RDataFrame enhancements for HEP analyses

E Guiraud[1], J Blomer[1], S Hageboeck[2], A Naumann[1], V E Padulano[1], E Tejedor[1], S Wunsch[1]

[1]ROOT team, EP-SFT, CERN
[2]IT-SC-RD, CERN

E-mail: enrico.guiraud@cern.ch

Abstract. In recent years, RDataFrame, ROOT's high-level interface for data analysis and processing, has seen widespread adoption on the part of HEP physicists. Much of this success is due to RDataFrame's ergonomic programming model that enables the implementation of common analysis tasks more easily than previous APIs, without compromising on application performance. Nonetheless, RDataFrame's interfaces have been further improved by the recent addition of several major HEP-oriented features: in this contribution we will introduce for instance a dedicated syntax to define systematic variations, per-data-sample call-backs useful to define quantities that vary on a per-sample basis, simplifications of collection operations and the injection of just-in-time-compiled Python functions in the optimized C++ event loop.

1. Introduction

RDataFrame ([1], [2]) is an efficient and ergonomic interface for HEP analysis tasks, in C++ and Python. Thanks to its API inspired by declarative programming principles (users state what results they want to obtain from a dataset, the system decides how computations are scheduled), the same high-level programming model covers a large variety of use cases: from the production of few histograms to thousands, from quick data exploration performed on a laptop to realistic analysis applications on many-core machines ([3]), to distributed execution on a cluster ([4]). Listing 1 presents a simple example usage.

Since its introduction in ROOT ([5]) in version 6.14, RDataFrame has seen widespread usage, and with that usage came valuable user feedback. Physicists are applying RDataFrame to more and more complex use cases, they employ it as the foundation for more specialized frameworks (e.g. [6], [7]) as well as dataset-to-dataset transformation tools, and require seamless integration of RDataFrame data processing with Python machine learning frameworks and other tools from the Python data science ecosystem. In time, RDataFrame also became a user-friendly point of entry to modern ROOT features (see Fig. 1). This was not a role that was initially foreseen for this interface, and it provides additional requirements that are influencing its evolution.

This work discusses several novel RDataFrame features introduced in ROOT v6.26 that address the most common user requirements emerging from recent feedback. Most notably, a dedicated syntax to express systematic variations is introduced in Sec. 4: it enables physicists to express systematics in an ergonomic fashion, without the burden of the related book-keeping, while fitting naturally with the rest of the programming model.

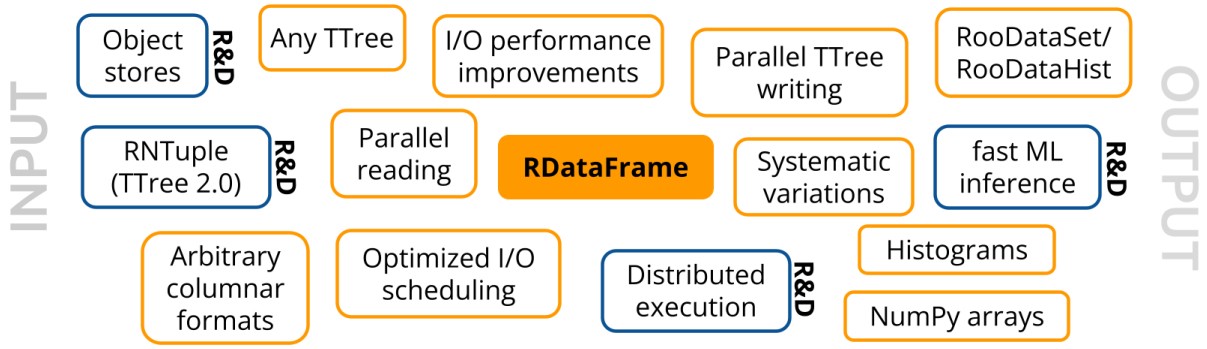

Figure 1. RDataFrame acts as a natural high-level entry point for many ROOT features, including recent developments such as RNTuple ([8]) and SOFIE (a fast machine learning inference engine, [9]).

```
ROOT.EnableImplicitMT() # enable multi-threading in ROOT
df = ROOT.RDataFrame(treeName="Events", filesList)
df = df.Filter("pts[abs(eta) < 1].size() > 0")
       .Define("myVec", myFunctor(), ["pts"])
h = df.Histo1D("myVec")
# write events that pass the Filter to a new tree (including column "myVec")
df.Snapshot("newtree", "newfile.root")
```

Listing 1: These few lines of Python code produce a skimmed dataset with an additional derived quantity and a control plot in a single multi-thread event loop.

2. Definition of per-sample values

HEP analyses must often treat data pertaining to different data taking periods slightly differently. Similarly, Monte Carlo samples might require different event weights than actual data. At the same time, in order to reduce the size of the dataset, the metadata useful to distinguish different samples is typically not part of the dataset itself, e.g. it might be stored in a small, separate dataset distributed together with the main one.

One way to address these scenarios with RDataFrame is to build different computation graphs for different samples: these graphs will look exceedingly similar, except for the nodes that need to deal with these differences in treatment of different sample types. `DefinePerSample` solves this problem more elegantly, enabling users to process different samples slightly differently in the same computation graph. Just like a `Define`, this transformation creates a new logical column holding the result of a user-defined callable invocation, with two important differences: firstly, the input to the callable are not other dataset columns, but rather metadata information regarding the sample being processed (e.g. the name of the tree, the file, the range of entries being processed in the form of a `RSampleInfo` object); secondly, the callable is not invoked at every event, but only once at the beginning of the processing of a new sample (e.g. when a single-thread event loop switches files or when a new multi-thread task starts executing). This second property makes `DefinePerSample` a useful tool also for injecting callbacks that can execute less frequently than at every entry and that need information on the data being processed: for example this is an appropriate hook for the injection of a progress bar display callback. Listing 2 shows an example invocation of `DefinePerSample`.

```
df.DefinePerSample("weight", [](unsigned slot, const RDF::RSampleInfo &s) {
  return s.Contains("MC") ? 0.5 : 1.; })
  .Histo1D("value","weight");
```

Listing 2: This C++ code snippet defines different weights for Monte Carlo and data samples.

3. Column redefinition

RDataFrame was initially designed as a high-level interface to select events and perform data reductions, but it soon became obvious that the programming model was flexible enough to extend to other use cases, in particular dataset-to-dataset transformations such as skims or dataset augmentations (e.g. addition of a derived column). In this context users might need to apply corrections to some column values before writing out a modified dataset, but RDataFrame made it unnecessarily cumbersome by completely disallowing column values from being overwritten: a `Define` always creates a new column, with a different name.

The recently introduced `Redefine` method makes it possible to modify the value and/or type of a column before further processing: this makes it easy to replace a single column in a large dataset, perform quick tests for the response of a selection to the variation of variables, or apply corrections to certain quantities, e.g. in specialized RDataFrame-based frameworks, before the dataframe object is further passed down to users. Debug printouts can also be easily injected by redefining the column as itself, with an identity function that performs a printout as a side-effect. Listing 3 provides a usage example.

```
df.Redefine("x", [](double x) { return float(x); }, {"x"})
  .Snapshot("tree", "newfile.root")
```

Listing 3: An example invocation of `Redefine` in C++ that reduces the precision of column "x" and writes out a modified dataset.

4. Systematic variations

RDataFrame's goal is to accompany HEP physicists from the stage of quick data exploration to a full-blown, large-scale analysis performing complex computations and requiring a large amount of computing resources. RDataFrame should make it possible to organically grow analysis code step by step, introducing nuances and complexities incrementally, without large code refactorings or changes of paradigm.

It is common for HEP analyses to eventually include the study of systematic variations. Handling systematic variations in RDataFrame used to require a certain increase in the analysis code complexity, mainly due to the tedious manual book-keeping required to keep track of the different results belonging to different systematics.

From the standpoint of a HEP physicist, the study of systematic variations involves many different, often conceptually complex cases. From the standpoint of the pure numerical computation, however, what typically happens is that the application must produce multiple results instead of a single one, each computed in a "universe" in which certain inputs take modified values. Our challenge is therefore to a) offer a user-friendly user API that nevertheless allows users to express most or all use cases for systematic variations present in HEP, and b) transparently propagate the variations through the RDataFrame computation graph (including event/object selections and the computation of derived quantities) in order to produce the varied results. In doing so, it is highly desirable to avoid replicating the whole computation graph, with its computations and the related I/O operations, for each separate "universe".

Starting from ROOT v6.26, RDataFrame provides a flexible syntax to define systematic variations that aims to satisfy these requirements. Listing 4 showcases the feature, which involves two new function calls: `Vary` lets users register varied values for one or more existing columns (e.g. up/down variations for "pt" in the example), while `VariationsFor` transforms the single "nominal" result into a dictionary-like object that contains results for each of the variations. Note that all other RDataFrame code remains the same as for the nominal case, and the presence of systematic variations is transparently propagated through `Filter`, `Define` and other calls.

```
auto nominal_hx =
   df.Vary("pt", "ROOT::RVecD{pt*0.9, pt*1.1}", {"down", "up"})
     .Filter("pt > k")
     .Define("x", someFunc, {"pt"})
     .Histo1D<float>("x");

// request the generation of varied results from the nominal
RResultMap<TH1D> hx = ROOT::RDF::VariationsFor(nominal_hx)
hx["nominal"].Draw()
hx["pt:down"].Draw("SAME")
hx["pt:up"].Draw("SAME")
```

Listing 4: An example usage of `Vary` and `VariationsFor`, in C++.

Some aspects of this interface will be improved with the help of a first round of user feedback; in particular, we expect to be able to further simplify the definition of simultaneous variations so that the explicit construction of nested arrays can be avoided in the most common cases.

4.1. Performance measurements

A dedicated syntax to express systematic variations not only simplifies user code, but by giving RDataFrame semantic information about user intention it allows certain optimizations in the construction of its computation graph. This results in less overhead during the event loop, as we can see from Fig. 2. The benchmark did not involve any disk I/O (entries were generated on the fly by RDataFrame, and consisted of a single floating point number) and it consisted in filling one or more histograms for the nominal case and multiple variations. We compare the performance of this task without using `Vary` (which requires defining varied columns with `Define` and booking the corresponding histograms in a for loop) and with `Vary`. The new syntax offers a significant performance boost for the heavier use cases (B. and C. in the figure) and it results in a small penalty when only one histogram with an up/down variation is filled.

5. Injecting Python functions into a C++ event loop via Numba

Python is increasingly relevant for the HEP community, and particularly so in the area of analysis. RDataFrame, like all of ROOT, offers dynamically generated Python bindings via PyROOT ([10]), while its internal event loop remains in C++: this offers greater opportunities for performance optimizations and allows multi-thread parallelism that would be more complicated in Python due to its Global Interpreter Lock (the *GIL*). At the same time, the Python ecosystem has developed tools to generate efficient machine code from Python code to speed up numerical applications; the most interesting such tool for our purposes is Numba ([11]), which is able to compile Python functions (that restrict themselves to use a specific subset of the language) to optimized machine code. Numba can transform Python functions in compiled functions accessible as C function pointers that can be made known to ROOT's C++ interpreter, Cling. The result is that RDataFrame can call these compiled Python functions from

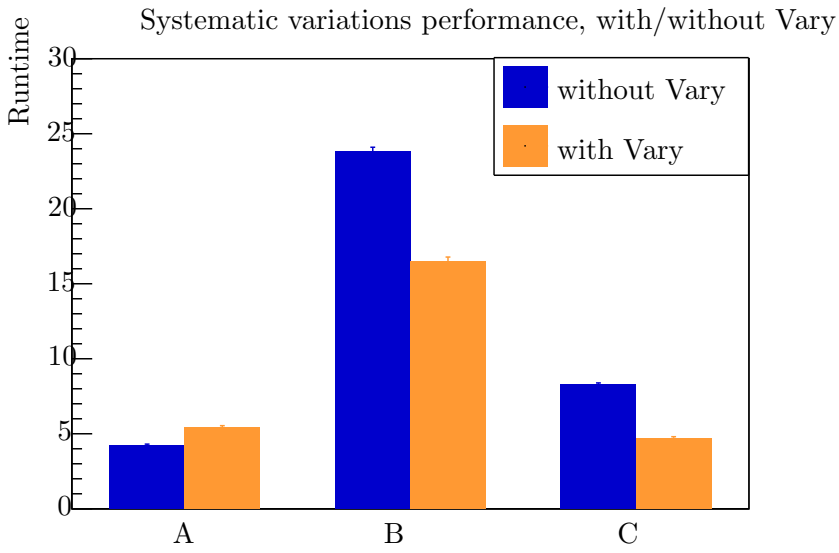

Figure 2. Runtimes for RDataFrame event loops for three different scenarios: **A**. 100M events, 1 histogram produced, 2 variations (nominal, up, down histograms filled). **B**. 10M events, 1 nominal histogram, 100 variations (101 histograms filled). **C** 100k events, 20 nominal histograms, 100 variations (2k histograms filled).

the C++ event loop without fear of the GIL. With this system it is also possible to use array columns, that RDataFrame exposes as the `RVec` type, as Numpy arrays in Python functions, with no copies in between: the Numpy arrays will be initialized to act as a view on the contents of the `RVec`. Listing 5 shows how this looks in practice.

```
@ROOT.Numba.Declare(["RVecD","RVecD"], "RVecD")
def good_pts(pts, etas):
    return pts[np.abs(etas) < 1]

df.Define("good_pts", "Numba::good_pts(pts, etas)")
```

Listing 5: Through `Numba.Declare`, the `good_pts` Python function is declared to Cling as a C function and can be used in RDataFrame as usual.

Most of the boilerplate code above could be generated automatically by RDataFrame, including the input and output types of the columns involved; therefore, in the future we expect to be able to simplify the API so that the code above will soon just be:

```
df.Redefine("pts", lambda pts, etas: pts[np.abs(etas) < 1])
```

6. Conclusions

RDataFrame keeps improving to simplify the life of HEP physicists, thanks to valuable feedback from the user community. Future work will introduce further enhancements, some of which are outlined in this work, as well as performance optimizations of the data handling in the inner event loop and more Pythonic interfaces for common use cases.

References

[1] Guiraud E, Naumann A and Piparo D 2017 TDataFrame: functional chains for ROOT data analyses (v1.0), *Zenodo*, https://doi.org/10.5281/zenodo.260230
[2] Piparo D, Canal P, Guiraud E, Pla XV, Ganis G, Amadio G, Naumann A and Tejedor E. 2018 RDataFrame: Easy Parallel ROOT Analysis at 100 Threads, *EPJ Web of Conferences 2019*, Vol. 214, p. 06029 EDP Sciences.
[3] Manca E Precision measurements of W detected at CMS *Doctoral dissertation, Universita & INFN Pisa (IT)*
[4] Padulano V E, Villanueva J C, Guiraud E and Saavedra E T Distributed data analysis with ROOT RDataFrame *EPJ Web of Conferences 2020* EDP Sciences Vol. 245, p. 03009
[5] Brun R and Rademakers F 1996 ROOT - An Object Oriented Data Analysis Framework, *Proceedings AIHENP'96 Workshop*, Nucl. Inst. & Meth. in Phys. Res. A 389 (1997) 81-86.
[6] David P Readable and efficient HEP data analysis with bamboo *EPJ Web of Conferences 2021* EDP Sciences Vol. 251, p. 03052
[7] CROWN framework by KIT-CMS, `https://github.com/KIT-CMS/CROWN`
[8] Blomer J, Canal P, Naumann A and Piparo D Evolution of the ROOT Tree I/O *EPJ Web of Conferences 2020* EDP Sciences, Vol. 245, p. 02030
[9] An S and Moneta L. C++ Code Generation for Fast Inference of Deep Learning Models in ROOT/TMVA *EPJ Web of Conferences 2021* EDP Sciences Vol. 251, p. 03040
[10] Galli M, Tejedor E and Wunsch S. A new PyROOT: Modern, interoperable and more pythonic. *EPJ Web of Conferences 2020* EDP Sciences, Vol. 245, p. 06004
[11] Lam S K et al., DOI 10.5281/zenodo.4343230