Lessons learned converting a production-grade Python CMS analysis to distributed RDataFrame

Diego Ciangottini[1], Enrico Guiraud[3], Vincenzo Eduardo Padulano[4], Daniele Spiga[1,2], Tommaso Tedeschi[1,2], Enric Tejedor Saavedra[3], Mirco Tracolli[1]

[1] INFN Perugia, Italy
[2] University of Perugia, Italy
[3] CERN, Switzerland
[4] Valencia Polytechnic University, Spain

Outline

- INFN analysis infrastructure objectives
- New distributed RDataFrame features
- A new INFN analysis facility prototype
- Performances
- Demo
- Conclusions

Thanks to other contributors:

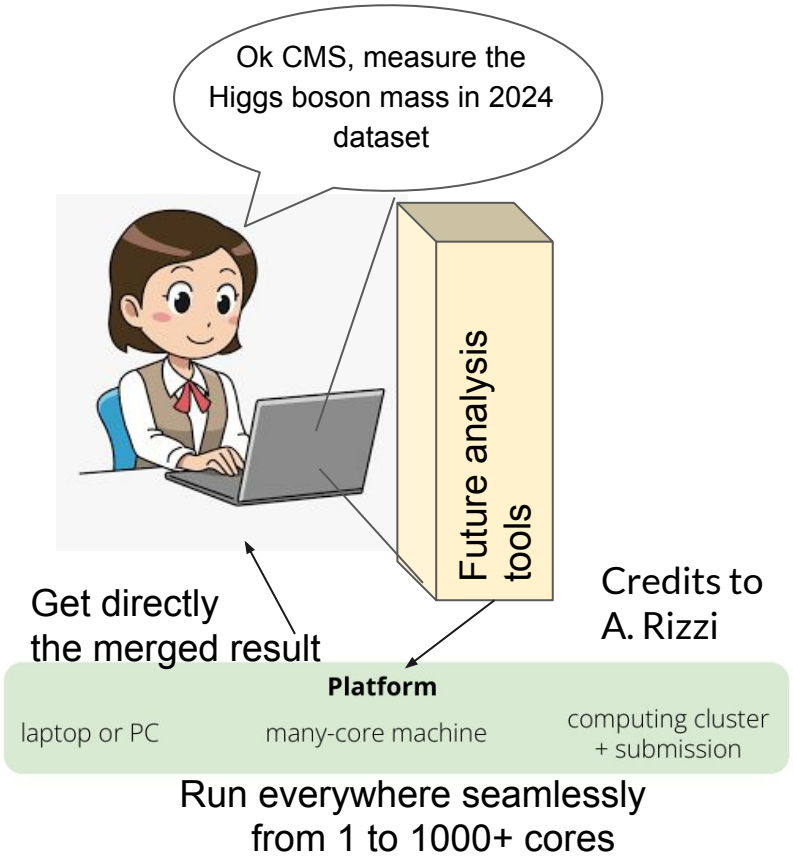
- Tommaso Boccali
- Massimo Biasotto
- Massimo Sgaravatto
- Stefano Nicotri
- Francesco Failla

The Objective

What are we talking about (in a nutshell)

An **R&D project** started at the end 2021 to study if / how to **improve resources usage** for data analysis and (more challenging) how to enable the **exploitation of new approaches, new paradigms for analysing data** at CMS. Looking at Phase2 but targeting already Run3.

- Avoid wheels, do physics
  - Do not code event loops, but rather **declare only what you want to do** in the end
- Let the **framework optimize things**
  - No configuration for data splitting or for explicit multi-threading
  - Get the best throughput out of the infrastructure
- Share and reuse **"code for humans"**
  - Easier to debug
  - Harder to get lost



User's perspective

Fast turnaround : how to allow the user to analyze billions on NanoAOD within N hours

- fast iteration time is essential for debugging experimental/theoretical/technical issues and for developing/improving the analysis
  - For O(billions) of events this means event throughput in the MHz

Instead of..

- **Submit** O(1000) single core jobs to condor batch system reading from mass storage, writing O(200MB) of histograms to afs
- **Resubmit** the fraction of jobs which failed the first time
  - and the others which failed the second time..
- **Merge**

What system do we need?

A system that grants access to computing resources for analysis and enables a hybrid model: batch and interactive patterns –> Not a one size fits all solution

Interactive - "Read this as: I can get a Jupyter notebook as big as a Tier2"

- Transparently parallelize over a huge amount of cores allows implementing the interactivity
- I'm writing Jupyter, you can read it as distributed python
- And more in general mitigate/avoid user waiting idle for grid jobs

Batch like processing - "Read this as: I have a place where I can submit (i.e condor_submit) my analysis jobs"

- Yes, "yet another batch"... completely dedicated to analysis

What analysis tool do we need?

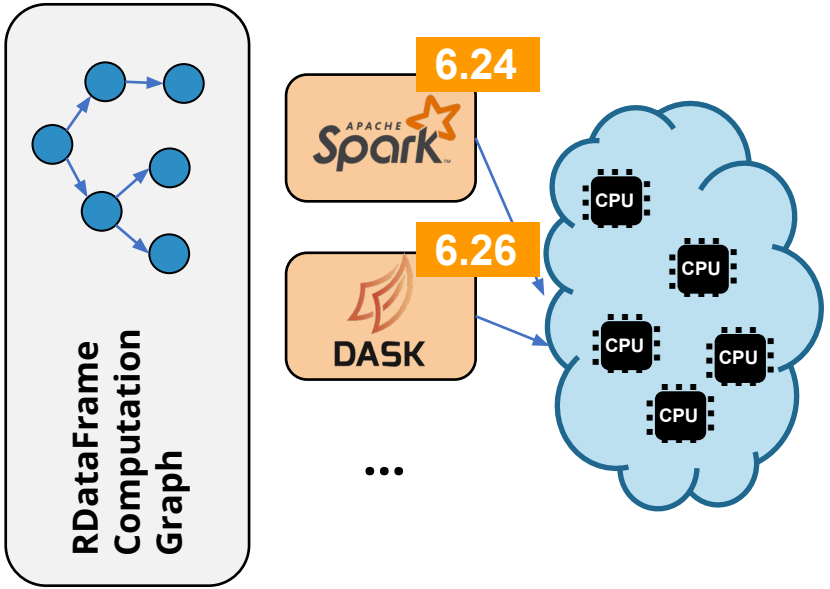
A declarative, efficient, distributed analysis tool: e.g. Distributed RDataFrame

The analysis tool

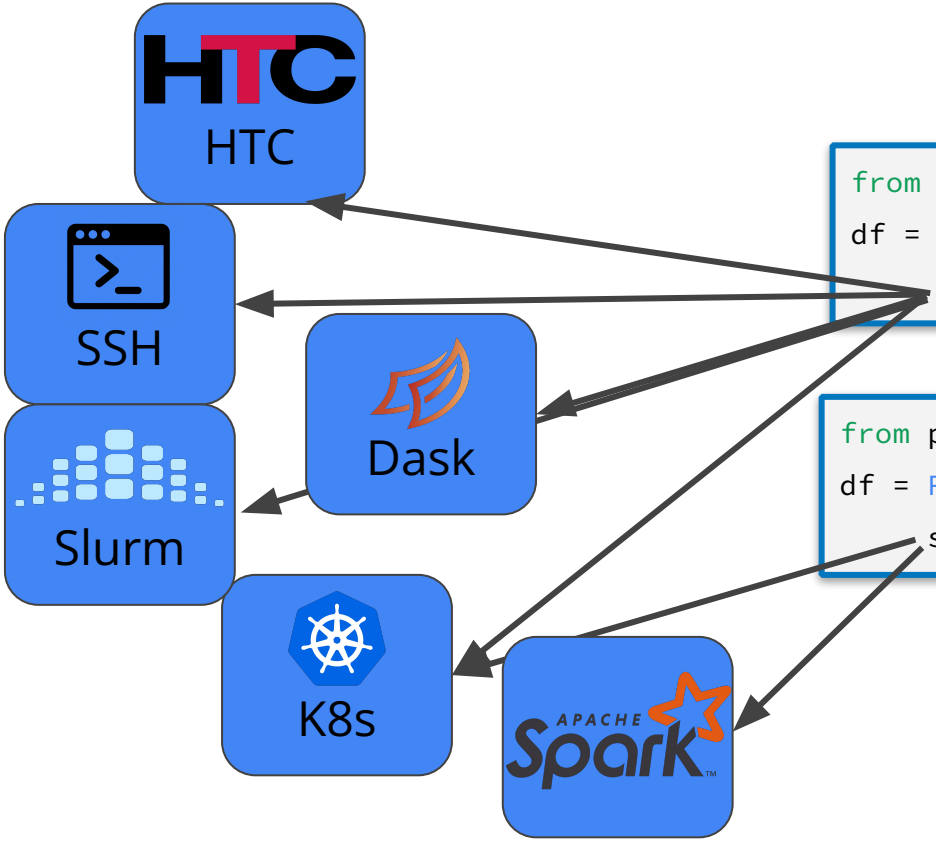
Distributed RDataFrame: what's new

Distributed RDataFrame - a recap

- Launches an **RDataFrame** application on a cluster
- **Automatic splitting** of the workflow
- Takes care of running jobs and **merging** results
- Can run with different schedulers: **Dask**, **Spark**, ...
- Analysis from start to end in a **single interface**



One API, Many Backends

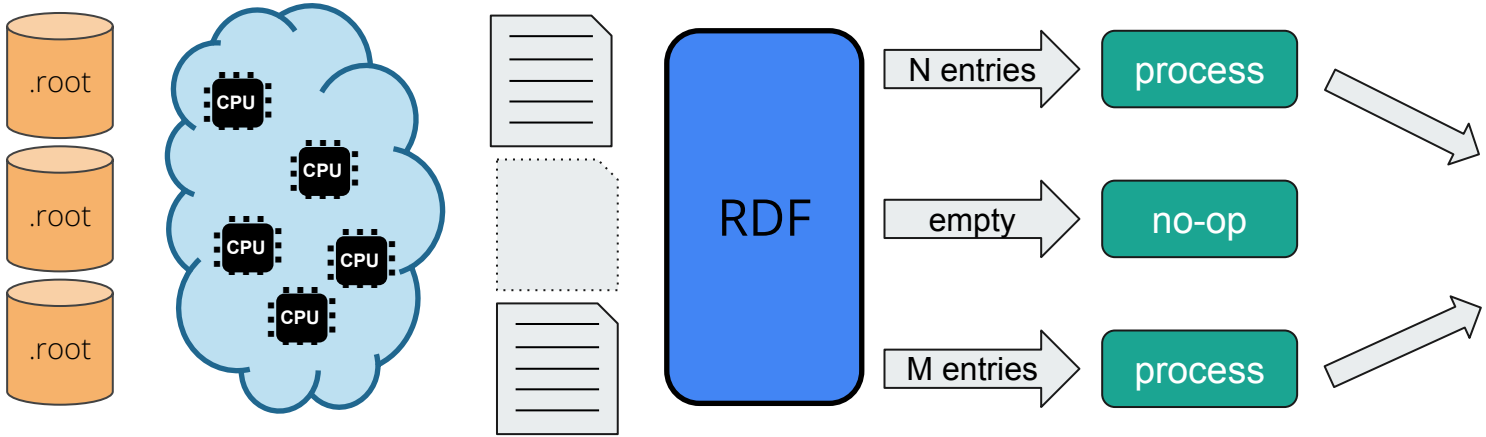


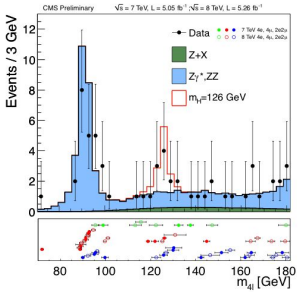

```python
from dask.distributed import Client
df = RDataFrame('treename', 'filename.root',
     daskclient = Client('tcp://hostname:port'))
```

```python
from pyspark import SparkContext
df = RDataFrame('treename', 'filename.root',
     sparkcontext = SparkContext('spark://IP:PORT'))
```

New: dealing with empty files in the pipeline

- Typical data pipelines go through a skimming step, before the final analysis starts
- This in practice can lead to irregular datasets, where some files have no entries at all
- A solid distributed engine must be able to deal with this situation gracefully
- In distributed RDataFrame, if a certain task has no entries to process it automatically becomes a no-op





New: bringing support for more operations

More operations of the RDataFrame API are now supported in distributed mode too:

- **HistoND**
- **DefinePerSample**
- **Redefine**
- **Vary + VariationsFor**
- **RunGraphs**

As per usual, the application code doesn't change at all

New: executing many distributed RDF graphs

Support for distributed RunGraphs means that it is possible to launch multiple RDF computation graphs to the same (or potentially even different) cluster resources, in parallel:

```
from dask.distributed import Client
df1 = RDataFrame('tree1', 'file1.root', daskclient = Client('tcp://hostname:port'))
df2 = RDataFrame('tree1', 'file1.root', daskclient = Client('tcp://hostname:port'))
# df1 and df2 need to be processed with different operations
p1 = df1.Filter(...).Filter(...).Define(...).Histo1D()
p2 = df2.Define(...).Define(...).Define(...).Histo3D()
# submit the graphs of p1, p2 concurrently
# (they will run independently with different processes)
ROOT.RDF.Experimental.Distributed.RunGraphs([p1, p2])
```

New: distributed systematic variations

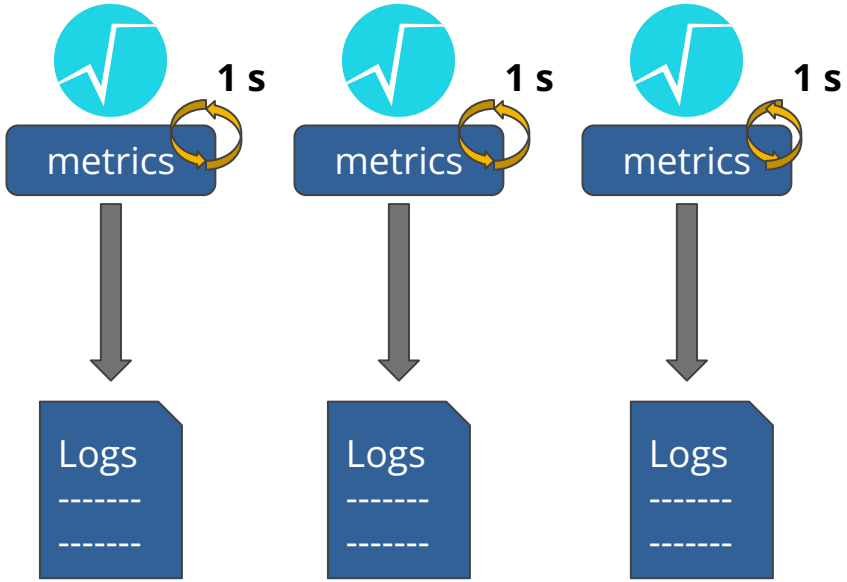
Support for distributed systematic variations:

```
nominal_hx = df.Vary("pt", "ROOT::RVecD{pt*0.9, pt*1.1}", ["down", "up"])
               .Filter("pt > k")
               .Define("x", someFunc, ["pt"])
               .Histo1D("x")
hx = ROOT.RDF.Experimental.Distributed.VariationsFor(nominal_hx)
hx["nominal"].Draw()
hx["pt:down"].Draw("SAME")
```

New: monitoring distributed RDF processes

For the purposes of this study, we enabled monitoring in the distributed processes

- Separate process pulls the **cpu/network/memory** data from the OS
- **Take** a new measurement **every second**
- Metrics **logs** are processed after the analysis



The infrastructure

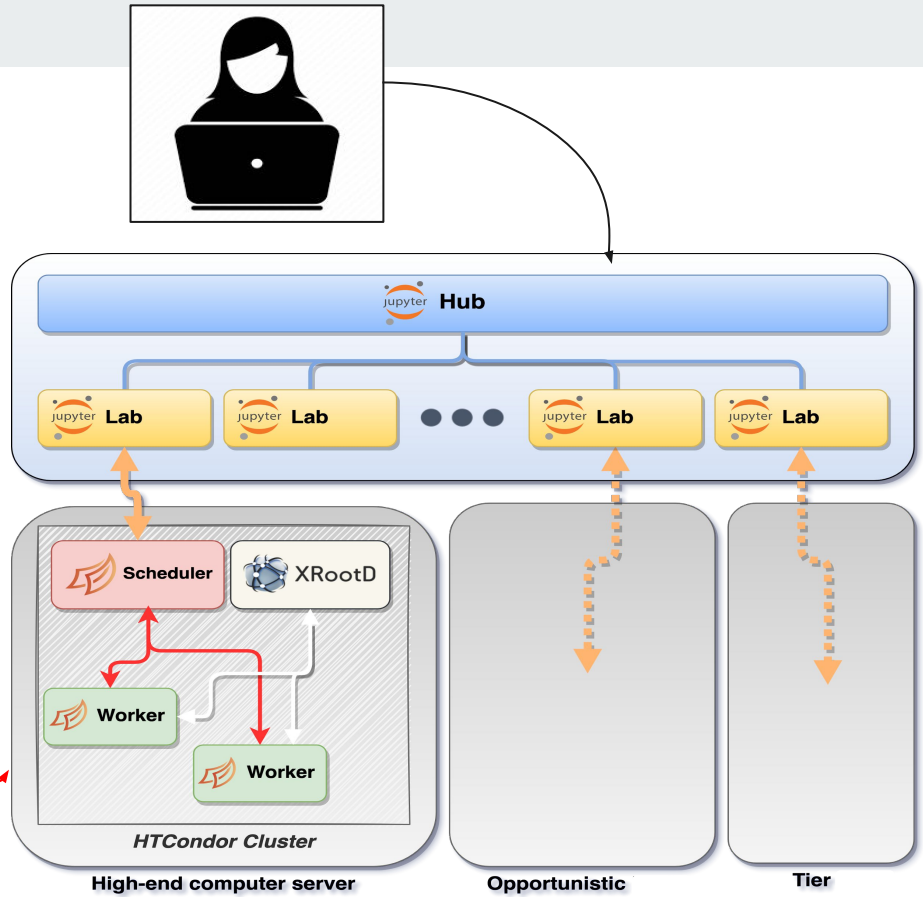
The analysis facility

Three building blocks:

- **JupyterHub (JHub)** and **JupyterLab (JLab)** to manage the user-facing part of the infrastructure:
  - This is not exclusive, also accessible via standard UI (“a la batch”)
- **DASK** to introduce the scaling over a batch system (**HTCondor**)
- **XRootD** as data access protocol toward AAA

We (CMS/INFN) have a **distributed and pledged resources** topology

- Integrate everything, possibly even opportunistic and “private” clusters
- With this configuration, “Grid vs Cloud vs HPC is not anymore a user issue“: everything is hidden behind a single Hub



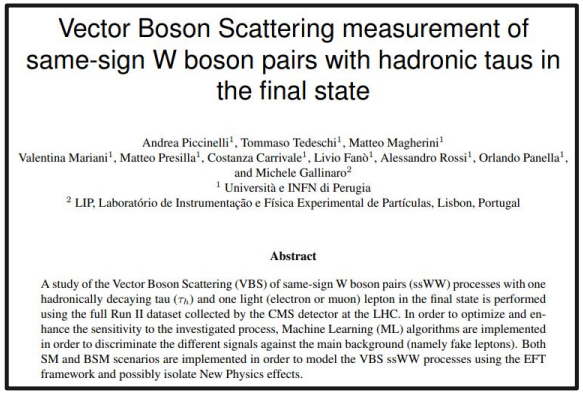
A cluster/ a single fat node / a cluster of fat nodes…

The use case

Interactive via distRDF: first use case

A VBS SSWW analysis based on NanoAOD inputs (~plain ROOT files) **has been ported** from legacy approach (nanoAOD-tools/plain PyROOT-based) **to distributed RDataFrame** in order to obtain:

- Enhanced **user experience** thanks to the declarative interface
- Improved **efficiency** thanks to intrinsic parallelization
- **Optimized operations on data**
  - obtained by merging analysis steps
- **Distribution of workflows** on different back-ends with ~0 changes in code base



Vector Boson Scattering measurement of same-sign W boson pairs with hadronic taus in the final state

Andrea Piccinelli[1], Tommaso Tedeschi[1], Matteo Magherini[1]
Valentina Mariani[1], Matteo Presilla[1], Costanza Carrivale[1], Livio Fanò[1], Alessandro Rossi[1], Orlando Panella[1], and Michele Gallinaro[2]
[1] Università e INFN di Perugia
[2] LIP, Laboratório de Instrumentação e Física Experimental de Partículas, Lisbon, Portugal

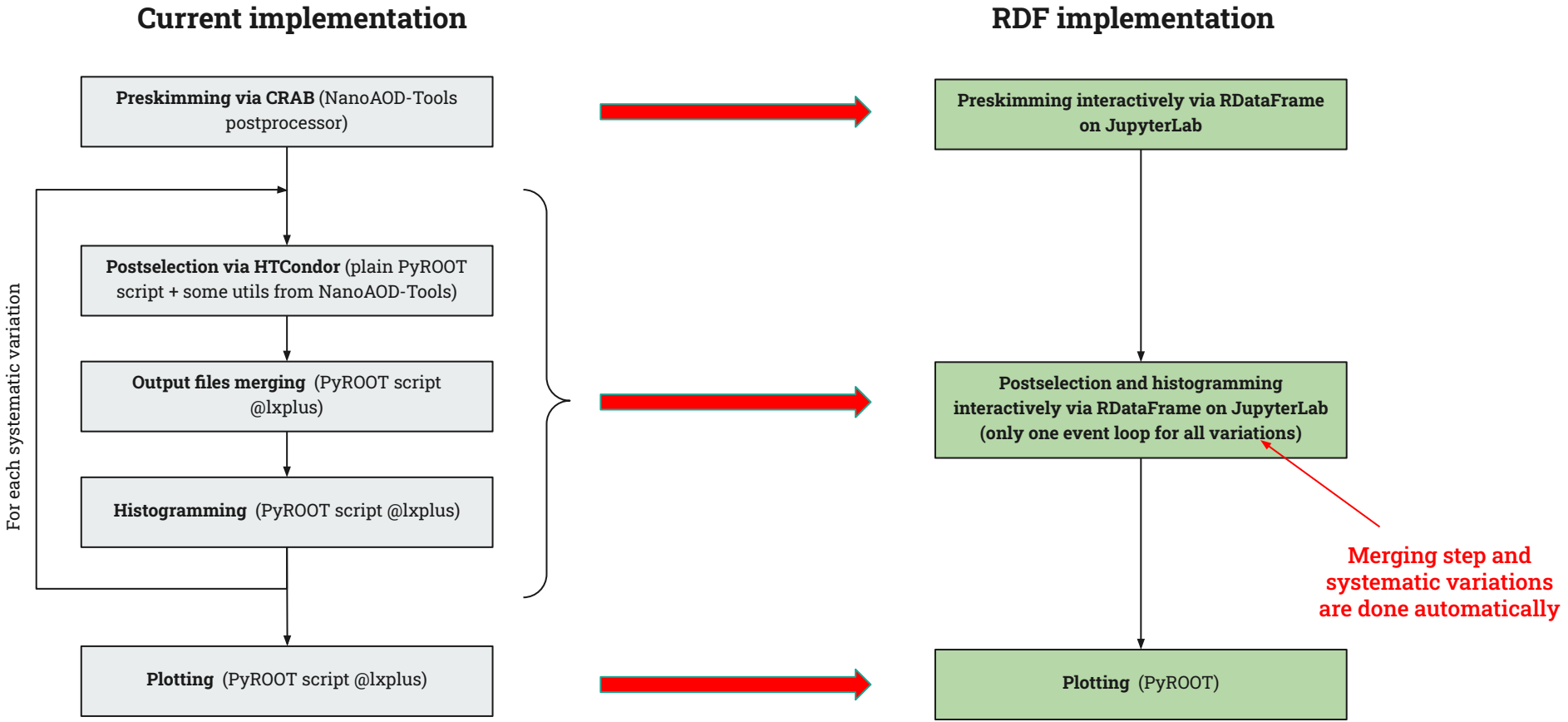
Abstract

A study of the Vector Boson Scattering (VBS) of same-sign W boson pairs (ssWW) processes with one hadronically decaying tau (τ_h) and one light (electron or muon) lepton in the final state is performed using the full Run II dataset collected by the CMS detector at the LHC. In order to optimize and enhance the sensitivity to the investigated process, Machine Learning (ML) algorithms are implemented in order to discriminate the different signals against the main background (namely fake leptons). Both SM and BSM scenarios are implemented in order to model the VBS ssWW processes using the EFT framework and possibly isolate New Physics effects.

VBS SSWW with a light lepton and an hadronic tau in final state on full Run2 (**NanoAOD**)

Continuous collaboration/interaction between INFN and ROOT teams for the **request/implementation/test of new specific distributed RDataFrame features needed to streamline a full-scale physics analysis** (see next slides) **to evaluate performance**

Legacy → distRDF migration



How the code looks like, in a nutshell

```
def inizialization_function():
    ROOT.gInterpreter.Declare('#include "utils_functions.h"')

df = ROOT.RDF.Experimental.Distributed.Dask.RDataFrame("Events", chain, nPartitions = N, client = client)  #define the dataframe

df_processed = df.Define("column_c", "function(column_a, column_b)")\
                 .Filter("filtering_function(column_d)", "A filter")
                 ...

# book a snapshot (i.e. a saving)-> used in preselection
opts = ROOT.RDF.RSnapshotOptions()
opts.fLazy = True
df_lazy_snapshot = df_processed.Snapshot("treeName", "fileName.root", opts)

# book an histogram -> used in postselection
lazy_histo = df_lazy_snapshot.Histo1D("column_c", "weights_column")

# to trigger execution
histo = lazy_histo.GetValue()

# to inspect data
df_saved.Display(["column_a", "column_b", "column_c"], nRows = 1).Print()
+-----+----------+----------+----------+
| Row | column_a | column_b | column_c |
+-----+----------+----------+----------+
| 0   | -1       | -1       | -1       |
+-----+----------+----------+----------+
```

How the code looks like, in a nutshell

```
def inizialization_function():
    ROOT.gInterpreter.Declare('#include "utils_functions.h"')

df = ROOT.RDF.Experimental.Distributed.Dask.RDataFrame("Events", chain, nPartitions = N, client = client)  #define the dataframe

df_processed = df.Define("column_c", "function(column_a, column_b)")\
                 .Filter("filtering_function(column_d)", "A filter")
                 ...

# book a snapshot (i.e. a saving)-> used in preselection
opts = ROOT.RDF.RSnapshotOptions()
opts.fLazy = True
df_lazy_snapshot = df_processed.Snapshot("treeName", "fileName.root", opts)

# book an histogram -> used in postselection
lazy_histo = df_lazy_snapshot.Histo1D("column_c", "weights_column")

# to trigger execution
histo = lazy_histo.GetValue()

# to inspect data
df_saved.Display(["column_a", "column_b", "column_c"], nRows = 1).Print()
+-----+----------+----------+----------+
| Row | column_a | column_b | column_c |
+-----+----------+----------+----------+
| 0   | -1       | -1       | -1       |
+-----+----------+----------+----------+
```

C++ functions that manipulate RVec objects
Target of the porting

How the code looks like, in a nutshell - with variations

```
def inizialization_function():
    ROOT.gInterpreter.Declare('#include "utils_functions.h"')

df = ROOT.RDF.Experimental.Distributed.Dask.RDataFrame("Events", chain, nPartitions = N, client = client)  #define the dataframe
```

```
df_processed = df.Vary("column_a", "...", "...")\
                 .Define("column_c", "function(column_a, column_b)")\
                 .Filter("filtering_function(column_d)", "A filter")
                 ...
```

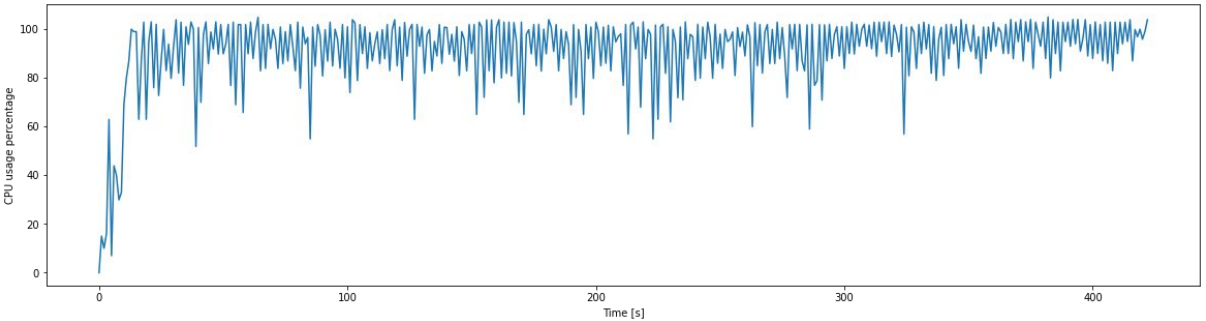
```
# book a snapshot (i.e. a saving)-> used in preselection
opts = ROOT.RDF.RSnapshotOptions()
opts.fLazy = True
df_lazy_snapshot = df_processed.Snapshot("treeName", "fileName.root", opts)

# book an histogram -> used in postselection
lazy_histo = df_lazy_snapshot.Histo1D("column_c", "weights_column")
```

```
lazy_histo_varied = ROOT.RDF.Experimental.Distributed.VariationsFor(lazy_histo)
```

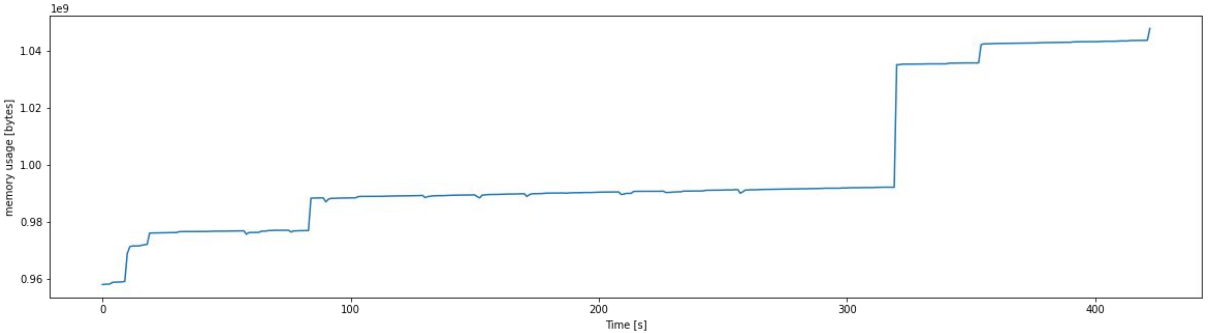
```
# to inspect data
df_saved.Display(["column_a", "column_b", "column_c"], nRows = 1).Print()
+-----+----------+----------+----------+
| Row | column_a | column_b | column_c |
+-----+----------+----------+----------+
| 0   | -1       | -1       | -1       |
+-----+----------+----------+----------+
```

distRDF monitoring

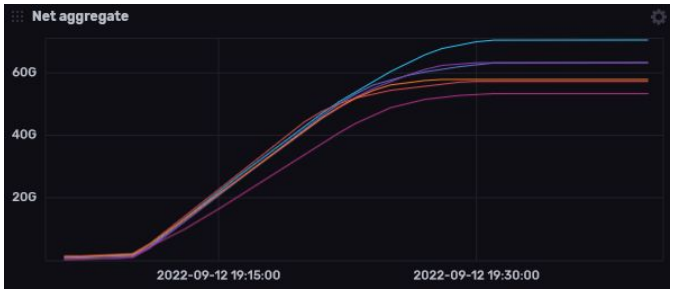
Task CPU usage percentage



Task memory usage

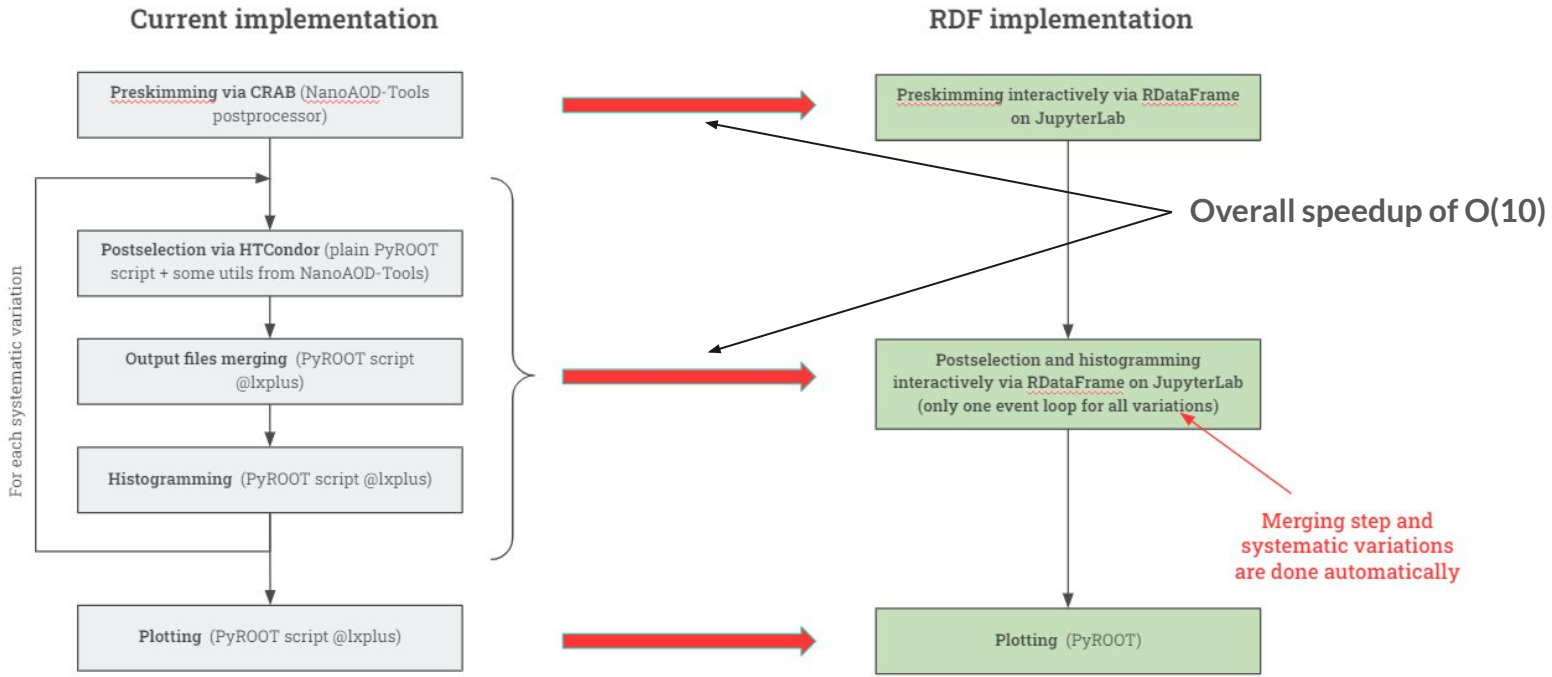


Network bytes read per worker node



Performance comparison

Taken as a benchmark the 2017 MC UltraLegacy (1 TB) analysis:



DEMO

https://github.com/comp-dev-cms-ita/pyHEP2022_distRDF_INFN_AF

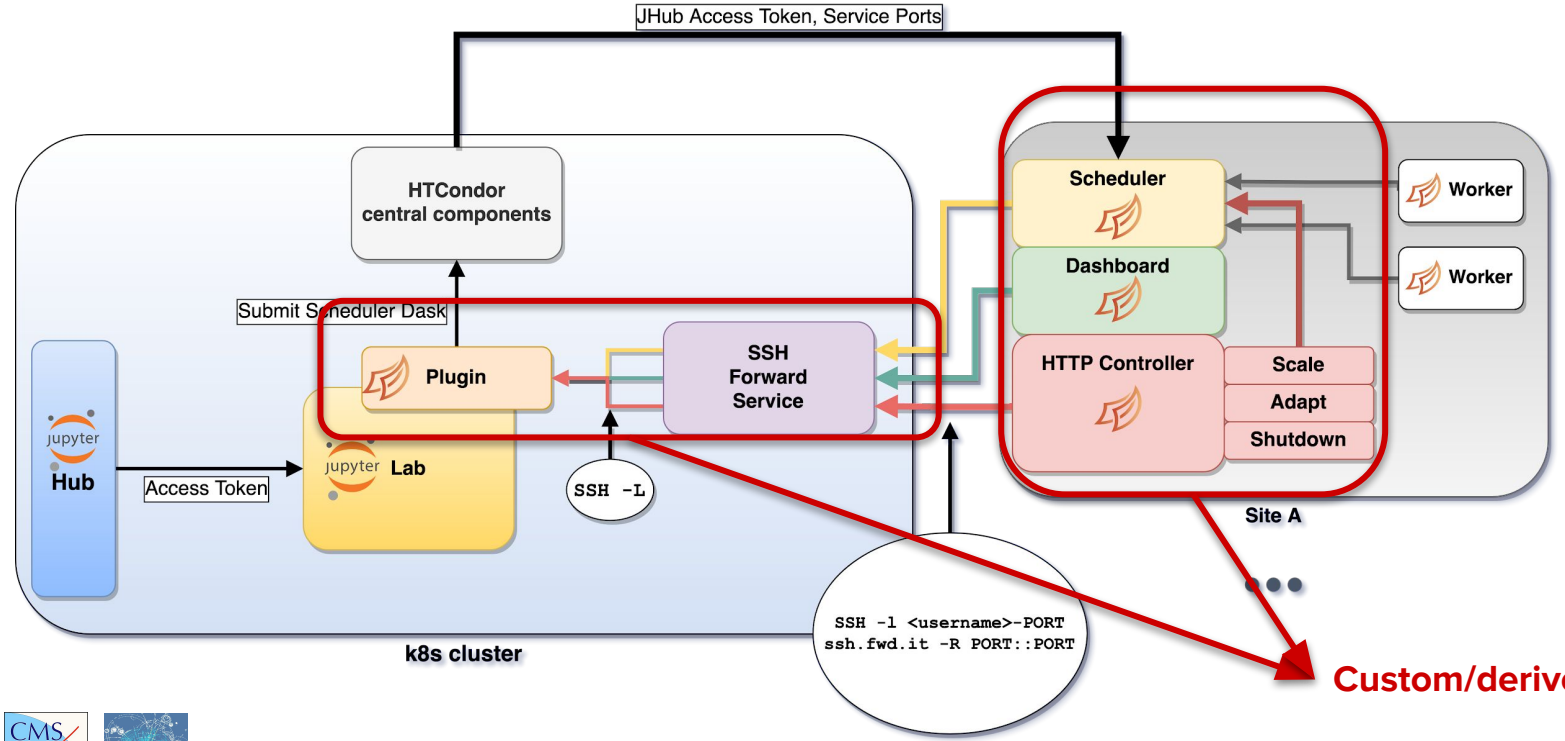
Conclusions

- Overall **the transition from a "legacy" analysis code looks reasonably easy**
  - At least for analyses based on NTuple-like data source
- The **capability to scale seamlessly** what I currently run locally is a great **added value**
  - Performance tests are also showing pretty good results
- Very **advanced features** that make analysis easy:
  - E.g. systematic variations
- Still to **improve on distributed logging and debugging procedure**
  - Sort of DASK/HTCondor native problem to be honest
  - occasional Dask task failure with no clear error pattern
  - Monitoring resources is crucial to understand possible bottleneck/errors
- Complete **DNN inference with an external model possible via onnx files** (and TMVA's SOFIE):
  - Tf2onnx necessary if starting from Tensorflow Models

BACKUP

Dask-remote-jobqueue implementation

From the network perspective **we came up with the following architecture deployed on the current testbed**
This would allow us to respect the DASK cluster locality needs, while running code on remote JupyterLAB

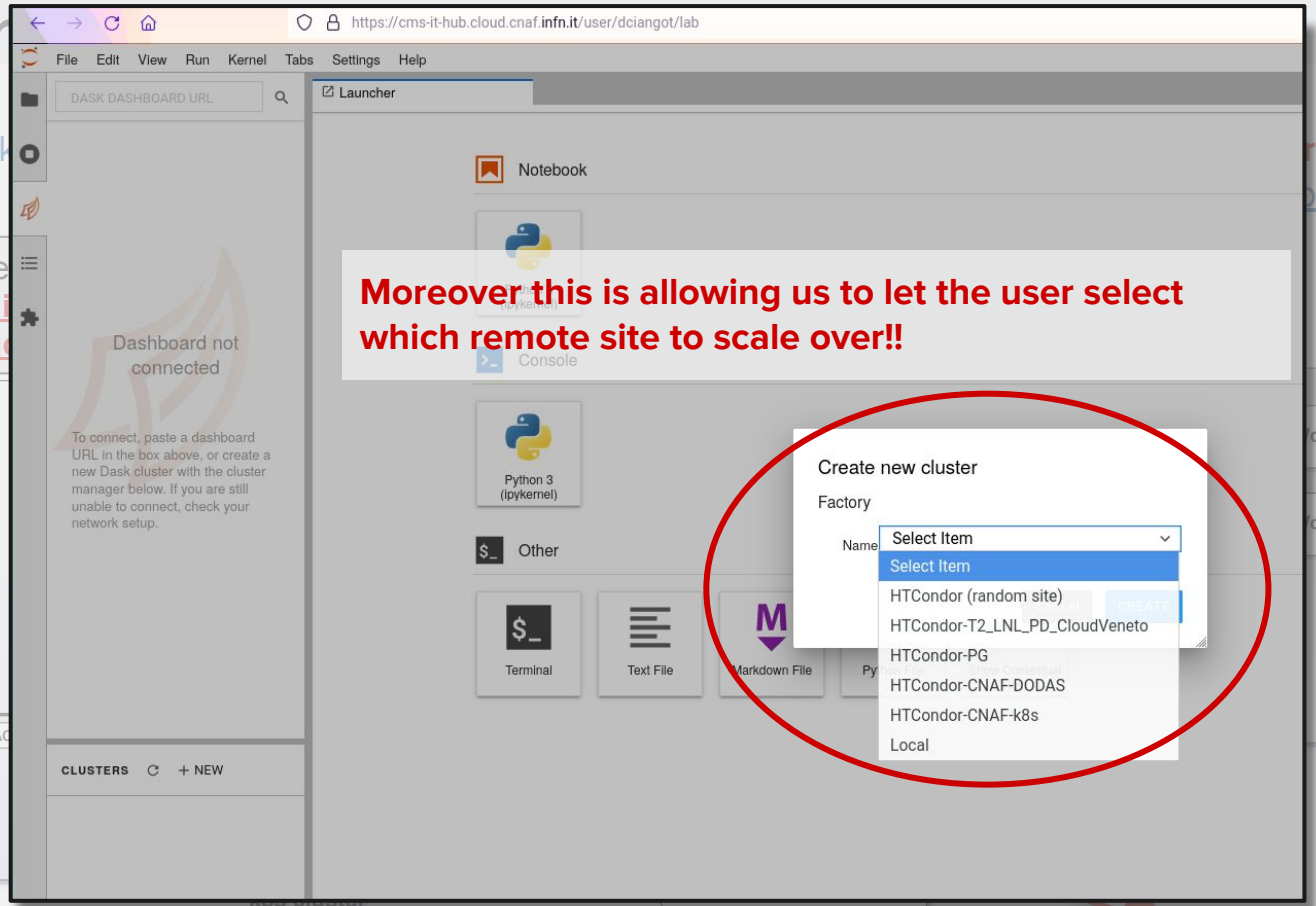


Dask-ren

From the network
This would allow

rent testbed
pyterLAB

More importantly we
the bar of the requi
new resources to j



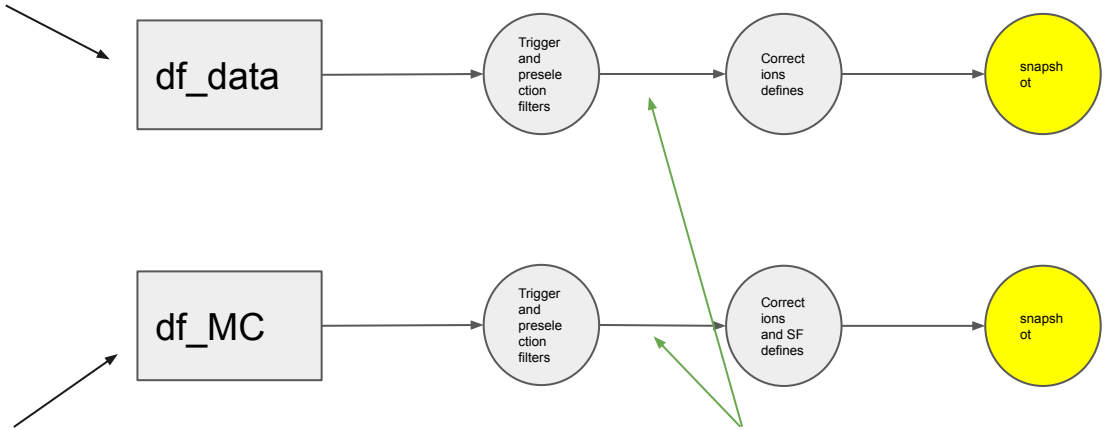
Moreover this is allowing us to let the user select which remote site to scale over!!

jupyter Hub

Custom/derived parts

Pre-selection.... in a graph

One for each era

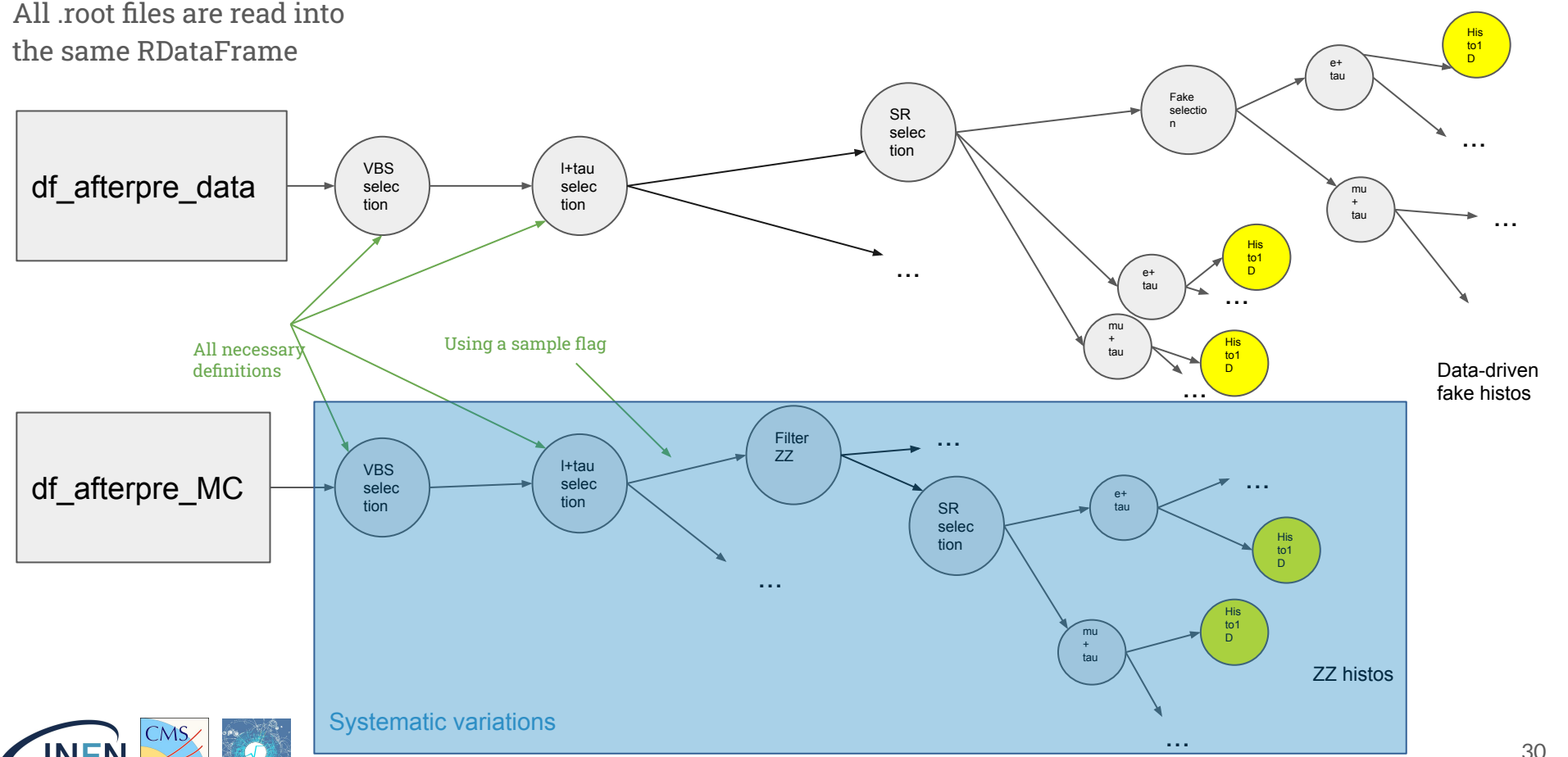


All .root files are read into the same RDataFrame

Adding a sample flag with `define_per_sample`

Post-selection…. in a graph

All .root files are read into the same RDataFrame



AuthN/Z: a “token native” system

We thought the system to be **token based**

- In other words: **from IAM @CMS → JupyterHUB/HTCondor/DASK reverse proxy**

Although a bit ahead of time, this comes with a significant benefit when we need to automatically define what resources can access who in a dynamic/modern fashion

So far in HTCondor we are using the “good old” mapfile to match user IAM id with condor user. This is in evolution though, **possibly toward capability based access** (in any case, **no changes would be needed to the infrastructure**)

```
apiVersion: v1
data:
  condormapfile: |
    SCITOKENS  https:\/\/dodas-iam.cloud.cnaf.infn.it\/,1e7074e5-96fe-43e8-881d-4d572c128931 dciangot
    SCITOKENS  https:\/\/dodas-iam.cloud.cnaf.infn.it\/,d0203717-30ad-407e-ab82-a48b93baed57 vpadulan
    SCITOKENS  https:\/\/cms-auth.web.cern.ch\/,51e0c369-345a-46b1-812f-61a4269cda8f  etejedor
    SCITOKENS  https:\/\/cms-auth.web.cern.ch\/,0c765e2d-993e-4ed2-abf8-9c54dcc34546  ttedesch
    SCITOKENS  https:\/\/cms-auth.web.cern.ch\/,4619b517-39d1-4b76-87df-69cbb15a0dee  mtracoll
    SCITOKENS  https:\/\/cms-auth.web.cern.ch\/,78f275d5-bb1a-4b2d-9956-f82316a8482e  spiga
    SCITOKENS  https:\/\/cms-auth.web.cern.ch\/,c4b22a94-6dda-4312-b77f-726813e963ae  dciangot
    PASSWORD (*.) condor
    GSI (.*) anonymous
```

What can I do in there?

- Batch/Legacy
  - Run your analysis on a batch system with resources collected over sites
  - Run your analysis on a batch system targeting specifically HPC resources
- Quasi-interactive python scripting
  - On-demand leverage big-notebooks on big machines (hpc node, dedicate hw) and run locally with a portable and ready-to-use environment
- Interactive python notebooks
  - Effortlessly scale local code with distributed mode RDataFrame workflows over a T2 site or over dedicated/specialized resources
  - Edit an reproduce plot interactively