Universitat Politècnica de València

Department of Computer Systems and Computation

Doctoral School of Computer Science

Distributed computing solutions for High Energy Physics interactive data analysis

Vincenzo Eduardo Padulano

Director: Prof. Pedro Alonso-Jordá

Co-director: Dr. Enric Tejedor Saavedra

March 2023

Abstract

The scientific research in High Energy Physics (HEP) is characterised by complex computational challenges, which over the decades had to be addressed by researching computing techniques in parallel to the advances in understanding physics. One of the main actors in the field, CERN, hosts both the Large Hadron Collider (LHC) and thousands of researchers yearly who are devoted to collecting and processing the huge amounts of data generated by the particle accelerator. This has historically provided a fertile ground for distributed computing techniques, which led to the creation of the Worldwide LHC Computing Grid (WLCG), a global network providing large computing power for all the experiments revolving around the LHC and the HEP field. Data generated by the LHC so far has already posed challenges for computing and storage. This is only going to increase with future hardware updates of the accelerator, which will bring a scenario that will require large amounts of coordinated resources to run the workflows of HEP analyses. The main strategy for such complex computations is, still to this day, submitting applications to batch queueing systems connected to the grid and wait for the final result to arrive. This has two great disadvantages from the user's perspective: no interactivity and unknown waiting times. In more recent years, other fields of research and industry have developed new techniques to address the task of analysing the ever increasing large amounts of human-generated data (a trend commonly mentioned as "Big Data"). Thus, new programming interfaces and models have arised that most often showcase interactivity as one key feature while also allowing the usage of large computational resources.

In light of the scenario described above, this thesis aims at leveraging cutting-edge industry tools and architectures to speed up analysis workflows in High Energy Physics, while providing a programming interface that enables automatic parallelisation, both on a single machine and on a set of distributed resources. It focuses on modern programming models and on how to make best use of the available hardware resources while providing a seamless user experience. The thesis also proposes a modern distributed computing solution to the HEP data analysis, making use of the established software framework called ROOT and in particular of its data analysis layer implemented with the RDataFrame class. A few key research areas that revolved

around this proposal are explored. From the user's point of view, this is detailed in the form of a new interface to data analysis that is able to run on a laptop or on thousands of computing nodes, with no change in the user application. This development opens the door to exploiting distributed resources via industry standard execution engines that can scale to multiple nodes on HPC or HTC clusters, or even on serverless offerings of commercial clouds. Since data analysis in this field is often I/O bound, a good comprehension of what are the possible caching mechanisms is needed. In this regard, a novel storage system based on object store technology was researched as a target for caching.

In conclusion, the future of data analysis in High Energy Physics presents challenges from various perspectives, from the exploitation of distributed computing and storage resources to the design of ergonomic user interfaces. Software frameworks should aim at efficiency and ease of use, decoupling as much as possible the definition of the physics computations from the implementation details of their execution. This thesis is framed in the collective effort of the HEP community towards these goals, defining problems and possible solutions that can be adopted by future researchers.

Resumen

La investigación científica en Física de Altas Energías (HEP) se caracteriza por desafíos computacionales complejos, que durante décadas tuvieron que ser abordados mediante la investigación de técnicas informáticas en paralelo a los avances en la comprensión de la física. Uno de los principales actores en el campo, el CERN, alberga tanto el Gran Colisionador de Hadrones (LHC) como miles de investigadores cada año que se dedican a recopilar y procesar las enormes cantidades de datos generados por el acelerador de partículas. Históricamente, esto ha proporcionado un terreno fértil para las técnicas de computación distribuida, conduciendo a la creación de Worldwide LHC Computing Grid (WLCG), una red global de gran potencia informática para todos los experimentos LHC y del campo HEP. Los datos generados por el LHC hasta ahora ya han planteado desafíos para la informática y el almacenamiento. Esto solo aumentará con futuras actualizaciones de hardware del acelerador, un escenario que requerirá grandes cantidades de recursos coordinados para ejecutar los análisis HEP. La estrategia principal para cálculos tan complejos es, hasta el día de hoy, enviar solicitudes a sistemas de colas por lotes conectados a la red. Esto tiene dos grandes desventajas para el usuario: falta de interactividad y tiempos de espera desconocidos. En años más recientes, otros campos de la investigación y la industria han desarrollado nuevas técnicas para abordar la tarea de analizar las cantidades cada vez mayores de datos generados por humanos (una tendencia comúnmente mencionada como "Big Data"). Por lo tanto, han surgido nuevas interfaces y modelos de programación que muestran la interactividad como una característica clave y permiten el uso de grandes recursos informáticos.

A la luz del escenario descrito anteriormente, esta tesis tiene como objetivo aprovechar las herramientas y arquitecturas de la industria de vanguardia para acelerar los flujos de trabajo de análisis en HEP, y proporcionar una interfaz de programación que permite la paralelización automática, tanto en una sola máquina como en un conjunto de recursos distribuidos. Se centra en los modelos de programación modernos y en cómo hacer el mejor uso de los recursos de hardware disponibles al tiempo que proporciona una experiencia de usuario perfecta. La tesis también propone una solución informática distribuida moderna para el análisis de datos HEP, haciendo uso del

software llamado ROOT y, en particular, de su capa de análisis de datos llamada RDataFrame. Se exploran algunas áreas clave de investigación en torno a esta propuesta. Desde el punto de vista del usuario, esto se detalla en forma de una nueva interfaz que puede ejecutarse en una computadora portátil o en miles de nodos informáticos, sin cambios en la aplicación del usuario. Este desarrollo abre la puerta a la explotación de recursos distribuidos a través de motores de ejecución estándar de la industria que pueden escalar a múltiples nodos en clústeres HPC o HTC, o incluso en ofertas serverless de nubes comerciales. Dado que el análisis de datos en este campo a menudo está limitado por E/S, se necesita comprender cuáles son los posibles mecanismos de almacenamiento en caché. En este sentido, se investigó un sistema de almacenamiento novedoso basado en la tecnología de almacenamiento de objetos como objetivo para el caché.

En conclusión, el futuro del análisis de datos en HEP presenta desafíos desde varias perspectivas, desde la explotación de recursos informáticos y de almacenamiento distribuidos hasta el diseño de interfaces de usuario ergonómicas. Los marcos de software deben apuntar a la eficiencia y la facilidad de uso, desvinculando la definición de los cálculos físicos de los detalles de implementación de su ejecución. Esta tesis se enmarca en el esfuerzo colectivo de la comunidad HEP hacia estos objetivos, definiendo problemas y posibles soluciones que pueden ser adoptadas por futuros investigadores.

Resum

La investigació científica a Física d'Altes Energies (HEP) es caracteritza per desafiaments computacionals complexos, que durant dècades van haver de ser abordats mitjançant la investigació de tècniques informàtiques en paral·lel als avenços en la comprensió de la física. Un dels principals actors al camp, el CERN, acull tant el Gran Col·lisionador d'Hadrons (LHC) com milers d'investigadors cada any que es dediquen a recopilar i processar les enormes quantitats de dades generades per l'accelerador de partícules. Històricament, això ha proporcionat un terreny fèrtil per a les tècniques de computació distribuïda, conduint a la creació del Worldwide LHC Computing Grid (WLCG), una xarxa global de gran potència informàtica per a tots els experiments LHC i del camp HEP. Les dades generades per l'LHC fins ara ja han plantejat desafiaments per a la informàtica i l'emmagatzematge. Això només augmentarà amb futures actualitzacions de maquinari de l'accelerador, un escenari que requerirà grans quantitats de recursos coordinats per executar les anàlisis HEP. L'estratègia principal per a càlculs tan complexos és, fins avui, enviar sol·licituds a sistemes de cues per lots connectats a la xarxa. Això té dos grans desavantatges per a l'usuari: manca d'interactivitat i temps de espera desconeguts. En anys més recents, altres camps de la recerca i la indústria han desenvolupat noves tècniques per abordar la tasca d'analitzar les quantitats cada vegada més grans de dades generades per humans (una tendència comunament esmentada com a "Big Data"). Per tant, han sorgit noves interfícies i models de programació que mostren la interactivitat com a característica clau i permeten l'ús de grans recursos informàtics.

A la llum de l'escenari descrit anteriorment, aquesta tesi té com a objectiu aprofitar les eines i les arquitectures de la indústria d'avantguarda per accelerar els fluxos de treball d'anàlisi a HEP, i proporcionar una interfície de programació que permet la paral·lelització automàtica, tant en una sola màquina com en un conjunt de recursos distribuïts. Se centra en els models de programació moderns i com fer el millor ús dels recursos de maquinari disponibles alhora que proporciona una experiència d'usuari perfecta. La tesi també proposa una solució informàtica distribuïda moderna per a l'anàlisi de dades HEP, fent ús del programari anomenat ROOT i, en particular, de la seva capa d'anàlisi de dades anomenada RDataFrame. S'exploren algunes àrees clau

de recerca sobre aquesta proposta. Des del punt de vista de l'usuari, això es detalla en forma duna nova interfície que es pot executar en un ordinador portàtil o en milers de nodes informàtics, sense canvis en l'aplicació de l'usuari. Aquest desenvolupament obre la porta a l'explotació de recursos distribuïts a través de motors d'execució estàndard de la indústria que poden escalar a múltiples nodes en clústers HPC o HTC, o fins i tot en ofertes serverless de núvols comercials. Atès que sovint l'anàlisi de dades en aquest camp està limitada per E/S, cal comprendre quins són els possibles mecanismes d'emmagatzematge en memòria cau. En aquest sentit, es va investigar un nou sistema d'emmagatzematge basat en la tecnologia d'emmagatzematge d'objectes com a objectiu per a la memòria cau.

En conclusió, el futur de l'anàlisi de dades a HEP presenta reptes des de diverses perspectives, des de l'explotació de recursos informàtics i d'emmagatzematge distribuïts fins al disseny d'interfícies d'usuari ergonòmiques. Els marcs de programari han d'apuntar a l'eficiència i la facilitat d'ús, desvinculant la definició dels càlculs físics dels detalls d'implementació de la seva execució. Aquesta tesi s'emmarca en l'esforç col·lectiu de la comunitat HEP cap a aquests objectius, definint problemes i possibles solucions que poden ser adoptades per futurs investigadors.

Contents

Contents **viii**

1 Introduction **1**
1.1 Scientific context 1
1.2 Data lifecycle at the Large Hadron Collider 6
1.3 Layout of physics events in an analysis dataset 10
1.4 The workflow of a data analysis application in High Energy Physics . . 12
1.5 Traditional HEP distributed computing and its limitations 15
1.6 Requirements for modern HEP software frameworks 22
1.7 Objectives 26
1.8 Related work 27
1.9 Structure of the document 28

2 Tools **30**
2.1 ROOT 30
2.1.1 I/O 31
2.1.2 Interoperability between Python and C++ 34
2.1.3 RDataFrame 34
2.2 Engines for large-scale data analysis 35
2.2.1 Apache Spark 36
2.2.2 Dask 36

3 Design of a programming model for distributed analysis in HEP **38**
3.1 State of the art 38
3.2 Maintaining the established API 41

3.3 The workflow of a distributed application 44
3.4 A modular implementation 45
3.4.1 Modularity with respect to the execution engine 45
3.4.2 Modularity with respect to the data format 46
3.5 Generalised task creation algorithm for distributed backends 47
3.5.1 Offloading the creation of task ranges to workers for parallelisation 47
3.5.2 Fast task generation on the client side 48
3.5.3 Remote-side conversion of the task 50
3.6 Efficient execution of C++ code in Python processes 52
3.7 Distributable representation of the computation graph 53
3.8 Passing partial results between different processes 54
3.9 Conclusions 56

4 Efficient distribution of physics computations 57
4.1 State of the art 58
4.2 Distributed backend implementation 61
4.2.1 Executing the computation graph with Spark 61
4.2.2 Executing the computation graph with Dask 62
4.2.3 Impact of the two execution engines on end user workflows . . . 64
4.3 Scaling distributed RDataFrame analysis to thousands of cores 66
4.3.1 Hardware setup 68
4.3.2 Methodology 68
Single node test with Dask 68
Tests comparing Dask and Spark backends 69
4.3.3 Results 72
4.3.4 Discussion 75
4.4 Example of full-scale distributed RDataFrame analysis on HEP grid resources 78
4.4.1 New RDataFrame developments 79
4.4.2 Experiments 82
Methodology 83
Hardware setup 84

Results 84
Discussion 87
4.5 Conclusions 88

5 Fine-grained caching of physics data **90**
5.1 State of the art 90
5.2 Tools 94
5.2.1 XRootD 94
5.2.2 TFilePrefetch 95
5.2.3 Intel DAOS 95
5.3 Caching strategies 96
5.3.1 Caching on a file server 96
5.3.2 Caching on the computing nodes 98
5.4 Evaluation of existing technologies for caching during an RDataFrame analysis 99
5.4.1 Methodology 100
5.4.2 Hardware setup 100
5.4.3 Results 100
Single node 101
Distributed cluster 102
5.4.4 Discussion 105
5.5 Exploiting object store for HEP data analysis 106
5.5.1 Exploration of a caching mechanism for RNTuple 107
5.5.2 Integration within the I/O pipeline 108
5.5.3 Considerations for HEP use cases 110
5.5.4 Interaction with DAOS 111
5.5.5 Experiments 112
Methodology 112
Hardware setup 113
Results 115
Discussion 120
5.6 Conclusions 122

6 Serverless computing for HEP data analysis workflows 124
6.1 State of the art . . . 125
6.2 Tools . . . 128
6.2.1 AWS Lambda . . . 128
6.2.2 OSCAR . . . 128
6.2.3 EOS . . . 129
6.3 AWS Lambda functions for distributed RDataFrame . . . 129
6.3.1 Overview of the interaction between RDataFrame and the serverless environment . . . 129
6.3.2 Controlling the invocations via Python threads . . . 132
6.3.3 Kerberos token placement . . . 133
6.3.4 Experiments . . . 133
6.3.5 Methodology . . . 134
6.3.6 Hardware setup . . . 134
6.3.7 Results . . . 134
6.3.8 Discussion . . . 140
6.4 Open source serverless framework for HEP analysis . . . 142
6.4.1 Implementation of the backend . . . 142
RDataFrame backend on the client side . . . 143
OSCAR services defined . . . 144
Reduction strategies . . . 144
Considerations on the implementation of the backend . . . 147
6.4.2 Experiments . . . 148
Methodology . . . 149
Hardware setup . . . 149
Results . . . 150
Discussion . . . 155
6.5 Conclusions . . . 157

7 Conclusions and future work 159
7.1 Publications . . . 162
7.2 Future work . . . 165

List of Figures

1.1 LHC project schedule [1]. 5

1.2 Examples of future computing and storage requirements for LHC experiments. First row: ATLAS experiment CPU consumption (on the left) and disk storage used (on the right) [2]. Second row: CMS experiment cpu consumption (on the left) and disk storage used (on the right) [3]. . 6

1.3 Examples of typical plots resulting from data analysis applications in HEP. (a): Invariant mass distribution observed by the ATLAS experiment with 2012 data [4]. (b): Invariant mass distribution observed by the CMS experiment with 2012 data [5]. 11

1.4 Topology of the WLCG tiers [6]. 18

1.5 CPU consumption across all grid sites for the LHCb experiment in 2018 [7]. 23

1.6 Predicted CPU usage for future ATLAS (on the left) and CMS (on the right) computing models, divided among the different parts of the data pipeline [2, 3]. 24

2.1 Data layout of RNTuple [8]. 32

3.1 Example application with RDataFrame (on the left) and distributed RDataFrame (on the right) 43

3.2 Simplified inheritance tree for the classes responsible to store results of distributed execution and defining how to merge them. 55

4.1 Visualisation of the Dask computation graph generated by calling the `delayed` mapper and reducers in distributed RDataFrame. The colours of the nodes represent their status: nodes that are waiting for results from others are in grey; those that are currently processing a task are in green; those that have completed their task but need to wait for another task before sending their result to the next reducer keep the result in memory and are shown in red; finally, the nodes that have completed their task and do not need to wait for others are shown in blue. 63
4.2 Processing throughput achieved on a single computing node, with increasing number of cores and tasks per core. Each task processes a separate partition of the original dataset. In each plot, three lines indicate the results of increasing the number of cores used in the benchmark. Each line corresponds to a different number of partitions per core. The result of running the original analysis sequentially is also indicated at the 1 core data point of the x axis. (a): Processing throughput expressed in Megabytes per second. (b): Processing throughput normalised by the number of cores. 72
4.3 Benchmark results with Dask or Spark backend on multiple nodes (32 processes per node). Left column: first run of the benchmark. Right column: average of consecutive runs. First row: total runtime of the benchmark. Second row: processing throughput. Third row: speedup relative to the result obtained with one node. 74
4.4 Time to plot (in minutes) achieved with an increasing number of nodes. 32 concurrent processes are run on each node. Data processed in the experiments are read remotely from the storage facilities at CERN. A box plot of the distribution of results is shown for each point on the x axis, ten runs per node count. The box spans from the first to the third quartile. Inside the box, a dashed line represents the mean, a solid line represents the median. “x” marks at 1 and 64 nodes represent outliers, those points that are outside the range of the whiskers. This range is defined as the distance between the first quartile and the third (also called interquartile range) multiplied by 1.5. 76

4.5 CPU usage (in percentage) while executing one mapper task of the distributed RDataFrame analysis. 86
4.6 Memory usage (in Megabytes) while executing one mapper task of the distributed RDataFrame analysis. 86
4.7 Network read throughput (in Megabytes per second) sustained by one computing node that was executing the distributed RDataFrame analysis. The duration on the x axis corresponds to the same runtime of the mapper task of Figures 4.5 and 4.6. 87

5.1 XRootD proxy cache. During user analysis, computing nodes (labeled "Worker" in the image) make read requests for their assigned ranges of entries to the proxy server, which in turn forwards such requests to the remote storage system. The proxy stores the requested entries in its local filesystem and will be able to serve them directly to the nodes during subsequent runs of the application. 98
5.2 TFilePrefetch cache. During user analysis, computing nodes (labeled "Worker" in the image) make read requests for their assigned ranges of entries directly to the remote storage system. On each node, TFilePrefetch intercepts the incoming blocks of entries and stores them on the local filesystem. In subsequent runs, each node will be able to read the same range of entries from the local disk instead of requesting it again from remote. 99
5.3 Single node scenario. Box plots of the distributions of execution times of one thousand test runs in three configurations: Caching disabled, XRootD cache, TFilePrefetch cache. The empty circles represent the median values of the distribution, the whiskers are drawn at $1.5 \cdot IQR$ (interquartile range) and the crosses outside the whisker boundaries represent distribution outliers. 102

5.4 Distributed scenario. Lines represent the execution times along one hundred consecutive runs for three configurations: Caching disabled, XRootD cache, TFilePrefetch cache. The first point of the two configurations with cache enabled correspond to a run where the caches were being populated. 104
5.5 Distributed scenario, with data locality aware scheduler. Lines represent the execution times along one hundred consecutive runs for three configurations: Caching disabled, XRootD cache, TFilePrefetch cache. In the configurations with cache enabled, the caches were already populated in every run. 105
5.6 Overview of the proposed system. The upper box includes the main ROOT components involved in an analysis. On the left of the dashed line (*Analysis layer*) is the user-facing API and the processing engine offered by RDataFrame. On the right is the *I/O layer* that brings compressed physics data from disk to uncompressed information in memory that is sent to RDataFrame for processing. The two orange boxes represent the parts introduced in this work: the introduction of RNTuple as a supported input data format for the distributed RDataFrame layer in the *Analysis layer* and a caching mechanism for RNTuple in the *I/O layer*. . . 107
5.7 Schema of the newly developed caching system in the RNTuple I/O pipeline. Blue horizontal arrows represent the current two steps of the pipeline: reading compressed pages and decompressing them. (a): The analysis is reading from some file-based source and a new RNTuple object is created to write data to a cache. (b): Data are read from the cache during the analysis. 109
5.8 Processing throughput (i.e. reading the dataset and running analysis computations on it) of a distributed RDataFrame analysis on a single node of the DAOS cluster. (a): Real throughput values compared with a linear throughput increase obtained by multiplying the throughput on one core by the number of cores on the x axis. (b): Speedup obtained by scaling the analysis to multiple cores on the node. 117

5.9 Processing throughput (i.e. reading the dataset and running analysis computations on it) of a distributed RDataFrame analysis on multiple nodes of the DAOS cluster (using 16 cores per node). (a): Real throughput values compared with a linear throughput increase obtained by multiplying the throughput on one node by the number of nodes on the x axis. (b): Speedup obtained by scaling the analysis to multiple nodes of the cluster. 118
5.10 Processing throughput (i.e. reading the dataset and running analysis computations on it) of a distributed RDataFrame analysis on multiple nodes of the cluster with the Lustre filesystem (using 16 cores per node). (a): Real throughput values compared with a linear throughput increase obtained by multiplying the throughput on one node by the number of nodes on the x axis. (b): Speedup obtained by scaling the analysis to multiple nodes of the cluster. 119

6.1 Overview of the distributed RDataFrame machinery adapted to run with AWS Lambda services. 130
6.2 Comparison of runtime scaling of the CPU-bound benchmark and the PPS analysis, with an increasing number of Lambda invocations. (a): comparison of the absolute runtimes. (b): comparison of the speedup with respect to a linear increase. 135
6.3 CPU usage (in orange) and network traffic (in blue) in a single Lambda execution running PPS analysis. This execution belongs to a run with 64 concurrent invocations. 136
6.4 Aggregated CPU usage (in percentage) of 64 Lambda invocations. On the y axis of both figures, 100% percent corresponds to full utilisation of a single core. (a): PPS analysis. (b): CPU-bound benchmark. 137
6.5 Comparison of distribution for start and end Times for 128 concurrent Lambda invocations running the cpu-bound benchmark. 138
6.6 Comparison of average CPU usage for every Lambda in both analyses for 512 Lambdas. The left column shows both analyses, while the right shows closeups of both. 139

6.7 Comparison of runtime variability in executions. The top row shows the actual time of a single analysis computation, aligned to the beginning of first Lambda. The bottom row has every Lambda invocation start aligned at 0. The left column shows synthetic CPU benchmark, the right PPS analysis. 141
6.8 Generation of the identifiers for mappers and reducers in the uncoordinated reduction scenario. 145
6.9 Component interaction in the uncoordinated reduction scenario. Numbers denote the order of the steps. 146
6.10 Component interaction in the coordinated reduction scenario. Numbers denote the order of the steps. 148
6.11 Time to plot with an increasing number of cores. 150
6.12 Speedup with an increasing number of cores. 151
6.13 CPU usage of mapper invocations. 152
6.14 Memory usage of mapper invocations. 153
6.15 Distribution of the runtime of mapper invocations for 48, 64 and 80 mappers, respectively from left to right in the image. 153
6.16 Time to plot comparison between using the coordinated reduction and the uncoordinated reduction patterns. 155
6.17 Time to plot comparison between using the Dask backend to run the CPU-bound benchmark on either the HPC cluster at CERN or the machines of the OSCAR cluster. 156

List of Tables

1.1 Event data sizes for different data formats in use by the various experiments in LHC [9]. 10

4.1 Dataset sizes in the proposed experiments. First row: original dataset. Second row: dataset used when testing the Dask backend on a single node. Third row: dataset used when comparing the Dask backend against the Spark backend on many nodes of the cluster. 67
4.2 Runtime of the analysis for the two different approaches. 85

5.1 Statistics for one thousand test runs along three configurations, with the single node setup. 103
5.2 Statistics for one hundred test runs along three configurations, with the distributed setup and a locality-aware scheduler. 104
5.3 Runtime metrics of tests reading an RNTuple dataset, 50 repetitions per configuration. 116

6.1 Variability in the execution time of mappers during a simulated workload run. 152
6.2 Time to plot results for different workload partitioning of the coordinated reduction strategy. 154

List of Listings

1 Simplified folder structure of distributed RDataFrame. The `Backends` folder contains implementations for the execution backends. 46

2 Approximate implementation of the creation of tasks on the client side. . 49

3 Approximate implementation of the conversion of a generic task into an actual task in a distributed worker. 51

4 Usage of the Spark API in the corresponding distributed RDataFrame backend 62

5 Implementation of the MapReduce pattern in the Dask distributed RDataFrame backend. 65

6 Setup function of a Spark benchmark. The analysis receives the created `SparkContext` object to distribute the application on the cluster. 70

7 Setup function of a Dask benchmark. The analysis receives the created `Client` object to distribute the application on the cluster. 71

8 Implementation of the submission of multiple computation graphs concurrently. 80

List of Algorithms

1 Invocation of the Lambda functions on the client side. 131
2 Engine Algorithm on the worker side. 132

Chapter 1

Introduction

Computer Science research can be applied to a wide range of topics and fields. The particular properties of High Energy Physics provide a fertile ground for research in computer science techniques. The next pages will go into detail in describing the context and status of such scientific field, highlighting the requirements to cope with future challenges.

1.1 Scientific context

> "High Energy Physics (HEP) explores what the world is made of and how it works at the smallest and largest scales, seeking new discoveries from the tiniest particles to the outer reaches of space." [10]

This field explores all physics phenomena that involve interactions between particles at the subatomic level. The most established theory that describes such interactions is called the Standard Model [11]. Since its definition in the 1970s, many experiments have been conducted to challenge the theory. The nature of the particles is such that only certain properties can be examined in ordinary conditions. A complex hardware setup is required in order to actually gain evidence about all the events described by the equations in the Standard Model, a task which requires accelerating and colliding particles, detecting their passage and finally analysing the gathered data.

The main goal of physicists in the field is to put this theory to the test, bringing it all to reality or possibly finding its limits. One of the most important research centers in

this context is the European Organization for Nuclear Research (CERN) [12]. It gathers scientists worldwide with expertise in a diverse range of fields with the objective of better understanding the most fundamental building blocks of the universe. In order 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 a huge amount of physical phenomena. In turn, this gives the physicists a means to study the interaction between matter at its most pure state and gain insights about the fundamental laws of nature.

The main technical instrument used to pursue this goal is currently the Large Hadron Collider (LHC), a two-ring superconducting hadron[1] accelerator with a circumference of 27 km of superconducting magnets. It can nominally provide a center of mass energy of 14 TeV[2] for proton-proton collisions, created by 2808 bunches containing $1.15 \cdot 10^{11}$ particles each, with a bunch spacing of 25 ns. 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 LHC [13].

Four experiments were designed and built to completely exploit the physics program for proton collisions. They are located at four interaction points of LHC, in which the actual collisions happen. Two general purpose experiments, ATLAS (A Toroidal Lhc ApparatuS) [14] and CMS (Compact Muon Solenoid) [15], were designed to explore all the possible aspects of LHC physics, from heavy-ions collisions and forward physics, to Higgs boson physics and direct search of new particles. The ALICE (A Large Ion Collider Experiment) experiment [16] is specifically devoted to research in heavy-ions physics, while the LHCb (LHC beauty) experiment [17] aims at maximising LHC potential in beauty and charm physics.

[1] A hadron is a subatomic particle. Examples of hadrons are protons and neutrons.

[2] The 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).

The accelerator alternates active periods, usually called *Runs*, with scheduled *Long Shutdowns* for maintenance and upgrade. During Run 1 (2010-2012), LHC could already deliver several tens 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 [18]. This number has steadily grown over time, surpassing the 200 PB mark in 2017 [19] and reaching a peak of 15.8 PB generated only in the month of November 2018, at the end of Run 2 [20]. Consequently, the volumes of data that are continuously accessed by researchers reach even higher limits. For example, the CERN storage service, EOS, has served 2.5 Exabytes (EB) of data to users only in the year 2020 [21]. This trend will continue in the current Run 3 and future runs as well.

One of the main driving factors behind the creation of LHC was the search for a specific particle, the Higgs boson, named after Peter Higgs who theorised in 1964 that all particles are given their mass thanks to a fundamental field that is present everywhere. In 2012, researchers announced the discovery of the Higgs boson, the last missing piece in the Standard Model. The finding of the Higgs boson was chosen as the "Breakthrough of the Year" by Science journal [22] and became a major milestone in the history of physics. Being the current most powerful accelerator in the world, LHC has the potential to go on and help to shed light on some of the unknown questions of this age: the existence of supersymmetry; the nature of dark matter; the existence of extra dimensions [23]. The impact of this discovery goes beyond the culmination of a long quest, it marked indeed the beginning of a new period in particle physics. Analyses about the behaviour of the Higgs boson have been performed ever since, helping in understanding other known and unknown particle properties.

However, that discovery alone was not the end of the story. Most of the hypotheses that have been explored require an even larger production of Higgs bosons to increase the statistical pools of the experiments. To extend its discovery potential, LHC needs a major upgrade in many areas, that has started since the end of Run 2 and will go on for most of the 2020s. 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 follow. In preparation for Run 3, which started in the first half of 2022, the ALICE experiment underwent

hardware upgrades in several subsystems and the online–offline system for data acquisition and processing. The experiment then registered the first Pb-Pb collision[3] of Run 3 in November of 2022 [24]. The goal set is to reach a nominal collision rate of 50 thousand events per second (KHz), finally producing more than 1 Terabyte (TB) data per second while functional [25]. Also the LHCb experiment upgraded its hardware for Run 3, 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. The goal is to process more than 40 times the number of collisions happening in previous runs (about 10 million collisions every second), thus enabling an increase in the output rate to storage of a factor 10, from 0.5 Gigabytes per second (GB/s) to up to 5 GB/s [26]. ATLAS and CMS have received a partial update during the last shutdown period, with a full hardware upgrade scheduled in the next shutdown [27, 28].

The specifications for Run 3 are not even close to those of the next LHC configuration, the High Luminosity LHC (HL-LHC) [29]. Figure 1.1 shows the timeline of the scheduled runs and shutdowns of LHC, including the upgrades involved between them leading to HL-LHC.

The updated accelerator, scheduled to begin taking data in 2029, is planned to collect only by ATLAS and CMS some 30 times more data than LHC has produced in its whole lifetime until now. As the total amount of LHC data already collected has already passed the Exabyte (EB) limit, it is clear that the problems to be solved require approaches beyond simply scaling current solutions. They reveal instead a strong necessity of collaboration in multiple areas, including those outside physics. Moore's Law [30] cannot be taken for granted anymore - even with a conservative 20% increase in computing power per year due to technological evolution alone - and LHC experiments will need efficient software to handle their computing needs [31].

Experiments at the LHC will face a step change from 2029 onwards, where just increasing resources will not be enough due to budget limitations. For example, Figure 1.2 shows the expected increase in CPU and disk usage respectively for the ATLAS and CMS experiments. In the images, the black curves (solid for ATLAS, dashed for CMS), represent the increase in budget with an interval between 10% and 20% per year.

[3]Collision between two ions of the lead atom, specifically Pb^{82+}.

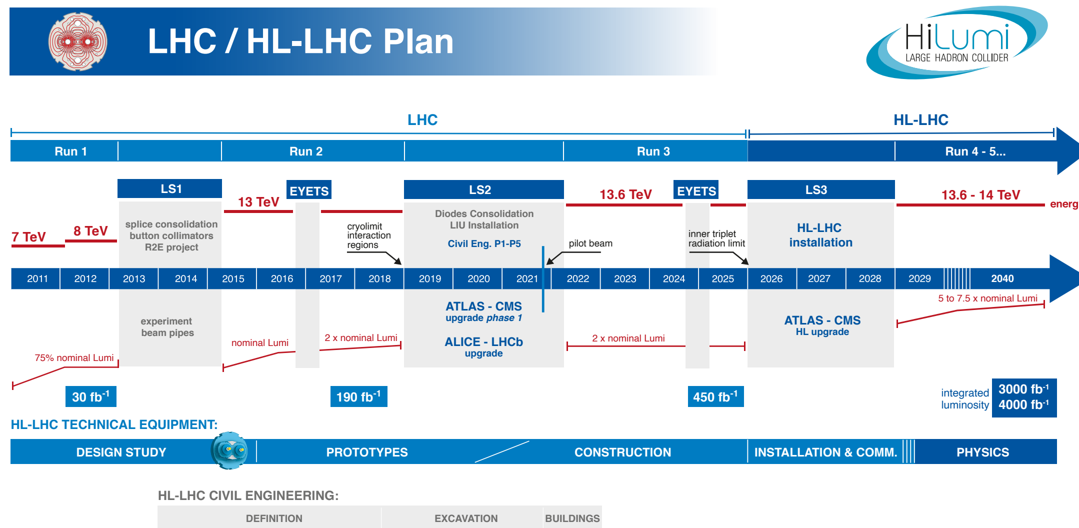


FIGURE 1.1: LHC project schedule [1].

In all cases except for the expected disk usage by CMS, proceeding without R&D improvements would mean a total failure in the physics programme even in the best case scenario of a continuous 20% budget increase per year. Nonetheless, both experiments agree that even the most optimistic perspective of a more aggressive and fruitful R&D would still fall behind schedule in case of restricted resources due to a lower budget. Evidently, this statement applies to both computing and storage requirements, with quite similar linearly increasing trends overall. Consequently, this thesis will touch upon both subjects, describing some of the current pitfalls and highlighting potential paths to a more sustainable future. Data collection, processing and storage will depend on affordable software and computing, so the physics reach during the HL-LHC era and beyond 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, a vast majority of the software currently in use was written with one single architecture in mind and following a sequential processing model. Consequently, squeezing the

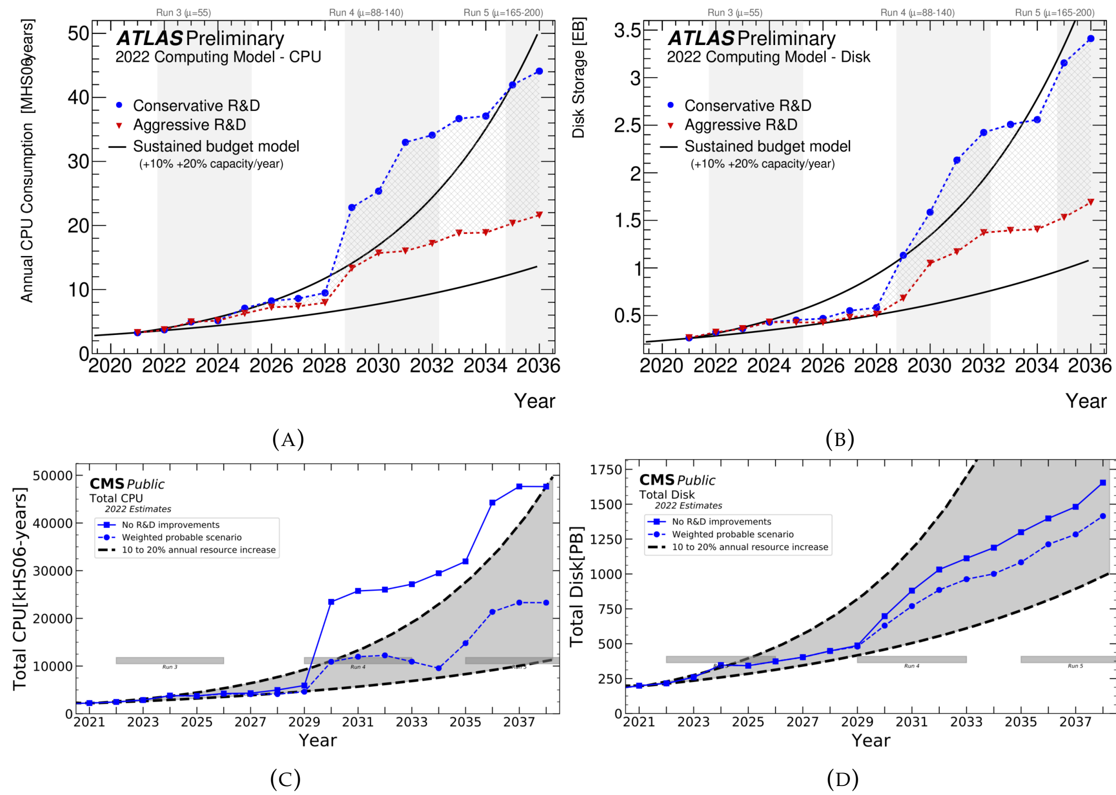


FIGURE 1.2: Examples of future computing and storage requirements for LHC experiments. First row: ATLAS experiment CPU consumption (on the left) and disk storage used (on the right) [2]. Second row: CMS experiment cpu consumption (on the left) and disk storage used (on the right) [3].

most out of computing resources often becomes an arduous task. Similarly, I/O implementations used to have sequential access to disk as their prime target, while concepts like asynchrony and concurrent transactions became promiment more recently.

1.2 Data lifecycle at the Large Hadron Collider

From the physics events that occur in the LHC when two bunches of protons collide to the final insights that researchers gather from those events, all happens through the

exchange of information between different steps of a more general data lifecycle. At the LHC, this usually involves: particle collisions with consequent collection of information by the detectors, storage of the relevant events in the CERN data center, reconstruction of the full event information from the raw detector data, reduction of the dataset sizes by changing to smaller and smaller data formats, final processing of the reconstructed events, visualisation and interpretation. In parallel, CERN experiments also run campaigns of generation of simulated events according to physics laws, so that they can be compared against the real data obtained via the accelerator.

As it was previously mentioned, physics events occur in LHC when two bunches of protons collide in correspondence of one of the four detectors. The collisions happen once every 25 nanoseconds, which translates to 40 million times per second (MHz), only considering single-pair collision (also called elastic collision). Considering that protons in the same bunch crossing collide multiple times (a phenomenon also called "pileup") the total number of collisions per second is actually higher. In the first active periods of the LHC pileup was around 15 (six hundred million collisions per second), whereas during Run 2 it was around 25 on average (one billion collisions per second). One parameter that is useful in characterising 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-section[4] (σ). The integral of the luminosity over time is often referred to as an important parameter, since the higher its value the more data available for studying [32]. The design luminosity of LHC was first reached in June 2016 [33]. By 2017 twice this value was achieved [34] and it was confirmed by the start of Run 3 [35]. The planned luminosity for HL-LHC will increase the current value by a factor 10, reaching $10^{35}\text{cm}^{-2}\text{s}^{-1}$ [36]. 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 around one Petabyte (PB) per second of raw data is produced at any given experiment while active:

$$1 \cdot 10^{9} \text{ collisions/s} \cdot 1 \text{ MB} = 1 \cdot 10^{15} \text{ bytes/s} = 1 \text{ PB/s}. \tag{1.1}$$

[4]Probability that two particles will collide in a certain way decaying into a certain product.

Clearly, storing information from all the events would represent a huge technical challenge. Furthermore, most of the events are not interesting in terms of the physics that they represent and are considered background noise. Thus, LHC experiments implement trigger systems for the purpose of filtering out events which do not have to be stored. The selectivity of the triggers, that is the rate at which events are kept, is usually very low, leading to keep only a small fraction of the generated events. For example, the order of magnitude of event production is in the millions of events per second, while the events that are usually kept are in the order of tens per second. It should be noted that the previously discussed figures regarding the data delivered by LHC experiments to the storage centers (O(100) PB/year) were already taking into account the trigger systems, without which those same metrics would be in the orders of Zettabytes (ZB) per year.

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 fifteen thousand servers and two-hundred-thirty thousand processor cores. At this stage, the information that arrives is still in its raw state, made of digits representing the signals coming from the detector hardware. Various algorithms of event reconstruction are applied to datasets in order to get them to a state that can be practically analysed later on. First, the trajectories of the particles involved in the collision must be computed. Later on, particles are identified based on some relevant dimensions such as their energy and the kinematics of the collision. These steps require external knowledge such as calibration constants for the detector or the magnetic field in place. A copy of the reconstructed data is archived to long-term tape storage, using the data management system called CERN Tape Archive [37].

The reconstructed events represent the real, gold-standard datasets from which the final insights and possibly new physics discoveries will be gathered by researchers down the line. But in order to reach that goal, reality has to be compared against theory. In High Energy Physics, the process of simulation creates dataset with volumes at least the same order of magnitude as real data, representing the physics phenomena described in the Standard Model. In practice this means running very long simulation campaigns (months at a time) with distributed programs that apply Monte Carlo simulations based on the physics equations until a big enough statistical sample has been

generated. Each experiment runs separate Monte Carlo simulations, which usually differ not only by the physics involved but also by the software algorithms used. In recent years, machine learning concepts have been extensively explored in this area, to speed up the simulations while maintaining a high degree of accuracy.

Up to this point, no actual data analysis has been run yet. Data managed centrally by the experiments is a non-interactive effort involving large parts of the community to produce curated datasets that can later be processed. Furthermore, the amount of steps involved in this centralised pipelines varies depending on the LHC experiment. Centrally-managed datasets are then queried and processed by individuals, interactively, leaving place to end-user analysis workflows. Data storage for end-user HEP analysis is thus a write-once-read-many scenario. Depending on the state in the reconstruction chain, different data models are in place among the experiments:

- 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 schemes of their data. Table 1.1 summarises the data size of each data format that the different experiments use in their workflows.

This thesis targets in particular requirements and improvements related to the final analysis stage (i.e. related to the data formats of the last two rows in Table 1.1). Over the years and with the different runs of LHC, experiments have introduced new data format specifications targeting lower disk size and faster I/O transactions. For example, CMS defined the MINIAOD targeting Run 2 as a successor of the AOD model used in Run 1. MINIAOD event size is around 40 KB, a factor 10 reduction with respect to the previous AOD format [38]. More recently, a new format called NANOAOD has been created, with an event size of around 1-2 KB, roughly twenty times smaller than before [39]. The ATLAS collaboration on its part also used an AOD model for Run

TABLE 1.1: Event data sizes for different data formats in use by the various experiments in LHC [9].

Data Format [KB/event]	ALICE	ATLAS	CMS	LHCb
RAW	1050 (p–p) 1625 (p--Pb) 7500 (Pb--Pb)	1000	500	60
ESD	15–30% RAW	2700	1000	110
AOD	10–15% RAW	600	400-500	120
Final analysis	AOD	*variable*	*variable*	10

1, with derived AOD (DAOD) formats being defined on an analysis group basis with somewhat smaller sizes but no formal definition. A similar approach was followed also for Run 2, with some DAOD files counting as low as 40 KB per event and others reaching around 450 KB per event. Two new formats have been defined for the latest run, in order to standardise the schema used by the various groups in the collaboration. These are called DAOS_PHYS (with a target of 50 KB/event) and DAOS_PHYSLITE (with a target of 10 KB/event) and they are centrally produced and managed by the experiment [40].

Data in this final stage can be processed and analysed by physicists. This is indeed the first point of contact between the final users and the information they want to process. At the highest level possible, this happens by selecting the interesting physics events, deriving some new quantities from them and plotting the results, mainly through histograms or closely related graphics as shown for example in Figure 1.3. Since this part of the lifecycle is the main focus of this thesis, it will be further explored and detailed in the following sections.

1.3 Layout of physics events in an analysis dataset

Due to the particular nature of the phenomena that happen in the LHC, particle collision information is stored in a quite peculiar way on disk, thus leading to dataset schemas that are not easily seen in other fields. First, it must be noted that events in the collider are statistically independent, that is processing any two different events can

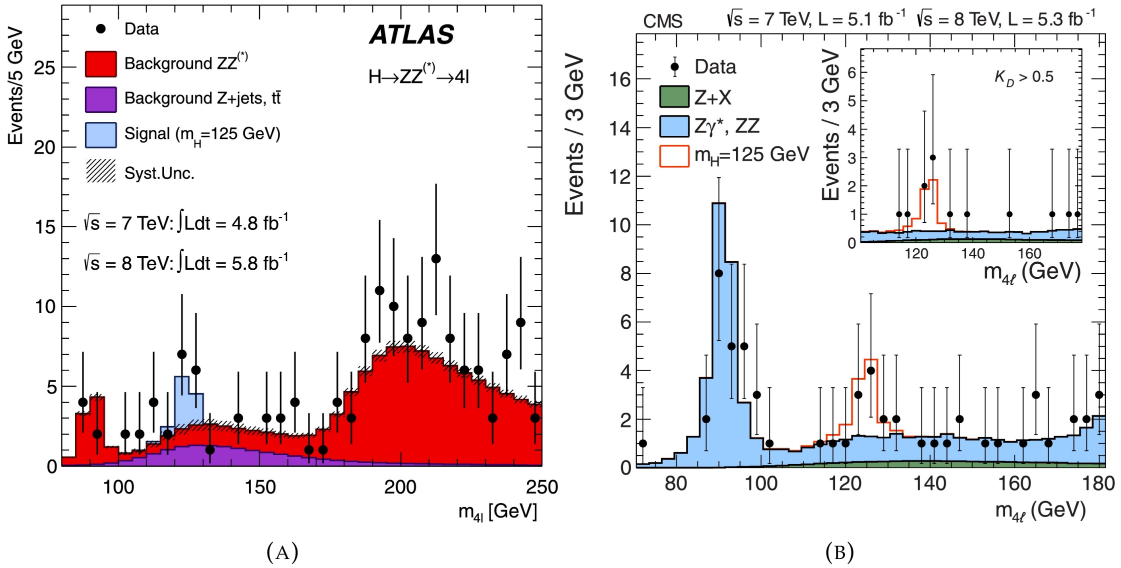


FIGURE 1.3: Examples of typical plots resulting from data analysis applications in HEP. (a): Invariant mass distribution observed by the ATLAS experiment with 2012 data [4]. (b): Invariant mass distribution observed by the CMS experiment with 2012 data [5].

happen in separate calls to the same function (or even in separate processes that apply the same function to the different events). The results of the different function calls will not depend on each other. Thus, by nature, data processing in High Energy Physics can be approached as an embarassingly parallel problem [41].

The information contained in a single event, which refers to the physics quantities of the particles resulting from that collision, can vary from simple scalar or floating point values to complex, arbitrarily-nested data structures. Furthermore, some quantities are inherently representable with collections of values, for example the collection of all the energy values of the particles involved in a certain collision. Two different events usually involve a different number of particles. If a dataset can be described by the events on the rows and the different quantities on the columns, it follows that the same column may have different number of elements in the collections that belong to different rows. In this sense, a column can be represented by a particular type of n-dimensional array, where only one dimension has a fixed length (corresponding to the number of

events in the dataset) and all other dimensions have a variable length. More generally, this class of arrays is known as ragged or jagged arrays [42].

As for the types of the values contained in different columns, this can vary a lot from experiment to experiment, analysis to analysis. It has been previously mentioned that in the past it was common to see large size per event in HEP formats. AOD formats were usually characterised not only by more information stored per each event, but also by large data structures used to contain it. With more recent and slimmer formats, the types employed in dataset columns are converging towards simpler structures, often just scalars and arrays thereof.

It has been established that physics datasets, once they pass the reconstruction stage, are not modified. They represent the reality of the processes that happened in the collider, that is why they are written once and then read many times by researchers afterwards. This implies a key consequence: it means that the information about many different particles involved in collisions is stored in one single dataset. Thus, it is quite common to see thousands of columns in the same dataset. It is the case most often that not all columns are actually needed in an analysis application. Any two different physics research groups may be interested in exploring the characteristics and phenomena related to some specific particles, or a specific subset of their dimensions. Consequently, it is desirable for a data format targeted at the HEP use case to be inherently columnar, so that I/O transactions are minimised to only the desired information.

1.4 The workflow of a data analysis application in High Energy Physics

A physics analysis is the main task for many research groups at CERN and partner institutions worldwide. It represents the actual interpretation of the physics objects that were created by the event reconstruction step, together with the comparison against those simulated according to the theoretical. Purely from the perspective of a computer scientist, this step entails the production of programs that will receive the reconstructed data as input and will output plots that help the physicists to reason about what happened in the accelerator (eventually providing them with good material for

publications). But for many researchers in the field data analysis includes a large series of considerations that must be done regarding physics laws and related statistical measurements. It so happens that a typical data analysis workflow is comprised of multiple steps, which can be all included in the same application or only some of them may be selected. The following list should provide most of the steps that a typical analysis goes through:

1. The first step is always defining which dataset should be processed. The dataset is defined usually by a list of files and some attached metadata. Notably, the same application may need to process data coming from different sources, for example real data versus Monte Carlo generated events. Metadata may then be used in code to configure different parameters or distinguish different operations that should be applied to different data. The parts of the dataset that share the same metadata or follow the same configuration parameters are sometimes also called "samples". The input dataset size can vary a lot, but a full-scale analysis typically processes at least a few Terabytes of data.

2. In the first stage of the analysis the source data is filtered in order to retrieve the interesting features, e.g. selecting only certain physics dimensions such as energy, angle or establishing thresholds such as particles with momentum higher than a certain value. This category of operations is mostly known as "cutting" and the filters themselves are also called "cuts".

3. Once the right physics events have been selected, an analyst usually designs the operations that create new quantities, also called "variables" or "observables". This means combining the remaining particles through functions that fall in a wide range of complexity and sometimes become computationally heavy. Historically, these functions share a common logic pattern which processes particles proceeding iteratively event by event, following their statistically independent nature. More recently, some analyses have begun to follow an approach which processes batches of events at a time.

4. Sometimes the defined quantities have to be calibrated according to detector specification. Furthermore, depending on the sample one or more observables may need to be reweighted before proceeding.

5. At the late stages of the analysis it is necessary to deal with the systematic uncertainties connected to the observables. These are those type of errors that lead to measurements which stray away from their true expected values in a precise way and little variability. Although these characterise many physical measurements, also in our daily lives, they pose a particular challenging issue in the context of HEP analysis. The difficulty comes mainly from two factors. First, that their origin is not always clear. Second, that there is no set of predefined equations that establishes how to deal with them. Thus, they require a mixture of knowledge and common sense to be properly addressed [43]. As for the analysis implementation, this usually means that for any observable, numerous systematic variations may be present, thus creating a multiplying factor in the computations needed.

6. In recent years, various groups have begun exploring various Machine Learning techniques to improve the efficiency or precision in defining some quantities. Not all analyses include this, but seeing some functions which leverage pre-trained models to infer variables from what is available in a dataset has become more common. Existing use cases include classification and signal-background selection [44].

7. Once all the relevant physics quantities are defined, they can be aggregated in useful statistics. Most analyses share a common set of summaries and plots, usually histograms and closely related graphics, which are also often stacked or drawn together. This is mainly due to the need for visualising and comparing the distributions of the physics events in the analysis, as done in the examples given in Figure 1.3.

8. The distributions seen in the results of the analysis have to be interpreted and this is usually done through statistical inference in order to get a better idea about the phenomena that generated a signal in the detectors. It is natural for physicists to model these distributions through Probability Density Functions (PDFs), which

describe the probability that a theory is correct given the output of the analysis. These functions are used to fit data according to a likelihood, that is the probability of obtaining a result x given a certain model parameter θ [45].

The analysis landscape in this field is characterised by a plethora of different software tools, with some generic patterns as described above but sometimes very different implementations. Each LHC experiment controls and manages a separate software stack that includes analysis frameworks tailored to the specific program of the experiment collaboration. Further down the line, any university group or even any single researcher may write custom code for their analysis need. This aspect will be further discussed in following chapters of this thesis.

1.5 Traditional HEP distributed computing and its limitations

The context described in the previous section clearly poses some serious computing and storage challenges that no single machine can tackle. It has already been discussed how experiments at the LHC rely on large storage facilities where data from the detectors is sent and further organised in various data formats. Consequently, these large amounts of data are processed following the lifecycle described earlier and this involves the coordination of the work of multiple different computing nodes. In general, this area of computer science is called distributed computing and involves both the hardware and software that contribute to manage and process very large datasets over a set of machines. There are different approaches to the distribution of a workload, the one that has been most extensively used and explored in HEP is the Grid [46]. 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 Grid middleware was developed and deployed to support such system. 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, a computing grid was defined by Ian Foster and Carl Kesselman as:

> "A hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities." [46]

A more recent definition by 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." [47]

Therefore, the Grid combines distributed computational and storage resources to serve an interface to final users through a middleware that can be used to solve various scientific problems. The benefit is faster, more efficient processing of different tasks in large computing centres while providing access to users sitting at their desks in their home institutions.

The first definition of the Grid architecture was presented in the article "The Anatomy of the Grid: Enabling Scalable Virtual Organizations". the Grid architecture identifies fundamental system components, specifies the purpose and function of these components, indicates how these components interact with one another [48]. It defines standard protocols and APIs to help with the creation of cooperative grid systems and portable applications. It is defined in terms of layers, where each layer has a dedicated scope.

At present, the coordinated efforts of HEP facilities around the world to have a coordinated computing system fall under the name of Worldwide LHC Computing Grid (WLCG) [49]. It is a large distributed computing infrastructure - more than 400,000 CPU cores, 300 PB of disk and more than 200 PB of tape - devoted to store, deliver and analyse data generated by the LHC, making them available to all partners, regardless

of their physical location. 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 computing 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 [50].

The WLCG features different layers, starting from the lower-level hardware and network provisioning, going through the definition and cooperation of the middleware, then leading to the user-facing applications and services. Hardware and networking can vary from facility to facility, thus the need for a layer in between the final users and the administrators of the local resources at a grid site. That is exactly the role of the middleware, consisting of various software components providing protocols and standard APIs to enable the communication between the hardware at the site and the rest of the Grid. This layer makes use of virtualisation to mask the underlying heterogeneous (in terms of different CPUs and network configurations) resources. 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 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 [48, 51, 52]. In WLCG, each LHC 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, they are usually classified by an increasing number from zero to three, each label defined by some common characteristics. For example, the Grid Tier-0 is at CERN, where raw data from the accelerator is sent to the data centre and either archived to tape or sent to other tiers to create replicas. Tier-1 centres are still quite large in terms

of resources available, usually they run the first processing steps of the data lifecycle and host the datasets for the experiment collaborations. Tier-2 and Tier-3 centres host, respectively, less resources, the first still being considered as part of the LHC collaboration, the latter being based on opportunistic resources available at the local institution with no formal commitment.

Figure 1.4 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 minimise chances of data loss. This is also done to reduce latency of analyses for scientists local to the corresponding research facilities who do not need to transfer data across long distances.

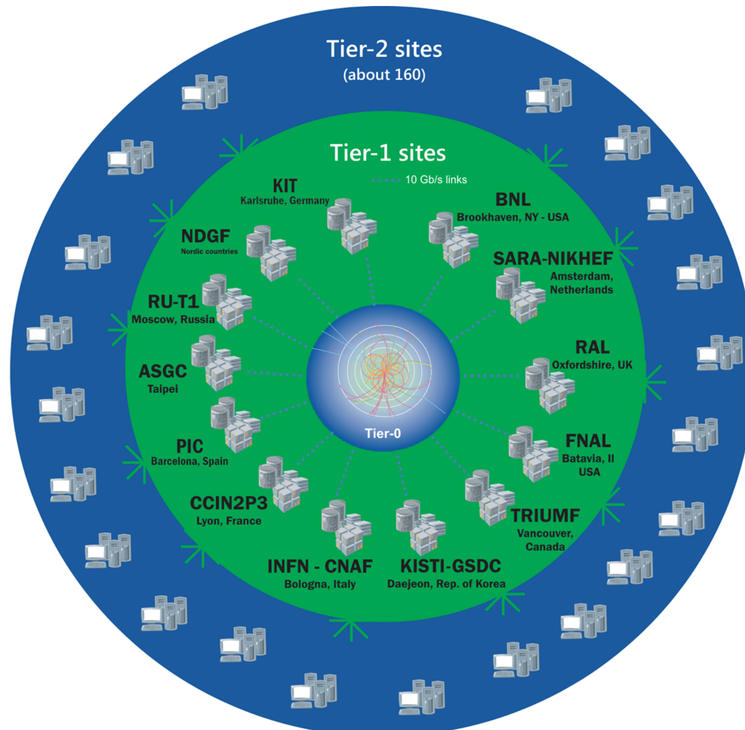


FIGURE 1.4: Topology of the WLCG tiers [6].

One of the main goals of employing a distributed computing system, including the

particular case of the Grid, is that final users should not be bothered by the deployment of the services, the orchestration of the resources or the splitting of the workload. Ideally, users would be presented with an interface which, in the most simple way, allows them to request resources and use them as if they were running an application on their own computer. This interface can actually be split according to at least different aspects. One is how users can connect to the Grid resources, e.g. with a web-based GUI or with SSH-based access from a terminal. Another is the software stack that is available to users once connected, whether it already contains all libraries that users may need, whether it can be customised. Lastly, how the resources can be actually used: what is the programming environment (e.g. GUI or code), what is the workflow to send computations to the remote resources et cetera.

Users may connect to the Grid in different ways and will land in different environments depending on the particular grid implementation. It is often the case that they will land on a node that is not responsible for running large scale applications, but is just a gateway to other resources. The interface provided may be graphical, in which case the connection will be web-based and the user will work within a web browser, or just a shell starting in the user home directory. Examples of graphical computing environments for the Grid date back to the early 2000s, showing concepts like the use of online notebooks that have then developed and gained popularity in the last few years [53, 54]. All physicists at CERN can use a common login platform to get access to remote resources, including the Grid. This service is called LXPLUS and is shell-based. Users receive a home directory with some disk quota and can submit applications to the CERN batch system resources from a terminal.

Once connected to a portal and armed with a working software environment, a HEP analyst can proceed with programming their application and launching it on one or more computing nodes. The traditional workflow for such a distributed scenario is as follows:

1. The main logic of the physics analysis is written in a program. This may include reading the dataset, selecting the interesting events, defining new quantities featuring various degrees of complexity. Sometimes, different parts of the analysis are split in different programs and run one after another, saving the intermediate

filtered datasets in between to run consequent steps faster.

2. The defined program represents a task, a kernel of operations that receive an input and produce an output (either some relevant plots or intermediary data). This needs to be distributed to the computing nodes. Grid computing inherently follows the approach of batch systems, large software libraries that take care of the scheduling of resources, the distribution of a workload and its execution in multiple nodes. Quite a few such systems exist, among which Slurm [55], MOAB [56], HTCondor [57] or PBS [58]. In HEP, HTCondor is the most common resource manager and batch queueing system found on distributed and HPC resources. The user needs to take care of writing a job submission file, describing the program they would like to run, requesting a certain number of resources and how many parallel jobs should execute the program. Generally, each job will receive as input a different part of the dataset that can be processed independently from others. Also this decision is in the hands of the user.

3. When the tasks finish running on the remote nodes, they will produce some output, each job its own. This poses the question of how the output should look like from each process, which usually is addressed by writing to disk the partial results of any task. Consequently, these need to be further processed so that they can be merged together into the final result of the analysis. This step needs intervention by the user, who needs to create another program to aggregate the partial results. Said program can then be run either on the local user session or launched again on remote resources. It must be noted that this whole part of the workflow supposes that all tasks produced a result correctly, which is often not the case. If any task fails, the user will need to keep track of them and perform retries accordingly until all results are gathered.

Reading the above list can give the idea that the usual workflow of a HEP analysis loosely follows the approach defined by the MapReduce paradigm [59]: the total workload is divided in smaller pieces which are processed in parallel by different executors and output partial results that have to be merged in order to obtain the final one. The traditional approach described above was employed before the definition of MapReduce and differs from it in some aspects.

In fact, there are some limitations related to this approach which should not exist in a modern computing framework. All of them revolve around one key problem: the user is responsible for too many parts of their workflow. This can be highlighted in various places:

- The MapReduce paradigm supposes that the execution framework takes care of aggregating the partial results of different tasks (at most giving the user the possibility to define a custom function that will do the reduction). In the traditional batch computing workflow the merging step is a responsibility of the user.
- Splitting the input dataset is not a trivial task. For example, a splitting strategy should be decided upfront: per-file, per-column, per-range of entries. The execution framework should support different reading granularities. The user then needs to perform the split and find a way to give to each job a subset of the whole dataset. They then need to think about how to guarantee that exactly all the required entries of the input dataset are read and processed, no less and no more.
- Task failures have to be handled by reading the log files from the jobs that failed, understanding what was the issue, fixing it and resubmitting that particular job.

All of the above notwithstanding, it is also important to note that MapReduce was designed with different use cases in mind than the ones specific to HEP, for example processing data of customer purchases and aggregating by product. One complexity of the MapReduce design, at least initially, was creating a shuffle mechanism to collect same-valued keys coming from different mappers to the same computer before being able to run the reduction step. Traditional HEP analysis as described above didn't suffer from this particular problem, but still presented the challenge of dealing with arbitrarily deeply nested data structures which are not addressed by MapReduce.

In the last few years there have been efforts to improve the model just described, both within HEP and in other fields, and also this thesis is framed in that same line of research. The core proposal, which will be better described in Section 1.7, is to introduce modern and ergonomic interfaces that remove as much burden as possible from the shoulders of the final users, letting them focus on their research activities.

1.6 Requirements for modern HEP software frameworks

The previous sections highlight the context that characterises the daily activities of analysts in this field. Section 1.2 shows that data analysis is only the final step in a long pipeline, since the information generated by the accelerator needs to be duly processed before reaching a reasonable state that can be actually useful for researchers. As a matter of fact, the presence of all the steps in the data lifecycle can also be observed by the fraction of the total CPU time spent on the Grid taken by each step. LHC experiments provide figures regarding the distribution of computing resource usage, as can be seen in Figure 1.6. The plot in Figure 1.5 shows historical data (from 2018, during Run 2) about the CPU usage across all grid sites for the LHCb experiment. Clearly, the highest portion of the time was spent in the Monte Carlo simulation campaigns, consistently during the whole year. Event reconstruction and related data processing comes in second place when aggregated together, followed by the "user" label representing the data analysis portion of the CPU usage accounting for roughly 5% of time spent. Estimates for future computing requirements given by the ATLAS and CMS experiments respectively in Figure 1.6a and Figure 1.6b show similar distributions. For CMS, generation and reconstruction of true events accounts for slightly more time spent than simulation, the opposite for ATLAS. All charts agree that user data analysis represents only a small fraction (around 5%-10%) of total resource usage on grid sites. And this remains true also in the two images showing future estimates with conservative or no R&D at all.

Given these insights, one could reason about whether the final analysis use case deserves much attention, and whether any effort to optimise it should be made in the first place. But the resource usage is not the only metric that should be taken into account. On the one hand, the parts of the data lifecycle that take up the most resources, i.e. simulation and reconstruction, are usually implemented in terms of large applications that perform all the algorithms needed and are then organised in long campaigns that run throughout the year, a few times over the course of an LHC run. Thus, whenever a collaboration decides to launch one of these campaigns, the computing resources will be extremely loaded, but the people that contributed to writing those applications will be relatively less stressed while they run. Most of the work that needs human intervention happens before the start of a campaign, while during the campaign only a few

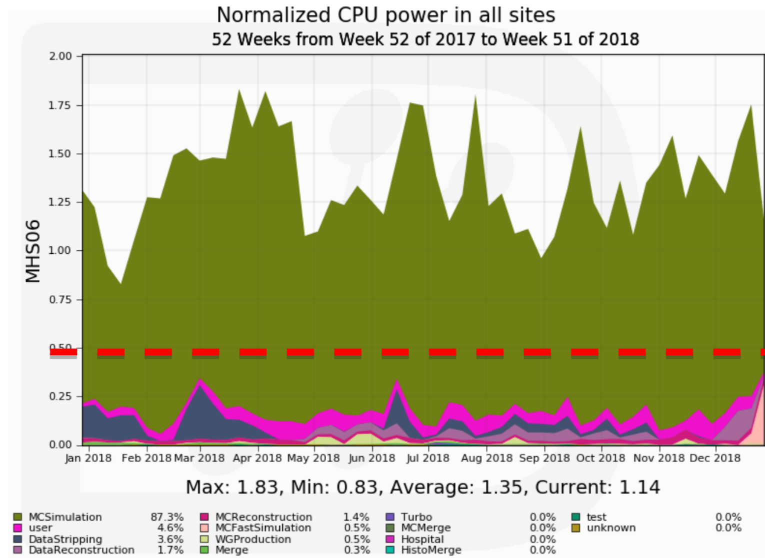


FIGURE 1.5: CPU consumption across all grid sites for the LHCb experiment in 2018 [7].

maintainers need to check the status of the jobs and deal with the coordination of the output files. On the other hand, the general category of data analysis in reality involves a plethora of different applications, research groups and software tools. Within any particular group that focuses on analysing a specific part of the phase space, an application may be run daily, multiple times, with slightly different configurations to check the corresponding behaviour of the particles. Often times the people in the group will discuss the results obtained, comparing their applications to better understand what are the aspects that brought to a particular insight. This approach then broadens to different groups which may meet to compare the tools they use to run their analyses or the decisions that led to a particular configuration. In practice, this last part of the data lifecycle actually involves the highest amount of time for the highest amount of people. It is usually the main occupation of students at CERN and partner institutions, who will spend a few years of their lives focusing on how to get the insights their group needs for the latest publication. Thus, it becomes clear that optimising the data analysis use case for HEP would have a very large impact in terms of personpower. For any speedup factor obtained by running an application with faster code, the equivalent

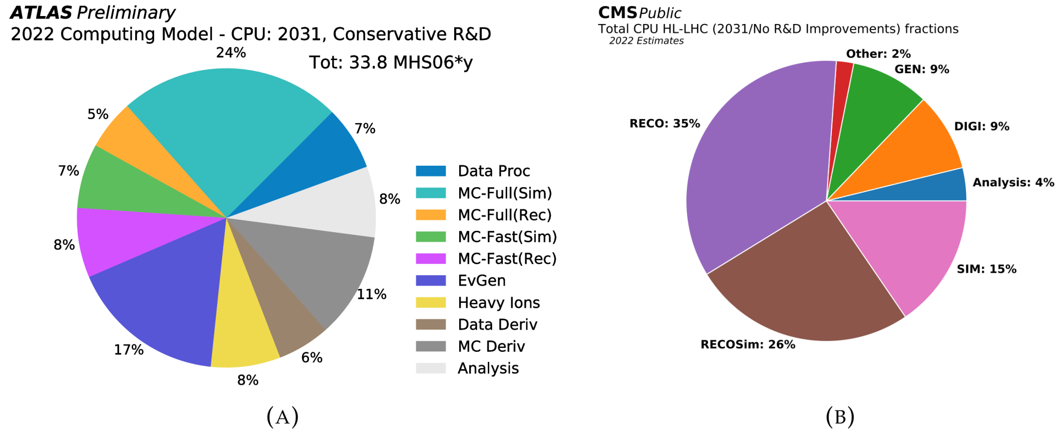


FIGURE 1.6: Predicted CPU usage for future ATLAS (on the left) and CMS (on the right) computing models, divided among the different parts of the data pipeline [2, 3].

amount of time would be spared for the user while at the same time the same resources could be used for some other application. Furthermore, if better, more modern and ergonomic interfaces become widely available, researchers would spend comparably less time writing the application code itself, gaining more time to focus on the insights. In practical terms, improving the current data analysis scenario would lead at the very minimum to a better experience for the physics community. But removing inefficiencies, thus having more time, may potentially also lead to enabling new possibilities for exploring other analysis within a certain group which previously were not considered due to lack of time.

Obtaining such improvements requires careful considerations regarding the HEP analysis software scenario. As mentioned in Section 1.4, there are quite a few frameworks which physicists rely upon for this task, some centrally managed by experiments and others closer tied to the activities of the analysis group. Many of them in their turn are based on a few key libraries, which serve as building blocks for higher-level functions and data structures more specialised for the specific workflow of the analysis framework. The work of this thesis is based upon and improves one such core

library, called ROOT [60], which will be further described later on. It should be the responsibility of this lower-level tools to address the problems and challenges of analysts head-on, so that all the rest of the toolchain at higher levels may benefit. This purpose can be spread across a few general lines of work: performance, user experience and collaboration with other parts of the data pipeline.

Performance is a very generic yet relevant topic for any kind of Computer Science application. For the HEP analysis use case, the main metric to optimise for is the so-called "time to plot", that is the time between the launch of an application by the user and the final visualisations materialising on their screen. On a second level, memory usage should also be kept under control, especially to avoid unpleasant situations with memory issues leading to unrecoverable crashes which are hard to debug on the part of the user. In this regard, it is worth mentioning that a design decision was made for Grid sites, which are required to have at least 2 GiB of memory per physical core. Physics analyses involve a high number of I/O operations, so making sure the software can read data from disk as fast as possible must be a focus. Not only that, but also getting the best out of a network connection is crucial since most often the datasets are stored in remote facilities and have to be read over the network. In the best case scenario, the computing facility running the analysis is also enabled for caching the data arriving to the nodes, so the analysis software must be able to coordinate with existing caches or even be able to serve as a caching system itself. The application runtime then involves a usually large amount of complex operations, which makes it also computationally intensive. In this regard, the analysis software should aim for the highest throughput (in terms of physics events analysed per second) per core. Historically, physics software was written based on single-core, sequential paradigm. Nowadays, multi-core computers are the norm and parallelisation should be part of the design of the software library, on a single machine and on a cluster alike. When dealing with a computing cluster, the analysis tool should be able to scale in terms of cores/nodes used, input dataset size and complexity of the analysis operations. Ideally, since the events are independent, the core logic of the execution of the analysis should scale linearly with the increase of the resources available.

There are many aspects related to user experience. The analysis programming model and its interface will be highlighted for the purposes of this document. It can

be argued that the main objective should be simplifying as much as possible the application development process for analysts, who should need to think primarily about the physics involved and much less about the programming. For example, the traditional programming model used by physicists employed iterative procedures to analyse the events one at a time. This was reflected also in code, characterised by for-loops over entries and bookkeeping the defined quantities and their aggregations by hand. Modern practices brought a shift towards a declarative model, where simple, composable APIs can be used to hide the underlying event loop while letting users focus on the "what" of their analysis and not on the "how". Once an API has been decided, it should be carefully preserved and grown in a sustainable way, decreasing the steepness of the learning curve for users and avoiding incompatible changes where possible. This approach has to go hand in hand with the necessity to perform well on many cores and many nodes discussed earlier. Thus, it is crucial to hide also the complexities of resource scheduling and workload distribution from the users, enabling them to use the same analysis code on their laptop as on any computing cluster.

The analysis stage is the last part of a long pipeline, and is itself divided further into many pieces as discussed. The analysis software should feel naturally encased in the pipeline. It should integrate with the I/O layer and make best use of it to gather data as fast as possible. Also, it should carefully restrict the data read to only those actually processed later in user code. Furthermore, it should be easy to integrate the analysis tool onto various facilities that the experiments may want to use, be it grid sites or other deployments. In this scenario, it may need to interact with other tools, such as data management systems, statistical or machine learning inference software.

1.7 Objectives

The main goal set for this thesis was to develop a practical solution to the problems and limitations of traditional distributed HEP data analysis previously described. Leveraging the experience and best practices that were investigated at the start of the thesis and that have been outlined in previous sections, it was chosen to focus the attention on developing a tool that physicists could use for their data analysis needs which would

remove most of the burdens they had to sustain in the past. In particular, this objective can be split in multiple parts:

1. The tool should feel modern and easy to use for physicists, ideally leveraging already established tools and patterns. This is well suited by high-level APIs, made to optimise the underlying runtime while providing a seamless experience that lets users focus on the physics. Distributing an application written with this interface should be transparent for users, adapting to both single and multi-node deployments to improve shareability and reproducibility of the analysis.
2. It should make effective use of available hardware resources, ideally scaling in a linear fashion even when distributing over thousands of cores. Furthermore, it should stay effective whether the application is a simple exploratory analysis with a few operations or a large-scale production-ready workflow.
3. It should be possible to use the analysis programming layer to make further optimisations for the users behind the scenes, particularly related to improving the I/O throughput during an analysis.
4. It should be agnostic with respect to the particular execution engine or resource manager, so that it can adapt to different sites of the Grid and also new types of infrastructure deployments.

The perfect starting point for the purposes described above can be actually found within a software library targeted at HEP use cases which is commonly used among physicists, ROOT. In particular, it features a high-level interface to data analysis called RDataFrame. This tool did not natively support the distributed computing use case before the work of this thesis. Thus, it was chosen as the main building block on top of which the objectives described above have been addressed.

1.8 Related work

The objectives described above have been implemented by first designing and creating a distributed layer on top of RDataFrame, then evaluating it in various contexts and expanding the research to a few key areas.

First, it was decided to follow the popular MapReduce paradigm for the distribution of tasks. In order to achieve this, the distributed RDataFrame tool needs to take care of packaging the user application in kernels of execution (i.e. the mappers) and then reduce their partial results along the way. Also, the tool takes care of automatically splitting the input dataset in chunks without user intervention, based on ranges of entries. Finally, the implementation is modular, so that the logic to split the dataset and create the MapReduce functions is separate from the logic that creates and distributes tasks to computing resources.

This allows to program different execution backends for distributed RDataFrame. In the course of this work, four different engines have evaluated. Two of them, namely Apache Spark and Dask, fit in the HPC scenario and can be compared similarly on managed computing clusters. Their scalability in using distributed RDataFrame has been tested on CERN resources up to 2048 cores, but in general they are used as execution engines throughout various works in this thesis. Another two engines, AWS Lambda and OSCAR, belong to the serverless computing approach, a different model than the traditional HEP distributed computing one, that is commonly used in cloud environments. Their usage is evaluated in fact on cloud resources and their implementation to fit with distributed RDataFrame provided some unique challenges that will be discussed later on in the thesis.

Another important line of research for this thesis was the improvement of throughput related to I/O for distributed RDataFrame. In particular, the issue of caching input datasets to speedup analysis runtime was evaluated. A first introductory work was conducted by comparing two caching strategies, namely caching on a single server vs. caching on the computing nodes, using existing tools in the field. Drawing from this experience, a prototype for a new caching mechanism was developed for the next-generation ROOT I/O system and it was tested with distributed RDataFrame on bleeding-edge object store technology offered by Intel.

1.9 Structure of the document

After the introduction outlined in this chapter, this thesis is structured as follows:

- The foundational tools for this work are better described in Chapter 2. Other pieces of software that are more specific to a certain work will be mentioned in the respective chapter.
- Chapter 3 contains a detailed description of distributed RDataFrame, its main design concepts and implementation, discussing its role as an interface for distributed analysis in HEP.
- Chapter 4 discusses the works related to scalability in managed computing infrastructures with distributed execution engines such as Spark and Dask.
- The research about caching strategies in this context is detailed in Chapter 5.
- The particularities of developing a serverless engine for distributed RDataFrame are outlined in Chapter 6, which also presents evaluations on different platforms aimed at cloud computing.
- Finally, Chapter 7 draws a few conclusions from all the work put into this thesis, outlining the current status of research and open questions for the future.

Chapter 2

Tools

This chapter contains descriptions of the software libraries that are deeply connected with the work pursued in this thesis. All the following chapters will feature the distributed analysis layer of ROOT as a key ingredient. Thus, the following sections of this chapter will first describe in more detail the functionalities offered by ROOT relevant for this work, then introduce other building blocks for most of the studies developed. Software which is relevant only for specific parts of this thesis is described in the appropriate chapter.

2.1 ROOT

ROOT is a modular scientific software toolkit. It represents a de facto standard for many parts of the HEP data pipeline, involving storage, processing and visualisation of datasets. Its core logic is written in C++, but all its functionalities can be accessed through a Python API as well. ROOT offers a complete framework and environment for developing and running physics analysis, and for storing data in an efficient way [60]. The ROOT components that are most relevant for this thesis include:

- A powerful I/O subsystem, capable of writing arbitrary C++ objects to files.
- A columnar data format that has been used to store more than an Exabyte of physics datasets so far [61].
- An interactive C++ interpreter, called Cling [62], which enables dynamic C++ code compilation and execution.

- Automatic and dynamic Python bindings, called PyROOT, powered by Cling.
- A tool to program and parallelise HEP data analysis workflows, called RDataFrame, the main reference for this thesis that builds new features on top of it.
- Graphical facilities to represent distribution of physics variables via plots.

2.1.1 I/O

Section 1.3 informs about the specific data layout that best represents particle information coming from the accelerator. This has been implemented in ROOT, which defines a data format (on disk and in memory) used in all HEP datasets. It is binary, columnar and capable of storing any kind of user-defined object in a file. Since the software framework is mainly implemented in C++, a ROOT file can store arbitrary C++ objects, with an automatic compression mechanism that splits complex classes into simpler components. Also, using a columnar layout further reduces the amount of read transactions needed, since different columns can be read independently from each other. Traditionally, the I/O layer in ROOT was implemented in the TTree class [63], and soon in RNTuple [64].

TTree A TTree is a generic container of data, capable of holding any type of C++ object. This goes from fundamental types to arbitrarily nested collections of user-defined classes. TTree organises data into columns, called *branches*. A branch can contain complete objects of a given class or be split up into sub-branches containing individual members of the original object. Each branch stores its data in one or more associated buffers on disk. Different branches can be read independently, thus TTree is a columnar data format at the top level of the branches (user-defined types stored in TTree are still accessed in their entirety). Furthermore, the granularity with which a TTree dataset can be read goes even further. When reading one or more columns, TTree is able to serve different groups of entries independently. The minimum amount of rows that can be read independently is also called an *entry cluster* or just *cluster*. By default, the size of a TTree cluster is 30 MB, but this value can be adjusted by the user when writing the dataset.

TTree has become the de facto standard data format in the area. Its on-disk columnar layout allows for efficient reading of a set of selected columns, a common case in HEP analyses. However, future experiments at the HL-LHC are expected to generate one order of magnitude larger datasets which makes researching the benefits of using high-bandwidth low-latency distributed object stores especially relevant. TTree only supports I/O transactions to file-based systems, which may prove not flexible enough for future challenges. This will be further explored in Chapter 5.

RNTuple RNTuple [8] is the new, experimental ROOT columnar I/O subsystem that addresses TTree's shortcomings and delivers a high read throughput on modern hardware. In RNTuple data is stored column-wise on disk, similarly to TTree and Apache Parquet [65]. The further advantage is that RNTuple is truly columnar at all level of structures, down to the fundamental types that make up even the most complex user-defined classes. An overview of the data layout design is depicted in Figure 2.1.

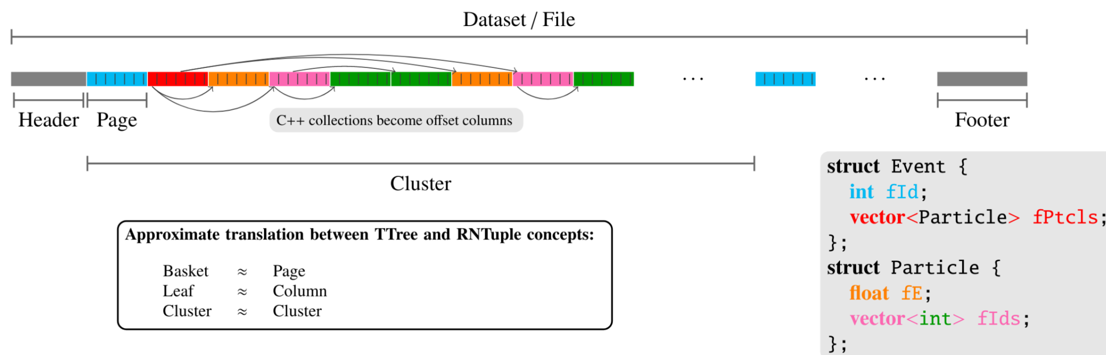


FIGURE 2.1: Data layout of RNTuple [8].

Specifically, data is organized into pages and clusters: a page is the smallest unit for storage and compression and they can also contain a certain amount of values of a column; clusters contain all the columns for a range of rows. The RNTuple meta-data are stored in a header and a footer directly within the RNTuple object. The header contains the schema of the RNTuple; the footer contains the locations of cluster metadata, from which information about columns and pages can be retrieved further. The pages, header and footer do not necessarily need to be written consecutively in a single file. As

long as the target container of the RNTuple specifies the location of header and footer, data can be stored in separate containers (e.g. different files or different objects in an object store).

The RNTuple class design comprises four decoupled layers. The *event iteration* layer provides the user-facing interfaces to read and write events and can be used from higher-level components in ROOT such as RDataFrame, which is described in Section 2.1.3. The *logical layer* defines the mappings to split arbitrarily complex C++ objects into different columns of fundamental types. The *primitives layer* manages deserialised pages in memory and the representation of fundamental types on disk.

RNTuple's layered design decouples data representation from raw storage of pages and clusters, therefore making it possible to implement backends for different storage systems, such as POSIX files or object stores. For example, a recent effort enabled using the Intel Distributed Asynchronous Object Storage (DAOS) as a backend for RNTuple and demonstrated promising performance results [66]. This thesis has explored the usage of this novel storage backend in the context of caching datasets for analysis (see Chapter 5).

TFile Datasets in the ROOT data format, be it through the TTree or RNtuple implementation, can stored on disk through the TFile class [67]. This class organises consecutive records of instances of objects on disk. In general, any type of object can be stored with TFile, it is not only limited to storing datasets. For example, it is often used to store histogram objects for easy sharing and portability. Objects in a TFile can be organised in a filesystem-like hierarchy, with subdirectories that can further contain other objects of arbitrary types. Through TFile, the ROOT I/O subsystem is able to read and write datasets both to a local disk on the computer and to remote machines with protocols such as HTTP or XRootD [68]. The latter is the most common protocol for remote data access used in the HEP field. This makes ROOT datasets easily transferable from one physicist's machine to another's for easy sharing among colleagues or from large storage facilities to the various computing nodes that may be used to run production analyses.

2.1.2 Interoperability between Python and C++

Cling is a C++ interpreter based on the Clang compiler from the LLVM software suite [69]. Thanks to it, ROOT can be used as a fast prototyping system for C++. The ability to open a ROOT prompt and type C++ code that gets executed on the fly is unique to this library. In fact, this feature has generic value and can be relevant for many other communities outside of High Energy Physics. This has become particularly evident with the recent integration of Cling functionalities in the LLVM suite with the name `clang-repl` [70]. Cling is used extensively within the various ROOT components. The I/O makes use of the reflection mechanisms offered by the interpreter to understand the objects that must be serialised to disk. RDataFrame sends user-defined functions to the interpreter for Just In Time (JIT) compilation before the execution of the computation graph. Furthermore, Cling is the foundational layer to enable bindings to other programming languages, for example Python.

The Python bindings in ROOT are named PyROOT. They are based on a library called cppyy [71], which in turn relies on the dynamic capabilities of Cling to create bindings of C++ functions to Python objects on the fly. PyROOT enables access from ROOT to any application or library that itself has Python bindings, and it makes all ROOT functionality directly available from a Python interpreter. An RDataFrame application can be written in Python thanks to both PyROOT and Cling, thus giving the basis for the work of this thesis.

2.1.3 RDataFrame

RDataFrame is the high level interface to data analysis offered by ROOT [72]. It features a programming model where the user calls lazy operations on the dataset through the API and the tool effectively builds a computation graph that is only triggered when the results are actually requested in the application. Through the Python bindings offered by ROOT, RDataFrame allows physicists to write their code with the user friendliness and flexibility offered by the Python language, while the underlying tool runs computations in C++. It also implements specific HEP features like support for systematic variations, jagged arrays and producing histograms with associated statistics.

The API of RDataFrame can be divided in three categories of operations:

- Transformations: operations that modify the dataset, such as removing uninteresting events (`Filter`) or creating new variables (`Define`).
- Actions: operations that return some result. Counting how many entries are left after filtering (`Count`), summing the values of a column (`Sum`) or creating a one-dimensional histogram (`Histo1D`) are all actions. When called, this type of operation registers itself in the RDataFrame computation graph but does not run immediately. The return value of this operation behaves like a future, it functions as a pointer to a result that is not there yet. Once the user needs the specified value, for example when they want to plot the histogram, the computation graph will be executed and actions will store the computed values.
- Other operations that are not included in the computation graph, but give some more information about the status of the RDataFrame. For example, users can ask for the list of column names (`GetColumnNames`).

Parallelisation has been a key feature of RDataFrame since its inception. The native C++ implementation allows to use all the cores in a single machine through implicit multithreading, which can be activated by a single function call at the beginning of the user application. RDataFrame stands on a very interesting advantange, that is the ability to know information about the input dataset such as the files and the columns that the user has requested. Thanks to the lazy API and the creation of a computation graph internally, it can thus leverage the ROOT I/O in the best way possible, for example by reading only the columns requested; or, in the multithreaded case, by splitting the execution in many tasks where each will read a portion of dataset entries that can be independently retrieved from the file.

2.2 Engines for large-scale data analysis

It has been discussed in Chapter 1 that Apache Spark and Dask are two of the most widely used distributed execution engines in industry for interactive data analysis. They have both seen usage within HEP and are thus perfect candidates to employ in the distributed RDataFrame effort. In particular, they are used as schedulers of distributed

execution. The RDataFrame machinery takes care of creating a series of tasks that can be processed independently, then passes them to the scheduler which will take care of sending the tasks to the remote nodes. Each scheduler has a different API and thus the implementation of this machinery may vary slightly. This section will briefly describe them and include their most relevant features for this thesis.

2.2.1 Apache Spark

Spark is an Apache project aimed at cluster computing and based on Hadoop MapReduce, extending it to more types of computations with a higher efficiency. The main feature of Spark is the ability to perform in-memory cluster computing to increase the processing speed of an application. Spark implements its own cluster management logic, separate from Hadoop, providing a faster and more general data processing platform [73].

Spark offers different client APIs to connect and send computations to a remote cluster from within the user application. Specifically, distributed RDataFrame employs the `SparkContext` class, which is connected to the concept of Resilient Distributed Dataset (RDD), a Spark abstraction of a sequence of elements that can be processed in parallel.

On the cluster side, one Spark scheduler is responsible for submitting computations to one or more Spark workers and retrieve the results before sending them again to the user. It offers built-in scheduling capabilities, useful when there is complete control of the nodes and the Spark services can be started manually. Most often, a Spark deployment will rely on external tools for the resource management. Among these: YARN [74], the default Hadoop resource and Kubernetes [75], a deployment with focus on containerised environments.

2.2.2 Dask

Dask [76] is a Python library that allows to easily parallelise existing workflows. It is mainly targeted at supporting other common Python analysis tools like Numpy [77] or Pandas [78], but is flexible enough to accommodate any type of computation. Thus, it offers many interfaces for data processing, including machine learning and real-time

analysis. Dask offers a wide set of configurations thanks to which an application can be scaled to different cluster setups, including:

1. Start all the remote nodes from a single machine through SSH.
2. Leverage existing cluster deployments with Kubernetes or YARN.
3. Connect to high performance computing resource managers that implement batch submission systems, such as HTCondor, Slurm or PBS.

Two ingredients are necessary in order to distribute computations in a Dask application. The first is the object representing the remote cluster itself, including how many resources will be assigned to it for the duration of the application. The second is an object representing the connection between the local machine and the remote cluster. This is called `Client` and can be used with any of the different implementations of resource managers available in Dask described above. The `Client` API allows users to asynchronously launch tasks to the remote cluster.

Chapter 3

Design of a programming model for distributed analysis in HEP

In order to address the objectives outlined in Section 1.7, ROOT RDataFrame was chosen as the starting point to develop a new distributed layer for HEP data analysis purposes. RDataFrame itself already addresses the vast majority of the use cases in this field, while at the same time being part of the most established and commonly used software for HEP data processing.

One of the contributions of this thesis is the development of an extension of RDataFrame to help physicists seamlessly distribute their analyses. This has been implemented in the form of a pure Python package natively integrated in the ROOT software library, referred to as distributed RDataFrame. Its purpose is providing an API to scale out an RDataFrame application to an arbitrary number of nodes. This chapter will go into more detail about different aspects of this package, highlighting some of the design choices and the machinery that was implemented consequently.

This package has served as a common tool for many different investigations throughout the course of the thesis. Thus, this chapter represents a foundational layer for the discussions in following chapters.

3.1 State of the art

Following the context introduced in Chapter 1, it is clear that HEP data analysis is characterised both by very computationally intensive, large-scale distributed applications

and a user base often lacking a computer science background and a varying degree of programming skills. This motivation has led many efforts over the years to investigate ways to abstract away from the lower-level programming logic, both in terms of easier programming interfaces and more ergonomic interfaces to access to computing resources.

The LHC experiments support their own software stack, which usually includes also data analysis tools. ATHENA [79] and CMSSW [80] for ATLAS and CMS, respectively, represent large code bases to cope with many requirements of the data pipeline in the collaboration and are sometimes also used for the final analysis step. Since these libraries can become quite heavy and their setup is not always trivial, it is also very common to see smaller analysis frameworks developed within one or more research groups of the collaboration, by those more interested in investigating certain particle interactions. Sometimes, these smaller libraries aim at being generic, so that many researchers could use them as building blocks for their code. A few examples include the Latinos framework [81], Bamboo [82], CMGTools [83], which were all developed in the context of various analyses of CMS data. In turn, these frameworks are usually just higher-level wrappers around other APIs, for example the nanoAOD-tools [84] by CMS, coffea [85] or RDataFrame.

This approach with multiple layers of software libraries between the low-level data processing and the high-level interface for final users is quite common in the field. One of its strengths is that it creates a better separation of roles, allowing a few domain experts that develop analysis frameworks to use performant processing tools while exposing an API that feels familiar to the physicists because it provides those operations that are peculiar to HEP analysis. In fact, this can be brought to the point where the analysis library exposes its own domain-specific language (DSL), keeping only those keywords that physicists are used to the most when talking about analysing physics events. Sometimes the DSL is embedded within a more generic interface and can be used as part of the programming interface. For example ROOT itself has offered the possibility to analyse TTree data by running the TTree::Draw function using a syntax to filter events and create variables called TTreeFormula [86]. Other times the DSL is the user-facing API itself, such as in the case of MadAnalysis [87] or ADL [88].

On the side of gaining access to remote resources and using them, it has been mentioned in Section 1.5 that CERN users have been able for a long time to connect to a terminal-based interface called LXPLUS that provides them with the software to run the analysis and access to resources through HTCondor. This workflow based on batch queues does not fit well with the goal of giving end users the ability to run their analyses in an interactive environment. This approach has gained increasing popularity in the HEP field in the last few years, also sustained by a similar, more generic trend happening at wide in the industry, which relies on widely used data science methods and tools. It is characterised by a focus on interactive workflows, supported by a mixture of infrastructure deployment that allows requesting computing and storage resources with very low-latency and interactive software tools supported by a web-based interface (most often based on Jupyter notebooks [89]). This combination is often referred to with the term "Analysis Facility". An example implementation is offered by SWAN [90], a service developed at CERN that connects the Jupyter notebook to personal user storage and all software libraries needed by physicists. The system also integrates natively with a Spark cluster at CERN that can be used for interactive usage of distributed nodes. The service has been bundled in a containerised environment for general purpose and scientific use called ScienceBox [91]. Following the same model, a team of CMS collaborators has developed an implementation of analysis facility on US resources [92].

With respect to the literature detailed above, RDataFrame positions itself as a coherent and simple interface that removes many frictions in the analysis definition for users. The composability of its API makes it a perfect candidate for the extension to distributed environments that is proposed in this thesis. Adding an automatic system for steering distributed computations natively in ROOT could benefit all higher-level tools such as Bamboo or CMGTools that currently need to maintain their own logic for splitting the dataset and creating tasks. The provided API, that is the one already offered by RDataFrame, opens the door for different kinds of distributed analysis definition, including executing any valid C++ code expression that is interpreted by Cling, loading C++ libraries with optimised functions or using native Python functions. The implementation of distributed RDataFrame is generic enough that it can be used efficiently on different types of deployment, ready for all the different implementations of

analysis facilities that may arise in the field. Evidence of this modularity is presented in Section 3.4 for what concerns managed facilities and Chapter 6 for scaling the applications in unmanaged cloud environments.

3.2 Maintaining the established API

The main features of RDataFrame have already been discussed in Section 2.1.3, among which its use of a declarative interface and lazy evaluation of the functions called by the user. These contribute to making a composable, user-friendly and generic API. Not only that, but the declarative approach, since it defines a limited, high-level set of operations that can be run on the dataset, also opens the door for a smooth integration with various execution engines to apply data parallelism, be it local or distributed. In fact, the first decision in designing the distributed layer was to keep exactly the same user interface. To demonstrate this, Figure 3.1 shows two listings containing a simple example of an application with RDataFrame (on the left) and the same application done with distributed RDataFrame (on the right). Beginning from the listing on the left, we can describe the following steps:

- The goal of this application is to compute the approximate value of π as four times the number of points that fall within a circle with radius $r = 1$ divided by the number of points of a square with side $l = 2$ that encloses the circle.
- The main function starts at line 8. The first step is creating the RDataFrame object (line 9), which is the entry point for the rest of the analysis. There is no input dataset in this case, the RDataFrame will simply generate sequential entries at runtime, from zero (included) until the value of `NPOINTS` (excluded).
- The calls to the `Define` method create new columns in the dataframe, i.e. the points in the square and the distance from the center. In line 20 the `Filter` operation removes those entries where the distance is lesser or equal to one, i.e. only the points inside the circle are considered. The `Count` operation tells RDataFrame to count how many entries pass that filter.

- So far, all the calls to the API have not triggered any computations. The execution starts at line 23, specifically by calling `GetValue` on the result of the previous call to `Count`.

Looking at the listing on the right side of the same image, it should be noted that:

- The two main functions (lines 8-25 on the left, lines 23-40 on the right) are exactly the same. That is, the user logic that expresses how the dataset should be processed is kept intact.
- There is only one extra ingredient needed for the creation of the RDataFrame object, as shown in lines 19-21. In order to distribute computations to a cluster, the RDataFrame object needs a way to connect to the cluster itself.
- When using Spark, Dask and other distributed execution engines, the connection to the cluster can be established in the application by creating a separate object that will be responsible for sending tasks and retrieving their results. In Dask, this is called a `Client`.
- The RDataFrame created at line 20 accepts an optional argument, called `daskclient` when using the Dask backend, that takes in input a connection object to the cluster. Thanks to this hook, the distributed RDataFrame can decouple completely the scheduling of the resources from the definition of the analysis.
- Lines 10-16 show the creation of the Dask client object. In this case, a `LocalCluster` will spawn multiple Python processes on the same machine to parallelise the execution of the RDataFrame computations. This is just one example, as already mentioned Dask offers facilities to send the computations to multiple different nodes.

```python
import ROOT

NPOINTS = 10_000_000

def create_rdf() -> ROOT.RDataFrame:
  return ROOT.RDataFrame(NPOINTS)

def main() -> None:
  df = create_rdf()
  pidf = (
    df.Define(
      "x",
      "gRandom->Uniform(-1.0, 1.0)")
      .Define(
      "y",
      "gRandom->Uniform(-1.0, 1.0)")
      .Define("r", "sqrt(x*x + y*y)")
  )
  incircle = (
    pidf.Filter("r <= 1.0").Count()
  )
  pi = (
    4.0 * incircle.GetValue() / NPOINTS
  )
  print(f"pi is approximately: {pi}")


if __name__ == "__main__":
  main()
```

```python
from distributed import LocalCluster
from distributed import Client
import ROOT
RDataFrame = (
  ROOT.RDF.Experimental
      .Distributed.Dask.RDataFrame
)
NPOINTS = 10_000_000

def create_connection() -> Client:
  cluster = LocalCluster(
    n_workers=8, threads_per_worker=1,
    processes=True, memory_limit="2GiB"
  )
  client = Client(cluster)
  return client

def create_rdf() -> RDataFrame:
  conn = create_connection()
  return RDataFrame(
    NPOINTS, daskclient=conn)

def main() -> None:
  df = create_rdf()
  pidf = (
    df.Define(
      "x",
      "gRandom->Uniform(-1.0, 1.0)")
      .Define(
      "y",
      "gRandom->Uniform(-1.0, 1.0)")
      .Define("r", "sqrt(x*x + y*y)")
  )
  incircle = (
    pidf.Filter("r <= 1.0").Count()
  )
  pi = (
    4.0 * incircle.GetValue() / NPOINTS
  )
  print(f"pi is approximately: {pi}")


if __name__ == "__main__":
  main()
```

FIGURE 3.1: Example application with RDataFrame (on the left) and distributed RDataFrame (on the right)

Figure 3.1 truly exemplifies the seamless transition from a traditional RDataFrame analysis to a distributed one. At the time of writing, this feature has been integrated in the main ROOT repository, but is still in experimental mode. This can be seen for example in the listing on the right, where the creation of the distributed RDataFrame depends on a previous alias to the correct class defined in lines 4-7. Once the extension is declared ready for production use, this small difference could be removed by dispatching the correct RDataFrame depending on whether the user provides a cluster connection or not.

3.3 The workflow of a distributed application

The listing on the right side of Figure 3.1 can be used also to describe what happens under the hood when a user runs a distributed RDataFrame application. A brief description would be that the tool creates a series of tasks by splitting the input dataset in logical parts, sends the tasks to some execution engine, each task will process the user-defined operations on its logical split, finally all task results are aggregated and sent back to the user. In more details, the workflow is enabled by the cooperation of different mechanisms:

1. The execution of the computations is triggered by usage of some action return value, as described in Section 2.1.3. This happens at line 38 of the listing with the call to `GetValue` on the result of the previous `Count` operation.

2. The action object calls into the head node of the computation graph to create all the information necessary to build a series of tasks for the distributed execution. The class responsible for this is named `HeadNode` and its functionality is better described in Section 3.7. Each task contains three items: a logical chunk of the dataset, a function that is able to create an RDataFrame object from that chunk, the computation graph that must be applied to that RDataFrame. These are all the ingredients needed for the execution of a mapper function. The head node takes care of packaging the list of tasks, the mapper function and the reducer function and give them to the distributed scheduler.

3. The scheduler receives the information from the head node and builds a MapReduce pattern. This can be implemented differently depending on the available API of each scheduler (more on that in Section 3.4). The tasks are sent to the distributed processes, both the mapping and reduction phases are executed on the workers.

4. The head node receives back from the scheduler the final results of the distributed execution. There is one result for each action operation the user asked. The head node takes care of filling the actions with their result values. This ensures that the results are available in the user application and connected to the variables created, as it would be expected.

3.4 A modular implementation

Another core design goal of this contribution is to decouple the logic that deals with creating the RDataFrame tasks from the logic to steer the computations distributedly. This translated into a modular structure of the package, providing base protocols for the parts of the workflow that may have different implementations. Currently, this is particularly evident in two areas, that are highlighted in the following sections.

3.4.1 Modularity with respect to the execution engine

The first aspect to analyse is how distributed RDataFrame can accomodate different execution backends with ease. Until the moment of writing, four backends have been developed and tested with this tool, of which two are natively integrated within ROOT, namely Spark and Dask, another two have been the focus of more specific research efforts detailed in Chapter 6.

A base class for all distributed RDataFrame backends is defined inside the `Backends` subfolder of the project, which serves as a protocol for other implementations. Three main ingredients are required from any backend:

- A method that creates an RDataFrame connected to the appropriate cluster type, accepting the same input arguments as the traditional RDataFrame class.

- A method that runs a MapReduce workflow, given the mapper and reducer functions and a set of dataset chunks to apply them.
- A method to send files from the client session to the remote machines. This is a crucial feature for distributing additional user code that may be needed during the analysis, such as C++ header files.

New backend implementations may subclass the base backend and implement the required methods. The package automatically picks up implementations placed under a subfolder of the `Backends` folder. This is shown in Listing 1. For more details about their implementation, see Chapter 4.

```
.
+-- Backends
|   +-- Base.py
|   +-- Dask
|       +-- Backend.py
|   +-- Spark
|       +-- Backend.py
+-- [...]
```

LISTING 1: Simplified folder structure of distributed RDataFrame. The `Backends` folder contains implementations for the execution backends.

3.4.2 Modularity with respect to the data format

Since RDataFrame supports different types of input data format, the same support is desirable for distributed RDataFrame too. In fact, different data formats will have different I/O rules and may need different treatment in terms of how they are split into smaller chunks. In distributed RDataFrame, the class responsible to handle the splitting of the dataset and generation of the tasks is called `HeadNode`. Instances of this class represent the manager of a certain RDataFrame computation graph. Also this class is a base for different implementations, which differ on the data format they target. The data formats supported currently in distributed mode are TTree and the simple generation of sequential entries (previously shown in Figure 3.1).

This approach makes it very easy to add support for other data formats which are also supported by the local version of RDataFrame. For instance, support for RNTuple has been developed and used in a research effort done during this thesis, described in Chapter 5.

3.5 Generalised task creation algorithm for distributed backends

Automatic creation of tasks for distributed execution is a core feature of the distributed layer for RDataFrame. The main goal is ensuring that two distinct tasks will operate on two distinct parts of the dataset specified by the user. This requires exploiting as much as possible the I/O granularity offered by the data format when reading a ROOT file. As a simple example, let us take a ROOT file containing a dataset with two columns and two entry clusters (as defined in Section 2.1.1). Two tasks can be possibly defined for this example, the first task operates on the first cluster, the second task on the second cluster. Both tasks need to also receive the information about the file(s) that contain the entries they were assigned. In general, the following rules can be derived about task creation:

- A task should be assigned with a range of entries to process from a particular set of files. This range should be aligned with respect to the cluster boundaries, to avoid reading a whole group of rows in memory and then just processing a few of them.
- For any given application, the maximum amount of tasks that should be created is equal to the total amount of clusters in the dataset. Creating more tasks would mean triggering unnecessary I/O requests to (potentially remote) ROOT files, adding a significant overhead to the analysis.

3.5.1 Offloading the creation of task ranges to workers for parallelisation

Creating the list of tasks to be passed to the execution engine used to be a serious bottleneck in initial iterations of distributed RDataFrame. This issue arises from the fact that

the information regarding clusters in a file can only be queried after the file has been opened. Before this work, all files of the dataset had to be opened on the client side to query the relevant metadata. Tests done at CERN have shown that a single open operation of a remote file stored within the CERN network from a client machine within the same network can take a few seconds to complete. A real HEP analysis can process O(1000) files, thus bringing a startup time cost of tens of minutes just to create the tasks. In this work, a new algorithm for task creation has been developed to completely remove the need to open remote files in the client, thus *bringing the startup time close to zero*. This is implemented via creating the task in two steps, one on the client side and the other on a distributed worker.

3.5.2 Fast task generation on the client side

The generic idea of the algorithm is as follows. On the client side, the only information that is readily available – because it is provided by the user – is some specification of the input dataset, namely the list of files to process. This list is split into a series of tasks with length equal to the number of chunks specified by the user. On the local machine, no file is opened. Instead, splitting the input files into tasks is done by considering each file as an entity that can be divided according to percentages. For example, an application that processes two files in two tasks would have one task processing 100% of the first file and the other task processing 100% of the other file. The granularity can go even further: a task can be assigned with a range of percentage of a single file (e.g. $[33, 66)$). This is done because such a generic task will then be sent to some distributed worker which will need to convert these percentages into actual cluster boundaries. The difference now is that files are only opened on the computing nodes that need to process them. This step of the algorithm thus treats all input files as having the same size (since they are all treated in terms of percentages of their content). Although this is not always true, some considerations can be made about it. An example that could lead to unbalanced load in the tasks would be if the dataset was made of two files, one with a very large size (and number of clusters), the other with a much smaller size. In case only two chunks were requested, then one task would process more data than the other. But, in practice, different files that form datasets of HEP experiments usually

have similar sizes and number of clusters. Furthermore, since a single dataset may be composed of thousands of files as mentioned above, the possibility of creating smaller tasks is mitigated.

```
1 def create_generic_tasks(filenames, n_partitions):
2   all_files = compute_files_in_tasks()
3   percs_f, percs_l = compute_percentages()
4   res = []
5   for files, perc_f, perc_l in zip(
6     all_files, percs_f, percs_l):
7     res.append((files, perc_f, perc_l))
8   return res
```

LISTING 2: Approximate implementation of the creation of tasks on the client side.

Listing 2 shows more details about the client side. In particular:

- Line 1: the function expects in input only the list of files that must be processed (i.e. the dataset) and a number of partitions in which they should be split.
- Line 2: gather a list of the files that should be processed in each task. This involves a few extra steps which are omitted from the listing:
  - First, the function creates a list of percentages according to how many partitions are required. For example, 5 files and 3 partition would give a list such as `[0, 1.66, 3.33, 5]`.
  - Then it retrieves the corresponding list of file boundaries as integers: `[0, 1, 3, 5]`.
  - Then it computes the difference element by element, to get the corresponding portion of the file for each percentage of the first list: `[0, 0.66, 0.33, 0]`.
  - From the list of file boundaries, the begin and end index (end exclusive) of the files in each task can be computed. Using this information, the `all_files`

variable is a list where each element is another list containing the files that a task should process (a subset of the list of files in input to the function).

- Line 3: gather two lists with the percentages of the first and last files, respectively, in each task where the processing should begin or end. Using the same example as above, `percs_f` is `[0, 0.66, 0.33]` and `percs_l` is `[0.66, 0.33, 1]`. Taking the first task for example, it will read files 0 and 1, file 0 will be read starting from percentage 0 (i.e. from its beginning) and file 1 will be read until percentage 0.66 (i.e. 66% of the entries in that file).
- Lines 4-7: construct the list of tasks by storing together the file indexes, the percentage of the first file and the percentage of the last file in a single object (i.e. the payload) for each task.

At the end of this function, each created task contains: a list of files (subset of the list of total files of the dataset), the percentage from which the task should start processing the first file, and the percentage until which the task should start processing the last file. Files in between the first and the last will be fully processed.

3.5.3 Remote-side conversion of the task

The payload obtained from the function described in the previous section will be sent to a remote worker. There, the approximate task needs to be concretised, i.e. the percentages need to be converted to actual entry numbers in the files. It must be noted again that this is deferred until the task is processed on the remote workers to avoid opening the files on the client side (a costly operation). Listing 3 shows an approximate implementation of this conversion from the payload task to an actual task. In particular:

- Lines 2-5: gather information from the payload. These are shown here to help readability, as they will be used in later parts of the function. `first_file_idx` and `last_file_idx` represent the index of the first and last file that should be processed from the list of files received in the payload. `perc_f` and `perc_l` represent the percentages of the first and last file obtained from the function in Listing 2.

```
def convert_task(task):
    first_file_idx = 0
    last_file_idx = len(files) - 1
    perc_f = task.perc_f
    perc_l = task.perc_l
    clusters, entries = get_clusters_entries(task.files)
    begin_entry_f = get_begin_entry(
        perc_f, entries[first_file_idx], clusters[first_file_idx]
    )
    end_entry_l = get_end_entry(
        perc_l, entries[last_file_idx], clusters[last_file_idx]
    )
    return task.files, begin_entry_f, end_entry_l
```

LISTING 3: Approximate implementation of the conversion of a generic task into an actual task in a distributed worker.

- Line 6: retrieve, for each file in the payload, a list of the entry clusters and the total number of entries in the file. Since the entry cluster is the minimum amount of entries that can be read independently from the rest of the file, it is crucial that concrete tasks are aligned with respect to cluster boundaries, so that I/O is minimised.
- Lines 7-9: convert the percentage of the first file from the payload to a real beginning entry of the first file. This is computed in the following steps:
  - The percentage is multiplied by the number of entries in the file, to get a candidate beginning entry.
  - This candidate is compared against the list of cluster boundaries for that file. The clusters are considered as bins and the corresponding cluster index is computed for the candidate entry.
  - The beginning entry of that cluster is taken as the beginning entry for the task.

  - The algorithm establishes whether a generic task should actually process a cluster or not based on whether the candidate entry coincides with the beginning entry of the cluster. For example, suppose a cluster spans entries from 10 (inclusive) to 20 (exclusive). If the candidate entry is equal to 10, the task will start processing from that cluster, otherwise not. This ensures that only one task takes that particular cluster of entries, so that there cannot be two tasks processing the same entries.

- Lines 10-12: convert the percentage of the last file from the payload to a real ending entry of the last file. This follows the same algorithm as in the previous item.
- Line 13: return the concrete task. This includes the list of files received in the payload, the beginning entry where the task should start processing the first file, the ending entry where the task should stop processing the last file. These entries are aligned with respect to cluster boundaries. Files in between the first and the last are processed fully.

3.6 Efficient execution of C++ code in Python processes

The details given so far already show the core ingredient needed to enable the new backend. But another important part of this work is taking care that parallelisation of RDataFrame computations is properly handled by Dask on the computing nodes. At the node level, the parallelisation is done through Python multiprocessing. Each process is a separate Dask worker that will receive one or more tasks to run. Inside each Python process, Dask will spawn multiple Python threads. The main thread is responsible for running the user-provided function, which in the context of distributed RDataFrame is a mapper (or reducer) task. Other threads involve Dask internal mechanisms such as inter-task and inter-node communication.

A distributed RDataFrame application is written in Python and also the functions that are serialised and sent to the Dask workers are in Python. But when a worker is executing a task and calls into the RDataFrame API, it is going to run C++ code because the actual implementation of RDataFrame is in C++. This is made possible thanks to the dynamic Python bindings available in ROOT, named PyROOT, which

can load a C++ library at runtime and call into it right away, thanks to the ROOT C++ interpreter. Thanks to PyROOT, there is no need to generate or send static Python-C++ bindings to the computing nodes, Dask will see only the Python layer of the tool and the implementation of distributed RDataFrame will transparently and automatically call functions implemented in C++.

Since Python is constrained by the Global Interpreter Lock (GIL), the different Python threads actually need to wait on each other before doing their job. Crucially, the main thread that is running the computations should never acquire the GIL for too long to avoid blocking resources for the others. Instead, by default, the thread that runs the task code through PyROOT holds the GIL and does not release it when calling into C++ functions. Thus, the threads that are responsible for communication starve, leading to major slowdowns and timeouts when running the tasks on the Dask workers.

In order to overcome this issue, this work changes the implementation of the distributed RDataFrame Python package to ensure that the Python GIL is released when calling into C++ RDataFrame code to run a task. This is done by exploiting a feature of the Python bindings in ROOT, that is the possibility to unlock the GIL for the duration of a specific function. This is done at the beginning of the mapper function in each task, before executing the computation graph. From the point of view of a Python process running in one of the computing nodes, all Dask mechanisms can now work freely while the C++ computations are running. Communication between different Dask workers or with the Dask scheduler is ensured and the main Python thread does not block them anymore, thus bringing a tangible performance increase. Without this change, the Dask backend would just not be convenient for users and would not be competitive with the already existing Spark backend.

3.7 Distributable representation of the computation graph

It is quite common in production HEP analyses to have very large computation graphs made of thousands of different calls to the RDataFrame API. In the distributed version, the computation graph needs to be sent as part of the information for the mapper function as briefly stated in Section 3.3. Thus, this opens the question of what is the best

representation of the graph that can at the same time hold a large number of nodes and be serialisable so that it can be packaged in a task and sent to the remote workers.

Let us start this discussion by saying that the computation graph connected to a single RDataFrame object can be represented with a tree structure. The root node of the tree is the RDataFrame object itself. All the calls to the API will branch out from that object and each node may have any number of children. The tree can have arbitrary depth, depending on how the user application is organised.

A first implementation would have every node of the graph store references to all its children, then the graph would be recreated in each task with a top-down approach. This approach had two downsides: it included unnecessary overhead in the serialisation step since every node had to also serialise its children recursively; no more than one thousand nodes could be present in the graph, due to the default Python recursion limit.

The current, more mature implementation overcomes the aforementioned issues by representing the graph as a flat tree, completely avoiding recursion. Every node is assigned an integer identifier at creation. The computation graph sent to a task will be a Python dictionary, where each key is the integer identifier and the value its corresponding node. Inside the mapper, the dictionary is iterated and every RDataFrame operation is called on the correct node thanks to a mechanism that associates the node to its parent via their identifiers.

With the current implementation, there is practically no limit to how large a computation graph can be when writing a distributed RDataFrame application. The flat representation also removes the overhead of the recursive one, so that the creation of the tasks on the client side is even faster.

3.8 Passing partial results between different processes

While running the MapReduce workflow, the most probable scenario is that different mapper tasks will run in different Python processes, the same applies for the reducer tasks. Thus, proper communication of the outputs of these functions between different processes must be ensured. In turn, this means that the results must be serialisable.

The return types of actions in the RDataFrame API are actually not properly serialisable due to some details of their C++ implementation. These types are fairly complex since they need to cooperate with the rest of the RDataFrame computation graph machinery. But in practice, there are only two ingredients required to address the described issue: storing the result obtained by running an action and knowing how to merge two results coming from the same type of action. To this end, a new set of C++ classes was developed.

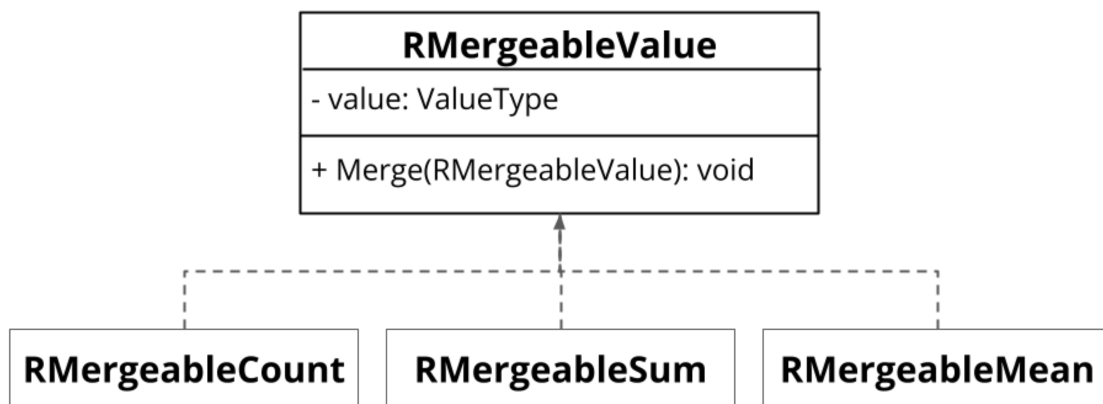


FIGURE 3.2: Simplified inheritance tree for the classes responsible to store results of distributed execution and defining how to merge them.

Figure 3.2 depicts a simplified inheritance tree of these new classes. It shows a base class, named `RMergeableValue`, which stores in its attribute `value` the result of running the action. The class is templated on the type of the result. It also defines a method `Merge`, that derived classes must implement to perform the merging steps between their results. For example, merging two instances of `RMergeableCount` is simply done by adding their respective values, merging `RMergeableMean` objects requires knowing both the mean and the count of each one involved, other types of actions may need more complex merging functions. The image only shows a few examples of the derived classes, but there is almost a different class for any operation supported in distributed mode. This class hierarchy allows efficiently communicating results between different processes, leveraging the serialisation/deserialisation machinery already in place within the distributed execution engines.

3.9 Conclusions

This section introduced distributed RDataFrame, an extension to the RDataFrame data analysis interface in ROOT. It is one of the main contributions of this thesis that targets the first objective outlined in the list of Section 1.7 and provides a foundational tool to address the other points of the list.

The extension is a Python package that wraps the operations requested by the user in a serialisable computation graph and automatically splits the input dataset in logical chunks. It implements a MapReduce pattern that is executed via a distributed engine, where each mapper applies the RDataFrame computation graph to a different chunk of the dataset and the reduction merges the partial results of the mappers two at a time until only one final set of results is obtained and sent back to the user application.

The package is designed with modularity in mind and this is shown on two levels. On the one hand, multiple execution engines can be fitted in this package by implementing a backend that uses the correct API to launch the distributed tasks. In fact, two backends are already integrated in the tool within the main ROOT repository, namely Apache Spark and Dask. Examples of other backends developed for the research of this thesis are shown in Chapter 6. On the other hand, different input data formats can be used for distributed processing. Currently, the package supports the traditional TTree I/O layer and in the future it will also support reading RNTuple data in the distributed tasks (a first example of this is shown in the work presented in Section 5.5).

Many optimisations were brought to this tool over the research period of this PhD, often based on input from the physicists and results of performance benchmarks. For example, a task generation algorithm was implemented to avoid the need for opening remote files on the local machine of the user, bringing the startup time close to zero.

This contribution thus provides a way for physicists to avoid the shortcomings of traditional distributed computing workflows and make a step forward towards enabling future large-scale interactive data analysis use cases.

Chapter 4

Efficient distribution of physics computations

Chapter 3 has discussed how distributed RDataFrame can fulfill the first goal set in Section 1.7 of allowing users to seamlessly distribute their analysis from one to many nodes. But if this is put into the perspective of large-scale production workflows as those described in Section 1.5, it becomes clear that the distribution of computations should not only be possible, but also efficient and scalable. Modern workstations can usually count on at least 8 or 16 physical cores, large server nodes that sometimes are available within research groups feature CPUs with a few hundred cores. For the cases when a single machine has enough computing power, the baseline RDataFrame implementation can already offer full exploitation of the available resources with multi-threading. Consequently, the target for distributed computations is coordinating tens or hundreds of computing nodes with an aggregated core count in the thousands.

This chapter introduces a few performance studies, specifically targeting the second requirement in the list of objectives of Section 1.7. In Section 4.3, the two backends for distributed RDataFrame integrated in ROOT, namely Spark and Dask, are put to the test on the same resources and the same analysis. The scaling achieved on many nodes with both backends is satisfactory and comparable, as expected since the main driver of the tool is the underlying C++ computation graph. In Section 4.4, a more concrete analysis example is given, using distributed RDataFrame on HTCondor resources to drive a full-scale application that involves the most complicated parts of the typical analysis workflow.

4.1 State of the art

Approaching distributed computing has some inherent challenges that are not present when working on a single machine. One has to deal with the coordination of work between different nodes, status synchronisation, network I/O, data and work replication, job failures, etc. Thus, specific software is needed to organise the different levels of the workflow in a distributed cluster: resource management, task scheduling, communication with the user. Sometimes the same tool takes care of more than one aspect of the overall process.

In HPC and scientific computing environments it is very common to manage the computing resources via a batch job submission system. How these resources are scheduled and used is then a responsibility of the final user or of some scheduling library they may employ in their application. The batch system knows about the available hardware of the cluster, receives in input from users programs to execute and requests for a certain amount of resources, then allocates and deallocates resources to user applications based on a priority queue. As previously mentioned in Chapter 1, HTCondor is the most widely used job queueing system in HEP and Slurm also sees frequent adoption. Although their usage varies dramatically depending on factors such as the field of research, level of expertise or the research group, some common patterns can be highlighted as done in the introduction of this thesis and in a work by Erickson et al. [93]. This approach often becomes elaborated and cumbersome for users, so it is common to build layers on top of this queue systems to better organise and schedule the work. For example, by employing master-worker concepts to schedule jobs and retrieve back their results [94]. This approach is quite generic and can be found in many applications, such as in an effort by Zheng et al. [95] to standardise the usage of the batch queues across different institutions for crop yield simulations [95]. Similarly, one can find examples of workload abstraction for HTCondor in HEP as well. Collaborations such as CMS and ATLAS use higher-level concepts such as pilot jobs to steer the work of batch pools across their sites world-wide [96, 97]. Sometimes these tools go even further and provide programmatic interfaces to help users in organising all the files needed for an analysis, as is the case with the ATLAS PanDA and the LHCb GANGA systems [97, 98].

While using job submission frameworks has always been common in scientific environments targeted at HPC, this was not the case for other fields in industry. Thanks to the huge increases in data volumes derived from massive usage of internet technologies and extraction of information from many more sources than before (a phenomenon collectively referred to as Big Data), large-scale distributed computing has become a pressing matter for a large majority of companies. In response, quite a few software libraries have emerged that allow to define computation graphs and then automatically parallelise the operations on distributed resources, usually through running a scheduler service that sends tasks to various executors.

One of the pioneering actors in this regard was Google with two whitepapers: one in 2003 regarding the definition of a new way to store files in a distributed cluster called Google File System [99] and another the following year describing a new paradigm to process those files, called MapReduce [100]. The latter in particular was one of the founding elements of the Apache Hadoop software ecosystem [59, 101] and is to this day one of the most widely used distributed computing paradigms. One important piece of software derived from Hadoop MapReduce is Apache Spark, an evolution of the concept with in-memory processing and strong scalability [73]. In many use cases it has become a de facto standard for distributed computations, from Extract-Transform-Load (ETL) to Machine Learning workflows alike [102]. Examples include multi-stage deep learning approaches [103], generic feature selection frameworks [104], streaming data analytics for IoT devices [105] and smart grid systems [106]. Usage extends as well to academia, for example in a geoscience effort to compare different storage systems with the objective of getting the best performance from a Spark-based analysis framework [107].

Over the last decade, the Data Science community at large has increasingly favored the usage of Python as the main programming language for data pipeline software stacks. Thus, it most common to find Python APIs as a front-end of distributed execution engines, Apache Spark making no exception. At the same time, there was no Python-only library for distributed computing. Around 2015, the Dask project was started, giving the possibility to parallelise the existing Python data science software packages in a familiar way for users [76]. Examples of using Dask to distribute computations can be found in Earth and Climate sciences [108, 109]. Also in those fields

datasets contain many years of data and can reach sizes of multiple TBs, with non-trivial multidimensional schemas similar to what can be found in HEP. In one of the cited works, data processing through Dask allows parallelising part of the data analysis workflow, the scalability results are shown up to 32 cores with a speedup value of around 9 in the best cases [108]. The same community made an effort to develop an ecosystem for distributed data analysis, recognising that previous approaches led to fragmentation and unproductivity [110]. In another work, molecular dynamics simulations were processed on a computing cluster using Dask as execution engine [111]. The split-apply-combine approach to distribute tasks shown in that work is similar to the workflow of the distributed layer for ROOT that is discussed in this thesis, but applied to a different field with different user workflows in mind. The Dask library itself provides a `dataframe` interface which can be directly used as an entry point for generic data analysis [112].

Although many tools exist to address data analysis needs of industries and academia, it should not be taken for granted that they can all work just as well in any other field. Particularly, it has been shown that for the HEP data analysis requirements a tailor-made tool like ROOT with its RDataFrame data analysis interface still has a major advantage over other industry frameworks [113]. The HEP field is not new to the investigation of large-scale distributed execution engines. The ROOT framework itself offered the Parallel ROOT Facility (PROOF), a tool to automatically parallelise HEP applications [114]. This provided a way to avoid all the manual submission work required by traditional batch systems, but it could only work with ROOT services that needed to be launched on the cluster resources (no other resource manager was supported). In 2017 two similar works presented a distributed data analysis system of the CMS experiment [15] data using Spark, but encountered limitations in having to convert data from the standard HEP ROOT format to formats that Spark could understand natively, incurring in major performance bottlenecks [115, 116]. A later study overcame this issue, but did not achieve higher scaling when using available CERN storage facilities [117]. An example of good scalability was provided by researchers of the TOTEM experiment at CERN, with a first approach at distributing a ROOT application over Spark resources in a cloud [118]. The presence of Spark in the HEP community has become relevant enough that CERN has invested in specific

infrastructure to support Spark analysis workflows [119].

No scalability test was presented with more than two thousand cores in the literature found for the HEP field. Also, while there are efforts to steer distributed computations to computing clusters through large-scale engines, none of them combines the possibility of using a user-friendly interface language like Python to actually distribute C++ computations. This thesis enables this kind of workflow through the distributed RDataFrame tool, which is able to natively distribute computations to Spark or Dask clusters. The Dask backend in particular fulfills a very strong need to make use of available HEP hardware resources which are tightly integrated with batch resource managers.

4.2 Distributed backend implementation

The implementations of the backends for distributed RDataFrame usually differ in the way the API of the respective distributed execution engine is used, but they all implement a MapReduce pattern to run the physics computations distributedly. Practically, the RDataFrame machinery creates a list of logical dataset chunks, a mapper function containing all the user-defined physics operations and a reducer function to automatically merge the partial results. The goal is then for each backend to take these objects and submit them to the computing cluster in the most efficient way. The next sections will demonstrate how this is done with Spark and Dask, respectively.

4.2.1 Executing the computation graph with Spark

The main logic of the function that implements the MapReduce pattern for the Spark backend can be seen in Listing 4. In particular:

- Line 4: the input list of chunks is passed to the `parallelize` method of the `SparkContext` instance stored within the class object. This function creates a Spark RDD.
- Line 6: the RDD supports the MapReduce process through the `map` and `treeReduce` methods, which both accept a user-provided function that should

```python
def map_reduce_spark(
  self, mapper, reducer, chunks):

  spark_rdd = self.spark_context.parallelize(chunks, len(chunks))

  return spark_rdd.map(mapper).treeReduce(reducer)
```

LISTING 4: Usage of the Spark API in the corresponding distributed RDataFrame backend

be run in the corresponding stage of the distributed execution. The `treeReduce` method differs from the `reduce` method in that it will run the reduction stage on the computing nodes, rather than bringing back the partial results from the mappers to the client node and reducing them locally.

Clearly, Apache Spark is built around the MapReduce pattern, so it is no surprise that the two functions defined by the RDataFrame machinery can be passed as-is to the engine API. The usage of the `SparkContext` and the RDD concepts, which are part of the core Spark library and there since its inception, has the added benefit of ensuring stability and minimal maintenance required.

4.2.2 Executing the computation graph with Dask

Similarly to the Spark backend, the Dask backend implementation must define a function that accepts the items described at the beginning of Section 4.2 and packages them in a way that works with the facilities available in the Dask library.

Dask offers many interfaces for data analysis, but the most interesting for the purposes of this work is called `dask.delayed` [120]. This is a Python decorator that effectively allows to run custom workflows of any type, by wrapping the user provided functions in objects that will delay the computations until actually requested by the application. In particular, calling a function that was previously decorated with `delayed` returns a `Delayed` object, i.e. a future, which will wait to start the computations until the user calls its `compute` method. Both the mapper and reducer functions created in

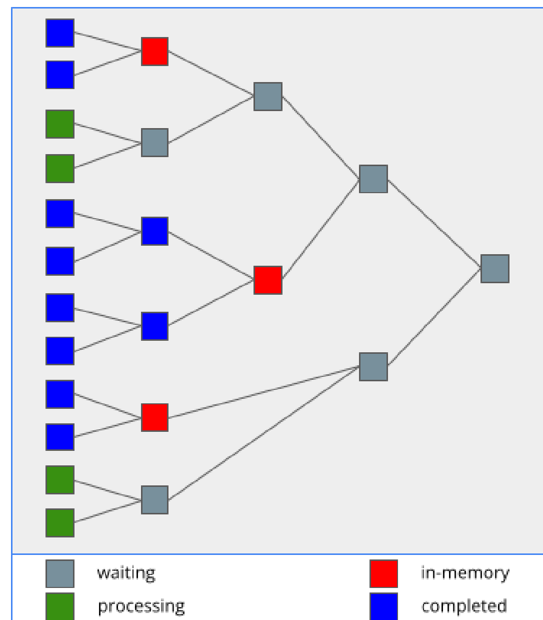


FIGURE 4.1: Visualisation of the Dask computation graph generated by calling the `delayed` mapper and reducers in distributed RDataFrame. The colours of the nodes represent their status: nodes that are waiting for results from others are in grey; those that are currently processing a task are in green; those that have completed their task but need to wait for another task before sending their result to the next reducer keep the result in memory and are shown in red; finally, the nodes that have completed their task and do not need to wait for others are shown in blue.

distributed RDataFrame are decorated with the `delayed` function, thus when they will be called on any given data chunk, they will not be executed right away but they will return a future. Dask `delayed` functions accept futures as input arguments, so that practically another computation graph is built at the Dask level: mappers return futures that are passed as arguments to the reducers, which in turn produce other futures that are passed as argument to following reducers. From the point of view of Dask, this is still a MapReduce graph (like it is done in the Spark backend), with the reduce phase done in a tree-like pattern. This is depicted in Figure 4.1. In the figure, each square represents a node of the Dask computation graph. Looking at each vertical set of nodes, the first one includes all mapper tasks, that is all the different applications of the RDataFrame

computation graph to a separate range of entries of the original dataset. All other tasks are reducing partial results of the mappers, two at a time until only the single, final result is left to be sent to the user.

Implementing this pattern on the backend side is shown in Listing 5. This shows the function that is responsible to take the computation graphs and the data chunks from the framework and connect it to Dask. In particular:

- Lines 4, 5: calls to the `delayed` interface. This makes the execution of the mapper and reducer functions deferrable. When they are called, they are registered in a computation graph internal to Dask.
- Lines 7-9: creation of the tasks by calling the delayed mapper function on each chunk of the dataset. This is represented by the first vertical line of nodes described in Figure 4.1.
- Lines 11-13: implementation of the reduce phase. The first two elements of the list of partial results are removed from the list and given to the reduce function; the result of the reduction is then appended to the same list; this is repeated until there is only one element in the list, meaning that all the partial results have been merged into the final result.
- Line 15: trigger the start of the computations by the Dask scheduler, by calling the `compute` method on the only remaining value of the described list.

4.2.3 Impact of the two execution engines on end user workflows

As stated in Section 3.4, one of the main design choices for distributed RDataFrame is to make it modular so that it can plug in different execution backends. Thus, for the final user, using the Spark backend or the Dask backend should make no difference when running a distributed RDataFrame application. In fact, they write exactly the same analysis code, with the only difference being in the setup of the connection to the cluster.

The two engines also have many similarities. Notably, they both support expressing custom computations using the MapReduce paradigm (although Dask extends the

```python
def map_reduce_dask(
  self, mapper, reducer, chunks):

  dmapper = dask.delayed(mapper)
  dreducer = dask.delayed(reducer)

  futures = []
  for chunk in chunks:
    futures.append(dmapper(chunk))

  while len(futures) > 1:
    futures.append(dreducer(
      futures.pop(0), futures.pop(0)))

  return futures[0].compute()
```

LISTING 5: Implementation of the MapReduce pattern in the Dask distributed RDataFrame backend.

support to any arbitrary execution pattern). Furthermore, they will both present the user with graphical dashboards showing real-time task execution and status of the nodes [121, 122]. Nonetheless, the two execution engines have different approaches which may result in nuanced differences regarding their usability.

One notable example is the cluster setup that is implicitly expected by the two different engines. Factoring out the standalone setups (i.e. manually launching either Spark or Dask services on various machines), Dask has the clear, crucial advantage of being able to interface to submission systems that are ubiquitous in HEP computing infrastructures. This means that setting up a Spark cluster will require either manual user intervention, or new facilities built ad-hoc for the purpose of interactive distributed analysis. With enough user demand this may become worth its cost, as mentioned in Section 4.1. But being able to use pre-existing computing resources with no added engineering, logistic or maintenance cost is undoubtedly a valuable feature.

This advantage is not only visible on the infrastructure side, but also on the end user side. Not all HEP analysts have access to Spark clusters, so their only alternative, if they would like to use the Spark functionalities, is to launch the services on the traditional distributed computing resources. This involves manually launching jobs on the cluster, then launching the Spark services when the job is ready. Furthermore, although this work and the distributed RDataFrame tool in general, are framed in the context of Python-based analysis workflows, the Spark library depends on Java in order to work, thus includes extra dependencies that need to be accounted for by the user. Dask on its side is a pure Python library and thus feels more natural in the workflow of a Python user. There is thus a larger overhead with choosing Spark over Dask for a user that wants to start their analysis workflow for the first time, and it will be shown with more details in Section 4.3.2.

4.3 Scaling distributed RDataFrame analysis to thousands of cores

In this section, the performance of the two available backends for distributed RDataFrame is tested on a computing cluster at CERN, running a physics analysis example with different configurations. First, the analysis is run on a single node with varying number of cores, in particular using only the Dask backend as a first baseline. The processing throughput per core is compared against the processing throughput of running the same analysis with RDataFrame in sequential execution. Later, the Dask backend is compared against the Spark backend running the same analysis on a larger dataset. By fixing the number of tasks executed by the two backends and the granularity of those tasks, the aim was to compare their scalability and investigate whether they would introduce any noticeable overhead with respect to each other.

The physics analysis used in this work processes data from events recorded by the CMS experiment at CERN in 2012. The analysis extracts the di-muon mass spectrum by computing the invariant mass of muon particles with opposite charge in the dataset. The result of the analysis is a histogram of the mass spectrum, showing peaks highlighting the presence of different particles in the physics events. This application is available

through the CERN open data portal [123] and is also called “dimuon benchmark” from now on.

The original application was implemented using RDataFrame to be executed on a single machine. In this work, it was adapted to run with distributed RDataFrame. The needed setup is minimal, similarly to Figure 3.1, leaving the calls to the RDataFrame API completely unmodified.

In the two different test configurations mentioned above, the original dataset is replicated in order to reach a higher size providing a more realistic computational workload. In the first configuration that runs the application on one node only, the dataset is replicated fifty times, whereas on the second configuration it is replicated four thousand times. The final dataset sizes for both configurations are reported in Table 4.1. Values in the table may present small rounding adjustments, the original dataset contains exactly 61 540 413 entries and its size is 2 244 449 133 bytes.

Replicating a ROOT dataset can be easily done by providing multiple times the path to a ROOT file to RDataFrame. Internally, the entries of the various files will be chained together and RDataFrame will be able to process them as a single coherent entity. This practice is valid for benchmarking purposes, since the physics events are statistically independent. In a first round of tests, the dataset is made available locally on each computing node, in order to factor out I/O performance from the results and read directly from filesystem cache. Subsequently, the dataset will be read remotely from its storage location at the CERN data center, providing a more concrete example of what physicists may experience.

TABLE 4.1: Dataset sizes in the proposed experiments. First row: original dataset. Second row: dataset used when testing the Dask backend on a single node. Third row: dataset used when comparing the Dask backend against the Spark backend on many nodes of the cluster.

Dataset	# Files	# Entries [K]	Size [MB]
Original	1	61 540	2 224
1st configuration	50	3 077 020	111 222
2nd configuration	4 000	246 161 652	8 897 796

4.3.1 Hardware setup

Resources from an HPC cluster at CERN are used for these tests. The computing nodes have the following characteristics:

1. 2x AMD EPYC 7302 16-Core Processor (total of 32 physical cores, no hyper-threading).
2. 512GB DDR4 3200Mhz memory.
3. Infiniband 100 Gbps network card.
4. Samsung PCI-e NVME SSD.

This cluster relies on the Slurm framework to manage hardware resources.

4.3.2 Methodology

In the following tests, the time to plot (as defined in Section 1.6) is measured in each test between the beginning and the end of the RDataFrame computations, irrespective of whether they are run locally or distributedly. The processing throughput of an application is then computed by dividing the size of the data that is actually read and processed in the benchmark by the corresponding time to plot. The considered analysis processes all columns and all entries of the input dataset.

Single node test with Dask

The original application is run on one of the nodes of the computing cluster, to get a first baseline measurement of the processing throughput on a single core. Then, it is converted to run with Dask, using a single computing node and increasing the number of cores used in that node. Details of the connection to Dask are specified in the next section. Furthermore, the number of chunks in which the dataset is split is also increased. Namely, 1, 2 and 4 chunks per core are tested.

Tests comparing Dask and Spark backends

Before actually running the tests with Spark, the needed resources must be requested to Slurm. When they are granted, the Spark scheduler and all the worker services on the computing nodes are started with a bash script that was provided in the Slurm request. Only after the Spark cluster is available, the Python application with distributed RDataFrame code is started. Thus, there is no direct way to start the Spark cluster by connecting to the batch system in the user application. The extra code needed by the Spark version of the benchmark is shown in Listing 6. In this case, the object that represents the connection to the cluster is called `SparkContext`. Its configuration options can be defined in the `SparkConf`, as shown in lines 8-14 in the listing. In this case, they just mirror the same resource configuration that was used to launch the services through the bash script mentioned above.

A different approach is shown in Listing 7 with the Dask setup. In this case, the object representing the connection to the cluster is a Dask `Client`. The cluster resources can be programmatically defined in the application itself, creating a cluster object from the types supported by Dask. In this work, the `SLURMCluster` class was chosen to connect to the cluster. It allows to automatically submit request for resources to Slurm, without the need to invoke a separate bash script, and shows the more user-friendly approach that Dask enables. This enables a direct interface with the resource manager, providing a different, more ergonomic approach for users with respect to what was described for Spark. The relevant steps needed for this setup are highlighted in Listing 7:

- Lines 6-14: The class `SLURMCluster` expects the information needed to create a job, which corresponds to the resources that it should request to the Slurm manager for every node. These include: the number of cores; the number of Dask worker processes, which is set to the amount of cores per node; the name of the Slurm queue where the job should be submitted; a few extra options that make sure the job will gain exclusive access to the node.

- Line 15: Calling the `scale` method on the created object will launch as many jobs as the number provided as argument. For these benchmarks, each job adds

```python
from pyspark import SparkConf
from pyspark import SparkContext
def main_spark(
  master, n_nodes, cores_per_node, dataset):

  total_cores = n_nodes * cores_per_node

  sconf = SparkConf().setAll([
    ("spark.master", f"spark://{master}:7077"),
    ("spark.executor.instances", n_nodes),
    ("spark.executor.cores", cores_per_node),
    ("spark.cores.max", total_cores),
  ])

  scontext = SparkContext(conf=sconf)

  run_analysis(scontext, dataset)
```

LISTING 6: Setup function of a Spark benchmark. The analysis receives the created `SparkContext` object to distribute the application on the cluster.

one more node to the reservation, and each node runs one (multiprocessed) Dask worker.

- Line 18: Before starting the RDataFrame analysis, the Dask client waits for all the Slurm jobs to be started and their corresponding Dask workers to be ready. This is done to have consistent time to plot measurements for the purposes of the benchmarks. In a real scenario, the application would start as soon as at least one job is ready, in order to minimise the waiting time for the user.

The benchmarks are executed both with the Dask backend and the Spark backend of distributed RDataFrame. Each benchmark is repeated ten times, using from one to 64 computing nodes to distribute the computations. 32 distinct processes are run

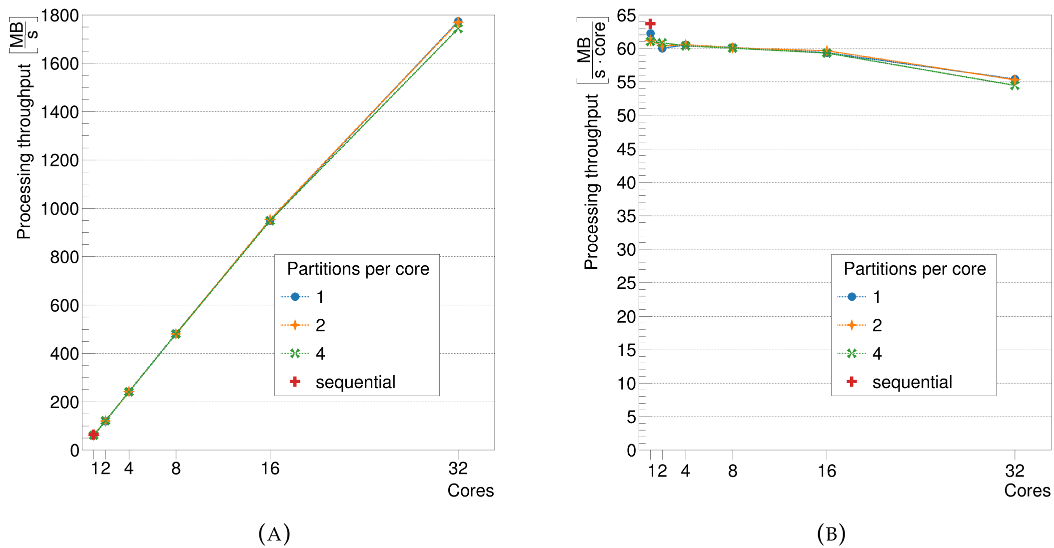
```python
from dask.distributed import Client
from dask_jobqueue import SLURMCluster
def main_dask(
  n_nodes, cores_per_node, dataset):

  cluster = SLURMCluster(
    cores=cores_per_node,
    processes=cores_per_node,
    queue=QUEUENAME,
    job_extra_directives=[
      "--exclusive",
      "--ntasks-per-node=1"
    ]
  )
  cluster.scale(n_nodes)

  dask_client = Client(cluster)
  dask_client.wait_for_workers(n_nodes)

  run_analysis(dask_client, dataset)
```

LISTING 7: Setup function of a Dask benchmark. The analysis receives the created `Client` object to distribute the application on the cluster.

concurrently on each node (one process per core). In Spark, this is handled through a Java Virtual Machine (JVM) which in turn spawns Python processes, whereas Dask spawns a different Python process for each core. For each test, the number of chunks in which the dataset is split is four times the number of cores used for that particular test. Source code of the tests discussed in this section is publicly available on GitHub [124].

4.3.3 Results

The results of the tests done on one computing node are shown in Figure 4.2. The lines in the figure refer to benchmarks of distributed RDataFrame with the Dask backend, using a variable number of cores of the node and also an increasing number of tasks per core. Each task processes a separate partition of the original dataset. The figures also report the result of the original application, which uses the traditional RDataFrame version processing the dataset sequentially.



(A) (B)

FIGURE 4.2: Processing throughput achieved on a single computing node, with increasing number of cores and tasks per core. Each task processes a separate partition of the original dataset. In each plot, three lines indicate the results of increasing the number of cores used in the benchmark. Each line corresponds to a different number of partitions per core. The result of running the original analysis sequentially is also indicated at the 1 core data point of the x axis. (a): Processing throughput expressed in Megabytes per second. (b): Processing throughput normalised by the number of cores.

In Figure 4.2a, the processing throughput is reported in terms of Megabytes of data processed per second. Generally, changing the number of partitions per core doesn't

affect the overall throughput of the benchmark, with the same number of cores. This is a positive result because it means that the Dask backend is able to handle the tested task granularities with no noticeable overhead, even if more tasks naturally ivolve more runtime. For the granularities tested in this case, the backend is able to properly balance the load.

In Figure 4.2b, the throughput shown in the previous figure is normalised by the number of cores used in each benchmark run. Overall, the highest throughput per core is achieved by the benchmark running the original analysis with RDataFrame sequentially. This can be explained easily since there is no overhead in initialising the Dask runtime and no extra communication and serialisation of data between scheduler and worker processes, so it runs slightly faster. It should be noted that the benchmark using the Dask backend and running with one core and one partition has a slightly higher throughput than the benchmarks using one core and two or four partitions. Since there is only one processing core, it can be expected that having more than one task brings some overhead.

When more cores are used and there is more than one task per core, different tasks can be sent to those cores that are free. Potentially, this can be very beneficial when the dataset is imbalanced (different files having very different number of entries) or when it is read remotely and network I/O becomes an issue.

Varying the number of partitions per core leads to the same throughput when considering the same amount of cores. The throughput per core shows an almost flat line with respect to the number of cores. The lowest point is reached when all 32 cores of the node are used at the same time. Since the node is being fully utilised, the processing tasks are influenced by other processes happening like I/O, Dask internal communication threads. Nonetheless, the drop in normalised throughput between using one core and 32 cores is around 10%, while achieving a 28 times higher nominal throughput.

Figure 4.3 shows the results of running the tests on multiple computing nodes, comparing the Spark and Dask backends. On each node, 32 concurrent processes are run. In the figure, the left column shows the time to plot of the first run of the benchmark at each node count, whereas the right column shows the average time to plot of the following runs at each node count (error bars are not visible because they are too small with respect to the scale of the y axis). The figure presents three distinct rows which,

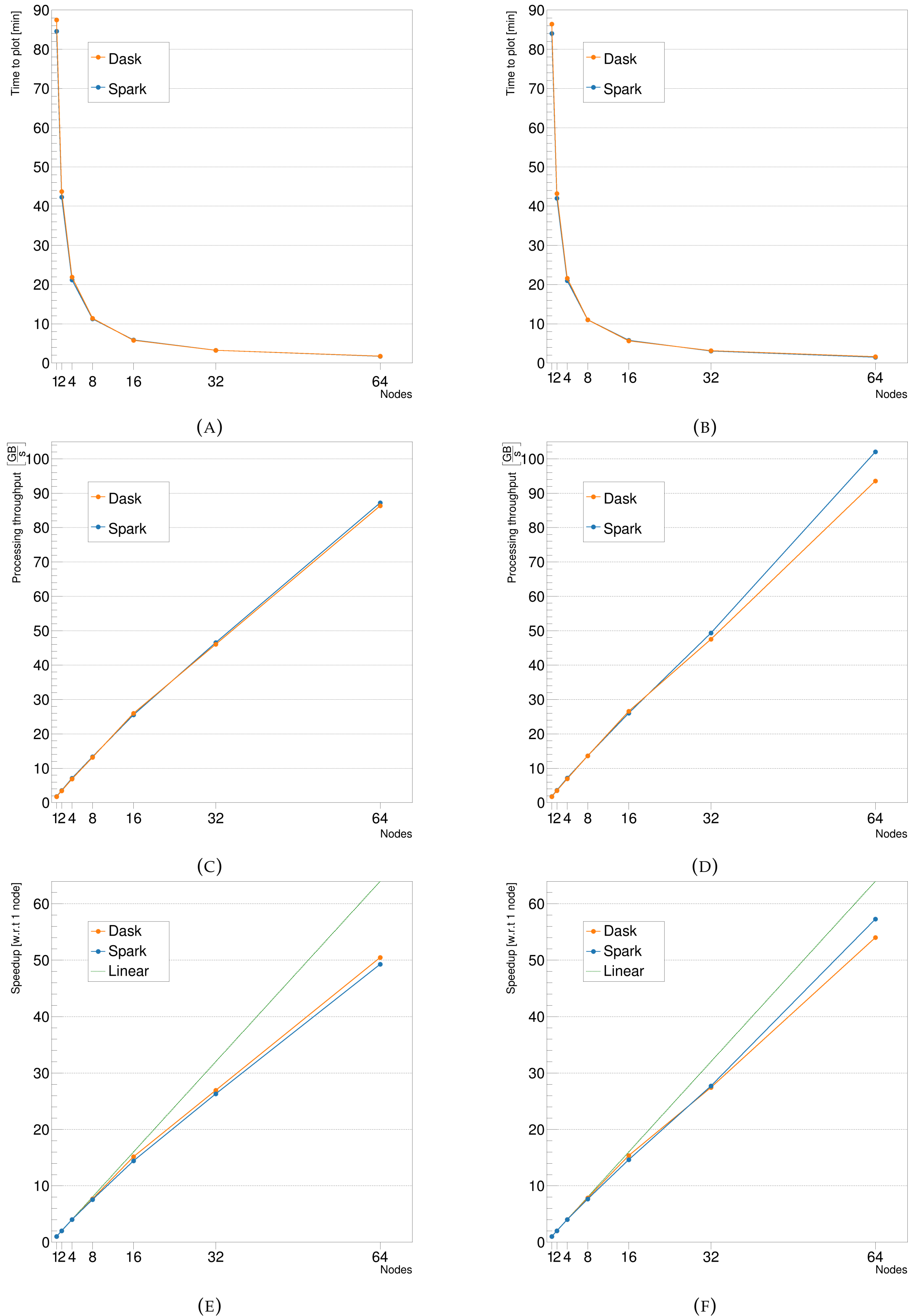


FIGURE 4.3: Benchmark results with Dask or Spark backend on multiple nodes (32 processes per node). Left column: first run of the benchmark. Right column: average of consecutive runs. First row: total runtime of the benchmark. Second row: processing throughput. Third row: speedup relative to the result obtained with one node.

from top to bottom, show the following metrics: time to plot (in minutes); processing throughput (in Gigabytes per second); speedup with respect to the result on one node. The time to plot experienced by the user continuously decreases with the increased number of nodes, from slightly less than 90 minutes to less than 2 minutes. As far as the time to plot is concerned, there is no appreciable overhead in the first run of the benchmark with respect to following runs. The overhead of the first run can be appreciated when considering the processing throughput results. The peak processing throughput achieved in the first run is 87 GB/s, whereas in following runs a peak of 102 GB/s is reached when using the Spark backend. The Dask backend can still achieve a very high throughput, although slightly lower than the Spark backend. The speedup in the first run shows a change of slope after the 16 nodes count and stays further away from the linear line. Consecutive runs show a better trend for both backends, closer to a linear behaviour with respect to the number of nodes used.

The results for the next configuration of this benchmark, reading data from remote CERN storage facilities rather than locally on the nodes, are shown in Figure 4.4. In this case, only the runs with the Dask backend are shown, so that the comparison with Spark is not influenced by variability due to the remote I/O. In fact, this image shows the distributions of the time to plot measurements with each different number of nodes.

4.3.4 Discussion

All these results show that the new Dask backend performs on par with the Spark backend. In general, the distributed RDataFrame tool can parallelise well even when thousands of computing cores are used.

It should be noted that, from the comparison of the time to plot measurements with the processing throughput of Figures 4.3, the experience for the user does not change significantly between the first run and consecutive runs. The highest difference in time to plot is present when using only one node and it is just less than two minutes with respect to an overall runtime of almost ninety minutes. The overhead present in the first run becomes more evident only when discussing the throughput and trying to lower it becomes important in the effort to best utilise cluster resources.

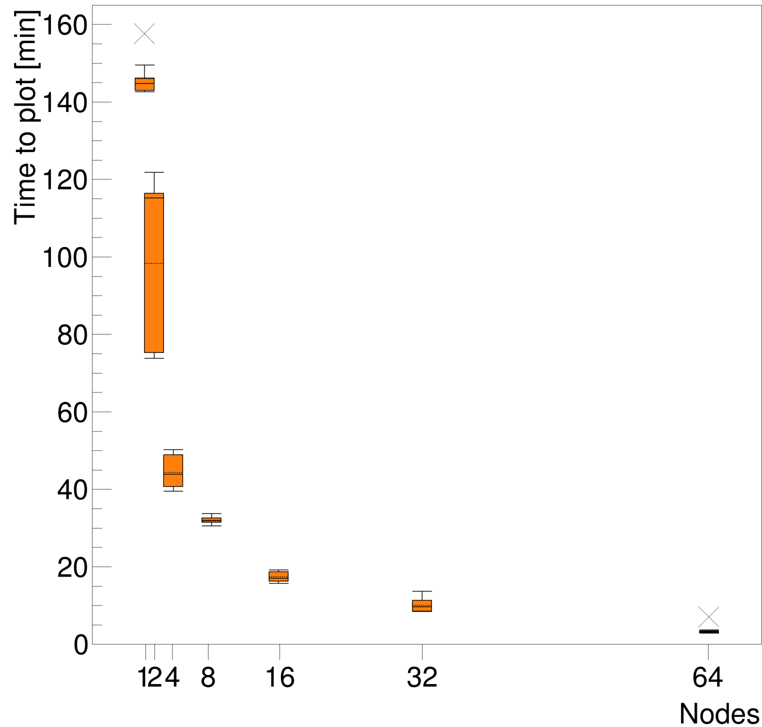


FIGURE 4.4: Time to plot (in minutes) achieved with an increasing number of nodes. 32 concurrent processes are run on each node. Data processed in the experiments are read remotely from the storage facilities at CERN. A box plot of the distribution of results is shown for each point on the x axis, ten runs per node count. The box spans from the first to the third quartile. Inside the box, a dashed line represents the mean, a solid line represents the median. "x" marks at 1 and 64 nodes represent outliers, those points that are outside the range of the whiskers. This range is defined as the distance between the first quartile and the third (also called interquartile range) multiplied by 1.5.

The second row in Figure 4.3 clearly shows that the first run of the benchmark has a lower throughput than consecutive runs. This is mainly due to the necessary startup routines performed by ROOT and its C++ interpreter. Furthermore, when using RDataFrame with the Python bindings, users still run their C++ functions by passing them as

Python strings to the RDataFrame API functions. These strings then need to be just-in-time (JIT) compiled by the ROOT interpreter. This operation, also called JITting, incurs in initialisation cost the first time it is done in a certain Python process and will cache some information internally for later use. Notably, most of the functions, C++ template instantiations, constant structures that belong to the computation graph. Since the execution of the distributed application is done with multiprocessing on each computing node, this means that the initialisation cost is incurred once per core used in the benchmark. Considering that in the interactive distributed data analysis use case several application runs happen on the same resources, each with slightly different parameters as needed by the user, this "cold run effect" is not a large bottleneck for the general user workflow.

The benchmark runs following the first run, presented in the right column of Figure 4.3, show a better throughput and speedup behaviour overall. Although the throughput achieved in this case is very high at around 100 GB/s with both backends, some non-idealities are still present and become more evident after the 32 nodes mark. This is due to the fact that even though the interpreter has been already initialised at this point, some JITting is still involved in each task. A clear improvement on this side could be brought by creating and compiling the RDataFrame workflow in a certain process when the first task starts, then caching it in memory and reusing it in subsequent task happening in the same process.

The distributions in Figure 4.4 are much wider than in the previous cases, sometimes also showing outliers (as indicated by the "x" marks at 1 and 64 nodes). This large variance could be helped by an even finer grained task distribution on the application side, but it is also influenced by the status of the network and possibly the implementation of the remote I/O. It is thus left for future studies directed at this specific issue.

Both in the first run and in consecutive runs, the amount of JITting is proportional to the number of tasks. This also raises the question of finding a good balance in parallelising the processing of the input dataset. On the one hand, creating more tasks means splitting the dataset in smaller chunks, thus leaving more room for the scheduler to assign work to the computing nodes, avoiding possible imbalances due to some nodes being slightly slower than others or some parts of the dataset requiring higher

computational load than others. On the other hand, creating more tasks leads to more overhead in spawning them and recreating the computation graph on-the-fly.

The slopes of the speedup lines shown with both backends do not diverge greatly from the linear behaviour shown for reference in the third row of Figure 4.3. Although it might be possible in principle to reach this behaviour when using multiple nodes, the non-idealities discussed in scaling the computation graph justify the missing performance gains. It should be especially noted that the overhead discussed is per-task, not per-core or per-node. This means that the more tasks, the more overhead on the whole runtime of the distributed execution. This fact clashes with the intuition that splitting the input dataset in more chunks should allow for a more fine-grained scheduling by the execution engine, thus avoiding potential slowdowns due to some tasks or nodes taking longer than others. In general, optimising dataset splitting is a non-trivial problem of distributed execution engines, with sparse literature suggesting different ways to choose the splitting value. For example, the Spark documentation suggests setting it to 2 or 3 times the amount of available CPUs in the cluster [125]. But other strategies exist depending on the number of files and size of the input dataset and the available resources [126] or even on the choice of the number of chunks being done statically or dynamically [127]. The results discussed previously refer to benchmarks with four tasks per computing core, which, for the purposes of this work, seems to strike a good balance between making the load even on the nodes and avoiding too much overhead.

4.4 Example of full-scale distributed RDataFrame analysis on HEP grid resources

Once the investigation on horizontal scaling axis of distributed RDataFrame execution has been taken into account, it is interesting to focus on the axis of the computational complexity of the analysis at hand. This section describes the main highlights of a collaboration with a research team affiliated with the CMS experiment at the LHC and the Italian National Institute for Nuclear Physics (INFN). In this context, the distributed RDataFrame tool was used for the first time to drive a real, production-level analysis

of Run 2 CMS data on INFN computing resources using the Dask backend to leverage the existing HTCondor resource manager. The analysis was run on a prototype analysis facility, an important topic within the R&D for the future computing model of CMS analysis. For this work, my contributions included the developments of new features for distributed RDataFrame that were not previously present (see Section 4.4.1), together with the supervision of the performance benchmarks executed. The continuous exchange of ideas and the constructive feedback loop in the software development with the external research group, both with the infrastructure administrators and the physicists programming the analysis, is a prime example of how the research done for this thesis really fits within a dynamic community with practical problems to address.

4.4.1 New RDataFrame developments

This section reports the developments that were brought to the distributed RDataFrame tool for the first time in the context of this work, in order to address some shortcomings that were identified by the users in their first approach with distributed RDataFrame.

Systematic variations in distributed mode Introducing the systematic variations API in the distributed mode leveraged the machinery developed for other types of actions described in Section 3.8. In this case, the function `VariationsFor` of the RDataFrame API is called, within the execution of a mapper, on the quantity for which the user requested to compute systematic variations. The object returned from the mapper in this case (an instance of type `RMergeableVariations`) will represent a collection of the varied results for the corresponding action. The implementation of the `Merge` method will just call the same method on all the stored results, both the nominal and the varied ones. The user will be then presented with the collection obtained after all the merge steps, following the usual distributed procedure of RDataFrame and not impacting the user experience in any way.

Distributing multiple computation graphs concurrently The advantage of seamlessly distributing an RDataFrame computation graph removes many of the responsibilities that were previously on users' shoulders. Nonetheless, every time a result

of an action is retrieved, it will launch the distributed execution of that particular RDataFrame graph, blocking the application until the results are back. There are cases where within the same analysis, more than one RDataFrame is needed to process different sets of files with different operations, depending on the nature of the contents of the different datasets. For such cases, whenever one distributed RDataFrame starts processing, it blocks the others.

To overcome this, users are offered the possibility to submit multiple different computations graph for distributed execution, concurrently within their application. This can be done through the `RunGraphs` function, which is briefly described in Listing 8.

```
1  from concurrent.futures import ThreadPoolExecutor, wait
2
3  def RunGraphs(actions):
4    unique_graphs = {}
5    for action in actions:
6      unique_graphs[action.get_headnode()] = action
7
8    with ThreadPoolExecutor(max_workers=len(unique_graphs)) as executor:
9      futures = []
10     for graph in unique_graphs.values():
11       futures.append(executor.submit(execute_graph, graph))
12
13     wait(futures)
```

LISTING 8: Implementation of the submission of multiple computation graphs concurrently.

In the listing, the following steps can be identified:

- Line 3: the function takes as input a list of actions that the user may provide. The actions may belong to different RDataFrame graphs and can be of any type. For example, a user may ask to submit a `Count` action to count entries of a certain dataset and a `Histo1D` action to create a histogram of a column from another different dataset.

- Lines 4-6: the list of actions is iterated to retrieve the corresponding head node. The head node becomes the key of a dictionary and the action its corresponding value. Using the head nodes as keys guarantees that only unique, different computation graphs will be stored in the dictionary.
- Lines 8-11: the unique computation graphs are submitted for distributed execution. On the client side, a Python thread pool is created with as many threads as unique computation graphs. Every graph is submitted to a different thread of the pool. The operation of sending the trigger to the respective backend (Spark, Dask or others) is non-blocking and not computationally intensive, so using Python threads is a safe choice in this case.
- Line 13: the different executions are waited, so that the user will get all the results once they are ready.

Dealing with empty files in the processing pipeline The analysis step is one of the last parts in a long chain of operations involved in the HEP data processing pipeline. Very often, physicists that approach the analysis of a dataset get the input files from some central database centrally managed by their experiment. These files may in turn be the result of some preliminary manipulation of the larger data formats. Thus, the amount of data and the number of entries in each file is not always known a priori by the final users (and it is not something they should need to know anyway). Still, this introduces a particular element of uncertainty in the distributed execution. It may in fact happen that, when a task running in some computing node opens a file, the dataset inside that particular file is actually empty. This may be the result of a very strict selection step which took place before the user started their analysis and some files were completely emptied by this procedure.

The issue to address then is how to construct the logic of each task in distributed RDataFrame such that it is resilient to this type of files, which are valid since they are physically present in some storage facility but still useless for the purposes of the analysis since they do not contain any entry. This is addressed on two sides: when retrieving the chunk of the dataset to be processed on the task and during the execution of the mapper and reducer functions. The algorithm of conversion from an approximate task

to a concrete task shown in Listing 3 now also keeps track of the accumulated number of entries that the task would process from all files assigned to it. When the algorithm would create a task with no entries to process at all, it instead returns a null value. A mapper that receives such empty value knows that it should transmit it as-is without running the computation graph. The reducer is then implemented such that if it receives a null value, it will not try to merge it with the other input result but it will return the non-empty input directly, or will just return another null value if both inputs are null.

This change turns what would have been previously a nuisance in the user workflow, the fact of tracking which files are effectively empty and removing them from the input list of files, into a completely transparent procedure.

4.4.2 Experiments

This effort targeted the analysis of two same-sign W bosons decaying into an hadronically-decaying τ lepton and a light lepton (μ or e), an example of the phenomenon called Vector Boson Scattering (VBS) [128]. Being an analysis already run in production by CMS physicists, this takes two main steps to process, which include the elements introduced in Section 1.4:

- Preselection: in this first step, all the information regarding the particles involved in the analysis is filtered with loose selection requirements. The operations involved in this part mainly relate to defining new quantities that will be used in the next stage of the analysis. The output of this stage is a new dataset containing only the events that passed the selection, with a set of columns made of a subset of the original columns together with the newly defined columns.
- Postselection: the analysis then proceeds by reading the output from the previous step and running the full set of complex operations needed. These notably include systematic variations that are applied to many of the quantities relative to the particles involved in the decay. The output of the postselection is a series of histograms, that can be both visualised and further processed in a separate statistical analysis workflow.

Roughly around 1 Terabyte of simulated data was created per each year of the LHC Run 2. The benchmarks performed in this work process a dataset with the following characteristics:

- Size: 1.1 Terabyte, corresponding to a simulation campaign relative to the year 2017.
- Number of events: around seven hundred million.
- Number of files: 1274.

Methodology

The software implementation of this analysis currently used in production will be also referred to as the "legacy" approach from now on. For the preselection step, it makes use solely of the nanoAOD-tools Python package. The postselection step is instead composed of a mix of nanoAOD-tools utility functions and a series of ad hoc Python scripts using PyROOT to define the physics quantities and plot the final results. Both steps are parallelised by manually submitting HTCondor jobs that take different parts of the total dataset as input. Also the logical splitting of the dataset is done manually, taking one file as input per each job. The postselection stage also includes procedures to merge the outputs from the different jobs.

The re-implementation of the analysis developed in this work makes instead full use of distributed RDataFrame for both the preselection and the postselection steps. Some Python utility functions previously used in the legacy approach were converted to C++ functions that could be easily distributed and declared to the various tasks on the distributed nodes running the RDataFrame C++ process. Porting the analysis to distributed RDataFrame also meant that the physicists were able to connect to the HTCondor resources ergonomically via a UI as explained in Section 4.4.2. Splitting the input dataset and merging the results from the jobs is automatically taken care of by the tool, without user intervention.

Hardware setup

The benchmark setup is part of the distributed analysis facility deployed by INFN over its CMS Tier 2 grid sites. To allow for a fair comparison all the tests were performed on the same hardware at the same site at Legnaro (IT). The federated facility uses HTCondor as overaly resource manager. Physicists would access the cluster via a Jupyter-based web interface, where they could create an instance of a Dask cluster that would automatically request resources to HTCondor under the hood. In particular, three computing nodes were available with exclusive access for the purposes of the benchmarks, each with the following characteristics:

- CPU: two Intel Xeon E5-2640 v3 @2.60 GHz, 8C/16T each.
- RAM: 128 GB.
- Storage: 1 TB spinning disk.
- Network: one ethernet controller Broadcom BCM5720, 1 Gb/s.

The input files for all the benchmarks are stored on different machines within the same grid site, so they will be read remotely via the XRootD protocol at runtime. The Dask cluster configuration is such that the scheduler will reserve 4 CPU cores for itself, so that of the total 96 cores available on the three nodes, 92 will be actually processing the analysis computations.

Results

The following results are relative to running only the preselection step of the physics analysis, on the computing nodes just described. Currently, the full set of results is being gathered in a separate document that will be submitted for approval to the CMS collaboration, then for publication in a journal. This section presents only a few key results related to the computing performance of RDataFrame.

The first interesting result to report is the overall runtime, for both the legacy approach and the new RDataFrame approach. The total runtime for an application run

is computed in slightly different ways for both approaches. For the RDataFrame approach, it is simply measured as the time between starting the Python application and its end. Thus, it includes all the time it takes to generate the distributed RDataFrame computation graph and tasks locally, submitting them to the cluster, executing the mappers and the reducers and sending the results back to the user. This approach employs three tasks per core available on the clsuter, for a total of 276 tasks. For the legacy approach, the users measure the starting time of the HTCondor job that starts first and the ending time of the job that ends last, then take their difference as the total runtime. This time does not include any part of the user application running locally before submitting the jobs and it does not include the merging step. The latter is not strictly relevant in this case, since the output of preselection step is saving the modified dataset from each job in some remote paths and there are no results that actually need merging. This approach runs one job per file of the dataset, for a total of 1274 jobs. Table 4.2 reports that RDataFrame obtains a tangible speedup, roughly eight times faster than the framework currently in production.

TABLE 4.2: Runtime of the analysis for the two different approaches.

Benchmark	Average [m]	Error [m]
Legacy	182.0	1.0
Distributed RDataFrame	23.8	0.6

During the execution of the distributed analysis, each task is monitored at intervals of one second to gather information about the resource usage on the nodes. In particular, Figure 4.5 shows the CPU usage in percentage for a single mapper task running the preselection step of the analysis. It can be seen that for roughly the first two minutes of execution, the CPU usage is consistently very high, above 90%. Afterwards, it begins oscillating widely.

For the same task, Figure 4.6 shows the corresponding memory usage in Megabytes. The increase in memory as the task continues is a sign of the clusters of entries being read into memory and processed. The profile of the curve is flat for most of the runtime.

Finally, Figure 4.7 shows the I/O throughput in Megabytes per second obtained

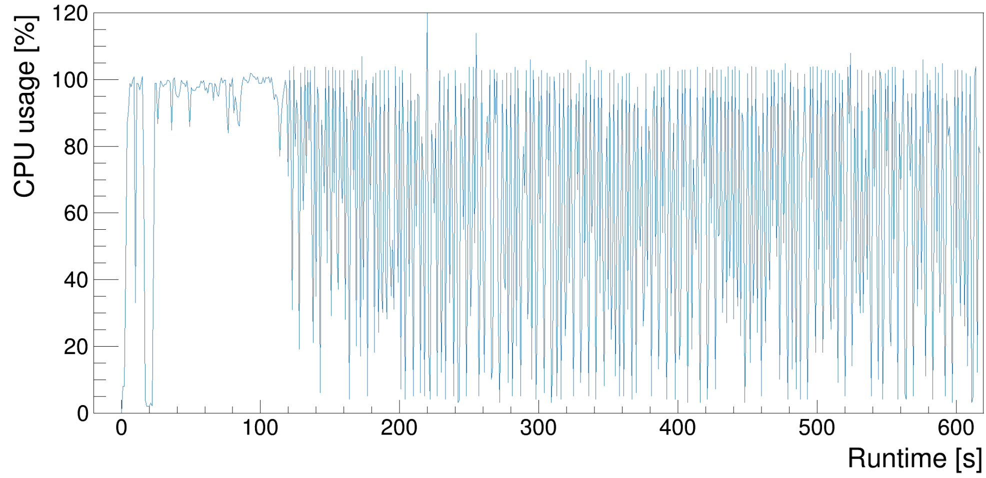


FIGURE 4.5: CPU usage (in percentage) while executing one mapper task of the distributed RDataFrame analysis.

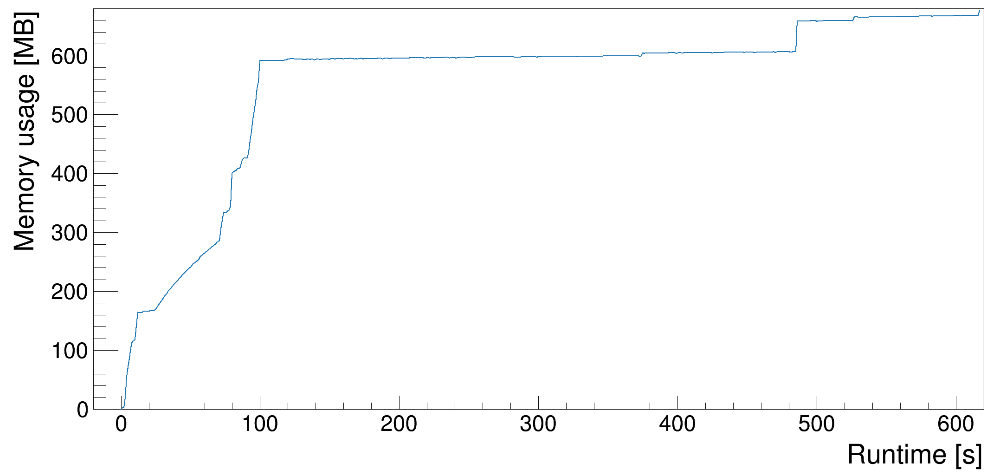


FIGURE 4.6: Memory usage (in Megabytes) while executing one mapper task of the distributed RDataFrame analysis.

from the network card of the node that was running the same mapper task of the previous two figures. In the image, the runtime on the x axis corresponds to the same runtime of the mapper. The throughput is instead relative to the whole computing

node. It can be seen that for roughly the first two minutes of execution, the network I/O was oscillating, with peaks of around 60 MB/s. Afterwards, the node sustains an almost continuous throughput of 120 MB/s until the end of the task.

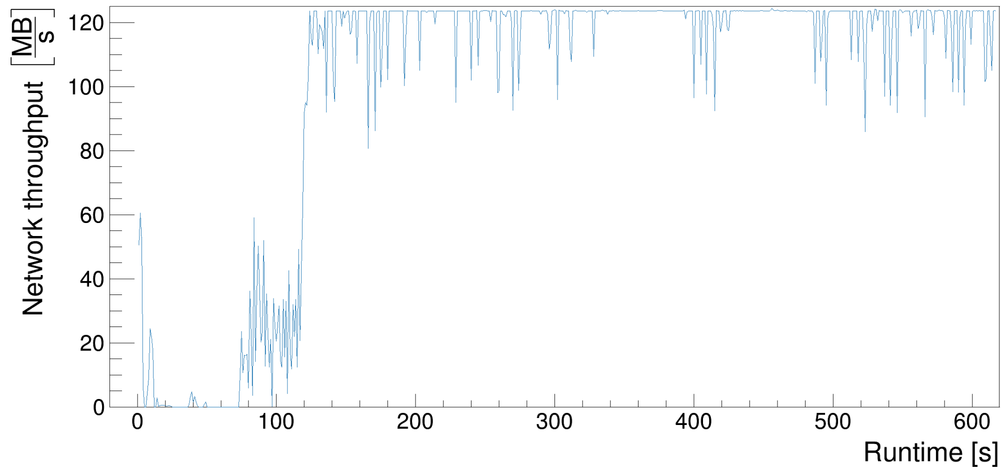


FIGURE 4.7: Network read throughput (in Megabytes per second) sustained by one computing node that was executing the distributed RDataFrame analysis. The duration on the x axis corresponds to the same runtime of the mapper task of Figures 4.5 and 4.6.

Discussion

The result shown in Table 4.2 demonstrates that distributed RDataFrame can provide a tangible benefit for physics users even in production environments. While developing this work, the INFN team put careful considerations into optimising the legacy analysis to avoid certain old patterns that were providing sensible bottlenecks, for example by removing unnecessary operations that were present in the legacy workflow. Even with the optimised legacy approach, RDataFrame can still provide an order of magnitude speedup. This is due to various factors, including for example the improved dataset splitting and job scheduling and the more lightweight environment, compared to nanoAOD-tools, that doesn't need to download external Python modules at runtime.

Furthermore, the images shown in the previous section suggest that the new approach with RDataFrame could deliver even better results. The wide variability in CPU usage seen after two minutes of execution in Figure 4.5 until the end of the task corresponds to the same time period where the node is sustaining a constant 120 MB/s in throughput from the network shown in Figure 4.7. This value corresponds to the nominal throughput of the network interface on the node, namely 1 Gb/s. This is due to many more tasks being running at the same time as the one that is shown in the images. Thus, the distributed RDataFrame execution is capable of saturating the I/O capabilities of the node. This brings a concrete insight useful for future computing models, that is analysis facilities should make sure to have at least a 10 Gb/s interface on each computing node and the storage nodes should be able to serve the aggregated nominal throughput of all the computing nodes at any given time. The memory usage shown in Figure 4.6 is well under 1 GB, thus leaving enough memory for even more complex analyses and other services of the analysis facility that may need to run in the background.

4.5 Conclusions

This section has demonstrated the scaling of distributed RDataFrame on many cores, many nodes scenarios, both with opendata benchmarks and real, production-grade analyses.

Initial tests were run on a single node of a computing cluster at CERN, to compare the throughput obtained running sequentially against the throughput obtained using multiple cores of the node. This was also normalised to the number of cores used in each run, showing that the extra work in scheduling the tasks does not account for a high drop in throughput. Also, it was demonstrated that the dataset can be split in many partitions without significant loss in performance, opening the door to heavier workloads involving remote I/O where the finer granularity may lead to improved balancing.

The comparison between Dask and Spark running on up to 2048 cores showed very high raw processing throughput values (more than 100 GB/s with the highest core count) and good scaling, with non-idealities showing up after the 512 cores mark. Both

backends perform similarly, with Dask having a slight disadvantage when more than 1000 cores are used, which can be due to being a less mature framework than Spark in the data science ecosystem. Overall, this shows the advantage of using a modular approach in the design of the backend system, that allows users to run their analysis with no changes on different types of infrastructure deployments expecting always a consistent performance.

Running a full-scale Run 2 CMS analysis demonstrated similarly good performance. The results shown in Section 4.4 bring a concrete benefit to users who port their existing analysis logic to the modern approach offered by distributed RDataFrame, with speedup of at least an order of magnitude. The Dask backend was used to seamlessly scale the physics analysis to a cluster of HTCondor resources, thus enabling new workflows that leverage existing infrastructure with no extra cost and a much smoother user experience. That work could make use of a subset of all the nodes of the grid site, being mostly an R&D effort, but the available hardware was fully saturated during the runtime of the RDataFrame version of the analysis, so that it would be possible to see even better performance runnning on more nodes or switching to better network interfaces.

Chapter 5

Fine-grained caching of physics data

The use cases provided by HEP data analysis workflows involve large amounts of I/O transactions, as described in Section 1.4. This characteristic, together with the fact that the big physics datasets are usually stored far from the nodes responsible for their processing, means any analysis can be bottlenecked by poor scheduling of reading operations or slow network connections. Sometimes, this limitation is mitigated by employing data management systems, that should be able to schedule analysis execution closer to the location of the datasets, but they are not always available to final users.

One common strategy to address this type of situation is through caching. The topic of caching is very broad and clearly it has been studied also within the context of LHC experiments (see Section 5.1 for the literature review). This chapter will highlight two studies that were carried out in this thesis. Initially, the current scenario was examined, evaluating existing tools and finding two different caching strategies detailed in Section 5.3. A series of tests was run comparing these two strategies, as shown in Section 5.4. Another study, detailed in Section 5.5, explores a different paradigm based on object stores with bleeding-edge libraries that can potentially introduce new caching systems for HEP.

5.1 State of the art

Early examples of large-scale data caching strategies originate from the needs provided by the advent of the Internet: distributed systems with a high number of servers with

requests coming from distant parts of the world needed some type of coordination to increase access speed to web resources [129, 130]. With the increased usage of distributed execution engines such as Spark, also the data analysis scenario faced the consequences of reading and processing large datasets over multiple nodes. Spark itself offers utilities to automatically cache the dataset while it is being processed at intermediate steps of the analysis, on user's request. Zhang et al. [131] have analysed the different caching strategies offered by default in Spark (in-memory, on-disk and a mix of both), highlighting the cost introduced by serialisation when caching on-disk. Another study by Jiang et al. [132] addresses the extra cost involved in the unpredictable nature of shuffling dataset chunks across nodes during the reduction phase. Very recently, Uta et al. [133] have proposed to address the shortcomings of traditional Spark dataframes with an indexed dataframe, an abstraction aimed at in-memory caching on the computing nodes with indexing to support lookup of rows for processing.

Clearly, one crucial aspect of caching in distributed computing is the topic of data locality. It would be most desirable that computations happened as close as possible to where data is located, in order to reduce access latency. LHC collaborations such as ATLAS or CMS employ data management systems to keep track of the location of particle physics datasets, so that the submissions to the grid may then use such databases and directly schedule jobs on the sites where datasets are stored. The ATLAS data management system is called Rucio [134], whereas CMS has developed and used PhEDEx [135] in the last two decades, although it is being phased out in favor of Rucio. These data management systems only track the presence of data on some sites, statically. But this poses the question of how and especially why data is stored at a specific location. Supposing that all sites could support a certain amount of storage, it would be beneficial to have datasets already present when jobs arrive, possibly even according to usage patterns. For this reason, the same two collaborations have investigated algorithms and techniques to place data dynamically on different sites by for example identifying popular datasets and creating replicas for them on multiple facilities when many users want to access them. C3PO [136] was developed inside the Rucio data management system, whereas Dynamo [137] targets the CMS use case.

The exploration of integrating the caching systems with the analysis layer is interesting for the field, both because the analysis layer brings precise knowledge on which

columns the user wants to process and because physicists are not always able to rely on some centralised data management system. Similarly to the previously cited works related to Spark, one can imagine that HEP data analysis interfaces are well suited to understand the dataset the user would like to process and possibly act accordingly in terms of caching for subsequent uses. After all, it is often the case that HEP analysts start their workflow with an exploratory phase, inspecting the physics events and deriving a first set of statistics, later repeating the analysis with different configurations. Furthermore, since datasets are often read remotely, it could be useful to hook directly into the remote transactions and stage data as it is being read for further processing. This is precisely the functionality offered by the caching mechanism in XRootD. A block-based mechanism to cache files on disk can be automatically enabled, so that incoming XRootD requests are staged to disk while they are being served to the client application. As a first step, this can be set independently on any machine that should cache data [138]. Furthermore, it can also be used to coordinate multiple caching nodes forming a federated system (a technology called XCache commonly used in grid sites [139, 140]). Although this caching system is common enough in HEP computing environments, not often can it be found used directly by the analysis framework, whereas usually it is activated at the level of the grid site. Many efforts from literature are focused on analysing access patterns of certain datasets and evaluating different strategies to improve network and I/O usage [141, 142, 143].

Another dimension to take into account when discussing storage in the context of HEP analysis is the architecture. Most often, storage layer and caching systems assume a file-based approach, whereas it is not uncommon in HPC and other distributed computing scenarios to see object stores being used for distributed databases and caches. Over the years, various object store implementations have spawned within industry. Notable examples are provided by vendors such as Amazon with the S3 service [144], Microsoft with the Azure Blob Storage [145] and Intel with the Distributed Asynchronous Object Store (DAOS) [146].

There is literature comparing different technologies according to established benchmark suites [147, 148]. In some cases, current knowledge allows to extract the best performance of a given object storage tool, through fine-tuning of user space parameters [149]. This work does not attempt to modify or tune the storage backend; rather,

it focuses on addressing analysis needs from the perspective of the data format and the layer that implements I/O of the data format to various backends. There are other examples of I/O libraries that have attempted an integration of their data format with fast object stores, such as the HDF5 connector for Intel DAOS [150].

Regarding the execution of distributed workflows that exploit object stores, some efforts can be found for industry products [151, 152, 153]. In the cited approaches, the object store is used as scalable storage layer to host big datasets and the computing nodes read data directly from the object store. Furthermore, it is shown that once the object store semantics are leveraged properly, read-intensive analysis workflows can get 3-6 times faster.

Regarding caching large input datasets, very rarely do other investigations highlight the possible benefits that it could have in distributed computing scenarios. In a work that compared different object store engines in geospatial data analysis workflows [107], a part of the benchmark presented the improvements in performance of the different engines with caching. But only the filesystem cache was used in that case, hence data and queries could be kept in the memory of the nodes and no separate caching mechanism was implemented.

In the HEP context, object stores still do not see widespread usage, even in large scale collaborations. Research studies from the early years of the LHC have tried integrating object stores in the grid through the Storage Resource Manager (SRM) interface [154]. This interface allowed accessing and managing storage resources on the grid. In a first effort, a plugin was developed to connect Amazon S3 resources to the grid storage layer [155]. Later on, a tier 2 grid facility of the ATLAS LHC experiment [14] was extended to use the Lustre filesystem [156], with benchmarks showing 8 GB/s of peak read speed [157]; the effort described in this chapter of the thesis is able to achieve a much higher throughput, as will be shown in Section 5.5.5. More recently, the focus has migrated towards the evaluation of such storage solutions on concrete examples of HEP software such as ROOT [158], where a physics dataset was used to evaluate data access patterns over an S3 API, leading to aggregated throughput of a few Gigabytes per second. An interesting example of investigation into data analysis needs can be found in a work by Charbonneau et al. [159], where 8 TB of physics events are stored in a Lustre cluster; this work reveals that although resource scaling helps achieve higher

throughput, remote data access while processing can become a burden for analysts. It is clear from this few examples that the potential of object stores, especially newer approaches that rely on low-latency high-bandwidth systems like DAOS, has not been extensively explored in this field. In fact, even very recent mentions of such systems are still a topic of discussion in internal workshops at CERN [160, 161].

This caching mechanism is explored in the context of this thesis and evaluated against traditional ROOT facilities for file-based caching in Section 5.4. This thesis includes a study of object stores for caching HEP data, detailed in Section 5.5.

5.2 Tools

5.2.1 XRootD

The XRootD framework is a C++-based suite targeting fast, low latency and scalable data access. Generically it can serve any kind of data that can fit in a hierarchical filesystem-like approach, abstracting away from the particular implementation of the data format. The core functionalities are greatly extended by a rich plugin system. It is widely used in High Energy Physics both for its remote I/O protocol and for the suite of data access tools that allow to expose the presence of large physics datasets from the storage facilities to other nodes of the Grid.

One of the most used plugins for XRootD is the Proxy Storage Service (PSS), which implements caching functionalities that can be activated on top of the data access layer. In its simplest form, the Proxy cache can be activated on a single node, which will be able to cache files streamed via the XRootD protocol, in a block-based fashion. This allows for partial caching, staging of the datasets and is independent from the particular data format. The caching system can be extended on multiple nodes to create a federated set of proxies, usually called XCache in the field, that is commonly employed in computing sites of the Grid.

ROOT natively supports reading/writing files from/to remote servers via the XRootD protocol, thanks to a plugin of the TFile class. Whenever a user specifies a path that contains the `root://` prefix, that file will redirect all I/O transactions through the XRootD API.

5.2.2 TFilePrefetch

The ROOT TFilePrefetch class [162], as the name suggests, implements a prefetching mechanism for TFile. It defines a readahead window that will request blocks of entries which are consecutive with respect to the one being processed, making them already available in memory once the application needs them. It spawns a thread that will run the prefetching in parallel with respect to the main execution. The class can also be used as a lightweight caching system, by providing a directory where the blocks of entries are stored on disk on the fly. This feature is optional and can be activated in any ROOT application by setting the appropriate environment variables, namely `TFile.AsyncPrefetching` and `Cache.Directory`.

5.2.3 Intel DAOS

Intel DAOS [146] is a fault-tolerant distributed object store targeting high bandwidth, low latency, and high I/O operations per second (IOPS). DAOS addresses traditional POSIX I/O limitations on two fronts in order to optimise data access. On the one hand, it bypasses kernel I/O scheduling strategies, e.g. coalescing and buffering, that are mostly relevant for high-latency few-IOPS spinning disks. On the other hand, it avoids using the virtual filesystem layer, since the strong consistency model enforced by POSIX is known to be a limiting factor in the scalability of parallel filesystems (an example of such limitations can be found in a work by Liu et al. [147]).

On a DAOS system, there are two different categories of nodes: servers and clients. All data in DAOS is stored on the server nodes. There can be many servers running a Linux daemon that exports local NVMe/SCM storage. This daemon listens on a management interface and several fabric endpoints for bulk data transfers. RDMA is used where available, e.g. over InfiniBand [163] or OmniPath [164] fabrics, to copy data from servers to clients. The client nodes are the ones responsible for running computations defined by the users in their applications. A DAOS client node does not store any data on itself, rather it requests the dataset from the servers when it is needed.

The storage is partitioned into pools and containers that can be referenced by means of a Universally Unique IDentifier (UUID) [165]. Objects can be partially read or written

into a container. Each of these objects is a key-value store that is accessed using a 128-bit Object IDentifier (OID). Object data may additionally have redundancy or replication.

5.3 Caching strategies

In a physicist's exploratory analysis workflow, it is common to rerun an application multiple times on the same input data, with slightly modified code. This opens the door to caching the dataset (or better yet the portion of it which is actually processed) during the first run of the user application. This will speed up subsequent runs where the computing nodes can read data from the cache rather than from the remote storage. The caching mechanism should be as transparent as possible for the user, in the sense that it should not modify their workflow or ask them to learn new tools. To this end, it should happen during the first run and asynchronously with respect to the main RDataFrame computation. Through the I/O libraries in ROOT, only the columns and the clusters of entries that are actually processed in the computation graph will be read from the remote storage. Thus, caching systems should try to leverage this behaviour by storing a subset of the input data that is as close as possible to what RDataFrame actually reads, preferably exactly the same amount.

A first milestone set for this part of the work in the thesis was to address already existing tools and evaluate how they can interact with a distributed RDataFrame analysis. In addressing these technologies, two main caching strategies were found: using dedicated cache servers that are close to the computing nodes, or storing the caches in the computing nodes themselves. The following sections will describe them in more detail.

5.3.1 Caching on a file server

When applying a caching technique, one aspect to keep in consideration is the physical destination of the cached files. For example, everything could be stored in a single machine, acting as caching server for the computing nodes. This approach would still require all the machines to be in the same network, preferably with a high-bandwidth connection between them, in order to have some chance to be faster than reading the

files from a remote storage facility. The technology that will be tested for this configuration is XRootD. In particular, its proxy storage plugin described in Section 5.2.1 fits in the requirements described so far. In this work, a single machine of the cluster acts as caching proxy, standing between the client application and the remote storage system. When the client asks for one or more remote files, the request will be redirected through the proxy and then to the final endpoint. Any file that is not already present on the proxy will be downloaded and stored in a specified directory.

An initial investigation on the behaviour of the XRootD proxy highlighted that its default configuration leads to suboptimal performance. In particular, these are the default behaviours that were modified for this work:

- Files downloaded to the proxy are prefetched in chunks. Practically, it is a staging phase that takes place at the beginning of the user application, slowing it down with no added benefit. Thus, this behaviour was disabled in the XRootD configuration on the server.

- Since the XRootD proxy streams data in blocks that are agnostic to the particular data format used, some consideration must be put in the difference between the cluster size of TTree and the size of the blocks downloaded by the proxy. With too large block sizes, the practical difference on disk between the real size of the TTree and the size of the cache may become non-negligible. For this reason, the minimum blocksize was used (4 KB), so that the size of a cached dataset would respect its true size as much as possible.

- By default, users would need to prepend the URL of the proxy server to the URL of any file they want to read from some remote location. To remove this extra burden, the proxy was configured such that its URL is automatically prepended to any user-supplied URL. This makes the proxy completely transparent in the user application.

In its final configuration, this mechanism runs as follows. During the first run, computing nodes make a request to the XRootD proxy to read a particular fragment of the input remote dataset. The proxy then fetches the requested portion remotely, caches it internally then serves it to the node which has made the request. During subsequent

runs, the request of a worker node is served directly from the local cache on the proxy. This workflow is shown in Figure 5.1.

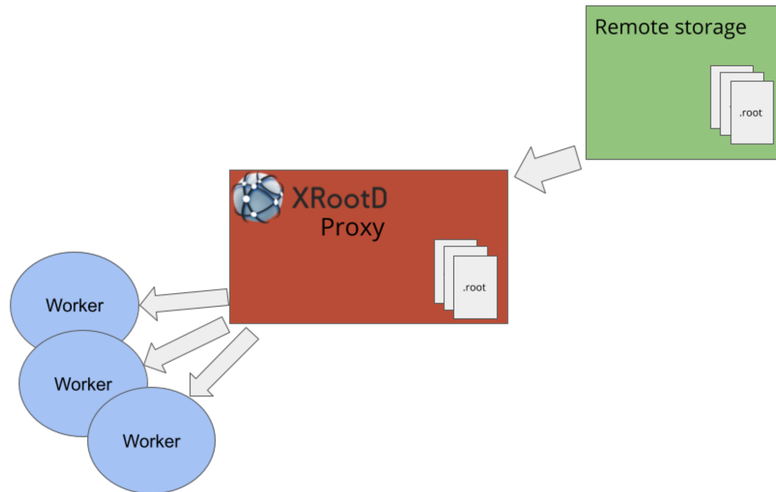


FIGURE 5.1: XRootD proxy cache. During user analysis, computing nodes (labeled "Worker" in the image) make read requests for their assigned ranges of entries to the proxy server, which in turn forwards such requests to the remote storage system. The proxy stores the requested entries in its local filesystem and will be able to serve them directly to the nodes during subsequent runs of the application.

5.3.2 Caching on the computing nodes

Another approach is to make each computing node store only its portion of processed data on its local filesystem. A first important takeaway is that the computing-node-local caches are exploited best if the distributed execution engine can guarantee data locality. That is, it always submits tasks where their input data fragments were previously cached. This aspect will be further explored in tests, discussed in Section 5.4.3.

The TFilePrefetch class of ROOT was used for this second strategy. This system was activated on each computing node, caching only the necessary TTree columns and entry clusters on the machine. The TFilePrefetch thread is only responsible for the I/O of the blocks of entries and in general does not put extra strain on the CPU running the

RDataFrame computations, especially in runs where the cache is already populated. A schema of this strategy is given in Figure 5.2.

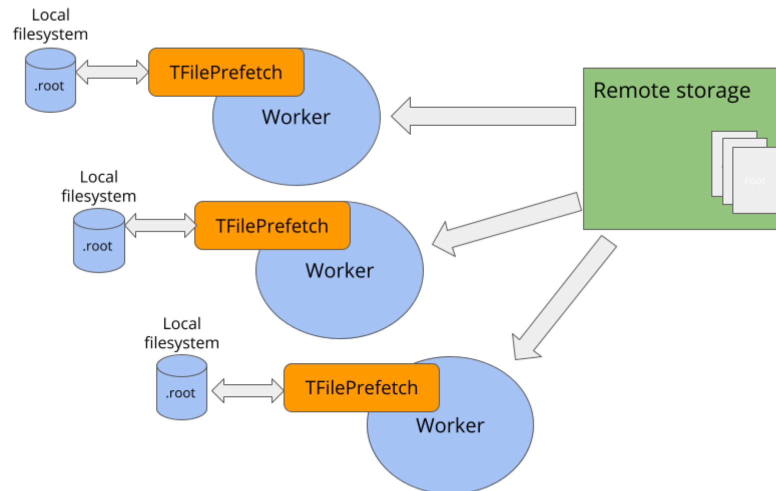


FIGURE 5.2: TFilePrefetch cache. During user analysis, computing nodes (labeled "Worker" in the image) make read requests for their assigned ranges of entries directly to the remote storage system. On each node, TFilePrefetch intercepts the incoming blocks of entries and stores them on the local filesystem. In subsequent runs, each node will be able to read the same range of entries from the local disk instead of requesting it again from remote.

5.4 Evaluation of existing technologies for caching during an RDataFrame analysis

The tests developed focus on showcasing the transfer of data from the remote storage system to the computing nodes or the caching server. A reference dataset has been created: a single ROOT file with a TTree of 10^5 entries and ten TTree clusters (exactly ten thousand entries per cluster). The dataset contains five columns of randomized data. The tests always try to read one specific column of type `double`. The total file size is 1.8 GB, while the column of interest is 700 MB. This file is uploaded to a CERN storage facility, from which it will be readable through the XRootD protocol.

5.4.1 Methodology

The RDataFrame computation graph is the same for all the tests: a very lightweight function running on the selected column of the dataset. This is enough to trigger the XRootD read requests from remote storage and observe the different effects depending on the caching mechanism enabled. The baseline is defined by running this RDataFrame application on a single machine, either with no cache or with one of the two caching mechanisms enabled. The following test configurations distribute the application to a set of nodes using the Spark backend of the distributed RDataFrame interface. The source code of each test is available in a GitHub repository [166].

5.4.2 Hardware setup

The hardware setup is made of a physical machine plus a set of virtual machines. The baseline of tests runs on the following machines:

- 1 physical node, CPU i7-6700 (4 physical cores, 8 logical), 256 GB SSD storage and 16 GB RAM. Serves as cache server in the tests with XRootD cache enabled.
- 1 virtual machine (VM) with 1 core, 10 GB spinning disk storage, 1024 MB RAM. Runs the RDataFrame application.

The second test configuration reuses the same physical node, but extends the number of total VMs to 5 (each with specifications identical to those of the VM described above) in order to form a Spark cluster. The virtual machines are created in the CERN OpenStack Cloud [167], while the physical machine is located at CERN. Thus, all machines used in the tests are inside the CERN network.

5.4.3 Results

In this section two different test scenarios are presented. In the first scenario, the RDataFrame application described in Section 5.4 is executed on the single node setup described above. In the second scenario, the same application is distributed over the Spark cluster described above. Each scenario in turn presents three tests: the baseline test with caching disabled, one test with XRootD cache enabled on a server separate

from the computing nodes and one test with TFilePrefetch cache enabled on the local filesystem of the computing nodes. The results of the tests are presented in this section and are discussed in Section 5.4.4.

Single node

In the single node scenario, the reference dataset is read from remote storage on the computing VM node during runtime. Only the selected column of the dataset is cached. If XRootD cache is enabled, data belonging to the selected column in the test will be stored on the caching server. The size of the cached data depends on the XRootD block size. For that reason its default value has been changed as explained in Section 5.3.1, so that the cache will contain approximately the same volume of data of the selected column.

If TFilePrefetch cache is enabled instead, data are stored directly in the local filesystem of the VM. This mechanism caches exactly the TTree clusters that the application requests. Subsequent runs will read data from the cache and not from remote storage. Each test is run a thousand times to get a significant distribution of the execution times of the application, since this value might vary especially when the cache is disabled. Figure 5.3 shows the execution time results. For the cache enabled cases, only the runs where the cache was already populated are considered (i.e. no cold cache runs are shown).

In the same figure, it is possible to observe the significantly higher variability in execution time of the application when reading data from remote storage rather than from the caches (the standard deviation with caching disabled is respectively 13 times higher than the standard deviation with XRootD cache and 10 times higher than the standard deviation with TFilePrefetch cache). At the same time, the average execution times with cache enabled are lower, respectively by 38% with XRootD and by 48% with TFilePrefetch. See Table 5.1 for a summary of these results.

The time to populate the caches was measured as well. XRootD cache takes on average 43 s with a standard deviation of 35 s, while TFilePrefetch takes on average 60 s with a standard deviation of 41 s.

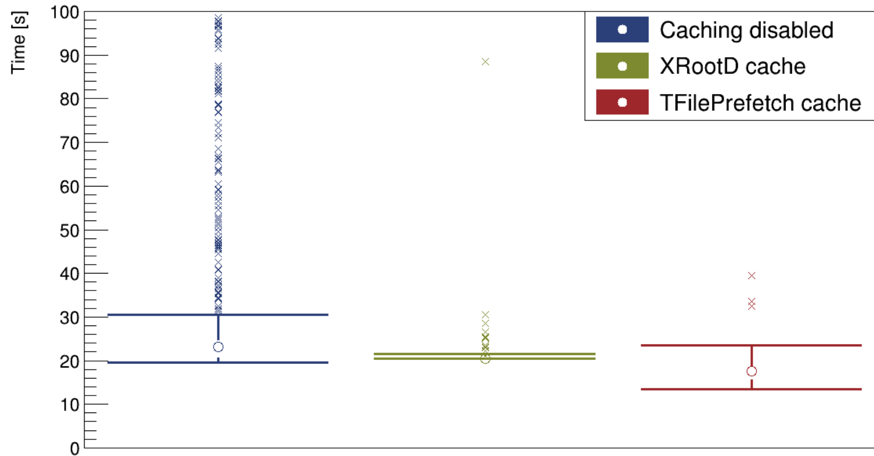


FIGURE 5.3: Single node scenario. Box plots of the distributions of execution times of one thousand test runs in three configurations: Caching disabled, XRootD cache, TFilePrefetch cache. The empty circles represent the median values of the distribution, the whiskers are drawn at $1.5 \cdot IQR$ (interquartile range) and the crosses outside the whisker boundaries represent distribution outliers.

Distributed cluster

In the second test configuration, all the VMs are included in the setup and it is possible to send tasks to the Spark cluster through the distributed RDataFrame interface. The Spark setup is thus made of one VM acting as Spark driver (the node from where the tests will be submitted), one acting as Spark master (the cluster coordinator and application scheduler) and the other three nodes acting as the Spark executors.

Each test is repeated 100 consecutive times, in order to simulate an interactive scenario where an exploratory user analysis is run once and then rerun subsequent times on the same data after some parameter modification. Less test repetitions are performed in this scenario, as the focus shifts towards evaluating the behaviour of the caching (both when populating the cache and when reading it in subsequent runs) and the variability of the distribution for runs after the first is less important.

The input dataset is transparently split in three logical partitions by the distributed

TABLE 5.1: Statistics for one thousand test runs along three configurations, with the single node setup.

	mean $[s]$	median $[s]$	σ $[s]$
Caching disabled	33	24	28
XRootD cache	21	21	2
TFilePrefetch cache	17	18	3

RDataFrame module: it is sufficient to give an optional parameter `npartitions` to the RDataFrame constructor. Each partition is sent together with the computation graph to one of the three worker nodes as described in Section 3.3. Each task then reads and processes data independently of the others. When the XRootD cache is enabled, the data corresponding to the whole selected column of the dataset is stored on the caching proxy server. With TFilePrefetch enabled instead, each task caches the logical portion of the column on the node which is processing it.

The results of this configuration are shown in Figure 5.4. The first points of the lines corresponding to tests with caching enabled show the run during which the caching mechanisms are downloading and storing the processed portions of the dataset. In the particular case shown in the figure, the run where the cache is being populated takes roughly 10 times longer than subsequent runs with both caching mechanisms. High spikes in the execution times of some of the runs with TFilePrefetch cache enabled are striking. They opened another topic of investigation in this work that will be further discussed in Section 5.4.4. This investigation led to modify the RDataFrame distributed module with the aim of forcing the Spark backend to apply data locality, i.e. to map tasks operating on the same logical range to the same node in subsequent runs. Following the points of the TFilePrefetch cache line show a higher execution time than the respective points on the XRootD cache line. This was not expected but could be due to some unpredictable strain on the host machines of the VMs.

Rerunning the same tests with the improvements of forcing data locality leads to Figure 5.5. In this figure the cold cache runs are not shown, instead the focus is on the subsequent runs with the cache already populated. The spikes previously observed using TFilePrefetch cache are no longer present. Average execution times with the

two caching mechanisms are similar and summarized in Table 5.2. On average, running with a cache mechanism enabled (either XRootD cache or TFilePrefetch cache) is slightly more than 2 times faster than running without cache.

TABLE 5.2: Statistics for one hundred test runs along three configurations, with the distributed setup and a locality-aware scheduler.

	mean [s]	median [s]	σ [s]
Caching disabled	36	26	17
XRootD cache	15	15	0.6
TFilePrefetch cache	16	16	0.5

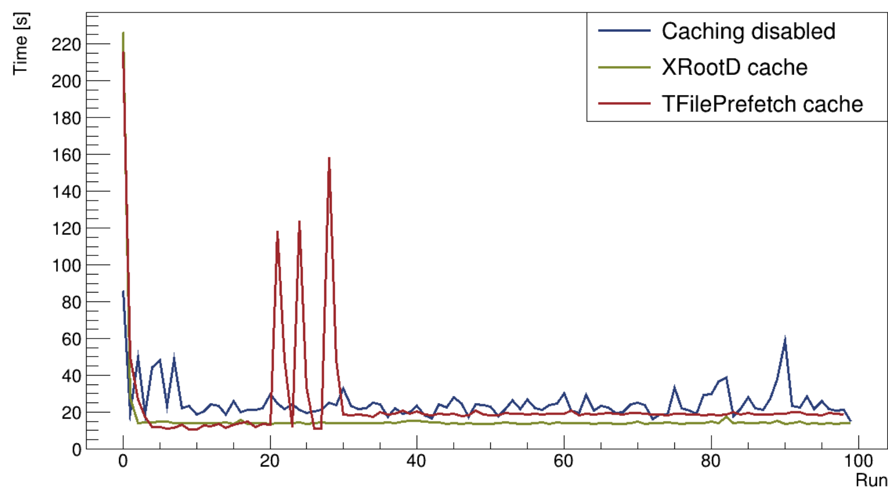


FIGURE 5.4: Distributed scenario. Lines represent the execution times along one hundred consecutive runs for three configurations: Caching disabled, XRootD cache, TFilePrefetch cache. The first point of the two configurations with cache enabled correspond to a run where the caches were being populated.

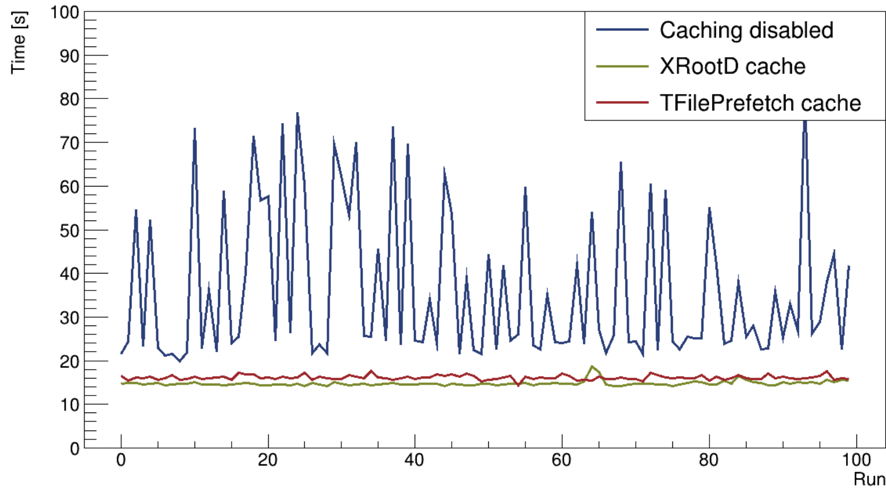


FIGURE 5.5: Distributed scenario, with data locality aware scheduler. Lines represent the execution times along one hundred consecutive runs for three configurations: Caching disabled, XRootD cache, TFilePrefetch cache. In the configurations with cache enabled, the caches were already populated in every run.

5.4.4 Discussion

The results presented in Section 5.4.3 generally show that enabling caching during the first run of the application makes subsequent runs faster. Further non trivial insights can be retrieved from the different scenarios investigated.

All test runs with caching disabled show a strikingly high variability in the execution time distribution, with a very long tail. This is a sign of the high load that storage systems at CERN have to sustain. This translates into unpredictable slowdowns in network I/O even when reading from within the CERN network as was done in this setup. This is already a strong point in favor of enabling caching for this kind of analysis, in order to protect the user from high latencies or overhead in remote data access.

In the single node scenario, XRootD cache shows the execution time distribution with the lowest standard deviation. In general this is not expected, but it is likely that the storage performance of the VM is responsible for the larger distribution in

the TFilePrefetch case. Nonetheless, TFilePrefetch shows the lowest average runtime, which is expected in general since data is stored directly on the same machine where it is processed.

The preliminary results of the distributed scenario shown in Figure 5.4 demonstrate that data locality is of utmost importance when caching on the computing nodes. This actually opens a new research question for this kind of effort: how to guarantee task pinning to nodes in a distributed computing environment. When distributing an RDataFrame application, the scheduler does not read the actual dataset and stream portions of it to the various nodes. Instead what the Spark master receives is a list of logical ranges of entries in the dataset and each element of that list corresponds to a task on some node of the cluster. In this context, task pinning would be beneficial. That means having the same logical range (i.e. a pair of integers) cached on the same worker node for all runs of the application.

Spark for example offers a `Cache` function in its API, but that still doesn't guarantee that tasks will always be sent to the same executors in the cluster. It is rather a way to signal the Spark scheduler that it is desirable to have that particular logical range cached on the cluster. In this sense, data locality is guaranteed eventually rather than at all times. It is possible though to have a stricter guarantee if some limits are set on the analysis workflow, namely that the application only runs computations of one single RDataFrame object and that the user does not exit the scope of their application until the end of their exploratory work. This was fully implemented in the distributed RDataFrame module for the purposes of this study, with no change in user code. Within this configuration, the distributed RDataFrame tests with the Spark backend indeed always pinned the same task to the same executor. This result is what Figure 5.5 shows, with the TFilePrefetch line overlapping the XRootD one for runs with the cache already populated.

5.5 Exploiting object store for HEP data analysis

The work described in the previous section focused on the traditional I/O system offered via TTree. Compared with the newer RNTuple implementation, it lacks both in terms of performance and flexibility in supporting different storage architectures [8,

168]. Thus, investigating caching mechanisms directly within RNTuple becomes interesting in the context of future HEP data pipelines. This section will describe the ideas, implementation and evaluation of caching RNTuple data using a bleeding-edge object store technology, namely Intel DAOS.

5.5.1 Exploration of a caching mechanism for RNTuple

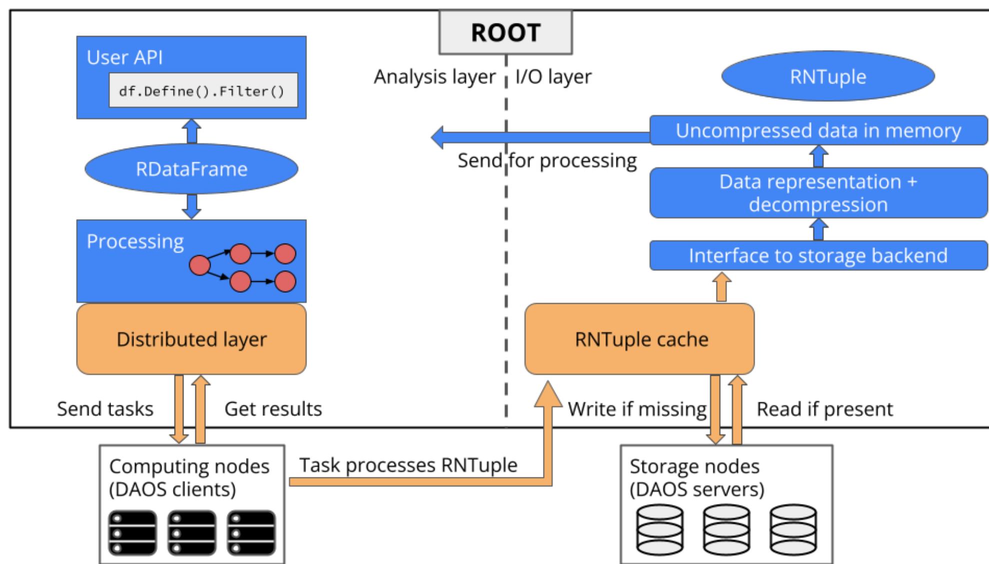


FIGURE 5.6: Overview of the proposed system. The upper box includes the main ROOT components involved in an analysis. On the left of the dashed line (*Analysis layer*) is the user-facing API and the processing engine offered by RDataFrame. On the right is the *I/O layer* that brings compressed physics data from disk to uncompressed information in memory that is sent to RDataFrame for processing. The two orange boxes represent the parts introduced in this work: the introduction of RNTuple as a supported input data format for the distributed RDataFrame layer in the *Analysis layer* and a caching mechanism for RNTuple in the *I/O layer*.

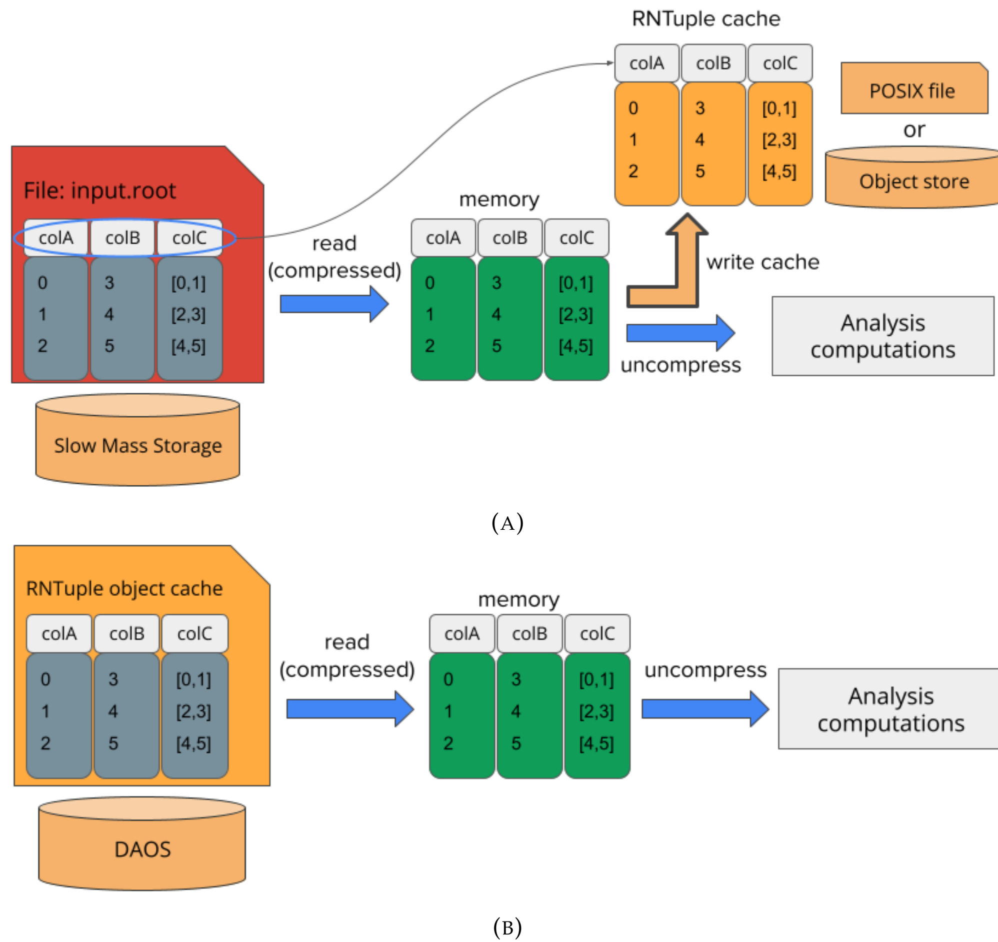
In this study, a caching machinery has been developed for data reading via RNTuple, so that any application that reads RNTuple data could benefit from it. This includes notably RDataFrame applications that read data from RNTuple. The caching

mechanism is independent of the storage backend, a crucial feature to maintain transparency for the user and contribute towards a sustainable development in a future where RNTuple will be able to read and write to even more storage systems than today.

Figure 5.6 gives a high-level view of the interaction between ROOT and DAOS after the proposed developments. A physics analysis with ROOT makes use of two main components: an analysis layer and an I/O layer. The analysis layer is implemented with RDataFrame, while the I/O layer in this case uses RNTuple. The user provides a dataset specification to RDataFrame, for example a list of files to process. In turn, RDataFrame will transparently invoke the low-level I/O layer which is in charge of opening the files from disk, uncompressing their data and sending them back to the processing layer. The image shows in particular the I/O layers defined within RNTuple as described in Section 2.1.1. All the blue boxes in the figure represent already established ROOT components, while the orange boxes demonstrate the parts that were specifically modified or developed in this work. In particular, the distributed RDataFrame layer was not able to process data coming from the RNTuple I/O. After this work, the algorithm that creates a distributed RDataFrame task on the client node also checks the origin of the dataset. This allows creating the correct RNTuple object when the task arrives on the computing node (bottom left part of the image). When a distributed task starts executing, it creates an RNTuple instance to read data from the selected storage. In case the RNTuple cache is activated, this can transparently start writing data from the original storage system to the target one. For the purposes of this work, the target storage system for the RNTuple cache is DAOS. If the DAOS server already contains the desired dataset, the developed cache will serve it directly to the rest of the RNTuple I/O pipeline which will in turn direct it towards the RDataFrame that requested it inside the distributed task.

5.5.2 Integration within the I/O pipeline

RNTuple I/O operations are scheduled in a pipeline. The current implementation of the pipeline is in two steps: first, data is read from storage into compressed pages in memory, then bunches of pages are decompressed together and sent to the rest of



(A)

(B)

FIGURE 5.7: Schema of the newly developed caching system in the RNTuple I/O pipeline. Blue horizontal arrows represent the current two steps of the pipeline: reading compressed pages and decompressing them. (a): The analysis is reading from some file-based source and a new RNTuple object is created to write data to a cache. (b): Data are read from the cache during the analysis.

the application for processing. The idea for the caching mechanism is to take place in between the two steps of the pipeline. This can be described as follows:

1. When programming an application, the user can enable the cache by simply providing a storage path (to a local directory or to an object store address for example)

as an extra option when opening an RNTuple.

2. In order to write the cache, a new RNTuple is created with the same metadata as the original RNTuple, this time pointing to the storage path provided by the user in step 1. Figure 5.7a shows the input dataset to the left, in red. The metadata, i.e. the list of three column names, are mirrored in the RNTuple cache that is shown on the right side of the image.

3. At a later stage, when the first step of the RNTuple pipeline is over, the compressed pages read into memory are grouped together in a cluster object. The cluster object contains a list of column names that the cluster is spanning. From any column name, a group of (compressed) pages belonging to that column can be retrieved. Consequently, the caching algorithm proceeds by traversing all column names and for each column it writes the corresponding pages into the newly created RNTuple object, thus populating the cache location (see Figure 5.7a). It is important to note here that the RNTuple system is implemented such that the column metadata is stored separately from the actual compressed pages (or groups thereof), so that information needed in the I/O pipeline is always available.

4. When the reading part is over, the RNTuple cache object finalises the writing operations and closes the open handle to the storage path (e.g. writes metadata about the number of pages and cluster layout to an attribute key in the DAOS case).

5. Any subsequent access to the same dataset by the user, will fetch the cached RNTuple rather than the original one. The caching mechanism is completely bypassed in order to avoid extra operations and the user is transparently presented with an RNTuple that resembles exactly their input dataset, but is read from a fast storage system like DAOS (Figure 5.7b).

5.5.3 Considerations for HEP use cases

Another important notion to discuss revolves around how the cache will interact with the ROOT I/O layer. As discussed in Section 1.2, HEP data is characterised by a “write-once, read-many” condition. Consequently, any caching system aimed at analysts’

needs in this field will need to optimise read operations as much as possible. Writing is also important to smoothen the user experience when the cache is still cold, but providing a faster reading performance directly translates into an increased productivity in physics analysis research. The machinery developed in this work tries to address both objectives: when writing, the cache does not need to wait for the decompression step of the RNTuple pipeline; when reading, it directly forwards all requests to low-level efficient RNTuple interfaces. During this investigation, the caching system could not make full use of an asynchronous pipeline since the DAOS backend for RNTuple could not support it at the time. Future iterations of this mechanism would write the compressed pages to the RNTuple cache in parallel with respect to the main pipeline.

5.5.4 Interaction with DAOS

The I/O workflow from the point of view of the caching node (which in this work corresponds to a DAOS server) looks like this: the user starts an analysis, requesting to process some dataset; the dataset is opened and both sent from disk to memory for processing and at the same time written as-is into the target caching node; after this, any other time an analysis is run and requests the same dataset, it is automatically read from the cache. Overall, the number of read operations is much higher than the number of write operations in this context.

The DAOS specification also establishes the use of caches to boost data access for its users. This is implemented at various levels, for example it is always enabled by default if DAOS is configured to use the *dfuse layer* [169]. At the hardware level, it exploits burst buffers on the server nodes. All of these characteristics are completely transparent and orthogonal to the caching system for RNTuple developed in this work. This is, from the point of view of DAOS, just like any other user application that reads or writes data stored in the DAOS servers. Thus, any improvement to the DAOS library or any site-specific tuning enabled on the server nodes will automatically be leveraged by the RNTuple cache.

This study focused only on the point of view of a single user. In a multi-user scenario, this caching system embedded in RNTuple should be synchronised with a

storage-facility-wide service. Taking for example two different users that want to access the same dataset, whoever does access it first will cache it in the object store thanks to the system developed in this work. But in order for the other user application to know about the presence of the dataset in the cache, some dataset register should be queried and report whether the same data is already present. This kind of challenge can be a topic for future studies.

5.5.5 Experiments

This section will present various test configurations that were employed to evaluate the capabilities of the proposed caching mechanism. At first, the cache is exercised on a small dataset, without running a physics analysis but just comparing the reading speed of the RNTuple cache on DAOS with the reading speed from a local SSD. Afterwards, a real HEP analysis is performed with the RDataFrame tool, either on one node or distributed to multiple nodes. For this second type of test, two different clusters have been used. The first cluster features a DAOS system where the RNTuple cache can store data on the DAOS servers and send it to the RDataFrame engine for processing. The second cluster has a Lustre shared filesystem and in this case the same distributed RDataFrame analysis processes data with the traditional ROOT I/O system using TTree.

Methodology

The following groups of tests have been set up for the caching system developed in this work:

1. A single-threaded C++ application that uses the RNTuple interface to read sequentially all the entries of a dataset stored in a ROOT file. No other computation is done in the application. The dataset size is 1.57 GB. The purpose of this test is comparing the runtimes of three different configurations: (i) reading the dataset stored in a file on the local SSD of a node; (ii) reading the dataset stored in a file on the local SSD of a node and at the same time caching data to DAOS; (iii) reading the dataset stored in the DAOS cache.

2. An open data analysis of the LHCb experiment at CERN [170], named from here on "LHCb benchmark". This analysis is run on both clusters described in Section 5.5.5. It uses the distributed RDataFrame tool to steer computations from one to multiple nodes. In the DAOS cluster, it reads data from the DAOS servers through the RNTuple cache. In the Lustre cluster, I/O is done with the traditional TTree implementation. The application processes the dataset used in the cited publication, replicated eight-hundred-fold to get a 1259 GB sample. In the tests with TTree and Lustre, the dataset is replicated simply by providing a list of paths to multiple copies of the original dataset. In the tests with RNTuple and DAOS, the replicated dataset is obtained by running a C++ program that reads all the entries in the original file (stored on the same SSD of the node in the previous group of tests) and copies them to one or more separate RNTuple objects stored in DAOS, until the desired size is reached. The number of objects is equal to the number of distributed RDataFrame tasks, so that each task processes exactly one RNTuple object. As previously discussed, particle physics events are statistically independent, so this approach is valid for benchmarking purposes.

The benchmarks in Section 5.5.5 request a variable amount of nodes and cores per node on the cluster through the distributed RDataFrame tool. In all tests, 2 GB of RAM are requested per core. For the DAOS tests, data is always cached on the DAOS server nodes, never on the computing nodes. The tests on the Lustre cluster present a similar situation.

The tests done on the cluster with the Lustre filesystem are run by submitting jobs to the Slurm resource manager. In each job, the desired number of nodes and cores for that test is requested. Furthermore, each job requests exclusive access to all the computing nodes involved in the test, to avoid unpredictable loads on the machines due to shared usage with other users of the cluster.

The test suite is available in a public code repository [124].

Hardware setup

DAOS cluster In the DAOS cluster there are seven client nodes and two servers. According to the DAOS specification, the dataset that is processed in the experiments

described in the next sections is always stored on the server nodes. Specifically, the server nodes are the caching nodes. Client nodes on the other hand only read the data from the server nodes and run the computations defined in the physics analysis. Each client node features the following hardware specifications:

- Motherboard: Newisys DoubleDiamond TCA-00638.
- CPU: 2x Intel Xeon E5-2640v3, for a total of 2 NUMA sockets, 8 physical cores per socket, 2 threads per core.
- RAM: eight Micron 36ASF2G72PZ-2G1A2 DIMMs, 16 GiB each, for 128 GB of total memory.
- Inifiniband interfaces: one Mellanox MCX354A-FCBT two port NIC, only one port was cabled, 56 Gb/s FDR speed; one HPE P23842-001 two-port NIC, only one port was cabled, 100 Gb/s EDR speed. Each interface is connected separately to one NUMA socket.

Each server node features the following hardware specifications:

- Motherboard: Supermicro X11DPU-Z+.
- CPU: 2x Intel Xeon Gold 6240, for a total of 2 NUMA sockets, 18 physical cores per socket, 2 threads per core.
- RAM: twelve Hynix HMA82GR7CJR8N-WM volatile DIMMs, 16 GB each, 6 per socket. 192 GB total memory.
- DAOS storage: twelve Intel HMA82GR7CJR8N-WM persistent memory DIMMs, 128 GB each, 6 per socket. Also, eight Samsung MZWLJ3T8HBLS-00007 3.84 TB NVMe SSD, four per socket.
- Infiniband interfaces: two Mellanox MCX654105A-HCAT one-port NICs 200 Gb/s (HDR). Each interface is connected to one NUMA socket. The node has PCIe Gen3 buses, so the actual bandwidth is 100 Gb/s per NUMA socket.

In practice, the maximum bandwidth that can be obtained when reading data on one of the client nodes is given by the sum of the two nominal bandwidths of its Infiniband interfaces. That is, a client node can read up to 156 Gb/s, or 19.5 GB/s.

The maximum bandwidth overall of the whole DAOS cluster is given by the sum of the nominal bandwidths of the server nodes. Thus the maximum reading throughput for the whole cluster is 400 Gb/s or 50 GB/s.

The DAOS version installed on this cluster is 1.2.

Lustre cluster The second cluster used for this work is a large computing cluster with hundreds of nodes and a shared Lustre filesystem. Access to the cluster was granted via a user account registered with the Slurm resource manager of the cluster. Cluster resources were shared among many other users. Also in this case there are server nodes where data is stored (on Lustre) and computing nodes that read the data from the server nodes and run the computations.

Each client node features the following specifications:

- Motherboard: Supermicro H11DST-B.
- CPU: 2x AMD EPYC 7551, for a total of 2 NUMA sockets, 32 physical cores per socket, 2 threads per core.
- Inifiniband interface: one Mellanox ConnectX-4 VPI adapter card, FDR IB 40GbE, 56 Gb/s.

The network topology is built like a fat-tree, with a 2 to 1 blocking factor on the computing nodes. More information about this cluster can be found in its user manual [171].

Results

Caching RNTuple to DAOS The first group of tests described in Section 5.5.5 is executed on a single node. The dataset is initially stored on SSD in order to gather more consistent measures and avoid possible network instabilities. Nonetheless, the same

tests could be repeated with the dataset stored in a remote file, since RNTuple data can be also read through HTTP.

Table 5.3 shows average runtime of the application with three different configurations. On the one hand, caching to DAOS while reading from SSD brings roughly 50% overhead with respect to only reading from SSD. On the other hand, reading from the DAOS cache is more than 6 times faster than reading from SSD.

TABLE 5.3: Runtime metrics of tests reading an RNTuple dataset, 50 repetitions per configuration.

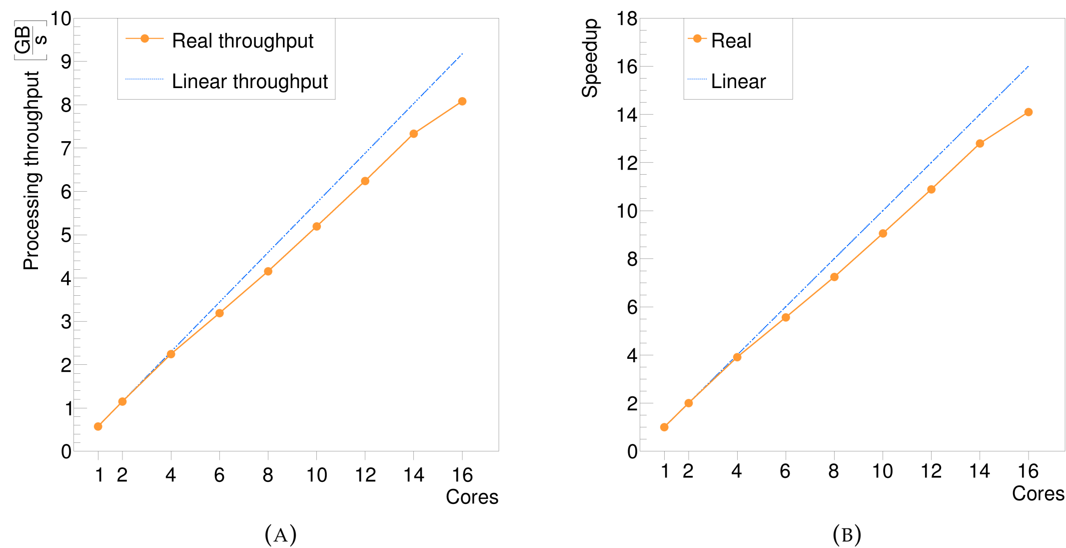
Read location	Average [ms]	Error [ms]
SSD	3694	7
SSD (while caching)	5606	5
DAOS	600	7

Distributed RDataFrame analysis benchmarks reading data from DAOS A second series of tests evaluate the performance of running an RDataFrame analysis on top of RNTuple data cached in DAOS. The LHCb benchmark described in Section 5.5.5 is executed in a Python application with the distributed RDataFrame tool. This allows to parallelise the analysis both on all the cores of a single machine and on multiple nodes, all with the same application. Furthermore, while the user code is written in Python, this is just used as an interface language and each task is actually running C++ computations through RDataFrame. Within the test, the dataset is split into multiple RNTuple objects stored in DAOS. Then, one task is defined to run the analysis on a single RNTuple in its own Python process. In general, for any given number of cores used in the following tests, there are as many Python processes and as many RNTuple objects stored in DAOS.

At runtime, the application is monitored with a timer that is used to compute the processing throughput (that includes time spent reading and time spent in the computations). 72% of the total dataset is read and processed, roughly 904 GB. The processing throughput is then computed dividing the processed dataset size by the execution time. Figure 5.8 and Figure 5.9 both show the absolute value of the processing throughput

(with increasing amount of cores either with a single node or multiple nodes) and the value relative to one core for the single-node case or one node for the multi-node case.

The following results are representative of tests where the application processes are pinned to run on either NUMA domain of the node. The backend of distributed RDataFrame is set up such that there are two executor services running on the node, one that will accept and process tasks running on the first NUMA domain, the other running its tasks on the second NUMA domain.



(A)

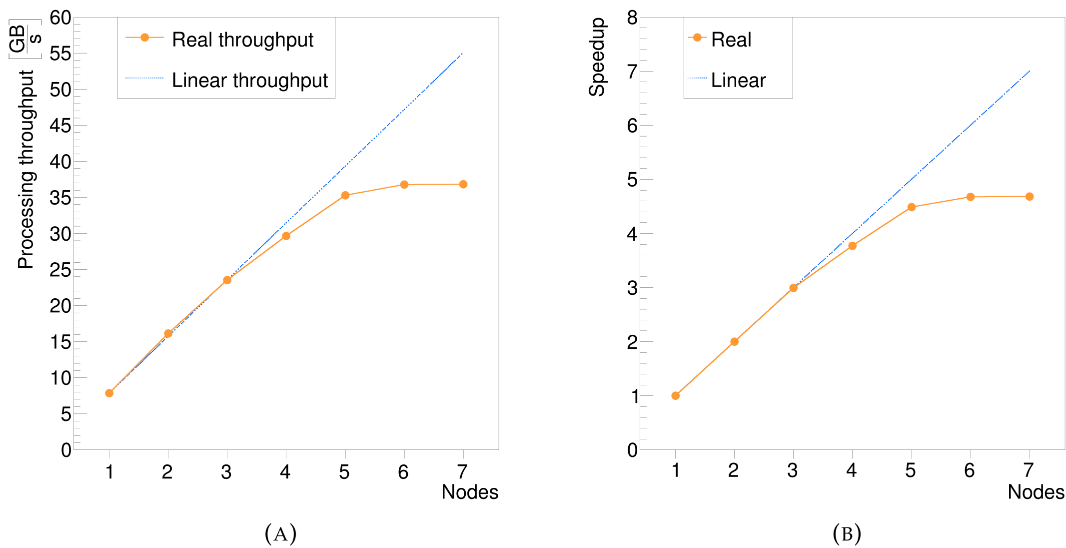
(B)

FIGURE 5.8: Processing throughput (i.e. reading the dataset and running analysis computations on it) of a distributed RDataFrame analysis on a single node of the DAOS cluster. (a): Real throughput values compared with a linear throughput increase obtained by multiplying the throughput on one core by the number of cores on the x axis. (b): Speedup obtained by scaling the analysis to multiple cores on the node.

Figure 5.8 shows the throughput obtained by running the application on one node, with an increasing amount of cores up to 16 (8 physical cores per NUMA domain). In particular, Figure 5.8a shows the absolute processing throughput on a single node with increasing amount of cores. Here it can be seen that this tool is able to reach a peak processing throughput of more than 8 GB/s on one computing node. Figure 5.8b

instead reports relative speedup on one node, which in this case is almost perfectly linear. In both images, up until 8 cores the test is using one of the NUMA domains on the node. When more than 8 cores are used, 8 of them are pinned to run on the first NUMA domain of the node, while the remaining are pinned on the second NUMA domain, to factor out NUMA effects.

The same analysis is then scaled to multiple nodes. Figure 5.9a shows that the peak processing throughput achieved is 37 GB/s, while the speedup plot in Figure 5.9b shows a plateau when more than five nodes are being used.



(A) (B)

FIGURE 5.9: Processing throughput (i.e. reading the dataset and running analysis computations on it) of a distributed RDataFrame analysis on multiple nodes of the DAOS cluster (using 16 cores per node). (a): Real throughput values compared with a linear throughput increase obtained by multiplying the throughput on one node by the number of nodes on the x axis. (b): Speedup obtained by scaling the analysis to multiple nodes of the cluster.

Distributed RDataFrame analysis benchmarks reading data from Lustre The same physics analysis is then run on the cluster that uses the Lustre shared filesystem for

data access. In this case, the traditional ROOT I/O system using TTree is put to the test with a file-based storage system. The LHCb benchmark described in Section 5.5.5 is executed in a Python application with the distributed RDataFrame tool. The same dataset with the same size is processed, with the only difference being that it is stored in TTree format rather than RNTuple.

One other difference in this case is that the TTree I/O, being more mature than RNTuple, already implements a way to read only a group of rows from a certain dataset when requested. Thus, the distributed RDataFrame tool is already capable of automatically splitting the user-provided dataset specification (i.e. the list of files to be processed) into multiple tasks, each containing a range of entries to process. When a task reaches a computing node, it will automatically create a local RDataFrame and open a TTree-based dataset reading only the entries supplied in the task metadata.

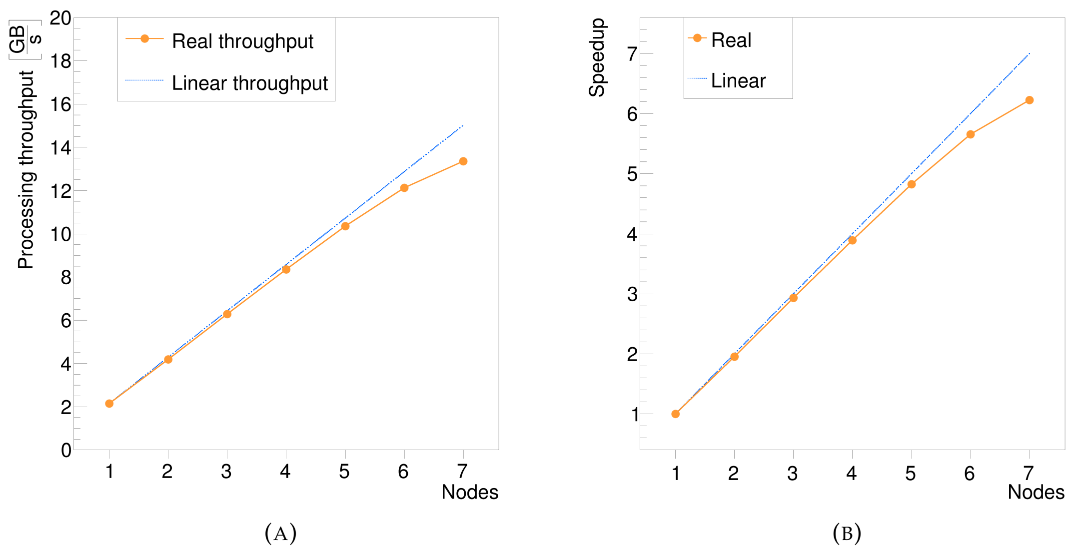


FIGURE 5.10: Processing throughput (i.e. reading the dataset and running analysis computations on it) of a distributed RDataFrame analysis on multiple nodes of the cluster with the Lustre filesystem (using 16 cores per node). (a): Real throughput values compared with a linear throughput increase obtained by multiplying the throughput on one node by the number of nodes on the x axis. (b): Speedup obtained by scaling the analysis to multiple nodes of the cluster.

Figure 5.10 shows the throughput obtained by running the application on an increasing number of nodes of the cluster, in order to recreate as closely as possible the same configuration used in the tests described in Section 5.5.5. In particular, from one to seven computing nodes are requested to the Slurm resource manager, with exclusive access in order to avoid unpredictable CPU load from other users of the cluster. On each node, the benchmark requests exactly 16 physical cores. Each core will be assigned with one task, that will read a portion of the dataset as described above from the Lustre filesystem.

In Figure 5.10a, the processing throughput obtained on an average of 10 benchmark runs per node count is compared with a linear throughput increase obtained by multiplying the value at the 1-node mark by the number of nodes on the x axis. The maximum throughput achieved is 13.3 GB/s. Figure 5.10b reports the speedup of running the analysis on multiple nodes relative to one node, comparing it with a linear speedup. The figure shows a perfect alignment between the two lines until the 4-node count, with a slight decrease in real speedup when using more nodes.

Discussion

Adding a new caching mechanism to a complex library such as ROOT requires careful design, both for usability and performance purposes. The proposed design is completely transparent to the user, who still only has to program their analysis through the RDataFrame high-level API. Drawing inspiration from the flexibility offered by the RNTuple layers described in Section 2.1.1, the caching system is injected in the I/O pipeline. The developed cache is backend-independent, thus enabling reading and writing RNTuple objects from/to any of the supported storage backends. This approach is the most sustainable in a field such as HEP, where large datasets can be stored in many different facilities around the world, each one with their own storage architecture.

In this work, the cache was exercised at different levels. At first, a physics dataset was either stored on an SSD or cached to DAOS with the developed tool. The results in Table 5.3 are promising for the caching mechanism. When caching data that is being read from the local SSD of a node, it is expected to have some overhead. But the speed

gained when reading from the DAOS cache more than compensates this overhead. It is also worth highlighting that the main use case for such a tool is when the dataset is still in a remote location. In fact, the same dataset used in these tests and the following ones was originally stored at CERN and downloaded locally on the cluster. The time to download the dataset was not included in Table 5.3, but it is safe to state that long-distance network I/O does not achieve the same reading speeds as a local SSD.

The following results presented in Section 5.5.5 demonstrate the capabilities of the distributed RDataFrame tool used in conjunction with the new storage layer offered by RNTuple and its DAOS backend. In this case the dataset was replicated to reach a size of more than 1 TB, in order to give enough workload to the computing nodes. This was chosen in accordance with the usual dataset size for an LHC Run 2 analysis, which is in the order of one to a few tens of Terabytes.

On the DAOS cluster, the analysis reached a peak processing speed value of 8 GB/s and 37 GB/s respectively with one node and seven nodes. Comparing these numbers with the maximum bandwidths described in Section 5.5.5 reveals that the peak processing throughput on one node is equal to 40% of the maximum reading throughput, while the peak processing throughput on seven nodes is equal to 74% of the maximum reading throughput of the whole cluster. It must be noted that the processing throughput numbers include all the steps of the analysis: opening the RNTuple objects, reading the data, bringing it to the RDataFrame processing layer and running the computations defined in the analysis application. Thus, this first result is promising considering that the HEP analysis are I/O bound. The scaling shown is close to ideal on one node with processes pinned to either NUMA domain, less than ideal when using multiple nodes. These results showed that the implementation of the DAOS backend for RNTuple at the time could not fully make use of the bandwidth, although that has been addressed in more recent developments of RNTuple and will be tested in future works.

The results obtained on the DAOS cluster can be compared with the benchmarks shown in Section 5.5.5, which involve running the same analysis on another cluster that uses the Lustre filesystem. The comparison, while not done on exactly the same hardware because the DAOS cluster does not provide a Lustre filesystem, is still representative of the advantages offered to ROOT by a low-latency high-bandwidth object store like DAOS. The Lustre cluster also uses Infiniband interfaces like the DAOS

cluster. All the same, the results presented in Figure 5.10 show an overall worse performance of the analysis. It should be considered that the slope of the speedup is better in this case, almost aligned to a linear speedup all the way up to seven nodes. This is due to TTree being a much more mature I/O system than RNTuple, especially considering the RNTuple interaction with DAOS. In fact, file-based I/O in ROOT dates back to its very beginning in 1995. In spite of this difference in maturity between the two I/O systems, RNTuple with DAOS is able to achieve an almost three times higher processing throughput than TTree with Lustre at the moment. This demonstrates the potential gain of exploiting a high-throughput system such as DAOS as a data source for HEP analysis in ROOT with RNtuple, compared with a traditional file-based approach with TTree, even when the latter is supported by a first-class parallel filesystem that is currently used by most of the top supercomputers in the world [172].

5.6 Conclusions

This chapter has shown a few approaches to drive the caching input data directly from within the analysis layer.

The first work made a first attempt at using existing technologies already available to HEP users and applying them to distributed analysis with RDataFrame. Two different caching strategies were evaluated, i.e. on a dedicated cache node or on the computing nodes. Both gave a factor 2 speed improvement with respect to the average baseline measurement with caching disabled. This evaluation also highlighted that a caching mechanism for HEP data analysis should store only the data that is actually read by the application. Thus, it should be aware of the data format used in the analysis so that the cache can be well integrated with it. For example, it should be able to cache a dataset at the level of the single cluster of entries of the columns that are read.

The second effort moved away from the traditional file-based approach for storing physics data. It featured a new integration between a bleeding-edge object store technology such as DAOS and the next-generation ROOT I/O offered by RNTuple, via a simple caching service that was storage-backend-agnostic. This idea allows to run an application that transparently reads a remote dataset from a POSIX filesystem and

writes it to an object store, opening the door to previously unavailable fast storage systems to cache HEP data in an analysis environment. The throughput achieved when pairing RNTuple with distributed RDataFrame was very high. Starting from a single core with close to 600 MB/s, a peak of 37 GB/s was reached using seven nodes (16 cores per node). A comparison was performed by testing the selected physics analysis with distributed RDataFrame processing a TTree dataset stored on a Lustre filesystem, using the traditional ROOT I/O system. This resulted in a 2.8 times lower processing throughput, peaking at 13.3 GB/s with the same number of nodes and cores used in the DAOS benchmarks. Consequently, this effort demonstrated the potential of using low-latency high-bandwidth object stores to accelerate HEP data analysis.

Chapter 6

Serverless computing for HEP data analysis workflows

The previous chapters outlined a more traditional computing approach for physics workflows, based on the exploitation of HPC resources and the use of distributed execution engines. As it was made clear in Section 1.1, future challenges in computing highlight the need for exploring different paradigms that may accompany established workflows, thus providing viable alternatives for parts of the community. Thus, this chapter of the thesis takes on a different research path that evaluates the model offered by serverless computing in the context of HEP data analysis. Two main efforts are presented in this sense. In both cases, the distributed RDataFrame tool was augmented with execution backends based on the serverless approach.

The first effort, described in Section 6.3, shows an implementation of the MapReduce workflow employed internally within distributed RDataFrame via the Amazon Web Services (AWS) cloud. The main advantages and limitations of using that service are discussed and the performance of the engine is evaluated with a set of benchmarks, including a real use case of physics analysis.

The second effort drew inspiration from this first prototype, creating a new serverless backend that leverages an open source cloud platform. The contribution, written in Section 6.4, also introduces different strategies to implement the MapReduce pattern, especially regarding the reduction phase. Furthermore, it presents a thorough analysis of the different sources of overhead that may interfere with the core physics event processing when using a serverless platform.

6.1 State of the art

Distributed computing for scientific applications is very often driven by large facilities or federated grids managed through job submission systems, as detailed in Section 1.5. In HEP this is exemplified by the WLCG, but the same type of resource managers is commonly employed in world-class supercomputers such as Frontier [173] or LUMI [174]. This kind of approach bears a cost for final users as already discussed in Section 1.5, who need to follow a strict series of steps in order to run their applications.

One alternative approach which has been explored in recent years, especially in the context of cloud computing, is a complete separation of all the logic to manage the hardware infrastructure and the task scheduling from the interface to input an application. This paradigm is broadly known as serverless computing, a model in which the cloud provider allocates machine resources on demand, taking care of the servers on behalf of their customers. It makes it easier for developers to access a high amount of computing power without having to think about setting up or managing the machines executing the code. Although the serverless paradigm is employed in a large suite of services such as serverless databases or storage layers or event streaming and messaging workflows, for the purposes of this thesis the main usage of this model is the so-called "Function as a Service" (FaaS). This concept operates on the notion of "function" - a piece of code that will be replicated and invoked on multiple nodes. It allows to execute user-provided functions in response to events without knowledge about the underlying infrastructure and the scheduling of the resources. The introduction of the serverless computing paradigm improved on some of the pitfalls of managed systems described above. For example, the end user does not need to have any knowledge of the underlying cluster infrastructure while submitting tasks to the remote machines. This allows to run massively parallel computations outside of typical supercomputing facilities, as less deployment-specific administration overhead is required.

Every serverless execution engine relies on a type of abstraction layer for the infrastructure itself. This is usually provided by big vendors, such as Cloud Functions by Google [175], Lambda by Amazon [176], or open source solutions such as OpenWhisk [177] running on real or virtual machines or Knative [178] running ephemeral Docker [179] containers directly on Kubernetes [75]. All the solutions offered by these

players share many features. In some cases, communities compare the products in terms of cost efficiency and availability to make informed choices about the workflow they will propose to users. An example is offered by a recent overview of the serverless computing scenario in bioinformatics by Grzesik et al. [180].

Efficient orchestrating frameworks are useful in utilising the power of serverless functions in data processing applications. One among these would be PyWren [181]. It allows to seamlessly distribute arbitrary Python code over multiple nodes with serverless functions. Needed objects and dependencies are serialised and sent to the Lambda execution environment in order to run the application on AWS resources natively. As of 2021, the original project is no longer maintained, but it was used as a basis for interesting extensions, including NumPyWren for numerical algebra [182]. The newer development, Wukong [183], builds on NumPyWren experience with a focus on data locality and improved, decentralised scheduling.

The serverless research scenario is quite wide. Other works in this line of research are MARLA [184] (MApReduce on AWS Lambda) and SCAR (Serverless Container-aware ARchitectures). MARLA is a framework that supports the MapReduce model in AWS Lambda for Python. One of the advantages of MARLA over other similar frameworks is that it is in charge of managing the entire MapReduce process, from the partitioning of the data to the generation of the final result, where the user only has to define the Map and Reduce functions of the process. SCAR [185] is a framework that offers the possibility of executing any programming language in AWS Lambda through the use of containers, allowing the execution of any type of application in a FaaS environment, which supported this execution model before AWS itself. In particular, the use of containerised environments to run the serverless functions has become more and more popular thanks to a few key advantages it brings in terms of reproducibility and ease of deployment. But this also comes with issues to be addressed. Bila et al. [186] discuss how containers may hide vulnerabilities which can be exploited at runtime. Li et al. [187] present the benefits of reusing the same container for multiple serverless functions to avoid cold startups. Oakes et al. [188] go as far as implementing a new container runtime especially aimed at serverless workflows to obtain factors of speedup with respect to using Docker.

Serverless workflows are not a common topic in High Energy Physics literature.

Nonetheless, there have been some efforts in this direction. A recent review has highlighted a few motivations that would make the FaaS model applicable to the online trigger systems employed by LHC experiments [189]. This paper theorises that inference using neural networks could be triggered by the events streaming from the accelerator. A more concrete example is provided by a paper related to CernVM-FS (CVMFS) [190], a file system that provides the software distribution backbone for collaborations in the field. The paper describes advances in the publishing system of the software distributions. The default model has a single server node responsible for the compilation of all the libraries and only upon a commit from this machine the distribution would be actually published and replicated. The envisioned changes would have any machine enabled with CVMFS publish the changes to cloud storage (e.g. Amazon S3) through a gateway serverless function [191]. Regarding the data analysis use case, a good example of serverless engine is provided by Lambada [192], that adds facilities on top of AWS Lambda to steer the serverless functions and shows a cost-efficient usage of the resources. Another notable effort is represented by funcX [193], a FaaS platform for HEP computing that allows registering user functions on arbitrary endpoints. Function registration and execution can be managed within the same Python application and the authors demonstrated good scaling of the solution up to two thousand function containers.

With respect to the efforts cited above, this chapter of the thesis introduces serverless engines that abstract even the last bit of interaction with the distributed system for the user, the creation and upload of the functions. The next sections will describe how the generic programming model adopted by RDataFrame can be fitted to generate and run serverless workflows, using industry-grade products such as AWS Lambda or open-source scalable frameworks such as OSCAR [194].

6.2 Tools

6.2.1 AWS Lambda

AWS Lambda is a service that allows running event-driven short-lived computations in a serverless environment. It supports writing serverless functions in different languages, among which Python. The functions can be executed with the native Amazon runtime or within a user-provided container. Lambda functions can be triggered by events such as uploading a file to AWS S3 [144] or sending an HTTP request. Once deployed, the number of function executions can be scaled dynamically based on the computational load.

One notable characteristic of this service is that once a container is spawned to execute a function, it is retained for some time that depends on the current workload on AWS resources [195]. Further invocations running on the same container will reuse it, thus reducing the startup time of the execution.

6.2.2 OSCAR

OSCAR is a FaaS execution environment aimed towards file-based event processing. The execution of the functions is done through the use of Docker containers. An OSCAR installation is based on a Kubernetes cluster that is deployed using the following tools: EC3 [196] to provide horizontal cluster elasticity; Infrastructure Manager, IM [197], to support multi-cloud deployment; and CLUES [198] to manage the elasticity of the cluster by taking care of the scale in and scale out of the nodes in the cluster, based on job demand.

An OSCAR manager service provides a REST API through which authorised clients may create, modify or delete serverless functions. When creating a new function, a user can specify a storage volume to associate it with. OSCAR employs object stores, in particular MinIO [199], to manage the storage of a function. Modifications to the storage, such as uploading a new object, will be considered as events by OSCAR. The input events need to be managed within the OSCAR platform, whereas the output of functions can also be uploaded externally to other object stores such as Amazon S3.

OSCAR was designed for file-driven workflows first. When a user uploads a file, this generates an event, to which a serverless function can react. The function defined within OSCAR spawns a Kubernetes pod launching the user-provided container image with the code that should process that file.

6.2.3 EOS

EOS [200] is the mass storage system used at CERN. LHC experiments' data are stored on EOS, which continuously serves files to users. Most of remote I/O transactions at CERN, e.g. when reading a file through the XRootD protocol, are served by EOS.

6.3 AWS Lambda functions for distributed RDataFrame

Allowing HEP scientists to benefit from the advantages of serverless computing without needing to teach them completely new tools requires a new method of communicating with existing platforms. The modular design of distributed RDataFrame offers a good opportunity to integrate the user-facing analysis framework with FaaS platforms. In order to achieve this goal, the MapReduce pattern needs to be adapted to the serverless model, in this work offered by the AWS Lambda service.

6.3.1 Overview of the interaction between RDataFrame and the serverless environment

The main idea for this contribution is to leverage the distributed RDataFrame machinery, developed for this thesis, to generate a MapReduce workflow and then execute it via the AWS Lambda service. When a user triggers the execution of a distributed RDataFrame action, the machinery described in Section 3.3. The input dataset is split in a list of logical partitions. The operations requested by the user in their code are registered with the head node instance, which then creates a computation graph object that can be serialised as described in Section 3.7. Finally, a new backend for distributed RDataFrame is implemented targeting specifically the AWS Lambda API. This backend is responsible for taking the list of partitions and the computation graph and creating a list of tasks for the Lambda functions. Each task contains one logical partition and the

computation graph object. This becomes the payload given to the AWS Lambda API, so that there is one Lambda invocation per task created. More specifically, a Lambda function is created on the AWS platform and configured such that it accepts a task created by the distributed RDataFrame machinery as payload. Practically, this is equivalent to the mapper of the MapReduce workflow. When one invocation has finished its job, it will store the partial results on an AWS S3 bucket. The backend on the client application waits for all the mapper invocations to finish. Afterwards, it retrieves all the partial results from all the buckets and runs the reduction locally on the client side, sequentially merging the partial results two by two. An overview of this workflow is depicted in Figure 6.1. The bottom left part of the image represents the moment when a user requests some result, for example drawing a histogram. From that point, the rest of the machinery as described above follows, from left to right in the image.

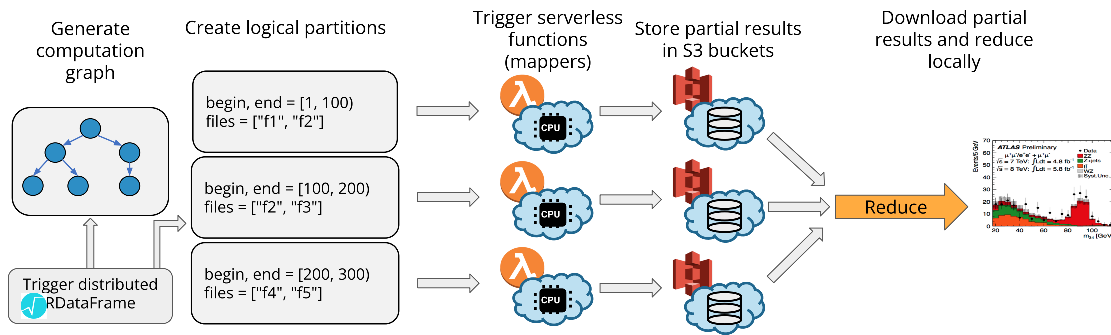


FIGURE 6.1: Overview of the distributed RDataFrame machinery adapted to run with AWS Lambda services.

The developed engine decouples the logic for the client side from the logic regarding the Lambda invocation (later called "worker side"). The first is run within the user application and the machine where it is started, through the newly developed AWS backend for RDataFrame. The latter is managed with Hashicorp Terraform [201] to deploy the required infrastructure on AWS resources with no user intervention. The whole execution environment is packaged in a Docker container which includes also a ROOT installation.

Client Side The client side is an implementation of a backend for distributed RDataFrame. The integration with AWS resources in Python is done via the Boto3 [202] software development kit. The overall mechanism of the client side is described in Algorithm 1. The machinery serialises the mapper function, metadata about the range of entries it should process and optionally an authorisation token. It spawns multiple Python threads that will invoke the Lambda functions. More detail about this specific part is given in Section 6.3.2. When the invocations finish, their partial results are streamed back to the client process and they are reduced sequentially, locally. In general, this could provide a bottleneck, both due to the extra remote I/O required to stream back the partial results and to the sequential operation that could be approached distributedly like in other RDataFrame backends. This was addressed in a later investigation, discussed in Section 6.4.

Algorithm 1 Invocation of the Lambda functions on the client side.

1: $mapper \leftarrow$ Function that applies RDataFrame computations
2: $token \leftarrow$ Kerberos authentication file
3: $headers \leftarrow$ Paths to optional C++ headers
4: $ranges \leftarrow$ Partitions of the input dataset
5: **for** $range$ in $ranges$ **do**
6:     *begin* THREAD($range$)
7:     $payload \leftarrow \{range, script, token, headers\}$
8:     $\{single_result, monitoring_data\} \leftarrow$ call Worker($payload$)
9:     call save($monitoring_data$)
10:     *end* THREAD
11: **end for**
12: $results \leftarrow$ call reduce($single_result$) foreach THREAD($range$)
13: **return** $results$

Worker Side The worker, i.e. the Lambda function that reacts to the event of submitting the serialised object from the client side, operates as shown in Algorithm 2. The only other service on which it depends is Amazon S3 for the storage of the output partial results. The execution of an RDataFrame computation graph via the deserialised mapper function is an example of a C++ runtime for AWS Lambda, which is not supported by default.

At the start of the function invocation, a monitoring service is launched in parallel, directly within the Python interpreter. It extracts information every second about the runtime of the function, CPU and network usage, which will be aggregated in Section 6.3.7.

Algorithm 2 Engine Algorithm on the worker side.

1: $\{range, mapper, token, headers\} \leftarrow payload$
2: **for** header in headers **do**
3:     declare *header* to ROOT interpreter
4: **end for**
5: $monitoring \leftarrow$ start ASYNC monitoring process
6: $result \leftarrow$ call $mapper(range)$
7: write *result* into S3 bucket
8: $monitoring_result \leftarrow$ stop($monitoring$)
9: **return** $monitoring_result$

6.3.2 Controlling the invocations via Python threads

AWS Lambda provides two kinds of invocations - synchronous and asynchronous [203]. The asynchronous one is based on the "fire and forget mechanism" and the response to the invocation request is sent straight away when Lambda queues the event for processing. In contrast to the previous one, the synchronous invocation waits until the computation is finished and reports the execution state directly to the caller.

In this first preliminary work, the synchronous invocation mechanism provided an easier way to implement a retry mechanism, to resubmit those invocations that may fail. After sending the synchronous invocation request, a connection with the Lambda is established and a thread calling an invocation procedure is waiting for the Lambda to complete processing. When the Lambda's work is done, the thread gets a response containing JSON serialized payload, the content of which can be specified inside the Lambda function's code. This allows for passing information about the success of the computation as well as the errors that occurred during the computation.

Error handling mechanism was achieved by running a pool of asynchronous Python threads, each managing a single synchronous Lambda execution. If the Lambda fails,

it sends a type of error and an error message to the client side, where those are logged and a retrial is done on the same thread. A single Lambda instance that still fails after several retrials will make the entire application fail, similar to the pattern provided by Apache Spark or Dask.

Using this approach did not provide any tangible overhead to the launch of the application. The only limitation was the limit of ten synchronous lambda invocations per second, imposed by AWS [204].

6.3.3 Kerberos token placement

To enable authentication to private storage at CERN, a transferable Kerberos ticket needs to be provided by the user on the client side. When the execution is triggered, the contents of the ticket are serialised and transported in the payload to each worker. Then, it will be unpacked at a specified location where XRootD will be able to read it (managed via the `KRB5CCNAME` environment variable). This will practically allow every Lambda invocation to have the same permissions as the user that invoked it.

6.3.4 Experiments

Two analyses are run on the AWS infrastructure:

1. A CPU-bound benchmark. This serves as a baseline for local resources utilisation, as opposed to the typical data-intensive analysis. Creates a simulated dataset with one billion randomly generated entries and three columns storing scalar floating-point values. The simulated dataset total size is 96 Gigabytes. The application computes the mean value of each column, ten times per column. This application will be called "CPU-bound benchmark" from now on.

2. A real physics analysis processing data from the PPS subsystem of the CMS experiment at CERN [205]. It consists of the selection of candidate events of exclusive dilepton production, $pp \to p \oplus \ell\ell \oplus p$, with $\ell \in \{e, \mu, \tau\}$. The measurement of exclusive production of lepton pairs involves two selections: (1) Exclusive cuts - leptons are produced exclusively, i.e., no other particles are produced during the proton-proton interaction, and (2) correlation between leptons and protons. The

total dataset size is 420 Gigabytes. This application will be called "PPS analysis" from now on.

6.3.5 Methodology

Each analysis is run with a distributed RDataFrame increasing the number of partitions of the input dataset, which is equal to the number of Lambda invocations. The values chosen are 8, 16, 32, 64, 128, 256 and 512. The distributed RDataFrame applications are started within an environment that has both the AWS credentials and the Kerberos ticket available (the latter is needed to access the dataset of the PPS analysis).

6.3.6 Hardware setup

There was no direct control over the hardware resources used for this work, since that is a characteristic of serverless computing. What could be controlled is the resource configuration for a Lambda function, which was:

- 1769 megabytes of RAM (corresponding to one vCPU-second of credit per second, which means that a full vCPU core is allocated to the Lambda function).
- The analysis runs on Intel(R) Xeon(R) Processor @ 2.50GHz as reported by monitoring tool.
- 15 minutes timeout, the maximum value allowed by AWS at the time of writing.
- The Lambda function is kept warm by running it as many times as partitions defined in the application, before the start of the main analysis, meaning that the ROOT initialization time is excluded thanks to avoiding additional time due to cold start.

6.3.7 Results

Runtime scaling The developed engine is put to test with the two analyses described in Section 6.3.4. The total runtime of the execution of the Lambda functions is measured by subtracting the minimum starting time among all executions belonging to a single

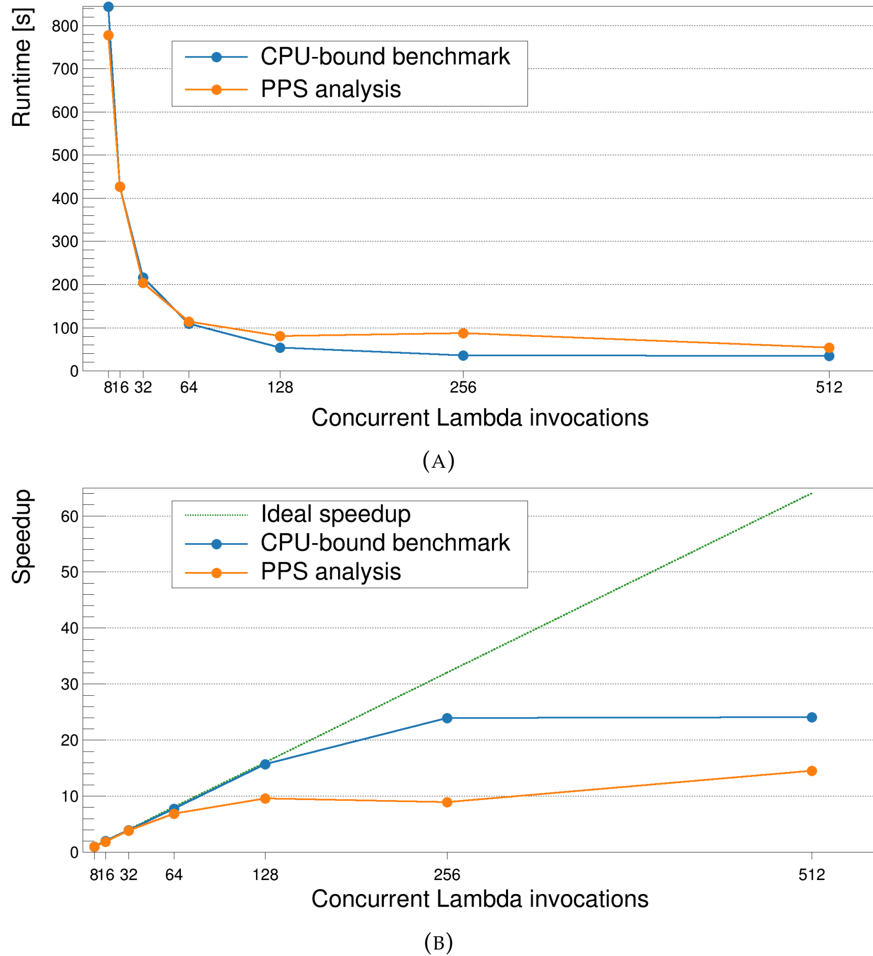


FIGURE 6.2: Comparison of runtime scaling of the CPU-bound benchmark and the PPS analysis, with an increasing number of Lambda invocations. (a): comparison of the absolute runtimes. (b): comparison of the speedup with respect to a linear increase.

test run from the maximum ending time within the same run. Figure 6.2 shows the behaviour of the system with both applications and an increasing number of Lambda invocations. Furthermore, the scalability of the tests in this work is subject to specific limitations imposed by AWS, which are highlighted further down in this section.

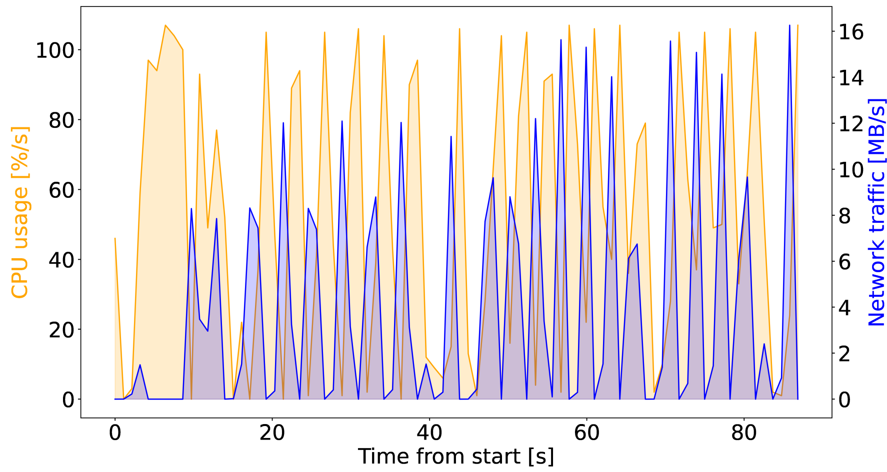
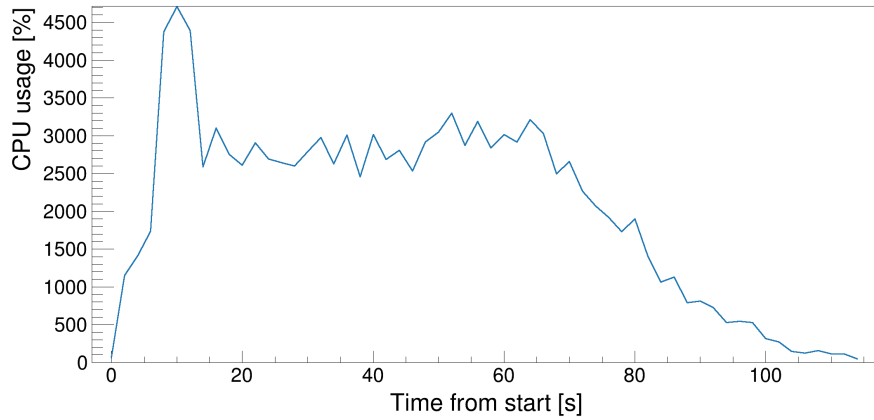


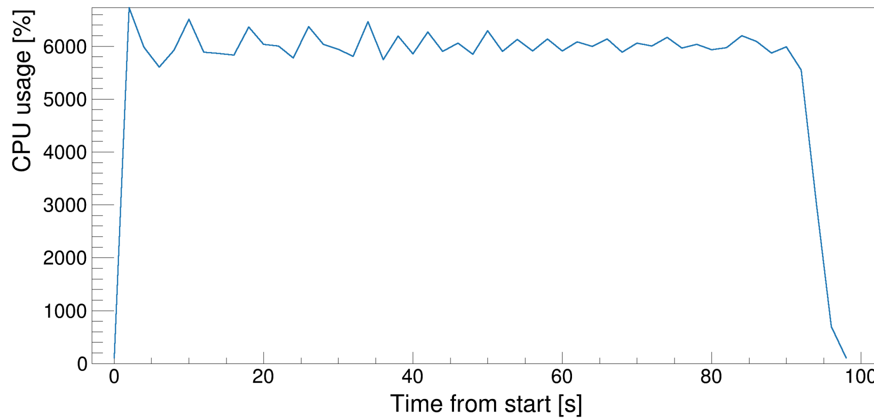
FIGURE 6.3: CPU usage (in orange) and network traffic (in blue) in a single Lambda execution running PPS analysis. This execution belongs to a run with 64 concurrent invocations.

Network and CPU usage patterns during Lambda execution Running the PPS analysis with the Lambda infrastructure requires streaming the pieces of the dataset needed for the analysis to the functions during their runtime. Given that a certain RDataFrame range can span one or more TTree clusters, these will need to be downloaded by the Lambda when they are needed for processing. The ROOT I/O streams a cluster of entries and processes them as they arrive, leading to the usage patterns shown in Figure 6.3. This figure focuses on a single Lambda invocation of the whole PPS analysis. During runtime, RDataFrame requests clusters of entries, thus triggering remote read requests that translate to higher spikes in network usage (in blue in the figure). Subsequently, the entries are processed leading to higher CPU usage (in orange in the figure). It can be seen that the spikes in the network and CPU usage alternate one another.

Aggregating the CPU percentage usage over all Lambda invocations of a particular PPS analysis run produces the pattern shown in Figure 6.4a. Initially, as all invocations finish downloading the first chunk of the dataset they need to process, a high spike in CPU usage is shown, reaching 74% of the total available 64 vCPUs. Consecutively, the CPU usage hovers around 40% until the first Lambda execution finishes and there is a corresponding drop that lasts until all functions finish their workload.



(A)



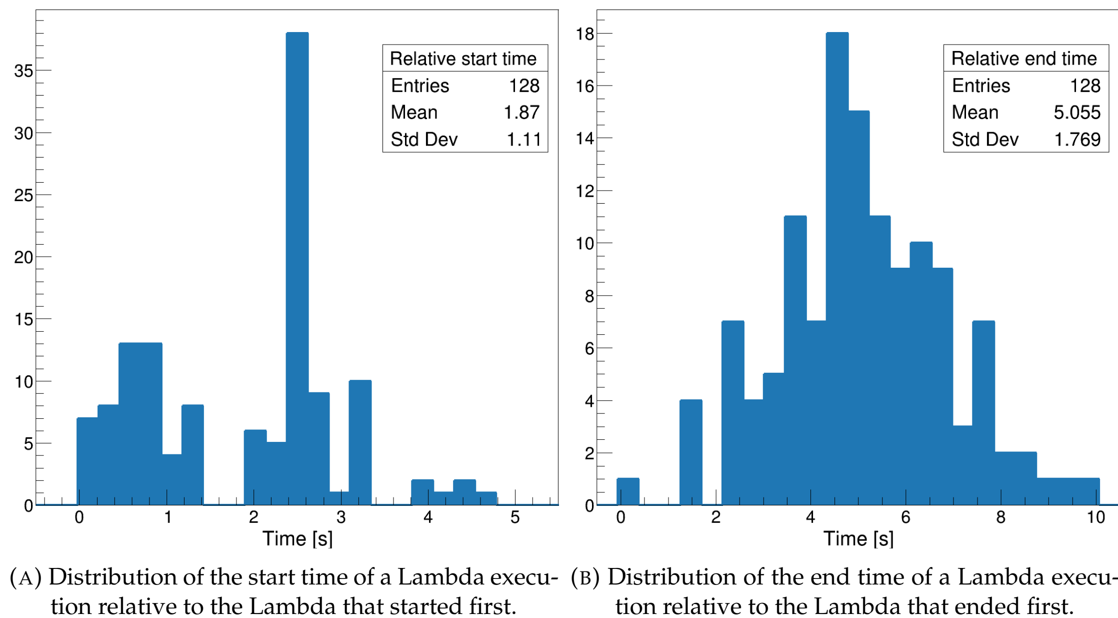
(B)

FIGURE 6.4: Aggregated CPU usage (in percentage) of 64 Lambda invocations. On the y axis of both figures, 100% percent corresponds to full utilisation of a single core. (a): PPS analysis. (b): CPU-bound benchmark.

The CPU-bound benchmark shows a different story. In Figure 6.4b the usage is always above 90% of the total vCPUs available until the end of the benchmark.

Variability in starting and ending time of Lambda executions The monitoring tool running in the Lambda execution also reveals delays in the starting and ending times

of different Lambda invocations in the same analysis run. With serverless computing, there is no direct control or access to the computing resources, and this means that the actual starting time of a certain computation after the Lambda has been invoked from the client can show some variability. Figure 6.5 shows that different Lambda invocations in the same test run can vary both in their starting time and in their ending times. In this particular CPU-bound run, the maximum delay between the first starting Lambda and the last one is around 4 seconds, while the maximum delay at the end is around 10 seconds. While quite limited, this delay is still noticeable and it is another peculiarity of the serverless workflow.



(A) Distribution of the start time of a Lambda execution relative to the Lambda that started first.

(B) Distribution of the end time of a Lambda execution relative to the Lambda that ended first.

FIGURE 6.5: Comparison of distribution for start and end Times for 128 concurrent Lambda invocations running the cpu-bound benchmark.

Real utilization of available resources on AWS Lambda The CPU-bound benchmark can be used to evaluate how efficiently the serverless invocation can use the underlying resources. Figure 6.6 shows that every invocation is able to use over 90% of allocated vCPU at all times. However, the actual time the analysis takes is two times as long

as that of the average execution, where the outliers are few and differ little from the average Lambda. This is due to the throttling imposed by AWS on the number of new synchronous invocations per second, leading to delays in the startup time when a large number of functions is invoked.

The PPS benchmark in the same figure shows greater variability in execution time, with some instances finishing much earlier than the few outliers. In this case, the overall analysis time is very similar to duration of the longest invocation.

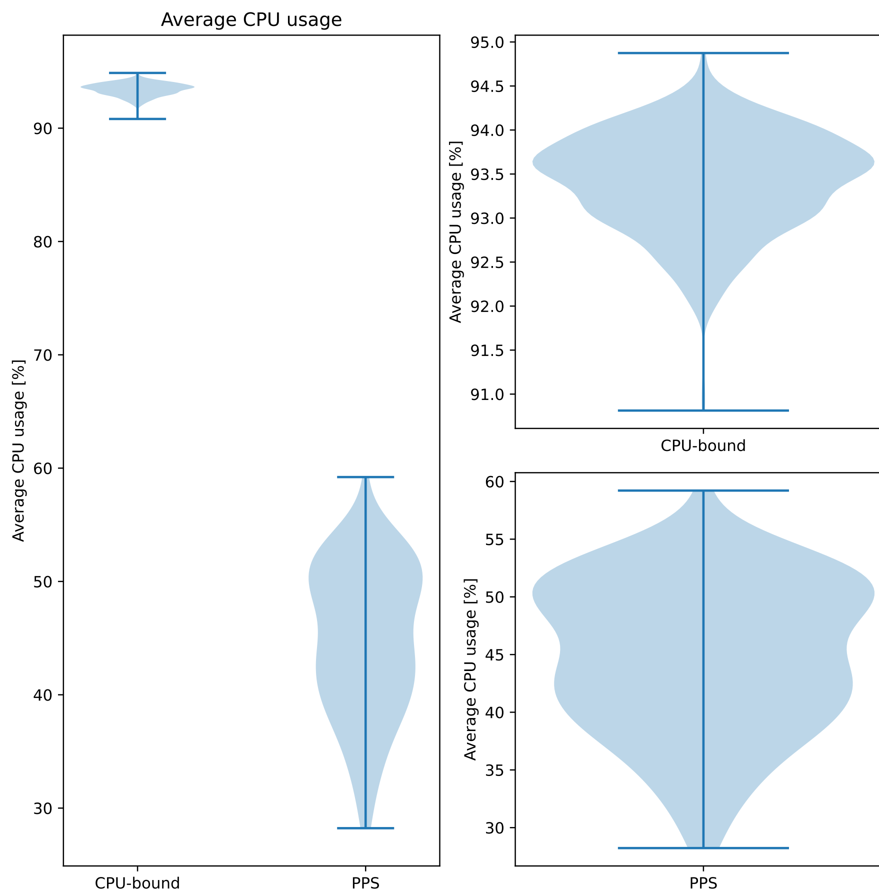


FIGURE 6.6: Comparison of average CPU usage for every Lambda in both analyses for 512 Lambdas. The left column shows both analyses, while the right shows closeups of both.

A run of the experiments with 512 invocations is shown in Figure 6.7. In the figure, each line represents the duration of a single invocation, sorted by starting time. The right side of the image shows invocations running the PPS analysis, showing some variability. Possibly, some instances stumble upon network slowdowns or slower I/O that results in a lower utilization of CPU on a particular Lambda. In comparison, the CPU-bound benchmark (left side of the image) shows a very uniform duration for all invocations. Thus, RDataFrame is able to saturate the CPU resources available on the AWS resources.

6.3.8 Discussion

Previous plots have shown that the new backend can make good use of the CPUs through the Lambda invocations, but there are several limitations imposed by AWS that have proven to be bottlenecks for proper execution. The choice of implementing synchronous calls to the AWS API is impacting the startup time with a limit of 10 invocations per second, although it allows for a fine-grained control over the status of the executions and allows to better act on errors.

Despite that, as seen on Figure 6.2, the scaling is promising, once the limitation on the number of invocations per second is lifted. While this may seem an issue if only looked at from the perspective of a single user running a single analysis, it is actually a demonstration of how the serverless approach would allow to fully utilize the underlying hardware. This can be seen for example with the execution pattern shown in Figure 6.7, which differs quite a lot from what HEP analysts are used to. In both the traditional HEP distributed computing and serverless scenarios, resources are shared among many users. Physicists are used to requesting resources by submitting jobs to a queue. Once the resources are granted, they can start using them. Similarly to the serverless case, some resources might be granted earlier, other resources later (although the difference in time between jobs starting could be much higher in a job queue system than on AWS). The key difference is that on an HPC cluster, it is the user's responsibility to utilise the resources they are granted as much as possible. Oftentimes though, cores are not utilised at 100% and this leads to an overall bad utilisation of the hardware. The AWS platform tries to optimise usage of resources by sharing them among as many

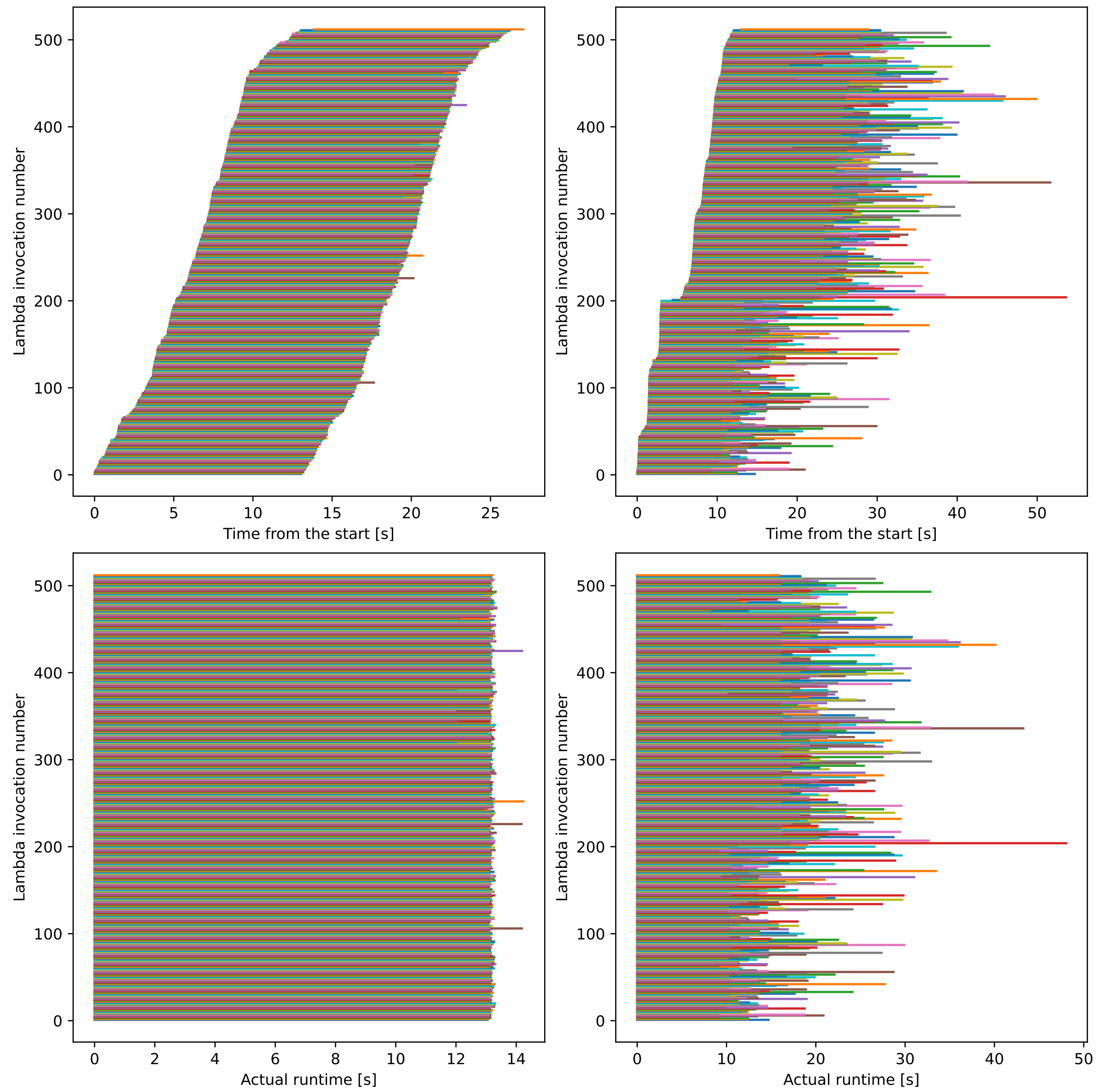


FIGURE 6.7: Comparison of runtime variability in executions. The top row shows the actual time of a single analysis computation, aligned to the beginning of first Lambda. The bottom row has every Lambda invocation start aligned at 0. The left column shows synthetic CPU benchmark, the right PPS analysis.

users as possible, for example letting users also request fractions of a CPU for their jobs, and at the same time forces users to take better care of their applications since they are paying for the amount of time they are using the resources.

Another important aspect to consider is data locality, where the stream of I/O coming directly from CERN causes the PPS analysis to stumble on the network connection between CERN and AWS data centre, as well as probably on CERN storage speed.

The variability both in starting and ending times of the mapper tasks, as shown in Figure 6.5, can be explained considering that the serverless workflow runs on non-dedicated resources. The AWS platform needs to sustain a very large amount of requests from many users and many geographical locations, thus the scheduling of Lambda invocations may show variability due to this kind of load.

6.4 Open source serverless framework for HEP analysis

Using the first implementation discussed in the previous section as a baseline, a new investigation focused on another serverless platform to address some of the previously found shortcomings. To start, the limitations imposed by the private cloud were hindering the research process. An open source platform could in principle provide a more direct control over the orchestration of the workflow. Furthermore, the previous effort did not explore the issue of having to launch only 10 mappers per second or the fact that the reduction phase was performed on the client side.

Thus, this section presents a new implementation of a serverless backend for distributed RDataFrame, which brings some improvements in the MapReduce workflow while also highlighting some common sources of overhead that can be common to many serverless platforms. The backend is based on OSCAR, described in Section 6.2.2.

6.4.1 Implementation of the backend

Similarly to the first work based on AWS Lambda, also in this case the implementation of the serverless backend is split in two parts, namely the client and the worker side.

RDataFrame backend on the client side

The implementation on the client side follows the usual schema. The distributed RDataFrame machinery generates a list of ranges, a mapper and a reducer. These need to be packaged as a single object and serialised so they can be sent to the remote workers. Sending the serialised objects represents the event that the serverless functions should act upon.

In the case of OSCAR, the events are file-driven, so the objects should be sent to some bucket on the platform, either a newly created one or a previously existing one. For simplicity, in the rest of the discussion the term "folder" may be used to refer to a bucket on the OSCAR platform.

For this first implementation, it was decided to use one bucket per application run. This bucket will be used a top-level "root folder" and more folders will be created inside of it to serve various parts of the workflow. Alternatively, multiple buckets per run could have been used, one per each part of the workflow that needs storage. Preliminary tests showed that there was no performance difference between using one bucket with a folder hierarchy and using multiple buckets, so the first option was chosen. The difference relates to configuration issues, certain properties and permissions. Therefore, the final decision on the configuration should be relegated to the requirements of the specific cluster deployment.

To identify an application, a universally unique identifier (UUID) is generated during the creation of the RDataFrame and then used as the name of the folder.

The serialised mapper and reducer are uploaded to a folder called `functions`, but this does not trigger the execution of any serverless function. Instead, the functions could be further reused if needed and the backend only needs to define a way to invoke them when the user asks for a result.

The act of invoking a function is a blocking call, the client will wait for the results. The final result can be written either to an external provider such as MinIO, Amazon S3, or ONEDATA, or to the cluster's own internal object store (which still uses MinIO underneath). In this work, the default system in OSCAR was chosen for storage of final results. Conveniently, MinIO provides a way to receive notifications of certain events such as the upload of a file to a specific location. Through this, the client application

does not need to poll at regular intervals for the status of the function executions. Instead, it can just wait for the notification from the storage service.

OSCAR services defined

The functions needed to perform the MapReduce approach must be defined within the OSCAR system. Access to the OSCAR services, and specifically to the MinIO buckets, requires specific access credentials, similar to AWS.

Mapper The mapper service listens to the `mapper-jobs` folder of the main bucket. Practically, each task generated on the client side will be written to this folder. At the time of invocation, the service reads the mapper function previously uploaded, deserialises it and applies it to its assigned range of entries. At the end of the mapper execution, the partial results are uploaded to the MinIO bucket in a folder called `partial-results`. An upload to this folder is an event for the reduction process.

Reducer The reduction phase is much less computationally intensive than the executing the mappers. Nonetheless, performing it in a stateless scenario is not a trivial problem. There is no central scheduler that can decide when the reduction phase should start and how the reducers should be run. The main issue to overcome is then how to check the state of execution of the MapReduce workflow, which sets of partial results should be merged together and in what order. Thus, two different reduction methods were explored and implemented in this work. They are better described in the next section.

Reduction strategies

Uncoordinated reduction In this model, each event of uploading a set of partial results from a mapper to MinIO will trigger the invocation of a reducer. The partial results are written to a file that is assigned an identifier, i.e. the index in the input list of ranges of the task that produced those results. The reduction phase follows a tree pattern. Sets of partial results are merged in pairs until only one set is left. Thus, an order can be established that takes into account the identifiers of the partial results. For every pair

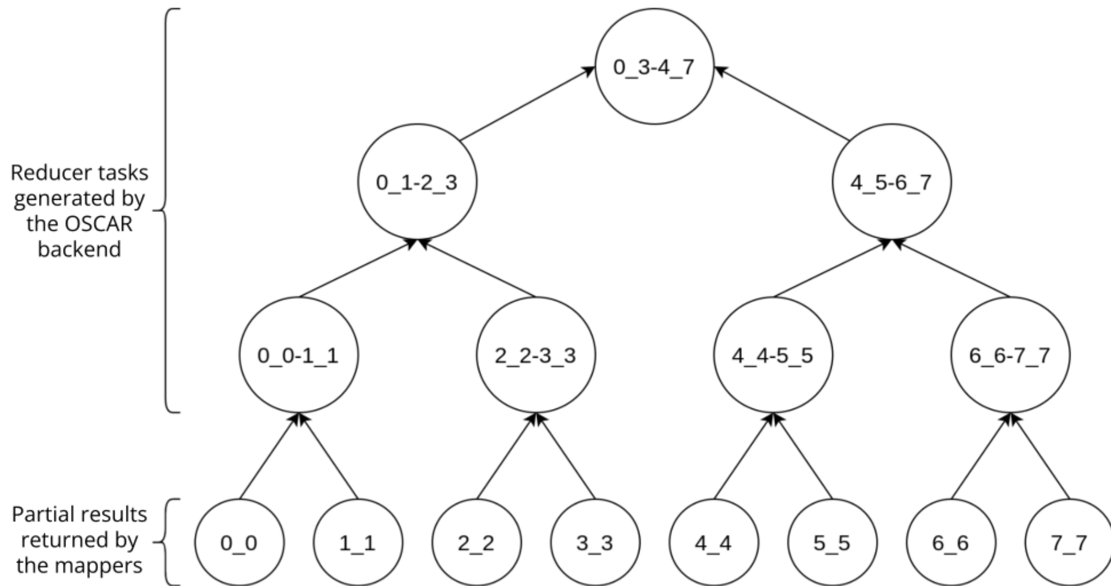


FIGURE 6.8: Generation of the identifiers for mappers and reducers in the uncoordinated reduction scenario.

of identifiers, a new file is created that contains both. Each file represents a reduction task. The files are stored in a separate folder named `reducer-jobs`. Upon completion of a mapper, the reduction task corresponding to its index will be triggered. If the partial results from the other mapper are already available, the task will proceed. Otherwise, that reduction task will not be executed. The reduction task that is triggered by the other assigned mapper index will then merge the partial results of the two mappers, now that both are available.

The algorithm used to determine the names of the files in the `reducer-jobs` folder is as follows. The MapReduce pattern is visualised as a binary tree. The leaves of the tree represent the mappers. All other nodes represent reduction tasks. The algorithm traverses the tree bottom-up, combining the names assigned to the mapper results two by two until reaching the root node. The name combination is done by taking the identifiers of the extremes that will match the minimum and maximum identifiers that have been reduced up to that point. Whenever a reducer is executed, it will write its merged results again in the `partial-results` folder, the same output folder as the

mappers. Once there is only one result file in the folder, it means that it is the set of final results that can be sent back to the user. An example of this name assignment is shown in Figure 6.8, assuming the analysis workload is split in 8 mapper tasks.

The main advantage of this approach is that it needs no extra process to manage the reduction phase, something which also does not naturally fit in the FaaS approach. There is also one disadvantage. If two mappers, which results should be later merged together, end at the same time, both will check whether there is already the counterpart set of partial results in the folder in order to trigger the corresponding reduction task. Consequently, two reductions will happen at the same time merging the same two sets of partial results. On the one hand, this is inconsequential to the results of the analysis, as the result of merging those two sets will be just one (the reducer that finishes last will overwrite the result of the previous reducer). On the other hand, it incurs in slightly more resources used. Figure 6.9 shows a schema of all the components involved in the uncoordinated reduction scenario.

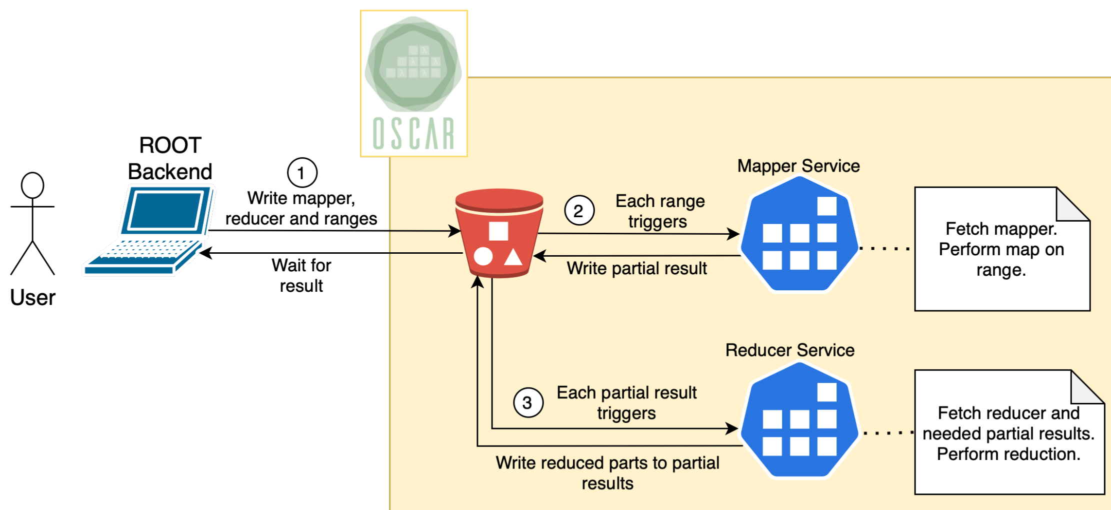


FIGURE 6.9: Component interaction in the uncoordinated reduction scenario. Numbers denote the order of the steps.

Coordinated reduction Another alternative consists in using a separate service which oversees the whole reduction phase. In this case, the RDataFrame backend invokes the

service at the beginning of the distributed execution, when the mappers are invoked. It also receives a list of integers when it is invoked, each value represents how many sets of partial results that should be merged by a single reducer invocation. The coordinator service will monitor the state of the mappers and the contents of the `partial-results` folder. Once there are at least as many partial results as the current value of the input list, the coordinator generates a reduction job that includes the names of all the mappers that it should merge. These names are assigned similarly to the uncoordinated reduction scenario and are also written in the `reducer-jobs` folder. Every time a reduction task is performed, the coordinator will pass to the next element in its input list.

An optimisation employed in this work is that the last reduction job is performed directly by the coordinator service, so that no extra time is spent waiting. With the coordinated reduction is not necessary to keep the order of the file names. Indeed, the coordinator can structure the process of reduction as convenient, e.g. into two parts or choosing an imbalanced splitting (80%-20%). It should be noted that since the reducer function only accepts two input parameters, these "multiple" reductions are performed by iterating over the assigned reduction tasks. Nonetheless, this avoids multiple reducer invocations and the limitations of the uncoordinated reduction approach. Figure 6.10 shows a schema of the coordinated reduction process.

Considerations on the implementation of the backend

Preliminary attempts of integrating the backend with OSCAR tried using the OSCAR API to call the mapper and reducer services. The final implementation instead accesses directly the files on MinIO to generate events. This choice conveniently provides the possibility to resume the execution at any point of failure during the MapReduce workflow.

Regarding the reduction strategies, in the uncoordinated reduction the jobs are written before the mappers start whilst in the coordinated one the jobs are written by the coordinator when necessary.

One of the current requirements of this backend is the need to define the number of mappers to be invoked for the analysis, similarly to the AWS Lambda backend. In the Spark and Dask backend a default value can be used which corresponds to the number

of available cores in the cluster. But in serverless platforms this information is not available at the time the services are invoked. In fact, the whole serverless approach relies on the idea that users can scale the number of function calls automatically depending on the amount of work provided. A possibility for future research would be to attempt at predicting the best number of functions that should be invoked, based on the workload given by the user application.

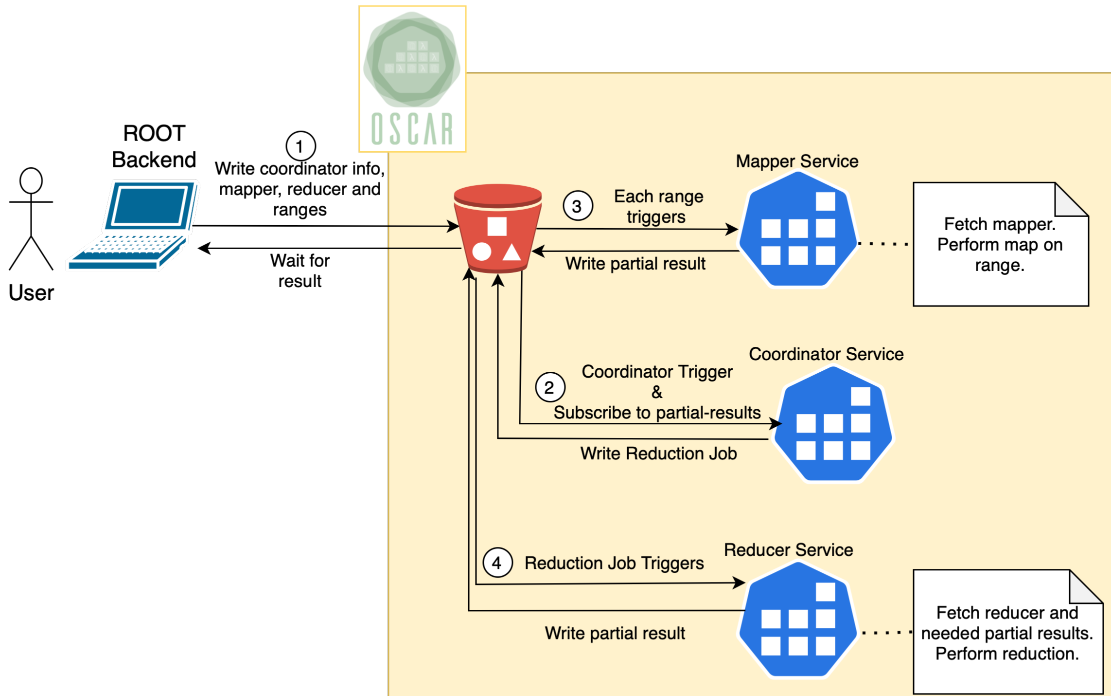


FIGURE 6.10: Component interaction in the coordinated reduction scenario. Numbers denote the order of the steps.

6.4.2 Experiments

Two types of analysis were employed as benchmarks in this work. The first type is an open data physics analysis, the same one used for the tests in Section 4.3. The dataset size for this experiment consists of roughly 200GB obtained from replicating one hundred times the original dataset file. The second type of analysis consists of a simulated

workload that reads no data from disk or network and can better highlight the best CPU usage the backend can drive. A simulated dataset is created at runtime in-memory, with roughly the same amount of data that would be processed in the other analysis.

Methodology

The experiments that will be described in this section all have to go through the usual two stages of the MapReduce scheme, plus some extra scheduling and orchestration that is more specific to OSCAR, such as container start and kill, interaction with the MinIO storage. The OSCAR services also contain monitoring code to evaluate resource usage and time spent during a mapper or a reducer.

The dimuon benchmark is carried out reading data from different locations: either from the CERN storage facility (EOS) or in the MinIO storage attached to the OSCAR cluster, much closer to the computing nodes.

For each of the three configurations (two for the dimuon benchmark, one for the cpu-bound), an increasing number partitions was studied: 1, 2, 4, 8, 16, 32, 48, 64, 80. Each partition corresponds to a task, thus to one invocation of the mapper service on OSCAR. The overall number of cores in the cluster would be 96, but in each node the Kubernetes service consumes a small amount of CPU thus hindering performance when all cores of a single VM are used.

As it was done in the work with AWS Lambda, the images needed by the serverless functions are cached on the cluster before the running the experiments to avoid cold starts.

At first, all the benchmarks are executed using the uncoordinated reduction approach. Subsequently, a comparison between the two reduction approaches will be demonstrated.

Hardware setup

A total of 6 working nodes were made available for the purposes of this work. Each working node is a virtual machine hosted on an OSCAR-enabled computing cluster, located in Valencia (Spain). Each node has a 16 physical cores Skylake Intel CPU (no hyperthreading) and 62.5 GB of RAM. The network connection among nodes of the

cluster is 10 Gb over ethernet. The object storage system (MinIO) is hosted on a dedicated SSD. Each OSCAR service created, i.e. the mapper and the reducer, is configured to invoke functions with 1 vCPU and 3 GB of RAM available.

Results

Figures 6.11 and 6.12 show, respectively, the time to plot (as defined in Section 1.6) and the relative speedup for all the experiments can be observed. As expected, the experiment with the data located within the cluster is faster than the experiment with the data located on EOS. For the two configurations of the dimuon benchmark, the speedup when reading data from CERN is higher than the speedup when reading data from MinIO. This is counterintuitive, as the MinIO storage is placed within the same network and the same machines that run the OSCAR services. Thus, probably a bottleneck in the I/O operation of the storage devices is hit in this case. The speedup in the dimuon benchmark remains constant when at least 48 cores are used, which may be due to network I/O. But a similarly poor scaling is shown also in the simulated workload example, reaching less than half of the optimal speedup as the core count increases. This opened an investigation into finding the possible causes of overhead in such serverless scenario.

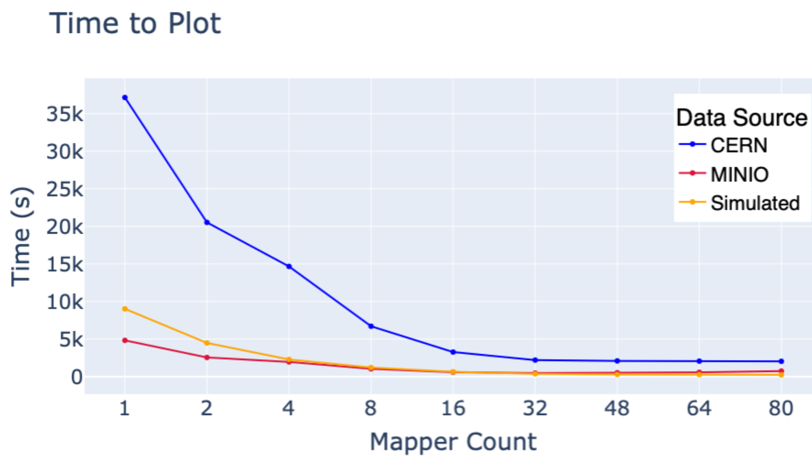


FIGURE 6.11: Time to plot with an increasing number of cores.

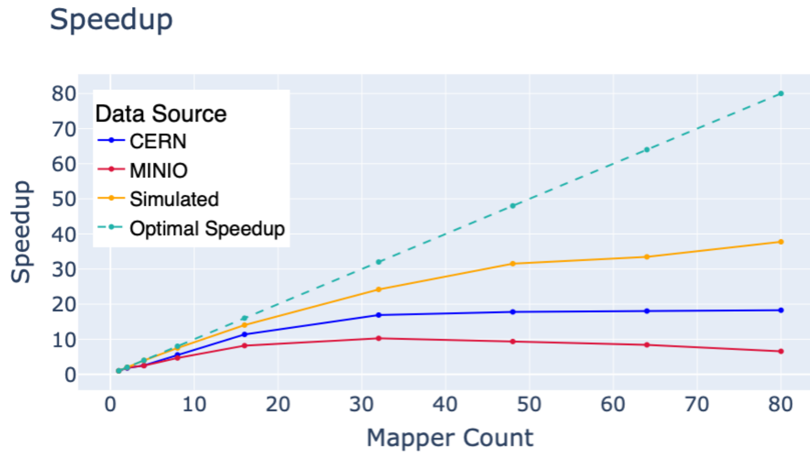


FIGURE 6.12: Speedup with an increasing number of cores.

Regarding the CPU and Memory utilization, Figure 6.13 shows the expected behaviour. The simulated workload has a CPU usage close to 100% and the experiment with the data stored in MinIO also makes a better usage of the CPU as it has to spend less time waiting for the data to arrive due to its locality. Looking at Figure 6.14 we can see that none of the experiments is close to reaching the 3 GB memory limit set, so we can discard this limit as a reason for the poor performance.

In order to better investigate the lack of scaling, the focus is shifted to the simulated workload. The other applications that read data may be influenced by I/O with the network or the local filesystem, so it is preferable to have a fully CPU bound workload to avoid any noise from other factors. Table 6.1 describes the difference in time between the fastest mapper and the slowest one for the simulated workload. In the table, the absolute time difference represents the difference between the runtime of the slowest mapper and the runtime of the fastest mapper. The relative time difference is computed with respect to the runtime of the fastest mapper. In this workload the difference should be within a reasonable margin (e.g. 2-3%), which may be justifiable due to noise from other processes on the nodes, but we can see that the difference in time reaches up to 29.45%. Figure 6.15 shows this information graphically. The same workload was run with distributed RDataFrame outside of the OSCAR cluster, on a single physical

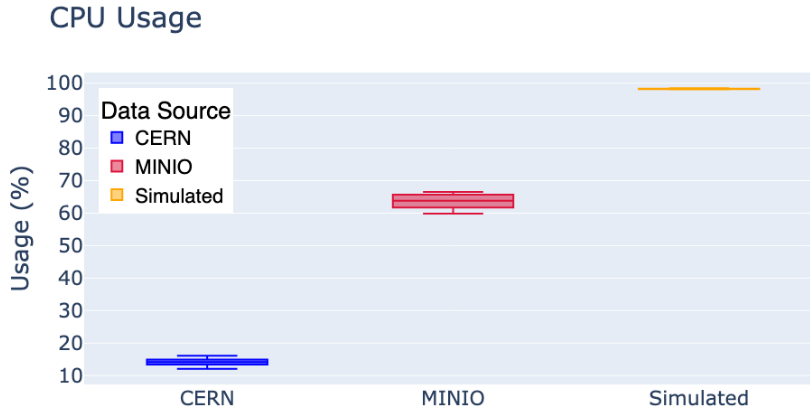


FIGURE 6.13: CPU usage of mapper invocations.

machine. The difference in time between the fastest and the slowest mapper in this controlled condition is on average 3%. With this information we can conclude that there are unknown sources of bottlenecks coming from the OSCAR cluster.

TABLE 6.1: Variability in the execution time of mappers during a simulated workload run.

Mapper Count	Absolute Time Difference [s]	Relative Difference Time [%]
1	0.0	0.0
2	119.0	2.79
4	154.0	7.41
8	105.5	10.03
16	63.5	11.72
32	40.5	13.64
48	32.5	15.44
64	48.0	29.45
80	39.0	27.76

Coordinated reduction For the comparison between the two coordinated strategies, only the CPU-bound benchmark was considered.

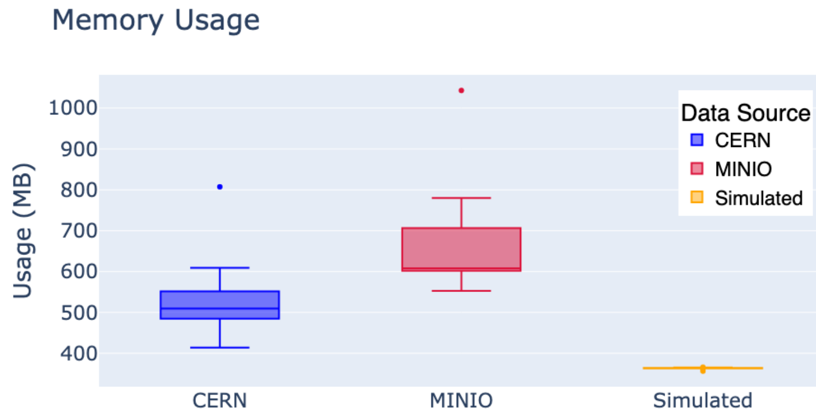


FIGURE 6.14: Memory usage of mapper invocations.

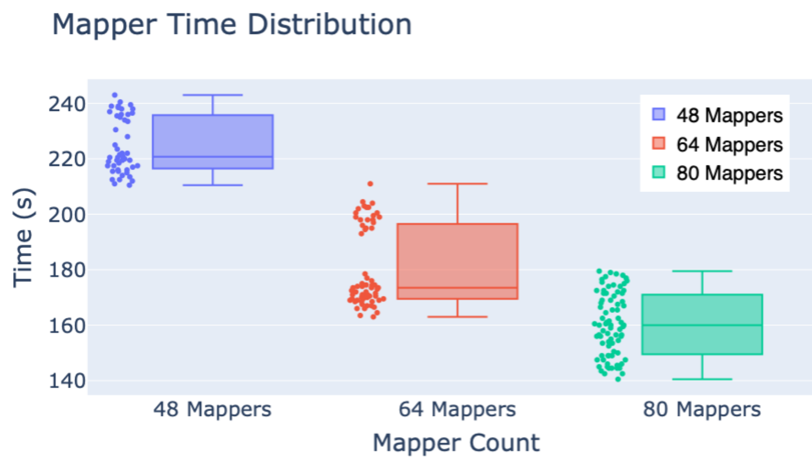


FIGURE 6.15: Distribution of the runtime of mapper invocations for 48, 64 and 80 mappers, respectively from left to right in the image.

A first test was performed which used exactly 80 mapper tasks, as that would create the highest number of subsequent reduction tasks. In an attempt to mitigate the overhead introduced by the extra scheduling needed for the serverless functions, a two-step reduction approach will be tried. This means that the coordinator will launch a reduction service to merge a determined amount of partial results, while the remaining

partial results will be merged by the coordinator itself. For this coordinated reduction three different configurations of load partitioning have been studied:

- 0%: all the reductions are performed by the coordinator.
- 50%: the invoked reducers process 50% of the reductions and the remaining reductions are performed by the coordinator
- 87.5%: the reducers are invoked for 87.5% of cases, the coordinator performs the remaining 12.5%.

Table 6.2 shows the results for this fine-tuning, all the configurations have similar results and the differences can be attributed to the runtime variability of the various mappers. For the following comparison with the uncoordinated reduction the 87.5% configuration will be used.

TABLE 6.2: Time to plot results for different workload partitioning of the coordinated reduction strategy.

Workload partitioning	Time to plot [s]
0%	205
50%	208
87.5%	202

The next test then runs the CPU-bound benchmark with an increasing amount of cores, the results of which can be seen in Figure 6.16. It is clear that, despite the variability of the system, the coordinated reduction is faster than the uncoordinated reduction, and this difference increases with the amount of mappers. This is due to the fact that for the uncoordinated reduction, the mapper that finishes last has to travel the entire path to the root of the tree invoking a service on each step. The greater the amount of mappers, the deeper the tree. This could be partially solved if instead of performing only one reduction in the uncoordinated reduction, the reducer also checks if it can perform any more reductions in the path to the root and also performs those automatically.

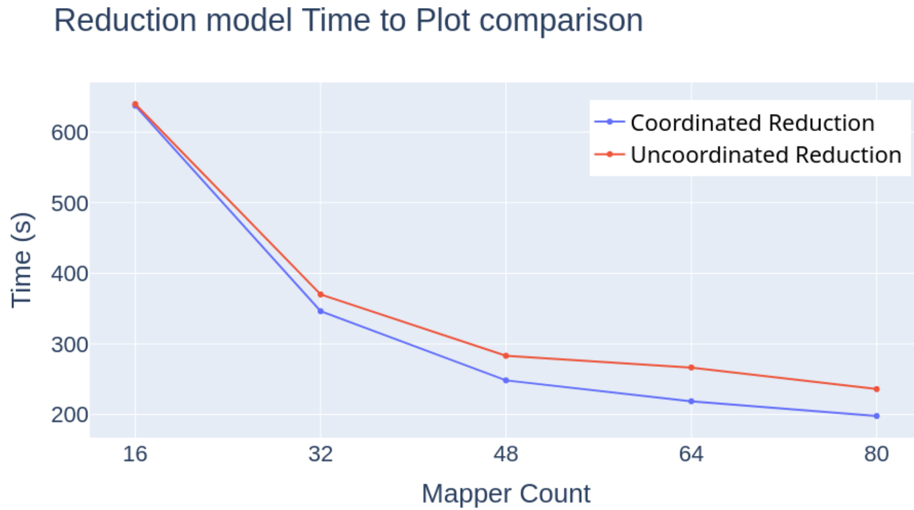


FIGURE 6.16: Time to plot comparison between using the coordinated reduction and the uncoordinated reduction patterns.

Discussion

The results presented in the previous section demonstrate that the distributed RDataFrame workflow can also be fitted in the file-driven serverless approach provided by OSCAR. CPU utilisation is close to 100% for all tasks when running the CPU-bound benchmark, and scales well also with data-locality, as the tests reading data from the MinIO object store result in higher utilisation than the tests reading data from CERN.

All the same, the scaling plots shown for example in Figure 6.12 show a worse scenario with respect to other works presented in this thesis. After a thorough examination of the resource usage and monitoring the status of the nodes on the cloud platform, it became clear that the overhead was due to bad performance of the underlying resources. To further demonstrate this claim, the CPU-bound benchmark was re-run on the same machines, using the Dask backend for distributed RDataFrame instead of the OSCAR one. The scalability of the Dask backend was already proven in previous works (see Chapter 4), so it provides a proper baseline. Since there are 6 nodes, an increasing number of cores per node (from one to fourteen) was used to run this test. The same number of nodes and cores per node was then used to run the test on the HPC cluster

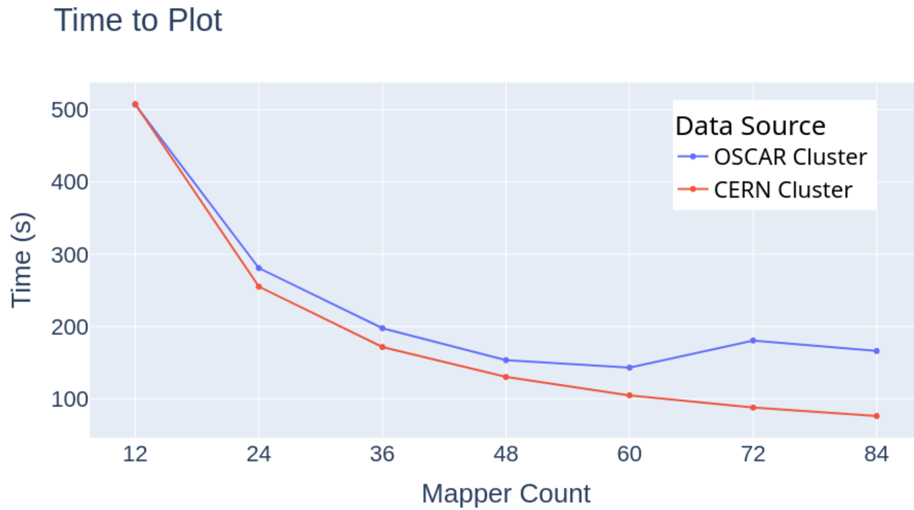


FIGURE 6.17: Time to plot comparison between using the Dask backend to run the CPU-bound benchmark on either the HPC cluster at CERN or the machines of the OSCAR cluster.

at CERN (the same one used for the experiments in Section 4.3). The comparison of the time to plot for these two configurations is shown in Figure 6.17. The plot focuses on core count starting from twelve and shows that for the higher core counts (sixty and onwards), the time to plot taken by the OSCAR cluster is almost double as the time taken by the CERN cluster. This is final proof of the issues that were found with the available hardware, while at the same time provide an interesting insight into the serverless computing scenario. It should be noted again that in all cases, either when using the OSCAR functions or in the tests with the Dask backends, the 6 nodes were actually virtual machines, on top of which a Kubernetes cluster was also running (used in the case of OSCAR, unused in the case of Dask). Thus, this suggests that careful consideration must be taken when using resources that build on top of multiple layers of virtualisation, which not always provides the stability usually taken for granted in HPC environments.

6.5 Conclusions

This chapter discussed the first efforts to fit the analysis layer available in ROOT in the context of serverless computing. This is still a very novel approach for HEP analysis in general, only one other similar effort was found in the literature review of Section 6.1. Two distinct research studies were developed with the aim of understanding the advantages and disadvantages of this new approach.

In the first work, RDataFrame computations were packaged and sent to be run on the AWS Lambda infrastructure. This could make use of Docker images to package ROOT and the needed code for distributed RDataFrame. Another advantage of using a Docker image is that all the needed libraries can be included there too, making this an easily customisable environment for users. The serverless functions defined in the cloud platform were made readily available to the tested benchmarks by executing them before testing, so that the Amazon engine would cache them. A real physics analysis was employed to also drive a first investigation on the use of authorization tokens in such scenario, which require careful considerations. In this work, the token lived as long as the function invocation, being destroyed immediately afterwards. The engine presented shows a promise for deployment at large scale as an alternative to a typical on-premises analysis run on the Grid. The results are promising, as the engine was capable of fully utilizing the AWS resources available during function execution and the only real bottleneck was the limit in Lambda invocations per second. Another interesting takeaway from that work was that, through Cling, it practically enabled a new C++ runtime environment for AWS, which is not supported natively by the platform. Furthermore, it showed that a better overall resource utilization of resources can be potentially achieved by serverless platforms, with a combination of sharing resources with as many users as possible (similarly to traditional HEP distributed systems) and an increased incentive for users to optimise their resource utilisation since they pay based on the CPU time used.

In the second work, the distributed RDataFrame machinery was connected to a different serverless platform, open source and with focus on file-driven applications, named OSCAR. This time, the invocation of mappers and reducers had to go through the creation of small files on the object store attached to the cloud platform, namely

MinIO. The reduction phase was explored further, with the definition of two reduction strategies, uncoordinated and coordinated. Both cases provide a way to push the reduction phase to the cluster, rather than keeping it on the client side as was done in the previous effort. Being OSCAR still in an experimental phase, some areas of improvement were found and discussed with its developers. In particular, the asynchronous function invocations rely on the underlying Kubernetes batch job scheduler and the layers of virtualisation seemed to provide a tangible overhead. This sparked a few interesting research questions regarding the possible sources of overhead of such a pattern with respect to classic MapReduce. It was highlighted that spawning the serverless functions and writing the partial results to buckets lead to additional steps in the workflow that can hinder the parallelisation offered by the RDataFrame backend.

Chapter 7

Conclusions and future work

This thesis was devoted to investigating a possible new solution for distributed and interactive data analysis in High Energy Physics. The distributed computing scenario in this field has been traditionally characterised by running large-scale complex applications on batch computing systems which are the only type of resource manager employed within the WLCG. For the specific case of the analysis, the last step in a long data pipeline, users are effectively compelled to write their applications in separate parts, manually submitting the computations as jobs that process different parts of the input dataset to the queue. Thus, it is an inherently non-interactive approach. More modern approaches that leverage interactive distributed execution engines, which have become popular thanks to the Big Data phenomenon and find ample use in the larger Data Science community, can provide in fact a much smoother experience also to HEP analysts.

The work examined in this thesis thus focused on contributing to the improvement of final analysis workflows in the field, also taking into account the fact that future computing requirements of the LHC will increase by one or two order of magnitudes with respect to the past. This aim can be approached with a new analysis interface that provides high-level facilities for users to express the physics computations and at the same time seamlessly distributes them with no change in code. A solid baseline was identified within ROOT RDataFrame, an already established analysis tool that features a declarative programming interface. This tool was augmented with a distributed execution machinery that is able to take existing applications and parallelise them on multiple nodes of a cluster. Distributed RDataFrame takes care of automatically splitting

the input dataset in logical chunks, sending each one as a payload for a different task on the computing nodes. This extension is built with modularity in mind, so that different input data formats and different execution backends can be fitted in the package.

Different aspects of distributed RDataFrame have been put to the test. First, the scalability that the users can expect from this tool, leveraging existing batch computing resources while keeping the more modern interface. Tests using open data benchmarks were run on up to 2048 cores showing an almost linearly increasing speedup with both the Apache Spark and Dask backends. This result is very promising for future physics computing workflows, opening the door for physicists to use many more resources to run their analysis without sacrificing ease of use and programmability, a crucial aspect for the HL-LHC era. In particular, the Dask backend also offers the highly valuable capability to directly connect and submit jobs to the batch systems that are commonly employed in the WLCG and other HEP computing facilities.

The same backend was then used by a team of researchers to develop a full-scale CMS analysis with distributed RDataFrame. The results were remarkable, showing an order of magnitude increase in speedup with respect to the legacy approach. The collaboration with these users highlighted some existing bottlenecks and missing features, which led to further developments of the software tool.

The typical data analysis workflow in HEP is I/O bound, with large datasets usually being read over the network or shared filesystems. Therefore, in order to make the new model of interactive distributed analysis viable and effective for final users, it should adapt to and rely on I/O optimisations that already play an important role in traditional analysis systems, notably caching. A first evaluation of existing technologies, found in the XRootD library and in ROOT, with different caching strategies highlighted the main requirements that such mechanism should fulfill, in particular with regards to storing only the portions of the dataset that are actually read. The programming model already offered by RDataFrame nicely plays with respect to that need, since it can examine what properties of the dataset the analyst is using and read only those. In addition, distributed computations that operate on different ranges of entries of the input dataset can be instructed to read strictly the entries required. Another study

regarding the same topic went a step further, employing technologies beyond the traditional caching stack seen in HEP. The aim was to switch from file-based caching to exploiting bleeding-edge object stores backed by highly-optimised HPC hardware. This study focused on the next-generation I/O layer offered by RNTuple, using Intel DAOS as the object store technology for caching. The developed system was actually independent of the storage backend, which enabled the possibility to read a file remotely and store it in different target caching systems, be it the object store itself or the local disk. This same mechanism can be reused for storage backends that will be supported in the future by RNTuple. The throughput obtained when reading data that was cached on the Intel DAOS cluster during a distributed RDataFrame execution achieved 37 GB/s, a remarkable achievement since it is anywhere from 2 times to an order of magnitude faster compared to benchmarks of the same type running on traditional I/O technologies. Furthermore, that result can still be improved considering that it is equal to 74% of the nominal throughput available on the cluster.

Although computing resources in the field are usually managed, on-premises infrastructures, the research of this thesis went beyond that and also evaluated a different paradigm: serverless computing in the cloud. Cloud resources have the potential of scaling better considering the future computing needs in HEP. This thesis thus included as well a few studies on the FaaS paradigm applied to distributed RDataFrame. Two backends were developed, one based on AWS Lambda and another on OSCAR. The benchmarks run with these new execution engines demonstrated a very promising scalability, although still lower than other benchmarks run with Dask or Spark on traditional dedicated infrastructures (HPC or batch computing facilities). Moving to the serverless approach also introduces a few extra scheduling steps that are usually not required, mainly related to the communication between the client and the serverless platform, or between the mappers, reducers and the storage layer used to pass around the information regarding their status and their results. Nonetheless, the serverless platforms can potentially lead to better overall resource utilisation, especially in multi-tenant scenarios, due to both the optimisations carried out by the overlay services and the model of paying per amount of CPU time used which increases users' awareness of their resource usage.

7.1 Publications

The contributions brought by this thesis have been included in different publications during its development. This section lists published articles, conference proceedings and presentations related to the work done for this thesis, grouped by the different research topics that were discussed in previous chapters.

Design of a programming model for distributed analysis in HEP The studies on addressing the limitations of traditional HEP distributed computing led to the development of distributed RDataFrame, a package to seamlessly distribute physics analyses efficiently, while keeping the already established user API intact. I was responsible for the development of this tool during the PhD. It was first mentioned, still in a prototype state and not integrated natively within RDataFrame, at the *24th International Conference on Computing in High Energy and Nuclear Physics (CHEP 2019)*. *CHEP* is the largest venue for sharing research regarding computing, network and software in the field, with hundreds of contributions presented at each edition. The following peer-reviewed conference proceedings are available:

- Vincenzo Eduardo Padulano, Javier Cervantes Villanueva, Enrico Guiraud and Enric Tejedor Saavedra. Distributed data analysis with ROOT RDataFrame. *In: 24th International Conference on Computing in High Energy and Nuclear Physics (CHEP 2019)*. `https://doi.org/10.1051/epjconf/202024503009`, 2022.

After I developed a new backend for distributed RDataFrame which leverages Dask as the distributed execution engine, I presented it at the Dask Summit 2021:

- Vincenzo Eduardo Padulano (speaker), Enric Tejedor Saavedra. Dask backend for distributed RDataFrame. *In: Dask in High-Energy Physics community workshop, Dask Distributed Summit 2021*. `https://summit.dask.org/schedule/presentation/24/dask-in-high-energy-physics-community`, 2021.

The first presentation of the tool being natively integrated with the main ROOT project was done at the *PyHEP 2021* workshop. *PyHEP* is a series of workshops providing a large, international venue for discussing the research regarding Python usage in the HEP field. The contribution regarding distributed RDataFrame follows:

- Vincenzo Eduardo Padulano (speaker), Enric Tejedor Saavedra. A Python package for distributed ROOT RDataFrame analysis. *In: PyHEP 2021*. https://indico.cern.ch/event/1019958/contributions/4419751/, 2021.

Some of the latest developments in terms of algorithms for task creation and scheduling are included in the contribution to *Journal of Grid Computing* mentioned in the next paragraph.

Efficient distribution of physics computations This area of research focused on demonstrating the scalability of distributed RDataFrame. A comparison of the Spark and Dask backend was performed on a Slurm-managed computing cluster at CERN. This contribution has been published in the *Journal of Grid Computing*:

- Vincenzo Eduardo Padulano, Ivan Donchev Kabadzhov, Enric Tejedor Saavedra, Enrico Guiraud, Pedro Alonso-Jordá. Leveraging state-of-the-art engines for large-scale data analysis in High Energy Physics. https://doi.org/10.1007/s10723-023-09645-2, 2023.

Another work was carried out in collaboration with a team of physics researchers of the CMS experiment at CERN. In this collaboration, a production-grade analysis was written with the RDataFrame API and distributed through the Dask backend on the resources of a grid site. There is currently no peer-review publication regarding this effort, the relative paper is being written and is pending submission to a scientific journal. A presentation at the latest *PyHEP* workshop is available:

- Tommaso Tedeschi, Vincenzo Eduardo Padulano (speakers). Lessons learned converting a production-grade Python CMS analysis to distributed RDataFrame. *In: PyHEP 2022*. https://indico.cern.ch/event/1150631/contributions/5002793/, 2022.

Fine-grained caching of physics data In this part of the thesis the focus was oriented towards evaluating caching mechanisms for the analysis layer. I carried out a comparison between existing technologies and different caching techniques, which was

presented at *CHEP 2021*. My presentation was later published in the following peer-reviewed conference proceedings:

- Vincenzo Eduardo Padulano, Enric Tejedor Saavedra and Pedro Alonso-Jordá. Fine-grained data caching approaches to speedup a distributed RDataFrame analysis. *In: 25th International Conference on Computing in High Energy and Nuclear Physics (CHEP 2021)*. https://doi.org/10.1051/epjconf/202125102027, 2021.

Later on, I developed and tested a new caching mechanism within RNTuple, with benchmarks that used Intel DAOS as the caching system. This contribution was published in the *Journal of Cluster Computing*:

- Vincenzo Eduardo Padulano, Enric Tejedor Saavedra, Pedro Alonso-Jordá, Javier López Gómez and Jakob Blomer. A caching mechanism to exploit object store speed in High Energy Physics analysis. https://doi.org/10.1007/s10586-022-03757-2, 2022.

Serverless computing for HEP data analysis workflows Another topic of research in this thesis was the serverless computing paradigm and how it can potentially be applied to the HEP analysis use case. Two new backends for distributed RDataFrame were developed and tested in this regard. A first implementation leveraged the AWS Lambda service to run the serverless functions. It featured a synchronous invocation mechanism and the reduction stage was performed locally on the client side. Two distinct publications contribute to this part of the research:

- Jacek Kuśnierz, Maciej Malawski, Vincenzo Eduardo Padulano, Enric Tejedor Saavedra, Pedro Alonso-Jordá. Distributed Parallel Analysis Engine for High Energy Physics Using AWS Lambda. *In: HiPS '21: Proceedings of the 1st Workshop on High Performance Serverless Computing*. https://doi.org/10.1145/3452413.3464788, 2021.

- Jacek Kuśnierz, Vincenzo Eduardo Padulano, Maciej Malawski, Kamil Burkiewicz, Enric Tejedor Saavedra, Pedro Alonso-Jordá, Michael Pitt, Valentina Avati. A Serverless Engine for High Energy Physics Distributed Analysis.

In: 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid). https:://doi.org/10.1109/CCGrid54584.2022.00067, 2022.

The other backend leveraged OSCAR, an open source serverless platform aimed at large-scale file-driven applications. The following contribution also introduced two different distributed reduction strategies which were compared. This work has been published in the *Journal of Supercomputing*:

- Vincenzo Eduardo Padulano, Pablo Oliver Cortés, Pedro Alonso-Jordá, Enric Tejedor Saavedra, Sebastián Risco, and Germán Moltó. Leveraging an open source serverless framework for High Energy Physics computing. https://doi.org/10.1007/s11227-022-05016-y, 2023.

7.2 Future work

The HEP landscape is rich with computational challenges already today, that will clearly become even more complex with the beginning of the HL-LHC era. This thesis contributes towards the collective effort of meeting those challenges, specifically in the data analysis scenario. As such, many more improvements and lines of research could be derived from the work that has been presented. For instance, the scalability of the tool must be demonstrated with many more different analyses, searching for different data access patterns and computation graphs of varying complexity in final user code. At the same time, running it on different infrastructures that may be leveraged through either Spark or Dask could give the tool further solidity with respect to the different computing resources available to HEP research groups, thus also maximising the portability and reproducibility of the analysis.

It would be also interesting to add some remaining steps of the full analysis workflow to the distributed execution, in particular the machine learning inference which is more and more commonly employed by physicists. Possibly, some parts of the distributed execution could be offloaded asynchronously to computing accelerators.

The serverless paradigm needs to be studied further in order to evaluate if it can be of effective assistance to the already available computing resources in the field, for

example to supply further resources during peaks of high demand. In particular, a thorough evaluation of its cost efficiency is needed. The implementation of the MapReduce pattern using only serverless functions demonstrated that it is possible to rely fully on this model for single analysis executions, but it would be necessary to remove the bottlenecks detected in the current implementation, which make it scale worse than the distributed execution engines running on managed resources presented.

Bibliography

[1] CERN. LHC Schedule. https://hilumilhc.web.cern.ch/content/hl-lhc-project. Accessed on 2022-12-20.

[2] ATLAS Collaboration. ATLAS Software and Computing HL-LHC Roadmap. Technical report, CERN, Geneva, 2022. URL: https://cds.cern.ch/record/2802918.

[3] CMS Offline Software and Computing. CMS Phase-2 Computing Model: Update Document. Technical report, CERN, Geneva, 2022. URL: https://cds.cern.ch/record/2815292.

[4] ATLAS Collaboration. Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC. *Physics Letters B*, 716(1):1–29, 2012. doi:https://doi.org/10.1016/j.physletb.2012.08.020.

[5] CMS Collaboration. Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC. *Physics Letters B*, 716(1):30–61, 2012. doi:https://doi.org/10.1016/j.physletb.2012.08.021.

[6] WLCG. Tier Centres. http://wlcg-public.web.cern.ch/tier-centres. Accessed on 2022-12-20.

[7] Concezio Bozzi. LHCb Computing Resource usage in 2018. Technical report, CERN, Geneva, 2019. URL: https://cds.cern.ch/record/2657833.

[8] Jakob Blomer, Philippe Canal, Axel Naumann, and Danilo Piparo. Evolution of the ROOT Tree I/O. In *24th International Conference on Computing in High Energy and Nuclear Physics (CHEP 2019)*, volume 245, November 2020. doi:https://doi.org/10.1051/epjconf/202024502030.

[9] Ian Bird et al. Update of the Computing Models of the WLCG and the LHC Experiments. Technical report, WLCG Collaboration, 2014.

[10] U.S.A. Department of Energy - Office of Science. High Energy Physics. https://www.energy.gov/science/hep/high-energy-physics, 2022. Accessed on 2022-10-20.

[11] CERN. The Standard Model. https://home.cern/science/physics/standard-model, 2022. Accessed on 2022-10-20.

[12] CERN. https://home.cern/, 2022. Accessed on 2022-12-20.

[13] CERN. The Large Hadron Collider. https://home.cern/science/accelerators/large-hadron-collider, 2022. Accessed on 2022-12-20.

[14] ATLAS Experiment. https://atlas.cern/, 2022. Accessed on 2022-12-20.

[15] CMS Experiment. https://cms.cern/, 2022. Accessed on 2022-12-20.

[16] ALICE Experiment. https://alice-collaboration.web.cern.ch/, 2022. Accessed on 2022-12-20.

[17] LHCb Experiment. https://lhcb.web.cern.ch/lhcb/, 2022. Accessed on 2022-12-20.

[18] CERN. End of LHC Run 1: First shutdown begins. https://timeline.web.cern.ch/end-lhc-run-1-first-shutdown-begins, 2013. Accessed on 2022-12-20.

[19] Mélissa Gaillard. CERN Data Centre passes the 200-petabyte milestone. https://home.cern/news/news/computing/cern-data-centre-passes-200-petabyte-milestone, 2017. Accessed on 2022-12-20.

[20] Esra Ozcesmeci. LHC: pushing computing to the limits. https://home.cern/news/news/computing/lhc-pushing-computing-limits, 2017. Accessed on 2022-12-20.

[21] Arsuaga-Rios, Maria, Bahyl, Vladimír, Batalha, Manuel, Caffy, Cédric, Cano, Eric, Capitoni, Niccolo, Contescu, Cristian, Davis, Michael, Fernandez Alvarez, David, Guenther, Jaroslav, Karavakis, Edouardos, Keeble, Oliver, Leduc, Julien, Luchetti, Fabio, Mascetti, Luca, Murray, Steven, Patrascoiu, Mihai, Peters, Andreas, Kamil Simon, Michal, Sindrilaru, Elvin, and Toebbicke, Rainer. LHC Data Storage: Preparing for the challenges of Run-3. *EPJ Web Conf.*, 251:02023, 2021. doi:10.1051/epjconf/202125102023.

[22] Adrian Cho. The discovery of the Higgs boson. *Science*, 338(6114):1524–1525, 2012. URL: https://www.science.org/doi/abs/10.1126/science.338.6114.1524, arXiv:https://www.science.org/doi/pdf/10.1126/science.338.6114.1524, doi:10.1126/science.338.6114.1524.

[23] Apollinari G., Béjar Alonso I., Brüning O., Fessia P., Lamont M., Rossi L., and Tavian L. High-Luminosity Large Hadron Collider (HL-LHC) : Technical Design Report V. 0.1. Technical report, CERN, 2017.

[24] ALICE. ALICE records the first lead-ion collisions of Run 3. https://alice-collaboration.web.cern.ch/leadtest1, November 2022. Accessed on 2022-12-20.

[25] Panos Charitos. ALICE upgrade plans. https://ep-news.web.cern.ch/content/alice-upgrade-plans-0. Accessed on 2022-12-20.

[26] Mark Whitehead. The upgrade of the LHCb trigger for Run iii. In *The European Physical Society Conference on High Energy Physics*, 07 2017.

[27] Ludovico Pontecorvo. ATLAS upgrades in LS2. https://cerncourier.com/a/atlas-upgrades-in-ls2/. Accessed on 2022-12-20.

[28] Panos Charitos. CMS upgrade plans for LS2. http://ep-news.web.cern.ch/content/cms-upgrade-plans-ls2. Accessed on 2022-12-20.

[29] CERN. High Luminosity LHC Project. https://hilumilhc.web.cern.ch/. Accessed on 2022-12-20.

[30] John L. Gustafson. *Moore's Law*, pages 1177–1184. Springer US, Boston, MA, 2011. URL: /https://doi.org/10.1007/978-0-387-09766-4_81, doi:10.1007/978-0-387-09766-4_81.

[31] Danilo Piparo. Scientific Software and Computing in the HL-LHC, EIC, and Future Collider Era. In *21st International Workshop on Advanced Computing and Analysis Techniques in Physics Research*, 2022. URL: https://indico.cern.ch/event/1106990/contributions/5021254/.

[32] 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 2022-12-20.

[33] Corinne Pralavorio. Lhc performance reaches new highs. https://home.cern/news/news/accelerators/lhc-performance-reaches-new-highs. Accessed on 2022-12-20.

[34] Corinne Pralavorio. Record luminosity: well done LHC. https://home.cern/news/news/accelerators/record-luminosity-well-done-lhc. Accessed on 2022-12-20.

[35] Mike Lamont. An encouraging start for Run 3. https://home.cern/news/opinion/accelerators/encouraging-start-run-3, August 2022. Accessed on 2022-12-20.

[36] Paul Rincon. Work starts to upgrade Large Hadron Collider. https://www.bbc.com/news/science-environment-44484062. Accessed on 2022-12-20.

[37] Davis, Michael C., Bahyl, Vladímir, Cancio, Germán, Cano, Eric, Leduc, Julien, and Murray, Steven. CERN Tape Archive - from development to production deployment. *EPJ Web Conf.*, 214:04015, 2019. doi:10.1051/epjconf/201921404015.

[38] G Petrucciani, A Rizzi, C Vuosalo, and on behalf of the CMS Collaboration. Mini-AOD: A New Analysis Data Format for CMS. *Journal of Physics: Conference Series*,

664(7):072052, dec 2015. URL: https://dx.doi.org/10.1088/1742-6596/664/7/072052, doi:10.1088/1742-6596/664/7/072052.

[39] Marco Peruzzi, Giovanni Petrucciani, Andrea Rizzi, and for the CMS Collaboration. The NanoAOD event data format in CMS. *Journal of Physics: Conference Series*, 1525(1):012038, apr 2020. URL: https://dx.doi.org/10.1088/1742-6596/1525/1/012038, doi:10.1088/1742-6596/1525/1/012038.

[40] Elmsheuser, Johannes, Anastopoulos, Christos, Boyd, Jamie, Catmore, James, Gray, Heather, Krasznahorkay, Attila, McFayden, Josh, Meyer, Christopher John, Sfyrla, Anna, Strandberg, Jonas, Suruliz, Kerim, and Theveneaux-Pelzer, Timothee. Evolution of the ATLAS analysis model for Run-3 and prospects for HL-LHC. *EPJ Web Conf.*, 245:06014, 2020. doi:10.1051/epjconf/202024506014.

[41] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 1st edition, May 2012. URL: https://www.elsevier.com/books/the-art-of-multiprocessor-programming-revised-reprint/herlihy/978-0-12-397337-5.

[42] K. N. King. *C Programming: A Modern Approach*. Wiley, 2nd edition, April 2008. URL: https://www.wiley.com/en-au/C+Programming%3A+A+Modern+Approach%2C+2nd+Edition-p-9780393979503.

[43] Rainer Wanke. *How to Deal with Systematic Uncertainties*, chapter 8, pages 263–296. John Wiley & Sons, Ltd, 2013. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9783527653416.ch8, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/9783527653416.ch8, doi:https://doi.org/10.1002/9783527653416.ch8.

[44] Dimitri Bourilkov. Machine and deep learning applications in particle physics. *International Journal of Modern Physics A*, 34(35):1930019, 2019. doi:10.1142/S0217751X19300199.

[45] Roger Barlow. *Fundamental Concepts*, chapter 1, pages 1–26. John Wiley & Sons, Ltd, 2013. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/

9783527653416.ch1, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/9783527653416.ch1, doi:https://doi.org/10.1002/9783527653416.ch1.

[46] Ian Foster and Carl Kesselman. *The Grid : Blueprint for a New Computing Infrastructure*. Morgan Kaufmannm publishers, 1998.

[47] Wolfgang Gentzsch. Response to Ian Fosters "what is the grid?". *Grid Today*, 1(8):5, 2002.

[48] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, 15, 2001.

[49] WLCG. Homepage. http://wlcg.web.cern.ch/. Accessed on 2022-12-20.

[50] Ian Bird. Computing for the Large Hadron Collider. *Annual Review of Nuclear and Particle Science*, 61, 2011.

[51] Ian Foster. What is the grid? a three point checklist. *GRID today*, 1:32–36, 01 2002. URL: https://www.researchgate.net/publication/215757948_What_is_the_Grid_A_Three_Point_Checklist.

[52] Eduardo Huedo, Ruben S. Montero, and Ignacio M. Llorente. A Framework for Adaptive Execution in Grids. *Software: Practice and Experience*, 34, 2004.

[53] T. Drews, B. Temko, J. Alameda, E. Webb, M. Govindaraju, D. Gannon, A. Slominski, R. Alkire, R. Indurkar, R. Bramley, and S. Krishnan. The XCAT Science Portal. In *SC Conference*, page 16, Los Alamitos, CA, USA, nov 2001. IEEE Computer Society. URL: https://doi.ieeecomputersociety.org/10.1109/SC.2001.10036, doi:10.1109/SC.2001.10036.

[54] Marlon E. Pierce, Choonhan Youn, and Geoffrey C. Fox. The Gateway computational Web portal. *Concurrency and Computation: Practice and Experience*, 14(13-15):1411–1426, 2002. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.681, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.681, doi:https://doi.org/10.1002/cpe.681.

[55] M Jette, C Dunlap, J Garlick, and M Grondona. SLURM: Simple Linux Utility for Resource Management. Technical report, LLNL, 7 2002. URL: https://www.osti.gov/biblio/15002962.

[56] Timothy James Tautges, Corey Ernst, Clint Stimpson, Ray J Meyers, and Karl Merkley. MOAB : a mesh-oriented database. Technical report, Sandia National Laboratories, 4 2004. URL: https://www.osti.gov/biblio/970174, doi:10.2172/970174.

[57] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny. Condor – A Distributed Job Scheduler. In Thomas Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, MA, USA, October 2001.

[58] Bill Nitzberg, Jennifer M. Schopf, and James Patton Jones. *PBS Pro: Grid Computing and Scheduling Attributes*, page 183–190. Kluwer Academic Publishers, USA, 2004.

[59] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large élusters. *Commun. ACM*, 51(1):107–113, January 2008. doi:10.1145/1327452.1327492.

[60] Rene Brun and Fons Rademakers. ROOT — An object oriented data analysis framework. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 389(1):81–86, 1997. New Computing Techniques in Physics Research V. doi:https://doi.org/10.1016/S0168-9002(97)00048-X.

[61] CERN Data Centre. Key Facts and Figures. https://information-technology.web.cern.ch/sites/default/files/CERNDataCentre_KeyInformation_April2022V1.pdf, April 2022. Accessed 2022-12-05.

[62] V Vasilev, Ph Canal, A Naumann, and P Russo. Cling – the new interactive interpreter for root 6. *Journal of Physics: Conference Series*, 396(5):052071, dec 2012. URL: https://dx.doi.org/10.1088/1742-6596/396/5/052071, doi:10.1088/1742-6596/396/5/052071.

[63] ROOT. TTree Class Reference. https://root.cern.ch/doc/master/classTTree.html, 2022. Accessed on 2022-12-20.

[64] ROOT team. RNTuple class reference guide. https://root.cern.ch/doc/master/structROOT_1_1Experimental_1_1RNTuple.html, 2021. Accessed on 2022-12-20.

[65] Deepak Vohra. *Apache Parquet*, pages 325–335. Apress, Berkeley, CA, 2016. doi:10.1007/978-1-4842-2199-0_8.

[66] Javier López-Gómez and Jakob Blomer. Exploring Object Stores for High-Energy Physics Data Storage. *EPJ Web Conf.*, 251:02066, 2021. doi:10.1051/epjconf/202125102066.

[67] ROOT. TFile Class Reference. https://root.cern.ch/doc/master/classTFile.html, 2022. Accessed on 2022-12-20.

[68] Alvise Dorigo, P. Elmer, Fabrizio Furano, and A. Hanushevsky. XROOTD - a highly scalable architecture for data access. *WSEAS Transactions on Computers*, 4:348–353, 2005.

[69] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004. doi:10.1109/CGO.2004.1281665.

[70] Clang. Clang-Repl. https://clang.llvm.org/docs/ClangRepl.html, 2022. Accessed on 2022-12-20.

[71] Wim T.L.P. Lavrijsen and Aditi Dutta. High-Performance Python-C++ Bindings with PyPy and Cling. In *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*, pages 27–35, 2016. doi:10.1109/PyHPC.2016.008.

[72] Danilo Piparo, Philippe Canal, Enrico Guiraud, Xavier Valls Pla, Gerardo Ganis, Guilherme Amadio, Axel Naumann, and Enric Tejedor Saavedra. RDataFrame: Easy Parallel ROOT Analysis at 100 Threads. *EPJ Web Conf.*, 214:06029, 2019. doi:10.1051/epjconf/201921406029.

[73] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, page 10, Boston, MA, USA, 2010. USENIX Association. URL: https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets.

[74] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2523616.2523633.

[75] Kubernetes. Homepage. https://kubernetes.io/, 2022. Accessed on 2022-12-20.

[76] Matthew Rocklin. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In Kathryn Huff and James Bergstra, editors, *Proceedings of the 14th Python in Science Conference*, pages 130–136, online, 2015. SciPy.

[77] NumPy. Homepage. https://numpy.org/. Accessed on 2022-12-20.

[78] Pandas. Homepage. https://pandas.pydata.org/, 2022. Accessed on 2022-12-20.

[79] ATLAS Collaboration. Athena Introduction. https://atlassoftwaredocs.web.cern.ch/athena/athena-intro/, 2022. Accessed on 2022-12-20.

[80] Gabriele Benelli, Balazs Bozsogi, Andreas Pfeiffer, Danilo Piparo, and Vidmantas Zemleris. Measuring CMS software performance in the first years of LHC collisions. In *2011 IEEE Nuclear Science Symposium Conference Record*, pages 108–112, 2011. doi:10.1109/NSSMIC.2011.6154461.

[81] Latinos. Common Analysis Tools. https://github.com/latinos/LatinoAnalysis, 2022. Accessed on 2022-12-20.

[82] Pieter David. Readable and efficient HEP data analysis with bamboo. *EPJ Web Conf.*, 251:03052, 2021. doi:10.1051/epjconf/202125103052.

[83] CMS. CMGTools. https://github.com/CERN-PH-CMG/cmgtools-lite, 2022. Accessed on 2022-12-20.

[84] CMS. Tools for working with NanoAOD. https://github.com/cms-nanoAOD/nanoAOD-tools, 2022. Accessed on 2022-12-20.

[85] coffea. Basic tools and wrappers for enabling not-too-alien syntax when running columnar Collider HEP analysis. https://github.com/CoffeaTeam/coffea, 2022. Accessed on 2022-12-20.

[86] ROOT. TTreeFormula Class Reference. https://root.cern.ch/doc/master/classTTreeFormula.html, 2022. Accessed on 2022-12-20.

[87] Eric Conte, Benjamin Fuks, and Guillaume Serret. MadAnalysis 5, a user-friendly framework for collider phenomenology. *Computer Physics Communications*, 184(1):222–256, 2013. URL: https://www.sciencedirect.com/science/article/pii/S0010465512002950, doi:https://doi.org/10.1016/j.cpc.2012.09.009.

[88] Harrison B. Prosper, Sezen Sekmen, and Gokhan Unel. Analysis Description Language: A DSL for HEP Analysis. https://arxiv.org/abs/2203.09886, 2022. doi:10.48550/ARXIV.2203.09886.

[89] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter development team. Jupyter Notebooks - a publishing format for reproducible computational workflows. In Fernando Loizides and Birgit Scmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87–90. IOS Press, 2016. URL: https://eprints.soton.ac.uk/403913/.

[90] Danilo Piparo, Enric Tejedor, Pere Mato, Luca Mascetti, Jakub Moscicki, and Massimo Lamanna. SWAN: A service for interactive analysis in the cloud. *Future Generation Computer Systems*, 78, 2016.

[91] Bocchi, Enrico, Canali, Luca, Castro, Diogo, Kothuri, Prasanth, Gonzalez Labrador, Hugo, Malawski, Maciej, Mo´scicki, Jakub T., and Mrowczynski, Piotr. ScienceBox Converging to Kubernetes containers in production for on-premise and hybrid clouds for CERNBox, SWAN, and EOS. *EPJ Web Conf.*, 245:07047, 2020. doi:10.1051/epjconf/202024507047.

[92] Matous Adamec, Garhan Attebury, Kenneth Bloom, Brian Bockelman, Carl Lundstedt, Oksana Shadura, and John Thiltges. Coffea-casa: an analysis facility prototype. *EPJ Web Conf.*, 251:02061, 2021. doi:10.1051/epjconf/202125102061.

[93] Richard A. Erickson, Michael N. Fienen, S. Grace McCalla, Emily L. Weiser, Melvin L. Bower, Jonathan M. Knudson, and Greg Thain. Wrangling distributed computing for high-throughput environmental science: An introduction to HTCondor. *PLOS Computational Biology*, 14(10):1–8, 10 2018. doi:10.1371/journal.pcbi.1006468.

[94] Yongyang Cai, Kenneth L. Judd, Greg Thain, and Stephen J. Wright. Solving Dynamic Programming Problems on a Computational Grid. *Computational Economics*, 45:261–284, February 2015. doi:10.1007/s10614-014-9419-x.

[95] Bangyou Zheng, Edward Holland, and Scott C. Chapman. A standardized workflow to utilise a grid-computing system through advanced message queuing protocols. *Environmental Modelling & Software*, 84:304–310, 2016. URL: https://www.sciencedirect.com/science/article/pii/S1364815216303450, doi:https://doi.org/10.1016/j.envsoft.2016.07.012.

[96] Bockelman, Brian Paul, Foyo, Diego Davila, Hurtado Anampa, Kenyi, Ivanov, Todor Trendafilov, Khan, Farrukh Aftab, Kotobi, Amjad, Larson, Krista, Letts, James, Mascheroni, Marco, Mason, David, and Pérez-Calero Yzquierdo, Antonio. Improving the Scheduling Efficiency of a Global Multi-Core HTCondor Pool in CMS. *EPJ Web Conf.*, 214:03056, 2019. doi:10.1051/epjconf/201921403056.

[97] T Maeno, K De, T Wenaus, P Nilsson, G A Stewart, R Walker, A Stradling, J Caballero, M Potekhin, D Smith, and (forThe Atlas Collaboration). Overview of ATLAS PanDA Workload Management. *Journal of Physics: Conference Series*, 331(7):072024, dec 2011. URL: https://dx.doi.org/10.1088/1742-6596/331/7/072024, doi:10.1088/1742-6596/331/7/072024.

[98] K. Harrison, W. T. L. P. Lavrijsen, C. E. Tull, P. Mato, A. Soroko, C. L. Tan, N. Brook, and R. W. L. Jones. GANGA: A User Grid interface for Atlas and LHCb. *eConf*, C0303241:TUCT002, 2003. arXiv:cs/0306085.

[99] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 20–43, Bolton Landing, NY, 2003.

[100] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[101] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010. doi:10.1109/MSST.2010.5496972.

[102] Salman Salloum, Ruslan Dautov, Xiaojun Chen, Patrick Xiaogang Peng, and Joshua Zhexue Huang. Big data analytics on Apache Spark. *International Journal of Data Science and Analytics*, 1:145–164, 2016. doi:10.1007/s41060-016-0027-9.

[103] Muhammad Ashfaq Khan, Md. Rezaul Karim, and Yangwoo Kim. A Two-Stage Big Data Analytics Framework with Real World Applications Using Spark Machine Learning and Long Short-Term Memory Network. *Symmetry*, 10(10), 2018. doi:10.3390/sym10100485.

[104] Sergio Ramírez-Gallego, Héctor Mouriño-Talín, David Martínez-Rego, Verónica Bolón-Canedo, José Manuel Benítez, Amparo Alonso-Betanzos, and Francisco Herrera. An Information Theory-Based Feature Selection Framework for Big Data

Under Apache Spark. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 48(9):1441–1453, 2018. doi:10.1109/TSMC.2017.2670926.

[105] Archana A. Chaudhari and Preeti Mulay. *SCSI: Real-Time Data Analysis with Cassandra and Spark*, pages 237–264. Springer Singapore, Singapore, 2019. doi:10.1007/978-981-13-0550-4_11.

[106] Shyam R., Bharathi Ganesh H.B., Sachin Kumar S., Prabaharan Poornachandran, and Soman K.P. Apache Spark a Big Data Analytics Platform for Smart Grid. *Procedia Technology*, 21:171–178, 2015. doi:https://doi.org/10.1016/j.protcy.2015.10.085.

[107] H Shin, K Lee, and HY Kwon. A comparative experimental study of distributed storage engines for big spatial data processing using GeoSpark. *Journal of Supercomputing*, 78:2556–2579, 2022. doi:10.1007/s11227-021-03946-7.

[108] Michael Rilee, Niklas Griessbaum, Kwo-Sen Kuo, James Frew, and Robert Wolfe. STARE-Based Integrative Analysis of Diverse Data Using Dask Parallel Programming Demo Paper. In *Proceedings of the 28th International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '20, page 417–420, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3397536.3422346.

[109] Jatin Gharat, Bipin Kumar, Leena Ragha, Amit Barve, Shaik Mohammad Jeelani, and John Clyne. Development of NCL equivalent serial and parallel python routines for meteorological data analysis. *The International Journal of High Performance Computing Applications*, page 10943420221077110, 2022. doi:10.1177/10943420221077110.

[110] Joseph J. Hamman, Matthew Rocklin, and R. M. Abernathy. Pangeo: A Big-data Ecosystem for Scalable Earth System Science. In *20th EGU General Assembly, EGU2018*, page 12146, online, 2018. The SAO/NASA Astrophysics Data System (ADS).

[111] Shujie Fan, Max Linke, Ioannis Paraskevakos, Richard J. Gowers, Michael Gecht, and Oliver Beckstein. PMDA - Parallel Molecular Dynamics Analysis. In Chris

Calloway, David Lippa, Dillon Niederhut, and David Shupe, editors, *Proceedings of the 18th Python in Science Conference*, pages 134–142, online, 2019. SciPy. doi:10.25080/Majora-7ddc1dd1-013.

[112] Dask. `dask.dataframe` documentation. https://docs.dask.org/en/stable/dataframe.html, 2021. Accessed on 2022-12-20.

[113] Dan Graur, Ingo Müller, Mason Proffitt, Ghislain Fourny, Gordon T. Watts, and Gustavo Alonso. Evaluating Query Languages and Systems for High-Energy Physics Data. *Proc. VLDB Endow.*, 15(2):154–168, oct 2021. doi:10.14778/3489496.3489498.

[114] D. Feichtinger, P. Canal, C. Reed, C. Loizides, M. Ballintijn, F. Rademakers, A.J. Peters, G. Kickinger, J. Iwaszkiewicz, G. Ganis, R. Brun, B. Bellenot, D. Feichtinger, P. Canal, C. Reed, C. Loizides, M. Ballintijn, F. Rademakers, A.J. Peters, G. Kickinger, J. Iwaszkiewicz, G. Ganis, R. Brun, and B. Bellenot. PROOF - The Parallel ROOT Facility. In *2006 15th IEEE International Conference on High Performance Distributed Computing*, pages 379–380, Les Ulis, France, 2006. EDP Sciences. doi:10.1109/HPDC.2006.1652193.

[115] Saba Sehrish, Jim Kowalkowski, and Marc Paterno. Spark and HPC for High Energy Physics Data Analyses. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1048–1057, Lake Buena Vista, FL, USA, 2017. IEEE. doi:10.1109/IPDPSW.2017.112.

[116] Oliver Gutsche, Matteo Cremonesi, Peter Elmer, Bo Jayatilaka, Jim Kowalkowski, Jim Pivarski, Saba Sehrish, Cristina Mantilla Surez, Alexey Svyatkovskiy, and Nhan Tran. Big data in HEP: A comprehensive use case study. *Journal of Physics: Conference Series*, 898:072012, oct 2017. URL: https://doi.org/10.1088%2F1742-6596%2F898%2F7%2F072012, doi:10.1088/1742-6596/898/7/072012.

[117] Oliver Gutsche, Luca Canali, Illia Cremer, Matteo Cremonesi, Peter Elmer, Ian Fisk, Maria Girone, Bo Jayatilaka, Jim Kowalkowski, Viktor Khristenko, Evangelos Motesnitsalis, Jim Pivarski, Saba Sehrish, Kacper Surdy, and Alexey Svyatkovskiy. CMS Analysis and Data Reduction with Apache Spark. *Journal of*

Physics: Conference Series, 1085:042030, sep 2018. doi:10.1088/1742-6596/1085/4/042030.

[118] Valentina Avati, Milosz Blaszkiewicz, Enrico Bocchi, Luca Canali, Diogo Castro, Javier Cervantes, Leszek Grzanka, Enrico Guiraud, Jan Kaspar, Prasanth Kothuri, Massimo Lamanna, Maciej Malawski, Aleksandra Mnich, Jakub Moscicki, Shravan Murali, Danilo Piparo, and Enric Tejedor. Declarative Big Data Analysis for High-Energy Physics: TOTEM Use Case. In Ramin Yahyapour, editor, *Euro-Par 2019: Parallel Processing*, pages 241–255, Cham, 2019. Springer International Publishing.

[119] Baranowski, Zbigniew, Kleszcz, Emil, Kothuri, Prasanth, Canali, Luca, Castellotti, Riccardo, Martin Marquez, Manuel, Matos de Barros, Nuno Guilherme, Motesnitsalis, Evangelos, Mrowczynski, Piotr, and Luna Duran, Jose Carlos. Evolution of the hadoop platform and ecosystem for high energy physics. *EPJ Web Conf.*, 214:04058, 2019. doi:10.1051/epjconf/201921404058.

[120] Dask. `dask.delayed` documentation. https://docs.dask.org/en/stable/delayed.html, 2021. Accessed on 2022-12-20.

[121] Spark. Web UI. https://spark.apache.org/docs/latest/web-ui.html, 2022. Accessed 2022-11-25.

[122] Dask. Dashboard Diagnostics. https://docs.dask.org/en/stable/dashboard.html, 2022. Accessed 2022-11-25.

[123] Stefan Wunsch. Analysis of the di-muon spectrum using data from the CMS detector taken in 2012. http://doi.org/10.7483/OPENDATA.CMS.AAR1.4NZQ, 2019. doi:10.7483/OPENDATA.CMS.AAR1.4NZQ.

[124] Vincenzo Eduardo Padulano. Test suite repository. https://github.com/vepadulano/rdf-rntuple-daos-tests, 2021. Accessed on 2022-12-20.

[125] Spark. Tuning Guide. https://spark.apache.org/docs/latest/tuning.html#level-of-parallelism, 2022. Accessed on 2022-12-20.

[126] Ajay Gupta. Building Partitions For Processing Data Files in Apache Spark. https://medium.com/swlh/building-partitions-for-processing-data-files-in-apache-spark-2ca40209c9b7, 2020. Accessed on 2022-12-20.

[127] Massimiliano Bertolucci, Emanuele Carlini, Patrizio Dazzi, Alessandro Lulli, and Laura Ricci. *Static and Dynamic Big Data Partitioning on Apache Spark*, volume 27, pages 489–498. IOS Press, online, 2016. doi:10.3233/978-1-61499-621-7-489.

[128] Lucia Di Ciaccio and Simone Pagan Griso. Unraveling Nature's secrets: vector boson scattering at the LHC. https://atlas.cern/updates/feature/vector-boson-scattering, 2020. Accessed 2022-11-25.

[129] P. Rodriguez, C. Spanner, and E.W. Biersack. Analysis of Web caching architectures: hierarchical and distributed caching. *IEEE/ACM Transactions on Networking*, 9(4):404–418, 2001. doi:10.1109/90.944339.

[130] Brad Fitzpatrick. Distributed Caching with Memcached. *Linux Journal*, 2004(124):5, aug 2004.

[131] Kaihui Zhang, Yusuke Tanimura, Hidemoto Nakada, and Hirotaka Ogawa. Understanding and improving disk-based intermediate data caching in Spark. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2508–2517, 2017. doi:10.1109/BigData.2017.8258209.

[132] Kun Jiang, Shaofeng Du, Fu Zhao, Yong Huang, Chunlin Li, and Youlong Luo. Effective data management strategy and RDD weight cache replacement strategy in Spark. *Computer Communications*, 194:66–85, 2022. URL: https://www.sciencedirect.com/science/article/pii/S0140366422002559, doi:https://doi.org/10.1016/j.comcom.2022.07.008.

[133] Alexandru Uta, Bogdan Ghit, Ankur Dave, Jan Rellermeyer, and Peter Boncz. In-Memory Indexed Caching for Distributed Data Processing. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 104–114, 2022. doi:10.1109/IPDPS53621.2022.00019.

[134] Martin Barisits, Thomas Beermann, Frank Berghaus, Brian Bockelman, Joaquin Bogado, David Cameron, Dimitrios Christidis, Diego Ciangottini, Gancho Dimitrov, Markus Elsing, Vincent Garonne, Alessandro di Girolamo, Luc Goossens, Wen Guan, Jaroslav Guenther, Tomas Javurek, Dietmar Kuhn, Mario Lassnig, Fernando Lopez, Nicolo Magini, Angelos Molfetas, Armin Nairz, Farid Ould-Saada, Stefan Prenner, Cedric Serfon, Graeme Stewart, Eric Vaandering, Petya Vasileva, Ralph Vigne, and Tobias Wegner. Rucio: Scientific Data Management. *Computing and Software for Big Science*, 3(1):11, Aug 2019. doi:10.1007/s41781-019-0026-3.

[135] Ricky Egeland, Tony Wildish, and Simon Metson. Data transfer infrastructure for CMS data taking. In *XII Advanced Computing and Analysis Techniques in Physics Research (ACAT08)*, page 033, 2009. doi:10.22323/1.070.0033.

[136] Maier, Thomas, Beermann, Thomas, Duckeck, Günter, Lassnig, Mario, Legger, Federica, Magoni, Matteo, and Vukotic, Ilija. Performance and impact of dynamic data placement in ATLAS. *EPJ Web Conf.*, 214:04025, 2019. doi:10.1051/epjconf/201921404025.

[137] Yutaro Iiyama, Benedikt Maier, Daniel Abercrombie, Maxim Goncharov, and Christoph Paus. Dynamo - Handling Scientific Data Across Sites and Storage Media. *CoRR*, abs/2003.11409, 2020. URL: https://arxiv.org/abs/2003.11409, arXiv:2003.11409.

[138] L A T Bauerdick, K Bloom, B Bockelman, D C Bradley, S Dasu, J M Dost, I Sfiligoi, A Tadel, M Tadel, F Wuerthwein, A Yagil, and the CMS collaboration. XRootd, disk-based, caching proxy for optimization of data access, data placement and data replication. *Journal of Physics: Conference Series*, 513(4):042044, jun 2014. URL: https://dx.doi.org/10.1088/1742-6596/513/4/042044, doi:10.1088/1742-6596/513/4/042044.

[139] L Bauerdick, D Benjamin, K Bloom, B Bockelman, D Bradley, S Dasu, M Ernst, R Gardner, A Hanushevsky, H Ito, D Lesny, P McGuigan, S McKee, O Rind, H Severini, I Sfiligoi, M Tadel, I Vukotic, S Williams, F Würthwein, A Yagil, and

W Yang. Using Xrootd to Federate Regional Storage. *Journal of Physics: Conference Series*, 396(4):042009, dec 2012. URL: https://dx.doi.org/10.1088/1742-6596/396/4/042009, doi:10.1088/1742-6596/396/4/042009.

[140] Elizabeth Copps, Huiyi Zhang, Alex Sim, Kesheng Wu, Inder Monga, Chin Guok, Frank Würthwein, Diego Davila, and Edgar Fajardo. Analyzing Scientific Data Sharing Patterns for In-Network Data Caching. In *Proceedings of the 2021 on Systems and Network Telemetry and Analytics*, SNTA '21, page 9–16, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3452411.3464441.

[141] Alex Sim, Ezra Kissel, and Chin Guok. Deploying in-network caches in support of distributed scientific data sharing. https://arxiv.org/abs/2203.06843, 2022. doi:10.48550/ARXIV.2203.06843.

[142] Julian Bellavita, Alex Sim, Kesheng Wu, Inder Monga, Chin Guok, Frank Würthwein, and Diego Davila. Studying Scientific Data Lifecycle in On-demand Distributed Storage Caches. In *Fifth International Workshop on Systems and Network Telemetry and Analytics*. ACM, jun 2022. URL: https://doi.org/10.1145%2F3526064.3534111, doi:10.1145/3526064.3534111.

[143] Espinal, X., Jezequel, S., Schulz, M., Sciabà, A., Vukotic, I., and Wuerthwein, F. The Quest to solve the HL-LHC data access puzzle. *EPJ Web Conf.*, 245:04027, 2020. doi:10.1051/epjconf/202024504027.

[144] Amazon. Amazon Simple Storage Service Documentation. https://docs.aws.amazon.com/s3/, 2021. Accessed on 2022-12-20.

[145] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows Azure Storage: A Highly Available Cloud Storage

Service with Strong Consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 143–157, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2043556.2043571.

[146] Zhen Liang, Johann Lombardi, Mohamad Chaarawi, and Michael Hennecke. DAOS: A Scale-Out High Performance Storage Stack for Storage Class Memory. In Dhabaleswar K. Panda, editor, *Supercomputing Frontiers*, pages 40–54, Cham, 2020. Springer International Publishing.

[147] Jialin Liu, Quincey Koziol, Gregory F. Butler, Neil Fortner, Mohamad Chaarawi, Houjun Tang, Suren Byna, Glenn K. Lockwood, Ravi Cheema, Kristy A. Kallback-Rose, Damian Hazen, and Mr Prabhat. Evaluation of HPC Application I/O on Object Storage Systems. In *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage Data Intensive Scalable Computing Systems (PDSW-DISCS)*, pages 24–34, 2018. doi:10.1109/PDSW-DISCS.2018.00005.

[148] Guoxin Kang, Defei Kong, Lei Wang, and Jianfeng Zhan. OStoreBench: Benchmarking Distributed Object Storage Systems Using Real-World Application Scenarios. In Felix Wolf and Wanling Gao, editors, *Benchmarking, Measuring, and Optimizing*, pages 90–105, Cham, 2021. Springer International Publishing.

[149] Seiz, M., Offenhäuser, P., Andersson, S. et al. Lustre I/O performance investigations on Hazel Hen: experiments and heuristics. *Journal of Supercomputing*, 77, 2021. doi:10.1007/s11227-021-03730-7.

[150] Jerome Soumagne, Jordan Henderson, Mohamad Chaarawi, Neil Fortner, Scot Breitenfeld, Songyu Lu, Dana Robinson, Elena Pourmal, and Johann Lombardi. Accelerating HDF5 I/O for Exascale Using DAOS. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):903–914, 2022. doi:10.1109/TPDS.2021.3097884.

[151] Lukas Rupprecht, Rui Zhang, and Dean Hildebrand. Big Data Analytics on Object Stores : A Performance Study. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*, 2014.

[152] Lukas Rupprecht, Rui Zhang, Bill Owen, Peter Pietzuch, and Dean Hildebrand. SwiftAnalytics: Optimizing Object Storage for Big Data Analytics. In *2017 IEEE*

International Conference on Cloud Engineering (IC2E), pages 245–251, 2017. doi:10.1109/IC2E.2017.19.

[153] Gil Vernik, Michael Factor, Elliot K. Kolodner, Effi Ofer, Pietro Michiardi, and Francesco Pace. Stocator: A high performance object store connector for spark. In *Proceedings of the 10th ACM International Systems and Storage Conference*, SYSTOR '17, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3078468.3078496.

[154] Paolo Badino, Olof Barring, Jean-Philippe Baud, Flavia Donno, and Maarten Litmaath. The Storage Resource Manager Interface Specification (v2.2), 2009. URL: https://sdm.lbl.gov/srm-wg/doc/SRM.v2.2.html.

[155] S Andreozzi, L Magnoni, and R Zappi. Towards the integration of StoRM on Amazon Simple Storage Service (S3). *Journal of Physics: Conference Series*, 119(6):062011, jul 2008. doi:10.1088/1742-6596/119/6/062011.

[156] Peter Braam. The Lustre Storage Architecture. https://arxiv.org/abs/1903.01955, 2019.

[157] C J Walker, D P Traynor, and A J Martin. Scalable Petascale Storage for HEP using Lustre. *Journal of Physics: Conference Series*, 396(4):042063, dec 2012. doi:10.1088/1742-6596/396/4/042063.

[158] María Arsuaga-Ríos, Seppo S Heikkilä, Dirk Duellmann, René Meusel, Jakob Blomer, and Ben Couturier. Using S3 cloud storage with ROOT and CvmFS. *Journal of Physics: Conference Series*, 664(2):022001, dec 2015. doi:10.1088/1742-6596/664/2/022001.

[159] A Charbonneau, A Agarwal, M Anderson, P Armstrong, K Fransham, I Gable, D Harris, R Impey, C Leavett-Brown, M Paterson, W Podaima, R J Sobie, and M Vliet. Data intensive high energy physics analysis in a distributed cloud. *Journal of Physics: Conference Series*, 341:012003, feb 2012. doi:10.1088/1742-6596/341/1/012003.

[160] John Carrier. Disrupting High Performance Storage with Intel DC Persistent Memory & DAOS. IXPUG 2019 Annual Conference at CERN. https://cds.cern.ch/record/2691951, Sep 2019.

[161] Johann Lombardi. DAOS: Nextgen Storage Stack for AI, Big Data and Exascale HPC. CERN openlab Technical Workshop, Mar 2021. URL: https://cds.cern.ch/record/2754116.

[162] ROOT. TFilePrefetch class reference. https://root.cern.ch/root/html608/classTFilePrefetch.html, 2022. Accessed on 2022-12-20.

[163] Dhabaleswar K. Panda and Sayantan Sur. *InfiniBand*, pages 927–935. Springer US, Boston, MA, 2011. doi:10.1007/978-0-387-09766-4_21.

[164] Mark S. Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D. Underwood, and Robert C. Zak. Intel Omni-path Architecture: Enabling Scalable, High Performance Fabrics. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 1–9, 2015. doi:10.1109/HOTI.2015.22.

[165] ISO Central Secretary. Information technology — Procedures for the operation of object identifier registration authorities — Part 8: Generation of universally unique identifiers (UUIDs) and their use in object identifiers. Standard ISO/IEC 9834-8:2014, International Organization for Standardization, Geneva, CH, 2014. URL: https://www.iso.org/standard/62795.html.

[166] Vincenzo Eduardo Padulano. Tests comparing xrootd and tfileprefetch. https://github.com/vepadulano/rdf-dist-cache, 2020. Accessed on 2022-12-20.

[167] CERN. OpenStack Overview. https://clouddocs.web.cern.ch/overview/overview.html, 2021. Accessed on 2022-12-20.

[168] Javier Lopez-Gomez and Jakob Blomer. RNTuple performance: Status and Outlook. https://arxiv.org/abs/2204.09043, 2022.

[169] DAOS developers. Caching. https://docs.daos.io/v2.0/user/filesystem/#caching, 2022. Accessed on 2022-12-20.

[170] LHCb Collaboration. Matter Antimatter Differences (B meson decays to three hadrons) - Project Notebook. http://opendata.cern.ch/record/4902, 2017. Accessed on 2022-12-20.

[171] Virgo Cluster. User Manual. https://hpc.gsi.de/virgo/preface.html, 2022. Accessed on 2022-12-20.

[172] Uli Plechschmidt. Lustre expands its lead in the Top 100 supercomputers. https://community.hpe.com/t5/Advantage-EX/Lustre-expands-its-lead-in-the-Top-100-supercomputers/ba-p/7141807#.YukqZUhByXJ, 2021. Accessed on 2022-12-20.

[173] Frontier Supercomputer. Job Submission Documentation. https://docs.olcf.ornl.gov/systems/frontier_user_guide.html. Accessed on 2022-12-20.

[174] LUMI Supercomputer. Job Submission Documentation. https://docs.lumi-supercomputer.eu/runjobs/. Accessed on 2022-12-20.

[175] Google. Cloud Functions. https://cloud.google.com/functions, 2022.

[176] Amazon Web Services. Lambda. https://aws.amazon.com/releasenotes/release-aws-lambda-on-2014-11-13, 2022.

[177] Apache Foundation. Apache openwhisk is an open source, distributed serverless platform that executes functions (fx) in response to events at any scale. https://openwhisk.apache.org/, 2022.

[178] Knative: Kubernetes-based platform to build, deploy, and manage modern serverless workloads. https://knative.dev.

[179] Dirk Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, 2014(239), mar 2014.

[180] Piotr Grzesik, Dariusz R Augustyn, Łukasz Wyciślik, and Dariusz Mrozek. Serverless computing in omics data analysis and integration. *Briefings in Bioinformatics*, 23(1), 09 2021. bbab349. arXiv:https:

//academic.oup.com/bib/article-pdf/23/1/bbab349/42229545/bbab349.pdf, doi:10.1093/bib/bbab349.

[181] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 445–451, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3127479.3128601.

[182] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. Serverless linear algebra. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 281–295, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3419111.3421287.

[183] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. Wukong: A scalable and locality-enhanced framework for serverless parallel computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 1–15, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3419111.3421286.

[184] Vicente Giménez-Alventosa, Germán Moltó, and Miguel Caballer. A framework and a performance assessment for serverless MapReduce on AWS Lambda. *Future Generation Computer Systems*, 97:259–274, 2019. doi:https://doi.org/10.1016/j.future.2019.02.057.

[185] Alfonso Pérez, Germán Moltó, Miguel Caballer, and Amanda Calatrava. Serverless computing for container-based architectures. *Future Generation Computer Systems*, 83:50–59, 2018. doi:https://doi.org/10.1016/j.future.2018.01.022.

[186] Nilton Bila, Paolo Dettori, Ali Kanso, Yuji Watanabe, and Alaa Youssef. Leveraging the serverless architecture for securing linux containers. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 401–404, 2017. doi:10.1109/ICDCSW.2017.66.

[187] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. Help rather than recycle: Alleviating cold startup in serverless computing through Inter-Function container sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 69–84, Carlsbad, CA, July 2022. USENIX Association. URL: https://www.usenix.org/conference/atc22/presentation/li-zijun-help.

[188] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with Serverless-Optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, July 2018. USENIX Association. URL: https://www.usenix.org/conference/atc18/presentation/oakes.

[189] Hai Duc Nguyen, Zhifei Yang, and Andrew A. Chien. Motivating High Performance Serverless Workloads. In *Proceedings of the 1st Workshop on High Performance Serverless Computing*, HiPS '21, page 25–32, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3452413.3464786.

[190] Jakob Blomer, Predrag Buncic, and Thomas Fuhrmann. CernVM-FS: Delivering Scientific Software to Globally Distributed Computing Resources. In *Proceedings of the First International Workshop on Network-Aware Data Management*, NDM '11, page 49–56, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2110217.2110225.

[191] Blomer, Jakob, Ganis, Gerardo, Mosciatti, Simone, and Popescu, Radu. Towards a serverless CernVM-FS. *EPJ Web Conf.*, 214:09007, 2019. doi:10.1051/epjconf/201921409007.

[192] Ingo Müller, Renato Marroquín, and Gustavo Alonso. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 115–130, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3318464.3389758.

[193] Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. FuncX: A Federated Function Serving Fabric for Science. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '20, page 65–76, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3369583.3392683.

[194] Alfonso Pérez, Sebastián Risco, Diana María Naranjo, Miguel Caballer, and Germán Moltó. On-Premises Serverless Computing for Event-Driven Data Processing Applications. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, 2019. doi:10.1109/CLOUD.2019.00073.

[195] Amazon Web Services. Lambda execution environments. https://docs.aws.amazon.com/lambda/latest/operatorguide/execution\protect\discretionary{\char\hyphenchar\font}{}{}environments.html.

[196] Miguel Caballer, Carlos de Alfonso, Fernando Alvarruiz, and Germán Moltó. EC3: Elastic Cloud Computing Cluster. *Journal of Computer and System Sciences*, 79(8):1341–1351, 2013. doi:https://doi.org/10.1016/j.jcss.2013.06.005.

[197] Miguel Caballer, Ignacio Blanquer, Germán Moltó, and Carlos de Alfonso. Dynamic Management of Virtual Infrastructures. *Journal of Grid Computing*, 13(1):53–70, Mar 2015. doi:10.1007/s10723-014-9296-5.

[198] Fernando Alvarruiz, Carlos de Alfonso, Miguel Caballer, and Vicente Hern'ndez. An Energy Manager for High Performance Computer Clusters. In *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, pages 231–238, 2012. doi:10.1109/ISPA.2012.38.

[199] MinIO. White paper: High Performance Multi-Cloud Object Storage. Technical report, MinIO Inc., Palo Alto, CA, USA, 2022. URL: https://min.io/resources/docs/MinIO-High-Performance-Multi-Cloud-Object-Storage.pdf.

[200] AJ Peters, EA Sindrilaru, and G Adde. EOS as the present and future solution for data storage at CERN. *Journal of Physics: Conference Series*, 664(4):042042, dec 2015. doi:10.1088/1742-6596/664/4/042042.

[201] HashiCorp. Terraform : Infrastructure as code for provisioning, compliance, and management of any cloud, infrastructure, and service. https://www.hashicorp.com/products/terraform, 2022.

[202] Amazon Web Services. Boto3 documentation. https://boto3.readthedocs.io. Accessed on 2022-12-20.

[203] Amazon Web Services. Types of Lambda invocations. https://docs.aws.amazon.com/lambda/latest/dg/lambda-invocation.html.

[204] Aws lambda quotas. https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html, 2022.

[205] Michael Pitt. PPS exclusive dilepton analysis. https://gitlab.cern.ch/mpitt/ppstools/-/blob/master/exclusive-dilep/PrepareDataFrames.ipynb.