Distributed Data Analysis with ROOT RDataFrame

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

ROOT
Data Analysis Framework
https://root.cern

Introduction

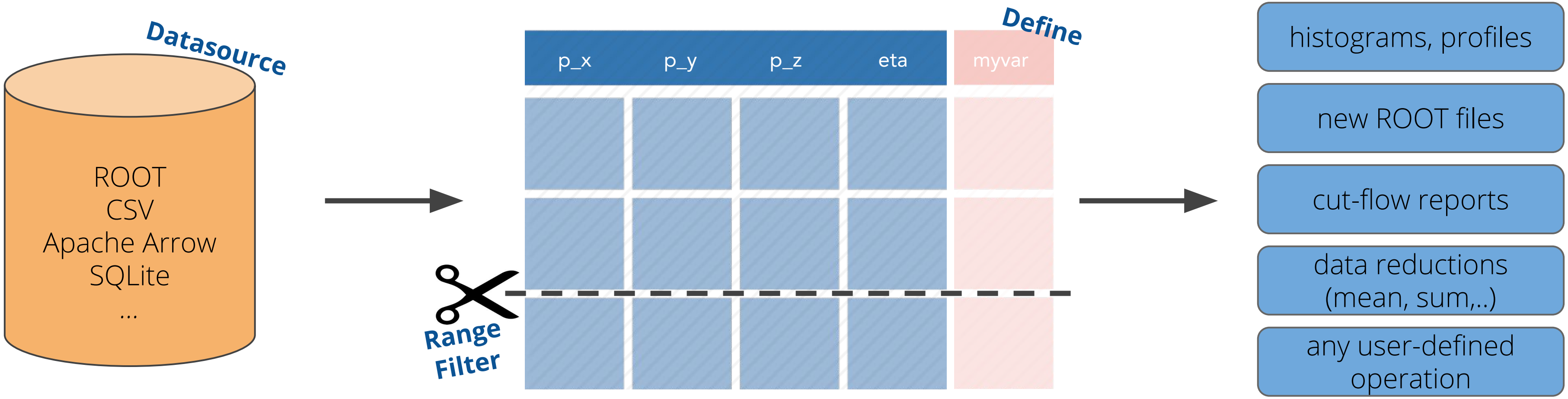
The HEP DataFrame

- strive for a **simple programming model** based on declarative programming
- expose modern, elegant interfaces that are **easy to use correctly** and hard to use incorrectly
- **transparently benefit from multi-core** hardware
- make **common tasks simple, complex tasks possible**
- consistent support for HEP languages: **C++ and Python**

RDataFrame, officially part of ROOT since v6.14, tries to incarnate these ideas in the context of HEP analyses and HEP data manipulation

[RDF user guide, cheat-sheet]
DOI 10.5281/zenodo.260230

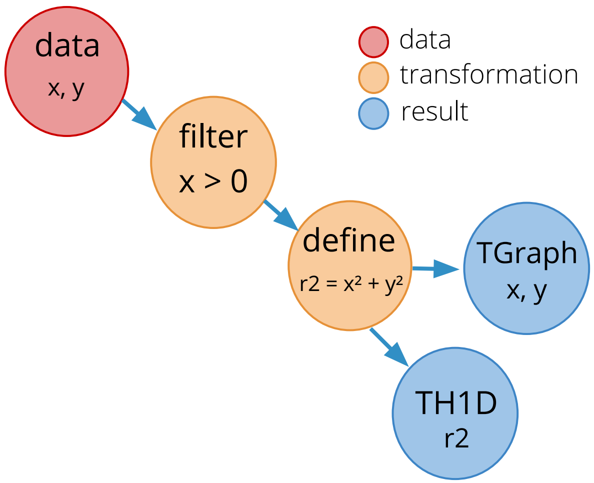
RDataFrame in a Nutshell



Analysis as Computation Graphs

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

rHist = df2.Histo1D("r2");

g = df2.Graph("x","y")
```

Internal computation graph

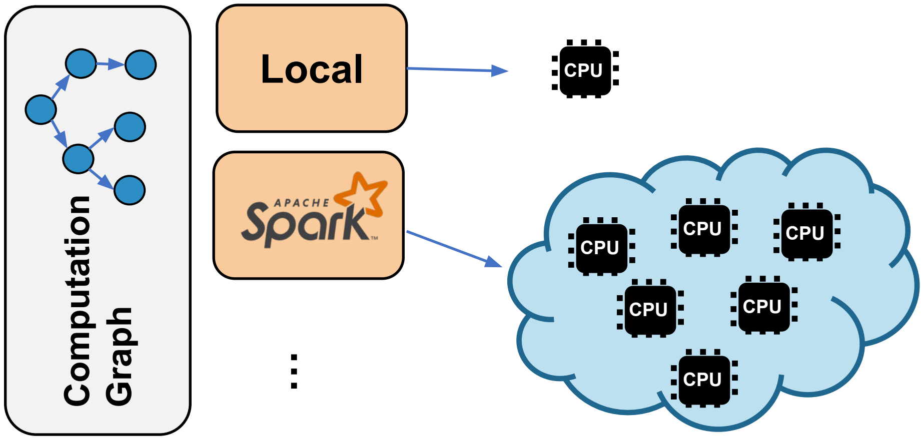


PyRDF: Distributed RDataFrame

- 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** backend implemented: submits RDataFrame computations to Spark clusters

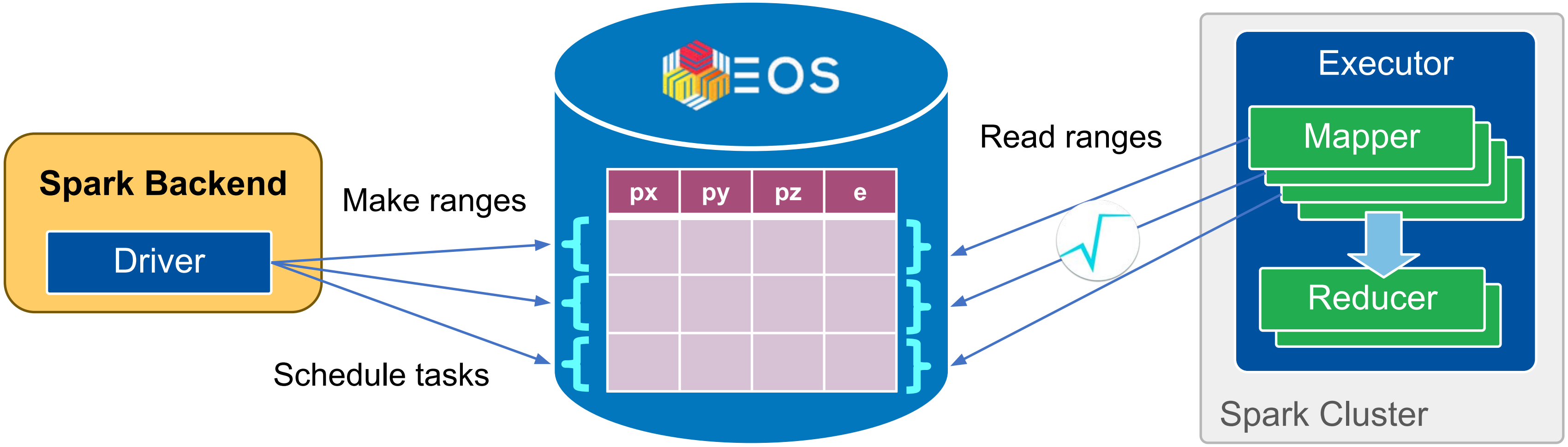
Code here!

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

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



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]



Features Overview

Programming Model

- Minimal changes on user's code

```
import ROOT

# Initialize RDataFrame object
df = ROOT.RDataFrame(dataset)

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

# Display histogram
rHist.Draw()
```

Programming Model

- Minimal changes on user's code

```python
import ROOT

# Initialize RDataFrame object
df = ROOT.RDataFrame(dataset)

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

# Display histogram
rHist.Draw()
```

RDataFrame via PyRDF

```python
import PyRDF

# Initialize RDataFrame object
df = PyRDF.RDataFrame(dataset)

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

# Display histogram
rHist.Draw()
```

API: Backend Selection

- Multi-backend support
  - Dynamic switch of backends

Move to local backend

Spark

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

# Initialize RDataFrame object
df = PyRDF.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")
```

Local

API: C++ Headers and Libraries

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

myfunc.hxx

```cpp
bool myfunc(int a, int b);
```

myfunc.cxx

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

```python
# Add analysis headers and libraries
PyRDF.include_headers("myfunc.hxx")
PyRDF.include_shared_libraries("myfunc.so")

# Initialize RDataFrame object
df = PyRDF.RDataFrame(dataset)

# Operations run in distributed backend
df2 = df.Define("res", "myfunc(x,y)")
```

Calls from JITted code

RDataFrame Snapshot

- RDataFrame Snapshot allows to save data to a file

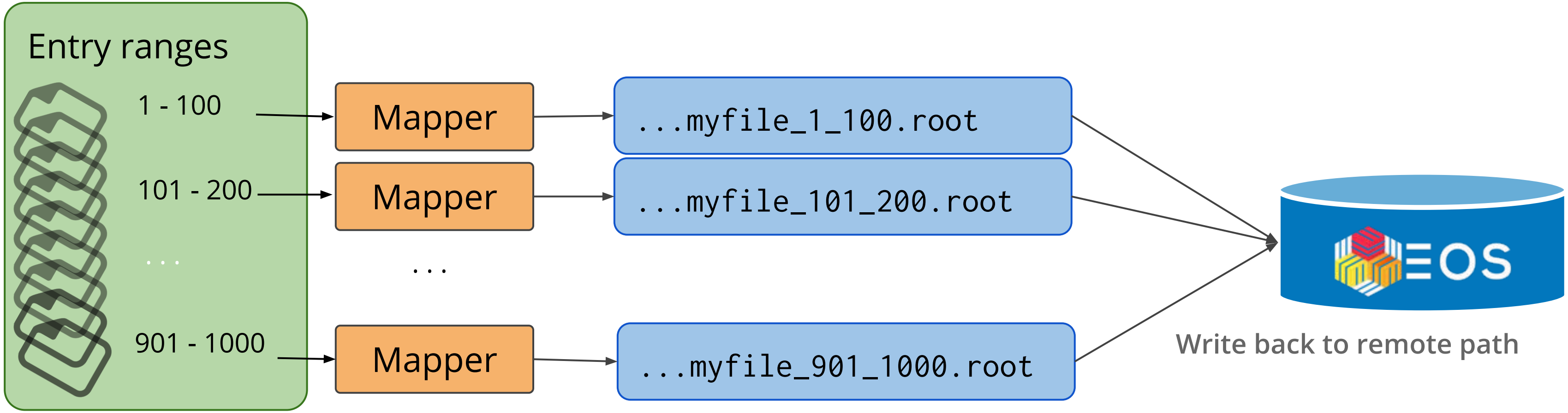
```
new_df = df.Filter("x > 0")
           .Define("z", "sqrt(x*x + y*y)")
           .Snapshot("tree", "newfile.root")
```

We filter the data, add a new column, and then save everything to file

Distributed Snapshot

```python
import PyRDF
PyRDF.use("spark")

# RDF Operations ...

new_df = df.Snapshot(remotepath)
```

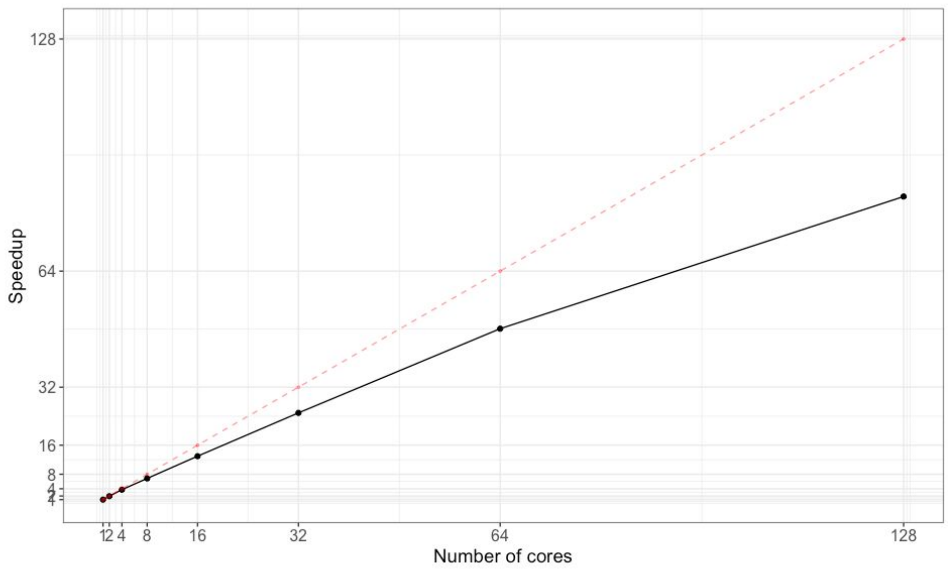
Path to a remote file:
`root://eosuser.cern.ch//mypath/myfile.root`



Use Case

Real Example: TOTEM Analysis

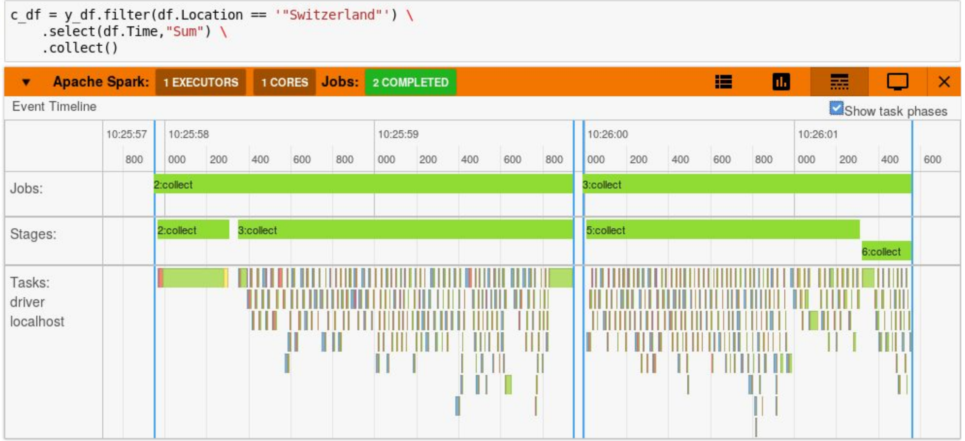
- TOTEM Experiment analysis coded with RDataFrame
- Spark backend
- 4.7TB dataset on EOS
- Get to physics results faster!
  - From 13 hours to 10 minutes
- Launched from SWAN to a dedicated Spark cluster

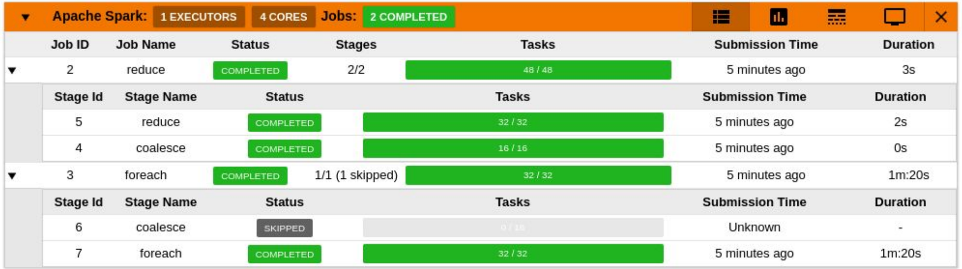


Distributed Monitoring

- **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

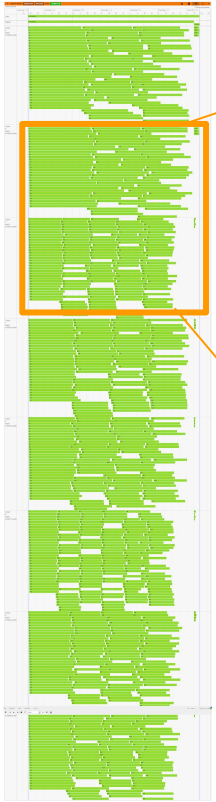
[Code here!]





Job ID	Job Name	Status	Stages	Tasks	Submission Time	Duration
2	reduce	COMPLETED	2/2	48 / 48	5 minutes ago	3s
Stage Id	Stage Name	Status		Tasks	Submission Time	Duration
5	reduce	COMPLETED		32 / 32	5 minutes ago	2s
4	coalesce	COMPLETED		16 / 16	5 minutes ago	0s
3	foreach	COMPLETED	1/1 (1 skipped)	32 / 32	5 minutes ago	1m:20s
Stage Id	Stage Name	Status		Tasks	Submission Time	Duration
6	coalesce	SKIPPED		0 / 16	Unknown	-
7	foreach	COMPLETED		32 / 32	5 minutes ago	1m:20s

Useful for Debugging



Conclusions

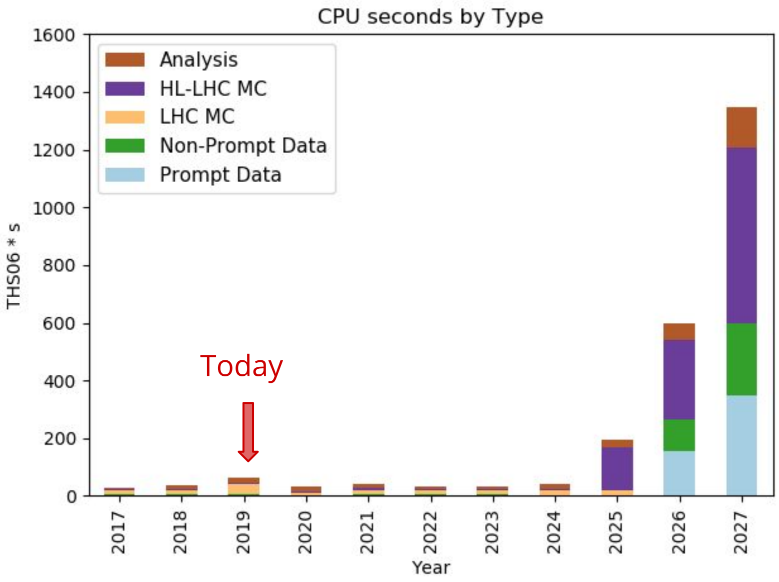
- The increase in physics data volumes and complexity is **pushing software** at CERN
  - Adoption of Spark and other big data technologies still in its early stages
- Adopting new programming paradigms takes time
  - Declarative analysis
  - Pushing computations to data
- RDataFrame and **PyRDF** try to combine:
  - Easy to use programming model
  - Implicit parallelization (local, distributed)

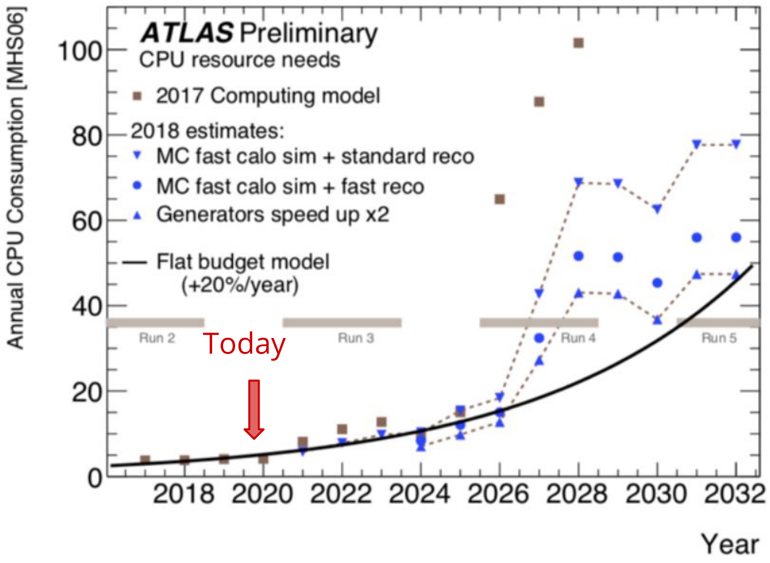
Thank you!

Backup

Reasons to run distributedly

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

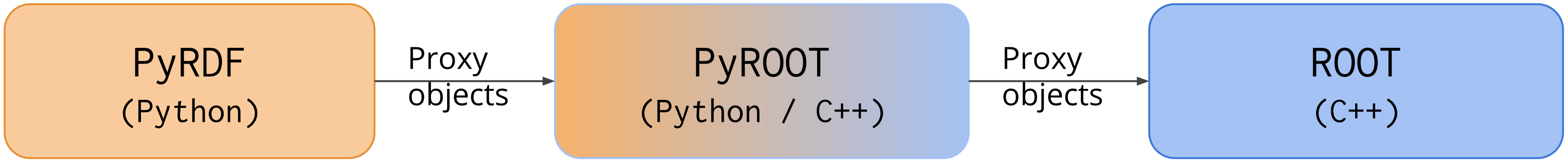




sources: CMS, ATLAS

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



Backend Configuration

- Entrypoint to backend configuration
  - Explicit parameters
  - Accept all backend parameters

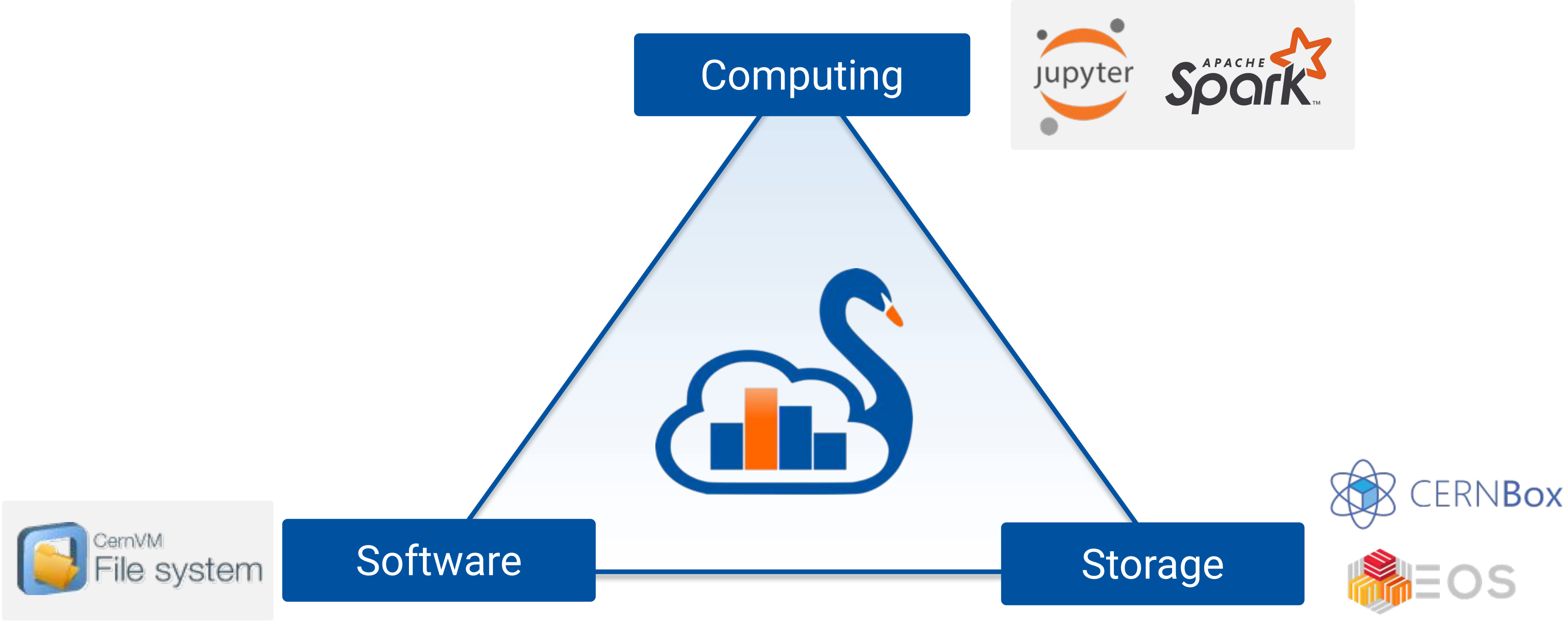
```python
import PyRDF

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

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

The SWAN Service

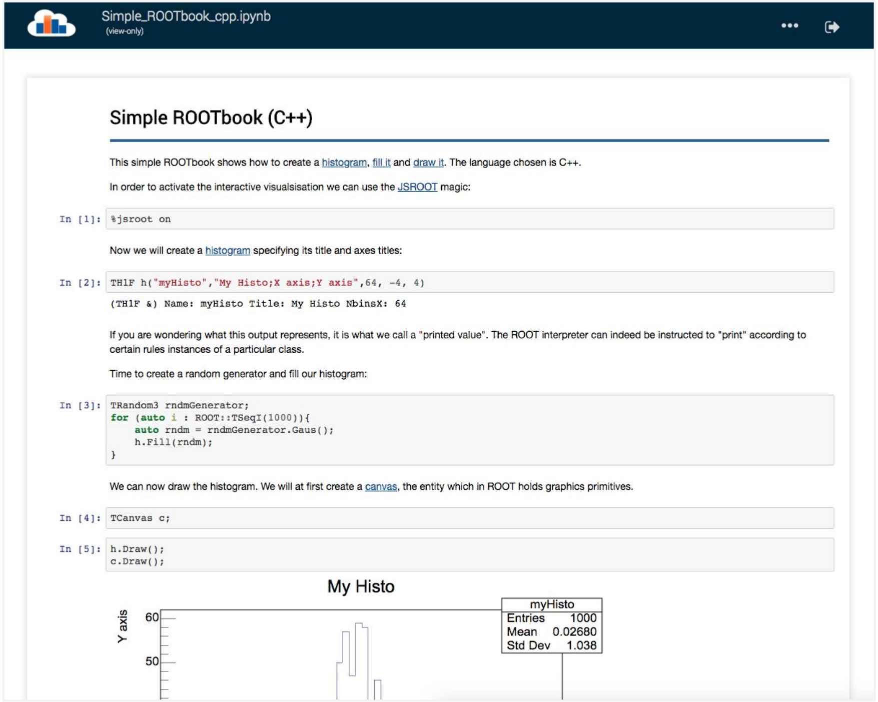
- **SWAN**: Service for Web-based Analysis
- **Interactive computing** platform for scientists
  - Based on Jupyter notebooks
- Analysis with only a web browser
- Easy **sharing of results**
- Integrated with CERN resources
  - Storage, software and computing

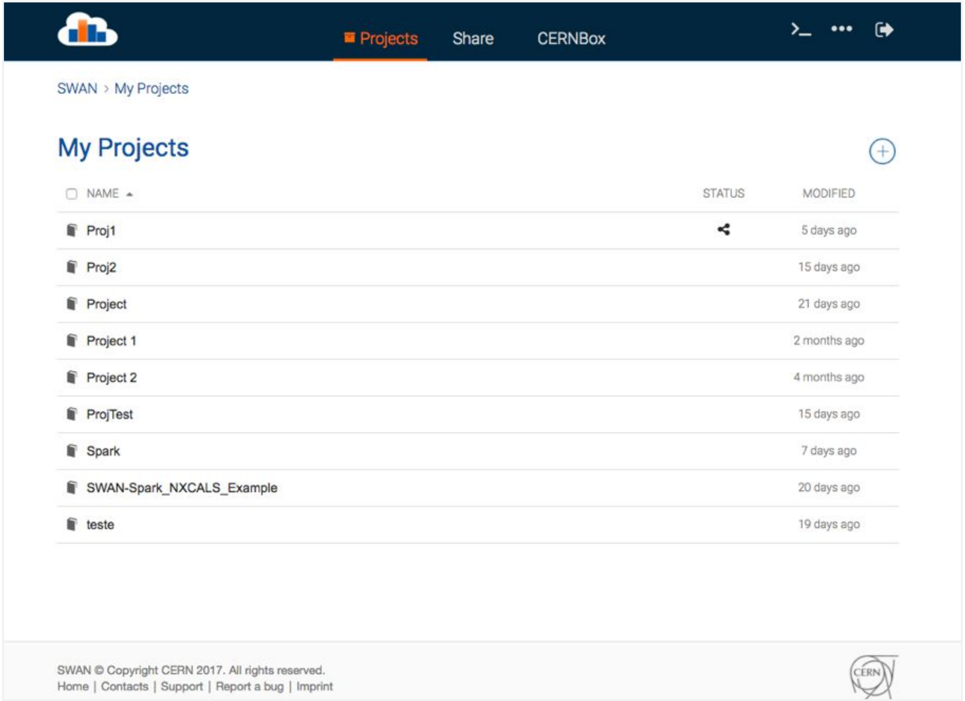
https://swan.web.cern.ch

SWAN Pillars

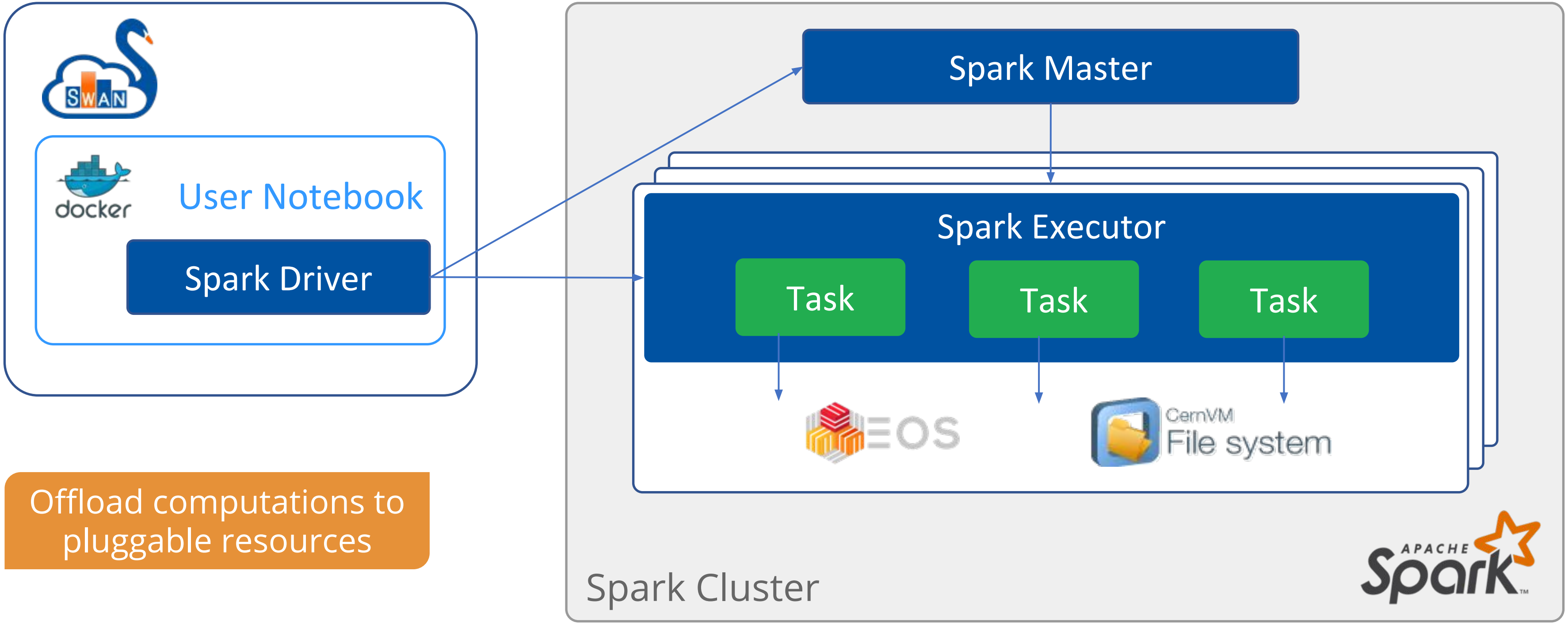


SWAN Interface: Notebooks



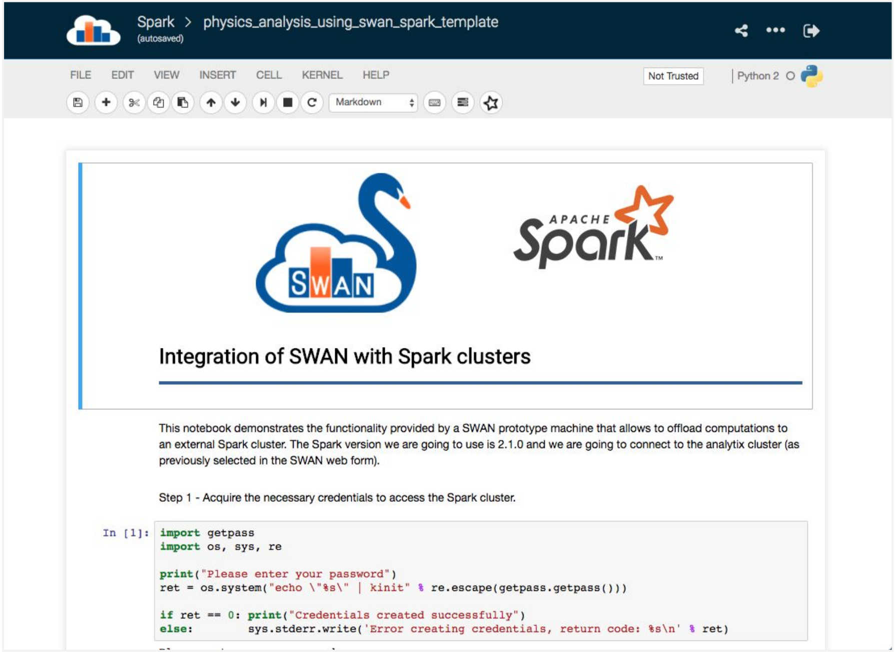


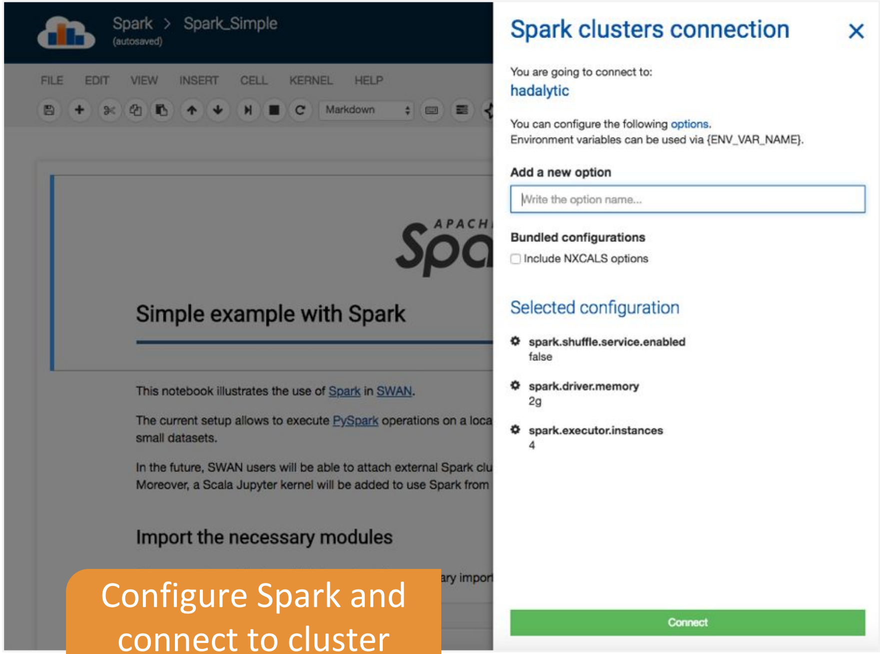
Integration with Spark



Offload computations to pluggable resources

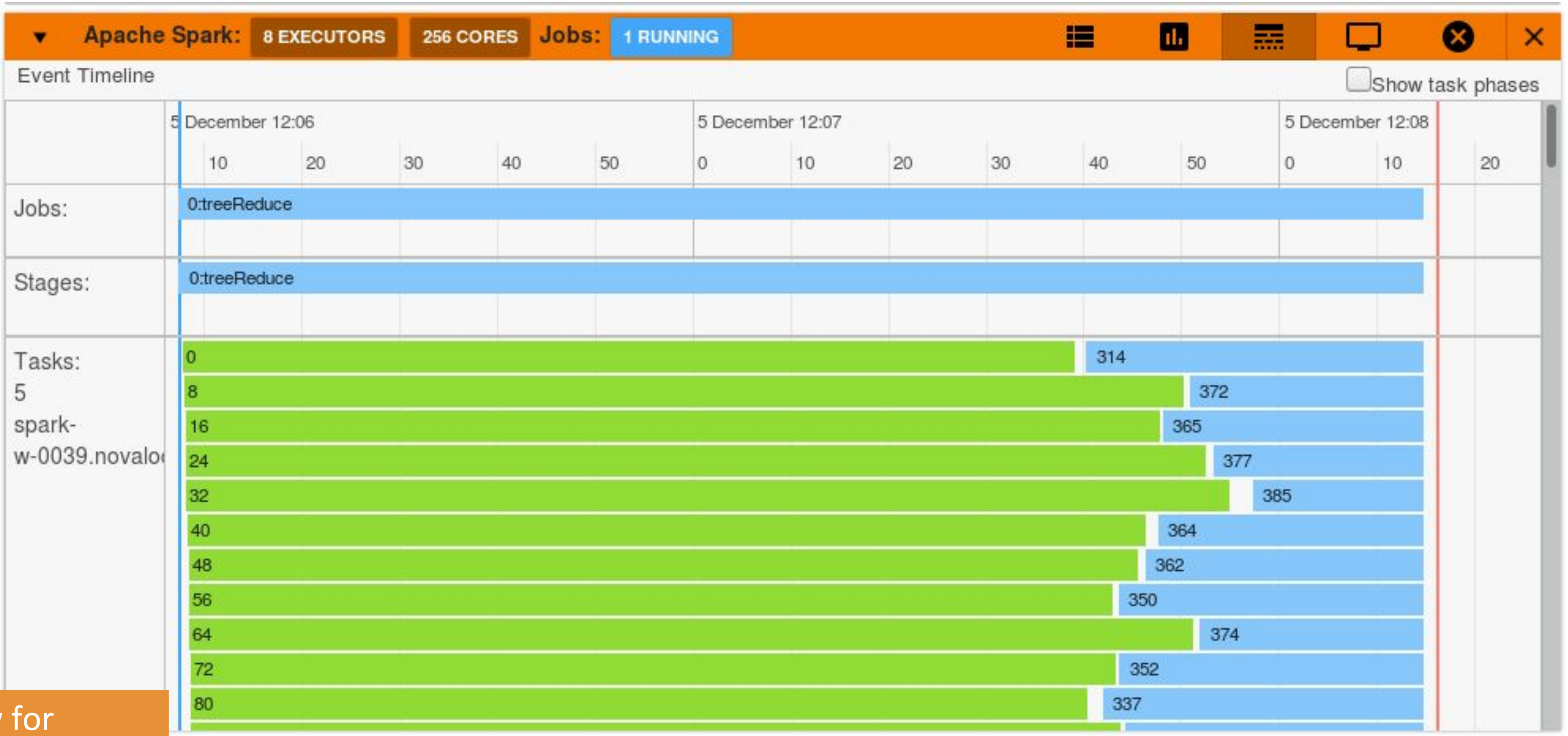
Spark Connector





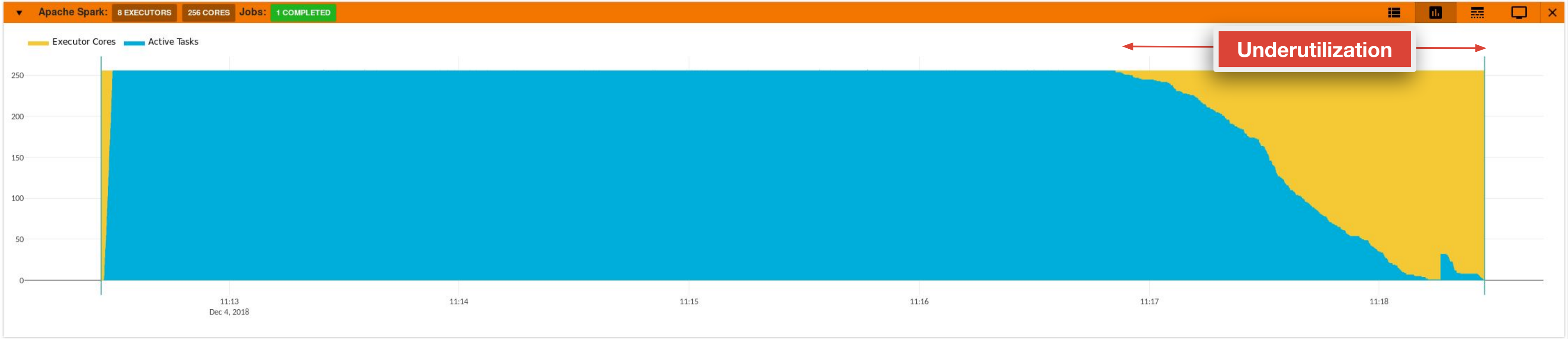
Configure Spark and connect to cluster with a click

Spark Monitor



Allow for optimizations

Useful for Debugging




```
CPU times: user 4.52 s, sys: 2 s, total: 6.53 s
Wall time: 6min 2s
```

- Easy to spot sources of inefficiencies
  - Optimize use of resources (cores)
- Led to a better way to manage the task ranges
  - ROOT I/O: prefetch/cache only within task ranges
  - Parallelize generation of ranges