TAP/QL

Version 0.1

IVOA internal working-draft 2008 May 12

This version:

0.1-2008-05-12

Latest version:

(not yet published)

Previous version(s):

None.

Author(s):

Kona Andrews, Patrick Dowler, Keith Noddle, Francois Ochsenbein, Iñaki Ortiz, Pedro Osuna, Guy Rixon (editor), Jesus Saldago, Aurélien Stébé

Abstract

We define the TAP/QL service-protocol which allows ADQL queries on tabular archives. The queries may be executed synchronous or asynchronously and the results may be directed to VOSpace. We also describe how to lay out the metadata for both the service and the tables in its archive.

Status of This Document

This is a working draft internal to 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/.

Contents

1 Introduction

TAP/QL stands for “Table Access Protocol, Query Language”. It is a web-service protocol for querying tabular data using Astronomical Data Query Language (ADQL) [1]. The protocol is intended primarily for querying relational-database systems and includes elements that exploit common RDBMS features. However, TAP/QL could also be implemented for an archive of tables stored in flat files.

The basic operation of TAP/QL is this:

1. The client sends an ADQL query as an HTTP request to an endpoint, specifying the output format and where to send the results.
2. The service implementation executes the query on its internal database-engine, receiving the results as a single table.
3. The service sends the results to the specified location, serializing them into the requested format.

The query can be executed asynchronously or asynchronously. In a synchronous execution, the query is finished and the results delivered when the client receives a response to its initial request. In an asynchronous execution, the response to the initial request is received before the query completes and the client then tracks and controls the execution using the Universal Worker Service (UWS) pattern [2].

The results of a query may be sent directly to the client (synchronous query), cached in the service for the client to download (asynchronous query) or sent to VOSpace (either mode).

In order to phrase the ADQL query, the client needs to know something of the schema of the database. TAP/QL allows a client to download a description of this schema from an HTTP endpoint. As ADQL evolves, it may also become possible to query the schema directly.

There is a related protocol called TAP/PARAM in which the query is expressed differently. That protocol is not defined here.

In the following text, the “specification” sections are normative and the “discussion” sections are not.

2 Query protocol

2.1 *Specification*

A TAP/QL service SHALL expose its querying capability as a tree of web resources accessible by HTTP or HTTPS. The resource at the root of the tree represents the TAP/QL service as a whole and its URI shall be written in the service's registration as described below.

2.1.1 Synchronously-executed queries

The service SHALL provide a resource representing the results of synchronously-executed queries. The URI relative to the root resource SHALL be */sync.*

On receiving a request at this resource, the service SHALL obtain the table resulting from the query stated in the *ADQL* parameter and shall write it to the location stated in the *DEST* parameter, serializing the table into the encoding specified by the *FORMAT* parameter (see below for details of these parameters). If the request method is HTTP-GET, the service MAY take the results from a cache, but if the method is HTTP-POST then the service MUST execute the query on the underlying database to obtain fresh data.

If the *DEST* parameter is not present in the request, the table of results SHALL be returned as the body of the response to the original HTTP-request. If the *DEST* parameter is present in the query, then the HTTP response to the query-request SHALL be a redirection to the given location of the results.

2.1.2 Asynchronously-executed queries

The service SHALL provide a resource representing the process of performing queries asynchronously. The URI relative to the root resource shall by */async*. This resource shall be structured as a job-list object as specified in the UWS pattern [2].

On receiving a request at this resource via HTTP method POST, the service SHALL obtain the table resulting from the query stated in the *ADQL* parameter and shall write it to the location stated in the *DEST* parameter, serializing the table into the encoding specified by the *FORMAT* parameter. The service SHALL treat this query-format-deliver process as a UWS job and SHALL record that job as a new web-resource subordinate to the job-list, as specified by UWS; the service shall then return a response to the original query-request while the job is in progress and the details of the response shall be as specified by UWS.

The client MAY then monitor and control the job using the controls specified by UWS.

2.1.3 *ADQL* query-parameter

The value of this parameter SHALL be the ADQL query to be executed, encoded as a UTF-8 string. All query requests MUST include this parameter.

2.1.4 *FORMAT* query-parameter

The value of this parameter SHALL be the MIME type in which the table of results is to be encoded.

When the output format is VOTable [ref], the MIME type shall be one of the following values;

- *application/x-votable+xml;tabledata* for VOTable with data written in XML as a *TABLEDATA* element;
- *application/x-votable+xml;fits* for VOTable with data written as an embedded FITS-file;
- *application/x-votable+xml;binary* for VOTable with data written as an embedded, binary structure.

The service SHOULD NOT produce VOTables in which the data are in a separate file from the metadata.

Where the output is to be a FITS table, the MIME type shall be *application/fits*.

A TAP/QL service MUST accept requests for *application/x-votable+xml;tabledata* and SHOULD accept requests for *application/x-votable+xml;fits* and *application/x-votable+xml;binary.*

This parameter is optional for the client. If the client does not provide the parameter then the service should assume *application/x-votable+xml;tabledata* as a default.

2.1.5 *DEST* query-parameter

The value of this parameter SHALL be the URI of the location to which the table of results is to be written.

This parameter is optional for the client. If the client does not include the parameter in the request, then the service SHALL use a default destination as specified above (synchronous execution) and below (asynchronous execution).

2.1.6 *table* result-parameter

When a query is executed and recorded as a UWS job, the table of results SHALL be represented as the output parameter named *table*.

If the parameter *DEST* was not given in the request that started the job, the service SHALL store the table of results internally. The client MAY then retrieve them from the web-resource for the parameter.

If the parameter *DEST* was present in the query request, then a request to the web-resource for this parameter shall be redirected to the URI specified by *DEST.*

2.1.7 Error handling

When a query is executed synchronously, the service SHALL use HTTP status-codes and error documents to indicate failed requests. Status codes MUST be used in strict accordance with RFC 2616 [3]. Error documents SHALL be of MIME type text/plain.

When a query is executed asynchronously the error handling is defined by UWS.

2.2 Discussion

The tree of web resources for a service might look something like this.

```
http://host/service/sync
http://host/service/sync?ADQL=SELECT *...&FORMAT=...&DEST=...
http://host/service/async
http://host/service/async/666
http://host/service/async/666/phase
```

http://host/service/async/666/termination

http://host/service/async/666/quote

http://host/service/async/666/results

http://host/service/async/666/results/table

where “666” is a UWS job; typically, there might be several jobs recorded at this level. “table”, inside “666” is the output parameter holding the table of results; an HTTP-get to this resource will either produce the results directly or will produce a redirection to the results (probably via VOSpace).

The various URIs obtained by extending “sync” with an HTTP query-string effectively point to different, potential tables of results. They are a kind of enumeration across the results space. The representation of each of these resources may be cached independently, either by the TAP/QL service or by HTTP proxies. The resource “sync” itself, with no query string, does not have a standard representation and should not be requested by the client.

The results of synchronously-executed queries may sometimes be cached but sometimes should not be cached; consider, e.g., a database of transient events where new records are added frequently. Therefore, the protocol treats HTTP-get and HTTP-post differently for the “sync” resource.

Early suggestions for TAP/QL had a single endpoint for synchronous and asynchronous execution of queries. An HTTP-get request caused synchronous execution and HTTP-post started an asynchronous job. Because we distinguish get and post even for synchronous queries, this arrangement cannot work. Instead, the asynchronous and synchronous queries have separate web-resources.

“Query parameters” are normal parameters of HTTP requests. “Result parameters” are a mechanism from the UWS pattern and are represented by web resources.

“Status codes must be used in strict accordance with RFC 2616” effectively restricts the set of codes that the service can return. E.g., 401 “unauthorized” requires a challenge response specific to HTTP authentication, so cannot be used to denote a failed authorization check; 403 “forbidden” should be used instead. The useful codes seem to be as follows.

400 “bad request” for invalid query-parameters, including invalid ADQL.

403 “forbidden” for security failures.

404 “not found” for invalid URIs.

503 “service unavailable” for services not accepting new jobs.

500 “internal server error” for anything else.

All the other codes seem to have specific semantics that do not match our use case.

3 Metadata and registration

3.1 Specification

3.1.1 Table metadata

A TAP/QL service SHALL provide metadata describing the relational schema against which queries can be written. These metadata SHALL be provided on demand according to the VOSI standard: i.e. the service shall provide a web-resource for which the representation is an XML document containing suitable elements from the VODataService [4] XML-schema. Please refer to Appendix A for a discussion of “suitable elements”. The client should obtain the representation using HTTP-get.

The metadata MUST list all the standard functions of ADQL in addition to any functions defined locally.

3.1.2 Service metadata

A TAP/QL service SHALL provide service metadata according to the VOSI standard [5]. These metadata consist in a sequence of capability elements as defined by the VOResource schema [6].

The service metadata SHALL include one capability specific to the TAP/QL protocol. This SHALL be written as an XML element of type *{http://www.ivoa.net/xml/VOResource/1.0}Capability* with its *standardID* attribute set to *ivo://ivoa.net/std/TAP/QL*. This capability SHALL include an interface element of type *{http://www.ivoa.net/xml/VODataService/1.0}ParamHTTP* in which the value of the *accessURL* element is the URI for the root web-resource specified for the query protocol, above.

A TAP/QL SHOULD make its service metadata available as a web resource, as defined by VOSI.

3.1.3 Availability metadata

A TAP/QL service SHOULD provide availability metadata according to the VOSI standard. On each availability request, the service SHOULD check that it is still in contact with its underlying database and report itself as unavailable if not.

3.2 Discussion

Although the tables part of VOSI is optional for services in general, it is mandatory for implementations of TAP/QL.

“Functions” here refers to functions that may be included in ADQL that are not inhertited from SQL 92.

VODataService 1.0 defines elements to describe tables, columns of tables and catalogues (i.e. groups of related tables). It does not address relational schemata, functions or joins. We hope that the missing elements can be added in VODataService 1.1.

This is a sample capability defining the TAP/QL endpoint:

```
<capability standardID=”ivo://ivoa.net/std/TAP/QL”>
  <interface xsi:type=”vs:ParamHTTP” role=”std”>
    <accessURL use=”base”>http://host/service</accessURL>
  </interface>
</capabilty>
```

Here, the access URL matches the example URIs shown above. Note that it does not end in */sync* or */async*: the client must add those suffices to find the query endpoints.

Normally, a TAP/QL service should emit its service metadata on demand from a suitable, VOSI endpoint. However, this standard allows the service to define the metadata but not to make them available via a web resource. In this case, one assumes that the metadata are somehow entered into the registry by some other means and that all clients look in the registry to find the endpoints.

4 Security

4.1 Specification

A TAP/QL service MAY require users to identify themselves and MAY further restrict access according to the user's identity, either denying access altogether or limiting the extent of the query that may run. If a TAP/QL service authenticates users' identities it MUST do so according to the IVOA single-sign-on standard [7] using the method *TLS-with-client-certificate* defined in that standard.

4.2 Discussion

“TLS-with-client-certificate” means that the query endpoints are in the https URI-scheme, that the encryption protocol is TLS (as opposed to the earlier SSL versions) and that the client is expected to authenticate with the user's certificate chain and private key.

TAP/QL services are not allowed to use password protection on their standard, registered query-endpoints. If passwords are needed for local operations, the service can have them on some extra, non-standard endpoints.

Appendix A: Draft specification for VOSI table-metdata

At the time of writing, the table-metadata part of VOSI has not been drafted. Here is an outline of what is needed.

Writing an ADQL query for a TAP service requires knowledge of the catalog(s), schema(s), table(s), column(s), function(s), and possible join(s) between tables. Catalog, schema, and table metadata MUST include names that follow the allowed syntax in the ADQL specification [1] and SHOULD include descriptive text aimed at users and a short display-name for each.

Column metadata MUST include a name that follows the allowed syntax in the ADQL specification (Section ?), the data type stored in the column, and the units for numeric values. Column metadata SHOULD include descriptive text and a UCD to describe the semantic meaning and a short display-name.

Function metadata MUST include the list of supported functions that are defined in the ADQL specification; no other metadata should be included for standard ADQL functions. Metadata for additional service-specific functions MUST include a name that follows the allowed syntax in the ADQL specification [1], the data type of the return value, and the units for numeric return values. It MUST also include the name, position, data type and units for each parameter. It SHOULD include descriptive text to describe the semantics of the operation.

Join metadata specifies that tables can be joined and how to specify the join. It MUST include a name and one or more pairs of fully qualified column names (or identifiers?). It SHOULD contain descriptive text to describe the semantics of the join.

A different method name/endpoint for each metadata type is used with an HTTP GET call to retrieve the XML encoded result. Method names/endpoints should be defined in coordination with the VOSI authors (GWS WG), and follow the example of the AstroGrid's DSA prototype: e.g. http://wfaudata.roe.ac.uk/ukidssWorld-dsa/wsa/vosi/tables.

The Table and Column metadata MUST be encoded in XML using the "Table" and "TableParam" elements from the VODataService schema [4], wrapped in a "Tables" element as specified in the XSD schema provided.

The Function metadata MUST be encoded in XML using the "InputParam" element from the VODataService schema [4] inside the defined "Function" element, wrapped in a "Functions" element as specified in the XSD schema provided.

The encoding of the Joins metadata is still to be decided, but the format will also follow a similar XML encoding.

References

[1] P. Osuna & I. Ortiz (eds.) ,*Astronomical Data Query Language Version 2.00*
http://www.ivoa.net/Documents/cover/ADQL-20080430.html

[2] G. Rixon, *Universal Worker Service v0.3*
http://www.ivoa.net/internal/IVOA/IvoaGridAndWebServices/UWS-0.3.pdf

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

[4] IVOA registry working-group, *VODataService: VOResource extension for Data and Services v1.0,* http://www.ivoa.net/xml/VODataService/VODataService-v1.0.xsd

[5] G. Rixon (ed.) *IVOA Support Interfaces: Mandatory Interfaces*, (VOSI v0.4)
http://www.ivoa.net/internal/IVOA/IvoaGridAndWebServices/VOSupportInterfacesMandatory-0.4.pdf

[6] IVOA registry working-group, *VOResource: resource description v1.0*, http://www.ivoa.net/xml/VOResource/VOResource-v1.0.xsd

[7] M. Graham & G. Rixon (eds.) I*VOA Single-Sign-On Profile: Authentication Mechanisms Version 1.0*, http://www.ivoa.net/Documents/latest/SSOAuthMech.html