RDataFrame: a flexible and scalable analysis experience

Vincenzo Eduardo Padulano[1,2], **Enrico Guiraud**[1], **Enric Tejedor Saavedra**[1], **Ivan Donchev Kabadzhov**[1,3] **and Pawan Johnson**[4]

[1]EP-SFT, CERN, Meyrin, 1211 Geneva, Switzerland.

[2]Department of Computation Systems and Computation, Universitat Politècnica de València, 46022 Valencia, Spain

[3]Department of Computer Science, Albert Ludwig University of Freiburg, 79098 Freiburg, Germany

[4]Indian Institute of Technology Kharagpur, 721302 Kharagpur, India

E-mail: vincenzo.eduardo.padulano@cern.ch, enrico.guiraud@cern.ch, enric.tejedor.saavedra@cern.ch, ivan.donchev.kabadzhov@cern.ch, pawan.p.johnson@gmail.com

Abstract. The growing amount of data generated by the LHC requires a shift in how HEP analysis tasks are approached. Usually, the workflow involves opening a dataset, selecting events, and computing relevant physics quantities to aggregate into histograms and summary statistics. The required processing power is often so high that the work needs to be distributed over multiple cores and multiple nodes. This contribution establishes ROOT RDataFrame as the single entry point for virtually all HEP data analysis use cases. In fact, the typical steps of an analysis workflow can be easily and flexibly written with RDataFrame. Data ingestion from multiple sources is streamlined through a single interface. Relevant metadata can be made available to the dataframe and used during analysis execution. A declarative API offers the most common operations to the users, while transparently taking care of data processing optimisations. For example, it is possible to inject user-defined code to compute complex quantities, gather them into histograms or other relevant statistics, include large sets of systematic variations and use machine-learning inference kernels. A Pythonic layer allows dynamic injection of Python functions in the main C++ event loop. Finally, any RDataFrame application can seamlessly scale out to hundreds of cores on the same machine or multiple distributed nodes by changing a single line of code.

1. Introduction

The Large Hadron Collider (LHC) at CERN is now in its third active period (also called "Run 3"). So far, it has already generated unprecedented volumes of data which require a complex pipeline to be properly stored and later processed. Nonetheless, it is already foreseen that the next major hardware upgrade (named HL-LHC [1]), due to start in 2029, will produce even more data than all the previous LHC runs combined. It has become more and more evident that LHC experiments and collaborations in the HEP community at large will need efficient software to handle the ever-increasing computing needs [2], also according to the future budget estimations [3].

Data generated by the collider is filtered and structured across multiple steps, which can vary in number and configurations depending on the specific LHC experiment. At some point, researchers will be able to access accelerator data, stored in a well-defined format. This final step of the HEP data lifecycle is characterised by a write-once, read-many scenario where the processing workflows vary widely depending on the experiment collaboration, institution, research group, down to the single researcher. As such, analysis software has seen many implementations in different frameworks and packages, differing in the execution logic, the user interface, the interaction with storage and distributed resources and naturally also in the configuration of the physics analysis involved.

Nonetheless, the whole field has shared a common building block for many steps of this data pipeline in the ROOT [4] software framework. The library defines a data format which features a columnar layout on disk and supports arbitrary user-defined objects, providing great customisation for physics events data models of experiments while at the same time minimising the I/O transactions needed. It also offers many facilities for data processing, statistical modeling, multivariate analysis and data visualisation. In particular, RDataFrame [5] is the suggested interface for HEP data analysis. This interface offers an API inspired by declarative programming principles, so users only need to think about the physics involved in their code, requesting the operations that should be run on the dataset. RDataFrame then takes care of expressing user logic in terms of a computation graph and executing it, while at the same time optimising the execution as much as possible under the hood.

This contribution shows how RDataFrame can be practically used by physicists to fulfill all the needs of their data analysis applications, from highly interactive simple analysis prototypes to large-scale, production executions on very large datasets and multiple nodes concurrently. The paper describes briefly the ingredients of a typical HEP data analysis application (Section 2), then proceeds to describe the implementation of the different parts in a composable and scalable manner thanks to RDataFrame (Section 3).

2. The physics analysis workflow

As stated in Section 1, the data analysis landscape in HEP is extremely heterogeneous and thus different applications may feature very different components and logic. Nonetheless, an attempt at defining the most high-level ingredients that are related to any analysis can be made. Figure 1 depicts such ingredients, represented in the order they usually appear (or are included) in the analysis code from left to right.

Some of these ingredients are strictly necessary or practically always present. Obviously, the "input samples" (i.e. the input dataset) are the starting point of any analysis. The word "sample" generally represents a part of the whole dataset and belongs to the jargon used in the HEP context: different samples may differ in the metadata that accompany them (for example different calibration or luminosity values) but they are processed with the same type of operations. A minimum amount of these operations is usually present in the form of filters (also called "cuts") to remove uninteresting events and creation of quantities derived from the input columns (also called "observables").

Other elements are not always present, but become increasingly important for more complex use cases. For example, observables may need to be weighted when performing some aggregation (usually histograms). Systematic variations need to be included e.g. to study the effect of measurement uncertainties on the analysis output. Thus, the same logic needs to be applied on slightly different versions of the same input columns (a process that traditionally required a lot of manual bookkeeping and usually involved a tangible runtime penalty).

Finally, the aggregated quantities are visualised, most often through histograms to represent the distribution of physical properties of interest for the analysis.

Although not directly belonging to this core set, there are also examples from many applications that include statistical modeling and machine learning approaches in the workflow.

All together, these ingredients concur in making every analysis a potentially very complex,

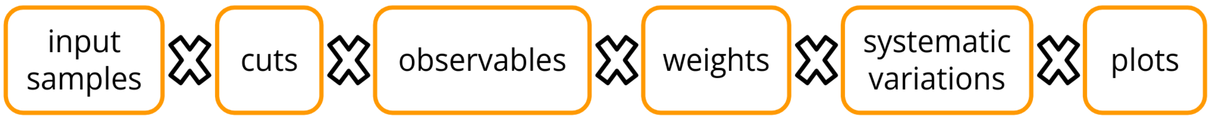

Figure 1. The elements that concur in the creation of a HEP data analysis application.

multi-dimensional workflow, with some dimensions that are optional, others that can get increasingly larger the more data, observables and systematic variations need to be taken into account. The more simply analysis software tools can express this workflow, the easier the task for the analyst.

3. The RDataFrame analysis experience

The various components highlighted in Section 2 find a common ground in the RDataFrame API, which can accompany analysts in their journey from collision data to new physics insights. Any analysis begins with an input dataset, which in practical cases is made of multiple different files.

```
{
  "sample_a": {
    "files": ["a.root","b.root"],
    "metadata": {"w":1., "xsec":21.3, "year":2017}
  },
  "sample_b": {
    "files": ["fb*.root"],
    "metadata": {"w":0.5, "type":"MC", "year":2018}
  }
}
```

Listing 1. A simple specification of a physics dataset.

The input files usually contain data in the TTree format (the traditional implementation of the ROOT data format), but RDataFrame also supports the next-generation ROOT data format RNTuple [6] and many others (e.g. Apache Arrow, CSV files, Numpy arrays). Furthermore, datasets very often are stored remotely in dedicated facilities due to their very large size and the need to be accessible by all the people involved in the collaboration. RDataFrame can seamlessly support remote reading and writing: supplying a remote path as input won't change anything with respect to the rest of the code logic. The specification of the input dataset can be defined in a semi-structured way akin to Listing 1 - including the input files and the division in samples with metadata - and passed to RDataFrame with the factory function `FromSpec`.

The exploration phase of the analysis can begin by calling `Describe` or `Display` on the RDataFrame, which respectively tell summary information about the input data and show the values stored in the columns. Then, the basic needs of the application are usually solved by a mix of `Filter` (for cuts) and `Define` (for the computation of derived observables) calls. Such parts of the API accept any type of user-defined callable.

An integral part of most analyses is dealing with columns that contain collections, which in the HEP case can vary in size among different rows since events may involve a different number

```python
df = ROOT.RDF.FromSpec("myspec.json")
histo = df.Define(
    "good_pt", "sqrt(px*px + py*py)[E>100]"
).Histo1D("good_pt")
histo.Draw()
```

Listing 2. RDataFrame code that fills and plots a histogram with the transverse momentum of the particles in the dataset if the energy is greater than 100 MeV.

of particles. Listing 2 shows how these first steps of an analysis can be constructed with just a few, very simple lines of code.

A physics analysis can then proceed by adding more observables and more precise filtering if needed. As the complexity starts to increase, other operations from the RDataFrame API may come in handy. For example, a call to the `Redefine` function allows to modify in-place the values of a column of the dataset, which may need to be reweighted or modified because they derived from an external pipeline that doesn't completely fit in the current application. The previous values of the column may be used when computing the new values, for example with:

```python
df.Redefine("electron_mass", "electron_mass * correction_value")
```

Another interesting example is dealing with the different samples, which can be done via the `DefinePerSample` function. Through it, users can adapt their logic according to the sample the event being processed belongs to (see Listing 3). The column `rdfsampleinfo_` is automatically created by RDataFrame and available in the call, where it can be used to retrieve the same metadata defined in the specification.

```python
df = ROOT.RDF.FromSpec("myspec.json")
df.DefinePerSample(
    "weight", 'rdfsampleinfo_.GetD("w")'
).Histo1D("pt","weight")
```

Listing 3. RDataFrame code to define the weight for the transverse momentum of particles in the dataset, depending on whether they come from simulation or from real events.

A turning point in the complexity of the analysis is usually reached when systematic variations start to be included in the logic. From the computational standpoint, this involves repeating the execution of the analysis on the same types of input quantities, but with varied values to represent the uncertainty of the detector measurements. It used to be the case that users not only had to deal with the complex considerations involved in the physics logic, but also with an amount of manual bookkeeping increasing linearly with the number of variations. RDataFrame allows to abstract away from all that bookkeeping with the pair of functions `Vary` and `VariationsFor` as seen in Listing 4. The values of the column "pt" which was already found in the dataset are also called "nominal" values. The variations are declared to RDataFrame at Line 3 with `Vary`, which creates two different "universes" of values labeled "down" and "up" by multiplying the nominal values respectively by 0.9 and 1.1. Following calls to the API will keep track of the varied universes in the computation graph. At Line 8, the `VariationsFor` function is used to request the histogram for the nominal values together with the histograms for all the variations. The user gets an associative container where the varied results can be gathered by the label that was supplied in the previous `Vary` call.

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

Listing 4. RDataFrame code to vary the values of column “pt” and consecutively apply some operations separately to the nominal column and the varied columns.

The `Vary` operation is quite flexible and allows for many more use cases. The simplest one is when multiple columns need to be varied separately, which can be satisfied by as many calls to `Vary`, chained together one after another. Another more interesting and typical case within HEP is when two or more observables need to vary at the same time, i.e. the varied universes contain more than one varied observable. To this end, `Vary` accepts as first argument a list of columns that can be varied together, as exemplified in Listing 5.

```python
nominal_hx = df.Vary(
    ["Jet_pt","Jet_mass","MET_pt","MET_phi"],
    getJetMETVariations,
    variationTags=["down", "up"],
    variationName="JetMET"
).Filter("Jet_pt > 30").Histo1D("nJet")
```

Listing 5. RDataFrame code to vary multiple columns in lockstep, as is commonplace in jets and MET variations.

Realistic HEP analyses make use of all the operations discussed so far, combined to get hundreds or even thousands of results from processing very large input datasets. In order to bring down the total runtime and make better usage of available hardware resources, RDataFrame is natively capable of parallelising the execution of the analysis, on a single machine as well as on multiple nodes of a computing cluster. In the first case, it is simply enough to let ROOT know that the application should be parallelised using C++ multi-threading with a single line of code at the beginning of the program:

```python
ROOT.EnableImplicitMT()
```

In the latter case, a Python layer enables the automatic splitting of the computation in tasks that are submitted to the remote cluster through some connection handle (currently the Dask or Spark libraries and related clusters are supported for distributed execution [7]). In any case, the API and the analysis logic stay exactly the same.

Finally, although all the example snippets in this paper were showing the ROOT Python bindings, it can be noted that the user-provided functions passed to the API were written as Python strings containing valid C++ code. This behaviour has been improved so that users can pass Python callables to the RDataFrame API directly, which are just-in-time (JIT) compiled via the Numba Python library [8] so that the C++ execution engine can run them. Listing 6 shows some usage examples of this feature, which is still under development to satisfy the vast amount of cases supported by RDataFrame.

```
df.Filter(lambda muon_eta: muon_eta < k)
df.Define("col_c", lambda col_a, col_b: col_a * col_b)
```

Listing 6. Passing Python callables within the RDataFrame API.

4. Conclusions

RDataFrame is a swiss-army-knife tool for HEP data analysis. Its simple yet powerful API relies on a declarative programming model, aimed at removing the burden of thinking about the computational logic from the physicists, and on a lazy execution pattern, which allows for automatic and implicit optimisations under the hood. Also thanks to the composability of its features, this API provides an ergonomic and scalable analysis experience throughout the various steps highlighted in Section 2. As such, it has already seen wide usage in the community as both a building block for larger frameworks and a direct tool for analysis applications [9, 10, 11]. Furthermore, its performance and scalable use of resources has been demonstrated in different occasions [12, 13, 14]. This tool will continue evolving together with the physics community towards future computing challenges.

References

[1] Apollinari G, Béjar Alonso I, Brüning O, Fessia P, Lamont M, Rossi L and Tavian L 2017 High-Luminosity Large Hadron Collider (HL-LHC). Technical Design Report V. 0.1 Tech. rep. Fermi National Accelerator Lab. (FNAL), Batavia, IL (United States) URL `https://www.osti.gov/biblio/1767028`

[2] Piparo D 2022 *21st International Workshop on Advanced Computing and Analysis Techniques in Physics Research* URL `https://indico.cern.ch/event/1106990/contributions/5021254/`

[3] Elsen E 2019 *Comput Softw Big Sci* **16**

[4] Brun R and Rademakers F 1997 *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* **389** 81–86 ISSN 0168-9002 New Computing Techniques in Physics Research V

[5] Piparo D, Canal P, Guiraud E, Valls Pla X, Ganis G, Amadio G, Naumann A and Tejedor Saavedra E 2019 *EPJ Web Conf.* **214** 06029

[6] Blomer J, Canal P, Naumann A and Piparo D 2020 *24th International Conference on Computing in High Energy and Nuclear Physics (CHEP 2019)* vol 245

[7] Padulano, Vincenzo Eduardo, Cervantes Villanueva, Javier, Guiraud, Enrico and Tejedor Saavedra, Enric 2020 *EPJ Web Conf.* **245** 03009 URL `https://doi.org/10.1051/epjconf/202024503009`

[8] Lam S K, Pitrou A and Seibert S 2015 *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* LLVM '15 (New York, NY, USA: Association for Computing Machinery) ISBN 9781450340052 URL `https://doi.org/10.1145/2833157.2833162`

[9] KIT-CMS CROWN URL `https://github.com/KIT-CMS/CROWN`

[10] Manca E 2021 *Precision measurements of W detected at CMS* Ph.D. thesis Scuola Normale Superiore, Pisa URL `https://cds.cern.ch/record/2800948`

[11] David P 2021 *EPJ Web Conf.* **251** 03052 URL `https://doi.org/10.1051/epjconf/202125103052`

[12] Graur D, Müller I, Proffitt M, Fourny G, Watts G T and Alonso G 2021 *Proc. VLDB Endow.* **15** 154–168 ISSN 2150-8097 URL `https://doi.org/10.14778/3489496.3489498`

[13] Padulano V E, Tejedor Saavedra E, Alonso-Jordá P, López Gómez J and Blomer J 2022 *Cluster Computing* ISSN 1573-7543 URL `https://doi.org/10.1007/s10586-022-03757-2`

[14] Padulano V E, Kabadzhov I D, Tejedor Saavedra E, Guiraud E and Alonso-Jordá P 2023 *Journal of Grid Computing* **21** 9 ISSN 1572-9184 URL `https://doi.org/10.1007/s10723-023-09645-2`