Distributed Analysis in Production with RDataFrame

Marta Czurylo[1,*], *Vincenzo Eduardo* Padulano[1], *Danilo* Piparo[1], and *Andrea Maria* Ola Mejicanos[2]

[1]CERN, Esplanade des Particules 1, 1211 Geneva 23, Switzerland
[2]Berea College, 101 Chestnut St, Berea, KY 40403, USA

Abstract. The ROOT software package provides the data format used in High Energy Physics by the LHC experiments. ROOT offers a data analysis interface called RDataFrame, which has proven to adapt well to the requirements of modern physics analyses. However, with the increasing data collected by the LHC experiments, the challenge to perform an efficient analysis expands. One of the solutions to ease this challenge is the leverage of modern high-performing distributed computing environments, for which RDataFrame provides an easy-to-use interface layer - the distributed RDataFrame.

In this paper, we show that the distributed RDataFrame is out of the experimental testing phase, and it is now ready for production thanks to a stabilized user interface. We delve into recent improvements of the distributed RDataFrame, including memory management, C++ code inclusion, and Pythonizations of the interface that allow running the workflows seamlessly. This includes running the distributed RDataFrame on various Analysis Facilities, which is discussed towards the end of the paper.

1 Introduction

RDataFrame (RDF) has been ROOT's analysis interface since ROOT 6.14 (2018) [1], and it has been thoroughly tested on single and multi-core machines ever since. In ROOT 6.24 (2021), the initial RDataFrame functionalities were expanded by introducing the distributed RDataFrame [2]. This fully Pythonic package allows users to benefit from the distributed computation on a number of nodes while using an already-known PyROOT RDataFrame API. The currently supported backends in distributed RDF are Apache Spark [3] and the Python library Dask [4].

Until now, the distributed RDataFrame has been part of the ROOT's experimental namespace, signifying ongoing development efforts without promising backward compatibility to the users. With the current level of development, especially considering the ergonomics of the user interface, available data input sources and Pythonizations within the ROOT framework, this paper claims that the distributed RDataFrame is ready for production. In particular, we will consider code stability improvements in Section 2, a set of new features in Section 3 and the performance of the distributed RDataFrame in Analysis Facilities in Section 4. The summary and next steps are given in Section 5.

*e-mail: marta.maja.czurylo@cern.ch

2 Code Stability

This section describes recent efforts to improve the code stability in the distributed RDataFrame, including improvements to the user-interfacing functions and reduced application memory usage.

2.1 User Interface

The user interface of the distributed RDataFrame has been improved so that the local and distributed RDataFrame versions are compatible in aspects such as RDF object construction and function calls.

Listing 1 shows different ways of constructing an RDataFrame object. Case 1 is a local RDataFrame constructor while case 2 is the distributed RDataFrame constructor, divided into two sub-cases based on the selected backend, either Dask or Spark. Albeit similar, the constructors shown in cases 1 and 2 differ in a few instances. Firstly, by the current use of the experimental namespace `RDF.Experimental` in the distributed RDataFrame constructors. Secondly, by an addition of `Distributed.Dask` or `Distributed.Spark` in case of the distributed RDataFrame constructors. These differences force users to foresee the execution type at the time of writing the application and to change the constructors every time they want to run an analysis on different resources. In order to simplify the RDF usage and allow for easier transition between the local and distributed scenarios, a unified constructor is introduced, see case 3. The optional third argument allows users to specify the executor. The options are to either not use the third argument, hence requesting a local execution, or to use either `daskclient` or `sparkcontext`, depending on the chosen backend. Both cases 2 and 3 of constructing the distributed RDataFrame remain available to the users, but the latter is recommended.

```
# Case 1: Local RDataFrame constructor
RDataFrame = ROOT.RDataFrame
df = RDataFrame(treeName, fileName)

# Case 2a: Distributed RDataFrame constructor, using Dask backend
RDataFrame = ROOT.RDF.Experimental.Distributed.Dask.RDataFrame
df = RDataFrame(treeName, fileName, daskclient=daskclient)

# Case 2b: Distributed RDataFrame constructor, using Spark backend
RDataFrame = ROOT.RDF.Experimental.Distributed.Spark.RDataFrame
df = RDataFrame(treeName, fileName, sparkcontext=sparkcontext)

# Case 3: New, unified constructor of either local or distributed RDataFrame
RDataFrame = ROOT.RDataFrame(treeName, fileName, executor=SupportedExecutor)
```

Listing 1: RDataFrame constructors depending on the mode of operation.

Once an RDataFrame object is constructed, the analysis workflow starts. The goal is that all RDF operations (actions, transformations and queries)[1] performed on the RDataFrame should be available for both local and distributed cases as long as they can be applied to both[2]. For this reason, many of the query functions were recently added to the distributed RDataFrame, for example, `GetColumnNames` or `GetColumnType("columnName")`. Additionally, some of the function calls differed between local and distributed RDataFrame.

[1] See [5] for details of the RDF operations categories.

[2] For example, `DefineSlot` function is only usable in the case of local, multi-threaded RDF, hence not needed in the distributed RDF package.

Listing 2 shows an example of such a function call. Every time the user's application was designed agnostically towards the type of execution (local vs distributed), a conditional statement had to be put in place (line 2 – 5). Now, the unified version of the function calls: `ROOT.RDF.Experimental.VariationsFor` (line 8) and `ROOT.RDF.RunGraphs` are available, making the user experience a true zero code change between local and distributed versions of RDataFrame.

```
1 # Without a unified API
2 if type(df).__module__ == "DistRDF.DataFrame":
3     variationsfor_func = ROOT.RDF.Experimental.Distributed.VariationsFor
4 else:
5     variationsfor_func = ROOT.RDF.Experimental.VariationsFor
6
7 # With a unified API: df can be either a local or a distributed RDataFrame
8 variationsfor_func =  ROOT.RDF.Experimental.VariationsFor(df)
```

Listing 2: The conditional statement needed to make an application usable in both local and distributed RDataFrame executions and the new unified API with an example of the VariationsFor function.

2.2 Memory Usage

As the distributed RDataFrame package was being enriched with new features and its performance was being improved, applications of increased complexity started to be written, demonstrating these increasing capabilities. For example, the benchmark provided by the Analysis Grand Challenge [6] was implemented in RDataFrame [7] and was executed distributedly on the SWAN platform [8] using the Dask scheduler and 64 HTCondor workers (1 core each). The workflow covered all analysis steps, including the machine learning inference for the jet tagging. We observed that during execution, the main memory consumption increased towards the 2GiB per core limit, which led to an unsuccessful execution. The problem was identified as improperly managed artefacts of the cached computation graphs, and a better management of these caches was introduced as a solution. Figure 1 compares the memory usage before and after this solution was found. The red line marks the main memory consumption of 1GiB per core, which, after solving the issue, is never reached by any of the workers, making the applications run smoothly in this and similar analyses contexts. However, note must be taken that memory management of an arbitrary analysis depends on various factors, independent of the RDataFrame machinery; hence, we cannot guarantee this solution will always suffice.

3 New Features

A number of new features were recently introduced to make the distributed RDataFrame more aligned with the needs of the users.

3.1 User Interface – C++ Code Inclusion

Many modern HEP analyses are written almost entirely in Python. However, the use of certain C++ libraries or functionalities is sometimes crucial and cannot be fully omitted. There is a straightforward method to include C++ code inside a local, Python-based RDataFrame application, but for the distributed RDataFrame situation, it is more complex. In order to

make the distributed RDF API more user-friendly, we add three main features: distributing headers, declaring C++ code, and distributing shared libraries.

The basis of including the C++ code in a PyROOT application is by using the `TInterpreter` class, which for the local case is documented in [9]. In the distributed scenario, we need to distribute the same code to all workers. Listing 3 shows how to include a necessary C++ header file at runtime of the Python application using the Dask backend when no specific API function is available. The crucial part is the use of the `initialize` function, which takes a Python callable as an argument (line 8). This callable will be executed both on the local machine (by default) and at the beginning of every distributed execution task. Inside the callable function itself, we first need to find the path to the header on the file system of the worker (lines 5 – 6). In the case of Dask, its API can be used to retrieve the local worker directory to which RDataFrame has sent the header in a previous step of the execution. Finally, by using `gInterpreter.Declare`, we declare to the C++ interpreter that we want to include all the code that is inside the header `myheader.h` (line 7). In order for the user not to get involved with this somewhat complex method to distribute headers, we introduced a new API function tailored for such distributed use cases. The user can now simply write one line, see line 11.

```
1  # Case 1: Using gInterpreter to include a C++ header - the old way
2  def load_header():
3      """Load C++ helper functions. When using distributed RDataFrame '
       my_header.h' is copied to the local_directory of every worker (via `
       distribute_unique_paths`)"""
4
5      localdir = get_worker().local_directory
6      cpp_header = Path(localdir) / "my_header.h"
7      ROOT.gInterpreter.Declare(f'#include "{str(cpp_header)}"')
8  ROOT.RDF.Experimental.Distributed.initialize(load_header)
9
10 # Case 2: Using the DistributeHeaders function - the new way
11 ROOT.Distributed.DistributeHeaders("my_header.h")
```

Listing 3: Examples of including a C++ header in otherwise Python based application.

Similarly, the problem would appear for the user wanting to declare the C++ code. The following code example (Listing 4) shows the code declaration and an example of using the declared function in a `Filter` operation of the RDataFrame. As an added technical but very important detail, users should not worry about C++ redefinition errors due to the same code being re-declared at every task, which can happen more than once per worker process. RDataFrame internally takes care of ensuring that the code is declared to the C++ interpreter only once per worker process.

```
1  ROOT.Distributed.DeclareCppCode("""
2          bool check_number_less_than_five(int num){
3              return num < 5;
4          }
5          """)
6
7  df = ROOT.RDataFrame(treeName, fileName, executor=SupportedExecutor)
8  df_filtered = df.Filter("check_number_less_than_five(rdfentry_)")
```

Listing 4: DeclareCppCode function.

Finally, the user can also distribute a shared library. In the following example (Listing 5) we demonstrate how to load a pre-compiled shared library in a distributed execution, which, as usual, also needs the declaration of the corresponding header.

```
ROOT.Distributed.DistributeHeaders("my_header.h")
ROOT.Distributed.DistributeSharedLibs("lib_my_header.so")
```

Listing 5: DistributeSharedLibs function.

3.2 Input Data Sources

Alongside the new functions, support for new input data sources was introduced. On top of the always supported TTree and the recently added RNTuple [10], RDataFrame now also starts to support the RDatasetSpec. RDatasetSpec offers a way to add specific data samples together with their metadata information. Listing 6 shows this in detail.

```
# Create and populate the metadata object
meta = ROOT.RDF.Experimental.RMetaData()
meta.Add("meta_key", "meta_value")

# Create a sample
mySample = ROOT.RDF.Experimental.RSample("mySampleName", treeName, fileName,
    meta)

# Use RDatasetSpec and create a distributed RDataFrame object
spec = ROOT.RDF.Experimental.RDatasetSpec()
spec.AddSample(mySample)
df = ROOT.RDataFrame(spec, executor=daskclient)
```

Listing 6: Using RDatasetSpec in ditributed RDataFrame to add samples with metadata information associated with them.

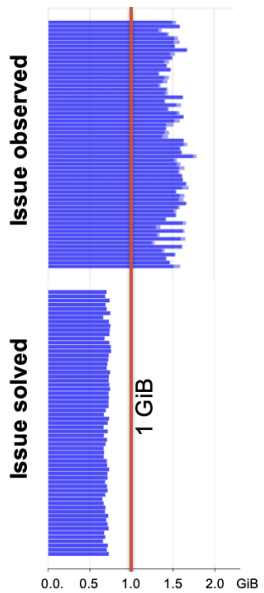

Figure 1: Main memory consumption per core before and after solving the memory usage issue. Each blue bar represents the memory usage of a single-cored worker. The red line is placed at 1GiB.

The basic functionality of this feature is already available to users. However, additional support for functions such as `FromSpec`, which allows for the creation of an RDataFrame object from a specification file in JSON format, is still a work in progress. The usage of this function is shown in Listing 7.

```
df = ROOT.RDF.Experimental.FromSpec("my_spec.json", executor=daskclient)
```

Listing 7: FromSpec function.

3.3 Pythonizations of the Interface

Yet another way to facilitate the distributed analysis workflow is a more extensive use of Pythonizations throughout the ROOT framework. As an example, we can look at the Analysis Grand Challenge [6] workflow, where Boosted Decision Trees (BDT) training is used for the jet tagging. The pre-trained models are saved using a popular XGBoost library [11], which allows for the efficient gradient boosting. The models are then loaded inside the workflow in order to select the correct jets. Until recently, an external C++ library was needed to process the BDT model data in the RDataFrame implementation of AGC.

To facilitate this kind of workflows, a Pythonization in the ROOT framework was introduced, allowing XGBoost models to be directly saved in ROOT format as shown in Listing 8, line 4. As a result, the management of dependencies, especially in the distributed analysis context, becomes much easier. For details on the implementation of this feature in the RDataFrame analysis flow, see the ROOT AGC repository [12].

```
from xgboost import XGBClassifier
myBdt = XGBClassifier()
myBdt.load_model(f"myModel.json")
ROOT.TMVA.Experimental.SaveXGBoost(myBdt, "myBdt", "myModel.root", num_inputs
    =num_inputs)
```

Listing 8: An example of using XGBoost inside ROOT.

4 Analysis Facilities

Throughout the years, the distributed RDataFrame has been tested on many Analysis Facilities (AF), for example, in HPC centers such as Jülich [13], CERN [2] and Lumi [14], proving to be the right tool for an interactive analysis usage. The improvements in stability and addition of the new features described in Sections 2 and 3 allow distributed RDataFrame to be even more suitable for the AF use.

One of the most recent testing grounds for the distributed RDF in the context of the Analysis Facilities is the CERN SWAN Analysis Facility. More details on this effort can be found in [15]. To summarise, SWAN is the web-based platform that provides the EOS storage access [16] and facilitates offloading of the RDataFrame analysis program to, for example, the CERN HTCondor pools [17] via Dask plugins.

By leveraging many of the new features and improvements described in this paper, a follow-up to the studies presented in [18] was performed. It was possible to run the complex Analysis Grand Challenge workflow, including the machine learning inference using the distributed RDataFrame with as many as 128 workers, with both TTree and RNtuple. Figure 2 shows the runtime of the full AGC execution vs the number of workers for data stored either in TTree or RNTuple formats. An ideal scaling can be observed for both data formats, while

RNTuple is between 1.5 to 2 times faster than TTree. Importantly, this significant improvement comes with no change to the user's analysis code between TTree and RNTuple cases. The work is still carried out to make an even better integration of the RNTuple within the distributed RDataFrame.

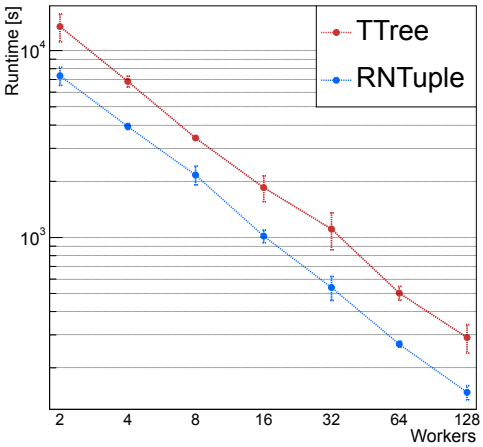

Figure 2: Runtime of the RDataFrame implementation of the Analysis Grand Challenge vs number of workers for the input data stored either in TTree or RNTuple formats.

5 Conclusion

The goal of this paper was to show that the distributed RDataFrame is ready for production. We presented enhanced code stability via both a unified RDataFrame object constructor and the management of the application memory. A number of new features allow for easier integration with C++ workflows and incorporation of various input sources, beyond TTree. Finally, also thanks to these improvements, distributed RDataFrame achieves excellent performance on various analysis facilities, including the most recent effort of running a complex Analysis Grand Challenge workflow on the CERN SWAN platform with both TTree and RNTuple. In the following months, we will enhance and extend functionalities related to the RDatasetSpec class within the distributed RDataFrame, and we will make integration of the distributed RDataFrame with RNTuple even stronger, aiming at even bigger performance gains in the future.

References

[1] Piparo, D. et al., RDataFrame: Easy Parallel ROOT Analysis at 100 Threads, EPJ Web Conf. **214**, 06029 (2019). 10.1051/epjconf/201921406029

[2] Padulano, V.E. et al., Leveraging State-of-the-Art Engines for Large-Scale Data Analysis in High Energy Physics, Journal of Grid Computing **21** (2023). 10.1007/s10723-023-09645-2

[3] Zaharia, M. et al., Spark: Cluster Computing with Working Sets, in *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)* (USENIX Association, Boston, MA, 2010), https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets

[4] Rocklin, M., Dask: Parallel Computation with Blocked algorithms and Task Scheduling, in *SciPy* (2015), https://api.semanticscholar.org/CorpusID:63554230

[5] The ROOT Team, ROOT Reference Guide - ROOT::RDataFrame Class Reference, accessed: 4th February 2025, https://root.cern/doc/master/classROOT_1_1RDataFrame.html/

[6] Held, A., Shadura, O., The IRIS-HEP Analysis Grand Challenge, p. 235 (2022). 10.22323/1.414.0235

[7] Padulano, V.E. et al., First implementation and results of the Analysis Grand Challenge with a fully Pythonic RDataFrame, EPJ Web of Conf. **295**, 06011 (2024). 10.1051/epjconf/202429506011

[8] Piparo, D. et al., SWAN: a Service for Interactive Analysis in the Cloud, Future Gener. Comput. Syst. **78**, 1071 (2018). 10.1016/j.future.2016.11.035

[9] The ROOT Team, ROOT Manual - Python interface: PyROOT, accessed: 4th February 2025, https://root.cern/manual/python/

[10] Naumann, A. et al., ROOT for the HL-LHC: data format (2022), 2204.04557.

[11] Chen, T., Guestrin, C., XGBoost: A Scalable Tree Boosting System, in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (ACM, New York, NY, USA, 2016), KDD '16, pp. 785–794, ISBN 978-1-4503-4232-2, http://doi.acm.org/10.1145/2939672.2939785

[12] The ROOT Team, RDataFrame implementation of the Analysis Grand Challenge, accessed: 4th February 2025, https://github.com/root-project/analysis-grand-challenge

[13] Boulis, J, Benchmarking Distributed Analysis at the Jülich HPC Center. CERN openlab Summer Student Lightning Talks (1/2) (2023), https://cds.cern.ch/record/2868375

[14] Mehrabi, A., Hahnfeld, J., Padulano, V.E., Evaluation of HPC Storage Systems for HEP Analysis (2024), https://doi.org/10.5281/zenodo.13847467

[15] Sciaba, A. et al., A Pilot Analysis Facility at CERN, Architecture, Implementation and First Evaluation (2024), https://indico.cern.ch/event/1338689/contributions/6010680/

[16] Peters, A.J., Sindrilaru, E.A., Adde, G., EOS as the present and future solution for data storage at CERN, J. Phys.: Conf. Ser. **664** (201). 10.1088/1742-6596/664/4/042042

[17] Thain, D., Tannenbaum, T., Livny, M., Distributed computing in practice: the Condor experience., Concurrency - Practice and Experience **17**, 323 (2005).

[18] Czurylo, M. et al., Seamless transition from TTree to RNTuple analysis with RDataFrame (2024), https://indico.cern.ch/event/1330797/contributions/5796495/