International

Virtual

Observatory

Alliance

Table Access Protocol Design Analysis

Version 0.1

IVOA Note 2007 September 20

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

Author(s):
D.Tody, ohers TBD

Abstract

This document presents a proposal for a basic Table Access Protocol (TAP) interface, noting motivations, describing interface elements which are thought to be understood, and identifying issues which are as yet unresolved. This attempts to build upon the work done by the ESAC group within the VOQL-TEG in early 2007, while conforming to the basic service profile and common service elements developed by the IVOA DAL, DM, Registry, and GWS working groups, and incorporating experience gained by the NVO and CADC with the SkyNode prototype and various related data center protocols.

Status of This Document

This is an IVOA Note. The first release of this document was 2007 September 20.

This is an IVOA Note expressing suggestions from and opinions of the authors. It is intended to share best practices, possible approaches, or other perspectives on interoperability with the Virtual Observatory. It should not be referenced or otherwise interpreted as a standard specification.

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

This document presents a proposed draft interface for the IVOA Table Access Protocol (TAP), describing those interface elements which we feel are fairly well understood while identifying issues which need further study. This is intended only as a draft to expose the issues and provoke discussion. The draft interface proposed here is based upon that developed in the spring of 2007 by ESAC and the VOQL-TEG, as well as work done within the DAL working group in the same time period, and reflects the experience of the NVO project with the earlier SkyNode prototype and various related data center query interfaces.

The following goals are addressed in the draft TAP interface presented here:

- The primary focus of TAP is to provide a standard interface for ADQL (SQL)-based queries, including providing support for large queries and distributed queries, and multi-table operations.

- At the same time TAP should define a minimal implementation which makes it as easy as possible for a small data provider to publish and individually query a few tables; ideally this will ultimately replace the legacy Cone Search interface. This minimal implementation need not require ADQL support, although a SQL DBMS might still be used at the back end.

- Both data access and metadata access are essential for any data access interface, and should be provided natively within the interface.

- Scalability is required, in particular, support for multi-position queries, where a table containing potentially thousands of source positions is input as part of the query (in effect this provides the first stage of a distributed cross match capability).

- Ultimately, integral support for asynchronous execution, data staging (e.g., via VOSpace), and SSO authentication are required, based upon IVOA GWS standards in this area, although the simplest version of the interface may not require any of these.

- For reasons of consistency and to enable code re-use, the basic form of the TAP interface should be consistent where possible with the other IVOA DAL interfaces.

- Registry integration is required to register service capabilities and possibly some information about the tables available via a TAP service, in order to support data and service discovery at the registry level.

2 Interface Summary

The basic TAP service interface described here is composed of multiple independent service operations. HTTP is adopted as the basis for the service protocol, using both GET and POST for service operations where appropriate, consistent with REST semantics (other protocols such as SOAP could optionally be supported as well). Data is returned in a variety of output formats including VOTable, CSV/TSV, and native XML (support for other formats such as HTML, FITS binary table, etc., is also possible but is not addressed here). A restricted subset of the SQL information schema, with the addition of VO specific extensions (UTYPE, UCD, etc.) is proposed for describing database and table metadata.

3 Service Operations

The following set of TAP service operations are suggested:

- AdqlQuery — ADQL-based queries, full functionality
- SimpleQuery — Simple parameter-based queries, metadata queries
- GetCapabilities — Return metadata describing the service
- GetAvailability — Monitor runtime service function and health

It appears that everything we want to do, including both synchronous and asynchronous ADQL queries, very large queries, multi-position queries, data staging, simple cone search type queries, general metadata queries, registry integration and service capability querying, service monitoring, etc., can be done with these few operations. This service interface is also thought to be adequate to support development of a higherl level large scale distributed cross match portal or application which relies upon TAP services for access to remote data.

There is some discussion of whether we need a "Simple" (non-ADQL) query, since in principle ADQL can provide everything required. Everyone agrees that the main focus of TAP should be to support ADQL-based queries. However, requiring ADQL, REGION, UTYPE-based queries, etc. just to do a cone search, or a simple query of a single data or metadata table, essentially requires a full-up ADQL implementation to do anything at all, and would violate our requirement that a minimal TAP service be defined which is easy for a small data provider to implement. For these reasons we feel that a SimpleQuery operation (described below) is still warranted.

3.1 AdqlQuery

The *AdqlQuery* operation provides a capability for ADQL (SQL)-based queries. Most of the complexity of the AdqlQuery operation involves the specification and processing of the ADQL expression itself; the service interface itself is fairly straightforward. The AdqlQuery operation has the following characteristics:

- Provides a capability for general ADQL-based queries, including large queries, multi-position queries, multi-table joins, etc.

- Both GET and POST versions are defined. The GET version permits only synchronous execution, and URL-encodes the ADQL query string, allowing arbitrary SQL syntax to be used. Both versions share the same parameters and semantics, although the GET version is a subset of what is permitted with the POST version.

- Use of the POST version is required for asynchronous queries, for multi-position queries which require upload of a source table, or for queries which are too large to compose as a GET.

3.1.1 AdqlQuery Parameters

The following parameters are defined for the AdqlQuery operation::

QUERY	The query string (ADQL; URL-encoded)
FORMAT	Output data format (VOTable, CSV, XML, etc.)
<*staging Params*>	Only used in POST version; for VOSpace
<*async Params*>	Only used in POST version; for driving UWS
MAXREC	Maximum records in the output table
RUNID	Pass-through; used for logging
(others TBD)	

The query string specifies all table-related aspects of the query hence no additional parameters are required to specify the query. Only ADQL queries are address here; if other SQL dialects or native SQL are supported by a service, this could be added as an optional capability with the dialect specified by an additional parameter (the use of native host SQL features within the ADQL query might however be a better approach).

FORMAT specifies the output format as for other DAL interfaces, with VOTable being the default output format.

MAXREC, provided primarily for synchronous queries, can be used to increase the maximum number of output records permitted in a query to prevent overflow. Overflow is indicated in the output table with the QUERY_STATUS INFO element, as for other DAL interfaces. MAXREC is unrelated to the SQL TOP construct.

3.1.2 UTYPE and UCD in Queries

We suggest that, rather than provide a separate query operation for UTYPE or UCD-based queries, these be handled instead in the process of field name resolution within a query. Although how it is handled is up to the internal

processing of a query, all field references in queries normally resolve to individual table fields. By default table fields are secified by name, using the field or column name given in the table metadata. If a field name in a query includes a UTYPE reference to a field of a data model, this is resolved by the service (if it supports the associated data model) into a literal table field, and processing proceeds normally. UCD can be handled the same way, and can be considered (for the purposes of table name resolution), as a special case of UTYPE. This is discussed further in section 3.2.3 below.

The proposed UFI syntax could also be used to specify table fields in terms of data model UTYPEs. While this could be a useful feature for automated resolution of UTYPEs, strictly speaking it is not required as the client could query the table metadata and resolve the data model reference to a literal table field on the client side. UTYPE and literal field name references could be mixed within the same ADQL expression.

3.1.3 Multi-Position Queries

Multi-position queries are required for scalability, as querying repeatedly by individual spatial positions is too slow when thousands of positions are involved (this is true for other interfaces such as SIA and SSA as well). The case of querying by spatial position requires special treatment as it is multidimensional and conventional SQL table indexes cannot easily be used. The use of custom indexing algorithms based upon HTM and other techniques greatly speeds up positional queries. The combination of custom spatial indexing algorithms plus the ability to process multiple spatial positions in a single query allows multi-position queries involving thousands of positions to be handled efficiently.

There are two main approaches for large multi-position queries: upload the source table as part of the query, or reference a previously uploaded or otherwise generated source table in the query.

To upload a source table directly as part of a query one would use the POST version of AdqlQuery, with a POST encoding of `Multipart/form-data`, which permits a mix of string parameters (as for GET) and file uploads to be packaged in the same request. Hence we can have request parameters as for the GET version of AdqlQuery, and at the same time upload a VOTable (or any other file, including binary files) containing any number of positions plus possibly other table fields to be passed through to the query output. Multi-position queries of this form are fully parallelizable and could be arbitrarily large (many thousands of positions).

While there are various ways that source data could be input for a multi-position query, we suggest that the standard format be VOTable, as this is already the format for the output of queries, as well as for storage of intermediate tables in a series of queries. In this case each source position is tagged with a source or

position ID. The query output may contain multiple records per input source; the records would be tagged with the source ID, allowing all data to be returned in a single table.

The HTTP `Multipart/form-data` mechanism allows submission of POST queries from any Web browser form, much as we already do for GET queries. In this case the VOTable of source positions could either be generated in advance, or on the fly by the Web form. User input would normally not be in the form of a VOTable, and would need to be converted for input to TAP.

This mechanism is also capable of uploading any auxiliary files which are referenced in a query. The REGION function in ADQL would reference the uploaded position table as a named table. VOSpace tables would be referenced with the same mechanism.

Execution of a multi-position query may be either synchronous or asynchronous, although POST must be used in both cases. Large multi-position queries may require asynchronous execution. Staging of the output is required only for the asynchronous version.

3.1.4 Data Staging

By data staging we mean staging data local to a TAP service for input to a query, or storage of any output data resulting from the query. Data staging is required for asynchronous queries (to define where the service should store the data) and is optional otherwise. Data would normally be staged to a VOSpace co-located with the service, or (for output) to a remote VOSpace, however other forms of data storage are also possible. In particular, output data staged local to a service could use some internal mechanism (such as a DBMS or file system) which is transparent to the client application. This means that asynchronous execution does not necessarily require VOSpace support.

Although the details are not yet clear, probably a similar mechanism can be used in queries to refer to all forms of data storage: staged user tables, normal archive data tables, or tables which are uploaded directly in a query request. For example, REGION might refer equivalently to data stored in any of these ways.

The details of data staging, including the parameters used to control staging in the AdqlQuery operation, are TBD. This is an advanced capability which does not have to be provided initially in TAP, although we would like to prototype this as soon as a basic TAP interface has been specified.

3.1.5 Asynchronous Queries

Use of the POST form of AdqlQuery would be required to initiate asynchronous queries. The details, including the parameters used to initiate asynchronous

execution, are TBD (as for the data staging capability). To a first approximation one would merely submit the query, including any staging instructions, and request that it execute asynchronously. The service would either return a job ID which could be used via the UWS mechanism to monitor job execution, or an error of some sort if there is a problem with the request. As with data staging, this does not have to be implemented in the initial version of TAP, but should be prototyped (along with data staging and SSO authentication) once the basic TAP interface has been specified.

3.2 SimpleQuery

The *SimpleQuery* operation provides a simple table data query mechanism and is also the primary mechanism provided in TAP for database and table metadata queries. The SimpleQuery operation has the following characteristics:

- The same interface is used to query both table data and metadata. In other words, data-oriented metadata is represented as tables. Service metadata is handled separately via a different mechanism (3.3).
- Only a single table (or view) can be queried at a time.
- Only a GET version is provided; input is via parameters, hence query parsing is not required.
- Only synchronous execution is permitted.
- Output may be returned in any supported output format.

Some of these limitations are not strictly necessary, .e.g., a POST form could also be permitted with support for multi-position queries and optional data staging; this would not complicate things much, particularly if the service also supports AdqlQuery. However, since our objective here is to define a simple query mechanism we will not consider such optional advanced capabilties further.

3.2.1 Motivation

The primary motivation for SimpleQuery is to provide a table access method which is both simple to implement, and easy to use by client applications for simple queries which do not require ADQL. Experience with real-world queries at our data centers shows that most (> 90%) of actual table data queries seen are simple queries selecting all or a few fields from a single table, with a minimal WHERE clause. In addition we would like to provide a simple mechanism to query database and table metadata which does not require ADQL.

Although some would argue that VO only requires full-function interfaces and that defining minimal implementations is not important, we feel that it is still important

to keep the needs of small data providers in mind. A small survey team for example, will want to publish data to the VO during the operational phase of the survey. Although the data may ultimately end up at a large data center (which can afford to implement complex, full-function services), during survey operations it is best if the survey team directly curates their data and makes new data accessible as soon as it is available from the survey pipeline.

Small data providers with limited resources and only a few tables to publish are more likely to implement a correct, robust TAP interface if it defines a simple interface; a full-up ADQL version is much more likely to either be incomplete or buggy, or not be implemented at all. A simple parameter-based, filter-type table query interface is much simpler to implement for non-SQL based systems; even for SQL-based systems it will be easier to parse and translate than ADQL-based input.

We may be able to ease this situation eventually by providing ready to use service frameworks, however we do not have these yet, and support will always be limited due to the number of target platforms out there.

3.2.2 SimpleQuery Parameters

The following parameters are defined for the SimpleQuery operation:

SELECT	Table fields to be returned (default all)
FROM	The table (or view) to be accessed
WHERE	A filter to be applied to the table (default none)
POS,SIZE	Find data only in this spatial region
FORMAT	Output data format
MAXREC	Maximum records out
RUNID	Pass-through for logging

(other params TBD)

The SELECT FROM WHERE parameters have an obvious motivation from SQL and will map directly upon an SQL back-end, but can be easily used with a non-SQL DBMS as well. The simplest possible query specifies only the FROM parameter, naming a single table or view to be queried. This may be all that is required for small data tables or for metadata tables. SELECT is a simple comma-delimited list of the table fields to be output; UTYPE/UCD field name resolution could be optionally performed upon these fields.

The POS, SIZE fields define a spatial region used to constrain the query. A query which specifies only FROM plus a spatial region is a simple cone search query. Both POS, SIZE and WHERE can be used in the same query. (TIME and BAND could also be provided, but we are concerned that these are not sufficiently well defined or useful for general tables hence have omitted them).

Various alternatives to POS, SIZE are possible, e.g., RA, DEC, SR, or use of a UTYPE or UCD to reference the spatial position. POS, SIZE is suggested because it is dimensionless and allows various coordinate systems to be specified, and because it is compatible with the other DAL interfaces allowing common code and semantics to be exploited. A UTYPE reference would also work, but only for the spatial position and not for the region size, which would still require a parameter. Use of parameters for all of the region-specific information seems simpler and more consistent.

The only parameter here of any complexity is the WHERE parameter. We want to keept this as simple as possible, as if any significant parsing is required we may as well use ADQL instead. A simple syntax would be to use a comma-delimited range list, where each field name is followed by a value which is either a fixed value (equality) or an open or close range list (range of valid values). For example,

```
FROM=foo&WHERE=objectType,galaxy,flux,5/&FORMAT=csv
```

would return all fields from table “foo” where the object type is “galaxy” and the value of the “flux” attribute is greater than or equal to 5, in CSV format. In this proposal only the AND relationship would be permitted in the WHERE clause.

Other schemes for WHERE are possible and should be explored, but something similar to this approach would work for many simple queries.

3.2.3 Field Names

As mentioned already in connection with AdqlQuery, we suggest that the choice of literal field names or UTYPEs be made individually for each field, using some predefined syntax (such as prepending a name space qualifier) to distinguish between the two. A possible field name syntax might be

FieldName = “<literal-name> | <name-space> ‘:’ <UTYPE>”

where “<literal>” is the literal field name as used in the table, and UTYPE is the UTYPE specifier for a field of the data model indicated by “<name-space>”. For the purposes of field name resolution, UCDs could be considered a special case of a data model, with its own name space “ucd”. All forms of field name would be resolved to literal field names prior to evaluating the query.

For example, the field `TargetName` from the SSA data model could be referred to by UTYPE as “`ssa:Target.Name`” or by UCD as “`ucd:meta.id;src`”. Any of these references would resolve to the literal table field name `TargetName` (whether this syntax might conflict with SQL syntax for field names is TBD but no doubt some solution can be found if this is the case).

3.2.4 Metadata Queries

We suggest that database, table, and query engine metadata be based upon (but not equivalent to) the *information schema* standard defined by SQL92. In this approach, standard views are defined to describe the database, its contents, and some aspects of the query engine, and the standard database query mechanism is used to query such metadata just as one would query actual data tables.

While the SQL information schema has some issues, we need something like this, it is a standard, and the concept of using the standard DBMS query mechanisms to query database metadata is an elegant approach. We cannot use the SQL information schema directly as, while it is implemented by most DBMS products (MySQL, PostgreSQL, SQL Server, etc.), it is not implemented by all, and each typically implements only a subset while adding its own custom metadata. This is essentially what we need to do for TAP as well, i.e., define a minimum subset of the information schema which a TAP service should provide, and extend this with additional custom metadata such as UTYPE, UCD, UNIT, etc. as required for our applications.

Aside from making use of an existing standard which is implemented in most SQL implementations, this approach has the advantage that the entire data path from the client application to the back-end DBMS can be the same for both data and metadata queries, allowing all related code, query facilities, output data formats, etc., to be used for both. In addition, the approach is easily extensible; if we want to describe some new aspect of the database, table, query engine, etc., we can add this by changing only the information schema without any changes to the service interface. The information schema is important not only to describe the database and the tables and views it contains, but to provide the information required for query optimization. This includes details such as the primary and foreign keys defined for each table (important for joins), any user defined functions, optional SQL/ADQL features, and so forth.

While ADQL could be useful for querying the information schema as an advanced optional capability, we are reluctant to require something as complex as ADQL for simple table metadata queries; the *SimpleQuery* operation is all that is needed in most cases.

A more complete view of the draft information schema recommended for TAP is provided in Appendix A. The most important elements of this are *SCHEMA.tables* and *SCHEMA.columns*, which list the database tables and describe their columns, respectively. Simple examples of queries against these tables are the following:

```
FROM=SCHEMA.tables
FROM=SCHEMA.columns&WHERE=tableName,foo
FROM=SCHEMA.columns&WHERE=tableName,foo&FORMAT=xml
```

The first merely lists the tables (or views) which the TAP service provides access to. The second lists the columns defined by table “foo”, in the default output format (VOTable). The third example does the same, except that the output format is native XML, which we could make compliant with whatever schema the Registry requires. This could be done for example, by implementing the registry view of a table as an actual View table in the database, allowing the registry to have its own custom view of the metadata for a table.

3.2.5 Simple Cone Search

In the proposal descrbed here, the TAP version of simple cone search reduces to a SimpleQuery using POS,SIZE:

```
REQUEST=SimpleQuery&FROM=foo&POS=180.0,12.5&SIZE=0.2
```

Additional constraints may be added, for example, if table “foo” has a field called “flux”, we could add `WHERE=flux,5/` to find only sources for which Flux is greater than or equal to 5.0. A FORMAT could be added to specify the desired output format. UCDs should be returned consistent with the UCD 1.1 specification or greater.

Note that the table to be queried is specified by name (this was missing in the legacy cone search interface). A SELECT clause could optionally be added to list the fields to be returned. POS defaults to ICRS, but other coordinate systems could be specified if supported by the service, e.g., to specify galactic coordinates, or to work with solar or planetary data.

If the TAP service supports AdqlQuery and REGION this could also be used to perform a cone search, with the option of more sophisticated expressions for the WHERE clause. In most cases this would still reduce to a simple GET query. By including a source table in the query a multi-position “cone search” could be performed.

3.2.6 Minimal TAP Service

The minimal TAP service supports *SimpleQuery*, including metadata queries over at least SCHEMA.tables and SCHEMA.columns. No data models need be supported other than that implied by POS, SIZE (i.e., no UTYPEs). The “ucd:” UTYPE could easily be supported even by a minimal service however. At least VOTable output format should be provided. An advanced service supports *AdqlQuery* as well. It is not clear whether or not *getCapabilities* and *getAvailability* should be required for a minimal service – probably they should since they should be simple to provide once defined.

3.3 GetCapabilties

The *getCapabilities* operation returns the Capabilities element of a registry *VOResource* descriptor, formatted as an XML document. A client application may call getCapabilities directly to query the capabilities of a TAP service instance. A special case of this is the registry itself, which calls the getCapabilities operation to download the service Capability element which is cached or updated in the registry description of the service.

An open question is how much information to include in the service Capability element. The main guideline is that this should be sufficient to describe the capabilities of the service in sufficient detail to permit service discovery. For example, does the service support the AdqlQuery operation, or any coordinate systems other than ICRS? Details on specific ADQL features should be given in the service Capabilities if they are needed for service discovery, but the main mechanism for describing ADQL or local SQL features, table columns, etc., is the information schema.

The details of the getCapabilities operation are TBD and are part of the emerging VOSI standard (GWS).

3.4 GetAvailability

The *getAvailability* operation is used to monitor service function, i.e., to determine if a service goes down. The details of the getAvailability operation are TBD and are part of the emerging VOSI standard (GWS).

4 Basic Service Elements

The basic form of a TAP service conforms to the standard service profile and HTTP semantics defined for the second generation DAL services and introduced with SSA V1.0 (see section 8, *Basic Service Elements*, of the SSA specification [1]). For example, REQUEST is used to specify the service operation to be invoked, and VERSION may be specified to enable runtime version checking or to select the version of an interface to be used, if the service supports multiple versions of a protocol. TAP protocol errors are returned as VOTable-formatted XML, using a QUERY_STATUS INFO element to return the query status and identify the error condition should error or overflow occur. Low level errors may result in an HTTP level error response. Range list syntax is used to specify list-valued parameters or ranges. Ultimately most of the mechanism used for asyncronous execution (based upon UWS), and data staging with VOSpace, will probably be common to all the DAL services as well, although this functionality has yet to be specified.

Appendix A: Database and Table Metadata

The following represents a first attempt (mainly by Pat Dowler) to identify a useful and widely available subset of the SQL information schema. Selected VO-specific metadata such as UTYPE, UCD, and UNIT has been added. This is very rough at this point and should not be considered as a serious proposal, but should illustrate the nature of what such a schema would provide.

```
// Available databases (schemata)
information_schema.schemata
(
      catalog_name            // physical database
      schema_name             // logical view of database
      schema_owner            // owner of schema or logical view
      sql_path
)

// Tables or views defined for a database
information_schema.tables
(
      table_catalog           // physical database
      table_schema            // logical view of database
      table_name              // owner of schema or logical view
      table_type              // base table, view, etc.
      table_description       // added for VO – purpose of table
)

// Describes all columns in all tables
information_schema.columns
(
      table_catalog
      table_schema
      table_name
      column_name
      ordinal_position
      column_default
      is_nullable
      data_type
      utype                   // added for VO
      ucd                     // added for VO
      unit                    // added for VO
      character_maximum_length
      character_octet_length
      numeric_precision
      numeric_precision_radix
      numeric_scale
      datetime_precision
      domain_catalog
      domain_schema
      domain_name
      udt_catalog?
      udt_schema? P
      udt_name? P
```

```
      dtd_identifier
)

// JOIN declaration? two rows with same constraint_* values, eg:
// catalog1.schema1.table1.col1 = catalog2.schema2.table2.col2

information_schema.key_column_usage
(
      constraint_catalog
      constraint_schema
      constraint_name
      table_catalog
      table_schema
      table_name
      column_name
      ordinal_position
)

// UDF declaration
information_schema.routines
(
      specific_catalog
      specific_schema
      specific_name
      routine_catalog
      routine_schema
      routine_name
      routine_type

      // describes return type:
      udt_catalog
      udt_schema
      udt_name
      data_type
      character_maximum_length
      character_octet_length
      numeric_precision
      numeric_precision_radix
      numeric_scale
      datetime_precision
      // end of return type description

      dtd_identifier
      routine_body
      routine_definition
      external_name
      external_language
      parameter_style
      is_deterministic
      sql_data_access
      sql_path
      created
      last_altered
)

// UDF argument declaration
information_schema.parameters (M: n/a)
```

```
(
      specific_catalog
      specific_schema
      specific_name
      ordinal_position
      parameter_mode
      parameter_name
      is_result
      as_locator
      data_type
      character_maximum_length
      character_octet_length
      numeric_precision
      numeric_precision_radix
      numeric_scale
      datetime_precision
      udt_catalog | user_defines_type_catalog
      udt_schema | user_defines_type_schema
      udt_name | user_defines_type_name
      dtd_identifier?
)

// Declaration of support for features/options?
information_schema.sql_features
(
      feature_id
      feature_name
      sub_feature_id
      sub_feature_name
      is_supported
      is_verified_by
      comments
)
```

For comparison, a graphical view of the SQL information schema as defined for the MySQL database is illustrated in Figure 1.

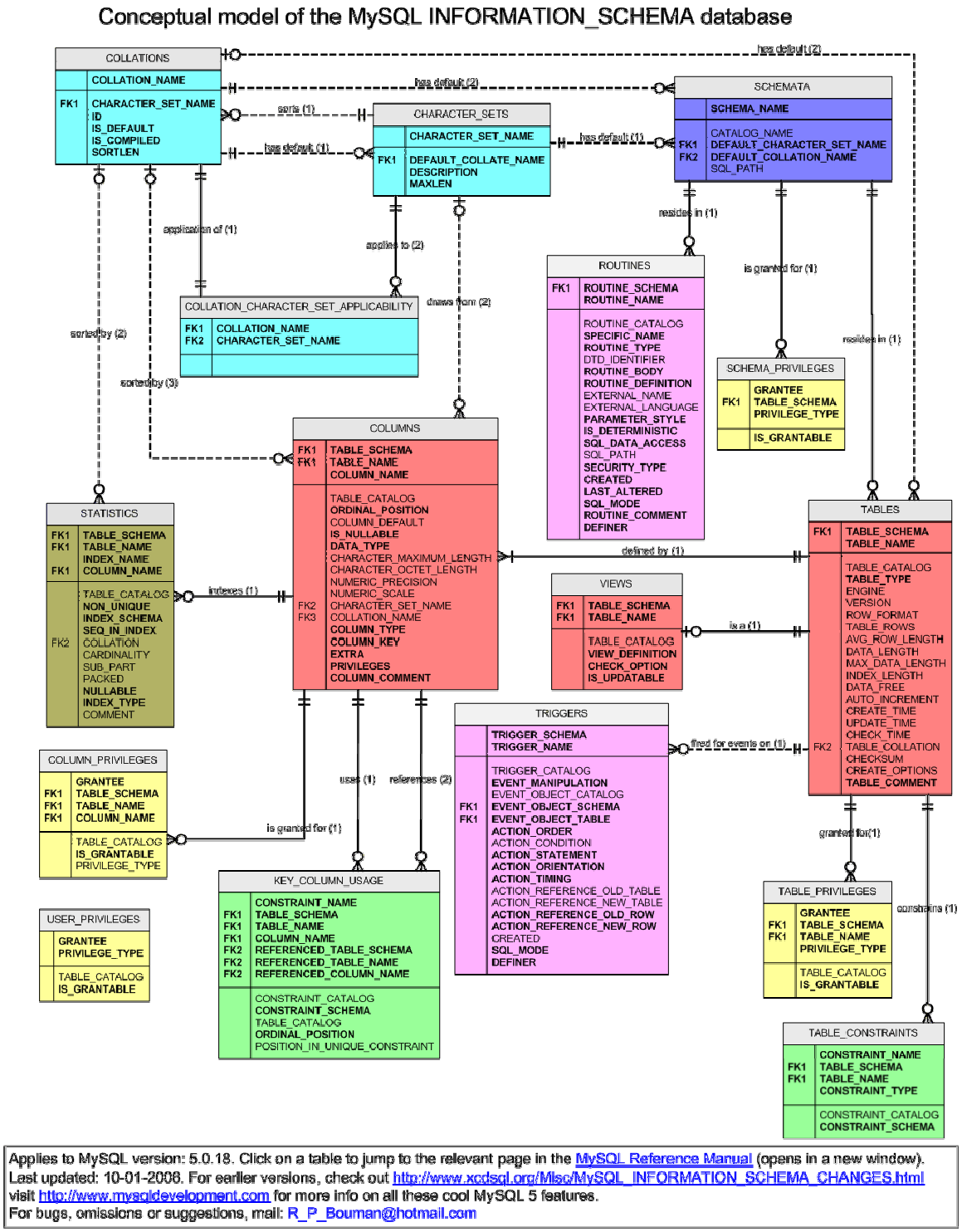

Figure 1. This illustrates the information schema as defined by the MySQL database. This represents only a subset of the full SQL92 information schema, and much of the metadata should is custom metadata specific to MySQL. These customizations are typical of SQL information schema in the real world so it is a realistic example!

References

[1] D.Tody, M.Dolensky, et.al, *Simple Spectral Access Protocol* , http://www.ivoa.net/Documents/latest/SSA.html
[2] R. Hanisch, *Resource Metadata for the Virtual Observatory* , http://www.ivoa.net/Documents/latest/RM.html
[3] R. Hanisch, M. Dolensky, M. Leoni, *Document Standards Management: Guidelines and Procedure* , http://www.ivoa.net/Documents/latest/DocStdProc.html