VNIVERSITAT DE VALÈNCIA

Escola Tècnica Superior d'Enginyeria

Departament d'Informàtica

Programa de Doctorat en Tecnologies de la Informació,
Comunicacions i Computació

Performance Improvements of EventIndex Distributed System at CERN

TESIS DOCTORAL

Álvaro Fernández Casaní

Directores:
Juan Manuel Orduña Huertas y Santiago González de la Hoz

Diciembre 2022

D. Juan Manuel Orduña Huertas,
Profesor del Departamento de Informática de la Universidad de Valencia, y

D. Santiago González de la Hoz,
Investigador del Instituto de Física Corpuscular (IFIC), y profesor del Departamento de Física Atómica, Molecular y Nuclear de la Universidad de Valencia.

CERTIFICAN:

Que la presente memoria que tiene por título *Performance Improvements of EventIndex Distributed System at CERN*, ha sido realizada bajo su dirección por D. Álvaro Fernández Casaní y constituye su trabajo de tesis doctoral en el Departamento de Informática de la Universidad de Valencia para optar al título de Doctor en Tecnologías de la Información, Comunicaciones y Computación.

Y para que conste, en cumplimiento de la legislación vigente, firman el presente certificado.

Fdo. Juan Manuel Orduña Huertas   Fdo. Santiago González de la Hoz

A Amanda,
mi kantele.

A mis padres,
por estar siempre.

Preface

The work described in this thesis is framed in the context of the EventIndex project of the ATLAS experiment, a big particle detector of the LHC (Large Hadron Collider) at CERN. When I was given the opportunity to be part of it, I found it very interesting to continue the work on distributed systems that we started many years ago with the development and deployment of grid technologies within big data environments.

I would like to thank my PhD. supervisors Juan Manuel Orduña Huertas and Santiago González de la Hoz, for the unwavering guidance and support during this work in partial fulfillment of my doctoral degree.

As part of an international collaboration, many people have contributed to the ATLAS EventIndex project success. I would like to acknowledge all the members of the project, for these years of excellent work and helping me when I needed it. I would also like to thank the CERN IT people, always available to share their knowledge and skills.

A special thank you to my colleagues at IFIC (Instituto de Física Corpuscular), joint centre of the University of Valencia and CSIC. The current and past members of the ATLAS Tier-2 computing group have shared motivation and challenges over these years, and I am grateful for all the things that I have learned. I would like to particularly mention the emotional support that I have received from leaders and colleagues to finish this work.

Finalmente, esto no sería posible sin la ayuda y apoyo de mi familia. A mi mujer Amanda, siempre a mi lado con amor incondicional. A mis padres, Toni y Marisa, que me han hecho estar orgulloso de mi y de ellos, y me enseñan cada día a valorar lo importante. A mi hermana Ana, soportándome desde siempre y siendo una trabajadora incansable, junto a Ismael y a mis sobrinos Pablo y Adrián. A Elvira por estar siempre a mi lado y a toda mi familia, gracias.

Contents

Preface vii

1 Introduction 1
1.1 Motivation 1
1.2 Objectives 2
1.3 Thesis Outline 2

2 The Large Hadron Collider and the ATLAS Experiment 5
2.1 Large Hadron Collider 5
2.2 ATLAS Experiment 8
2.2.1 Inner Detector 10
2.2.2 Calorimeters 12
2.2.3 Muon Spectrometer 14
2.2.4 Trigger and Data Acquisition System 16
2.3 ATLAS Distributed Computing 20
2.4 ATLAS Computing and Data Challenges in Run 2 and Run 3 22

3 EventIndex Project 27
3.1 Introduction and Goals 27
3.2 Use Cases 29
3.3 Data Model 30
3.4 Requirements 31
3.5 Architecture 32
3.5.1 Data Production 32
3.5.2 Data Collection 35
3.5.3 Data Storage 35
3.5.4 Data Access 36
3.5.5 Monitoring 37

4 Data Collection 39
4.1 Legacy Messaging Data Collection . . . 40
4.1.1 Data ingestion . . . 42
4.1.2 Data Validator Controller . . . 43
4.1.3 Shortcomings . . . 44
4.2 New Design of Distributed Data Collection . . . 45
4.2.1 Object Store data staging . . . 46
4.2.2 Push versus pull model data ingestion . . . 47
4.3 Evaluation . . . 51
4.3.1 Single dataset indexing results . . . 52
4.3.2 Complete results . . . 55
4.4 Conclusions . . . 65

5 Storage 69
5.1 HDFS . . . 70
5.1.1 Data organization . . . 71
5.1.2 File format and contents . . . 73
5.1.3 Limitations . . . 76
5.2 Kudu . . . 77
5.2.1 Data organization . . . 77
5.2.2 Data ingestion . . . 80
5.3 HBase and Phoenix . . . 82
5.3.1 Data organization . . . 85
5.3.2 Data ingestion . . . 92
5.4 Conclusions . . . 106

6 Access 109
6.1 Requirements and use cases . . . 109
6.2 EventIndex Analytics Platform . . . 110
6.2.1 Spark . . . 112
6.2.2 Data discovery . . . 113
6.2.3 Duplicate calculation . . . 114
6.2.4 Helper functions . . . 118
6.2.5 Overlaps calculation . . . 120
6.3 Conclusions . . . 126

7 Conclusions 129
7.1 Contributions . . . 129
7.2 Publications . . . 131

Resumen 135
Introducción 136
CERN, LHC y el experimento ATLAS 136
Proyecto EventIndex 140
Objetivos 140
Metodología 141
Recolección distribuida de datos 142
Almacenamiento 144
Acceso 147
Conclusiones 149

Bibliography 153

1 Introduction

1.1 Motivation

Our knowledge of the Universe is limited, and one way to search for answers is to look at the smallest things to understand the bigger ones.

The ATLAS experiment in the Large Hadron Collider (LHC) at CERN, the European Organization for Nuclear Research, is devoted to studying the tiniest entities in Nature. The temperature and energy density produced in the particle colliders is very similar to that just a few moments after the Big Bang, which means looking into the past to the very beginning of the Universe.

With the massive data production of the LHC experiments at CERN, a distributed computing solution was required to access the data in a distributed manner for hundreds of research institutions worldwide. The grid was developed and has evolved within the WLCG (Worldwide LHC Computing Grid) to provide computing and storage resources distributedly owned and managed, but with central coordination.

The ATLAS EventIndex project aims to provide a complete catalogue of all the particle collisions or events, real and simulated, produced in the experiment over the different years of operation. To accomplish this objective, a small quantity of metadata per event is collected in a distributed manner worldwide on the grid, and then conveyed to a central database at CERN. Thus, use cases related to selecting particular events based on constraints, or analytical studies over large quantities of data can be solved in an easy and performant manner.

A number of challenges are foreseen for the next years with increasing production rates. These are related to the storage, cataloging and access to petabytes of data, and to the computing resources to analyze them. Also thousands of physicist require friendly tools for locating and accessing the data of their interest for their analyses.

A distributed data collection system is required to convey the indexed meta-

data from the grid to a central catalogue at CERN. A previous implementation based on a messaging system was developed and deployed previously to Run 2 (2015-2018), but faced scalability problems. The backend storage system to host all the collected EventIndex data at CERN was first implemented with a NoSQL approach using plain Hadoop HDFS files. A hybrid approach using Oracle for faster access to a subset of the data was implemented and deployed during Run 2. This implementation has performed correctly, at the cost of duplicating data in both systems. Yet with the increasing ingestion rates and the total volume of data expected on Run 3 (2022-2025) and the next Runs, a more modern scalable and easily manageable solution to satisfy all use cases is needed.

1.2 Objectives

The objectives of this thesis are related to the enhancements required for indexing and cataloguing all produced ATLAS data in a large scale distributed system, for the current and next Runs of the experiment.

A first objective is to find the shortcomings of the previous distributed data collection approach and demonstrate that a new pull model design with temporary data staging in an object store and with dynamic data selection can scale for the production rates of the experiment. In addition the previous approach can be improved, reducing the overall complexity and resource usage.

Collected metadata is stored at a central catalogue at CERN, and the current approach is to provide a hybrid solution using Hadoop as main storage and an Oracle database to provide faster access to a subset of the data. The second objective is to study a Big Data data storage solution able to scale with the data ingestion rates, without duplication and data coherence issues. Columnar-storages like Kudu as a hybrid transactional and analytical processing tool, and HBase for its fast random-access support are considered. A new SQL layer with Phoenix is considered for schema definition and data ingestion standard access.

The selected backend storage must be able to support all required use cases. The third and last objective is to prove that a schema enforcement provides benefits for data model and access, and that HBase/Phoenix is versatile enough to support our analytical workloads.

1.3 Thesis Outline

This thesis is divided into seven chapters.

Chapter 1 introduces the thesis, motivation, and objectives.

Chapter 2 introduces the LHC and the ATLAS experiment at CERN, with the related data challenges in Run 2 (2015-2018). Long Shutdown 2 (2018-2022) was devoted to preparing for the new conditions of the current Run 3 (2022-2025), with the upgrades on the detector and the computing infrastructure including EventIndex. The future requirements on the High Luminosity LHC (HL-LHC) are also outlined in this chapter.

Chapter 3 presents the EventIndex project, its objectives and requirements to solve the needed use cases, and the architecture of the different components, including distributed data collection, storage, and access.

Chapter 4 discusses the challenges of distributed data collection. A producer-consumer architecture based on an Object Store as a staging data repository is presented, with a pull based model for data ingestion. Results on real production scenarios are discussed.

Chapter 5 includes the studies on different data storage backends. First the original HDFS implementation is discussed, showing its shortcomings. Studies on columnar-based data storage are presented with the usage of Kudu. Chapter ends with the HBase and Phoenix model, showing that it can support the required data ingestion rates and that it can store a single copy of the data to solve all use cases.

Chapter 6 shows the tools based on Spark for accessing the data based on HBase and Phoenix. We show that our system can support our analytical use cases. We discuss the tools for data discovery, duplicate detection and a new algorithm for detecting overlaps among events.

Finally the conclusions for this work are discussed in the last Chapter 7.

2 The Large Hadron Collider and the ATLAS Experiment

2.1 Large Hadron Collider

The Large Hadron Collider (LHC) [1] is the largest machine built, and the highest energy particle collider of the world, designed to run at a maximum energy in the center of mass of 14 TeV. It is built in a circular tunnel located at an average of 100 metres underground, and with a total length of 26.7 km. Charged particles (protons and heavy ions, which are hadrons) are accelerated in beams in two rings in opposite directions until reaching the desired energy.

The journey of a particle in the LHC starts, in the case of the protons, in a bottle of compressed hydrogen. The CERN accelerator complex, as can be seen in figure 2.1, contains a chain of injectors working together to increase the energy of the particles.

The starting point of the trip, at the bottom of the figure, is the LINAC4 (A Linear Accelerator), where negative Hydrogen ions are accelerated up to an energy of 160 MeV. Before entering the next stage, electrons are stripped off the ions and the bare protons are injected into the Proton Synchrotron Booster (PSB). This is a small 25-metre circular accelerator, which makes it possible to increase the energy to 2 GeV and up to 100 times the number of protons accepted in the next stage. The Proton Synchrotron (PS) accepts protons from the previous stage or heavy ions from the Low Energy Ion Ring (LEIR). It is composed of a 628-metre circumference ring that increases the energy up to 25 GeV. Then a transfer line conveys the protons to the Super Proton Synchrotron (SPS), housed in a circular tunnel of 6.9 km, currently capable of reaching up to 450 GeV.

The LHC operates in cycles, each considered a fill of the accelerator composed of different phases.

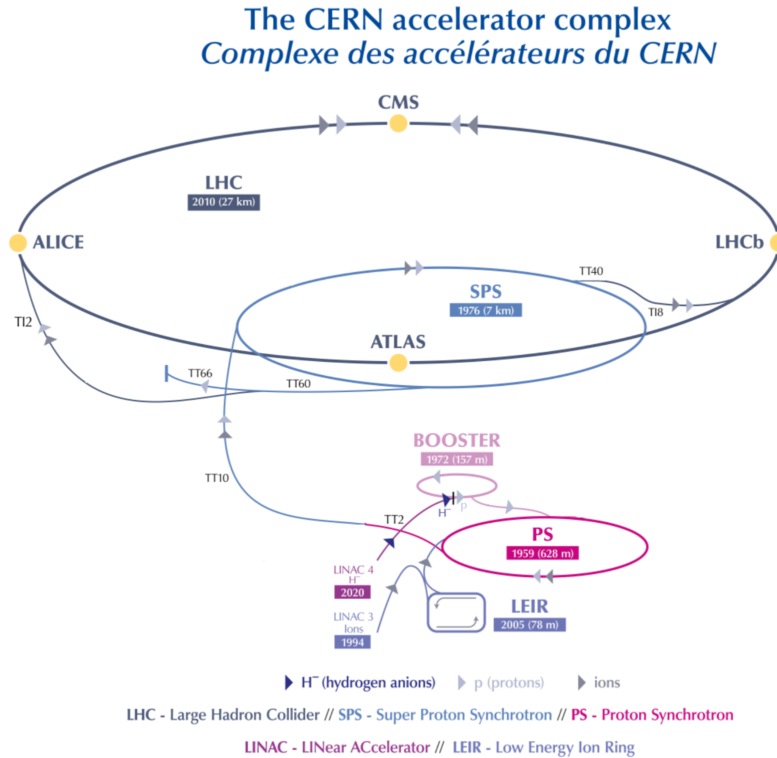


Figure 2.1: CERN Accelerator Complex, layout in 2022. Figure adapted from [2].

The injection phase starts with the particles crossing over to the two beam pipes of the LHC, in a process that lasts 4 minutes and 20 seconds per beam. Instead of a continuous beam, the proton injection is done in bunches in all the chain. The designed objective is 2,808 bunches when reaching the LHC.

In order to maintain the particles in their circular beamlines, 1,232 dipole magnets, 15 metres long each, are used along the 26.9 kilometres track. Coils are made of niobium-titanium (Nb-i) fibres, and a current of 11,850 amps is needed to create the necessary 8.33 T magnetic field. Additionally 392 quadrupole magnets, 5 to 7 metres each, are installed to maintain the beams focused and

the most powerful are located close to the interaction points where the particles will collide.

Beamline pipes are kept at ultra high vacuum to avoid beams of particles colliding with gas molecules. Ultra high vacuum is also used as an insulator in the cryogenically cooled magnets, and in the helium distribution line. More than 120 tonnes of superfluid helium-4 are needed in order to maintain the superconducting properties of the magnets at 1.9 K (-271.3°C).

During the ramp phase, the charged particles are accelerated in the LHC by means of the electric fields in 16 radiofrequency (RF) cavities (8 per beam). Cavities are equipped with a high-power klystron allocated in cryomodules to work at superconducting state. These reach 2 megavolts (MV) per cavity, corresponding to 16 MV per beam. The electric field oscillates at 400 MHz, making the particles line up and keeping them in their bunches, accelerating or decelerating the particles arriving at the cavity. They bring the original 450 GeV energy of the protons to 6.5 TeV in a 20 minutes cycle, when they have crossed the cavities more than 10 million times.

During the squeeze and adjust phases, the beams are prepared in the interaction points where the 2 opposite beams are directed by the quadrupole magnets against each other to produce particle collisions.

At this point there are stable beams that fulfil the LHC conditions and therefore collisions can be registered. There are 4 huge caverns underground, where experiments place their detectors to record the details of the particle collisions that are produced at a bunch crossing rate of 40 MHz or every 25 ns. Several independent interactions can happen at almost the same time within 0.5 ns during the bunch crossings.

The signals left by the particles produced by these interactions are recorded by the detectors as an event from a particular bunch-crossing. The number of independent particle interactions per bunch-crossing is referred to as the pile-up, and this varies with the parameters of the LHC. One of the most important parameters is the luminosity, or the number of potential collisions per surface unit over a given period of time. When we increase the luminosity we have greater amount of data to better analyze physical processes, but we also increase the pile-up making it more difficult to select the desired interactions.

When the number and energy of the particles have decreased to a certain level, the final dump and ramp-down phases take place to absorb the excess particle energy, and to decrease the magnetic fields.

The LHC has been in operation since 2009, and alternates long data-taking periods with long shutdown times for maintenance and upgrades. Since Run 1 (2009-2013), and Run 2 (2015-2018) the luminosity was increased and the

pile-up reached up to 50-60. After the Long Shutdown 2, this is expected to be further increased by the end of the Run 3 (2022-2025).

ATLAS and CMS are 2 big general purpose detectors devoted to a broad range of physics studies, which can independently confirm their discoveries. LHCb and ALICE are dedicated to specific phenomena, like physics of B and D mesons [3], or the formation of quark–gluon plasma (QGP) [4].

2.2 ATLAS Experiment

The ATLAS detector [5] is a multi-purpose detector, in the form of a big cylinder 25 metres high, 46 metres long and 7,000 tonnes in weight that is situated in a big cavern 100 metres underground, at point 1 of the LHC. As can be seen in figure 2.2, it is composed of several subsystems, layered and concentric to the interaction point of the accelerator beams. These include the inner detector, the electromagnetic and hadronic calorimeters, the magnet system, and the muon spectrometer. Each of these is designed with a single objective: either to detect particular kinds of particles or measure individual characteristics such as the trajectory, energy or momentum.

Heavy particles expected to be produced in the LHC, will decay into hundreds of lighter particles, like photons, muons, and electrons, after the collisions. Taking this into account, their tracks and characteristics can be reconstructed and identified.

The trajectory of the particles is usually straight, but the big magnets contribute to the identification bending the tracks, and make measuring the momenta possible. High momentum particles will go in almost straight tracks, while lower energy ones will travel in slower spirals around the inner subsystems.

The trackers measure the trajectory at different points to reconstruct the charge and energy from the particle. The inner layers are the least dense to avoid interacting. Calorimeters, on the other hand, are dense and absorb most of the particles, making it possible to measure the energy. Electromagnetic (EM) calorimeters measure the energy of particles like electrons or photons when they interact with the charged matter of the detector. Hadronic calorimeters measure hadrons (particles made of quarks) energy while they interact with atomic nuclei.

In the trackers, a long enough trajectory must be measured to calculate the curvature radius of particles with high kinetic energy. The calorimeters are designed to be very large to absorb as much energy as possible. So these are the reasons why the LHC detectors are so large.

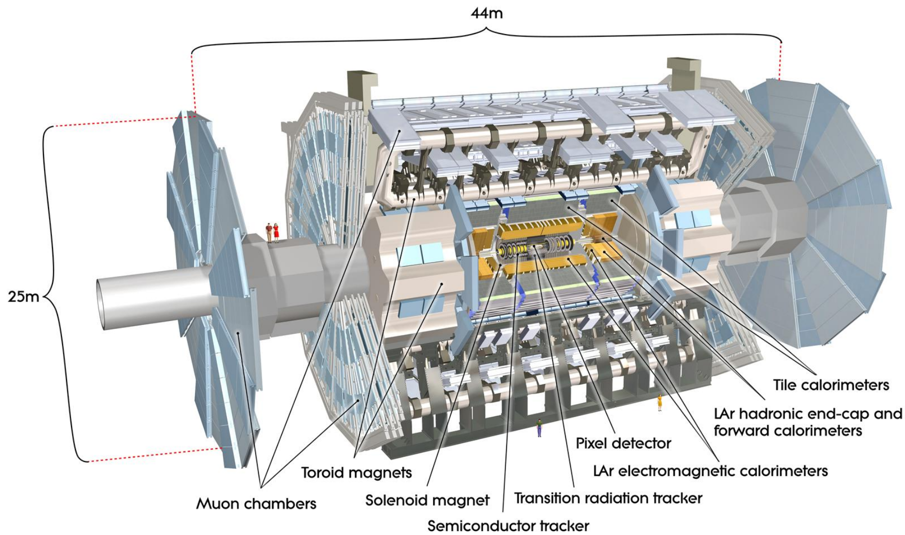


Figure 2.2: Computer generated image of the whole ATLAS detector with detailed subsystems [6].

A visualization of the different kinds of particles and their tracks along the different ATLAS subsystems can be seen in figure 2.3. Dashed tracks mean that part is invisible to the detector, while straight lines indicate that the detector sees the particle. Electrons are light particles that lose their energy in the EM calorimeter, while charged hadrons like protons go further up to the Hadronic Calorimeter. Photons are not detected by the inner trackers but leave traces measured in the EM Calorimeter when decaying into one electron and one positron. Neutrons are measured indirectly when interacting in the Hadronic calorimeter and converted into protons. The calorimeters do not stop particles like muons, so additional tracking devices like muon chambers are placed above them. Particles like neutrinos are the only known particles neither detected nor stopped by the different layers.

From the billion (10^9) particle interactions that happen at the collision point, only 1 in a million are flagged by the trigger subsystem as potentially of interest.

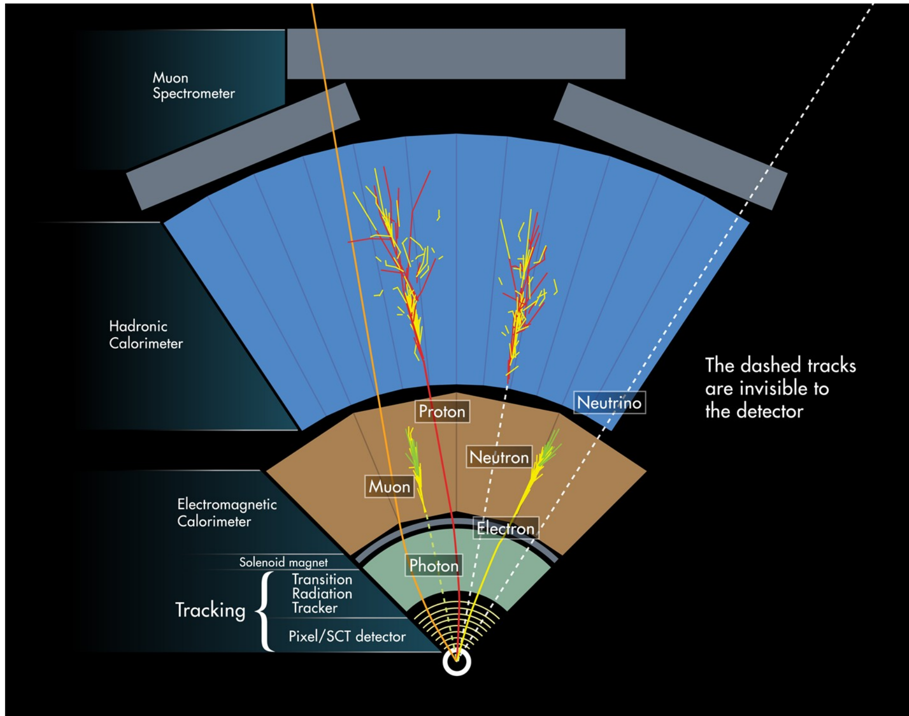


Figure 2.3: Diagram of particle tracks on the ATLAS detector subsystems [7].

2.2.1 Inner Detector

The Inner Detector (ID), as it can be seen in figure 2.4, is a 6.2 by 2.1 metres barrel situated just around the collision point of the beam lines. It is composed of three sub-detectors with different technologies, including pixel detectors, silicon strip detectors, and straw drift tubes. All of these are immersed in a high 2 T magnetic field parallel to the beams axis. They record the particles at the interaction points, known as hits, so with different hits along the sub-detectors the complete track can be reconstructed.

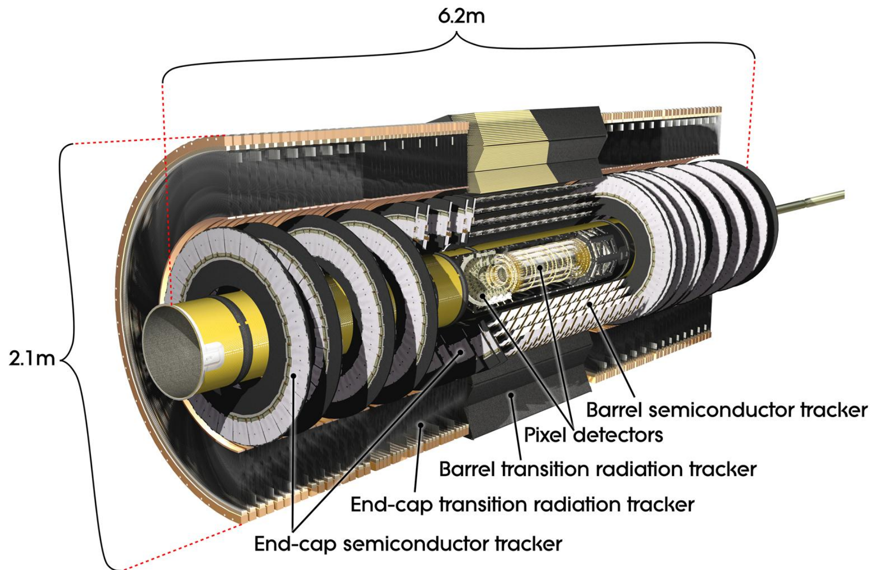


Figure 2.4: Diagram of the ATLAS Inner Detector showing the Pixel Detector, Semiconductor Tracker, and Transition Radiation Tracker [8].

Pixel Detector

The Pixel Detector is the innermost part and is composed of a 4-layer barrel of pixel sensor arrays arranged in 1,736 modules, and two end-caps of three-pixel disks each with 288 modules. The first layer located just 33.2 mm from the interaction point is the Insertable B-Layer (IBL) [9], which was added in 2014 prior to the start of the Run 2. The objective was to tolerate extreme radiation and the higher luminosity following the shutdown, so new radiation-tolerant sensor and electronic technologies were employed. The pixel size is 50 x 400 μm^2 for the external layers and 50 x 250 μm^2 for the innermost layer (IBL), comprising a total of 92 million pixel and electronic channels. This allows precise measurements of the decay vertex position, and to better distinguish particles.

Semiconductor Tracker

The next part of the inner detector is the Semiconductor Tracker (SCT), located in the range of 299 mm to 560 mm from the center of the detector. This is composed of 4 double layers of silicon microstrips in the barrel, and nine disks in each of the endcaps. With a total of 4,088 modules and 6 million readout channels, it measures particle tracks with a precision of 17 μm in the transverse direction of the modules, and 580 μm in the longitudinal plane (z or R).

Transition Radiation Tracker

The Transition Radiation Tracker (TRT) is the last part of the ID, and this is different since it uses 350,000 kapton straw tubes filled with a noble gas mixture of $Xe/CO_2/O_2$, separated with polypropylene between each of them. Inside the tubes there is a thin gold-plated tungsten wire, which is able to collect the electrons produced by ionization of the mixture gas when the particles travel through the straws. This is used for track reconstruction. The difference in refraction indexes of the materials produce an amount of transition radiation related to the mass of the particle, which allows electrons and positrons to be distinguished from pions.

2.2.2 Calorimeters

ATLAS uses sampling calorimeters that are composed of layers of absorbing materials to stop the particles as much as possible, and sensing materials that measure the energy. As can be seen in figure 2.5, the calorimeters come after the Inner Detector and the solenoid magnet. These are composed of the inner Liquid Argon (LAr) calorimeters, and the Tile Calorimeter, which fill the barrel and end cap sections.

Liquid Argon (LAr) Calorimeter

Liquid Argon is used as active material for several reasons, including linear behaviour, stability response and radioactivity resistance. Lead, copper and tungsten are used as passive materials.

The LAr Electromagnetic Calorimeter (ECAL) is a 6.4 m long barrel, 53 cm thick, with a carefully designed accordion structure to detect all the particle showers and to specially identify electrons and photons. It is immersed in a cryostat to maintain the temperature of the medium at -183 degrees Celsius.

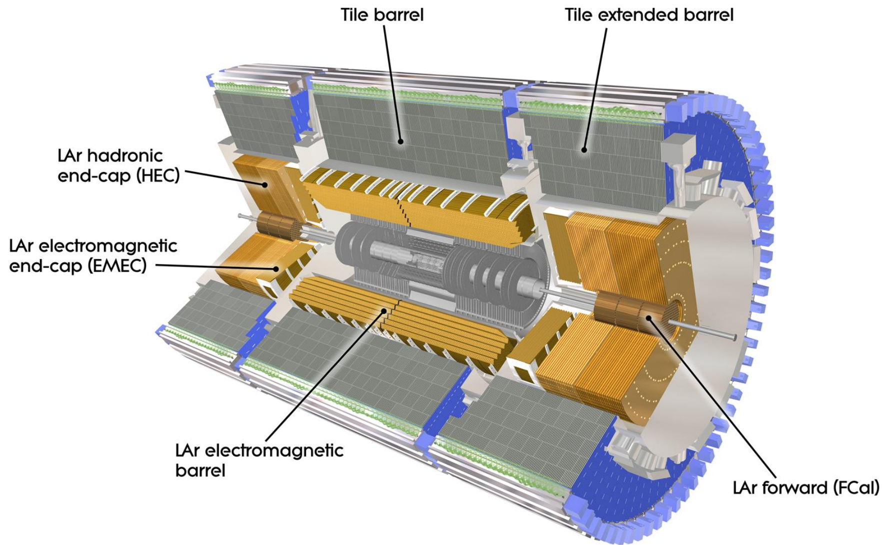


Figure 2.5: ATLAS Calorimeters [10]

There are 110,000 read-out channel cables situated in a vacuum environment to preserve them.

End-caps are contained in another cryostat and house the liquid-argon hadronic end-cap calorimeter (HEC), electromagnetic end-cap (EMEC), and the liquid-argon forward calorimeter (FCAL). The HEC is composed of two wheels 0.8 m and 1 m thick each, with a radius of 2.09 m. FCAL is composed of three modules of a radius of 0.455 m and thickness of 0.450 m each.

This calorimeter measures a wide range of particle energies, from 50 MeV to 3 TeV, and plays a critical role in selecting only an interesting selection of collision events, which is also known as the trigger.

During Run 3 (2022-2025) the luminosity, or number of interactions per collision, will be increased and so the number of background and unidentified processes by typically a factor of 5. During Long Shutdown 2, improvements were made to the electronics [11], including the installation of new Super-Cells which provide increased granularity for the calorimeter layers, and a 23.6 Tbps read-out data rate. Also 1,524 Front-End readout boards were refurbished, 124

new LAr Trigger Digitizer readout Boards, and 5,000 fibres installed. Running the new digital trigger will be done in parallel with the legacy trigger [12], until it is fully validated, as it is a critical part of the system.

Tile Calorimeter

Particles that escape the LAr calorimeter are measured by the Tile Calorimeter (TileCal), inducing hadronic showers via the ionization and strong interaction. This is composed of alternating layers of steel producing the particle showers, and plastic scintillating tiles that induce photons, whose electric current is measured.

As can be seen in figure 2.5, the structure is divided into a 5.6 m central long barrel, and 2 extended barrels which are 2.6 m in length. This is the heaviest part of all the ATLAS detector, at 2,600 tons in weight.

The 420 k scintillating tiles are read-out with wavelength-shifting fibers, grouped in bundles that are then read-out by 9,852 photomultiplier tubes.

Front-end electronics sum channels in trigger towers, which will be the L1 trigger basis.

2.2.3 Muon Spectrometer

As previously discussed, muons pass by unnoticed in inner parts of the detectors, and it is specifically the task of the last parts to measure them. The Muon spectrometer is divided into three parts, one in the central barrel and two in the endcaps. The different sub-detectors can be seen in figure 2.6.

These are immersed in the iconic toroidal magnets that provide a magnetic field of up to 3.5 T, in order to measure the momentum of the muons. At 25.3 m in length and weighting 830 tons, the central magnets are the biggest toroidal magnets ever built. The parts in the end-cap are 10.7 m in diameter and weight 240 tonnes each.

To accomplish the muon tracking objectives two set of detectors are used, the muon trigger system chambers and the muon tracking system chambers.

Muon trigger system chambers

The muon trigger system is composed of the Resistive Plate Chambers (RPCs) and the Thin Gap Chambers (TGCs) as can be seen in figure 2.6, and the New Small Wheels (NSW).

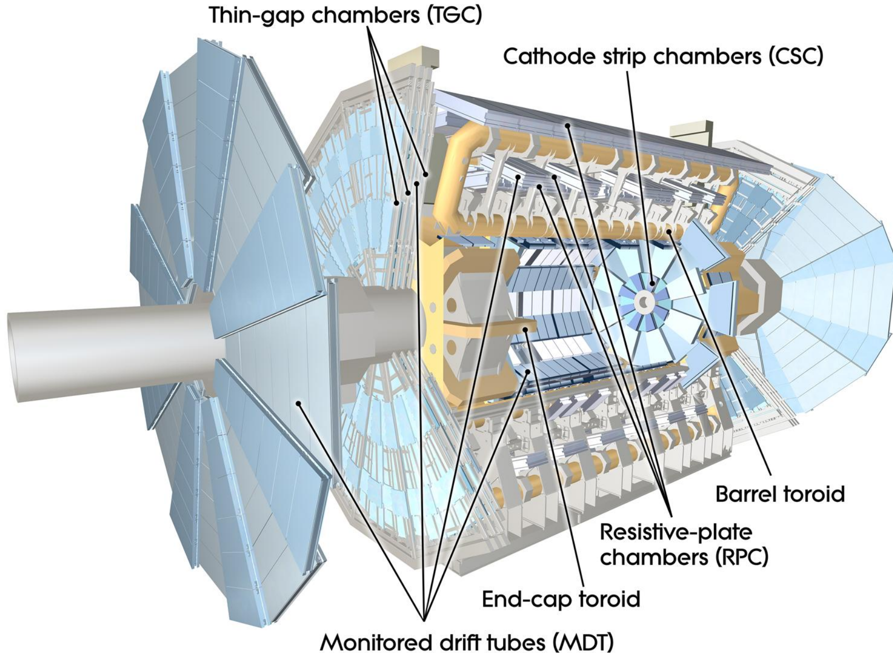


Figure 2.6: Computer generated image of the muon spectrometer and its parts [13].

The Resistive Plate Chambers (RPC) are made of parallel metal strips filled with a gas mixture that is ionized when a muon particles travels through them. The electric current induced is detected providing high resolution (1 ns), which is very efficient in rapid triggering. The Thin Gap Chambers (TGC) work on the same principle, but their chambers are composed of graphite-coated cathodes and a plane of high voltage wires along the chamber, providing a time resolution of 25 ns.

The New Small Wheels (NSW) [14] were installed during the long shutdown 2, in order to be ready for Run 3. They consist of two wheels 5 metres in radius, situated in each of the end-caps. They are built with two outer small strip thin gap chambers (sTCG) for trigger, vertex and bunch crossing identification; and two internal micromegas (MM) wedges for tracking with 100 microns space resolution. The objective of the New Small Wheels is to work in conjunction

with the previous Big Wheels to discard ghost hits from muon not originating in the bunch collisions, improving the previous Run 2 resolution and performance in this matter.

Muon tracking system chambers

As can be seen in figure 2.6, the muon tracking system is composed of the Muon Drift Tubes(MDTs) and the Cathode Strip Chambers (CSCs), which produce higher resolution position information, but have slower read-out systems. Thus, they are only used when a trigger decision is made by the trigger subsystem.

2.2.4 Trigger and Data Acquisition System

The rate of 40 MHz in bunch crossings and the luminosity affect the number of particle interactions that can be potentially detected. Recording all of them would mean storing 60 TB/s, which is unmanageable. Not all events are equally interesting, so the Trigger and Data Acquisition System (TDAQ) [15] is in charge of the online selection of the events according to the physics analysis requirements at any particular moment, defined in the thus called trigger menu.

The TDAQ is made of a tiered structure, where the event rate is sequentially reduced to acceptable levels. During Run 1 it was originally made of a 3-level infrastructure: Level 1 (L1), Level 2 (L2) and Event Filter(EF), but starting with Run 2, L2 and EF were combined in the High Level Trigger (HLT).

As can be seen in figure 2.7, the hardware based Level 1 (L1) trigger is the first step in the event filtering that operates synchronously at 40 MHz, and reduces the event rate to 100 kHz. This also reduces the original data rate of the 100,000 read-output channels of the detector from 60 TB/s to roughly 160 GB/s. The next level is the High Level Trigger, which is a completely software based selection that further reduces the event rate to the order of 1.5 kHz and a corresponding data rate of 1.5 GB/s, taking a mean event size of 1 MB.

Level 1 (L1) trigger

The L1 trigger uses custom electronics to first decide events of interest. From the 2.5 μs time slot available for the L1 trigger, only 0.5 μs is devoted to the actual event selection due to the transfer times needed on the data acquisition subsystem. It is internally composed of a Central Trigger Processor (CTP) that interacts with the calorimeters and the muon detectors with three components: the L1Calo, L1Muon and L1Topo.

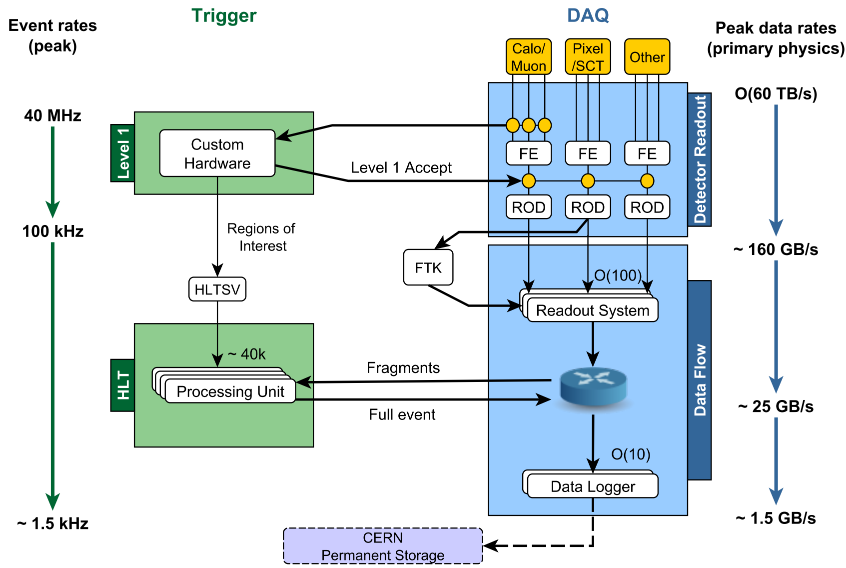


Figure 2.7: ATLAS Trigger and Data Acquisition subsystems. Image Credit: ATLAS.

The L1Calo reads lower granularity analogue signals from the calorimeters (see section 2.2.2), digitizing and preprocessing them. It searches for high energy regions above the programmable menu threshold, which are defined as regions of Interest (RoI). The results are sent to the L1Topo and CTP modules, including total and missing transverse energy values.

The L1Muon searches for hits coming from the RPCs and TGCs, including the NSW presented in section 2.2.3. It sends the information to the L1Topo and CTP modules in search of muon particles coming from the center of the detector.

The FPGA-based L1 Topology (L1Topo) module calculates topological information like angular distances and invariant masses from the trigger objects calculated in the previous modules, and forwards this data to the CTP.

The Central Trigger Processor (CTP) makes a decision taking into account

the amount of energy of the event, but also the number of objects above the defined thresholds and the topological information. It is also in charge of honoring dead times to prevent overrunning of the Data Acquisition (DAQ) buffers for consecutive L1 accepts, or for the number of L1 accepts accumulated in a given number of bunch crossings.

When a L1 accept is signaled the data is read from the Front-End (FE) electronics of the DAQ system first into the ReadOut Drivers (ROD) which performs the first processing. Then it is sent to the ReadOut system (ROS), that buffers the data for the next stage. From there it is transferred only when requested by the High Level Trigger (HLT), in addition to the identified Regions-of-Interest (RoIs).

High Level Trigger (HLT)

The High Level Trigger (HLT) is a pure software trigger approach with 40,000 Processing Units (PUs) running in commodity processors. A fast rejection approach is followed using first information from the RoIs, and requesting partial or full event data from the different subsystems stored in the ROS buffers when necessary. This allows requests of event data to be made from within RoIs and to apply algorithms to reconstruct features, applying the decisions within a few hundreds of milliseconds. The software infrastructure is based on Athena [16, 17] allowing multi-threading complex algorithms to be applied and detector configuration information to be used if needed.

When the HLT accept is signaled, the event data is stored by the Sub-Farm Output (SFO) data logger to permanent storage at CERN Tier-0 [18]. During Run 2, the rates during a typical physics data-taking run (see following section 2.2.4) were on average about 1.2 kHz on the number of events, and 1.2 GB/s to permanent storage.

Trigger configuration

Event trigger decisions are done by means of trigger chains, where each chain consists of a L1 trigger item and a list of HLT algorithms. The list of the trigger chains is defined in the trigger menu.

Designing a good trigger menu is crucial to achieving the goals defined in the physics program. This ideally means having events for all processes, including rare ones. Some more common processes can be pre-scaled, randomly selecting some of them according to defined values. A value of n for a particular pre-scale means we want to select 1 in n events. There might be L1 and HLT defined

pre-scales for every value in the trigger menu, in order to reduce the bandwidth consumed.

The trigger menu along with other configuration values are needed in order to the correct interpretation and reproducibility of the trigger in the offline analysis framework.

A relational database called TriggerDB [19] is used to maintain the L1 and HLT configuration parameters and trigger menus which are fixed during a run.

Information that can be updated during physics data-taking, like condition parameters, is stored in a dedicated database [20], but also referenced in the TriggerDB. Other information stored are the pre-scales defined with each L1 item or HLT trigger chain, and the bunch group set key.

Data files record the trigger decisions per event in trigger bit masks, where each bit refers to a particular trigger chain. Since the configuration can change, the same bit can refer to a different trigger chain, and this relation is maintained in the TriggerDB, in a table indexed by the trigger supermaster key (SMK).

Bunch identification

In the 25-ns bunch fill spacing scheme used during Run 2, there are a total of 3,564 bunch spaces or crossings that can be defined in the LHC revolution. These spaces can be emptied of protons, only with one bunch, or with the two bunches colliding (up to the 2,808 designed). A Bunch Crossing Identifier (BCID) is generated in the range from 0 to 3,563. BCIDs are merged in bunch groups, which are used to be paired with L1 trigger accept decisions.

ATLAS run operational mode

An ATLAS run is defined as a data acquisition period of time at stable LHC conditions, which usually lasts from a few hours to over a day. For the physics data-taking this coincides with a typical LHC fill cycle (section 2.1) and is ideally 10 to 15 hours in the stable beams for physics data-taking. Other kinds of runs are possible beside physics, for example with cosmic ray data-taking when there is no beam at the LHC, in order to do calibration or study detector performance.

When a run starts, the Data Acquisition (DAQ) system assigns a consecutive and unique run number. An event number unique identifier inside the run is assigned to every recorded event, which is reset at 0 with each run start.

A run is divided into Luminosity Blocks (LB) with an approximate length of one minute, although this can configured and changed online in the same run.

During a LB, luminosity is kept constant, and detector conditions and trigger configuration are kept stable. This time period is considered the smallest period of time that can be declared correct or incorrect for data quality purposes.

Main physics data is output from SFO in byte-oriented streams in raw format [21], but also other streams are set up for calibration, express data quality checking, debugging and other tasks. At Tier-0 facility first calibration and express streams are processed, in order to provide detector calibration and alignment configuration variables. Then bulk physics data is processed in 24-48 hours with this calculated time dependent information, in order to produce reconstructed real event properties and physical quantities. The main output of this reconstruction phase are the Analysis Object Data (AOD) files that are stored at CERN but also distributed to other centers.

2.3 ATLAS Distributed Computing

ATLAS follows a distributed tiered model for the distribution, reprocessing and analysis of the data. The Worldwide LHC Computing Grid (WLCG) [22], or the grid for short, comprises a set of distributed computing technologies to distribute and access data and computing resources. Tier-0 at CERN keep all the raw data, and in the order of ten Tier-1 centers duplicates copies of this data. Then around one hundred Tier-2/3 centers are mostly devoted for data analysis. A visualization map of the WLCG centers can be seen in figure 2.8.

ATLAS currently uses in this distributed infrastructure over 700 k CPU cores, 230 PB of online disk and 270 PB of tape storage.

It is common to have better knowledge of the conditions with improved detector calibration and alignment constants, usually after long data taking periods. Also new and improved algorithms might be produced, making the reprocessing of the data possible. In this case, new additional versions of the previous event AOD files are produced at the Tier-1 centres, and then distributed.

AOD files are too big to be used directly, so more convenient formats are produced to be distributed. Derived AOD (DAOD) files are produced centrally according to the requirements of the physics analysis groups with only the events and required contents. These derivation processes can be run very frequently, producing in practice multiple versions of the same original information distributed in different files, according to the requirements and software releases used.

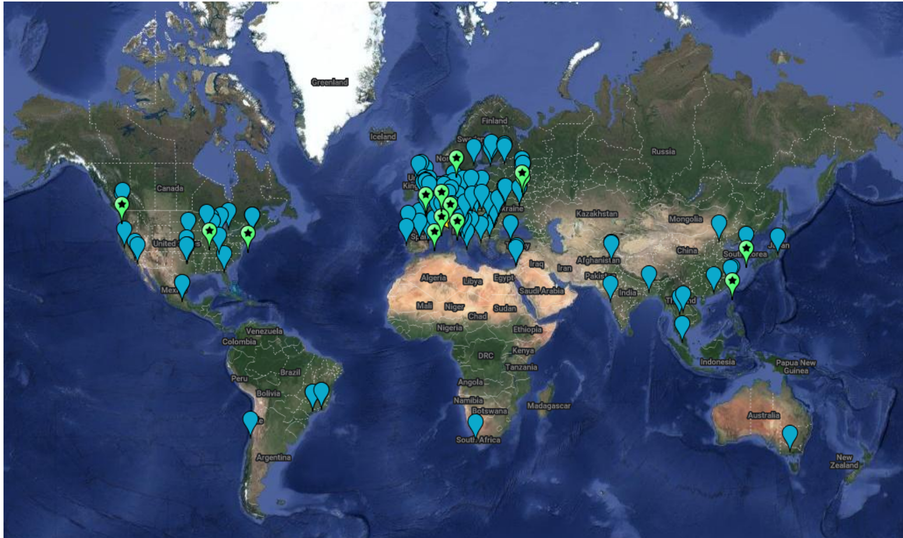


Figure 2.8: Wordwide LHC Computing Grid (WLCG) sites in September 2022. Tier-0 is at CERN (Geneva); Tier-1 centers are marked with a green star sign; Tier-2/3 centers are marked with blue sign [23].

Simulated data is also produced in the grid with MonteCarlo probabilistic methods [24, 25], in order to test the understanding of the performance of the detector, calculating reconstruction efficiencies, and modelling other processes. The first step is the event generation in a common EVNT file format. Then the detector simulation is run to check which generated particles interact with the detector materials, and the energy deposit within them. The digitization process converts the simulated energy into a response in the detector, to look like the real recorded raw data. After this step the procedure is the same as for real data, generating AOD and DAOD files.

Real and simulated event data are stored in files that reside on disk or tape, and that are identified in the grid by a Global Unique Identificator (GUID) [26]. Each of these files contain a number of events ranging from 1,000 to 10,000 records, depending on the format and the event size. This is controlled in order to have output file sizes from 1 to 10 GB which are manageable in the grid, but there are no limitations to producing smaller or bigger files.

Files are logically grouped into datasets, and datasets are grouped hierarchi-

cally into containers. These are named according to a defined nomenclature standard [27].

These data identifiers for files, datasets and containers are registered in the distributed data management Rucio [28] tool, which tracks the original physical location but also the replicas along the grid. With this tool ATLAS users can access over 100 million files on disk distributed in the grid, containing more than 400 billion event records. A number of metadata attributes can be assigned to files. Dataset and container attributes can be explicitly created or be inferred from its constituents. These attributes include system defined data (file size, creation and modification times, checksums), and physics data attributes (number of events, campaign, project, datatype, run_number, stream_name, prod_step, version, campaign, lumiblock). Workflow management attributes can help to identify which processes created the data. Data management attributes can control how the replicas must be created, effectively ordering data placement if necessary.

2.4 ATLAS Computing and Data Challenges in Run 2 and Run 3

Since its conception the LHC and its experiments have been planned to be progressively upgraded to reach higher energies and increase the luminosity in order to better study the physics processes, the fundamental components of the matter and the forces among them.

The schedule of the LHC and the next High Luminosity LHC can be seen in figure 2.9. When the LHC started in Run 1 in 2011, the energy at center of masses reached 7 TeV. Energy has been increasing and when the Run 2 started in 2015, the energy at center-of-mass reached a much higher 13 TeV with a integrated luminosity of 190 fb^{-1} (inverse femtobarn). The inverse femtobarn is a measurement of particle collisions per unit of area (barn), which represents a measure of both the number of collisions and the amount of data collected. One inverse femtobarn corresponds to approximately 100 trillion (10^{12}) proton-proton collisions.

Currently we are on Run 3, which started in July 2022 with a collision energy of 13.6 TeV, the highest energy reached by a particle accelerator. It will collect approximately 450 fb^{-1} when it ends in 2025.

It is planned that the High Luminosity LHC will start at the end of the decade, with an unprecedented record from 7 to 10 times the current data taking. With an increase in luminosity, the pile-up will also increase from the current 30-60 to a future 200. When the Run 5 ends it is expected to have

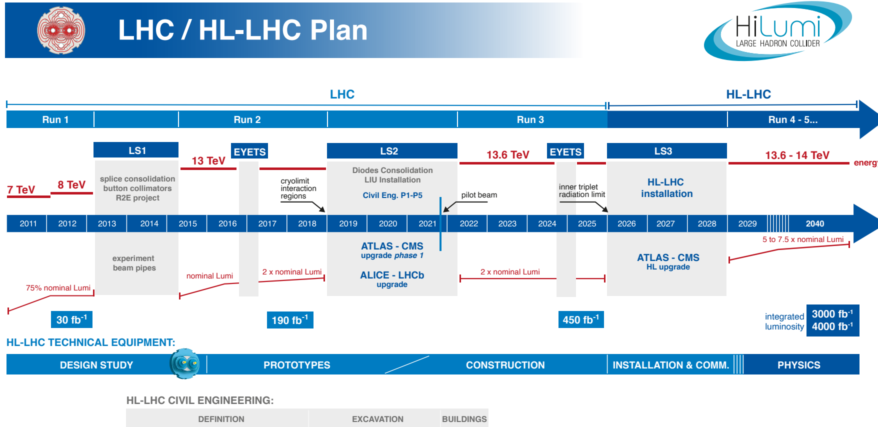


Figure 2.9: LHC/ HL-LHC Plan (last update February 2022) [29].

recorded as much as ten times the data recorded than in the three first Runs of the LHC.

Challenges arise with the LHC and ATLAS upgrades [30] as there is an increasing number of events to be analyzed, and their complexity (due to higher pile-up and track multiplicity) increases. This has been the case in the recent past and will be of an order of magnitude higher in the next Runs. In addition a number of simulated events has to be produced with higher fidelity. The current flat budget compromises the success of the following runs, if there is no aggressive research devoted to the computing and data challenges.

Estimated requirements on CPU usage (in Million HS06-years [31])for the following years can be seen in figure 2.10. The blue dotted line, and red triangle lines represent conservative and aggressive research and development as defined in [30]. The black lines forecast a 10% and 20% increase in resources capacity due to technologies or budget improvements for every year, with a factor 2 gap if there are no improvements due to research advances.

An update of the ATLAS computing model was performed for Run 2 [32], with a more distributed and not so strict hierarchical model where Tier-2 computing nodes communicate with each other without the need for all data going through their parent Tier-1. Additional non pledged opportunistic resources

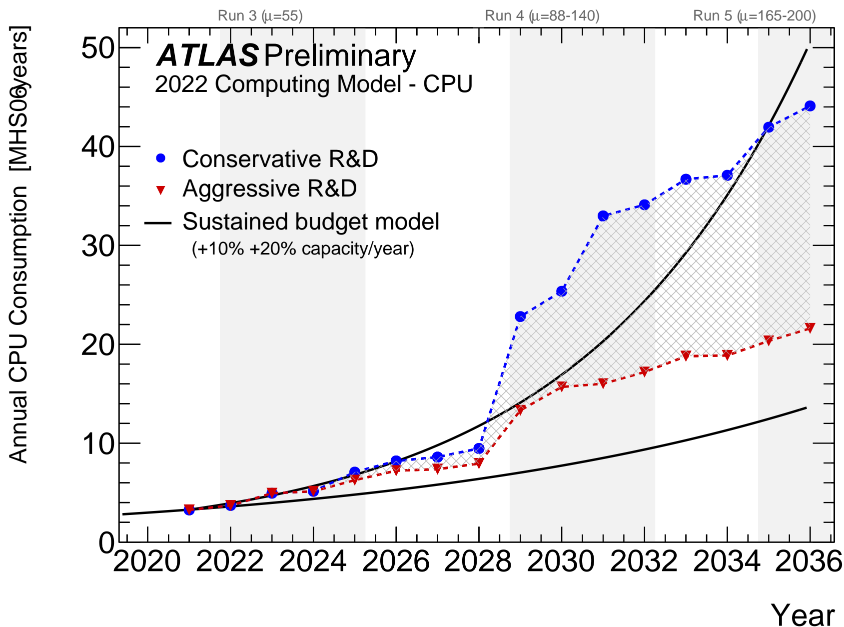


Figure 2.10: LHC / HL-LHC projected evolution of compute usage from 2020 until 2036 [30].

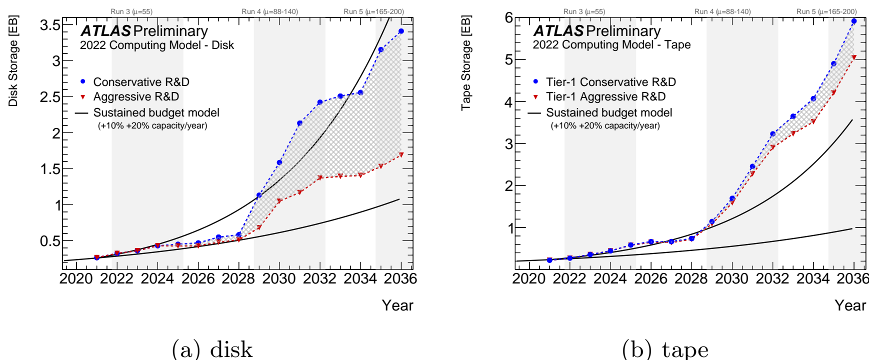
on High Performance Computing (HPC) resources have been used in the last years and may improve the situation.

The commercial computing capabilities are reaching the limits on single core performance as can be seen in the microprocessors trends in last years [33]. In order to increase computing power, more cores are added which makes the optimal usage challenging with a paradigm shift.

A multiprocessing version of Athena was produced, but the requirements in terms of memory usage have not profited for all CPU cores. For Run 3 it was clear that another approach was needed so it was completely redesigned to produce AthenaMT [34], yielding dramatic improvements in memory utilisation.

Regarding storage, ATLAS Run 2 analysis model was very successful but expensive on disk space usage.

We can see the storage requirement projections on figure 2.11, with disk usage in 2.11a and tape usage in 2.11b. Like in the case with the CPU consumption,



(a) disk

(b) tape

Figure 2.11: LHC / HL-LHC projected evolution of disk and tape usage from 2020 until 2036 [30].

we can see also the shortage of resources that justify the investment on research programs.

During Run 2, a new Event Data Model (EDM) was implemented, called xAOD [35]. The information is organized in a columnar way, which allows accessing on-demand partial parts of objects. Also improves performance increasing the locality of memory references. The xAOD format is ROOT compatible, so the current production files are named AOD by extension, although having this new internal format.

In Run 2 the centralized Derivation Framework [36] was introduced, in order to produce specific derived data (DAOD) for analysis groups. The format is derived from the output of the reconstruction (AOD) by removing non required variables (slimming), defined internal objects (thinning) and complete events (skimming). Also it was possible to include new variables or defined objects as required by the analysis groups. All individual information per-event of the maintained objects is included in order to perform calibration and assessment of the instrumental uncertainties without explicit user intervention. Overlaps among different derivations is however produced by the different groups. Therefore many derivations which might duplicate information and the file format are still occupying an amount of data per event that makes it challenging to be able to continue with this model for the next runs.

A new analysis model is being implemented [37], to achieve a significant reduction on disk usage with new formats like DAOD_PHYS (with 50 kB/event) and DAOD_PHYSLITE (with 10 kB/event). The first DAOD_PHYS format

contains all variables to apply calibration on reconstructed objects. It will be introduced in Run 3 in order to halve the disk usage for all analysis data formats. The DAOD_PHYSLITE format contains precalibrated reconstructed quantities, reducing the footprint by removing all the variables for explicit calibration procedures. It will also be progressively introduced in preparation for the next runs, which might require adaptations on the users. In addition, lossy compression techniques are being applied, by reducing the precision on some variables. With these techniques a reduction of 10% might be achieved when applied to reconstructed objects.

There is also the need to improve metadata (data about data) in order to quickly locate events and to assure data quality. There is a number of constraints [38] regarding whether it is stored in-file or on external databases. Metadata can refer to data provenance as to the origin, software versions and procedure to generate the data. Also bookkeeping information for filtering samples, or data quality for good or bad data list at event, dataset or run granularity. Information about the calibration and time dependent data is also needed to real data reprocessing, or parameters for Monte Carlo generators of simulated data.

In-file metadata was recently redesigned in the Athena framework [39], in order to support concurrent event processing in multi-threaded simulation and reconstruction workflows. This is also the first step to support heterogeneous architectures, like GPU.

Central catalogs have been developed in ATLAS for accessing metadata at dataset level (AMI [40]) and run level (COMA [41]). In the following section we will present EventIndex, our catalog for accessing event-level metadata.

3 EventIndex Project

3.1 Introduction and Goals

Physicists usually work through big quantities of data, but access to individual events is also needed in order to check the details. This is done for example for correctness checking during reconstruction phases, or just to produce event displays for figures in publications.

We have seen that an event can be reconstructed with different conditions and software releases, producing in practice multiple event versions. Also production procedures create event records with different information in different file formats, and thus having information on the event lineage is a plus.

Access at event record level is also convenient for data quality checking. Duplicate records that might be produced during data acquisition temporary failures or during data processing can be detected. Also the overlap calculation of events across different files produced during derivation procedures is valuable in order to optimize resource usage.

Studies on the trigger correlations and overlaps between data streams can be also conducted at the event level if the data is available.

Tools like Rucio [28] provide access to data at file, dataset or container level. They are not designed however to provide insights at the event granularity level.

Therefore an event catalogue that indexes all real and simulated events, at all processing stages, is needed.

ATLAS was using a previous event-level catalog called TAG database [42] during LHC Run 1 (2009-2013). The idea was to obtain event wise information with a special kind of TAG files, produced at Tier-0 from the AOD files. These TAG files were directly imported into an Oracle database, which was the only proven technology at that time capable of sustaining the required amount of data. The TAG DB information could be then used directly for selecting and

extracting some events (skimming) on the basis of physical variables for many user analysis directly, before accessing the full AOD contents.

The concept was very promising but suffered from two main design and operational issues.

First, it did not differentiate between immutable and mutable data. Immutable data consist of system parameters taken during data acquisition phase such as event number, run and other identifiers, and trigger decisions. Mutable data refer to physics variables that are dependent on the accelerator and detector conditions, and that might change in successive reprocessings. As the reprocessings occur frequently and in the grid, the corresponding TAG files were not always created, leading to outdated physics information in the TAG DB.

Second, the implementation on the Oracle database was directed by the structure of the TAG files. This requirement was established in order to be capable to generate TAG files (to do event skimming), directly from the Oracle tables. This approach actually grew into Oracle tables with hundreds of mixed mutable and immutable columns, with indexes over all of them (in order to do searches efficiently within the tables). The higher storage necessities for this approach were not effectively of benefit to the users due to the outdated physics information aforementioned.

The EventIndex was designed to catalog and access billions of event records in an easy and efficient manner. Only a small quantity of metadata per event was required to solve the envisaged use cases. Before starting this work, it was decided to avoid mutable data, as starting with Run 2 there were other mechanisms for applying event skimming with the new production data formats and derivation framework [36]. Even storing only immutable information translates into big amounts of data when indexing billions of events (in millions of distributed files, occupying petabytes of storage).

EventIndex leverages the rise in the research on big data and NoSQL databases in order to scale the increasingly higher amount of data to be stored for the successive runs of the ATLAS experiment. This technology allows a cluster of machines to be used, which can be increased to scale horizontally. In addition, the usage of free open source products dramatically lowers the cost over using a commercial database like Oracle.

Data is always updated thanks to a new indexing and data collection strategy, which is able to extract the required information from any kind of produced data format and not only AODs. We deploy our own EventIndex distributed indexing jobs in the grid, collecting the metadata centrally and making this immediately available to final users. Thus EventIndex can be considered a

representative real application collecting big data on large scale distributed infrastructures.

3.2 Use Cases

The use cases that the EventIndex is currently solving can be grouped in OLTP-like workloads, where only one or a few records are accessed, and OLAP-like workloads for analytics over larger quantities of data.

The *event peeking* use case allow users to select single or few events depending on provided constraints. The final objective is to check for data availability, and to obtain location details on the full event information in a particular format. With this location information the full event record can be retrieved with the proper tool (Rucio) from the files in the different formats, up to the original source (RAW).

A second group of use cases relates to data consistency and quality checks. The *duplicate event checkings* looks for event records with the same identifiers appearing at different storage granularities, from a file to within a dataset or containers. This undesired situation might be caused by temporary problems during the detector data acquisition, or by production issues during the reconstruction chains. For simulated data, it can detect software glitches for generating unique identifiers, or when merging several sources of data. EventIndex indexing jobs are usually the first ones to access the data, so they provide additional checking about availability and correctness for further user analysis. There are other cases where we need to check for data over multiple datasets. For example the derivation framework can produce derived datasets from an original one selecting only useful events for particular analyses. There are about one hundred official derivations in ATLAS, and the same event can end in multiple output derivation streams (in files, grouped in datasets). In this case it is advisable to check what streams are that contain overlapping events, in order to remove them and optimize the grid storage. With *event overlap calculation* tools we can produce the overlapping matrix identifying common events across different files.

The last group of use cases is related to the trigger decisions. We can apply *trigger checks* counting events satisfying a particular trigger, or providing and event list and details based on trigger selections. We can also apply similar *trigger overlap calculations* on a particular real data run or stream, detecting the events satisfying particular trigger pairs. In addition a number of trigger statistics can be calculated with the available data.

3.3 Data Model

In order to solve the required use cases we need to index information for real and simulated events. We can obtain all the required information from final stored files, as they have common formats. Only immutable information is stored, so there is no need to update information due to recalculation of physics variables on the reprocessing procedures.

Bearing this in mind, we need to store the following information for each event record:

- *Event identifiers.* Information for event identification of real and simulated events. An event record is uniquely identified by its run number, event number, the trigger stream, data format and version. Information about LHC conditions is also included, like the luminosity block (LB) number identifier and the bunch crossing identifier (BCID). Also information about the date of the event at the nanosecond level, and other metadata information about the stream that produced it including calibration, testbeam, or simulation. For simulated Monte Carlo events, generation information like the event weight and channel number is also stored, which eases the identification in case of software issues.

- *Trigger information.* Information about the passed trigger masks for the Level 1 (L1) trigger, the Level 2 (L2) trigger and event filter (EF) only for Run 1 data, and the High Level Trigger (HLT) for data starting with Run 2. Super master key (SMK) is needed to decode the trigger chains associated with the particular bits of the trigger masks, so it is also stored. In addition prescales keys for L1 and HLT are also included.

- *Location information.* Information where to find the full event record information in final files stored in the grid, in order to access and analyze the data for final users if necessary. The basic information is the GUID of the file to be able to retrieve it from the distributed grid storage. In order to access the specific event record inside the file, we need to include information from the persistency framework that stores the events in native format, which is currently based on Athena Pool/ROOT [43]. We want not only the current indexed record information, but also the provenance (data lineage) of the event. With this functionality we can obtain the information of the upstream files: for example indexing a DAOD file to obtain information about the AOD that generated it, and the RAW file before it.

3.4 Requirements

The proposed use cases require the indexing of all ATLAS produced data, real and simulated, during all the years of operation. In addition, all processing stages have to be taken into account, so reprocessings have to be also indexed. Indexing the data means accessing to the files to read them and extracting a small quantity of metadata per event, that varies from 300 bytes to 1 kB.

The files are distributively stored worldwide in the datacenters belonging to the ATLAS WLCG infrastructure, using grid technologies for computing and data access.

The indexed metadata has to be always accessible for all the members and processes of the experiment, so a central catalog is needed. A distributed data collection procedure is required to convey the metadata from the grid centers to a central catalog.

The metadata catalog has to be able to scale with the requirements of the current and future runs of the ATLAS Experiment.

The production rates have been in the order of 1 kHz of real data events from the detector during Run 2. To this number has to be added the production of simulated data, which can be more than double of the real data. In addition the reprocessing of the data that is done from time to time adds production peaks that have to be absorbed. For Run 3 an increase is expected in the rates of up to 3 kHz in real data and a similar simulated data proportion to previous runs. For next runs however a factor 10 is expected in the rates, meaning the production of 100 billion (10^9) real events, and 300 billion simulated events.

The query number and rates are much lower than the insertion rates. For event picking use cases, requests that search for one single event is the common case. Then there are other requests that search for a list of events, which might be up to the order of thousands. Rates are low with those up to 1 Hz, but users expect a fast response in the interactive service, in the order of a second. Other analytical use cases, like duplicate or overlap checking, can be done with specific requests or in the background for all the data. For many background cases the response time need not be immediate but rather in the order of one hour. In any case background jobs can be delayed if the system needs to be prioritized for ingestion and interactive query tasks.

The command line interface is preferred by users and automatized processes. The web interface is also very useful for data discovery and in general for first time users.

3.5 Architecture

An overview of the architecture of the EventIndex can be seen in figure 3.1.

The Data Production component is in charge of selecting the data and submitting the event metadata extraction jobs in the grid, in order to produce the required metadata. The distributed Data Collection component sets up the transport infrastructure to convey the metadata to the central catalog at CERN, assuring its completeness and quality. It also ingests the data in the final data storage backend in the correct format. The Data Storage is the core component implementing the metadata catalog in scalable storage technologies.

The Data Access component provides the interfaces for users to access the data. The Monitoring component checks the servers and services of the rest of components, collecting metrics and running functional tests.

During the lifetime of the project several components have evolved in order to fulfill the requirements and solve issues detected during the first runs of the project. Thanks to the differentiation of the components and the evolution of the technology it has been easier to evolve the system without compromising the production level of the provided service.

This thesis is focused on the contributions to Data Collection (chapter 4, Data Storage (chapter 5) and Access (chapter 6) components. In the following sections we will detail the challenges and issues that motivated the changes and the studies and solutions that were proposed.

3.5.1 Data Production

Extracting the metadata starts as soon as data coming from the ATLAS detector is reconstructed in Tier-0 and the AOD files are produced. Only physics datasets are indexed, excluding calibration, express and debug streams. This is the main output and format used for successive derivations and user analysis. Indexing RAW files is not necessary because the event location information will be in the AOD file data provenance records, and therefore extracted.

Reprocessings and derivations are carried on the WLCG grid, with the production of new releases on AOD and the derivation AOD (DAOD) datasets which are also indexed. These also include the newer DAOD_PHYS and the DAOD_PHYSLITE file formats [37].

Simulation data is also generated in the grid and other opportunistic resources like HPC for all the production chain, including event generation and detector simulation. For the simulated data, all EVNT files and all the produced

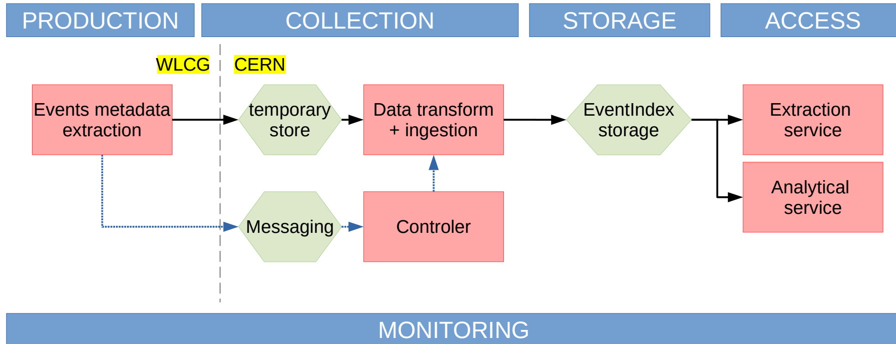


Figure 3.1: High level view of the EventIndex architecture composed of 5 areas: Data production, data collection, data storage, data access and monitoring. Green hexagons correspond to temporary or permanent data, and pink rectangles represents processes. Black arrows show the EventIndex data flow, and blue arrows show the flow of information related to data processing.

AOD files are indexed. Selected derivations in DAOD are also indexed as per explicit request in the physics user groups.

Selection of the datasets produced in the grid is done searching the registered data with the AMI tool [40]. A number of variables is checked in this dataset catalog, including that the datasets are part of the official production; that are not for debug or test purposes; and that are complete for all its file constituents and with a valid state.

This search and selection is done continuously and the list of datasets to be indexed is processed by the grid Production and Distributed Analysis system (PanDA) [44, 45]. The list of datasets is taken by the production system and a number of computing jobs are generated that are submitted to the distributed grid centers where the data is available.

Dataset naming

A dataset name follows the ATLAS nomenclature [27], and it is composed of six fields separated by dots. For real data it has the form:

```
Project.runNumber.streamName.prodStep.dataType.
↪ AMITag[_tidnnnnnn[_SS]
```

Similarly for simulated data has the form:

```
Project.datasetNumber.physicsShort.prodStep.dataType.⌟
↪  AMITag[_tidnnnnnn[_SS]
```

The list and description of the fields follows:

- *Project* represents a particular physics or computing context. For example *mc16_13TeV* means simulation of data at 13 TeV during the Monte Carlo campaign of 2016, and *data22_13p6TeV* corresponds to real data taken in 2022 at collision energy of 13.6 TeV.

- The second is a pure numeric field used to indicate the real data DAQ run number or, in the case of Monte-Carlo simulatad data, the datasetNumber.

- The third field is the *streamName* in the case of real data, which identifies the data stream (for example *physics_Main* for the primary physics stream). For simulated data the *physicsShort* is a text description of the event generation and detector simulation in this dataset.

- *prodStep* gives the last production step which was used to create the data, like for example *recon* (reconstruction), *merge* (after processing several inputs), *deriv* (group production), and others.

- *dataType* field identifies the format of the files in a dataset, with the first part (dataTypeFormat), for example: *RAW, ESD, AOD*. Some contain a second optional part (dataTypeGroup) to describe the group for which the given dataset was created: *DAOD_EGAM7, DAOD_TOP4*.

- The last field is the version, or *AMITag* chains (as defined in AMI). Each processing step changes adds the related AMI tag used separated by a underscore. For example: *f476_m1223* describes Tier-0 bulk reconstruction (f), and file merging (m) with a particular ATLAS release.

If the *_tidnnnnnn_SS* suffix is included, it is called a tid (task identifier) dataset, that was created by the Panda production system task number nnnnnn and subtask SS. The output of several tasks can be placed in a container, usually with the same name without the tid suffix (called a dataset container in this case).

Events metadata extraction

The EventIndex computing jobs are based on an Athena transformation [46] which defines a standard interface to access and process the data. The indexing transformation is implemented in python [47], as only access to event headers (EventInfo) is required to extract the event metadata. The python interfaces to C++ Athena Classes rarely changes among software releases, giving additional stability to the producer transformation.

3.5.2 Data Collection

Distributed data collection follows a producer-consumer architecture, where the producers run the events metadata extraction procedures at the grid, and the consumers run at CERN doing the ingestion to the backend data storage.

There is an infrastructure for conveying the metadata payload from the grid sites where the producers are running. Originally this was based on a messaging infrastructure in the first deployments, with several brokers to distribute the payload to the consumers.

Although the performance was adequate during the first deployments, some blockings were detected in situations with slow consumers that led to big backlogs of data to be ingested. These situations compromised the scalability of the system with the increased rates foreseen in the following Runs. The work developed in this thesis resulted in part of the development of a new system based on an object store for the payload staging, and which was put in production during Run 2 [48, 49] as we will see in chapter 4.

3.5.3 Data Storage

Data Storage is the essential component that maintains the data, scaling to the order of terabytes or even petabytes of event metadata information for the future runs.

During the first years of the project a pure Hadoop [50] implementation was designed and deployed. Hadoop Filesystem (HDFS) [51] has been used to maintain all collected data, with an organized directory structure based on ATLAS production system nomenclature, and with MapFiles [52] as data containers.

This has served as a data lake without the need for schema enforcement, although our data model has been defined since the beginning without much change.

The performance of random access for use cases like event picking was not satisfactory, so a part of the real data has been also stored in Oracle database [53]. A hybrid approach was also implemented [54] with good results but without storing some parts of the data (trigger) that occupy most of the space and without the simulated events metadata, reducing the total data volume.

Use cases also evolved in the meantime from event picking and production completeness checks, to more analytical use cases studying trigger and event overlaps, and duplicate event detection.

It was clear that a most general solution able to fulfil all use case requirements was needed. Also without incurring in duplication of part of the data like and the management burdens of having several data subsystems, consolidating all the data on a single platform.

With the development of big data technologies newer options arose, and studies on file formats and storage technologies [55] directed the exploration of new storage systems like Kudu [56]. The use of an enforced schema data model and columnar storage was promising for solving the EventIndex use cases. A prototype was proposed and tested (section 5.2) with good results but eventually its adoption was declined since Kudu was seen as a not mature enough product and without long time support, as required for production usage in the EventIndex service.

HBase [57] was also being used as a way to catalog the registered metadata in HDFS, and to cache some of the information to speed up random searches.

During LS2 and in preparation for Run 3, the work developed in this thesis has contributed in part of the redesign of the system to consolidate all the data in HBase as a unique store, with Apache Phoenix [58] providing data schema enforcement during data ingestion (section 5.3).

3.5.4 Data Access

With the original system with HDFS as main storage, two data access paths were used. The first path is for direct access like the event-picking use case, and this had to check what file to access (with the catalog) and then direct random access to a MapFile through its built-in index. The second path was data scanning over multiple MapFiles for the consistency checks and overlap calculations involving more data. For this second case a MapReduce [59] job is used for distributed data processing. After the first review and implementing the hybrid storage in Oracle, a web frontend interface was provided to access it.

With the redesign of the data backend system consolidating all data in HBase, the inclusion of Apache Phoenix allows new data access paths with low latency SQL-like access. The work developed in this thesis resulted in part of the development of new data access methods (see chapter 6). Queries are automatically transformed in a series of HBase scans instead of using MapReduce, although this framework is still available. Also, the SQL interface allows us to use JDBC protocol in order to connect the system with the web frontends already available to access the relational backends, as well as any other external tools. The analytic use cases are solved using Apache Spark [60], a data processing framework with its own data abstractions and support to higher level languages (Java, Scala) more suitable than plain SQL sentences.

3.5.5 Monitoring

The monitoring components track the servers and services availability for the EventIndex components and also collect metrics of their performance. During the lifetime of the project, these have gone through some upgrades to add functionality and adapt to newer tools used at central CERN services.

The first version of the monitoring tools [61] was based on Kibana [62] for collecting the metrics and showing the metrics for the developers and maintainers of the service.

For the Data Collection part it collected the status messages for the producers and the consumers, with details about the processing status at a particular moment [63]. The messaging monitoring also helped to identify blockings and message payload backlogs that were later solved in the new object store based architecture (chapter 4).

The monitoring also has evolved within the project [64] and is now using InfluxDB [65] as a time-series database for the monitoring metrics, and Grafana [66] for the visualization panels.

4 Data Collection

The indexing and extraction of the metadata in a distributed environment like the WLCG grid requires a data collection mechanism in order to convey and register the data in the central EventIndex catalog at CERN. From a functional point of view the duties of the data collection task define an ETL (Extract-Transform-Load) [67] pipeline, with the ATLAS data distributed worldwide as source and the core EventIndex storage as destination.

Requirements for the design of the data collection framework were identified as the following:

- Metadata extraction should be synchronized as much as possible with ATLAS data production and reprocessing procedures. This will reduce possible inconsistencies.
- The used resources should be as small as possible, in particular computing, storage and networking.
- The volume of metadata extracted per event should be small and therefore limited to the defined data model.
- The transport method should be independent of the common ATLAS data flow and management with Rucio (see section 2.3), as EventIndex metadata are temporary transient small files.
- Availability at central catalog should be fast, allowing users to locate and access data as soon as possible.

The metadata extraction procedure runs at Tier-0 as soon as AOD (analysis object data) files are produced, but also at the worldwide distributed grid centers. The architecture of the data collection was conceived as a producer-consumer distributed architecture. Producers are short-lived processes running

at Tier-0 and the grid centers, with potentially a high number of them indexing data simultaneously. Consumers are long-lived processes that run constantly at CERN, transforming the received data and ingesting in the EventIndex storage.

The data is conveyed reliably between producers and consumers with a transport method originally implemented with a messaging system. During the first deployment in production the performance was satisfactory, but also blockings were detected in peak production moments, leading to backlogs of data to be ingested. This was a risk for increasing data rates coming in the following runs, particularly a factor 3 only taking into account real data starting with Run 3 (2022-2025), and up to a factor 10 with Run 4. Thus a new design and implementation of a new data collection was conducted in order to solve these shortcomings.

4.1 Legacy Messaging Data Collection

The architecture of the original messaging data collection system [47] is shown in figure 4.1. There are a number of short-lived producers running at Tier-0 and grid sites that are launched by the data production task (as seen in section 3.5.1) when there is data to index. The procedure extracts a small quantity of metadata per event, in the order of 150 to 1,000 bytes per event depending on the file format and the trigger contents, which are the larger component of the metadata. After indexing procedure ends, the information is conveyed using a messaging system to the consumers, which will transform and ingest the data at the EventIndex storage, in this case the Hadoop core system using HDFS.

The messaging infrastructure consists of ActiveMQ (Red Hat JBoss A-MQ) [68] brokers that are managed by CERN, and receive the data in a dedicated message queue. The communication is based on the Streaming Text Oriented Messaging Protocol (STOMP) [69], which is a text based protocol chosen for its reliability and efficiency. The broker setup included a redundancy deployment with 5 brokers under a DNS round robin configuration, physically located at CERN Meyrin (Switzerland) and Wigner (Hungary) centers. The objective of the brokers is to decouple the data production from the data reception, effectively increasing the reliability in case of temporary issues on the consumers or the data backend systems.

The payload from the producers varies from 100 kilobytes up to several megabytes and is divided into small messages around 10 kB each, in order to maintain the brokers reliable and agile. Each message contains from 20 to

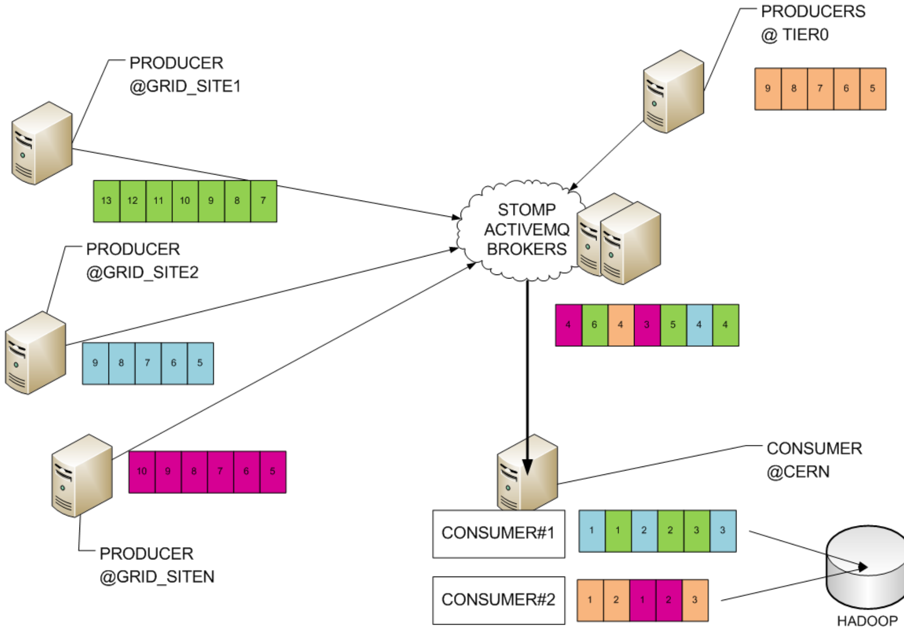


Figure 4.1: Data collection legacy messaging architecture. Producers communicate with consumers through messaging brokers. Color-coded numbered frames represents the messages from a particular payload, until stored in final Hadoop HDFS backend.

60 events depending on the aforementioned trigger information. Messages are encoded in JSON (ECMA-404 standard) [70] format, compatible with the text based requirements of the transport protocol. Standard protocols allow this to be language agnostic and use different programming languages for the entities of the system.

Messaging brokers do not ensure by default messaging ordering or atomic processing of all messages from a particular sender. This is necessary in our application since all messages from a sender are related and need to be consumed in order and by a single receiver, to ingest the data into HDFS. To overcome this shortcoming, messages are tagged at the source with a message group identifier (using the standard JMSXGroupID header), ensuring that all messages from a producer will be sent to one consumer allowing correct decoding and data

ingestion. This is shown in figure 4.1 with the color code and numbering of the message frames. Also transactions are used to group all related messages, assuring atomic processing.

4.1.1 Data ingestion

Consumers are in charge of the data reception and ingestion of the data with the correct format in the Hadoop HDFS storage. The duties of the messaging implementation architecture are the reception of the messages, the reordering and transformation of the data if necessary, and writing the files with a defined structure. Consumers are implemented as Java multi-threaded applications. A consumer connects to a single broker, so at least one instance should be started per each of the brokers. A consumer instance will have a unique identifier, and more instances can be started per broker to increase the throughput if necessary and scale up.

When connected to the broker messages start flowing to the consumer, and when the first message of a particular message group is received, then the broker assures the distribution of all the messages from that group to the same consumer. Each message JSON payload is decoded and control information about the production (Panda job id, dataset name, GUID of the file being indexed) is followed by the actual metadata. Data is internally organized in memory by GUID, so when successive messages related to a file arrive, they are appended in a queue until the last message from that file arrives. At this point, the memory queue is ready to be taken by the writer thread, which orders the received data by run number and event number (data might not be sorted in the original file or when processed by the producer) to be written in HDFS. Ordering is needed when writing HDFS MapFiles, which are internally sorted by a key composed of the run number and event number (see section 5.1).

Files in HDFS are grouped in directories named after the dataset name, and each directory contains a Mapfile per each one of the original GUID files that form the dataset. The MapFiles are named with a conjunction of the GUID, the consumer identifier, the panda job identifier, and the message group (JMSXGroupID) identifier, so they are unique file names in the dataset directory. These files are small (order of 10 MBytes) when compared to HDFS designed file size (order of GBytes). In addition files can contain duplicated data due to the production, so validation steps and consolidation into bigger files are needed.

Consumers send control messages to the broker to signal its state. A control message is sent when a MapFile is written with variables like the GUID of the

original file, the number of events, number of messages processed for that file, total size, job id, and processing. These messages are used by the data validator controller to ensure correct process. In addition a heartbeat message is send periodically (every 60 seconds) to inform about the consumer instance liveness, including the statistics (number of events, number of messages, etcetera.) of the process during that period. These messages are used by the monitoring tasks to assure the correctness of the data ingestion service.

4.1.2 Data Validator Controller

There are scenarios when the data is not completely received, or it is received more than once. Extraction jobs at the grid sites can fail without providing the output. What can also happen is that the output has been produced, but is incorrectly detected as not. Grid jobs can restart automatically up to a number or retries, but will maybe produce duplicated information. Messaging brokers can be in maintenance, consumers can crash or HDFS service can be restarted. Although not common, these failures can happen and therefore a fail tolerant system requires validation of the process and the data received. Data validation is done at dataset granularity, in several steps:

- Use the control messages to match produced with consumed metadata: number of events, number of GUID files, etc.
- All written HDFS files of a dataset are available and correct in format.
- If there are several candidate files for the same data (GUID) then apply a selection policy.
- A selection policy chooses best candidate using heuristics based on file name (formed of the identifiers aforementioned) and date. For example, use modification time only if there are no highest producer panda job identifier found.

After data is validated, some consolidation steps are needed. Extra files are removed, and a Hadoop core task is signaled so the dataset can be imported. The signal is done creating a small file in a shared directory (in AFS [71]), containing the details of the dataset. At this moment all the files from a dataset directory will be merged into a single bigger MapFile, which is ready to be usable.

4.1.3 Shortcomings

This system handled more than 10^9 messages, at an average rate of 100 messages/second produced, and peaks of more than 3,500 messages/second. It performed well most of the time, but we detected that the consumption rates were degraded in situations related to production peaks or when there was a backlog of messages at the brokers to be consumed [72].

Data collection was designed to be fault tolerant within the components of the infrastructure. Messaging brokers are configured in a high availability setup, so should any of them fail, others can take over the production load. Producers try to contact any of these in round-robin fashion, so a successful communication is achieved. In addition, producers themselves can be restarted by the data production system if a failure occurs. Regarding consumers, they can stop consuming messages if there is a scheduled shutdown on the backend HDFS Hadoop filesystem, or any other failure occurs. In this case the system is designed to continue working, with the brokers temporarily keeping a backlog of messages until they are consumed. Similar backlogs can be produced if the consumption rate does not keep up with the production rate.

Usually messages are processed in real time keeping up the production rates, and also when recovering from the failure scenarios aforementioned. We have seen however situations when messages are not being consumed, even when there are idle consumers attached.

This problem is related to the usage of messaging group tagging (JMSXGroupID). Brokers restrict the number of messages to be handled at a time, loading a reference into memory. When using message groups, this refuses to serve more messages until all message from particular group are consumed. This leads to Head-of-line (HOL) [73] blockings, with consumers starving without processing any other messages. Results are that brokers do not distribute the workload correctly to consumers, reducing the consumption throughput to 10 messages/second in some situations. This situation is specially detected when there is a slow consumer, which seems to block others. Adding more consumers does not solve the problem due to the blockings [72]. Adding more brokers could alleviate the situation, as we observed during the lifetime of the messaging implementation increasing the brokers progressively from 2 to 6. This did not solve however the issues with individual instances when backlogs reached few thousand messages. In addition brokers are expensive resources in terms of hardware, so could not be added indefinitely.

It was clear that a new transport method was needed in order to solve these issues, and to support the next Runs of the ATLAS experiment with

increased data rates. In addition other design decisions that were imposed by the messaging system could be reformulated to improve the general outcome of the project.

4.2 New Design of Distributed Data Collection

With the detection of aforementioned shortcomings, there was a need for a redesign in the distributed data collection architecture. In addition, during the experience of the first deployment in production, a series of weaknesses that should be addressed in the following areas were collected:

- **Complexity** The current messaging segmentation approach is complex and needs grouping and transaction mechanisms that impose limits on throughput. A simpler approach is required.
- **Scalability** The messaging architecture and implementation would not scale for the increasing rates of the coming years. We need a system that can scale even with slow consumers or with backlogs of data to be ingested.
- **Performance and resource consumption** The inherent complexity supposes a performance cost in the payload segmentation in production, transport and consumption in terms of CPU and memory. The usage of a transport text protocol, although simpler, imposes limits on the encoding and compression on the payload. Ingested data could have duplicate HDFS files be removed during validation steps. Also the data ingestion at GUID file level requires extra data consolidation steps, to reduce the number of HDFS files. A resource-consumption aware and better performing approach is needed for data consolidation.
- **User Experience** Traversal time is defined as the time since a particular data is started to be indexed until it is available for users. Aforementioned issues impose limits on total traversal time that can be improved.

Messaging systems are designed to efficiently handle a large number of small messages, and our use case faces increasingly higher payloads. Although newer systems can handle large messages, there is performance degradation over time specially when slow consumers arise. These also impose immediate and in order consumption, thus, other messaging system were discarded. The grid production data movement mechanisms with Rucio and grid storage were

originally discarded due to the temporary and small size character of the EventIndex data. We therefore explored other possibilities.

4.2.1 Object Store data staging

A completely different approach was considered using an object-based storage (OBS) [74] for temporal data staging. Instead of payload segmentation in multiple frames, the complete payload is stored in an object in the OBS, and a reference to it is submitted for later data ingestion in the final data backend. Thus, we avoid the congestion problems found with the messaging systems. In addition we open the possibility of data selection before data ingestion, which is convenient to simplify later data validation and consolidation steps. Therefore the new architecture implies a change in the data collection model, moving from a push-based to a pull-based approach. In the previous model the data flows through the producer-broker-consumer chain without discrimination storing all data in HDFS, requiring later data validation and consolidations steps. With the pull model, consumers can be signaled which data retrieve, without duplicates due to production issues and avoiding extra cleaning steps. In addition there are other advantages like the consumption of the same data several times. This feature is useful for overcoming spurious consumer crashes, which was not possible with the previous messaging push model.

The new architecture is represented in figure 4.2. The distributed producer-consumer is maintained, but now the EventIndex data is directed into an object store that acts as a data staging area until it is ingested into the Hadoop data backend. This data flow is represented with black solid arrows. Producers send a small control message when the indexing procedure ends, with a reference to the created object and statistics about the process. Therefore, the messaging infrastructure and the brokers are maintained, but with a considerably less amount of messages and payload distributed. The control messages and statistics are represented in the figure by the blue dashed arrows, which are distributed to the new supervisor control entity. This entity substitutes the data validator controller of the previous architecture, checking online which data has been produced and its status. When a desired granularity is achieved, the supervisor signals this with a new control message. This message is received by one of the consumers, which will access the indicated objects in the object store. A black arrow starting from the consumers retrieving selected data from the object store represents the pull model approach.

The object store itself is implemented with CEPH [75], managed by CERN with a total of 2 PB temporary storage provided, shared with multiple projects.

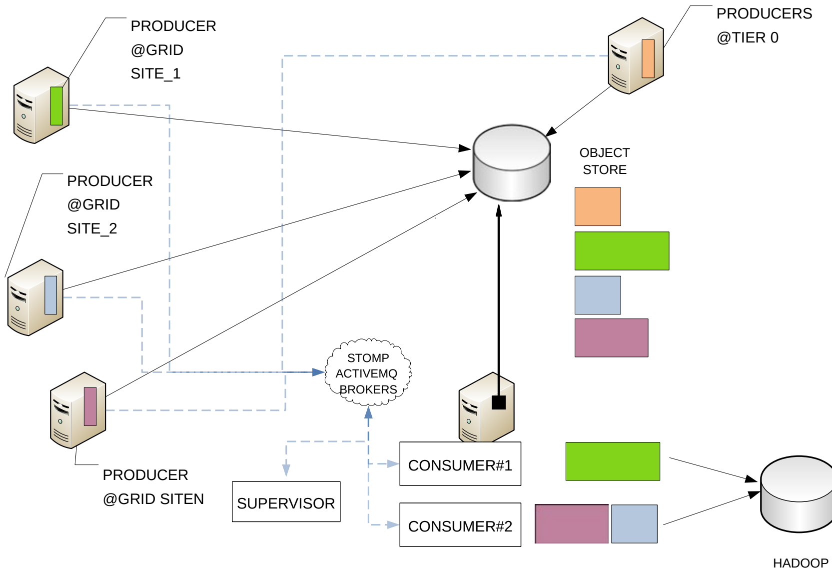
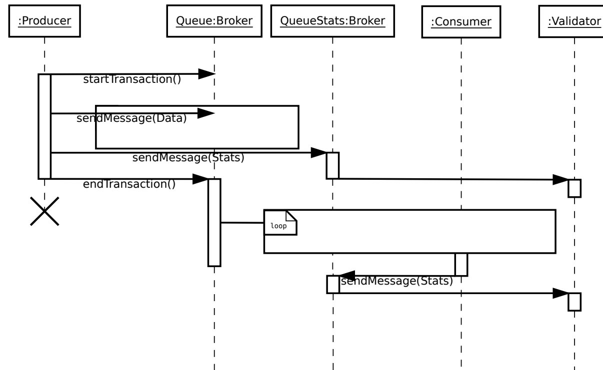


Figure 4.2: Data collection object store architecture. Black solid arrows represent the EventIndex data flow. Blue dashed arrows represent data processing information and control messages. Color-coded frames represent the payload stored by producers as objects in the object store. Consumers retrieve objects when signaled by a supervisor controller.

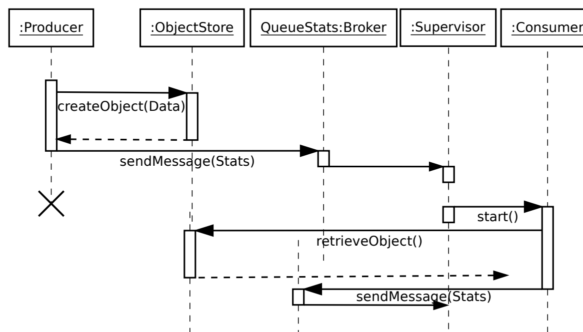
An S3 [76] interface is provided and the associated transport method allows binary payloads, so we are not attached to text only data like in the messaging approach. This allows us to select other encoding methods and we are using Google Protocol Buffers (protobuf) [77] to encode and potentially compress (gzip) our data.

4.2.2 Push versus pull model data ingestion

UML (Unified Modelling Language) [78] sequence diagrams comparing both messaging and object store scenarios can be seen in figure 4.3. In the messaging scenario (figure 4.3a) the producer initiates a messaging transaction submitting



(a) Messaging scenario.



(b) Object store scenario.

Figure 4.3: UML sequence diagram comparison for messaging and object store scenarios.

the payload divided into multiple messages into the broker data queue. When this ends it closes the transactions and sends a small control message to another statistics queue, ending the process. The consumer is connected to the broker data queue and receives all the messages as soon as the producer transaction was closed. It also signals when it ends the data ingestion procedure in Hadoop with a small control message in the statistics queue. The validator receives the statistics messages from both producers and consumers, and will trigger the data validation and consolidation procedures at regular intervals. In the object store scenario (figure 4.3b) the producer creates an object without dividing the payload into multiple chunks. When this ends it sends the control message to the statistics queue, which is received by the supervisor. This process is repeated continuously, as containers and datasets can be formed of hundreds of files indexed by multiple producers. When reaching the desired validation granularity, currently at dataset level, it constructs a new small message directed to the consumer through another broker queue. This validation message contains, among others, a validation identifier (validationId) field and the URL [79] of the validation object that contains all the references to the objects to be consumed. The consumer contacts the object store for the validation and the rest of the required objects, and again sends a small control message signaling the end to the supervisor.

Table 4.1 shows a summary of the concepts related to data and workload distribution in the messaging push-model and the new object store pull-based model. There are two key differences regarding the new object store approach. First, the payload from a given producer can be potentially written in a single object. This avoids the complex payload segmentation and reconstructions procedures, avoiding the inherent previously detected issues. Transport protocol is no longer required to be textual only, so new encoding and compression schemes can be applied. Second, the pull model approach allows more versatile data workload distribution. Now the payload can be distributed to several consumers, instead of the single consumption imposed by messaging groups and transactions. In addition, selective and out-of-order object retrieval is possible, accessing only actual validated data. This avoids the validation and consolidation steps at the data backend, creating from the beginning bigger and validated HDFS files. The lifetime of the objects is definable, clearing the data staging area at desired intervals.

Table 4.1: Summary of concepts of the messaging push model versus object store pull model.

	Messaging push-model	*Object Store pull-model*
Data		
Payload	many messages per input file	single object per input file
Segmentation	transactions group messages	not needed
Reconstruction	complex	not needed
Encoding	textual	binary
Compression	low	high
Workload distribution		
Producer to Consumer	1-to-1	1-to-many
Production	a transaction is reconstructed by a single consumer	multiple produced objects can be retrieved by different consumers
Consumption	FIFO (message queues abstraction) assure all data is consumed complete and in order	arbitrary, consumption out of order possible
Blockings	slow consumers can block others.	no blockings, more scalable
Lifetime	short (messages removed when consumed)	definable (object can be retrieved multiple times in case of failure)

4.3 Evaluation

With the implementation of the new design of the Distributed Data Collection, several tests were done prior to deploying it in production [72]. These tests indicated that the implementation was solving previous issues with the messaging system, and that the performance baseline was correct for the requirements of the project.

After the first synthetic experiments, the next objective was to check the performance in real production scenarios. We therefore carried out a rolling deployment in production, having both the previous messaging and the new object store systems working in parallel.

We can see the details of this deployment setup on the figure 4.4, where we can see both messaging and object store data paths running in parallel. It must be noted that only required parts are duplicated, with single instances where possible.

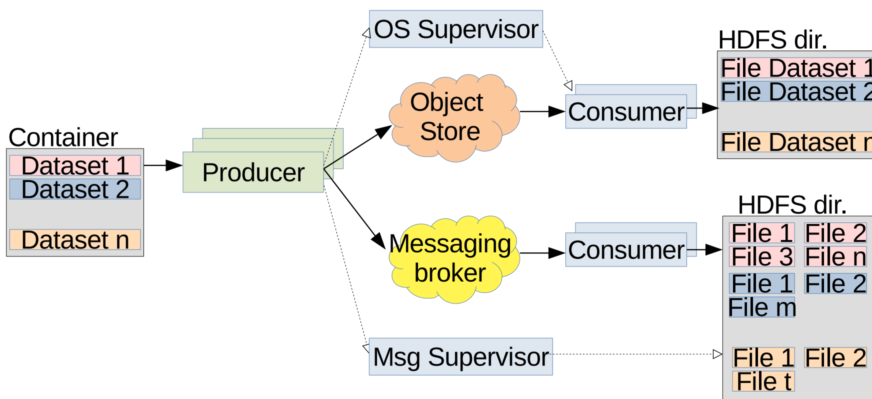


Figure 4.4: Evaluation setup with messaging and object store in parallel [80].

In the left part of the figure we see the single set of producer processes. These index the information just once, but send the index data to both paths. In this way the procedure saves a lot of CPU and I/O from the distributed grid centers as the indexing part is the most demanding one in terms of resource consumption. Then starting from the center of the figure we see both data paths, where we have duplicated instances of the elements of the architecture. In the lower part we see the legacy messaging only data path, where we use the brokers to distribute the indexed data to a set of consumers. These consumers perform the data ingestion continuously to the data backend, in this case with the legacy data format strategy in HDFS. The messaging supervisor (the original data validator controller) just signals in the data backend when the information is ready for next stage.

In the upper part of the figure we see the new object store data path, which serves as a temporary data staging entity. In this case the consumers ingest the information to the data backend only when signaled by the supervisor.

In the following subsections we will review the results of the experiment. First we analyze the results of indexing a single dataset, reviewing the concepts aforementioned with the experimental data obtained. Later we review the results of the complete experiment over the 3 months that it lasted.

4.3.1 Single dataset indexing results

Indexing a dataset is a typical EventIndex step that is done continuously as directed by the Data Production component. In this case we are analyzing the process and results of indexing a complete dataset within the previously presented evaluation scenario. A summary of the process and results can be seen in table 4.2. In this table we have 3 columns detailing the analyzed concept, and then the results in both the messaging and the object store approaches. Concepts are divided into the indexed input data, the Data production with producers processes, the data ingestion with consumer processes, and the result output in the final backend.

Input data

A real ATLAS AOD dataset produced during June 2017 was used in this example. Its name is *data17_13TeV.00327636.physics_Main.merge.AOD.f838_m1824*. It contains a bit over 21 million events, distributed in 1,160 files. The total input data comes to 6.229 TB, which will be indexed by our producer processes.

Table 4.2: Results of indexing a single dataset comparing both messaging and object store approaches.

	Messaging	*Object Store*
Input Data		
Total Files	1,160	
Total Events	21,103,653	
Total Size	6.229 TB	
Producers		
Instances (short-lived)	1,447	
Index results	1,447 (287 duplicated)	
Events indexed	262,87,826 (5,184,173 duplicated)	
Time spent	55.5 hours	
Transport method	1,515,131 messages (text)	1,447 objects (binary)
Index size	15 GB	3.3 GB (Compressed)
Consumers		
Instances (long-lived)	6	6
Files received (Duplicated)	1,447 (287)	1,160 (0)
Events received (Duplicated)	26,287,826 (5,184,173)	21,103,653 (0)
Receiving method	1,515,131 messages	1,160 objects
Produced data used	all messsages	287 objects not accessed
Data received	15 GB	2.5 GB (Compressed)
Time spent	244 min	34 min
Reading rate (kB/s)	1,098 kB/s	1,302 kB/s
Throughput (events/s)	1,793 events/s	10,344 events/s
Output data (HDFS)	1,447 HDFS files / 11.2 GB	1 HDFS file / 9 GB
Writing rate (kB/s)	627 kB/s	4,464 kB/s
Output Index Data		
Validation Method	check and remove duplicates	not needed
Traversal Time	69 hours	56.1 hours
Consolidated Events	21,103,653 events	21,103,653 events
Consolidated Data (HDFS)	1,160 HDFS Files / 9 GB	1 HDFS File / 9 GB

Data production

The data production procedure on this kind of dataset should have launched a producer per input file (1,160), but we see that a total of 1,447 instances were spawned and the indexing procedure lasted 55.5 hours. This means that some processes failed or were identified as failed at some point and were relaunched. There were produced 1,447 index results, 287 of them with probably duplicated information (5,184,173 event records extra). In the aforementioned scenario, the index data is sent with both messaging and object store systems. The index results payload was segmented into more than 1.5 million text messages, compared with the 1,447 objects without segmentation. The transient space was came to a total of 15 GB for messages, compared to 3.3 GB for the object store. The binary encoding using protobuf format allows us to compress the data much more in the object store, compared to the textual JSON encoding of the messaging approach.

Data ingestion

The pull model object store approach allows our system to avoid accessing the objects that contain duplicate data, effectively reducing the amount of data received a 25% in this example (2.5 GB received compared with 3.3 GB produced). Push model messaging consumers receive the 100% of data, including duplicates. The reading rate represents the amount of data received, divided by time spent. Similarly, the writing rate is the output data (in HDFS), divided by time spent. In both cases we obtain better numbers in the object store scenario. Each set of consumers writes in its own HDFS area; object store consumers write a unique consolidated file compared to 1,447 files by messaging consumers. Throughput in events processed per seconds is the most representative metric, which is almost 6 times higher in object store consumers.

Output results

After writing the data in HDFS, a validation method is needed in the messaging approach for cleaning the duplicated data. After the validation step, a total of 1,160 HDFS files will be ready. An extra consolidation step will be done by the core Hadoop system (after Data Collection) to create a single file. This is not the case in the object store approach, which has previously selected which data to write in HDFS. A single HDFS file is written from the beginning per dataset. The total size and collected events is the same in both approaches, as the HDFS file format is the same. The traversal time (from the beginning of

the process until all data is validated and available in HDFS) is shorter in the object store approach. We have seen that producers take the same time, but the data ingestion process with object store consumers is shorter. Also, the validation step is removed with the pull model, using less resources and with faster results with the object store approach.

4.3.2 Complete results

We analyze now the complete experiment lasting 3 months in the same described scenario, comparing both messaging and object store approaches. We will characterize and review in detail every component of the system and the results obtained.

Table 4.3 shows a summary of the results of the experiment, divided into input data, producer processes, consumer processes, and output index data.

Input data

A total of 26,367 datasets were indexed, with more than 60 billion (10^9) events distributed in more than 10 million files. Total volume sums up 17 PB of input data, with 10% stored at CERN and the remaining 90% stored at grid ATLAS sites in the WLCG.

Data production

A total of 587,967 producers were launched over the duration of the experiment at CERN and around the tens of sites that form the ATLAS part of the WLCG grid. The following figures represent the real workload that was submitted to the production system.

In figure 4.5 a histogram shows the duration of the producer event metadata extraction process. The x-axis represents the time (in Hours), and the y-axis the number of appearances (frequency) of each case on logarithmic scale. The indexing time depends on the number of input files and type of data, but also on the hardware and configuration of machine where the producer executes. The majority of the jobs finish in less than 1 hour, with a mean of 0.61 hours. There is a lot of variance in the duration time, with the longest job taking 71 hours. In total all the producers sum up 360 k hours of duration time during the 3 months of the experiment.

Figure 4.6 shows the results of the event metadata extraction phase, the EventIndex data. This is represented as the number of events indexed per

Table 4.3: Complete results of the experiment comparing messaging and object store approaches

	Messaging	*Object Store*
Input Data		
Experiment Duration	3 months	
Total Input Datasets	26,367	
Total Files	10 million	
Total Events	60 billion	
Total Size	˜17 PB	
Producers		
Instances (short-lived)	587,967	
Index results	12,311,330	
Events indexed	70,549,949,057	
Time spent	3 months (361k CPU hours)	
Transport method	994,796,792 messages (text)	587,967 objects (binary)
Index size	10 TB	2.2 TB (Compressed)
Consumers		
Instances (long-lived)	6	6
Files received (Duplicated)	12,311,330 (2,379,720)	8,663,430 (0)
Receiving method	994,796,792 messages (text)	538,442 objects (binary)
Data received	10 TB	2 TB
Time spent	16,728.6 hours	1,005.38 hours
Reading rate (kB/s)	173.6 kB/s	571 kB/s
Throughput (events/s)	1,171 events/s	14,877 events/s
Output data (HDFS)	12,311,330 HDFS files / 17.7 TB	26,367 HDFS files / 14.3 TB
Writing rate (kB/s)	282 kB/s	3,950 kB/s
Output Index Data		
Validation Method	Check and remove duplicates	Not needed
Traversal Time (typical)	>10h (94% of datasets)	<10h (85% of datasets)
Consolidated Events	60 B events	60 B events
Consolidated Data (HDFS)	9,931,610 Files / 14.3 TB	26,367 Files / 14.3 TB

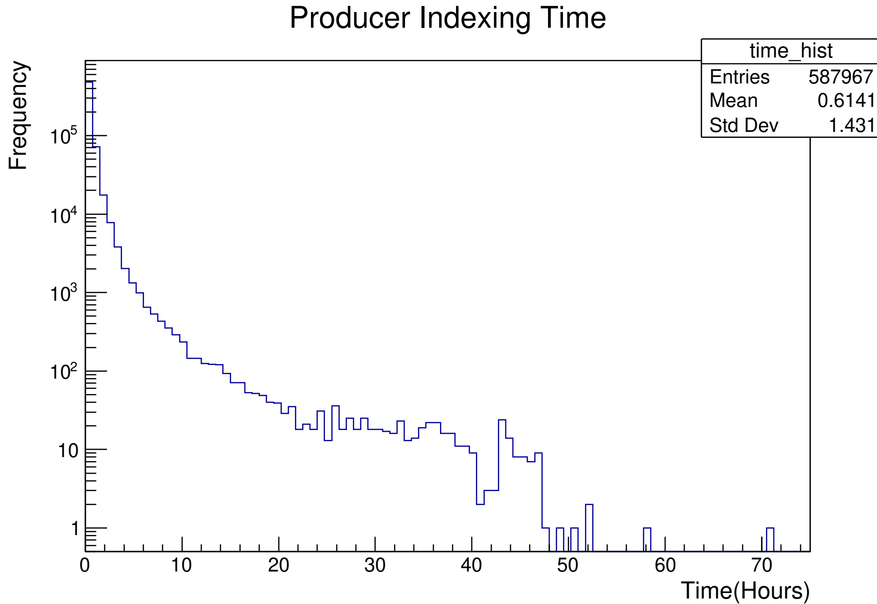


Figure 4.5: Producer Indexing Time histogram. The x-axis represents the time (hours); y-axis represents the number of appearances (frequency) [80].

producer. The x-axis represents the number of events, and the y-axis the frequency of the producers. The mean is to index around 100 k events per producer. Then we have variability with the biggest production of more than 30 million events for a single producer. The total number produced event index records is more than the original 60 billion events. This is again because some files are indexed more than once due to some job failure and restart. This will create duplicate data that needs to be detected in later phases.

In the following figures we show a characterization and comparison on the data volume produced, differentiating between messaging and object store approaches.

Figure 4.7 shows a histogram on the EventIndex data volume created per producer, comparing the two approaches. Messaging, in red, is based on JSON textual encoding. Object store, in blue, is based on binary protobuf compressed encoding. The x-axis represents the EventIndex data volume size (in megabytes), and the y-axis represents the frequency or number of occurrences, in logarithmic

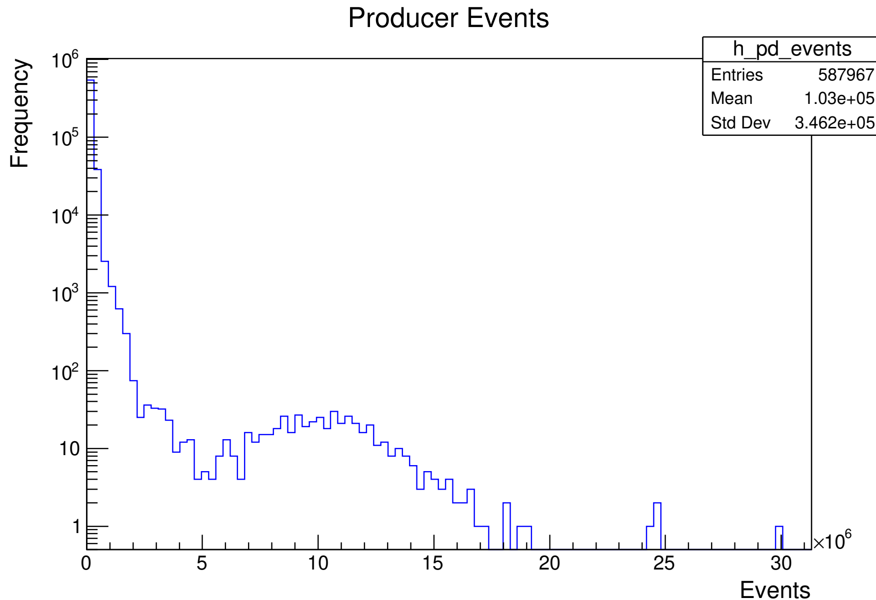


Figure 4.6: Number of events indexed per producer. The x-axis represents the number of events; The y-axis the frequency of the producers. [80].

scale. Regarding messaging approach (in red), the number of entries correspond to the 12 million (12,311,330) index results as seen in table 4.3. These index results are small, with a mean index size of 0.8 MB. 99% of the results are less than 12 MB, and the biggest index result is 64 MB. We have to remember that these results will be divided into smaller 10 kB messages when transmitted. The total integrated volume index data size is 10 TB. With the object store approach (in blue) we find less entries, as every producer stores a single object in this scenario we have 587,967 objects as index results. The mean object size is 3.6 MB. 90% of the objects are less than 8MB, but we have a great variance with objects up to 670 MB.

As a conclusion we can see that the workload is concentrated in fewer index results ($\sim$ 587 k objects against $\sim$ 12 million index files), with greater volume of data (3.6 MB mean object size, against 0.8 MB mean index data file with the messaging approach). This is due to the fact that with objects we group the information for several input files, with a binary encoding that in addition

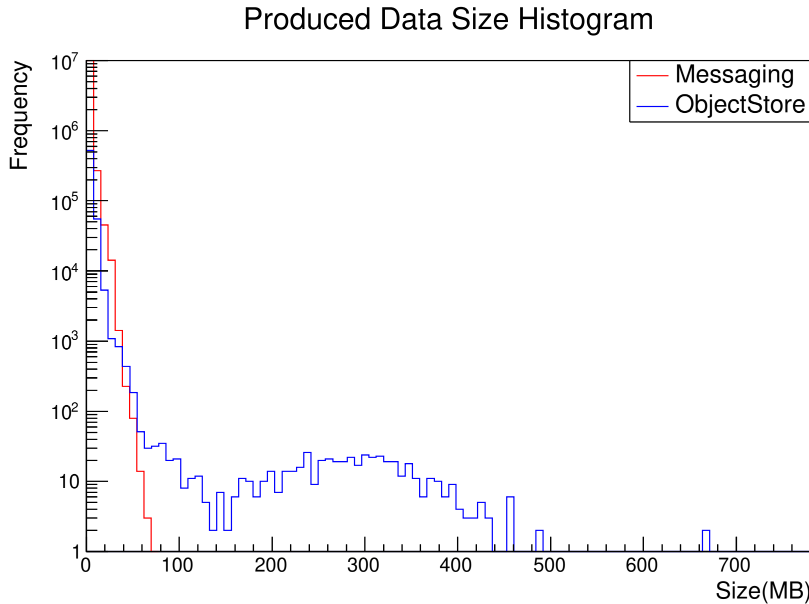


Figure 4.7: Comparison of produced index data (messaging vs. object store). The x-axis represents the data volume size; the y-axis represents the frequency. [80].

allows better compression.

In the following figures we can see the produced index data produced and sent with both methods, depending on time over all the duration of the experiment.

Figure 4.8 shows the volume of data produced per hour, for messaging (red) and object store (blue) approaches. The x-axis represents the number of elapsed hours since the beginning of the experiment until 2,270 hours corresponding to 3 months of the experiment. The y-axis represents the data volume produced in megabytes. The peak in messaging (in red) is produced at hour 1,880, with 60 GB of index data (sent with approximately 6 million messages). The object store (in blue) data production peak is about 14.25 GB (in 3,538 objects).

In the following figure 4.9 we observe the produced index data sent accumulated until a particular moment since the beginning of the experiment. The total volume of the produced and transmitted data with messaging (in red) is 10 TB at the end of the experiment, which corresponds to 102 GB per day.

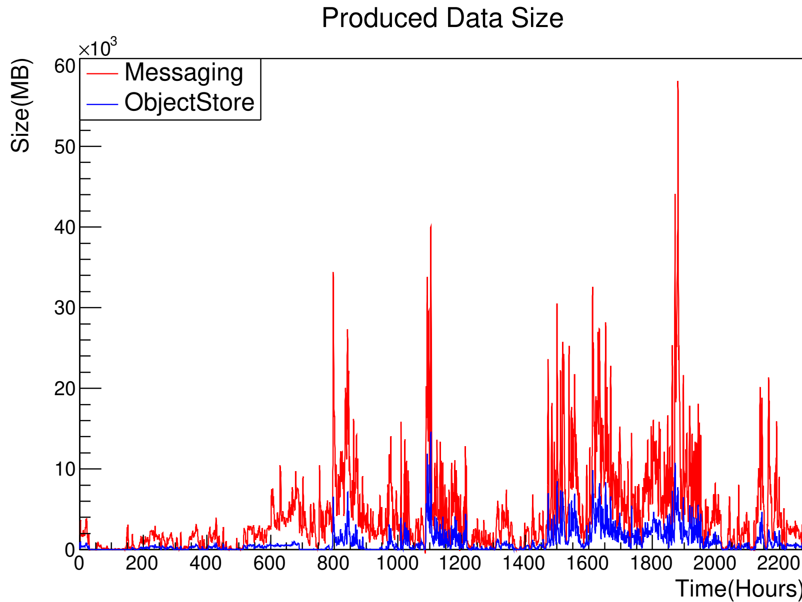


Figure 4.8: Produced index data along the duration of the experiment. Red graph corresponds to messaging; blue graph corresponds to object store. The x-axis represents time since the beginning of the experiment; The y-axis represents the data volume of index data. [80].

The same variable for the object store approach (in blue) adds up to 2.2 TB of index data, corresponding to 22.2 GB per day. Comparing both approaches, we achieve transmitting 4.5 times less data with the object store approach.

Data ingestion

Differences in consuming and data ingestion procedures are represented in the following representations, where we show produced against consumed data depending on the approach.

In the figure 4.10 we show the produced (red line) versus consumed (green line) messages, over the duration of the experiment. The total number of consumed messages reaches almost the billion messages (994,796,792), which corresponds to the volume of data of 10 TB aforementioned. It has to be noted

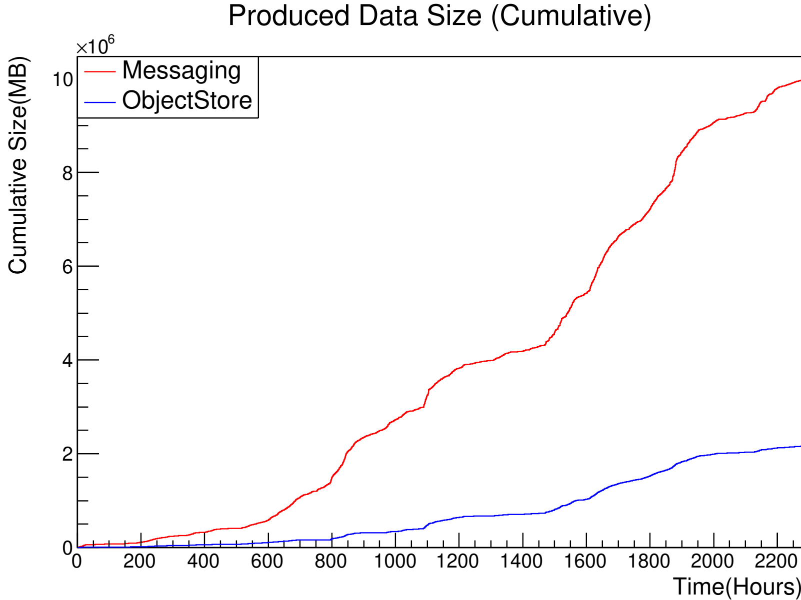


Figure 4.9: Cumulative produced data for the duration of the experiment. Red graph corresponds to messaging; blue graph corresponds to object store. The x-axis represents time from the beginning of the experiment; The y-axis represents cumulative data volume of index data [80].

that both lines superpose during most of the time of the experiment, showing the real-time consumption of the produced data. There are however time periods where the corresponding red line is visible, indicating an issue in the consumption rates. This is the case for hour $x \in [1087, 1162]$, $[1189, 1254]$, $[1611, 1748]$ and $[1879, 1912]$. In this case the brokers create a backlog of messages that are not consumed in line with production. Eventually the backlog is reduced and messages are consumed, with both green and red lines of the graph converging. In these situations slower consumption is detected, due to head-of-line blocking issues, potentially worsening the situation. Other situations that can cause backlogs are disconnection of consumers due to maintenance, or problems in the backend HDFS system. Should messages not be consumed at production rate over larger periods, the backlog would increase, exhausting the brokers memory, and rejecting new produced messages. Another issue is that a problem

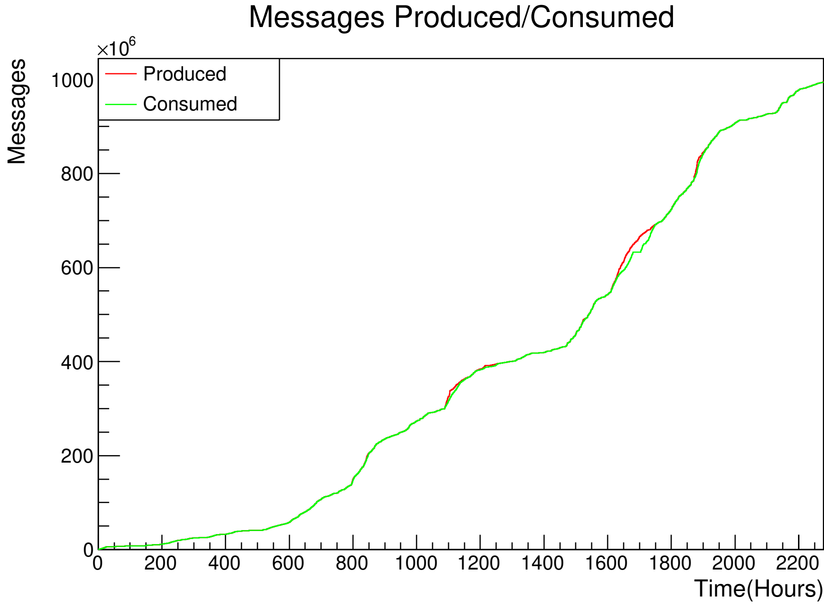


Figure 4.10: Produced versus consumed messages for the duration of the experiment. Red line represents produced messages; green line represents consumed messages [80].

handling one single message of a message group, invalidates all the group. This means indexing again the original data, with the corresponding submission of new data production jobs.

Figure 4.11 shows the number of produced versus consumed objects. Produced and stored data in the object store (in red) reach a total of 587,967 objects. In contrast to the messaging approach, objects are not inserted into a queue and are not needed to be consumed in order. In addition, as the produced object might contain duplicate information, not all of them need to be accessed. There are validation steps that select the correct objects. This is shown in green, representing the 538,442 accessed objects. Thus, 49,525 of the originally produced objects are not accessed.

The consumed objects sum up a total of 2 TB index data. There are 200 GB from the original volume produced (2.2 TB) that are not accessed at all, representing a 10% of the total.

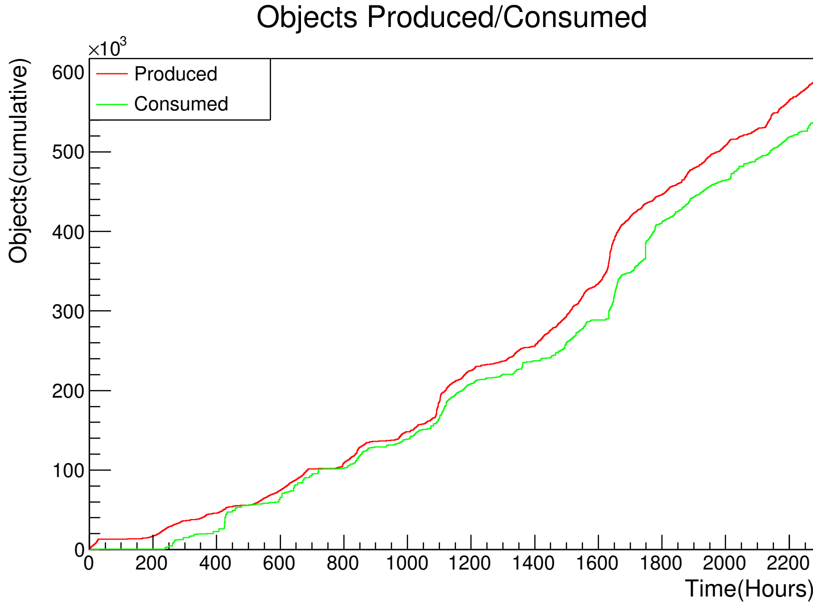


Figure 4.11: Produced versus consumed objects for the duration of the experiment. Red line represents produced objects; green line represents consumed objects [80].

Compared with the 10 TB of the messaging approach, with the new object store approach we consume overall 5 times less data. These improvements result in less resources used, and faster data ingestion rates.

Finally, an overall picture of the data ingestion process is shown in figure 4.12. The consumers throughput is represented in terms of processed events per second, comparing the messaging approach (in red) with the object store approach (in blue). The histogram x-axis shows the rate (events/s), and the y-axis shows the relative frequency of appearances in the consumption. The messaging consumers yield a mean rate of 1 k events processed per second, with a maximum of 5 k events per second. The object store consumers yield a mean rate of 15 k events processed per second, with a maximum of 28 k events per second. Therefore, we observe a factor 15 improvement in the object store performance, compared with the messaging consumers. With head-of-line blockings detected in the messaging brokers, we could not effectively scale the system. On the other hand,

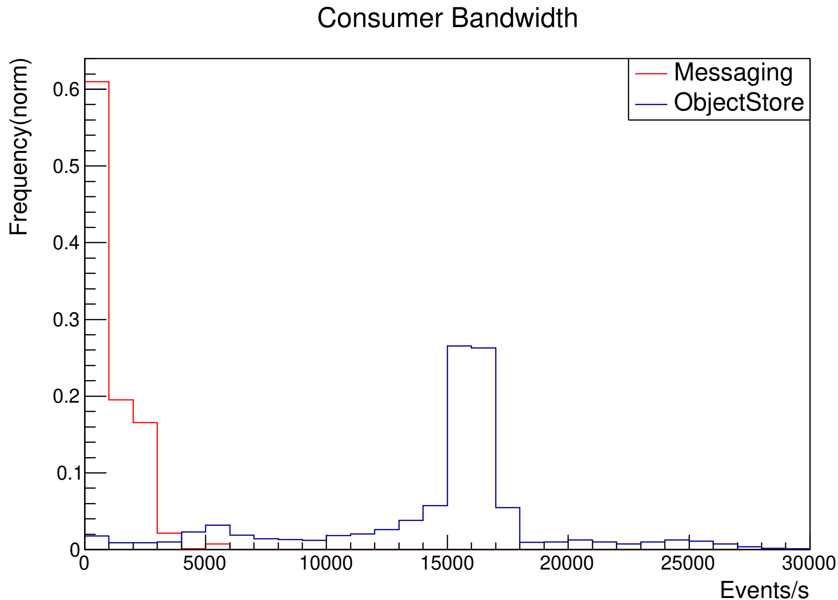


Figure 4.12: Comparison of consumer throughput in events processed per second. Messaging consumers are shown in red, and object store consumers are shown in blue. The x-axis represents the events processed per second; y-axis represents the frequency normalized [80].

with the new approach we don't observe such limitations and we can scale up adding new object store consumers.

Another set of benefits are achieved with the pull model of the new object store approach. Only validated and unique data is consumed and ingested into HDFS, and therefore can be done in a single step, writing a unique file per dataset. Therefore later validation and consolidation steps are avoided, compared with the messaging approach.

Objects are not removed when accessed. There is a defined data staging lifetime policy, which directs when to get rid of non needed objects, usually after some data production campaign.

Thus, objects can be accessed again in case of a problem with consumers or with the backend filesystem. This is an improvement with respect to the previous messaging approach.

Traversal time

A valuable metric for users on the overall performance on the system is traversal time. We have defined traversal time for a particular dataset, as the period of time starting with the indexing of the first input data file, until the last EventIndex data is consumed and is available in the final backend.

We show in figure 4.13 a histogram of the traversal time for the analyzed datasets, comparing both messaging (in red) and object store (in blue) approaches. The x-axis represents the traversal time in hours, and the y-axis represents the relative frequency on the number of dataset occurrences (normalized to 1, in logarithmic scale).

Traversal time for datasets ingested with the object store path are mostly (85% of them) available in less than 10 hours. The mean is $\bar{x} = 12$ hours, with $\mathrm{Var}[x] = 47$.

In contrast, traversal time for the same datasets ingested with the messaging path take longer. Only 6% of them are available in less than 10 hours. We observe much more variance with $\mathrm{Var}[x] = 247$.

Few datasets take much longer to be indexed and validated. In some cases part of the input data is not available online (because the grid storage has problems, or because it is on long-term storage tape systems). In other cases some input has to be re-indexed again because of problems in the collection. We observe that with messaging a larger number of datasets takes longer to be available, increasing the total traversal time for these datasets by over 1,000 hours. Overall, we observe lower time traversal values with the object store approach, meaning they are sooner available for final users.

4.4 Conclusions

The work described in this thesis has resulted in several contributions on real production grid distributed systems collecting big data.

We contributed to the design and part of the implementation of a big data distributed collection system for our application. We characterized the shortcomings of messaging systems to convey our production data. The segmentation of data payload and reassembly is complex and produces head-of-line (HoL) blockings at the messaging brokers, limiting scalability and imposing limits on the data ingestion rates.

We contributed to a new pull model data ingestion design for the distributed data collection of grid produced metadata of the EventIndex project. Its

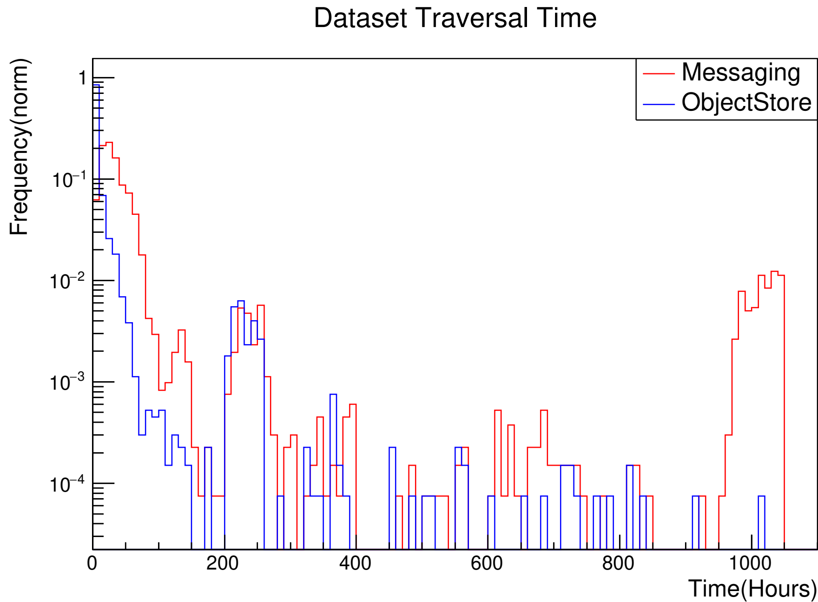


Figure 4.13: Comparison of typical traversal times for complete datasets for messaging (in red) and object store (in blue) approaches. The x-axis represents the traversal time in hours, and the y-axis represents the relative frequency (normalized to 1, in logarithmic scale) [80].

implementation is supported by the usage of an object store for temporary data staging, and a dynamic data selection mechanism.

In order to experimentally prove the validity of the approach, we set up a evaluation deployment in real scenarios with both the previous messaging push model, and the new object store pull model in parallel. We ran an experiment over 3 months in a real distributed environment, indexing more than 60 billion events, distributed in 10 million files worldwide, with a total volume of 17 PB of data.

Results show that the new design improves the performance in several areas. With the new design we can avoid payload segmentation, storing the index information from a producer in a single object. Better data compression ratios are achieved with larger payloads and binary data encoding. Thus a factor 4.5 reduction in the total volume of conveyed and ingested data is achieved.

Blockings are not observed with the new object store implementation, and the workload distribution is improved, potentially achieving better scalability. Throughput during data ingestion is improved 15 times compared with the messaging approach. In addition, the pull model approach allows the dynamic selection of data, avoiding the ingestion of duplicated information that in our experiment is a 10% of the produced data.

Data validation and consolidation steps have been improved, reducing the number of backend written data to a single HDFS file per dataset. This supposes a factor 300 reduction in the number of files written during data ingestion.

Reduction in the complexity of the system, and resource consumption during transport, data ingestion and validation phases have improved the reliability and overall performance of the system. These changes have shortened the traversal times of the EventIndex data, eventually improving the user experience.

After successful validation of the new object store implementation, a rolling deployment was carried out during Run 2 and has been running in production ever since.

5 Storage

Data storage is one of the most important components because it maintains all the EventIndex data and has to scale for the current and future Runs of the experiment. The first design for the project implemented in 2014 was based on the available Hadoop Big Data technologies at that time. It was conceived to use HDFS as the main storage with MapFiles as the file format used. HBase was also used, but only to maintain a catalog of the data. Limitations were found in the data ingestion and data access performance. Random access required for event-picking use case was not well suited to this model, and therefore not performant enough.

A first advance was made with a hybrid system indexing the most relevant data (only real data, without trigger information) in Oracle [54], and having the main backend with all data in HDFS. In addition, included a general lookup method by event number and run number in a small HBase database, for faster access to a GUID (event picking use case) and including pointers to the complete records stored on HDFS. The implementation was run successfully in production during Run 2 (2015-2018) [48] as shown in figure 5.1, at the cost of duplicated data, with an increase in volume size used. Overall system architecture complexity supposed higher maintenance costs and data coherence concerns.

New use cases have appeared since the beginning of the project. They have been extended from event picking and production completeness checks to trigger overlap studies, duplicate event detection and derivation streams overlaps. The event rate steadily increased in Run 2, and now in Run 3 it is expected to be a factor 3 of the previous Run. For Run 4, a factor 10 increase in the event rate is expected. New Big Data technologies suitable to support bigger ingestion rates, and the required transactional and analytical workloads of our project, have appeared. Previous studies [55, 81] concluded that columnar storages like Kudu could be suitable for our application. Also HBase was found to be an

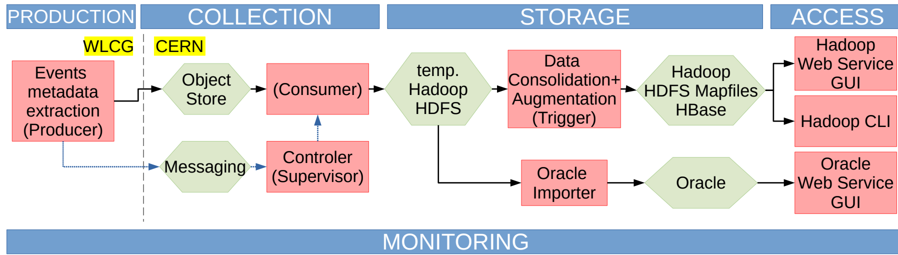


Figure 5.1: EventIndex architecture as implemented at the end of the LHC Run 2. Green hexagons correspond to temporary or permanent data, and pink rectangles represents processes. Black arrows show the EventIndex data flow, and blue arrows show the flow of information related to data processing. Data collection uses a pull-model based on an object store. Data storage uses a temporary HDFS store, from where data is consolidated and augmented into the final storage using HDFS MapFiles and HBase for the catalog. A subset of the information is copied to Oracle tables for fast data retrieval.

option to host now all the EventIndex data and not only pointers, due to the improvements in the platform.

In this chapter we review the storage technologies suitable for our project that have been or are suitable for use in the following Runs. First, we review the HDFS storage and how it has been used in the project focusing on the data ingestion parts. Next, we present Kudu's main advantages, along with the data model and prototype that was setup to evaluate it. Finally, we show HBase as candidate to host all the data with an interface called Phoenix to provide SQL capabilities for data ingestion and access. An evaluation is carried out, finishing with some conclusions.

5.1 HDFS

The Hadoop Distributed File System (HDFS) [51] is designed for storing very large files (hundreds of megabytes up to terabytes in size) and streaming data access patterns (write-once, read many times; and accessing a large proportion or all the dataset) into commodity hardware clusters. It is not designed for low-latency data access, storing lots of small files, or having multiple writers or arbitrary data modifications. HDFS files are organized into big blocks replicated in 3 data nodes by default. With this feature it achieves fault tolerance on

hardware failures, and increases data availability. It is specially well suited to data processing with the MapReduce [59] framework. A MapReduce job splits the input into chunks (usually at the block level), which are processed by the map tasks in a completely parallel manner. The framework orders the outputs of the map tasks and distribute them to the reduce tasks. Tasks are usually executed where data is stored, yielding a broad aggregate bandwidth.

5.1.1 Data organization

The EventIndex data collection consumers are in charge of data ingestion procedures into the data backend. Upon receiving a validation message, the consumer creates the required HDFS directory and HDFS files at the defined granularity.

EventIndex data is organized into HDFS directories named after the dataset container name, and will include one HDFS file per dataset. The HDFS file name is composed of the dataset name (see subsection 3.5.1) and the validation identifier separated by a dot (*datasetname.validationId*).

A representation of an HDFS directory of a dataset container can be seen next:

```
datasetContainer/datasetname_tid1.validationId_a
                /datasetname_tid2.validationId_b
                /datasetname_tidN.validationId_c
```

The datasetContainer corresponds to the datasetname without the *_tid* suffix.

A dataset can represent a container itself (datasetContainer = datasetname) if it is composed of a unique dataset (no tid datasets). This is the case for Tier-0 datasets. As an example for a 2022 Tier-0 dataset, an HDFS directory is created with a single HDFS file:

```
data22_13p6TeV.00429026.physics_Main.merge.AOD.f1249_m2112/
↪  data22_13p6TeV.00429026.physics_Main.merge.AOD.f1249_m2112.
↪  9f7087f2ec9d46f986da4f69d9ecc1f5
```

In general a dataset container is composed of several (tid) datasets, so an HDFS file will be created per each of them. In the following example a verbatim listing on an HDFS directory of a 2020 Monte-Carlo dataset container composed of 3 tid datasets is shown:

```
-rw-r--r--   3 atlevind atlas-db-eventindex-access    243.0 M 2022-06-28
↪  15:31 mc20_13TeV.700335.Sh_2211_Znunu_pTV2_BFilter.merge.AOD.⌋
↪  e8351_s3681_r13144_r13146/mc20_13TeV.700335.⌋
↪  Sh_2211_Znunu_pTV2_BFilter.merge.AOD.⌋
↪  e8351_s3681_r13144_r13146_tid27230884_00.⌋
↪  1b35faccaaed4039862631caece3efd6
-rw-r--r--   3 atlevind atlas-db-eventindex-access    361.3 M 2022-06-28
↪  14:36 mc20_13TeV.700335.Sh_2211_Znunu_pTV2_BFilter.merge.AOD.⌋
↪  e8351_s3681_r13144_r13146/mc20_13TeV.700335.⌋
↪  Sh_2211_Znunu_pTV2_BFilter.merge.AOD.⌋
↪  e8351_s3681_r13144_r13146_tid27230909_00.⌋
↪  86514c7bb05742828bfc4794e1de3ebc
-rw-r--r--   3 atlevind atlas-db-eventindex-access      3.1 G 2022-06-29
↪  01:10 mc20_13TeV.700335.Sh_2211_Znunu_pTV2_BFilter.merge.AOD.⌋
↪  e8351_s3681_r13144_r13146/mc20_13TeV.700335.⌋
↪  Sh_2211_Znunu_pTV2_BFilter.merge.AOD.⌋
↪  e8351_s3681_r13144_r13146_tid27230936_00.⌋
↪  0511bd49377c4572b6ed955fec557c2c
```

The listing is shown with the following format: *permissions number_of_replicas userid groupid filesize modification_date modification_time filename*. Several things have to be noted:

- number_of_replicas is equal to 3, the usual HDFS replication factor to increase reliability.
- filesize varies from 243 MB, 361.3 MB and 3.1 GB.
- All files share the parent directory (dataset container name), and the file name (datasetname) until the *_tid* suffix. Every file has different tid number and validationId:

  ```
  ..._tid27230884_00.1b35faccaaed4039862631caece3efd6
  ..._tid27230909_00.86514c7bb05742828bfc4794e1de3ebc
  ..._tid27230936_00.0511bd49377c4572b6ed955fec557c2c
  ```

  Having the validationId included in the HDFS filename is done for tracking purposes related to the data collection task. This also adds versatility to change the granularity of HDFS file writing. We could potentially have several HDFS files per (tid) dataset, sharing the datasetname, and with several validationId suffix.

5.1.2 File format and contents

To store the EventIndex information in HDFS, we use SequenceFiles and MapFiles per different part in the system. SequenceFiles are produced during data ingestion. They are binary flatfiles that require a key and a value per record, but do not require key ordering. MapFiles are ordered binary files, that are in fact composed of two ordered SequenceFiles, one for the data and one for an index on the keys. This index allows faster retrieval. They are produced after data consolidation procedures to be used by final users.

Data ingestion

Every record in the files represents an event record in EventIndex. The *key* is a zero-padded string composed of "RunNumber-EventNumber", with 8 digits for the RunNumber, and 11 digits for the EventNumber (ie. "00429026-00424949126"). With this key approach we can index the records in a lexicographical order. It has to be noted that although the records for a particular file are written by a single writer, a complete order is not guaranteed, as was not the case in the transient input EventIndex protobuf encoded objects (sec subsection 4.2.1).

The *value* is a string composed of the comma separated values (CSV) of the fields of an event record. It contains 40 fields for event identification, the trigger information, and location information.

A visualization of a record is shown here:

```
00429026-00424949126    293,5,1658545466,418836785,0.0,0,872532⌟
↪  710,0,0,0,Bn!!!!!!!B!!!!!!!!!!!!!!!!!!!!!!!!!!K!!!D!!O!!!!!⌟
↪  !!!!!!!F!B!C!E!D!!!!!!!!!B!!!!!!!!!H!!!!!!!C!D!D!!!b!!!!!!!⌟
↪  !!!!!J!!!!B!!!D!!!E!!!!!!BT!G!F!!!B!H!F!B!!u!DQ!G!B!C!E!BP!⌟
↪  F!GY!G!B!C!E!BP!F!Dl,;;,UI!!Bd!CD!M!B!H!Gd;;,3096,1628,1647⌟
↪  ,StreamAOD,DDFC6B19-66ED-E442-AA50-82E57B2CB79F,POOLContain⌟
↪  er(DataHeader),4DDBD295-EFCE-472A-9EC8-15CD35A9EB8D,0000020⌟
↪  3,00000270-00000000,StreamRAW,20008429-340A-ED11-8810-3CECE⌟
↪  F0D9A38,,00000000-0000-0000-0000-000000000000,00001000,0000⌟
↪  0000-00000120,,,,,,,,,,,,
```

The list of record fields in order of appearance is shown below. We have divided them into blocks regarding their nature: event identification, trigger information, and location information. Internal name of the field is followed by the original Protocol Buffers Type (C++ like), and a description of the

field. When writing the HDFS file record the types are converted to the string representation.

The key of the record is composed by the following fields:

- **RunNumber** (uint32) Event run number.
- **EventNumber** (uint64) Event number.

Value of the records follows with information on the event identification, composed of the following fields (1-10):

- **LumiBlock** (uint32) Luminosity block number identifier.
- **BunchId** (uint32) Bunch crossing identifier.
- **TimeStamp** (uint32) Event POSIX time in seconds since the epoch (1970-01-01 UTC).
- **TimeStampNSOffset** (uint32) Nanoseconds offset with respect to the event time (*TimeStamp*).
- **McEventWeight** (uint32) MonteCarlo generator event weight.
- **McChannelNumber** (uint32) MonteCarlo generator channel number.
- **ExtendedLevel1ID** (uint32) Extended Level-1 trigger identifier [21].
- **IsSimulation** (bool) true: simulation, false: data.
- **IsCalibration** (bool) true: calibration, false: full detector.
- **IsTestBeam** (bool) true: testbeam, false: physics.

The trigger related information is in the following records (11-16):

- **L1PassedTrigMask** (string) Level 1 (L1) trigger mask in compressed text format.
- **L2PassedTrigMask** (string) Level 2 (L2) trigger mask in compressed text format.
- **EFPassedTrigMask** (string) EventFilter (EF) trigger mask in compressed text format.
- **SMK** (uint32) Trigger supermaster key.

- **HLTPSK** (uint32) High Level Trigger (HLT) prescale set identifier.
- **L1PSK** (uint32) Level 1 Trigger (L1) prescale set identifier.

The trigger compressed text format has been used by the Data Collection task since the first EventIndex release [47, 82]. Originally it was driven by the text only payload requirement on the messaging platform (section 4.1), but has been maintained on the new platform (section 4.2). The format is using a run length encoding (RLE) approach where the trigger mask bits equal to 1 are substituted by '!' character, and consecutive bits equal to '0' are substituted by its repetition count encoded in the corresponding character from the Base64 alphabet [83].

The next set of fields relate to the provenance (data lineage) information of the event. A set of these fields represents a *Token* in the Athena persistency POOL/ROOT framework [43], which allows navigation capabilities to access particular event data within a file.

In the EventIndex record we store 4 sets of these Tokens. The first Token (fields 17-22) represents the current event as found in the indexed file. The following ones (fields 23-28, 29-34, and 35-40) represent the lineage of the data in files from earlier upstream processing stages. The fields of a Token set are the following:

- **Name** (string) Token container Name (Stream data type)
- **DB** (string) Database identifier where to find the event. The GUID string representation is stored for files.
- **CNT** (string) Container identifier.
- **CLID** (string) Class identifier UUID.
- **TECH** (string) Token technology identifier.
- **OID** (string) Object identifier. Represented as string "OID1-OID2".

The field naming is following the persistency framework naming. Token *Name* field represents in EventIndex the stream data type (StreamAOD, StreamRAW...); *DB* contains the GUID as file identifiers where we can find the full event data; *CNT* is a container identifier and *CLID* is a UUID that can refer to a internal C++ object useful in the persistency framework. It must be noted that RAW data files are written in ByteStream format [21] and not accessible by POOL, thus CNT and CLID are not meaningful. *TECH* is the

token technology identifier, and *OID* is composed of two offsets and represented as the "OID1-OID2" string. OID1 is an offset in a file container list, and OID2 is an offset of the event inside the file container.

Data augmentation

After data ingestion, data augmentation (like trigger decoding) and consolidation procedures are done by the Hadoop core tasks [48]. The internal CSV format is maintained, effectively appending 14 more fields. The first fields include the dataset naming fields (already available in the file name) decomposed (fields 41-45) including project, streamName, prodStep, dataType, and version.

Next fields include the decoded trigger chains decomposed in their components:

- L1 trigger (fields 46-48) : L1trigChainsTAV, L1trigChainsTAP, L1trigChainsTBP.
- L2 trigger (fields 49-51) : L2trigChainsPH, L2trigChainsPT, L2trigChainsRS.
- EF/HLT trigger (fields 52-54) : EFtrigChainsPH, EFtrigChainsPT, EFtrigChainsRS.

These are long string fields with all the trigger names decoded, which makes the file contents much larger.

The file records are lexicographical ordered by the key (RunNumber - EventNumber), and this process makes the final Map Files available for users in the production area.

5.1.3 Limitations

The procedure requires several stage areas that contain temporary replicated data in HDFS (ingestion and production areas). Several steps have to be done until data is consolidated and finally available, complicating the procedure and increasing the points of failure. The data ingestion and consolidation into MapFiles is complex and requires sorting the contents by key when writing them in HDFS. The average ingestion speed into MapFile format is 6.4 kHz per dataset [55]. Thus overall performance is reduced, in the end increasing traversal time. Therefore, the implementation of a new technology was needed to overcome these limitations. The following sections describe possible technologies and the implementation of the chosen solution.

5.2 Kudu

Apache Kudu [56] is a distributed table-based storage designed for Hybrid Transactional and Analytical Processing (HTAP) systems. Instead of providing a file format on top of HDFS, it provides its own technology for column-based storage and indexing of the data. A columnar storage groups logically and physically similar data together, allowing efficient encoding and compression. A relational data model is followed in the design with tables with a defined schema based on named columns, types and a primary index.

Kudu partitions tables based on hashing, range partitioning, or a combination of these. Partitions are replicated allowing fault tolerance and availability. Kudu employs the Raft [84] consensus algorithm to replicate all operations for a given partition.

Other features and limitations define the usage:

- Each table has only one (clustered) index and it is built on a primary key.
- The unit of table scan parallelism is a table partition.
- There is no automatic splitting on the key range. A partition range has to be known during creation and cannot be modified.

Benefits come also in the data access capabilities, as multiple frameworks for parallel data processing and computation like Apache Spark [60] and Apache Impala [85] can be used.

5.2.1 Data organization

We contributed to the development of a new data model based on Kudu tables and resembling a relational schema [86]. The idea is to have the events stored in big tables (one for real and another for simulated Monte-Carlo data), with typed columns representing the fields in the EventIndex data model.

The EventIndex events schema that was used for validation of the model and used for the ingestion tests in the next subsection 5.2.2 is represented in table 5.1 with an entry per column. For every column we specify a *column name*, a *column type*, *encoding* based on the type of the column, and if it is part of the *Primary Key (PK)*. In the last column of the table we also include *comments* on the meaning of the entry.

The key of the Kudu events table is composed of the first eight entries (signaled with PK). The fields that represent the dataset are *project, streamname,*

Table 5.1: Kudu *Events* table schema

Column name	Column type	Encoding	Primary Key (PK) / Comment
epoch	int32	rle	(PK) partition
project	string	dict	(PK) dataset project
streamname	string	dict	(PK) dataset data stream
prodstep	string	dict	(PK) dataset production step
datatype	string	dict	(PK) dataset format
version	string	dict	(PK) dataset version (AMITag)
runnumber	int32	rle	(PK) dataset run number
eventnumber	int64	bs	(PK) event number
lumiblockn	int32	rle	lumiblock number
bunchid	int32	bs	bunch crossing identifier
eventtime	int32	bs	event timestamp
eventtimens	int32	bs	nanoseconds offset
lvl1id	int32	bs	L1 trigger identifier
hltpsk	int32	rle	HLT trigger prescaler key
l1psk	int32	rle	L1 trigger prescaler key
l1trigmask	string	plain	L1 trigger mask (json)
l1trigmask_bytes	binary	plain	L1 trigger mask (binary)
eftrigmask	string	plain	EF/HLT trigger mask (json)
eftrigmask_bytes	binary	plain	EF/HLT trigger mask (binary)
tbp0-7	int64	bs	L1 TBP component (binary)
tap0-7	int64	bs	L1 TAP component (binary)
tav0-7	int64	bs	L1 TAV component (binary)
hltph0-63	int64	bs	HLT PH component (binary)
htlrs0-63	int64	bs	HLT RS component (binary)
hltpt0-63	int64	bs	HLT PT component (binary)
db	string	dict	GUID of file
oid1	int32	rle	object identifier offset 1
oid2	int32	bs	object identifier offset 2
provenance	string	plain	provenance (json)

prodstep, datatype, version, runnumber as defined in subsection 3.5.1. A refinement to this approach was done within the project by grouping these fields in a generated *dspid* or dataset primary id as presented in [86] as there was no need to search for individual rows based on this, and dspid results in a more compact key. These columns were moved to another *datasets* table, referring to the generated *dspid* and include other cataloguing and bookkeeping fields. Eventnumber type of bigint (int64) allows referring to 2^{64} events for any dataset. Column *epoch* (*rgid* in [86]) is included for range partitioning and is assigned applying a function over a column which can be *runnumber* or *dspid*.

Regarding *encoding*, run-lengh encoding (rle) is effective for compressing consecutive repeated values, because it stores only the value and the count. The dictionary (dict) encoding is effective for columns with low cardinality, as a dictionary of unique values is constructed, and each column value is stored as its corresponding index in the dictionary. Bitshuffle (bs) encoding [87] rearranges values by blocks storing the most significant bit of every value, then the second most sisginificant bit, and so on. The block is compressed with the LZ4 compression algorithm. This encoding shows the best compression ratios for numeric values that change very little when sorted by primary key. The last plain encoding is stored in the natural format, for example UTF-8 format for strings.

Then other fields are included for event identification, like the lumiblock number (*lumiblockn*), the bunch identifier (*bunchid*), and the timestamp of the event (*eventtime*) including the offset in nanoseconds resolution (*eventtimens*).

Trigger related values are included in the following fields: the extended Level-1 trigger identifier (*lvl1id*), the High Level Trigger prescaler (*hltpsk*), and the Level 1 trigger prescaler (*l1psk*).

Regarding trigger masks, several options were tested in the prototype: A unique binary column, multiple binary columns encoded as int64 values (allowing us to push down query predicates with Impala), and JSON encoding.

Complete trigger masks are reflected in the columns *l1trigmask, eftrigmask* encoded as JSON strings, corresponding to Level 1, and Event Filter/ High Level Trigger. As level 2 trigger only exists in Run 1, the related fields allowed us to test previous data, but will not be included in the model. In Run 2, level 2 and EF were merged into the High Level Trigger (HLT) and contained the same information.

JSON encoding allows versatile operations and can be directly used by higher level tools accessing this information. On the other hand, all the trigger information has to be accessed, and there are use cases when we need only

access some triggers. For this reason, the same information is exploded in several binary columns, which allows the selection on individual triggers.

Level 1 trigger is divided into trigger before prescaler (TBP), trigger after prescaler (TAP), and trigger after veto (TAV). Each component is composed of up to 512 bits so this is reflected in the schema with 8 columnns of int64 type (8 x 64 = 512 bits), which can be individually selected, making up a total of 1,536 bits. In this way, this can be directly mapped to Impala bitwise operations (limited to 64 bit operands).

Event filter (EF) or High Level Trigger (HLT) are internally divided into three components: physics (PH), passthrough (PT), and resurrected (RS). It is represented with 192 columns of int64 type, so 192 x 64 = 12,288 bits in total. Therefore 64 columns are used for physics (PH) (*lhltph0-63*), 64 columns for passthrough (PT) (*hltpt0-63*), and 64 columns for resurrected (RS) (*htlrs0-63*).

Decoded augmented trigger chains (*l1trigchainstav, l1trigchainstap, l1trigchainstbp, eftrigmask, eftrigchainsph, eftrigchainspt, eftrigchainsrs*) as in the original augmented HDFS schema (section 5.1) are not included, due to their big size and that this can be decoded from the already included fields.

Event location information is available in columns named with *db* prefix, which represents the GUID of the file that stores these particular events, and columns *oid1, oid2* which are the object identifiers inside that file. The *provenance* is also a JSON encoded field that represents the origin of this events from previous processing versions.

5.2.2 Data ingestion

We developed a new plugin inside the Data Collection framework for the ingestion into the Kudu storage backend [88]. In this way we can benefit from the available methods of accessing the input data and report the results in the framework, completing the ingestion workflow.

The *KuduWriter* is in charge of transforming the data into the new schema, and uses the available Kudu client API to store the data.

For our tests a local IFIC cluster was used with the specifications defined in the table 5.2.

The ingestion tests consisted of a typical data collection scenario, with input data stored in the object store corresponding to Tier-0 and grid produced datasets during 1 month (May 2018). In this scenario a consumer receives data at dataset (tid) granularity, which can be processed in a multithreaded way. This process involves reading all the objects from the object store corresponding

Table 5.2: IFIC Kudu cluster specifications

Hardware (5 x machines)	Software/Configuration
2 x Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz (14 cores/CPU)	Kudu 1.7
16 x 16 GB RAM DDR4 @ 2400 MHz (256 GB)	Impala 2.11
8 x data disks SATA SEAGATE ST6000NM0034 (6TB)	RAID 10 (22TB)
1 x OS disk SSD SAMSUNG MZ7KM240 (240GB)	Spark 1.6 (cdh5.14.2)
1 x Intel SSD DC P3700 (1.5 TB) pci nvme	WAL on Intel SSD
2 x 10 Gpbs ethernet controller	

to a particular dataset, transforming the data into the Kudu schema, and writing the data into the backend storage.

The schema transformations involve decoding the trigger bits packed in the original format with B64 encoded strings, as written in the HDFS files and the Object Store.

L1 trigger mask field contains the numerical indexes of the trigger bits fired, with the components TBP, TAP and TAV sequentially packed. Decoding can be done directly in a byte array, which is stored in the *l1trigmask_bytes* column. The length of each component (L1_COMPLEN) is 256 bits for RUN1, or 512 bits for RUN2. So the first L1_COMPLEN bits of the byte array can be directly mapped to the *tbp0-7* columns, next L1_COMPLEN bits to the *tap0-7*, and the final L1_COMPLEN bits to the *tav0-7* columns.

High level trigger (HLT) mask is stored in three B64 encoded strings, separated by a ';'. Each of these strings represent the physics (PH), passthrough (PT) and resurrected (RS) components with a length of 4096 bits (HLT_COMPLEN). Similarly to the L1 trigger mask, the three HLT components can be decoded in the *eftrigmask_bytes* byte array, and directly mapped in order to the (*lhltph0-63*), (*hltpt0-63*), and (*htlrs0-63*) columns.

Several tables and configuration were tested:

- Base: base schema.

- Base-t1: partitioning on HASH (eventnumber)=8 and RANGE (runnumber).

- Base-t2: same partitioning as t1 and key ending on <...,runnumber, eventnumber>.
- Epoch-t1: partitioning on HASH(eventnumber)=4 and RANGE(epoch)=4 with epoch=runnumber%4.
- Epoch-t2: partitioning on HASH(eventnumber)=8 and RANGE(epoch)=4 with epoch=runnumber%4.

The *Base* class of table configurations refers to the original schema used and some variations. The base tests use no partitioning, and then on *Base-t1* we include RANGE partitioning over the *runnumber* and HASH partitioning on the *eventnumber*. On *Base-t2* we use the same partitioning and the key ending as in the final schema presented.

The *Epoch* class of table configurations includes range partitioning based on the *epoch* field, which is calculated during ingestion as the runnumber modulus the number of partitions. In addition HASH partitioning based on the *eventnumber* is used, with configurations using 4 (*Epoch-t1*) and 8 buckets (*Epoch-t2*).

Results on the ingestion can be seen on figure 5.2. There is a small queue of lower rates with mixed configurations, while the common ingestion rate is 5-6 kHz (events per second) per thread. A bit higher rates of up to 7 kHz can be reached with *Epoch* configurations. As we can see, schemas that distribute the load evenly over the key space produce better ingestion results.

An analysis of the time per stage during ingestion can be seen on figure 5.3. A writer thread spends less than 1% of the time on waiting for data from the Object store. Then parsing and data transformations takes 4% of the time. Inserting on Kudu client buffers takes about 23% of the time, and the final stage on flushing the buffers to the server takes most of the time (72%). Only the first base schema configuration produces different results, as it tests the ingestion when there is duplicate data (same key). By comparison the flush time is reduced considerably as the duplicates are not sent over the network.

5.3 HBase and Phoenix

HBase is a distributed highly scalable key-value database, designed for real-time read and write random access. HBase sits on top of HDFS where the data resides, but it uses transient in-memory storage to provide these features.

Data is organized in columns, and one or more columns can form a row. Rows are addresses by a unique row key, which are always lexicographically

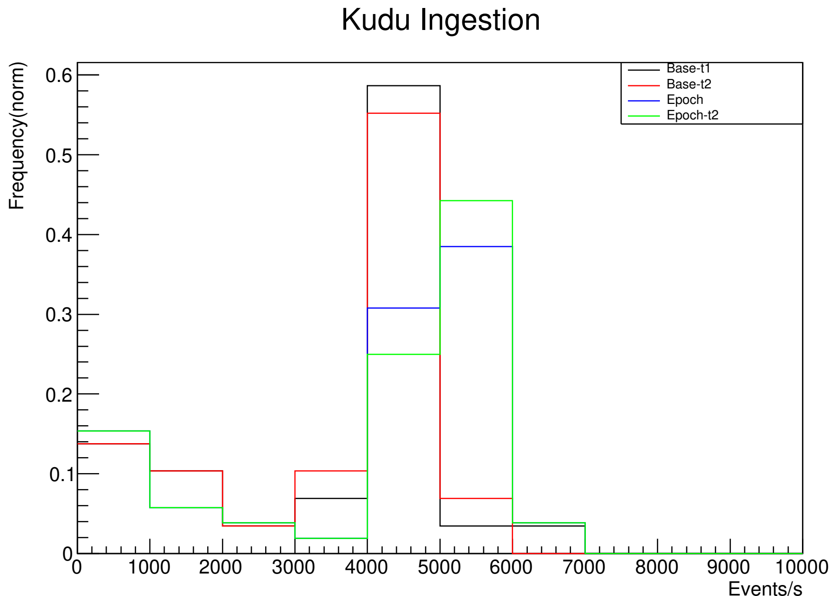


Figure 5.2: Kudu ingestion

sorted. A number of rows form a table, and there can be many of them. Key and values are stored using raw byte arrays, there are no defined types. Columns are grouped in columns families, which denote semantically but also physical relations. These are stored together (affecting compression for example) in an HFile in HDFS. Access to row data is atomic including any column read or written, but the guarantee does not span to other rows or tables.

The basic unit of scalability and load balancing is a region. Regions are continuous ranges on the key space, and are dynamically created. Each region is served by exactly one region server, that in fact can host many of them. Initially there is only one region, but when data is added and reaches a maximum size, the region will be split in two automatically (*autosharding*). Regions can be moved among regions servers when the load is under pressure, achieving load balancing.

When data is written it is first written to a commit log called a Write-Ahead Log (WAL) in HBase, and then stored in the in-memory memstore. When data reaches a limit it flushes to a HFile, discarding the commit logs. Data can

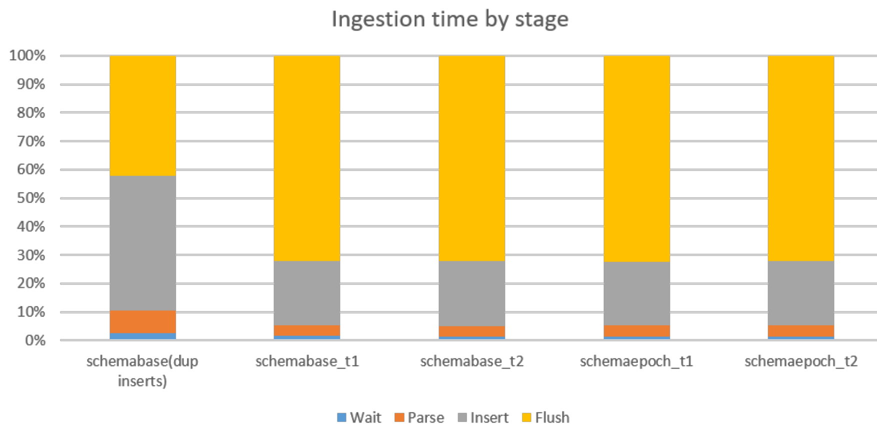


Figure 5.3: Kudu ingestion time per stage

be served meanwhile to readers and writers rolling the memstore in memory. Data is sorted from the beginning, so no sorting of other data consolidation procedures has to be performed. Flushing can cause more and more HFiles to be created, so HBase has a compaction mechanism running in the background merging files into larger ones.

Apache Phoenix [58] provides an SQL access layer on top of HBase. It also provides flexibility of late-bound, schema-on-read capabilities with dynamic type binding. A relational model can be defined with tables with defined column types, and a composite primary key. Internally Phoenix will map columns and serialize the data types to bytes stored in HBase. The SQL planner and optimizer is in charge of taking the SQL queries, and compile them in a series of HBase scans. This uses directly the HBase API, along with co-processors and custom filters that can run server-side. The access is done with standard JDBC, making the integration with other relational tools very easily. A number of optimizations are available to be supported at server side. Using a salting prefix to the key is automatically supported, and guarantees spread of all rows across regions. Statistics are collected of selected keys called guideposts, that act as guides to improve the parallelization of queries on a given region.

The EventIndex architecture using HBase and Phoenix in the storage component is shown in figure 5.4.

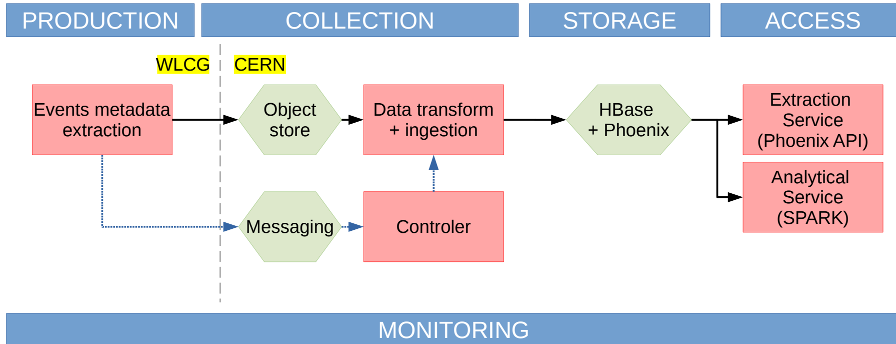


Figure 5.4: EventIndex architecture with HBase as unique data storage with a Phoenix layer to provide SQL capabilities.

5.3.1 Data organization

The EventIndex data structures using HBase Storage and a Phoenix schema were developed within the project to include a big events table that stores all the real and simulated event records [89]. Other auxiliary tables are defined with values referred in the main events table, like datatypes.

The contributions in this work are related to the data ingestion and book-keeping of the data. We have defined other meta-information tables to maintain basic hierarchical relations of datasets and dataset containers. In addition, we need to import the previous consolidated data from HDFS into the new HBase/Phoenix data structures. Some other temporary auxiliary tables are defined to help in this task.

Events table

In HBase, the best performance is obtained when searching for data using row keys. Random access by the complete key is translated into an HBase get operation, with excellent performance. A range scan over a prefix of the key is also a common operation that yields best performance. The EventIndex schema is designed to include the most accessed information to solve the required use cases in the key, leaving extra information in the value. Searching by value is internally transformed into full HBase scan operations, which has a lower performance. Having billions of entries means that keeping the key length to

a minimum is important both for performance and for total occupied volume reasons.

A representation of the events schema can be found in table 5.3. Phoenix allows us to specify the tables with a DDL (data definition language), which will map Phoenix types (second column in the table) to HBase native types. In this schema every entry is defined by a composite row key, seen in the table in the first four rows. The values of every entry are grouped in four families (a, b, c, and d) as seen in the prefix used in the column names.

Row Key is a 16-byte binary HBase value, composed internally of four parts in order to be used for range prefix searches: *dspid.dstype.eventno.seq.*

- *dspid* (Integer : 4 bytes) is an identifier generated at ingestion time. It takes into account internally the dataset name, except the datatype. Thus, datasets with the same Project.runNumber.StreamName.prodStep.AMITag will share the same dspid. This is intended to allow range scan by dspid in HBase.
- *dstypeid* (smallint: 2 bytes) is an identifier for the datype extracted from the key. All the different datatype of a dataset will sit close in the backend storage, making the search very performant and useful for solving the dataset overlaps computation use case. This is internally computed into dataTypeFormat (5 bits = 32 values) and dataTypeGroup (11 bits = 2048 values) as defined in the dataset nomenclature (see subsection 3.5.1), and its actual values are defined in the Data Types table (5.3.1).
- *eventno* (Long: 8 bytes) is the event number.
- *seq* (Short: 2 bytes): sequence used to deduplicate event entries when the EventNumber collides. This makes the row key unique in case of datasetName and EventNumber duplication, and is computed as the crc16 value of (GUID:OID1-OID2) which is unique. GUID is the identifier of the file containing the event, and OID1-OID2 are the internal pointers within that file. With this approach using hash values, there is a small possibility of key clashing but this is estimated low enough in our production system to not cause problems.

Column families. The values of every entry are grouped in 4 families, with the following configuration:

- *Family A*. Event Location information (and Monte-Carlo information for simulated data).

Table 5.3: HBase/Phoenix *Events* table schema as defined in [89].

Column name	Column type	Primary Key (PK) / Comment
dspid	integer	(PK) dataset primary id
dstypeid	smallint	(PK) dataset type id
eventno	biging	(PK) event number
seq	smallint	(PK) event sequence
a.tid	integer	dataset tid
a.sr	binary(24)	self reference (guid, oid1, oid2)
a.mcc	integer	monteCarlo channel number
a.mcw	integer	monteCarlo event weight
b.pv	binary(26) array	provenance
c.lb	integer	lumiblock number
c.bcid	integer	bunch crossing identifier
c.lpsk	integer	L1 trigger prescaler key
c.etime	timestamp	event time (nanoseconds)
c.id	bigint	L1 trigger identifier
c.tbp	smallint array	L1 trigger before prescaler (TBP)
c.tap	smallint array	L1 trigger after prescaler (TAP)
c.tav	smallint array	L1 trigger after prescaler (TAV)
d.lb1	integer	lumiblock number
d.bcid1	integer	bunch crossing identifier
d.hpsk	integer	HLT prescaler ley
d.lph	smallint array	L2 trigger physics (PH)
d.lpt	smallint array	L2 trigger passthrough (PT)
d.lrs	smallint array	L2 trigger resurrected (RS)
d.ph	smallint array	HLT trigger physics (PH)
d.pt	smallint array	HLT trigger passthrough (PT)
d.rs	smallint array	HLT trigger resurrected (RS)

- *tid* (Integer : 4 bytes) production task identifier: the numeric value found in the dataset name suffix "_tidNNNNNNNN_X" for this kind of tid datasets, or 0 otherwise. This further identifies the part of the dataset that it belongs to, for data bookeping purposes.
- *sr* (binary: 24 bytes) self-reference: a binary composed of the GUID (16 bytes) file identifier where the event is found, OID1 (4 bytes) and OID2 (4 bytes) representing pointers inside the file to locate the actual event.
- *mcc* (Integer : 4 bytes) Monte-Carlo channel number: in case of simulated events, not used otherwise.
- *mcw* (float: 4 bytes) Monte-Carlo weight: in case of simulated events, not used otherwise.

- *Family B.* Event Provenance. Provides information about data lineage.
  - *pv* (binary array: 26 bytes per entry) provenance: a binary array with one entry per element of the data lineage in the production chain of the event as found in the analyzed file. Every entry is 26 bytes binary composed of the dstype (2 bytes) and the self-reference (24 bytes) as previously presented.

- *Family C.* Level 1 (L1) Trigger information.
  - *lb* (Integer : 4 bytes) lumiblock.
  - *bcid* (Integer : 4 bytes) bunch crossing identifier.
  - *lpsk* (Integer : 4 bytes) L1 trigger prescaler key.
  - *etime* (timestamp: 16 bytes) using the java Timestamp type with an internal representation of the number of nanoseconds from the epoch.
  - *id* (bigint : 8 bytes) L1 trigger id
  - *tpb* (smallint array) trigger before prescaler: a variable length array of smallint (2 bytes) entries.
  - *tap* (smallint array) trigger after prescaler: a variable length array of smallint (2 bytes) entries.
  - *tav* (smallint array) trigger after veto: a variable length array of smallint (2 bytes) entries.

- *Family D.* Level 2 (L2) and Event Filter (EF) Trigger information for Run1 / High Level Trigger (HLT) information for Run2 onwards.
  - *lb1, bcid1.* Contains same values as the counterpart in family C, named with a suffix due to a limitation in apache phoenix spark to access the same field names in different families.
  - *hpsk* (Integer : 4 bytes) HLT trigger prescaler key
  - *lph, lpt, lrs* (smallint array): Level 2 (L2) physics, passthrough and resurrected variable length of arrays.
  - *ph, pt, rs* (smallint array): HLT physics, passthrough and resurrected variable length of arrays.

Datasets table

We have defined a new table schema to keep information about which events are stored in what (tid) datasets. This is needed for data discovery by any field on the dataset name, or by the included summary data. The schema also provides bookkeeping information on the dataset states, starting with data ingestion. In addition, this stores calculated information during the analytic procedures calculated over the related events data (see section 6.2).

A representation of the datasets schema can be seen in table 5.4.

Row key is defined to reflect information at (tid) dataset granularity. It includes the dataset identification fields, and fields that act as foreign keys on the events table (although Phoenix and HBase do no support foreign key constaints):

- *project, runnumber, streamname, prodstep, datatype*, and *version* (AMItag without _tid suffix). These represent the string fields that compose a dataset name. Note that runnumber is located first in order to be able to prefix search by run number. Although this table is orders of magnitude smaller than the events table, it is still desirable to optimize the key for searches.

- *tid* for those datasets that include them in the *"_tidNNNNNNNN_X"* suffix is the numerical value of the NNNNNNNN string, or 0 in other cases.

- *Dspid* and *dstypeid* refer to the values stored in the events table, and is the way to link these two tables. As a useful side effect, it defines

Table 5.4: HBase/Phoenix *Datasets* table schema.

Column name	Column type	Primary Key (PK) / Comment
runnumber	integer	(PK) dataset / run number
project	varchar(200)	(PK) dataset project
streamname	varchar(200)	(PK) dataset streamName
prodstep	varchar(200)	(PK) dataset production step
datatype	varchar(200)	(PK) dataset format
version	varchar(200)	(PK) dataset version (AMITag)
tid	integer	(PK) dataset tid
dspid	integer	(PK) dataset primary id
dstypeid	smallint	(PK) dataset type id
smk	integer	trigger supermaster key
events_rucio	bigint	number of events by rucio
rucio_at	timestamp	last request to rucio time
files	integer	number of files (GUIDs)
events	bigint	number of event records
events_uniq	bigint	number of unique events
events_dup	bigint	number of duplicated events
files_dup	integer	number of files with duplicate events
state	varchar(200)	dataset state
state_modification	timestamp	last modification of state
state_details	varchar(20000)	state details
insert_start	timestamp	insertion start time
insert_end	timestamp	insertion end time
source_path	varchar(20000)	original input data path
updated_at	timestamp	updated time
dups_at	timestamp	duplicates calculation time
trigger_at	timestamp	trigger calculation time
is_open	boolean	dataset is open/closed
has_raw	boolean	dataset has raw data
has_trigger	boolean	dataset has trigger data
prov_seen	smallint array	dataset provenance

a canonical container [90] (see subsection 5.3.1), which includes all the events that share the same dspid.dstypeid, grouping all dataset fields but the tid.

Values. Rest of the rows are listed in three groups of metatadata about the dataset:

- Caching or computed values: *smk, events, events_uniq, events_dup, files_dup, prov_seen.*
- Booleans identifying characteristics of the dataset: *is_open, has_raw, has_trigger.*
- Information and timestamps about bookkeeping of this dataset: *state, state_details, state_modification, source_path, insert_start, insert_end, updated_at, dups_at, trigger_at.*

We will detail the usage of these fields in the next subsections.

Datasets containers table

In this table we define a representation of a particular kind of container, a canonical dataset container that groups all the information about entries in the datasets table that share the same dspid.dstypeid. The structure of the table is basically the same as the datasets table, but the key does not include the tid. Therefore, an entry in this table represents a canonical dataset container, which might reflect one or more entries in the datasets table. The fields of an entry represent the sum of the fields of the related entries in the datasets table, or the calculated values at the (canonical dataset) container granularity.

Table 5.5: HBase/Phoenix *dstypes* (data types) table schema [89].

Column name	Column type	Primary Key (PK)/ Comment
type	tynyint	(PK) type class
name	varchar(20)	(PK) type name
id	smallint	type numerical identifier

Table 5.6: HBase/Phoenix *dsguids* (dataset GUIDs) table schema [89].

Column name	Column type	Primary Key (PK) / Comment
dsname	varchar(255)	(PK) dataset name
id	integer	(PK) dataset identifier
tid	integer	(PK) dataset task identifier
guid	varchar(127)	(PK) file (GUID) identifier

Auxiliary tables

Other auxiliary tables help in the process of data identification, or during the ingestion and importing procedures.

- Data types table (*dstypes*). As can be seen in table 5.5, this stores the relation about the type *name* and numerical identifiers about the data type (*dstypeid* in events and datasets tables). Internally *type* class distinguishes between data type formats (type=0) and data type groups (type=1). For every entry we can find its *name*, and its numerical *id*. The id values will be used for building the dstype id field in the row key, and other fields in the data location and provenance families.

- Files table (*dsguids*) as can be seen in table 5.6. This schema reflects information about the dataset name, the contained files (*guid*), and their *tid* production task identifier. This is needed because during the data ingestion procedure (see subsection 5.3.2) this guid data might not be available in the source, and we need to consult it in this table to fill in the proper events (location and provenance fields) and datasets (tid field) tables.

5.3.2 Data ingestion

We carried out several developments to support the data ingestion on the HBase and Phoenix backend [89].

Figure 5.5 shows the part of the EventIndex architecture related to the data ingestion on the new HBase/Phoenix storage. A new *PhoenixWriter* plugin for the consumer of the distributed data collection task was implemented and evaluated for performance. With it the complete data ingestion chain was complete and we could easily exchange the previous data backend in our

framework. In addition, an import procedure (*PhoenixImporter*) was developed to move the massive amount of data already consolidated in the production HDFS backend, to the new HBase/Phoenix one.

Several experiments were done to evaluate importing individual and massive amounts of datasets. Experiments were conducted in parallel to evaluate the best schema and table configuration parameters for our data model, taking into account different possibilities.

PhoenixWriter consumer

The *PhoenixWriter* internally uses the JDBC driver provided by Apache Phoenix in order to interact with the backend HBase/Phoenix storage implementation. The data consumption and ingestion uses the same logic within the production data collection system, reading the input data in Object Store in protobuf format, transforming the data to the new data schema, and effectively ingesting the data in HBase/Phoenix backend system.

The used data model schema has been previously explained, and it contains several destination tables. Several data transformation classes were implemented (eventindex.lang.phoenix package) in order to facilitate the data transformation and augmentation.

The first ingestion tests were done loading 48 Tier-0 datasets sequentially. The consumer was configured with a single writer thread. Figure 5.6 shows a histogram of the results of the ingestion tests. The x-axis shows throughput in number of events ingested per second by a single thread. The y-axis shows the normalized frequency, or number of appearances of each case. We observe that the mean is around 3 kHz per thread, with some datasets reaching up to 5 kHz. Lower performance can be seen in around 18% of the datasets due to small dataset size, as in this case the JDBC driver initialization and in particular the HBase data streams setup can dominate most part of the ingestion time.

We can observe the distribution in phases during ingestion time in figure 5.7. The first one is the *Wait* phase, 1.5% of the time is waiting for input data. The actual reading of the input data is done within the consumer by another thread, in order to decouple input data access from processing. In this case the consumer is accessing the Object store and retrieving the objects that contain the input data. The second *Parse* phase is devoted to data conversion from the input protobuf format, to the output HBase/Phoenix types and schema. This phase takes 8.5% of the time. Next phase is the *Insert* phase of the transformed data into the JDBC client buffers, which takes 41% of the time. And the last phase is the *Flush* phase, which commits the buffers to HBase/Phoenix on a

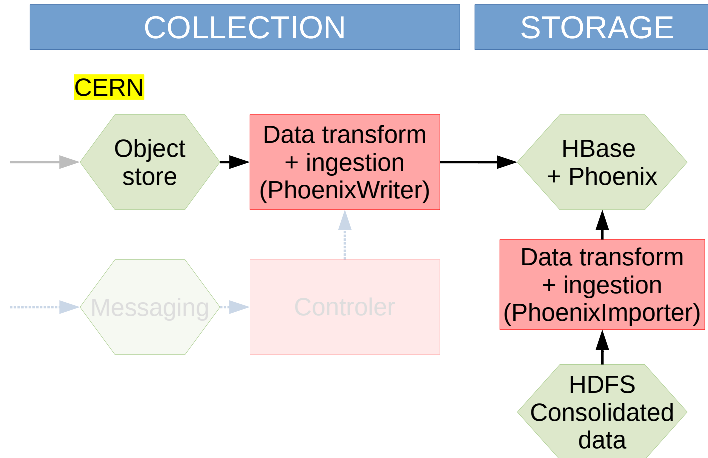


Figure 5.5: Close-up view of the EventIndex architecture with HBase/Phoenix storage on the data ingestion and import methods. The PhoenixWriter is integrated in the Data Collection component to ingest data in line with production. The PhoenixImporter is a transient component in charge of importing previous consolidaded data (LHC Run 2) from HDFS into the new HBase/Phoenix storage.

regular basis (in this experiment tested every 100 inserts). This last phase takes the rest 48% of the time.

As we can see, most of the time is dedicated to the setup and communication with the HBase servers, with a small amount of time required for data access and transformation.

Consolidated data import method

We developed an import procedure using MapReduce [59] jobs to read input production data, transform to the Phoenix Schema, and store in HBase/Phoenix tables. These *PhoenixImporters* share code and use the same data mangling methods as the PhoenixWriter Consumer.

The input data resides in HDFS file system within the production CERN Hadoop Cluster used by EventIndex. As we have seen, format of the input data is MapFiles or SequenceFiles, which might be ordered or not, compressed or

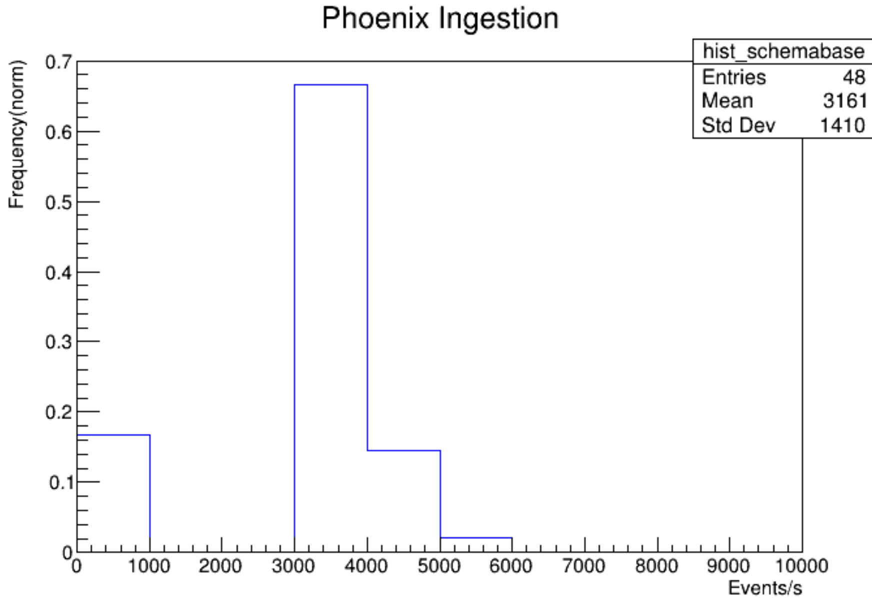


Figure 5.6: Hbase/Phoenix ingestion throughput.

not, using the EventIndex Hadoop Core CSV schema with an event entry per line.

The output data will be in the proposed HBase/Phoenix schema, using the events and rest of the meta-information tables which serve also for the bookkeeping purposes of the ingestion procedure.

The procedure itself consists of the submission of MapReduce jobs to the Hadoop cluster, with the input data path's desired granularity (individual dataset, entire project, etc.). The map-only tasks transform each event from the original format to the HBase/Phoenix schema, writing it in the events tables. The additional metadata that is not available in the original format and is needed for the event keys (*dspid*, *dstypeid*) is read from the meta-information tables. This is also what happens for the *tid* and *guids* relations, which might not be available in HDFS consolidated data. This metadata will be provided by the Data Collection task when the import process is done in production, but for these tests the data can be pre-filled or generated dynamically by this import procedure. The procedure looks for the needed metadata entry and if

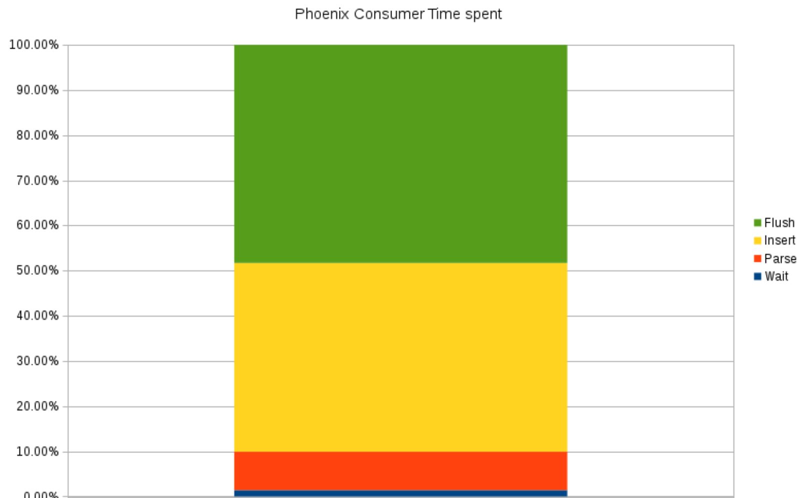


Figure 5.7: Hbase/Phoenix ingestion time per stage.

not available uses an internal *sequence* (a standard SQL feature supported by Phoenix for generating monotonically increasing identifiers) to generate it.

The rest of the fields can be transposed directly with the new format, or computed on the flight from the original input data.

We have implemented a check pointing mechanism at dataset level, this being able to stop and restart the procedure. This procedure avoids restarting successful jobs, unless an overwrite option is configured explicitly. We are writing the bookkeeping data in the setup() and cleanup() map-reduce phases. They update the datasets tables with information about what data is imported (logical datasets, but also referring to the physical HDFS files), the date, the status, and dataset metadata (number of events, etc.).

The import procedure fills the following fields on the dataset tables:

- *state*: with values indicating the procedure phase like INSERTING, DONE, or FAIL.
- *state_details*: is written with details of the state phase like in case of failure.
- *source_path*: with the actual HDFS path of input data.

- *state_modification, insert_start, insert_end*: These timestamps are written when changes are produced in the related fields.

- *events*: with the number of events inserted.

Schema configuration evaluation

We did some experiments in order to decide about final configuration parameters on the Phoenix events table like : salting, pre-spliting, storage format and column encoding bytes. This was also conducted to determine the capabilities on the backend system during ingestion and query.

We defined several Phoenix event tables with different options, and we imported a substantial amount of data into the Hadoop Cluster at CERN supporting HBase/Phoenix, to see performance and differences.

The configuration parameters that are part of the experiment can be grouped in the following categories:

- *Salting*: to automatically include a prefix byte hashed to distribute load among regions and avoid hot-spotting when using same or monotonically increasing keys. Tested in Phoenix with *SALT_BUCKETS* configuration parameter equal to 0, 4, and 10 buckets.

- *Region pre-splitting*: A pre-split table can be created with different regions, in order to distribute the load and avoid creating and moving many regions with the first loads. Phoenix supports it with the DDL clause *SPLIT ON* and providing ranges on the row key space.

- *Column mapping*: column names are encoded in every row, so it is recommended to keep them short to reduce store space used. Column mapping can be used to record a short identifier as indirection to the real column name. We tested 0 (no encoding) and 1 byte encoding.

- *Immutable rows*: encoding all column data from a family in a single cell. This is supposed to reduce total space used and improve writing and reading performance, at the cost of accessing individual cells from HBase not possible without Phoenix layer.

- *WAL (Write Ahead Log)*: avoiding writing a WAL can improve writing performance, at the cost of losing data if a server crashes during writing. This is not a high risk if the main source of data is on HDFS or Object Store as it can be replayed.

- Encoding and compression: as previously tested the best option is to use default diff encoding and Snappy compression [90].

Several experiment cases were designed in order to test data ingestion with several different configuration parameters:

- *Case a (base)*: select basic options, with no salting, no limit on encoded column bytes, fast_diff encoding and snappy compression, and without initial region splits. The events table Phoenix DDL syntax applied was the following:

```
DATA_BLOCK_ENCODING='FAST_DIFF', COMPRESSION='SNAPPY',
↪  COLUMN_ENCODED_BYTES=0;
```

- *Case b (split)*: use base options and pre-split with 15 initial regions.

- *Case c (inm)*: use pre-split, and immutable and coded columns. The applied DDL syntax was the following:

```
IMMUTABLE_STORAGE_SCHEME =
↪  SINGLE_CELL_ARRAY_WITH_OFFSETS, COLUMN_ENCODED_BYTES
↪  = 1, DATA_BLOCK_ENCODING = 'FAST_DIFF', COMPRESSION
↪  = 'SNAPPY', SPLIT ON ( ...15 SPLITS);
```

- *Case d (salt)*: use pre-split, and salting on 4 buckets. The applied DDL syntax was the following:

```
IMMUTABLE_STORAGE_SCHEME =
↪  SINGLE_CELL_ARRAY_WITH_OFFSETS, COLUMN_ENCODED_BYTES
↪  = 1, DATA_BLOCK_ENCODING='FAST_DIFF',
↪  COMPRESSION='SNAPPY', SALT_BUCKETS = 4 SPLIT ON (
↪  ...15 SPLITS);
```

- *Case e (wal)*: use pre-split, and disable write aheal log (WAL). The applied DDL syntax was the following:

```
IMMUTABLE_STORAGE_SCHEME =
↪  SINGLE_CELL_ARRAY_WITH_OFFSETS, COLUMN_ENCODED_BYTES
↪  = 1, DISABLE_WAL = true,
↪  DATA_BLOCK_ENCODING='FAST_DIFF',
↪  COMPRESSION='SNAPPY', SPLIT ON ( ... 15 SPLITS);
```

Table 5.7: Results of the experiments with several Phoenix schema configuration cases.

Case	Input (HDFS) Datasets	Events	Output (HBase) Size	Regions	Time	Throughput Max (WO)
a (base)	580	5.2×10^9	-	-	1 days	44 kHz
b (split)	20460	5.8×10^9	1.31 TB	227	10 days	55 kHz
c (inm)	20460	5.8×10^9	1.33 TB	239	10 days	85 (110) kHz
d (salt)	20458	5.8×10^9	1.35 TB	300	10 days	75 (262) kHz
e (wal)	16635	5.4×10^9	-	-	2 days	96 (119) kHz

All the cases were tested submitting 10 writing threads per tested case. A special experiment was run reading and transforming the data only once, and writing in the three output tables simultaneously for cases c, d and e (write-only parts of the experiment). Results of the import experiments according to different schema configuration cases can be seen in table 5.7. Input data resides in HDFS, detailed with the number of datasets and the number of events that have been read. Output data is written in HBase with parameters defined in each case; the result table specifies the occupied disk (Size) and the number of HBase regions (Regions) used. Time of the experiment is included, which was run for 10 days for all, except special, cases. The throughput measures processing rates per thread in Hz. This represents the number of events processed (read, transformed, and written) per second. Enclosed in parentheses, this shows the maximum writing-only (WO) rate per thread, when taking into account only writing procedures (leaving out data read and transformation procedures).

Results on *case a (base)* show the baseline configuration and performance. The experiment was run for 1 day, importing 580 big datasets. Output results in terms of volume size and regions occupied were not available at that time. Experiments on *case b, c, and d* were run for 10 days each, adding in the order of 20,000 more smaller datasets. The results eventually showing similar occupied space and HBase regions used. Only *case d (salt)* shows higher numbers on the regions affected. During the experiment with *case e (wal)* some datasets were removed in production, so only throughput values are meaningful.

Results on *case a (base)* show the baseline performance, which yields a maximum rate of 44 kHz (events per seconds). Throughput is improved a bit in *case b (split)* with up to 55 kHz when using a pre-splitted table configuration. The next cases *case c, d, and e* add options, effectively increasing rates over the base and split cases. The best regular performance is achieved when avoiding a

WAL as shown in *case e (wal)*, at the cost of increasing the risk on data loss during ingestion. Using a 4 bucket salting configuration in *case d(salt)* yields the best performance in the write-only experiment.

Salting configuration increases writing performance, but it also affects reading as it has to collect several region's data [91]. Column mapping was tested without encoding, and with 1 byte encoding, and the observed occupied space is similar. We therefore recommend not using this feature, so defining a table configuration with *COLUMN_ENCODED_BYTES=0*. Regarding immutable rows, in our tests we did not see space reduction compared with mutable rows, although we saw small increases in writing performance. On the other hand, using immutable rows makes reading from HBase directly more difficult, as it imposes a custom encoding done by Phoenix. Removing the Write Ahead Log (WAL) allows an increment in writing performance. Our main source of data is on the Object Store (or HDFS during import/conversion phase), so in case of machine crash during data ingestion there is no real risk of losing data as it can be always replayed (this was not the case with the messaging approach as seen in section 4.1). We could avoid the WAL if we need to increase writing performance during bulk loading on import phases. This feature can always be altered later.

Although there are options that might be interesting for both writing and reading performance, all these options depend on Phoenix functionalities that introduce a dependency on this. We have decided to maintain HBase compatibility without using extra features that cannot be used if we remove the Phoenix layer.

Individual dataset performance

In this experiment we are testing the importing of individual datasets from HDFS into the HBase/Phoenix backend. We want to check the baseline performance that we obtain using minimal resources, namely one map-reduce job with a single mapper task, and a single file split. The job is launched within a single YARN container with 1 vcore (virtual core). With this configuration we avoid dividing the task in multiple threads, to measure single thread baseline performance.

During the experiment we sequentially launch a job for indexing a single dataset of different size. In this case we are using real data datasets in the order of 1 MB, 100 MB, 1 GB and 10 GB.

Results of the experiment are shown in table 5.8. Due to the nature of the real data, there is variability on the actual size of the datasets (second column), but also on the number of events contained (third column) and the event size

Table 5.8: Results of single dataset ingestion into HBase/Phoenix.

Dataset	Size (bytes)	Events count	Evt size (bytes)	Time (s)	Throughput (evt/s)	Throughput (bytes/s)
1 MB	1,873,145	11,486	163.08	22	529.00	85,142
100 MB	120,861,878	81,134	1489.65	50	1,639.70	2,417,237
1 GB	1,556,266,245	18,151,016	85.73	4,332	4,199.38	359,248
10 GB	11,867,886,730	38,863,293	305.37	12,521	3,103.68	947,838

(fourth column). We find small events (order of 100 bytes/event) in the dataset samples of 1 MB and 1 GB. The largest dataset in number of events and total size is as expected the 10 GB dataset, but on the contrary, this has medium size records with 305 bytes per event. The biggest events can be found in the 100 MB dataset, with 1.5 kB per event.

Throughput in events/s vary from 500 Hz (1 MB dataset) up to 4.2 kHz (1 GB dataset). The 10 GB dataset has a slightly lower rate of 3.1 kHz. The highest rates in events/s are reached on the 1GB dataset, where the event size is the smallest. On the other hand, we can see that with bigger events we can reach higher rates in bytes/s as we can see in the 100 MB dataset.

The lowest numbers on the 1 MB dataset are probably due to the time it takes for the preparation of the jobs, and the establishment of the reading channels (HDFS) and the writing channels (HBase). In case of small datasets this setup might dominate the total time, reducing the rates for small data payloads like this one.

After writing, we check the validity of the written data in the events table, counting the records. This operation takes 1 second for the 1 MB (11,486 events) dataset, up to 42 seconds counting the 10 GB (38 M events) dataset.

Massive ingestion performance

One of requirements to adopt in the new storage backend is to import all the EventIndex data consolidated in the production HDFS backend. We analyze now a massive ingestion experiment that we run in order to check the response of the system under high loads. In addition this ingestion campaign will allow us to have data in the new system for the query access developments and experiments (see chapter 6).

As experimental platform we utilized the CERN Hadoop clusters. The input production data resides in HDFS (lxhadoop.cern.ch cluster). The destination

data will be the newer analytix cluster (analytyx.cern.ch). At the time of experiment it was composed of 39 nodes (32 HBase region servers), with a total memory of 18 TB, and 1,658 vcores (virtual cores). It has to be noted that this is a shared cluster by multiple projects at CERN, including EventIndex. The configuration includes the following software distributions: Hadoop 3.2.1, HBase 2.2.4, and Apache Phoenix 5.0.

The input data corresponds to several HDFS directories organized at dataset container level. Our importing tool makes a mapping with a MapReduce job sent per dataset container, so eventually a number of tasks are submitted. The experiment ran for 1 week in total, with several invocations of our tool:

- A first batch to index real data from 2018 (*data_18* project prefix), launched 6,254 tasks.
- Second batch to index data from 2017 (*data_17* project prefix), launched 6,796 tasks.
- Other jobs were sent to index a variety of datasets, including Monte-Carlo and real data from other years.

The output data will be in HBase according to the defined Phoenix schema (see subsection 5.3.1). The events table configuration was defined with the following options:

```
DATA_BLOCK_ENCODING='FAST_DIFF', COMPRESSION='SNAPPY',
↪  SALT_BUCKETS=10
```

These configuration options specify using the diff encoding and Snappy compression, and a salting key prefix strategy. In these tests we are using a SQL sequence to generate needed identifiers not available in the input data (for example *dspid*) so there is the risk of using monotonically increasing keys. To overcome possible issues, we use 10 salt buckets to distribute the load among regions and avoid hot-spotting.

We started the experiment with a shared cluster configured to allow fair use up to 1,000 shared vcores and 3 TB of memory, depending on the load on the system. We observed several metrics over the duration of the experiment. Figure 5.8 shows the number of HBase write operations on the upper panel, and the volume of data transmitted to the server. Each line represents one of the HBase region servers, and the metrics are stacked so the aggregated values at a particular temporal point can be observed.

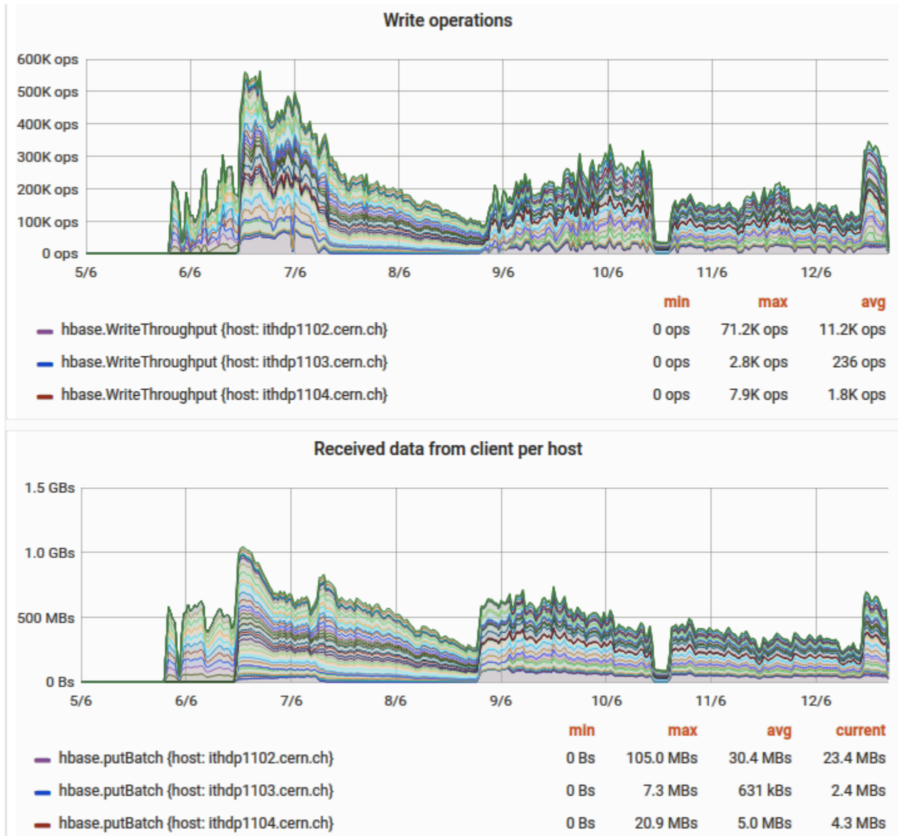


Figure 5.8: HBase region servers metrics over the duration of the experiment. The upper panel shows write operations; the lower panel shows volume of received data.

We submit the first batch of 2018 datasets (*data_18*) and we observe that when the importing processes starts at point 1 (date 6/06), the write operations are evenly distributed to only 10 Region server machines (initial 10 region split), reaching a maximum rate of 300 Kwrites/s (corresponding to 500 MB/s). Later that day (point 2, midday 6/06) we manually triggered a distribution of the regions to other unoccupied regions server machines. At this moment we see the performance rise due to the distribution load, reaching up to 550 Kwrites/s and a rate of 1 GB/s.

This leads us to think about a pre-splitting of the tables to better accommodate the writing load from the beginning. It seems there is no optimal selection for the adequate number of regions for a given load. The accepted recommendation is to start with a number of splits as a multiple of the number of region servers. Then HBase automated splitting will do the next splits by itself, but in our case we can accommodate higher loads from the beginning, with no slow start.

During the next day the number of write operations starts to decrease progressively, as the number of batch jobs related with the first batch of 2018 data are ending, although the bigger datasets are still running.

At point 3 (date 09/06) we start to submit the second batch of jobs, now related to *data_17* datasets, and we observe the metrics start to rise again.

At point 4 (midday 10/06), we killed all the tasks, so we see a steep decline. We relaunched the second batch of jobs to check the check-pointing mechanisms and that the process was restarted correctly. After restart the number of writes was reduced to 200 Kwrites/s, but maintaining the performance of 500 MB/s. At point 5 (afternoon 12/06) there was no change on job side, but the queue configuration was changed by the CERN cluster managers. A dedicated YARN queue was created for EventIndex with complete access to a maximum of 1k vcores, an using up to 4TB of memory. At this point the performance was raised to 300 Kwrites/s and 750 MB/s.

We can see that we don't reach the performance that we have seen at previous point 2.

After the experiment we checked that the vast majority of datasets were imported correctly. There were no failures with corrupted data or bad records. We compared the output data with original HDFS for selected datasets. The data transformation method for converting the previous data format to the new Phoenix schema was proven correct.

There were, however, some issues related to the job management inside the cluster. Some tasks were killed by YARN (Container preempted by the scheduler), due to the priorities configured in the cluster. Our import procedure

takes into account processing issues like this with bookkeeping procedures. Already correctly finished tasks (all dataset events are written correctly in HBase) will not be started again. There is no data integrity problem writing the same data in the events table, as the event keys are the same and the results will be idempotent. Yet this restarting means many CPU hours lost for processing the same data. To solve these issues, a dedicated YARN queue was created devoted to our project, as previously mentioned, so this preemption over running jobs should not happen again.

An issue for tasks running over 24 hours was discovered. In fact, tasks can run much longer without problem, writing all data, but when closing the connection, the internal JDBC driver loses the last batch of data (order of 100 events). During the experiment we tested changing the relevant configuration option (set phoenix.client.connection.max.duration $\geq$ 24H) without improving the situation. Our analysis discovered an issue in the JDBC usage in the Phoenix implementation, which was reported to the developers.

The measured resource usage on the cluster finally accounted for 2,347 HBase regions opened (using the 10 buckets splits as configured). The originally configured shares resources supposed a maximum of 1,000 vcores and 3 TB of memory. Our jobs were running at some points with up to 977 concurrent containers, saturating the memory. The cluster was configured to increase resources to 1,000 exclusive vcores and 4 TB of memory, supposing 20% of the cluster.

A compilation of the results can be seen on table 5.9. We have inserted a total of 7,941 datasets, containing over 70 billion (10^9) events. The occupied space goes up 21.95 TB in the events table (distributed into 2,347 regions in 32 HBase region servers as we have seen before).

The total volume size is divided into the four families of our data schema:

- *family A.* Event location: 1.89 TB.
- *family B.* Event provenance: 2.38 TB

Table 5.9: Results of complete experiment ingestion into HBase/Phoenix.

Datasets	Size (TB)	Events count	Evt size (bytes)	Time (days)	Mean Throughput (evt/s)	(MB/s)
7,941	21.95	70.77×10^9	310.19	7	117,027	36.30

- *family C.* L1 trigger: 9.25 TB
- *family E.* L2 and HLT physics trigger: 8.41 TB

The mean event size is 310.19 bytes. The rest of tables (datasets, containers and auxiliary tables) contain the auxiliary and bookkeeping data but this is negligible in terms of volume occupied.

During the experiment we have seen high rates reached of up to 550 Kwrites/s and 1 GB/s. in the servers. We have to take into account that submitting the information related from one event can be translated into multiple writes to the HBase servers, due to the schema division into families and the payload per write. In addition, the amount of data transmitted to the servers will be later compacted by the standard HBase compaction procedures. The throughput values in the table represent mean values for all the experiment and refer to the actual event rates (event count divided by the time of the experiment) and data rates (Size divided by the time of the experiment). Therefore this represents the real value of the imported event rate, which is 117 kHz (117,027 events imported per second).

5.4 Conclusions

We have presented several contributions regarding storage for metadata applications like the EventIndex. A study on different NoSQL storage technologies suitable for the EventIndex has been carried out and presented. The particularities of the implementations of our data model were discussed for HDFS, Kudu and HBase/Phoenix, focusing on data ingestion into these backends. We developed the data collection system required parts to substitute these backends and evaluate them. We contributed to the development of a new data model based on Kudu tables resembling a relational schema. The contributions on HBase/Phoenix in this work are related to the data ingestion and bookkeeping of the data, both for the data collection system from the grid and for importing the massive amount of consolidated data in the previous HDFS backend. Several experiments involving up to billions of events characterized in terabytes of data were performed.

Some conclusions can be drawn regarding throughput, data organization and complexity of the system. Although the throughput per single thread is higher with HDFS, we can achieve overall better performance in our application with HBase/Phoenix and Kudu. This is due to two main reasons. First, the writing mode and granularity of our data is more parallelizable with HBase and

Kudu. One HDFS file has to be written by a single writer due to the Hadoop design (single-writer, multiple-reader model). Currently our system writes at the dataset level to have big enough files. Kudu and HBase/Phoenix allow writing at the record level, so we can have multiple writers per dataset, increasing overall throughput of the system. Second, with Kudu and HBase/Phoenix we do not require extra augmentation and consolidation steps. These have built-in global ordered key space by design, and the data is ordered at ingestion without needing extra steps. An analysis of the time spent at different ingestion phases show that less than 10% of the time is dedicated to data transformations needed to accommodate the data schema, with the bulk of the time dedicated to the data transmission to the backend servers. Therefore, the decoding of all fields into these schemas can be done online during data ingestion, without needing extra procedures.

Other improvements are related to the use of a fixed data schema for all data and use cases. With HDFS we had to replicate part of the data in Oracle to solve some use cases, effectively having a complex hybrid system. Now we can maintain all the data into HBase avoiding complex management steps, data coherence issues, and reducing the total volume of used space. Compared with a schema-less system in HDFS, we can benefit from a schema enforced model in HBase/Phoenix or Kudu. This approach allows us to group related information into columns, define its types and have better compression ratios. In addition we don't store decoded trigger chain names as literal strings, saving a considerable amount of space compared with the previous implementation.

Kudu did not get large support in the open-source community, so HBase was finally selected for its good performance and better support in production. The study on different parameters on the HBase/Phoenix schema shown the best options for our application. An initial pre-splitting on the events table reduces the slow-start and increases data ingestion rates from the beginning. A key distribution strategy is needed to avoid hot-spotting on particular region servers due to monotonically increasing keys. A salting strategy was proven useful starting with 4 buckets. Avoiding a Write Ahead Log (WAL) is possible without risk of losing data because our data collection model has temporary data staging in the Object Store. The performance gain is not needed now, but can be switched on in case of necessity. Column mapping does not provide any improvement on reducing the volume occupied, since our schema already has compact column names. Immutable rows neither suppose an improvement in our application, and they impose a dependency on this Phoenix feature. We have avoided strict dependencies so we can have direct access to HBase data without Phoenix layer, in the case its support is not maintained, at the cost of

losing SQL access capabilities.

Overall with the new HBase approach, we observe that it can maintain the ingestion rates required by the current and future Runs of the EventIndex project. In addition, it can serve as the unique storage for all use cases. Therefore, we improve on the previous hybrid model avoiding duplicate data and coherence issues, and reducing the complexity of the system.

In the next chapter 6 we discuss the data access features and capabilities on the HBase/Phoenix backend.

6 Access

New tools are required for solving the required access cases within the new storage. The work developed in this thesis resulted in the development of new tools [92] for accessing the big data quantities of the EventIndex project stored in HBase/Phoenix using Spark [93] and implemented in Scala [94]. We provide data discovery capabilities at different granularities, producing Spark Dataframes that can be used or refined within the same framework. Data analytical cases of the EventIndex project are implemented, like the search for duplicates of events from the same or different datasets. An algorithm and implementation for the calculation of overlap matrices of events across different datasets is presented.

These tools can be used by other higher-level tools of the EventIndex project, to ease access to the data in a performant and standard way using Spark abstractions. They decouple the data access from the actual data schema, which makes it convenient to hide complexity and possible changes on the backed storage.

6.1 Requirements and use cases

The EventIndex main use case was event picking, or selection of particular event records based on criteria applied on a large metadata catalog.

Use cases have evolved over the duration of the project and more analytical cases have been considered. The requirements of these cases are related to pattern detection over large quantities of data.

Data consistency checks are the first group of use cases considered. ATLAS production processes during real data taking can temporarily fail, producing duplicate events with the same identifiers. In addition, Monte-Carlo procedures for simulating events are also vulnerable to generating incorrect data. Detection

of duplicate event data is required at different levels, from files to complete dataset containers.

The ATLAS derivation framework outputs the selected events that are requested by physics analysis groups from already available AOD files, acting as an offline trigger. It would be useful to be able to detect the event overlaps among the derived datasets, identifying them to optimize the procedures and used resources. This can be solved with the EventIndex stored metadata.

Other use cases requiring analysis over large quantities of data may arise in the future, so general access methods must be supported.

A new data access layer is required to leverage the backend data storage improvements being developed in HBase/Phoenix (see section 5.3). To support data retrieval and extraction, a new SQL interface opens the possibility of integration with JDBC [95]. Previous web front-ends designed to access relational data back-ends can be rapidly adopted. In addition, a new low-latency access framework is needed to support the analytic use cases with semantics expressed in higher level languages, instead of the restricted SQL syntax.

6.2 EventIndex Analytics Platform

The EventIndex analytics platform provides services for solving OLAP (online analytical processing) use cases and obtaining insights from the data. Figure 6.1 shows the proposed architecture which is based on Apache Spark [93], an engine for large-scale data analytics that provides abstractions for data modelling and in-memory efficient operations. This can be accessed interactively with command line consoles or web interfaces like notebooks. A programmatic interface is also available, making it easy to run background processes on the provided resources. Spark interfaces natively with resource management tools, as in our case, YARN [96] to provide access to the CERN cluster, which comprises dozens of machines that host data and computing servers.

Data storage for the next generation EventIndex has selected HBase to provide a unique and unified backend for all data and use cases (see section 5.3).

Regarding data access, HBase works best for random access, which is perfect for the event picking case where we want to use low latency access to a particular event to get its location information. Use cases when we need information retrieval (trigger information, provenance) for particular events are served with fast HBase *gets*, with good performance.

In addition, analytical use cases where we need to access a range of event information for one or several datasets (derivation or trigger overlap calculation),

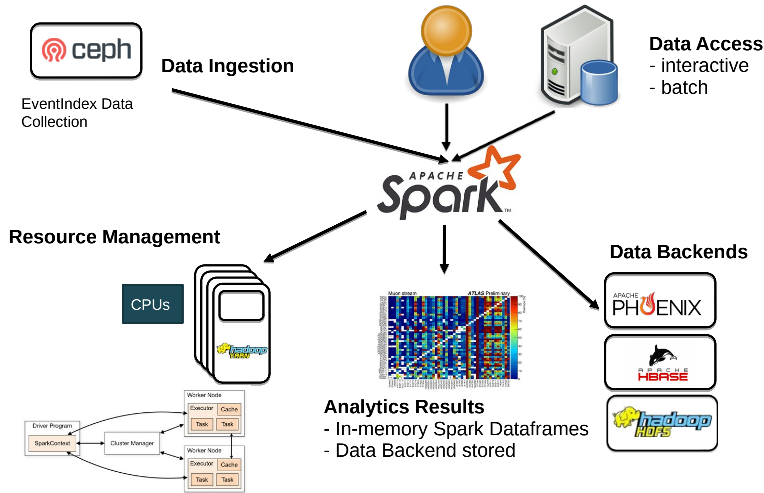


Figure 6.1: Architecture of the EventIndex analytics platform based on Apache Spark.

can be solved with scans on these data. They can be optimized with a careful table and key design to maintain related data near the storage, reducing access time. HBase is a column-family grouped key-value store, so we can benefit from dividing the event information into different families according to the data accessed in separated use cases. Further analytic use cases with larger amounts of data are not foreseen, but still can be achieved running MapReduce/Spark jobs on the HBase files (HFile), as they are stored in HDFS.

We have seen that Apache Phoenix is the layer over HBase that enables SQL access and provides an easy entry point for users and other applications. Although HBase is schema-less storage, Phoenix requires a schema and data typing to provide its SQL functionality. Schema versioning and dynamic late binding for the same tables are supported. In EventIndex, our data rarely varies its schema, so we can benefit from Phoenix designing the required schema and tables accordingly (see subsection 5.3.1).

We use Apache Spark as a framework for dealing with data in a distributed manner that provides some abstractions that are very useful and performant.

Scala is the language that Spark was written in and is the most supported programming language to access all Spark APIs. Its compiled bytecode is running in the JVM (Java Virtual Machine), so Java classes can be called from Scala, with the benefits of a concise and high level language.

Spark and Scala are used in this work to access the backend HBase/Phoenix storage data with the defined EventIndex data model.

In the following sections, we discuss the Spark functionalities that are key for our work. Analytical tools for data discovery, duplicates and overlap calculations, and helper functions are also presented.

The referenced data model implemented on HBase/Phoenix and the data ingested that we presented on section 5.3.2 supported the development and evaluation of these data access analytical tools.

6.2.1 Spark

Spark is a distributed data processing framework that provides a series of abstractions and services, effectively acting as an analytics operating system.

One of the most important abstractions is the Spark DataFrame, which represents a set of data that might be physically residing on several physical nodes. This data abstraction allows us to apply and chain operations to produce other DataFrames that are maintained in memory, or that can be stored in the backend storage. Data is described in columns and rows, like traditional relational schemas; but new columns can be easily added or removed, and there are no constraints, index, restrictions on primary or foreign keys or triggers.

When connecting to our EventIndex schema and data mode, we use the Apache Phoenix Spark Plugin to load Phoenix tables as DataFrames.

As data is distributed, operations can be applied in parallel by Spark executors in several partitions and nodes. One important characteristic is that operations are applied lazily in memory, only when needed. Thus, optimizations are applied dynamically on chained operations called transformations. Actions, like collecting data in a particular node or writing in the backend storage, trigger the execution of the chained transformations and effectively materialize output data. Spark framework and the DataFrames are failure resilient, and the computations can be reapplied automatically in case of node failure.

6.2.2 Data discovery

As we have seen in the data model, the EventIndex data resides in a big table linked to the dataset and container tables by means of a constructed composite key.

We need a set of tools to quickly find event data using the meta data information at dataset and dataset container level. After locating data we could access the events tables with the provided Spark abstractions to solve the required use cases.

Data searching can be potentially done with any value on the dataset model (see subsection 5.3.1). These can be the identification fields (Project, runNumber, streamName, prodStep, dataType, version, tid). Access can also be done by any other field that represents summary data for every dataset and container, including:

- Total, unique, and duplicate number of events.
- Number of total files (guids) and number of files containing duplicates (as dataset or container granularity, depending what table accessed).
- Data Collection bookkeeping information: status of the dataset and date of updated information.
- Metadata about contained events: if this contains raw data, trigger, and provenance details.

We have produced the tools to look for data of interest that can be used later for further use cases. In particular, first entry functions are *findDatasets()* and *findCanonical()* (canonical dataset container), which access the related tables. They produce a Spark DataFrame representing a dataset or container with results that can be consulted and refined by Spark operations.

DataFrames are defined on the schema with named columns of the backend HBase/Phoenix tables aforementioned. The incarnation of the data is done with a lazy evaluation policy, so the results are only available when actions are called. Another advantage when modelling the data with Spark DataFrames is the possibility of applying Spark SQL functions, which allows the usage of SQL queries.

Some examples of the functionality are shown next.

In the figure 6.2 we show an interactive Spark shell session where we use our data discovery tools to find all containers available and count them by project.

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.4.8
      /_/

Using Scala version 2.11.12 (OpenJDK 64-Bit Server VM, Java 1.8.0_302)
Type in expressions to have them evaluated.
Type :help for more information.

scala> import eventindex.analytics.spark.Util._
import eventindex.analytics.spark.Util._

scala> val canonicalDF = findCanonical()
canonicalDF: org.apache.spark.sql.DataFrame = [RUNNO: int, PROJECT: string
 ... 24 more fields]

scala> canonicalDF.count
res0: Long = 1132


scala> canonicalDF.groupBy("PROJECT").count.show
+------------+-----+
|     PROJECT|count|
+------------+-----+
|   data15_hi|    1|
|  mc15_13TeV|   12|
|data16_13TeV|  108|
|data15_13TeV|   14|
|  mc16_13TeV|  259|
| data17_5TeV|    9|
|   data18_hi|    4|
|data17_13TeV|   25|
|data18_13TeV|  705|
+------------+-----+
```

Figure 6.2: Data discovery functions to find all canonical dataset containers and count them grouping by project.

We first import our tool with the `eventindex.analitycs.spark` package. With the *findCanonical()* function we will produce the Spark DataFrame that represents the canonical datasets table in the HBase/Phoenix backend. The first action with the *count* function will give us the number of entries which corresponds to 1,132 container datasets. Then we use the *groupBy()* spark transformation by the “PROJECT” named column, and then count and show how many datasets are available per project.

6.2.3 Duplicate calculation

There is a need to detect event duplication at different granularities, starting from files (GUID) containing duplicates, but also at a higher level so we

developed analytic tools to check for event duplicates.

Our tool provides a set of functions to detect duplicates at several granularities. It also expands the functionality of the Spark DataFrames containing events, providing custom transformation functions related to the calculation of duplicates.

The following functions are provided and can be used to calculate the missing values in the dataset and container tables. The function names match those of the fields in the data model (see subsection 5.3.1):

- *events()* : number of event records.
- *events_dup()* : number of events with duplication.
- *events_uniq()* : number of unique events (identifiers).
- *files()* : number of files (GUIDs) seen.
- *files_dup()* : number of files with duplicates.

In the following example we apply the previous functions to a particular *mc16_13TeV* Monte-Carlo canonical container dataset :

```
scala> val canonicalDF = findCanonical("mc16_13TeV.451926.
↪  MadGraphPythia8EvtGen_A14NNPDF23LO_X280tohh_bbtautau_hadhad.deriv.
↪  DAOD_HIGG4D3.e8353_e5984_a875_r9364_r9315_p3978")
canonicalDF: org.apache.spark.sql.DataFrame = [RUNNO: int, PROJECT:
↪  string ... 24 more fields]

scala> val eventsDF = canonicalDF.findEvents
eventsDF: org.apache.spark.sql.DataFrame = [DSPID: int, DSTYPEID:
↪  smallint ... 24 more fields]

scala> eventsDF.events
res0: BigInt = 128077

scala> eventsDF.events_dup
res1: BigInt = 27805

scala> eventsDF.events_uniq
res2: BigInt = 94256

scala> eventsDF.files
res3: BigInt = 5
```

```
scala> eventsDF.files_dup
res4: BigInt = 4
```

First we apply data discovery functions to represent this container within a DataFrame named *canonicalDF*.

Then we refine it into a new *eventsDF* that represents the events from that container. The subsequent functions are applied in that DataFrame to obtain the number of total events (128,077), the number of events with duplication (27,805), the number of unique event identifiers (94,256), the total number of files (5), and the number of files affected by duplicates (4).

These calculated values can be stored in the relevant fields of datasets and container tables by the user, or by an automated higher-level tool.

The same functions could be applied at any granularity as a DataFrame can contain a single event, events from a file (GUID), a dataset or a container.

Evaluation

A Spark application using these functions was implemented to be submitted automatically to the production system. Granularity can be selected at submission time with some parameters.

This application calculates all the aforementioned values: events, events_dup, events_uniq, files, files_dup.

In addition, it obtains the list of files (GUIDs) with duplicated event identifiers, and the number of that event records within that file.

The program produces a DataFrame that can be used to update the tables online, or be written in another backend to be later used. The output of this program is a summary JSON file with these variables, with the final object of filling the missing values in the datasets and canonical dataset container meta-tables.

We measured the performance of this application with a set of examples of a size in the order of 200 k, 1 M, 20 M, and 100 M event. These samples contain groups of datasets with up to 7 derivations (identified by its datatypes). We have chosen a representative set of examples that are known to contain duplicates, although we have also tested the base cases for samples when there are no duplicates.

The procedure measures only real data access and computation time, from the start of the main Scala process to the end of the calculations. It does not include the interactive or background setup times, in case the application is sent

to be executed in a YARN cluster. We submitted the same application for the same samples several times, to measure variability in cluster utilization. The cluster configuration makes it possible to distribute the load among 4 Spark executors initially, but can automatically scale up to 32 executors.

Results in figure 6.3 shows the duplication calculation rates of the application when submitted to the CERN cluster. The horizontal axis represents the size of the dataset in events in log scale. The vertical axis shows the rate in processed events per second (or Hz) in log scale.

In the figure we can see the mean rate for 200k event datasets is a rate of 1.5 k events/s. Observing the raw results in detail we observe that when applying to a 200 k dataset without any duplicates, the procedure time takes 150 seconds as a median (ranges from 117 to 216 s). When finding duplicates, a 200 k dataset takes 150 seconds as a median as well. Time ranges from 112 to 171 seconds, detecting from hundreds to 24 k duplicates in 54 files.

1 M event datasets take from 171 to 314 seconds (yields 3 to 6 k events/s). 20 M event datasets take from 347 to 564 seconds (yields 35 to 57 k events/s). In all of these samples, the number of detected duplicates and files containing duplicates varies without determining a clear weight on the resulting processing times. Samples of the same size with more duplicated events and number

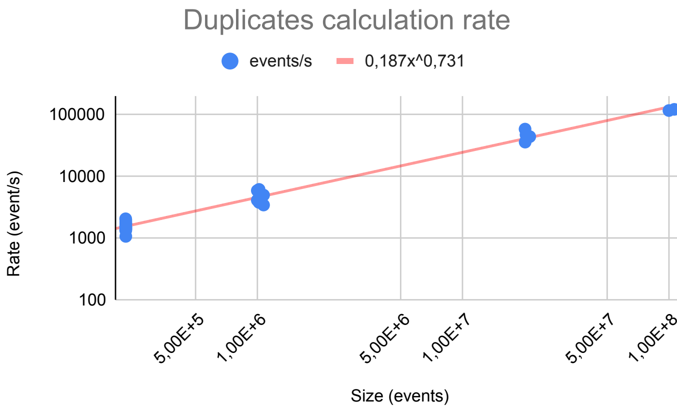


Figure 6.3: Duplicates calculation rate. The horizontal axis shows the size of the dataset in events; the vertical axis shows the rate in processed events per second (log scale).

of affected files can yield better results, making the size of the sample the determinant variable for predicting the processing time.

Bigger datasets with their derivations comprising about 100 M events take 869-884 seconds (yields 120k events/s). Again without much difference to the processing time attributable to the number of duplicates detected (8 to 23 million) and affected files (500 to 200 files).

The load is distributed initially among 4 executors initially for every submission, but the cluster configuration is configured with auto-scaling up to 32 executors.

It has to be noted that in the case of a high number of files affected by duplicates, the detailed list of GUIDs might occupy a non-negligible amount of space (530 MB JSON files with details of about 4M events duplicated on 422 files). Time to produce the output summary file is not shown here. In particular, in this sample it was about 60 seconds for collecting the output of the driver process and writing the file. This time can be decreased if the output is written directly by the Spark executors in a distributed manner in the meta tables, instead of collecting the output in the driver.

6.2.4 Helper functions

Our tool also provides some addition helper functions:

- *withColumnGUID()* : adds a new column to the DataFrame with decoded GUID file (from SelfReference field).
- *groupByGuid()* : groups the entries by file (GUID), effectively calculating the number of event entries per file.
- *groupByEventno()* : number of records per event number identifier.

In the next example listing, we observe that we can obtain the list of event ids (event number or eventno) and the number of entries of that event id in the dataframe (representing the same *mc16_13TeV* container from the example in subsection 6.2.3 :

```
scala> eventsDF.groupByEventno.show
+-------+-----+
|EVENTNO|count|
+-------+-----+
|   3506|    1|
|   5385|    1|
```

```
|   5409|    1|
|   7279|    1|
|   8440|    1|
|   8484|    1|
|   9233|    1|
|  11190|    1|
|  11619|    2|
|  12044|    2|
|  13248|    1|
|  13401|    1|
|  13638|    2|
|  14117|    2|
|  14719|    1|
|  15057|    1|
|  15322|    1|
|  15375|    2|
|  17043|    1|
|  18147|    2|
+-------+-----+
only showing top 20 rows
```

This example shows the first 20 rows, and we can observe that some event id has a count of 2, so they are duplicated events. We could refine even more this result DataFrame, for example, applying a *"where(count => 2)"* clause to obtain only the event ids that have duplicates.

In this next example we show the usage of *groupByGuid* function to show the files identified by GUID and the number of event entries that contain:

```
scala> eventsDF.groupByGuid.show
+--------------------+-----+
|                GUID|count|
+--------------------+-----+
|F539A8C4-9D16-974...|30406|
|99F2E92C-048F-DC4...|30285|
|416D28E9-0DC8-0C4...| 6390|
|7ABEE196-3CCE-964...|30395|
|4821F1F3-F214-994...|30601|
+--------------------+-----+
```

Function *groupByGuid()* internally uses *withColumnGUID()* to decode the GUID fields that are not directly available in the backend tables, and have to be decoded from the *sr* (self-reference) field. This latter function is also publicly open for users, as it is convenient for decoding and accessing the GUIDs.

Other fields can be accessed directly from the DataFrame with the column name. Trigger is encoded with fields in the events table as an Array of shorts (see families C and D in the events table in subsection 5.3.1). The Apache Phoenix Spark connector was incorrectly interpreting this data type, making access impossible from Spark and therefore from our tool. Modifications to the Phoenix source code and integration tests were developed to solve this issue in the Phoenix Spark connector, and these were submitted to the Apache Phoenix project and finally adopted [97]. With this we managed to correctly access these types of fields from Spark, which was not possible before.

6.2.5 Overlaps calculation

Calculation of event overlap matrices among the derivations of a dataset is one the use cases of the ATLAS EventIndex. This is because an event is reprocessed and stored in several formats (several output files over time). In particular, for the derivation framework, currently there are n streams being produced which will be spread among several trains (processing jobs) and will end in n files. Therefore 1 input file, n output files: The event overlap between these needs to be monitored. We wish to determine how many and which events end up in each stream, for a number of datasets.

The provided function *calculateOverlaps()* calculates the values needed to build the matrix for all derivations of a given dataset with its identifier.

In the following example we will calculate the overlaps for all the derived datasets from the following original AOD dataset:

```
data18_13TeV.00350144.physics_Main.merge.AOD.f933_m1960
```

For reference the DAOD (Derived AOD) datasets to apply the algorithm have the following names:

```
data18_13TeV.00350144.physics_Main.deriv.DAOD_BPHY1.f933_m1960_p3553
data18_13TeV.00350144.physics_Main.deriv.DAOD_BPHY4.f933_m1960_p3553
data18_13TeV.00350144.physics_Main.deriv.DAOD_BPHY5.f933_m1960_p3553
...
```

They comprise 83 datasets summing up around 500 M events.

Figure 6.4 shows the results of our algorithm in the form of a Spark DataFrame, which can be shown in the spark-shell as in the figure.

The first two columns *stream1, stream2* represent the pairs of derived streams that are considered in every row. Their names correspond to the *dataTypeGroup* that are part of the *dataType* of the dataset. For example, the

stream1	stream2	events_stream1_only	events_stream2_only	events_bothstreams	ratio
EXOT2	TAUP1	13349380	3320739	99177	0.005914201764939924
JETM3	TOPQ5	816220	2042208	103154	0.03483070872256787
HIGG8D1	JETM9	7234206	12877697	886261	0.04220659482419511
FTAG3	SUSY4	194182	5875417	39379	0.006446086399394465
EGAM3	EXOT5	154333	4893179	76331	0.014897216796064984
BPHY5	JETM10	304176	239310	423	7.777036232163836E-4
JETM1	TOPQ5	14218507	1832525	312837	0.019117544878903638
HIGG4D6	JETM3	1576903	916560	2814	0.001127278743504...
EXOT15	EXOT22	1944729	11487201	117684	0.008685413473771282
BPHY1	FTAG2	2822458	11442482	1294851	0.08321776301494024
SUSY3	TOPQ5	8139199	1719860	425502	0.04137288893517185
EXOT17	SUSY5	765866	9520498	1001714	0.08874088219447102
HIGG6D2	SUSY3	14075689	3296962	5267739	0.23266997609140125
HIGG2D5	MUON1	44994	10110601	13540	0.001331480012803449
EXOT8	SUSY4	7519121	3686039	2228757	0.1659052233239196
BPHY4	SUSY4	1764120	5477059	437737	0.05700505123379394
HIGG1D1	SUSY6	805433	13103610	335273	0.023537318324024826
EXOT6	TOPQ5	1784015	2122378	22984	0.005849273307193481
EXOT0	SUSY11	1468703	2636598	13308	0.003231188005464...
FTAG3	MUON2	231798	439645	1763	0.002618812072382...

only showing top 20 rows

Figure 6.4: Overlap event calculation for a set of 83 datasets with 500 M events.

corresponding stream from DAOD_BPHY1, would be named BPHY1. Next column *events_stream1_only* contains the number of event records only found in stream1 and therefore unique. Similarly with stream2 in the next column *events_stream2_only*. Column *events_bothstreams* contain the number of records found in both streams, therefore overlapped events or the intersection. Last column *ratio* is calculated as the intersection (events_bothstreams) divided by the union (events_stream1_only + events_stream2_only + events_bothstreams).

The output DataFrame with these results contains an entry for every pair of derived streams that contain overlapped events. Therefore for S derived streams we would have $S \times S$ entries (representing a matrix), but we have to bear in mind that the events overlapping in a pair of streams (i,j) will hold the same results as the pair of streams (j,i), so we will have a symmetric matrix. In addition, the elements of the leading diagonal (i,i) that contains the values of one stream against itself will be always the same (ratio=1 as all events are by definition the same events_stream1_only = events_stream2_only = 0, and events_bothstreams will equal the total number of events in the stream). Therefore instead of $S \times S$ elements, we will have to explicitly calculate only the $\frac{S(S-1)}{2}$ elements in the upper right (or lower left), which are the independent entries of the matrix.

In this example, the results on the screen show the first 20 rows or entries of the result overlapsDF DataFrame, which in fact contains $\frac{83(83-1)}{2} = 3,403$

entries.

As an example, if we take the first entry, the (EXOT2, TAUP1) streams will produce the same results as (TAUP1, EXOT2), as the overlapped events in both streams are the same, and so the rest of the values. In this case, we see that there are 13,349,380 events that are only in EXOT2 (events_stream1_only), and 3,320,739 events in TAUP1 (events_stream2_only). Then there are 99,177 events that are in both streams, so this is the number of overlapped events or the intersection of both streams. We calculate the ratio (0.005914...) which represents the events that are in both streams, divided by the sum of total events (intersection over the union).

This DataFrame can as well be stored in an output file, or in another Phoenix Table, like in the following example. We have stored the results in another new table *DATASETS_OVERLAPS* with the same schema as the overlaps DataFrame and with an identifier of the calculation.

We have previously seen in our data model that millions of events reside in a big events table, with a row per event entry. We will apply the algorithm only to the needed data, namely, the event entries stored for every derived dataset that we are taking into account.

Algorithm

The overlap calculation algorithm has 4 main steps:

1. For every event record, select the event identifier (eventnumber), and the stream (datatype). The result of this step is a set of (EventId, Stream). This set might contain several entries with the same EventId.

2. Group the streams by the event identifier. The result of this step is a Set of (EventId, EventStreams), where EventStreams is a set (Stream1, Stream2, ... StreamN). The number of elements of this set of streams corresponds to the number of event entries of a particular event identifier.

3. For every event identifier, build all pairs of streams (i,j) that might contain this particular event, signaling where it is found. The result of this step is a set of tuples [(StreamI, StreamJ) , (is_in_I, is_in_J)], where is_in_X is a boolean that signals that this event entry is found in that stream. Values emitted might be:

    - (false, false) : not found in any, so this value is not emitted at all and will not be found in the set of tuples.

- (true, false) : the event entry is found in StreamI, but not in StreamJ.
- (false, true) : the event entry is found in StreamJ, but not in StreamI.
- (true, true) : the event entry is found both in StreamI and StreamJ, so this is an overlap.

It has to be noted that we have to build pairs of streams not only from the EventStreams set in step 2 (as will contain only the overlaps), but to travel all possible pairs of streams (i,j). With s streams, this means as much as (s*(s-1))/2 entries. This counts the events that might be in one but not in the other stream. As stated, if both i and j streams are not found in the EventStreams set considered in this step, then the (false,false) value is not emitted, reducing the potential (s*(s-1))/2 values emitted per entry generated from the previous step.

4. Group the tuples by pairs of streams, counting the number of previously generated values. Result is a set of Tuples [(StreamI, StreamJ) , (events_stream1_only, events_stream2_only, events_bothstreams, ratio)] The set contains an entry per (StreamI, StreamJ) pair possibility, with as many as (s*(s-1))/2 entries. When grouping by pair (StreamI, StreamJ), the values (is_in_I, is_in_J) previously emitted are counted in the mentioned variables:

   - (true, false) : sum 1 to events_stream1_only.
   - (false, true) : sum 1 to events_stream2_only.
   - (true, true) : sum 1 to events_both_streams.

   ratio is calculated as the intersection over the union, so

$$ratio = \frac{events_bothstreams}{(events_stream1_only + events_stream2_only + events_bothstreams)}$$

The result of Step 4 is what we find in the output of the example shown before, the set of unique entries of the matrix that represents the possible $N \times N$ overlaps of the N derived streams of the analyzed dataset.

Implementation

The implementation of the algorithm was done in Scala language and using Spark abstractions and functions.

First, we are reading the events from the source (Events HBase/Phoenix table in subsection 5.3.1), which are from the streams of interest, in this case all streams (dstypeid) found in the canonical container table for a particular dataset by its dataset identifier (dspid).

Since all data share the dspid which is the key prefix, we assure the locality of the data. In addition, all events from a stream datatype will share the dstypeid, that is, the next data in the key prefix, so they will be together in disk and not spread, assuring also the locality of the data.

Step 1 of the algorithm is achieved with a `map()` transformation, retrieving only the fields of interest of every event entry, namely, the event identifier (eventnumber or eventno), and the stream (dstypeid).

In Step 2, we apply the `aggregateByKey()` transformation which is much more efficient as it can be applied in parallel in different partitions where the data is, instead of moving the data as in `groupByKey()`. We want the result of the aggregation to be a set of values, which is a different type to the values that are strings (the sum of strings is a concatenation of the string), so we use this function instead of `reduceByKey()`.

The backend data is organised into all streams (dstypeid) of a dataset (dspid) to be consecutive in the HBase row key space, and therefore in the storage (disks). We can therefore benefit from most of these calculations being done in the same machine and in memory in particular, without too much data shuffling across the Region Servers of the cluster. It is however possible that for big datasets and lots of derivation streams, their data expands along several Region Servers. In this case, the `aggregateByKey()` transformation will shuffle the data to aggregate by the event identifier.

In Step 3, we use the `flatMap()` transformation to the DataFrame result of step 2, which has an entry per event identifier, into the set of tuples that identify pairs of streams, and where in that pair the event entry is found (as boolean pairs described in the algorithm). This `flatMap()` transformation applies a custom eventsStreamsPairMapper() function to every original entry (event identifier) for that purpose.

The last Step 4 applies again an `aggregateByKey()` transformation to reduce the previous results by pairs of streams. The first parameter of `aggregateByKey()` will be the combiner function for merging values within a partition, taking the boolean pairs, and converting them to tuples (events_stream1_only, events_stream2_only, events_both_streams, ratio). Ratio is not computed in this combiner function, yet is emitted as 0. Yet the summing of the events are done at the partition level. The second parameter of `aggregateByKey()` will be the reducer function, grouping a pair of tuples produced by the previous

function, and applied when merging values between partitions. In this case, the output is another tuple, summing the 3 first values, and computing the ratio as the intersection over the union (events_both_streams / (events_stream1_only + events_stream2_only + events_both_streams)). It has to be noted that this reducer function is not applied when all data is in a single partition, so the ratio will not be computed correctly. In this case, there will be another step to explicitly compute the ratio when producing the final results.

The last part of the implementation deals with showing user-friendly values in the result DataFrame, which implies converting the dstypeid to the user-friendly stream names stored in the dstypes tables.

Evaluation

We have tested the algorithm on several datasets and derivation streams. The most common data currently has few stream derivations ($s < 10$), with some up to the order of 100 (s) streams. Datasets of size n range from thousands to millions of events.

The sizes of the problem (n) of the dataset the samples tested are: 200 k, 1 M, 20 M, 100 M, and 500 M events. These events are divided into a number (s) of streams, which is: 1, 2, 5, 6, 7, and 83 (unique case).

Figure 6.5 shows the overlap calculation time depending on the size of the dataset on the x-axis and the number of streams (in lines with different colours).

For dataset samples up to 200 k events, the overlap procedure took from 15 seconds for a dataset with just 1 stream. Therefore this is the baseline as no matrix elements are calculated. It takes 27 seconds for 2 streams, and 40 seconds for 6 streams.

For dataset samples of 1 M events it takes from 43 to 67 seconds. A 6 stream sample takes 43 seconds , the same as a 200k dataset, so in this case this might be an issue in the former calculations.

The number of computations performed is $n\frac{s(s-1)}{2}$, where n is the number of events, and s is the number of streams. As we can see also in the figure 6.5, the cost is dominated by the size of the problem n, while the cost of computing the $\frac{s(s-1)}{2}$ pairs per entry takes less time compared to the n term. The dominant term is n, so the temporal cost is $\mathcal{O}(n)$. The intermediate data produced is at most the number of computations $n\frac{s(s-1)}{2}$ in Step 3 of the algorithm presented. However, this data is constantly reduced per spark partition at Step 4. Thus, the final space cost is $\mathcal{O}(s^2)$.

Figure 6.6 shows the overlap calculation rate. For datasets of 20 M events, processing takes 368 to 483 seconds and yields a processing rate of about 50 k

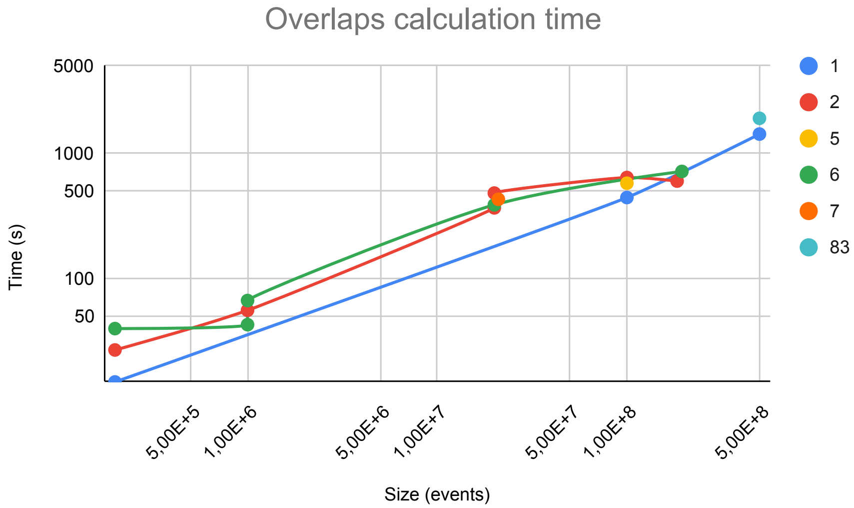


Figure 6.5: Overlaps calculation time. Every line represents the number of streams (s) of the problem. Horizontal axis represents the size of the problem in events, in logarithmic scale; the vertical axis represents the time in seconds (log scale).

events/s. 100 M event baseline dataset with just 1 stream takes 445 seconds and yields a performance of 240 k events/s processed. Then datasets with more streams and overlap processing takes 579 to 643 seconds, with a mean 160 k events/s processed. The biggest 500 million events sample yields a performance of 380 k events/s.

Datasets with higher number of streams might yield better results, revealing again the preponderance of the size (number of events) factor.

6.3 Conclusions

We have presented the contributions on data access and analytics on big data applications like the EventIndex. We have developed tools and algorithms for data access, duplicate and overlap detection over big amounts of data. We have shown the usage over the HBase/Phoenix data storage proposed on the

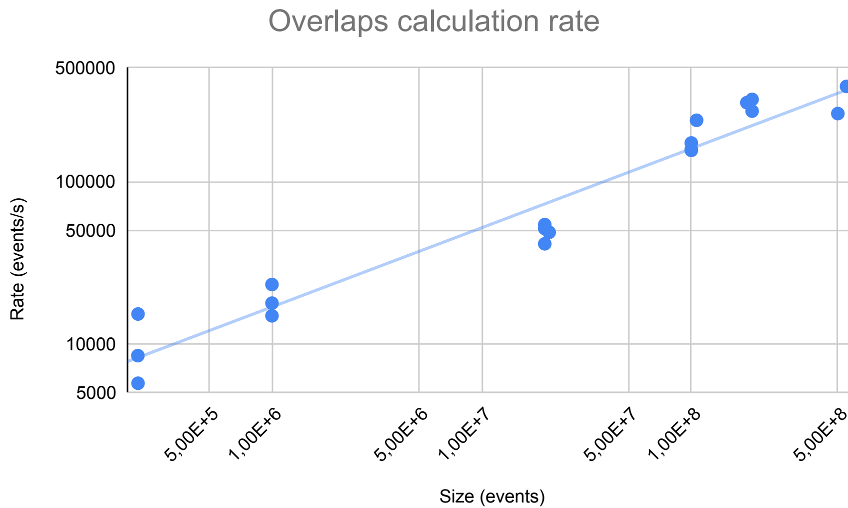


Figure 6.6: Overlaps calculation rate. Horizontal axis represents the size of the dataset in events; the vertical axis represents the rate of events processed per second (log scale).

previous section. With the new contributed interactive capabilities the users have new data access paths not available before.

We have presented an analytics platform based on Spark and a set of analytical tools using Spark abstractions and implemented in a Scala package (`eventindex.analitycs.spark`). With this, we can access the billions (10^9) of event records of the EventIndex project stored in the HBase data backend. An Apache Phoenix layer provides schema enforcement and SQL capabilities to access the data.

With our tools, we abstract the backend data model, decoupling the data access from the actual data schema and used technologies. This approach is very convenient to hide model complexity with an accessible defined interface. It also masks changes in the data model, which are invisible to the user as the defined interfaces have not changed.

The package can be used interactively within a command line spark-shell session. It also can be used with batch standalone Spark jobs, as we have shown

when evaluating our tool algorithms.

We have previously shown in section 5.3 that our data model is defined in a big events table with data organised in 4 families, and other meta-tables for data discovery and bookkeeping. For the evaluation of our data access tools, we have used the 21.95 TB dataset samples of EventIndex production data that were imported into our HBase/Phoenix system (see subsection 5.3.2).

The tool and algorithms presented solve data access for our application use cases in areas like data discovery, duplicate detection, and overlap calculation. Data discovery capabilities produce Spark DataFrames usable by the rest of the tools. They also use custom helper functions to access the encoded fields of the data model. During the process of development, some limitations in the Apache Phoenix Spark connector were solved and contributed back to the community.

An overlap calculation algorithm was presented with computational cost $\mathcal{O}(n)$ with the number of events and spatial cost $\mathcal{O}(s^2)$ with the number of streams. Implemented in Scala and using Spark abstractions, it translates automatically to scans over the HBase key, which are fast and performant. All data for the derivation of a dataset are adjacent in the key space, and therefore storage, reducing input/output operations. The algorithm scales automatically within the Spark cluster up to 32 processes, yielding a performance of 380 k events processed per second for a 500 M event dataset.

The duplication detection case accesses the event table key and family A (event location) data. It is slower compared with the overlap case due to the accessing of different values in a data family and not only the HBase key. It yields a performance of 120 k events per second for 100 M events, compared with 150 k events/second for the same size overlap calculation.

The processing rates show a penalization for smaller datasets that are due to the set-up of HBase data streams, dominating the accounted time. We obtain better rates for bigger dataset sizes, so one possibility to apply in the future is to increment spark job granularity when possible, for example, calculating features at the container level, instead of its constituent datasets.

The presented framework approach solves the analytic use cases of the ATLAS EventIndex project in a performant manner, providing convenient data access paths which will be exploited starting with the LHC Run 3 (2022-2025).

7 Conclusions

In this last chapter we summarize the contributions and conclusions of this work. A list of publications related with the work in this thesis and within the ATLAS EventIndex project is included.

7.1 Contributions

Several contributions have been made in the EventIndex project in the areas of distributed data collection, big data storage backends and data access and analytics.

We have shown the improvements on the distributed data collection system on large distributed infrastructures like the grid. The previous messaging data collection system was found complex and limited the scalability for our EventIndex payloads. We contributed to a new pull model design with an object store for temporary data staging and dynamic data selection for data ingestion. Results show that the new design improves the performance in several areas. With the new design we can avoid payload segmentation, storing the index information from each metadata extraction process in a single object. We achieve better data compression ratios with larger payloads and binary data encoding. Thus a factor 4.5 reduction in the total volume of conveyed and ingested data is achieved. We have got rid of blockings with the new object store implementation and the workload distribution is improved, potentially achieving better scalability. Throughput during data ingestion is improved 15 times compared to the messaging approach. In addition, the pull model approach allows the dynamic selection of data, avoiding the ingestion of duplicated information that in our experiment is 10% of the produced data. The pull model object store implementation was deployed in production already during Run 2 (2015-2018) and has shown excellent performance in production and is able to scale to the required rates of the next Runs.

Regarding data storage, HBase was selected because of its good performance and better support in production. The flexibility of our architecture allows us to change the data backend storage while maintaining the pull-model distributed data collection approach. With HBase as the main and unique storage for all our data we simplify storage, avoiding duplicating data with complex procedures, potential data coherence issues and reducing total volume of data. In this manner we move from a hybrid system that maintained data in HDFS and Oracle, to a unique system based on HBase. With a Phoenix layer over HBase we allow schema enforcement with defined types, also providing SQL capabilities for later data access, which was not available before. In addition we assure the schema application with defined types, which was not available with HDFS, that led to improvements applied at column level reducing total volume of data. Data augmentation procedures are not needed anymore as all the procedures can be done online during data ingestion. Regarding performance and the data model schema parameters, we have shown that a pre-splitting on the events table and a salting distribution strategy can improve throughput during data ingestion. HBase/Phoenix includes built-in global ordered key space by design, avoiding data consolidation procedures for ordering data, as we had in HDFS. HBase also allows writing at the record level, so we can have multiple writers per dataset compared with only one in the previous approach and therefore improving the data ingestion performance. We have contributed to reduce the complexity on the storage system and the resource usage, optimizing the data volume used while improving the throughput on data ingestion. These improvements have increased the reliability and overall performance of the system required for the LHC Run 3 (2022-2025) and beyond.

With our contributions in the data access area, now is possible interactive data access and analysis, which was not possible with the previous model. We have improved analytical use cases with the EventIndex data stored in HBase/Phoenix, and accessing with a platform based on Spark, using its abstractions and a set of analytical tools implemented in Scala. The new tool and algorithms presented solve data access for our application use cases in areas like data discovery, duplicate detection, and overlap calculation among datasets, that are now integrated. Data discovery capabilities produce Spark DataFrames usable by the rest of the tools. In addition data and results might be maintained in cache, which was not possible before, allowing algorithm chaining and improving overall resource usage. The overlap calculation algorithm has computational cost $\mathcal{O}(n)$ with the number of events, and spatial cost $\mathcal{O}(s^2)$ with the number of streams. Its performance is improved as it only has to access the key of the event, instead of all the event register like in HDFS. In

addition the columnar organization allows to reduce the number of input/output operations, improving the performance. With our tools, we abstract the backend data model, decoupling the data access from the actual data schema and used technologies. This approach is very convenient to hide model complexity with an accessible defined interface for tools and users. It also masks possible changes in the data model, which are invisible for the user as the defined interfaces do not change. The presented framework approach solves the analytic use cases of the ATLAS EventIndex project in a performant manner, providing convenient data access paths which were not available before and which will be exploited starting with the LHC Run 3 (2022-2025).

7.2 Publications

The contributions described in this work have resulted in the following publications:

- D Barberis, J Cranshaw, G Dimitrov, A Favareto, Á Fernández Casaní, S González de la Hoz, J Hřivnáč, D Malon, M Nowak, J Salt Cairols, J Sánchez, R Sorokoletov, and Q Zhang and. The ATLAS Eventindex: an event catalogue for experiments collecting large amounts of data. In: *Journal of Physics: Conference Series* 513.4 (June 2014), p. 042002. DOI: 10.1088/1742-6596/513/4/042002
- D. Barberis, S.E. Cárdenas Zárate, J. Cranshaw, A. Favareto, Á. Fernández Casaní, E.J. Gallas, C. Glasman, S. González de la Hoz, J. Hřivnáč, D. Malon, F. Prokoshin, J. Salt Cairols, J. Sánchez, R. Többicke, and R. Yuan. The ATLAS EventIndex: architecture, design choices, deployment and first operation experience. In: *Journal of Physics: Conference Series* 664.4 (Dec. 2015), p. 042003. DOI: 10.1088/1742-6596/664/4/042003
- J Sánchez, A Fernández Casaní, and S González de la Hoz. Distributed Data Collection for the ATLAS EventIndex. In: *Journal of Physics: Conference Series* 664.4 (Dec. 2015), p. 042046. DOI: 10.1088/1742-6596/664/4/042046
- D. Barberis, J. Cranshaw, A. Favareto, A. Fernández Casaní, E. Gallas, S. González de la Hoz, J. Hřivnáč, D. Malon, M. Nowak, F. Prokoshin, J. Salt, J. Sánchez Martínez, R. Többicke, and R. Yuan. The ATLAS EventIndex: Full chain deployment and first operation. In: *Nuclear and Particle Physics Proceedings* 273-275 (Apr. 2016). Q3. Corresponding

author A. Fernández Casaní, pp. 913–918. DOI: 10.1016/j.nuclphysbps.2015.09.141

- D. Barberis, S.E. Cárdenas Zárate, A. Favareto, A. Fernandez Casani, E.J. Gallas, C. Garcia Montoro, S. Gonzalez de la Hoz, J. Hrivnac, D. Malon, F. Prokoshin, J. Salt, J. Sanchez, R. Toebbicke, and R. Yuan and. ATLAS Eventlndex monitoring system using the Kibana analytics and visualization platform. In: *Journal of Physics: Conference Series* 762 (Oct. 2016), p. 012004. DOI: 10.1088/1742-6596/762/1/012004

- A Fernandez Casani, D Barberis, A Favareto, C Garcia Montoro, S González de la Hoz, J Hřivnáč, F Prokoshin, J Salt, J Sanchez, Többicke, R Yuan, and ATLAS Collaboration. ATLAS EventIndex general dataflow and monitoring infrastructure. In: *Journal of Physics: Conference Series* 898.6 (2017), p. 062010. DOI: 10.1088/1742-6596/898/6/062010

- Álvaro Fernández Casaní, Juan Orduña, and Santiago González de la Hoz. Performance Improvements of an Event Index Distributed System. In: *ICPP 2018: Proceedings of the 47th International Conference on Parallel Processing*. Extended abstract. Eugene, OR, USA: Association for Computing Machinery, 2018. ISBN: 9781450365109. URL: http://oaciss.uoregon.edu/icpp18/publications/pos110s2-file1.pdf

- Zbigniew Baranowski, Luca Canali, Alvaro Fernandez Casani, Elizabeth J Gallas, Carlos Garcia Montoro, Santiago González de la Hoz, Julius Hrivnac, Fedor Prokoshin, Grigori Rybkine, Jose Salt, Javier Sanchez, and Dario Barberis. A prototype for the evolution of ATLAS EventIndex based on Apache Kudu storage. In: *EPJ Web of Conferences* 214 (2019). Ed. by A. Forti, L. Betev, M. Litmaath, O. Smirnova, and P. Hristov, p. 04057. DOI: 10.1051/epjconf/201921404057

- Álvaro Fernández Casaní, Dario Barberis, Javier Sánchez, Carlos García Montoro, Santiago González de la Hoz, and José Salt. Distributed Data Collection for the Next Generation ATLAS EventIndex Project. In: *EPJ Web of Conferences* 214 (2019), p. 04010. DOI: 10.1051/epjconf/201921404010

- Santiago González de la Hoz, Carlos Acosta-Silva, Javier Aparisi Pozo, Manuel Delfino, Jose del Peso, Álvaro Fernández Casani, José Flix Molina, Esteban Fullana Torregrosa, Carlos García Montoro, Julio Lozano Bahilo, Almudena del Rocio Montiel, Andreu Pacheco Pages, Javier Sánchez

Martínez, José Salt, and Aresh Vedaee. Spanish ATLAS Tier-1 & Tier-2 perspective on computing over the next years. In: *EPJ Web of Conferences* 214 (2019). Ed. by A. Forti, L. Betev, M. Litmaath, O. Smirnova, and P. Hristov, p. 03013. DOI: 10.1051/epjconf/201921403013

- González de la Hoz, Santiago, Acosta-Silva, Carles, Aparisi Pozo, Javier, del Peso, Jose, Fernández Casani, Álvaro, Flix Molina, José, Fullana Torregrosa, Esteban, García Montoro, Carlos, Lozano Bahilo, Julio, Montiel, Almudena, Pacheco Pages, Andrés, Sánchez Martínez, Javier, Salt Cairols, José, and Vedaee, Aresh. Computing activities at the Spanish Tier-1 and Tier-2s for the ATLAS experiment towards the LHC Run3 and High-Luminosity periods. In: *EPJ Web of Conferences* 245 (2020). Ed. by C. Doglioni, D. Kim, G.A. Stewart, L. Silvestris, P. Jackson, and W. Kamleh, p. 07027. DOI: 10.1051/epjconf/202024507027

- M Villaplana Perez, E Alexandrov, I Aleksandrov, Z Baranowski, D Barberis, G Dimitrov, A Fernandez Casani, E Gallas, C Garcia Montoro, S Gonzalez de la Hoz, J Hrivnac, I Alexander, A Kazymov, M Mineev, F Prokoshin, G Rybkin, J Sanchez, J Salt, and P T Vasileva. The ATLAS EventIndex and its evolution towards Run 3. In: *Journal of Physics: Conference Series* 1525.1 (Apr. 2020), p. 012056. DOI: 10.1088/1742-6596/1525/1/012056

- Álvaro Fernández Casaní, Juan M. Orduña, Javier Sánchez, and Santiago González de la Hoz. A Reliable Large Distributed Object Store Based Platform for Collecting Event Metadata. In: *Journal of Grid Computing* 19.3 (Aug. 2021). Q1, p. 39. ISSN: 1572-9184. DOI: 10.1007/s10723-021-09580-0

- Elizaveta Cherepanova, Evgeny Alexandrov, Igor Alexandrov, Dario Barberis, Luca Canali, Alvaro Fernandez Casani, Elizabeth Gallas, Carlos Garcia Montoro, Santiago Gonzalez De La Hoz, Julius Hrivnac, Andrei Kazymov, Mikhail Mineev, Fedor Prokoshin, Grigori Rybkin, Francisco Javier Sanchez Martinez, Jose Salt, Miguel Villaplana, and Alexander Iakovlev. The ATLAS EventIndex Using the HBase/Phoenix Storage Solution. In: *9th International Conference on Distributed Computing and Grid Technologies in Science and Education.* 2021, pp. 17–25. DOI: 10.54546/mlit.2021.68.25.001

- Dario Barberis, Igor Aleksandrov, Evgeny Alexandrov, Zbigniew Baranowski, Luca Canali, Elizaveta Cherepanova, Gancho Dimitrov, Andrea

Favareto, Alvaro Fernandez Casani, Elizabeth J. Gallas, Carlos Garcia Montoro, Santiago Gonzalez de la Hoz, Julius Hrivnac, Alexander Iakovlev, Andrei Kazymov, Mikhail Mineev, Fedor Prokoshin, Grigori Rybkin, Jose Salt, Javier Sanchez, Roman Sorokoletov, Rainer Toebbicke, Petya Vasileva, Miguel Villaplana Perez, and Ruijun Yuan. *The ATLAS EventIndex: a BigData catalogue for all ATLAS experiment events.* For publication in Computing and Software for Big Science. Q1. 2022. DOI: 10.48550/ARXIV.2211.08293

- Álvaro Fernández Casaní, Carlos García Montoro, Santiago González de la Hoz, Jose Salt, Javier Sánchez, and Miguel Villaplana Pérez. Big Data analytics for the ATLAS EventIndex project with Apache Spark. In: *[Manuscript submitted for publication] Computational and Mathematical Methods* (2022). Presented at 2022 International CMMSE conference and the Second conference on high performance computing (CHPC). Awarded "Best computational applications on line presentation". ISSN: 2577-7408

Resumen

El trabajo de esta tesis se enmarca dentro del proyecto EventIndex del experimento ATLAS, un gran detector de partículas del LHC (Gran Colisionador de Hadrones) en el CERN. El objetivo del proyecto es catalogar todas las colisiones de partículas, o eventos, registrados en el detector ATLAS y también simulados a lo largo de sus años de funcionamiento. Con este catálogo se pueden caracterizar los datos a nivel de evento para su búsqueda y localización por parte de los usuarios finales. También se pueden realizar comprobaciones en la cadena de registro y reprocesado de los datos, para comprobar su corrección y optimizar futuros procesos. Debido al incremento en las tasas y volumen de datos esperados en el Run 3 (2022-2025) y el HL-LHC (finales de la década del 2020), se requiere un sistema escalable y que simplifique implementaciones anteriores.

En esta tesis se presentan las contribuciones al proyecto en las áreas de recolección de datos distribuida, almacenamiento de cantidades masivas de datos y acceso a los mismos. Una pequeña cantidad de información (metadatos) por evento es indexada en el CERN (Tier-0), y de forma distribuida en el grid en todos los centros de computación que forman parte del experimento ATLAS (10 Tier-1, y del orden de 70 Tier-2). En esta tesis se presenta un nuevo modelo de recolección de datos en el grid basado en un *object store* como almacenamiento temporal, y con selección dinámica de datos para su ingestión en el almacén de datos final. También se presentan las contribuciones a una nueva solución en un único y gran almacén de datos basado en tecnologías de macrodatos (*Big Data*) como HBase/Phoenix, capaz de sostener las tasas y volumen de ingestión de datos requeridos, y que simplifica y soluciona los problemas de las anteriores soluciones híbridas. Finalmente, se presenta un marco de computación y herramientas basadas en Spark para el acceso a los datos y la resolución de cargas de trabajo analíticas que acceden a grandes cantidades de datos, como el cálculo del solapado (*overlaps*) entre eventos de distintos *datasets*, o el cálculo

de eventos duplicados.

Introducción

CERN, LHC y el experimento ATLAS

El Gran Colisionador de Hadrones (LHC) [1] es la máquina más grande construida, y el colisionador de partículas de mayor energía, diseñado para correr a una energía máxima en el centro de masas de 14 TeV. Está situado en el CERN, la Organización Europea para la Investigación Nuclear, en la frontera entre Suiza y Francia. Se localiza en un túnel circular con una longitud total de 26.7 km, y enterrado bajo tierra a una media de 100 metros de profundidad. Partículas con carga (protones e iones pesados, es decir hadrones) son aceleradas en haces dentro de dos anillos en sentidos opuestos, hasta que alcanzan la energía deseada.

Cuando los haces de partículas que han sido acelerados alcanzan la energía y condiciones deseables se hacen colisionar en puntos determinados del LHC. Hay 4 enormes cavernas bajo tierra que alojan los detectores de los experimentos para grabar los detalles de las colisiones que son producidas a una tasa de 40 MHz en cruce de paquetes partículas (*bunch-crossing*), o cada 25 ns. Las señales dejadas por las partículas producidas por estas interacciones son grabadas por los detectores como un evento de un *bunch-crossing* determinado.

El detector ATLAS [5] es un gran detector multi-propósito, en la forma de un gran cilindro de 46 metros de alto, 25 metros de largo y 7,000 toneladas de peso, situado en la caverna del punto 1 del LHC. Como se puede ver en la figura 1, el detector está compuesto de varios subsistemas dispuestos en forma de capas concéntricas al punto de interacción de los haces del acelerador. Estos subsistemas son el detector interno, los calorímetros electromagnéticos y hadrónicos, el sistema de imanes y el espectrómetro de muones. Cada uno de ellos está diseñado con un único objetivo, detectar tipos específicos de partículas o medir características individuales como la trayectoria, la energía o el momento.

Trigger y Sistema de Adquisición de Datos

Las tasas de 40 Mhz en el cruce de paquetes partículas (*bunch-crossing*) y la luminosidad afectan al número de interacciones entre partículas que pueden ser potencialmente detectadas. Grabar todas ellas significaría almacenar 60 TB/s, lo que es inmanejable. No todos los eventos son igualmente interesantes, así que el Sistema de *Trigger* (disparo) y Adquisición de datos (TDAQ) [15] se encarga

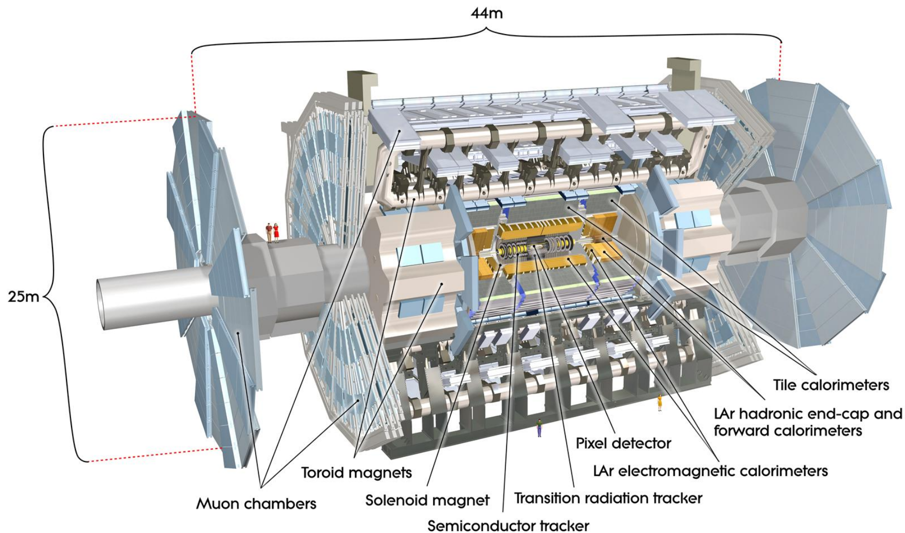


Figura 1: Imagen generada por ordenador del detector ATLAS con sus subsistemas detallados [6].

de la selección en línea de los eventos en función de los requerimientos de análisis de física establecidos en cada momento, definidos en lo que se llama el *trigger menu*. El TDAQ está compuesto de una estructura en niveles, donde la tasa de eventos se reduce secuencialmente a niveles aceptables. Durante el Run 1, el *trigger* estuvo originalmente compuesto por 3 niveles: L1, L2 y EventFilter (EF), pero empezando con el Run 2, el L2 y EF fueron combinados en el HLT (*trigger* de alto nivel). El primer nivel (L1) está implementado en *hardware* (FPGAs) y reduce la tasa de 40 MHz a 100 kHz, lo que significa bajar de los 60 TB/s originales a 160 GB/s aproximadamente. El HLT es un nivel implementado completamente en *software*, corriendo en una granja de unas 40,000 unidades de proceso. En este nivel se reduce a 1.5 kHz la tasa de eventos, lo que corresponde a 1.5 GB/s producidos con un tamaño de evento medio de 1 MB.

Computación distribuida

El experimento ATLAS sigue un modelo de computación distribuida para la compartición, reprocesado y análisis de los datos. La infraestructura de computación distribuida grid del LHC (*Wordwide LHC Computing Grid* o WLCG) [22] está formada por un conjunto de tecnologías y recursos de computación y almacenamiento en los centros que forman parte del experimento, organizados por niveles (*tiers*). El Tier-0 en el CERN mantiene todos los datos en crudo (*raw data*). Del orden de 10 centros Tier-1 distribuidos por regiones mantienen copias de estos datos y hacen reprocesados de los mismos. En el nivel inferior, del orden de 100 centros Tier-2/3 (70 para ATLAS) están dedicados al análisis de los datos y dar facilidades de acceso a los usuarios finales. Una visualización del mapa de los centros del WLCG puede verse en la figura 2. ATLAS actualmente usa en esta infraestructura distribuida más de 700 k CPU cores de computación, 230 PB de almacenamiento en disco y 270 PB de almacenamiento en cinta.

Figura 2: Centros del grid del LHC (WLCG) en septiembre de 2022. El Tier-0 está en el CERN (Ginebra); los centros Tier-1 están marcados con una estrella dentro de un icono de ubicación verde; los centros Tier-2/3 están marcados con un icono de ubicación azul [23].

Los datos principales de física son registrados desde el detector en flujos de

datos en crudo [21], pero también otros flujos son establecidos para la calibración, las comprobaciones rápidas de calidad (*express streams*), el depurado y otras tareas. En el Tier-0 los primeros flujos de datos de calibración y los flujos *express* son procesados para proveer variables de calibración y de alineamiento del detector. Luego, el grueso de datos de física es procesado en 24-48 horas con esta información dependiente del tiempo, para así producir eventos reales reconstruidos y sus propiedades y cantidades físicas. El resultado principal de esta fase de reconstrucción son los ficheros de tipo *Analysis Object Data* (AOD), que son guardados en el CERN y distribuidos a otros centros.

Es común realizar reprocesados de los datos cuando se tiene mejor conocimiento de las condiciones en las que se registraron los eventos (tras el cálculo mejorado de constantes de calibración y alineamiento). También, algoritmos nuevos y mejorados pueden ser producidos haciendo necesario el reprocesado de los datos. En este caso, nuevas versiones de los ficheros AOD son producidos en los centros Tier-1, para luego ser distribuidos.

Operación y retos

El LHC está en operación desde el año 2009, y alterna largos periodos de toma de datos (conocidos como *Runs*), con paradas programadas para su mantenimiento y actualización (conocidos como *Long Shutdowns*). Cuando empezó el Run 1 en 2011, la energía fue de 7 TeV en el centro de masas. Durante el Run 2 en 2015, la energía alcanzó los 13 TeV en el centro de masas, con una luminosidad integrada de 190 fb^{-1} (femtobarn inverso). El femtobarn inverso es una medida de colisiones de partículas por unidad de área (barn), lo que representa ambos el número de colisiones y la cantidad de datos recolectados. Un femtobarn inverso corresponde aproximadamente a 100 trillones (10^{12}) de colisiones protón-protón. Actualmente está en proceso el Run 3, que empezó en julio de 2022 con una energía de colisión en el centro de masas de 13.6 TeV, la más alta alcanzada en un acelerador de partículas. Se recolectarán del orden de aproximadamente 450 fb^{-1} cuando acabe el Run 3 en 2025. Está planeado que el HL-LHC (LHC de Alta Luminosidad) [29] empiece al final de la década, con un récord sin precedente en la toma de datos del orden de 7 a 10 veces más que las actuales. Con un incremento en la luminosidad también se incrementará el apilado de colisiones (*pile-up*) o número de colisiones simultáneas, desde el actual 30-60 hasta un futuro 200, incrementando el volumen total de datos.

Proyecto EventIndex

Los físicos usualmente trabajan sobre grandes cantidades de datos, pero también necesitan el acceso a eventos individuales para comprobar sus detalles. Esto se hace por ejemplo, durante las fases de reconstrucción o para producir visualizaciones de eventos. Como hemos visto, los eventos pueden ser reconstruidos con diferentes condiciones y algoritmos produciendo en la práctica múltiples versiones de los mismos. Los procesos de producción crean registros de los eventos con distina información en tipos diferentes de ficheros. Por lo tanto, es muy útil tener información sobre el historial o linaje de un evento. El acceso a nivel de evento es conveniente para comprobaciones de la calidad de la producción. De esta manera se pueden detectar eventos duplicados que podrían ser producidos debido a fallos temporales del sistema de adquisición o derivados de los procesos de reconstrucción. También el cálculo de solapes de eventos entre diferentes ficheros producidos por los procesos de derivación es muy útil, para optimizar futuras derivaciones con menos solapes y por lo tanto menos recursos gastados. Estudios en la correlación del *trigger* y solapes del mismo entre flujos de datos también pueden ser realizados a nivel de evento si los datos están disponibles.

En el grid los eventos se guardan en ficheros de distintos formatos, identificados por un *GUID* (identificador único global). Los ficheros se agrupan lógicamente en conjuntos de datos (*datasets*), y éstos se agrupan en contenedores. Una pequeña cantidad de información (metadatos) por evento se debe extraer de forma distribuida en el grid, donde residen los datos en ficheros en los diferentes centros. Estos metadatos extraidos deben ser transportados a un catálogo central en el CERN, donde estarán disponibles para resolver los distintos casos de uso planteados.

Objetivos

Los objetivos que aborda esta tesis están relacionados con las mejoras requeridas para indexar y catalogar los billones de datos producidos en el experimento ATLAS en un sistema distribuido a gran escala, para la actual y siguientes ejecuciones (*Runs*) del mismo.

El primer objetivo persigue encontrar las deficiencias del sistema de recolección de datos distribuido en el grid basado en un sistema de mensajería (modelo push) y proponer un nuevo sistema que mejore el rendimiento y escale mejor, reduciendo además, la complejidad y el uso de recursos. De esta forma, se trata de demostrar que un nuevo modelo *pull* basado en un almacenamiento de objetos (*object store*) como área de preparación de datos temporal, y con

selección dinámica para ser guardados en el almacenamiento final, satisface los requerimientos del proyecto mejorando en todas las áreas mencionadas.

Los metadatos recolectados han sido hasta ahora guardados en un catálogo central en el CERN, con un modelo híbrido basado en Hadoop HDFS como almacenamiento general, y con un subconjunto de los datos copiados a una base de datos Oracle para facilitar accesos más rápidos. El segundo objetivo es estudiar y proponer un nuevo almacenamiento final basado en tecnologías *big data* capaz de soportar las tasas de ingestión de datos requeridas, y que solucione los problemas anteriores relacionados con la complejidad de mantener un sistema híbrido y la coherencia de datos entre los subsistemas. Se estudia Kudu, basado en almacenamiento columnar y con soporte a cargas de trabajo híbridas, transaccionales y analíticas. También se estudia HBase por su soporte a accesos aleatorios de forma eficiente, y su versatilidad para soportar otras cargas de trabajo. Se analiza una capa sobre HBase denominada Phoenix que provee capacidades de modelado del esquema de datos y una interfaz SQL para el acceso a los datos.

El almacenamiento final seleccionado debe poder soportar todos los casos de uso. El tercer y último objetivo es probar que un esquema estricto en el modelado de los datos también provee beneficios en el acceso a los mismos, y que cargas de trabajo analíticas como en nuestros casos de uso también pueden ser satisfechas de forma eficiente con un sistema basado en HBase/Phoenix.

Metodología

La metodología seguida en este trabajo se basó en el estudio y caracterización de las necesidades para escalar y mejorar el sistema EventIndex para las siguientes ejecuciones del proyecto, incluyendo el Run 2 (2015-2018) y el Run 3 (2022-2025), y en preparación de los sucesivos *Runs*. Se realizó la implementación y evaluación de propuestas en entornos a gran escala y con datos reales.

En la figura 3 puede verse una representación de alto nivel de la arquitectura del proyecto EventIndex. Esta arquitectura flexible nos permite cambiar los componentes de un área modificando mínimamente las interfaces con el resto.

La producción de datos del EventIndex se basa en el indexado y extracción de metadatos de los ficheros guardados alrededor del mundo en el grid. Después de la indexación, los metadatos son recolectados y enviados al CERN. Durante el proceso de recolección puede haber procesos de transformación y consolidación de los datos para ser ingeridos en el sistema de almacenamiento final. El área de almacenamiento guarda los metadatos en un sistema *big data* escalable, y provee

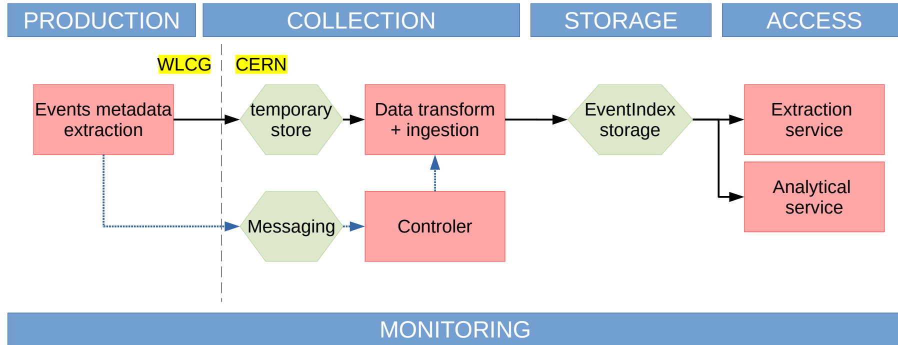


Figura 3: representación de alto nivel de la arquitectura del EventIndex compuesta por 5 áreas: producción de datos, recolección de datos, almacenamiento, acceso y monitorización. Los hexágonos verdes corresponden con almacenes temporales o permanentes de datos, y los rectángulos rosados representan procesos. Las flechas negras muestran el flujo de datos del EventIndex, y las flechas azules representan el flujo de información relacionada con el procesado.

de las interfaces necesarias para resolver los casos de uso requeridos. El área de acceso a los datos provee los servicios para resolver los casos de uso, englobados por clases en un sistema de extracción de datos (acceso aleatorio principalmente) y en el servicio de analítica de datos (acceso a grandes cantidades).

Las contribuciones de esta tesis se centran en las áreas de recolección distribuida de datos, almacenamiento final, y acceso a los mismos.

Recolección distribuida de datos

Se realizó primeramente un análisis de la arquitectura productor-consumidor para la recolección de datos desde el grid, usando un sistema de mensajería como transporte. Se detectaron una serie de problemas de rendimiento, ya que se producían bloqueos debido a la segmentación de la carga en múltiples mensajes impuesta por el sistema, y esto limitaba en la práctica la escalabilidad del sistema.

Se propuso un nuevo modelo *pull* para el transporte e ingestión de los datos en el almacenamiento final (*backend*). La nueva arquitectura se representa en la figura 4. En esta arquitectura se usa un almacenamiento de objetos (*object store*) para guardar de forma temporal la carga producida en el grid en un único

objeto por productor, de tal forma que no se segmenta la carga. Estos objetos son seleccionados y consumidos de forma dinámica para ser consolidados en el sistema de almacenamiento final (puede ser HDFS como originalmente o cualquiera que implementemos en el área de almacenamiento). De esta forma, pasamos de un sistema de mensajería con un modelo *push*, en el que toda la información se transmite hasta el *backend*, a un sistema con un modelo *pull*, en el que solo la información válida es guardada y finalmente consolidada en el *backend*.

Después de las primeras pruebas satisfactorias, se evaluó el sistema en el entorno de producción real. Se realizó un despliegue en paralelo de ambos sistemas, duplicando solo las partes necesarias. Por ejemplo, solo se desplegó un proceso de indexación y producción de datos, ahorrando tiempo de CPU consumida, pero enviando los metadatos indexados a ambos sistemas, a través de mensajería y del *object store*. El consumo e ingestión de datos se realizó en dos áreas diferentes dentro del sistema de almacenamiento final del CERN. El experimento para validar este modelo se llevó a cabo durante 3 meses, indexando más de 60 billones (10^9) de eventos distribuidos en 10 millones de ficheros alrededor del mundo, sumando un volumen total de 17 PB de datos de entrada leídos.

Los resultados demostraron que se mejoró el rendimiento en varias áreas. Con el nuevo diseño, evitamos la segmentación de la carga, almacenando los metadatos en un único objeto por proceso. Además, se consiguieron mejores tasas de compresión con objetos más grandes, agrupando información similar y de forma binaria (en vez de segmentar la carga en miles de mensajes y con codificación textual). De esta forma, logramos reducir en un factor 4.5 el volumen total de datos transmitidos y finalmente, guardados en el *backend* final. No se observan bloqueos en la nueva implementación usando el *object store*, de tal forma que se puede lograr mejor escalabilidad. El rendimiento durante la ingestión de datos mejora 15 veces comparado con el sistema anterior basado en mensajería. Además, el modelo *pull* permite la selección dinámica sobre los datos a consumir, evitando la ingestión de datos duplicados que en nuestro experimento alcanzó el 10 % de los datos producidos.

Después de la validación de la nueva implementación basada en el *object store*, se desplegó como único método de recolección de datos durante el Run 2 (2015-2018) y ha estado corriendo en producción desde entonces.

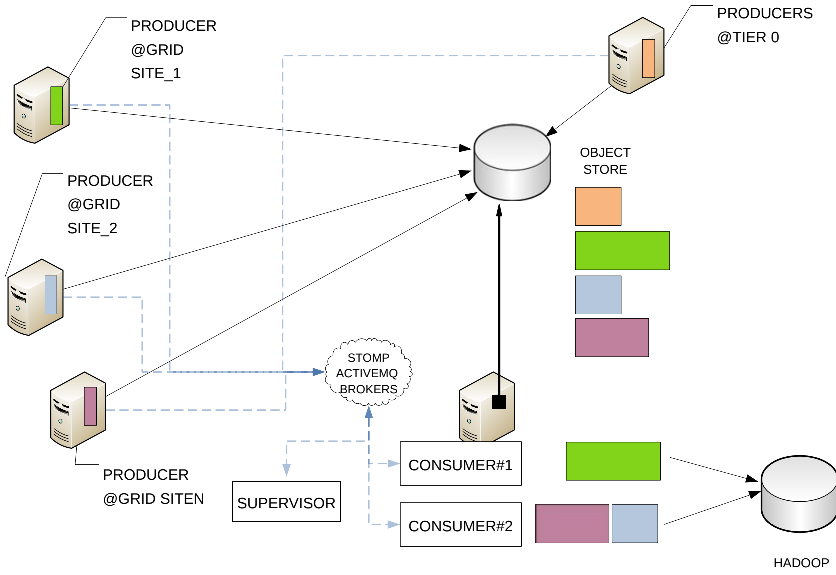


Figura 4: arquitectura de la recolección distribuida de datos basada en un *object store*. Las flechas negras representan el flujo de datos del EventIndex. Las flechas azules a lineas discontinuas representan la información de monitorización del procesado y mensajes de control. Los rectángulos coloreados representan la carga de datos guardada por los productores como objectos en el *object store*. Los consumidores acceden a los objetos con un modelo *pull* cuando es señalado por un proceso supervisor.

Almacenamiento

En este área se analizaron las tecnologías de almacenamiento adecuadas para nuestro proyecto y que pudieran escalar para las siguientes ejecuciones a partir del Run 3 (2022-2025).

Primero, se realizó una revisión de HDFS como almacenamiento principal que ha sido usado en producción desde el principio del proyecto. El trabajo se centró en el método de ingestión de datos y la organización de los mismos. El procedimiento de ingestión requiere de varias áreas de almacenamiento temporal

diferentes hasta consolidar los datos en el área final de producción. Primero, los datos son guardados de forma no necesariamente ordenada en un fichero SeqFile por *dataset.* En un paso posterior que realizan otras áreas del proyecto, se requiere de un proceso de aumentado de los datos (por ejemplo decodificando el *trigger*) y de ordenación y consolidación de los mismos para guardarlos finalmente en un MapFile por contenedor. Estos procesos son complicados y reducen la tasa efectiva de ingestión de datos, a la vez que aumentan el tiempo hasta que los datos son accesibles por el usuario final (definido como *traversal time* o tiempo de recorrido de los datos). Además, durante el Run 2 la implementación no pudo satisfacer los casos de uso de acceso aleatorio (por ejemplo selección de eventos particulares) debido al diseño inherente de HDFS, por lo que dentro del proyecto EventIndex se implementaron soluciones híbridas copiando un subconjunto de los datos a tablas de una base de datos Oracle o HBase. Esto complicó los procedimientos de copia, replicando de nuevo los datos en distintas plataformas y con posibles problemas de coherencia entre ellos.

Se buscaron otras soluciones para resolver las cuestiones de rendimiento y escalabilidad, de simplificación de procesos y de duplicidad y coherencia de los datos. La idea principal era mantener todos los datos en un almacenamiento único, para resolver todos los casos de forma eficiente, y que pudiera escalar con el volumen y tasas de ingestión esperadas. Primero se evaluó Kudu debido a su orientación como sistema híbrido para resolver cargas de trabajo transaccionales (acceso aleatorio) y analíticas (acceso secuencial a grandes cantidades de datos). Este sistema se basa en la definición de un modelo de datos con un esquema y tipos fijos (en vez de sin forzado de esquema como es HDFS) y almacenado físico de forma columnar. Se colaboró en el desarrollo de un modelo de datos basado en tablas similares al modelo relacional de bases de datos. En este modelo dos tablas guardan todos los eventos (una para los eventos reales y otra para los eventos simulados por métodos de Montecarlo).

Se desarrolló un nuevo complemento (*plugin*) dentro de nuestro sistema de recolección de datos para la ingestión en un *backend* basado en Kudu. Se realizaron experimentos de ingestión de datos reales en un *cluster* local, probando varios esquemas diferentes de organización interna de los datos. Las pruebas consistieron en un escenario típico en el que se van procesando *datasets*, lo que implica leer todos los objetos a consumir pertenecientes al mismo, se transforman los datos al esquema en Kudu, y se escriben los datos en el almacén final. Las transformaciones para el esquema implican decodificar los bits de los *triggers* empaquetados en los datos intermedios en *object store* (reduciendo el volumen de datos durante el envío). Una diferencia con respecto a la implementación

anterior basada en HDFS es que no hace falta que sea un solo hilo (*thread*) por *dataset* el que escriba (debido al diseño de HDFS con un único-escritor y multiples-lectores). Durante nuestras pruebas se comprobó que en el nuevo sistema, gracias a la implementación multi-hilo, menos del 1 % del tiempo es invertido en esperar datos del *object store*. Además, solo el 4 % del tiempo es invertido en la transformación de los datos al esquema en Kudu, siendo el resto del tiempo ocupado en la comunicación y transmisión de los datos. El rendimiento por hilo escritor en Kudu es de 5-6 kHz, y ahora se pueden emplear múltiples hilos de escritura por *dataset* para incrementar el rendimiento global. El mejor rendimiento se obtiene con esquemas que distribuyen los eventos de forma equitativa entre particiones. Además, el sistema dispone de un espacio de claves ordenado por diseño desde la primera ingestión, evitando otros procesos de consolidado de datos.

Aunque Kudu satisfacía los requerimientos de nuestra aplicación en cuanto a capacidades y rendimiento, la falta de un soporte claro en producción hizo buscar otras alternativas.

Estudiamos HBase como candidato a mantener todos los datos (anteriormente había sido usado para un subconjunto pequeño), junto con una capa implementada por Phoenix para proveer de tipado y esquemas fijos al modelo de datos, así como interfaces SQL para la ingestión y el acceso a los mismos. El esquema de datos con una gran tabla para los eventos fue desarrollado dentro del proyecto, y las contribuciones principales de este trabajo están relacionadas con la ingestión de datos y el registro de los mismos. Hemos definido una serie de tablas con meta-información y para mantener las relaciones jerárquicas de los *datasets* y contenedores tras la ingestión. Además, se necesitaba importar los datos previamente consolidados en HDFS para *Runs* anteriores en las nuevas estructuras de HBase/Phoenix, por lo que otras tablas auxiliares adicionales fueron definidas. Llevamos a cabo varios desarrollos para soportar la ingestión de datos en el *backend* de HBase/Phoenix. Un nuevo complemento fue desarrollado para soportar la ingestión en el sistema de colección de datos, para así completar la cadena de ingestión de datos desde el *object store* hasta el nuevo almacén final. Además, se desarrolló un procedimiento basado en MapReduce para importar la ingente cantidad de datos ya consolidados en HDFS hasta HBase/Phoenix. Se condujeron varios experimentos para evaluar el rendimiento y la mejor configuración de las tablas en el esquema de datos final. Se probaron varias técnicas con varias configuraciones: distribuir la carga entre regiones (funciones *hash* o *salting* añadiendo un prefijo a la clave); dividir con anterioridad la tabla en regiones (*region pre-splitting*); mapear nombres de columnas (*column mapping*) con un nivel de indirección para reducir su

tamaño (los nombres de las columnas se guardan en HBase en cada registro); codificar todas las columnas de una familia en una única celda (*immutable rows*); y deshabilitar el registro anticipado de transacciones o WAL (*Write Ahead Log)*. Se usó el *cluster* de computación del CERN compuesto por 39 nodos (32 servidores de regiones de HBase) con un total de 18 TB de memoria y 1,658 *vcores*. Es un *cluster* compartido por varios proyectos, incluído EventIndex. La configuración incluía las siguientes distribuciones de *software*: Hadoop 3.2.1, HBase 2.2.4 y Apache Phoenix 5.0. Se llevaron a cabo experimentos con varias tablas y configuraciones, y una ingestión masiva de aproximadamente 8,000 *datasets*, con 70 billones de eventos y ocupando finalmente un volumen de 22 TB en HBase. Se obtuvieron rendimientos de 117 kHz de media durante la ingestión. Los mejores rendimientos individuales se obtienen con *datasets* más grandes y con configuraciones que distribuyen la carga en el espacio de claves. La pre-división en regiones evita los rendimientos lentos al comienzo, lo que puede ser beneficioso cuando se pone en producción por primera vez. Deshabilitar el WAL mejora el rendimiento de la escritura y es factible en nuestro caso, al poder reproducirse los datos desde el *object store* en caso de fallo. Otras características como el mapeo de columnas o la codificación en celdas inmutables no proveen de mejoras sustanciales en nuestra aplicación y modelo, y en cambio imponen una dependencia sobre Phoenix innecesaria. El tiempo invertido para la conversión de los datos a nuestro modelo en HBase está en el 8.5 %, siendo el resto ocupado por el envío efectivo de los datos a los servidores de regiones de HBase. Los resultados mostraron que gracias al modelo en HBase se pueden utilizar varios hilos por *dataset* escalando convenientemente y obtener valores de rendimiento suficientes para el Run 3 y siguientes.

Acceso

Se incorporó un entorno basado en Spark para acceder a los datos guardados en nuestro modelo en HBase/Phoenix, y resolver casos de uso analíticos que requieren acceso a grandes cantidades de datos. En la figura 5 se muestra la arquitectura propuesta donde Spark es la herramienta principal que provee las abstracciones y el modelado de datos orientado particularmente para operaciones eficientes en memoria. Spark tiene interfaz nativa con los gestores de recursos como en nuestro caso YARN para proveer los recursos computacionales. La interfaz con el almacenamiento es versátil y puede acceder a varios tipos de almacenamiento, en nuestro caso HBase/Phoenix. El acceso por parte de los usuarios o herramientas de alto nivel puede ser tanto interactivo desde la línea de comandos o *notebooks*, como en segundo plano (*batch*) a través de interfaces

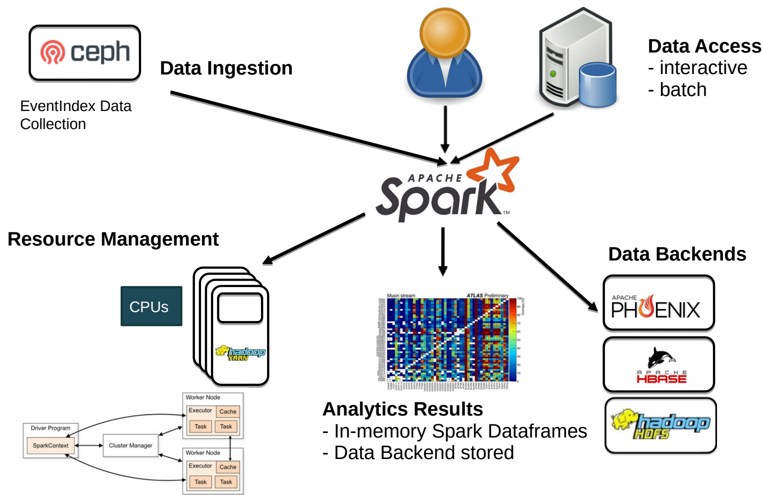


Figura 5: arquitectura de la plataforma analítica de EventIndex basada en Apache Spark.

de programación (API). Los resultados calculados son modelados como Spark DataFrames que pueden ser utilizados para otras computaciones o almacenados de nuevo con los modelos adecuados en HBase/Phoenix.

Se desarrollaron herramientas para acceder a los datos y en particular para la resolución de casos de uso analíticos en forma de un paquete de Spark usando Scala como lenguaje de programación. Se proveen abstracciones sobre el modelo propuesto para la localización de los datos a nivel de contenedor, *dataset* o evento. Además, se modelan como DataFrames que pueden ser usadas por el resto de herramientas. Se resolvieron casos de uso como la detección de duplicados o de solapes (*overlaps*) de eventos entre distintos *datasets*. Para la evaluación se utilizaron tanto datos reales guardados en HBase/Phoenix como el *cluster* del CERN mencionado en la sección anterior, disponiendo de Spark 2.4.8 y 3.3.0.

En el caso de detección de *overlaps* se presentó un algoritmo con coste

computacional $\mathcal{O}(n)$ con el número de eventos y con coste espacial $\mathcal{O}(s^2)$ con el número de flujos (*streams*). El algoritmo escala automáticamente hasta 32 procesos gracias a la configuración del *cluster*, obteniendo un rendimiento de 380 kHz (eventos procesados por segundo) para 500 M de eventos. Solo es necesario acceder a la clave de la tabla y no a ninguna familia de datos de HBase, con lo que se obtiene un rendimiento óptimo en cuanto a la cantidad de datos accedidos. La detección de duplicados accede a más datos por evento (en nuestro modelo los metadatos sobre localización están en una familia especifica de HBase), por lo que se obtiene un rendimiento de 120 kHz para 100 M de eventos, comparado con los 150 kHz que se obtienen para el mismo número de eventos en el algoritmo de *overlaps*. Las tasas de procesado muestran una penalización para *datasets* pequeños debido a que la inicialización y establecimiento de los flujos de datos a los servidores de regiones de HBase dominan el tiempo total.

Conclusiones

En este trabajo hemos hecho varias contribuciones al proyecto EventIndex en las áreas de recolección de datos distribuidos, almacenamiento de grandes cantidades de datos y acceso a los mismos.

Hemos mejorado el sistema de recolección de datos en grandes infraestructuras distribuidas como el grid. El anterior sistema basado en mensajería era complejo y limitaba la escalabilidad de nuestra aplicación para los siguientes *Runs*. Con un nuevo diseño basado en un almacenamiento de objetos (object store) temporal, y con un modelo *pull* de selección de datos dinámica para su ingestión en el almacenamiento final, mejoramos el rendimiento en varias áreas. Con el nuevo modelo evitamos la segmentación de la carga, almacenando toda la información de cada proceso de extracción de metadatos en un único objeto. De esta manera conseguimos mejores tasas de compresión al aplicar codificación binaria a datos relacionados y agrupados con mayor granularidad. Se logra por lo tanto, una reducción de un factor 4.5 en el volumen de datos totales enviados por la red a través del sistema de recolección, e ingeridos en el almacenamiento final. Desde que hemos aplicado esta mejora basada en el *object store*, no se han observado bloqueos y la distribución de la carga de trabajo se ha optimizado consiguiendo mayor escalabilidad. Así mismo el rendimiento ha mejorado 15 veces comparado con la implementación anterior basada en mensajería. Además, la aproximación del modelo *pull* permite la selección dinámica de datos a consumir, evitando ingerir datos duplicados generados en el grid, que en nuestro experimento es del orden del 10 % de los datos producidos. El modelo

pull basado en un *object store* fue desplegado en producción durante el Run 2 (2015-2018) resolviendo las limitaciones anteriores y muestra un excelente desempeño en producción desde entonces, siendo capaz de escalar para las tasas requeridas en los siguientes años.

En el área de almacenamiento de datos, se seleccionó HBase por su buen rendimiento y por su soporte superior en producción. La flexibilidad de nuestra arquitectura permite cambiar el almacenamiento final manteniendo el modelo *pull* de ingestión de datos. Con HBase como principal y único almacenamiento final para todos nuestros datos logramos simplificar el almacenamiento, evitar duplicidades entre subsistemas y posibles problemas de coherencia entre los mismos, y reducir el volumen total de datos. De esta forma pasamos de un sistema híbrido que mantenía los datos en HDFS y Oracle, a un sistema único basado en HBase. Con una una capa de acceso basada en SQL que no estaba disponible anteriormente, logramos dotar de una interfaz única a terceros, y además nos aseguramos del cumplimiento del esquema de datos con tipos definidos. Esto no estaba disponible anteriormente en el modelo sin esquema de HDFS, de tal forma que con la nueva aproximación se pueden aplicar mejoras en el almacenamiento de datos a nivel columnar, reduciendo el volumen total de datos ocupados. Así mismo con este esquema ya no se necesitan procedimientos añadidos de aumento de datos, ya que todos los necesarios pueden hacerse en línea durante el proceso de ingestión. En cuanto a rendimiento y relacionado con los parámetros del esquema de nuestro modelo de datos, hemos mostrado que una división de regiones previa en HBase y una distribución de la carga entre regiones basada en *hashes* sobre la clave (*salting*) mejora las tasas de ingestión de datos. HBase/Phoenix incluye por diseño una ordenación global basada en la clave, con lo cual evitamos los procesos de ordenación y consolidación que teníamos anteriormente en la implementación con HDFS. HBase permite escribir a nivel de celda (registro de evento en EventIndex), con lo que se pueden tener múltiples hilos escritores por *dataset*. De esta manera, aumentamos el rendimiento comparado con el único hilo escritor por *dataset* que teníamos en la implementación anterior con HDFS. Hemos contribuido a reducir la complejidad en el sistema de almacenamiento y el uso de recursos, optimizando el volumen de datos a la vez que se mejora el rendimiento durante la ingestión de datos. Estas mejoras incrementan la fiabilidad y el rendimiento global del sistema de almacenamiento requerido para el Run 3 (2022-2025) y sucesivos.

Con nuestras contribuciones en el área de acceso a los datos, ahora es posible el acceso y análisis interactivo, cosa que no era posible con el modelo anterior. Hemos mejorado los casos de uso analíticos en el nuevo sistema almacenando los datos con un modelo en HBase/Phoenix y accediendo mediante una plataforma

basada en Spark, usando sus abstracciones y un conjunto de herramientas implementadas en Scala. La nueva herramienta y algoritmos resuelven el acceso a los datos y los casos de uso analíticos en áreas como el descubrimiento de datos, detección de duplicados y cálculo de solapes (*overlaps*) entre *datasets*, que ahora son utilizables de forma integrada. Las herramientas producen DataFrames que son usadas por el resto de funcionalidades, en particular para el cálculo de duplicados y *overlaps*. Además los datos y resultados analizados se mantienen en caché, cosa que no era posible anteriormente, facilitando el encadenamiento de algoritmos, y la mejora global en la utilización de recursos. El algoritmo de cálculo de *overlaps* tiene un coste computacional $\mathcal{O}(n)$ con el número de eventos, y coste espacial $\mathcal{O}(s^2)$ con el número de flujos (*streams*) de *datasets*. Su desempeño en HBase/Phoenix es óptimo, ya que solo requiere acceder a la clave de cada evento. El rendimiento mejora con respecto a la anterior aproximación ya que ahora el acceso a datos específicos de un registro de evento se puede hacer de forma individual, en vez de acceder a todo el registro como en HDFS. También la nueva organización columnar permite reducir el número de operaciones de entrada/salida, contribuyendo a la mejora del rendimiento. Con nuestras herramientas abstraemos el modelo de datos final, desacoplando el acceso a los datos del esquema y tecnologías reales usadas. Esta aproximación simplifica las complejidades del sistema con unas interfaces definidas e intuitivas para herramientas y usuarios. También enmascara posibles cambios en el modelo de datos, que son invisibles para el usuario ya que las interfaces que usa no cambian. Con esta aproximación logramos resolver los casos de uso analíticos del proyecto EventIndex de una manera eficiente, dando caminos de acceso que no existían anteriormente. De esta forma hemos contribuido a lograr el objetivo de construir un catálogo de todos los eventos reales y simulados del proyecto ATLAS durante todos sus años de ejecución, y en particular para los retos previstos para el Run 3 (2022-2025) y en adelante.

Bibliography

[1] Lyndon Evans and Philip Bryant. LHC Machine. In: *Journal of Instrumentation* 3.08 (Aug. 2008), S08001–S08001. DOI: 10.1088/1748-0221/3/08/s08001 (cit. on pp. 5, 136).

[2] Ewa Lopienska. The CERN accelerator complex, layout in 2022. Complexe des accélérateurs du CERN en janvier 2022. In: (Feb. 2022). General Photo. URL: https://cds.cern.ch/record/2800984 (cit. on p. 6).

[3] *Letter of Intent for the LHCb Upgrade*. Tech. rep. Geneva: CERN, 2011. URL: https://cds.cern.ch/record/1333091 (cit. on p. 8).

[4] The ALICE Collaboration. Transverse-momentum and event-shape dependence of D-meson flow harmonics in Pb–Pb collisions at sNN=5.02TeV. In: *Physics Letters B* 813 (2021), p. 136054. ISSN: 0370-2693. DOI: https://doi.org/10.1016/j.physletb.2020.136054 (cit. on p. 8).

[5] G. Aad et al. The ATLAS Experiment at the CERN Large Hadron Collider. In: *JINST* 3 (2008), S08003. DOI: 10.1088/1748-0221/3/08/S08003 (cit. on pp. 8, 136).

[6] Joao Pequenao. *Computer generated image of the whole ATLAS detector*. Mar. 2008. URL: https://cds.cern.ch/record/1095924 (cit. on pp. 9, 137).

[7] Joao Pequenao and Paul Schaffner. How ATLAS detects particles: diagram of particle paths in the detector. Jan. 2013. URL: https://cds.cern.ch/record/1505342 (cit. on p. 10).

[8] Joao Pequenao. Computer generated image of the ATLAS inner detector. Mar. 2008. URL: https://cds.cern.ch/record/1095926 (cit. on p. 11).

[9] Brad Abbott, Justin Albert, Fabrizio Alberti, Markus Alex, Gianluca Alimonti, Steven Alkire, P Allport, Silke Altenheiner, Lucian Stefan Ancu, E Anderssen, et al. Production and integration of the ATLAS Insertable B-Layer. In: *Journal of instrumentation* 13.05 (2018), T05008 (cit. on p. 11).

[10] Joao Pequenao. Computer Generated image of the ATLAS calorimeter. Mar. 2008. URL: `https://cds.cern.ch/record/1095927` (cit. on p. 13).

[11] M (CERN) Aleksa, W (Pittsburgh) Cleland, Y (Tokyo) Enari, M (Victoria) Fincke-Keeler, L (CERN) Hervas, F (BNL) Lanni, S (Oregon) Majewski, C (Victoria) Marino, and I (LAPP) Wingerter-Seez. *ATLAS Liquid Argon Calorimeter Phase-I Upgrade: Technical Design Report*. Tech. rep. Final version presented to December 2013 LHCC. Sept. 2013. URL: `https://cds.cern.ch/record/1602230` (cit. on p. 13).

[12] Ellis Kay. *Commissioning the Phase-1 LAr Upgrade*. Tech. rep. Geneva: CERN, Aug. 2021. URL: `https://cds.cern.ch/record/2779554` (cit. on p. 14).

[13] Joao Pequenao. Computer generated image of the ATLAS Muons subsystem. Mar. 2008. URL: `https://cds.cern.ch/record/1095929` (cit. on p. 15).

[14] Bernd Stelzer. The New Small Wheel Upgrade Project of the ATLAS Experiment. In: *Nuclear and Particle Physics Proceedings* 273-275 (2016). 37th International Conference on High Energy Physics (ICHEP), pp. 1160–1165. ISSN: 2405-6014. DOI: `https://doi.org/10.1016/j.nuclphysbps.2015.09.182` (cit. on p. 15).

[15] The ATLAS collaboration. Operation of the ATLAS trigger system in Run 2. In: *Journal of Instrumentation* 15.10 (Oct. 2020), P10004–P10004. DOI: `10.1088/1748-0221/15/10/p10004` (cit. on pp. 16, 136).

[16] ATLAS Collaboration. *Athena*. Version 21.0.127. May 2021. DOI: `10.5281/zenodo.4772550` (cit. on p. 18).

[17] Rafal Bielski. ATLAS High Level Trigger within the multi-threaded software framework AthenaMT. In: *Journal of Physics: Conference Series* 1525.1 (Apr. 2020), p. 012031. DOI: `10.1088/1742-6596/1525/1/012031` (cit. on p. 18).

[18] Markus Elsing, Luc Goossens, Armin Nairz, and Guido Negri. The ATLAS Tier-0: Overview and operational experience. In: *Journal of Physics: Conference Series* 219.7 (Apr. 2010), p. 072011. DOI: 10.1088/1742-6596/219/7/072011 (cit. on p. 18).

[19] Carlos Chavez, Michele Gianelli, Alex Martyniuk, Joerg Stelzer, Mark Stockton, and Will Vazquez. The Database Driven ATLAS Trigger Configuration System. In: *Journal of Physics: Conference Series* 664.8 (Dec. 2015), p. 082030. DOI: 10.1088/1742-6596/664/8/082030 (cit. on p. 19).

[20] E J Gallas, S Albrand, M Borodin, and A Formica and. Utility of collecting metadata to manage a large scale conditions database in ATLAS. In: *Journal of Physics: Conference Series* 513.4 (June 2014), p. 042020. DOI: 10.1088/1742-6596/513/4/042020 (cit. on p. 19).

[21] C P Bee, D Francis, L Mapelli, R McLaren, Giuseppe Mornacchi, J Petersen, and F J Wickens. *The raw event format in the ATLAS Trigger & DAQ*. Tech. rep. Revised version number 5 submitted on 2016-11-03 11:47. Geneva: CERN, Feb. 2016. URL: https://cds.cern.ch/record/683741 (cit. on pp. 20, 74, 75, 139).

[22] K. Bos, N. Brook, D. Duellmann, C. Eck, I. Fisk, D. Foster, B. Gibbard, C. Grandi, F. Grey, J. Harvey, A. Heiss, F. Hemmer, S. Jarp, R. Jones, D. Kelsey, J. Knobloch, M. Lamanna, H. Marten, P. Mato Vila, F. Ould-Saada, B. Panzer-Steindel, L. Perini, L. Robertson, Y. Schutz, U. Schwickerath, J. Shiers, and T. Wenaus. *LHC computing Grid: Technical Design Report. Version 1.06 (20 Jun 2005)*. Technical design report. LCG. Geneva: CERN, 2005. URL: https://cds.cern.ch/record/840543 (cit. on pp. 20, 138).

[23] *Worldwide LHC Computing Grid (WLCG) Map. September 2022.* https://wlcg.web.cern.ch/using-wlcg/monitoring-visualisation. (Visited on 09/01/2022) (cit. on pp. 21, 138).

[24] The ATLAS Collaboration, Georges Aad, B Abbott, J Abdallah, AA Abdelalim, Abdelmalek Abdesselam, B Abi, M Abolins, H Abramowicz, H Abreu, BS Acharya, et al. The ATLAS simulation infrastructure. In: *The European Physical Journal C* 70.3 (2010), pp. 823–874. DOI: 10.1140/epjc/s10052-010-1429-9 (cit. on p. 21).

[25] Andrea Valassi, Efe Yazgan, Josh McFayden, Simone Amoroso, Joshua Bendavid, Andy Buckley, Matteo Cacciari, Taylor Childers, Vitaliano Ciulli, Rikkert Frederix, et al. Challenges in Monte Carlo event generator software for High-Luminosity LHC. In: *Computing and Software for Big Science* 5.1 (May 2021), p. 12. ISSN: 2510-2044. DOI: 10.1007/s41781-021-00055-1 (cit. on p. 21).

[26] Paul J. Leach, Rich Salz, and Michael H. Mealling. *A Universally Unique IDentifier (UUID) URN Namespace*. RFC 4122. July 2005. DOI: 10.17487/RFC4122 (cit. on p. 21).

[27] S Albrand, J Chapman, D Cote, L Fiorini, EJ Gallas, V Garonne, C Gwenlan, P Laycock, A Klimentov, D Malon, E Torrence, G Unal, T Wenaus, J Catmore, D Charlton, L Gossens, A Nairz, D Barberis, F Gianotti, C Guyot, R Hawkings, I Hinchliffe, B Heinemann, A Höcker, G Lehmann, P Nevski, and H von der Schmitt. *ATLAS Dataset Nomenclature*. Tech. rep. Geneva: CERN, Nov. 2007. URL: https://cds.cern.ch/record/1070318 (cit. on pp. 22, 33).

[28] 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. In: *Computing and Software for Big Science* 3.1 (Aug. 2019), p. 11. ISSN: 2510-2044. DOI: 10.1007/s41781-019-0026-3 (cit. on pp. 22, 27).

[29] *High Luminosity LHC*. https://hilumilhc.web.cern.ch/content/hl-lhc-project. (Visited on 09/01/2022) (cit. on pp. 23, 139).

[30] ATLAS Collaboration. *ATLAS Software and Computing HL-LHC Roadmap*. Tech. rep. Geneva: CERN, 2022. URL: https://cds.cern.ch/record/2802918 (cit. on pp. 23–25).

[31] *HepSpec06*. https://w3.hepix.org/benchmarking.htm. (Visited on 09/01/2022) (cit. on p. 23).

[32] I Bird, P Buncic, F Carminati, M Cattaneo, P Clarke, I Fisk, M Girone, J Harvey, B Kersevan, P Mato, R Mount, and B Panzer-Steindel. *Update of the Computing Models of the WLCG and the LHC Experiments*. Tech.

rep. Apr. 2014. URL: https://cds.cern.ch/record/1695401 (cit. on p. 23).

[33] Karl Rupp. *48 Years of Microprocessor Trend Data*. Version 2020.0. July 2020. DOI: 10.5281/zenodo.3947824 (cit. on p. 24).

[34] Charles Leggett, John Baines, Tomasz Bold, Paolo Calafiura, Steven Farrell, Peter van Gemmeren, David Malon, Elmar Ritsch, Graeme Stewart, Scott Snyder, Vakhtang Tsulaia, and Benjamin Wynne and. AthenaMT: upgrading the ATLAS software framework for the many-core world with multi-threading. In: *Journal of Physics: Conference Series* 898 (Oct. 2017), p. 042009. DOI: 10.1088/1742-6596/898/4/042009 (cit. on p. 24).

[35] A Buckley, T Eifert, M Elsing, D Gillberg, K Koeneke, A Krasznahorkay, E Moyse, M Nowak, S Snyder, and P van Gemmeren. Implementation of the ATLAS Run 2 event data model. In: *Journal of Physics: Conference Series* 664.7 (Dec. 2015), p. 072045. DOI: 10.1088/1742-6596/664/7/072045 (cit. on p. 25).

[36] James Catmore, Jack Cranshaw, Thomas Gillam, Eirik Gramstad, Paul Laycock, Nurcan Ozturk, and Graeme Andrew Stewart. A new petabyte-scale data derivation framework for ATLAS. In: *Journal of Physics: Conference Series* 664.7 (Dec. 2015), p. 072007. DOI: 10.1088/1742-6596/664/7/072007 (cit. on pp. 25, 28).

[37] 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. In: *EPJ Web Conf.* 245 (2020), p. 06014. DOI: 10.1051/epjconf/202024506014 (cit. on pp. 25, 32).

[38] TJ Khoo, A Reinsvold Hall, N Skidmore, S Alderweireldt, J Anders, C Burr, W Buttinger, P David, L Gouskos, L Gray, et al. Constraints on future analysis metadata systems in High Energy Physics. In: *Computing and Software for Big Science* 6.1 (2022), pp. 1–9. ISSN: 2510-2044. DOI: 10.1007/s41781-022-00086-2 (cit. on p. 26).

[39] Berghaus, Frank, Krasznahorkay, Attila, Martin, Tim, Novak, Tadej, Nowak, Marcin, Schaffer, A.C., Tsulaia, Vakho, and van Gemmeren, Peter. ATLAS in-file metadata and multi-threaded processing. In: *EPJ Web Conf.* 251 (2021), p. 03006. DOI: 10.1051/epjconf/202125103006 (cit. on p. 26).

[40] Jerome Fulachier, J Odier, F Lambert, ATLAS Collaboration, et al. ATLAS Metadata Interface (AMI), a generic metadata framework. In: *Journal of Physics: Conference Series*. Vol. 898. 6. IOP Publishing. 2017, p. 062001 (cit. on pp. 26, 33).

[41] EJ Gallas, Solveig Albrand, Mikhail Borodin, Andrea Formica, Atlas Collaboration, et al. Utility of collecting metadata to manage a large scale conditions database in ATLAS. In: *Journal of Physics: Conference Series*. Vol. 513. 4. IOP Publishing. 2014, p. 042020 (cit. on p. 26).

[42] D Malon, J Cranshaw, and Q Zhang. An extensible infrastructure for querying and mining event-level metadata in ATLAS. In: *Journal of Physics: Conference Series* 396.5 (Dec. 2012), p. 052053. DOI: 10.1088/1742-6596/396/5/052053 (cit. on p. 27).

[43] P van Gemmeren, D Malon, and M Nowak and. Next-Generation Navigational Infrastructure and the ATLAS Event Store. In: *Journal of Physics: Conference Series* 513.5 (June 2014), p. 052036. DOI: 10.1088/1742-6596/513/5/052036 (cit. on pp. 30, 75).

[44] F H Barreiro Megino, K De, A Klimentov, T Maeno, P Nilsson, D Oleynik, S Padolski, S Panitkin, and T Wenaus and. PanDA for ATLAS distributed computing in the next decade. In: *Journal of Physics: Conference Series* 898 (Oct. 2017), p. 052002. DOI: 10.1088/1742-6596/898/5/052002 (cit. on p. 33).

[45] F H Barreiro, M Borodin, K De, D Golubkov, A Klimentov, T Maeno, R Mashinistov, S Padolski, and T Wenaus and. The ATLAS Production System Evolution: New Data Processing and Analysis Paradigm for the LHC Run2 and High-Luminosity. In: *Journal of Physics: Conference Series* 898 (Oct. 2017), p. 052016. DOI: 10.1088/1742-6596/898/5/052016 (cit. on p. 33).

[46] G A Stewart, W B Breaden-Madden, H J Maddocks, T Harenberg, M Sandhoff, and B Sarrazin. ATLAS Job Transforms: A Data Driven Workflow Engine. In: *Journal of Physics: Conference Series* 513.3 (June 2014), p. 032094. DOI: 10.1088/1742-6596/513/3/032094 (cit. on p. 35).

[47] J Sánchez, A Fernández Casaní, and S González de la Hoz. Distributed Data Collection for the ATLAS EventIndex. In: *Journal of Physics: Conference Series* 664.4 (Dec. 2015), p. 042046. DOI: 10.1088/1742-6596/664/4/042046 (cit. on pp. 35, 40, 75, 131).

[48] Dario Barberis, Igor Aleksandrov, Evgeny Alexandrov, Zbigniew Baranowski, Luca Canali, Elizaveta Cherepanova, Gancho Dimitrov, Andrea Favareto, Alvaro Fernandez Casani, Elizabeth J. Gallas, Carlos Garcia Montoro, Santiago Gonzalez de la Hoz, Julius Hrivnac, Alexander Iakovlev, Andrei Kazymov, Mikhail Mineev, Fedor Prokoshin, Grigori Rybkin, Jose Salt, Javier Sanchez, Roman Sorokoletov, Rainer Toebbicke, Petya Vasileva, Miguel Villaplana Perez, and Ruijun Yuan. *The ATLAS EventIndex: a BigData catalogue for all ATLAS experiment events.* For publication in Computing and Software for Big Science. Q1. 2022. DOI: `10.48550/ARXIV.2211.08293` (cit. on pp. 35, 69, 76, 133).

[49] Santiago González de la Hoz, Carlos Acosta-Silva, Javier Aparisi Pozo, Manuel Delfino, Jose del Peso, Álvaro Fernández Casani, José Flix Molina, Esteban Fullana Torregrosa, Carlos García Montoro, Julio Lozano Bahilo, Almudena del Rocio Montiel, Andreu Pacheco Pages, Javier Sánchez Martínez, José Salt, and Aresh Vedaee. Spanish ATLAS Tier-1 & Tier-2 perspective on computing over the next years. In: *EPJ Web of Conferences* 214 (2019). Ed. by A. Forti, L. Betev, M. Litmaath, O. Smirnova, and P. Hristov, p. 03013. DOI: `10.1051/epjconf/201921403013` (cit. on pp. 35, 132).

[50] Tom White. *Hadoop: The Definitive Guide.* O'Reilly Media, Inc., 2012. ISBN: 9781449311520 (cit. on p. 35).

[51] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST2010.* 2010, pp. 1–10. DOI: `10.1109/MSST.2010.5496972` (cit. on pp. 35, 70).

[52] *Hadoop MapFile.* `https://hadoop.apache.org/docs/r3.3.0/api/org/apache/hadoop/io/MapFile.html`. (Visited on 09/01/2022) (cit. on p. 35).

[53] *Oracle DB.* `https://www.oracle.com/database/technologies/`. (Visited on 09/01/2022) (cit. on p. 36).

[54] E J Gallas, G Dimitrov, P Vasileva, Z Baranowski, L Canali, A Dumitru, and A Formica and. An Oracle-based event index for ATLAS. In: *Journal of Physics: Conference Series* 898 (Oct. 2017), p. 042033. DOI: `10.1088/1742-6596/898/4/042033` (cit. on pp. 36, 69).

[55] Z. Baranowski, L. Canali, R. Toebbicke, J. Hrivnac, and D. Barberis. A study of data representation in Hadoop to optimize data storage and search performance for the ATLAS EventIndex. In: *Journal of Physics: Conference Series* 898 (Oct. 2017), p. 062020. DOI: 10.1088/1742-6596/898/6/062020 (cit. on pp. 36, 69, 76).

[56] Todd Lipcon, David Alves, Dan Burkert, Jean-Daniel Cryans, Adar Dembo, Mike Percy, Silvius Rus, Dave Wang, Matteo Bertozzi, Colin Patrick McCabe, et al. Kudu: Storage for fast analytics on fast data. In: *Cloudera, inc* 28 (2015) (cit. on pp. 36, 77).

[57] Lars George. *HBase: the definitive guide: random access to your planet-size data.* " O'Reilly Media, Inc.", 2011 (cit. on p. 36).

[58] *Apache Phoenix.* https://phoenix.apache.org/. (Visited on 09/01/2022) (cit. on pp. 36, 84).

[59] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In: *Communications of the ACM* 51.1 (2008), pp. 107–113 (cit. on pp. 36, 71, 94).

[60] *Apache Spark.* https://spark.apache.org/. (Visited on 09/01/2022) (cit. on pp. 37, 77).

[61] D. Barberis, S.E. Cárdenas Zárate, A. Favareto, A. Fernandez Casani, E.J. Gallas, C. Garcia Montoro, S. Gonzalez de la Hoz, J. Hrivnac, D. Malon, F. Prokoshin, J. Salt, J. Sanchez, R. Toebbicke, and R. Yuan and. ATLAS Eventlndex monitoring system using the Kibana analytics and visualization platform. In: *Journal of Physics: Conference Series* 762 (Oct. 2016), p. 012004. DOI: 10.1088/1742-6596/762/1/012004 (cit. on pp. 37, 132).

[62] Vishal Sharma. Getting Started with Kibana. In: *Beginning Elastic Stack.* Berkeley, CA: Apress, 2016, pp. 29–44. ISBN: 978-1-4842-1694-1. DOI: 10.1007/978-1-4842-1694-1_3 (cit. on p. 37).

[63] A Fernandez Casani, D Barberis, A Favareto, C Garcia Montoro, S González de la Hoz, J Hřivnáč, F Prokoshin, J Salt, J Sanchez, Többicke, R Yuan, and ATLAS Collaboration. ATLAS EventIndex general dataflow and monitoring infrastructure. In: *Journal of Physics: Conference Series* 898.6 (2017), p. 062010. DOI: 10.1088/1742-6596/898/6/062010 (cit. on pp. 37, 132).

[64] E Alexandrov, A Kazymov, and F Prokoshin. BigData Tools for the Monitoring of the ATLAS EventIndex. In: *CEUR Workshop Proceedings*. Proceedings of the VIII International Conference "Distributed Computing and Grid-technologies in Science and Education" (GRID 2018), Dubna (Russia). 2018, pp. 91–94. URL: http://ceur-ws.org/Vol-2267/91-94-paper-15.pdf (cit. on p. 37).

[65] *InfluxDB*. https://www.influxdata.com/products/influxdb-overview/. (Visited on 09/01/2022) (cit. on p. 37).

[66] Mainak Chakraborty and Ajit Pratap Kundan. Grafana. In: *Monitoring Cloud-Native Applications*. Springer, 2021, pp. 187–240 (cit. on p. 37).

[67] Ralph Kimball and Joe Caserta. *The data warehouse ETL toolkit*. John Wiley & Sons, 2004 (cit. on p. 39).

[68] *ActiveMQ*. http://activemq.apache.org/. (Visited on 09/01/2022) (cit. on p. 40).

[69] *Stomp protocol*. https://stomp.github.io/. (Visited on 09/01/2022) (cit. on p. 40).

[70] ECMA International. *Standard ECMA-404. The JSON Data Interchange Format*. 2017 (cit. on p. 41).

[71] *Andrew File System (AFS)*. https://www.openafs.org/. (Visited on 09/01/2022) (cit. on p. 43).

[72] Alvaro Fernandez Casani, Javier Sanchez, Santiago Gonzalez de la Hoz, and Juan M. Orduña. Designing Alternative Transport Methods for the Distributed Data Collection of ATLAS EventIndex Project. In: (Nov. 2016). URL: http://cds.cern.ch/record/2235644/files/ATL-SOFT-SLIDE-2016-869.pdf (cit. on pp. 44, 51).

[73] M. Karol, M. Hluchyj, and S. Morgan. Input Versus Output Queueing on a Space-Division Packet Switch. In: *IEEE Transactions on Communications* 35.12 (1987), pp. 1347–1356. DOI: 10.1109/TCOM.1987.1096719 (cit. on p. 44).

[74] M. Mesnier, G. R. Ganger, and E. Riedel. Object-based storage. In: *IEEE Communications Magazine* 41.8 (Aug. 2003), pp. 84–90. ISSN: 0163-6804. DOI: 10.1109/MCOM.2003.1222722 (cit. on p. 46).

[75] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-performance Distributed File System. In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. OSDI '06. Seattle, Washington: USENIX Association, 2006, pp. 307–320. isbn: 1-931971-47-1 (cit. on p. 46).

[76] *Amazon S3, Cloud Computing Storage for Files, Images, Videos*. http://aws.amazon.com. (Visited on 09/01/2022) (cit. on p. 47).

[77] *Google Protocol Buffers: Google's Data Interchange Format*. https://developers.google.com/protocol-buffers/. (Visited on 07/15/2021) (cit. on p. 47).

[78] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004. isbn: 0321245628 (cit. on p. 47).

[79] Tim Berners-Lee, Roy T. Fielding, and Larry M Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986. Jan. 2005. doi: 10.17487/RFC3986 (cit. on p. 49).

[80] Álvaro Fernández Casaní, Juan M. Orduña, Javier Sánchez, and Santiago González de la Hoz. A Reliable Large Distributed Object Store Based Platform for Collecting Event Metadata. In: *Journal of Grid Computing* 19.3 (Aug. 2021). Q1, p. 39. issn: 1572-9184. doi: 10.1007/s10723-021-09580-0 (cit. on pp. 51, 57–64, 66, 133).

[81] Álvaro Fernández Casaní, Juan Orduña, and Santiago González de la Hoz. Performance Improvements of an Event Index Distributed System. In: *ICPP 2018: Proceedings of the 47th International Conference on Parallel Processing*. Extended abstract. Eugene, OR, USA: Association for Computing Machinery, 2018. isbn: 9781450365109. url: http://oaciss.uoregon.edu/icpp18/publications/pos110s2-file1.pdf (cit. on pp. 69, 132).

[82] D. Barberis, J. Cranshaw, A. Favareto, A. Fernández Casaní, E. Gallas, S. González de la Hoz, J. Hřivnáč, D. Malon, M. Nowak, F. Prokoshin, J. Salt, J. Sánchez Martínez, R. Többicke, and R. Yuan. The ATLAS EventIndex: Full chain deployment and first operation. In: *Nuclear and Particle Physics Proceedings* 273-275 (Apr. 2016). Q3. Corresponding author A. Fernández Casaní, pp. 913–918. doi: 10.1016/j.nuclphysbps.2015.09.141 (cit. on pp. 75, 131).

[83] Simon Josefsson. *The Base16, Base32, and Base64 Data Encodings*. RFC 4648. Oct. 2006. doi: 10.17487/RFC4648 (cit. on p. 75).

[84] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX ATC'14. Philadelphia, PA: USENIX Association, 2014, pp. 305–320. ISBN: 9781931971102 (cit. on p. 77).

[85] Butch Quinto. High Performance Data Analysis with Impala and Kudu. In: *Next-Generation Big Data: A Practical Guide to Apache Kudu, Impala, and Spark*. Berkeley, CA: Apress, 2018, pp. 101–111. ISBN: 978-1-4842-3147-0. DOI: 10.1007/978-1-4842-3147-0_4 (cit. on p. 77).

[86] Zbigniew Baranowski, Luca Canali, Alvaro Fernandez Casani, Elizabeth J Gallas, Carlos Garcia Montoro, Santiago González de la Hoz, Julius Hrivnac, Fedor Prokoshin, Grigori Rybkine, Jose Salt, Javier Sanchez, and Dario Barberis. A prototype for the evolution of ATLAS EventIndex based on Apache Kudu storage. In: *EPJ Web of Conferences* 214 (2019). Ed. by A. Forti, L. Betev, M. Litmaath, O. Smirnova, and P. Hristov, p. 04057. DOI: 10.1051/epjconf/201921404057 (cit. on pp. 77, 79, 132).

[87] K. Masui, M. Amiri, L. Connor, M. Deng, M. Fandino, C. Höfer, M. Halpern, D. Hanna, A.D. Hincks, G. Hinshaw, J.M. Parra, L.B. Newburgh, J.R. Shaw, and K. Vanderlinde. A compression scheme for radio data in high performance computing. In: *Astronomy and Computing* 12 (2015), pp. 181–190. ISSN: 2213-1337. DOI: https://doi.org/10.1016/j.ascom.2015.07.002 (cit. on p. 79).

[88] Álvaro Fernández Casaní, Dario Barberis, Javier Sánchez, Carlos García Montoro, Santiago González de la Hoz, and José Salt. Distributed Data Collection for the Next Generation ATLAS EventIndex Project. In: *EPJ Web of Conferences* 214 (2019), p. 04010. DOI: 10.1051/epjconf/201921404010 (cit. on pp. 80, 132).

[89] Elizaveta Cherepanova, Evgeny Alexandrov, Igor Alexandrov, Dario Barberis, Luca Canali, Alvaro Fernandez Casani, Elizabeth Gallas, Carlos Garcia Montoro, Santiago Gonzalez De La Hoz, Julius Hrivnac, Andrei Kazymov, Mikhail Mineev, Fedor Prokoshin, Grigori Rybkin, Francisco Javier Sanchez Martinez, Jose Salt, Miguel Villaplana, and Alexander Iakovlev. The ATLAS EventIndex Using the HBase/Phoenix Storage Solution. In: *9th International Conference on Distributed Computing and Grid Technologies in Science and Education*. 2021, pp. 17–25. DOI: 10.54546/mlit.2021.68.25.001 (cit. on pp. 85, 87, 91, 92, 133).

[90] M Villaplana Perez, E Alexandrov, I Aleksandrov, Z Baranowski, D Barberis, G Dimitrov, A Fernandez Casani, E Gallas, C Garcia Montoro, S Gonzalez de la Hoz, J Hrivnac, I Alexander, A Kazymov, M Mineev, F Prokoshin, G Rybkin, J Sanchez, J Salt, and P T Vasileva. The ATLAS EventIndex and its evolution towards Run 3. In: *Journal of Physics: Conference Series* 1525.1 (Apr. 2020), p. 012056. DOI: 10.1088/1742-6596/1525/1/012056 (cit. on pp. 91, 98, 133).

[91] Apache Phoenix. *Apache Phoenix Tuning Guide*. Sept. 2022. URL: https://phoenix.apache.org/tuning_guide.html (visited on 09/01/2022) (cit. on p. 100).

[92] Álvaro Fernández Casaní, Carlos García Montoro, Santiago González de la Hoz, Jose Salt, Javier Sánchez, and Miguel Villaplana Pérez. Big Data analytics for the ATLAS EventIndex project with Apache Spark. In: *[Manuscript submitted for publication] Computational and Mathematical Methods* (2022). Presented at 2022 International CMMSE conference and the Second conference on high performance computing (CHPC). Awarded "Best computational applications on line presentation". ISSN: 2577-7408 (cit. on pp. 109, 134).

[93] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache Spark: A Unified Engine for Big Data Processing. In: *Commun. ACM* 59.11 (Oct. 2016), pp. 56–65. ISSN: 0001-0782. DOI: 10.1145/2934664 (cit. on pp. 109, 110).

[94] Dean Wampler. *Programming Scala*. O'Reilly Media, Inc., 2021 (cit. on p. 109).

[95] Sun Microsystems. *The JDBC Database Access API*. http://java.sun.com/products/jdbc/index.html. 2022. (Visited on 09/01/2022) (cit. on p. 110).

[96] 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. Santa Clara, California: Association for Computing Machinery, 2013. ISBN: 9781450324281. DOI: 10.1145/2523616.2523633 (cit. on p. 110).

[97] *Apache Phoenix ticket 6559: Spark connector access to SmallintArray / UnsignedSmallintArray columns.* `https://issues.apache.org/jira/browse/PHOENIX-6559`. (Visited on 09/01/2022) (cit. on p. 120).

[98] D Barberis, J Cranshaw, G Dimitrov, A Favareto, Á Fernández Casaní, S González de la Hoz, J Hřivnáč, D Malon, M Nowak, J Salt Cairols, J Sánchez, R Sorokoletov, and Q Zhang and. The ATLAS Eventindex: an event catalogue for experiments collecting large amounts of data. In: *Journal of Physics: Conference Series* 513.4 (June 2014), p. 042002. DOI: `10.1088/1742-6596/513/4/042002` (cit. on p. 131).

[99] D. Barberis, S.E. Cárdenas Zárate, J. Cranshaw, A. Favareto, Á. Fernández Casaní, E.J. Gallas, C. Glasman, S. González de la Hoz, J. Hřivnáč, D. Malon, F. Prokoshin, J. Salt Cairols, J. Sánchez, R. Többicke, and R. Yuan. The ATLAS EventIndex: architecture, design choices, deployment and first operation experience. In: *Journal of Physics: Conference Series* 664.4 (Dec. 2015), p. 042003. DOI: `10.1088/1742-6596/664/4/042003` (cit. on p. 131).

[100] González de la Hoz, Santiago, Acosta-Silva, Carles, Aparisi Pozo, Javier, del Peso, Jose, Fernández Casani, Álvaro, Flix Molina, José, Fullana Torregrosa, Esteban, García Montoro, Carlos, Lozano Bahilo, Julio, Montiel, Almudena, Pacheco Pages, Andrés, Sánchez Martínez, Javier, Salt Cairols, José, and Vedaee, Aresh. Computing activities at the Spanish Tier-1 and Tier-2s for the ATLAS experiment towards the LHC Run3 and High-Luminosity periods. In: *EPJ Web of Conferences* 245 (2020). Ed. by C. Doglioni, D. Kim, G.A. Stewart, L. Silvestris, P. Jackson, and W. Kamleh, p. 07027. DOI: `10.1051/epjconf/202024507027` (cit. on p. 133).