SQL API Guide

Contents:

1. Introduction

The purpose of this document is to define the SQL interface for MADlib algorithms.

2. Naming Conventions

Names should use lower case characters separated by underscores.

This is applicable to all database objects (tables, views, functions, function parameters, data types, operators, etc).

2.1. Schema

All database objects should be created in the default MADlib schema. Use `MADLIB_SCHEMA` as the schema prefix for your tables/views/functions/etc. in any scripts. This literal will be replaced during the installation with the target schema name (configured by the user in `Config.yml`). Code examples below use prefix `madlib` for illustration purposes only.

2.2. Functions and Aggregates

All non-user facing routines should be named with a "__" (double underscore) prefix to make the catalog easier to read.

Module specific routines should have a SHORT and COMMON prefix based on the module they belong to. For example:

- Multi-linear regression functions could start with **mregr_**:

```
madlib.mregr_coef(...)
```

- Naive-Bayes classification functions could start with **nb_**:

```
madlib.nb_create_view(...)
```

See the current function catalog for more examples.

General purpose routines should be named without a reference to any module and should be created inside a general purpose MADlib module (e.g. Array Operations). For example:

- Function returning the key of the row for which value is maximal:

```
madlib.argmax (integer key, float8 value)
```

3. Functions and Languages

To simplify this guide, we'd like to introduce three categories of user-defined functions:

- **UDAs** - user-defined aggregates, which perform a single scan of the data source and return an aggregated value for a group of rows. All UDA component functions should be written in PL/C (C/C++) for performance and portability reasons.
- **Row Level UDFs** - functions that operate on their arguments only and do not dispatch any SQL statements. These functions generate a result for each argument set, or for each tuple they are executed on. Recommended language is the same as for UDAs.
- **Driver UDFs** - functions that usually drive an execution of an algorithm, and may perform multiple SQL operations including data modification. In order to make this part of the code portable we suggest using PL/Python wrapper functions based on plain Python modules. The DB access inside the Python modules should be implemented using "classic" PyGreSQL interface (http://www.pygresql.org/pg.html).

4. Function Name Overloading

The suggestions below on name overloading apply to all the above-mentioned types of user-defined functions.

Data Types

Some platforms (like PostgreSQL) allow one to use the ANYELEMENT/ANYARRAY datatype, which can be used by MADlib routines (whenever it makes sense) in order to minimize code duplication.

If ANYELEMENT/ANYARRAY functionality is not available or not feasible, function name overloading can be used for different argument data types. For example, function F1 from module M1 can have the following versions:

- TEXT data type example:

```
madlib.m1_f1( arg1 TEXT)
```

- NUMERIC data type example:

```
madlib.m1_f1( arg1 BIGINT/FLOAT/etc.)
```

Argument Sets

Overloading mechanisms should also be used for different sets of parameters. For example, if `(reqarg1, ..., reqargN)` is a set of required parameters for function F1 from module M1, then the following definitions would be correct:

- A version for required arguments only:

```
madlib.m1_f1( reqarg1, ..., reqargN)
```

- A version for both required and optional arguments:

```
madlib.m1_f1( reqarg1, ..., reqargN, optarg1, ..., optargN)
```

5. Guide to Driver UDFs

- Should follow the naming conventions described in Section 2.
- Should follow the function overloading rules as described in Section 4. On Greenplum and PostgreSQL this can be achieved via PL/Python wrapper UDFs based on the same main Python code.

5.1. Input Definition

Parameters of the execution should be supplied directly in the function call (as opposed to passing a reference ID to a parameter-set stored in a table). For example:

```
SELECT madlib.m1_f1( par1 TEXT/INT/etc, par2 TEXT[]/INT[]/etc, ...)
```

Data should be passed to the function in the form of a text argument `schema.table` representing an existing table or a view, which:

- Can be located in any schema as long as the database user executing the function has read permissions.
- Should be defined in the method documentation. For example:

```
TABLE|VIEW (
  col_x INT,
  col_y FLOAT,
  col_z TEXT
)
```

- The input relation and its attributes needed by the function should be validated using primitive functions from the `helper.py` module. See Section 5.4 for more information.

5.2. Output Definition

Returning Simple Results or Models

We recommend using Standard Output to return a predefined single record structure for all cases when the results of a method or a model definition is in a human readable format. See examples below:

- Returning a model (Linear Regression):

```
SELECT mregr_coef(price, array[1, bedroom, bath, size]) from houses;
            mregr_coef
-----------------------------------
 {27923.4,-35524.8,2269.34,130.794}
```

- Returning results (FM Sketch):

```
SELECT madlib.fmsketch_dcount(pronargs) FROM pg_proc;
 fmsketch_dcount
-----------------
              10
```

Note: If it turns out that a large user population would prefer to have the model output saved in a table, you can add optional parameters as described in the following section.

Returning Complex Models

If a particular method returns a complex model that is represented in multiple rows it should be saved into a table with a pre-defined structure. The name of this table (including target schema) should be passed in the function call as an argument.

- Example (Decision Tree):

```
SELECT * FROM madlib.dtree_train( 'user_schema.user_table', 3, 10);
 output_table
------------------------
 user_schema.user_table
```

Returning Large Result Sets

The case for returning one or many data sets is similar to returning a complex model. The name(s) for all tables that will be created during the execution of the function must be supplied by the user in the function call. See the below section (Summary Output) for an example of multiple table output method.

Summary Output

Each *Driver UDF* should return a summary output in the form of a pre-defined record/row. Each attribute of the result should be clearly defined in the method documentation. If any tables/views are created or populated during the execution their full names should be returned in the summary. For example, the output of a k-means clustering algorithm could look like this:

```
 clusters | pct_input_used | cluster_table           | point_table
----------+----------------+-------------------------+--------------------
       10 |            100 | my_schema.my_centroids  | my_schema.my_points
```

The above output can be achieved in the following way:

1) Create data type for the return set `madlib.results`:

```
CREATE TYPE madlib.kmeans_result AS (
  clusters          INTEGER,
  pct_input_used    PERCENT,
  output_schema     TEXT,
  cluster_table     TEXT,
  point_table       TEXT
);
```

2) If using the recommended PL/Python language (see Section 3 for more info) you can use the following example to generate a single row of output inside a Python routine:

```
CREATE OR REPLACE FUNCTION madlib.kmeans_dummy()
  RETURNS SETOF madlib.kmeans_result
AS $$
  return ( [ 10, 100.0, 'my_schema', 'my_centroids', 'my_points' ]);
$$ LANGUAGE plpythonu;
```

5.3. Logging

- **ERROR**

If a function encounters a problem it should raise an error using the `plpy.error( message)` function (see section 6.1). This will ensure the proper end of the execution and error propagation to the calling environment.

- **INFO**

If specified by the user (*verbose* flag/parameter), long-running methods can send runtime status to the log. But be aware that this information may not be propagated to clients in many cases, and it will enlarge the stored log file. Informational logging should be turned off by default, and activated only with an explicit user command. Use `plpy.info( message)` function (see Section 6.1) to properly generate information logs. Example log output:

```
SQL> select madlib.kmeans_run( 'my_schema.data_set_1', 10, 1, 'run1', 'my_schema', 1);
INFO: Parameters:
INFO:  * k = 10 (number of centroids)
INFO:  * input_table = my_schema.data_set_1
INFO:  * goodness = 1 (GOF test on)
INFO:  * run_id = run1
INFO:  * output_schema = my_schema
INFO:  * verbose = 1 (on)
INFO: Seeding 10 centroids...
INFO: Using sample data set for analysis... (9200 out of 10000 points)
INFO: ...Iteration 1
INFO: ...Iteration 2
INFO: Exit reason: fraction of reassigned nodes is smaller than the limit: 0.001
INFO: Expanding cluster assignment to all points...
INFO: Calculating goodness of fit...
...
```

5.4. Parameter Validation

Parameter validation should be performed in each function to avoid any preventable errors.

For **simple arguments (scalar, array)** sanity checks should be done by the author. Some common parameters with known value domains should be validated using SQL domains, for general use. For example:

- Percent

```
CREATE DOMAIN percent AS FLOAT
CHECK(
   VALUE >= 0.0 AND VALUE <= 100.0
);
```

- Probability

```
CREATE DOMAIN probability AS FLOAT
CHECK(
   VALUE >= 0.0 AND VALUE <= 1.0
);
```

For **table/view and column arguments** please see Section 6.2 (describing usage of the helper.py module).

5.5. Multi-User and Multi-Session Execution

In order to avoid unpleasant situations of over-writing or deleting results, MADlib functions should be ready for execution in multi-session or multi-user environment. Hence the following requirements should be met:

- Input relations (tables or views) should be used for read only purposes.
- Any user output table given as an argument must not overwrite an existing database relation. In such case an error should be returned.
- Any execution specific tables should be locked in EXCLUSIVE MODE after creation. This functionality will be implemented inside the Python abstraction layer. There is no need to release LOCKS as they will persist anyway until the end of the main UDF.

6. Support Modules

A set of Python modules to make MADlib development easier.

6.1. DB Connectivity: plpy.py

This module serves as the database access layer. Even though currently not used this module will provide easy portability between various MADlib platforms and interfaces. To clarify: PostrgreSQL PL/Python language currently uses an internal plpy.py module to implement seamless DB access (using "classic" PyGreSQL interface - see http://www.pygresql.org/pg.html). By adding a MADlib version of plpy.py we'll be able to more easily port code written for MADlib.

Currently implemented functionality:

```
def connect ( dbname, host, port, user, passwd)
def close()
def execute( sql)
def info( msg)
def error( msg)
```

6.2. Python Abstraction Layer

This module consists of a set of functions to support common data validation and database object management tasks.

Example functions:

- table/view existence check

```
def __check_rel_exist( relation):
    if relation ~ schema.table:
       check if exists
    if relation ~ table:
       find the first schema using SEARCH_PATH order
    returns:
       - (schema,table)
       - None (if not found)
```

- relation column existence check (assuming relation exists)

```
def __check_rel_column( relation, column):
    returns:
       - (schema,table,column)
       - None (if not found)
```

- relation column data type check (assuming table & column exist)

```
def __check_rel_column( relation, column, data_type):
    returns:
       - (schema,table,column,type)
       - None (if not found)
```