SCHOOL OF ENGINEERING AND SCIENCES
JACOBS UNIVERSITY BREMEN

Integrating Rasdaman with Python

Siddharth Shukla

January 11, 2016

Prof. Dr. Peter Baumann
Independent Study Course : Big Data Lab Project

ABSTRACT

With data science becoming one of the most popular fields of the 21^{st} century, the need has grown to provide convenient access to data to the data scientists. Since, a majority of data scientists use Python as their primary language, we figured that an intuitive and easy to use Python interface to Rasdaman[1] is in order. RasPy allows the end programmer to execute RasQL[2] queries to a remote Rasdaman database instance and return the resulting array. This document underlines different approaches that can be taken to complete the goal of developing the said library.

CONTENTS

1 MOTIVATION

Python is the most popular language among computer scientists and data scientists alike because of it?s powerful yet readable syntax. Besides syntax, it?s a “batteries included" language and has a wide array of libraries such as SciPy, NumPy, Pandas, SymPy, et cetera which make it the primary choice of workbench for scientists. However, Python is limited to main memory data sizes and throw a MemoryError whenever it’s asked to handle huge files. The idea is to create a python interface which allows the end user to connect to and query an array database, here: Rasdaman, hosted on another server and hence delegate the data handling tasks to the server that does not suffer from similar issues. This would allow the user to get the arrays from the database in a python usable format (NumPy arrays). Support for Pandas dataframe objects and SciPy sparse matrices is also envisioned sometime in the future. The proof of concept is similar to PHP’s $mysql\_connect()$ command except it’s being executed on the client side. From hereon, the proof of concept will be referred to as RasPy. RasPy is the framework which glues together rasdaman with the Python language.

2 STATE OF THE ART

At the moment of writing, there is no python interface for Rasdaman and hence the need to implement on to make Rasdaman more Python-friendly. There have been similar Python interfaces for popular database technologies such as MySQLdb’s Python API, SciDB with SciDB-py which allows remote query execution, etc. MonetDB implements a python API called python-monetdb. SciDB-Py[3], includes a NumPy like syntax where the data processing takes place on the database side by lazy evaluation of arrays and creating the necessary queries (Abstract Functional Language and Abstract Query Language) on the fly. The requests to connect to the SciDB server are sent via HTTP to the SciDB shim client which is a package dependency along with Requests (A HTTP library for Python). An example of remote query execution from SciDB would be:

```
from scidbpy import connect
sdb = connect()
arr = sdb.new_array()
    sdb.query("store("
              "  redimension("
              "    filter({J}, {J.a0}={J.a1}),"
              "    <{J.d1}:int64>[{J.d0}=0:{N},{chunk},0]),"
              "  {arr})",
              J=J, arr=arr,
              N=J.shape[0] - 1,
              chunk=J.datashape.chunk_size[0])
arr = arr.toArray()
```

Python-MonetDB provides a similar library for executing queries on a remote database instance in the following manner:

```
> import monetdb.sql
> connection = monetdb.sql.connect(username="monetdb", password="monetdb", hostname="localhost", data
> cursor = connection.cursor()
> cursor.arraysize = 100
> cursor.execute('SELECT * FROM tables')
> cursor.fetchall()
[[1067, 'types', 1061, None, 0, True, 0, 0],
 [1076, 'functions', 1061, None, 0, True, 0, 0],
 [1085, 'args', 1061, None, 0, True, 0, 0],
 [1093, 'sequences', 1061, None, 0, True, 0, 0],
 [1103, 'dependencies', 1061, None, 0, True, 0, 0],
```

```
[1107, 'connections', 1061, None, 0, True, 0, 0],
[1116, '_tables', 1061, None, 0, True, 0, 0],
...
[4141, 'user_role', 1061, None, 0, True, 0, 0],
[4144, 'auths', 1061, None, 0, True, 0, 0],
[4148, 'privileges', 1061, None, 0, True, 0, 0]]
```

3 IMPLEMENTATION PLAN

3.1 ENVISIONED PROOF OF CONCEPT

```
import raspy
# Connect to default port 7001 if not specified
ras = raspy.connect("0.0.0.0", "myuser", "mypass", port=7000)

# Open the database given a db name
db = ras.database("dbname")

# List of all the collections available
collection_list = db.collections()

# Begin transaction
txn = db.transaction()

# Get array data from rasdaman server
query = txn.query("select m[0:10 ,0:10] from mr as m")
data = query.execute()

# End transaction
del txn

# Close the database connection
del db

# Convert to Numpy Array
data = data.toArray()

# Numpy Operations
data += 1
```

3.2 FLEXIBLE OPTIONS AND SANE DEFAULTS

The goal is to provide the end user (i.e. the Python programmer) a convenient method that allows them to execute RasQL queries to the server. While we should provide configuration possibilities for all possible options, we should also provide sane defaults for the convenience of the user and ease of usability.

The way this can be done is to have configuration options passed on as keyword arguments (kwargs), in absence of which, the library would default to given defaults:

- connect:host - 0.0.0.0

- connect:port - 7001
- connect:user - rasdaman
- connect:password - rasdaman
- open:name - rasdaman
- toArray:type - numpy

Hence, in absence of keyword arguments:

```
ras = raspy.connect()
db = ras.open()
...
data = data.toArray()
```

The library would try to connect to the rasdaman instance running on localhost at port 7001, with the credentials "rasdaman" and "rasdaman", open the database titled "rasdaman", and serialize the data into numpy arrays. However, the keyword arguments can be specified to customize the experience such that this statement

```
ras = raspy.connect(host="8.8.8.8", port="7000", user="myUser", pass="myPass")
db = ras.open(name="db")
...
data = data.toArray(type="scipy")
```

would connect to the rasdaman instance running on 8.8.8.8 at port 7000 using the credentials "myUser" and "myPass", open the database titled "db", and serialize the data into the scipy sparse arrays.

3.3 Choice of Python version

Since Python 3.x broke backwards compatibility, a lot of developers and scientists decided to stick with Python 2.x (legacy branch) for compatibility reasons since there are a few valid Python 2.x constructions that won't parse under Python 3.x

For this project, we decided to take a bigger overview and look at under the hood changes which could aid our purposes and help us make an informed decision instead of choosing 2.x because of the status quo.

As it turns out, Python 3.x is 7+ years old, much more consistent, and is nearly a superset of Python 2.7. Besides a few exceptions (Twisted, Gevent, etc.), by now, almost all Python libraries should work on 3.x or on both major versions.

Some of the major benefits that we can reap from using the 3.x branch would be:

- Saner bytes/unicode separation
- Unicode support (all text strings unicode by default)
- The heavily used range() fucntion returns a memory efficient iterable instead of a typical python list
- Decimal division by default (The default division in python 2.x was integer division and was the cause of a lot of confusion. It was eventually deemed non-pythonic and hence removed in 3.x)
- Much better optimizations for decimals under the hood making operations 30x - 80x faster that 2.x
- All the classes in 3.x are new-style python classes and we won't have to deal with occasional class incompatibility errors

- Better tuple unpacking (extended with more generalizaitons)
- Exception chaining
- Support for function annotation if needed

In essence, development efforts would focus on developing with python 3.x in mind. Support for legacy python version is also envisioned in future, but currently, that is not the focus.

3.4 POSSIBLE APPROACHES

There can be multiple approaches to solve this problem and we will discuss some of them here.

One approach would basically rely on using the existing client side functionality of RasODMG or RasJ and build Python wrappers around the said libraries to achieve the task. The benefit of this approach is that since all the code would be client sided, it can be used by anybody without updating their Rasdaman installation. These wrappers can either work using static loading or using dynamic linking. Usually, dynamic linking is considered easier to implement. However, writing wrappers for large libraries is a tedious task and one ends up writing thousands of lines of code to wrap a simple library. Besides, the length, debugging the wrappers in itself is quite a tedious task.

We can automate some of these tasks using interface generators such as SWIG, PyRex, or SIP which require us to write an interface file and with the help of that generate the wrapper files. However, following this approach using SWIG resulted in a never ending debugging spree that spawned more than a week with no conclusive results while trying to compile the wrapper with the rasnetprotocol into object files for usage within Python.

Alternatively, we can hook into the rasnet protocol which is based on grpc for remote procedure calls and use it for our remote query execution. We'll have a client side library which would execute functions on our server side libraries using grpc. In concept it's a brilliant idea since it allows us to have a language agnostic API i.e. have multiple client side libraries in different languages (Python, Ruby, Go, PHP, etc.) which communicate with our server side library using protocol buffers.

However, grpc is still in beta and still feels like alpha. It's a nightmare from a devops perspective since everything needs to be built from source. Besides, the different language plugins usually lag behind the general grpc branch which causes inconsistencies. Also, the examples present don't particularly work because of incompatible protocol buffer headers and building the Python plugin is quite a struggle. However, this is a much cleaner and more elegant approach since this is extensible to other languages such as Ruby if we decide to branch out into other languages besides Python. So, for the purpose of the project we will go with the latter approach.

The proto files define the protocol buffers along with the rpc methods. Using the protobuf 3.0 compiler, we need to generate the files that provide us with a python stub using which we can develop an interface for our client library which will call the server methods.

To generate the stubs, use the following example command adjusted accordingly for all the protofiles after switching to the directory with protofiles:

```
protoc -I . --python_out=. --grpc_out=. --plugin=protoc-gen-grpc=‘which grpc_python_plugin‘
 ./error_message.proto
```

After generating the stub files (ending in _pb2.py), we need to create client side functions that interface with these stubs that interact with the server using protocol buffer over grpc. We need to create the request

builders and methods that call the rpc function through stubs. An example of such functions is demonstrated here in the form of Connect function for reference. For the request builder:

```
def make_connect_req(username, password):
    con_req = rasmgr.ConnectReq(userName=username, passwordHash=password)
    if not con_req:
        raise Exception("Can't create Connect request")
    return con_req
```

and for the rpc calling:

```
def rasmgr_connect(stub, username, password):
    connection = stub.Connect(make_connect_req(username,password), _TIMEOUT_SECONDS)
    if not connection:
        raise Exception("Remote function 'Connect' did not return anything")
    return connection
```

3.5 Python Classes

The classes required would be the following:

- Connection: Class to represent the connection from the python client to the rasdaman server
  - __init__(self, hostname, port, username, password): Constructor for the Connection class
  - database(self, name): Returns a database object initialized with this connection
- Database: Class to represent a database stored inside a rasdaman server
  - __init__(self, connection, name): Constructor for the Database class
  - transaction(self, name): Returns a new database object initialized with this database
  - collections(self): Returns all the collections for this database
- Collection: Class to represent a given collection inside a database
  - __init__(self, transaction): Constructor for the Collection class
  - name(self): Returns the name of the collection
  - arrays(self): Return all the arrays in this collection
  - insert(self, array): Inserts an array in this collection
  - update(self, array): Updates the array in the collection
- Transaction: Class to represent a transaction on the selected database
  - __init__(self, database): Constructor for the Transaction class
  - query(self, query_str): Returns a new query object initialized with this transaction and the query string
- Query: Class to represent a rasql query that can be executed in a certain transaction
  - __init__(self, transaction, query_str): Constructor for the Query class
  - execute(self): Executes the query and returns back a result
- BandType: Enum containing possible band types in rasdaman

```
INVALID = 0,
CHAR = 1,
USHORT = 2,
SHORT = 3,
ULONG = 4,
LONG = 5,
FLOAT = 6,
DOUBLE = 7
```

- SpatialDomain: Class to represent a spatial domain in rasdaman
  - __init__(self, *interval_parameters): Constructor for the SpatialDomain class
- ArrayMetaData: Class to represent the metadata associated to an array
  - __init__(self, spatial_domain, band_types): Constructor for the ArrayMetaData class
- Array: Class to represent an array produced by a rasdaman query
  - __init__(self, metadata, values): Constructor for the Array class
  - __getitem__(self, item): Operator overloading for [ ], it will slice the array on the first dimension available
  - subset(self, spatial_domain): Subsets the array based on the given spatial domain
  - get(self): Returns the array contents as a one dimensional list containing a tuple of each band value for the array cell
  - point(self, *dimensional_indices): Returns the point at the given position
  - toArray(self, type): Returns the NumPy array for a given RasArray

The given class structure on the client side will work with either of the approaches mentioned beforehand and can be adjusted accordingly to which ever approach we choose.

4 CONCLUSION

This document has underlined two methods that can be used for developing the project and as mentioned some of them might work better than others. Based on those, the document has set the direction for the project by taking an active decision on which approach to follow.

5 REFERENCES

[1] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann, "The multidimensional database system rasdaman," *ACM SIGMOD Record*, June 1998.

[2] rasdaman GmbH, *RasQL - Query Language Guide*, August 2003.

[3] J. VanderPlas and C. Beaumont, "Scidb-py : A python interface to scidb," August 2013. [Online]. Available: http://scidb-py.readthedocs.org/