Distributed Data Analysis with ROOT RDataFrame

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

ROOT
Data Analysis Framework

https://root.cern
RDataFrame, officially part of ROOT since v6.14, tries to incarnate these ideas in the context of HEP analyses and HEP data manipulation.
RDataFrame in a Nutshell

Datasource

- ROOT
- CSV
- Apache Arrow
- SQLite
- ...

Define

- p_x
- p_y
- p_z
- eta
- myvar

Range Filter

- histograms, profiles
- new ROOT files
- cut-flow reports
- data reductions (mean, sum,..)
- any user-defined operation

Datasource operations

- Define
- Range Filter
- Filter

Datasource types

- ROOT
- CSV
- Apache Arrow
- SQLite
- ...
from ROOT import RDataFrame

df = RDataFrame(dataset);
df2 = df.Filter("x > 0")
  .Define("r2", "x*x + y*y");

rHist = df2.Histo1D("r2");
g = df2.Graph("x","y")
The RDataFrame programming model is implicitly parallel
- Runs on multi/many core architectures
- But it can also exploit **distributed infrastructures**!

**PyRDF**: Python library on top of ROOT RDataFrame
- Enables distributed execution of RDataFrame workflows
- Modular design: multiple backends can be plugged in

**Spark** plugin implemented: submits RDataFrame computations to Spark clusters

```python
import ROOT

d = RDataFrame(dataset)
f = d.Define(...)
  .Define(...)
  .Filter(...)

h1 = f.Histo1D(...)  
h2 = f.Histo2D(...)  
g   = f.Graph(...)```

Distributed data analysis with ROOT RDataFrame, CHEP 2019
Spark Backend

- **Map-reduce** workflow where every mapper runs the RDataFrame computation graph on a range of collision events
- Run **analysis in C++ with Spark**
  - Exploiting its Python API and [PyROOT](https://pyroot.readthedocs.io)

![Spark Backend Diagram]

**Spark Plugin**

- **Driver**
- Make ranges
- Schedule tasks

**Spark Cluster**

- **Executor**
- **Mapper**
- **Reducer**
- Read ranges

**Data Structure**

```
<table>
<thead>
<tr>
<th>px</th>
<th>py</th>
<th>pz</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Features Overview
Minimal changes on user’s code

```python
from ROOT import RDataFrame

# Initialize RDataFrame object
df = RDataFrame(dataset)

# Define operations
df2 = df.Filter("x > 0")
    .Define("r2", "x*x + y*y")
rHist = df2.Histo1D("r2")

# Display histogram
rHist.Draw()
```
Minimal changes on user’s code

```python
from ROOT import RDataFrame

# Initialize RDataFrame object
df = RDataFrame(dataset)

# Define operations
df2 = df.Filter("x > 0")
    .Define("r2", "x*x + y*y")
rHist = df2.Histo1D("r2")

# Display histogram
rHist.Draw()
```

```python
from PyRDF import RDataFrame

# Initialize RDataFrame object
df = RDataFrame(dataset)

# Define operations
df2 = df.Filter("x > 0")
    .Define("r2", "x*x + y*y")
rHist = df2.Histo1D("r2")

# Display histogram
rHist.Draw()
```
Multi-backend support

- Dynamic switch of backends

```python
# Select Spark backend
PyRDF.use("spark")

# Initialize RDataFrame object
df = RDataFrame(dataset)

# Operations run in Spark
df2 = df.Filter("x > 0")
    .Define("r2", "x*x + y*y")
rHist = df2.Histo1D("r2")
# Trigger event loop
sd = rHist.GetStdDev()

# Switch back to Local backend
PyRDF.use("local")

# Operations run locally
df3 = df2.Filter("r2 % 2 == 0")
```
API: C++ Headers and Libraries

- Include **C++ headers and libraries**
  - PyRDF makes them available in the distributed nodes

```cpp
#include <myfunc.hxx>

PyRDF.include_headers("myfunc.hxx")
PyRDF.include_shared_libraries("myfunc.so")

# Initialize RDataFrame object
df = RDataFrame(dataset)

df2 = df.Define("res", "myfunc(x,y)")
```

```cpp
#include <myfunc.hxx>

bool myfunc(int a, int b);

myfunc.hxx
bool myfunc(int a, int b);

myfunc.cxx
bool myfunc(int a, int b) {
    return a < b;
}
```

Calls from JITted code
RDataFrame Snapshot

- RDataFrame Snapshot allows to save data to a file

```python
new_df = df.Filter("x > 0")
    .Define("z", "sqrt(x*x + y*y)")
    .Snapshot("tree", "newfile.root")
```

We filter the data, add a new column, and then save everything to file.
import PyRDF
PyRDF.use("spark")

# RDF Operations ...
new_df = df.Snapshot(remote_path)
Use Case
Real Example: TOTEM Analysis

- TOTEM Experiment analysis coded with RDataFrame
  - Proton-proton elastic scattering
- Spark backend
- 4.7TB input dataset on EOS
- Get to physics results faster!
  - From 13 hours to 10 minutes
- Launched from SWAN to a dedicated Spark cluster

[Link to Euro-Par 2019 paper]

![Graph showing speedup vs. number of cores](image)
- **Bridge the gap** between interactive computing and distributed data processing
- Automatically appears when a Spark job is submitted from a cell
- Progress bars, task timeline, resource utilisation

Code here!
Useful for Debugging
The increase in physics data volumes and complexity is **challenging** software in HEP

- Adoption of Spark and other Big Data technologies still in its early stages

Adopting new programming paradigms takes time

- Declarative analysis

RDataFrame and **PyRDF** try to combine:

- Easy to use programming model
- Implicit parallelization (local, distributed)

**To try it out:** PyRDF is distributed with LCG releases (from LCG 96)

- SWAN provides additional goodies: easy connect to Spark clusters, live monitoring
Thank you!
Backup
Reasons to run distributedly

- The amount of data processed by HEP scientists is going to increase drastically

Distributed data analysis with ROOT RDataFrame, CHEP 2019

sources: CMS, ATLAS
PyRDF: Main design principles

- Delay computations as much as possible
- Avoid data format conversion
- Change the backend dynamically
- Minimal changes on user's code
  - Changing the mindset of programmers takes time
  - Keep the same interface offered by RDataFrame in Python
  - Support local as a backend
Entrypoint to backend configuration
- Explicit parameters
- Accept all backend parameters

```python
import PyRDF

# Configure Spark backend
PyRDF.use("spark", {
    "npartition": 4,
    "spark.executor.instances": 5
})

# Initialize RDataFrame object
df = PyRDF.RDataFrame(dataset)
```
Integration with Spark

Spark Cluster

Spark Master

Spark Executor

Task

Task

Task

User Notebook

Spark Driver

Offload computations to pluggable resources
Spark Connector

Configure Spark and connect to cluster with a click.
Spark Monitor: Debugging

- Easy to spot inefficiency situations
  - Optimize use of resources (cores)