI-tables outputs. These **must** be formatted as in the VOSI standard [6].

*[This requirement is not stated in TAP 0.3. I have added it since VOSI itself requires it – Ed.]*

The resource document should include the table metadata, except where the database-schema of the archive changes frequently.[4] Where table metadata are

4 If the database schema changes faster than the changes can be propagated through the publishing registries to the full registries, then it is pointless to register the table metadata. If the details change hourly then clearly the registries cannot keep up; if the details change

provided, they **must** be represented as XML elements drawn from *VODataService* 1.1.

## 4 Extended capabilities (normative)

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 **must** not be required or expected to make use of such extensions. Extended capabilities or operations **must** 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 **must** 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 **must** supply a default value for any extended capabilities it defines), or if parameters are supplied that are not known to the server.

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

## 5 Use of UWS (informative)

The UWS pattern is specified in [3] and its application to TAP in section . This section explains the exchange of messages between a TAP client and service when using UWS to run an asynchronous query.

Consider a TAP service at *http://x.y.z/TAP*. TAP mandates that the asynchronous requests be directed to *http://x.y.z/TAP/async*. This URL points to the list of 'jobs'; i.e. the list of queries currently or recently executed.

To start a new query, the client posts a request to the job list.

HTTP POST to http://x.y.z/TAP/async

REQUEST=ADQLquery&ADQL=SELECT TOP 100 * FROM foo

The service then creates a job and assigns that job a name and a URL based on the name. Suppose that the name is *j42*, then the URL will be *http://x.y.z/TAP/async/j42* because the jobs are always children of the job list.

The service then issues an HTTP redirection to the job's URL.

```
HTTP status 303 'See other'
Location: http://x.y.z/TAP/async/j42
```

Beneath the job URL there are further URLs for aspects of the job:

yearly, then clearly they can. Intermediate cases are less certain, but weekly changes are probably too fast and monthly changes probably slow enough.

```
http://x.y.z/TAP/async/j42/phase
http://x.y.z/TAP/async/j42/results
http://x.y.z/TAP/async/j42/error
```

(there are more, but these are the one that the client has to deal with).

The *phase* URL shows the progress of the job. When the job is created by the service it will normally be set to *PENDING*, but might be set to *ERROR* if the service has rejected the job. If the phase is *ERROR*, then the *error* URL should lead to a an error document explaining the problem. If the phase is *PENDING*, then the client needs to commit the job for execution.

The client commits the job by posting to the phase URL

```
HTTP POST to http://x.y.z/TAP/async/j42/phase
PHASE=RUN
```

The service replies with a redirection to the job URL

```
HTTP status 303 'see other'
Location: http://x.y.z/TAP/async/j42
```

The phase will now have changed to either *QUEUED* or *EXECUTING*, depending on the service implementation. The client tracks the execution by polling the phase URL:

```
HTTP GET http://x.y.z/TAP/async/j42/phase
```

When the query is complete, the phase changes to *COMPLETED*. The client then retrieves the result from the results list:

```
HTTP GET http://x.y.z/TAP/async/j42/results/result
```

The client knows that the table of results is at the URL /result relative to the results list because the TAP protocol requires this naming.

If the service cannot run the query, then the final phase is *ERROR* and there is no table of results. In this case, the client should expect an HTTP 404 'not found' status if it tries to retrieve the result. The client should look instead at the error URL to find out what went wrong

```
HTTP GET http://x.y.z/TAP/async/j42/error
```

The service remembers the job for a limited period after which it forgets the job information and discards the result of the query. After job expires, the client will receive an HTTP 404 'not found' status if it tries to get any information about the job. The destruction time of the job is chosen by the service and the client can read it from the job:

```
HTTP GET http://x.y.z/TAP/async/j42/destruction
```

The service may allow the client to change the destruction time:

```
HTTP POST to http://x.y.z/TAP/async/j42/destruction
DESTRUCTION=2008-11-11T11:11:11Z
```

The basic sequence can be executed from a web browser or from a shell script using the *curl* utility:

```
curl -d 'REQUEST=paramquery&POS=12,34&SIZE=0.5&FROM=foo' \
       http://x.y.z/TAP/async
 [read Location header from curl output]
curl -d 'PHASE=RUN' http://x.y.z/TAP/async/j42
curl http://x.y.z/TAP/async/j42/phase
[repeat until phase is COMPLETED]
curl http://x.y.z/TAP/j42/results/result
```

# 6 VOSpace Integration (informative)

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 subsequent 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.

# 7 Use of HTTP (informative)

A TAP service is a web service and TAP implementations are constrained by the general rules for use of HTTP, which are contained in IETF RFC documents. This section collates some of the requirements. For authoritative specifications, please refer to the original RFCs.

## *7.1 General HTTP request rules*

### 7.1.1 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 [11]). 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 **must** conform to the description in IETF RFC 2616 (section 3.2.2 “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.

### 7.1.2 Reserved characters in HTTP GET URLs

The URL specification (IETF RFC 2396 [5]) 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 **must** 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 **must** 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.

### 7.1.3 HTTP GET

A TAP service **must** support the "GET" method of the HTTP protocol (IETF RFC 2616 [11]).

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 [5] 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 **must** 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] | 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. |

### 7.1.4 HTTP POST

TAP uses the “POST” method of the HTTP protocol (IETF RFC 2616 [11]) 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.

## 7.2 General HTTP response rules

Upon receiving a valid request, the server **must** send a response corresponding exactly to the request as detailed in section  of this document, or send a service exception if unable to respond correctly. Only in the case of Version Negotiation (see 2.9.4) may the server offer a differing result. Upon receiving an invalid request, the server **must** issue a service exception as described in section .

A server may send an HTTP Redirect message (using HTTP response codes as defined in IETF RFC 2616 [11]) 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 **must** be a TAP response that corresponds exactly to the original request (or a service exception).

Response objects **must** be accompanied by the appropriate Multipurpose Internet Mail Extensions (MIME) type (IETF RFC 2045 [12]) 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.

# 8 References

[1] I. Ortiz, J. Lusted, P. Dowler, A. Szalay, Y. Shirasaki, M. Nieto- Santisteban, M. Ohishi, W. O'Mullane, P. Osuna, VOQL-TEG & VOQL-WG, *IVOA Astronomical Data Query Language version 2,* IVOA recommendation 30th October 2008. http://www.ivoa.net/Documents/REC/ADQL/ADQL-20081030.pdf

[2] S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels,* IETF RFC 2119. http://www.ietf.org/rfc/rfc2119.txt

[3] G. Rixon & P. Harrison, *Universal Worker Service Version 0.5,* IVOA internal working-draft 8th October 2008. http://www.ivoa.net/internal/IVOA/AsynchronousHome/UWS-0.5.pdf

[4] A. Rots, *Space-Time Coordinate Metadata for the Virtual ObservatoryVersion 1.33*, IVOA Recommendation 30 October 2007. http://www.ivoa.net/Documents/REC/DM/STC-20071030.html

[5] T. Berner-Lee, R. Fielding  L. Masinter, *Uniform Resource Identifiers (URI): Generic Syntax,* IETF RFC 2396.  http://www.ietf.org/rfc/rfc2396.txt

[6] G. Rixon (ed.) & GWS-WG, *IVOA Support Interfaces Version 1.00*, IVOA Working Draft 2008 October 23. http://www.ivoa.net/Documents/WD/GWS/VOSI-20081023.pdf

[7] R, Plante, (ed.), A. Stébé, K. Benson, M. Graham, G. Greene, P. Harrison, A. Linde, G. Rixon & IVOA Registry-WG, *VODataService: a VOResource Schema Extension for Describing Collections and ServicesVersion 1.01.* IVOA Working Draft 16 October 2008. http://www.ivoa.net/internal/IVOA/VODataService/VODataService-v1.1wd.html

[8] R. Plante (ed.), K. Benson, M. Graham, G. Greene, P. Harrison, G. Lemson, A. Linde, G. Rixon, A. Stébé, & IVOA Registry-WG, *VOResource: an XML Encoding Schema for Resource MetadataVersion 1.03*, IVOA Recommendation 22 February 2008. http://www.ivoa.net/Documents/REC/ReR/VOResource-20080222.html

[9] F. Ochsenbein (ed.), R. Williams, C. Davenhall, D. Durand, P. Fernique, D. Giaretta, R. Hanisch, T. McGlynn, A. Szalay, M. Taylor, A. Wicenec, *VOTable Format DefinitionVersion 1.1*, IVOA Recommendation 11 August 2004. http://www.ivoa.net/Documents/REC/VOTable/VOTable-20040811.html

[10] P. Biron & A. Malhotra, *XML Schema Part 2: Datatypes Second Edition,* W3C Recommendation 28 October 2004. http://www.w3.org/TR/xmlschema-2/

[11] R. Fielding, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, *Hypertext Transfer Protocol – HTTP/1.1*, IETF RFC 2616. http://www.rfc-editor.org/rfc/rfc2616.txt

[12] N. Freed & N. Borenstein, *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies,* IETF RFC 2045. http://www.ietf.org/rfc/rfc2045.txt