Declarative Big Data Analysis for High-Energy Physics: TOTEM Use Case

Valentina Avati[1], Milosz Blaszkiewicz[1], Enrico Bocchi[2], Luca Canali[2], Diogo Castro[2], Javier Cervantes[2], Leszek Grzanka[1], Enrico Guiraud[2], Jan Kaspar[2], Prasanth Kothuri[2], Massimo Lamanna[2], Maciej Malawski[1], Aleksandra Mnich[1], Jakub Moscicki[2], Shravan Murali[2], Danilo Piparo[2], and Enric Tejedor[2]

[1] AGH Universtity of Science and Technology, Krakow, Poland
{grzanka,malawski}@agh.edu.pl
[2] CERN, CH-1211 Geneva 23, Switzerland,
{first.last}@cern.ch

Abstract. The High-Energy Physics community faces new data processing challenges caused by the expected growth of data resulting from the upgrade of LHC accelerator. These challenges drive the demand for exploring new approaches for data analysis. In this paper, we present a new declarative programming model extending the popular ROOT data analysis framework, and its distributed processing capability based on Apache Spark. The developed framework enables high-level operations on the data, known from other big data toolkits, while preserving compatibility with existing HEP data files and software. In our experiments with a real analysis of TOTEM experiment data, we evaluate the scalability of this approach and its prospects for interactive processing of such large data sets. Moreover, we show that the analysis code developed with the new model is portable between a production cluster at CERN and an external cluster hosted in the Helix Nebula Science Cloud thanks to the bundle of services of Science Box.

Keywords: High-Energy Physics · distributed data analysis · Apache Spark · scalability.

1 Introduction

The High-Energy Physics (HEP) community of thousands of researchers around the world processing massive amounts of data has always been renown for driving the development of distributed processing tools and infrastructures. Regarding the tools, the predominant software toolkit for data analysis is ROOT [6]. ROOT provides all the functionalities required to deal with big data processing, statistical analysis, visualisation and storage. To give an idea of its importance, all the data collected so far by the Large Hadron Collider (LHC), the particle accelerator hosted at CERN, is stored in ROOT format (around 1 EB). Regarding the infrastructures, batch processing on clusters and grid technologies are the typical means of operations on HEP data.

On the other hand, recent developments in commercial big data processing tools and infrastructures, which include toolkits such as Apache Spark [19] and cloud computing, show the importance of high-level interfaces and user-friendly APIs to exploit the full potential of new data analysis infrastructures. This becomes even more apparent with the upgrades of the LHC experiments foreseen for Run III [2] and High-Luminosity LHC (HL-LHC) [3] and the consequent increase in the amount and complexity of data to be collected and processed. Specifically, the HL-LHC will operate at at least 10 times higher data rate than the current machine. With such an increased demand for data storage and processing, together with the need for more user-friendly analysis tools, investigating new approaches for big data analysis, become an important research problem.

To address this challenge, we present, in this paper, a new declarative programming model of ROOT, called RDataFrame, and its distributed processing backend based on Apache Spark. The developed framework enables high-level operations on the data, while preserving performance, scalability and compatibility with existing HEP data files and software. Specifically, while using the parallel processing of Spark, optimized C++ code runs on the backend, reading ROOT files directly, and uses the wide set of existing ROOT tools for creating high quality histograms, plots and statistical analysis calculations.

In order to evaluate the developed framework, we present our experience with porting a real production analysis of 4.7 TB data from the TOTEM [8] experiment at LHC. With dedicated experiments on the Helix Nebula Cloud and a production Spark cluster at CERN, we show the (1) correctness of the results, (2) scalability of this approach, (3) prospects for interactivity with the use of ScienceBox [7], and (4) portability of the high-level code across infrastructures.

2 Related work

Apache Spark [19] is probably the most popular, open-source framework for distributed analysis on clusters, providing scalability and fault tolerance capacity. Its Directed Acyclic Graph (DAG) and task schedulers features allow for the deployment of data transformation operations such as filtering and reductions in a scalable way on large clusters. In this work, we leverage these features to parallelize and scale ROOT jobs across a cluster.

Several approaches to usage of big data tools in HEP have been proposed. The main challenges come from the fact that ROOT is based on C++ and relies heavily on its native objects for data serialization into its specific data format, while tools such as Spark are Java based.

One of the approaches [18] is thus to transform the ROOT input data to HDF5 format before porting the analysis code from ROOT C++ to Spark operations. This has the advantage of allowing usage of standard HDF5 libraries and common Spark operations. Limitations come from the fact that data need to be converted first, and then stored for processing, however our approach allows unmodified input ROOT files to be used.

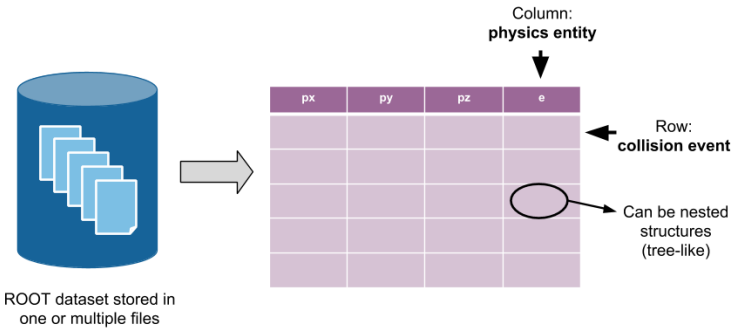

Fig. 1. Layout of a columnar ROOT dataset.

A different approach consists of reading ROOT files into the native Spark structures. It can be done using the Spark-Root connector developed by the CMS Big Data project [10]. This approach requires data processing jobs to be written using native Spark Dataframe APIs, which has the advantage of being compatible with popular big data toolkits, at the cost of the effort needed to re-implement all the code in a new programming language. Moreover, performance overheads of the code running in Java Virtual Machine may become non-negligible in comparison to direct usage of C++ as we propose.

In addition to Java-based frameworks, other toolkits such as Python's Dask [1] could be used. The framework allows running the same analysis at the local machine as well as at the cluster with minimal changes to the code. Unfortunately, those benefits come at a certain price. Dask requires the developer to go very deep into the technical details of the parallelization, adding another responsibility for the developer, in this case the physicist.

We should also mention here earlier approaches [12] of using Hadoop for implementing HEP data analysis using MapReduce model. Their experience shows that using HDFS for data storage can reduce network I/O thanks to using data locality. In our approach, we directly use EOS, a standard storage at CERN, which does not take advantage of data locality as with HDFS, but still provides a good scalability.

3 Declarative Data Analysis with ROOT RDataFrame

3.1 From Imperative to Declarative: RDataFrame

A ROOT file can contain multiple properties of every collision event, arranging them in a columnar format. A typical ROOT dataset is therefore similar to a table whose column values – the event properties – can vary from floating point numbers to arbitrarily complex objects and nested structures (Fig. 1).

Traditionally, HEP analysis programs have been based on an loop over a ROOT dataset, where the properties of an event are read at every iteration and

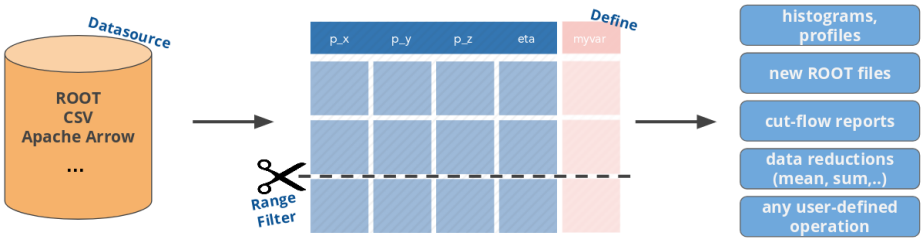

Fig. 2. RDataFrame design integration with the processing of ROOT datasets.

information is aggregated in some way for later analysis (e.g. filling histograms with some physics quantities). This corresponds to a more imperative programming approach, where users explicitly manage the reading and processing of a dataset via interfaces offered by ROOT. Such interfaces provide more control over the whole process but, at the same time, they can make programs more error-prone, since the user deals with lower-level details in longer programs, which can hinder their programming productivity.

More recently, ROOT has proposed a new interface for data analysis, called **RDataFrame** (formerly known as TDataFrame) [13], whose objective is twofold:

- Providing a *high-level interface for data analysis* that overcomes the productivity issues mentioned above, making it simpler for physicists to express their analyses, and to focus on physics rather than on implementation.
- Opening the door to runtime optimisations, such as parallelisation, thanks to a *declarative* expressing the analysis, stating *what* to do but not *how*.

Therefore, in a similar vein to other modern data analysis frameworks such as Apache Spark's DataFrames [19] and Python's data analysis library pandas [15], 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++, although it provides a Python interface too), the integration of just-in-time compilation of user-defined expressions to make analysis definition concise and a tight integration with the rest of the ROOT.

An RDataFrame program basically expresses a set of operations to be applied on a dataset. At runtime, the implementation reads from a columnar data format via a data source, applies the required operations 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). Fig. 2 illustrates this process, while Fig. 3 shows an example of an RDataFrame program.

3.2 Local Parallelisation

The abstraction provided by the RDataFrame programming model paves the way for crucial run time optimisations that can potentially lead physicists to their results faster. Indeed, the future requirements of the LHC, introduced in

```
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
```

Fig. 3. Example of a simple RDataFrame application. The first line, which is optional, enables the implicit parallelisation of the program, as explained in Section 3.2.

Section 1, make it necessary not only to simplify the programming of analyses, but also to exploit the underlying resources in the most efficient way possible.

User-transparent task-based parallelism has been a goal of RDataFrame since its inception, as detailed by [13]. A sequential RDataFrame program can be easily parallelised just by adding one line, as shown in Fig. 3. When the implicit multi-threading mode is activated, RDataFrame concurrently reads chunks of data from the source and spreads the work among multiple threads, which will apply the required operations to their fragment. Moreover, a reduction step to obtain the final results is also performed under the hood at the end.

3.3 Distributed Parallelisation

This paper presents a new python library built on top of ROOT RDataFrame that extends its parallelisation scheme, allowing not only local cores but also a set of distributed resources to be exploited. Moreover, in order to go from local to distributed, no changes are required to the application, since the library offers the same API as the RDataFrame Python interface. The extension to C++ will be investigated in the future.

The modular design of the distributed RDataFrame is presented in Fig. 4, where the application generates a computation graph that reflects the set of operations applied on the input dataset. Underneath, multiple backends can be implemented for either local or distributed execution. Regarding the latter, a Spark backend has been developed to be able to exploit Spark clusters.

When starting an RDataFrame application, the Spark backend inspects the metadata of the input dataset to know about its total number of rows (or entries). This dataset can correspond to one or more ROOT files, which are typically stored on CERN's mass storage system, EOS [16]. With the information about the number of entries and the available resources, the Spark backend creates logical ranges of entries, which represent the partitioning of the data. After that, it launches a map-reduce computation where every mapper task will process a given range of entries. Thus, the mapper code receives the lower and upper boundaries of its range and uses the ROOT I/O libraries to read the entries of that particular range (e.g. remotely from EOS). Then, it applies the RDataFrame

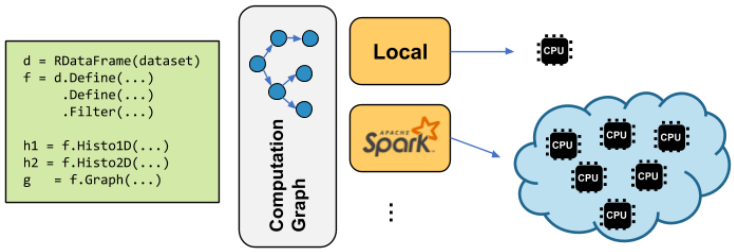

Fig. 4. Applications that use RDataFrame can run both on local and distributed resources thanks to its multiple backends.

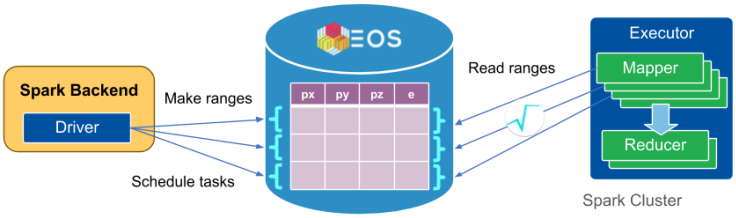

Fig. 5. The Spark backend of RDataFrame launches a map-reduce computation, where mapper tasks read ranges of entries and apply the computation graph to them, before a reduce phase aggregates the final results.

computation graph to its entries and generates partial results, which are finally merged during a reduce phase. This whole process is illustrated in Fig. 5.

It is worth pointing out that, even if the RDataFrame program is written in Python, most of the computation happens in C++, which is the language in which RDataFrame is implemented. The Python interface of ROOT just provides a thin layer on top of its C++ libraries (for I/O, histogramming, etc.).

3.4 Data management

Moreover, our framework does not need any interface between Java and C++ code since the Spark runtime does not manage the input data but instead it is the ROOT C++ library that reads remotely that data. As a result, no reading is involved in the Java layer. On this approach, Spark is only used as a task scheduler: creation of the map-reduce tasks to be run on the remote workers and the coordination the tasks. At runtime, each worker spawns its own Python subprocess, which in turn uses C++ code to run the actual processing.

Furthermore, the implementation of RDataFrame has been optimised to be as efficient as possible when reading the input data in the mapper tasks: only the entries of the assigned range are considered and only the columns that are actually used in the RDataFrame computation graph are read and internally cached.

4 TOTEM Analysis Use Case

The physics analysis we used in the evaluation is the analysis of the elastic scattering data gathered by the TOTEM experiment in 2015 during a special LHC run. The dataset comprises 1153 files totalling 4.7 TB of data in ROOT Ntuple format, and stores 2.8 Billion events representing proton-proton collisions. The choice of TOTEM data was in part motivated by the fact that TOTEM is a relatively small experiment: the test dataset was fully available on EOS (not distributed on the grid). Moreover, a small collaboration facilitated the direct interactions between physicists doing analysis, software engineers and students supporting their work, data administrators granting access rights, and ROOT team developing RDataFrame, as well as CERN IT team responsible for cloud and Spark setup.

4.1 Original Analysis

The original analysis was written using the ROOT framework and includes 2 stages: (1) data reduction, and (2) filtering based on physics cuts. It followed a traditional approach of implementing an imperative processing loop. The first stage is a simple filtering which rejects a majority of input entries. The remaining set is subject to more complex computations. The output is a set of one and two dimensional histograms representing distributions of interesting trends.

4.2 Porting of Analysis to RDataFrame

The conversion from ROOT C++ code to the new RDataFrame interface is a required preparation step before running the analysis on Spark. Unlike [18], the input data for this analysis can keep the original format and still be run on Spark, as the RDataFrame interface delegates every I/O operations to the ROOT internals. While the original code has to be adapted to the new programming paradigm offered by RDataFrame, all operations can be reused since RDataFrame is part of ROOT. Consequently, all C++ headers, data structures and custom functions defined by users can be adopted from the original analysis with minimal changes on the new version. As a result, the migration of the existing analysis to the new interface requires significantly less effort than using e.g. Scala language (native for Spark) where everything needs to be rewritten.

Dealing with C++ code from Python As stated in Section 3.1, RDataFrame exposes a Python interface with support for all methods available in C++. Therefore, every line of the original C++ code can be expressed with an equivalent Python syntax keeping the same functionality. Besides, manipulating C++ code from Python benefits from a simpler and quicker interface while still getting a level of performance closer to C++ than raw Python. These advantages do not come for free as all C++ expressions on the Python interface will be *just-in-time* compiled to C++, leading to a known overhead which may vary depending on

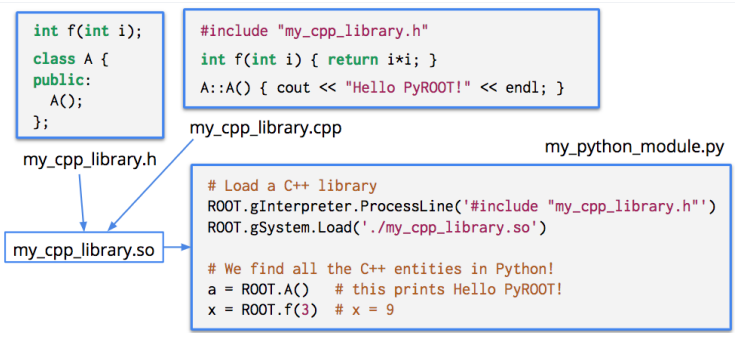

Fig. 6. Dynamic library loading mechanism with ROOT

the use-case. On the other hand, this Python interface lowers the hurdles to run C++ code on Spark clusters. Libraries and C++ headers defined on the original code can be reused as shown in Listing 1.1.

```
1 ROOT.gInterpreter.Declare('#include "common_algorithms.h"')
2 ...
3 f1 = rdf.Filter("!SkipTime(time)")
```

Listing 1.1. Equivalent example of Listing 1.2 with RDataFrame

`SkipTime` is defined in the `common_algorithms.h` which is written in C++. Before starting the computation, the string `!SkipTime(time)` is compiled by the ROOT C++ Interpreter and ready to be called during the execution.

This process can be slightly improved by providing compiled libraries rather than just headers, thereby saving one step to the interpreter. In this regard, all headers ought to be modified to keep only structs and function definitions while real implementations go to a different source file. These two files can be later compiled into a shared library and injected to Python. Fig. 6 illustrates an example of this mechanism.

From an imperative to a declarative model Following a declarative programming model rapidly reduce the verbosity of the code compared to the imperative version where half of the lines are boilerplate. Listings 1.2 and 1.3 show a simplified example of filling a histogram written in both ways. The first listing requires the creation of temporal variables (`rp_L_1_N`, `rp_L_2_N`, `rp_L_2_F`) to store the values read from the dataset (lines 9 - 14) using manual memory assignments. Then, the iteration over all the dataset entries needs to be specified in form of a `for` loop. Finally, a histogram, previously created, is filled with the valid entries, which are filtered by a `IsValidEntry` function. In contrast, the RDataFrame version (Listing 1.3) just specifies the required actions rather than the real implementation, hence it circumvents the necessity of temporal variables, explicit loops and manual memory assignments.

```
// Input data with all events
TChain data = new TChain(TotemData);

// Custom object with Totem data structure
rp_L_1_N = new TotemDataStructure();
rp_L_2_N = new TotemDataStructure();
rp_L_2_F = new TotemDataStructure();

// Read three columns
data->SetBranchStatus("track_rp_5.*", 1);
data->SetBranchAddress("track_rp_5.", &rp_L_1_N);
...
data->SetBranchStatus("track_rp_25.*", 1);
data->SetBranchAddress("track_rp_25.", &rp_L_2_F);

// Loop over all entries
long int ev_index = 0;
for (; ev_index < data->GetEntries() ; ev_index++){
    // Assigns entry values to corresponding custom objects
    data->GetEvent(ev_index);
    if (IsValidEntry(rp_L_1_N, rp_L_1_N, rp_L_2_F...)
        histogram->Fill(rp_L_1_N);
}
```

Listing 1.2. Reading from file, selection of branches and filtering in the original analysis

```
rdf = ROOT.ROOT.RDataFrame(TotemData)
histo = rdf.Filter(IsValidEntry, {"track_rp_5",
                                  "track_rp_21",
                                  "track_rp_26"})
           .Histo1D("track_rp_5");
```

Listing 1.3. Equivalent example of Listing 1.2 with RDataFrame

The original analysis codebase written in ROOT C++ has around 4000 lines of code. Approximately 60% of this code describes the main process of the analysis, so-called the event-loop, while the remaining 40% defines data structures, algorithms and global parameters on headers files.

The code corresponding to the main process was completely rewritten with the new RDataFrame interface, leading to a 76% reduction of the code length, applying similar changes to the ones describes in Listings 1.2 and 1.3. The 40% of the code corresponding to header files can be reused by RDataFrame without any conversion, since it can be loaded by the ROOT C++ Interpreter and used from RDataFrame at runtime.

Besides decreasing the amount of code, local executions of the analysis expressed in RDataFrame Python code on a single core showed that it performs

three times faster than the original version. Although it has not been properly analysed, one possible reason for this difference in performance may be the presence of inefficiencies on the original code that are repeated on every event loop. This demonstrates that the fact of using a high-level interface based on Python does not add any major overhead for this analysis since underneath the real computation runs on C++.

5 Evaluation – Interactive data analysis in the cloud

The main objective of our evaluation was to verify that the proposed RDataFrame framework can handle the workload of real physics data analysis. We used the TOTEM experiment dataset as described in Sec. 4. Our experiments were designed to demonstrate (1) the correctness of obtained results, (2) the scalability of parallel processing along with the increasing number of cores, (3) the interactivity provided by the user interface and the reduced computing times, and (4) the code portability between clusters located at CERN and on the Open Telekom Cloud (T-Systems), the latter being provided exclusively for our experiments thanks to the Helix Nebula initiative [11] (referred to as HNSciCloud).

5.1 Science Box software bundle

To achieve portability and interactivity, we used the Science Box software bundle [7]. As described in [4], the main components are:

- EOS [16], the distributed storage system used to host all physics data at CERN. A dedicated EOS instance hosting a subset of the TOTEM experiment data is deployed on the HNSciCloud.
- SWAN [17], a web-based platform to perform interactive data analysis. It builds on top of Jupyter notebooks by integrating the ROOT analysis framework and the computational power of Spark clusters.
- A dedicated Spark cluster accessible from the SWAN notebook interface.
- CERNBox [14], the synchronization and sharing service for personal and scientific files that uses EOS as storage backend.

All the Science Box services run in containers orchestrated by Kubernetes with the only exception of Spark, which runs on VMs with Cloudera Manager.

5.2 Testbed details

The testbed used for the physics analysis consisted of two independent clusters:

- Analytix, a general purpose Hadoop and Spark cluster located at CERN that is used as a shared resource for various analytics jobs.
- the Spark cluster deployed on the HNSciCloud.

All the performance and scalability experiments were executed on the cluster on the HNSciCloud. It consists of 57 nodes of 32 cores and 128 GiB each, giving a total of 1,824 vCPUs and 7296 GiB of memory. The cluster was equipped with 21.5 TiB of storage, out of which 16.4 TiB were available to EOS (the actual space available is 8.2 TiB due to the `replica 2` layout of stored files). Network connectivity among the different Science Box containers and the Spark cluster was provided by general purpose 10 Gigabit Ethernet interfaces.

5.3 Correctness

One of the most important requirements for physicists was to make sure that the analysis re-implemented in RDataFrame produces the same results as the original one. First, all the output histograms and their main statistics (mean and standard deviation) have been compared with the help of physicists and ROOT experts. Second, a set of scripts was developed to automate the comparison of resulting outputs [9]. These checks confirmed that the results are correct. We also observed that the way ROOT displays the number of entries in weighted histograms may be misleading, since it depends on the number of partitions, but it is a known issue and does not influence the other physics results. For more details, we refer to the report [5].

5.4 Scalability

The main goal of these experiments was to measure the performance and parallel scalability of the analysis running on Spark depending on the number of CPU cores. We conducted multiple experiments in order to identify the bottlenecks and tune the Spark configuration, e.g. the number of partitions, executors per node and worker reuse policy. From these experiments it turned out that the best configuration is to use one Spark executor per node and the number of partitions data is divided into should be equal to the number of cores allocated.

Here we report on the results of the largest run, when we allocated up to 1470 cores total and we varied the number of partitions, which limited the number of cores actually used. In order to measure the effect of Python worker restart policy in Spark, which may affect the performance for larger deployments, we include two series of data contrasting the results obtained when using a fixed number of Python workers that do not `fork()` a process for every task (i.e., `spark.python.worker.reuse = True`) against the ones obtained when forking (i.e., `spark.python.worker.reuse = False`).

The results are shown in Fig. 7. As we can see, the policy of reusing workers gives better results for larger number of cores. The best computing time achieved was 1m43s at 736 cores, while on 16 cores it was 34m15s and on 1 core 8h15m20. It results in a best speedup of about 280x compared to single core execution, but we observe that beyond 800 cores the execution time begins to grow. This shows that we reached the limits of parallelism for this particular analysis application. The scalability limits can be attributed to: (a) overheads during start-up time and work distribution, (b) not ideal load balancing of computing tasks, and

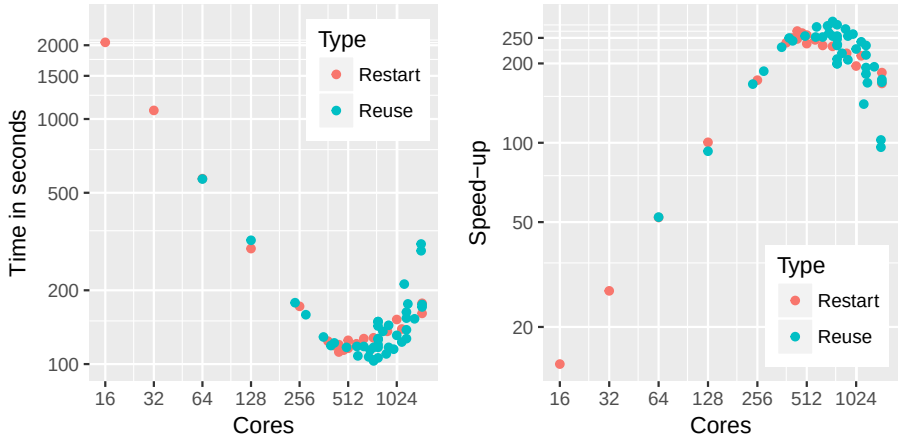

Fig. 7. Execution time and speedup versus the number of CPU cores used for the two configuration options of `spark.python.worker.reuse`.

(c) possible memory leaks in the current implementation of the framework. We expect that two former issues may become less pronounced when dealing with larger data sets, while the latter will be investigated in further development.

5.5 Interactivity

One of the challenges we addressed is to examine whether it is possible to perform an *interactive* data analysis of the full 4.7 TB dataset at a large scale. We consider two relevant aspects: (1) the provisioning of user interfaces to run code and visualize the results interactively, and (2) the reduction of the computing time to consider the waiting time acceptable. According to the collected results, we argue that we have shown interactivity in both aspects. (1) was achieved by using the SWAN interface so that the whole analysis was implemented and executed via Jupyter notebook (see repo: [9]). SWAN together with ScienceBox builds on top of Jupyter, providing such additional features as notebook sharing between users of CERNBox, integration with Spark provides interactive monitoring of job progress directly in the notebook, and direct access to CERN resources (Spark clusters, EOS). (2) was achieved by reducing the computing time below 2 minutes, as discussed in Sec. 5.4. While interactivity is not simple to quantify, we are convinced that our approach is a big step towards interactivity compared to traditional batch processing, which takes hours or even days and requires the use of multiple command-line tools and intermediate copies of data.

5.6 Portability

Implementing the analysis in the RDataFrame model and the components described in Sec. 5.1 allowed us to achieve transparent portability of the code

across two infrastructures, namely the Analytix cluster at CERN and the Spark cluster at HNSciCloud. In particular, the whole dataset was replicated on the EOS instance at HN with the same directory structure, so that the same paths to data files could be used in both clusters. Next, the same version of SWAN was deployed on both clusters and local files could be synchronized via CERNBox. Thanks to this setup, the same Jupyter Notebook [9] could be executed on both clusters. We emphasize that our solution is not specific to the HNSciCloud but it can be used on any public cloud infrastructures or private OpenStack clouds. Specifically, services provided through Science Box can be deployed on any Kubernetes cluster or Kubernetes-based container orchestration engine, while the Spark cluster can be deployed on commodity virtual machines.

6 Conclusions

In this paper, we presented recent developments in the data analysis frameworks for HEP community. The declarative RDataFrame extension to the popular ROOT toolkit allowed us to transparently parallelize the data analysis process using Apache Spark, while still preserving compatibility with the existing ecosystem of tools. Moreover, by the usage of Science Box Tools we were able to perform the whole analysis interactively with modern Web-based notebooks.

We consider the results of our evaluation very promising to the HEP community planning their data analysis for the future experiments expecting higher data volumes. We have shown that the framework works correctly on a real analysis of 4.7 TB of data from the TOTEM experiment, and that thanks to distributed processing we can reduce the computing time to less than 2 minutes. The analysis we used for evaluation was not simplified by any means, and we can consider it as representative for typical data processing in HEP. Other, more complex analysis programs sometimes use external numerical libraries (e.g. for integration, approximation, etc.), but as they are available for use with Python or C++, we foresee no major issues with their integration in our framework.

Our results are important also to other scientific or commercial data analysis applications. We have shown that it is possible to combine efficient widespread High-Energy Physics C++ libraries with a Java- and Python-based Apache Spark platform. Moreover, a combination of open source tools that comprise the Science Box environment can be dynamically deployed in an external cloud, providing additional resources for similar big data science and engineering projects.

Future work includes studies on performance with the larger data sets or the multi-user and multi-application workloads, including comparisons with other solutions described in Sec. 2. We also plan to investigate other emerging frameworks and infrastructures, such as serverless or containerized clouds.

Acknowledgments: This work was supported in part by the Polish Ministry of Science and Higher Education, grant DIR/WK/2018/13.

References

1. Python Dask. http://docs.dask.org/en/latest/why.html
2. Alves, Antonio Augusto, J., et al.: A Roadmap for HEP Software and Computing R&D for the 2020s. Tech. Rep. HSF-CWP-2017-001 (Dec 2017), http://cds.cern.ch/record/2298968
3. Apollinari, G., Bjar Alonso, I., Brning, O., Fessia, P., Lamont, M., Rossi, L., Tavian, 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
4. Avati, V., et al.: Big data tools and cloud services for high energy physics analysis in TOTEM experiment. In: 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion, Zurich, Switzerland, December 17-20, 2018. pp. 5–6 (2018), https://doi.org/10.1109/UCC-Companion.2018.00018
5. Blaszkiewicz, M., Mnich, A.: Interactive data analysis of data from high energy physics experiments using Apache Spark. Tech. rep. (2019), http://cds.cern.ch/record/2655457, BSc Thesis Presented 2019
6. CERN: ROOT a data analysis framework (2018), https://root.cern.ch
7. CERN: Science Box (2018), https://sciencebox.web.cern.ch
8. CERN: The TOTEM Experiment (2018), https://totem.web.cern.ch
9. Cervantes, J.: Rdataframe-totem repository (2018), https://github.com/JavierCVilla/RDataFrame-Totem/
10. Cremonesi, M., et al.: Using big data technologies for hep analysis, https://indico.cern.ch/event/587955/contributions/2937521/attachments/1684310/2707721/chep_bigdata.pdf
11. Gasthuber, M., Meinhard, H., Jones, R.: HNSciCloud - Overview and technical challenges. J. Phys. : Conf. Ser. **898**(5), 052040. 5 p (2017), http://cds.cern.ch/record/2297173
12. Glaser, F., Neukirchen, H., Rings, T., Grabowski, J.: Using mapreduce for high energy physics data analysis. In: 2013 IEEE 16th International Conference on Computational Science and Engineering. pp. 1271–1278 (Dec 2013). https://doi.org/10.1109/CSE.2013.189
13. Guiraud, E., Naumann, A., Piparo, D.: TDataFrame: functional chains for ROOT data analyses (Jan 2017), https://doi.org/10.5281/zenodo.260230
14. Mascetti, L., Labrador, H.G., Lamanna, M., Moscicki, J., Peters, A.: CERNBox + EOS: end-user storage for science. J. Phys.: Conf. Ser. **664**(6), 062037. 6 p (2015)
15. McKinney, W., et al.: Data structures for statistical computing in python. In: Proc. of the 9th Python in Science Conference. vol. 445, pp. 51–56. Austin, TX (2010)
16. Peters, A., Sindrilaru, E., Adde, G.: EOS as the present and future solution for data storage at CERN. J. Phys.: Conf. Ser. **664**(4), 042042. 7 p (2015), http://cds.cern.ch/record/2134573
17. Piparo, D., Tejedor, E., Mato, P., Mascetti, L., Moscicki, J., Lamanna, M.: SWAN: a Service for Interactive Analysis in the Cloud. Future Gener. Comput. Syst. **78**(CERN-OPEN-2016-005), 1071–1078. 17 p (Jun 2016), http://cds.cern.ch/record/2158559
18. Sehrish, S., Kowalkowski, J., Paterno, M.: Spark and HPC for High Energy Physics Data Analyses. In: Proceedings, 31st IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW): Orlando, Florida, USA, May 29-June 2, 2017. pp. 1048–1057 (2017). https://doi.org/10.1109/IPDPSW.2017.112
19. The Apache Software Foundation: Apache Spark (2018), https://spark.apache.org/