Quick Start Guide for Developers

blocked URL

This guide explains all of the elements needed to successfully develop and plug in a new MADlib® module.

1. Prerequisites
2. Docker Image
3. Adding a New Module
4. Adding an Iterative UDF

The files for the examples in this guide can be found in the hello world folder of the source code repository.

Prerequisites

Install MADlib by following the steps in the Installation Guide for MADlib or use the Docker image instructions below.

MADlib source code is organized such that the core logic of a machine learning or statistical module is located in a common location, and the database-port specific code is located in a `ports` folder. Since all currently supported databases are based on Postgres, the `postgres` port contains all the port-specific files, with `greenplum` and `hawq` inheriting from it. Before proceeding with this guide, it is recommended that you familiarize yourself with the MADlib module anatomy.

Docker Image

We provide a Docker image with necessary dependencies required to compile and test MADlib on PostgreSQL 9.6. You can view the dependency docker file at `./tool/docker/base/Dockerfile_postgres_9_6`. The image is hosted on docker hub at `madlib/postgres_9.6:latest`. Later we will provide a similar Docker image for Greenplum Database.

Some useful commands to use the Docker file:

```
## 1) Pull down the `madlib/postgres_9.6:latest` image from docker hub:
docker pull madlib/postgres_9.6:latest
## 2) Launch a container corresponding to the MADlib image, mounting the source code folder to the container:
docker run -d -it --name madlib -v (path to incubator-madlib directory):/incubator-madlib/ madlib/postgres_9.6
where incubator-madlib is the directory where the MADlib source code resides.
############################################## * WARNING * ##################################################
# Please be aware that when mounting a volume as shown above, any changes you make in the "incubator-madlib"
# folder inside the Docker container will be reflected on your local disk (and vice versa). This means that
# deleting data in the mounted volume from a Docker container will delete the data from your local disk also.
#############################################################################################################
## 3) When the container is up, connect to it and build MADlib:
docker exec -it madlib bash
mkdir /incubator-madlib/build-docker
cd /incubator-madlib/build-docker
cmake ..
make
make doc
make install
## 4) Install MADlib:
src/bin/madpack -p postgres -c postgres/postgres@localhost:5432/postgres install
## 5) Several other madpack commands can now be run:
# Run install check, on all modules:
src/bin/madpack -p postgres -c postgres/postgres@localhost:5432/postgres install-check
# Run install check, on a specific module, say svm:
src/bin/madpack -p postgres -c postgres/postgres@localhost:5432/postgres install-check -t svm
# Run dev check, on all modules (more comprehensive than install check):
src/bin/madpack -p postgres -c postgres/postgres@localhost:5432/postgres dev-check
# Run dev check, on a specific module, say svm:
src/bin/madpack -p postgres -c postgres/postgres@localhost:5432/postgres dev-check -t svm
# Reinstall MADlib:
src/bin/madpack -p postgres -c postgres/postgres@localhost:5432/postgres reinstall
## 6) Kill and remove containers (after exiting the container):
docker kill madlib
docker rm madlib
```

Adding A New Module

Let's add a new module called `hello_world`. Inside this module we implement a User-Defined SQL Aggregate (UDA), called `avg_var` which computes the mean and variance for a given numerical column of a table. We'll implement a distributed version of Welford's online algorithm for computing the mean and variance.

Unlike an ordinary UDA in PostgreSQL, `avg_var` will also work on a distributed database and take advantage of the underlying distributed network for parallel computations. The usage of `avg_var` is very simple; users simply run the following command in `psql`:

```
sql select avg_var(bath) from houses
```

which will print three numbers on the screen: mean, variance and number of rows in column `bath` of table `houses`.

Below are the main steps we will go through:

1. Register the module.
2. Define the SQL functions.
3. Implement the functions in C++.
4. Register the C++ header files.

The files for this exercise can be found in the hello world folder of the source code repository.

1. Register the module

Add the following line to the file called `Modules.yml` under `./src/config/`

```
- name: hello_world
```

and create two folders: `./src/ports/postgres/modules/hello_world` and `./src/modules/hello_world`. The names of the folders need to match the name of the module specified in `Modules.yml`.

2. Define the SQL functions

Create file `avg_var.sql_in` under folder `./src/ports/postgres/modules/hello_world`. Inside this file we define the aggregate function and other helper functions for computing mean and variance. The actual implementations of those functions will be in separate C++ files which we will describe in the next section.

At the beginning of file `avg_var.sql_in` the command `m4_include`('SQLCommon.m4') is necessary to run the m4 macro processor. M4 is used to add platform-specific commands in the SQL definitions and is run while deploying MADlib to the database.

We define the aggregate function `avg_var` using built-in PostgreSQL command `CREATE AGGREGATE`.

```
DROP AGGREGATE IF EXISTS MADLIB_SCHEMA.avg_var(DOUBLE PRECISION);

CREATE AGGREGATE MADLIB_SCHEMA.avg_var(DOUBLE PRECISION) (
    SFUNC=MADLIB_SCHEMA.avg_var_transition,
    STYPE=double precision[],
    FINALFUNC=MADLIB_SCHEMA.avg_var_final,
    m4_ifdef(`__POSTGRESQL__', `', `prefunc=MADLIB_SCHEMA.avg_var_merge_states,')
    INITCOND='{0, 0, 0}'
);
```

We also define parameters passed to `CREATE AGGREGATE`:

- `SFUNC`
  - The name of the state transition function to be called for each input row. The state transition function, `avg_var_transition` in this example, is defined in the same file `avg_var.sql_in` and implemented later in C++.
- `FINALFUNC`
  - The name of the final function called to compute the aggregate's result after all input rows have been traversed. The final function, `avg_var_final` in this example, is defined in the same file `avg_var.sql_in` and implemented later in C++.
- `PREFUNC`

    - The name of the merge function called to combine the aggregate's state values after each segment, or partition, of data has been traversed. The merge function is needed for distributed datasets on Greenplum and HAWQ. For PostgreSQL, the data is not distributed, and the merge function is not necessary. For completeness we implement a merge function called `avg_var_merge_states` in this guide.
- `INITCOND`
    - The initial condition for the state value. In this example it is an all-zero double array corresponding to the values of mean, variance, and the number of rows, respectively.

The transition, merge, and final functions are defined in the same file `avg_var.sql_in` as the aggregate function. More details about those functions can be found in the PostgreSQL documentation.

3. Implement the functions in C++

Create the header and the source files, `avg_var.hpp` and `avg_var.cpp`, under the folder `./src/modules/hello_world`. In the header file we declare the transition, merge and final functions using the macro `DECLARE_UDF(MODULE, NAME)`. For example, the transition function `avg_var_transition` is declared as `DECLARE_UDF(hello_world, avg_var_transition)`. The macro `DECLARE_UDF` is defined in the file `dbconnector.hpp` under `./src/ports/postgres/dbconnector`.

Under the hood, each of the three UDFs is declared as a subclass of `dbconnector::postgres::UDF`. The behavior of those UDFs is solely determined by its member function

```
AnyType run(AnyType &args);
```

In other words, we only need to implement the following methods in the `avg_var.cpp` file:

```
AnyType avg_var_transition::run(AnyType& args);
AnyType avg_var_merge_states::run(AnyType& args);
AnyType avg_var_final::run(AnyType& args);
```

Here the `AnyType` class works for both passing data from the DBMS to the C++ function, as well as returning values back from C++. Refer to `TypeTraits_impl.hpp` for more details.

`Transition function`

```
AnyType
avg_var_transition::run(AnyType& args) {
    // get current state value
    AvgVarTransitionState<MutableArrayHandle<double> > state = args[0];
    // update state with current row value
    double x = args[1].getAs<double>();
    state += x;
    state.numRows ++;
    return state;
}
```

- There are two arguments for `avg_var_transition`, as specified in `avg_var.sql_in`. The first one is an array of SQL double type, corresponding to the current mean, variance, and number of rows traversed, and the second one is a double representing the current tuple value.
- We will describe `class AvgVarTransitionState` later. Basically it takes `args[0]`, a SQL double array, passes the data to the appropriate C++ types and stores them in the `state` instance.
- We compute the average and variance in an on-line manner by overloading the operator `+=` in the class `AvgVarTransitionState.`

`Merge function`

```
AnyType
avg_var_merge_states::run(AnyType& args) {
    AvgVarTransitionState<MutableArrayHandle<double> > stateLeft = args[0];
    AvgVarTransitionState<ArrayHandle<double> > stateRight = args[1];

    // Merge states together and return
    stateLeft += stateRight;
    return stateLeft;
}
```

- Again: the arguments contained in `AnyType& args` are defined in `avg_var.sql_in`.

- The details are hidden in the method of class `AvgVarTransitionState` which overloads the operator +=

`Final function`

```
AnyType
avg_var_final::run(AnyType& args) {
    AvgVarTransitionState<MutableArrayHandle<double> > state = args[0];

    // If we haven't seen any data, just return Null. This is the standard
    // behavior of aggregate function on empty data sets (compare, e.g.,
    // how PostgreSQL handles sum or avg on empty inputs)
    if (state.numRows == 0)
        return Null();

    return state;
}
```

- Class `AvgVarTransitionState` overloads the `AnyType()` operator such that we can directly return state, an instance of `AvgVarTransition State`, while the function is expected to return a `AnyType`.

Bridging class

Below are the methods that overload the operator += for the bridging class `AvgVarTransitionState`:

```
   /**
     * @brief Update state with a new data point
     */
    AvgVarTransitionState &operator+=(const double x){
        double diff = (x - avg);
        double normalizer = static_cast<double>(numRows + 1);
        // online update mean
        this->avg += diff / normalizer;
        // online update variance
        double new_diff = (x - avg);
        double a = static_cast<double>(numRows) / normalizer;
        this->var = (var * a) + (diff * new_diff) / normalizer;
    }

/**
 * @brief Merge with another State object
 *
 * We update mean and variance in a online fashion
 * to avoid intermediate large sum.
 */
template <class OtherHandle>
AvgVarTransitionState &operator+=(
    const AvgVarTransitionState<OtherHandle> &inOtherState) {

    if (mStorage.size() != inOtherState.mStorage.size())
        throw std::logic_error("Internal error: Incompatible transition "
                               "states");
    double avg_ = inOtherState.avg;
    double var_ = inOtherState.var;
    uint64_t numRows_ = static_cast<uint64_t>(inOtherState.numRows);
    double totalNumRows = static_cast<double>(numRows + numRows_);
    double p = static_cast<double>(numRows) / totalNumRows;
    double p_ = static_cast<double>(numRows_) / totalNumRows;
    double totalAvg = avg * p + avg_ * p_;
    double a = avg - totalAvg;
    double a_ = avg_ - totalAvg;

    numRows += numRows_;
    var = p * var + p_ * var_ + p * a * a + p_ * a_ * a_;
    avg = totalAvg;
    return *this;
}
```

Given the mean, variance and the size of two data sets, Welford's method computes the mean and variance of the two data sets combined.

4. Register the C++ header files

The SQL functions defined in `avg_var.sql_in` need to be able to locate the actual implementations from the C++ files. This is done by simply adding the following line to the file `declarations.hpp` under `./src/modules/`

```
#include "hello_world/avg_var.hpp"
```

5. Running the new module

Now let's run an example using the new module. First, rebuild and reinstall MADLib according to the instructions from Installation Guide. We use the `patients` dataset from the MADlib Quick Start Guide for Users for testing purposes. From the `psql` terminal, the result below shows that half of the 20 patients have had second heart attacks within 1 year (yes = 1):

```
SELECT madlib.avg_var(second_attack) FROM patients;

    -- ************ --
    --    Result    --
    -- ************ --
    +-------------------+
    | avg_var           |
    |-------------------|
    | [0.5, 0.25, 20.0] |
    +-------------------+
-- (average, variance, count) --
```

Adding An Iterative UDF

In this session we demonstrate a slightly more complicated example which requires invoking a UDA iteratively. Such cases can often be found in many machine learning modules where the underlying optimization algorithm takes iterative steps towards the optimum of the objective function. In this example we implement a simple logistic regression solver as an iterative UDF. In particular, the user will be able to type the following command in `psql` to train a logistic regression classifier:

```
SELECT madlib.logregr_simple_train('patients','logreg_mdl', 'second_attack', 'ARRAY[1, treatment,
trait_anxiety]');
```

and to see the results:

```
SELECT * FROM logreg_mdl;
```

Here the data is stored in a SQL TABLE called `patients`. The target for logistic regression is the column `second_attack` and the features are columns `treatment` and `trait_anxiety`. The `1` entry in the ARRAY denotes an additional bias term in the model.

We add the solver to the `hello_world` module created above. Here are the main steps to follow:

- In `./src/ports/postgres/modules/hello_world`
  - create file `__init__.py_in`
  - create file `simple_logistic.py_in`
  - create file `simple_logistic.sql_in`
- In `./src/modules/hello_world`
  - create file `simple_logistic.cpp`
  - create file `simple_logistic.hpp`
- In `./src/modules`
  - modify file `declarations.hpp`: append a new line `#include "hello_world/simple_logistic.hpp"` to the end.

Compared to the steps presented in the last session, here we do not need to modify the `Modules.yml` file because we are not creating new module. Another difference is that we create an additional `.py_in` python file along with the `.sql_in` file. That is where most of the iterative logic will be implemented.

The files for this exercise can be found in the hello world folder of the source code repository. Please note that `__init__.py_in` is not included in this folder as an empty file will be sufficient for the purposes of this exercise.

1. Overview

The overall logic is split into three parts. All the UDF and UDA are defined in `simple_logistic.sql_in`. The `transition`, `merge` and `final` functions are implemented in C++. Those functions together constitute the UDA called `__logregr_simple_step` which takes one step from the current state to decrease the logistic regression objective. And finally in `simple_logistic.py_in` the `plpy` package is used to implement in python a UDF called `logregr_simple_train` which invokes `__logregr_simple_step` iteratively until convergence.

Note that the SQL function `logregr_simple_train` is defined in `simple_logistic.sql_in` as follows:

```
CREATE OR REPLACE FUNCTION MADLIB_SCHEMA.logregr_simple_train (
    source_table        VARCHAR,
    out_table           VARCHAR,
    dependent_varname   VARCHAR,
    independent_varname VARCHAR,
    max_iter            INTEGER,
    tolerance           DOUBLE PRECISION,
    verbose             BOOLEAN
) RETURNS VOID AS $$
PythonFunction(hello_world, simple_logistic, logregr_simple_train)
$$ LANGUAGE plpythonu
m4_ifdef(`__HAS_FUNCTION_PROPERTIES__', `MODIFIES SQL DATA', `');
```

where `PythonFunction(hello_world, simple_logistic, logregr_simple_train)` denotes that the actual implementation is provided by a python function `logregr_simple_train` inside the file `simple_logistic` in module `hello_world`, as shown below:

```
def logregr_simple_train(
        schema_madlib, source_table, out_table, dependent_varname,
        independent_varname, max_iter=None,
        tolerance=None, verbose=None, **kwargs):
    """
    Train logistic model

    @param schema_madlib Name of the MADlib schema, properly escaped/quoted
    @param source_table Name of relation containing the training data
    @param out_table Name of relation where model will be outputted
    @param dependent_varname Name of dependent column in training data (of type BOOLEAN)
    @param independent_varname Name of independent column in training data (of type
                    DOUBLE PRECISION[])
    @param max_iter The maximum number of iterations that are allowed.
    @param tolerance The precision that the results should have
    @param kwargs We allow the caller to specify additional arguments (all of
           which will be ignored though). The purpose of this is to allow the
           caller to unpack a dictionary whose element set is a superset of
           the required arguments by this function.

    @return A composite value which is __logregr_simple_result defined in simple_logistic.sql_in
    """

    return __logregr_train_compute(
        schema_madlib, source_table, out_table, dependent_varname,
        independent_varname, max_iter, tolerance,
        verbose, **kwargs)
```

2. Iterative procedures in `plpy`

The iterative logic is implemented using the PL/Python procedural language. In the beginning of `simple_logistic.py_in` we import a Python module called `plpy` which provides several functions to execute database commands. Implementing the iterative logic using `plpy` is simple, as demonstrated below:

```
update_plan = plpy.prepare(
    """
    SELECT
        {schema_madlib}.__logregr_simple_step(
            ({dep_col})::boolean,
            ({ind_col})::double precision[],
            ($1))
```

```
    FROM {tbl_source}
    """.format(
        tbl_output=tbl_output,
        schema_madlib=schema_madlib,
        dep_col=dep_col,
        ind_col=ind_col,
        tbl_source=tbl_source), ["double precision[]"])

state = None
for it in range(0, max_iter):
    res_tuple = plpy.execute(update_plan, [state])
    state = res_tuple[0].values()[0]
```

- The `__logregr_simple_step` is a UDA defined in `simple_logistic.sql_in` and implemented using `transition`, `merge` and `final` functions provided in C++ files in `./src/modules/hello_world`.
- `__logregr_simple_step` takes three arguments, the target, the features and the previous state.
- The state is initialized as `None` which is interpreted as `null` value in SQL by `plpy`.
- A more sophisticated iterative scheme for logistic regression would also include optimality verification and convergence guarantee procedures, which are neglected on purpose here for simplicity.
- For a production-level implementation of logistic regression, refer to the module `regress`.

3. Running the new iterative module

The example below demonstrates the usage of `madlib.logregr_simple_train` on the `patients` table we used earlier. The trained classification model is stored in the table called `logreg_mdl` and can be viewed using standard SQL query.

```
SELECT madlib.logregr_simple_train(
    'patients',                                 -- source table
    'logreg_mdl',                               -- output table
    'second_attack',                            -- labels
    'ARRAY[1, treatment, trait_anxiety]');      -- features
SELECT * FROM logreg_mdl;

-- ************ --
--    Result    --
-- ************ --
+-------------------------------------------------+------------------+
| coef                                            |   log_likelihood |
|-------------------------------------------------+------------------|
| [-6.27176619714, -0.84168872422, 0.116267554551] |         -9.42379 |
+-------------------------------------------------+------------------+
```