PyRDF: Distributed RDataFrame

Summary of the current status

V. E. Padulano, J. Cervantes, E. Tejedor
Outline

▶ What is it?
▶ New Features
▶ Tests
▶ Use cases
What is it?
Started as a GSOC project in 2018

**Python layer** on top of RDataFrame

Enable RDataFrame analysis to run with **distributed resources**

PyROOT to access ROOT from Python

Spark backend

[PyRDF repo](#)
Map-reduce workflow where every mapper runs the RDataFrame computation graph on a range of collision events.

Spark provides APIs in Scala, Java, Python.

Run **analysis in C++ with Spark**

- Exploiting its python API and PyROOT.
New Features
New releases

- **04/19**: No release
- **05/19**: PyRDF 0.1
  - Available in LCG 96
- **09/19**: PyRDF 0.2
  - In the nightlies and LCG 96b

New features, improvements and bugfixes
At my arrival:

- Minimal implementation
- Only python 2
- Unstable graph pruning
- No official docs
- Missing RDF operations

PyRDF 0.1:

- Functional implementation
- First PyRDF release on LCG
- Full Python 3 support
- Stable and improved graph pruning
- Send C++ headers and shared libraries to Spark

PyRDF 0.2:

- Add friend trees compatibility
- Add logging capabilities
- Official docs created
- Many RDF operations implemented with Spark
C++ headers to distributed executors

- ROOT allows user-defined headers and shared libraries
- But they are not sent to the Spark executors at runtime

RDataFrame computational graph

- Data: x, y
- Filter: x > 0
- Define: \( r_2 = x^2 + y^2 \)
- TGraph: x, y
- TH1D: r2

myheader.h
libmylib.so
import PyRDF
PyRDF.use("spark")
PyRDF.include_headers("myheader.h")
df = PyRDF.RDataFrame(256)
df.Filter("f(rdfentry_)").Count();

myheader.h

#ifndef myheader
#define myheader

bool f(int num){
    return num < 5;
}
#endif

header = SparkFiles.get("myheader.h")
Utils.declare_headers(header)

Also working with directories of headers and shared libraries

Users can send headers at runtime
*before: only at connection time*
Many operations which were previously unavailable in a distributed environment now are:

- Count, Sum, statistics in general (through Stats)
- Snapshot
- AsNumpy
import ROOT
df = ROOT.RDataFrame(10000)
    .Define("x", "rdfentry_")
    .Define("y", "x*x")
df.Snapshot("treename", "file.root")

import PyRDF
PyRDF.use("spark")
df = PyRDF.RDataFrame(10000)
    .Define("x", "rdfentry_")
    .Define("y", "x*x")
df.Snapshot("treename", "file.root")

Save to a file in the local machine.

Save the distributed chunks of data to an EOS path the user decides.
import PyRDF
PyRDF.use("spark")

[RDF operations...]
df.Snapshot(eospath)

Path to a EOS file:
root://eosuser.cern.ch//mypath/myfile.root

Cluster ranges
1 - 100
101 - 200
901 - 1000

Mapper
Mapper
Mapper

TChain
...myfile_1_100.root
...myfile_101_200.root
...myfile_901_1000.root

Files stored
Save back to EOS

1313
import ROOT
df = ROOT.RDataFrame(10000)
    .Define("x","rdfentry_")
    .Define("y","x*x")
npy = df.AsNumpy()

import PyRDF
PyRDF.use("spark")
df = PyRDF.RDataFrame(10000)
    .Define("x","rdfentry_")
    .Define("y","x*x")
npy = df.AsNumpy()

A collection (dictionary) of numpy arrays corresponding to the columns of the RDataFrame
Documentation is built with **sphinx**

Automatically created from docstrings

Hosted on [Read the Docs!](https://readthedocs.org)
Logging PyRDF

Logs implemented with Python standard module **logging**
Create logger in the Python **script**

```python
import logging
logger = logging.getLogger()
logger.setLevel(logging.DEBUG)
[add handlers and customize formatting]
```

```python
import PyRDF
logger = PyRDF.create_logger("DEBUG")
[ RDF stuff ... ]
```
A comparison between different configurations for running the dimuon tutorial on SWAN:

- locally, with 4 threads enabled
- on the spark executors (1 core each)
  - 4 executors
  - 8 executors
  - 16 executors
  - 32 executors
Multithreaded [4 threads]
- AVG time [s]: 60.5 ± 12.7
- Most frequent time [s]: 61.2

Spark [4 executors, 1 core each]
- AVG time [s]: 59.1 ± 32.5
- Most frequent time [s]: 46.8
Dimuon Analysis Tests

Multithreaded [4 threads]
- AVG time [s]: 60.5 ± 12.7
- Most frequent time [s]: 61.2

Spark [4 executors, 1 core each]
- AVG time [s]: 59.1 ± 32.5
- Most frequent time [s]: 46.8

Most frequent time is much lower with Spark
Dimuon Analysis Tests

- Good scaling up to 8 cores
- But different scaling factor
- Speedup ~ 1 after 8 cores
Use cases
ATLAS use case

ATLAS user with a ROOT RDataFrame based analysis

Intricated package with multiple dependencies

Simple PyRDF test for the Friend TTree functionality

Install latest PyRDF version

```python
# Configuration to the SparkContext is added via the SWAN interface as follows:
# https://github.com/JavierCullin/PyRDF/tree/master/demos
import sys
sys.path.insert(0, "~/es6/user/v/spadlan/ATLAS/ATLAS-MBD/ROOT/PyRDF")
sys.path.append("~/es6/user/v/spadlan/ATLAS/ATLAS-MBD/ROOT/PyRDF")
sys.path.append("~/es6/user/v/spadlan/ATLAS/ATLAS-MBD/ROOT/PyRDF")
# Add python module (temporal)
sx.addFile("~/PyRDF.zip")
```

Configure

```python
import ROOT
import PyRDF

# PyRDF = ROOT

#setenv on
PyRDF.use("spark")

Welcome to JupyterKernel 6.18/81

Base TTree: define and plot

```python
In [3]:
baseTree = ROOT.TTree("Nominal")
# File from the project
baseTree = Add("root://es6/user/v/spadlan/ATLAS/ATLAS-MBD/ROOT/PyRDF")
```

New Feature developed: Friend trees with a Spark backend!
CMS user developing his own physics analysis within a python framework.

- Analysis translated from traditional ROOT to RDataFrame
- PyRDF used to connect to the Spark cluster and retrieve numpy arrays after column definition and filtering

Very early stage, but this approach can be easily extended to any NanoAOD analysis.

preliminary presentation, analysis repo
A challenging use case

Data from the TOTEM experiment

- Analysis requires **CMSSW** framework to run.
- SWAN and the Spark clusters take software from CVMFS, but only include the **SFT** repository.

Currently working towards integrating CMSSW in SWAN (first successful attempt in the user session) and the Spark executors.
Thank you!
During map phase:

1. Retrieve info about friend trees
2. Friend trees share same data range as their main tree
3. Build a chain of friend trees and add it to the main tree.

```python
def get_friend_info(TTree):
    friends = TTree.GetListOfFriends()
    for friend in friends:
        friend_tree = friend.GetTree()  # ROOT.TTree
        real_name = friend_tree.GetName()
        friend_filename = friend_tree.GetCurrentFile().GetName()

# In map function
if friend_info:
    for friend_treename, friend_filenames in tree_files_names:
        # Start a TChain with the current friend treename
        friend_chain = ROOT.TChain(friend_treename)
        # Add each corresponding file to the TChain
        for filename in friend_filenames:
            friend_chain.Add(filename)
        # Set cache on the same range as the parent TChain
        friend_chain.SetCacheEntryRange(start, end)
        # Finally add friend TChain to the parent chain
        chain.AddFriend(friend_chain)
```
Graph Pruning

Before release 0.1:

1. Check if node has children
2. If it has, check the children first
3. Check how many python objects are referencing the node
4. Check if the node is an action with an already computed value
5. If either (3) or (4) are true then the node can be pruned
6. Repeat on all the children

The condition on the number of referrers was very unstable!
Graph Pruning

After release 0.1:

1. Check if node has children
2. If it has, check the children first
3. Check if the proxy flagged the node to be non-referenced by the user
4. Check if the node is an action with an already computed value
5. If either (3) or (4) are true then the node can be pruned
6. Repeat on all the children

Now we can know precisely when the user doesn’t need a node operation anymore.