
Parallelization and optimization of a High Energy Physics analysis with ROOT's RDataFrame and Spark.

Master thesis



Facultad de Ingeniería Informática
UNIVERSIDAD DE MURCIA

Author

JAVIER CERVANTES VILLANUEVA
javier.cervantes@um.es
javier.cervantes@cern.ch

Directors

JOSÉ TOMÁS PALMA MÉNDEZ
jtpalma@um.es

ENRIC TEJEDOR SAAVEDRA
etejedor@cern.ch

SEPTEMBER 2018



*To my parents and my brother, whose love knows no distance
Thank you for giving me everything*

*And my lovely future wife, Laura
Cause... nothing else matters*

AUTHOR'S DECLARATION

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

SIGNED: DATE: 05/09/2018

CONTENTS

Resumen	xiii
Abstract	xv
1 Introduction and goals	1
1.1 Context and motivation	1
1.2 Project description	5
1.3 Goals	5
1.4 Methodology	6
1.5 Context	7
1.6 Structure of this document	7
2 Environment and technologies used	9
2.1 TOTEM Experiment	9
2.2 ROOT Data Analysis Framework	10
2.3 ROOT Data Storage	11
2.4 RDataFrame, a modern tool to manipulate and analyze ROOT datasets	11
3 Design	17
3.1 TOTEM dataset	17
3.2 TOTEM original code	19
3.3 Encoding analysis problem in Python RDataFrame	20
3.4 Distributed execution	25
4 Results and discussion	29
4.1 Test setup	29
4.2 Local comparison between reference code and RDataFrame version . .	31
4.3 Distributed analysis with Spark	38
4.4 Result validation	42
5 Related work	43
5.1 Programming models for distributed data processing at HEP	43
5.2 Declarative models for parallel execution	44
6 Conclusions and future work	47
6.1 Conclusions	47
6.2 Publications	49
6.3 Future work	49

Bibliography	51
A Code: <code>distill.py</code>	57
B Code: <code>distributions.py</code>	61
C ROOT Workshop presentation	71
D UCC 2018	73

LIST OF TABLES

3.1	Composition of dataset recorded by TOTEM experiment during a LHC run in 2015	17
-----	---	----

LIST OF FIGURES

1.1	Inside view of the ATLAS Detector at the LHC accelerator at CERN. Source: ATLAS Experiment© 2014 CERN	2
1.2	The current schedule for the LHC and HL-LHC upgrade and run [1]	2
1.3	CMS estimated CPU (3a) and disk space (3b) resources required into the HL-LHC era, using the current computing model with parameters projected out for the next 12 years.	3
1.4	Estimated CPU resources (in kHS06 ⁻¹) (1.4(a)) and total disk resources (in PB) (1.4(a)) needed for the years 2018 to 2028 for both data and simulation processing. The blue points are estimates based on the current software performance estimates and using the ATLAS computing model parameters from 2017. The solid line shows the amount of resources expected to be available if a flat funding scenario is assumed, which implies an increase of 20% per year, based on the current technology trends.	4
2.1	Computational graph produced by Listing 2.1 [2]	15
2.2	Schema of parallel distribution in several environments with RDataFrame [3]	16
3.1	Part of the TOTEM dataset composition by branches and leaves	18
3.2	Flowchart of the RDataFrame version of the analysis	23
3.3	Dynamic library loading mechanism with PyROOT	24
4.1	Architecture and deployment of the distributed analysis components	30
4.2	Execution time of the TOTEM analysis for each code version. Equivalent code expressed with the RDataFrame model runs on average 3 times faster.	32
4.3	Execution time of the event loop in distributions for different versions of the RDataFrame code compared to the original code using dataset DS1 as input.	34
4.4	Evolution of the execution time by number of produced histograms using the Python JIT version of RDataFrame	35
4.5	Total execution time with RDataFrame running with different threads.	36
4.6	Scaling of RDataFrame version with multithreading enabled.	37
4.7	Scaling with smaller datasets in Spark. Number of partitions coincides with number of cores for each case.	39
4.8	Execution time of the original code run at CERN batch system compared to the RDataFrame version running in parallel with 64 Spark workers.	40
4.9	Execution time for 4.7TB of data running distributed with RDataFrame and Spark. Number of partitions does not coincide with number of cores for each case.	41

4.10 Scaling with 4.7TB of data up to 1024 partitions.	42
--	----

RESUMEN

Tras una época memorable repleta de grandes descubrimientos, el campo de la física de partículas se enfrenta a un amplio y ambicioso programa con el principal objetivo de extender los límites de nuestro conocimiento acerca del universo que nos rodea. Con el Gran Colisionador de Hadrones, más conocido como LHC, como principal herramienta de investigación, la hoja de ruta para los próximos años presenta grandes retos para una amplia variedad de entornos. Desde el descubrimiento del bosón de Higgs en 2012, multitud de nuevas teorías han surgido basadas en su presencia y en lo que conocemos acerca de su comportamiento hasta el momento. Situado en el CERN, Organización Europea para la Investigación Nuclear, el LHC seguirá siendo el colisionador de protones más potente del mundo aún durante algunos años y la principal prioridad europea es obtener su máximo rendimiento posible. Con vistas a aumentar su potencial, durante la década del 2020 el acelerador será sometido a una gran actualización de *hardware* que le permitirá producir hasta cinco veces más colisiones por encima del valor para el que fue diseñado.

Con la nueva configuración, se prevé que el LHC de Alta Luminosidad (HL-LHC) sea capaz de producir una cantidad de datos 30 veces mayor a la producida hasta el momento. Según estas previsiones, la cantidad de datos a analizar para seguir descubriendo nuevas propiedades del universo superará las decenas de *exabytes* (10^{18} bytes). En este aspecto, nuevos grandes avances serán necesarios, en particular a nivel de computación, para lograr los objetivos establecidos.

Numerosas soluciones han sido desarrolladas dentro de la comunidad de física de partículas a fin de procesar las enormes cantidades de datos generadas por los distintos experimentos. No obstante, en los últimos años la producción de este volumen de datos ha pasado a ser más habitual en otros campos, lo que ha dado lugar a nuevos avances por parte de la industria como la erupción de tecnologías *Cloud* y *Big Data*. Con potentes entidades dedicando cuantiosos esfuerzos a la investigación del procesamiento eficiente de datos y el desarrollo de innovadoras infraestructuras que lo soporten, surge una clara necesidad de analizar estos nuevos paradigmas como posibles alternativas a las soluciones usadas hasta ahora en el dominio de análisis de datos físicos.

Esta tesis se desarrolla en el marco de una colaboración entre varios departamentos del CERN con el objetivo de proporcionar una plataforma en la nube destinada al análisis de datos interactivo que sirva a los científicos como punto de entrada a tecnologías innovadoras en el procesamiento de datos distribuidos, como Spark. En este contexto, el trabajo presentado utiliza los servicios ofrecidos por dicha plataforma para reproducir un análisis real de 4.7TB de datos recogidos por el experimento TOTEM en el CERN. Este experimento es uno de los siete situados a lo largo del acelerador LHC y sus investigaciones se centran en el estudio de las colisiones entre protones a fin de comprender mejor la

estructura de esta partícula subatómica. Los detectores del experimento TOTEM son altamente precisos y su diseño permite medir el ángulo de desviación de los protones tras su colisión a más de 150 metros del punto de interacción. La información recogida por estos detectores es filtrada y almacenada en el centro de datos del CERN. Posteriormente, estos datos son analizados por los científicos con ROOT, un librería C++ de análisis de datos especializada en el estudio de partículas de alta energía (*High-Energy-Physics*, HEP).

Esta librería constituye el núcleo de la gran mayoría de análisis hechos en el campo de la física de partículas durante los últimos 20. Los propios gráficos que presentaron la evidencia científica de la existencia del bosón de Higgs fueron desarrollados con ROOT. A pesar de su éxito en el campo, sus desarrolladores siguen trabajando para satisfacer nuevas necesidades y ofrecer un mejor rendimiento que repercuta en el resultado de los estudios llevados a cabo por los distintos experimentos físicos. En busca de nuevos modelos capaces de hacer frente a la enorme cantidad de datos que se espera producir en los próximos años, ROOT está dedicando grandes esfuerzos al desarrollo de herramientas que sean eficientes a la hora de gestionar estos datos y fáciles de usar por programadores con distintos niveles de experiencia. Como resultado, recientes versiones de la librería incluyen una nueva interfaz denominada RDataFrame. Con ella, ROOT ofrece una interfaz de alto nivel fácil de usar pero con optimizaciones a bajo nivel que permiten que los usuarios saquen el máximo rendimiento de sus máquinas.

El presente trabajo comienza a partir de un análisis del experimento TOTEM basado en la antigua interfaz de ROOT. La primera parte del documento se centra en la reimplementación del código del análisis con la nueva interfaz. Por primera vez, se describe en detalle el proceso de conversión de un análisis con este volumen de datos al nuevo modelo de programación ofrecido por RDataFrame. Mientras que la interfaz antigua de ROOT sigue un modelo imperativo de programación, RDataFrame está diseñado para ofrecer un modelo declarativo donde el usuario puede centrarse en definir el *qué* de su análisis sin preocuparse tanto por el *cómo*. En esta parte del documento, se comparan distintas secciones del código implementando la misma funcionalidad con ambas interfaces, destacando las principales diferencias y ventajas.

Entre las novedades de este trabajo se encuentra el uso de Python como principal lenguaje a la hora de definir el código del análisis. Aunque el código original está implementado en C++, ROOT ofrece mecanismos para ejecutar código C++ desde Python. De este modo, toda su funcionalidad es accesible desde el lenguaje Python. Parte de este estudio considera el impacto de un lenguaje interpretado como Python en el rendimiento de la aplicación.

En la segunda parte, se realiza un estudio de rendimiento comparando principalmente las dos versiones del código. Este estudio se estructura de manera que se consideran distintos escenarios. En primer lugar, se ofrece un resumen de los resultados obtenidos al ejecutar el mismo análisis con la nueva interfaz en una máquina local. Entre las características de la nueva interfaz, destaca la habilidad de ejecutar el código en paralelo sin requerir cambios en el código por parte del usuario. Así pues, se estudia cómo evoluciona el tiempo de ejecución del análisis conforme aumenta el número de procesadores usados en paralelo. Esta primera fase del análisis en una máquina local es importante dado que nos permite conocer cuáles son las limitaciones de la interfaz en un entorno controlado. De esta forma, se excluyen factores externos que puedan agregar incertidumbre a los resultados. Como resultado de someter RDataFrame a un volumen de datos muy superior al usado por otros análisis previos, se han descubierto ineficiencias en los patrones de lectura de los datos, produciendo un deterioro significativo en el

rendimiento. Estos resultados han sido reportados al equipo de desarrolladores y su resolución tendrá un efecto positivo para la comunidad de experimentos en el LHC.

Tras obtener resultados acerca del rendimiento del nuevo código en un entorno local, se desarrollan pruebas en un ambiente distribuido basado en Spark. La conexión entre RDataFrame y un clúster basado en Spark da lugar a la presentación de PyRDF, una librería python desarrollada en el contexto del programa *Google Summer of Code*. Esta iniciativa de Google promueve la integración de estudiantes universitarios en el mundo del Código Abierto. Durante los tres meses de verano, Google subvenciona estudiantes que trabajen en proyectos de código abierto en colaboración con distintas organizaciones de todo el mundo. El autor del presente trabajo ha participado en la supervisión del estudiante encargado de desarrollar PyRDF. Esta librería ofrece una interfaz para implementar distintos entornos de ejecución distribuida de modo que RDataFrame pueda ser ejecutado en ellos. En consecuencia, análisis basados en la interfaz de RDataFrame pueden ejecutarse en paralelo en distintos nodos, los cuales retornan resultados parciales del análisis a la máquina local, donde se reducen a un único resultado final.

Con la implementación del adaptador entre RDataFrame y Spark, se da paso al estudio de rendimiento del código corriendo en varios nodos a la vez. Primero se presenta un pequeño análisis de los posibles recursos a utilizar en el clúster de Spark para establecer una configuración común a todas las pruebas. Un primer estudio muestra los tiempos de ejecución para un conjunto reducido de datos así como su ganancia con respecto al número de particiones consideradas. Estos tiempos se comparan con los obtenidos durante la ejecución local, ya que en ambos casos el código corre en paralelo. Para finalizar las pruebas en distribuido con Spark, se analizan los 4.7TB de datos con distintas particiones y se estudia cómo se evoluciona para un mayor volumen de datos y particiones. Los resultados muestran que el uso combinado de RDataFrame junto con Spark, permite reducir el tiempo de ejecución del análisis varios órdenes de magnitud. Esta comparación toma como referencia el código original y los métodos de ejecución habituales en sistemas procesamiento por lotes (*batch systems*). Considerando ambos escenarios, el tiempo de procesamiento se reduce de más 24 horas a unos 7 minutos, produciendo los mismos resultados.

Como conclusión, esta tesis aporta varios resultados de interés tanto para la comunidad de usuarios de ROOT como para sus desarrolladores. Además, otros experimentos pueden beneficiarse de los análisis presentados con RDataFrame en distintos entornos. En general, se muestra un estudio detallado del rendimiento al ejecutar RDataFrame en distintos entornos. A nivel de aplicación, se describen las distintas opciones ofrecidas por la interfaz para optimizar el código tanto desde C++ como desde Python. Se presentan también distintos escenarios de ejecución de RDataFrame en entornos locales y distribuidos, haciendo uso del paralelismo implícito o en varios nodos respectivamente. Para la ejecución en distribuido, se aporta además una nueva librería que extiende la funcionalidad de RDataFrame. Estos resultados han sido puestos en conocimiento de los desarrolladores de ROOT y han sido gratamente valorados.

ABSTRACT

After a remarkable era plenty of great discoveries, particle physics has an ambitious and broad experimental program aiming to expand the limits of our knowledge about the nature of our universe. The roadmap for the coming decades is full of big challenges with the Large Hadron Collider (LHC) at the forefront of scientific research. The discovery of the Higgs boson in 2012 is a major milestone in the history of physics that has inspired many new theories based on its presence. Located at CERN, the European Organization for Nuclear Research, the LHC will remain the most powerful accelerator in the world for years to come. In order to extend its discovery potential, a major hardware upgrade will take place in the 2020s to increase the number of collisions produced by a factor of five beyond its design value.

The upgraded version of the LHC, called High-Luminosity LHC (HL-LHC) is expected to produce some 30 times more data than the LHC has currently produced. As the total amount of LHC data already collected is close to an exabyte (10^{18} bytes), the foreseen evolution in hardware technology will not be enough to face the challenge of processing those increasing volumes of data. Therefore, software will have to cover that gap: there is a need for tools to easily express physics analyses in a high-level way, as well as to automatically parallelize those analyses on new parallel and distributed infrastructures.

The High Energy Physics (HEP) community has developed specialized solutions for processing experiments data over decades. However, HEP data sizes are becoming more common in other fields. Recent breakthroughs in Big Data and Cloud platforms inquire whether such technologies could be applied to the domain of physics data analysis.

This thesis is carried out in the context of a collaboration between different CERN departments with the aim of providing web-based interactive services as the entry-point for scientists to cutting-edge distributed data processing frameworks such as Spark. In such context, this thesis aims to exploit the aforementioned services to run a real analysis of the CERN TOTEM experiment on 4.7 TB of data. In addition, this thesis explores the benefits of a new high-level programming model for physics analysis, called RDataFrame, by translating the original code of the TOTEM analysis to use RDataFrame.

Following sections describe for the first time the detailed process of translating a data analysis of this magnitude to the programming model offered by RDataFrame. Moreover, we compare the performance between both codes and provide results gathered from local and distributed analyses. Results are promising and show how the processing time of the dataset can be reduced by multiple order of magnitude with the new analysis model.

INTRODUCTION AND GOALS

1.1 Context and motivation

CERN, the European Organization for Nuclear Research, is one of the world's largest and most emblematic centres for scientific research. Physicists and engineers from a diverse range of fields look for a better understanding of the structure of the universe. To achieve this, they use the world's largest and most complex scientific instruments to study the basic constituents of matter – the fundamental particles. The particles are made to collide together at close to the speed of light. This process gives the physicists clues about how the particles interact, and provides insights into the fundamental laws of nature [4].

The instruments managed at CERN are purpose-built particle accelerators and detectors. Accelerators boost beams¹ of particles to high energies before the beams are made to collide with each other or with stationary targets. Detectors observe and record the results of these collisions. At present, CERN hosts the two biggest devices ever constructed for particle studies, the ATLAS detector (Figure 1.1) and the Large Hadron Collider (LHC) [5], the largest particle accelerator ever built thus far. The LHC consists of a 27-kilometre ring of superconducting magnets with a number of accelerating structures to boost the energy of the particles along the way [5] before make them collide.

After a remarkable era with the LHC at the forefront of attempts to discover the fundamental nature of our universe, the roadmap of particle physics is full of big challenges for the upcoming years. In 2012, researchers announced the discovery of the Higgs boson, the last missing piece in physicists' standard model of fundamental particles and forces. The finding of the Higgs boson was chosen as the Breakthrough of the Year by Science journal [6] and became a major milestone in the history of physics.

Being the current most powerful accelerator in the world, the LHC has the potential to go on and help to shed light on some of the unknown questions of the age: the existence of supersymmetry; the nature of dark matter; the existence of extra dimensions [7]. The impact of Higgs discovery goes beyond the culmination of a long quest, it marks indeed the beginning of a new period in particle physics. Analyses on the Higgs boson's behaviour might help to better understand questions about other known and unknown particle properties. However, most of the hypothesis require a larger production of Higgs bosons to support the statistical results of the experiments. To extend its discovery potential, the LHC will need a major upgrade in the 2020s in order to increase the total

¹The particle stream produced by an accelerator usually clustered in bunches.

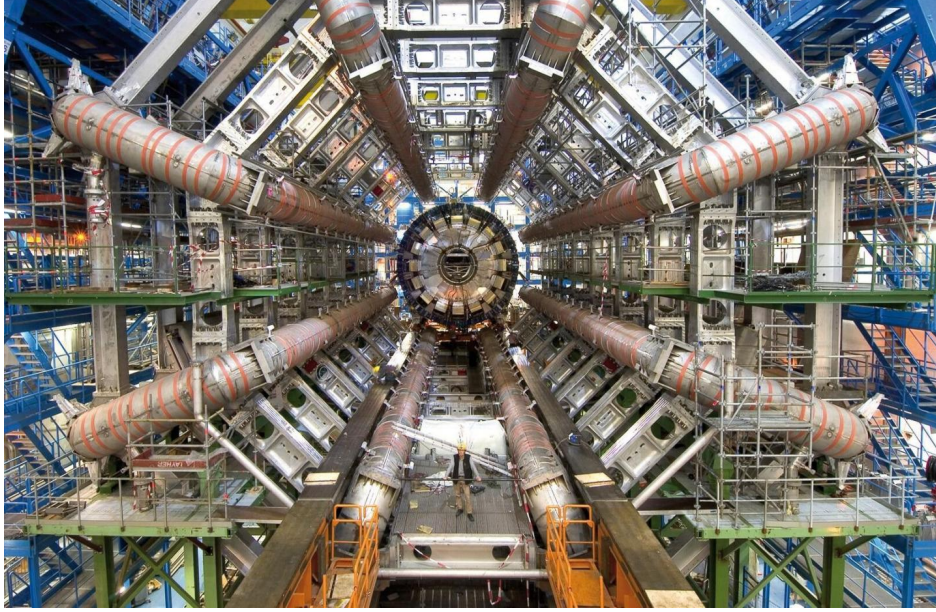


Figure 1.1: Inside view of the ATLAS Detector at the LHC accelerator at CERN. Source: ATLAS Experiment© 2014 CERN

number of collisions by a factor of ten beyond its design value. A more powerful LHC would provide more accurate measurements of new particles and enable observation of rare processes that occur below the current sensitivity level. How this should be done is at the heart of the novel machine configuration, the High Luminosity LHC (HL-LHC) [1, 8]. The long shutdowns, LS2 and LS3, will be used to upgrade both the accelerator and the detector hardware as illustrated on Figure 1.2.

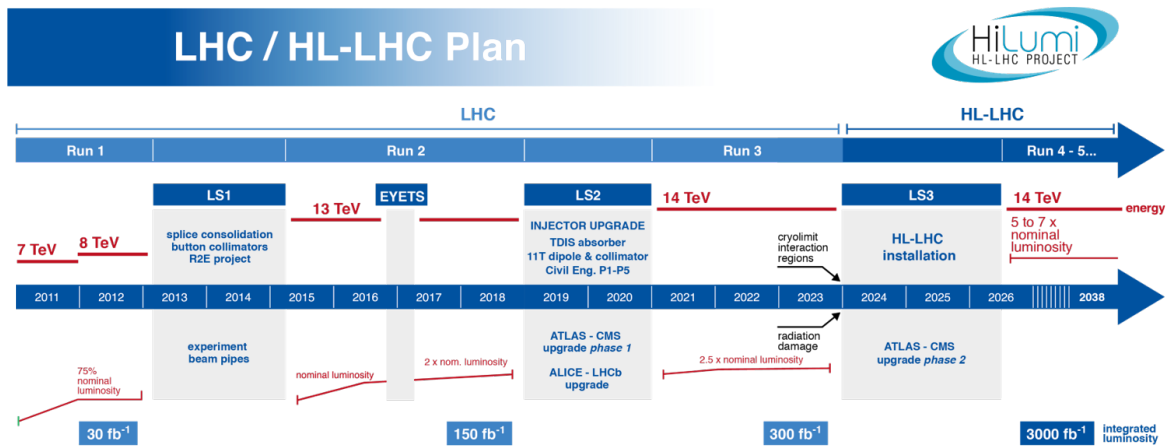


Figure 1.2: The current schedule for the LHC and HL-LHC upgrade and run [1]

Current total amount of data produced at the LHC already presents a substantial processing challenge. During a run, LHC detectors witness particle collisions at a rate of approximately 10^9 times per second. The amount of raw data per event (collision) is around one megabyte, hence about one petabyte of collision data is generated per second.

$$10^9 \text{ collisions/s} \times 1^6 \text{ bytes/collision} = 10^{15} \text{ bytes/s} = 1 \text{ Petabyte/s}$$

Nonetheless, this is several orders of magnitude greater than what present detectors data acquisition system can even handle. Such quantities of data is filtered by the experiments, running sophisticated algorithms to reduce the number of events and select only those considered interesting for analysis purposes. The trigger system of each experiment carries these algorithms out to reject the undesired events. As a result of this process, the filtered LHC data are then aggregated in the CERN Data Centre (DC), where a copy is archived into long-term tape storage. By the end of June 2017, the CERN DC had already passed a data storage milestone with a total of 200 petabytes of data permanently archived on tape. The velocity at which data volume is produced at the LHC experiments grows at an almost exponential rate. As a matter of fact only during October 2017 about 12.3 petabytes of data were stored in the data center.

Data-wise predictions for the post-LHC age reveal a strong necessity of collaboration in multiple areas outside physics to accomplish these challenges. Already in Run 3 the CERN experiment LHCb [9] will process more than 40 times the number of collisions that it does today (about 10 million collisions every second). For the HL-LHC these numbers go even further. The updated accelerator, scheduled to begin taking data in 2026 (Figure 1.2), is planned to collect by only ATLAS [10] and CMS [11] some 30 times more data than the LHC has currently produced. As the total amount of LHC data already collected is close to an exabyte, it is clear that the problems to be solved require approaches beyond simply scaling current solutions, assuming Moore’s Law. An approximated number of needed resources to manage this data can be estimated from an extrapolation of the Run 2 computing model and is shown in Figures 1.3 and 1.4.

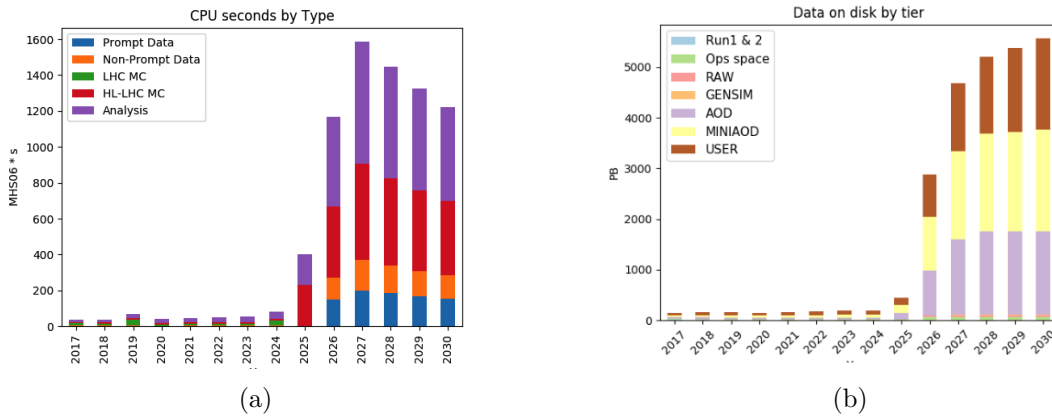


Figure 1.3: CMS estimated CPU (3a) and disk space (3b) resources required into the HL-LHC era, using the current computing model with parameters projected out for the next 12 years.

Both experiments will encounter a step change from 2026 on where just increasing resources will not be enough due to budget limitations. The amount of data to be collected and processed will be limited by affordable software and computing, so the

²The units of CPU are kilo-HEP-Spec06 [kHS06], which is a measure of power more appropriate to HEP than TFlops

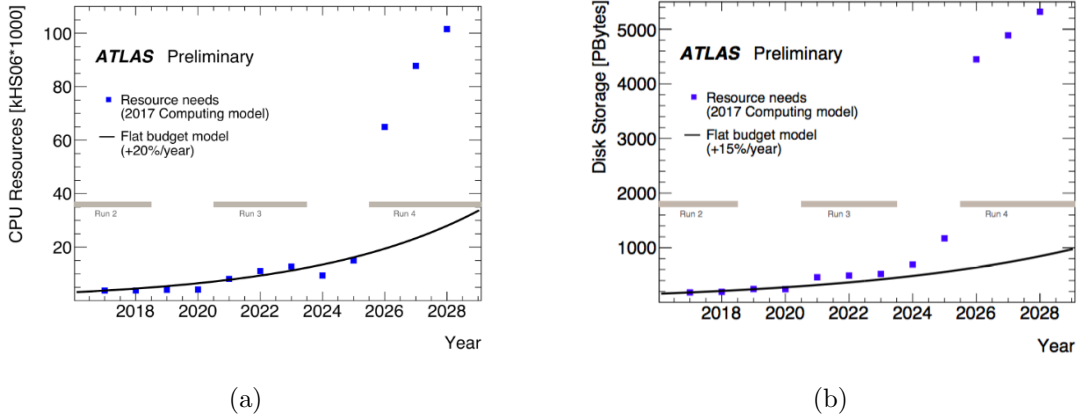


Figure 1.4: Estimated CPU resources (in kHS06^2) (1.4(a)) and total disk resources (in PB) (1.4(b)) needed for the years 2018 to 2028 for both data and simulation processing. The blue points are estimates based on the current software performance estimates and using the ATLAS computing model parameters from 2017. The solid line shows the amount of resources expected to be available if a flat funding scenario is assumed, which implies an increase of 20% per year, based on the current technology trends.

physics reach during this period will be limited by how efficiently these resources can be used.

Over the last 15 years, thousands of physicists, computer professionals and students with a wide range of skills and abilities have contributed writing more than 20 millions lines of code for the 4 largest LHC experiments. In spite of the fact hardware is rapidly evolving with new paradigms and architectures, most of the current software was written with one single architecture in mind and following a sequential processing model. Consequently, squeezing the most out of computing resources becomes an arduous task. In this regard, the High Energy Physics (HEP) community is aware of the risks this may lead and it is making continuous efforts to raise awareness that software upgrades have to happen in parallel with hardware improvements [12].

Such advances in software must be aligned with the volume of data expected to be produced but also with that fact that software is written by many people in collaborations, with varying levels of expertise. The former calls for taking advantage of new architecture and programming models to increase the ability of the code to deliver results efficiently. The latter sets out both a technical and a social challenge. Current and future developed and deployed software ought to be sustainable for future and upgraded experiments. At the same time, heterogeneous platforms have become a more commonplace: from the many-cores architectures for distributed computing to numerous alternatives such as GPUs for new machine learning techniques. Although it is inevitable that some software developments will remain within the scope of a very concrete platform, providing software tools to make the most of the underlying hardware will be critical to achieving success in the future.

Establishing common libraries and projects that will provide generic solutions to the community is one of the main messages that [12] wants to spread across the community. It is on these projects to push software evolution without degrading user-friendly interfaces and programming models:

- Progresses on performance ought to be transparent for users while easy to use

with minimal or no changes in user code.

- Software tools must provide a high-level abstraction to facilitate its use, without sacrificing performance.
- Taking advantage of local and distributed parallelism will play a crucial role to succeed.

The present work goes in the line of the aforementioned points. On the following sections we focus on one of the most recent developments inside the ROOT Data analysis framework, RDataFrame. This new interface aims to lower the hurdle to write efficient and effective code able to run in parallel while putting special attention on the programming model offered to the user. This study therefore sets out to assess the process of converting a real analysis code from a CERN experiment to RDataFrame and its performance running within multiple environments.

1.2 Project description

This thesis focuses on the parallelization and optimization of a real High Energy Physics analysis using ROOT's RDataFrame interface to operate with the physics data in combination with Spark as the backend framework for the distributed execution. An efficient usage of all the resources available both on local user machines and shared computing-clusters will be crucial to overcome the ambitious experimental programme planned for the coming years in High Energy Physics. In preparation for this post-LHC era, the ROOT Team at CERN has recently put a lot of efforts to develop this promising RDataFrame library which offers a high level interface for analyses of data stored on ROOT format or other data formats.

The baseline of this project is a real LHC analysis, performing an in-depth study of the proton structure with the TOTEM detectors. The original code of the TOTEM experiment will be rewritten in order to adapt it to the RDataFrame programming model, showing the differences offered by the new interface with respect to the classical style of expressing HEP analysis over the last 20 years. Resultant code is required to produce an equivalent outcome of the physics analysis, otherwise any further research on performance will be irrelevant.

Once verified the correctness of the results, different environments will be considered for the efficiency studies including local and distributed machines. On this topic, we will put special attention on the number of needed changes to go from one environment to another.

1.3 Goals

As it has been advertised by the HEP community, it is necessary to look into new approaches to take the most out of the available resources without compromising the user experience. The goals of this work go along the same lines and can be depicted as follows:

- Investigate how a real HEP analysis code can be expressed in a high-level declarative programming model like RDataFrame, and what are the benefits this brings in terms of programming productivity.

- Test the performance and scalability of the new RDataFrame analysis code, when exploiting both local (thread-level) and distributed parallelism, in particular with Spark.

1.4 Methodology

TOTEM's reference analysis was developed using the ROOT Data Analysis Framework around three years ago. During this time, the analysis framework has been quickly evolving including new features and functionality with more than forty official releases. On this thesis, one of the most recent and powerful developments will be considered, RDataFrame together with its integration in Spark.

In the first place, the original ROOT C++ code will be rewritten into ROOT's RDataFrame which follows a completely different programming model. Main differences between both paradigms will be also presented comparing various non-technical aspects such length of the code, readability or interfaces.

As one of the main purposes is to run the analysis distributed through Spark, the connection between RDataFrame and the latter will be also tackled on the current work. Although the ROOT Framework is mainly written in C++, it is highly integrated with other languages such as Python. This feature will be exploited to connect Spark and RDataFrame.

Once the RDataFrame version of the analysis code is ready, the first priority will be to ensure that both versions of the code provide the same results. Since the amount of data produced during the different stages of the analysis is substantially large, proper automatic tools should be provided to guarantee the exactness of the results. Throughout the analysis, a wide range of output results with different nature is produced: ROOT files (binaries), one-dimensional histograms, two-dimensional histograms, profile histograms, canvas combining multiple plots and graphs. For the sake of robustness, all of them should be considered for the aforementioned tools.

Despite its relatively short time in the ROOT releases, RDataFrame has been warmly welcomed by the community and put into practice in several scenarios from different experiments. This work aims to evaluate some other aspects of RDataFrame not tested in detail yet:

- Performance on a large amount of data, $\mathcal{O}(TB)$
- Parallelize the execution of the RDataFrame analysis on a cluster of nodes by using Spark, a tool for distributed big data processing
- Full conversion of an existing analysis based on C++ code to the Python interface of RDataFrame
- Cost of using Python as the main interface

Behind these goals, a broad variety of technologies is implied so a painstaking performance analysis will be required in order to thoroughly understand the outcome. On top of it, different scenarios will be analysed:

- Sequential: running in one single core with one thread
- Multithreading: running multiple threads on the same machine

- Distributed: running on a cluster backed by Spark

Taking the original code as a reference, all these cases will be measured and compared. Subsequently, possible scalability problems or bottlenecks should be considered for study.

Before efficiently running distributed in a big cluster, it is vital to ensure an adequate use of the local resources. During the single-threaded performance analysis, two different execution modes offered by ROOT will be considered: statically compiled and *just-in-time* (dynamically) compiled. The latter is possible thanks to Cling [13], the ROOT C++ interpreter.

All mentioned analyses will solely focus on modifications on the new code. Optimizations of the reference code itself are not considered, since our goal is not to optimize the current code but to understand how new interfaces can provide a better efficiency no matter the user code. Physics results will be contemplated during the performance tests as a validation check to ensure the correctness, however from the point of view of the physics analysis any conclusions nor results will be presented on this work.

1.5 Context

This thesis is the result of a collaboration between the TOTEM Experiment and the EP-SFT group [14] at CERN, where the author has been working for the last two years: firstly undertaking a Technical Student programme during 14 months, and secondly with a Fellowship contract as a Software Engineer in the Future Circular Collider (FCC) experiment [15] since November 2017.

1.6 Structure of this document

The structure of this document is as follows: Chapter 2 describes the organizational scope where this work has been developed and presents the technologies that lay down the foundations of physics data analysis. On this same chapter recent innovative models are also introduced as they are the baseline where our analyses rely on. Chapter 3 details the composition of the dataset used for this study, describes the process of re-writing the reference code to the new model and introduces the available strategies to run it in parallel. Chapter 4 presents the performance analysis between the two code versions considering both a local and a distributed scenario. Chapter 5 goes through related works considering technologies in use inside and outside CERN. Finally, Chapter 6 provides conclusions and a description of future directions of our work.

2.1 TOTEM Experiment

A total of seven experiments at the LHC use detectors to analyse the myriad of particles produced by collisions in the accelerator. These experiments are run by collaborations of scientists from institutes all over the world. Each experiment is distinct, and characterized by its detectors [16]. These experiments are located on different points throughout the LHC ring. Listed in descending order of size, the LHC experiments are:

- ATLAS: A Toroidal LHC Apparatus
- CMS: Compact Muon Solenoid
- ALICE: A Large Ion Collider Experiment
- LHCb: LHC-beauty
- TOTEM: Total Cross Section, Elastic Scattering and Diffractions Dissociation
- LHCf: LHC-forward
- MoEDAL: Monopole and Exotics Detector At the LHC

The TOTEM experiment [17], small in size compared to the others at the LHC, is dedicated to the precise measurement of the proton-proton interaction cross section¹, as well as to the in-depth study of the proton structure which is still poorly understood.

TOTEM's physics programme is focused on studying elastic scattering with large momentum transfers partly in cooperation with CMS as both experiments are located at the same interaction point, IP5. Hence the TOTEM collaboration aims at physics complementary to the programmes of the general-purpose experiments at the LHC. The experiment had to invest heavily in the design of detectors that will be capable of meeting the challenge of triggering and recording events in the very forward region. To perform such measurements, TOTEM requires a good acceptance for particles produced at very small angles with respect to the beam. Both sides of the interaction point are covered with advanced detectors, so-called Roman Pots. These detectors, placed at

¹Cross section is a measurement of the probability that an event occurs.

about 147 m and 220 m from the interaction point, have been designed to detect leading protons at merely a few mm from the beam centre [18].

The work presented on this thesis takes as a reference analysis code run during a real TOTEM study with a large amount of data gathered by the mentioned Roman Pots detectors. The current analysis is still on progress and has not been published yet. Therefore any results can be shown on this work, albeit an analysis of a different dataset obtained in similar conditions has been published on [19].

2.2 ROOT Data Analysis Framework

ROOT [20] is a modular scientific software framework for large scale data analysis and the de-facto standard tool to do data processing in HEP. It provides all the functionalities needed to deal with big data processing, statistical analysis, visualisation and storage. It is mainly written in C++ but integrated with other languages such as Python and R. Over the past years, the HEP community has developed and gravitated around an analysis ecosystem centered on this toolkit [21]. According to the latest estimates, ROOT is used to store more than 120 petabytes of data, which makes it a central pillar in high-energy physics. Given that it was designed to operate with great amounts of data and it is also widely used outside the HEP field [13].

Through a number of foundation and utility class libraries, for storing, analysing, processing and visualizing, ROOT can be used for developing a whole HEP analysis. Thousands of successful publications support ROOT as an integrated and validated toolkit which is able to cover the full event processing chain. Although the different experiments use tons of diverse hardware and software combinations, the majority agree on having ROOT present at the core of their analysis. In this sense, ROOT has enabled the communication using a common analysis language, making easy to leverage improvements between experiments [12].

Since its early development, started in 1995, ROOT has included some of the features that nowadays are considered state of the art [22]:

- Efficient object data store scaling from KB's to PB's
- Storage of C++ classes into ROOT files within a machine-independent compressed binary format
- Columnar data storage techniques
- Optimizations for statistical data analysis over very large data sets
- Complex data modeling and fitting
- Multivariate classification methods
- Extensive 2D+3D scientific data visualization capabilities
- Parallelization multiprocess and multithreading

One of the most powerful and key features of ROOT is Cling, an interactive C++ interpreter, built on top of Clang and LLVM compiler infrastructure. Cling brings the possibility to do Rapid application development (RAD), where software prototypes and proof-of-concept applications can be quickly developed. Prototyping is an effective

way to gain understanding of the requirements, reduce the complexity of the problem and provide an early validation of the system design and implementation. However, it depends on the time consumed by edit-run cycles during development. Cling offers users a way to develop their applications rapidly and effortlessly. Its advantages over the standard interpreters are that it has command line prompt and uses just-in-time (JIT) compiler for compilation.

Despite its longevity, ROOT keeps evolving and adding new features to the toolkit. Its ongoing program of work addresses important new requirements, in both functionality and performance, in order to be ready for the ever-increasing amount of data generated by the new accelerators. In this regard, recent releases have shown strong efforts to be aligned with the challenges mentioned in Section 1, by promoting a new programming model easy to use yet still powerful: RDataFrame.

2.3 ROOT Data Storage

ROOT provides system-independent binary files in which users can store C++ objects of any class as well as other formats such as SQL databases or XML files [23]. Information stored in a ROOT file can be organized in several subfolders where users can navigate as they were browsing a file system. Every single object produced during an analysis can be saved into a ROOT file such as histograms, data structures or functions. Event data that fills these objects are also stored in ROOT files, normally organized in columnar data, the so called ROOT *trees*. Each tree structure is composed by a list of branches. At the same time, each branch has its own definition and internal organization. A branch may contain leaves, other nested branches or a combination of both. Leaves describe the individual elements of a branch. Since a branch can be matched with any object structure, branches of the same tree may vary on their format and be arbitrarily complex. This is possible since the ROOT file format supports unstructured data that does not fit neatly into rows and columns.

2.4 RDataFrame, a modern tool to manipulate and analyze ROOT datasets

RDataFrame [24] is a modern C++ high-level interface for interacting with data in ROOT. The interface design is inspired by other dataframe APIs, such as pandas [25] or Spark DataFrames, and takes ideas from the functional and declarative programming paradigms [26]. RDataFrame's development is strongly guided by the following goals aligned with HEP future challenges as well as data science industry:

- Simple programming model
- Expose modern and elegant interfaces, easy to use correctly and hard to use incorrectly
- Allow to transparently benefit from parallelism
- Being the fastest way to manipulate HEP data
- Being the go-to ROOT analysis interface from 1 to 100 cores (laptop to cluster)

- Full support for and consistent interfaces in both Python and C++

Thanks to the internal machinery, users can focus on their analysis as a sequence of operations to be performed on the C++ data-frame object, while the framework takes care of the management of the loop over entries as well as low-level details such as I/O and parallelisation. `RDataFrame` provides methods to perform most common operations required by ROOT analyses; at the same time, users can just as easily specify custom code that will be executed in the event loop.

2.4.1 ROOT and RDataFrame code comparison

The following listings show an example of analyses based on the old ROOT interfaces and their respective translation to `RDataFrame`. First five lines of code in Listing 2.1 get a dataset from a `treename` tree saved inside a file called `filename`. They also initialize three variables `a`, `b`, `c` that will point to attributes stored in the tree. With `reader` starts the so-called *event-loop*, this loop will iterate over all the elements (events) of the tree. This particular analysis will apply the `DoStuff` operation to values `a`, `b` and `c` of a given event if they fulfil the condition checked by `IsGoodEvent`.

Same analysis implemented in `RDataFrame` is shown in Listing 2.2. In this case, `RDataFrame` interface offers a compact one-liner method to create a dataframe including handling of the file, access to the tree (dataset) and treatment of the pointers to the list of desired attributes. Next, the `Filter` method, which must contain a valid C++ expression, is applied to `a`, `b` and `c` for each event, being those elements from the `A`, `B` and `C` columns, respectively. Data in `RDataFrame` flows through the chain of calls, being transformed, filtered and finally used to perform actions. This example ends applying the method `DoStuff` to each event that passed the filter.

```
1 TFile f(filename);
2 TTreeReader reader("treename", &f);
3 TTreeReaderValue<A_t> a(reader, "A");
4 TTreeReaderValue<B_t> b(reader, "B");
5 TTreeReaderValue<C_t> c(reader, "C");
6 while(reader.Next()) {
7     if(IsGoodEvent(a, b, c))
8         DoStuff(a, b, c);
9 }
```

Listing 2.1: Reading and filtering events with ROOT

```
1 ROOT::RDataFrame d("treename", filename, {"A", "B", "C"});
2 d.Filter(IsGoodEvent, {"A", "B", "C"}).Foreach(DoStuff);
```

Listing 2.2: Reading and filtering events with RDataFrame

As we can see, the new interface not only reduces the amount of boilerplate but also avoids users to implement common and repetitive tasks. Reading ten attributes rather than three will lead to ten definitions of `TTreeReaderValue`'s values pointing to attributes of the dataset, while the main part of the analysis (filter and actions) would keep the same. Moreover, the interface still provides full control over the analysis.

2.4.2 RDataFrame features overview

Additionally, moving from *pure* ROOT to `RDataFrame` brings some powerful features for free. User-wise, the only remarkable cost is the change from an imperative programming

model to a **declarative** one. Next example is helpful to better illustrate this, together with the rest of features. Listings 2.3 and 2.4 contain the code to plot a one-dimensional histogram filled with the momentum (**pt**) of those even particles whose energy (**E**) is greater than 100.

```

1  TFile f(filename); // Open file with one or more datasets
2  TTreeReader tree("treename", &f); // Point `tree` to `treename` tree
3  TTreeReaderArray<double> px(tree, "px"); // Each event contains an array of px,
4  TTreeReaderArray<double> py(tree, "py"); // an array of py
5  TTreeReaderArray<double> E(tree, "E"); // and an array of E
6  // Each event contain particles, whose x, y position
7  // and Energy is define by px, py and E, respectively
8
9  TH1F h("pt", "pt", 16, 0, 4); // Initialize a histogram of pt against pt (momentum)
10 // with 16 bins from 0 to 4
11
12 while (tree.Next()) { // Event loop
13     for (auto i = 0; i < px.GetSize(); ++i){ // For each event, iterate all particles
14         if (E[i] > 100) // Checks if the particle energy is greater than 100
15             h.Fill( sqrt(px[i] * px[i] + py[i] * py[i]) // Calculate pt of a particle and add it to the Histogram
16         }
17     }
18 h.Draw(); // Plot filled histogram

```

Listing 2.3: ROOT code to read a file, create a custom variable and plot a histogram out of it

```

1  ROOT::EnableImplicitMT();
2  ROOT::RDataFrame d("treename", filename);
3  d.Define("good_pt", "sqrt(px[i] * px[i] + py[i] * py[i])[E>100]")
4  .Histo1D({"pt", "pt", 16, 0, 4}, "good_pt")->Draw()

```

Listing 2.4: RDataFrame code to read a file, create a custom column and plot a histogram out of it

Besides a significant gain in compactness, the RDataFrame interface provides a more convenient way of writing the same analysis as well as friendlier to read. Code in Listing 2.3 defines the analysis in a imperative way:

- Need to know concrete type of the columns read from dataset.
- Explicitly define the operations (*the what*) and its implementation (*the how*).
- Run an explicit double loop, for each event particle does a check.
- Deal with different iterators according to the type of data (**Next** iterator for events, **for** loop for particles).
- Manually introduce each element of the histogram.

On the other hand, the RDataFrame version in Listing 2.4 is featured by:

- **Declarative model**, user can focus on the what and internals of RDataFrame decide the implementation (functional chaining helps greatly).
- New property **pt** is calculated, stored as a new column (**good_pt**) and filtered in one-go (**[E > 100]**, **vectorised operations** on collections *ala* numpy or pandas).
- **Histogram** is created with the result of the previous defined. **pt** value of the selected particles is directly filled into the histogram.

- No type definition, **type safety** is handled underneath by inference.
- **Define** operation in line 3, makes use of **JITted code** to simplify. The string must contain a valid C++ expression which is interpreted by Cling and applied to the corresponding data. It might also receive a C++ lambda function thereby reducing the conversion time from string-ed code to a real C++ expression.
- `ROOT::EnableImplicitMT()` in line 1, activates **implicit parallelism**. If any number is passed as a parameter, code will run with as many threads as cores available in the machine.
- **Lazy execution** guarantees that all operations are performed in one event loop.

2.4.3 Computational graph

Internally, `RDataFrame` creates a direct acyclic graph composed with the operations defined by the user. There are two kind of operations:

- Transformation (lazy): a way to manipulate data being filtering or defining new columns in the dataset.
- Actions: produce a results that can be synchronized

```
1 ROOT::RDataFrame df(dataset);
2 auto df2 = df.Filter("x > 0")
3     .Define("r2", "xx + yy");
4 auto rHist = df2.Histo1D("r2");
5 df2.Snapshot("newtree", "newfile.root");
```

Listing 2.5: `RDataFrame` code to read a dataset, filter values from a column, create a custom column with the result of the previous filter and write result on disk

Code shown in Listing 2.5 produces the computational graph presented in Figure 2.1. Method calls add new elements to the graph. This structure is key to provide some of the most outstanding features of `RDataFrame` such us laziness or cached results. As we will see in future sections this graph allows the distributed execution with different backends.

2.4.4 Going from C++ to Python

`RDataFrame` has been designed to support an easy integration of both languages Python and C++. Therefore moving from one to the other requires minimal changes. The following pieces of code contain the same logic expressed in three different ways (Listing 2.6, 2.7 and 2.8). This code discards from a given dataset `d` those rows whose *theta* value is smaller than 0 and writes on disk a `f.root` file that contains only the `pt_x` values of the filtered rows.

```
1 d.Filter([](double t) { return t > 0.; }, {"theta"})
2 .Snapshot<vector<float>>("t", "f.root", {"pt_x"});
```

Listing 2.6: `RDataFrame` with C++ interface

First code is pure C++ and defines the **Filter** with a lambda function. In this case all types are explicitly defined. Hence the execution of this code will be the fastest one among these examples.

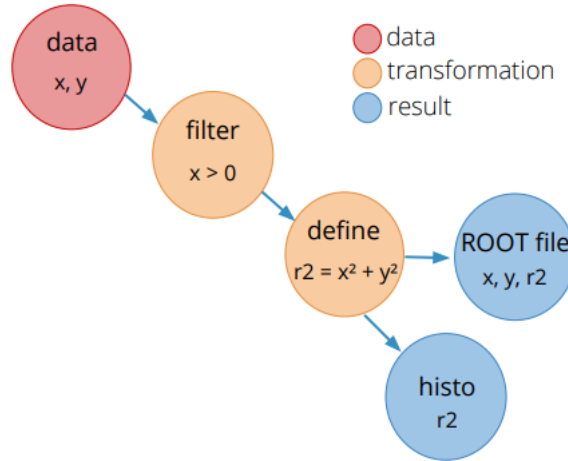


Figure 2.1: Computational graph produced by Listing 2.1 [2]

```
1 d.Filter("theta > 0").Snapshot("t", "f.root", "pt_x");
```

Listing 2.7: RDataFrame with C++ interface and jitting code

The second piece of code makes use of C++ code together with Cling’s just-in-time compilation. This is still C++ code whose `Filter` is defined with a simple string which will be compiled at runtime. This version is less verbose however its execution implies type inference and calls to the interpreter thus it also takes slightly more time.

```
1 d.Filter("theta > 0").Snapshot("t", "f.root", "pt_x")
```

Listing 2.8: RDataFrame with Python interface with jitting code

As for the last code in spite of being exactly the same as the second one it runs in Python. Through PyROOT [27] all Python binding are automatically generated so calls to ROOT/C++ methods can be executed together with pure Python code.

2.4.5 Parallel data processing

2.4.5.1 Implicit parallelism

Over the last years, ROOT has evolved to ease the expression of parallelism for the community. As a result both shared-memory and distributed-memory environments has been tackled [28]. Different types of parallelism are currently supported by ROOT:

- Multi-threading
- Multi-processing
- Cluster-wide executions

From the programming model point of view, two useful categories to describe parallelism can be identified: explicit parallelism and implicit parallelism. The former gives full control on the expression of parallelism to the user, so it’s targeted for those with a strong knowledge on parallelism. On the other hand, the latter only requires the user to adopt certain high-level interfaces and data treatment hiding all internal details.

In order to activate the implicit parallelism within a program, a single, global switch is necessary. The following line of code in Listing 2.9 is the only change required in user code to read, decompress and deserialize branches of a dataset in parallel. In addition, `RDataFrame` has been completely written with this design in mind, consequently most of the actions supported by the interface take advantage of these low-level optimization completely transparently for users.

```
1  ROOT::EnableImplicitMT()
```

Listing 2.9: Single line required to enable implicit parallelism in `RDataFrame`

2.4.5.2 Distributed parallelism

Parallelism on many nodes is also possible through `RDataFrame` thanks to the its internal computational graph. As illustrated in Figure 2.2 the processing of the graph can be implemented in different backends. In this regard, a local environment can be treated as a backend. For each distributed technology an intermediate layer is required to connect both interfaces, so the distribution task is transparent for the users. On top of that the adapter layer allows the distributions with minimal or even no changes in the analysis code.

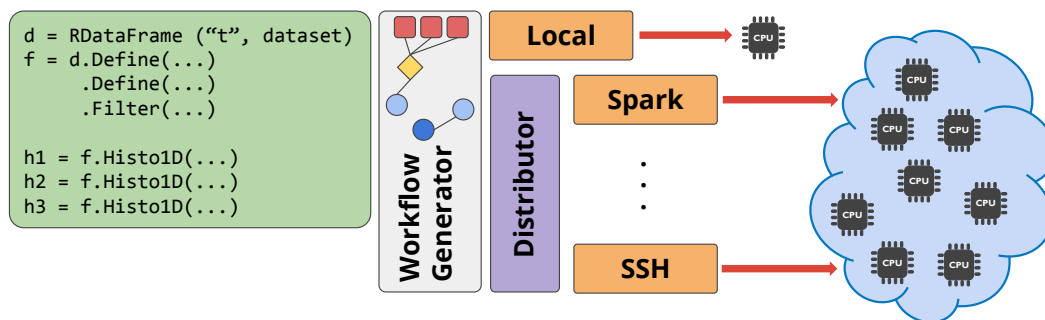


Figure 2.2: Schema of parallel distribution in several environments with `RDataFrame` [3]

3

CHAPTER DESIGN

The following sections presents the process of converting the reference analysis code to RDataFrame. On section 3.1 we describe the composition of the dataset and its structure. Section 3.2 provides an overview of the original code which is split in two stages: data reduction and production of the histograms. Section 3.3 introduces the main techniques applied to re-implement the code into the RDataFrame declarative model and describes possible code optimizations to be considered. Finally, Section 3.4 focuses on the distributed execution presenting two complementary approaches that allow RDataFrame code to run in parallel on different machines.

3.1 TOTEM dataset

The original TOTEM analysis data¹ was gathered in 2015 during a special LHC fill with optics parameter² β^* adjusted to 90 m [19]. Further details of this special optics configuration are described in [29–32]. Collected data correspond to properties of charged and scattered particles coming from proton-proton collisions.

The dataset comprises 1153 files of total 4.7 TB of data in ROOT format, and stores 2.8 Billion events. The full dataset was split in seven different data sample, whose concrete numbers are summarized in Table 3.1. Each data sample is different from others and contains unique collision data. The size of the data sample tells us the total magnitude that will be analysed while the number of events is directly related to the number of iterations carried out for the algorithms. All this data is distributed in different files which contain part of the data sample. The size of each file varies between 1.5GB or 4.7GB depending on the dataset.

Data sample	DS1	DS2	DS3	DS4	DS5	DS6	DS7
Size (GB)	91	185	750	377,5	1850	550	750
Events (Million)	53	113	444	233	1162	328	483
Files	59	77	177	130	446	127	175

Table 3.1: Composition of dataset recorded by TOTEM experiment during a LHC run in 2015

All samples belong to the same experiment and were collected in similar conditions, however each data sample may contain events with different particularities, visible

¹This data is not public yet.

² β^* means amplitude function, which is the term to express the beam size.

during the analysis and reflected on the results. For this thesis, the smallest dataset (DS1) will be used to ensure the exactness of the translated code, comparing results between both analysis versions. Performance-wise bigger datasets will be considered as well as a combination of all them, even though the physical meaning of the results might not be precise since some modifications in the original code would be required.

In terms of attributes, the dataset follows a complex tree structure composed by main branches, nested subbranches and leaves. Figure 3.1 shows a partial representation of the dataset composition. From the figure below, it can be seen that the ROOT format does not match with the typical concept of dataframe offered by other API's such as R or pandas.

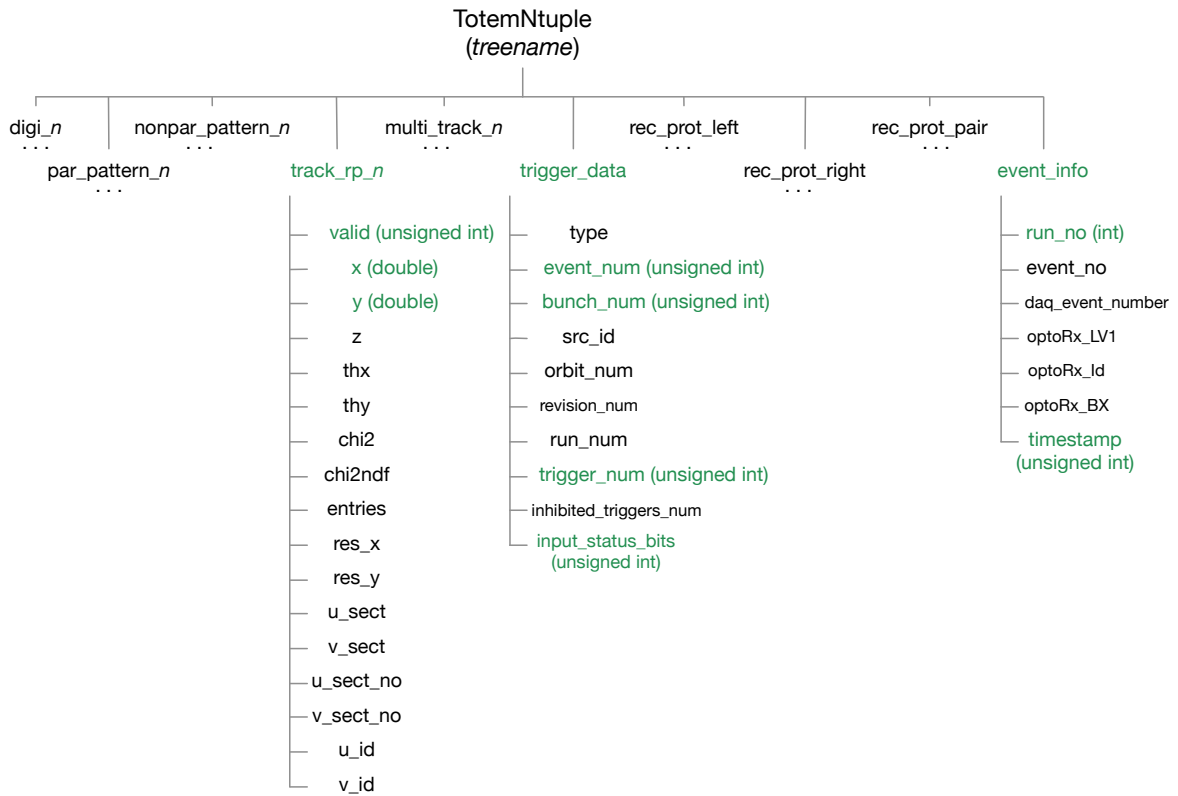


Figure 3.1: Part of the TOTEM dataset composition by branches and leaves

First five branches shown in Figure 3.1 contains a suffix n that codifies the detector that recorded those events, where n takes the following range of values:

[0, 1, 2, 3, 4, 5, 20, 21, 22, 23, 24, 25, 100, 101, 102, 103, 104, 105, 120, 121, 122, 123, 124, 125]

Each of these main branches contains subbranches and leaves. A comparison between this dataset and the mentioned concept of a dataframe could be drawn, considering a flatten version of this tree, where every leaf would be a column. In that case, the dataset would count with a total of 1608 columns. For the sake of space, only the most relevant attributes for this analysis are shown in Figure 3.1. Namely, only those green coloured will be used in these analyses, which means 78 attributes, a 4.85% of the total.

3.2 TOTEM original code

Original code [33] is written in C++ using the ROOT Framework. It is comprised by a set of headers defining:

- Main data structures, mostly structs:
 - **EventRed**: an event (collision) composed by a **HitData** and some metadata information such as **timestamp**.
 - **HitData**: Values **x**, **y** and **valid** shown in Figure 3.1 from six different **track_rp_n** match to elements of this struct.
 - **Kinematics**: defines a complex movement of the particles.
 - **CutData**: contains information to filter events.
 - **Analysis**: contains configuration of the analysis.
 - **Environment**: contains information about the experiment conditions.
- Algorithms:
 - Reconstruction (from raw x and y positions creates a *HitData* struct)
 - Definition of the bin positions in the plot axes.
 - Calculus of corrections.
 - Alignment of the events.

As for the analysis logic, two applications are involved for each stage of the analysis:

- Data reduction - **distill.cc**:
 - Select leaves (attributes) to read from disk.
 - Apply first filter to remove invalid data, thereby reducing extensively the amount of event to be processed in the subsequence steps.
 - Create a new ROOT file out of the reduced data (input for the next part).
 - Since the execution time is quite long ($\mathcal{O}(\text{hours})$), it is meant to be run once.
- Filtering and histogram production - **distributions.cc**:
 - Read branches from the ROOT file created by **distill.cc**.
 - Apply multiple filters at different stages of the event loop.
 - Produce a variety of plots: one-dimensional and two-dimensional histograms (TH1D, TH2D), graphics object (TGraphs) and profile histograms (TProfile).
 - Save all results to a ROOT file, i.e. the aforementioned plots.
 - Execution time of this part is way shorter, hence it can be run multiple times while tuning the analysis parameters defined in the headers.

3.3 Encoding analysis problem in Python RDataFrame

The conversion step from ROOT C++ code to the RDataFrame interface is essential to run the analysis on Spark. In some related works such as [34], firstly authors have to convert the input data from ROOT format to HDF5 format and secondly they reimplement the analysis code from ROOT C++ to Spark operations. Unlike that work, the input data for this analysis can keep the same format, since RDataFrame is just a new interface built as part of ROOT using the ROOT internals. Regarding the analysis code, even though a conversion is also required for this case, only the interface is modified which indeed is part of the same framework as the original code. One of the main advantages of this approach is that all data structures and logic defined in libraries for this analysis can be also used in the new version with minimal changes.

3.3.1 RDataFrame conversion

As previously stated, the RDataFrame version of the analysis employs the Python interface so that going to a distributed execution with Spark is straightforward. In addition, manipulating C++ code from Python benefits from a simpler and quicker interface while still getting a level of performance which is closer to C++ than raw Python. However, these advantages do not come for free since all C++ expressions in form of string will be *just-in-time* compiled to C++, leading to a known overhead which may vary depending on the use-case. Next sections will analyse and consider this aspect.

3.3.1.1 From an imperative to a declarative model

The original code follows an imperative programming model where almost half of the lines are boilerplate. This section presents some of the key aspects to translate the starting code to RDataFrame. Due to the extension of the code only the most representative parts will be considered. A full version of the resultant RDataFrame code with comments referring to the equivalent original one is available in [35] and Appendixes A, B.

Select and link input branches (distill)

Let the variable `ch` be a reference to the collection of files containing the input dataset, within the imperative version every branch needs to be marked in order to be read (being 1 to be processed and 0 not to). In order to access to their values, a variable of the correct data type has to point to the address in memory (Listing 3.1).

```
1 TChain ch = new TChain("TotemNtuple");
2 printf(">> input_files\n");
3 for (unsigned int i = 0; i < input_files.size(); i++)
4 {
5     ch->Add(input_files[i].c_str());
6     printf("%s\n", input_files[i].c_str());
7 }
8
9 // select and link input branches
10 ch->SetBranchStatus("", 0); // Mark every branch to Not Process
11
12 EventMetaData metaData = new EventMetaData(); // Declare data structure
```

```

13 ch->SetBranchStatus("event_info.", 1); // Mark to Process
14 ch->SetBranchAddress("event_info.", &metaData); // Link variable and address
15
16 ...
17
18 RPRootDumpTrackInfo rp_L_1_F = new RPRootDumpTrackInfo();
19 ...
20 ch->SetBranchStatus("track_rp_5.", 1);
21 ch->SetBranchAddress("track_rp_5.", &rp_L_1_F);

```

Listing 3.1: Select and link input branches in distill.cc, original code

Assuming the ROOT python module already imported, Listing 3.2 shows the equivalent RDataFrame version:

```

1 treename= "TotemNtuple"
2 rdf = ROOT.ROOT.RDataFrame(treename, input_files)

```

Listing 3.2: Select and link input branches in distill.py, with RDataFrame

The ROOT.ROOT.RDataFrame method receives the name of the tree (line 1 in Listing 3.1) and the list of files to read (note that original code explicitly adds all of them). The rest of the code shown in Listing 3.1 is not needed since RDataFrame will lazily read and process branches on demand.

Filter, output branches definition and creation of new file (distill)

As we have already seen in Listing 2.1, branches of a tree stored on disk can be read by assigning its memory address to a variable already defined. In the following code, in order to access to the values inside the branch `track_rp_5`, the variable `rp_L_1_F` has been created. The type of this variable is `RPRootDumpTrackInfo` which is a struct that fits with the structure of the branch. In line 2, branch and variable are linked so that values of the branch can be accessed. After that, a file and a tree are created to store the results at the end of the loop. Branches of this new tree `outT` are defined in lines 10 to 12, and linked to the memory address of the `ev.h.L_1_F.{v,x,y}` variables. Inside the event loop (line 14 to 39) the variable `ev` is assigned to values of the current event. In terms of other language dataframes, this loop would be iterating over the rows of a dataset and the variables `ev.h.L_1_F.v`, `ev.h.L_1_F.x` and `ev.h.L_1_F.y` would be assigned to the elements of those rows. Later, a filter checks whether at least two out of three variables are `True` for each group (where `L` stands for *left* and `R` stands for *right*, these letters provide information about the detector that gathered the collision information). Finally, line 38 flushes the current values of the variables to disk.

```

1 RPRootDumpTrackInfo rp_L_1_F = new RPRootDumpTrackInfo();
2 ch->SetBranchAddress("track_rp_5.", &rp_L_1_F);
3 ...
4
5 TFile f_out = TFile::Open(fn_out.c_str(), "recreate");
6
7 // set up output tree
8 EventRed ev;
9 TTree outT = new TTree("distilled", "bla");
10
11 outT->Branch("v_L_1_F", &ev.h.L_1_F.v);
12 outT->Branch("x_L_1_F", &ev.h.L_1_F.x);
13 outT->Branch("y_L_1_F", &ev.h.L_1_F.y);
14
15 for (; evi < ch->GetEntries() && !interrupt_loop; evi++)
16 {
17     ch->GetEvent(evi);

```

```

18
19 ev.h.L_1_F.v = rp_L_1_F->valid;
20 ev.h.L_1_F.x = rp_L_1_F->x;
21 ev.h.L_1_F.y = rp_L_1_F->y;
22 ...
23
24 // Filter
25 unsigned N_L = 0;
26 if (ev.h.L_1_F.v) N_L++;
27 if (ev.h.L_2_N.v) N_L++;
28 if (ev.h.L_2_F.v) N_L++;
29
30 unsigned N_R = 0;
31 if (ev.h.R_1_F.v) N_R++;
32 if (ev.h.R_2_N.v) N_R++;
33 if (ev.h.R_2_F.v) N_R++;
34
35 bool save = (N_L >= 2 && N_R >= 2);
36 if (!save)
37     continue;
38
39 outT->Fill(); // Event is filled into the file
40 }

```

Listing 3.3: Filtering, custom column definition and file creation in original code

Same logic written in RDataFrame is not only less verbose but easier to understand. In this case, the filter is expressed in form of a string and the new custom columns are defined directly referring to the column read from disk (`track_rp_5`) so there is not need to create temporary variables to match with the structure on disk. Another big difference is the way to write data on disk, RDataFrame uses the method `Snapshot` for this purpose that receives the name of the tree, the name of the file to write and the list of columns to write.

```

1 branchList = ["v_L_1_F", "x_L_1_F", "y_L_1_F", ... ]
2 ...
3
4 filter_code = ""
5 (track_rp_5.valid + track_rp_21.valid + track_rp_25.valid) >= 2 &&
6 track_rp_104.valid + track_rp_120.valid + track_rp_124.valid >= 2
7 ""
8
9 r = rdf.Filter(filter_code) \
10     .Define("v_L_1_F", track_rp_5.valid) \
11     .Define("x_L_1_F", track_rp_5.x) \
12     .Define("y_L_1_F", track_rp_5.y) \
13     ...
14     .Snapshot(outTreeName, outFileName, outbranchlist)

```

Listing 3.4: Filtering, custom column definition and file creation with RDataFrame

3.3.1.2 Workflow

Flow chart in Figure 3.2 illustrates the sequence of operations executed for each part of the analysis. Both flows are expressed in terms of RDataFrame. For `distill` only a big `Filter` is applied before defining the new columns, which indeed boils down to rename the structs stored on disk. More sophisticated is the corresponding flow in `distributions`, on the right side of the figure. A total of four filters are applied throughout the program. Rows that do not fulfil the condition are discard for the next step in the diagram, and the equivalent for `distill`. From some operations two lines connect to different actions, this means that the result is used to either produced a histogram (`Histo1D` or `Histo2D`) or used as an input for the next new column. After reaching *Stop*, all histograms are saved to disk.

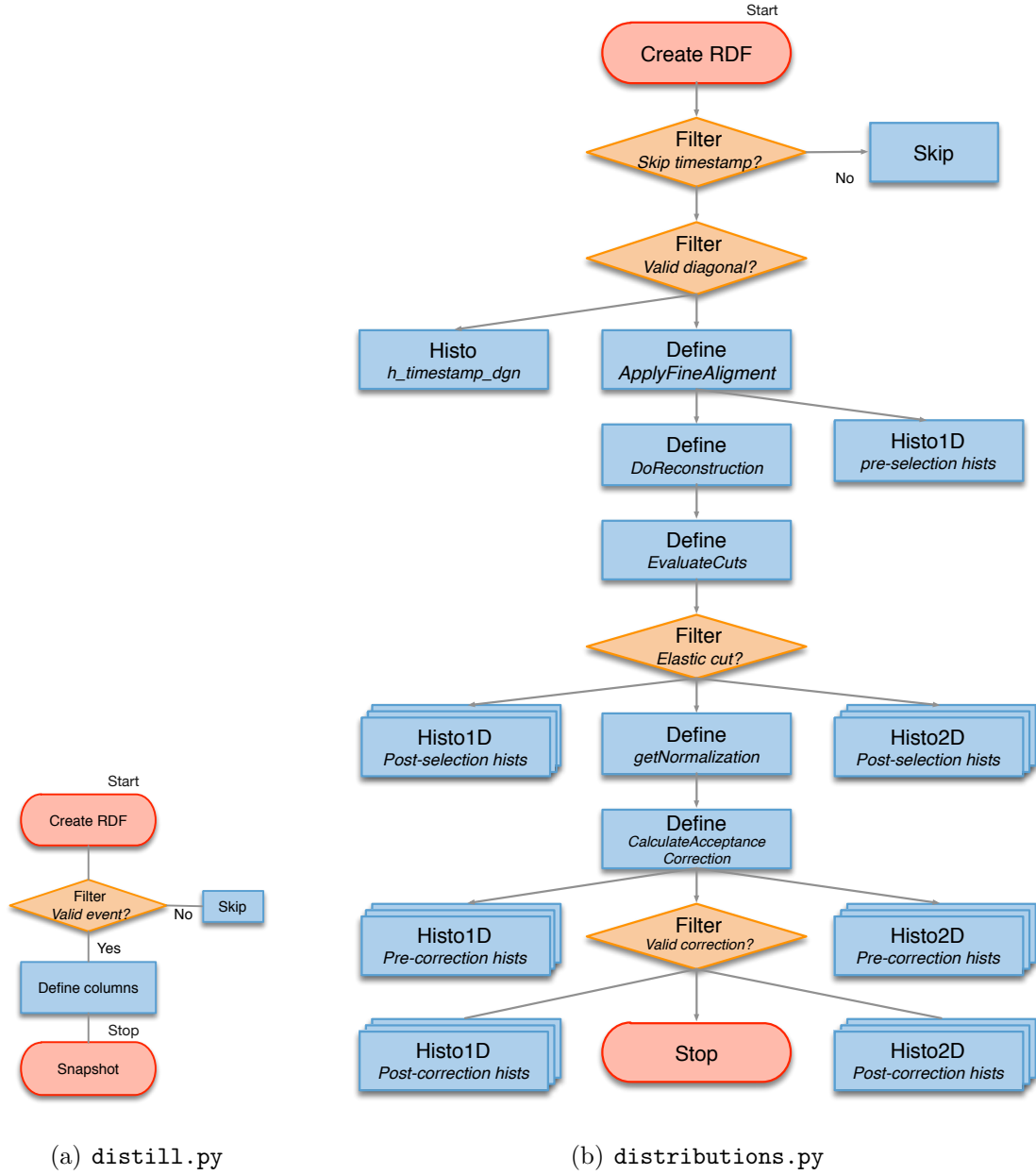


Figure 3.2: Flowchart of the RDataFrame version of the analysis

3.3.2 Optimizations

3.3.2.1 Shared library for jitted code

Current RDataFrame Python code invokes C++ code through jitting mechanism. For instance, the following Listing 3.5 contains a `Filter` whose C++ expression is passed in form of a string.

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

Listing 3.5: Jitting code with RDataFrame

Function `SkipTime` is defined in the `common_algorithms.h` header which is purely written in C++. The cost of calling C++ code in such a way is some offset before starting the internal event loop: the string `! SkipTime( timestamp )` goes to Cling which returns a C++ lambda function with an explicit definition of the involved types. Alternatively stated, it gets compiled and ready to be called during the event loop.

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. Figure 3.3 illustrates an example of this mechanism.

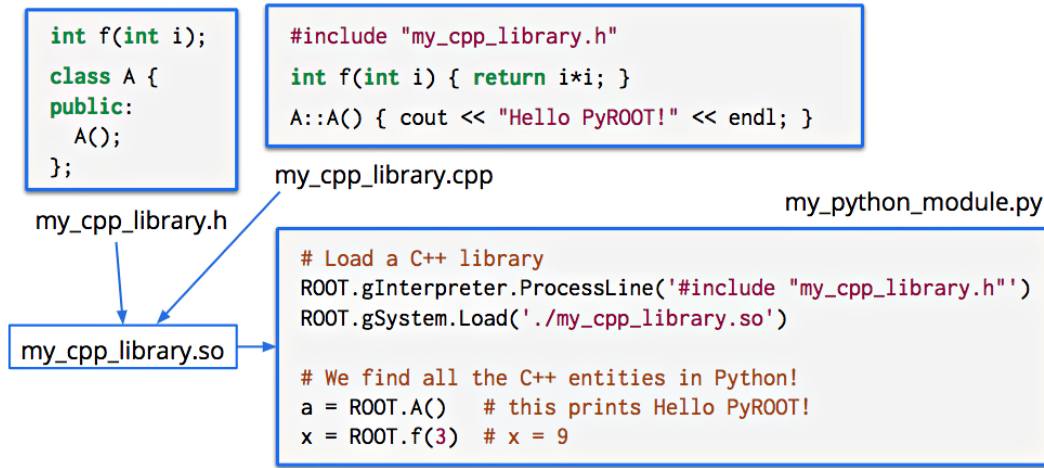


Figure 3.3: Dynamic library loading mechanism with PyROOT

3.3.2.2 RDataFrame C++

On this project, Python is required in order to be able to run the analysis code using Spark. Nonetheless, it is important to fully understand what is the best configuration in a local execution before going distributed. Although previous real experimental analyses have been also ported to RDataFrame, one of the novelties of this work is the use of Python interface as the main language. Therefore, it is worth to compare the performance of the same analysis using two different languages, in pursuance of possible bottlenecks when only using Python as the main interface.

Bright side is that going from Python to C++ is straightforward since the API does not change. Listings 3.6 and 3.7 show both versions of the same code. Besides differences implicit in the language such as type declaration or syntax, the code is fairly similar.

```

1 auto f1 = rdf.Filter("! SkipTime( timestamp )", "check time — selected");
2 // Diagonal cut (L831)
3 auto f2 = f1.Filter("v_L_2_F && v_L_2_N && v_R_2_F && v_R_2_N", "allDiagonalRPs");
4
5 auto model = ROOT::RDF::TH1DModel("h_timestamp_dgn", ";timestamp;rate (Hz)",
6   int(timestamp_bins), timestamp_min-0.5, timestamp_max+0.5);
7 auto h_timestamp_dgn = f2.Histo1D(model, "timestamp");
8 ...
9 auto r3 = r2.Define("kinematics", "DoReconstruction( h_al )");
10   .Define("k_th_x_R", "kinematics.th_x_R")
11   .Define("k_th_y_R", "kinematics.th_y_R")
12   .Define("k_th_x_L", "kinematics.th_x_L")

```

Listing 3.6: RDataFrame code in Python to create a histogram and define some columns

```

1 f1 = rdf.Filter("! SkipTime( timestamp )", 'check time - selected')
2
3 # Diagonal cut (L831)
4 f2 = f1.Filter("v_L_2_F && v_L_2_N && v_R_2_F && v_R_2_N", 'allDiagonalRPs')
5
6 model = ("h_timestamp_dgn", ";timestamp;rate (Hz)",
7         int(ROOT.timestamp_bins), ROOT.timestamp_min-0.5, ROOT.timestamp_max+0.5)
8 h_timestamp_dgn = f2.Histo1D(model, "timestamp")
9 ...
10 r3 = r2.Define("kinematics", 'DoReconstruction( h_al )')
11     .Define("k_th_x_R",      "kinematics.th_x_R") \
12     .Define("k_th_y_R",      "kinematics.th_y_R") \
13     .Define("k_th_x_L",      "kinematics.th_x_L") \

```

Listing 3.7: RDataFrame code in C++ to create a histogram and define some columns

From C++ is easier to optimize the code for two reasons: First, the interpreter layer can be removed so the mentioned overhead (compiling time and extra virtual codes) is eliminated. Considering the heavy use of jitted code in the RDataFrame version, this should have a visible impact on the performance and we measure it in Section 4.2.1. Second, all parts of the code can be fully expressed in C++, consequently the code can be previously compiled getting profit of the various compiler optimizations. C++ versions of the code have been compiled using `-O0` and `-O3` flags, the overall performance is analysed and discussed in Section 4.

As for the fully conversion to C++, all `Filter` and `Define` functions need to be replaced by valid C++ expressions (lambda functions, functions, functors, ...). Taking the filter in Listing 3.8 as an example, the equivalent C++ code may adopt the form of a lambda function which receives four parameters (explicitly defined) and applies the same operation (Listing 3.9).

```

1 # Diagonal cut
2 f2 = f1.Filter("v_L_2_F && v_L_2_N && v_R_2_F && v_R_2_N")

```

Listing 3.8: RDataFrame code using jitted code in Python

```

1 // Diagonal cut
2 auto allDiagonalRPs = [](unsigned int &v_L_2_F,
3                          unsigned int &v_L_2_N,
4                          unsigned int &v_R_2_F,
5                          unsigned int &v_R_2_N){
6     return v_L_2_F && v_L_2_N && v_R_2_F && v_R_2_N;
7 };
8
9 auto f2 = f1.Filter(allDiagonalRPs, {"v_L_2_F", "v_L_2_N", "v_R_2_F", "v_R_2_N"});

```

Listing 3.9: RDataFrame code without jitted code in C++

While the former is quicker and allows a more interactive first exploration, the latter code provides a more efficient though verbose implementation. At compilation time, the compiler benefits from the explicit type declaration leading to an optimized execution of the code.

3.4 Distributed execution

Connection between Spark and RDataFrame code requires a third element able to submit RDataFrame-based tasks to a Spark instance without requiring the user to modify the code. On this section we describe two complementary approaches: DistROOT and PyRDF.

3.4.1 DistROOT python library

DistROOT [36] is a Python library that allows the Spark parallelization of ROOT analysis, using the ROOT Python interface (PyROOT). The underlying parallelization applies the map-reduce pattern to the processing of a ROOT TTree. In the map phase, each mapper reads and processes a sub-range of TTree entries and produces a partial result, while the reduce phase combines all the partial outputs into a final result. The number of partitions in the Spark configuration determine the number of ranges to be considered by DistROOT.

Spark works with the abstract concept of a resilient distributed dataset (RDD), which is a collection of elements partitioned across the nodes of the cluster, while in this case the collection is composed by events that need to be operated in parallel. In that sense, when working with multiple ROOT files each one contains part of the full event collection. DistROOT helps to abstract the user from these low-level operations. In order to run a ROOT analysis distributed, the user specifies the input of the analysis as a list of files containing a TTree and the TTree name³, as well as the mapper and reducer functions. Internally, DistROOT inspects these functions to extract all the required context information and sends them to Spark.

Despite offering an adequate integration with ROOT, one drawback of this library is that it still demands the user to adapt the code, splitting the analysis workflow into two functions, mapper and reducer, with specific inputs and outputs parameters. As described on the previous section, the reference physics analysis is split in two main stages: `distill` and `distributions`. To enable the `RDataFrame` version to run in Spark easily, these two parts are merged into a single function that acts like a mapper. It can be seen from the code in Listing 3.10 that the result of the `distill` is directly passed to `distributions`. What is interesting in this example is that no intermediate ROOT file is created between both parts as it happens on the original code, leading to a significant reduction in the I/O operations. The larger these intermediate files are the bigger the impact is on the execution time, since more I/O operations are involved.

The result of the mapper function (`fillHist`) is a list of partially filled histograms. This is another change required in the code in order to go distributed. While the `distributions` part originally creates histograms which are later saved on disk, this one saves partial histograms into a list which is passed to the reduce phase. Since all lists contain a sorted collection of histograms, the reduce phase can easily merge them by iterating each list (11 and 12, lines 5 to 7 in Listing 3.10) and adding one to the other. The functionality of merging two histograms is offered by ROOT.

```
1 def fillHist(rdf):                                # Mapper
2     distilled = distill(rdf)
3     return distributions(distilled)
4
5 def mergeHist(l1, l2):                             # Reducer
6     [first.Add(second) for (first, second) in zip(l1, l2)]
7     return l1
8
9 dTree = DistTree(input_files,
10                  treename = "TotemNtuple",
11                  npartitions = 4)
12
13 histos = dTree.ProcessAndMerge(fillHist, mergeHist)
```

Listing 3.10: `RDataFrame` code to run distributed with Spark

³TTree is the structure that contains the events spread in multiple ROOT files

3.4.2 PyRDF

In the context of the Google Summer of Code [37] program⁴, the python library PyRDF⁵[39] has been developed to extend the functionality provided by DistROOT.

This library aims to run a RDataFrame analysis on distributed resources without requiring changes on the original code. To achieve this, the complexity of submitting distributed ROOT jobs is hidden on the library internals. Moreover, it defines an interface to deal with multiple backends. Currently only *local* and Spark backends are supported, however this list can be extended to other technologies such as Dask or even for clusters based on SSH. One advantage of PyRDF is that users can select which part of the analysis run locally or distributedly, since switching backends is as simple as a function call. In that sense, PyRDF offers distributed parallelism following the same principle as for the implicit parallelism offered by RDataFrame, where users do not have to adapt their code since everything happens on the underlying layer. This library frees the users from the definition of mapper and reduce function, removing one of the main drawbacks of the DistROOT approach. Nonetheless, DistROOT keeps being employed on the implementation of the library.

Listing 3.11 shows an example of code running in two different backends. In line 5, we select the Spark backend as well as define the configuration for the Spark instance. After that the RDataFrame is created, reading the events from a TTree split in two files. As we have selected the Spark backend, this operation will be in fact run in a remote node, and hence these files need to be reachable from there. Locals files can be also sent to the remote resources by using `PyRDF.include("/path/to/file")`. Given that the number of partitions specified is 4 (Line 5), PyRDF will generate 4 ranges of events out of those two files. Each partitions process the same number, applying the operations defined in lines 11-13. As a result, a partial histogram `h1` is produced in each node and merged into a single one afterwards. This histogram is now accessible from the local machine. In line 16, we switch back to a local backend, so that the following histogram `h2` is created in the same local node.

```

1  import PyRDF
2
3  # This is the only extra statement to run distributed
4  # Select backend and pass configuration
5  PyRDF.use('spark', {'npartitions':4, 'spark.executor.instances':5})
6
7  # Read list of files
8  rdf = PyRDF.RDataFrame("myTreeName", ["file1.root", "file2.root"])
9
10 # Run distributed
11 rdf2 = rdf.Filter("x > 5")
12     .Define("new_column", "xx + yy" )
13 h1 = rdf2.Histo1D("new_column")
14
15 # Switch to local backend
16 PyRDF.use('local')
17
18 # Run in local
19 h2 = rdf2.Histo2D("x", "y")

```

Listing 3.11: Seamlessly execution of RDataFrame code in Spark

⁴Google Summer of Code is a global program focused on introducing students to open source software development. Students work on a 3 month programming project with an open source organization during their break from university.

⁵The author of this work has been co-mentor of this project, collaborating in the conceptual design and guidance of this new library developed by the student: Shravan Murali [38].

The ability to write code able to run using different backends at different stages of the process becomes specially convenient for interactive analysis in platforms such as Jupyter. Therefore, analysis defined in notebooks can alternate cells running with different backend depending on the workload of the operations.

4

CHAPTER

RESULTS AND DISCUSSION

On the previous chapter we have described the conversion of the original code to an analysis based on `RDataFrame`. On the following sections we present a study of performance comparing both versions of the analysis. First the test bed for the runs is presented on Section 4.1, considering the two scenarios of our interest: local and distributed. Later on Section 4.2 we discuss the results in a local environment where both sequential and multithreaded executions are examined. Section 4.3 shows scalability tests for different partitions running in a distributed environment with Spark. Finally, in Section 4.4 the validation of the results is explained.

4.1 Test setup

4.1.1 Local execution

All runs shown on Section 4.2 were performed on a computer with an Intel Core i7-6700 3.40GHz processor with 4 physical cores and 2 virtual cores each (total of 8 threads) and 16 GB of DDR4 2133 MHz memory. As for data storage, due to the big amount of data, only the first four datasets (DS1, DS2, DS3 and DS4) have been copied to a 2TB hard drive disk. This approach ensures that results from performance tests are not affected by network latencies out of our control. The intermediate results produced by the first part of the analysis (`distill`) are written into a SDD and hence read from there at the beginning of the second stage (`distributions`).

Concerning the software environment, both codes ran with an equivalent configuration, ROOT version 6.15, Python 2.7.15 and same software dependencies, all of them taken from CERN distributed File System (CVMFS) [40] which is mounted in the same node with the software stack cached in memory.

4.1.2 Distributed execution

All distributed analyses have been run in the Helix Nebula Science Cloud [41] platform (HNSciCloud in short), an european hybrid cloud platform led by CERN that aims to support high-performance, data-intensive scientific use-cases. The architecture of the deployment is shown in Figure 4.1 and it involves the following components:

- EOS [42] is a distributed storage system used to host all physics data at CERN. A dedicated EOS instance storing a subset of TOTEM experiment data (presented

in Section 3.1) is deployed on the HNSciCloud, allowing for fast data access from the computing nodes operating in the HNSciCloud.

- SWAN [43] is a web-based platform to perform interactive data analysis. It inherits the Jupyter notebook interface and integrates the ROOT analysis framework with the dedicated ROOT C++ kernel. In addition, it is capable of offloading massive computations to a Spark cluster, it uses CVMFS to access scientific software packages, and provides access to EOS emulating a local filesystem access.
- A dedicated Spark cluster is deployed on the HNSciCloud and is integrated with SWAN. Specifically, we make use of RDataFrame, DistROOT and Spark for spreading the computational load across worker nodes.
- CERNBox [44] offers a web interface to manage files stored on EOS and allows for easy sharing of SWAN notebooks between users as well as for synchronizing selected folders with personal laptops.

EOS, SWAN and CERNBox services run in Docker containers orchestrated by Kubernetes, while Spark runs on plain VMs and is configured via Cloudera Manager. The overall deployment consists of 25 VMs, 388 CPUs, 1,450 GB of memory, and 21.5 TB of storage. 288 CPUs and 1,088 GB of memory are reserved for Spark, while 16.4 TB is the physical storage space available for EOS (the actual space available is 8.2 TB due to a `replica 2` layout of the stored files).

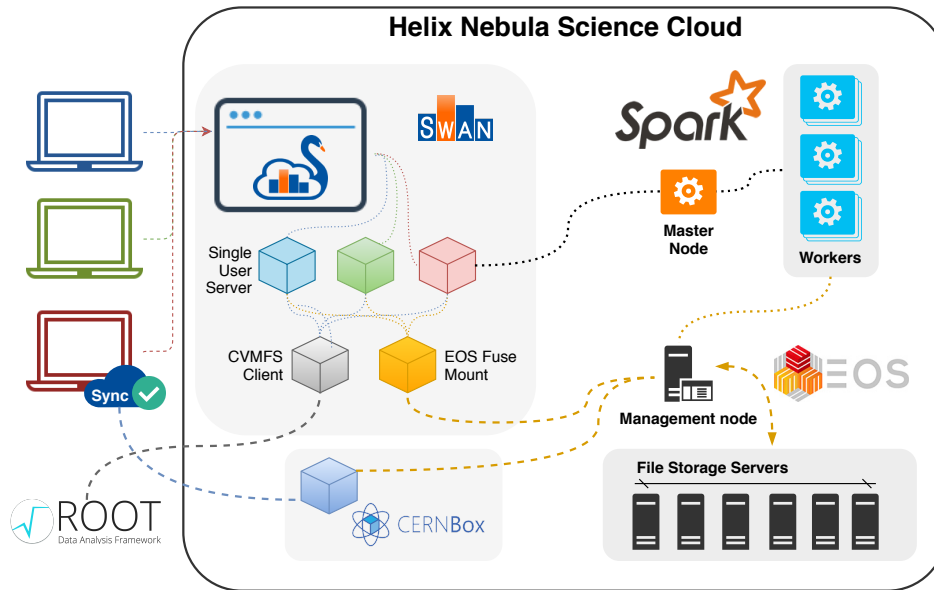


Figure 4.1: Architecture and deployment of the distributed analysis components

Results presented on Section 4.3 correspond to executions on a Spark cluster with up to 256 total cores allocated to Spark workers. All datasets are available through EOS which is mounted on every node, therefore the 4.7TB of data is read remotely. The software configuration of the cluster is:

- Hadoop: CDH_5.15
- Spark: 2.2.1

- Java: 1.8.0_161-b12

The software dependencies required by the analysis itself are picked from CVMFS using the same software distribution employed for the analysis on a local machine, with ROOT version 6.15 and Python 2.7.15.

Regarding the configuration of the Spark context (partitions, executors, memory) different configurations are tested and presented in Section 4.3.1 before running the analyses.

4.2 Local comparison between reference code and RDataFrame version

This first series of tests aims to compare the performance of the original version of the TOTEM analysis code with the code translated into an RDataFrame form. For the RDataFrame code, we will show results both for sequential and multi-threaded runs. This RDataFrame version has been implemented to only produce the most relevant plots for the final analysis, the original code has been consequently modified to generate the same subset of results. Both codes provide an equal number of histograms so that the comparison is not biased by external factors other than their internal implementations.

4.2.1 Sequential execution

This first analysis runs two versions of the code in a local machine with a subset of the total amount of data. In addition to the disk space limitation in the node, the biggest datasets have been discarded for this comparison since the first stage of original code takes an excessively high amount of time to process medium-sized datasets (order of hours even for less than 200GB of data). Each of the considered dataset is independent from others; each one can be analysed at different stages, as indeed it's done in the reference analysis.

Regarding the code configuration, the RDataFrame version runs using the Python interface. As previously stated, this means that the C++ code present on `Filter` and `Define` functions in form of strings is compiled at runtime by the interpreter, which internally sets the optimization level to the equivalent of the Clang `-O0` flag. On the other hand, the original code is compiled with the GCC `-O3` flag, thus all possible optimizations are considered.

A related point to consider is that times shown below refer to the whole process which includes: import of the headers, initializations of the modules, reading the input files, the full event loop and finally writing to disk both results (intermediate file after running `distill` and final results produced by `distributions`).

As Figure 4.2 shows, there is a significant difference between the two codes in terms of performance. RDataFrame versions achieve a major time reduction for all datasets analysed, taking a third of the time for running the full analysis compared to the original code.

However, having a closer look at the plot we realise that this improvement is not proportionally reflected in both parts of the analysis. Portions of time dedicated to run `distill` and `distributions` are unbalanced with respect to the overall gain in the total execution time. Differences in time are related as follows:

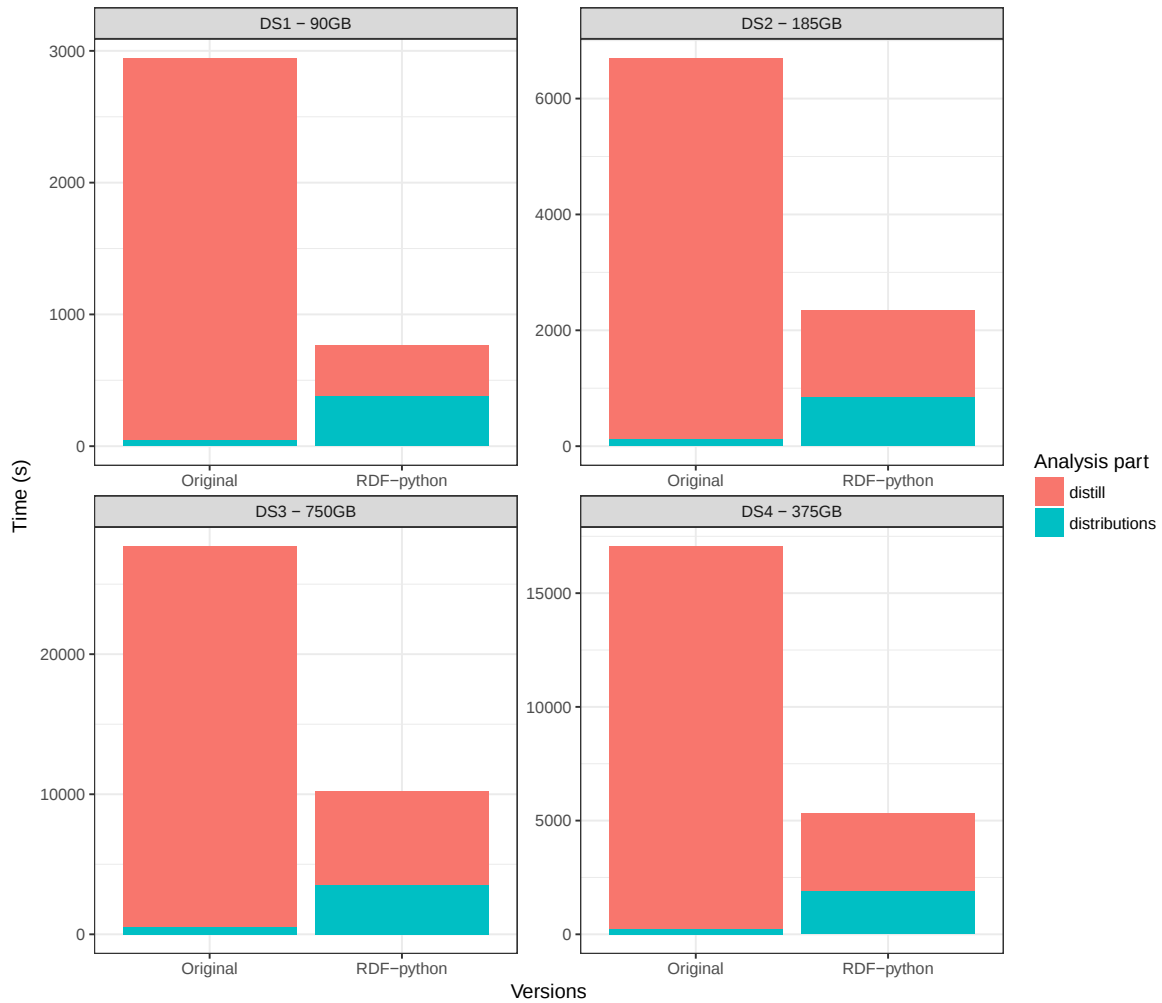


Figure 4.2: Execution time of the TOTEM analysis for each code version. Equivalent code expressed with the RDataFrame model runs on average 3 times faster.

- **distill** is faster by a factor of 7 running with RDataFrame.
- **distributions**, which is the fastest part for the original code, runs 6 times slower in the RDataFrame version.

The single most striking observation that can be done based on the the previous results is that RDataFrame is able to run the same analysis in a much shorter time, even being slower for some parts of the code and running with a configuration that favours a better execution time for the original code:

- Original code employs only compiled code which is known to be faster than interpreted code (optimization level 02 with respect to optimization level 00)
- Although RDataFrame does compile certain parts of the analysis, this compilation indeed adds a slight overhead since the code is *just-in-time* compiled at runtime.

In addition, RDataFrame offers the option to run multithreading without changes on the code. This feature has not been considered for this analysis which lead us to think that execution times shown in Figure 4.2 could be further reduced using implicit parallelism.

Nonetheless, these results provide support for the assumption that RDataFrame has still room for improvement in some areas. For the sake of better understanding, a more-in-detail analysis for the `distributions` part is described below.

Distributions comparison

Previous results (Figure 4.2) show a significant increment in the execution time for the second part of the analysis, `distributions`. This section covers a detailed study conducive to find out the reason. The version computed in the aforementioned figure does not provide enough information for debugging purposes. As it exploits the *just-in-time* compilation of the code, most of the symbols produced at runtime cannot be resolved by profiling tools. Therefore the code needs to be expressed with the C++ interface of RDataFrame, which in addition offers some other mechanisms to optimize the code. The following modifications are considered in order to reduce the work on the compiler side and thus speed up the runtime:

- Explicit type declaration of the columns plotted in 1D and 2D-histograms, reducing calls to the interpreter (at runtime) to infer the type.

```
1 f4.Histo2D<double, double>(al_sel_models[0], "h_al_x_L1_F", "h_al_y_L1_F");
```

- Replace jitted code in `Defines` and `Filters` methods by using:

- Compiled functions defined in the headers

```
1 auto r3 = r2.Define("kinematics", DoReconstruction , { "h_al" } )
```

- Lambda functions

```
1 // Example 1
2 auto SkipTimeBis = [](unsigned int &t){
3     return ! SkipTime(t);
4 };
5
6 auto f1 = rdf.Filter( SkipTimeBis, {"timestamp"});
```

```
1 // Example 2
2 #define GETMEMBER( type, member ) [](type &st){ return st.member; }
3 ...
4 auto r4 = r3.Define("k_th_x_R", GETMEMBER( Kinematics, th_x_R ), {"kinematics"})
```

- Make use of compiler optimizations at the same level as the original code, which are those part of the `-O3` flag.

Additionally, RDataFrame offers an option to define *named-filters*. These are just normal `Filters` that receives a name so that later reports can be generated with summary information about how many events have successfully passed each filter. For this analysis, we have considered two versions of the code removing these named filters from one of them.

As for the Python code, a few optimizations can be also taken into account. The first one is the removal of the mentioned named-filters. The second one, described in

Section 3.3.2.1, consists on compiling the C++ headers so that functions defined on them do not need to be compiled at runtime.

A summary of the results is shown in Figure 4.3, where we can compare the original code against several versions of RDataFrame using both languages. All results correspond to the dataset DS1.

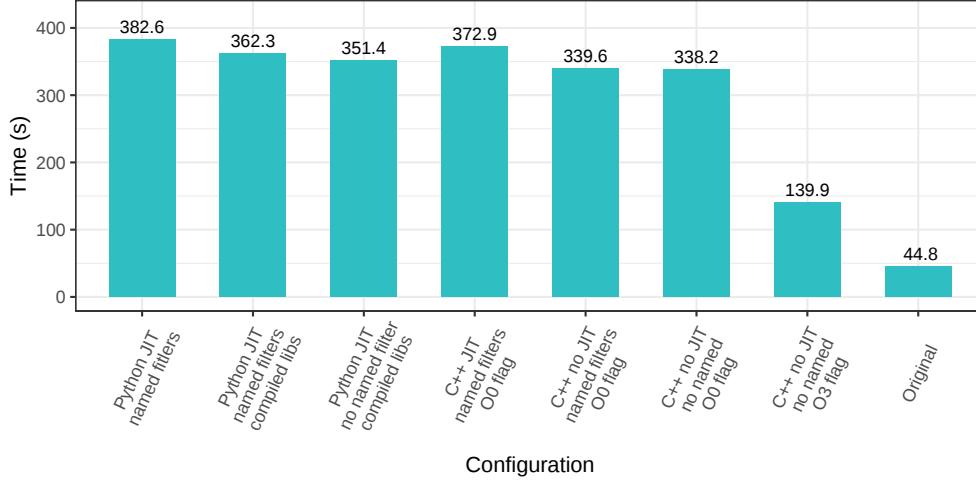


Figure 4.3: Execution time of the event loop in `distributions` for different versions of the RDataFrame code compared to the original code using dataset DS1 as input.

Comparing the first six results we notice a slight improvement using C++ rather than Python. This outcome is expected since we are basically removing one layer from the computation but still running the core of the analysis using C++ code. In order to run C++ code from Python, every call passes first by PyROOT that acts as a glue between both languages. The three first results correspond to Python code. Here we see that both optimizations provide a better performance of the code. Considering the equivalent results for the C++ version, it can be seen that the use of named-filters adds an overhead of 30 seconds on this analysis. Named filters work as normal filters, but also keep track of how many entries they accept and reject. For each event every named filter needs to be updated and so the total number of operations increase, which explains the perceptible overhead given the amount of events.

As it can be expected, the most significant improvement comes after using compiler optimizations together with the aforementioned modifications in the code. Nonetheless, the difference between the original code and this optimized version is still a factor of 3. Preliminary tests have shown that this overhead is also distributed among threads when running in parallel with good results (Figure 4.5 in next Section).

Additional profiling analyses have been carried out to better understand the reasons behind this slowdown. Namely, we have employed the `igprof` tool to track down the execution of the RDataFrame version with the C++ interface, looking into the details at a smaller granularity. The conclusion of this analysis is that the adoption of the ROOT method `TTreeReaderValue` on the RDataFrame implementation for reading operations adds an overhead of about 30% more time with respect to the reading strategy followed by the original code. This explains why the execution time grows as a side effect of increasing the number of histograms (Figure 4.4) generated during the analysis, since the number of calls to the `TTreeReaderValue` method also increases.

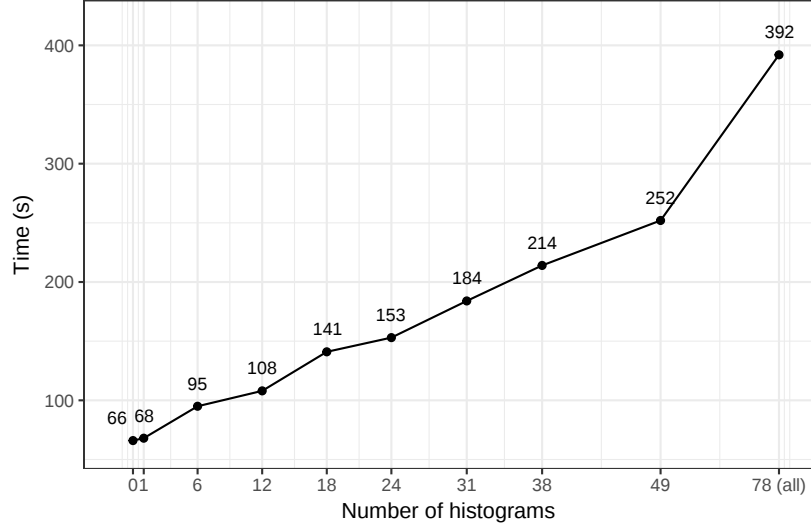


Figure 4.4: Evolution of the execution time by number of produced histograms using the Python JIT version of RDataFrame

The fact that `TTreeReader` produces an overhead when accessing files on disk is one of the already known problems for the ROOT team. However, this has not been a critical problem at the sizes tested so far in previous analyses. As a result of this work we have highlighted an undetected performance degradation pattern in I/O operations and identified the ROOT component to improve, with an immediate advantage for the analysis community of the LHC experiments.

4.2.2 Multithreading executions

Once seen how good RDataFrame behaves compared to the original code, we focus on the implicit parallelism offered by the framework. Results shown below consider the same test configuration described in Section 4.2. Times presented on these plots include the main event loop but also the writing time for the output files since these operations are also parallelized by RDataFrame.

Measuring the performance of RDataFrame running multithreading in a local machine helps us first of all to calibrate the real potential of the framework. Secondly it serves as a reference to determine the size of data from which running in a distributed infrastructure as Spark begins to be worth. This comparison is addressed on Section 4.3.2.

The results obtained from the same analysis running with different number of threads are summarised in Figure 4.5. For this study, five different configurations have been tested per dataset: a sequential execution and multithreading up to 8 threads. Given that the test machine counts with 4 physical cores, the hyperthreading of the CPU will be exploited when running with 8 threads. The first result of each dataset in Figure 4.5 corresponds to the sequential execution so it is equivalent to the result plotted in Figure 4.2 for the RDataFrame version.

The execution time of RDataFrame running with one single thread is represented by the second bar. Here we identify certain overhead added by the implicit parallelization of the code. In particular, such extra time is minimal for DS1 and DS2 while it tends to increase along with the size of the dataset. These results contrast with the fact that there is not visible difference in time between the sequential mode and the one-thread

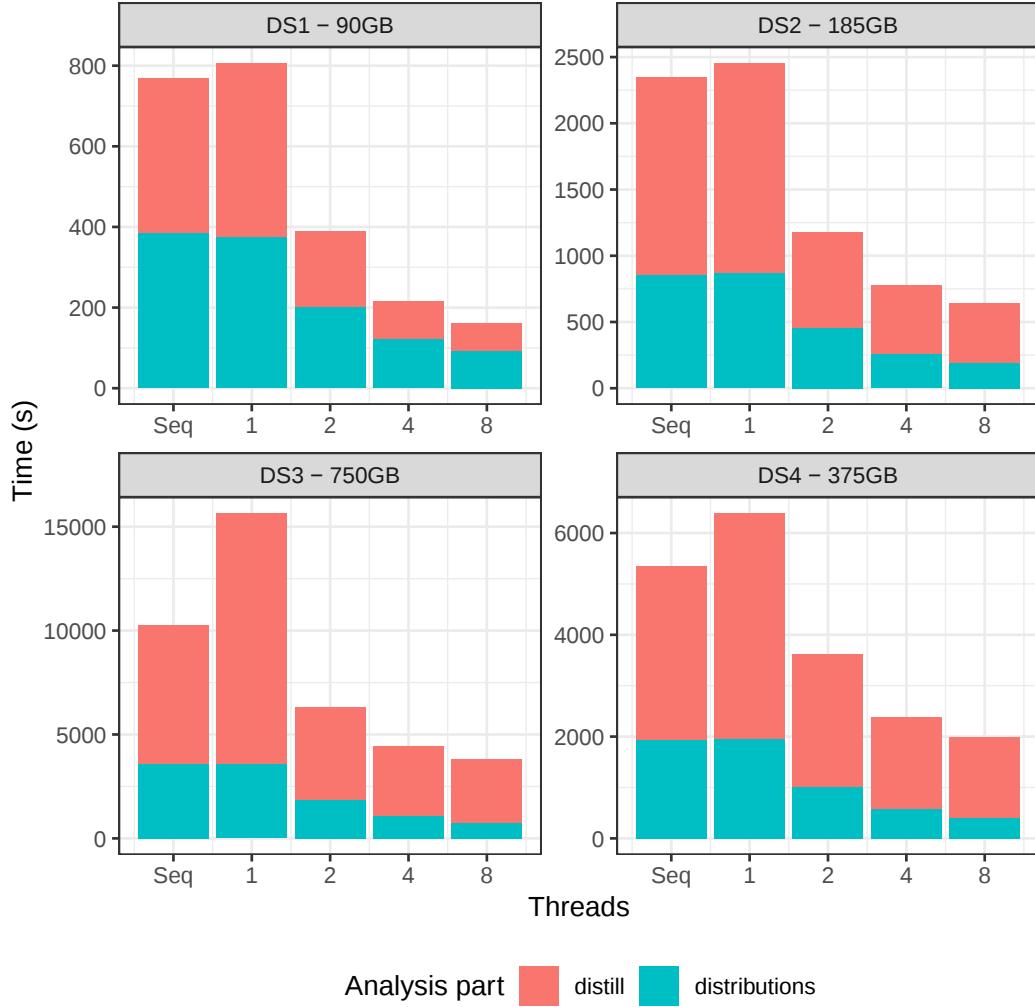
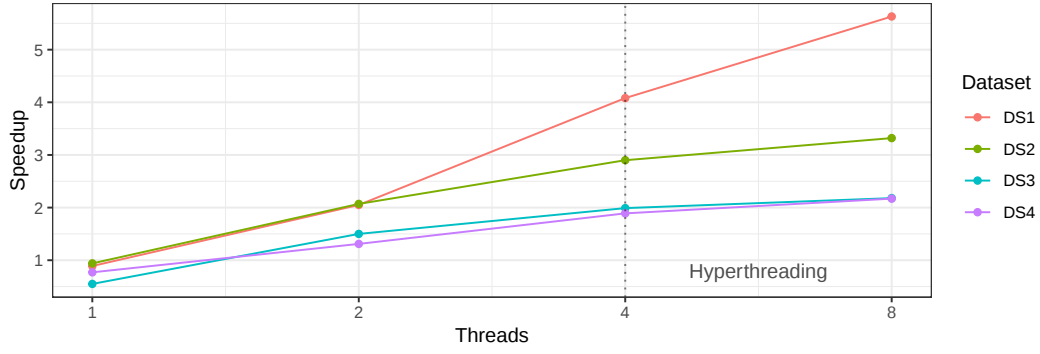
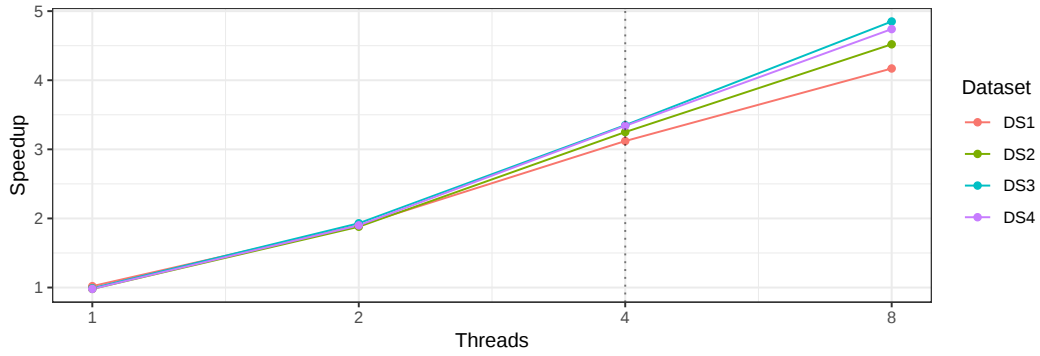
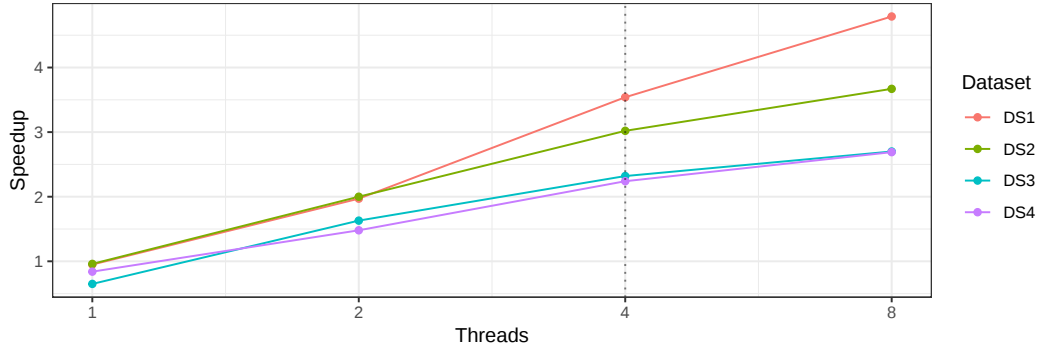


Figure 4.5: Total execution time with RDataFrame running with different threads.

execution when running `distributions` (blue part of the bar) for any datasets. It is known that the creation of tasks for the implicit parallelization adds an overhead when applying multithreading with one single thread, however results for DS3 and DS4 datasets show an unexpected increase. This behaviour needs to be further investigated. Our assumption is that it may be related to I/O operations: The input for `distill` is read from a HDD and written to a SDD, whereas `distributions` directly reads from a SDD and barely writes a few megabytes. In consequence, the discrepancy would be larger for bigger amount of data.

As long as the number of threads increases, the execution times are reduced being specially significant from one to two threads, due to the overhead suffered. For more than two threads, time gets reduced at a slower rate. In spite of running the analysis on a 4 physical cores machine, an execution with 8 threads was also included to force the hyperthreading. All datasets reveal that they can scale up to 8 threads although the improvement from 4 to 8 threads is minimal with respect to the speed up obtained for less threads. The proportion of time between `distill` and `distributions` is not constant since each part of the analysis evolves differently with the number of threads.

Figure 4.6 illustrates better this progression. It contains three plots with the speedup achieved on each dataset running with a different amount of threads taking as a baseline

(a) Scaling of `distill.py`(b) Scaling of `distributions.py`

(c) Overall scaling of full analysis

Figure 4.6: Scaling of RDataFrame version with multithreading enabled.

the sequential model for the speedup calculation. The plot 4.6(a) on the top presents the speedup for the `distill` stage. It can be seen from this plot that smaller datasets get a more significant speedup. DS1 stands out from the rest, reaching the ideal speedup for 2 and 4 threads and getting the highest value for 8 threads. Bigger datasets also improve their speedups but at a slower rate. The middle plot 4.6(b) represents the speedup for the `distributions` part. Here we see exactly the opposite results. The bigger the dataset is, the better speedup, although all of them have a similar trend. Finally, the bottom plot 4.6(c) shows the average speedup considering both parts of the analysis.

Overall we see that code which features more I/O intensive patterns (`distill`)

presents a worse scaling, except for small datasets such as DS1. On the other hand, the `distributions` part scales up with a more constant speedup as the number of threads increases. Taking into account that `distributions` performs the most relevant part of the physics analysis (filters, physics cuts, histograms) it can be considered a positive result that it gets the best scaling. Even more considering that this stage of the analysis is meant to be run multiple times for the physicists while tuning the parameters.

As a conclusion of this comparative, we identify that I/O operations have a substantial impact on the multithreading scaling of `RDataFrame`, being particularly worse as the size of the datasets grows. Although it is out of the scope of this work, some additional tests have shown a significant improvement on time when reproducing the same tests with data read from a SSD. In general, `RDataFrame` has demonstrated to succeed on providing an efficient implicit parallelism where users do not need to modify a single line of code to take the most of the available resources on their machines.

4.3 Distributed analysis with Spark

4.3.1 Spark configuration

Despite the combinatorial in the number of possible Spark configurations is quite high, a non-exhaustive first analysis is discussed in the following section. Since our goal is not to figure out the best Spark configuration for the given amount of data but to measure the final performance of the programming model in a distributed scenario, we will select a configuration general enough to deal with all the considered partitions. Once chosen, the same configuration will perform all the tests. Parameters considered on this study are the following: range strategy in `DistROOT`, executor instances, executor cores and executor memory.

The number of executor instances times number of cores per executor gives us the total amount of available cores for each configuration. In multithreading execution, `RDataFrame` splits the workload on threads. A range of values from the total input data is assigned to each thread and the maximum number of threads is limited by the number of cores available on the host. The current approach is slightly different since in Spark the workload is split by tasks. The concept of partition determines how many tasks are considered. Each core runs a single task/partition of the total data at the time and hence the total number of parallel tasks is limited by the number of cores. A workload split in many tasks means a lot of small partitions which will run faster but will have a bigger overhead. On the other hand, a few big partitions will reduce the overhead but take longer.

In our case, we know from [2] that `RDataFrame` is able to scale up to 100 threads without adding a significant overhead. Therefore the chosen configuration should count with enough cores to exert the most of the interface given its potential to scale. Additionally, for a small number of partitions we want to have one single Spark task running in a core at the same time. For a given configuration where the number of partitions is greater than the total number of cores, multiple tasks will be allocated to the same core leading to an inefficient use of the resources if the tasks are notably big. We are interested in running from 1 to 64 partitions, increasing by powers of 2. Such range allows us to compare the parallel execution between threads in a local machine and partitions in a clusters. Besides, we can measure the performance when having a bigger number of partitions than the number of threads tested on the previous section.

Preliminary tests have shown a significant increase in the initialization time of DistROOT before starting the Spark jobs. By default DistROOT creates partitions out of the entries of ROOT files. This means that every single ROOT file has to be previously read to determine the number of entries, sum the total amount of entries per file and define the ranges. Since each ROOT file has to be accessed and opened, there is a correlation between number of input files and needed time to compute the ranges. As an alternative approach, a custom version of DistROOT has been tested, considering files instead of entries for the ranges, thereby reducing the granularity of the division and increasing the unbalance between cores. As a result, each core works with a list of files instead of with a list of entries. This distributions is not optimal for small number of files, but fits well if the number of partitions is smaller than the number of files, as it is our case. Besides a reduction of the execution time, splitting by files seems to produce a better scaling up with the number of partitions

4.3.2 Scaling with Spark

Considering only the configuration described in the previous section, we run the analysis for the same subset of datasets analysed in Section 4.2 for the local comparison. A sequential mode is not executed in this case since running on Spark already adds an overhead equivalent to enabling the implicit multithreading with a single thread.

Figure 4.7 compares the speedup obtained for each dataset using the a range of partitions from 1 to 64. As we can see, all datasets have a positive speedup up to 64 partitions, except for DS1 which barely improves beyond 32. What is interesting in this figure is that bigger datasets seem to perform better with a larger number of partitions. For example, DS4 beats the rest of datasets for all partitions. DS2 (185GB) however scales worse than DS1 (90GB) until 32 partitions but keeps improving for 64.

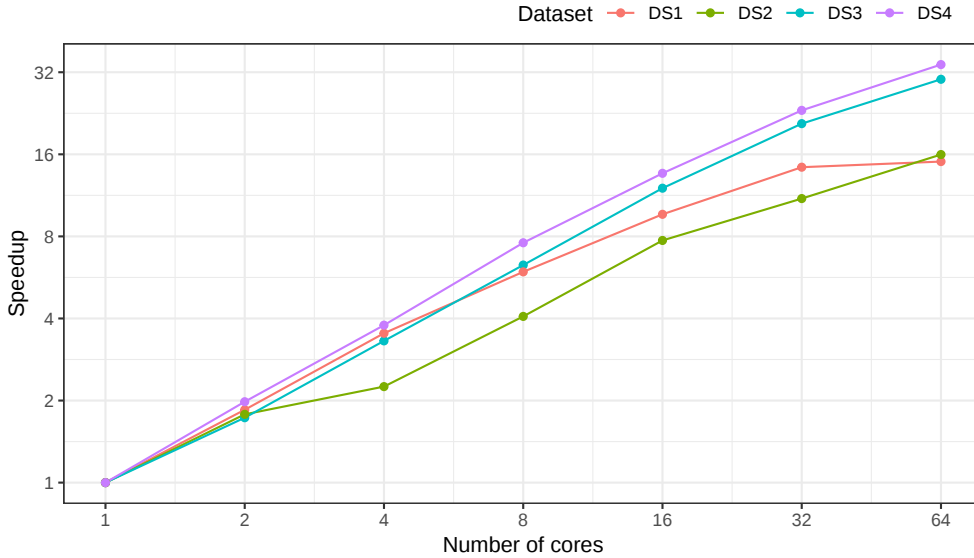


Figure 4.7: Scaling with smaller datasets in Spark. Number of partitions coincides with number of cores for each case.

These results contrast with the ones shown in Figure 4.6(c) for the multithreaded execution in local, where the speedup for the total execution time was worse for bigger datasets. However, both scenarios differ significantly: while in the local execution all

datasets were stored and read from a local hard drive disk, in this case input data is read remotely through the network. Memory-wise, we have used a configuration with 60GB of memory per executor against the 8GB present at the local machine.

In summary, we see that all datasets scale up easily with the number of partitions reducing the processing time to a few minutes for a hundred of gigabytes. In general, the distributed platform benefits from a better scaling than the multithreaded execution in a local environment. On top of that, the platform offers access to a larger number of cores, bigger data storage and interactivity on the analysis, creating an ecosystem suitable for scientific studies.

4.3.3 Comparison between distributed Spark execution and original code

Finally, we run the analysis code for all available datasets using Spark. The execution time of each run is compared against the time it gets when using the original way of running this code. For large datasets, TOTEM analyses are usually submitted to the CERN batch system, called LSF, where they run in a sequential mode as well as reading remotely from EOS, alike for the Spark cluster.

Figure 4.8 is quite revealing in several ways. First, unlike the other figures it shows for the first time execution times for the rest of datasets (DS5, DS6 and DS7). In this case we are comparing two different code versions of the same analysis in completely different environments. The reference code, written in C++, has been run in the LSF batch system whereas the RDataFrame version uses the Python interface and was executed exploiting the integration with Spark using 64 partitions for each dataset.

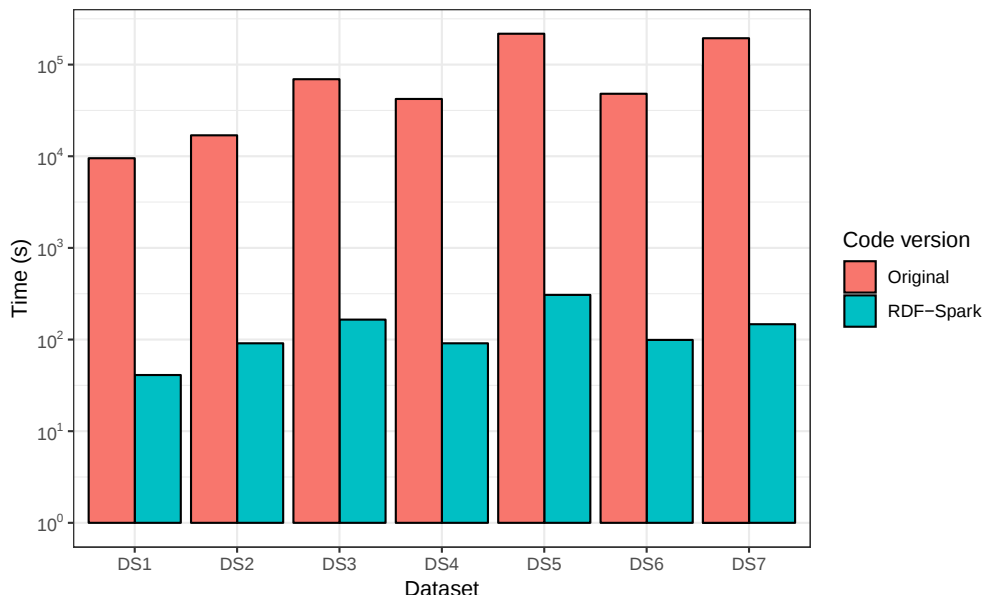


Figure 4.8: Execution time of the original code run at CERN batch system compared to the RDataFrame version running in parallel with 64 Spark workers.

The purpose of this comparison is to demonstrate that current tools have already the potential to run same analyses in a more efficient way. Combining developments in

different areas we are able to reduce the execution time by multiple orders of magnitude. From the figure, on which both axes feature a logarithmic scale, it can be seen that on average the data analysis goes from hours to a few minutes. Furthermore, it takes around five minutes to run the biggest dataset (DS5) which contains 1.85TB of data and 1.162 million of events. In terms of computation, this can be translated to a total of 6GB processed per second and 3.8 million of events processed every second. The same dataset takes more than one day to be run with the original code and using the batch system.

4.3.4 All datasets

To conclude the series of analyses we put to test the potential of the platform and the programming model by running the total amount of 4.7TB of data for different partitions. The configuration in this case has been modified to allocate 240 cores so it can scale up to a bigger number of partitions. Figure 4.9 illustrates the evolution of the execution time by increasing the number of parallel workers. Running with a single core, the analysis takes around 13 hours and 39 minutes. By contrast, creating 512 partitions split among the 240 cores, the execution time reaches its minimum value, taking about 7 minutes to process the whole dataset. In this scenario, when the number of partitions is bigger than 240, more than one is allocated to each core. In terms of events, 6.7 millions of event are processed per second and around 11GB per second. For a bigger number of partitions, the processing time does not decrease further which means that the number of tasks allocated to each core adds more overhead than speedup.

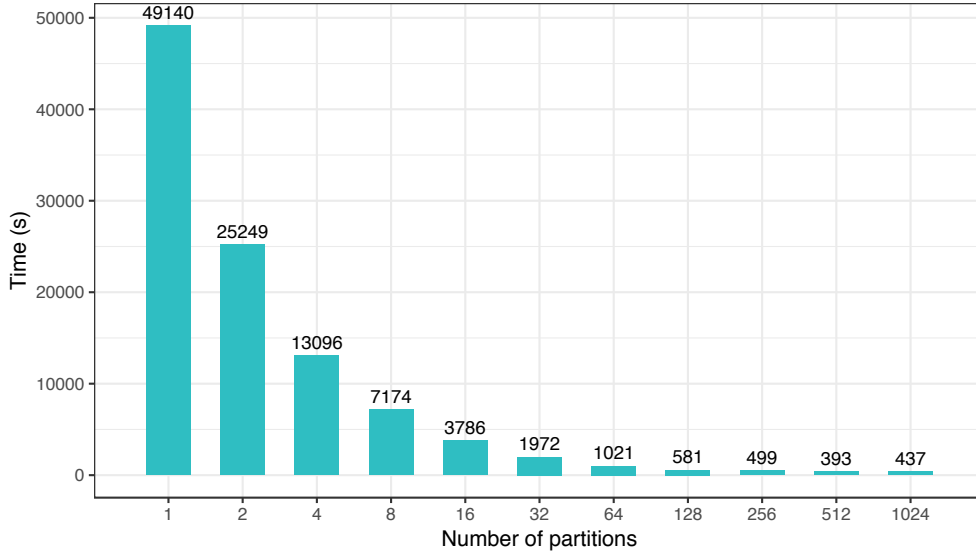


Figure 4.9: Execution time for 4.7TB of data running distributed with RDataFrame and Spark. Number of partitions does not coincide with number of cores for each case.

For such volume of data, the speedup grows well up to 32 partitions and stops increasing after 128. This result is consistent with the conclusions presented in Section 4.3.2, running a subset of datasets: bigger datasets tend to scale better along with the number of partitions. Despite the time gets significantly reduced, for many partitions we see a speedup which is far from being optimal. Further investigations in this aspect is needed to figure out the reasons.

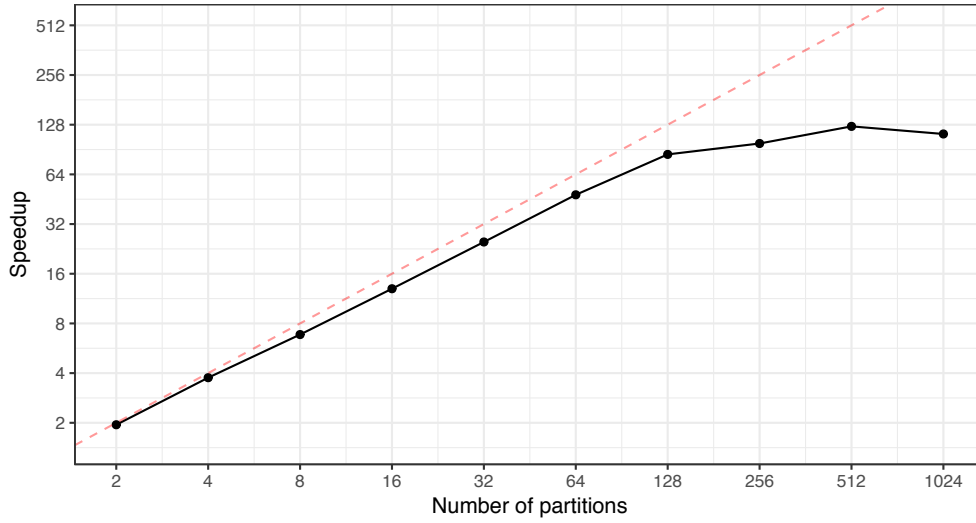


Figure 4.10: Scaling with 4.7TB of data up to 1024 partitions.

4.4 Result validation

Correctness of the results is a big concern for the experiments so before moving to a new model we must be able to prove that the new implementation of the analysis still provides the same results. For the sake of validation, two mechanisms have been used during this work:

- **rootcompare**, a tailored C++ script have been developed to compared the intermediate ROOT files produced after the first stage of the analysis (**distill**). In summary, every value stored at the file produced with the new code is paired with its homologous from the original version. This step ensures that both files are equivalent and so the same number of events is distilled.
- Visual validation, the most relevant histogram for the physics analysis was identified by the TOTEM experiment. Since the chosen plot is the result of all filters and transformation applied during the second stage of the analysis, its correctness ensures that all previous histograms are also correctly produced. Additionally, each ROOT histogram provides a statistical summary of the represented data (mean, std dev and total number of events plotted). The validation has taken into account these values besides checking the visual similarities. Moreover TOTEM experts on this analysis have supervised and approved the validation process.

5

CHAPTER

RELATED WORK

Particle physics is no longer alone in facing the efficient processing of massive data volumes. Huge amounts of data are also produced by experiments in other fields, such as astronomy, cosmology or genomics, which will need to overcome the same challenges. Additionally, there is a clear interest from the computer science community with the irruption of Web 2.0 and Internet of Things paradigms. Furthermore, the portfolio of Machine Learning techniques and tools has rapidly evolved during the last years, drastically changing the way data is analysed. Consequently, a vast array of revolutionary ML-based applications has brought to light successful use cases based on data as well as the necessity of gathering more data to go even further. Massive data handling and efficient scientific programming is also a challenge faced by industry which developed novel and major new technologies under the name of *Big Data* that may be applicable to most of the use cases and problems on the physics domain.

5.1 Programming models for distributed data processing at HEP

HEP data analysis have been successfully applied over many years to produce physics results, including more than 2000 publications during LHC Runs 1 and 2. On this period, different analysis methods have been developed in order to improve the performance and usability. Although each experiment may follow a different approach, LHC data analysis typically starts from multiple data reduction and refinement phases, ultimately producing simplified data structures of arrays of simple data types able to "fit on a laptop". The aim of this extreme reduction, often from hundred of TBs, is to facilitate low-latency, high-rate access to a manageable amount of data thereby boosting the interactivity during the final stages of an analysis. The latter stage contrasts with the high-latency of the reduction phase usually run in batch systems, which is the approach followed by the original analysis used on this thesis. In order to deal with this latency, some strategies has been developed such as the ALICE *analysis trains* [45], which combines analyses from different users and execute them with the same batch jobs, reducing the number of times the data needs to be read from the storage systems. However, this does not solve the iterative and interactive HEP analysis problems.

Modern technologies have emerged proposing new models of data analysis where the sequential reduction is still present but combining interactivity and batch processing. In this sense, Apache Spark [46, 47] has become industry de-facto in recent years as an open source computing framework for efficient and interactive analysis on big

data workloads. These features make Spark an attractive tool for HEP analyses as discussed by [34] and shown on this thesis. The use of big data technologies coming from industry is not a new area in HEP. Groups from CMS experiments have already presented some exploratory work on the reach of Spark in HEP. At a first approximation, an event classification algorithm was re-implemented using Spark but the algorithm itself was unsuitable to use caching and in-memory repeated processing provided by Spark resulting in poor performance as compared with the MPI implementation [48]. Initially, big data tools such as Spark, were neither designed for scientific applications nor to exploit high performance computing (HPC), which were broadly adopted by the scientific community. However in recent years, Spark has been made available for the scientific applications at National Energy Research Scientific Computing Center (NERSC) and other facilities [49] making it even more attractive for physics analysis tasks. Such developments were successfully used by [34] to implement an active LHC analysis, searching for Dark Matter with the Compact Muon Solenoid (CMS) detector, and evaluate its performance on supercomputing resources.

Other works focus on a different tool for the distributed process, such as Hadoop. Although it is based on the same idea as Spark, MapReduce [50], both tools follow an utterly different implementation. The key difference between them lies in the approach to processing: Spark can do it in-memory, while Hadoop MapReduce has to read from and write to disk. Authors of [51] reimplemented a HEP analysis with Hadoop, starting from ROOT-based code. In spite of being able to work with far larger data sets than Spark, Hadoop still does not solve the lack of iterative and interactive processes. On these approaches, all analyzed data would need to be converted to the Spark's own format which is a hindering inconvenience. Thus, the solution studied in this thesis uses only part of the Spark platform which is responsible for the task scheduling.

As we can see, multiple works show the positive impact of using big data tools in HEP analyses. However, the data format (ROOT format) in which the vast majority of the LHC data is stored becomes a hurdle for these tools to get integrated with the existing HEP workflows. As a result, numerous libraries have been created as an intermediate layer to carry out ROOT-based data analysis using new technologies. Regarding Spark, the tool used on this work, DistROOT [36], offers the possibility to use a simple map reduce interface to run existing root code with a few changes on Spark resources. The key is the use of the PyROOT [27] interface in combination with PySpark. Alternatively, [52] offers the possibility to read ROOT files connecting the format directly to Spark's DataFrames.

5.2 Declarative models for parallel execution

In the past decade, size of the data sets produced has rapidly grown. The issue of data handling and analysis is not specific to particle physics but to many other fields. The needs of the broader scientific community together with recent developments in the industrial community provide a large pool of scientific software developers to tackle these common problems. In recent years, this growing community has established Python as its preferred language due to its efficacy, its interconnectivity with other environments and the speed of developments it allows. This has led to the development of a robust scientific software ecosystem with packages such as Numpy [53], Scipy [54] or Pandas [25] among others where the common factor is a powerful yet simple and declarative syntax.

Recently, the scientific Python community has begun to converge on parallel and

high-performance computing techniques to increase the utilisation of a single machine or workstation. In this regard, the most popular frameworks are Apache Spark, already presented on this work, and Dask [55] a flexible parallel computing library for analytic computing developed at the core of the Python scientific community. Thus, it provides full integration with many of the tools on the Python scientific ecosystem. In particular, Pandas dataframes can be stored on disk on a single machine or on many different machines in a cluster. Thanks to the orchestration provided by Dask, many operations can be triggered on several local or distributed Pandas dataframes as part of a more abstract Dask dataframe. This allows Python libraries to make a better utilization of the system resources and a distributed computation.

Given the current design of RDataFrame and the new developments on the PyRDF module presented as part of this thesis, ROOT files could be also distributed backed by Dask. The implementation of a Dask backend for PyRDF is considered as a future work of this thesis.

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

Software and computing domains are at the core of the High Energy Physics and their evolution will be crucial to succeed on the challenges ahead. However, a simply extrapolation of the challenges faced today will not be enough for the step change foreseen with the upgrade version of the Large Hadron Collider. The needs of the HEP programme in the high luminosity era far exceed those that can be met by simply making incremental changes to today's code and scaling up computing facilities.

Fortunately, HEP community is no longer alone in facing the efficient processing of ever-increasing amount of data. Recent breakthroughs in industry have come up with promising Big Data technologies and Cloud platforms that could be applied to the domain of physics data analysis proficiently. On this regard, several works have presented solutions to move from current code and formats to completely new models based on non-HEP technologies. However, being close to the exabyte of gathered data makes these approaches unaffordable in terms of time and space.

As stated by the community, exploiting parallelism is one of the techniques that may overcome the performance limitation in single core CPU's. Nevertheless, its correct execution is complex and the amount of legacy code written with a single core in mind makes it a tough task. The work presented on this thesis embraces recent developments by the ROOT team aiming to face the future challenges. Namely, we focus on the new high-level interface RDataFrame, where users face to a friendly and declarative programming model which transparently performs low-level optimizations and parallel execution in an easy way.

Unlike other related works, the use of RDataFrame has allowed us to natively work with files in ROOT format and at the same time to benefit from the Spark task scheduler to run distributedly on an external cluster. Although there has been previous real analysis translated into the RDataFrame model, this work distinguishes from the rest for being pioneer in three aspects:

- Expressing a real analysis, written by the TOTEM CERN Experiment, in a declarative way by using the RDataFrame ROOT API.
- Processing a dataset of $\mathcal{O}(TB)$ size during the aforementioned RDataFrame analysis.
- Parallelize the execution of the RDataFrame analysis on a cluster of nodes by using Spark, a tool for distributed big data processing.

Main goals of this thesis have been successfully achieved by following a detailed bottom to top approach:

- In the beginning, the analysis code was rewritten to leverage on the PyROOT and RDataFrame model, moving from an imperative paradigm to a declarative model.
- First series of tests between both versions showed a reduction on the processing time by a factor of 3 running the RDataFrame-based code.
- Same results led us to an unexpected increase of the time needed for the second stage of the analysis (**distributions**) when using the RDataFrame version.
 - In-depth analyses of this stage revealed **TTreeReaderValue** as the cause of the slowdown.
 - Thanks to these results, a report was submitted to the ROOT team leading to an improvement of the tool, a further improvement in time for future analysis and a direct impact on the LHC community.
- Once understood the main differences between both versions, performance analyses have been carried out in two distinct environments:
 - Local, measuring the performance of the multi-threaded execution up to 8 threads.
 - Distributed with Spark, using up to 64 partitions for a subset of dataset, comparing the results with the local execution.
- Finally, combining the experience gathered from the previous tests, the total amount of data (4.7TB) has been analysed in 7 minutes running with 512 partitions and 240 cores allocated in a Spark cluster, in contrast with the 13 hours that it takes with a single partition.

As a result of this project many interesting outcomes for the community can be shared. Firstly, overall the speedup observed in these tests is promising. On the one hand, for a reasonable amount of data (up to 1TB) even a local machine has proven to be a successful scenario where the maximum volume of data can be processed in less than 5 minutes, exploiting all the available resources. This allows scientists to develop first stages of the analysis for a considerable size of data before needing extra infrastructure. For bigger datasets, development on cloud platforms such as Helix Nebula together with recent features in RDataFrame has demonstrated the capacity to run analysis of a few terabytes of data using several external cores. Moreover, if further reduction of processing time is achieved, it will be possible to use the distributed approach for nearly interactive analysis of the whole dataset, instead of multi-step batch processing. Such reduction has a direct impact on the community increasing the possible scientific potential of the data, whilst minimising the *time to insight* for a large number of different analyses performed in parallel.

6.2 Publications

Results from this thesis will be publicly presented at the ROOT Workshop [56] held in Sarajevo (Bosnia and Herzegovina). The abstract acceptance mail is attached to this work in appendix [C](#)

Additionally, a more general abstract regarding the collaboration where this work is contextualized have been submitted to the 11th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2018) held at Zurich, the submission mail is attached to this document in appendix [D](#).

6.3 Future work

Further developments on the new PyRDF python module will be needed to fully integrate RDataFrame with a distributed environment such as Spark without requiring any change on the user's code side. Besides, the implementation of other distributed backends such as Dask or SSH-based clusters will enlarge the reach of feature for the community inside and outside the HEP domain.

Current results presented on this work have been considered valuable from the core of the ROOT team at CERN so the test bed developed for this thesis will be extended with recent features aiming to tackle the I/O limitations experienced on our analyses.

Moreover, some of the results presented throughout the performance analyses requires further investigation to fully understand the underlying reasons, such as the increase on the processing time when enabling the implicit parallelism for one thread on big datasets.

BIBLIOGRAPHY

- [1] CERN.
The High Luminosity LHC project.
<http://hilumilhc.web.cern.ch/about/hl-lhc-project>, 2018.
[Online; accessed 11-August-2018].
- [2] Enrico Guiraud.
RDataFrame: easy parallel ROOT analysis at 100 threads.
<http://cern.ch/go/9Vmn>, .
[Online; accessed 25-August-2018].
- [3] Danilo Piparo.
Supporting Future HEP Data Processing with a Parallelised ROOT.
https://indico.cern.ch/event/587955/contributions/2938148/attachments/1679109/2706928/Supporting_Future_HEP_Data_Processing_with_a_Parallelised_ROOT1.pdf.
[Online; accessed 15-August-2018].
- [4] CERN.
CERN.
<https://home.cern/>, .
[Online; accessed 15-August-2018].
- [5] CERN.
The Large Hadron Collider project.
<http://home.cern/topics/large-hadron-collider>, 2014.
[Online; accessed 11-August-2018].
- [6] Science.
Breakthrough of the Year, 2012.
<http://www.sciencemag.org/site/special/btoy2012>, 2012.
[Online; accessed 11-August-2018].
- [7] Apollinari G. et al.
High-Luminosity Large Hadron Collider (HL-LHC): Technical Design Report V. 0.1.
CERN Yellow Reports: Monographs. CERN, Geneva, 2017.
URL <https://cds.cern.ch/record/2284929>.
- [8] P La Rocca and F Riggi.
The upgrade programme of the major experiments at the large hadron collider.

- Journal of Physics: Conference Series*, 515(1):012012, 2014.
URL <http://stacks.iop.org/1742-6596/515/i=1/a=012012>.
- [9] CERN.
The Large Hadron Collider Beauty Experiment at CERN.
<http://lhcb-public.web.cern.ch/lhcb-public/>, .
[Online; accessed 11-August-2018].
- [10] CERN.
A Toroidal LHC Apparatus experiment at CERN.
<https://atlas.cern/>, .
[Online; accessed 11-August-2018].
- [11] CERN.
Compact Muon Solenoid experiment at CERN.
<https://cms.cern/>, .
[Online; accessed 11-August-2018].
- [12] Antonio Augusto Alves, Jr et al.
A Roadmap for HEP Software and Computing R&D for the 2020s.
2017.
- [13] V Vasilev, Ph Canal, A Naumann, and P Russo.
Cling—the new interactive interpreter for root 6.
In *Journal of Physics: Conference Series*, volume 396, page 052071. IOP Publishing, 2012.
- [14] CERN.
Experimental Physics - Software group.
<https://ep-dep-sft.web.cern.ch/>, .
[Online; accessed 15-August-2018].
- [15] CERN.
Future Circular Collider Study.
<https://fcc.web.cern.ch/>, .
[Online; accessed 31-August-2018].
- [16] CERN.
LHC Experiments.
<https://home.cern/about/experiments>, .
[Online; accessed 15-August-2018].
- [17] CERN.
The TOTEM Experiment at the LHC.
<http://totem-experiment.web.cern.ch/totem-experiment/>, .
[Online; accessed 15-August-2018].
- [18] Giovanni Anelli, G Antchev, P Aspell, V Avati, MG Bagliesi, V Berardi, M Berretti, V Boccone, U Bottigli, M Bozzo, et al.
The totem experiment at the cern large hadron collider.
Journal of Instrumentation, 3(08):S08007, 2008.

- [19] G Antchev, P Aspell, I Atanassov, V Avati, J Baechler, CB Barrera, V Berardi, M Berretti, E Bossini, U Bottigli, et al.
First measurement of elastic, inelastic and total cross-section at $\sqrt{s}=13$ tev by totem and overview of cross-section data at lhcb energies.
arXiv preprint arXiv:1712.06153, 2017.
- [20] Rene Brun and Fons Rademakers.
Root—an object oriented data analysis framework.
Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, 389(1-2):81–86, 1997.
- [21] HSF Community.
HEP Analysis Ecosystem Workshop Report, 2017.
URL <https://hepsoftwarefoundation.org/assets/AnalysisEcosystemReport20170804.pdf>.
[Online; accessed 31-August-2018].
- [22] Ilka Antcheva, Maarten Ballintijn, Bertrand Bellenot, Marek Biskup, Rene Brun, Nenad Buncic, Ph Canal, Diego Casadei, Olivier Couet, Valery Fine, et al.
Root—a c++ framework for petabyte data storage, statistical analysis and visualization.
Computer Physics Communications, 180(12):2499–2512, 2009.
- [23] ROOT.
ROOT Files, 2018.
URL <https://root.cern.ch/root-files>.
[Online; accessed 21-August-2018].
- [24] Enrico Guiraud, Axel Naumann, and Danilo Piparo.
TDataFrame: functional chains for ROOT data analyses, January 2017.
URL <https://doi.org/10.5281/zenodo.260230>.
- [25] Wes McKinney et al.
Data structures for statistical computing in python.
In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. Austin, TX, 2010.
- [26] Enrico Guiraud.
RDataFrame, a modern tool to manipulate and analyze ROOT datasets.
<https://root-forum.cern.ch/t/rdataframe-a-modern-tool-to-manipulate-and-analyze-r/29384>, .
[Online; accessed 15-August-2018].
- [27] CERN.
PyROOT.
<https://root.cern.ch/pyroot>, .
[Online; accessed 13-August-2018].
- [28] D Piparo, E Tejedor, E Guiraud, G Ganis, P Mato, L Moneta, X Valls Pla, and P Canal.
Expressing parallelism with ROOT.
J. Phys. : Conf. Ser., 898(FERMILAB-CONF-16-738-CD. 7):072022. 7 p, 2017.

- URL <https://cds.cern.ch/record/2296792>.
- [29] G Antchev, P Aspell, I Atanassov, V Avati, J Baechler, V Berardi, M Berretti, E Bossini, M Bozzo, P Brogi, et al.
First measurement of the total proton-proton cross-section at the lhc energy of.
EPL (Europhysics Letters), 96(2):21002, 2011.
- [30] G Antchev, P Aspell, I Atanassov, V Avati, J Baechler, V Berardi, M Berretti, E Bossini, M Bozzo, P Brogi, et al.
Measurement of proton-proton elastic scattering and total cross-section at.
EPL (Europhysics Letters), 101(2):21002, 2013.
- [31] Frigyes Nemes, T Csörgo, and M Csanád.
Elastic scattering of protons at the totem experiment at the lhc.
Technical report, PhD thesis, Eötvös U., 2015. Presented 15 Oct, 2015.
- [32] G Antchev, P Aspell, I Atanassov, V Avati, J Baechler, V Berardi, M Berretti, E Bossini, U Bottigli, M Bozzo, et al.
Lhc optics measurement with proton tracks detected by the roman pots of the totem experiment.
New Journal of Physics, 16(10):103041, 2014.
- [33] Jan Kaspar.
Analysis elastic 6500 gev, beta 90, 10 sigma.
https://github.com/jan-kaspar/analysis_elastic.6500GeV.beta90.10sigma, 2015.
- [34] Saba Sehrish, Jim Kowalkowski, and Marc Paterno.
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*, pages 1048–1057, 2017.
doi: 10.1109/IPDPSW.2017.112.
URL <http://lss.fnal.gov/archive/2017/pub/fermilab-pub-17-078-cd.pdf>.
- [35] Javier Cervantes.
RDataFrame-TOTEM code, 2018.
URL <https://github.com/Javiercervilla/RDataFrame-TOTEM>.
[Online; accessed 4-September-2018].
- [36] Danilo Piparo Enric Tejedor.
PyROOT Parallelization with Spark.
<https://github.com/etejedor/root-spark>.
[Online; accessed 13-August-2018].
- [37] Google.
Google Summer of Code program.
<https://summerofcode.withgoogle.com/>.
[Online; accessed 25-August-2018].

- [38] Shravan Murali.
Google Summer of Code student.
<https://github.com/shravan97>, .
[Online; accessed 25-August-2018].
- [39] Shravan Murali.
PyRDF : The Python ROOT DataFrame Library.
<https://github.com/shravan97/PyRDF/>, .
[Online; accessed 25-August-2018].
- [40] Jakob Blomer, Predrag Buncic, and Thomas Fuhrmann.
Cernvm-fs: Delivering scientific software to globally distributed computing re-sources.
In *Proceedings of the First International Workshop on Network-aware Data Man-agement*, NDM '11, pages 49–56, New York, NY, USA, 2011. ACM.
ISBN 978-1-4503-1132-8.
doi: 10.1145/2110217.2110225.
URL <http://doi.acm.org/10.1145/2110217.2110225>.
- [41] Martin Gasthuber, Helge Meinhard, and Robert Jones.
HNSciCloud - Overview and technical challenges.
J. Phys. : Conf. Ser., 898(5):052040. 5 p, 2017.
URL <http://cds.cern.ch/record/2297173>.
- [42] AJ Peters, EA Sindrilaru, and G Adde.
EOS as the present and future solution for data storage at CERN.
J. Phys.: Conf. Ser., 664(4):042042. 7 p, 2015.
URL <http://cds.cern.ch/record/2134573>.
- [43] Danilo Piparo, Enric Tejedor, Pere Mato, Luca Mascetti, Jakub Moscicki, and Massimo Lamanna.
SWAN: a Service for Interactive Analysis in the Cloud.
Future Gener. Comput. Syst., 78(CERN-OPEN-2016-005):1071–1078. 17 p, Jun 2016.
URL <http://cds.cern.ch/record/2158559>.
- [44] L Mascetti, H Gonzalez Labrador, M Lamanna, JT Mościcki, and AJ Peters.
CERNBox + EOS: end-user storage for science.
J. Phys.: Conf. Ser., 664(6):062037. 6 p, 2015.
- [45] Markus Zimmermann and for the ALICE collaboration.
The alice analysis train system, 2015.
- [46] The Apache Software Foundation.
Apache Spark, 2018.
URL <https://spark.apache.org/>.
[Online; accessed 30-August-2018].
- [47] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica.
Apache spark: A unified engine for big data processing.

- Commun. ACM*, 59(11):56–65, October 2016.
ISSN 0001-0782.
doi: 10.1145/2934664.
URL <http://doi.acm.org/10.1145/2934664>.
- [48] Saba Sehrish, Jim Kowalkowski, and Marc Paterno.
Exploring the performance of spark for a scientific use case.
In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 1653–1659. IEEE, 2016.
- [49] Apache Spark.
Spark Distributed Analytics Framework at NERSC.
<https://www.nersc.gov/users/data-analytics/data-analytics/spark-distributed-analytic-framework>.
[Online; accessed 13-August-2018].
- [50] Jeffrey Dean and Sanjay Ghemawat.
Mapreduce: simplified data processing on large clusters.
Communications of the ACM, 51(1):107–113, 2008.
- [51] F. Glaser, H. Neukirchen, T. Rings, and J. Grabowski.
Using mapreduce for high energy physics data analysis.
In *2013 IEEE 16th International Conference on Computational Science and Engineering*, pages 1271–1278, Dec 2013.
doi: 10.1109/CSE.2013.189.
- [52] Viktor Khristenko and Jim Pivarski.
diana-hep/spark-root: Release 0.1.14, October 2017.
URL <https://doi.org/10.5281/zenodo.1034230>.
- [53] Travis E Oliphant.
A guide to NumPy, volume 1.
Trelgol Publishing USA, 2006.
- [54] Eric Jones, Travis Oliphant, and Pearu Peterson.
{SciPy}: open source scientific tools for {Python}.
2014.
- [55] Matthew Rocklin.
Dask: Parallel computation with blocked algorithms and task scheduling.
In *Proceedings of the 14th Python in Science Conference*, number 130-136. Citeseer, 2015.
- [56] ROOT Users’ Workshop.
Sarajevo, Bosnia and Herzegovina, 2018.
URL <https://cern.ch/root2018>.
[Online; accessed 1-September-2018].

APPENDIX A

CODE: DISTILL.PY

Full version of the code hosted in:

<https://github.com/JavierCVilla/RDataFrame-Totem>

```
1 import ROOT
2 import glob
3 import sys
4
5 RDF = ROOT.ROOT.RDataFrame
6
7 # Branches clasified by diagonal
8 diagonals = {
9     # return a tuple: ([left] verticals in 45, [right] verticals in 56))
10     "d45b_56t" : ([ "track_rp_5", "track_rp_21", "track_rp_25"],
11                  [ "track_rp_104", "track_rp_120", "track_rp_124"]),
12     "ad45b_56b" : ([ "track_rp_5", "track_rp_21", "track_rp_25"],
13                   [ "track_rp_105", "track_rp_121", "track_rp_125"]),
14     "d45t_56b" : ([ "track_rp_4", "track_rp_20", "track_rp_24"],
15                  [ "track_rp_105", "track_rp_121", "track_rp_125"]),
16     "ad45t_56t" : ([ "track_rp_4", "track_rp_20", "track_rp_24"],
17                   [ "track_rp_104", "track_rp_120", "track_rp_124"])
18 }
19
20 DS = {
21     'DS1' : '4495',
22     'DS2' : '4496',
23     'DS3' : '4499',
24     'DS4' : '4505',
25     'DS5' : '4509',
26     'DS6' : '4510',
27     'DS7' : '4511'
28 }
29
30 threads_description = "no_MT"
31
32 if len(sys.argv) < 3:
33     print('Usage: python distill.py <diagonal> <DS> [threads number]')
34     sys.exit(1) # no diagonal specified
35
36 if len(sys.argv) == 4:
37     if int(sys.argv[3]) < 1:
38         print('Threads number should be > 0')
39         sys.exit(1) # wrong threads number
40     ROOT.ROOT.EnableImplicitMT(int(sys.argv[3]))
41     threads_description = "threads_" + sys.argv[3]
42
43 # Select branches
44 selected_diagonal = sys.argv[1]
45 selected_DS = sys.argv[2]
46 if selected_diagonal not in diagonals.keys():
47     print('Invalid diagonal: %s' % selected_diagonal)
48     print('Choose between: %s' % diagonals.keys())
49     sys.exit(1)
50
```

```
51 if selected_DS not in DS.keys():
52     print('DS not available: %s' % selected_DS)
53     print('Choose between: %s' % DS.keys())
54     sys.exit(1)
55
56 rp_left, rp_right = diagonals[selected_diagonal]
57
58 # Extracted from: DS1/block1/input_files.h
59 source_file = "input_files_{}.txt".format(selected_DS)
60 input_ntuple_name = "TotemNtuple"
61 prefix = "root://eostotem.cern.ch//eos/totem/data/cmstotem/2015/90m/Totem/Ntuple/version2/{}/".
62     format(DS[selected_DS])
63 input_files = [prefix + line.rstrip('\n') for line in open(source_file)]
64
65 # Convert to PyROOT vector
66 vec_input_files = ROOT.vector('string')()
67 [vec_input_files.push_back(f) for f in input_files]
68
69 # Columns per branch
70 attributes = ['valid', 'x', 'y']
71
72 full_branches = ["{}.{}".format(c,a) for a in attributes for c in rp_left+rp_right ]
73
74 # Split left and right branch on valid, x and y
75 valids = [ "(unsigned int) {}".format(v) for v in full_branches[0:6]]
76 xs = full_branches[6:12]
77 ys = full_branches[12:18]
78
79 print("Selected branches: \n" + "\n\t".join(full_branches))
80
81 # Filter and define output branches
82 filter_code = ""({0}.valid + {1}.valid + {2}.valid ) >= 2 &&
83 ({3}.valid + {4}.valid + {5}.valid ) >= 2
84 """".format((rp_left+rp_right))
85
86 print("Filter code: \n" + filter_code)
87
88 # Input tree
89 treename= "TotemNtuple"
90 rdf = RDF(treename, vec_input_files)
91
92 # Output tree, file and branches
93 outTreeName = "distilled"
94 outFileName = "distill_{0}_{1}_{2}_new.root".format(selected_DS, threads_description, selected_diagonal)
95 branchList = ["v_L_1_F", "x_L_1_F", "y_L_1_F",
96             "v_L_2_N", "x_L_2_N", "y_L_2_N",
97             "v_L_2_F", "x_L_2_F", "y_L_2_F",
98             "v_R_1_F", "x_R_1_F", "y_R_1_F",
99             "v_R_2_N", "x_R_2_N", "y_R_2_N",
100             "v_R_2_F", "x_R_2_F", "y_R_2_F",
101             "timestamp",
102             "run_num",
103             "bunch_num",
104             "event_num",
105             "trigger_num",
106             "trigger_bits"
107             ]
108
109 # Convert to PyROOT vector
110 vec_outbranchlist = ROOT.vector('string')()
111 [vec_outbranchlist.push_back(b) for b in branchList]
112
113 # Filter and define output branches
114 r = rdf.Filter(filter_code) \
115     .Define("v_L_1_F", valids[0]) \
116     .Define("x_L_1_F", xs[0]) \
117     .Define("y_L_1_F", ys[0]) \
118     .Define("v_L_2_N", valids[1]) \
119     .Define("x_L_2_N", xs[1]) \
120     .Define("y_L_2_N", ys[1]) \
121     .Define("v_L_2_F", valids[2]) \
122     .Define("x_L_2_F", xs[2]) \
123     .Define("y_L_2_F", ys[2]) \
124     .Define("v_R_1_F", valids[3]) \
```

```

124 .Define("x_R_1_F", xs[3]) \
125 .Define("y_R_1_F", ys[3]) \
126 .Define("v_R_2_N", valids[4]) \
127 .Define("x_R_2_N", xs[4]) \
128 .Define("y_R_2_N", ys[4]) \
129 .Define("v_R_2_F", valids[5]) \
130 .Define("x_R_2_F", xs[5]) \
131 .Define("y_R_2_F", ys[5]) \
132 .Define("timestamp", "(unsigned int) (event_info.timestamp - 1444860000)") \
133 .Define("run_num", "(unsigned int) event_info.run_no") \
134 .Define("bunch_num", "trigger_data.bunch_num") \
135 .Define("event_num", "trigger_data.event_num") \
136 .Define("trigger_num", "trigger_data.trigger_num") \
137 .Define("trigger_bits", "trigger_data.input_status_bits")
138
139 # All above actions are not executed at the moment they are called,
140 # but they are lazy, i.e. delayed until the moment one of their results
141 # is accessed (in this case by .GetValue() )
142 print("Distilled events: %s" % r.Count().GetValue())
143
144 # Save output tree
145 r.Snapshot(outTreeName, outFileName, vec_outbranchlist)

```

Listing A.1: Data reduction using PyROOT and RDataFrame

B

APPENDIX

CODE: DISTRIBUTIONS.PY

Full version of the code hosted in:

<https://github.com/JavierCVilla/RDataFrame-Totem>

```
1 import ROOT
2 import sys
3 import os.path
4
5 # Alias to RDataFrame
6 RDF = ROOT.ROOT.RDataFrame
7
8 # Load C++ headers with data structures and algorithms
9 ROOT.gInterpreter.Declare('#include "common_definitions.h"')
10 ROOT.gInterpreter.Declare('#include "parameters_global.h"')
11 ROOT.gInterpreter.Declare('#include "common_algorithms.h"')
12 ROOT.gInterpreter.Declare('#include "parameters.h"')
13 ROOT.gInterpreter.Declare('#include "common.h"')
14
15 # Pi
16 M_PI = 3.14159265358979323846264338328
17
18 # Parsing command line arguments
19 th_tag="no_MT"
20 if len(sys.argv) < 2:
21     print('Usage: python distributions.py input_filename')
22     sys.exit(1) # no input file specified
23 if len(sys.argv) == 3:
24     nthreads = int(sys.argv[2])
25     if nthreads != 0:
26         th_tag="threads_"+sys.argv[2]
27         ROOT.ROOT.EnableImplicitMT(nthreads)
28
29 # Read input files
30 fname = sys.argv[1]
31
32 # Get input
33 treename = "distilled"
34 selected_diagonal = "d45b_56t"
35 prefix = ""
36 outputDir = "."
37 input_file = prefix + fname # Created with distill.py
38
39 if not os.path.isfile(input_file):
40     print('File does not exists: %s' % input_file)
41     sys.exit(1)
42
43 # Init diagonal settings
44 ROOT.Init("45b_56t");
45
46 # Default parameters
47 detailsLevel = 0 # 0: no details, 1: some details, >= 2 all details
48 overrideCutSelection = False # whether the default cut selection should be overridden
49 cutSelectionString = None
50 outputDir = "."
```

```
51 inputDir          = "."
52 input_n_si        = 4.0
53 time_group_divisor = 0
54 time_group_remainder = 0
55 event_group_divisor = 0
56 event_group_index  = 0
57 evIdxStep          = 1
58 maxTaggedEvents    = 0    # 0 means no maximum
59
60 # Print parameters
61 print(" detailsLevel = %s" % detailsLevel)
62 print(" outputDir = %s" % outputDir)
63 print(" inputDir = %s" % inputDir)
64 print(" input_n_si = %s" % input_n_si)
65 print(" time_group_divisor = %s" % time_group_divisor)
66 print(" time_group_remainder = %s" % time_group_remainder)
67 print(" event_group_divisor = %s" % event_group_divisor)
68 print(" event_group_index = %s" % event_group_index)
69 print(" evIdxStep = %s" % evIdxStep)
70 print(" maxTaggedEvents = %s" % maxTaggedEvents)
71
72 # Select cuts
73 ROOT.anal.BuildCuts()
74 ROOT.anal.n_si = input_n_si
75
76 # Print info
77 print("\n");
78 print("----- environment
79 -----\n");
80 ROOT.env.Print();
81 print("\n");
82 print("----- analysis
83 -----\n");
84 ROOT.anal.Print();
85 print("\n");
86
87 # Alignment
88 for i,_ in enumerate(ROOT.alignmentSources):
89     print("\n----- alignment source %s ----- \n" % i);
90     ROOT.alignmentSources[i].Init();
91
92 print("\n\n");
93
94 # Binnings for histograms
95 binnings = ROOT.vector('string')()
96 binnings.push_back("ub");
97 binnings.push_back("ob-1-10-0.2");
98 binnings.push_back("ob-1-30-0.2");
99
100 ##### READ INPUT FILE, INITIALIZE RDATAFRAME #####
101 #####
102
103 # Read all branches
104 rdf = RDF(treename, input_file)
105
106 # Get time-dependent corrections
107 corr_g_pileup = None
108 if ROOT.anal.use_pileup_efficiency_fits:
109     path = inputDir + "/pileup_fit_combined.root"
110     puF = ROOT.TFile.Open(path)
111     if not os.path.exists(puF):
112         print("ERROR: pile-up correction file '%s' cannot be opened.\n" % path);
113     if diagonal == "d45b_56t":
114         corr_g_pileup = puF.Get("45b_56t/dgn")
115     if diagonal == "d45t_56b":
116         corr_g_pileup = puF.Get("45t_56b/dgn")
117
118 # Get th_y dependent efficiency correction
119 f_3outof4_efficiency_L_F = None;
120 f_3outof4_efficiency_L_N = None;
121 f_3outof4_efficiency_R_N = None;
122 f_3outof4_efficiency_R_F = None;
```

```

123
124 if ROOT.anal.use_3outof4_efficiency_fits:
125     path = inputDir + "/eff3outof4_details_fit_old.root"
126     effFile = ROOT.TFile.Open(path)
127     if (os.path.exists(effFile)):
128         print("ERROR: 3-out-of-4 efficiency file '%s' cannot be opened.\n" % path);
129
130     diagonal = selected_diagonal;
131     f_3outof4_efficiency_L_F = effFile.Get(diagonal + "/L_F/fit");
132     f_3outof4_efficiency_L_N = effFile.Get(diagonal + "/L_N/fit");
133     f_3outof4_efficiency_R_N = effFile.Get(diagonal + "/R_N/fit");
134     f_3outof4_efficiency_R_F = effFile.Get(diagonal + "/R_F/fit");
135
136     print("\n>> using 3-out-of-4 fits: %s, %s, %s, %s\n" %
137           (f_3outof4_efficiency_L_F, f_3outof4_efficiency_L_N,
138            f_3outof4_efficiency_R_N, f_3outof4_efficiency_R_F))
139
140 # Book metadata histograms
141 ROOT.timestamp_bins = ROOT.timestamp_max - ROOT.timestamp_min + 1.;
142
143 # Create bh_t_hists
144 bh_t_Nev_before = dict()
145 bh_t_Nev_after_no_corr = dict()
146 bh_t_before = dict()
147 bh_t_after = dict()
148 bh_t_after_no_corr = dict()
149 bp_t_phi_corr = dict()
150 bp_t_full_corr = dict()
151
152 binning_setup = dict()
153 for b in binnings:
154     binning = ROOT.BuildBinningRDF(ROOT.anal, b)
155     binning_setup[b] = binning
156
157 # Zero counters
158 n_ev_full = 0;
159 n_ev_cut = dict();
160 for ci in range(ROOT.anal.N_cuts):
161     n_ev_cut[ci] = 0
162
163 th_min = 1E100;
164 th_y_L_min = +1E100; th_y_R_min = +1E100
165
166 N_anal=0; N_anal_zeroBias=0;
167 N_zeroBias_el=0; N_zeroBias_el_RP_trig=0;
168 N_4outof4=0; N_el=0;
169 N_el_T2trig=0; N_4outof4_T2trig=0;
170 N_el_raw=0;
171
172
173 #####
174 ##### FILTER, BUILD HISTOGRAMS — START EVENT LOOP #####
175 #####
176
177 # Initial cuts
178 f1 = rdf.Filter("! SkipTime( timestamp )", 'check time — selected')
179
180
181 if time_group_divisor != 0:
182     f1 = f1.Filter("! SkipTimeInterval( timestamp, %s, %s )"
183                   .format(time_group_divisor, time_group_remainder),
184                       'time interval')
185
186 # Diagonal cut
187 f2 = f1.Filter("v_L_2_F && v_L_2_N && v_R_2_F && v_R_2_N", 'allDiagonalRPs')
188
189 # Timestamp histogram
190 model = ("h_timestamp_dgn", ";timestamp;rate (Hz)", int(ROOT.timestamp_bins), ROOT.timestamp_min-0.5,
191         ROOT.timestamp_max+0.5)
192 h_timestamp_dgn = f2.Histo1D(model, "timestamp")
193
194 # Filter zero bias events
195 f_zerobias = f2.Filter("! ((trigger_bits & 512) != 0)", 'zero_bias_event')

```

```

196 xs = ["x_L_1_F", "x_L_2_N", "x_L_2_F", "x_R_1_F", "x_R_2_N", "x_R_2_F"]
197 ys = ["y_L_1_F", "y_L_2_N", "y_L_2_F", "y_R_1_F", "y_R_2_N", "y_R_2_F"]
198
199 # Apply fine alignment
200 r2 = f2.Define("h_al", "ApplyFineAlignment( timestamp ,{}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {})".
    format((xs+ys) )) \
201     .Define("h_al_x_L_1_F", "h_al.L_1_F.x") \
202     .Define("h_al_x_L_2_N", "h_al.L_2_N.x") \
203     .Define("h_al_x_L_2_F", "h_al.L_2_F.x") \
204     .Define("h_al_y_L_1_F", "h_al.L_1_F.y") \
205     .Define("h_al_y_L_2_N", "h_al.L_2_N.y") \
206     .Define("h_al_y_L_2_F", "h_al.L_2_F.y") \
207     .Define("h_al_x_R_1_F", "h_al.R_1_F.x") \
208     .Define("h_al_x_R_2_N", "h_al.R_2_N.x") \
209     .Define("h_al_x_R_2_F", "h_al.R_2_F.x") \
210     .Define("h_al_y_R_1_F", "h_al.R_1_F.y") \
211     .Define("h_al_y_R_2_N", "h_al.R_2_N.y") \
212     .Define("h_al_y_R_2_F", "h_al.R_2_F.y") \
213
214 # Fill pre-selection histograms
215
216 al_nosel_models = map(ROOT.ROOT.RDF.TH2DModel, [
217     ("h_y_L_1_F-vs-x_L_1_F_al_nosel", ";x^{L,1,F};y^{L,1,F}", 150, -15., 15., 300, -30., +30.),
218     ("h_y_L_2_N-vs-x_L_2_N_al_nosel", ";x^{L,2,N};y^{L,2,N}", 150, -15., 15., 300, -30., +30.),
219     ("h_y_L_2_F-vs-x_L_2_F_al_nosel", ";x^{L,2,F};y^{L,2,F}", 150, -15., 15., 300, -30., +30.),
220     ("h_y_R_1_F-vs-x_R_1_F_al_nosel", ";x^{R,1,F};y^{R,1,F}", 150, -15., 15., 300, -30., +30.),
221     ("h_y_R_2_N-vs-x_R_2_N_al_nosel", ";x^{R,2,N};y^{R,2,N}", 150, -15., 15., 300, -30., +30.),
222     ("h_y_R_2_F-vs-x_R_2_F_al_nosel", ";x^{R,2,F};y^{R,2,F}", 150, -15., 15., 300, -30., +30.)
223 ])
224
225 h_y_L_1_F-vs-x_L_1_F_al_nosel = r2.Histo2D(al_nosel_models[0], "h_al_x_L_1_F", "h_al_y_L_1_F")
226 h_y_L_2_N-vs-x_L_2_N_al_nosel = r2.Histo2D(al_nosel_models[1], "h_al_x_L_2_N", "h_al_y_L_2_N")
227 h_y_L_2_F-vs-x_L_2_F_al_nosel = r2.Histo2D(al_nosel_models[2], "h_al_x_L_2_F", "h_al_y_L_2_F")
228 h_y_R_1_F-vs-x_R_1_F_al_nosel = r2.Histo2D(al_nosel_models[3], "h_al_x_R_1_F", "h_al_y_R_1_F")
229 h_y_R_2_N-vs-x_R_2_N_al_nosel = r2.Histo2D(al_nosel_models[4], "h_al_x_R_2_N", "h_al_y_R_2_N")
230 h_y_R_2_F-vs-x_R_2_F_al_nosel = r2.Histo2D(al_nosel_models[5], "h_al_x_R_2_F", "h_al_y_R_2_F")
231
232 # Run reconstruction
233
234 ### Kinematics struct
235 r3 = r2.Define("kinematics", 'DoReconstruction( h_al )')
236
237 r4 = r3.Define("k_th_x_R", "kinematics.th_x_R") \
238     .Define("k_th_y_R", "kinematics.th_y_R") \
239     .Define("k_th_x_L", "kinematics.th_x_L") \
240     .Define("k_th_y_L", "kinematics.th_y_L") \
241     .Define("k_th_x", "kinematics.th_x") \
242     .Define("k_th_y", "kinematics.th_y") \
243     .Define("minus_k_th_y", "- kinematics.th_y") \
244     .Define("k_vtx_x", "kinematics.vtx_x") \
245     .Define("k_vtx_x_L", "kinematics.vtx_x_L") \
246     .Define("k_vtx_x_R", "kinematics.vtx_x_R") \
247     .Define("k_vtx_y", "kinematics.vtx_y") \
248     .Define("k_vtx_y_L", "kinematics.vtx_y_L") \
249     .Define("k_vtx_y_R", "kinematics.vtx_y_R") \
250     .Define("k_th_y_L_F", "kinematics.th_y_L_F") \
251     .Define("k_th_y_L_N", "kinematics.th_y_L_N") \
252     .Define("k_th_y_R_F", "kinematics.th_y_R_F") \
253     .Define("k_th_y_R_N", "kinematics.th_y_R_N") \
254     .Define("k_th_x_diffLR", "kinematics.th_x_R - kinematics.th_x_L") \
255     .Define("k_th_y_diffLR", "kinematics.th_y_R - kinematics.th_y_L") \
256     .Define("k_th_x_diffLF", "kinematics.th_x_L - kinematics.th_x") \
257     .Define("k_th_x_diffRF", "kinematics.th_x_R - kinematics.th_x") \
258     .Define("k_th_y_L_diffNF", "kinematics.th_y_L_F - kinematics.th_y_L_N") \
259     .Define("k_th_y_R_diffNF", "kinematics.th_y_R_F - kinematics.th_y_R_N") \
260     .Define("k_vtx_x_diffLR", "kinematics.vtx_x_R - kinematics.vtx_x_L") \
261     .Define("k_vtx_y_diffLR", "kinematics.vtx_y_R - kinematics.vtx_y_L") \
262     .Define("k_t", "kinematics.t") \
263     .Define("k_th", "kinematics.th") \
264     .Define("k_phi", "kinematics.phi")
265
266 # Cut evaluation
267 r5 = r4.Define("cutdata", "EvaluateCutsRDF( h_al, kinematics )")
268

```

```

269 # Elastic cut
270 f4 = r5.Filter("cutdata.select", "elastic cut")
271
272 # Define normalization and norm_corr colums
273 r6 = r5.Define("norm_corr", "getNorm_corr( timestamp )" ) \
274     .Define("normalization", "getNormalization( norm_corr )" )
275
276 # Fill raw histograms
277 model = ("h_timestamp_sel", ";timestamp;rate (Hz)",
278         int(ROOT.timestamp_bins),
279         ROOT.timestamp_min-0.5,
280         ROOT.timestamp_max+0.5)
281
282 h_timestamp_sel = f4.Histo1D(model, "timestamp");
283
284 # Fill histograms
285 noal_sel_models = map(ROOT.ROOT.RDF.TH2DModel, [
286     ("h_y_L_1_F_vs_x_L_1_F_noal_sel", ";x^{L,1,F};y^{L,1,F}", 100, -3., +3., 300, -30., +30.),
287     ("h_y_L_2_N_vs_x_L_2_N_noal_sel", ";x^{L,2,N};y^{L,2,N}", 100, -3., +3., 300, -30., +30.),
288     ("h_y_L_2_F_vs_x_L_2_F_noal_sel", ";x^{L,2,F};y^{L,2,F}", 100, -3., +3., 300, -30., +30.),
289     ("h_y_R_1_F_vs_x_R_1_F_noal_sel", ";x^{R,1,F};y^{R,1,F}", 100, -3., +3., 300, -30., +30.),
290     ("h_y_R_2_N_vs_x_R_2_N_noal_sel", ";x^{R,2,N};y^{R,2,N}", 100, -3., +3., 300, -30., +30.),
291     ("h_y_R_2_F_vs_x_R_2_F_noal_sel", ";x^{R,2,F};y^{R,2,F}", 100, -3., +3., 300, -30., +30.)
292 ])
293
294 h_y_L_1_F_vs_x_L_1_F_noal_sel = f4.Histo2D(noal_sel_models[0], "x_L_1_F", "y_L_1_F")
295 h_y_L_2_N_vs_x_L_2_N_noal_sel = f4.Histo2D(noal_sel_models[1], "x_L_2_N", "y_L_2_N")
296 h_y_L_2_F_vs_x_L_2_F_noal_sel = f4.Histo2D(noal_sel_models[2], "x_L_2_F", "y_L_2_F")
297 h_y_R_1_F_vs_x_R_1_F_noal_sel = f4.Histo2D(noal_sel_models[3], "x_R_1_F", "y_R_1_F")
298 h_y_R_2_N_vs_x_R_2_N_noal_sel = f4.Histo2D(noal_sel_models[4], "x_R_2_N", "y_R_2_N")
299 h_y_R_2_F_vs_x_R_2_F_noal_sel = f4.Histo2D(noal_sel_models[5], "x_R_2_F", "y_R_2_F")
300
301 al_sel_models = map(ROOT.ROOT.RDF.TH2DModel, [
302     ("h_y_L_1_F_vs_x_L_1_F_al_sel", ";x^{L,1,F};y^{L,1,F}", 100, -3., +3., 300, -30., +30.),
303     ("h_y_L_2_N_vs_x_L_2_N_al_sel", ";x^{L,2,N};y^{L,2,N}", 100, -3., +3., 300, -30., +30.),
304     ("h_y_L_2_F_vs_x_L_2_F_al_sel", ";x^{L,2,F};y^{L,2,F}", 100, -3., +3., 300, -30., +30.),
305     ("h_y_R_1_F_vs_x_R_1_F_al_sel", ";x^{R,1,F};y^{R,1,F}", 100, -3., +3., 300, -30., +30.),
306     ("h_y_R_2_N_vs_x_R_2_N_al_sel", ";x^{R,2,N};y^{R,2,N}", 100, -3., +3., 300, -30., +30.),
307     ("h_y_R_2_F_vs_x_R_2_F_al_sel", ";x^{R,2,F};y^{R,2,F}", 100, -3., +3., 300, -30., +30.)
308 ])
309
310 h_y_L_1_F_vs_x_L_1_F_al_sel = f4.Histo2D(al_sel_models[0], "h_al_x_L_1_F", "h_al_y_L_1_F")
311 h_y_L_2_N_vs_x_L_2_N_al_sel = f4.Histo2D(al_sel_models[1], "h_al_x_L_2_N", "h_al_y_L_2_N")
312 h_y_L_2_F_vs_x_L_2_F_al_sel = f4.Histo2D(al_sel_models[2], "h_al_x_L_2_F", "h_al_y_L_2_F")
313 h_y_R_1_F_vs_x_R_1_F_al_sel = f4.Histo2D(al_sel_models[3], "h_al_x_R_1_F", "h_al_y_R_1_F")
314 h_y_R_2_N_vs_x_R_2_N_al_sel = f4.Histo2D(al_sel_models[4], "h_al_x_R_2_N", "h_al_y_R_2_N")
315 h_y_R_2_F_vs_x_R_2_F_al_sel = f4.Histo2D(al_sel_models[5], "h_al_x_R_2_F", "h_al_y_R_2_F")
316
317 # (k.th_x_R - k.th_x_L)
318 # (k.th_y_R - k.th_y_L)
319 models = map(ROOT.ROOT.RDF.TH1DModel, [
320     ("th_x_diffLR", ";#theta_{x}^{R} - #theta_{x}^{L}", 1000, -500E-6, +500E-6),
321     ("th_y_diffLR", ";#theta_{y}^{R} - #theta_{y}^{L}", 500, -50E-6, +50E-6)
322 ])
323 th_x_diffLR = f4.Histo1D(models[0], "k_th_x_diffLR")
324 th_y_diffLR = f4.Histo1D(models[1], "k_th_y_diffLR")
325
326 # (k.th_x_L - k.th_x)
327 # (k.th_x_R - k.th_x)
328 models = map(ROOT.ROOT.RDF.TH1DModel, [
329     ("th_x_diffLF", ";#theta_{x}^{L} - #theta_{x}^{R}", 400, -200E-6, +200E-6),
330     ("th_x_diffRF", ";#theta_{x}^{R} - #theta_{x}^{L}", 400, -200E-6, +200E-6)
331 ])
332 th_x_diffLF = f4.Histo1D(models[0], "k_th_x_diffLF")
333 th_x_diffRF = f4.Histo1D(models[1], "k_th_x_diffRF")
334
335 # (k.th_x, k.th_x_R - k.th_x_L)
336 # (k.th_y, k.th_y_R - k.th_y_L)
337 # (k.vtx_x, k.th_x_R - k.th_x_L)
338 models = map(ROOT.ROOT.RDF.TH2DModel, [
339     ("h_th_x_diffLR_vs_th_x", ";#theta_{x};#theta_{x}^{R} - #theta_{x}^{L}", 100, -300E-6, +300E-6, 120,
340         -120E-6, +120E-6),
341     ("h_th_y_diffLR_vs_th_y", ";#theta_{y};#theta_{y}^{R} - #theta_{y}^{L}", 100, -500E-6, +500E-6, 120,
342         -120E-6, +120E-6),

```

```

341     ("h_th_x_diffLR_vs_vtx_x", ";vtx_{x};#theta_{x}^{L} - #theta_{x}^{R}", 100, -300E-3, +300E-3, 120,
    -120E-6, +120E-6)
342 ])
343 h_th_x_diffLR_vs_th_x = f4.Histo2D(models[0], "k_th_x", "k_th_x_diffLR")
344 h_th_y_diffLR_vs_th_y = f4.Histo2D(models[1], "k_th_y", "k_th_y_diffLR")
345 h_th_x_diffLR_vs_vtx_x = f4.Histo2D(models[2], "k_vtx_x", "k_th_x_diffLR")
346
347 # (k.th_x_L, k.th_y_L)
348 # (k.th_x_R, k.th_y_R)
349 # (k.th_x, k.th_y)
350 models = map(ROOT.ROOT.RDF.TH2DModel, [
351     ("h_th_y_L_vs_th_x_L", ";#theta_{x}^{L};#theta_{y}^{L}", 100, -115E-6, +11E-5, 100, 22E-6, +102E-6)
    ,
352     ("h_th_y_R_vs_th_x_R", ";#theta_{x}^{R};#theta_{y}^{R}", 100, -125E-6, +12E-5, 100, 27E-6, +102E-6)
    ,
353     ("h_th_y_vs_th_x", ";#theta_{x};#theta_{y}", 100, -300E-6, +300E-6, 100, -150E-6, +150E-6)
354 ])
355 h_th_y_L_vs_th_x_L = f4.Histo2D(models[0], "k_th_x_L", "k_th_y_L")
356 h_th_y_R_vs_th_x_R = f4.Histo2D(models[1], "k_th_x_R", "k_th_y_R")
357 h_th_y_vs_th_x = f4.Histo2D(models[2], "k_th_x", "k_th_y")
358
359 # (k.th_y_R, k.th_y_L)
360 model = ROOT.ROOT.RDF.TH2DModel("h_th_y_L_vs_th_y_R", ";#theta_{y}^{R};#theta_{y}^{L}",
361     300, -150E-6, +150E-6, 300, -150E-6, +150E-6)
362 h_th_y_L_vs_th_y_R = f4.Histo2D(model, "k_th_y_R", "k_th_y_L")
363
364 # (k.th_x)
365 # (k.th_y)
366 models = map(ROOT.ROOT.RDF.TH1DModel, [
367     ("h_th_x", ";#theta_{x}", 250, -500E-6, +500E-6),
368     ("h_th_y", ";#theta_{y}", 250, -500E-6, +500E-6)
369 ])
370 h_th_x = f4.Histo1D(models[0], "k_th_x")
371 h_th_y = f4.Histo1D(models[1], "k_th_y")
372
373 # (-k.th_y)
374 model = ("h_th_y_flipped", ";#theta_{y}", 250, -500E-6, +500E-6)
375 h_th_y_flipped = f4.Histo1D(model, "minus_k_th_y")
376
377 # (k.th_x_L)
378 # (k.th_x_R)
379 models = map(ROOT.ROOT.RDF.TH1DModel, [
380     ("h_th_x_L", ";#theta_{x}^{L}", 250, -500E-6, +500E-6),
381     ("h_th_x_R", ";#theta_{x}^{R}", 250, -500E-6, +500E-6)
382 ])
383 h_th_x_L = f4.Histo1D(models[0], "k_th_x_L")
384 h_th_x_R = f4.Histo1D(models[1], "k_th_x_R")
385
386 # (k.th_y_L)
387 # (k.th_y_R)
388 models = map(ROOT.ROOT.RDF.TH1DModel, [
389     ("h_th_y_L", ";#theta_{y}^{L}", 250, -500E-6, +500E-6),
390     ("h_th_y_R", ";#theta_{y}^{R}", 250, -500E-6, +500E-6)
391 ])
392 h_th_y_L = f4.Histo1D(models[0], "k_th_y_L")
393 h_th_y_R = f4.Histo1D(models[1], "k_th_y_R")
394
395 # (k.th_y_L_F)
396 # (k.th_y_L_N)
397 # (k.th_y_R_N)
398 # (k.th_y_R_F)
399 models = map(ROOT.ROOT.RDF.TH1DModel, [
400     ("h_th_y_L_F", ";#theta_{y}^{L_F}", 250, -500E-6, +500E-6),
401     ("h_th_y_L_N", ";#theta_{y}^{L_N}", 250, -500E-6, +500E-6),
402     ("h_th_y_R_N", ";#theta_{y}^{R_N}", 250, -500E-6, +500E-6),
403     ("h_th_y_R_F", ";#theta_{y}^{R_F}", 250, -500E-6, +500E-6)
404 ])
405 h_th_y_L_F = f4.Histo1D(models[0], "k_th_y_L_F")
406 h_th_y_L_N = f4.Histo1D(models[1], "k_th_y_L_N")
407 h_th_y_R_N = f4.Histo1D(models[2], "k_th_y_R_N")
408 h_th_y_R_F = f4.Histo1D(models[3], "k_th_y_R_F")
409
410
411 # Fill vertex histograms

```

```

412
413 # (k.vtx_x)
414 # (k.vtx_x_L)
415 # (k.vtx_x_R)
416 models = map(ROOT.ROOT.RDF.TH1DModel, [
417     ("h_vtx_x", ";x^{", 100, -0.5, +0.5),
418     ("h_vtx_x_L", ";x^{,L}", 100, -0.5, +0.5),
419     ("h_vtx_x_R", ";x^{,R}", 100, -0.5, +0.5)
420 ])
421 h_vtx_x = f4.Histo1D(models[0], "k_vtx_x")
422 h_vtx_x_L = f4.Histo1D(models[1], "k_vtx_x_L")
423 h_vtx_x_R = f4.Histo1D(models[2], "k_vtx_x_R")
424
425 # (k.vtx_y)
426 # (k.vtx_y_L)
427 # (k.vtx_y_R)
428 models = map(ROOT.ROOT.RDF.TH1DModel, [
429     ("h_vtx_y", ";y^{", 100, -0.5, +0.5),
430     ("h_vtx_y_L", ";y^{,L}", 100, -0.5, +0.5),
431     ("h_vtx_y_R", ";y^{,R}", 100, -0.5, +0.5)
432 ])
433 h_vtx_y = f4.Histo1D(models[0], "k_vtx_y")
434 h_vtx_y_L = f4.Histo1D(models[1], "k_vtx_y_L")
435 h_vtx_y_R = f4.Histo1D(models[2], "k_vtx_y_R")
436
437 # (k.vtx_x_R, k.vtx_x_L)
438 # (k.vtx_y_R, k.vtx_y_L)
439 models = map(ROOT.ROOT.RDF.TH2DModel, [
440     ("h_vtx_x_L-vs-vtx_x_R", ";x^{,R};x^{,L}", 100, -0.5, +0.5, 100, -0.5, +0.5),
441     ("h_vtx_y_L-vs-vtx_y_R", ";y^{,R};y^{,L}", 100, -0.5, +0.5, 100, -0.5, +0.5)
442 ])
443 h_vtx_x_L-vs-vtx_x_R = f4.Histo2D(models[0], "k_vtx_x_R", "k_vtx_x_L")
444 h_vtx_y_L-vs-vtx_y_R = f4.Histo2D(models[1], "k_vtx_y_R", "k_vtx_y_L")
445
446 # (k.th_x_L, k.vtx_x_L)
447 # (k.th_x_R, k.vtx_x_R)
448 # (k.th_y_L, k.vtx_y_L)
449 # (k.th_y_R, k.vtx_y_R)
450 models = map(ROOT.ROOT.RDF.TH2DModel, [
451     ("h_vtx_x_L-vs-th_x_L", ";#theta_{x}^{L};x^{,L}", 100, -600E-6, +600E-6, 100, -0.5, +0.5),
452     ("h_vtx_x_R-vs-th_x_R", ";#theta_{x}^{R};x^{,R}", 100, -600E-6, +600E-6, 100, -0.5, +0.5),
453     ("h_vtx_y_L-vs-th_y_L", ";#theta_{y}^{L};y^{,L}", 100, -600E-6, +600E-6, 100, -0.5, +0.5),
454     ("h_vtx_y_R-vs-th_y_R", ";#theta_{y}^{R};y^{,R}", 100, -600E-6, +600E-6, 100, -0.5, +0.5)
455 ])
456 h_vtx_x_L-vs-th_x_L = f4.Histo2D(models[0], "k_th_x_L", "k_vtx_x_L")
457 h_vtx_x_R-vs-th_x_R = f4.Histo2D(models[1], "k_th_x_R", "k_vtx_x_R")
458 h_vtx_y_L-vs-th_y_L = f4.Histo2D(models[2], "k_th_y_L", "k_vtx_y_L")
459 h_vtx_y_R-vs-th_y_R = f4.Histo2D(models[3], "k_th_y_R", "k_vtx_y_R")
460
461 # (k.vtx_x_R - k.vtx_x_L)
462 # (k.vtx_y_R - k.vtx_y_L)
463 models = map(ROOT.ROOT.RDF.TH1DModel, [
464     ("h_vtx_x_diffLR", ";x^{,R} - x^{,L}", 100, -0.5, +0.5),
465     ("h_vtx_y_diffLR", ";y^{,R} - y^{,L}", 100, -0.5, +0.5)
466 ])
467 h_vtx_x_diffLR = f4.Histo1D(models[0], "k_vtx_x_diffLR");
468 h_vtx_y_diffLR = f4.Histo1D(models[1], "k_vtx_y_diffLR");
469
470 # (k.th_x, k.vtx_x_R - k.vtx_x_L)
471 # (k.th_y, k.vtx_y_R - k.vtx_y_L)
472 models = map(ROOT.ROOT.RDF.TH1DModel, [
473     ("h_vtx_x_diffLR", ";x^{,R} - x^{,L}", 100, -0.5, +0.5),
474     ("h_vtx_y_diffLR", ";y^{,R} - y^{,L}", 100, -0.5, +0.5)
475 ])
476 h_vtx_x_diffLR-vs-th_x = f4.Histo1D(models[0], "k_th_x", "k_vtx_x_diffLR");
477 h_vtx_y_diffLR-vs-th_y = f4.Histo1D(models[1], "k_th_y", "k_vtx_y_diffLR");
478
479 # (k.vtx_x_R, k.vtx_x_R - k.vtx_x_L)
480 # (k.vtx_y_R, k.vtx_y_R - k.vtx_y_L)
481 models = map(ROOT.ROOT.RDF.TH2DModel, [
482     ("h_vtx_x_diffLR-vs-vtx_x_R", ";x^{,R};x^{,R} - x^{,L}", 100, -0.5, +0.5, 100, -0.5, +0.5),
483     ("h_vtx_y_diffLR-vs-vtx_y_R", ";y^{,R};y^{,R} - y^{,L}", 100, -0.5, +0.5, 100, -0.5, +0.5)
484 ])
485 h_vtx_x_diffLR-vs-vtx_x_R = f4.Histo2D(models[0], "k_vtx_x_R", "k_vtx_y_diffLR");

```

```

486 h_vtx_y_diffLR_vs_vtx_y_R = f4.Histo2D(models[1], "k_vtx_y_R", "k_vtx_y_diffLR");
487
488 # Calculate acceptance divergence correction
489 r7 = f4.Define("correction", "CalculateAcceptanceCorrectionsRDF( kinematics )") \
490     .Define("corr", "correction.corr") \
491     .Define("div_corr", "correction.div_corr") \
492     .Define("one", "One()")
493
494 for bi in binnings:
495     bis = binning_setup[bi]
496
497     model = ROOT.RDF.TH1DModel("h_t_Nev_before", ";|t|;events per bin", bis.N_bins, bis.bin_edges)
498     bh_t_Nev_before[bi] = r7.Histo1D(model, "k_t", "one");
499
500     model = ROOT.RDF.TH1DModel("h_t_before", ";|t|", bis.N_bins, bis.bin_edges)
501     bh_t_before[bi] = r7.Histo1D(model, "k_t", "one");
502
503
504 model = ROOT.RDF.TH2DModel("h_th_y_vs_th_x_before", ";#theta_{x};#theta_{y}", 150, -300E-6, +300E-6, 150,
    -150E-6, +150E-6)
505 h_th_y_vs_th_x_before = r7.Histo2D(model, "k_th_x", "k_th_y", "one");
506
507 # Filter skip
508 f5 = r7.Filter("! correction.skip", "acceptance correction")
509
510 for bi in binnings:
511     bis = binning_setup[bi]
512
513     model = ROOT.RDF.TH1DModel("h_t_Nev_after_no_corr", ";|t|;events per bin", bis.N_bins, bis.bin_edges)
514     bh_t_Nev_after_no_corr[bi] = f5.Histo1D(model, "k_t", "one");
515
516     model = ROOT.RDF.TH1DModel("h_t_after_no_corr", ";|t|", bis.N_bins, bis.bin_edges)
517     bh_t_after_no_corr[bi] = f5.Histo1D(model, "k_t", "one");
518
519     model = ROOT.RDF.TH1DModel("h_t_after", ";|t|", bis.N_bins, bis.bin_edges)
520     bh_t_after[bi] = f5.Histo1D(model, "k_t", "corr");
521
522 model = ROOT.RDF.TH2DModel("h_th_y_vs_th_x_after", ";#theta_{x};#theta_{y}", 150, -300E-6, +300E-6, 150,
    -150E-6, +150E-6);
523 h_th_y_vs_th_x_after = f5.Histo2D(model, "k_th_x", "k_th_y", "div_corr");
524
525 model = ROOT.RDF.TH2DModel("h_th_vs_phi_after", ";#phi;#theta", 50, -M_PI, +M_PI, 50, 150E-6, 550E-6);
526 h_th_vs_phi_after = f5.Histo2D(model, "k_phi", "k_th", "div_corr");
527
528 # Apply normalization
529 model = ROOT.RDF.TH1DModel("h_t_normalized", ";|t|", 128, 0., 4.)
530 bh_t_normalized_ob_1_30_02 = f5.Define("corr_norm", "corr normalization") \
531     .Histo1D(model, "k_t", "corr_norm")
532
533 model = ROOT.RDF.TH2DModel("h_th_y_vs_th_x_normalized", ";#theta_{x};#theta_{y}", 150, -600E-6, +600E-6,
    150, -600E-6, +600E-6);
534 h_th_y_vs_th_x_normalized = f5.Define("div_corr_norm", "correction.div_corr normalization") \
535     .Histo2D(model, "k_th_x", "k_th_y", "div_corr_norm");
536
537 for bi in binnings:
538     bh_t_before[bi].Scale(1., "width");
539     bh_t_after_no_corr[bi].Scale(1., "width");
540     bh_t_after[bi].Scale(1., "width");
541
542 #####
543 ##### END OF EVENT LOOP #####
544 #####
545
546
547 print("-----after event loop
    -----\n");
548
549 print(">>> th_min = %s\n" % th_min);
550 print(">>> th_y_L_min = %s\n" % th_y_L_min);
551 print(">>> th_y_R_min = %s\n" % th_y_R_min);
552
553 print("\n");
554 print("N_anal = %s\n" % N_anal);
555 print("N_anal_zeroBias = %s\n" % N_anal_zeroBias);

```

```

556 print("N_zeroBias_el = %s\n" % N_zeroBias_el);
557 print("N_zeroBias_el_RP_trig = %s\n" % N_zeroBias_el_RP_trig);
558
559 print("N_4outof4 = %s\n" % N_4outof4);
560 print("N_el = %s\n" % N_el);
561 print("N_el_T2trig = %s\n" % N_el_T2trig);
562 print("N_4outof4_T2trig = %s\n" % N_4outof4_T2trig);
563
564 th_y_diffLR.Scale(1., "width");
565 th_x_diffLR.Scale(1., "width");
566
567
568 #####
569 #####  SAVE TO DISK  #####
570 #####
571
572
573 # Save histograms
574 c = ROOT.TCanvas();
575
576 outputname = "/distributions_{ }_{ }_{ }.root".format("DS1", th_tag, selected_diagonal)
577 outF = ROOT.TFile.Open(outputDir+outputname, "recreate");
578 ROOT.gDirectory = outF.mkdir("metadata");
579
580 c = ROOT.TCanvas("rate cmp");
581 h_timestamp_dgn.Draw();
582 h_timestamp_sel.SetLineColor(2);
583 h_timestamp_sel.Draw("sames");
584 c.Write();
585
586 hitDistDir = outF.mkdir("hit distributions");
587 ROOT.gDirectory = hitDistDir.mkdir("vertical, aligned, before selection");
588 h_y_L_2_F_vs_x_L_2_F_al_nosel.Write();
589 h_y_L_2_N_vs_x_L_2_N_al_nosel.Write();
590 h_y_L_1_F_vs_x_L_1_F_al_nosel.Write();
591 h_y_R_1_F_vs_x_R_1_F_al_nosel.Write();
592 h_y_R_2_N_vs_x_R_2_N_al_nosel.Write();
593 h_y_R_2_F_vs_x_R_2_F_al_nosel.Write();
594
595 ROOT.gDirectory = hitDistDir.mkdir("vertical, not aligned, after selection");
596 h_y_L_2_F_vs_x_L_2_F_noal_sel.Write();
597 h_y_L_2_N_vs_x_L_2_N_noal_sel.Write();
598 h_y_L_1_F_vs_x_L_1_F_noal_sel.Write();
599 h_y_R_1_F_vs_x_R_1_F_noal_sel.Write();
600 h_y_R_2_N_vs_x_R_2_N_noal_sel.Write();
601 h_y_R_2_F_vs_x_R_2_F_noal_sel.Write();
602
603 ROOT.gDirectory = hitDistDir.mkdir("vertical, aligned, after selection");
604 h_y_L_2_F_vs_x_L_2_F_al_sel.Write();
605 h_y_L_2_N_vs_x_L_2_N_al_sel.Write();
606 h_y_L_1_F_vs_x_L_1_F_al_sel.Write();
607 h_y_R_1_F_vs_x_R_1_F_al_sel.Write();
608 h_y_R_2_N_vs_x_R_2_N_al_sel.Write();
609 h_y_R_2_F_vs_x_R_2_F_al_sel.Write();
610
611 ROOT.gDirectory = outF.mkdir("selected - hits");
612 ROOT.gDirectory = outF.mkdir("selected - angles");
613
614 th_x_diffLR.Sumw2(); th_x_diffLR.Write()
615 th_y_diffLR.Sumw2(); th_y_diffLR.Write()
616
617 th_x_diffLF.Sumw2(); th_x_diffLF.Write()
618 th_x_diffRF.Sumw2(); th_x_diffRF.Write()
619
620 h_th_x_diffLR_vs_th_x.Write();
621 h_th_y_diffLR_vs_th_y.Write();
622 h_th_x_diffLR_vs_vtx_x.Write();
623
624 h_th_y_L_vs_th_x_L.Write();
625 h_th_y_R_vs_th_x_R.Write();
626 h_th_y_vs_th_x.Write();
627
628 c = ROOT.TCanvas();
629 c.SetLogz(1);

```

```
630 c.ToggleEventStatus();
631 c.SetCrosshair(1);
632 h_th_y_L_vs_th_y_R.Draw("colz");
633 c.Write("canvas_th_y_L_vs_th_y_R");
634
635 h_th_x.SetLineColor(1); h_th_x.Write();
636 h_th_y.SetLineColor(1); h_th_y.Write();
637 h_th_y_flipped.SetLineColor(1); h_th_y_flipped.Write();
638
639 h_th_x_L.SetLineColor(2); h_th_x_L.Write();
640 h_th_x_R.SetLineColor(4); h_th_x_R.Write();
641
642 h_th_y_L.SetLineColor(2); h_th_y_L.Write();
643 h_th_y_R.SetLineColor(4); h_th_y_R.Write();
644
645 h_th_y_L_F.SetLineColor(2); h_th_y_L_F.Write();
646 h_th_y_L_N.SetLineColor(6); h_th_y_L_N.Write();
647 h_th_y_R_N.SetLineColor(4); h_th_y_R_N.Write();
648 h_th_y_R_F.SetLineColor(7); h_th_y_R_F.Write();
649
650 ROOT.gDirectory = outF.mkdir("selected - vertex");
651 h_vtx_x.SetLineColor(1); h_vtx_x.Write();
652 h_vtx_x_L.SetLineColor(2); h_vtx_x_L.Write();
653 h_vtx_x_R.SetLineColor(4); h_vtx_x_R.Write();
654
655 h_vtx_y.SetLineColor(1); h_vtx_y.Write();
656 h_vtx_y_L.SetLineColor(2); h_vtx_y_L.Write();
657 h_vtx_y_R.SetLineColor(4); h_vtx_y_R.Write();
658
659 h_vtx_x_L_vs_vtx_x_R.Write();
660 h_vtx_y_L_vs_vtx_y_R.Write();
661
662 h_vtx_x_L_vs_th_x_L.Write();
663 h_vtx_x_R_vs_th_x_R.Write();
664 h_vtx_y_L_vs_th_y_L.Write();
665 h_vtx_y_R_vs_th_y_R.Write();
666
667 h_vtx_x_diffLR.Sumw2(); h_vtx_x_diffLR.Write();
668 h_vtx_y_diffLR.Sumw2(); h_vtx_y_diffLR.Write();
669
670 h_vtx_x_diffLR_vs_th_x.Write();
671 h_vtx_y_diffLR_vs_th_y.Write();
672
673 h_vtx_x_diffLR_vs_vtx_x_R.Write();
674 h_vtx_y_diffLR_vs_vtx_y_R.Write();
675
676 ROOT.gDirectory = outF.mkdir("optics");
677
678 accDir = outF.mkdir("acceptance correction");
679 for bi in binnings:
680     ROOT.gDirectory = accDir.mkdir(bi);
681     bh_t_Nev_before[bi].Sumw2(); bh_t_Nev_before[bi].Write();
682     bh_t_Nev_after_no_corr[bi].Sumw2(); bh_t_Nev_after_no_corr[bi].Write();
683     bh_t_before[bi].Sumw2(); bh_t_before[bi].Write();
684     bh_t_after_no_corr[bi].Sumw2(); bh_t_after_no_corr[bi].Write();
685     bh_t_after[bi].Sumw2(); bh_t_after[bi].Write();
686
687 ROOT.gDirectory = accDir;
688
689 h_th_y_vs_th_x_before.Sumw2(); h_th_y_vs_th_x_before.Write();
690 h_th_y_vs_th_x_after.Sumw2(); h_th_y_vs_th_x_after.Write();
691 h_th_vs_phi_after.Sumw2(); h_th_vs_phi_after.Write();
```

Listing B.1: Filtering based on physics cuts and production of histograms using PyROOT and RDataFrame

ROOT WORKSHOP PRESENTATION

Part of the results obtained on this thesis will be presented on the ROOT Users' Workshop held at Sarajevo (Bosnia and Herzegovina). The abstract acceptance mail is shown below:

Subject: [Indico] ROOT 2018 Abstract Acceptance notification (#12)
From: <noreply-indico-team@cern.ch>
Date: 02/07/2018, 23:12
To: <javier.cervantes.villanueva@cern.ch>, <enric.tejedor.saavedra@cern.ch>

Dear Enric Tejedor Saavedra,

We're pleased to announce that your abstract "Overtaking PROOF: The Future of Distributed Analysis in ROOT" with ID #12 has been accepted as an oral presentation of 15 minutes.

See below a summary of your submitted abstract:
Submitted by: Enric Tejedor Saavedra
Title: Overtaking PROOF: The Future of Distributed Analysis in ROOT
Primary Authors: Enric Tejedor Saavedra, Javier Cervantes Villanueva
Co-authors:
Presentation type: oral

For a more detailed summary please visit the page of your abstract:
<https://indico.cern.ch/event/697389/abstracts/92071/>.

Kind regards,
The organizers of ROOT Users' Workshop 2018

--

Indico :: Call for Abstracts
<https://indico.cern.ch/event/697389/>

APPENDIX D

UCC 2018

An abstract for the 11th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2018) has been proposed with the results of the collaboration. The abstract submission mail is shown below:

----- Forwarded Message -----
Subject: UCC 2018 - Posters submission 3
Date: Fri, 31 Aug 2018 19:35:37 +0200
From: UCC 2018 - Posters <ucc2018posters@easychair.org>
To: Maciej Malawski <malawski@agh.edu.pl>

Dear authors,

We received your paper:

Authors : Valentina Avati, Milosz Blaszkiewicz, Enrico Bocchi, Luca Canali,
Diogo Castro, Javier Cervantes, Leszek Grzanka, Enrico Guiraud,
Jan Kaspar, Prasanth Kothuri, Massimo Lamanna, Maciej Malawski,
Aleksandra Mnich, Jakub Moscicki, Shravan Murali, Danilo Piparo
and Enric Tejedor
Title : Processing High Energy Physics Data in the Cloud - Experience with
Helix Nebula, ROOT, Spark, SWAN and TOTEM Experiment
Number : 3

The paper was submitted by Maciej Malawski <malawski@agh.edu.pl>.

Thank you for submitting to UCC 2018 - Posters.

Best regards,
EasyChair for UCC 2018 - Posters.