RDataFrame: Easy Parallel ROOT Analysis at 100 Threads

Danilo Piparo[1], *Philippe* Canal[2], *Enrico* Guiraud[1,3,*], *Xavier* Valls Pla[1], *Gerardo* Ganis[1], *Guilherme* Amadio[1], *Axel* Naumann[1], and *Enric* Tejedor[1]

[1]CERN
[2]FNAL
[3]University of Oldenburg

Abstract. The Physics programmes of LHC Run III and HL-LHC challenge the HEP community. The volume of data to be handled is unprecedented at every step of the data processing chain: analysis is no exception. Physicists must be provided with first-class analysis tools which are easy to use, exploit bleeding edge hardware technologies and allow to seamlessly express parallelism. This document discusses the declarative analysis engine of ROOT, `RDataFrame`, and gives details about how it allows to profitably exploit commodity hardware as well as high-end servers and manycore accelerators thanks to the synergy with the existing parallelised ROOT components. Real-life analyses of LHC experiments' data expressed in terms of `RDataFrame` are presented, highlighting the programming model provided to express them in a concise and powerful way. The recent developments which make `RDataFrame` a lightweight data processing framework are described, such as callbacks and I/O capabilities. Finally, the flexibility of `RDataFrame` and its ability to read data formats other than ROOT's are characterised, as an example it is discussed how `RDataFrame` can directly read and analyse LHCb's raw data format MDF.

1 Introduction

The ROOT project is committed to take physicists from data acquisition to publication as effectively as possible. The need to offer analysts simpler and yet powerful interfaces that could easily let them exploit the full potential of their hardware became all the more apparent with the increased luminosity and the upgrades of the LHC experiments foreseen for Run III [1], HL-LHC [2] and FCC [3] – with the consequent increase in the amount and complexity of available data. `ROOT::RDataFrame` [1], has been developed in order to address these requirements. In a similar vein to other modern data analysis frameworks such as Apache Spark's DataFrames [5] and Python's data analysis library pandas [6], `RDataFrame` exposes a declarative API designed to be easy to use correctly and hard to use incorrectly. Novel elements introduced by `RDataFrame` are the choice of programming language (C++), which allows usage of template metaprogramming to avoid runtime overhead while maintaining generality of interfaces, the integration of just-in-time compilation of user-defined expressions to make analysis definition concise when top performance is not required, and of course a tight integration with the rest of the ROOT data analysis toolkit. User-transparent

[*]e-mail: enrico.guiraud@cern.ch
[1]in the following just `RDataFrame`, first introduced in [4] as `ROOT::Experimental::TDataFrame`

task-based parallelism has been a goal since inception, and recent R&D [7] demonstrates that the programming model lends itself to multi-node distributed execution with no changes. Firstly, this paper will offer a high-level overview of `RDataFrame`'s software design (section 2). Section 3 will follow with a review of the most important, recently introduced features that contribute to make `RDataFrame` a fully fledged lightweight data processing framework. Finally, one real-world application of the framework and performance benchmarks are discussed in sections 4 and 5 respectively.

2 RDataFrame's software design

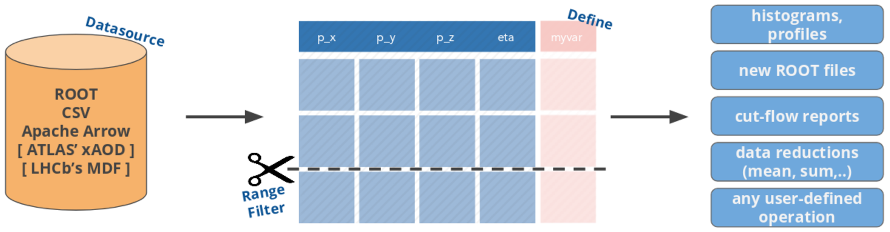
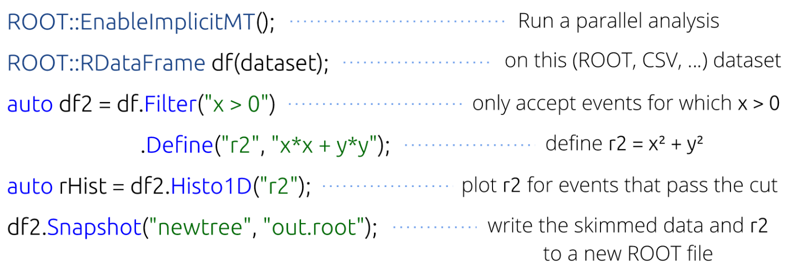


Figure 1. The `RDataFrame` framework reads from a columnar data format via a data source, applies transformations to the data (i.e. selects rows and/or defines new columns) and produces results (i.e. data reductions like histograms, new ROOT files, or any other user-defined object or side effect). Data sources for ATLAS' xAOD data format and LHCb's MDF binary data format exist but are not distributed with ROOT.



```
ROOT::EnableImplicitMT();                    Run a parallel analysis
ROOT::RDataFrame df(dataset);                on this (ROOT, CSV, ...) dataset
auto df2 = df.Filter("x > 0")                only accept events for which x > 0
             .Define("r2", "x*x + y*y");     define r2 = x² + y²
auto rHist = df2.Histo1D("r2");              plot r2 for events that pass the cut
df2.Snapshot("newtree", "out.root");         write the skimmed data and r2
                                               to a new ROOT file
```

Figure 2. A simple C++ `RDataFrame` analysis that performs event selection, defines a new quantity, produces a histogram and writes processed data to disk. All registered operations will be executed in a single event loop.

Design principles

At a high level, `RDataFrame` strives to expose modern, elegant and safe interfaces. The introduction of elements of **declarative programming** in the design (users say *what* they need to compute, `RDataFrame` chooses *how* to compute it) provides user-visible advantages such as less typing, increased readability and abstraction of complex operations. At the same time, by decoupling API from underlying implementation, the declarative paradigm allows for transparent optimisations (e.g. user-transparent parallel processing of range of events, lazy

evaluation, caching) that it would not have been possible to introduce in previous ROOT data analysis facilities such as `TTree`.
As shown in figure 2, the design also features elements of **functional programming** such as pure and higher order functions which encourage users to program in terms of small and reusable components with less side effects and less shared state; this in turn increases thread safety and code correctness: pure functions are thread safe by construction and are easier to test as they do not carry dependencies on global state. Furthermore, thanks to PyROOT's [8] automatic python binding generation, most of the framework's functionality is seamlessly available in Python, guaranteeing a consistent user experience.

Functional parts

Concretely, the framework is composed of three kinds of objects:

- *data sources* read columnar data and expose a common format-independent interface. This is a customisation point: expert users can implement a data source for their columnar data format of choice. Data sources are discussed in more detail in section 3.

- *nodes* are objects that represent one step of the data analysis workflow specified by the user. They form a computation graph which represents the full analysis workflow. With reference to figure 2, each `Filter`, `Define` or `Histo1D` invocation creates a node object which is appended to the node on which the method was called, hence forming a graph.

- *results*: most `RDataFrame` methods return a smart pointer (an `RResultPtr`, see [9]) to an object produced through data processing. For example, `Histo1D` returns a smart pointer to a histogram (`TH1D`) object. These smart pointers are the mechanism through which lazy execution is implemented: the actual data processing is only triggered upon access to one of the results produced by an `RDataFrame` (i.e. dereferencing of the smart pointer). During the data processing all previously booked results are produced simultaneously.

We distinguish between `RDataFrame` methods which return new `RDataFrame`-like objects (such as `Filter` or `Define`) and methods which return results. We refer to the former as *transformations* and to the latter as *actions*, following Spark's nomenclature.

Parallelisation scheme

The actual event loop is parallelised by processing different chunks of data in different tasks. Tasks are scheduled for execution on a thread pool, and the execution of each task updates a thread-local copy of each desired result with the output of the processing of the relevant data chunk. As an optional final step, thread-local partial results are merged into a single result object that will be handed to users. ROOT's task scheduler is currently Intel's Threading Building Blocks (TBB) library (see [10]).
Assuming the task scheduler employs a pool of worker threads of appropriate size (as it happens with TBB), task-based parallelism offers several advantages: there is no risk of overcommitting computing resources, as the task scheduler ensures that each thread runs one task at a time, also taking into account dependencies between tasks; this scheme also integrates well with other entities (e.g. experiments' frameworks) which schedule their own tasks, as long as all tasks are submitted to a common scheduler. Finally, redundant decompression of ROOT data is avoided by chunking inputs with the same granularity at which they were compressed.

3 Recently introduced features

3.1 Data sources

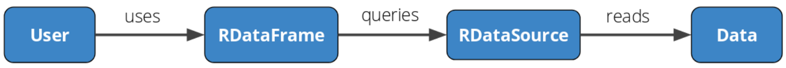


Figure 3. RDataFrame can read any columnar data format through a dedicated data source implementation. Expert users can implement and seamlessly integrate data sources for their format of choice.

Although the ROOT data format certainly plays a large role in HEP analyses, it is however not the only format of interest in science: a common user requirement is to read text files, in CSV format or similar, into ROOT; the ALICE experiment also expressed [11] interest in the possibility to use `RDataFrame`'s declarative analysis paradigm to process Apache Arrow in-memory tables, as part of their data analysis framework renovation effort. In order to address these use cases, the framework provides the interface type `RDataSource` (see [12]) which defines a minimal API that `RDataFrame` can use to read arbitrary tabular data formats.

In practice, `RDataSource` is a C++ abstract base class which imposes certain functional requirements onto its implementations; concrete derived types will provide adaptors that `RDataFrame` can leverage to read any kind of tabular data formats. `RDataFrame` calls into `RDataSources` to retrieve information about the data, to obtain (thread-local) readers or "cursors" for selected columns and to advance such readers to the desired data entry.

`RDataSource` not only extends `RDataFrame` to support other formats than ROOT, but it decouples the analysis code from the format analysed so that users can use the same exact code to process potentially very different datasets.

CSV and Apache Arrow inputs are currently supported through this mechanism and prototypes for LHCb's MDF binary data format and ATLAS' xAOD event data model (see [13]) have been developed, which goes to show the flexibility of the approach.

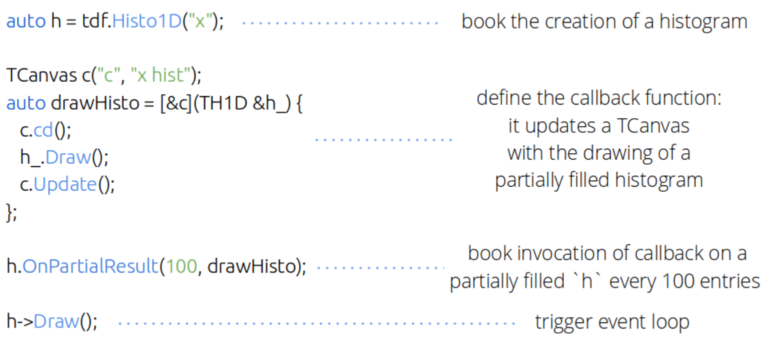
3.2 User-defined callbacks

It is possible to schedule execution of arbitrary functions (callbacks) during the event loop. Callbacks can be used, for example, to inspect the ongoing filling of a histogram as the event loop is running, to save results to a file every time a certain number of new entries are processed, or to display a progress bar that indicates event loop progress.
As an example, figure 4 shows how one can draw an up-to-date version of a result histogram every 100 entries.

At user's discretion, callbacks can be called once at the beginning of the event loop or every time a certain amount of new entries have been processed. Users can also decide whether the callback should be called only by one thread at a time (*which* thread calls the callback might vary during execution, but the framework guarantees that the given amount of entries will have been processed between calls) or by all threads, potentially concurrently, in which case the user is responsible for providing a thread safe callback function (more information is available at [14]).
This feature is currently only available in C++, not Python.

4 A real-world RDataFrame application

As a case study, we would like to discuss a real-world `RDataFrame` C++ application developed by ROOT users from the ATLAS collaboration [15]. Refer to figure 5 for the applica-



```
auto h = tdf.Histo1D("x");

TCanvas c("c", "x hist");
auto drawHisto = [&c](TH1D &h_) {
  c.cd();
  h_.Draw();
  c.Update();
};

h.OnPartialResult(100, drawHisto);

h->Draw();
```

book the creation of a histogram

define the callback function: it updates a TCanvas with the drawing of a partially filled histogram

book invocation of callback on a partially filled `h` every 100 entries

trigger event loop

Figure 4. An example callback usage, commented line by line. The `drawHisto` function is called by one of the worker threads every one hundred entries processed, and it refreshes a canvas on which the state of the histogram is displayed. The callback need not be thread-safe, as it is never executed concurrently. In multi-thread executions, the partial result that the callback will receive as argument will be the thread-local copy that the relevant worker thread is employing.

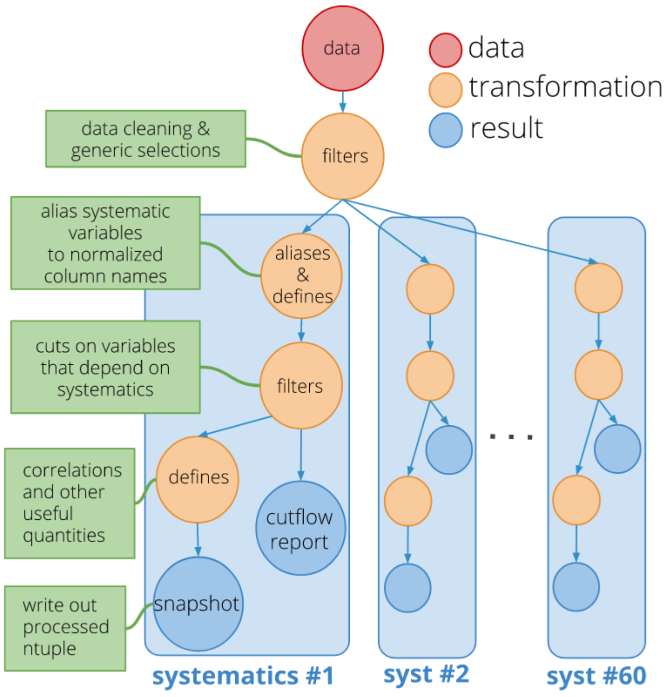

data
data
transformation
result
data cleaning & generic selections
filters
alias systematic variables to normalized column names
aliases & defines
cuts on variables that depend on systematics
filters
correlations and other useful quantities
defines
cutflow report
write out processed ntuple
snapshot
. . .
systematics #1
syst #2
syst #60

Figure 5. A representation of the computation graph of an `RDataFrame` C++ application that performs ntuple to ntuple processing of simulated data. For each of sixty different systematics, the input ntuple is skimmed and several columns containing quantities relevant for further processing and dependent on systematics are added. Sixty new output ROOT files are produced within the same event loop.

tion's computation graph and a brief explanation of its purpose. It is worth highlighting a few striking features of the application's codebase: first of all, thanks to the declarative programming model, the program's `main` function is a simple sequence of `Filter` and `Define` calls, followed by a call to schedule a write-out of the processed ntuple (`RDataFrame`'s `Snapshot` method); the definition of the analysis workflow is clearly separated from the definition of the smaller helper functions. Given a little familiarity with `RDataFrame`'s API, the workflow is grasped quickly, with no need to dive into finer details if not required: such details are encapsulated in several small, pure helper functions. Of course good software design practice is to always make applications as readable and modular as possible; however, `RDataFrame`'s programming model makes it natural and encourages users to write code with such qualities.

Finally, it is worth noting that event selection, calculation of new quantities and writing of the sixty output ROOT datasets all happen within a single parallelised event loop, an achievement that would have required significant effort and attention to low-level details with ROOT interfaces preceding `RDataFrame`.

5 Scaling and performance benchmarks

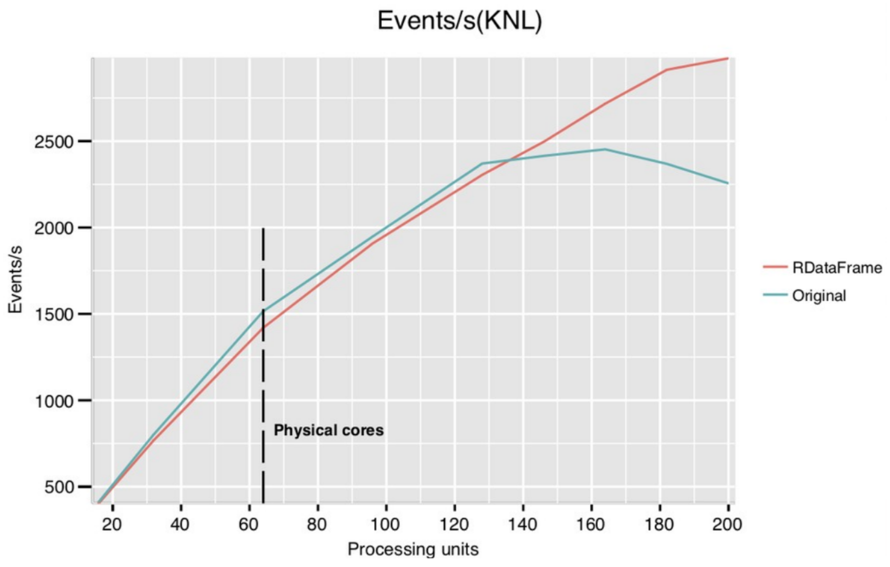


Figure 6. Scaling of a Monte Carlo QCD Low-PT event generation and analysis on the fly for an ad-hoc implementation using a patched ROOT 5 and POSIX threads (labeled "original" in the plot) and an `RDataFrame` rewrite of that same application (yielding identical results). No disk reads or writes are performed by either application. KNL architecture, 64 physical cores.

In order to measure the scaling of a realistic `RDataFrame` application, we take a pre-existing parallel code that generates low-pt QCD events, processes them and plots some quantities of interest as ROOT histograms. This application has been developed for research purposes by an expert ROOT user, who based it on a fork of ROOT version 5 patched to allow multi-thread data analysis. Data is produced and analysed on the fly: absence of direct disk I/O makes it possible to scale to a large number of cores without being limited by the hardware's reading speed. We compare the original code with an `RDataFrame`-based rewriting which produces identical results and reuses most of the numerical computation logic. As figure 6 shows, `RDataFrame` introduces a small overhead with respect to the original ad-hoc code, which made direct use of lower-level ROOT interfaces. On the other hand, `RDataFrame`

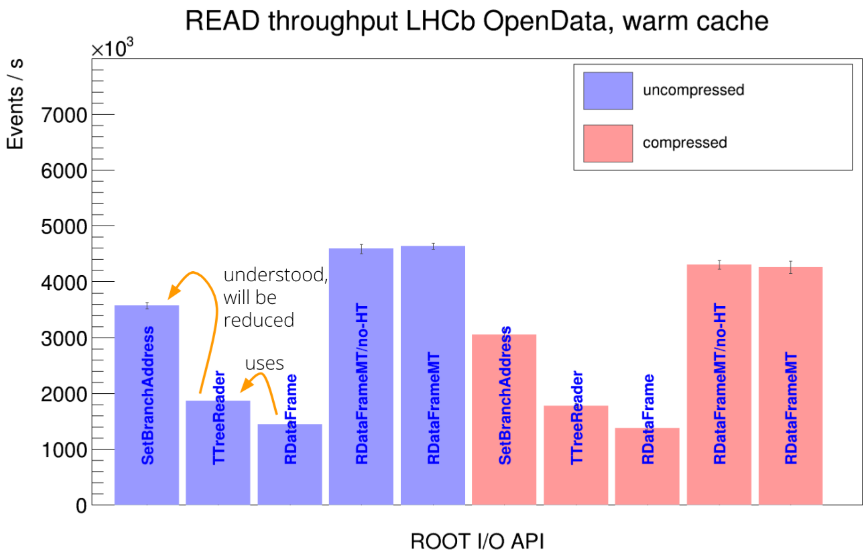


Figure 7. Read speed (events/s) on an LHCb OpenData dataset for three different reading APIs available in ROOT: `TTree+SetBranchAddress`, `TTreeReader`, `RDataFrame` (with and without implicit multi-threading enabled). The benchmark was run on a machine with four physical cores, 3.6 GHz each, and an off-the-shelf SSD. `TTreeReader` adds a non-negligible overhead on top of direct `TTree` usage, whose origin is understood. This overhead will be reduced in future ROOT releases. `RDataFrame` employs `TTreeReader` internally, inheriting the overhead as a consequence.

scales to a larger amount of cores thanks to task-based parallelism and less lock contention during the event loop.

In order to assess `RDataFrame`'s I/O performance, the measurements of [16] were repeated with the latest release of ROOT; the results are displayed in figure 7. No significant changes with respect to the original measurements of [16] were detected: `RDataFrame` is the fastest interface ROOT offers to analyse ROOT data if one takes into account the simplicity of expressing parallelism, although single-thread execution suffers from an important overhead with respect to direct usage of `TTree`. The cause of such overhead has been identified and mitigations will be introduced in future releases.

6 Conclusions and Future Work

We presented a modern, declarative, parallel framework for data analysis and manipulation. `RDataFrame` is officially part of ROOT as of version 6.14, offers a C++ interface as well as Python bindings and it has already been employed successfully in real-world HEP analyses. `RDataFrame`'s task-based parallelism scales successfully to many-core architectures.

Future work will revolve around offering more "pythonic" PyROOT bindings, including import/export of numpy arrays, low-level performance optimization, especially aimed at reducing single-thread overhead with respect to direct `TTree` usage, and integration with other ROOT interfaces such as TMVA. In addition, promising R&D on distributed execution of `RDataFrame`-based analyses is being carried on (see [7]), framing `RDataFrame` as a real "Swiss Army knife" for HEP data processing.

References

[1] J. Alves, Antonio Augusto, G. Amadio, N. Anh-Ky, L. Aphecetche, J. Apostolakis, M. Asai, L. Atzori, M. Babik, G. Bagliesi, M. Bandieramonte et al., Tech. Rep. HSF-CWP-2017-001 (2017), `http://cds.cern.ch/record/2298968`

[2] A. G., B.A. I., B. O., F. P., L. M., R. L., T. L., *High-Luminosity Large Hadron Collider (HL-LHC): Technical Design Report V. 0.1*, CERN Yellow Reports: Monographs (CERN, Geneva, 2017), `https://cds.cern.ch/record/2284929`

[3] J.L.A. Fernandez, C. Adolphsen, A.N. Akay, H. Aksakal, J.L. Albacete, S. Alekhin, P. Allport, V. Andreev, R.B. Appleby, E. Arikan et al., Journal of Physics G: Nuclear and Particle Physics **39**, 075001 (2012)

[4] G. Amadio, J. Blomer, P. Canal, G. Ganis, E. Guiraud, P.M. Vila, L. Moneta, D. Piparo, E. Tejedor, X.V. Pla, Journal of Physics: Conference Series **1085**, 042008 (2018)

[5] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica, *Spark: Cluster Computing with Working Sets*, in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing* (USENIX Association, Berkeley, CA, USA, 2010), HotCloud'10, pp. 10–10, `http://dl.acm.org/citation.cfm?id=1863103.1863113`

[6] W. McKinney et al., *Data structures for statistical computing in python*, in *Proceedings of the 9th Python in Science Conference* (Austin, TX, 2010), Vol. 445, pp. 51–56

[7] S. Murali, E.T. Saavedra, E. Guiraud, D. Castro, J.C. Villanueva, D. Piparo, *Pyrdf* (2018), `https://doi.org/10.5281/zenodo.1464080`

[8] J. Generowicz, W.T. Lavrijsen, M. Marino, P. Mato (2004)

[9] *RResultPointer user guide*, `https://root.cern.ch/doc/master/classROOT_1_1RDF_1_1RResultPtr.html`

[10] D. Piparo, E. Tejedor, E. Guiraud, G. Ganis, P. Mato, L. Moneta, X.V. Pla, P. Canal, Journal of Physics: Conference Series **898**, 072022 (2017)

[11] G. Eulisse, personal communication

[12] *RDataSource user guide*, `https://root.cern.ch/doc/master/classROOT_1_1RDF_1_1RDataSource.html`

[13] W.U. Dharmaji, *xAOD-DataSource: An implementation of ROOT's RDataSource interface for reading xAOD files through RDataFrame interface.* (2018), `https://doi.org/10.5281/zenodo.1410372`

[14] *OnPartialResult user guide*, `https://root.cern.ch/doc/master/classROOT_1_1RDF_1_1RResultPtr.html#a4ece251b1dd30ba269cc67812ed94418`

[15] F. Meloni, personal communication

[16] J. Blomer, Journal of Physics: Conference Series **1085**, 032020 (2018)