



Università degli Studi di Milano-Bicocca

Dipartimento di Informatica, Sistemistica e Comunicazione

Corso di Laurea Magistrale in Data Science

Blurring High Energy Physics Data Analysis Techniques and Data Science Approaches

Relatore: Prof. Marco Paganoni

Co-relatore: Dott. Danilo Piparo

Tesi di Laurea Magistrale di:

Vincenzo Eduardo Padulano

Matricola 789450

Anno Accademico 2018-2019



Acknowledgements

I thank Professor Marco Paganoni with deepest gratitude. His passion for research and cooperation has guided me through time and he has always inspired me to push forward despite of the hardships.

I would like to thank all the brilliant scientists I have cooperated with for this thesis. First of all, to Dr. Danilo Piparo for welcoming and guiding me during my first contributions at CERN. To Javier Cervantes Villanueva, he has been the most supportive supervisor, letting me express myself at work but at the same time steering me in the right direction whenever I needed it the most. To Dr. Enric Tejedor Saavedra, for the invaluable help in some tricky situations. And finally to all the EP-SFT team, for their incredible positive attitude towards me and my work from day one.

To Dalila, words can only do so much to describe how important you are to me. You have been key in all these months of hard work, thank you.

To all the family and friends I am so lucky to have. Whether you were close or far away, your continuous support has strengthened me in so many different ways.

Abstract

Scientific research has always been intertwined to a certain degree with Computing. Even more so over the last few years, during which the needs for resources in terms of storage and processing power have increased exponentially. This holds true for many different joint collaborations in fields such as biology, medicine, earth sciences, physics and astrophysics, among which CERN definitely represents a notable example. Being the largest centre for research in the High Energy Physics (HEP) field, it has always kept pushing for new discoveries in its executive program. The collaborative efforts of thousands of scientists worldwide have led to important results, most notably in recent years the discovery of the Higgs boson, officially announced in 2012 by the researchers at CMS and ATLAS, the two main experiments taking data at the LHC collider. This strenuous work demands the most advanced technological instruments to recreate the physics events and at the same time hardware and software that keep up with the computing needs. But while HEP has been historically at the forefront in developing solutions to cope up with these requirements, in the recent years other fields and industries have experienced steady advances, helped by an unprecedented abundance of data. The research field born to exploit data, namely Data Science, has brought to the table new computing techniques that may well fit the needs of HEP. In this thesis, a programming model commonly used in Data Science, namely MapReduce, will be exploited to work with the most prominent software for HEP analysis, ROOT. The first will be used in the implementation available under the Apache Spark framework to allow for distributing computations over a remote cluster, while the latter will provide the interface to common HEP data formats and analysis models through one of its latest additions, namely RDataFrame. PyRDF, a purposely developed package, will glue all the components together and will be used to showcase how this new model can affect the workflow of a physics analysis.

Contents

Acknowledgements	ii
Abstract	iii
Contents	iv
1 Introduction	1
1.1 Context	1
1.2 Motivation	5
2 Data lifecycle at CERN	10
2.1 Generation	11
2.2 Collection	12
2.3 Processing	14
2.4 Storage	15
2.4.1 Grid technologies in HEP	16
2.5 Management	21
2.6 Analysis	22
2.7 Visualization	24
2.8 Interpretation	26
3 Tools and methodologies	27
3.1 ROOT	27

3.1.1	ROOT data model	29
3.1.2	ROOT RDataFrame	35
3.1.3	Automatically generated Python bindings for ROOT	40
3.2	Apache Spark	40
3.2.1	Spark components	41
3.2.2	Spark program overview	42
3.3	SWAN	43
3.3.1	Storage	45
3.3.2	Software	46
3.3.3	Computing	47
3.4	PyRDF	47
3.4.1	Distributing ROOT RDataFrame with Spark	48
3.5	Blending everything together	51
4	Distributing physics analyses with Spark and RDataFrame	53
4.1	Dimuon spectrum	54
4.1.1	Conversion to PyRDF	55
4.1.2	PyROOT vs PyRDF comparison	56
4.1.3	Scalability of the analysis	58
4.2	Higgs boson decay to two τ leptons	62
4.2.1	Conversion to PyRDF	64
4.2.2	Comparison with original analysis	67
4.3	VBS Analysis	71
4.3.1	Conversion to PyRDF	74
4.3.2	Results	77
5	Related Work	78
5.1	TOTEM distributed analysis on AOD data	78
5.1.1	Integrating two different software stacks	79
6	Conclusions	81

6.1 Presentations	82
Bibliography	83

List of Figures

1.1	The LHC accelerator complex. [3]	3
1.2	LHC project schedule [26].	7
2.1	Physics data stored within the CASTOR system.	15
2.2	Topology of the WLCG tiers.[40]	20
2.3	Histogram plots produced using ROOT presented for the announcement of the Higgs boson discovery, taken from the official ROOT gallery [51].	25
3.1	Diagram layout of the ROOT modules [53].	29
3.2	Physical layout of a TFile [54].	31
3.3	Logical layout explaining the connections between TFile, TKey, TDirectory and the actual data contained in the file [55].	32
3.4	The ROOT tree data structure [56].	34
3.5	The Apache Spark ecosystem.	42
3.6	Components of a Spark program from the perspective of the cluster [62].	43
3.7	An overview of the main components in SWAN. From the user perspective (top of the figure) SWAN presents as a web portal with easy access to files and to the Jupyter notebooks for programming. The user can login to the portal with her CERN credentials, as a consequence a new Docker container will be spawned. The container will be automatically hooked to a suite of CERN services for software and storage. Finally, inside a Jupyter notebook, the user will be able to connect to external computing resources such as the Spark cluster.	44

3.8	Python object layout created by the code in Listing 4.4. The blue boxes represent nodes in the graph, while the green ones show the corresponding proxies. When a user access the last action node in the graph (an histogram node in this case), the ActionProxy triggers the execution of the whole graph and the achieved result is given to the user.	50
4.1	Di-muon spectrum plot resulting from the analysis on CMS open data derived from Run B and C in 2012.	54
4.2	Execution time of 1000 Dimuon analysis runs in a SWAN session. a: multithreaded configuration with 4 threads. b: Spark configuration with 4 workers, 1 core each.	57
4.3	Execution time (in seconds) of the Dimuon analysis run on the Spark clusters with different number of cores. The blue dots represent the average time of 50 runs.	58
4.4	Speedup of adding more cores for running the Dimuon analysis. The green dashed line shows the ideal scaling, that is linear with the number of cores.	59
4.5	Execution time (in minutes) of the Dimuon analysis with the augmented input dataset, with increasing number of cores. The blue dots represent the average time of 20 runs.	61
4.6	Speedup of adding more cores for running the Dimuon analysis with the augmented input dataset. The green dashed line shows the ideal scaling, that is linear with the number of cores.	61
4.7	Sample plot of the Higgs to τ analysis, showing the mass distributions of the particles and the various background data.	62
4.8	Average Execution time of 20 runs of the Higgs to τ analysis. On the left side of the plot the result of the C++ multithreaded execution is shown, while on the right side the different Spark executor configurations are shown.	68

4.9	Average Execution time of 20 runs of the Higgs to τ analysis, after repartitioning the input datasets. On the left side of the plot the result of the C++ multithreaded execution is shown, while on the right side the different Spark executor configurations are shown.	69
4.10	Speedup using the Run2012C_TauPlusX dataset, average of 20 runs of the Higgs to τ analysis.	70
4.11	Feynman diagram of a VBS process contributing to the EW-induced production of events containing a hadronically decaying gauge boson (V) and a $W^\pm$ boson decaying into a lepton (l) and two jets (q/q') [71].	72
4.12	Workflow of the VBS analysis. The blue boxes represent transformations on data, while the green ones represent actions. The results can be retrieved at different levels of the computational graph, according to the users' needs. .	75

List of Tables

2.1	Event data sizes for different data formats in use by the various experiments in LHC. [46]	23
4.1	Description of the variables in the di-muon dataset. The square bracket notation in the Type column indicates the length of the corresponding vector [68].	55
4.2	Summary statistics for the different files used in the Higgs to τ leptons analysis.	63
4.3	Summary of the data available for the analysis in the year 2017.	74

Listings

3.1	Traditional ROOT code to read a file, create a custom variable and plot an histogram out of it.	38
3.2	RDataFrame code to read a file, create a custom variable and plot an histogram out of it.	38
3.3	Simple script using PyRDF to create the RDataFrame computational graph.	49
4.1	Code changes between PyROOT and PyRDF versions of the di-muon analysis	55
4.2	Python code for the FindMuonTauPair function. Note that build_pair will be declared to the interpreter at runtime from the header, so it's not defined in the Python code.	65
4.3	C++ code for the build_pair function, written in the skim.h header file. . .	65
4.4	Minimal code to retrieve a collection of numpy arrays from a variable of the processed VBS dataset distributedly.	76

List of Abbreviations

ALICE	A Large Ion Collider Experiment
ATLAS	A Toroidal LHC Apparatu S
CMS	C ompact M uon S olenoid
CMSSW	CMS Soft W are
CVMFS	CERN Virtual M achine F ile S ystem
DBMS	D ata B ase M anagement S ystem
DC	D ata C entre
EP-SFT	E xperimental P hysics - S oft W are T echnologies
HEP	H igh E nergy P hysics
HL-LHC	H igh L uminosity LHC
IO	I nterface and O utput
JIT	J ust I n T ime
LHC	L arge H adron Collider
LHCb	LHC beauty
PS	P roton S ynchrotron
RDF	R oot D ata F rame
SPS	S uper P roton S ynchrotron
SWAN	S ervice for W eb-based A nalysis
WLCG	W orldwide LHC C omputing G rid

Chapter 1

Introduction

1.1 Context

Scientific research has always been intertwined to a certain degree with Computing. Even more so over the last few decades, during which the needs for resources in terms of storage and processing power have increased exponentially. This holds true for many different joint collaborations in fields such as biology, medicine, earth sciences, physics and astrophysics, among which CERN definitely represents a notable example.

CERN, the European Organization for Nuclear Research, is one of the world's largest centres for scientific research. It gathers scientists worldwide with experience in a diverse range of fields with the objective of better understanding the most fundamental building blocks 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. These are isolated and accelerated in a specific environment that brings them close to the speed of light, until a collision is forced, generating an incredible amount of physical phenomena. This gives the physicists a means to study the interaction between matter at its most pure state and insights about the fundamental laws of nature [1].

The main technical instrument used to pursue this goal is currently the LHC (Large Hadron

Collider), a two-ring superconducting hadron¹ accelerator with a circumference of 27 km of superconducting magnets, laying 100 m underground. It can nominally provide a center of mass energy of 14TeV^2 for proton-proton collisions, created by 2808 bunches containing $1.15 \cdot 10^{11}$ particles each, with a bunch spacing of 25ns . The design luminosity of the collider is $10^{34}\text{cm}^{-2}\text{s}^{-1}$. Primary protons are obtained from Helium, they are first accelerated in a linear collider (Linac), to be then injected in the chain of circular accelerators that allows the achievement of their maximum energy (6.5 TeV). Protons pass through the Proton Synchrotron (PS, 25 GeV), the Super Proton Synchrotron (SPS, 450 GeV) and finally are injected in the LHC [2]. Figure 1.1 shows the whole accelerator system.

Four experiments were designed and built to completely exploit the physics program for proton collisions. They are located at four interaction points of the LHC, in which the actual collisions happen. Two general purpose experiments, ATLAS (A Toroidal Lhc ApparatuS) [4] and CMS (Compact Muon Solenoid) [5], were designed to explore all the possible aspects of the LHC physics, from heavy-ions collisions and forward physics, to Higgs boson physics and direct search of new particles. Specially dedicated to heavy-ions physics is the ALICE experiment (A Large Ion Collider Experiment) [6], while the LHCb experiment [7] has the aim to maximize the LHC potential in beauty and charm physics.

The accelerator alternates active periods, the so-called *Runs*, to scheduled *Long Shutdowns* for maintenance and upgrade. During Run 1 (2010-2012), LHC could already deliver several 10s of Petabytes (PB) per year and the mass storage systems at CERN surpassed the remarkable figure of 100 PB of stored data by the beginning of 2013 [8]. This number has steadily grown over time, reaching close to a hundred PB per year (at the end of Run 2 in 2018) and is expected to grow even more by the time the new detector upgrade (High Luminosity LHC) will be operational in 2026 [9].

¹A particle made of two or three quarks, held together by the strong force. Protons and neutrons are hadrons.

²The electronvolt (eV) is a unit of measure, representing the energy an electron has while passing inside an electric potential of 1 Volt (V). a Teraelectronvolt (TeV) is equivalent to a thousand-billion electronvolts (10^{12} eV).

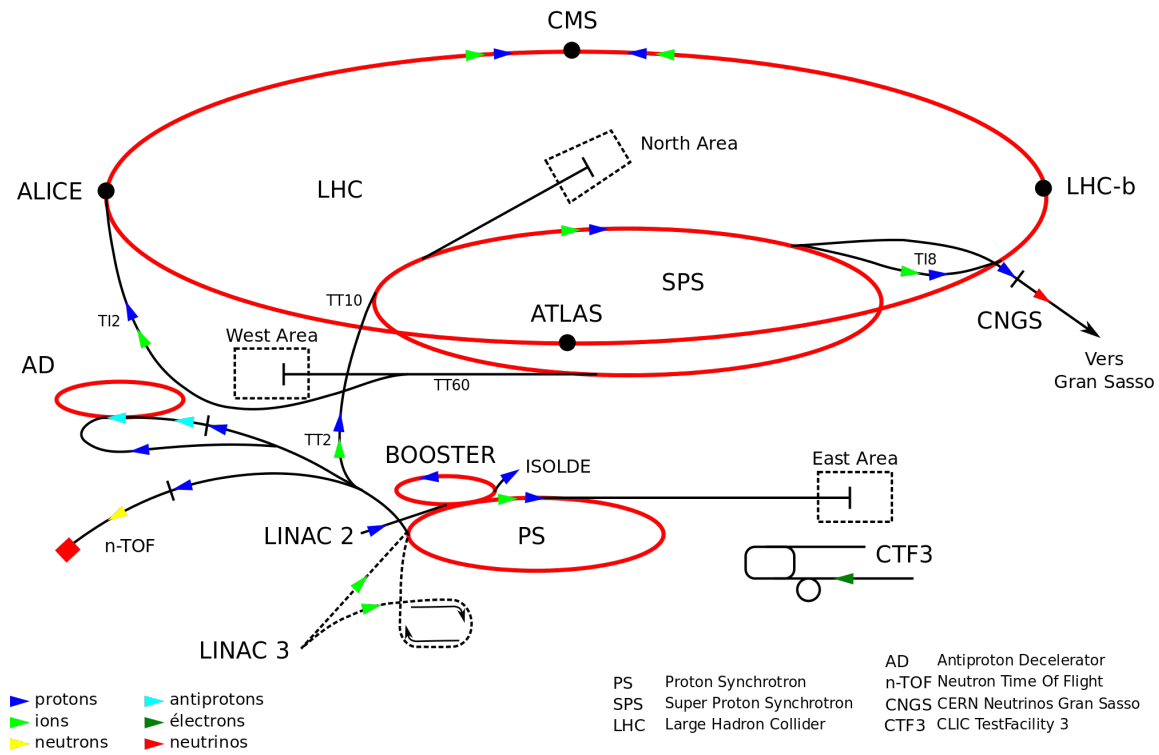


FIGURE 1.1: The LHC accelerator complex. [3]

Storage and computing solutions had to be created to deal with all of this information. Most evidently, it wasn't the case as before that all could fit in a single facility's systems. A number of studies focused on distributed and parallel models, in order to effectively utilise the computing power and storage available at all the different institutions collaborating with CERN. The most widely adopted solution in this regard is represented by Grid technologies, which enable effective work by the members of collaborations regardless of their geographical location. Currently, the Grid technologies are used in various fields of science all over the world, but the one developed specifically for the CERN physics program is the WLCG (Worldwide LHC Computing Grid).

These kind of needs, and for that matter the solutions involved, have been historically a concern for few, highly keen on research use cases such as the ones found within High Energy Physics (HEP) and CERN. But in the past few decades, with the development and widespread adoption of the World Wide Web, the steady decline in computing cost (a concept implied by Moore's Law [10]) and the constant evolution in how people create and use digital media, a new, ever more conscious field has emerged that encompasses different industries, research fields, society as a whole, broadly known as Data Science. This term is used in so many different contexts and with so many different goals that is difficult to define it in a single statement. One firm focus remains though: the will to manage, analyse and extract value from data. In fact, Data Science is usually coupled with the aforementioned phenomenon of an unprecedented abundance in information known as Big Data. Though it has drawn great attention in the research, it is difficult to say what the impact of this trend will be when we look at it a few years from now. It seems clear, however, that the superabundance of data is a firm trend and that, if anything, will increase in the future. Handling these big amounts of data has been a staple for the scientific community, and recently this need has extended to the industry and the society in general. Even if there may be peculiarities in the requirements of the research community and in their computing workflows, mainstream companies are now concerned with similar problems [11]. Scientists are looking into industry solutions to tackle their problems, mainly for the

obvious advantage that such solutions do not require dedicated development and maintenance efforts from the community itself (often subject to uncertain funding). [12]

Actually, HEP has always been a fertile soil for Data Science. Even before the LHC was operational, the upcoming challenges in computing and storage were already a focus for researchers. In 2006, a CERN thesis studied how to optimise data indexing on the grid for faster retrieval [13]. In 2009, another thesis work in collaboration with CMS tested file transfer protocols over the grid with a focus on security [14]. The following years tell similar stories. In 2012, a tool for distributed computing with a graphical user interface was discussed and at the same time graphics processors were explored for their impact on the physics workflow [15] [16]. In 2015, workflow execution scheduling was addressed with a system that made use of distributed hash tables [17]. In 2017, an algorithm was implemented to assess the popularity (e.g. the frequency of access) of a particular dataset on the grid so that the data could be redistributed accordingly [18]. This is merely a short list of the enormous amount of contribution that CERN and HEP in general have made to research in some of the topics that make up Data Science, but it shows how there's always been an overlap between the two.

1.2 Motivation

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 [19] 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 [20]. 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.

Current total amount of data produced at the LHC already presents a substantial processing challenge. During a run, LHC detectors witness particle collisions at very high rates, much higher than what present detectors data acquisition system can even handle. Most of the events that happen in the accelerator have thus to be filtered out. 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 centre.

However, most of the hypothesis require an even 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 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.

Thus, all experiments have different upgrade schedules to keep up. In preparation for Run 3 starting in 2021, the ALICE experiment will undergo upgrades in several subsystems and the online–offline system for data acquisition and processing. its goal is to reach an interaction reading rate of 50 thousand events per second (KHz) in Pb-Pb collisions³, finally producing more than 1 Terabyte (TB) data per second while functional [21]. LHCb is in the process of being upgraded as well, with the triggering system (the one taking care of keeping only good events to be later processed) changing from hardware based to only software based. In Run 3, it will process more than 40 times the number of collisions that it does today (about 10 million collisions every second), thus enabling an increase in the

³Collisions between two Pb^{82+} lead ions

output rate to storage of a factor 10, from 0.5 Gigabytes per second (GB/s) to up to 5 GB/s [22]. ATLAS and CMS are not shifting their gears as much during the current shutdown, but they are both expected to read 200 different kind of interactions per each bunch crossing (compared to the current 35) and they will receive hardware upgrades accordingly [23] [24].

The specifications for Run 3 are not even close to those of the next LHC configuration, the High Luminosity LHC (HL-LHC) [25]. Fig. 1.2 shows the timeline of the scheduled runs and shutdowns of LHC, including the upgrades involved between them leading to HL-LHC.

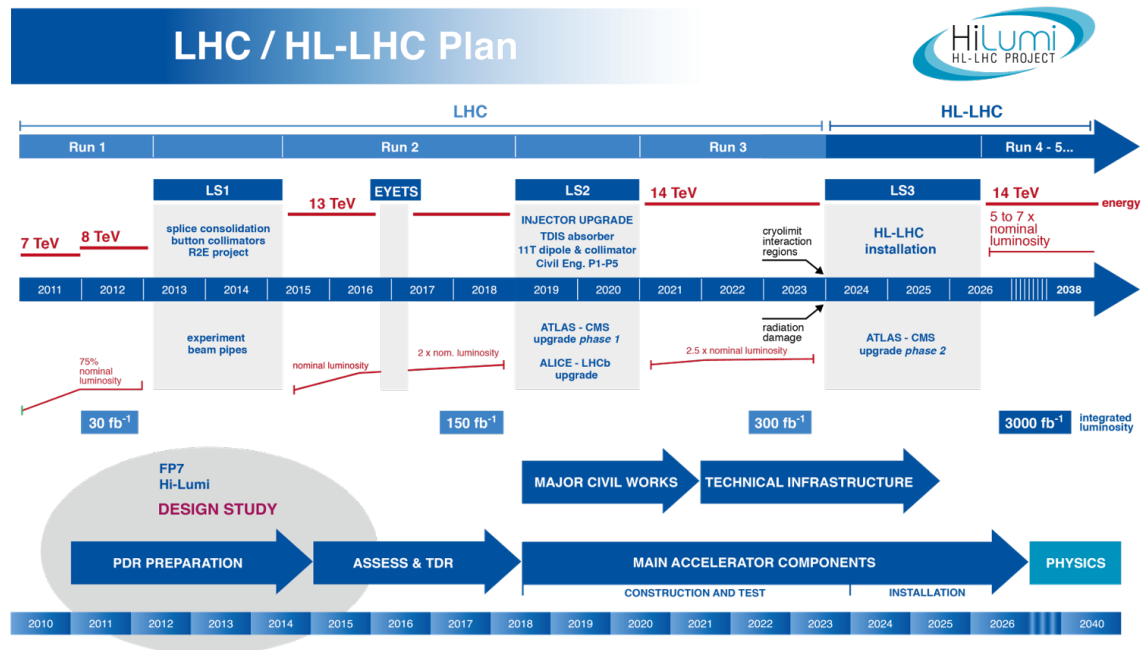


FIGURE 1.2: LHC project schedule [26].

The updated accelerator, scheduled to begin taking data in 2026, is planned to collect only

by ATLAS and CMS some 30 times more data than the LHC has produced in its whole life-time until now. As the total amount of LHC data already collected is close to an Exabyte (EB), it is clear that the problems to be solved require approaches beyond simply scaling current solutions. Instead they reveal a strong necessity of collaboration in multiple areas, including those outside physics. Even assuming that Moore's Law [10] would still hold for the years to come, at least with a conservative +20% increase in computing power per year due to technological evolution alone, the experiments would be off by a factor 5 with respect to their needs [27]. Both ATLAS and CMS will face a step change from 2026 onwards, where just increasing resources will not be enough due to budget limitations. Data collection and processing will be depend on affordable software and computing, so the physics reach during this period will be limited by how efficiently those resources can be used. In spite of the fact that hardware is rapidly evolving with new paradigms and architectures, most of the software currently in use 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 HEP community is aware of the risks ahead and it is making continuous efforts to raise awareness that software upgrades have to happen in parallel with hardware improvements [28]. Such advances in software must be aligned with the volume of data expected to be produced but also with the 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 experiments around the world. At the same time, heterogeneous platforms have become 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 thus of paramount important. These projects may find advantageous then to push software evolution without disregarding user-friendly interfaces and programming models.

Chapter 2

Data lifecycle at CERN

Let's take a broader look at the different steps that data take during the physics program at CERN. Generically speaking, the data lifecycle can be described through the following stages:

- Generation
- Collection
- Processing
- Storage
- Management
- Analysis
- Visualization
- Interpretation

Each of these can vary depending on the field of interest, the source of the data and the goal. In this chapter the specificities of the HEP data lifecycle will be highlighted, with a focus on data coming from the LHC.

2.1 Generation

People generate data: every search query performed, link clicked, movie watched, book read, picture taken, message sent and place visited contribute to the massive digital footprint generated by any human being each year. Sensors generate data: more and more sensors monitor the health of the physical infrastructures, e.g., bridges, tunnels, and buildings; provide ways to be energy efficient, e.g., automatic lighting and temperature control in rooms at work and at home; and ensure safety on the roads and in public spaces, e.g., video cameras used for traffic control and for security protection. As the promise of the Internet of Things plays out, more and more sensors will be generating an ever increasing amount of data. At the other extreme from small, cheap sensors, there are also large, expensive one-of-a-kind scientific instruments, which also generate unfathomable amounts of data.

LHC is definitely one among them. In the detector, bunches of protons or lead (Pb) ions are accelerated and collided in key points along the circumference at the impressive energies of 13 TeV and 5.1 TeV, respectively. The collisions, also called events, happen once every 25 nanoseconds, which translates to roughly 40 million times per second (MHz), only considering single-pair collision (also called elastic collision). Since in the accelerator the protons collide multiple times at each bunch crossing, the real number of collisions is closer to 600 million times per second. And since the bunches cross at any of the four main experiments, this number has to be further multiplied by 4. One parameter that is useful in characterizing the performance of a particle accelerator is the luminosity, defined as the ratio of the number of events detected (N) in a certain time (t) to the interaction cross-section¹ (σ):

$$L = \frac{1}{\sigma} \frac{dN}{dt} \quad (2.1)$$

¹Probability that two particles will collide in a certain way decaying into a certain product.

Luminosity represents the physical quantities of events per time per area, and is usually expressed in the cgs² units of $cm^{-2}s^{-1}$. A related quantity is the integrated luminosity, defined as the integral of the luminosity over time:

$$L_{int} = \int L dt \quad (2.2)$$

The integrated luminosity is another important parameter for the LHC, since it gives the total number of events generated in the detector during a certain time. That is, the more the integrated luminosity, the more the data available for studying [29].

The design luminosity of the LHC is $10^{34}cm^{-2}s^{-1}$, which was first reached in June 2016 [30]. By 2017 twice this value was achieved [31]. The planned luminosity for HL-LHC will increase the current value by a factor 10, reaching $10^{35}cm^{-2}s^{-1}$ [32]. Engineers and physicists strive to maximize these two quantities as they are a direct measurement of the amount of data that will be available for analyses. In particular, each event generates approximately 1 megabyte (Mb), so that:

$$6 \cdot 10^8 \text{ collisions/s} \cdot 1 \text{ MB/s} = 6 \cdot 10^{14} \text{ bytes/s} = 0.6 \text{ PB/s} \quad (2.3)$$

of raw data is produced at any given experiment when the detector is on.

2.2 Collection

The huge amount of raw data generated by the LHC represents an extreme challenge, both for the management of the data flow and the storage. Furthermore, only a small part of these data needs to be kept for further analysis, while the rest is considered background noise composed by spurious collisions or uninteresting event from the physics point of view. For these reasons, experiments at CERN implement specific techniques to filter out most of the information (about 99.9%) with the so-called trigger systems.

²Variant of the metric system based on the centimeter, gram and second units.

In particle physics, a *trigger* is a system that uses simple criteria to rapidly decide which events in a particle detector to keep when only a small fraction of the total can be recorded. Since experiments are typically searching for interesting events (such as decays of rare particles) that occur at a relatively low rate, trigger systems are used to identify the events that should be recorded for later analysis. As previously stated, the LHC has an event rate of 40 MHz, while trigger rates can be as low as 10 Hz. The ratio of the trigger rate to the event rate is referred to as the *selectivity* of the trigger. For example, the Large Hadron Collider has an event rate of 1 GHz (10^9 Hz), and the Higgs boson is expected to be produced there at a rate of at least 0.01 Hz, therefore the minimum selectivity required is 10^{-11} .

Triggers usually make heavy use of a parallelized design, exploiting the symmetry of the detector: the same operation may be performed at the same time on different parts of the detector. Yet on a global scale they are essentially serial devices: in fact, they are usually divided into levels with different granularity. The idea is that each level selects the data that become an input for the following, which has more time available and more information to take a better decision. Custom hardware processors make an initial decision to keep an event in a few microseconds (μs) using coarsely segmented data from a subset of the detectors, while holding all the high-resolution data in pipelined memories. Commodity processors make subsequent decisions using more detailed information from all of the detectors and more sophisticated algorithms.

Each detector has its own trigger design and features. For example, in CMS the first-level trigger allows to store data produced in a time frame of $3.2 \mu s$, after which no more than 100 kHz of the stored events are forwarded to the next level of the system, called High Level Trigger (HLT). This must be done for all channels without dead time. The Level-1 (L1) system is based on custom electronics. The HLT system, relies upon commercial processors. Down the line the trigger system will output data to offline storage services at an average rate of 400 Hz which in turn means 400 MB/s since each event carries 1 MB worth of data. Assuming that during runs the LHC is active for 300 days per year in two shifts of ten hours per day then the amount of data output by CMS would be:

$$400 \text{ MB/s} \cdot 2 \cdot 10 \cdot 3600 \cdot 300 \approx 9 \text{ PB/year} \quad (2.4)$$

From equation 2.3 can be derived that any experiment produces approximately 19 million PBs, or 19 zettabytes (ZBs), of data per year (before triggering). This means that for CMS the ratio between data collected after trigger and data produced in the detector each year is:

$$\frac{9 \text{ PB/year}}{19 \cdot 10^6 \text{ PB/year}} = 4.75 \cdot 10^{-7} \quad (2.5)$$

Thus, only a tiny fraction of the original data is kept. Nonetheless, the different experiments with their own trigger systems collectively store several tens of petabytes of data each year for later analyses.

2.3 Processing

Data processing can mean many different things: cleaning, wrangling, formatting etc. After passing the trigger systems, data are sent from the experiments to the CERN Data Centre for their first processing. The data centre itself is made up of 15 thousands servers and 230 thousands processor cores, running 24/7. Over 90% of the resources are provided to CERN users via a private cloud system based on OpenStack. The first processing steps are comprised of a series of algorithms that perform initial reconstruction of the physics events. A copy of the reconstructed data is archived to long-term tape storage, thanks to the CERN Advanced STORage manager (CASTOR). This is a hierarchical storage (i.e. with disk and tape) management system specifically developed at CERN for the needs of physics data (e.g. the large volumes shown in Fig. 2.1, the distribution to different research centres and the peculiarities of the events' structure). It is made of different components that are centrally managed in order to safeguard their state. It offers a client API that allows users to interface with the other components in charge of allocating and

reclaiming space and controlling the file metadata. In this particular stage of the data lifecycle CASTOR also manages the tape infrastructure.

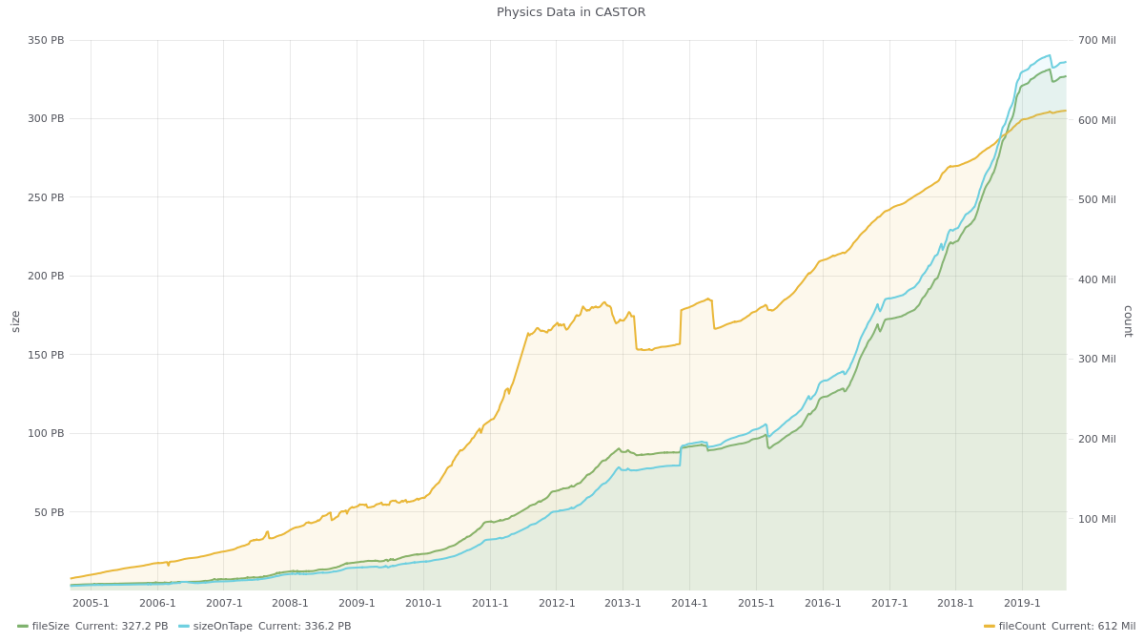


FIGURE 2.1: Physics data stored within the CASTOR system.

2.4 Storage

The initial copy on tape of the reconstructed data is just the tip of the iceberg when it comes to storing data coming from the LHC. The sheer amount of information provided by the experiments is too precious to risk having one single point of failure in case that anything goes wrong. At the same time, leaving data stagnating on disk would not be of any use. It needs to be readily available for scientists, not only at CERN but around the world in multiple research labs, to be analysed and further processed. This is made possible through the various entities that participate in the LHC collaboration, each having their own computing and storage facilities.

2.4.1 Grid technologies in HEP

This approach, which has become fundamental in HEP, belongs to the grid technologies area. It was already under development years before the LHC was operational, when it became evident that, with the available funding, CERN alone would not be able to satisfy all the computing and storage requirements that the activities of the LHC experiments required. The solution to this problem was found in the local computing facilities of the associated institutes participating in the LHC experiments. A distributed system that made use of all the available resources was proposed and the grid technologies were chosen to implement such system, since they seemed to tackle exactly the problem at hand. This distributed resource model matched well with the funding structure of the entities involved with the LHC and would have made it easier for physicists at the different institutes to gain access to data.

Independent of the architecture and technologies, the Grid was defined by Ian Foster and Carl Kesselman, two computer scientists both famous for their work in the distributed computing domain, as:

“A hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities”[33]

A more recent definition from Wolfgang Gentzsch is :

“A Grid is a hardware and software infrastructure that provides dependable, consistent, and pervasive access to resources to enable sharing of computational resources, utility computing, autonomic computing, collaboration among virtual organizations, and distributed data processing, among others”[34]

Therefore, the Grid combines heterogeneous and distributed computational resources to a large virtual computer through a middleware that can be used to solve various scientific problems. The benefit is faster, more efficient processing of different tasks.

The first definition of the Grid architecture was presented in the article 'The Anatomy of Grid' [44]. The Grid architecture identifies fundamental system components, specifies the purpose and function of these components, indicates how these components interact with one another[44]. It defines standard protocols and APIs to help the creation of cooperative Grid systems and portable applications. The Grid architecture is defined in terms of layers, where each layer has a dedicated scope.

Today, the coordinated efforts of HEP facilities around the world fall under the name of Worldwide LHC Computing Grid (WLCG). It is a large distributed computing infrastructure - more than 400,000 CPU (Central Processing Unit) cores, 300 PB of disk and more than 200 PB of tap - devoted to store, deliver and analyze the data generated by the LHC, making the data available to all partners, regardless of their physical location [35]. The WLCG is operated by a global collaboration of more than 170 computing centres, in 40 countries. The WLCG is supported by many associated national and international grids, such as EGI (Europe-based) and OSG (USA-based), as well as many other regional grids. While both EGI and OSG provide computation resources to researchers from many different scientific disciplines, the high energy physics community is their most important consumer (in terms of resource needs) and the main driving force behind the projects. As a whole, the WLCG is the world's largest computing grid [36].

The four main component layers of the Worldwide LHC Computing Grid (WLCG) are (from top to bottom):

- Applications and services
- Middleware
- Hardware
- Networking

The Applications and services layer is defined by web portals and development toolkits. This layer provides many management-level functions such as accounting, and measurement of usage metrics. The next layer is the Middleware layer which consists of a

software system located between applications and the operating system, providing protocols that enable multiple elements (network, storage, servers, etc.) to participate in an integrated Grid environment. The Grid middleware purpose is to define standard protocols and standard APIs realizing communication and data exchange between the various elements. It enables virtualization which is used to mask the heterogeneous computing resources. The Hardware layer (or resource layer) consists of the actual resources that are part of the Grid, including primarily servers and storage devices while the Network layer is the underlying connectivity for the resources in the Grid.

The resources available with the WLCG are much larger than any single research centre could provide. The various participants in the sharing of the resources have been called Virtual Organizations (VO). Each VO acts according to well-defined rules stating which grid resources are shared, who is allowed to access them and under which conditions[37] [38] [39].

In WLCG, each experiment is represented by a VO. Each resource centre (site) may decide to support one or more of these VOs. The sites are organized in tiers, according to their relative size (in terms of resources) and the functions they accomplish. Even if the duties of each tier vary from one experiment to the other, we can say that, in general terms, the centres are classified as follows:

- Tier-0 (CERN): Receives the raw data from the detector, caches it on disk, archives it to tape and distributes a second copy among the Tier-1 sites. Calibration and alignment are also run at Tier-0 and a small sample of raw data is promptly processed for quality checks. In addition, a first-pass processing of the raw data is performed too (and the results archived and distributed to Tier-1 centres).
- Tier-1 : Usually equipped with mass storage systems and archival facilities (tape), Tier-1 centres are responsible for reprocessing the raw data once adequate calibrations are available, as well as for distributing it to the collaboration. They also host simulated data produced at the Tier-2 sites.

- Tier-2 : They host a share of the processed datasets and provide analysis facilities, as well as resources for producing certain amount of simulated data.
- Tier-3 : Opportunistic resources, with little or no formal commitments (pledges) to LHC experiments, and mostly offering CPU only, with no permanent storage available.

Fig. 2.2 depicts the tier layout for WLCG. Though the duties of the grid encompass many phases of the data lifecycle, as far as storage is concerned the different institutions enable the replication of data and its distribution in order to minimize chances of data loss. At the same time, they reduce latency of analyses for scientists local to the corresponding research facilities who do not need to transfer data across long distances. Other goals achieved through the grid will be discussed in later sections.

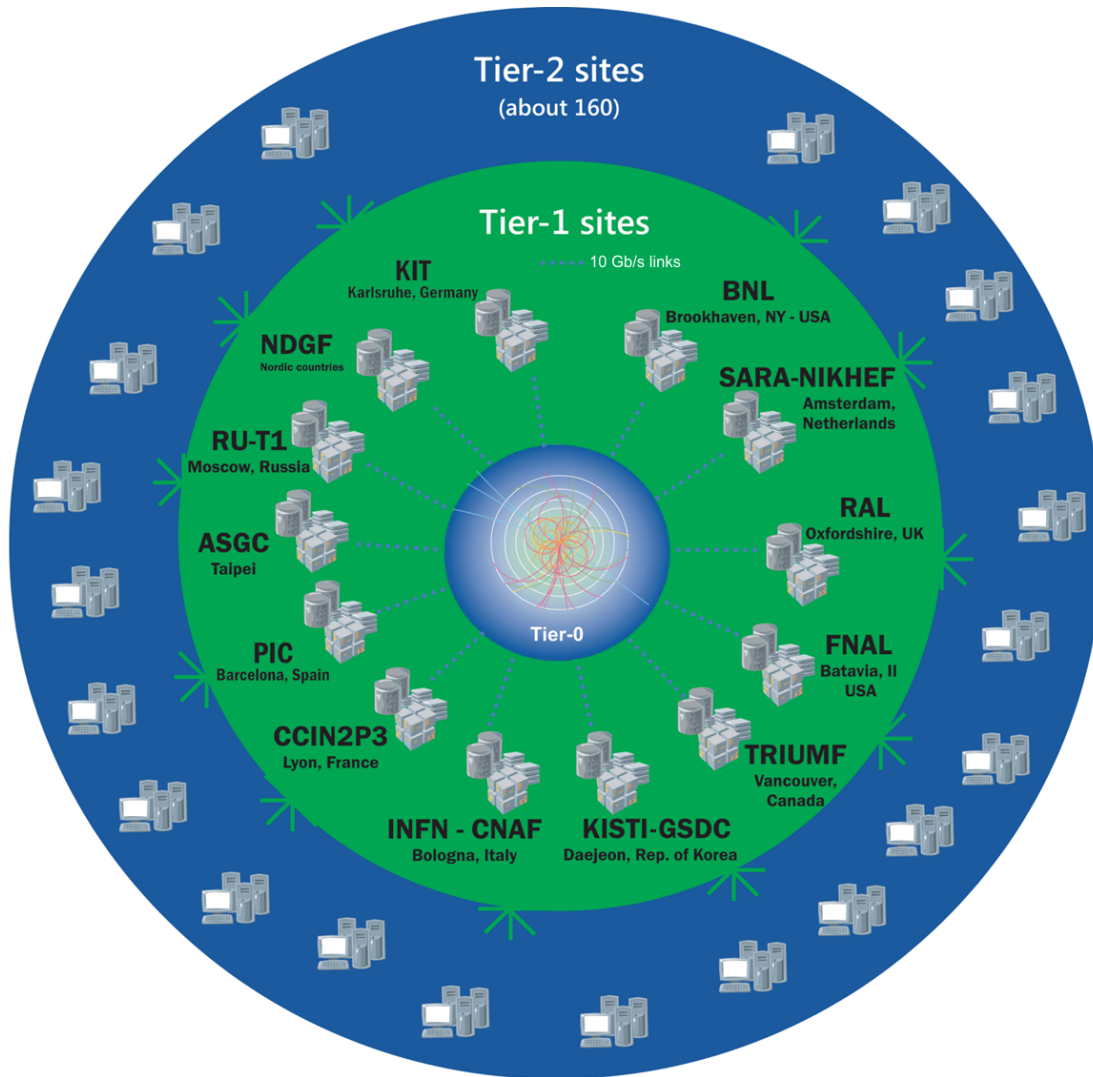


FIGURE 2.2: Topology of the WLCG tiers.[40]

2.5 Management

Data management is another goal handled by the different communities that contribute to the WLCG. Each of the different actors has its own agreements with CERN and may provide different ways and technologies for accessing, retrieving and managing data. So far data coming from the LHC has been implicitly considered as physics data, related to the events happening in the accelerator, but there are non-event data as well that may include calibration and condition data about the detector. While the first are generally stored in files, the latter are managed through Relational Database Management Systems (RDBMS). Oracle, MySQL and PostgreSQL relational databases are used at CERN for a variety of use cases for high energy physics and for other services in the Organisation. In particular, Oracle databases find several different applications at CERN:

- Backend deployment for online control systems and LHC experiments' conditions data [41] [42].
- Offline processing and distributed data management [43].
- Logging accelerator data, at rates in excess of 500 GB/day from a large number of data sources and control systems that are critical for the functioning of the LHC [44].

Though these databases have been the staple for different years now, recently different types of systems that do not specifically belong to the traditional tools of HEP have been explored for potential usage and integration, for instance the Hadoop ecosystem. The Hadoop ecosystem has emerged in the last decade as one of the leading platform for large scale distributed data processing, often referred to as "Big Data". Hadoop started as an open-source project to reproduce the architecture and functionality of systems built internally at Google for their data processing use cases [5]. It has since evolved and greatly profited from the contribution of many players in the open source community. In 2017 a work from the CERN databases research group demonstrated the benefits of integrating this big data framework with the traditional one, at the same time enabling an hybrid database system and maintaining a transparent interface for the user [45].

2.6 Analysis

While the previous steps in the data lifecycle are mainly supported from the IT department at CERN or other institutes, this stage is probably the main focus in the work of physicists. The core of their task is to transform, filter and extract information from data while looking for new physics.

Any analysis starts from data. At this stage, they are stored in files distributed over the grid. The information comes in the form of already reconstructed data that represent the physics events that happened inside LHC. There are different data formats depending on the type of information:

- RECO, ESD, DST: files formatted as such contain first reconstruction of raw data coming from LHC. They usually represent tracks of the particles with associated hits of the modules in the detectors and calorimetry quantities that are kept after the triggering system.
- AOD: Analysis Object Data, containing full tracks, descriptions of particles and vertices inside specific objects.

The different experiments have different data models and therefore different schemas of their data. Tab. 2.1 summarizes the data size of each data format that the different experiments use in their workflow.

TABLE 2.1: Event data sizes for different data formats in use by the various experiments in LHC. [46]

Data Format [KB]	ALICE	ATLAS	CMS	LHCb
RAW	1050 (p-p)			
	1625 (p--Pb)	1000	500	60
	7500 (Pb--Pb)			
ESD	15–30% RAW	2700	1000	110
AOD	10–15% RAW	350	300	120
Analysis level	AOD	<i>variable</i>	<i>variable</i>	10

The analysis relies on simulation in order to compare physics theory (the Standard Model when speaking about CERN) with the reality of the events. Generally, per each event generated in LHC there will be one event generated fully by software that emulates the whole structure of the accelerator and the detector, including all the interactions involved between the particles and between the different parts of the machine. The software packages involved in this process today are mostly two:

- Pythia: a C++ based high energy physics event generator. It simulates collisions at high-energies between elementary particles [47].
- GEANT4: a toolkit for the simulation of the passage of particles through matter. It simulates the interactions between the particles and the materials of the detectors [48].

The data generated through simulation follow the same path as the event data. They are generated at CERN and then distributed all over the grid, replicating the data formats discussed previously.

There is no predefined workflow for a physics analysis: it may vary depending on the context, data, goal and often it undergoes lots of small changes in the code in attempt at

comparing slightly different configuration to achieve the desired results. Some common aspects can be nonetheless identified:

- **Skimming:** in the first stage of the analysis the source data is filtered in order to retrieve the interesting features and configurations, e.g. selecting only certain physics dimensions such as energy, angle or establishing thresholds such as particles with momentum higher than a certain value.
- **Performing operations:** having obtained the relevant data then custom functions and operations tailored to the specific analysis are performed to construct new variables that may be of interest.
- **Storing results:** the newly formed data are finally stored. Usually the final result of an analysis is some kind of visualization, mostly histograms or derived, that will be further discussed in the next section.

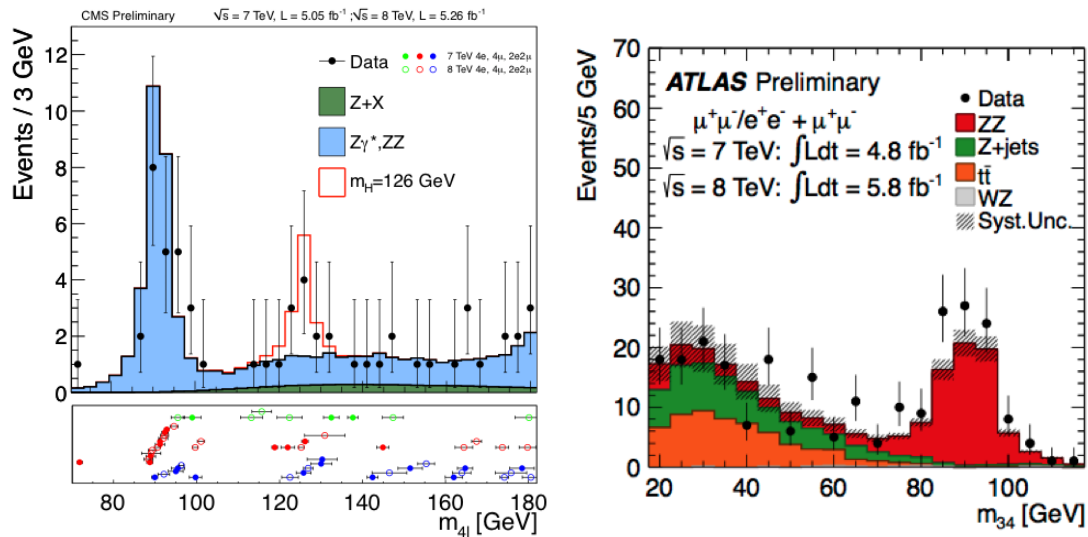
There have been a plethora of different tools that enabled data analysis for physicists along the years. Furthermore, each experiment has its own software stack that includes analysis frameworks tailored to the specific program of the experiment collaboration. Even more down the line, any university or single researcher may write custom code for their analysis. But there is one common tool that is widespread over the HEP community and is the building block for many of the frameworks utilised, ROOT [49]. Since it is also one of the technologies that this thesis relies upon, it will be better described in a later chapter.

2.7 Visualization

Data visualization is the presentation of quantitative information in a graphical form. In other words, data visualizations turn large and small datasets into visuals that are easier for the human brain to understand and process. Good data visualizations are created when communication, data science, and design collide. Data visualizations done right offer key insights into complicated datasets in ways that are meaningful and intuitive.

American statistician and Yale professor Edward Tufte believes excellent data visualizations consist of "*complex ideas communicated with clarity, precision, and efficiency*" [50].

In HEP, a subset of the many different visualizations is most often utilised, consisting of histograms and closely related graphics. This is mainly due to the nature of physics data of millions of events that should reproduce a physics phenomenon and the need for comparing these with simulations. Various examples can be seen scattered around the web, Fig. 2.3 shows a couple of plots published in 2012 after the discovery of the Higgs boson.



(A) CMS data compared against monte carlo simulation. (B) ATLAS data compared against monte carlo simulation.

FIGURE 2.3: Histogram plots produced using ROOT presented for the announcement of the Higgs boson discovery, taken from the official ROOT gallery [51].

2.8 Interpretation

The last step of the data lifecycle is where all the previous efforts are aimed at, to answer the question "what does the data tell?". The work of physicists and scientists to fill the gap between theory and reality, to go from the phenomenon to raw data then back again to the phenomenon through analysis culminates in the insights and understandings that are achieved in the end. Through these insights some key discoveries have been made, most notably the discovery of the sought-after Higgs boson, but also many other observations that have shed light on aspects of theory previously undefined, both in the Standard Model and beyond (e.g. supersymmetry theories). It is still the duty of the researchers and collaborations around the world to strive for improvement in their quest for unveiling the most fundamental mysteries of the universe.

Chapter 3

Tools and methodologies

The issues and concepts discussed until now regarding computing and storage of data have been declined in different software tools and paradigms during the years. This chapter will discuss and further explore specifically the ones that are more closely tied to the work of this thesis. It will start describing them individually, to then highlight some of the ways that have enabled the connection between different tools and programming models previously unexplored by the HEP community.

3.1 ROOT

ROOT is a modular scientific software toolkit. 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. ROOT offers a complete framework and environment for developing and running physics analysis, and for storing data in an efficient way [49]. ROOT was and still is developed at CERN specifically for the needs of the HEP community. The development began in 1994 in the context of the NA49 experiment [52], which was generating about 10 terabytes of data per run, an impressive number for the period. This provided the original authors (René Brun and Fons Rademakers) the ideal environment to develop and test a tool that should have succeeded the twenty-year-old FORTRAN libraries HBOOK and PAW. Indeed, ROOT would

have become the go-to software for data analysis of the LHC experiments in the years to come.

ROOT is an object oriented framework that aims to solve problems related to high energy physics. To better understand what ROOT is, it is important to start with understanding what is a framework: in computer science a framework is a structure that helps the programmers providing them a set of already working utilities and services (for example, IO, graphics, etc.) often related to the sector in which the framework was designed (for example, the services of a web development framework are related to the layout of a web page, to the organization of databases etc.). ROOT in particular offers services, functions and packages related to the world of high energy physics research, that allow to save much work. It provides the possibility to use a computer's graphics subsystem and operating system with abstraction, allowing the developer to create a graphical user interface and a GUI builder. ROOT provides also an abstract platform that allows to run C++ and command line scripts. Even more, ROOT has its own C++ interpreter, Cling, that enables running C++ code in a prompt or a macro, without the need to compile it. More precisely ROOT includes (among others):

- Libraries related to histogramming and graphing, that allow the developer to easily represent graphically statistical distributions. Also 3D visualization is allowed.
- Libraries related to regression analysis.
- Various statistics tools.
- The possibility of interfacing in both directions with code written in Python or R.
- The possibility of interfacing with Javascript, allowing the developer to access ROOT functionalities within a browser.

Fig. 3.1 depicts a slightly outdated but still relevant layout of the various ROOT modules (some modules have been added or updated meanwhile).

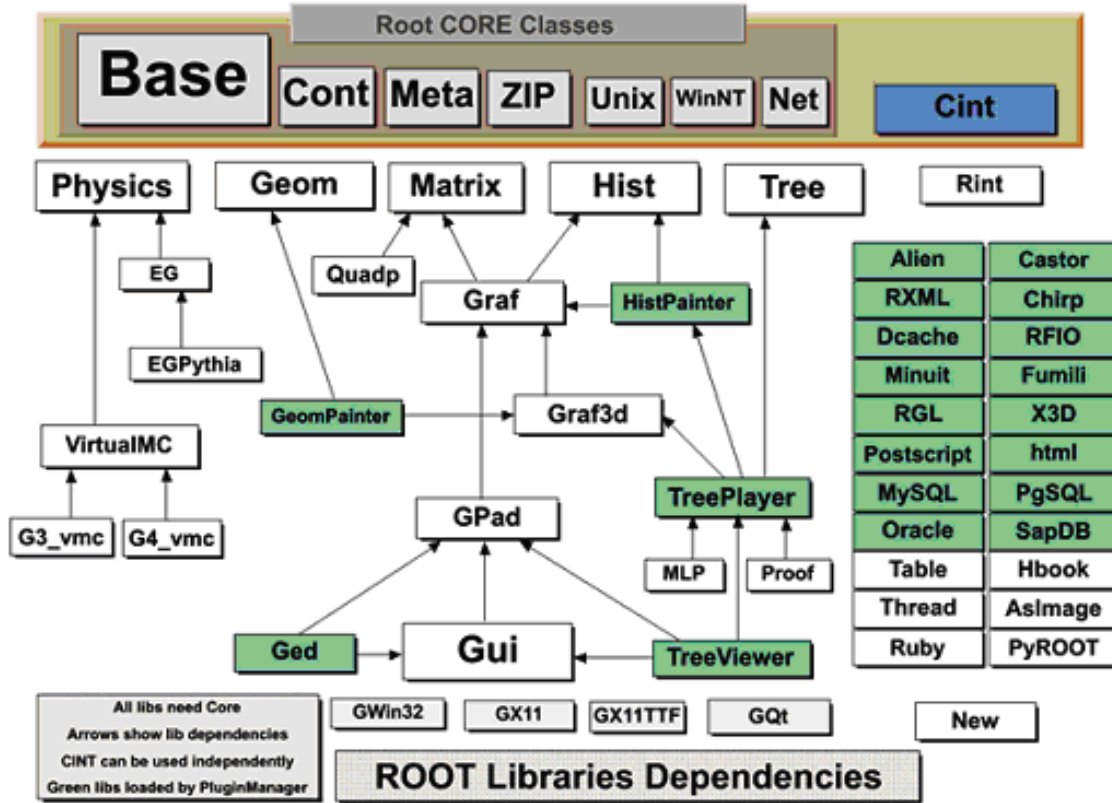


FIGURE 3.1: Diagram layout of the ROOT modules [53].

ROOT offers some neat capabilities that have been extensively exploited in the development of this thesis. The next sections will highlight the most important ones.

3.1.1 ROOT data model

ROOT implements a specific data model to optimise input and output (IO) operations peculiar to HEP. Though data can be stored in external databases (e.g. SQL) or in common

formats such as XML, the approach used by ROOT is of having files that are machine-independent, compressed in binary format, including both data and their description. This means that in a ROOT file one can store whatever kind of object. An automated tool will generate the data description (or *dictionary*, in the ROOT jargon) when saving data and C++ classes corresponding to this description when reading back the data. The *dictionary* is used to build and load the C++ code, to load the binary objects saved in the ROOT file and to store them into instances of the automatically generated C++ classes. Whether data is in the form of fundamental types (e.g. integers, floats, strings) or more convoluted C++ classes with deep hierarchies, ROOT can manage them in a seamless way. ROOT files (saved with the `.root` extension) can be structured into directories similar to the layout of files in an operating system, so that one ROOT file is more similar to an entire file system than to an ordinary file. Finally, ROOT files are designed with the enormous amount of data produced by LHC experiments in mind, optimising data access via caching and reading only the minimum interesting chunk of data required for the analysis.

The class that defines these characteristics is called `TFile`¹. Fig. 3.2 shows its physical layout in memory when created. It is essentially a collection of (C++) objects, with a file header containing metadata for better content management. Each element of the collection can then hold completely different information with respect to the others (e.g. a ROOT file can simultaneously store a table, an histogram the instance of a custom C++ class) and is stored within a `TKey` class instance. Separating different objects in different sections of the file improves read and write performance, since this allows both sequential and random access to data. Each key then contains a logical record header to store metadata about the single object. A full logical layout of the combination of file, key and object data can be seen in Fig. 3.3.

¹in ROOT most of the classes follow this `TClassName` schema for their name.

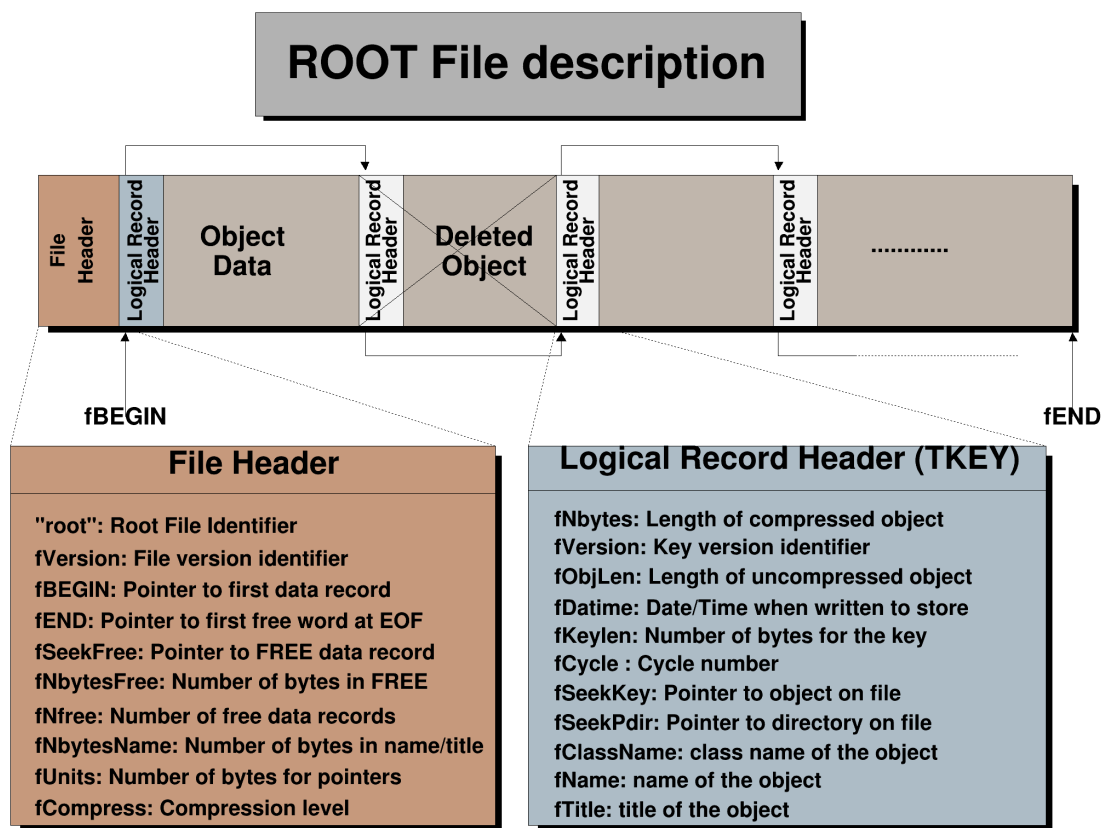


FIGURE 3.2: Physical layout of a TFile [54].

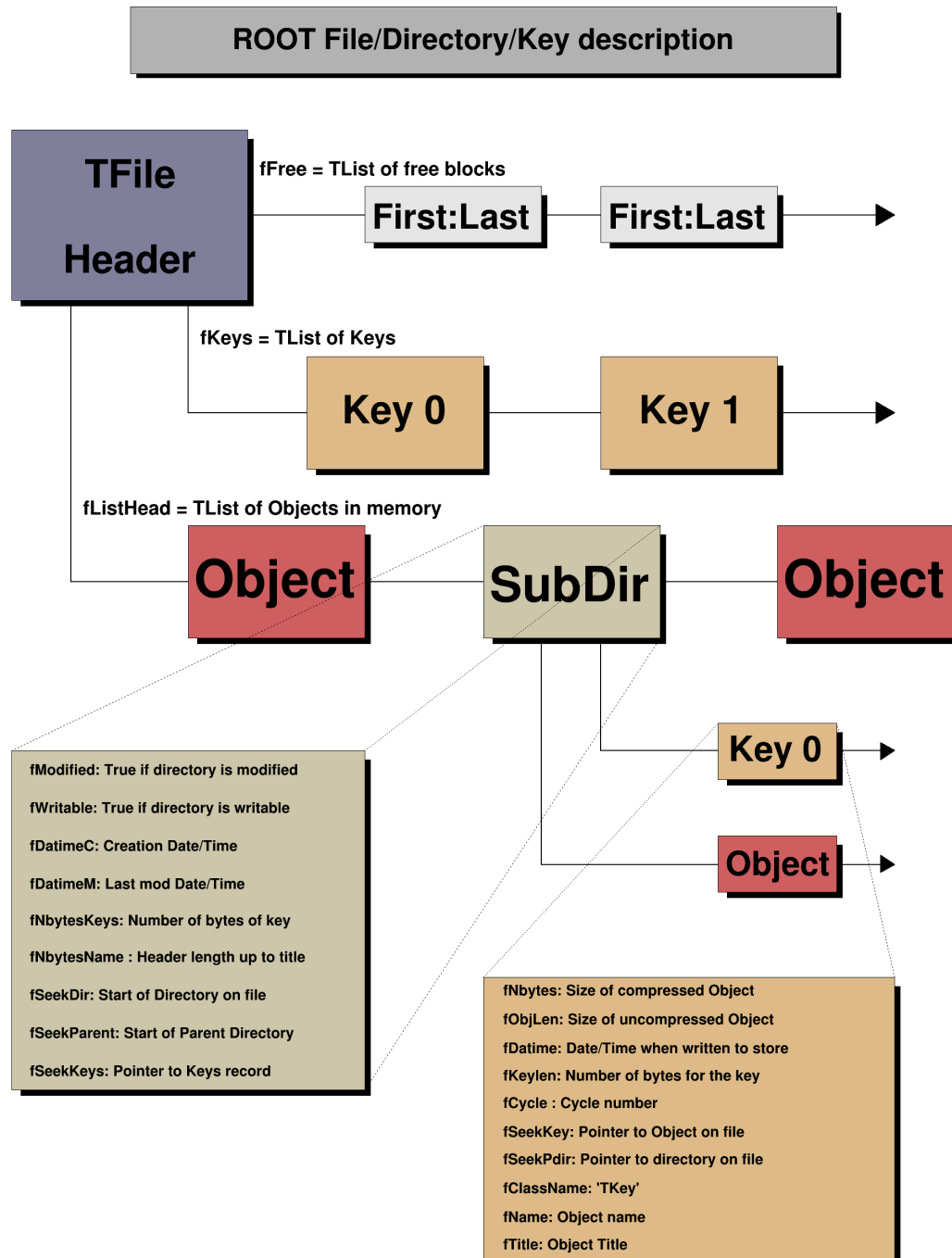


FIGURE 3.3: Logical layout explaining the connections between TFile, TKey, TDirectory and the actual data contained in the file [55].

Though it is quite handy to be able to save different objects in one single, rapidly accessible location, most of the time data is better managed when stored in a coherent structure holding same-class information together. The most dominant model for structuring data together since the 1980s has been the *relational data model*, in which data are sorted into entities connected by relations, that are physically translated to tables and variables that allow linking different tables together. In recent years, different data formats (as well as database management systems or DBMS) have been created in order to cope with the new requirements discussed until now. Generically, they fall under the name of *NoSQL*, as opposed to SQL, the language specific to the management of relational databases. The columnar data format belongs to this suite and is what ROOT specifically implements for its data structure. The columnar data format is such that all the values of an attribute of a table are stored as a vector using multiple memory blocks, and all the vectors of attributes of a table are stored sequentially. Organizing the values as a vector of attributes enables an easier understanding of the data, and also a high scanning and filtering speed. This results in significant sequential processing, where the columnar format has an enormous advantage compared with the traditional row-oriented disk database.

The class designed for this purpose in ROOT is called TTree. A TTree consists of a list of independent columns or branches, represented by the TBranch class (Fig. 3.4). Each branch holds same-class objects and can be read independently of the others. In fact, a branch could be completely "switched off" so that ROOT wouldn't even try to read the corresponding data. This improves data access speed and allows for good compression of the files. In the background ROOT automatically allocates separate memory buffers for each branch and writes them to disk after they have reached a certain threshold size. This also creates a so-called *cluster* in the tree, that represents the minimal portion of data that can be read without opening the whole file. This feature enables caching parts of the datasets for faster reading and greatly improves IO performance when distributing computations on the grid. A single tree can span multiple files, in fact when it reaches a certain size the file where it is currently being written is closed and a new file is opened. The TChain class was implemented to process the multiple files that contain a single tree.

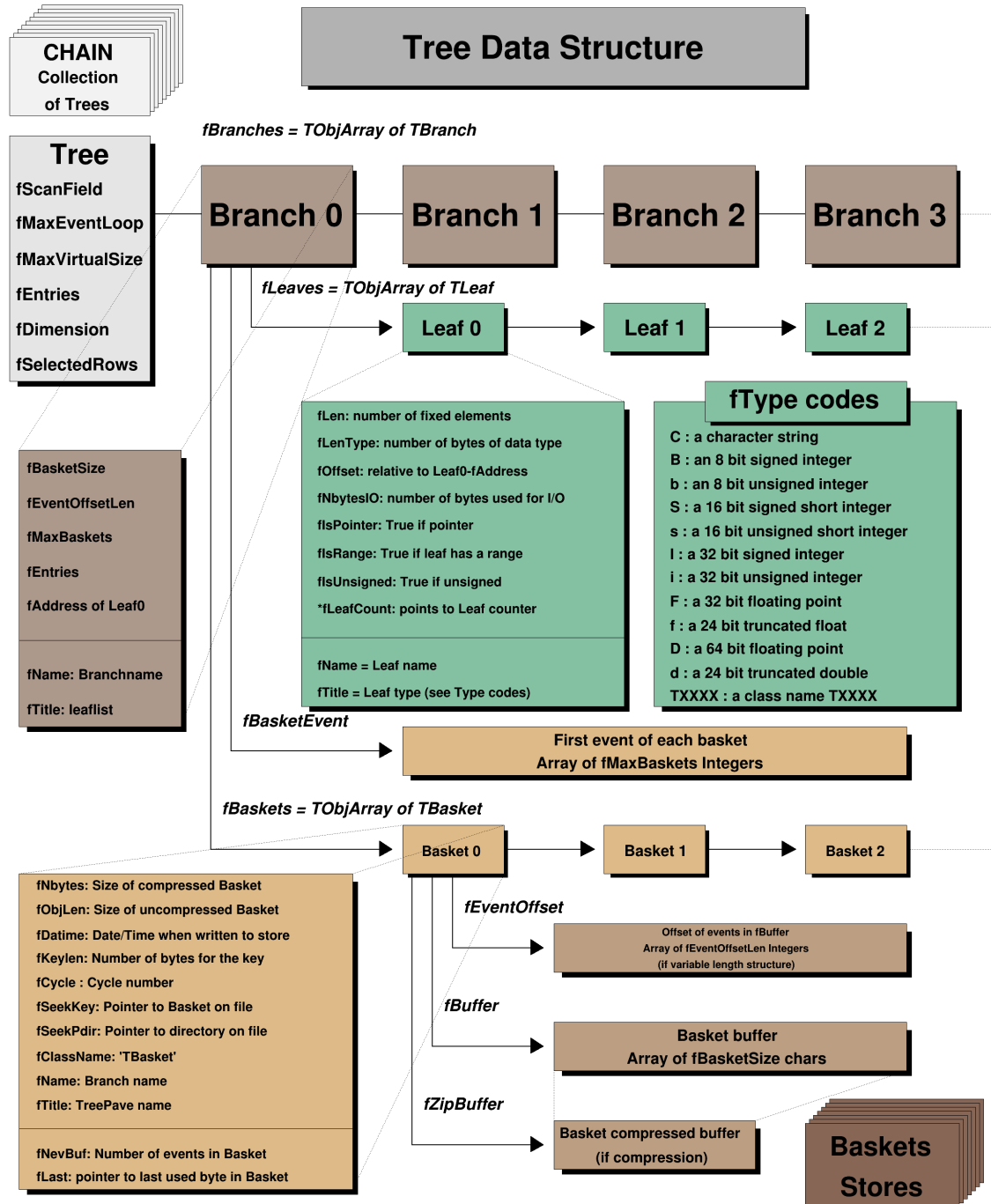


FIGURE 3.4: The ROOT tree data structure [56].

3.1.2 ROOT RDataFrame

RDataFrame (RDF) is one of the latest additions to the toolkit offered by ROOT. It is an high-level interface to data and supports different data sources, from ROOT files to plain text files (e.g. `csv`) or formats coming from frameworks specific to the Data Science field such as Apache Arrow.

Traditional physics analyses with ROOT followed an *imperative* programming model. The physicist had to focus on describing how the analysis should proceed, writing code that specifically took care of everything happening to data, from opening the file to filling the desired object result entry by entry. This approach gave complete freedom to users and meant both that the code was inherently more difficult to parallelise properly (scalar computations) and more error prone.

In contrast to imperative programming, *declarative* programming focuses on what the program should accomplish without specifying how the program should achieve the result. Many languages that apply this style attempt to minimize or eliminate side effects by describing what the program must accomplish in terms of the problem domain, rather than describe how to accomplish it as a sequence of the programming language primitives. RDataFrame follows exactly this paradigm, exposing a modern interface to users that is easy to use correctly and hard to use incorrectly.

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 IO operations and parallelisation, effectively creating a *computational graph*, that is a directed graph where each node corresponds to one of the operations to be performed on data. 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.

The interface design of RDataFrame is inspired by other dataframe APIs such as pandas [57] or Spark DataFrames, dividing the operations that can be performed on a dataset into

two classes, transformations and actions.

Transformations are those operations that do not compute anything on the values of the columns, but just modify the original dataframe returning a new one. RDataFrame offers the following transformations:

- **Define:** Create a new column in the dataset. This can be done through linear transformation of other columns in the dataset (e.g. summing or multiplying two or more columns), boolean operators on a subset of columns or just any valid C++ function returning some value. Two more transformations exist doing with the same duty, namely DefineSlot and DefineSlotEntry, that are just needed as helpers for thread-safety when running ROOT in multithreaded mode.
- **Filter:** Removes rows in the dataset based on some boolean masking. It can receive any valid C++ chain of boolean operations and also understands names of the columns as variables (this is true for any RDataFrame operation actually).
- **Range:** Only considers rows of the dataset up to the value issued by the user. This may be helpful to run the analysis quickly on few entries for validation.

Actions on the other hand are operations that return an answer to a question the user may have about the data, such as "How many entries are there in the dataset?", "What is the sum of this column of values?". The handy feature with RDataFrame (inspired by Spark in this case) is the lazy evaluation, that is an action is actually run only when the user states in their program they want to get the result. Otherwise RDataFrame will just hold the information of the action in a C++ smart pointer. This makes it easy to produce several different results in one go, ensuring at the same time that only the rows passing any filter preceding the actions are processed. RDataFrame implements many actions, the main ones are highlighted in the following list:

- **Cache:** Caches in contiguous memory entries of columns specified by the user.
- **Count:** Returns the number of rows in the dataset (after filtering).

- Visualization actions: There are many actions that return ROOT objects for visualization, such as Histo(1D,2D,3D), Profile(1D,2D,3D) or Graph. These objects all have their respective class implementation in ROOT and are very commonly used in a HEP analysis.
- Statistics about the data: Min, Max, Mean, Stdev all return the corresponding statistic of the column the user specifies.
- Sum: Returns the sum of the values in a column.
- Take: Extract a column from the dataset as a collection of values. This is of paramount importance for retrieving the raw data from the source into some flat format. A specific feature that this action enables is outsourcing the content of a ROOT tree into numpy arrays, which will be discussed later.

Finally, there are some operations that override the usual laziness of the RDataFrame interface in favor of providing some needed feature. These class of operations is called *Instant Actions* and is comprised of the following:

- Foreach: Execute a user-defined function on each entry. This actually seems in contrast with the declarative programming approach, but sometimes the users have needs that may not be satisfied with all the vast set of features RDataFrame already covers, at the expense of performance and thread-safety. The user is totally responsible for the multithreading execution with this action, though ForeachSlot is meant as a helper in that situation.
- Snapshot: Writes processed data to disk. This is RDataFrame's own interface for saving data in a new TTree and TFile, a much needed feature, and it returns a new RDataFrame object to the user as well. This means that the user can store partial results during their analysis for validation. Nonetheless, Snapshot can be forced to be a lazy action through a specific option in the interface.

RDataFrame also provides some API functions to query metadata, such as getting the type of some column or its name. These functions do not interfere with the computational graph.

Now an example will be provided to show some key differences between the traditional, imperative approach and the declarative approach of RDataFrame. Listings 3.1 and 3.2 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 // Each event contains
4 TTreeReaderArray<double> px(tree, "px"); // an array of px,
5 TTreeReaderArray<double> py(tree, "py"); // an array of py
6 TTreeReaderArray<double> E(tree, "E"); // and an array of E
7 // Each event contain particles, whose x, y position
8 // and Energy is define by px, py and E, respectively
9 // Initialize an histogram of pt against pt (momentum)
10 // with 16 bins from 0 to 4
11 TH1F h("pt", "pt", 16, 0, 4);
12 while (tree.Next()) { // Event loop
13     // For each event, iterate all particles
14     for (auto i = 0; i < px.GetSize(); ++i){
15         if (E[i] > 100){// Checks if the particle energy is greater than 100
16             // Calculate pt of a particle and add it to the Histogram
17             h.Fill( sqrt(px[i] * px[i] + py[i] * py[i]));
18         }
19     }
20 h.Draw(); // Plot filled histogram
```

LISTING 3.1: Traditional ROOT code to read a file, create a custom variable and plot an 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 3.2: RDataFrame code to read a file, create a custom variable and plot an 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 it is friendlier to read. Code in Listing 3.1 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).

On the other hand, the RDataFrame version of Listing 3.2 features:

- 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 in the style of Numpy [58] or Pandas [59]).
- Histogram is created automatically with the resulting vector of values from the previous operation.
- No type definition, type safety is handled underneath by inference.
- Define operation in line 3 makes use of JIT (Just in Time) compiled 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.

3.1.3 Automatically generated Python bindings for ROOT

Python is a popular, open-source, dynamic programming language with an interactive interpreter. Its interoperability with other programming languages, both for extending Python as well as embedding it, is excellent and many existing third-party applications and libraries have therefore so-called “Python bindings.”. PyROOT provides Python bindings for ROOT: it enables cross-calls from ROOT/Cling into Python and vice versa, the intermingling of the two interpreters, and the transport of user-level objects from one interpreter to the other. PyROOT enables access from ROOT to any application or library that itself has Python bindings, and it makes all ROOT functionality directly available from the python interpreter.

3.2 Apache Spark

The programming paradigm introduced by Hadoop MapReduce pioneered a model for computing in which parallel computations on data are executed on clusters of unreliable machines by systems that automatically provide locality-aware scheduling, fault tolerance, and load balancing. The initial idea behind this programming model was to work on an input dataset with a set of operators collectively forming an *acyclic graph*. This allows the underlying system to manage scheduling and to react to faults without user intervention. Knowing that not all applications could be expressed in the form of acyclic

graphs, a new project was introduced that would eventually succeed in providing the user with a general purpose programming language for cluster computations: Spark [60].

Spark is an Apache project aimed at cluster computing and based on Hadoop MapReduce, extending it to efficiently use it for more types of computations, including interactive queries and stream processing. The main feature of Spark is the ability to perform in-memory cluster computing to increase the processing speed of an application.

It has a thriving open-source community and is the most active Apache project at the moment. Spark provides a faster and more general data processing platform. Spark lets you run programs up to 100x faster in memory, or 10x faster on disk, with respect to the original implementation of Hadoop MapReduce [61].

Spark holds intermediate results in memory rather than writing them to disk which is very useful especially when the data is processed multiple times. It is designed to be an execution engine that works both in-memory and on-disk. Spark operators perform external operations when data does not fit in memory. Spark can be used for processing datasets larger than the aggregate memory in a cluster. Spark will attempt to store as much data as possible in memory and then will spill the remaining to disk.

3.2.1 Spark components

Spark is written in the Scala programming language and runs in a Java Virtual Machine (JVM) environment. It can be developed in many programming languages, including Scala, Java, Python, R. Other than the Spark core API, there are additional libraries that are part of the Spark ecosystem and provide additional capabilities in Big Data analytics and Machine Learning areas, shown in Fig 3.5.

Normally, Spark uses the Hadoop Distributed File System (HDFS) for data storage. This ensures compatibility with many different data formats and DBMS developed within the Hadoop ecosystem (HBase, Cassandra, Parquet etc.). The specifics of this system are such that data gets automatically distributed over different machines in the cluster for fault-tolerance (with at least three replicas) and availability purposes. The data management

can then be handled in different modes, starting from the standalone configuration and scaling to a diverse range of distributed computing frameworks such as YARN, Mesos or even Kubernetes.

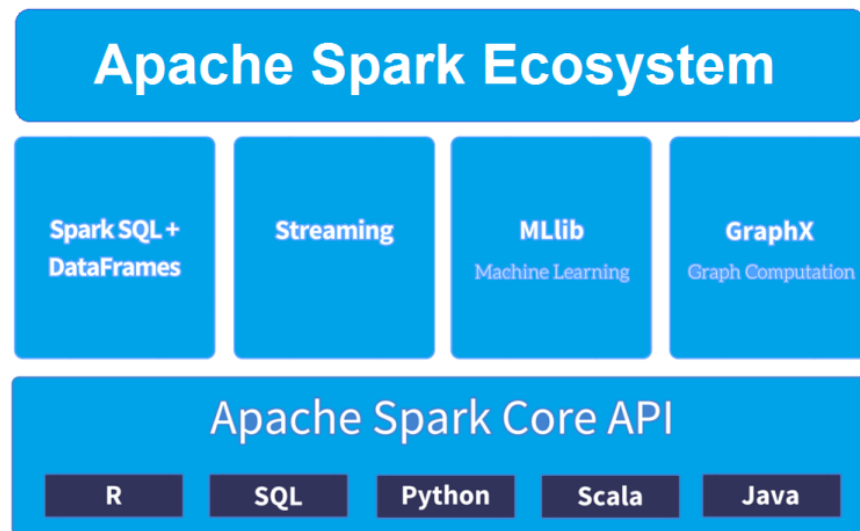


FIGURE 3.5: The Apache Spark ecosystem.

3.2.2 Spark program overview

The architecture of a Spark program is as follows. On the user side, where the program is written and started, a specific object called `SparkContext` is instantiated that will serve as an endpoint to send and retrieve back information from the cluster. The user application that is attached to the `SparkContext` is also known as driver program or simply driver. Once connected to the available distributed resources, Spark takes over executors on distributed nodes in the cluster, which are processes in the distributed nodes that run computations and store data for your application. Next, it sends your application code to the executors through the `SparkContext` and spawns tasks to be run and completed. During the execution of the tasks, the reduction phase kicks in to start merging the partial

results. When all the results have been merged, they get sent back to the user through the SparkContext. Fig. 3.6 shows the anatomy of a Spark program from the point of view of the cluster.

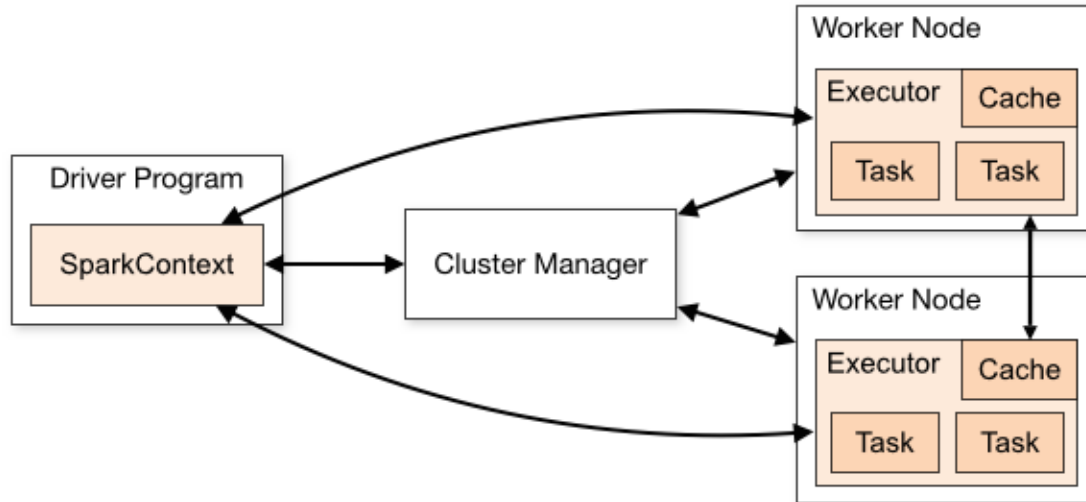


FIGURE 3.6: Components of a Spark program from the perspective of the cluster [62].

3.3 SWAN

SWAN (Service for Web based ANalysis) is a platform to perform interactive data analysis in the cloud. The analysis are powered by a common tool in the Data Science community, known as (Jupyter) notebooks. In particular, these are documents that allow to combine code, text, plots and other media in the same place, presenting the user empty cells where they can type, execute and see the results of code inline. Notebooks usually run in a browser, and SWAN connects them together with a suite of CERN services that allow for an easier reproducibility and sharing of scientific code, accessing and demonstrating scientific software and finally preserving the analyses for future reference (Fig. 3.7). One of

the goals of the service is to boost the productivity of scientists and engineers by allowing them to focus solely on the solution of their problems without investing resources in the creation, configuration and maintenance of software and hardware environments [63].

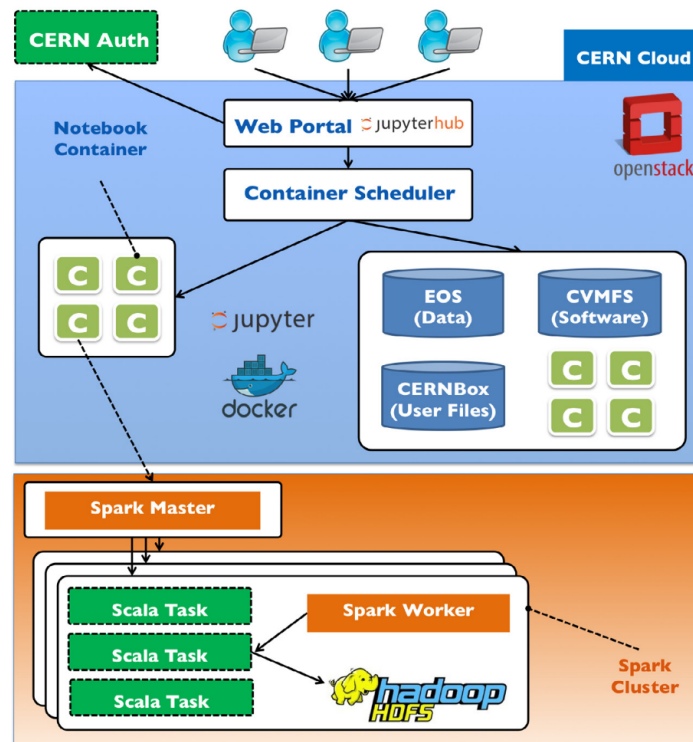


FIGURE 3.7: An overview of the main components in SWAN. From the user perspective (top of the figure) SWAN presents as a web portal with easy access to files and to the Jupyter notebooks for programming. The user can login to the portal with her CERN credentials, as a consequence a new Docker container will be spawned. The container will be automatically hooked to a suite of CERN services for software and storage. Finally, inside a Jupyter notebook, the user will be able to connect to external computing resources such as the Spark cluster.

SWAN stands on three conceptual pillars:

- Storage
- Software
- Computing

3.3.1 Storage

SWAN leverages cloud storage, to provide users with an easy access to their documents, code and data. The service responsible for data storage is EOS, the disk-based low-latency storage service developed at CERN.

The system became operational in 2011 for the ATLAS experiment with 2 PB in size and has grown steadily to 140 PB in four years afterwards [64]. The total storage space today is segmented into six independent failure domains (instances) for the four LHC experiments, a shared experiment instance for smaller experiments (EOSPUBLIC) and a generic user instance (EOSUSER) for all CERN users. Main difference is the average file size with the EOSUSER instance storing the smallest files. The architecture is based on separate metadata access and data access services with a message queue to exchange information between the two. All the services have been implemented using the XRootD client-server framework. EOS provides a very flexible and extendable disk storage platform for a large user community at CERN that has demonstrated an unprecedented scalability for a low-cost storage system in high energy physics.

To cope with the expectations that users have to access their data from different devices than a computer, such as smartphones and tablets, a new synchronization and sharing tool was introduced called CERNBox. It is based on components provided by the OwnCloud open software stack and is federated with EOS. This enables a coherent view on data plus access and synchronization from different devices, while the entire data storage system is accessible online by the traditional batch jobs for running physics analyses. Each user

is given her own personal quota on EOS of 1 TB that can be accessed through CERNBox also on SWAN.

3.3.2 Software

The software needed for the analysis can already be found in the user session, without the need to install additional packages most of the times. This feature is brought by CVMFS, a distributed aggressively-cached file system that is mounted directly on the SWAN virtual machine that the user is given access to.

CVMFS stands for CernVM File System and it provides a scalable, reliable and low-maintenance software distribution service. It was developed to assist High Energy Physics (HEP) collaborations to deploy software on the WLCG. To the users, it appears as a structure of files and directories, mounted on the universal namespace `/cvmfs`, while they are really hosted on CERN web servers. Software on CVMFS usually comprises many small files that are frequently opened and read as a whole. Furthermore, the software use case includes frequent look-ups for files in multiple directories when search paths are examined.

CVMFS is actively used by small and large HEP collaborations. In many cases, it replaces package managers and shared software areas on cluster file systems as means to distribute the software used to process experiment data. For the experiments at the LHC, CVMFS hosts several hundred million files and directories that are distributed to the order of hundred thousand client computers.

Different experiments have different namespaces to host their software in CVMFS. SWAN mainly relies on the official releases of the software for experimental physics (EP-SFT) department at CERN, namely LCG releases which are available under the namespace `/cvmfs/sft.cern.ch/`.

3.3.3 Computing

SWAN allows hardware resources to be plugged in to the user session, to provide more computing power or generically more performance. Currently, one prominent example would be the access to an external Spark cluster, where the user can send their operations. In turn, Spark workers are spawned and the computations are parallelised on the available resources that can be dynamically scaled to cope with the requirements of the analysis.

The cluster framework implementation that will be used in this thesis work is Spark on Kubernetes [65]. This is a container orchestrator that enables running multiple instances of containers on different machines in a cluster environment [66]. To be able to use Spark within this framework, a Docker image has to be supplied with the needed software. This is done automatically in SWAN before the analysis of the user starts.

3.4 PyRDF

PyRDF is a python library that sits on top of ROOT RDataFrame. It creates a layer that allows for distributed computation on the data, without changing the programming model and with extremely small modifications to the original ROOT analysis. Some of the key features of the package are:

- Blending C++ and Python code
- Distributing a declarative analysis to a (Spark) cluster

In the scope of this thesis, the library has been brought from a proof of concept stage to a fully functional implementation, accounting for the needs of different physics analyses.

3.4.1 Distributing ROOT RDataFrame with Spark

In PyRDF, each node of the ROOT RDataFrame computational graph is associated with one (or more) python objects, specifically tailored to handle different tasks. More specifically, PyRDF includes the following classes to address the different parts of the graph:

- **Node**: a generic node in the computational graph, holding information about the operation that it represents and its children nodes.
- **HeadNode**: a specialized node, representing the first object in the graph, i.e. the ROOT RDataFrame. This node can also retrieve information on the underlying data such as the name of the TTree, the number of entries etc.
- **Operation**: a class holding metadata on a ROOT RDataFrame operation. This is always coupled to a Node object and is used to make a call to the corresponding real RDataFrame function.
- **Proxy**: a wrapper object to a node in the computational graph. This object helps to keep track of whether the node is assigned to a variable by the user, to ease the graph pruning process.
- **ActionProxy**: a Proxy with specialized functionalities for actions of the RDataFrame API. When they are accessed for the first time by the user, they trigger the execution of the computational graph.
- **TransformationProxy**: a Proxy with specialized functionalities for transformations of the RDataFrame API. Ensures that the user can still call other operations and create the functional chain without triggering the execution, or triggering it when an instant action is called, e.g. AsNumpy or Snapshot.

In any python script, the first node created is always the one representing the RDataFrame, i.e. `PyRDF.RDataFrame()` in the PyRDF API. This call creates an `HeadNode` object wrapped by a `TransformationProxy`. Subsequent calls to transformations, e.g. `Define` and `Filter` will always create `Node` objects wrapped by `TransformationProxy` and linked to an `Operation`.

The first time an action is called on data, e.g. Count or Histo1D, a Node object will be created wrapped by an ActionProxy. Then, like with the RDataFrame API, accessing this proxy will result in the execution of the graph and the retrieval of the result for the user. An example workflow is provided in Listing 3.3 and in Fig. 3.8.

```
1 import PyRDF
2 # Create an RDataFrame with 100 sequential entries between 0 and 99.
3 df = PyRDF.RDataFrame(100)
4 # Filter entries less than 5
5 flt_1 = df.Filter("rdfentry_ < 5")
6 # Define a new column with the square value of the entries
7 def_1 = df.Define("x", "rdfentry_ * rdfentry_")
8 # Create an histogram out of the new column
9 h_1 = def_1.Histo1D("x")
10 # Retrieve the histogram and plot it
11 h_1.Draw()
```

LISTING 3.3: Simple script using PyRDF to create the RDataFrame computational graph.

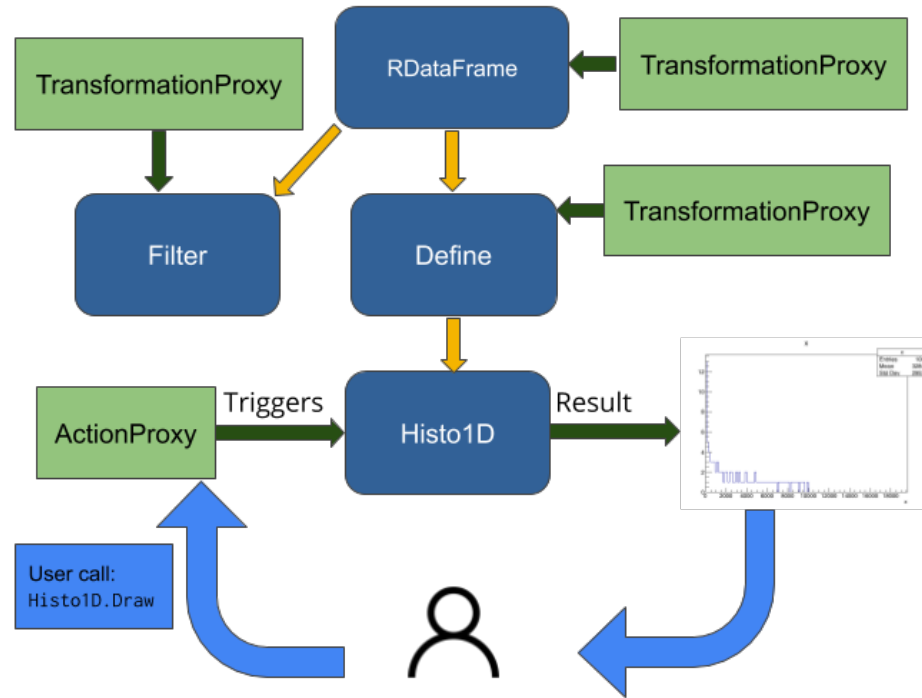


FIGURE 3.8: Python object layout created by the code in Listing 4.4. The blue boxes represent nodes in the graph, while the green ones show the corresponding proxies. When a user access the last action node in the graph (an histogram node in this case), the ActionProxy triggers the execution of the whole graph and the achieved result is given to the user.

To distribute the graph of operations as a Spark job, specific map and reduce functions were implemented, in particular:

- Map: each PyRDF node holding an operation is mapped to the corresponding PyROOT RDataFrame API function, which is then executed on the chunk of data being processed on the current executor. The chain of operation results is then stored in a Python list.

- Reduce: the reduce step happens on the cluster, using the `TreeReduce` function of the Spark API. Two lists of partial results coming from the execution of the graph on different executors are iteratively merged, aggregating same-operation results (e.g. histogram entries are added to other histogram entries, a partial sum of values in a column is added to another partial sum etc.). When all the partial lists of results have been merged remotely, the list with the final results is then serialized and sent back to the user.

3.5 Blending everything together

This chapter described the various tools used in this thesis work. PyRDF, a thin python layer for distributing ROOT RDataFrame, has been developed extensively to provide new features for physics analyses. To summarize, a realistic HEP analysis scenario that uses the techniques described in this chapter could be as follows:

1. The physicist logs in to SWAN with CERN credentials.
2. She creates a jupyter notebook where the analysis code will be written, a python script that can still make use of C++ functions thanks to ROOT. The logic of the program will have to follow the declarative paradigm, operations to be processed on data will be simply described through the RDataFrame API, without the need for the user to handle each step of the processing.
3. Before launching the program, a connection with the CERN computing cluster will be established through the specific user interface available in the SWAN jupyter notebook.
4. At program start, a number of Spark executors will be spawned and dynamically increased or decreased following the needs of the analysis
5. PyRDF will take care of converting the RDataFrame computational graph into a Spark job, with custom map and reduce functions.

6. Data are read remotely by the Spark executors from the source indicated by the user, most of the time data will reside on EOS. Chunks of data are read separately to minimize IO operations and improve parallelization.
7. The Spark job is thus executed on the executors, which take care of producing the partial results and merging them.
8. The final results of the analysis are serialized and sent from the executors back to the user SWAN session.

The work that has been done to harmonize the MapReduce technique, in the implementation given by Spark, with the more established CERN tools thus provides a completely new way to access and process HEP data on the fly and interactively. Examples of applications will be shown in the next chapter.

Chapter 4

Distributing physics analyses with Spark and RDataFrame

The various tools described in chapter 3 collectively enable a framework that could be used in HEP analyses, especially considering the interactive and friendly approach provided to the user. In this chapter, different examples of analysis will be shown with various degrees of data size, computational weight and scope. The first example will be taken from an established reference analyses in the ROOT framework and will be updated to run with Spark on the clusters. It will provide a quick benchmark both for the software developed and for the distributed framework. Its results and the insights derived will guide the work of the following two use cases. The second use cases will involve a physics analysis going through some of the more common steps and at the same time converting it to run distributedly. The third and final example will show how the workflow enabled by PyRDF can be used as an input for different analysis models.

4.1 Dimuon spectrum

This analysis takes data from the CMS experiment recorded in 2012 during Run B and C and extracts the di-muon spectrum, that is the distribution of mass in the reference frame¹ of the collision between two muons [67]. The di-muon spectrum is computed from the data by calculating the invariant mass² of muon pairs with opposite charge. The resulting plot shows particle resonances³ in a wide energy range from the η meson at about 548 MeV up to the Z boson at about 91 GeV (Fig. 4.1).

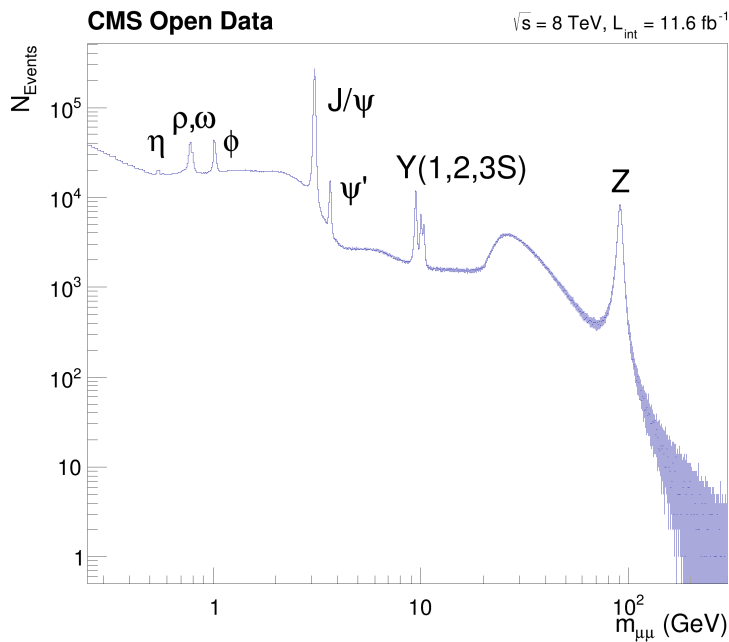


FIGURE 4.1: Di-muon spectrum plot resulting from the analysis on CMS open data derived from Run B and C in 2012.

¹A system of coordinates and physical reference points that fix objects and measurements in that system.

²Portion of mass independent of the motion of objects in the reference frame. In the center-momentum frame, where the linear sum of the momentum of the two colliding particles vanishes, the invariant mass corresponds to the whole mass of the system.

³Peak located around a certain energy in scattering experiments related to their cross section.

The input data is comprised of two files that collectively amount to ~ 8 GB and ~ 60 million events. These in turn are the result of a previous conversion from the original AOD dataset to a flat collection of vectors, called the NanoAOD data format. In particular this NanoAOD dataset contains only the needed information regarding the muons produced after the collisions in LHC for that particular Run. The description of the variables in the dataset is given in Tab. 4.1

TABLE 4.1: Description of the variables in the di-muon dataset. The square bracket notation in the **Type** column indicates the length of the corresponding vector [68].

Variable	Type	Description
nMuon	unsigned int	Number of muons in this event
Muon_pt	float[nMuon]	Transverse momentum of the muons
Muon_eta	float[nMuon]	Pseudorapidity of the muons
Muon_phi	float[nMuon]	Azimuth of the muons
Muon_mass	float[nMuon]	Mass of the muons
Muon_charge	int[nMuon]	Charge of the muons (either 1 or -1)

4.1.1 Conversion to PyRDF

The reference code for the analysis is available as a Python script in the official ROOT tutorials. The C++ classes and functions are made available through the PyROOT bindings. Since the code doesn't involve any complex C++ function, the changes required in order to run the code with Spark on the cluster are very minimal and are shown in Listing 4.1 by the commented `++++` tags.

```

1 import ROOT
2 import PyRDF # ++++
3 PyRDF.use("spark") # ++++
4

```

```
5 # ---- Enable multi-threading
6 # ---- ROOT.ROOT.EnableImplicitMT()
7
8 # Create dataframe from NanoAOD files
9 files = ROOT.std.vector("string")(2)
10 files[0] = "root://eospublic.cern.ch//eos/root-eos/
    cms_opendata_2012_nanoaod/Run2012B_DoubleMuParked.root"
11 files[1] = "root://eospublic.cern.ch//eos/root-eos/
    cms_opendata_2012_nanoaod/Run2012C_DoubleMuParked.root"
12
13 # ---- df = ROOT.ROOT.RDataFrame("Events", files)
14 df = PyRDF.RDataFrame("Events", files) # ++++
15 # The analysis continues as the original one
16 # ...
```

LISTING 4.1: Code changes between PyROOT and PyRDF versions of the di-muon analysis

So, with a net increase of one line of code, PyRDF enables running the code on a cluster of workers instead of executing it on a single machine.

4.1.2 PyROOT vs PyRDF comparison

The first comparison for this analysis was drawn between running the original code with multithreading enabled and running the converted code with Spark using the same number of cores as the threads used locally. The whole comparison was run in SWAN with the following setup:

- PyROOT code: multithreaded with 4 threads.
- PyRDF with Spark: 4 executors with 1 core per each.

Each of these two configurations was run repeatedly for a total of 1000 runs each. The distributions of the execution time are shown in Fig. 4.2.

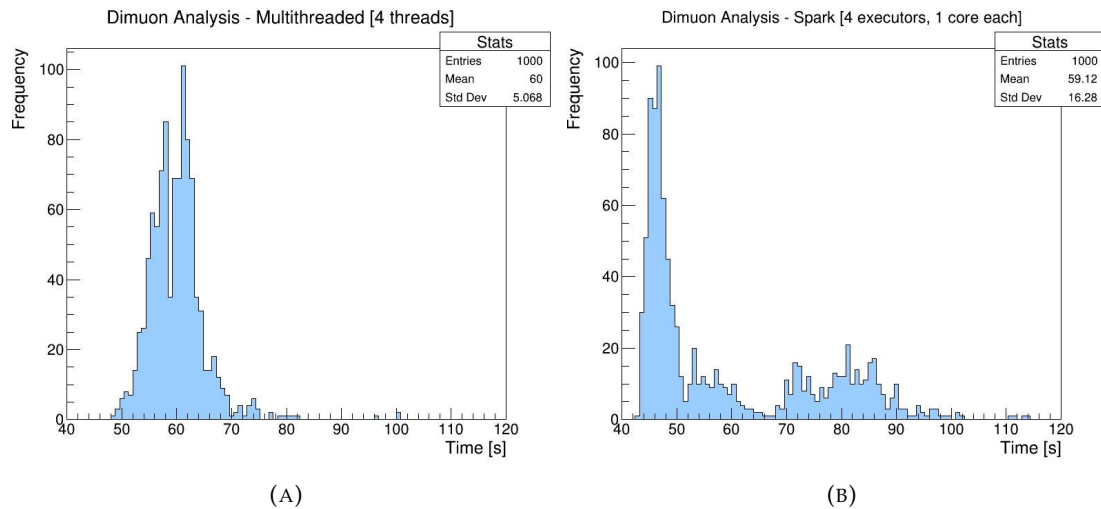


FIGURE 4.2: Execution time of 1000 Dimuon analysis runs in a SWAN session. a: multithreaded configuration with 4 threads. b: Spark configuration with 4 workers, 1 core each.

The main takeaways of this comparison are:

- Even though executing the code on the Spark clusters intrinsically involves some overhead due to spawning the executors at runtime, on average the execution time is equal to the one achieved by the multithreaded version on the SWAN machine.
- The two histograms show quite different distribution shapes. The one on the left shows a clear spike around 60 s suggesting that if reliability is a concern the multi-threaded configuration would achieve the same execution time most often. The one on the right instead shows a broader curve that could be due to different factors, not the least being the variability of shared resources on the cluster, but in this case the most frequent time achieved for the analysis is 46.8 s, compared to the most frequent time of the multithreaded configuration that is 61.2 s.

4.1.3 Scalability of the analysis

The comparison with the original PyROOT code was meant to demonstrate that on equal ground there is no loss in performance when using PyRDF. Hopefully the performance should scale linearly with the number of cores, at least up to a certain point. In this regard, a new set of time measurements for the analysis was taken with more cores available each time, namely 1, 4, 8, 16, 32 cores. Fig. 4.3 shows the average execution time for each configuration.

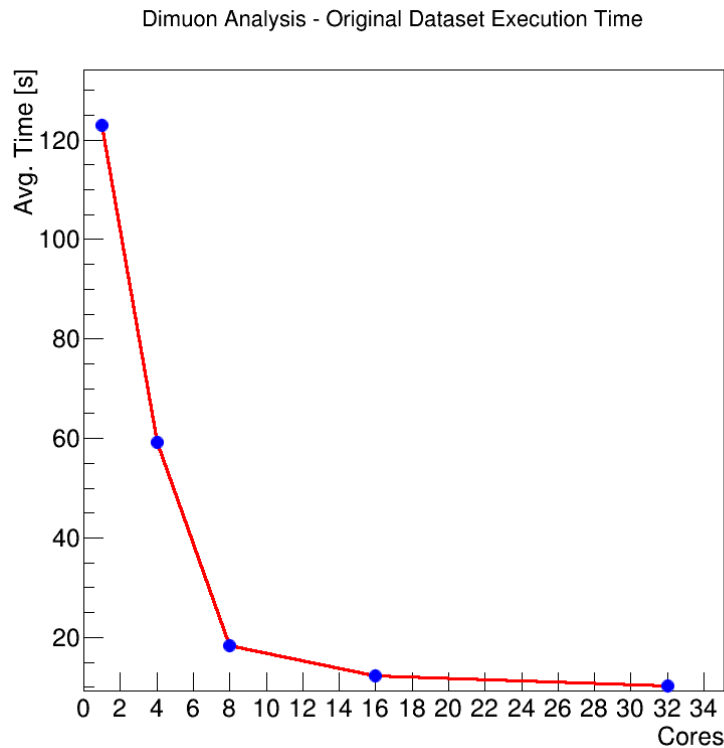


FIGURE 4.3: Execution time (in seconds) of the Dimuon analysis run on the Spark clusters with different number of cores. The blue dots represent the average time of 50 runs.

The figure shows the time decrease of the analysis execution with the addition of more workers to process data. Nonetheless, it looks like the speedup is not always ideal and that using more than 8 cores for this analysis doesn't deliver the expected performance. This is better shown in Fig. 4.4.

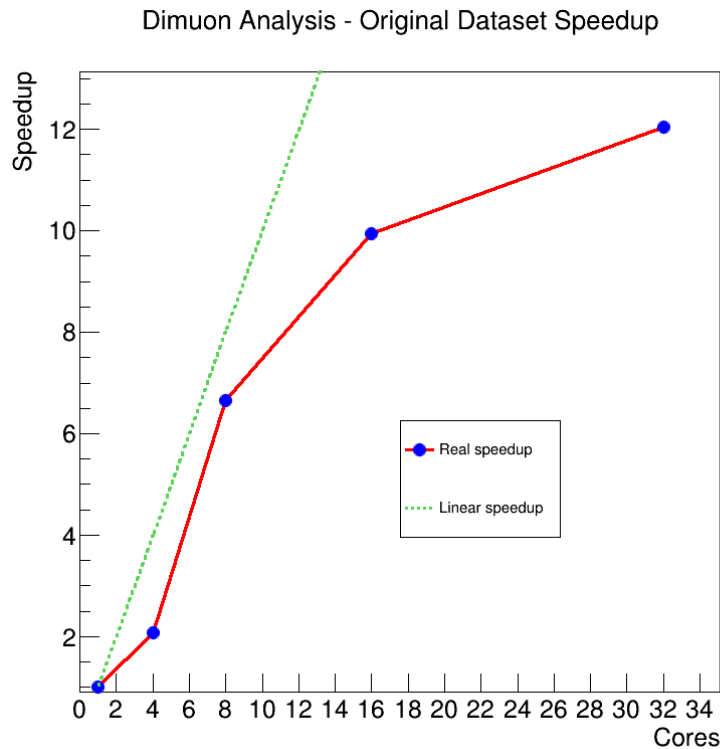


FIGURE 4.4: Speedup of adding more cores for running the Dimuon analysis. The green dashed line shows the ideal scaling, that is linear with the number of cores.

Looking at the results of this first approach to the analysis, no configurations above 32 cores were tried, instead the scalability of the analysis was addressed in terms of its data size. In fact, an input dataset of 8 GB could be the bottleneck in this case, since under a

certain size the overhead given by spawning the executors is higher than the actual time spent in processing the chunks of data. To verify this hypothesis, a data augmentation technique was applied by replicating the events of the initial dataset to reach the size of ~ 208 GB. It is worth noticing that this procedure doesn't invalidate the analysis, since physics collision data is made of statistically independent events. The code was run again with all the previous configurations as well as adding 64 and 128 cores runs.

The results are presented in Fig. 4.5 and Fig. 4.6. They collectively confirm the initial suppositions on the data size, especially looking at the second one it is clear that the software can achieve perfectly linear performance up to a certain number of cores given the right amount of data to process. In this case the speedup curve falls greatly at 128 cores, maybe due to the dataset being still too little for that number of workers or possibly for new reasons that could be further studied and explored.

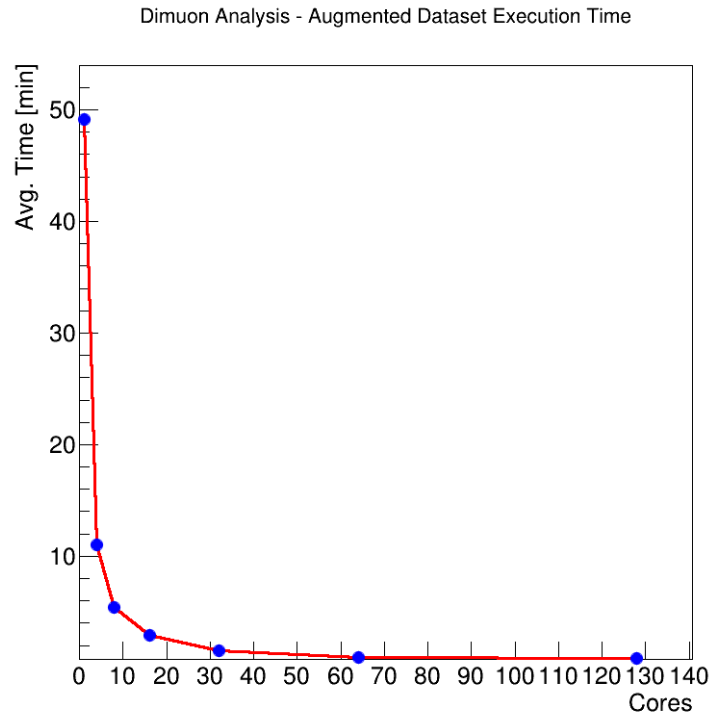


FIGURE 4.5: Execution time (in minutes) of the Dimuon analysis with the augmented input dataset, with increasing number of cores. The blue dots represent the average time of 20 runs.

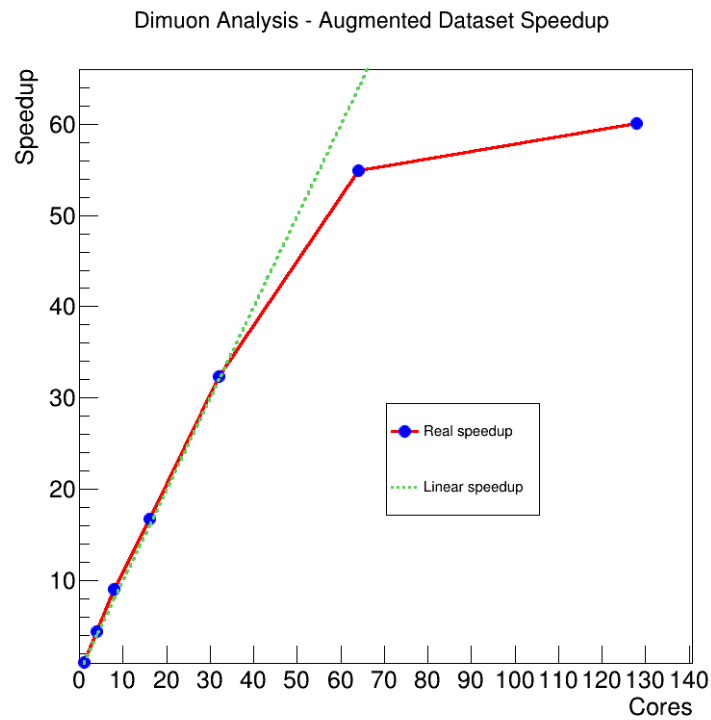


FIGURE 4.6: Speedup of adding more cores for running the Dimuon analysis with the augmented input dataset. The green dashed line shows the ideal scaling, that is linear with the number of cores.

4.2 Higgs boson decay to two τ leptons

The second use case in this thesis focuses on translating a full physics analysis written with RDataFrame, loosely based on the official CMS publication of the decay of an Higgs boson to two τ leptons. Originally, the analysis was run on events recorded by the CMS experiment at the LHC in 2011 and 2012. The dataset corresponds to an integrated luminosity of 4.9 fb^{-1} at a centre-of-mass energy of 7 TeV, and 19.7 fb^{-1} at 8 TeV [69]. The original analysis though was conducted with traditional ROOT classes, using an imperative programming model. The new analysis uses the declarative programming model and is a good starting point for the distribution to the cluster. The main outcome of the analysis is a set of plots representing different attributes of the particles (see 4.7 for an example).

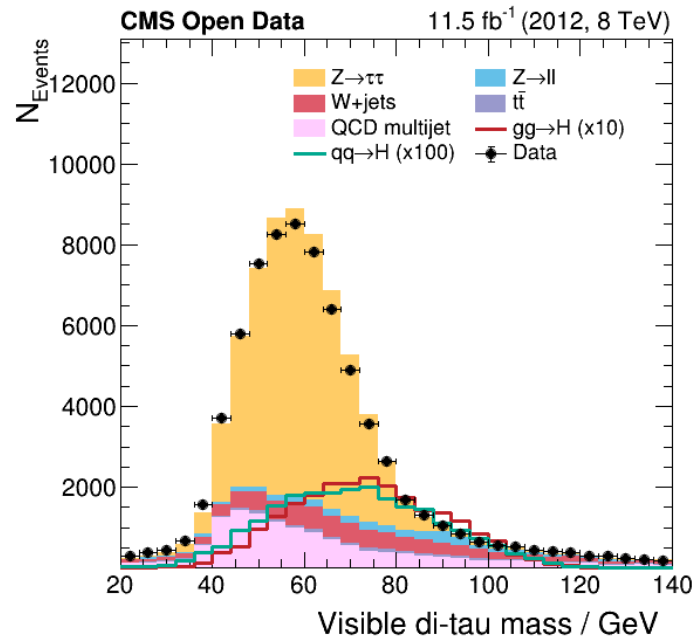


FIGURE 4.7: Sample plot of the Higgs to τ analysis, showing the mass distributions of the particles and the various background data.

The input dataset is spread across 9 ROOT files, each containing information about different particles in the event but with a similar schema. The files again follow the NanoAOD format, so they consist of collections of vectors of fundamental data types. The whole dataset sums up to 70 GB, summary statistics about the files are presented in Tab. 4.2

TABLE 4.2: Summary statistics for the different files used in the Higgs to τ leptons analysis.

Dataset Name	Size [GB]	Records	Clusters	Columns
GluGluToHToTauTau	0.20	476 963	8	69
VBF_HToTauTau	0.23	491 653	9	69
DYJetsToLL	9.25	30 458 871	349	69
TTbar	3.53	6 423 106	134	69
W1JetsToLNu	10.73	29 784 800	398	69
W2JetsToLNu	12.16	30 693 853	453	69
W3JetsToLNu	6.54	15 241 144	243	69
Run2012B_TauPlusX	10.92	35 647 508	436	62
Run2012C_TauPlusX	15.89	51 303 171	588	62

The analysis is composed of three main steps:

- **Skimming:** The input files are processed in order to find good particle candidates for the decay, filtering on some predefined thresholds and defining new variables from the original ones. The newly created dataset is saved for later use. This is the most computationally demanding task and is written in C++. The program is then compiled and run on the user's machine.
- **Producing Histograms:** Starting from the saved snapshots after the Skimming task, histogram objects are produced after some final filtering. Each histogram represents a different attribute and is thus created with different parameters (number of bins,

range on the axis). The objects are saved again in a separate ROOT file. This task is less demanding than the first and is written in a Python script.

- Plotting Histograms: The ROOT file containing the histogram objects is opened and each object is plotted with a specified set of graphical attributes. This part is the least demanding and is also written in Python.

The following section will describe the needed modifications to run the analysis on the cluster.

4.2.1 Conversion to PyRDF

The first step was to understand which parts of the analysis actually needed to be converted. Since the skimming step is the most computationally intensive, it has been the focus of the work for the conversion to PyRDF of this analysis. The other two steps, while they can be converted to run with Spark distributedly thanks to the seamless integration between PyRDF and ROOT, do not represent demanding tasks so they can be as easily performed on the user's machine in a timely manner.

The original skimming task is written in a file called `skim.cxx`. The general logical steps for its conversion are:

- Translate the RDataFrame C++ API to its Python counterpart, keeping all the transformations and actions that make up the computational graph in the main file called `skim_pyrdf_spark.py`.
- Keep the more complicated functions as C++ code in a separate header called `skim.h` that will be sent at runtime to the Spark executors and declared to the ROOT interpreter to allow for using C++ code within the Spark Python API.

An example of such workflow is given by the function `FindMuonTauPair`, used to retrieve $\mu - \tau$ pairs representing good candidates for the Higgs decay. This function is translated from C++ to Python with just few modifications (the minimal required to account for the syntax changes between the two languages), but it makes a call to another function

`build_pair` which contains some more complex computations that are better left in C++ (see Listings 4.2 and 4.3). The complete translation can be found in the project repository [70].

```

1 def FindMuonTauPair(df):
2     return df.Define("pairIdx",
3         "build_pair(goodMuons, Muon_pt,\
4             Muon_eta, Muon_phi, goodTaus,\
5             Tau_relIso_all, Tau_eta, Tau_phi)" \
6         .Define("idx_1", "pairIdx[0]") \
7         .Define("idx_2", "pairIdx[1]") \
8         .Filter("idx_1 != -1", "Valid muon in selected pair") \
9         .Filter("idx_2 != -1", "Valid tau in selected pair")

```

LISTING 4.2: Python code for the `FindMuonTauPair` function. Note that `build_pair` will be declared to the interpreter at runtime from the header, so it's not defined in the Python code.

```

1 std::vector<int> build_pair(RVec<int>& goodMuons, RVec<float>& pt_1,
2     RVec<float>& eta_1, RVec<float>& phi_1, RVec<int>& goodTaus,
3     RVec<float>& iso_2, RVec<float>& eta_2, RVec<float>& phi_2)
4 {
5     // Get indices of all possible combinations
6     auto comb = Combinations(pt_1, eta_2);
7     const auto numComb = comb[0].size();
8     // Find valid pairs based on delta r
9     std::vector<int> validPair(numComb, 0);
10    for(size_t i = 0; i < numComb; i++) {
11        const auto i1 = comb[0][i];
12        const auto i2 = comb[1][i];
13        if(goodMuons[i1] == 1 && goodTaus[i2] == 1) {
14            const auto deltar = sqrt(
15                pow(eta_1[i1] - eta_2[i2], 2) +
16                pow(Helper::DeltaPhi(phi_1[i1], phi_2[i2]), 2));

```

```
17     if (deltar > 0.5) validPair[i] = 1;
18   }
19 }
20 // Find best muon based on pt
21 int idx_1 = -1;
22 float maxPt = -1;
23 for(size_t i = 0; i < numComb; i++) {
24   if(validPair[i] == 0) continue;
25   const auto tmp = comb[0][i];
26   if(maxPt < pt_1[tmp]) {
27     maxPt = pt_1[tmp];
28     idx_1 = tmp;
29   }
30 }
31 // Find best tau based on iso
32 int idx_2 = -1;
33 float minIso = 999;
34 for(size_t i = 0; i < numComb; i++) {
35   if(validPair[i] == 0) continue;
36   if(int(comb[0][i]) != idx_1) continue;
37   const auto tmp = comb[1][i];
38   if(minIso > iso_2[tmp]) {
39     minIso = iso_2[tmp];
40     idx_2 = tmp;
41   }
42 }
43 return std::vector<int>({idx_1, idx_2});
44 }
```

LISTING 4.3: C++ code for the build_pair function, written in the skim.h header file.

Converting the histogram creation code is as simple as changing ROOT.RDataFrame to PyRDF.RDataFrame, thus enabling the submission of Spark jobs transparently to the user. The two translated steps of the analysis are then configured to be run repeatedly on the cluster as benchmarks.

4.2.2 Comparison with original analysis

Similarly to the Dimuon analysis, the code was run on SWAN, both in the user machine and on the cluster with Spark, to guarantee the usage of the same physical hardware and software stack between the two configurations. For the skimming task, the C++ code was compiled and run on the SWAN user session, while the PyRDF version was used to connect to the cluster and run Spark jobs. Each run was repeated 20 times and its execution time was taken with the ROOT builtin stopwatch, in order to have the same means of time measurement between the two different programming languages used.

A first set of measurements was taken with the following configurations:

- C++: Multithreaded with 4 threads
- PyRDF: Spark with 4, 8, 16 executors.

The configurations stop at 16 cores maximum because of early signs of loss in performance. In Fig. 4.8 is clear that the 16 executor configuration is misbehaving in some regard, since its mean execution time is even higher than what 4 executors take on average. It happens that this is in part due to the peculiar data layout for this analysis. The smaller datasets, namely **GluGluToHToTauTau** and **VBF_HToTauTau**, also have less clusters because of the standard way in which ROOT flushes data to disk. PyRDF exploits the modularity of the clusters to read only the minimal amount of data needed to process on a particular executor, thus parallelizing on the number of clusters. Thus, while the workers could be parallelizing on many more partitions if they were only dealing with datasets split along a high number of clusters, they are instead limited to deal with 8 partitions, that is the minimum number of clusters available through all the files of the dataset.

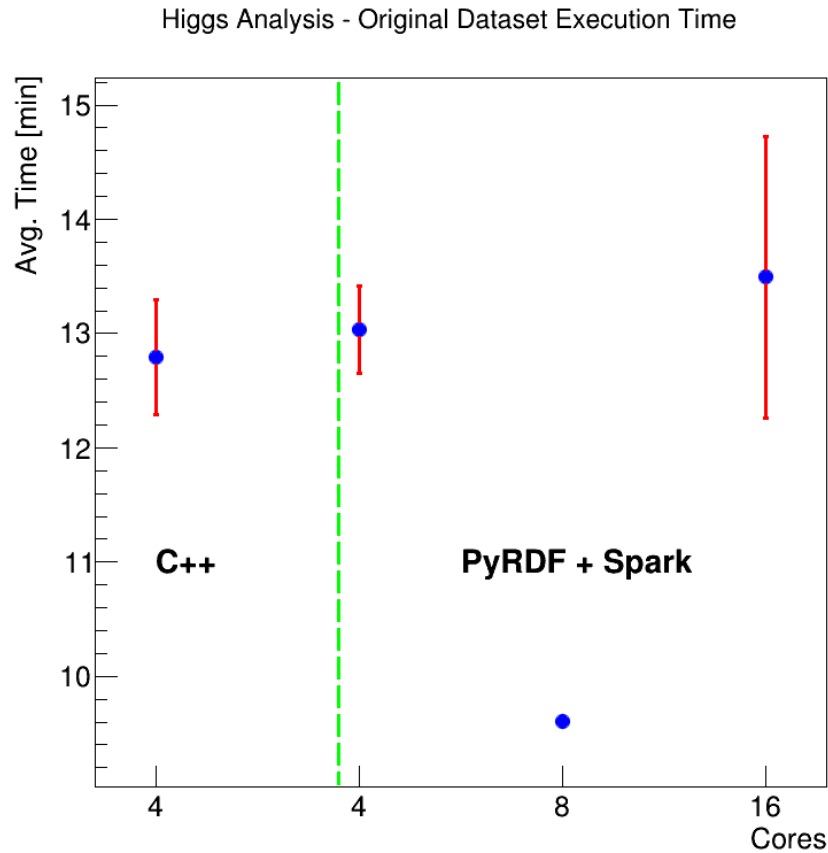


FIGURE 4.8: Average Execution time of 20 runs of the Higgs to τ analysis. On the left side of the plot the result of the C++ multithreaded execution is shown, while on the right side the different Spark executor configurations are shown.

One solution to this could be to repartition the input datasets in order to have at least the same number of clusters as available executors. This approach leads to restabilizing the performance of the 16 executors configuration, as shown in Fig. 4.9

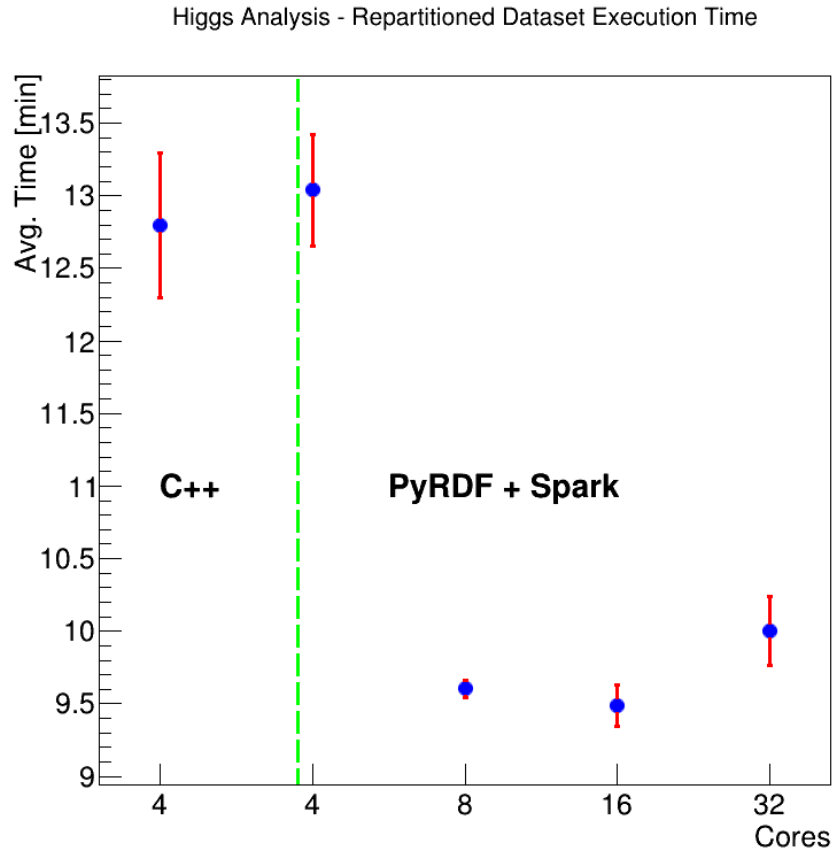


FIGURE 4.9: Average Execution time of 20 runs of the Higgs to τ analysis, after repartitioning the input datasets. On the left side of the plot the result of the C++ multithreaded execution is shown, while on the right side the different Spark executor configurations are shown.

Nonetheless, the analysis still doesn't seem to scale as the Dimuon one. One key difference between the two cases is that the current one is not processed as one dataset, e.g. one single RDataFrame computational graph. Instead, each of the 9 input files is processed separately with its own computational graph, meaning that the parallelisation is active on

the single file, but not globally on the whole analysis. It could be then that the smaller files are the bottleneck, effectively taking the same time to process even when scaling to multiple cores, resulting in a waste of time instead of a gain. This can be shown with a speedup graph on the analysis run on a single input file, the biggest one **Run2012C_TauPlusX**. Fig. 4.10 shows the scaling factor up to 32 cores.

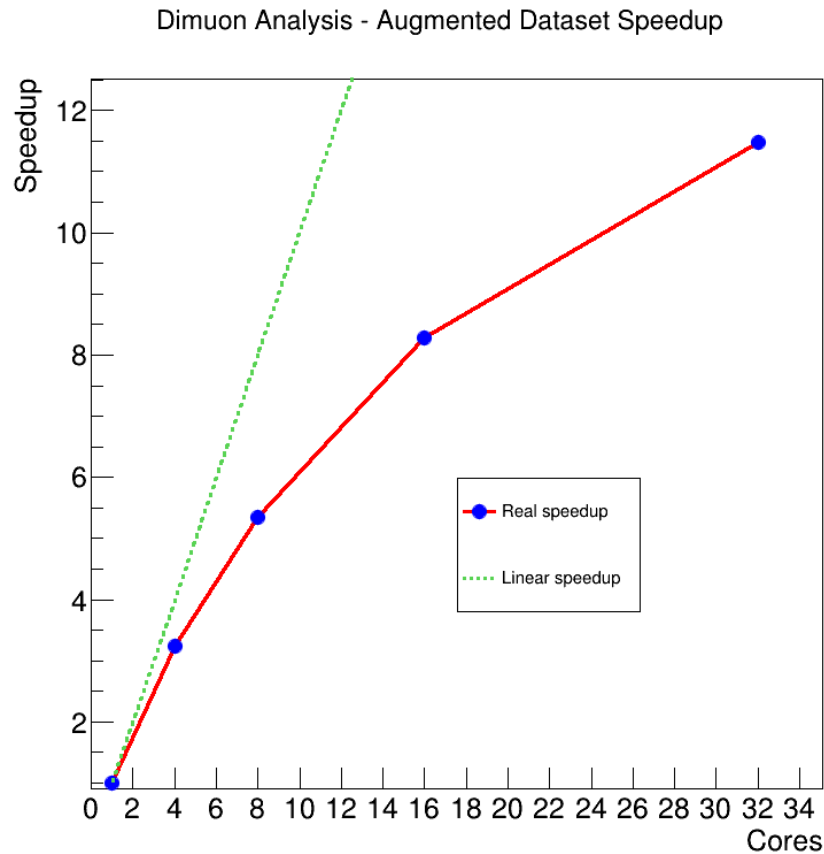


FIGURE 4.10: Speedup using the Run2012C_TauPlusX dataset, average of 20 runs of the Higgs to τ analysis.

Though not ideal, the scaling is still positive up to 32 cores, with no real loss in performance. The investigation could further continue on the reasons that make the speedup curve deviate from linearity, such as the user defined functions doing the most intensive computations. This analysis presents quite a different layout with respect to the Dimuon spectrum discussed in the previous section, not only because it is divided in multiple steps, but also for the structure of the input dataset and for all the user functions that involve heavier computations. The variety of file sizes and the fact that they are processed as separated computational graphs has proved to be a bottleneck for performance that should be further addressed in the future.

4.3 VBS Analysis

Even though the first confirmed observation of the Higgs boson dates back to 2012, there are still unresolved fundamental issues and different experiments and analysis to address them. In this context, various research groups are studying phenomena in a particular energy range where it may happen that when a pair of protons collide a quark from each one of them radiates a vector boson. The two bosons interact with each other and their decay products are detected in LHC. This phenomenon is called Vector Boson Scattering (VBS). One way to depict particle decays, very common in physics, is through the Feynman diagrams, that are much more intuitive than the complex mathematical expressions involved in the event. The Feynman diagram of the VBS phenomena for the current analysis is shown in Fig. 4.11

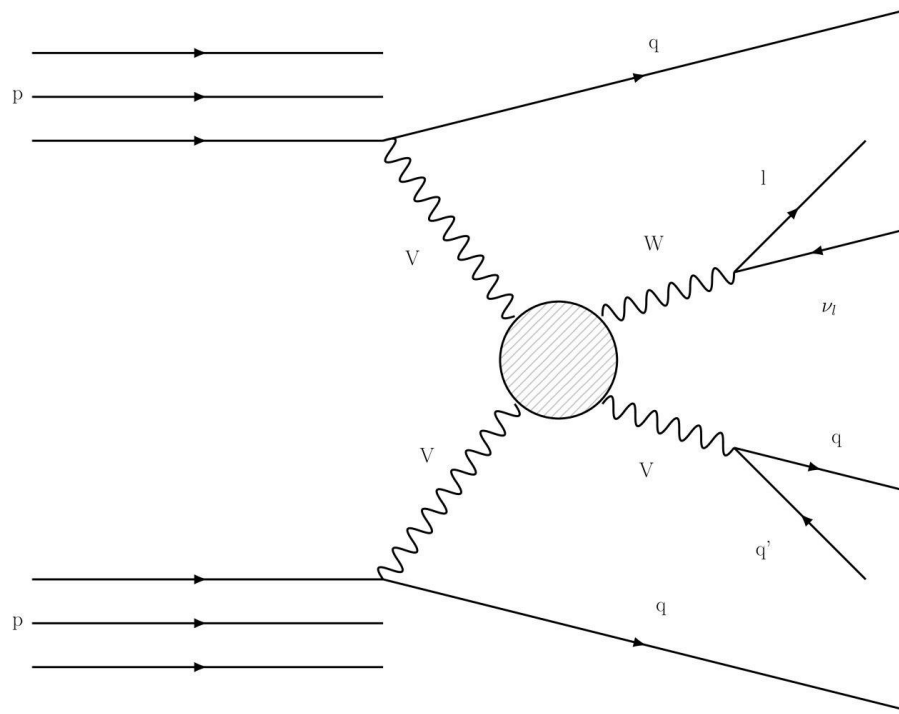


FIGURE 4.11: Feynman diagram of a VBS process contributing to the EW-induced production of events containing a hadronically decaying gauge boson (V) and a $W^\pm$ boson decaying into a lepton (l) and two jets (q/q') [71].

VBS represents a physics model independent test bench in its characteristic energy range and the analysis described focuses on data collected during Run 2 in the CMS experiment. The main goal of the analysis is to correctly classify those particles that are actually the product of a VBS event and its requirements are such that the input data has not been filtered greatly with respect to the storage stage of the data lifecycle, meaning that there is much more data to be processed than in other similar analyses. Another key aspect is that the cross section of a VBS event is approximately one thousandth the total cross section, after preselections based on the physics process involved. This roughly means that after initial filtering on the data to retrieve only the right physics characteristics, it is a thousand

times more probable to find any other event that is not coming from VBS.

It may then be the case that multivariate methods, including machine learning and deep learning, may help in classifying these events and in fact this is the path the researchers are following in this case. The workflow of the analysis can be described as follows:

- Gather the data according to its sample of origin. Each sample belongs to a different data taking process involving different particles.
- Define new variables based on the initial ones, filter them and finally weight them according to predefined specifics.
- Retrieve Numpy arrays from the processed data.
- Utilize the Numpy arrays as input data for deep learning models.

In particular my contribution to this analysis has been to provide an easy to use interface for the retrieval of the Numpy arrays and enabling distributed processing of the input data through SWAN and PyRDF. Actually, the task could have been done with at least three different tools:

- **root_numpy**: A python package that allows interfacing ROOT files to the Numpy library. It can be useful in some situations, especially when the data is in a flat format, e.g. a table. But in this case the data has a more complex, columnar structure.
- **uproot**: Another library, written completely in Python, that handles reading and writing files in the ROOT format, while being completely detached from the ROOT framework itself. It has two main drawbacks when it comes to the current analysis: remote IO operations (data resides may reside in any location of the WLCG) are quite slow and reading is not enough since the physicists need to perform their cuts and use their configurations in a reproducible way, with established operations.
- **ROOT RDataFrame**: The API provided by RDataFrame allows for performing all the common HEP operations on data, as well as converting to Numpy arrays at any point during the computational graph thanks to the AsNumpy function available in

PyROOT. As a plus, PyRDF and SWAN enable distributed computing of the chain of operations as well as saving the arrays remotely on EOS without any need of extra work from the user.

The final choice of the research team has been to rely on established CERN tools and frameworks that ensured both reproducibility and comparability with legacy code and try out the latest features that enable new workflows previously unavailable while being performant.

The input dataset for this analysis in particular contains events observed in 2017. The full dataset would contain the whole Run 2 period, 2016 - 2018. Tab. 4.3 gives roughly the amount of data and files that should be processed for each of these years. Considering the whole Run 2, the dataset could reach ~ 900 GB.

TABLE 4.3: Summary of the data available for the analysis in the year 2017.

Dataset	Total Size [GB]	# Files
Monte Carlo	250	800
Events	50	600

4.3.1 Conversion to PyRDF

The original code structure makes use of configuration files to define variables and physics cuts, while leaving other operations, such as IO, histogram creation and plotting, in a separate Python script(s).

The configuration files are written as Python scripts with the information about the physics phenomena stored in Python dictionaries. For this reason, these won't be changed during the conversion. Indeed, the scope of this part of the thesis was to change the workflow based on traditional ROOT methods to exploit RDataFrame and the Spark cluster.

The structure of the operations for the analysis resembles a graph and fits very well within the RDataFrame logic. The initial dataset goes through an initial processing, involving

definition of some new variables and a global set of cuts that selects the right physics for the VBS events. After that, new variables are defined to hold information specific to only certain aspects of the decay and following cuts can be applied to any of the variables. This can be easily expressed with the RDataFrame API, actually creating a computational graph and retrieve the results from it at different levels (Fig. 4.12).

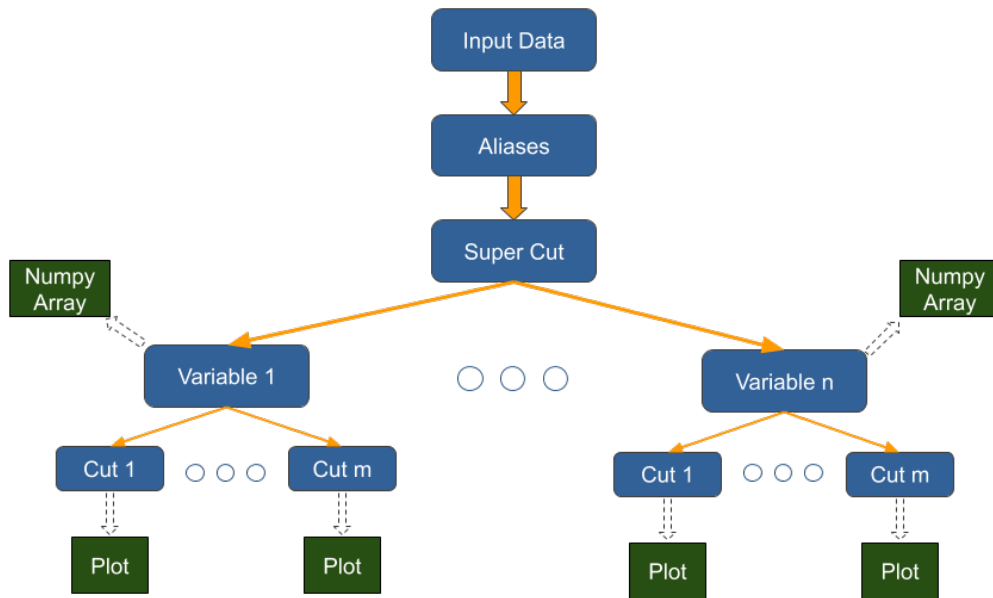


FIGURE 4.12: Workflow of the VBS analysis. The blue boxes represent transformations on data, while the green ones represent actions. The results can be retrieved at different levels of the computational graph, according to the users' needs.

In this regard, PyRDF can be achieved to replicate this graph and send its operations to the cluster as Spark jobs. A minimal example of analysis code is given in Listing 4.4. The full analysis is available in the project repository [72].

```

1 import PyRDF
2 # Creating the dataframe from the original files
3 df = PyRDF.RDataFrame("Events", VBS_DATASET_PATH)
4 # The various configuration dictionaries are imported from external scripts
5 def define_aliases(df, aliases):
6     for key in aliases.keys():
7         df = df.Define(key, aliases[key]["expr"])
8     return df
9
10 def define_cut(parent_cut, cuts):
11     for key, cut in cuts.items():
12         if cut["parent"] == parent_cut:
13             cuts[key]["rdf_node"] = cuts[parent_cut]["rdf_node"]\
14                 .Filter(cut["expr"], key)
15             define_cut(key, cuts)
16
17 def define_variable(cuts, variables):
18     for cutkey, cutvalue in cuts.items():
19         if cutvalue["doVars"] == True:
20             cutvalue["histos"] = {}
21             cutvalue["rdf_node_cache"] = [cutvalue["rdf_node"]]
22
23     for varkey, varvalue in variables.items():
24         last_def = cutvalue["rdf_node_cache"]\
25             .append(cutvalue["rdf_node_cache"][-1]\
26                 .Define(cutkey+"_var_"+varkey, varvalue["name"]))
27         if cutvalue["doHisto"] == True:
28             cutvalue["histos"][varkey] = last_def.Histo1D(
29                 (varkey, varkey,
30                  varvalue["range"][0],
31                  varvalue["range"][1],
32                  varvalue["range"][2] ),
33                 cutkey+"_var_"+varkey )
34
35         cutvalue["rdf_node"] = cutvalue["rdf_node_cache"][-1]
36
37 # Create the computational graph

```

```
38 aliases_df = define_aliases(df, aliases)
39 define_cut("supercut", cuts)
40 define_variable(cuts, variables)
41 # Take one node of the computational graph
42 rdf_node = cuts[variable][cut]
43 # Retrieve the collection of Numpy arrays as a pandas dataframe
44 numpy_coll = rdf_node.AsNumpy()
```

LISTING 4.4: Minimal code to retrieve a collection of numpy arrays from a variable of the processed VBS dataset distributedly.

4.3.2 Results

The whole 2017 dataset was processed with the same configuration files, using both traditional ROOT, RDataFrame alone and distributed with PyRDF. Since retrieving numpy arrays is a completely new functionality never tried before, the timing of the traditional ROOT analysis can only be used as a generic measure, because it refers to drawing plots. Nevertheless, RDataFrame has been used both with multithreaded enabled and with Spark to retrieve the arrays and those results can be rightfully compared. Finally, this task has led to the following execution times:

- Traditional ROOT: ~ 1 hour.
- RDataFrame multithreaded (10 threads): 45 minutes.
- RDataFrame distributed with PyRDF and Spark (10 cores): 40 minutes.

In the end, though the scaling can still be improved, this preliminary results are quite encouraging, since the tool provides completely new functionalities that can lead to different applications for the HEP analysis.

Chapter 5

Related Work

The declarative programming model that RDataFrame is based on and the distributed capabilities offered by PyRDF are applicable to any real physics analysis. During the work for this thesis, many different use cases have been studied and explored. In this chapter the most notable example of these will be briefly described to give a better insight on the wide scope of this kind of research.

5.1 TOTEM distributed analysis on AOD data

The TOTEM experiment, small in size compared to the others at the LHC, is dedicated to the precise measurement of the proton-proton interaction cross section, 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 on the accelerator ring. Hence the TOTEM collaboration aims at physics complementary to the programmes of the general-purpose experiments at the LHC. One of the datasets they are currently dealing with contains 500 TB data belonging to the reconstruction step of the data lifecycle. This data in particular is encoded in the AOD data format of the CMS experiment. The tasks they proposed to perform on this data are mainly two:

- Data Quality: analyse the input data for benchmarking purposes, retrieving standard plots and making sure the data is representing real physics events.
- Data Reduction: filter the huge amount of initial data in smaller chunks, to be easier and quicker to analyse by physicists.

On paper both tasks should be quite feasible and could potentially follow patterns similar to those described in chapter 4. The real obstacle in this case is represented by data itself, specifically by its format. The CMS AOD data format is an intricate combination of different aspects of the particles involved in the decay, defined as custom C++ classes inside a bigger framework for analysis, namely CMSSW (CMS SoftWare). CMSSW is developed and distributed as a standalone software, available publicly on github or for CERN users via LXPLUS, the traditional user console to remote resources.

This software is actually a requirement for the TOTEM tasks, since without the classes definitions available inside it the input dataset cannot be properly read, thus preventing from reaching any real result.

5.1.1 Integrating two different software stacks

The problem becomes even more evident considering that the only way to connect to the Spark clusters at CERN, for the moment, is via a jupyter notebook in SWAN. SWAN itself picks all the needed software, including jupyter extensions and Spark, from the `sft.cern.ch` CVMFS repository. CMSSW instead resides in the `cms.cern.ch` repository. The core of the work has thus revolved around trying to integrate the two software stacks, to have access to the classes needed to read data on the one hand and to connect to the cluster and run Spark jobs on the other hand.

The first steps tried included recreating the full CMSSW environment in the user SWAN session. This was ultimately achieved thanks to some configuration scripts, injected in the SWAN system at startup. Eventually, the whole process of creating the right environment for using CMSSW could be just reduced to changing some environment variables in the SWAN user system.

The Spark executors can be spawned using a different docker image than the base one, thus a new image was created and hosted on DockerHub, using the same environment script as the one created for the SWAN user session.

The data was successfully opened and some simple operations were performed in the SWAN user session. But this workflow was already hiding some errors due to the clashing of the two software stacks. Those errors became then evident when trying to connect to the cluster, which was simply impossible. Forcing to use ROOT from the CMSSW repository also removed the correct paths to the Jupyter extensions and Spark, thus preventing from even starting the connection to the cluster.

Finally, there is still work in progress, but the first problem to solve would be to enable usage of CMSSW and other experiment-specific software on SWAN, with the ability to connect to the cluster and run the analysis with Spark. The knowledge acquired during this work has been summarized and presented publicly at CERN during the first SWAN workshop [73].

Chapter 6

Conclusions

The challenges provided by the future of High Energy Physics at CERN cannot be addressed without research in the software and computing domains. The next big achievement to be reached is to handle the requirements for the HL-LHC upgrade in 2026 that will far exceed any need in computing resources and code performance present so far. Parallelization will be key in exploiting the power provided by future computing facilities, ranging from traditional CPU clusters to more heterogeneous architectures. In recent years, these needs have started to be shared by outside communities, that have brought to light new paradigms and techniques that fall under the broad area of Data Science. Though other industries have had many success stories, the data scale, the established codebase and the techniques proper to the HEP domain mean that changes take time and that feasibility and scalability of these kind of tools have to be proved several times. The work presented in this thesis builds upon recent exploratory works made at CERN [74], enriching them with new features and more thorough testing on previously unconsidered use cases. The parallelism paradigm has been used through MapReduce and Spark and applied to the high level interface to data provided by ROOT RDataFrame. Unlike other related works, the use of RDataFrame has allowed to natively work with files in ROOT format and at the same time to benefit from the Spark task scheduler to run distributably 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:

- Bringing PyRDF to a stable functional state, effectively enabling distributing ROOT RDataFrame analysis to a cluster of Spark executors without virtually any code change for the user.
- Adding completely new features for the distributed analysis, for instance saving ROOT files directly on EOS or producing Numpy arrays from processed ROOT data directly during a Spark job.
- Thoroughly exploring new use cases, including standard benchmarks for ROOT RDataFrame analysis, distributed preprocessing of ROOT data as input for machine learning models and the integration of different software stacks for distributed analysis of experiment-specific data formats.

6.1 Presentations

Results of this work have been presented in several occasions, both at CERN and abroad. A talk has been given at the INFN CCR workshop in June 2019 [75]. Presentations of the PyRDF development status have been held both for the team specifically involved in parallelization and programming models [76] and for the whole EP-SFT research group at CERN [77].

Bibliography

- [1] CERN. *Homepage*. <https://home.cern/>. Accessed on 2019-10-10.
- [2] CERN. *The Large Hadron Collider*. <https://home.cern/science/accelerators/large-hadron-collider>. Accessed on 2019-10-10.
- [3] Christiane Lefèvre. *The CERN accelerator complex*. CERN-DI-0812015. Dec. 2008.
- [4] ATLAS Experiment. *Homepage*. <https://atlas.cern/>. Accessed on 2019-10-10.
- [5] CMS Experiment. *Homepage*. <https://cms.cern/>. Accessed on 2019-10-10.
- [6] ALICE Experiment. *Homepage*. <http://aliceinfo.cern.ch/Public/Welcome.html>. Accessed on 2019-10-10.
- [7] LHCb Experiment. *Homepage*. <https://lhcb.web.cern.ch/lhcb/>. Accessed on 2019-10-10.
- [8] CERN. *End of LHC Run 1: First shutdown begins*. <https://timeline.web.cern.ch/end-lhc-run-1-first-shutdown-begins>. Accessed on 2019-10-10.
- [9] William E. Johnston et al. *Enabling high throughput in widely distributed data management and analysis systems: Lessons from the LHC*. Tech. rep. ESnet, 2013.
- [10] Gordon E. Moore. *Moore's Law*. <http://www.moorelaw.org/>. Accessed on 2019-10-10.
- [11] I.J. Taylor et al. *Workflows for e-Science*. Springer-Verlag, 2007.
- [12] I Bird et al. *Update of the Computing Models of the WLCG and the LHC Experiments*. Tech. rep. CERN, 2014.
- [13] Sebastien Ponce. "Issues in petabyte data indexing, retrieval and analysis". PhD thesis. École Polytechnique Fédérale de Lausanne, 2006.

- [14] Julie E. Managan. "Data Logistics and the CMS Analysis Model". PhD thesis. Vanderbilt University, 2009.
- [15] Mihai Niculescu. "Processing experimental data and analysis of simulation codes from Nuclear Physics using distributed and parallel computing". PhD thesis. Universitatea din Bucuresti, Sept. 2012.
- [16] Zoltan Mathe. "Feicim: A browser and analysis tool for distributed data in particle physics". PhD thesis. UCD School of Physics, May 2012.
- [17] Antonio Delgado Peris. "Distributed Late-binding Micro-scheduling and Data Caching for Data-Intensive Workflows". PhD thesis. Universidad Complutense de Madrid, June 2015.
- [18] Thomas Beermann. "A popularity prediction and dynamic data replication study for the ATLAS distributed data management". PhD thesis. University of Wuppertal, July 2017.
- [19] Science. *Breakthrough of the year*. <https://science.sciencemag.org/content/338/6114/1524>. Accessed on 2019-10-10.
- [20] Apollinari G. et al. *High-Luminosity Large Hadron Collider (HL-LHC) : Technical Design Report V. 0.1*. Tech. rep. CERN, 2017.
- [21] Panos Charitos. *ALICE upgrade plans*. <https://ep-news.web.cern.ch/content/alice-upgrade-plans-0>. Accessed on 2019-10-10.
- [22] Mark Whitehead. "The upgrade of the LHCb trigger for Run III". In: *The European Physical Society Conference on High Energy Physics*. July 2017.
- [23] Ludovico Pontecorvo. *ATLAS upgrades in LS2*. <https://cerncourier.com/a/atlas-upgrades-in-ls2/>. Accessed on 2019-10-10.
- [24] Panos Charitos. *CMS upgrade plans for LS2*. <http://ep-news.web.cern.ch/content/cms-upgrade-plans-ls2>. Accessed on 2019-10-10.
- [25] CERN. *High Luminosity LHC Project*. <https://hilumilhc.web.cern.ch/>. Accessed on 2019-10-10.
- [26] CERN. *LHC Schedule*. <https://hilumilhc.web.cern.ch/content/hl-lhc-project>. Accessed on 2019-10-10.

- [27] Tommaso Boccali. "Computing strategies for HL-LHC and beyond". In: *I2FEST2019 – IHEP-INFN Future Experiments seek Smart Technologies – Workshop*. Feb. 2019.
- [28] HSF. *A Roadmap for HEP Software and Computing R&D for the 2020s*. Tech. rep. HSF Collaboration, 2017.
- [29] James Gillies. *Luminosity? Why don't we just say collision rate?* <https://home.cern/news/opinion/cern/luminosity-why-dont-we-just-say-collision-rate>. Accessed on 2019-10-10.
- [30] Corinne Pralavorio. *LHC performance reaches new highs*. <https://home.cern/news/news/accelerators/lhc-performance-reaches-new-highs>. Accessed on 2019-10-10.
- [31] Corinne Pralavorio. *Record luminosity: well done LHC*. <https://home.cern/news/news/accelerators/record-luminosity-well-done-lhc>. Accessed on 2019-10-10.
- [32] Paul Rincon. *Work starts to upgrade Large Hadron Collider*. <https://www.bbc.com/news/science-environment-44484062>. Accessed on 2019-10-10.
- [33] Ian Foster and Carl Kesselman. *The Grid : Blueprint for a New Computing Infrastructure*. Morgan Kaufmann publishers, 1998.
- [34] Wolfgang Gentzsch. "Response to Ian Foster's "What is the Grid?" 2002.
- [35] WLCG. *Homepage*. <http://wlcg.web.cern.ch/>. Accessed on 2019-10-10.
- [36] an Bird. "Computing for the Large Hadron Collider". In: *Annual Review of Nuclear and Particle Science* 61 (2011).
- [37] Ian Foster, Carl Kesselman, and Steven Tuecke. "The Anatomy of the Grid: Enabling Scalable Virtual Organizations". In: *International Journal of High Performance Computing Applications* 15 (2001).
- [38] Ian Foster. *What is the Grid? A Three Point Checklist*. Article. 2002.
- [39] Eduardo Huedo, Ruben S. Montero, and Ignacio M. Llorente. "A Framework for Adaptive Execution in Grids". In: *Software: Practice and Experience* 34 (2004).
- [40] WLCG. *Tier Centres*. <http://wlcg-public.web.cern.ch/tier-centres>. Accessed on 2019-10-10.
- [41] M. Gonzalez-Berges. "The High-performance Database Archiver for the LHC experiments". In: *ICALEPCS*. 2007.

- [42] Andrea Valassi. “LCG Conditions Database Project Overview”. In: *International Conference on Computing in High Energy and Nuclear Physics*. 2004.
- [43] G. Dimitrov et al. “Next generation database relational solutions for ATLAS distributed computing”. In: *International Conference on Computing in High Energy and Nuclear Physics*. 2013.
- [44] C. Roderic et al. “The LHC Logging Service: Handling terabytes of on-line data”. In: *ICALEPCS*. 2009.
- [45] L. Canali et al. “Integration of Oracle and Hadoop: Hybrid Databases Affordable at Scale”. In: *International Conference on Computing in High Energy and Nuclear Physics*. 2016.
- [46] Ian Bird et al. *Update of the Computing Models of the WLCG and the LHC Experiments*. Tech. rep. WLCG Collaboration, 2014.
- [47] Pythia. *Homepage*. <http://home.thep.lu.se/Pythia/>. Accessed on 2019-10-10.
- [48] GEANT4. *Homepage*. <http://geant4.web.cern.ch/>. Accessed on 2019-10-10.
- [49] ROOT. *Homepage*. <https://root.cern.ch/>. Accessed on 2019-10-10.
- [50] Edward R. Tufte. *The Visual Display of Quantitative Information*. The Visual Display of Quantitative Information, 1983.
- [51] ROOT. *Higgs Plots*. <https://root.cern.ch/higgs-plots>. Accessed on 2019-10-10.
- [52] NA49 Experiment. *Homepage*. <https://na49info.web.cern.ch/na49info/>. Accessed on 2019-10-10.
- [53] ROOT. *User’s Guide*. <https://root.cern.ch/root/html/doc/guides/users-guide/ROOTUsersGuide.html>.
- [54] ROOT. *TFile Class Reference*. <https://root.cern.ch/doc/master/classTFile.html>. Accessed on 2019-10-10.
- [55] ROOT. *TDirectoryFile Class Reference*. <https://root.cern.ch/doc/master/classTDirectoryFile.html>. Accessed on 2019-10-10.
- [56] ROOT. *TTree Class Reference*. <https://root.cern.ch/doc/master/classTTree.html>. Accessed on 2019-10-10.
- [57] Wes McKinney et al. “Data structures for statistical computing in python”. In: *Python in Science Conference*. 2010.

- [58] NumPy. *Homepage*. <https://numpy.org/>. Accessed on 2019-10-10.
- [59] Pandas. *Homepage*. <https://pandas.pydata.org/>. Accessed on 2019-10-10.
- [60] Matei Zaharia et al. "Spark: Cluster Computing with Working Sets". In: *HotCloud*. 2010.
- [61] Apache Spark. *Homepage*. <https://spark.apache.org/>. Accessed on 2019-10-10.
- [62] Apache Spark. *Cluster Mode Overview*. <https://spark.apache.org/docs/latest/cluster-overview.html>. Accessed on 2019-10-10.
- [63] Danilo Piparo et al. "SWAN: A service for interactive analysis in the cloud". In: *Future Generation Computer Systems* 78 (2016).
- [64] A.J. Peters, E.A. Sindrilaru, and G. Adde. "EOS as the present and future solution for data storage at CERN". In: *International Conference on Computing in High Energy and Nuclear Physics*. 2015.
- [65] Apache Spark. *Running Spark on Kubernetes*. <https://spark.apache.org/docs/latest/running-on-kubernetes.html>. Accessed on 2019-10-10.
- [66] Kubernetes. *Homepage*. <https://kubernetes.io/>. Accessed on 2019-10-10.
- [67] Stefan Wunsch. "Analysis of the di-muon spectrum using data from the CMS detector taken in 2012". 10.7483/OPENDATA.CMS.AAR1.4NZQ. 2019.
- [68] Stefan Wunsch. "DoubleMuParked dataset from 2012 in NanoAOD format reduced on muons". 10.7483/OPENDATA.CMS.LVG5.QT81. 2019.
- [69] CMS Collaboration. "Evidence for the 125 GeV Higgs boson decaying to a pair of τ leptons". In: *Journal of High Energy Physics* 05 (2014).
- [70] Vincenzo Eduardo Padulano. *Repository of the Higgs to two tau leptons analysis translated to run with Spark*. <https://github.com/vepadulano/HiggsTauTauNanoAODOutreachAnalysis>. Accessed on 2019-10-10.
- [71] CMS Collaboration. "Search for anomalous electroweak production of vector boson pairs in association with two jets in proton-proton collisions at 13 TeV". In: *Physics Letters B* 798 (2019).
- [72] Vincenzo Eduardo Padulano. *Repository of the VBS analysis translated to run with Spark with the goal of outputting Numpy arrays from High Energy Physics data*. <https://github.com/vepadulano/mkShapePyRDF>. Accessed on 2019-10-10.

- [73] Valentina Avati et al. “Integrating CMSSW in SWAN”. In: *1st SWAN Workshop*. 2019.
- [74] Javier Cervantes Villanueva. *Parallelization and optimization of a High Energy Physics analysis with ROOT's RDataFrame and Spark*. Master Thesis. Sept. 2018.
- [75] Vincenzo Eduardo Padulano, Enric Tejedor Saavedra, and Javier Cervantes Villanueva. “Web-based interactive data analysis for HEP with Spark and ROOT DataFrame”. In: *Workshop di CCR: La Biodola*. <https://agenda.infn.it/event/17962/timetable/>. June 2019.
- [76] Javier Cervantes Villanueva and Vincenzo Eduardo Padulano. “PyRDF: new features and use cases”. In: *60th ROOT Parallelism, Performance and Programming Model Meeting*. <https://indico.cern.ch/event/831343/>. June 2019.
- [77] Vincenzo Eduardo Padulano. “PyRDF: The Distributed RDataFrame”. In: *SFT Group Meeting*. <https://indico.cern.ch/event/845230/>. Sept. 2019.