International

Virtual

Observatory

Alliance

IVOA Table Access Protocol Parameterized Query Language

Version 0.2

IVOA Internal Working Draft 2009 May 20

This version:

PQL-0.2-20090520

Latest version:

Not yet issued

Previous version(s):

Authors:

TBD

Contributors:

TBD

Abstract

This document describes the Parameterized Query language (PQL). PQL has been developed for parameter-based table queries as part of the DAL Table

Access Protocol (TAP). This document formalizes the syntax and meaning of PQL as a general parameter-based query language for querying tabular data. PQL complements the Astronomical Data Query Language (ADQL), with PQL providing a higher level, data model based query facility optimized for specific astronomical use cases, while ADQL provides a more expressive and powerful but lower level general query language based upon SQL.

Status of This Document

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

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

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

Acknowledgements

"Ack here, if any"

Contents

1 Introduction

The Parameterized Query Language (PQL) provides a simple parameter-based mechanism for querying tabular data within the context of the IVOA Data Access Layer (DAL) Table Access Protocol (TAP). PQL complements the Astronomical Data Query Language (ADQL), providing optimized support for common queries of individual astronomical catalogs as well as table metadata, while ADQL provides a general language based upon the Structured Query Language standard (SQL), providing a powerful and general mechanism for querying relational databases. When used within the context of TAP, PQL and ADQL provide alternate ways to pose a query, with both sharing the same TAP query execution and output processing engine as well as a common service interface.

Unlike the more generic ADQL, PQL includes built-in support for astronomical data models, in particular the DAL *generic dataset* data model (sometimes also referred to as the *Observation* data model). Most astronomical catalogs as well as many collection-specific *index catalogs* implement portions of the generic dataset model, in particular the concept of spatially indexed data records. Data can also be described using the full generic dataset model, permitting a richer range of queries based upon a variety of standard attributes such as data collection, dataset type, time of observation, spectral coverage, object name, object classification, dataset identifier, spatial resolution, and so forth.

Parametric queries are simple to express and to implement for cases where the data model is sufficiently well defined for the data to be queried, hiding many of

the details required to pose and evaluate the query. In this sense PQL, while simple in terms of interface, provides a higher level of abstraction than richer and more detailed languages such as ADQL.

While the parameter query can be used to query any individual data table or to query table metadata, it provides enhanced support for querying both astronomical catalogs as well as index catalogs, based upon the generic dataset data model. Astronomical catalogs typically describe sources which have a spatial position or other physical parameters which can be directly queried with the parameter query. Index tables describe and link to other data, for example images, spectra, other catalogs, services used to access such data, or even instrumental data. A table describing linked datasets using standard generic dataset metadata is a special case of an index catalog, but custom collection-specific index tables often share a subset of the generic dataset metadata and can be directly queried as well.

While the parameterized query described herein could potentially be used in contexts other than TAP, it is specific to querying tables and table metadata and as specified here is assumed to execute within the context of a TAP service, relying upon the TAP service for query execution, table uploads, output processing, and other TAP-specific functionality. The version of the parameter query specified herein is intended to be used only with version 1.0 of the TAP specification

> *[The intention is to produce a new version of the parameter query whenever TAP is updated, and always keep these two versioned and closely linked interfaces in sync - Ed]*

2 General Parameter Rules

2.1 Single-Valued Parameters

Parameters which are single-valued have a constant value, the semantics of which are defined by the individual parameter. The value may not use any reserved characters unless these are URL-encoded 5.1).

2.2 Multi-Valued Parameters

Parameters which are multi-valued (list valued, such as positions) use the comma (“,”) as the separator between successive items in the list. Embedded white space is not permitted. List items may be constants or ranges. A list-valued parameter composed of ranges is referred to as a range-list parameter (2.5).

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.

2.3 Range-Valued Parameters

Parameters thats specify a range of values 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,a range consisting of all values from 5E-7 to 8E-7 inclusive would be:

```
Example: 5E-7/8E-7
```

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, an open range which accepts all values less than or equal to 5 would be encoded as shown below.

```
Example: /5
```

Range values can only be used with parameters which specify numeric and date values. [*String ranges could potentially be defined as well but advanced string processing would probably require a different and more complex facility.*]

2.4 Qualifiers

If specified by the definition of a particular parameter, a single-valued parameter, range, or list **may** be qualified by appending the character “;” (semicolon) followed by a qualifier string. This could be used to specify an alternate coordinate system, e.g.

```
Example: 180.0,1.0;GALACTIC
```

could specify a position in galactic coordinates. In some cases, multiple semicolons may be used to delimit separate sub-lists or clauses within the parameter value.

2.5 Range-List Syntax

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.

2.6 Parameter Indirection

The value of any parameter can be taken indirectly from an external object rather than specified directly, if specified in the description of an individual parameter. The '@' character is used to denote parameter indirection:

```
Example: POS=@something
```

The meaning or interpretation of the referenced symbolic value ('@' target) is defined by the individual parameter.

2.7 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. Whether or not a null parameter value is significant is defined by the individual parameter. If only the parameter name is given, e.g., "POS", it is the same as if the parameter was not specified, and the parameter value is unset.

2.8 Inapplicable Constraints

Unless otherwise specified, if a parameter constraint is specified which does not apply to the referenced table the constraint is ignored without error. This allows the same query to applied to multiple tables without error, automatically applying the specified query constraints only where appropriate. Ignoring inapplicable constraints in this fashion is only done if the constraint parameters refer to generic data model attributes which may or may not apply to a given table. If a table or field is explicitly referenced but does not exist an error should be reported.

2.9 Case of parameters

Parameter names **must not** be case sensitive, but parameter values **must** be so. While the value of a parameter is considered case-sensitive by the parameter handling mechanism, whether or not a parameter value is treated in a case sensitive fashion is specified by the individual parameter. In this document, parameter names are typically shown in uppercase for typographical clarity, not as a requirement.

2.10 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 the parameter. Clients **should not** repeat parameters in a request.

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.

Positive, negative and zero values are allowed unless explicitly restricted by a service specification making use of PQL.

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.

3 Parameter Query Operation

How a parameter query is posed within the TAP context is defined by the TAP service interface. The parameters used to compose a parameter query are described below. The parameter query executes within the runtime context of the TAP service and relies upon TAP for all functionality not specified explicitly herein. A brief summary the shared TAP context used with parameter queries is given in section TAP Context below (3.2).

3.1 Standard Parameters

The parameters specific to the parameter query are defined below. Unless otherwise specified the service **must** implement all the parameters defined in this section.

3.1.1 POS, SIZE

The POS and SIZE parameters provide an easy to use, optimized facility for performing spatial queries of astronomical catalogs, index tables, or other tables which are spatially indexed. Spatial query constraints are supported only for tables which contain positional information; most astronomical catalogs or index tables are of this type. It is an error if a positional constraint is specified but cannot be applied to the specified target table.

POS and SIZE define a circular search region in the specified coordinate system (default ICRS). A table which supports the POS and SIZE parameters implements them as a query constraint for tables containing records tagged with spatial positions.

The coordinate values for POS are specified in list format (comma separated) with no embedded white space, as defined in section 2.2 .

```
Example: POS=52,-27.8
```

The POS parameter defaults to right-ascension and declination in decimal degrees in the specified 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].

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

It is an error if the specified coordinate system reference frame is not supported by the service.

POS may optionally point to a separate table of positions, used to specify positions for a *multi-position* query (see section 4.2 Multi-Position (Multicone) Queries).

The SIZE parameter specifies the angular 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.

> [*Note: BAND and TIME have been moved to the discussion of the Generic Dataset query (4.4) below. – Ed.*]

3.1.2 REGION

The service **should** implement a REGION parameter, used to provide a more general spatial search capability than can be defined using POS and SIZE. The value of REGION **must** be a STC/S (REF) region specifier, e.g.

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

In the example above the embedded spaces are shown for clarity, but in real use they **must** be URL encoded.

If POS,SIZE and REGION are both specified in the same query, they both apply. In this case REGION defines a mask used to further qualify the circular region or regions specified by POS and SIZE. This is most useful for multi-position queries (Multi-Position (Multicone) Queries) to cross-correlate two tables within the specified mask region.

3.1.3 SELECT

The SELECT parameter specifies the fields to be returned by the query, specified either as a comma delimited list of table 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).

```
Example: SELECT=ra,dec,flux
Example: SELECT=$ALL
```

By default only the $STD fields are returned. The “primary” fields are specified on a per-table basis, and define a subset of the most important table fields chosen by the service implementer. This is used to provide a more readable view of very wide tables. The service **must** permit *$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 $STD view is defined for a table the service **should** ignore *$STD* and merely return all table fields.

The names of available table fields may be determined by a prior table metadata query (Table Metadata Queries), or may be specified by a pre-defined schema such as the TAP_SCHEMA (TAP Schema).

3.1.4 FROM

The FROM parameter indicates the target table to be queried. Only a single table name is allowed. Any table name permitted by TAP is legal here.

```
Example: FROM=hdfv2
```

There is no default value and it is an error if no table name is specified. The minimum possible parameter query would specify only FROM, resulting in the entire table being returned in the default output format, displaying the default (SELECT=$STD) view of the table.

3.1.5 WHERE

The WHERE parameter is used to specify optional filtering constraint(s) to be applied to the query target to determine which records are returned. By default all table records are returned. The WHERE parameter may be combined with other query constraints such as POS and REGION to further refine the query.

The syntax of the WHERE parameter value 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.

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

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

> *[NOTE: In the WHERE parameter, the semi-colon is used as the list delimiter to separate constraints, while in the range-list section above comma separates list items and semi-colon separates an item from a qualifier. That is, in the BNF below field-list uses semi-colon and the other lists use comma. Clearly the separators need to be different, but maybe re-using the qualifier separator is not such a*

good idea... just in case we want to allow (now or in future) the use of qualifiers in the WHERE parameter. -Ed.]

The names of available table fields may be determined by a prior table metadata query (Table Metadata Queries), or may be specified by a pre-defined schema such as the TAP_SCHEMA (TAP Schema).

The WHERE syntax has deliberately been kept simple as ADQL already provides a fully general expression evaluation capability which should be used for more advanced 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 condition. 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 (Multi-Valued Parameters) for the WHERE expression. An effort has been made to define a minimal set of meta-characters so as to minimize the need for URL encoding – specifically not using reserved characters in the URI and URL query string syntax (e.g. =, &, ?, #). 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> ]
```

Here 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.
- Field expressions are of the form <field-name>’,’<value> (meaning field-name=value), where <value> is a single value, a range, or a list of single values or ranges all of the same type. Constraint expressions within the overall WHERE expression are combined with a logical AND operation. Values within a range-list are combined with a logical OR operation, i.e. a range or list for a specific field gives a list of acceptable 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, and 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 (with extra spaces for readability here):

```
WHERE=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
```

Note the special treatment of the jmag constraint; the list of ranges are combined with the OR operator while the jmag constraint itself is combined with the others with the AND operator.

3.2 TAP Context (informative)

When a parameter query executes within the context of a TAP service it relies upon the TAP service for much of its functionality, including most aspects of query execution, and all output processing. For the most part the parameter query is merely an alternative way of posing a query, with all other processing and execution context in common with other query methods. The TAP specification should be referred to for a full specification of the TAP context, but we introduce the functionality most relevant to the parameter query here.

3.2.1 Common Parameters

The following parameters are provided by TAP and are shared by all query methods, including the ADQL and parameter queries.

FORMAT	The format of the output table, e.g., votable, csv, tsv, text, and html (default VOTable).
UPLOAD	Used to upload a table referenced by its URL. Tables can also be uploaded in-line with the query. Uploaded tables can be used like any other table in a query, for example to input a list of positions for a multi-position query. Uploaded tables are referred to in queries as TAP_UPLOAD.<*table_name*>.
MAXREC	The maximum number of table records to be returned by the query. An overflow indication is returned to the client if overflow occurs. Used to avoid returning arbitrary amounts of data to the client, unless the client explicitly requests a large response.
MTIME	Used to find only table records which have been modified within a given time interval, specified as an open or closed range. “Modified” records are table records which are inserted, deleted, or updated.
RUNID	Used to tag service requests with the job ID of a larger distributed job of which the current request may be only one part.

While all of these parameters should be transparent to a parameter query implementation, they are important to client applications using a TAP parameter query.

3.2.2 TAP Schema

The TAP Schema defines standard metadata describing the *tableset* available from a TAP service (a tableset is the set of tables provided by a given TAP service). The TAP Schema is represented in TAP as a set of tables which can be queried using the standard table query interface. The tables are part of a database "schema" known as the TAP_SCHEMA. The TAP_SCHEMA currently defines the following metadata tables:

TAP_SCHEMA.schemas	Describes the database *schemas* defined by the tableset. A schema is a logical grouping of tables which are related in some fashion.
TAP_SCHEMA.tables	Describes all tables in the tableset.
TAP_SCHEMA.columns	Describes all columns (table fields) of *all* tables in the tableset (in one big table).
TAP_SCHEMA.keys	Describes all primary and foreign key relationships linking tables within the tableset.

To determine the schemas (table groupings) or tables available from a TAP service, or the columns of a given table, a client can simply query the TAP_SCHEMA using the standard TAP query facilities. The parameter query provides specialized facilities for this purpose (see section Table Metadata Queries). For a description of the detailed content of the TAP_SCHEMA see the TAP specification.

3.2.3 Table Names

All table names used in parameter queries **must** conform to the same table name specification as is used in TAP. In particular, a fully qualified table name has the form

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

where *catalog_name* is the the name of the DBMS 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*. To reference tables within a specific schema the schema name should be included, e.g., TAP_SCHEMA.tables, or TAP_UPLOAD.foo.

3.2.4 Table Field (Column) Names

Table field names as used in queries should equate to the “column_name” specified in the TAP_SCHEMA.columns metadata for the given table. While in a given implementation table field names (or table names) may be case insensitive, this cannot be relied upon, and it is best to use the exact column name returned by a prior query of the TAP_SCHEMA for the table.

3.2.5 Asynchronous Execution

By default parameter queries execute synchronously, and upon successful execution return the output table directly to the client. Parameter queries may also execute asynchronously using the UWS mechanism provided by the main TAP service.

4 Advanced Topics

While section 3 specifies the basic usage of all parameters for the parameter query, they can be used with additional semantics to implement the advanced functionality described in this section.

4.1 Cone Search Queries

The parameter query **must** support use of the POS and SIZE parameters as a spatial constraint for “cone search” type queries, in cases where the table to be queried contains an appropriate spatial index. It is an error to attempt to apply a spatial constraint to a table which does not support one.

For example, the following would perform a cone search of table “fp_psc” using the specified position and search region diameter, selecting only objects for which the J magnitude is less than or equal to 10 (here and below we omit the preamble of the GET required to invoke the parameter query operation, showing only the parameters specific to the parameter query):

```
FROM=fp_psc&POS=180.0,0&SIZE=0.2&WHERE=j_m,/10.0
```

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

> [*Note: We may want to specify UCDs for the primary output columns, as was done for legacy cone search. – Ed.*]

4.2 Multi-Position (Multicone) Queries

The parameter query **may** implement the *multi-position (“multicone”)* query, used to simultaneously query a table at an arbitrary number of spatial positions. The multi-position query generalizes POS, SIZE to a table of positions, allowing an

arbitrarily large number of spatial queries to be executed simultaneously.

The multi-position query performs what is essentially a spatial cross-match of two tables, optionally combined with a WHERE filter on the target table to further refine the query and reduce the volume of data produced. In a typical scenario the user uploads a list of the positions of their favorite objects, then executes the multi-position query against some data table. The resulting table can be downloaded to the desktop or put into a VOSpace. While the multi-position query does not provide an astrophysically informed crossmatch, being little more than nearness in the sky, further processing can be done on the resulting table to weed out inappropriate matches, and thereby create a true crossmatch. Hence a simple multi-position query can participate in a sophisticated multi-parametric distributed cross match, without the need to complex cross-matching algorithm at the remote site where the target table is stored.

- ### Query Input

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

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

```
POS=@TAP_UPLOAD.positions&SIZE=0.2
```

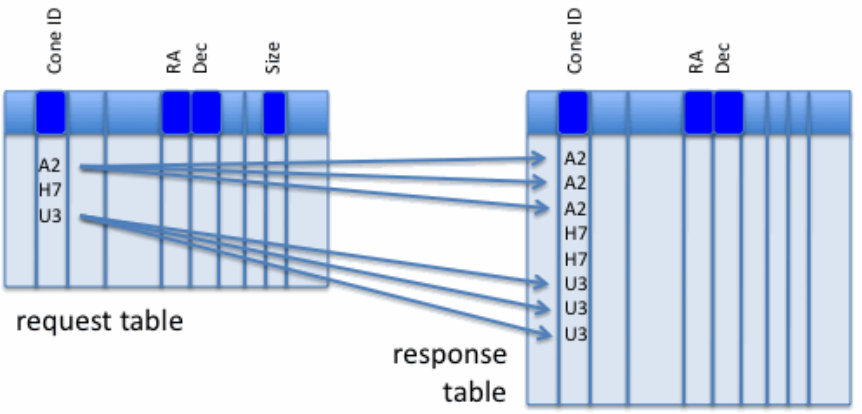
In the most general case any table containing position information may be used. For example we could use the 2MASS point source catalog from our earlier examples, assuming a copy is available to the service. This table contains nearly half a billion sources, so the REGION parameter would be used to apply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

In the uploaded table, columns must be identified by the service to be used for ConeID, sky position, and cone diameter. These can be specified either by UTYPE, by UCD or by column name, as shown in this table:.

Quantity	*utype*	*ucd*	*Name*
Cone ID	src:Position.ID	meta.id	ConeID

First coordinate (e.g., Right Ascension), degrees	src:Position.Coord1	pos.eq.ra	RA
Second coordinate (e.g., Declination), degrees	src:Position.Coord2	pos.eq.dec	DEC
Diameter of search region	src:Position.Size	pos.angDistance	SIZE

UTYPE and UCD **should** be used for reliable field identification when possible. UTYPE must be used if the anything other than equatorial coordinates are input. If neither UTYPE nor UCD can be used for field identification the service will try to use the field names shown.



Request and response tables for the multicone service. The uploaded table must have a unique identifier (Cone ID), RA, Dec (or other coordinate) for each source, and may also have a cone size (diameter) for each search. The response table must show the identifier of the source it is close to (cone ID) in addition to any other identifier.

The Multicone service will try to find which columns of the input table to use in the following method. If UTYPE is available and its value is in the table above, then that column of the table is used for that purpose. If the UCD of a column contains one of the UCDs above as a substring, then that column of the table is used for that purpose; however, the UCD that also contains the string "meta.main" will take priority of there is more than one match. If neither suitable UTYPE or UCD is available, then a precise match of the Name column will identify the relevant columns.

In the case of VOTable, the following example would provide all four of the columns required for valid input:

```
<FIELD name="SerialNo" utype="src:Position.ID"  />
<FIELD name="otherRA"  ucd="pos.eq.ra"          />
```

```
<FIELD name="mainRA"   ucd="pos.eq.ra;meta.main"/>
<FIELD name="DEC"                               />
<FIELD name="diam"     utype="src:Position.Size"/>
```

Notes for VOTable inputs:

- Table columns can appear in any order.
- The ConeID FIELD must have datatype="char" and arraysize="*"
- The RA and Dec fields must have datatype="float" or datatype="double"
- Other FIELDs can also appear in the table
- The ID attribute of VOTable is not used in this definition

In the event that only field names can be used, the following would provide valid input:

```
ConeID, a, b, RA, DEC, c, d, SIZE
```

As noted above, all positions may share the same search region diameter, or the search region diameter may be specified separately for each region. If there is a parameter SIZE in the input request, then the table column for SIZE will not be used, even if it exists. Otherwise, there **must** be a SIZE column found in the input table.

- ## Form of the Output Table

Where possible output table columns **should** be assigned UCDs 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.

There are two types of identifiers for the output of a multicone search:

- The ID of the catalog entry
- The ConeID of the input cone in which this entry lies

In general, each row of the output table will be a copy of a row from the queried table; there may be a reduction of the number of columns if SELECT=$STD. But there must be a new column also, which is the ConeID from the input table that is the reason for inclusion.

The ID of that catalog entry must have UCD = meta.id or have UTYPE = src:Position.ID, and the originating ConeID column must have either UCD = meta.id.cross or have UTYPE = src:Position.ID.cross, as shown in this table:

Quantity	*utype*	*ucd*
Catalog ID	src:Position.ID	meta.id

Cone ID	src:Position.ID.Cross	meta.id.cross

4.3 Table Metadata Queries

The parameter query **must** implement basic table metadata queries based upon the TAP_SCHEMA (TAP Schema). The service **must** support listing all tables provided by the service, listing all the fields of a single table, as well as providing the full tableset metadata in both XML and VOTable format. A more advanced TAP service **may** support general table metadata queries using the full parameter query interface. Some simple examples of table metadata queries follow.

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

```
FROM=TAP_SCHEMA.tables
```

To list all columns of table “fp_psc”:

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

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

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

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

A TAP service **must** support the above primary table metadata queries. A TAP service **may** support more advanced table metadata queries which allow other parameters to be used to further refine a table metadata query. Some examples of advanced table metadata queries follow.

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

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

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

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

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

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

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

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

As above, but return tableset metadata in VOTable format:

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

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

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

List only tables modifed since July 4 2005:

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

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

4.4 Generic Dataset Queries

The parameter query implementation **may** support queries of tables using the *Generic Dataset* (GDS) data model. While POS,SIZE and REGION allow queries of tables based upon a simple model of spatial position, GDS defines a full model capable of describing a range of standard attributes of any “dataset” (image, spectrum, table, etc., as well as complex data associations and even the more generic attributes of instrumental data). The GDS query in a sense represents a generalization of the simple cone search to encompass all generic dataset metadata.

As with all TAP parameter queries the GDS query is used to query tables, but due to the requirement that the target table implement some subset of the GDS model (not necessary for a query but necessary for the GDS query to be useful), the GDS query will generally be limited to queries of *index tables* of some sort. An index table provides a uniform description and index of all data contained within a *data collection*. Typical data collections might be a survey or instrumental data collection, a user-defined collection associating an arbitrary set of data objects, or a master index of all the data at a site. Index tables may contain arbitrary content and still be queried with TAP, using either the parameter

query or ADQL, but must contain at least some fields containing GDS metadata for queries based upon the GDS model to be used.

Index tables not only describe the datasets comprising a data collection, but typicallly include *data links* which can be used to download the associated data object, upload or link the object to a VOSpace, or invoke another service to access the object (for example an SIA service could be used to cutout portions of the referenced image or data cube).

While the most generic, physics based aspects of the GDS model can be used to query most astronomical source catalogs (spatial queries at least are usually possible), the real utility of the GDS query becomes evident when it is used to query index tables. Primary datasets may be associated within the query to model *complex data*. Data links may be followed to browse or access any associated data products. The parameter query provides the simplest way to query GDS index tables as the parameter mechanism provides explicit support for querying GDS data model attributes. The ADQL query mechanism can also be used once support for UTYPE-based queries becomes available.

> [*We could continue at this point to define the parameters for the GDS query, however this is not required for the initial version of the TAP param query, and while the GDS metadata and associated query is already in use in DAL2 interfaces such as SSA, more work as well as prototyping is needed on new features such as data linking before the GDS query is ready for standardization. Hence the remainder of this section merely discusses the aspects of the GDS query where further work is needed in the next phase of development. – Ed.]*

The most basic parameters defined by the GDS query are POS,SIZE, BAND, and TIME. Equally fundamental for a dataset query which can discover many types of data is the dataset type, e.g., image, spectrum, table, or any other VO type. Extension of the dataset type to characterize external data (arbitrary data from the local archive) is also possible.

Parameter	*Description*
POS, SIZE	Spatial position in the specified coordinate system.
BAND	Spectral coordinate, range, or band identifier.
TIME	Time coordinate or range.
POL (?)	Polarization type [*may be more useful for data access. – Ed]*
DSTYPE (?)	Dataset type (table, image, spectrum, custom, etc.)

A more complete set of GDS parameters (from the DAL2 architecture document [2] and originally SSA) are given in the following table.

Parameter	*Description*
SPECRP	Spectral resolving power
SPATRES	Spatial resolution
TIMERES	Temporal resolution
TARGETNAME	Name of observed target
TARGETCLASS	Astronomical object type
ASTCALIB	Level of astrometric calibration of data
WAVECALIB	Level of spectral coordinate calibration of data
FLUXCALIB	Level of flux calibration of data
PUBDID	Publisher dataset identifier (e.g., from data center)
CREATORDID	Creator dataset identifier (e.g., assigned by survey)
COLLECTION	Data collection name or identifier

Additional refinement of the GDS query is likely as the concept is developed beyond just what was done for SSA and the generic dataset model.

An issue with the GDS query is what to do if a parameter is specified but the metadata required to evaluate the parameter is not defined. The general rule in this case for data discovery queries in DAL is to ignore the constraint without error if it cannot be applied. Data queries thus err on the side of including data if a query constraint cannot be applied, leaving it up to the client to further refine the query. This has the advantage of allowing queries to be posed without requiring detailed knowledge of what metadata is available for a given index table (this differs from a simple table query where the referenced table fields must exist).

5 Use with HTTP (informative)

An HTTP service which accepts PQL as input is constrained by the general rules for use of HTTP, which are contained in IETF RFC documents. This section collates some of issues in using PQL with such services. For authoritative specifications, please refer to the original RFCs.

The PQL parameters described in this document may be mapped directly to HTTP request parameters in the query string portion of the URL (HTTP GET) or included in the request (HTTP POST). As noted above, it may not be necessary to URL encode the parameter values in all cases, but it is generally good practice to do so.

5.1 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 HTTP URLs

Character	Reserved usage
?	Separator indicating start of the URL query string
&	Separator between parameters in the query string
=	Separator between name and value of a parameter
#	Separator indicating start of a URL fragment (anchor?)
, / ;	Separator between individual values in range or list parameters

For example, while PQL does not specify any use for the fragment or anchor (#) separator, any parameter value contains this character must be URL encoded to be legally included in a URL.

6 References

[1] P. Dowler, G. Rixon, D. Tody, DAL-WG, *Table Access Protocol,* IVOA Internal Working Draft, May 2009.

[2] D. Tody, F. Bonnarel, M. Dolensky, J. Salgado, DAL-WG, *IVOA Data Access Layer Service Architecture and Standard Profile,* IVOA Note 5 October 2008. http://www.ivoa.net/internal/IVOA/SiaInterface/DAL2_Architecture.pdf

[3] 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

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

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

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

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

[8] 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

TODO: add references to previous DAL services (e.g. SSA)