AGH

AGH UNIVERSITY OF SCIENCE AND TECHNOLOGY

FIELD OF SCIENCE: ENGINEERING AND TECHNOLOGY

SCIENTIFIC DISCIPLINE: INFORMATION AND COMMUNICATION TECHNOLOGY

DOCTORAL THESIS

Design and evaluation of data quality control methods for a very high bandwidth data acquisition and processing system in the future CERN/ALICE O2 framework

Author: Piotr Jan Konopka

First supervisor: prof. dr hab. inż. Marek Gorgoń
Second supervisor: dr hab. Jacek Otwinowski

Completed in: AGH University of Science and Technology, Faculty of Electrical Engineering, Automatics, Computer Science and Biomedical Engineering, Department of Automatic Control and Robotics

Kraków, 2021

I would like to offer my sincere thanks to Prof. Marek Gorgoń, Dr. Jacek Otwinowski and Barthélémy von Haller for their support and guidance during my doctoral studies, and for giving me this great opportunity to work at CERN.

Abstract

Modern physics experiments acquire very large amounts of data which require diligent quality monitoring and assessment. Data Quality Monitoring (DQM) systems help with identifying problems with particle detectors, data transfer and initial processing, where timely and accurate feedback is crucial. They are complemented with Quality Assurance (QA) systems, which allow to perform extensive assessment of the data quality before preparing them for physical analyses.

This dissertation covers the new data Quality Control (QC) framework for the upgraded ALICE experiment at the CERN LHC. Starting from the year 2022, the QC system will accompany the main computing software during the acquisition of a 3.5 TB/s raw data stream, its compression on-the-fly and final quality assessment. Thus, it combines the functionalities of DQM and QA within a complete system.

The QC framework is a highly parallel system, which can split computations among thousands of nodes. According to the reviewed literature, it is the first system of this kind to rely fully on message-passing and the actor model. In the dissertation, its general design and components are presented, giving the most detail to those which required novel solutions. A mathematical model of two available variants for multinode QC setups is proposed. The framework benchmarks results are presented and discussed.

Streszczenie

Współczesne eksperymenty fizyczne generują bardzo duże ilości danych, które są poddawane gruntownej kontroli jakości. Systemy monitoringu jakości danych (ang. Data Quality Monitoring, DQM) są wykorzystywane do szybkiej identyfikacji problemów dotyczących detektorów cząstek oraz transferu i przetwarzania danych podczas ich akwizycji. Z drugiej strony, systemy zapewniania jakości (ang. Quality Assurance, QA) pomagają w prowadzeniu całościowej kontroli zarejestrowanych danych zanim są uwzględniane w badaniach.

W rozprawiej doktorskiej przedstawiono nowy framework kontroli jakości danych (ang. Quality Control, QC) dla eksperymentu ALICE przy LHC w CERN. Oprogramowanie łączy w sobie cechy systemów DQM i QA. Począwszy od roku 2022, system kontroli jakości będzie wykorzystywany w trakcie akwizycji i przetwarzania strumienia danych o szerokości 3.5 TB/s oraz ostatecznej certyfikacji jakości.

Zaprezentowany framework umożliwia prowadzenie obliczeń równoległych na tysiącach węzłów sieci komputerowej. Jest to pierwszy tego rodzaju system opisany w literaturze oparty w całości o przekazywanie komunikatów (ang. message passing) i model aktorów. W rozprawie zaprezentowano jego architekturę i komponenty, ze szczególnym uwzględnieniem tych, które wymagały oryginalnych rozwiązań. Ponadto, zaproponowano model matematyczny dwóch dostępnych wariantów topologii aktorów oraz zamieszczono wyniki i dyskusję testów.

Contents

1. Introduction

Modern physics experiments often produce very large amounts of data. Only in November 2018 the data centre of the European Organization for Nuclear Research CERN (fr. *Organisation Européenne pour la Recherche Nucléaire*) recorded an unprecedented 15.8 petabytes of information [1]. While there is over a dozen experiments at CERN now, vast majority of the recorded data came from the detectors which register products of numerous particle collisions (Fig. 1.1) at the Large Hardon Collider (LHC)[2], the largest and most powerful particle accelerator in the world. Collecting such enormous data volumes is justified by searching for very rare signals, e.g. productions of heavy bosons and quarks.

Particle detectors designed for cutting-edge physics research are quite complex devices. They are actually composed of multiple sub-detectors which take advantage of different physical phenomena in order to observe certain particle characteristics and then convert them into electric signals. Properties of detectors change over time, in a short and long term - noise levels vary, electronic components break, their sensitive elements degrade due to ionizing radiation, etc. Retrieving raw data makes only the first step of the event reconstruction process, when electronics signals from detectors readout channels are digitized. Then the following reconstruction steps, including signal clusterisation, tracking and particle identification, are performed. While data acquisition takes place synchronously to the activity at the LHC, the further data processing and preparation might continue days after. Event reconstruction software also tends to be very complex, reaching thousands of hundreds lines of code (see e.g. the ALICE reconstruction software (AliRoot) repository [3]), which leaves plenty of room for potential problems and bugs.

These facts definitely raise some doubts about correctness of whole experimental process. If particle detectors may malfunction and so may their processing software, how experimental crews should know that they acquire valid data and process it correctly instead of wasting costly equipment and time? Moreover, should we believe experimental physics results which are based on a chain of so many, potentially faulty components and steps?

These issues are usually addressed by implementing data quality control systems. The existence of Data Quality Monitoring (DQM) systems helps with quickly identifying and overcoming problems with detectors, data transfer and processing during acquisition. They are usually designed to provide feedback information in timely manner, so experiment shift crews and detector experts can spot and fix potential problems to collect good quality data. During data preparation for analysis, Quality Assurance (QA) is held. While it should not unnecessarily delay the full process, this stage may take longer time, so acquired

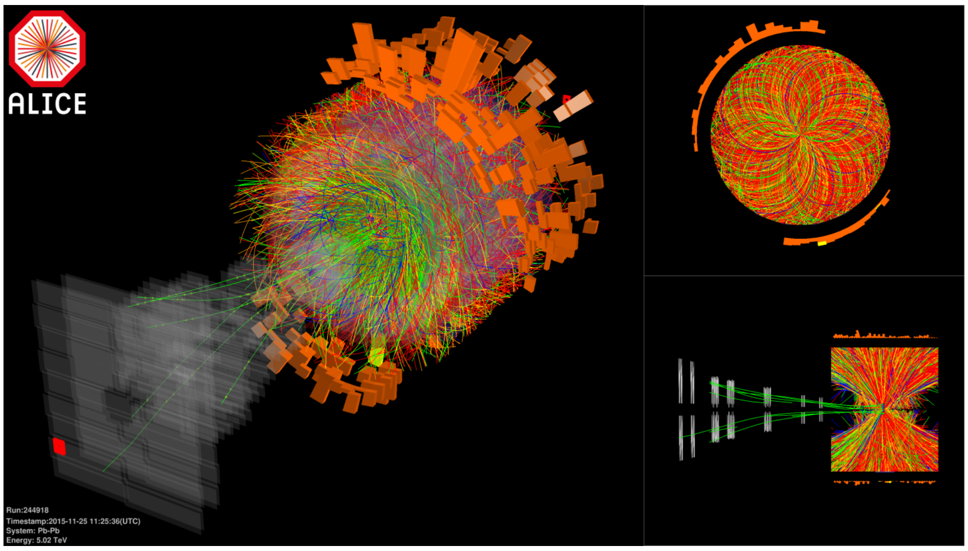


Fig. 1.1. A graphical representation of a heavy ion collision recorded by the ALICE detector (the experiment's press materials).

data are validated with best possible knowledge and physicists may rely on them when conducting their research. These two classes of software may be more or less intertwined, which largely varies between particular physics experiments. They have long lifetimes - the original quality control systems of the four major LHC experiments were used since the beginning of the data taking in 2009, until the shutdown in 2018.

The ALICE experiment [4] is having a major upgrade before the LHC resumes its operation in early 2022. Most of the sub-detectors is being modernised or completely replaced in order to achieve a higher spatial and time resolution, and much greater amount of acquired data [5]. The updated experimental setup will be able to observe 50000 led ion collisions per second and produce 3.5 TB/s of raw data in such conditions. To sustain the new data throughput requirements, a new computing system and associated software are being prepared. Consequently, the process of data quality control was redesigned, encompassing the two previously separated systems, DQM and QA, into one - Quality Control (QC).

1.1. Dissertation context and aim

This thesis was realised following the joint CERN and AGH UST doctoral programme. The aim of the project was to investigate and propose solutions to the most challenging aspects of the new Quality Control system developed for the ALICE experiment. First, the author got acquainted with the state of art in the field of Data Quality Montoring and Assurance in High Energy Physics (HEP) experiments. Then, he reviewed the initial state of the QC framework [6]. Finally, he proposed changes and implemented the following components:

– *Data Sampling* – the component collecting data samples to feed the QC system in an efficient, statistically sound and reliable way.

– *Automatic checkers* – the framework component which allows to analyse statistical data structures generated by the QC system, such as histograms and decide whether they indicate a good or bad quality of the data.

– *Correlation and trending (post-processing)* – the infrastructure to post-process data generated by QC in dedicated asynchronous processes. Physicists often need to see transformed data over time (trending) and to correlate different observables.

Apart from having successfully delivered the aforementioned framework components, the author also took over the following tasks:

– *Quality Control Tasks* – the framework component which allows to run algorithms generating data structures based on sampled data. The initial prototype was further developed and extended.

– *Merging software* – the infrastructure to merge results of algorithms running in parallel on multiple cores and/or processing nodes. This functionality was also needed by other parts of the new computing system. The first prototype proposed in [7] was reworked and put under extensive performance benchmarks.

– *Infrastructure management* – finding methods which help to choose the best arrangement of message-passing processes for performing the tasks mentioned above in terms of computing, memory and bandwidth resources. Also, designing the QC framework components in a way which allows for their reconfiguration during runtime.

The Quality Control framework was benchmarked in order to evaluate validity of the applied solutions. Moreover, the author took part in many discussions regarding the general ALICE software framework development [8] and contributed with multiple bug fixes and small extensions.

The entirety of the described work led the author to put forward the following thesis:

Parallelisation of data quality control systems allows them to sustain very large data throughputs in data acquisition systems.

The Data Sampling software was presented in the form of a poster at the 19th International Workshop on Advanced Computing and Analysis Techniques in Physics Research (ACAT), then extended and published in [9]. A detailed study regarding the merging software was submitted to a peer-reviewed journal and is in the review process, as of January 2021. The complete Quality Control system was presented at the 24th International Conference on Computing in High Energy and Nuclear Physics and published in peer-reviewed proceedings thereof [10].

1.2. Content description

The dissertation is split into 9 chapters and 2 appendices. The context of this work, which is outlined in this chapter, is followed in detail in Chapter 2. It contains an introduction to the research conducted at

CERN from the computer science point of view. Then, the recent modernisation activities of the ALICE experiment are described. Most important detector upgrades are mentioned and the new computing system architecture is explained. Finally, the new data Quality Control system design is presented as it was described in [6], before the author began his work. The QC software relies on several of frameworks, libraries and other underlying technologies, which are described in Chapter 3. The state of the art in the field of Data Quality Monitoring and Assurance systems was extensively covered in Chapter 4.

The next two chapters make the body of the dissertation. Chapter 5 contains a detailed description of the QC framework in its latest version, including studies of its most crucial components. As the implemented piece of software leaves a certain amount of flexibility in the way it is executed, it received a mathematical model, described in Chapter 6. There, the effects of different variables on the performance were evaluated and methods to find the optimal software configuration were proposed.

The performance of the QC framework was benchmarked to assess if the design and implementation choices indeed allow to sustain the unprecedented data rates of the new ALICE data acquisition system. The benchmarks results are presented and discussed in Chapter 7. Chapter 8 contains the latest statistics about the QC framework usage and provides the reader with the most interesting applications of the software so far. The dissertation is concluded in Chapter 9, where all major achievements of the work are highlighted and the discussion about potential improvements is held.

The thesis is complemented with two appendices. App. A contains a list of abbreviations and uncommon terms used in the dissertation. In App. B, benchmark results of the commonly used data types in the QC system are presented and discussed.

2. CERN and the ALICE experiment upgrade

In this chapter an introduction to the environment of the described data Quality Control system is presented. The details of the particle accelerator, the new ALICE detector and its new computing system are provided and put into context. Moreover, the software architecture overview of the Quality Control systems is also presented, followed by the computational requirements which come from the chosen approach.

2.1. CERN and Large Hadron Collider

The European Organization for Nuclear Research CERN is located on the Franco-Swiss border in the proximity of Geneva. During its more than 60 years history, this scientific institute greatly contributed to the development of particle physics and information technologies. The research is funded by 23 member states, including Poland, since 1991.

Without doubt, CERN is most famous for its Large Hadron Collider - a particle accelerator located in a 27 km torus-shaped tunnel. The apparatus is able to accelerate protons (heavy-ions) up to the energy of 7 (5.5) TeV per nucleon pair [2]. Two particle beams moving in parallel in the opposite directions cross each other in four collision points. The four major detectors and experiments: ALICE [4], ATLAS [11], CMS [12] and LHCb [13] are located around them. Each is designed to study various physics phenomena. For example, the ATLAS and CMS experiments empirically confirmed the existence of the Higgs boson [14][15], which is responsible for generation of current quark masses creation in the Standard Model [16].

The LHC schedule is organised in operational runs and long shutdowns, both lasting several years. The first run took place in years 2009-2013, then it was followed by the first Long Shutdown (LS1). The second run lasted from 2015 until the end of 2018. The LHC generates particle collisions during these operational runs, letting experiments record physics data.

The LHC working cycle (Fig. 2.1) commences with the insertion of particle beams into the accelerator, then radiofrequency cavities increase their energy so they can reach a target value. After final calibrations, the beams are squeezed near the interaction points and particle collisions start to appear. They last until there is not enough particles in the beams to keep high collision rates or other technical problems occur. Then the beams are dumped into two blocks of graphite and the LHC shift crew prepares the machine for the next injection. Under favourable circumstances, the accelerator may generate

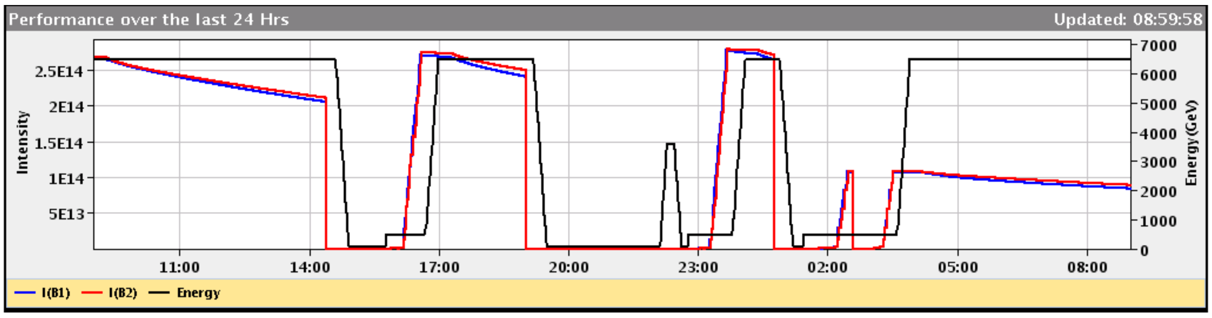


Fig. 2.1. An example of the LHC performance plot. **The blue and red lines** indicate the intensities of the two parallel beams. When the lines rise, particles are injected to the accelerator ring. A slow decrease usually can be observed when the beams are collided, so the amount of particles declines (data is acquired then). Sudden intensity drops correspond to beam dumps. **The black line** indicates the expected beam energy, driven by radiofrequency cavities. It is gradually increased after particles are injected, then kept steady until the beams are dumped.

collisions continuously for between 10 and 20 hours. During the operational runs, it works 24 hours a day, 7 days per week with the exception of end of year breaks.

2.2. The ALICE experiment

The ALICE experiment (A Large Ion Collider Experiment) gathers 174 institutes and universities with the aim of studying quark-gluon plasma - an exotic state of matter, which is said to exist shortly after the Big Bang. It was confirmed that one can form quark-gluon plasma in high-energy led (Pb) ion collisions [17][18]. The detector (Fig. 2.2) provides information about the general features of a collision and properties of observed particles such as charge, momentum, mass and energy. The ALICE detector and the data acquisition system which used to operate in years 2009-2018 were able to observe up to 8000 heavy-ion collisions in a second and trigger recording of around 400 of those, which would correspond to roughly 17 GB/s of data throughput, then compressed to almost 5 GB/s. By having large statistics of such events it is possible to study quark-gluon plasma using large variety of physics observables [19].

2.3. Lifetime of physics data

Studying the physics phenomena occurring during and after particle collisions makes the last step of a long process of acquiring and preparing data, which begins with running an experiment.

The operation schedule of the LHC experiments is closely aligned to the collider’s calendar. During operational runs, detectors and data acquisition systems have to follow and adapt to the LHC state (Fig. 2.1). When the accelerator team announces that the beam injection is planned in a short time, the detector is prepared accordingly by activating all its components and performing their final calibration. Analogously, the data acquisition hardware and software is started and configured, so it can transition

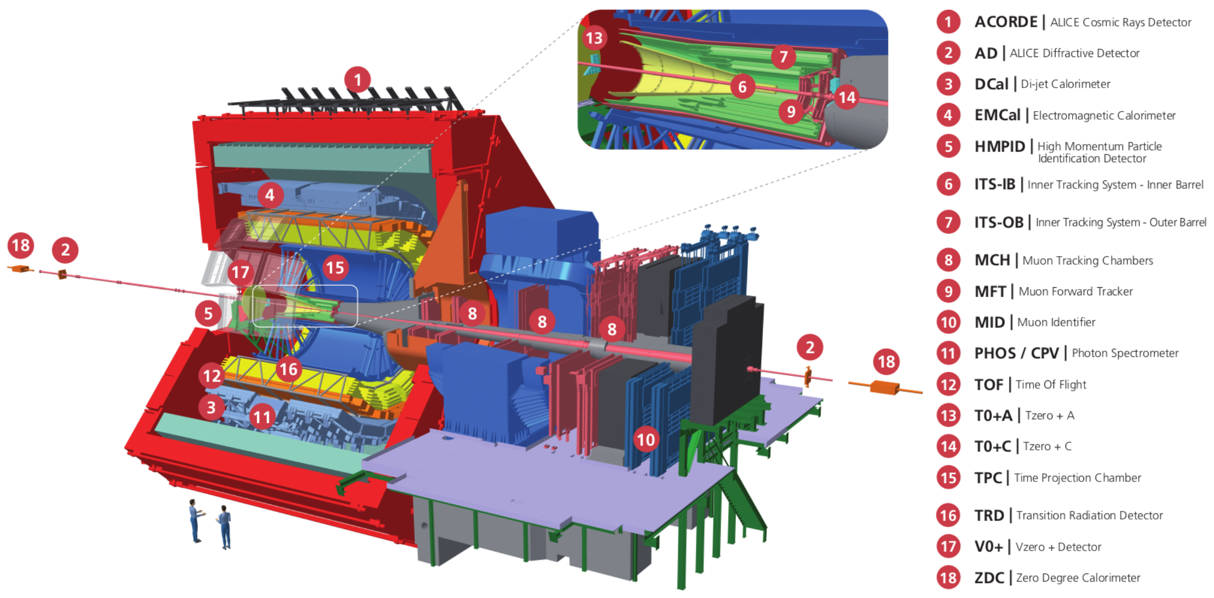

1 ACORDE | ALICE Cosmic Rays Detector
2 AD | ALICE Diffractive Detector
3 DCal | Di-jet Calorimeter
4 EMCal | Electromagnetic Calorimeter
5 HMPID | High Momentum Particle Identification Detector
6 ITS-IB | Inner Tracking System - Inner Barrel
7 ITS-OB | Inner Tracking System - Outer Barrel
8 MCH | Muon Tracking Chambers
9 MFT | Muon Forward Tracker
10 MID | Muon Identifier
11 PHOS / CPV | Photon Spectrometer
12 TOF | Time Of Flight
13 T0+A | Tzero + A
14 T0+C | Tzero + C
15 TPC | Time Projection Chamber
16 TRD | Transition Radiation Detector
17 V0+ | Vzero + Detector
18 ZDC | Zero Degree Calorimeter

Fig. 2.2. The ALICE detector scheme (the experiment's press materials).

into running state as promptly as possible. When the LHC declares the presence of *stable beams*, the experiment begins to record data and it continues so until the beams are dumped or an unexpected technical problem appears.

In the case of the ALICE experiment, the detector is usually supervised by trained crews consisting of 4-5 people which take 8-hour shifts, including nights, weekends and holidays. The shifters are mostly employees of CERN or other institutes taking part in the experiment. A shift crew in ALICE consists of:

– an Experiment Control System (ECS) Shifter, who is in charge of supervising the data acquisition system and attempts to solve basic problems with its operation,

– a Detector Control System (DCS) Shifter, a person responsible for the safety and correct behaviour of the detector itself,

– a Data Quality Monitoring (DQM) Shifter, who makes sure that the recorded data are of good quality and helps to quickly identify problems with the detector and the data acquisition system,

– a Shift Leader (SL), who has a global understanding of the detector system, including the responsibilities of the three aforementioned shifters, and takes any decisions and actions needed to guarantee the best efficiency of the detector,

– a Run Coordinator (RC, optionally), a person who coordinates the experiment operations within the collaboration and with the accelerator teams.

The shifters work in the ALICE Control Room (ARC), which is located on the experimental site. In case they cannot solve a problem with a sub-detector or one of the computing systems, they may call corresponding experts for help, who take 24-hour on-call shifts for specific sub-systems.

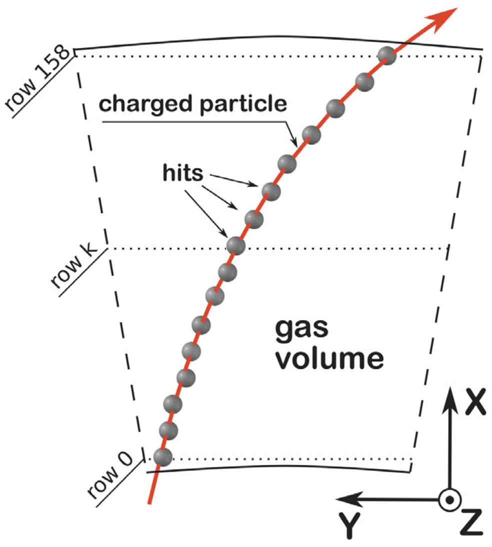


Fig. 2.3. An example illustrating tracking in one of the ALICE sub-detectors [21]. The particle hits are connected into lines which approximate the real particle tracks.

The data acquisition system receives a raw detector data stream. Its format and contained information varies significantly across sub-detectors, but it usually contains rudimentary data about time, location and strength of electric signals been observed. In order to be efficiently analysed by physicists, these data are be transformed into easily accessible information about tracks and properties of the particles observed by the detector. This process is called reconstruction [20][21]. It may be performed synchronously to data-taking or afterwards, if raw data is preserved.

The first step of reconstruction usually involves clusterisation. When a particle crosses a detector it may leave a trace on multiple adjacent detection cells or, in case of calorimeters, cause a particle shower. In both cases, such a group of signals corresponds to one particle hit (energy deposition), therefore they are combined into clusters. With this information, one can compute a more precise position of a particle hit and the energy which it deposited.

Having identified the particle hits, the software proceeds with finding the primary vertex (collision point coordinates), recognizing the tracks and fitting them to the corresponding hits. Data from different sub-detectors are matched. The ALICE detector may observe up to 8000 particles in one Pb-Pb collision, which makes the tracking particularly difficult and resource-demanding. Moreover, some particles appear away from the primary vertex, as decay products of undetectable particles. In the last step, the particles are identified based on the curvature of their tracks, energy deposition, time of flight and other information. The collected data requires calibration, which is derived from additional calibration runs and is complemented with data obtained during reconstruction.

Physicists may analyse large quantities of the reconstructed data by using the Worldwide LHC Computing Grid (WLCG) [22]. The acquired data is usually complemented with simulated data with the same condition and calibration parameters, which allows the researchers to estimate the detector efficiency and the statistical significance of analysis results [23].

2.4. ALICE detector upgrade

Since December 2018, the LHC is in the Long Shutdown 2 (LS2), designated for an exchange and modernisation of its components and an improvement of the machine's working parameters [24]. The third LHC run (Run 3) is expected to start at the beginning of 2022.

The shutdown creates also a good opportunity for an extension and modernisation of the ALICE experiment setup. Statistical analyses in high energy physics often rely on a very high amount of data in order to provide statistically significant results. The ALICE physics programme includes searching for very rare signals [5], which requires large statistics of collision data. Therefore, most of the sub-detectors are being modernised or completely replaced to achieve higher spatial and time resolution, and a much higher data readout rate. To mention the most important upgrades, the new Inner Tracking System (ITS)[25], which is the detector closest to the collision point, will consist of 7 cylindrical detection layers, each covered with silicon-based Monolithic Active Pixel Sensors (MAPS) - square-shaped pixels with sides of 30 µm. In total, the sub-detector will have 12.5 billion pixels spanning on an active surface of about 10 m^2 and will be able to sustain the rate of 100 kHz of heavy ion collisions and 400 kHz of proton collisions.

A new sub-detector, the Muon Forward Tracker (MFT), will be incorporated into the existing detector setup [26]. It will make an extension to the existing muon detector set by giving it vertexing capabilities, thus allowing for new measurements, which were not accessible with the previous apparatus. The detector is built from the same MAPS sensors as the ITS.

The Time Projection Chamber (TPC) upgrade [27], a gas detector to track charged particles, will make the most impactful change. Due to the physics phenomenon used to detect particles - registering electrons from ionised gas drifting in the detector - there is a significant latency since the appearance of a particle until its detection. It reaches 125 µs, which greatly inhibits data acquisition in triggered mode - when each collision is quickly evaluated and decided if it is worth recording. As soon as the data acquisition is triggered, succeeding collisions cannot be evaluated until the current one is completely registere. In this case, it limits the trigger rate to less than 3.5 kHz. The new TPC will sustain interaction rate of 50 kHz with heavy-ion beams. However, to actually register more events, the sub-detector will produce a continuous, not triggered flow of data, and by that, increase the amount of generated data by two orders of magnitude.

The MFT will occupy a part of the space previously used by the trigger detectors, thus they will be replaced with the new, more compact Fast Interaction Trigger (FIT) detector [28]. It will be used as the main forward trigger and to measure the collision time and centrality, and to monitor LHC beam luminosity. Due to the high demands of the other sub-detector upgrades, the FIT will have to maintain its latency below 425 ns, where more than a half will be induced by the signal propagation time in cables, leaving just 200 ns of the available processing time.

The new ALICE detector is estimated to produce 3.5 TB/s of raw data, which requires a new data acquisition and processing system, operating on a largely extended computing farm.

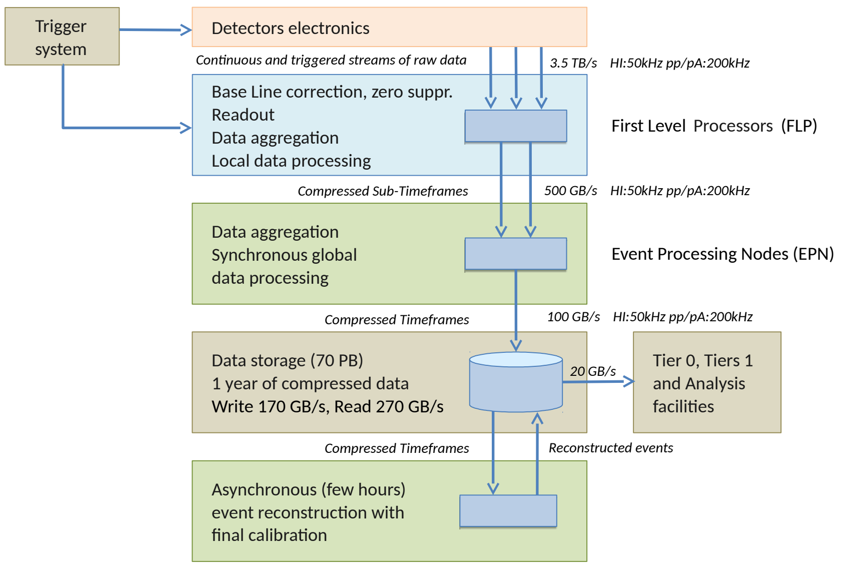

Trigger system
Detectors electronics
Continuous and triggered streams of raw data
3.5 TB/s HI:50kHz pp/pA:200kHz
Base Line correction, zero suppr.
Readout
Data aggregation
Local data processing
First Level Processors (FLP)
Compressed Sub-Timeframes
500 GB/s HI:50kHz pp/pA:200kHz
Data aggregation
Synchronous global data processing
Event Processing Nodes (EPN)
Compressed Timeframes
100 GB/s HI:50kHz pp/pA:200kHz
Data storage (70 PB)
1 year of compressed data
Write 170 GB/s, Read 270 GB/s
20 GB/s
Tier 0, Tiers 1 and Analysis facilities
Compressed Timeframes
Reconstructed events
Asynchronous (few hours) event reconstruction with final calibration

Fig. 2.4. The O^2 system architecture [29].

2.5. The new online-offline computing system

In order to cope with such requirements, a new Online-Offline Computing system, called O^2 [29], is being developed. It is characterized by the continuous readout of all interactions, their compression by means of partial online reconstruction and calibration, and sharing of common computing resources during and after data taking.

As shown in Fig. 2.4, there are two major computing layers, the *First Level Processors* (FLPs) and the *Event Processing Nodes* (EPNs). Both are highly heterogeneous, with specialized acquisition cards embedding FPGAs on the FLPs and GPUs on the EPNs. A temporary data storage facility, consisting of 70 PB disk space, will receive around 100 GB/s of data, store them until further compressed and then transfer to the main data centre at CERN.

The detector electronics located in the experimental cavern send raw data via optical fibers to the first computing farm, which contains almost 200 FLPs. Common Readout Units (CRUs), FPGA cards, receive data, optionally perform some pre-processing and transfer them into the computer memory of the FLPs. Then, data is aggregated into *Sub-TimeFrames* (STFs), which contain all event information gathered from a set of fiber links in a certain amount of time. Detector-dependent processing algorithms reduce or compress most of the raw data, sometimes irreversibly. STFs matching the same time periods reach one of the EPNs, where they are combined into *TimeFrames* (TFs) and further processed. With the use of GPUs, the EPNs perform the most computationally heavy consuming operations, such as tracking. The O^2 system design assumes that the same computing resources are used during data-taking (online) and while waiting until the LHC injects the particle beams again (offline), which gives the *Online-Offline* component in its name. During acquisition the EPNs have limited time available for data processing, therefore partially compressed TimeFrames are temporarily stored in the ALICE storage facility at the experimental site. Then, the additional processing is performed asynchronously on the EPNs or on the WLCG. When the acquired data are fully reconstructed, they make their way to permanent storage and analysis facilities.

The O^2 system relies on a number of software components. The ALICE Experiment Control System (AliECS)[30] is responsible for the automated management of the computing cluster resources and all applications running within. Monitoring [31] collects and visualizes information about the software and hardware performance in order to provide a complete overview of the system health and help identifying failures of its components. InfoLogger [32] takes care of collecting and aggregating log messages coming from the O^2 processes. It is complemented with a highly functional GUI which allows to browse and filter logs. The Configuration system stores and distributes nested key-value pairs to configurable processes. The O^2 framework [8] consolidates most of data processing in a form of message-passing topologies and hides technical difficulties of implementing the data transport underneath.

2.6. Data Quality Control and Assessment

In such an immensely complex system problems might occur at each step of data acquisition. Moreover, during the synchronous reconstruction most of raw data will be irretrievably removed. Therefore, potential problems with detector performance or data processing should be immediately identified by the Quality Control system to collect good quality data. It should allow to quickly recognize issues with data during acquisition and determine the final data quality for later physics analyses.

2.6.1. Definition

The Data Quality Control and Assessment (QC) replaces the former online Data Quality Monitoring (DQM) and offline Quality Assurance (QA), used in the ALICE experiment during previous years. The QC [6] should confirm that collected data has good quality and if not, identify potential problems with the detector and processing in a timely manner, especially when running synchronously with the data taking. It should also provide tools to track changes and correlate different properties of detectors in order to allow for a high-level analysis of their performance.

The unification of the online and offline worlds, as well as the discarding of raw input data in favour of reconstructed data, make the need for a reliable data quality control even more essential. The challenge is made greater due to the more than 15 different detector teams involved and the very large amount of data to look after (3.5 TB/s).

2.6.2. Architecture overview

The originally proposed design of the Quality Control [6] is split into a number of components shown in Fig. 2.5. The most important requirements for each component are mentioned.

The *Data Sampling* (blue arrows connected to purple boxes) is responsible for selecting and distributing data samples according to the needs of the proceeding components. Sampling should be possible at each processing step, which is especially important in case of non-permanent data. It may not slow down the main processing, unless explicitly specified. The software should provide a random, statistically sound representation of data and also allow to implement more advanced sampling techniques. The *QC tasks* (purple boxes) execute detector-specific algorithms either locally on the FLPs and the EPNs or remotely on dedicated Quality Control Servers. They publish their results in a form of data structures, such as histograms or tables, generally referred as QC Objects or Monitor Objects. As most Tasks run in parallel on many nodes, their incomplete results have to be merged. The *Mergers* perform this task. The *Checkers* then take care of evaluating the quality of the objects by running user algorithms developed under a common interface. Finally, the QC Objects and the Qualities are stored in the QC repository. Two components - *Correlation* and *Trending* - can post-process data derived from QC Objects and Qualities stored in the repository and inject the results back into the processing chain for additional checks. They are triggered periodically, manually or on certain events (e.g. start of a data acquisition run or end of an LHC fill). Once the QC Objects are stored, along with some Quality Objects, the shifters and experts can use the QC GUI (QCG) to visualize them.

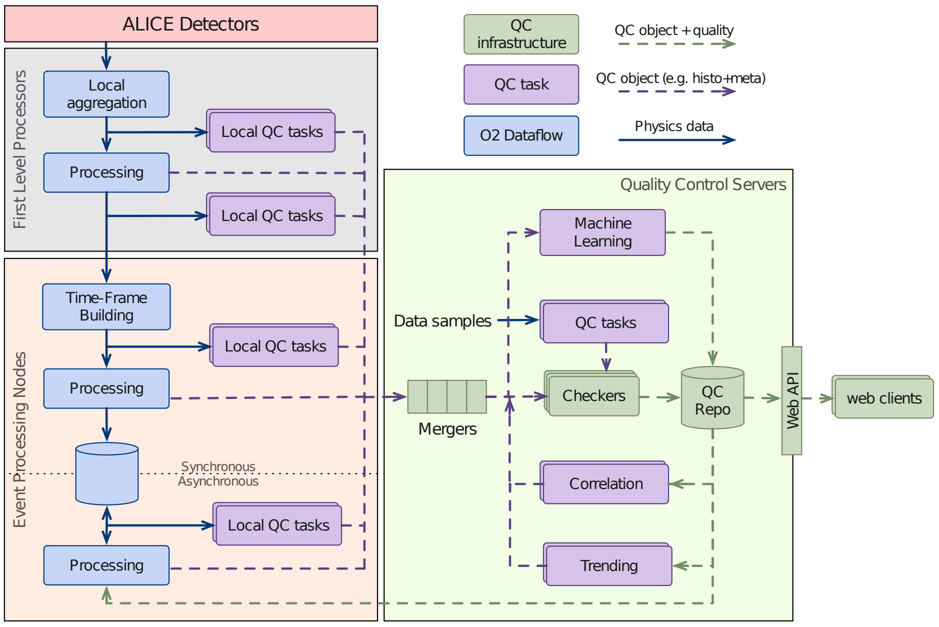


Fig. 2.5. The QC architecture as presented in [6].

2.6.3. Data rates estimations

The new Quality Control system should be able to cope with an unprecedented data throughput. To achieve this feat, beginning with the main processing data, each processing step should reduce the data volume (see Fig. 2.6).

The latest estimations show that the full O^2 system will receive around 3.5 TB/s of raw data input. During the online processing, the recorded information will be reduced down to 635 GB/s in the FLPs and then even further to 100 GB/s in the EPNs. However, additional temporary data might appear in the midst of processing. Message rates will vary significantly across the system as a direct consequence of different data granularity (being as little as 2 MBs up to a dozen of GBs). Assuming 2 MB payloads, one process might generate around 7500 messages per second.

According to surveys conducted with the sub-detector experts, the complete system should consist of around 100 distinct QC tasks. Mostly, they will use between 1% and 10% of messages of a selected data type, although in rare cases they will work correctly only with the full data stream. A QC Object will have the size of 250 kB on average. All QC Tasks which run in parallel will produce incomplete results that Mergers will have to combine into 10000 complete objects each minute. The QC Tasks are expected to produce in total about 25000 complete objects updated each minute. These objects will be analysed resulting in around 100000 lightweight Quality Objects, needing only a few kB each.

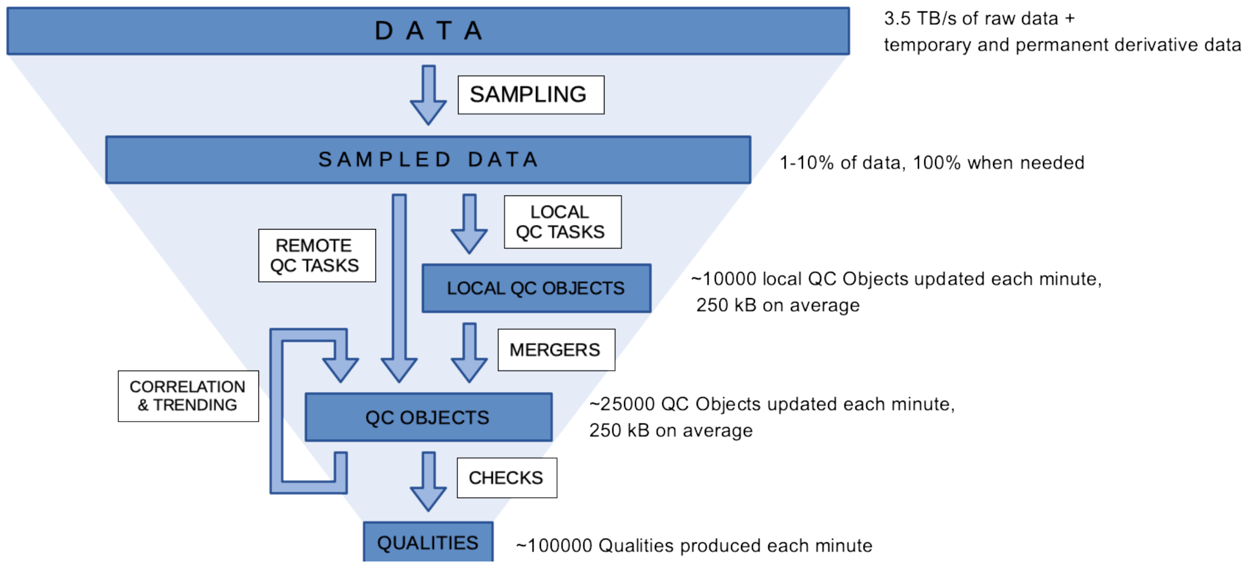

D A T A
3.5 TB/s of raw data +
temporary and permanent derivative data
SAMPLING
SAMPLED DATA
1-10% of data, 100% when needed
LOCAL QC TASKS
REMOTE QC TASKS
LOCAL QC OBJECTS
~10000 local QC Objects updated each minute, 250 kB on average
MERGERS
CORRELATION & TRENDING
QC OBJECTS
~25000 QC Objects updated each minute, 250 kB on average
CHECKS
QUALITIES
~100000 Qualities produced each minute

Fig. 2.6. The QC processing chain and its data rates [10].

3. Message-passing in the O^2 system

3.1. Actor model

The actor model was first proposed in 1973 as an architecture for artificial intelligence applications [33]. This concept has been gaining popularity ever since. It allows to divide processing tasks into several meaningful components with clearly separated responsibilities, thus also facilitating the use of parallel computing resources. As the O^2 system relies on this model to a large extent, its basic assumptions and benefits are presented in this section.

Actor is the basic unit of computation. It has one or more addresses, which can be used e.g. to communicate with other actors (and also with itself) by receiving and sending messages. Incoming messages have an indeterministic time of arrival and can be sequentially processed by an actor, in the order of arrival. An actor may decide how to treat future messages, thus it has a state. It may also create other actors.

Implementing a complex computation task within the actor model helps to divide it into clearly separated processing units which can operate independently. This means that topologies of actors can well use the multicore architecture of modern processors and even large computer farms. Since actors can influence each other via messages only, they do not require a strict synchronisation, but they might have to be started and stopped in an order which guarantees that all messages are processed.

3.2. Message passing software stack in O^2

This section gives a broad perspective on the software stack used in the O^2 system and should help to understand how the Quality Control software is implemented and how it interacts with other components.

3.2.1. Messaging in Linux systems

The Linux operating systems family [34] strongly dominates the high performance computing market. All of the most powerful 500 supercomputers use one of the Linux distributions [35]. CERN maintains a customised CentOS distribution which is suited for its computing environment [36]. The O^2 system is expected to run on CERN CentOS 7 or 8.

Unix-based systems offer several ways to interface separate computer programs [37], which might serve as building blocks of higher-level message passing libraries and frameworks. Some of them allow to connect processes within one computing node, while others are designated for computer networks.

Probably the easiest form of communication in Unix systems can be achieved by using ordinary files. A process might create and/or write into a file, closing it after having finished. Other applications might read the file while it is being written into or anytime after. Operations on files are however costly, as they require an access to a filesystem using system calls. Memory-mapped files show an increased I/O performance by avoiding system calls and allowing to treat file contents as a piece of memory.

Pipes are unidirectional data channels. Anonymous pipes can connect a standard output of one process with a standard input of another, which is expressed by inserting the pipe character "|" between shell commands executing respective processes. This approach allows to interface only applications running within one shell command, which is, however, not a restriction of named pipes. These, also called FIFOs (First In First Out), might be created at any time and appear as standard files. On modern Linux distributions a pipe buffer can contain 64kB of data by default. The communication is blocking, unless specified otherwise. In contrast to standard files, data travelling through FIFOs are not stored in filesystem, but only in kernel, effectively allowing to achieve higher throughput and lower latency.

Linux offers also the socket API - endpoint interfaces for sending and receiving messages within one computer node or a network. Unix domain sockets allow for stream-oriented and datagram-oriented inter-process communication. Internet sockets implement popular networking protocols: Transmission Control Protocol (TCP), Stream Control Transmission Protocol (SCTP), User Datagram Protocol (UDP), which can operate both locally and between separate computing nodes.

The previously described inter-process communication methods always involve kernel in passing data. In order to communicate processes directly and possibly achieve higher performance, one can create a shared memory region which is accessed and modified by a group of co-owners. While being the fastest, this method also makes a developer responsible for synchronisation of read/write operations on a shared memory region, as well as its proper clean-up after use. On Linux, it appears as a mounted file system under the path `/dev/shm`, which uses virtual memory instead of a real storage device.

3.2.2. ZeroMQ

ZeroMQ is a free and open-source messaging library [38]. It supports the most popular operating systems, including Windows Server, multiple Linux distributions and MacOS. The core engine is written in C++, but it offers various bindings for other languages, with C, C#, Java, Python, Go, Node.js and Ruby among others.

The main feature of the library is its high-level socket application programming interface (API). By using different kinds of sockets, one can create applications which exchange messages and signals in the following fundamental communication patterns [39]:

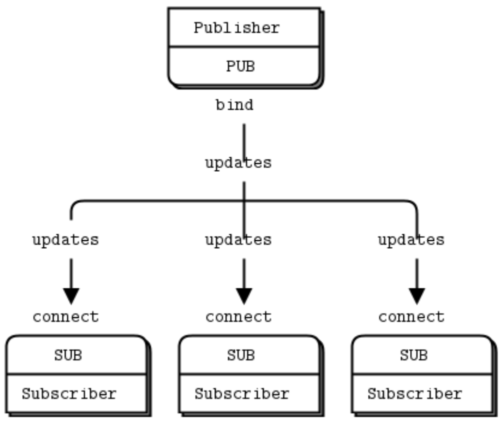


Fig. 3.1. An example of a pub-sub pattern [39].

– Publish-subscribe (PUB/SUB) – connects a set of publishers to a set of subscribers. It is a data distribution pattern.

– Pipeline (PUSH/PULL) – connects nodes in a fan-out/fan-in pattern that can have multiple steps and loops. It is a parallel task distribution and collection pattern.

– Request-reply (REQ/REP) – connects a set of clients to a set of services. It is a remote procedure call and task distribution pattern.

– Exclusive pair (PAIR) – exclusively connects two sockets in the same process, but different threads.

The first two have the most relevance to the O^2 system and will be described in more detail in this subsection.

The library does not impose any format to transferred data - they are treated as binary blocks of any size, including 0. A developer can choose an optimized transport type accordingly to their use-case. These include in-process, inter-process, TCP and multicast communication. Each socket has corresponding message queues (or buffers), where incoming and outgoing data are kept before being received and sent, respectively. Therefore, buffering messages protects them from being dropped when a receiver is not able to respond immediately. So called *high-water marks* prevent queues from consuming too much memory by limiting the number of messages inside. If a socket consumes data from multiple sources, the queues have equal-priority due to the fair-queuing mechanism, which organizes receiving messages in the round-robin order. In most cases ZeroMQ allows to create and destroy sockets in arbitrary order. Also, they can manually or automatically reconnect without unnecessary disturbances in other nodes.

The PUB/SUB communication pattern allows to connect a set data publishers with another set of data subscribers (Fig. 3.1). Published messages are duplicated among listening sockets, so each subscriber receives each message. The pattern is non-blocking - when queues between communication nodes become full, a publisher trying to send more messages will drop them without waiting. Furthermore, if there are no subscribers connected to a publisher, the latter will not store outgoing messages in a buffer.

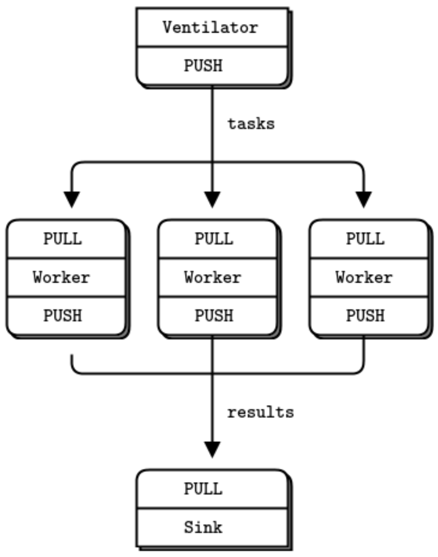


Fig. 3.2. An example of a pipeline (push-pull) pattern [39].

The O^2 system also relies heavily on the pipeline pattern, often called PUSH/PULL due to its socket types names (Fig. 3.2). It allows to distribute a workload among multiple workers. A set of *pull* sockets receive messages produced by a ventilator in round-robin order, therefore, a particular message is seen by only one worker. Processing results are gathered by a common sink with another *pull* socket. In contrast to PUB/SUB, here the communication is blocking - having full output buffers forces the sender to wait until the congestion resolves (with an optional timeout). This guarantees that no data is lost during the transport, even if there is no node available to receive new messages.

The ZeroMQ authors list Microsoft, Samsung, AT&T, Spotify, Digital Ocean, Auth0 among companies which use their library. There is also an increasing trend to rely on ZeroMQ as the communication fabric in software frameworks of data acquisition systems in high energy physics experiments, e.g. in DAQling [40] and the library discussed in the next section.

3.2.3. FairMQ

FairMQ [41][42] is an open-source message queuing library developed in GSI Helmholtz Centre for Heavy Ion Research, which maintains a strong connection with the ALICE experiment. By implementing the actor model, the library authors aim to tackle the large data processing requirements of the recent particle physics experiments. It is written in C++.

The basic building block of the framework is *FairMQDevice* (or just *device*), an independently running application which can receive and send messages to other devices, and process them with user-defined algorithms. Multiple devices connected to each other can form a data processing topology, naturally taking advantage of multi-core processors. The library supports communication patterns offered by ZeroMQ - pipeline, publish-subscribe, request-reply and exclusive pair. Message passing is agnostic to data format and gives the responsibility of defining a data model to the developer.

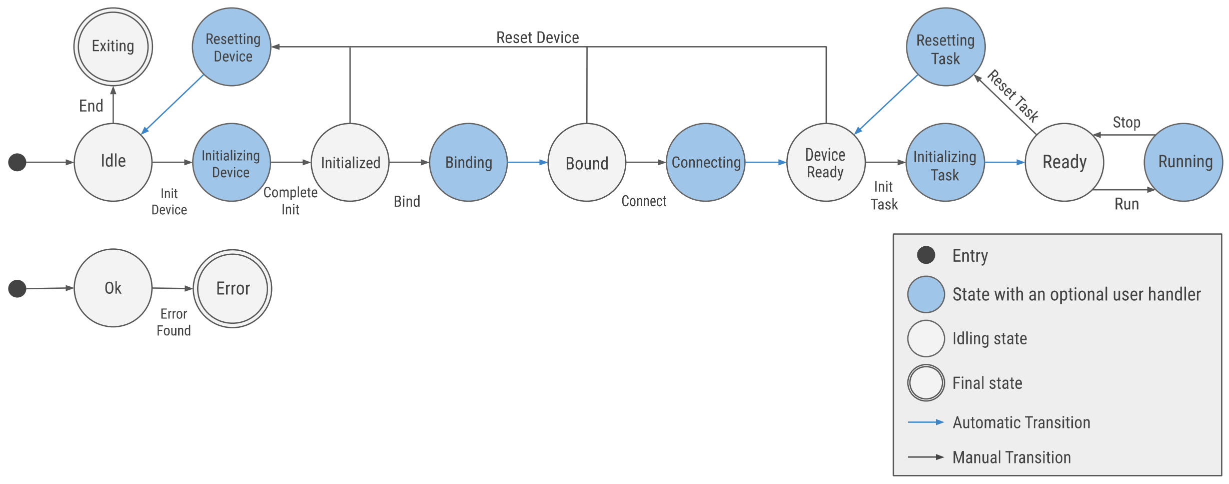


Fig. 3.3. The state machine of FairMQDevice ([42], increased font size).

Two transport types are supported - the first is based fully on ZeroMQ, the second implements shared memory communication with the help of `boost::interprocess` [43] for the memory management and ZeroMQ for passing meta-data required to locate the message contents. Sharing a common memory region among processes is especially beneficial for data processing topologies, as it does not entail making expensive copies of messages when it is not necessary. To avoid problems related to concurrent access to the same memory region and shared ownership, the library allows for having only one owner of a given message at a time. Therefore, the publish-subscribe communication pattern cannot be combined with the shared memory transport, as one message could have multiple subscribers otherwise.

The software of data acquisition and processing systems in high energy physics often operate within finite state machines [44]. This approach allows to define the expected behaviour of each system component with regard to an environment (e.g. particle beam insertion, presence of stable beams, beam dump) and control those components in a coherent manner, avoiding potential race hazards. FairMQDevice implements the state machine shown in Fig. 3.3. The state sequence starts with an *Idle* device. Then it may follow a series of transitions, which involve e.g. binding communication channels, into the *Device Ready* state. Returning back to *Idle* is possible by *Resetting device*. By *Initializing Task* and *Resetting Task* one can reinitialize a processing algorithm without reallocating acquired resources in the preceding steps, which would make the whole process longer and potentially impose state changes to other devices. While *Ready*, a device can finally enter the *Running* state. Then it may perform its task until the *Stop* transition is initiated. Regardless of the main state machine, FairMQDevice has a second one, which indicates its health with the *OK* and *Error* states.

3.2.4. The O^2 framework

The FairMQ library provides the necessary foundations to build a large data processing system and it was chosen as the messaging library of the O^2 system. The O^2 data model expects that each message has two components - a stack of headers and a payload. The framework should never inspect the payload and may tamper only with the headers. The header stack might consist of one or more headers, all following the same template. Some headers are created and managed by the framework, while other, optional might correspond only to specific kind of data. The compulsory *Data Header* consists of (among others):

– *Data Origin* – a three-letter detector code or the name of a facility which produced data

– *Data Description* – a data type description

– *Subspecification* – an arbitrary index to differentiate parallel data streams of the same type

– *Serialization method* – the method which was used to store data in the payload

– *First orbit of a TimeFrame* – a unique TimeFrame identifier within a data taking run, which is derived from the LHC clock signal.

The data model is language-agnostic. The headers are allocated in contiguous blocks of memory and their fields have fixed size.

The O^2 system is expected to execute at least hundreds of unique devices and most of them might run in parallel. As a consequence, configuring and managing such a large number of processes would require a lot of effort. Also, each device would have to make sure it follows the O^2 data model, serialise, deserialise and match messages. Therefore, the O^2 system contains an additional framework which accommodates these tasks - the Data Processing Layer (DPL) [8].

A DPL topology (or workflow) consists of Data Processors, which are pipeline stages built on top of FairMQDevices. Data Processors take care of receiving incoming messages and synchronizing them if more than one input data stream is expected. Aside from supporting standard message inputs, the framework may also generate timers or retrieve objects from databases. A user-defined algorithm still makes the core of a Data Processor. A developer can declare a number of callbacks corresponding to state machine changes and a set of events, including readiness of a new collection of input messages or the reception of an *End Of Stream* notification. Incoming data is accessed via a high-level interface which takes care of safe message deserialisation, a process which is error-prone and involves a lot of boilerplate code in the case of more complicated serialisation methods. Similarly, the framework handles creation and sending of new messages. Another interface gives access to more advanced services, such as the resources monitoring library, configuration key-value store, state transition callback registration, logging facility and the FairMQDevice API.

The DPL takes care of arranging a topology of Data Processors given a workflow specification written in C++. A developer declares a set of structures containing, among others, required data inputs (which might contain wildcards), produced data outputs and a processing algorithm in form of the C++11's lamdba function or a function pointer. If more than one device needs a certain type of data, they are arranged in a queue. The same message is processed in a pipeline in order to respect the FairMQ's limitation of the singular data ownership. Unique devices can be replicated by setting an additional parameter. In such case the parallel workers receive data in round-robin order and process them independently.

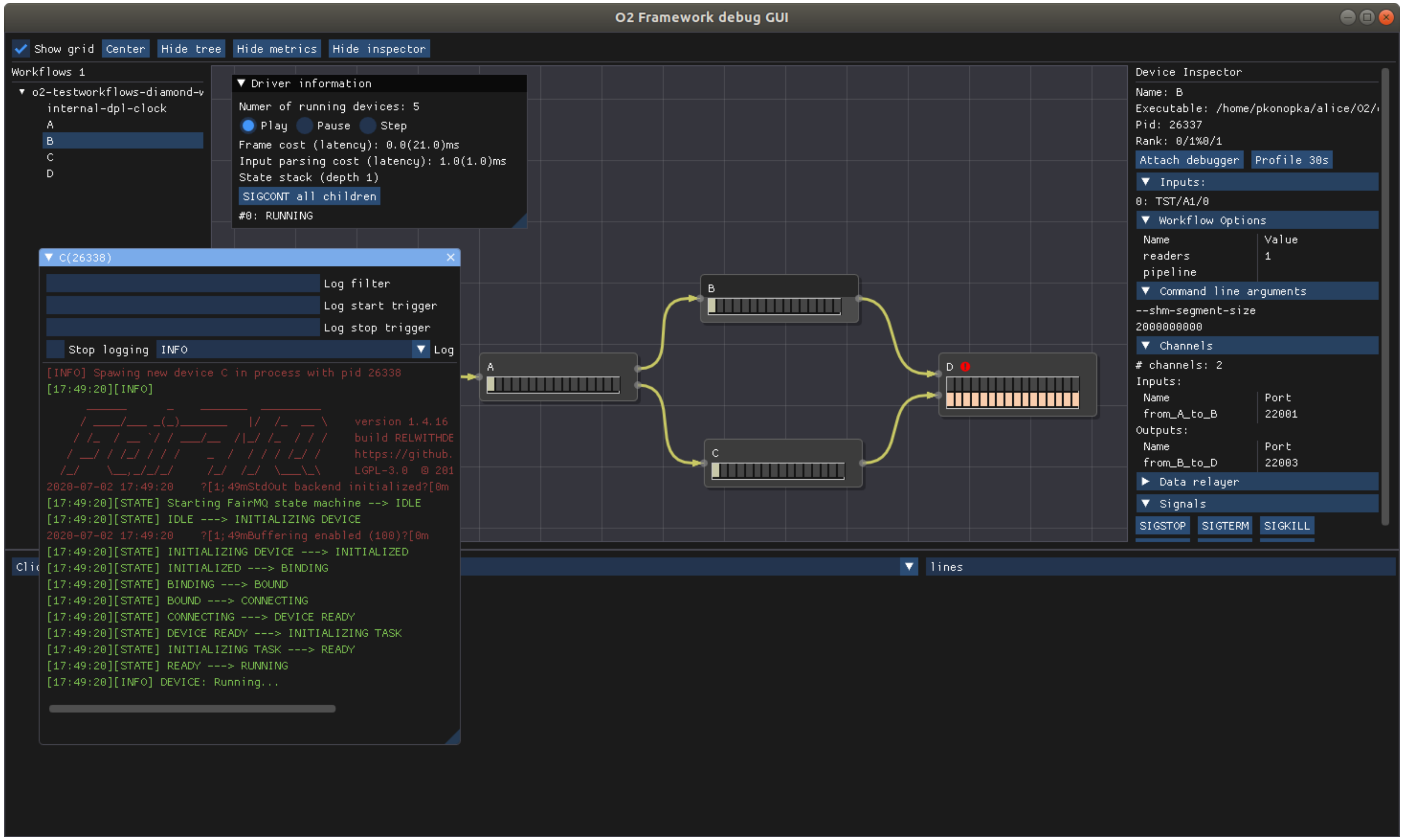
```
#include "Framework/runDataProcessing.h"

using namespace o2::framework;

AlgorithmSpec source() {
  return AlgorithmSpec{
    [](ProcessingContext &ctx) {
      auto aData = ctx.outputs().make<int>(OutputRef{ "a1" }, 1);
      auto bData = ctx.outputs().make<int>(OutputRef{ "a2" }, 1);
  }};
}
AlgorithmSpec simplePipe(std::string const& what) {
  return AlgorithmSpec{ [what](ProcessingContext& ctx) {
      auto bData = ctx.outputs().make<int>(OutputRef{what}, 1);
  }};
}
AlgorithmSpec sink() {
  return AlgorithmSpec{ [](ProcessingContext&) {} };
}

WorkflowSpec defineDataProcessing(ConfigContext const& specs) {
  return WorkflowSpec{
    { "A", Inputs{}, Outputs{{{"a1"}, "TST", "A1"}, {{"a2"}, "TST", "A2"}}, source()},
    { "B", Inputs{{"x", "TST", "A1"}}, Outputs{{{"b1"}, "TST", "B1"}}, simplePipe("b1")},
    { "C", Inputs{{"x", "TST", "A2"}}, Outputs{{{"c1"}, "TST", "C1"}}, simplePipe("c1")},
    { "D", Inputs{{"b", "TST", "B1"}, {"c", "TST", "C1"}}, Outputs{}, sink()}
  };
}
```

Listing 3.1. A condensed example of a DPL specification of a diamond shaped topology (slightly modified version from [8]).

The listing 3.1 shows a basic diamond-shaped topology specification that passes very simple messages. A source file with the `runDataProcessing.h` header included can be compiled into an executable which contains the declared workflow. For development purposes, running the created binary is enough to spawn the full process topology, as all devices are executed as its child processes. A dedicated GUI (Fig. 3.4) facilitates debugging. It visualises a graph of devices and connections between them, shows the flow of data and configuration of processes. It also allows to inspect the monitoring metrics and log messages of each component.

Developers usually prepare self-contained parts of the full O^2 process topology using simulated data. In order to test the integration of separate workflows, one can execute them in one terminal command by connecting their binaries within a Unix pipeline. During data taking all the DPL workflows will be driven by the AliECS and a dedicated control software for the EPN farm.

After the acquired data is completely reconstructed, it is ready for physics analyses. The ALICE Run 3 analysis framework [45] facilitates writing analysis tasks by standardising the data format, providing convenient access to data and imposing a pipeline structure of the processing. It also relies on the Data Processing Layer, thus it allows to reuse the O^2 code and combine it with the analysis software.



```
▼ Driver information
Numer of running devices: 5
● Play ● Pause ● Step
Frame cost (latency): 0.0(21.0)ms
Input parsing cost (latency): 1.0(1.0)ms
State stack (depth 1)
SIGCONT all children
#0: RUNNING
```

```
▼ C(26338)
[INFO] Spawing new device C in process with pid 26338
[17:49:20][INFO]
                                                 version 1.4.16
                                                 build RELWITHDE
                                                 https://github.
                                                 LGPL-3.0  © 201
2020-07-02 17:49:20    ?[1;49mStdOut backend initialized?[0m
[17:49:20][STATE] Starting FairMQ state machine --> IDLE
[17:49:20][STATE] IDLE ---> INITIALIZING DEVICE
2020-07-02 17:49:20    ?[1;49mBuffering enabled (100)?[0m
[17:49:20][STATE] INITIALIZING DEVICE ---> INITIALIZED
[17:49:20][STATE] INITIALIZED ---> BINDING
[17:49:20][STATE] BINDING ---> BOUND
[17:49:20][STATE] BOUND ---> CONNECTING
[17:49:20][STATE] CONNECTING ---> DEVICE READY
[17:49:20][STATE] DEVICE READY ---> INITIALIZING TASK
[17:49:20][STATE] INITIALIZING TASK ---> READY
[17:49:20][STATE] READY ---> RUNNING
[17:49:20][INFO] DEVICE: Running...
```

```
Device Inspector
Name: B
Executable: /home/pkonopka/alice/O2/
Pid: 26337
Rank: 0/1%0/1
Attach debugger  Profile 30s
▼ Inputs:
0: TST/A1/0
▼ Workflow Options
 Name        Value
 readers     1
 pipeline
▼ Command line arguments
--shm-segment-size
2000000000
▼ Channels
# channels: 2
Inputs:
 Name        Port
 from_A_to_B 22001
Outputs:
 Name        Port
 from_B_to_D 22003
▶ Data relayer
▼ Signals
SIGSTOP  SIGTERM  SIGKILL
```

Fig. 3.4. The debug GUI of the Data Processing Layer.

4. Overview of data quality control systems

In this chapter, the reader is introduced to some examples of usual methods of data quality monitoring and assessment in high energy physics experiments. Then, a thorough review of the literature about such quality control system designs and implementations, as well as the used tools, is carried out.

4.1. Examples of data quality control methods

There are plenty of approaches to the problem of data quality monitoring, which cannot be all well covered in this dissertation due to the large scope of the topic. Particle detectors convert physics phenomena associated with passing particles into electric signals. They are built in different technologies and they are usually designed to detect only some kind of particles depending on their species and energy. Moreover, they are designed to measure only some specific characteristics of these particles such as electric charge, energy, transverse momentum and time of flight. In this section a few examples are presented to give a brief overview on how detector performance and produced data might be evaluated.

The Time-Of-Flight (TOF) [46] in the ALICE experiment is one of the key sub-detectors used to identify particles produced in the central barrel. It can measure the time of flight of charged particles coming from the interaction vertex to reach the TOF detector, with precision better than 100 ps. One of the ways to verify if data produced by the TOF detector is correct involves checking the distribution of raw hit times [47], as shown in Fig. 4.1. Particles produced in a collision are expected to reach the detector in a certain time range between 150 ns and 225 ns. If there are measurements outside of this spectrum or more than one maximum is observed, one should investigate this issue as it is suggests a faulty behaviour of the detector or the data acquisition system.

The second biggest accelerator at CERN, the Super Proton Synchrotron (SPS), hosts an experiment called NA61/SHINE [49]. Similarly to ALICE, the detector is designed to register high-multiplicity particle collisions, but the interaction occurs between a particle beam and a fixed target, with the detector system behind it. In order to precisely determine the interaction vertex location, as well as to track and identify the produced particles, a group of Time Projection Chambers (TPCs) is installed behind the target. Two of them, called Vertex-TPCs (VTPCs), are placed in the magnetic field.

TPCs [50] are detectors filled with a mix of noble gases. Electromagnetically interacting particles ionize the gas and the freed electrons drift in an electric field to one side of the chamber. There, they are amplified and turned into a measurable electric signal on the array of readout pads. The particle position

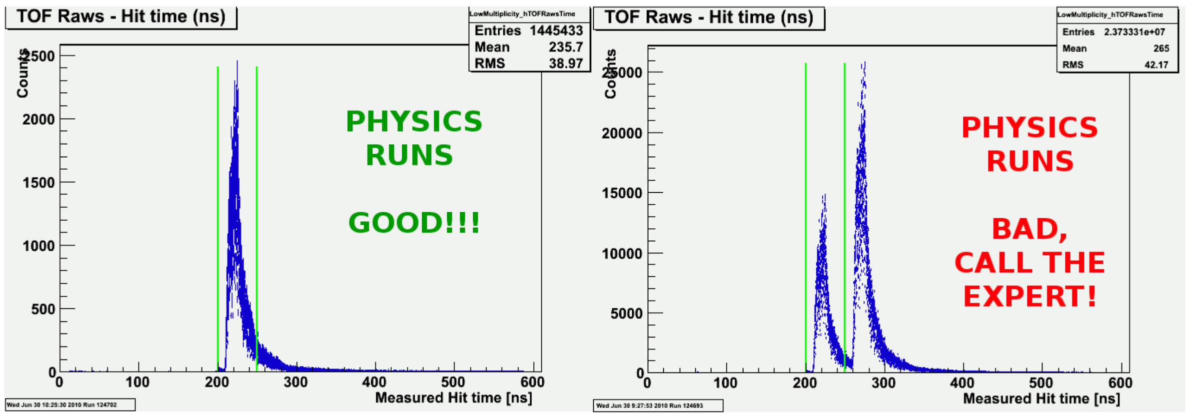


Fig. 4.1. Examples of correct and incorrect distribution of hit times in the ALICE TOF detector (the documentation of the ALICE Run 1&2 Data Quality Monitoring system [48]).

on the readout pad plane is determined by the pad position, while the third coordinate is derived from the electron drift time in the TPC gas volume, thus name the Time Projection Chamber. The measured charges on neighbouring pads or timeslices are then combined into clusters, since they usually correspond to the same particle. Finally, clusters which are distributed across straight or curved lines are combined into particle tracks.

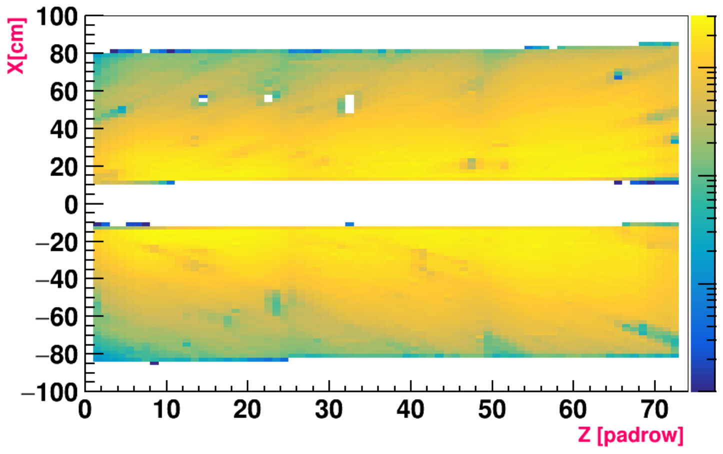
The quality of acquired data in the NA61/SHINE experiment is assessed daily based on a number of plots. For example, for the VTPC sub-detectors the following observables are monitored:

– **Occupancy plots** (Fig. 4.2a). Each recorded cluster is allocated into corresponding position on the histogram. The result is a heat map of hits, which shows how often particles are observed by the detector in each location. Unexpectedly low or high values suggest problems with e.g. the readout electronics.

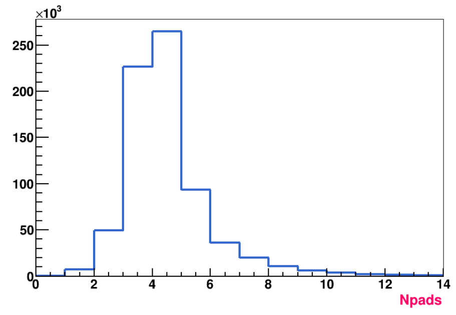
– **Distribution of the number of pads per cluster** (Fig. 4.2b). As almost each charged particle generates an electric signal on a few pads, the distribution of the number of pads per cluster should follow a certain expected shape. Seeing a high amount of small clusters might indicate a high level of noise in the detector electronics.

– **Distribution of the number of timeslices per cluster** (Fig. 4.2c). Similarly to the previous example, a cluster might be registered across certain amount of timeslices (units of time). One may inspect their distribution in order to recognise many problems, e.g. noisy electronics.

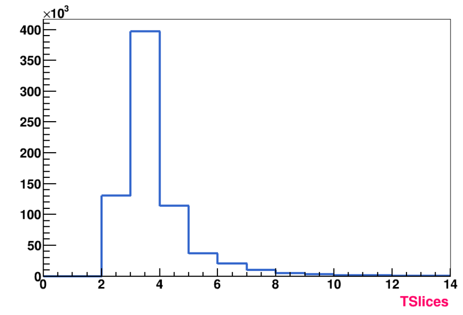
– **Distribution of the maximum analogue-to-digital converter (ADC) values per cluster** (Fig. 4.2d). The maximum registered charge value for each cluster is stored in a histogram. A large amount of ADC overflows (the right side of the plot) would indicate e.g. an incorrectly adjusted signal amplification.



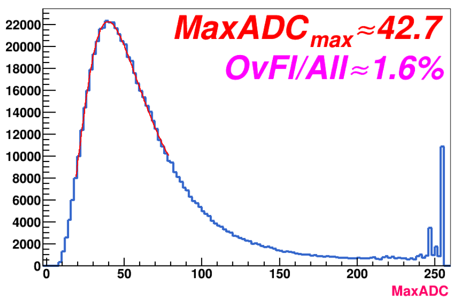
(a) An occupancy plot of VTPC1 in the x and z axes.



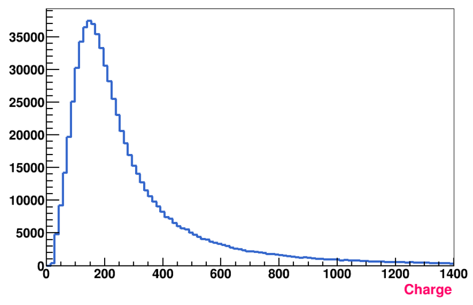
(b) A distribution of the number of pads per cluster in VTPC1.



(c) A distribution of the number of timeslices per cluster in VTPC1.



(d) A distribution of the maximum ADC values per cluster in VTPC1.



(e) A distribution of the total charge per cluster in VTPC1.

Fig. 4.2. Examples of Quality Attestation plots of the VTPC1 in the SHINE experiment (source - private communication with NA61/SHINE experts).

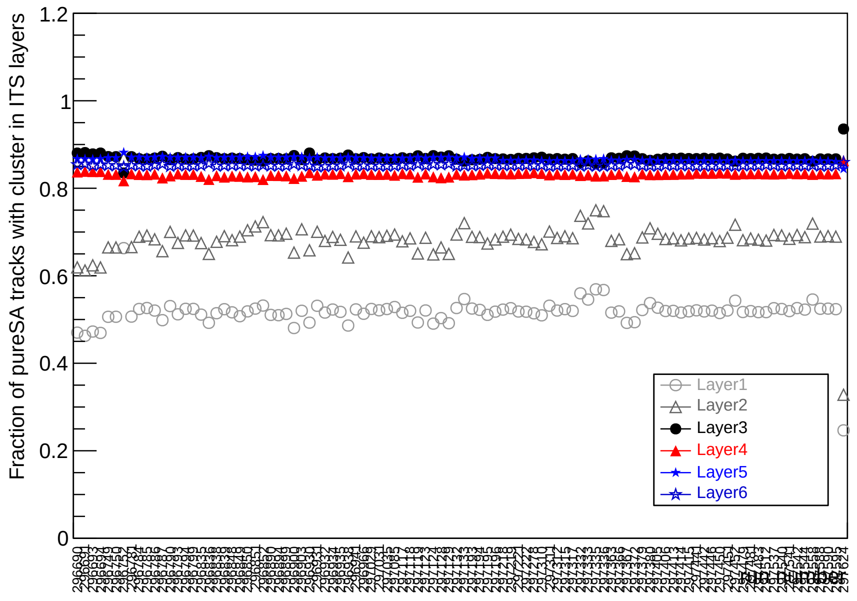


Fig. 4.3. An example of Quality Assurance trend in the ALICE ITS detector during Run 2 (ALICE QA repository).

– **Distribution of the total charge per cluster** (Fig. 4.2e). The total charge of each cluster is calculated and stored in a histogram. As in the previously described plot, an abnormal distribution may suggest problems with the signal amplification.

The former ITS detector in the ALICE experiment consisted of six layers build with three different technologies [51]. It was used for finding the collision vertex, tracking and particle identification. During the event reconstruction, the clusters detected by the ITS and TPC detectors are used together to find particle tracks. However, tracks are also reconstructed using only the ITS clusters to monitor the detector performance and for specific data analyses. They are referred to as pure stand-alone (SA) tracks. Fig. 4.3 illustrates the shares of different detector layers in these reconstructed pure SA tracks across one hundred data acquisition runs. By finding outliers in such trends, the detector experts may identify runs with potentially bad quality data.

The presented examples were selected specifically to provide a general idea about methods of data quality monitoring and assessment, although without introducing too much details about the detectors technology. In fact, many data quality control tasks might be much more complicated and require deeper knowledge of this matter.

4.2. ROOT - data analysis framework

ROOT is a framework for statistical data analysis [52] developed at CERN. It is currently the most popular framework for computing in high energy physics, as it will be shown in the summary of the last generation of data quality control systems. This section lists the most important features of ROOT which brought it popularity, as well as the recent developments and plans.

The functional core of the framework are its data types, which are optimised for handling large data quantities. ROOT offers a wide range of histogram classes. Low-dimensional histograms are implemented with concrete classes (i.e. `TH1`, `TH2`, `TH3`), while higher dimensions are covered by the template classes `THn` and `THnSparse`. `THn` allocates all the necessary memory for each bin, while `THnSparse` decreases the RAM consumption by allocating only non-zero bins. Columnar data can be stored in the `TTree` type, which optimises random access and facilitates statistical analysis of its contents. Other data types include e.g. graphs (sets of points), functions, matrices, four-vectors and a set of containers - lists, arrays, maps. They can be serialised and deserialised, which is helpful for messaging and data storage.

The aforementioned classes can interact with a powerful mathematical toolbox. It allows to perform curve fitting, optimisation, matrix algebra, four-vector computation, statistical and multivariate data analysis as well as pseudo-random number generation.

ROOT has a built-in reflection mechanism, which is based on dictionaries. They are data structures generated during compilation which contain instructions indicating how to construct and handle any class object. This allows to create data types by specifying a class name (with a string), inspect its ancestry, find and execute its available methods (also with a string). It should be noted that, as of 2021, C++ does not offer this feature in contrast to several more modern languages. Dictionaries also facilitate permanent storage with class schema evolution. The information about the way to reconstruct an object can be stored together with the object itself. While C++ is a compiled language by nature, the *cling* interpreter comprised in ROOT allows to execute the code with a command prompt, using Just-In-Time (JIT) compilation. A lot of physics processing and analysis software is written in form of macros - C++ scripts, which are then interpreted and executed by *cling*, or its predecessor CINT.

The ROOT package includes 2D and 3D visualisation tools for its data types and analysis results. Developers might also use them to build applications with interactive graphical user interfaces. The JSROOT package allows to visualise the ROOT data types on websites [53].

While being very popular and appraised, ROOT also receives a certain dose of criticism [54]. Beginners complain on the steep learning curve, due to a complicated class inheritance structure (including forbidden methods), difficult memory management leading to leaks and overall complexity of the package. Since the development commenced before the existence of the Standard Template Library (STL) in C++, the framework uses and supports only its own containers. The name ROOT collides with the administrator account name and the root directory in Unix-like systems, confusing web search engines and obfuscating build configuration files.

The criticism is being addressed by the developers as they are rewriting ROOT for the upcoming version 7. It should support standard containers and modern C++, break with some string parameters to class methods and provide interfaces which are clearer and safer to use. Among others, a new high-level interface for tabular data, `RDataFrame` was introduced. It supports lazy actions, which means that data processing is triggered when results are accessed. This allows to perform requested operations faster and with use of multi-threading.

4.3. The latest generation of data quality control systems

Data quality control systems have a history almost as long as electronic data acquisition systems and were often inseparable parts of those. In this chapter, examples of contemporary quality control systems in physics experiments are presented.

4.3.1. The ALICE experiment

The ALICE experiment had several data quality tools during its commissioning and data-taking runs, some of them existing in parallel, others replaced by their successors. Instead of preparing a complete and highly functional data quality control system at the very beginning, the developers of the ALICE Run 1 software decided to start with a moderately simple monitoring tool, called MOOD (Monitor Of Online Data) [48]. This way, the authors of the planned complete system could observe the usual use-cases and find potential problems which would appear in larger-scale applications. MOOD was a single executable C++ program interfaced with the main data acquisition (DAQ) software [55]. It could access a single monitoring source at once, i.e. just one type of data, for example raw data coming from a specific sub-detector. The piece of software heavily relied on the ROOT framework to provide a graphical interface, data structures, such as histograms and graphs, and tools for their statistical analysis. The sub-detectors had dedicated modules unified under a common interface class with a set of methods which should be overridden. The tool helped to commission the ALICE detector before the start of the first data-taking run in 2009.

The AMORE software (Automatic MOnitoRing Environment) [48] served as a main tool to control the quality of the raw detector data during acquisition across Run 1 and 2. It provided a quick feedback to shifters in the experiment control room and allowed them to quickly identify and solve problems with the detector and data.

Its architecture relies on the publish-subscribe paradigm - a large group of processes, called agents, perform computations and publish results into a common data pool, which is based on the MySQL database management system [56]. The processing results are subscribed to by graphical user interfaces, used by shifters in the experiment control room, and by the ALICE *eLogbook*, which provides access to that data from any point in the web. The Distributed Information Management (DIM) system [57] informs the subscribers about presence of new objects in the database.

The quality control algorithms very much depend on a kind of sub-detector being used and the format of received data. The software loads the corresponding code as external libraries which implement a set of methods of a common interface. The right classes are constructed using the reflection mechanism in ROOT. This approach does not impose any dependencies of the framework on the detector code, making it indifferent to potential compilation problems with the user modules. Also, just as in the original

MOOD, the processing results are usually the ROOT data types - histograms of any dimension, graphs and sometimes data arrays (TTrees).

The experience gained during the first years of operations was presented in [58]. More than 40 AMORE agents used to carry out data quality monitoring computations, updating around 10000 objects per minute, which corresponds to roughly 10 MB/s of recorded data. 95% of those was aimed for detector experts use, while the remaining 5% was shown to shifters.

The authors also noticed that generating histogram plots for the inspection tools requires a significant amount of processing power, which would slow down the AMORE agents. Thus, a new web-based solution was implemented, replacing the previously used C++/ROOT application [59].

The initial data quality monitoring would take place during the acquisition, providing preliminary information about the correctness of the detector operation and the data recording process. However, in the following days, a thorough data Quality Assurance (QA) would be performed. The detector calibration parameters were updated and evaluated based on the particle tracks reconstruction performance. Each offline data processing would also create QA histograms and other kinds of ROOT objects. Different physics observables were aggregated into trending plots and correlated with each other in order to track the detector performance across time, which was then discussed at weekly meetings. Final approval of good quality data was done with the data-taking run granularity (i.e. up to 15 hours). QA coordinators would create separate run lists per several physics analysis types, which required a good data quality of different sets of sub-detectors. During Runs 1 and 2, the experiment recorded raw detector data, which allows the Data Preparation Group to rerun the original reconstruction passes even years after, increasing the quality of data for the physics analyses. The author holds there the position of the QA coordinator since May 2020.

To complement the quality control systems described above, the EMCal sub-detector team prepared an additional tool, Overwatch [60], which facilitates the analysis and visualisation of QA data generated during online reconstruction performed on the High Level Trigger (HLT) computing system [21]. The Overwatch allows to define data processing in the Python language, to observe trending plots in arbitrarily chosen time ranges and to implement automatic alarms based on those. Intially, the software was developed and used only by the EMCal team, however, thanks to its extensible architecture, also other detector teams created their modules.

While Data Quality Monitoring objects in the AMORE software did not require merging due to its architecture, the HLT was a message-passing system with parallel computations performed. Thus, the data generated on the HLT were merged before further analysis.

4.3.2. The ATLAS experiment

ATLAS (A Toroidal LHC ApparatuS) is a multi-purpose detector, used to study the Higgs boson and to search for extra spatial dimensions and particles which could form dark matter. 1.7 billions of proton collisions are observed during each second and around 1000 among those, which fulfil specified criteria, are registered [61]. To control the quality of data, the team of scientists developed the Data Quality Monitoring Framework [62].

Similarly to the ALICE experiment, the basic quality monitoring takes place during data acquisition [63], while the additional, power-consuming computations are performed asynchronously to the data-taking [64]. However, both stages of the processing rely on the same software. The data quality monitoring is built on the same distributed computing technology as the main ATLAS processing framework [65] - the Common Object Request Broker Architecture (CORBA). The parallel computations are performed by objects, which can access data within other, remote objects and invoke their methods.

The quality control in ATLAS begins with sampling of recorded collision data and storing their selected properties in histograms. In case that multiple servers create them in parallel, they are merged before any further analysis is performed [66]. Sometimes the results are aggregated in many stages in order to better distribute the load among processes [67]. Groups of histograms are put under automatic evaluation, which results in information about the quality of recorded data.

Quality monitoring takes place in multiple stages of data recording and processing. For example, the assessment of muon detection quality is performed based on the original data stream coming from the detector, reconstructed muon tracks and the final, high-level information about registered collisions [68]. The monitoring of calorimeter's state involves assessing partially reconstructed events together with a list of fulfilled trigger conditions during their acquisition [69]. This software is also used to check the detector performance with cosmic radiation during breaks in the LHC operation.

The graphical user interface [70], which can present the results of data quality monitoring, is based on the Qt ROOT library. The experiment shift crew can inspect the health of a given system component with the generated histograms organised in a from of a tree. One also has a possibility to create a graphical representation of the detector structure, which facilitates locating a potential failure.

4.3.3. The CMS experiment

CMS (Compact Muon Solenoid) is another multi-purpose detector at the LHC [12]. Its physics research programme largely overlaps with the one of ATLAS. However, CMS uses different technical solutions for the data acquisition and analysis.

The general idea of the data quality monitoring resembles the previously described systems [71][72]. During the data acquisition, only a fraction of collision events is analysed, but on each reconstruction level. Only the quality control of the track detection system generates roughly 350 000 histograms which are automatically checked. The evaluation results are aggregated into more consolidated form and both are stored as ROOT files. The experts and shift crew can inspect the system status in a graphical user interface, where histograms and check results are presented [73]. Then, the quality control is performed again, but with the complete set of recorded events and additional histograms created by the data acquisition system itself. The software algorithms also allow to observe changes of sub-detector parameters across multiple data-taking runs.

The DQM software of CMS relies on its main processing framework [74]. Data is organised in form of *Events*, while the processing code is divided into *modules*. Modules are arranged into *Schedules*, which determine the order in which the modules should process an Event. The *Schedule Executor* application executes the modules one after another, providing them with Events.

Certification of recorded data is the crucial step of its preparation [75]. During 6-hour shifts the crew monitors the correctness of the detector's behaviour based on aggregated information provided by the quality control system. Afterwards, the experts representing the separate experimental sub-systems mark the quality of the components they are responsible of. The final decision about approval of each data chunk (physics run) is taken by a group of data quality monitoring experts.

At the time of early development, the designers considered moving a large portion of real-time processing into external computing farms [76]. Transferring 10% of recorded data and results of their quality control would make load of 1 TB daily. Despite the promising test results in a collaboration with a facility in Bari (Italy), this solution was not used.

The software was further developed as the experiment ran [77]. The developers put emphasis on automatisation of the data certification by increased number of tests; starting and stopping the quality control system by integrating it with the data acquisition software; improving the processing performance and disk space monitoring on the hosting servers. The authors also updated the quality control system following processing parallelization support in the framework. In 2017 the team reported a successful application of the machine learning methods to automatically classify the generated objects [78]. The system flags the good and bad data where possible and leaves ambiguous cases for an evaluation by experts.

4.3.4. The LHCb experiment

The LHCb detector and experiment (Large Hadron Collider beauty) was founded to study differences between matter and anti-matter by observation of the beauty hadrons in particular.

The data quality control is carried out similarly as in previously described experiments [79]. Part of the main data stream is processed and stored into histograms, which are merged when needed and saved on the disk [80]. Merging can be performed in many stages - first to collect all data on the same node, then on sub-farm and farm levels, then finally aggregating results of the complete processing system. Their graphical user interface uses the ROOT framework to present plots to the shift crew. In the first version, the software did not carry out automatic tests - a decision about correctness of data was made by the shifters.

Later, the developers team modernised the user interface and introduced an automated histogram evaluation [81], based solely on Machine Learning (the AdaBoost BDT algorithm). Training data was obtained from the historical base of histograms and their manually assigned good and bad marks. Since the automatic classification had been implemented, the shift crew was asked to inspect only the histograms considered as outliers.

The software is based on the GAUDI framework [82], which is also used for the High Level Trigger system in LHCb. The code is organised into *Algorithms*, which can contain other Algorithms. The data they receive and produce are transferred through *transient data stores* - centralised buffers organised in a tree-like directory structure. The event data is shipped to monitoring tasks with the *Transient Event Store*, but the generated histograms are then published with DIM. This is the same technology that was applied in the AMORE software of ALICE, but in this case, also histograms contents are transferred instead of the information about their presence in the database. The histograms for the data quality monitoring are produced not only by dedicated tasks, but also by other algorithms, which have primarily a different purpose, such as calibration, triggering or analysis. Nevertheless, all results are presented to users in a unified and consistent way in the Histogram Presenter application.

In the data monitoring of the High Level Trigger system, the developers also report an existence of Histogram Adders, which merge partial results coming from tasks running in parallel on multiple processing nodes. They expect two kinds of messages: full messages with metadata and increments - objects which contain only the most recent changes [83]. In the latest developments, a part of the DQM system was reimplemented with *Apache Kafka*. The authors recommend this approach for the ease of use and maintenance, and automatic optimisation of message queues.

4.3.5. The Tevatron experiments

Tevatron was the second highest-energy particle accelerator in history, located in the proximity of Chicago in USA, active from 1983 until 2011. The two Tevatron experiments - the Collider Detector at Fermilab (CDF) [84] and DØ [85] - are probably most famous for the discovery of the top quark in 1995 [86].

The CDF's data quality monitoring was initially strongly intertwined with its data acquisition system [87]. For the Run II in years 2001-2011 the developers prepared a Consumer Framework [88], which allows to perform additional processing on acquired data. A fraction of collisions accepted by the last level of trigger serves as an input to 10 parallel analysis tasks, which monitor the data quality and produce diagnostic histograms and trending graphs. The framework takes care of data transport, displaying the processing results, logging errors and provides interfaces for initialisation and event processing. The monitoring plots are presented to the shift crew in a graphical user interface, which retrieve the histograms in a client-server scheme, GUIs being the clients. The software is written in C++ and uses the ROOT framework for data analysis and inter-process socket communication. It can run both synchronously to data-taking and asynchronously - on reconstructed, simulated or test data. In the first case it receives events from a dedicated server, in the latter it reads them from files or generates random data. The framework is complemented by the State Monitor - a watchdog process which monitors the state of monitoring tasks, shows their configuration and keeps track of the amount of processed events.

Like most of expensive particle accelerators, Tevatron operated 24-hours a day. This required that a shift crew had to control and operate the experiments daily and nightly. In order to distribute the workload more evenly, the developers have provided an access to the data quality monitoring tools via WWW [89].

The report on the initial DAQ system of the DØ experiment does not indicate an existence of a strictly data-related quality monitoring [85]. However, the software included an alarm system that keeps track of measurable detector parameters which might have an influence on the data quality - temperature, power, voltage, current, humidity etc. If a monitored variable crosses the allowed limits, the system raises an alarm.

During the Run II there was a data quality monitoring software [90], however, there is no publication concerning the framework itself. In the presentation about the DØ calorimeter's [91], the presenter indicated that the quality is monitored both during data taking and after, with trigger decisions information and reconstructed events. As in other examples, the ROOT package is used for histogramming and visualisation of the obtained results.

4.3.6. Gravitational waves detectors

The literature describes also data quality control methods used by the LIGO [92] and GEO600 [93] interferometers, which serve as tools to measure gravitational waves. The computation scheme resembles the previously mentioned systems. Acquired data is put under a preliminary real-time analysis on a computing farm. Thanks to a modular architecture and possibility to share their results, the algorithms can be organised into pipelines, improving the overall efficiency of the system. Data analysis results are stored in a database and then explored by data mining tools.

4.4. Unification efforts

Despite the apparent similarity of data quality control systems, their unification is not a simple task due to the use of different data models, computing architectures, tools and analysis algorithms. A software framework which could find its application in almost any physics experiment should allow to configure and adapt the aforementioned elements. The authors of [94] took up this challenge. They are developing DQM4HEP - a piece of software which should fulfil the usual requirements of high energy physics experiments regarding data quality control. In order to adapt this framework in a data acquisition system, one has to take care of supplying input data, defining data model and message serialisation methods, implementing quality control tasks as well as specifying the look and contents in a graphical user interface. As in the other cases, the framework relies on ROOT. The system was successfully adopted in two detector prototype setups - the AHCAL+beam telescope [95] and SDHCAL+SiWECAL [96].

4.5. Summary

The literature review shows that only bigger physics experiments describe their data quality control systems and there is very limited information available about smaller setups. The general idea of each system described in the literature is in fact similar - some portion of acquired data is processed, some statistics are derived and stored mostly into histograms. These results might be evaluated either automatically or manually by shifters. However, the terms used to describe these concepts are very different, which makes exchanging knowledge more difficult within the data quality control community. Tab. 4.1 contains an overview of particular concept names among the four large LHC experiments. One might notice slight differences even in the most fundamental concepts, such as division between quality control during and after data acquisition. The term 'Monitoring' sometimes relates both to data quality and basic software/hardware health - chip temperatures, resources usage etc. Also, only ALICE and ATLAS seem to have a generalised term for any object with statistics within the framework, while the others assume that all such objects are histograms. What ALICE calls Mergers, is actually described distinctly in each system. CMS performs merging directly in the process which runs Quality Tests. Similarly, these processes received very distinct terms among the experiments. Only the new ALICE and the existing ATLAS frameworks seem to allow to combine the results of automatic evaluation into more general information. All four systems had a common name of good runs lists, however, ALICE is going to withdraw, as its data quality will be described with finer granularity.

Tab. 4.1. The data quality control Rosetta stone, showing names corresponding to the same concept in data quality systems across different experiments. The '-' sign indicates that no such concept name was found (it still might exist, but undocumented in the literature). Names in *italics* are only indirect equivalents to the terms in the QC.

ALICE (Run 1&2)	ALICE (Run 3)	ATLAS	CMS	LHCb
-	QC	DQ	DQ	DQ
DQM	online QC	DQM	online DQM	DQM
QA	offline QC	DQA	offline DQM	offline DQ
event sampling	Data Sampling	Event sampling service	*reducing data rate*	-
AMORE Agent	Task Runner	DQM Producer	-	-
Publisher	QC Task	-	DQM Producer	Monitoring Task
Monitor Object	Monitor Object	*histogram* MonObject	*histogram*	*histogram*
Merger	Merger	*Gatherer*	*DQM Consumer*	Histogram Adder
-	Checker Check Runner	DQAgent	DQM Consumer	Histogram Analysis RoboShifter
QAChecker	Check	DQAlgorithm	Quality Test QTest	Analysis task
Quality Flag	Quality	DQResult	QTest result	-
-	High Level Quality	DQRegion	-	-
QA Trending	Post-processing	Post-processing	Historic DQM	*History Mode* *in Histogram Presenter*
AMORE GUI DQM Web Client	QC GUI (QCG)	DQM Display	DQM GUI	Histogram Presenter
Run Condition Flag	Timestamp	DQFlag	-	-
Good runs list	Data tags	*Good runs list*	*Good runs list*	*Good runs list*

Unfortunately, the literature does not contain much information about the ways in which data are sampled before their quality is analysed. The sparse information may suggest that data are selected randomly or in regular intervals, but the frameworks do not allow for customised sampling procedures. Similarly, the topic of merging distributed objects is not well covered in the literature.

The existing data quality control systems do not rely vastly on the message-passing approach nor the actor model. One of the few identified examples includes the Monitoring Tasks in LHCb, which send objects to Histogram Adders inside messages. The two applications could be treated as actors, however, it was not explicitly stated. The CDF experiment data quality monitoring system was based on the client-server model and a combination of file and socket communication, which could suggest the presence of some form of messaging. However, the internal communication socket bandwidth was reported to reach around 10 Mb/s. Therefore, it is concluded that no existing data quality control frameworks rely on the message-passing and the actor model to large extent.

The new ALICE QC framework should unify the two previously separate pieces of software - DQM and QA. The Mergers were previously present only in the HLT software where they combined QA histograms generated by parallel workers. However, the AMORE agents could not be replicated, limiting the amount of data they could process. This limitation is lifted in the new QC framework.

5. Quality Control system

The Quality Control framework is described in detail in this chapter. First, the general architecture is introduced. It is followed with a detailed description of each system component together with the reasoning for the selected solutions.

5.1. Architecture overview

The Quality Control framework consists of a number of components illustrated in Fig. 5.1. The conference paper [10] includes the summary below, with the exception of the *Aggregators* module, which was implemented after the publication.

The *Data Sampling* (blue doted arrows) is responsible for the selection and distribution of data samples based on configurable policies. These include, among others, pseudo-random sampling of parallelly distributed data, selecting messages of certain size and any custom filtering. Its main component, the *Dispatcher*, runs on each node where data are sampled and can be reconfigured during data taking.

The *QC Tasks* are the detector-specific algorithms executed either with the main data processing chain (on the FLPs or the EPNs) or remotely on dedicated Quality Control Servers. Their regularly published outputs are called QC Objects (Monitor Objects) and typically consist of ROOT histograms [52]. In case the tasks are running on many servers in parallel, their incomplete results should be merged into objects containing complete data, which is taken care of by the *Mergers*.

The *Checkers* carry out automatic evaluation of the data quality by running defined algorithms over QC Objects. They follow a common interface, which is inherited by a generic set of reusable checks and custom user algorithms. The *Aggregators* let users combine fine granularity Qualities into more general results, describing e.g. the global behaviour of a sub-detector. Finally, the QC Objects and the Quality Objects are stored in the QC repository.

Several members of the upgrade project are investigating the usage of the machine learning methods as a way to perform more sophisticated checks in multidimensional parameter space. They may interact directly with the repository or use *Checkers*.

Previously described by *Correlation* and *Trending* blocks (Fig. 2.5), the *Post-processing* module runs any task running asynchronously to the main data flow, on data derived from QC Objects and Quality Objects. It can be triggered manually, in regular time intervals or upon a number of declared events (e.g.

start of an LHC fill or end of a data acquisition run). Its results are stored directly in the database or inserted back in the QC processing chain.

The experiment shift crew and experts may use the QC GUI (QCG) to visualize stored QC Objects and Quality Objects. A generic user interface allows to navigate the objects and display them using JSROOT [53].

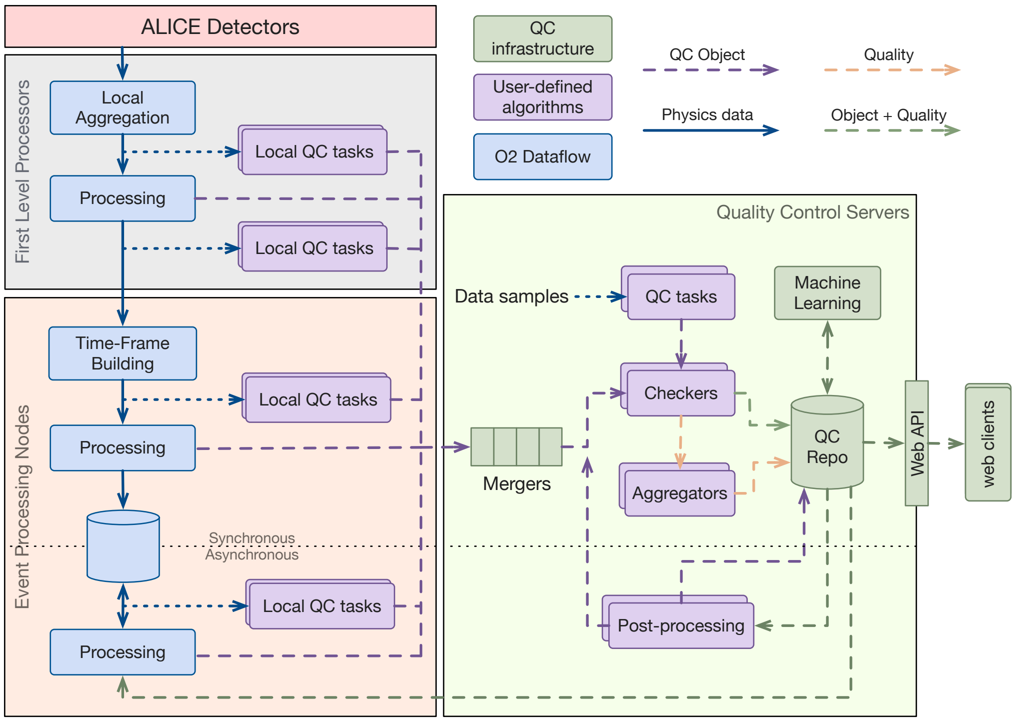


Fig. 5.1. The QC architecture as reported in this work. As opposed to the original design in Fig. 2.5, one may notice the addition of the *Aggregators* and the unification of the *Correlation* and *Trending* components into *Post-processing*.

The vast majority of the Quality Control framework - Data Sampling, QC Tasks, Checkers, Mergers and Post-processing - relies on the Data Processing Layer. The QC repository is interfaced with the Hypertext Transfer Protocol (HTTP), it communicates with other parts of the system by receiving requests messages to save or retrieve objects, processing them and sending replies. Therefore, the Quality Control framework follows the message-passing approach and the aforementioned components can be considered as actors within the actor model, since they carry distinct responsibilities and interact with each other only by sending and receiving messages.

5.2. Data Sampling

A robust and functional way to access acquired data is the first step to build a quality monitoring system. However, transferring the full data stream to QC Tasks, as well as analysing it, would require unreasonably large amounts of computational and bandwidth resources. Thus the majority of QC Tasks should perform the quality monitoring with a fraction of the events which might be selected randomly or

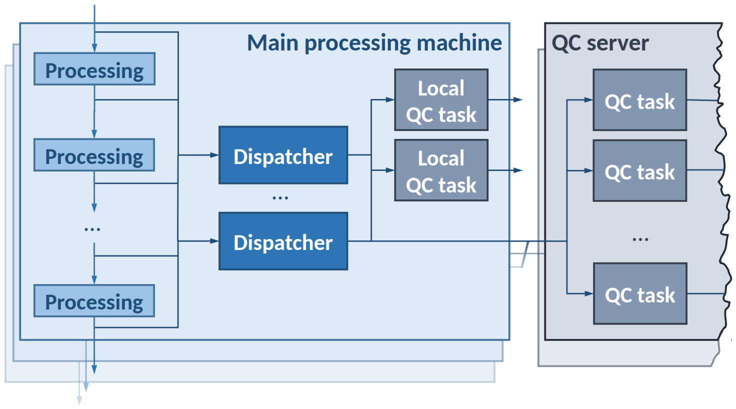


Fig. 5.2. A message-passing processing workflow with Dispatchers and QC [9].

by matching certain criteria. Thus, the 100% of data is shipped only to the algorithms which indisputably need it.

The data processing chain consists of many stages and it generates different permanent and temporary data which should undergo some evaluation. Optionally, the software should allow to block the main data stream, so all messages can reach a Task even at the cost of slowing down the main processing.

In the QC, the Data Sampling software is responsible for selecting and providing the data messages to QC Tasks and other clients. Its design and the considerations about random sampling were also discussed in [9].

5.2.1. Data Sampling design

Dispatcher is the key component and actor responsible for sampling and forwarding data. It is placed within message-passing topologies between data producers and clients which require sampled data (e.g. the Quality Control infrastructure), as shown in Fig. 5.2. The design allows for one or several Dispatchers working in parallel, subscribing to messages coming from the main processing topologies and receiving them in a round-robin schedule. Messages which fulfil declared sampling requirements are passed to local and remote clients.

Data Sampling Policies define the features of desired data, i.e. their provenance and description (Fig. 5.3). Data are matched to Policies according to their message headers. The grounds for accepting or rejecting data are defined as a group of Data Sampling Conditions. For a message to be forwarded to subscribers, all Conditions should return a positive decision. Their inner logic might perform for example: random sampling; choosing messages matching a defined payload size range; filtering a certain amount of consecutive messages in a sequence; sampling by keeping data throughput lower than a specified value; or any other custom sampling, implemented by inheriting an interface class and loaded dynamically by Dispatcher.

Enabling and disabling Data Sampling Policies and changing the list of Data Sampling Conditions is possible at run-time. Moreover, when forwarding a message payload, Dispatcher attaches an additional

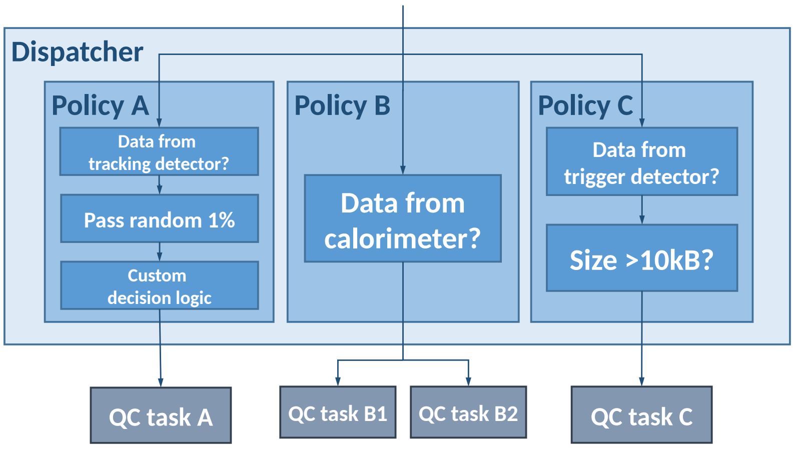


Fig. 5.3. An example of Dispatcher's configuration [9].

header to the header stack. It contains information which might prove extremely useful when developing tasks requiring advanced sampling, e.g. time of the last sampling decision, total amounts of messages seen and passed which matched the criteria.

5.2.2. Investigating data sampling techniques

In this section, data sampling methods in the current context are discussed. The reasoning for the proposed pseudo-random sampling is given and potential risks are mentioned. Advantages and risks of other sampling methods are also considered.

5.2.2.1. Bias risks in sampling

In many cases analysing only a portion of the full data stream is completely sufficient, provided that the selected samples do not carry any unwanted biases. Firstly, this situation may occur if size limits of message queues depend on their total memory consumption and they can drop messages when they hit the maximum capacity (e.g. in the ZeroMQ's publish-subscribe pattern). When a process has momentary or permanent problems with sustaining the input data throughput, its message queue will grow till its limits. With certain amount of free memory in such queue, only messages small enough might find their place in the queue and others may be dropped. In such case, one might see a bias towards smaller messages, which will make the sampled data unrepresentative. In ZeroMQ, thus also in the O^2, this situation is avoided, as message queue capacity is defined by a maximum number of messages. However, every computer has a limited amount of memory and reaching its capacity might result in similar behaviours.

Other risk is related to using certain sampling policies. For example, if a process samples an amount of data which should be constrained within a defined average throughput limit, then smaller messages might fit easier in the current capacity. Of course, such situation will not occur if all expected messages have identical size. Otherwise, one cannot avoid this problem, so it should be taken into account.

In case that a randomly selected representation of data is needed, to implement it, one may resolve to sources which claim to offer true random numbers, for example by using external entropy [97]. Otherwise, one can base their sampling algorithm on Pseudo-Random Number Generators (PRNGs), which use mathematical methods to produce numbers which seem to be random. In both cases, the chosen method should be put under evaluation to check if its statistical soundness is satisfying in given context.

5.2.2.2. Considered pseudo-random sampling methods

Data corresponding to the same time range of particle collisions will be distributed among multiple FLPs, as they are connected to different parts of the detector. However, for the convenience of matching them in the EPNs, they will carry a common 64-bit identifier, later referred to as *TimesliceID*. Still, such a data distribution brings additional challenge when sampling data from multiple nodes. For example, a QC Task might need pseudo-randomly selected messages with the same *TimesliceID* which appear on all nodes connected to a specific sub-detector.

This requirement can be fulfilled in two ways. The first assumes that one entity could take decisions about sampling data and propagate them to the concerned servers. However, sending and storing a list for every possible *TimesliceID* before data-taking would require great amounts of memory. On the other hand, requesting and receiving decisions in real-time would introduce a high latency, given the expected message rates of tens of thousands messages per second on each server. Also, a failure of the central decision-making node would propagate to all downstream components and its load would rise proportionally to the number of requests, making this solution more difficult to scale.

The second approach assumes that each Dispatcher takes the same sampling decision independently. It requires a PRNG, which, given a seed and an input number, can deterministically generate a pseudo-random value, independently from the computer architecture. In this context, a data acquisition run number may be used as the seed, while the message *TimesliceID* could be the input number.

Since the vast majority of PRNGs can generate either random sequences of bits or equally distributed numbers in the range of [0, 1], one has to perform an additional operation to obtain positive sampling decisions with the expected probability, as in Eq. (5.1). A pseudo-random number $rand$ is compared with a threshold value, which is defined as a product of the $fraction$ parameter and the highest possible random number $rand_{max}$. To prevent any messages being dispatched when $fraction = 0$ is set, the greater or equal comparison operator is used.

$$Decision = \begin{cases} 1 & rand < fraction \cdot rand_{max} \\ 0 & rand \geq fraction \cdot rand_{max} \end{cases} \tag{5.1}$$

Several ways of obtaining pseudo-random number which fulfil the aforementioned requirements were evaluated in [9]. The most straight-forward assumes using a PRNG which gives access both to its seed and inner state. The latter can be set correspondingly to a *TimesliceID*, so one can deterministically retrieve a pseudo-random value for each sample. Among this group of PRNGs, the Permuted

Congruentail Generator (PCG) [98] was tested, as it had an open-source and ready-to-use implementation in C++. It allows to iterate its state backward and forward. Among the available variants of the PCG, `pcg32_fast` was used. It is optimised for speed and generates 32-bit pseudo-random integers.

Obtaining a deterministic pseudo-random number is also possible by initialising a PRNG with a product of some constant seed number and *TimesliceID* each time. Then, the first output value is used. However, this method might not perform well if setting the seed requires a long processing time. It also may have a negative impact on the randomness quality. The set of `TRandom` PRNGs available in the ROOT framework was used to evaluate this method.

The last proposed approach relies on hash functions to produce values which are uniformly distributed over certain range. In this case, one may use a product of a seed value and a *TimesliceID* as the hash function key. The `splitmix64` generator [99] implemented in [100] was used this way. Other types of hash function may take two values and combine them into a hashed result. The `hash_combine` function from the boost library [101] was evaluated, using a seed value and a *TimesliceID* as the two arguments.

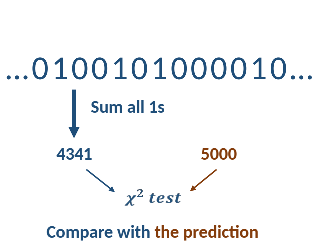
5.2.2.3. Benchmarking statistical soundness of data sampling methods

The presented sampling methods were benchmarked in order to evaluate their statistical soundness and performance. The pseudo-random numbers, which were compared with the threshold value in Eq. (5.1), were tested with the `dieharder` suite [102]. This benchmark set contains randomness tests which expect binary sequences as their input and return p-values describing how much likely was the sequence random. If a PRNG generates truly random numbers, the p-values should fall randomly in the range [0, 1]. Thus, each kind of test is run 100 times (by default) and the p-value distribution is checked to be uniform using the Kolmogorov-Smirnov test (KS test). The testing procedure is documented in [103].

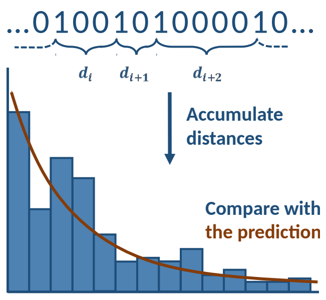
The `dieharder` tests expect that received bit is truly random. Because of this, the PRNGs in the ROOT framework could not be well evaluated, as they produce floating-point numbers between 0 and 1. Even after scaling them to the maximum integer values, their bit representation might contain some patterns due to the floating-point notation method [104]. Thus, additional benchmarks (Fig. 5.4) were proposed in [9]. They were specifically designed to test the bit sequences of uneven proportion of 0s and 1s, so the sampling decision methods could be benchmarked directly. They were implemented with use of the statistics toolbox in the ROOT framework.

The test (A) involves summing up all 1's in a sequence of *N* decisions, computing a χ^2 value (Eq. (5.2)) and a p-value. Similarly to the `dieharder` suite design, the test is ran 100 times and the uniformity of the p-value distribution is checked using the KS test. Effectively, this benchmark evaluates if a sampling method provides the expected amount of positive decisions with certain randomness.

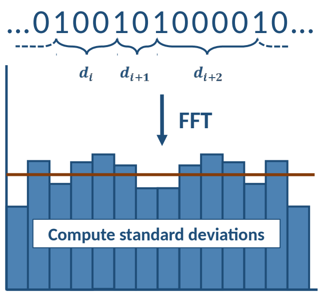
$$\chi^2 = 2\frac{(N \cdot fraction - trues)^2}{N \cdot fraction} \tag{5.2}$$



(a) The χ^2 test (A).



(b) The Runs test (B).



(c) The FFT test (C).

Fig. 5.4. The three randomness benchmarks proposed in [9].

In contrast to the previous benchmark, the test (B) inspects the order of bit sequences. The distances between each two consecutive 1's in a sequence of N decisions are computed and accumulated in a histogram. The distribution derived from truly random values should follow the negative binomial distribution Eq. (5.3), which in this case can be expressed as Eq. (5.4). The similarity of the distribution is measured with the χ^2 test, then a p-value is calculated. The test is run 100 times and the results are checked to be distributed uniformly with the KS test.

$$f(k;r,p) \equiv Pr(X=k) = \binom{k+r-1}{k} p^k (1-p)^r \tag{5.3}$$

$$f(k;1,1-f) = (1-f)^k f \tag{5.4}$$

The test (C) computes the Fast Fourier Transform (FFT) of the distances between consecutive 1's. The standard deviations of frequency powers (excluding the constant component) are calculated for 100 test runs and summed up. For a good PRNG, the test should return a similar value as for truly random sequences.

Table 5.1 contains the benchmark results. The `/dev/urandom` source available in Linux was used as a reference for a high quality random number generator. The performance benchmarks were performed on a server with Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60 GHz. The `dieharder` tests received raw pseudo-random numbers generated with each method instead of the sampling decisions. Due to the previously mentioned differences in PRNGs, their results were not treated as definitive, but rather as suggestions. The (A) and (B) tests generated p-values. The values in the range of [0.005, 0.995] were interpreted as good results. The (C) test should return a value close to the one obtained for the reference generator.

The results indicate that setting the seed before generating each pseudo-random value severely impairs the computational performance. Even though the more sophisticated `TRandom` generators pass the randomness benchmarks, they could not be applied in this context due to the slow response. However, one should note that these PRNGs were used in an unusual manner, therefore the test result do not indicate bad quality of `TRandom` overall.

Tab. 5.1. The test results of random sampling methods for $N = 10^7$ and $fraction = 0.01$ [9]. (*) The PCG performance result was obtained with incremental *TimesliceIDs*, but with realistically distributed values it needs 5-20 ns more time for each decision.

Test / PRNG	`dieharder`	ns/call	A (KS test p-value)	B (KS test p-value)	C (σ) (lower is better)
Hash function 1	PASSED	2.66	0.0101	0.2809	1552
Hash function 2	a dozen FAILED	2.07	0.5806	0.3667	1551
PCG	PASSED	2.95*	0.2106	0.8127	1552
TRandom	all FAILED	7.97	0.0000	0.0000	6067
TRandom1	all FAILED	300.2	0.0000	0.0000	2665
TRandom2	a half FAILED	30.60	0.1545	0.0000	1554
TRandom3	PASSED	1378	0.0243	0.6994	1552
TRandomMT64	PASSED	2360	0.0541	0.9671	1552
TRandomRanlux48	almost all FAILED	118.1	0.0000	0.0000	2418
TRandomMixMax17	too slow	22224	0.0101	0.5806	1551
TRandomMixMax	too slow	4487360	too slow	too slow	too slow
`/dev/urandom`	PASSED	923.0	0.0366	0.2105	1552

The hash functions and the PCG showed great performance in terms of required CPU time. Also, they provided good results in the randomness benchmarks with the exception of the second hash function, which failed a couple of `dieharder` tests. Based on these data, the PCG was adopted in the Data Sampling software, as other studies also indicated its high quality [105][106]. The first hash function was kept as a backup option in case that additional performance improvements are needed.

5.2.2.4. Equalizing the load with random sampling

Being able to control the pseudo-randomness of sampling gives us interesting possibilities, but also certain risks if used incorrectly. On one hand, it allows us to sample messages corresponding to the same events within one Data Sampling Policy, as its PRNGs receive the same seed. This can be extended by having multiple sampling policies for different tasks with a common seeding value, when it is expected.

However, when possible, such sampling coherence should be avoided. If all policies decided to copy data at the same time, they could create significant spikes in resources usage, which would result in an unnecessary increase of elements in message queues. Having the sampling decisions distributed as evenly as possible should help with avoiding such problems.

Therefore, the sampling seeds in different policies should vary, unless specifically requested. Moreover, to obtain unrepeatable sampling sequences across data acquisition runs, the constant seed values can be modified with a run number, e.g. by addition, multiplication or xor operation of the two parameters.

5.2.2.5. Advantages of advanced filtering

The features provided by the design allow for an advanced data selection, which should help to avoid transporting unnecessary messages, therefore minimizing the amount of required CPU, memory and bandwidth resources. Especially, one can consider moving away from the traditional approach of evaluating frequency of certain events in a histogram to a more lightweight method, which involves filtering and counting only the unwanted ones.

The following example is considered. A data quality control tasks monitors a certain parameter derived from sampled data and allocates it on a one-dimensional histogram. If a certain percentage of these values is contained within a safe margin, the quality of data is considered as good, otherwise it is marked as bad. Computing the exact value of the parameter requires a significant amount of processing time, but one can quickly identify majority of good data without marking any false positives, i.e. mistakenly perceiving bad data as correct. The task cannot be located on the same server as the main data processing. One might approach such a problem in a number of ways:

– **The task subscribes to 100% of data (Fig. 5.5a).** This requires a lot of computational power and data transfer, but it gives the most accurate results.

– **The task subscribes to a random fraction of data (Fig. 5.5b).** The task requires less computational power and transfer, because it uses less data. However, it needs more time to produce statistically significant results (if it is acceptable to omit any bad data in the first place).

– **The task subscribes only to filtered, potentially bad data (Fig. 5.5c).** The task receives only filtered data and the total amount messages which matched the sampling policy. Therefore, it uses less resources, accordingly to the amount of potentially bad data, and it still produces the most accurate results. However, sudden spikes in data throughput might saturate message queues. This might be mitigated by adding a second sampling condition, which keeps the throughput usage below a specified value, but it would break the guarantee of getting all messages containing incorrect data.

– **The task subscribes to randomly sampled and filtered, potentially bad data (Fig. 5.5d).** In this case, filtering is applied only after messages first pass random sampling. Therefore, the task requires even less computational resources, but it needs more time to produce statistically significant results. On the other hand, the risk of saturating message queues is much lower, as it would require a long sequence of positive random sampling decisions and detections of bad quality data by the filter.

The presented example shows that using custom filters might be beneficial to overall system performance, assuming that filtering requires much less computational resources than the actual processing of data and a quality control task does not need to process all messages or its random representation.

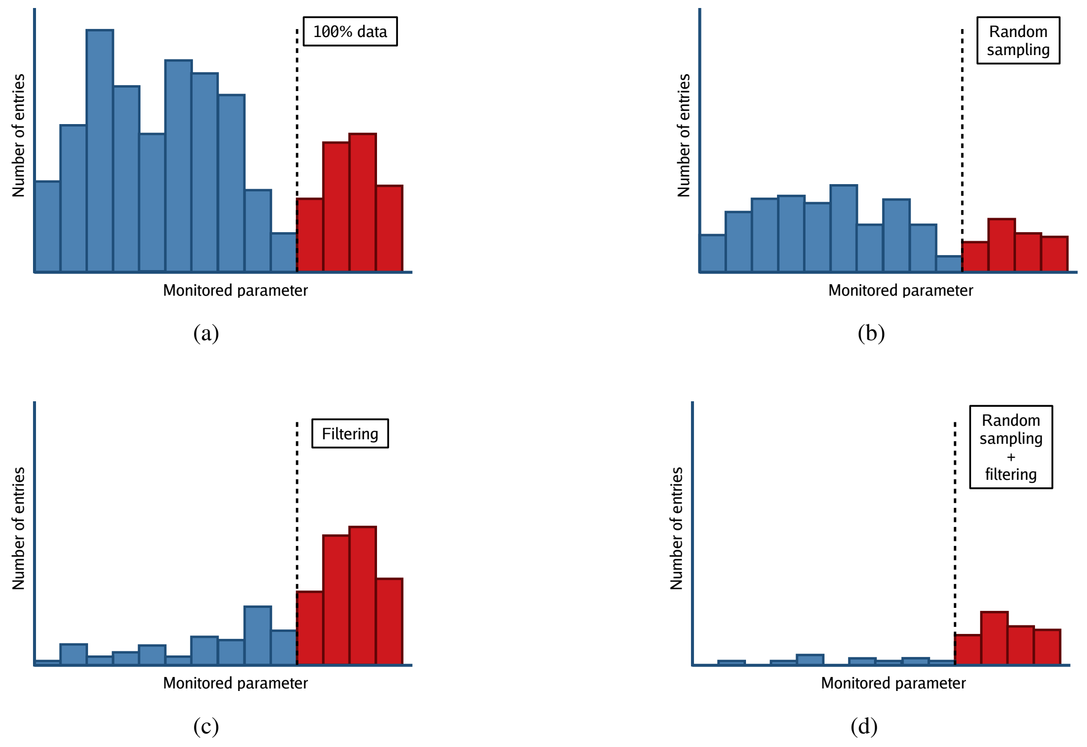


Fig. 5.5. An illustrative comparison of amounts of monitored data with different sampling methods. Blue bins represent the correct data, red bins correspond to bad quality data.

5.2.3. Summary

Choosing the message-passing approach for Data Sampling provided multiple benefits in terms of functionality and scalability. It allowed to easily sample data produced by any other actor which also relies on the Data Processing Layer. As Dispatcher is a separate process with its own state machine, it can be reconfigured before and during data acquisition without affecting the rest of the processing topology, unless it becomes too slow and the upstream message queues are filled up. Dispatchers may operate independently on multiple workflows and servers. Not only QC Tasks, but also any other Data Processor may subscribe to sampled data, which allows to use this component outside of the QC system. This possibility was already used during commissioning tests of the MFT sub-detector.

5.3. Quality Control tasks and automatic checks

QC Tasks and QC Checks are the main processing entities in the synchronous quality control. Tasks should receive declared data stream, process it accordingly and generate some kind of condensed statistical information, which is encapsulated by Monitor Objects. They can subscribe to Data Sampling Policies or to certain data type directly in case they require all messages. Checks are supposed to evaluate data produced by Tasks and other actors in the processing chain and return quality labels in form of Quality Objects. The Task Runner and Check Runner processes execute the respective algorithms, provide them with data and publish their results. They are implemented on the top of the Data Processing Layer, so they can be treated as any other Data Processors in the O^2 system.

Tasks and Checks follow the template method pattern [107]. User algorithms should inherit the interface classes, which are handled in a uniform fashion by the framework. Each detector team creates its own library which contains the algorithms, but it does not impose a dependency on the QC core library. The framework loads Tasks and Checks using the ROOT reflection mechanism.

5.3.1. Task Interface

Tasks' lifetime consists of Activities and Cycles. One Activity spans the total time when a Task is running, since the *Start* transition until the *Stop* transition. While it may correspond directly to one data-taking run, it is not strictly required. For example, a shifter might execute a short-lived Task several times during a run.

An Activity can consist of one or more Cycles. The framework publishes Monitor Objects at the end of each Cycle. The Cycle duration is configurable with default values 10 s and 60 s for the development and production systems, respectively. By manipulating the Cycle duration, one can easily scale the resources usage of a large portion of the QC infrastructure. Setting a longer duration decreases the amount of merged objects, performed checks and the repository I/O operations, however, at the cost of sparser monitoring updates to a shift crew.

A developer might register data structures as Monitor Objects with a dedicated manager class. The most recent version supports only ROOT data types, but this limitation will be lifted in future releases. The interface also allows to set metadata to Monitor Objects in form of keys and corresponding values.

The Task Interface consists of several abstract methods which user classes should implement. These include:

– `startOfActivity, endOfActivity` – the two methods should include nonrecurring operations, done when Activities are started and finished.

– `startOfCycle, endOfCycle` – similar to the methods above, but referring to the Cycle lifetime.

– `initialize` – here, the statistical data structures should be constructed and registered as Monitor Objects. Any other procedures, which are held once and may take a longer time, should also be performed in this method. This way, Activities and Cycles can start as fast as possible.

– `reset` – the published Monitor Objects should be cleaned up. This step is especially important for correct merging, as described in Sec. 5.4.

– `monitorData` – it is invoked each time when there is a new data set available. The user algorithm should fill its data structures by processing the incoming messages, which are directly accessible by the interface provided by the Data Processing Layer.

The Task lifetime might end either when it reaches a specified maximum number of Cycles, or by a manual intervention or due to reception of an End Of Stream message, which is a special payload propagated by the DPL notifying that no more data will come to given Data Processor. In each case, the QC framework will safely execute the end of Cycle and Activity methods, and publish the latest, complete versions of Monitor Objects.

A QC Task might run on the same machine as the main processing chain (locally) or on a QC server farm (remotely). In the first case, if the same Task is duplicated on parallel machines, their Monitor Objects need to be merged (see Sec. 5.4). Remote Tasks might receive sampled data from one or more FLPs or EPNs. These alternatives were introduced to better utilise computing resources accordingly to sampled data throughput and Monitor Objects sizes. The procedure on how to choose one option over the other was described in Sec. 6.4.

5.3.2. Timing of Monitor Objects publication in QC Tasks

QC Tasks exist on the border of two time domains. Input data is in general based in TimeFrame domain and the processing is invoked upon reception of a complete data set. On the other hand, Task output messages are published at the end of each Cycle, therefore, in specified and constant time intervals.

The design of the O^2 system expects that each processing device is data-driven and in absence of incoming messages, no new messages should be produced. Task Runners deliberately bend this rule by using timer inputs. This way, Cycle durations are strictly preserved and Monitor Objects are published even when no input data arrives. It gives an additional debugging information. By publishing results, it proves that a QC Task is alive, but receives no data. Also, it maintains the expected Activity duration if it is set to have a limited number of Cycles.

Thousands of parallel QC Tasks which start their activities at the same time might actually cause problems related to load distribution. If all Monitor Objects are published at the same time throughout the full O^2 system, they might suddenly and drastically increase the CPU usage, the burden on network cards and the RAM usage. Therefore, a load distribution mechanism is needed. Ideally the AliECS should start the respective processes in an equally diffused intervals. If this is not possible, the first Cycle of each Task will be randomly shorter.

5.3.3. Check Interface

Checks perform automatic evaluation of data produced by QC Tasks. The checking result is a Quality, which is stored in Quality Object together with additional information, such as metadata added by the framework or user. The set of Qualities to choose include *Good*, *Medium*, *Bad* and *Null* (one cannot determine the Quality). By accessing metadata, user algorithms can provide additional comments indicating the reason of returning certain Quality.

The initial Check prototype supported a quite restrictive relationship between Monitor Objects and Quality Objects. Each Monitor Object would have a list of Checks which should be performed on that specific object. One could not run a Check which would collectively evaluate multiple Monitor Objects produced by the same or different Tasks. Moreover, the Check assignment used to be configured inside the C++ source code, thus it could not be changed without recompiling a detector library.

The author proposed an extension the framework which should allow for an N to M relationship between Monitor Objects and Quality Objects. Also, Checks should be declared inside configuration structures, not in the compiled code. An AGH UST master student, Rafał Pachołek, implemented the proposed changes as a part of his project for master's thesis [108].

The Check Interface requires developers to override the following methods:

– `check` – receives a map with one or more Monitor Objects which were ordered by this Check. It should return a Quality.

– `beautify` – allows to perform additional changes in Monitor Objects based on the recent Check results. For example, in the case of ROOT data types, one can add a text box on a chart, configure axes ranges and set a background or plot colour.

Other components of the O^2 system might also generate objects which could be put under quality control. To support such cases, Checks can also accept external data sources which are available within other DPL workflows.

The possibility to check more than one Monitor Object at one time raises a question on synchronisation of multiple data sources. If just one object among the expected set is actually updated, should it trigger a Check? The framework lets users decide with the following update policies:

– `OnAny` – run a Check if any Monitor Object was updated.

– `OnAnyNonZero` – run a Check if any Monitor Object was updated, but all of them arrived at least once.

– `OnAll` – run a Check if all Monitor Objects were updated.

– `OnEachSeparately` – run a Check on each updated object separately. This policy allows to receive a separate Quality per each expected Monitor Object.

5.3.4. Dealing with complex Task and Check mapping

The N to M relationship between Monitor Objects and Quality Objects makes the framework more flexible, however, it introduces another level on complexity in handling objects' beautification and storage. In order to avoid unnecessary messaging, Check Runners are responsible for storing both beautified Monitor Objects and produced Quality Objects. Here, one has to distinguish several cases (Fig. 5.6a) to avoid storing duplicates, while allowing many Checks to perform beautification on the same object and to distribute the processing load among multiple processes.

Checks are performed within the same process if they require the same set of Task inputs (Fig. 5.6b). They may beautify input objects only if they come from one Task. Then, the Quality Objects are sent to the QC repository by the same process. If a set of objects generated by a Task is not required by any Check, or it is, but always in conjunction with an output of other Task, then they are stored by a dedicated *writer* process. This scheme guarantees that only one version of an object is stored in the repository and it can be beautified by Checks which operate on an output from one Task. This algorithm was proposed and implemented by Rafał Pachołek in [108].

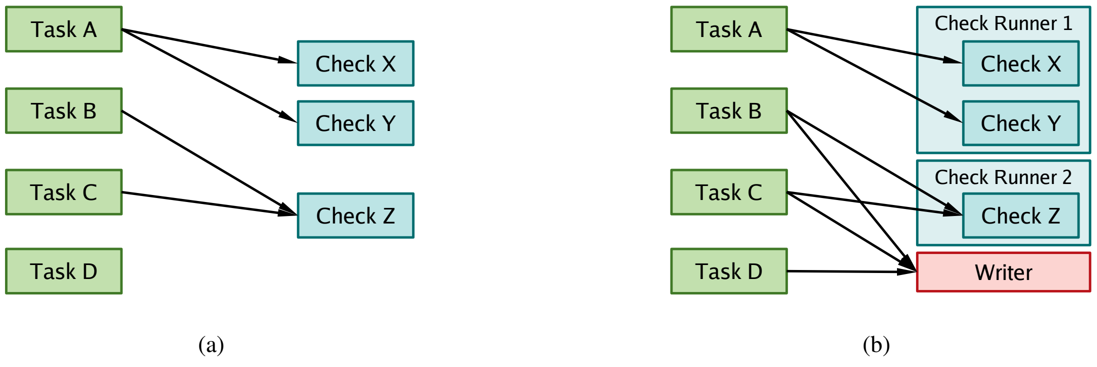


Fig. 5.6. An example of Tasks and Checks mapping (a) and how it is arranged by the framework (b).

5.3.5. Quality aggregation

The whole system is expected to generate around 100000 new Quality Objects every minute. While this level of detail is necessary to clearly pinpoint the issues with acquired data, such an amount of information might overwhelm both shifters and experts at first. Hence, a possibility to create Quality Object aggregates was requested by many detector module developers, even though it was not present in the original design [6].

Aggregators allow to group Check results into categories which correspond to specific features of sub-detectors and even to obtain a global sub-detector data quality (Fig. 5.7). User-defined Aggregators and a set of common algorithms inherit a simple interface with the `aggregate` method to implement.

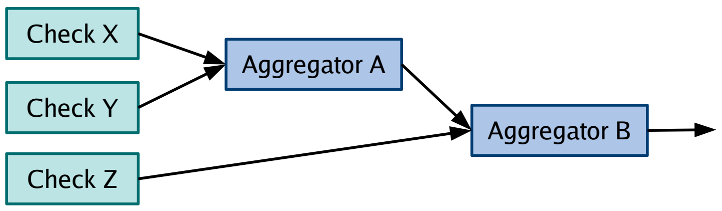


Fig. 5.7. An example of Checks and Aggregators arrangement. Aggregator A combines the Quality Objects produced by Checks X and Y. Since Aggregators also produce Quality Objects, they may serve as data inputs for other Aggregators and can be mixed with Check results, as it is done in Aggregator B.

The execution of Aggregators can be subjected to the same update policies as Checks, with the exception of the `OnEachSeparately` policy which would not provide any meaningful results.

In its initial implementation, done by Barthélémy von Haller, the aggregation is performed by one actor, Aggregator Runner, which subscribes to all Quality Objects which are required by declared Aggregators. As the task performed by this process does not require a lot of computing resources, it is expected to sustain the data stream of all Qualities produced by Checks. If needed, one may group Aggregators within one process per sub-detector to increase the maximum load capacity.

5.4. Mergers

The QC framework allows to execute QC Tasks locally, i.e. on the same servers where the main processing is performed. The existence of this possibility is dictated by potentially excessive costs of transferring and combining large amounts of data to machines dedicated for Quality Control. The results of parallel data processing have to be merged before further analysis.

The software responsible for merging data from multiple parallel sources is the topic of this section. The author developed the solution used in the QC framework and performed an extensive review of different approaches to this problem. The work also benefits from the experiences of merging data in the ALICE High Level Trigger software [109] and initial studies done for the O^2 by a master student, Patryk Lesiak [7]. The author submitted a paper covering this topic to a peer-reviewed journal, it is in the review process. App. B contains a thorough performance study of mergeable data types available in the ROOT framework and a comparison of standard histograms in ROOT and `boost`.

5.4.1. Assumptions and naming convention

As indicated in Sec. 4.5, the literature review showed that this topic does not have a commonly used naming convention. Even the software itself is referred by different names: Mergers, Gatherers and Adders. It is highly possible that other terms exist, which could have limited the amount of related publications found. In this dissertation the following taxonomy is used:

– *Data point* – a set of one or more measurements performed during one observation, e.g. a table row or singular entry in a histogram bin

– *Data type*, *data structure* – a storage format for data points

– *Object* – an instance of a data structure

– *Incomplete object* – an object which contains a subset of all data points currently present in the system

– *Complete object* – an object which contains all data points currently present in the system

– *Merging* or *merger* – an operation which combines two or more *input objects* into one

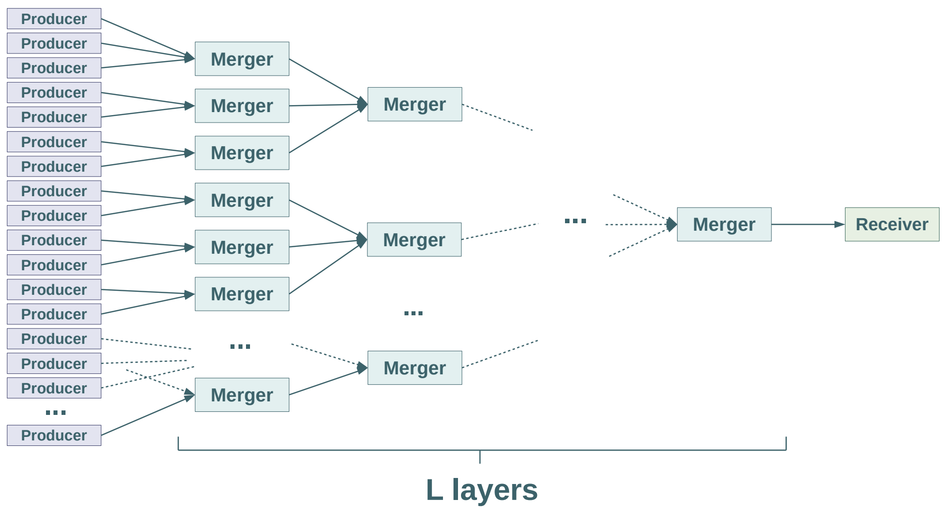


Fig. 5.8. A multi-layer topology of Mergers.

- *Delta* – an input object which contains only the data points which a Merger has not yet received
- *Entire object* – an input object which contains all data points gathered by a data source so far
- *Data source*, *Data producer* – an actor which publishes objects. They are incomplete, unless it is the only data source in the system
- *Merger* – an actor which merges incomplete objects
- *Data receiver* – an actor which receives objects from Mergers

For the convenience of discussing differences between merging entire objects and deltas, *fixed-size objects* and *growing objects* are defined. The first do not change their size after merging, e.g. histograms with constant binning. The latter are larger than the largest input object after merging. For example, tables fall into this category.

It is assumed that the merger result do not depend on the merging order and it is an instance of the same data structure as input objects. Also, having the same input objects, one should obtain the same result regardless of the merging algorithm used.

Data structures may contain other types within. Collections of histograms, tables and other custom objects are expected in the QC system. In such case, they are merged by matching corresponding objects names. If a previously unseen object is found, it is copied into the target collection.

5.4.2. Mergers design

The implementation of Mergers follows the general design principles of the O^2 software - they are message-passing actors respecting the O^2 data model. They may be used in other context, unrelated to the QC system.

The input load can be distributed among several actors with a multi-layer topology of Mergers (Fig. 5.8). The first layer of Mergers receives input objects from data producers, merges them and passes to the next layer. The last layer always consists of only one Merger, which publishes the complete object. If we require that each Merger should handle a similar amount of input channels, then the number of processes per layer M_i can be calculated with Eq. 5.5.

$$M_i = \lceil M_0^{\frac{L-i}{L}} \rceil \tag{5.5}$$

where L - the number of layers, i - the layer index, 0 being the producer layer, while Mergers start from layer 1. Thanks to the ceiling function, an integral number of Mergers is obtained.

Alternatively, one may define a global reduction factor R as the maximum number of inputs handled by one Merger. Then one may find the amount of processes per layer with Eq. 5.6.

$$M_i = \lceil \frac{M_{i-1}}{R} \rceil \tag{5.6}$$

Consequently, the reduction factor per each layer is:

$$R_i = \frac{M_i}{M_{i-1}} \tag{5.7}$$

The total number of layers is:

$$L = \lceil log_R M_0 \rceil \tag{5.8}$$

The author designed and implemented two kinds of Mergers, one of them dedicated for merging entire objects, the other one supports deltas. They are characterised with different advantages, complexity and corner cases, which will be discussed in the following sections.

5.4.3. Merging entire objects

The general behaviour of the entire objects Merger is illustrated in Fig. 5.9 and Fig. 5.10. The cache stores the most recent version of an incomplete object from each data source. Merging is performed in regular intervals. Each time, a new complete object is created and published.

The presence of cache is necessary to avoid merging the same data points twice. Input objects should not be merged as soon as they are received, since their order of arrival is not guaranteed. Thus, certain data source could send an object more than once before another data source sends it. By storing them in cache, the merged object always is created from the most recent objects. Moreover, if a data source accidentally stops working, a Merger can still use the last version of its incomplete object.

When merging entire objects, one gains some advantages over the other alternative. First of all, a Merger may start later than the data sources and its temporary failure does not cause any loss of data. Since data sources send entire objects, no data points are lost unless the sources start to malfunction. As soon as a Merger receives objects from all input channels, it may produce the correct complete object.

However, the presence of a cache puts relatively large CPU, memory and bandwidth requirements compared to the other alternative. In case of growing objects (e.g. tables), all data points are transferred and merged repeatedly instead of once. Also, the presence of a cache requires memory for one incomplete object from each data source.

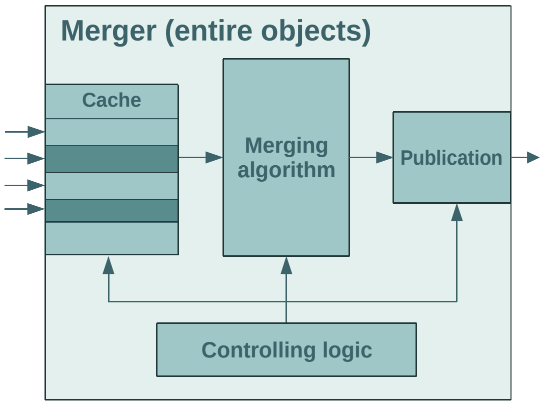


Fig. 5.9. The inner logic components of the entire objects Merger.

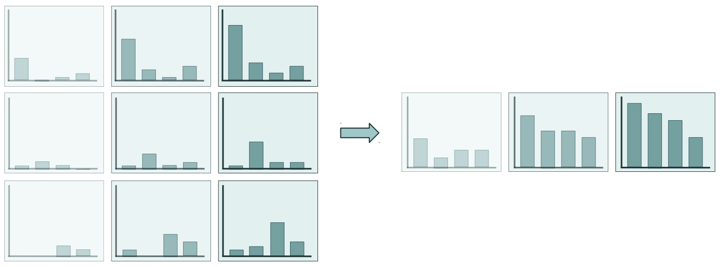


Fig. 5.10. Merging entire objects. Each received object (on the left) replaces the previous one from the same data source. The complete object (on the right) is created from scratch by merging the most recent incomplete objects.

5.4.4. Merging deltas

The logic behind the delta Merger is depicted on Fig. 5.11 and Fig. 5.12. First, there is an optional cache to store new deltas. They are merged with the stored object, which gathers all data points received so far. It is published in regular time intervals. In multi-layer Merger topologies, all the Mergers excluding the final layer reset their stored objects after publishing them in order to avoid data points duplication.

In this case, the cache is optional as there are no prerequisites to delay merging input objects. However, it may potentially provide performance benefits by allowing to merge many objects at the same time (discussed in Sec. 5.4.5). The provenance of objects is insignificant, since no data points are sent more than once. Therefore, the objects can be stored in one queue. Due to the considerations and benchmark results in Sec. 5.4.5, the cache was not included in the final version.

In terms of the fail-safety, if a delta Merger abruptly stops functioning and needs a restart, then all contained data points so far are lost. To prevent this, one could extend it to store backup versions of complete objects. On the other hand, if such situation occurs in a data source, the system will lose only the data points which were never published by the producer.

Since only object differences are passed to the Merger, the hardware requirements for merging growing objects become lower. Transferring deltas puts smaller load on the bandwidth. Merging only new

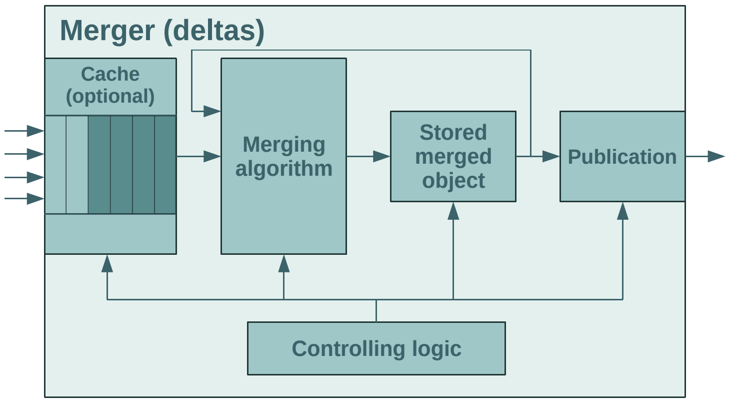


Fig. 5.11. The inner components of the delta Merger.

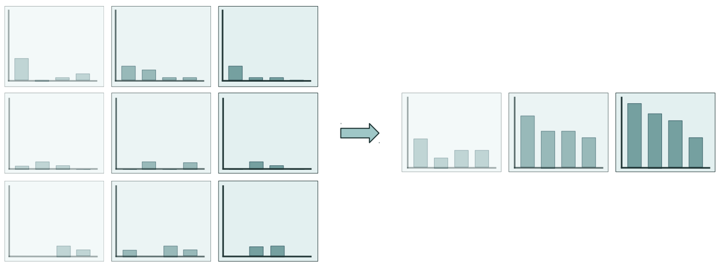


Fig. 5.12. Merging deltas. Each received object update (on the left) is integrated with the most recent complete version (on the right).

data points might require less CPU time, depending on the data structure used. Removing cache also lowers the amount of memory needed.

5.4.5. Merging collections of objects

Merging certain data structures in larger collections might provide performance benefits. For example, the mergeable ROOT data structures support merging collections of objects. Because of this, the potential performance improvements were evaluated.

The five most popular ROOT types among the planned and existing QC Tasks were benchmarked: standard histogram types with one, two and three dimensions (`TH1I`, `TH2I`, `TH3I`); multi-dimensional sparse histogram (`THnSparseI`), which allocates memory only for non-zero bins; and columnar data storage (`TTree`). Fig. 5.13 illustrates the CPU time needed to merge an object as a function of the total amount of objects in a collection. The size of all the input objects was aligned to 250 kB, with the exception for `THnSparseI`, whose in-memory size is dependent on the distribution of the contained data points. Its size could not be reliably determined, as it would be related to the pseudo-random values used to fill it

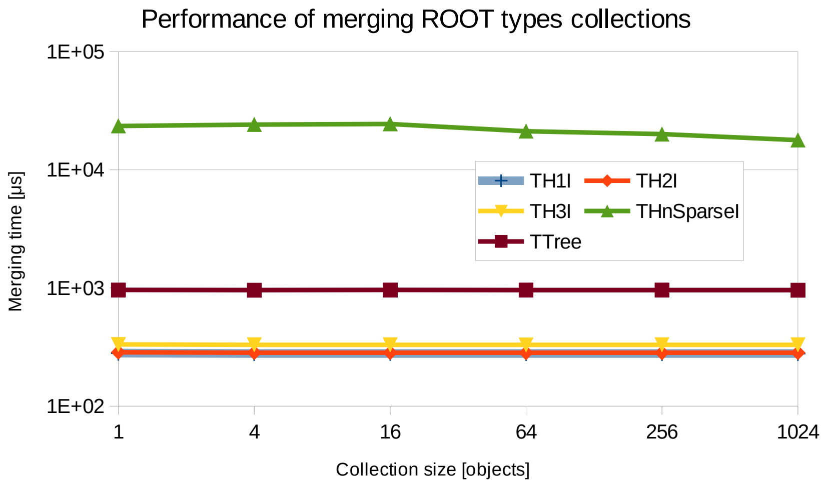


Fig. 5.13. Merging time per object as a function of the number of objects in a collection.

No performance differences were observed for the standard histogram types and the columnar storage. On the other hand, a collection of 1024 sparse histograms is merged around 24% faster than 1024 individual objects, which is a relatively small, but noticeable improvement.

According to the benchmarks results, the delta Merger would not benefit much from merging objects collection. Therefore, the input objects are merged as soon as they are received in the current version of Mergers. Only sparse histograms show performance benefits in the CPU time, which does not compensate the amount of memory needed for caching objects and the increased complexity which comes with having a cache.

5.4.6. Comparison of merging entire objects and deltas

The two Mergers, one for entire objects and another for deltas, were implemented. Any of the two may be chosen for specific parallel QC Tasks. This sections contains a comparison between these alternatives and the default solution is chosen based on the considerations.

Fig. 5.14 shows the ratio of CPU time needed to merge an entire object and a delta if the latter contains 100 times less data points. Fixed-size objects, such as `TH1I`, `TH2I` and `TH3I`, require the same computing resources regardless of the option chosen. Large performance gain was observed for sparse histograms - merging 100 times less data points corresponds to 62 times shorter merger operation on average. The precise improvement might be very dependent on the distribution of data stored within `THnSparse`. `TTrees` are merged 75 times faster with deltas containing 100 times less entries.

Summarising other differences, delta Mergers do not require caching objects, which translates to smaller memory needs. Also, the temporary failure of data sources causes less data losses with this alternative. Additionally, they might require less bandwidth, memory and CPU time when merging growing

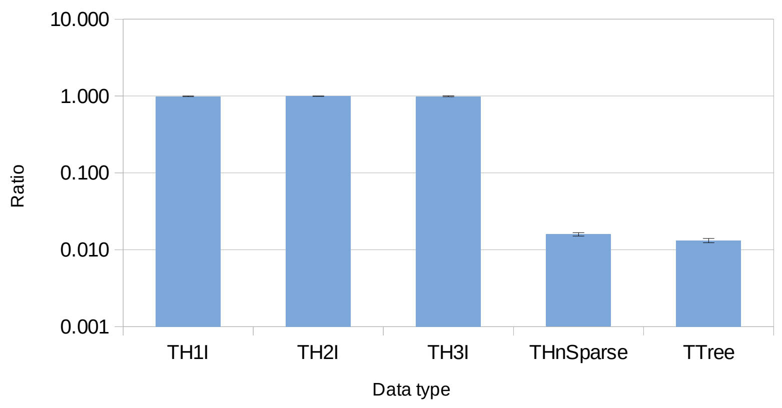


Fig. 5.14. Ratio of CPU time needed to merge entire objects and deltas. The former contained 5000 entries, while the latter only 50.

objects. On the other hand, entire object Mergers can restart during data acquisition, as each time they receive all data points gathered so far by data sources. In case that only the final complete object is needed, they may merge objects only once, at the end of data-taking.

Given the benchmark results and the functionalities comparison, Mergers will operate in the delta mode by default, unless it is required by any specific needs of certain QC Tasks.

5.5. Quality Control repository

The work described in this section was done primarily by Barthélémy von Haller. The author contributed by taking part in discussions about the objects storage design and reviewing the implementation code.

Monitor Objects and Quality Objects are stored in the QC repository, where they can be accessed by graphical user interfaces, experts and further processing tasks. Initially, the QC repository was implemented on the top of MySQL [6] - an open source SQL database management system [56], which was also used in the ALICE Data Quality Monitoring software during Run 1 and 2.

However, the original solution was replaced with the Condition and Calibration DataBase (CCDB) – a technology built especially for the ALICE's needs regarding storage of quickly changing detector parameters and calibration values. The change was justified by a decrease of the system complexity, effectively requiring the users to learn how to use one kind of database in the O^2 system instead of two. The CCDB offers a convenient Representational state transfer (REST) interface [110] to store, browse, retrieve and delete entries across time. It fulfils the Quality Control needs for data organisation, where objects under the same name are published recurrently and have a limited validity in time, as they refer only to a specific time period of the detector operation. The interface also allows to add metadata to stored objects and then apply filters to choose entries with specific metadata values. Also, the CCDB was proven to sustain the projected load of the QC system (see Sec. 7.5). It should be noted that two instances of the CCDB will be present in the O^2 system. One will serve its primary purpose - storing the detector condition and calibration values. A separate instance will be used as the QC repository.

The framework stores QC objects under structured paths in the form *qc/<detector name>/<object type>/<component name>/<object name>*. A 3-letter detector name, e.g. ITS, TPC or MCH, organise objects into detector-specific groups. The object type can be either *mo* (Monitor Object) or *qo* (Quality Object), thus the two kinds of data are put in separate directories. The component name might be e.g. a QC Task or Check name and they aggregate all their QC objects under a common path. The component and object names might consist of additional '/' separators, which allows the detector teams to organise their quality control results into custom sub-categories.

Monitor Objects encapsulate objects specified by users. In the current version of the framework, it is possible to use any ROOT object which inherits from `TObject` - the main ROOT's interface which consolidates most of its data types and standardises their behaviour in I/O operations, error handling, reflection and printing. In order to allow users to inspect their Monitor Objects in the database without the dependency on the QC framework, they are stored unwrapped from the encapsulating interface class object. However, this is not the case for Quality Objects, which consist only of custom fields - Qualities and metadata, not present in the ROOT framework.

The QC framework stores objects as ROOT's `TFiles`. This approach ensures data persistence, since such file contains information about file contents and data types, including their versions. Thus, even obsolete classes can be properly read back.

Most of the QC objects is updated each 60 seconds during data-taking, so the experiment shift crew can inspect data quality in a timely manner. However, after a data acquisition run is over, one does not need all the corresponding objects - one version per hour should usually suffice. Reducing the number of permanently stored objects should decrease the disk storage requirements and response times to object retrieval and filtering. The removal of unnecessary objects is performed by a periodically executed Python script. It is able to clean up a declared directory path according to specific policy - e.g. keeping only one object per hour, one per run, only the most recent version or removing all objects. Since operations on databases involve a risk of corrupting or removing data, the script can perform a *dry run*. Then, it shows the list of operations it would carry out without actually applying them.

5.6. Post-processing

QC Tasks operate on data coming from the main processing chain, synchronously or asynchronously to data-taking runs. This, however, does not cover the needs of higher-level quality analysis to inspect trends in data and to perform correlation studies between different observables.

The post-processing framework, a part of the Quality Control software, is responsible for post-processing of Monitor Objects and Quality Objects, including information from other data sources. It should store the derived objects in the QC repository, thus also allowing to inspect them using the same graphical interfaces and other tools. The processing should be triggered on event basis, such as at the end of each data acquisition run, or once per day. The exact list of triggers is discussed in 5.6.1.

While developers can experiment and run the software on their own anytime, this framework is mainly dedicated for processing which should be triggered in an organised manner. To play with data created by the Quality Control, users are encouraged to use e.g. ROOT macros or the Jupyter Notebook [111] instance hosted at CERN [112], which provides an access to ROOT.

5.6.1. Post-processing Interface

Post-processing in the QC framework, as it is for Tasks and Checks, also relies on the template method pattern [107], minimising the effort which users would have to put to start developing their modules. The Post-processing Interface unifies all the Post-processing Tasks under a common, basic interface with four methods to implement - `configure`, `initialize`, `update` and `finalize`. The first pushes configuration parameters to an algorithm. The remaining three are invoked according to configured triggering events. Initialisation and finalisation are executed only once, in contrast to the update method, which is invoked any time after having been initialised and when corresponding triggers apply. So far, the following triggers are defined:

– Start Of Run, End Of Run – triggers when a data acquisition run is started or finished, respectively.

– Start Of Fill, End Of Fill – triggers when particle beams are being inserted into the LHC and after they are dumped, respectively.

– Periodic – triggers regularly with a specified interval time.

– New Object – triggers when a specified object in the QC repository is updated.

– Once, Always, Never – trigger once, each time or never. They are useful for testing the implemented software.

– User Or Control – triggers when a user performs some kind of manual intervention. This includes receiving a program interrupt request (e.g. by pressing CTRL+C in terminal) or a *Stop* state change request sent by the AliECS.

In contrast to QC Tasks and QC Checks, which rely solely on the framework to pass data around, Post-processing Tasks should retrieve objects from databases. They may also store data voluntarily. However, using a dedicated interface is recommended, because it allows to pass the objects to Checks. Summarising, this part of the QC framework leaves a lot of freedom to users. It allows to design and implement unusual algorithms, which cannot achieved with other QC components. On the other hand, numerous usage requests cover moderately simple and common applications which can be unified under one algorithm, described in the next section.

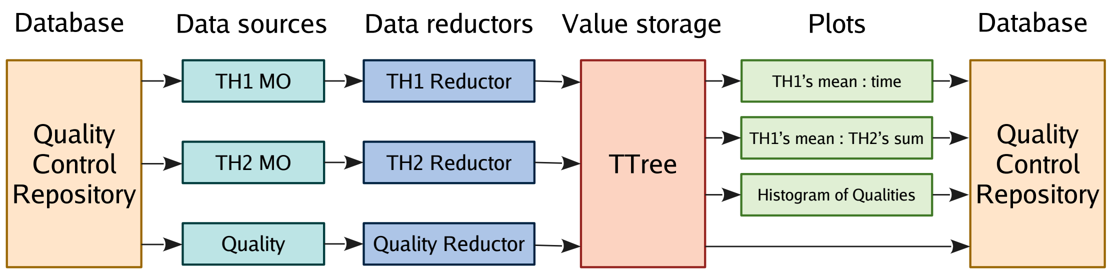


Fig. 5.15. The Trending Task design.

5.6.2. Trending Task

Based on gathered requirements for post-processing in Quality Control, the usual desired sequence of operations consists of regularly retrieving one or more QC objects, deriving some kind of statistics and plotting them over time (trending) or against each other (correlation). The Trending Task is a specialisation of the Post-processing Interface class. It is supposed to perform commonly requested activities related to the post-processing without requiring users to write any code.

Fig. 5.15 depicts the Trending Task structure. The task may request any data from the QC repository. It can be a Monitor Object, Quality Object or any other class object which inherits `TObject`. Reductor classes are responsible for computing or retrieving high-level information out of specific data types, e.g. the average and standard deviation of a histogram. The framework provides a common set of Reductors for the standard ROOT histograms, sparse histograms and Quality Objects. The last one allows users to trend data quality in time and correlate them with other observables. The task writes sets of observables as rows into the ROOT's tabular data storage facility - `TTrees`. Aside from I/O operations, this class offers a powerful visualisation tool, which allows to create plots and histograms from underlying columnar data. Effectively, it allows to correlate values and draw trends. The Trending Task reads a list of plot configuration values and passes them to the `TTree::Draw` interface. The created figures and the tree itself are stored in the QC repository during each update and finalisation of the task.

5.6.3. Running Post-processing Tasks

As previously stated, Post-processing Tasks can operate on data retrieved from databases. This allows to run them as standalone processes, without the Data Processing Layer. There are two dedicated executors which can run Post-processing Tasks, one is dedicated for development purposes, while the second allows the AliECS to drive its state machine by remote procedure calls. Still, the tasks can run within the Data Processing Layer as data sources and it is the only way which lets the objects reach Check Runners, so they might be automatically evaluated.

The Post-processing Tasks executors are mainly responsible for regularly checking validity of triggers. Doing this too often might put a lot of strain on other system components, for example on the

QC repository, by continuously inquiring if an object under specific path was updated. To mitigate the potential problems, the task executors run triggers periodically, in a configurable time interval. As the Post-processing results do not have to appear as soon as possible, having the tasks sleep for a few minutes should be acceptable in most cases. The workload could be further reduced by implementing a centralised trigger manager, which would decrease amount of operations done in parallel, and by letting the AliECS system handle triggers related to data acquisition runs and beam fills.

5.7. Execution

Any QC process topology can be started with one executable, without any need for its recompilation after changes the user code. The binary reads the provided configuration file and generates the corresponding infrastructure within the Data Processing Layer, which can immediately run it on development setups or it may be exported to the AliECS for running in production. A lot of attention was given to provide meaningful error messages when a QC topology cannot be correctly created due to missing or incorrect configuration parameters.

Thanks to the possibility of merging DPL workflows, one can easily attach a QC workflow to any other by extending the execution command with a Unix pipe and the QC binary name with necessary arguments (Lst. 5.1).

```
o2-some-processing | o2-qc --config json://path/to/configuration/file.json
```

Listing 5.1. An example of command which runs a main data flow workflow with the Quality Control software attached.

In the early development phase the users usually run the software on one server and test it with simulated or pre-recorded detector test data. Later, during the detector commissioning or data taking, the real detector can provide data to several machines in parallel. In such case, some elements of the QC infrastructure can run on the servers receiving and processing acquired data, while other parts can be executed on a dedicated QC server in order to merge and check the results. Then, the QC executable can generate only the appropriate part on each machine and allow them to function properly on such a multi-node setup.

Since both the QC and the analysis framework take advantage of the Data Processing Layer, they may interact with each other just as any actors in a workflow. One can configure QC Tasks to read Analysis Objects Data (AOD) files directly or to subscribe to tables and histograms generated by analysis tasks. The workflows may run locally or on distributed computing clusters.

5.8. Quality Control GUI

Shift crews and experts can access the results of Quality Control processing – Monitor Objects and Quality Objects – by directly interacting with the database or via the Quality Control GUI (QCG). It

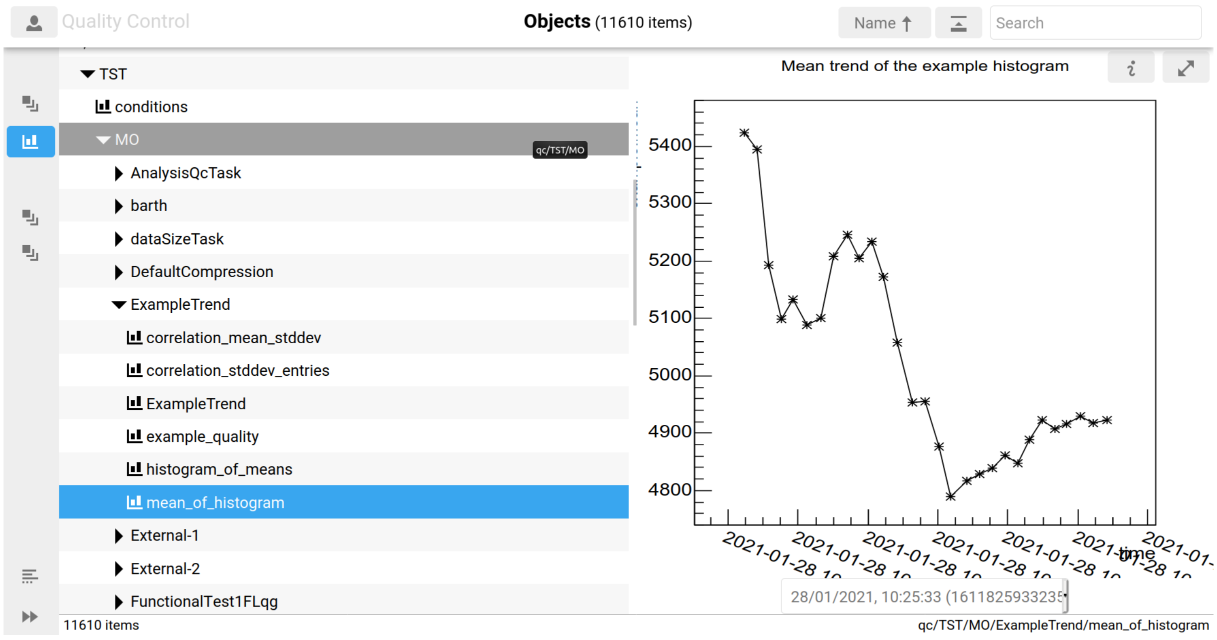


Fig. 5.16. The Quality Control GUI in the browser mode. The panel on the left allows to choose an object in the tree-like structure, which is then visualised on the right side. The selection panel in the right-bottom of the picture allows to choose any older version of the object. The selected object contains a trend obtained with the Trending Task.

is based on the O^2 WebUI framework [113], relying on Web technologies to build graphical interfaces which do not require anything else than a web browser to run and can be accessed from anywhere in the Internet. Both the framework and the QCG were developed by George Raduta, Vladimir Kosmala and Adam Węgrzynek.

The GUI allows to inspect objects stored in the QC repository in two major ways. The first (Fig. 5.16) illustrates available entries in a tree-like structure which represents their arrangement in the repository (see the path description in Sec. 5.5). User can filter out only currently running Tasks, Checks and their corresponding objects. By default, the most recent version of an object is shown, but one can select also past objects with a selection panel. The other mode allows the users to arrange several plots into layouts (Fig. 5.17), which are saved by the server application and can be shared among other users.

The QC propagates the information about available services to the QCG via Consul - a software technology for service discovery, distributed key-value storage, segmentation and configuration [114]. The ROOT objects are visualised with JSROOT [53], which implements ROOT graphics for web browsers. Quality Objects are illustrated as maps of keys and values, as shown in Fig. 5.18.

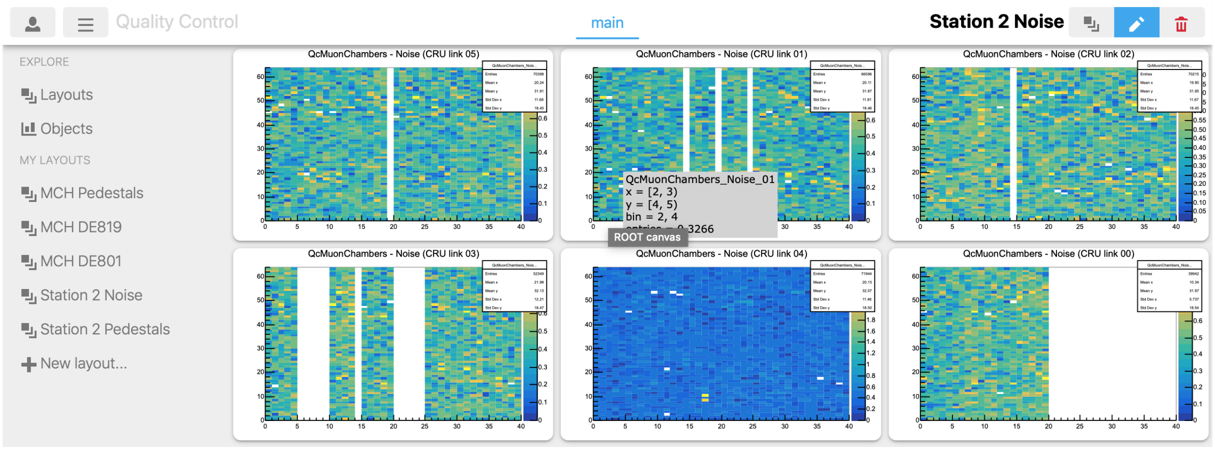
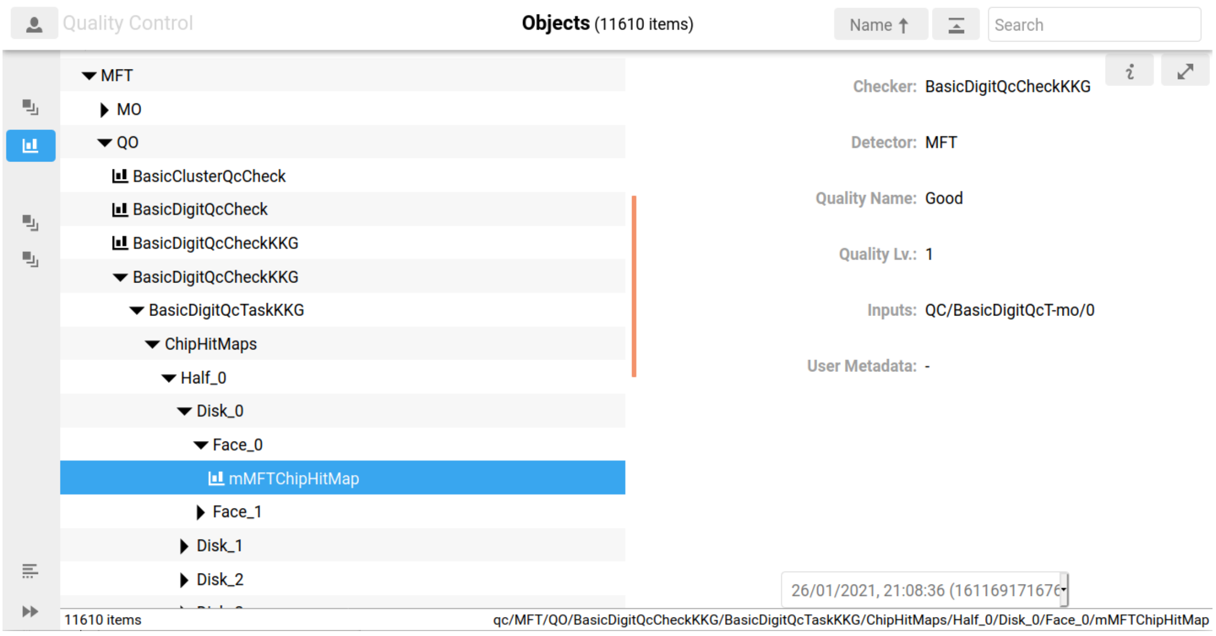

Quality Control
main
Station 2 Noise
EXPLORE
Layouts
Objects
MY LAYOUTS
MCH Pedestals
MCH DE819
MCH DE801
Station 2 Noise
Station 2 Pedestals
New layout...
QcMuonChambers - Noise (CRU link 05)
QcMuonChambers - Noise (CRU link 01)
QcMuonChambers - Noise (CRU link 02)
QcMuonChambers - Noise (CRU link 03)
QcMuonChambers - Noise (CRU link 04)
QcMuonChambers - Noise (CRU link 00)
QcMuonChambers_Noise_01
x = [2, 3)
y = [4, 5)
bin = 2, 4
ROOT canvas

Fig. 5.17. The Quality Control GUI in the layout mode. On the left, one can choose between the available layouts or request to create a new one. On the right, a layout example is shown, it is composed of six histograms with noise measurements of the Muon Chambers (courtesy of Andrea Ferrero).



MFT
MO
QO
BasicClusterQcCheck
BasicDigitQcCheck
BasicDigitQcCheckKKG
BasicDigitQcCheckKKG
BasicDigitQcTaskKKG
ChipHitMaps
Half_0
Disk_0
Face_0
mMFTChipHitMap
Face_1
Disk_1
Disk_2

Checker: BasicDigitQcCheckKKG
Detector: MFT
Quality Name: Good
Quality Lv.: 1
Inputs: QC/BasicDigitQcT-mo/0
User Metadata: -

26/01/2021, 21:08:36 (161169171676

11610 items qc/MFT/QO/BasicDigitQcCheckKKG/BasicDigitQcTaskKKG/ChipHitMaps/Half_0/Disk_0/Face_0/mMFTChipHitMap

Fig. 5.18. The Quality Control GUI visualising a Quality Object as a key-value map. In the browsed tree one can notice the fine grained structure of objects, achieved by using the '/' separators within the object names.

5.9. Maintenance of the QC software

As most of the ALICE O^2 software, the Quality Control is publicly and freely available [115] under the GNU General Public License v3 [116]. It uses Git [117] as a distributed version control system. The software builds are done with CMake, which facilitates the build process within singular software projects, and with AliBuild, which takes care of building and installing any direct and indirect dependencies (as of January 2021, there is 49 of them in the minimal version!). The build infrastructure is handled by the Work Package 3 team of the O^2 project.

The project is divided into the framework code and detector module libraries, in a way that the first do not depend on the latter. Most of the framework is covered with unit and functional tests in order to guarantee its expected behaviour after any changes in the project or upstream dependencies. Unit tests in detector modules are not mandatory, but encouraged.

All contributions to the project have to follow a common set of the ALICE O^2 coding guidelines, which include formatting rules and good code practices. New contributions are managed and discussed in form of pull requests, which contain proposed code changes by a certain contributor. The Continuous Integration system builds latest versions of the Quality Control with the proposed changes, runs the tests and code formatting checks. A pull request should receive an approval from a repository maintainer and pass the test builds in order to find its way into the main code branch. First-time contributors should receive a positive review before their code is built to avoid security-related issues due to an untrusted code being executed on the servers at CERN. The software is always built on CERN CentOS 7 (the target system) and MacOS. Having two distinct platforms supported helps to find and identify bugs which are caused by using code which is not guaranteed by the standard to function in a defined and cohesive manner (undefined behaviour). The code is complemented with an online documentation. Users can introduce themselves to the framework with a step-by-step tutorial.

The framework development, detector teams requirements and their progress are reported and discussed in bi-weekly meetings. Information about new releases and their contents are communicated via mailing lists and plenary meetings of the O^2 projects.

5.10. Further development

The presented state of the Quality Control software was released as the version v1.9.0. The framework contains all the features described in the O^2 project Technical Design Report [29] and the Doctoral Student Programme description. However, the software will be further developed according to upcoming requests.

One of the planned features should provide a possibility to raise alarms upon pre-declared events, which could be results of aggregated Quality Objects. Alarms should be propagated to various information channels, such as mailing system, GUIs, text messages or popular chat services. With this mechanism, detector experts and experiment shift crew would quickly receive information that they should react to a potential problem during data acquisition.

In the current version, the Quality Control supports only ROOT types as Monitor Objects. While it is possible to have any class inherit `TObject` and, by that, allow the framework to support such a data type, it also unnecessarily increases the number of classes and code. This limitation should be removed by generating ROOT dictionaries of any class which required support, which is not intrusive to data structures.

The software is currently configured with JavaScript Object Notation (JSON) files, which are then translated into property trees available in the `boost` library. This approach, however, cannot scale well in the production system. Configuration key and values will be stored and propagated to processes by Consul [114]. Also, a graphical interface should allow to modify selected values in the centralized configuration store accordingly to user privileges.

Data recorded during the years 2009-2018 was assigned quality flags on a run number basis. These marks, stored in the Run Condition Table, were the criteria for deciding if a given run is appropriate for certain type of physics analysis (different analysis types require different sets of sub-detectors working well). However, since Run 3, due to constantly changing detector conditions and calibration, the final quality will be evaluated with a finer granularity. Each event will have an associated timestamp. Since the events will be gathered in TimeFrames, the final assessment of the data quality will be performed with such or larger time intervals. In the end, data tags will be derived, which will allow to select only the good quality data during the physics analysis.

6. Optimisation of message-passing topologies in the Quality Control

The Quality Control framework leaves a certain amount of freedom for the arrangement of its actors. This allows to choose configurations which use less computational resources. QC Tasks can run either on machines with the main processing workflows - locally (Fig. 6.1), or on dedicated servers - remotely (Fig. 6.2). The choice between the two options is dictated by the following requirements. On one hand, the Quality Control should have no negative influence on the main processing, i.e. it should not disrupt it or slow it down. Thus, QC Tasks running on dedicated servers are less likely to cause troubles with data acquisition and processing. However, the computations should also use the minimal possible amount of resources. In case that a QC Task requires 100% of data to produce valid results, it might not be feasible or efficient to transfer such a data stream to another machine. Then, locally running Tasks might use significantly less resources.

In the case when incomplete results are produced by local QC Tasks running in parallel, one should also consider finding an optimal Merger topology. If merging does not consume a considerable portion of a CPU core, then having one Merger process should be completely sufficient. However, when one-layer topology can barely sustain the input data stream, one might consider using multi-layer merging topologies.

In this chapter, methods to find optimal message passing topologies are proposed and complemented with an exploration of the model parameter space, which allows the reader to learn how different factors influence the performance and costs of QC setups.

6.1. Queueing theory

Queueing theory [118] is the mathematical study of queues, which was initially applied in telecommunication to estimate the amount of resources needed to provide communication paths between waiting clients within acceptable level of delays. In the most basic form, a single queueing node provides services to arriving clients. If the clients cannot be served immediately, they wait for the service in a FIFO queue. The actors in the O^2 framework can be modelled with the same mathematical methods, since their main activity is processing messages which arrive in message queues.

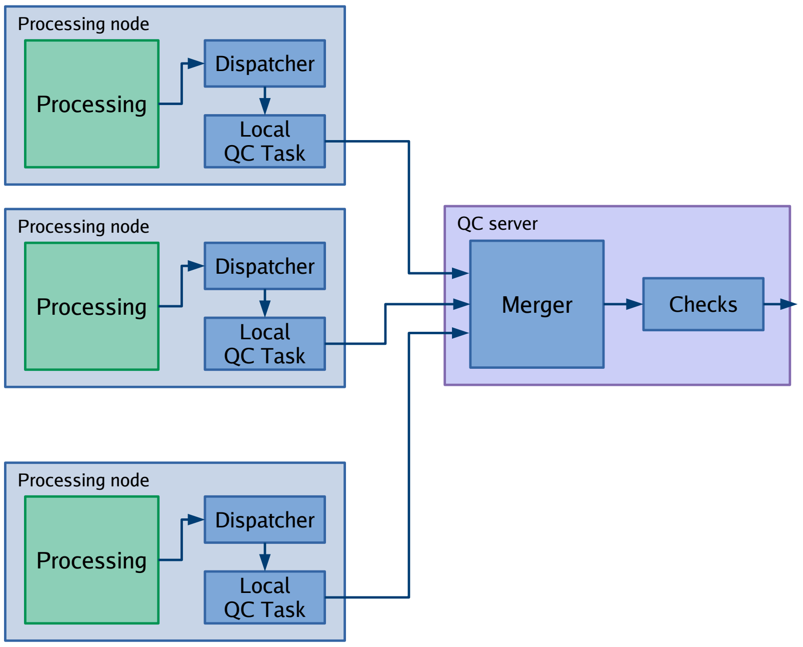


Fig. 6.1. A setup with local QC Tasks.

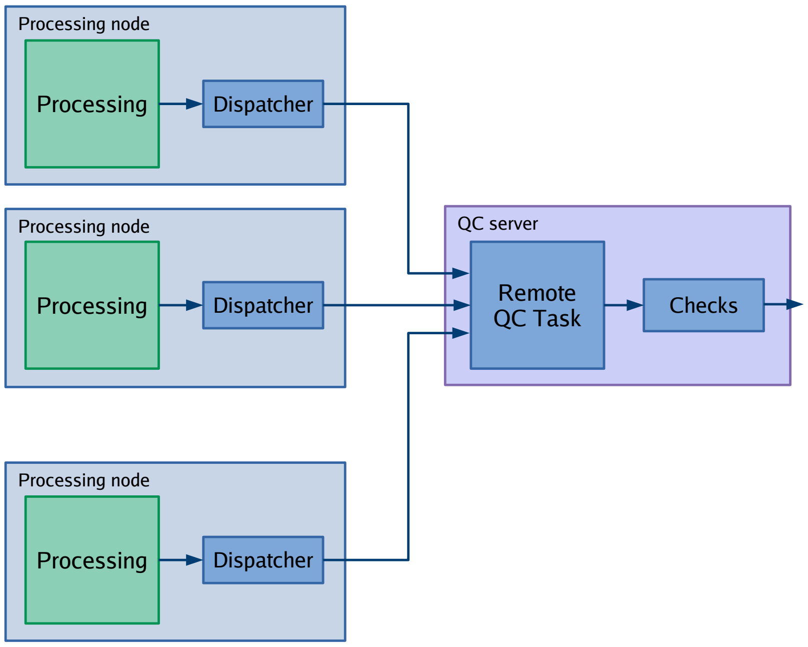


Fig. 6.2. A setup with a remote QC Task.

In the next sections two kinds of queue models are used. Their names follow the Kendall's notation, which arranges three specifiers in form of *a/b/c*. In this convention, *a* describes the client arrival process, *b* defines the service time distribution and *c* corresponds to the number of servers (workers). As defined in [118], *the M/G/1 queueing model assumes that customers arrive according to a Poisson process (M); the distribution of service times is arbitrary (G); there is one server (1); and all blocked customers wait until served.*

The average length ξ of a M/G/1 queue can be described with Eq. 6.2, assuming the utilisation factor ρ as in Eq. 6.1.

$$\rho = \frac{\lambda}{\mu} \tag{6.1}$$

$$\xi = \frac{\rho^2}{2(1-\rho)}\left(1 + \frac{\sigma^2}{\tau^2}\right) \tag{6.2}$$

where: λ - client arrival rate, μ - maximum service rate, τ - average service time and σ^2 - variance of service times. In the context of processing messages, the τ and σ components may correspond to the mean and variance of message sizes, assuming that the processing time is proportional to message size. Moreover, the utilisation factor ρ can be interpreted as the average CPU usage, if the process uses one worker thread.

If the service times are deterministic (D), then the queue is described as M/D/1. Consequently, the variance σ^2 in Eq. (6.2) is equal to zero and the formula for the average queue length ξ is reduced to Eq. (6.3).

$$\xi = \frac{\rho^2}{2(1-\rho)} \tag{6.3}$$

In this case, the queue length ξ depends only on the utilisation factor, thus one can use the set of calculated values in Tab. 6.1 to estimate it.

Tab. 6.1. Average length ξ of the M/D/1 queue for selected utilisation factor ρ values.

ρ	0.500	0.732	0.854	0.916	0.955	0.981	0.990
ξ	0.25	1.00	2.50	5.00	10.13	25.33	49.00

6.2. Model requirements

In order to clearly indicate applicability of the shown methods and avoid unnecessary complexity in the models, the following conditions are assumed:

- There are P main processing nodes, each produces a data stream subject to quality control with throughput D, average message size τ_d and its variance σ_d^2.
- Modelled processes can handle one message at a time, i.e. they have one worker thread.

– Time needed to process a message is proportional to its size. For QC Tasks it is described as Q_p - CPU usage per data throughput. For the commonly used ROOT types, this can be assumed for objects larger than 200 kB, as shown in Sec. B.4. Thus, CPU usage grows proportionally to the amount of received data, up to 100%.

– Hardware costs scale linearly. Of course, this statement does not apply in reality, but approximate costs can be found based on the price of already bought computing infrastructure. Therefore, we define: c_p - cost of a CPU core, c_m - cost of RAM per memory unit, c_b - cost of network bandwidth per throughput unit.

– Messages and contained objects generated by QC Tasks have a fixed total size o_{size}, i.e. they do not grow or shrink due to insertion of new data or merging. This is not true e.g. for tables and sparse histograms, so the worst case may be assumed - the highest expected size of an object, achieved after many hours of data acquisition.

– Aside from their message queues, local and remote QC Tasks need the same amount of memory, defined as Q_m.

– Merger needs memory only for objects stored in its message queue and cache.

– CPU cost of publishing messages is minimal and can be omitted.

– Transferred data has the same volume as processed data. In other words, serialisation and deserialisation do not have influence on objects size.

Probably all of the aforementioned requirements can be lifted up by extending the models below. The shown formulas are structured in a manner which allows for independent modification of any component. All parameters of the model are summarised in Tab. 6.2.

Tab. 6.2. The complete set of parameters used in the model.

Symbol	Description
c_p	Cost of a CPU core
c_b	Cost of a unit of bandwidth
c_m	Cost of a unit of memory
P	Number of main processing nodes running in parallel
D	Throughput of data produced by one node
τ_d	Average data message size
σ_d	Standard deviation of data message size
o_{size}	Size of all objects produced by one QC Task
T	QC Task's cycle duration
Q_p	Number of CPU cores per input data throughput needed by a QC Task
Q_m	Memory needed by one QC Task
$\mu(R)$	Object processing rate of one Merger with respect to the reduction factor R

6.3. Optimal Merger topology

The decision about finding the right amount of data sources per one Merger might involve multiple factors, two of them are highlighted in this introduction. First of all, one should take into account that due to the indeterminism of distributed message passing systems, random data throughput fluctuations might occur. Therefore, a Merger which is able to sustain certain maximum data flow should not be put under such strain. In case of a temporary increase of input data throughput, the Merger might have troubles resolving the congestion and, as a result, occupy more memory with its input message queues. Thus, a safe margin should be applied either arbitrarily or with statistical estimations, for example by minimizing the total computational resources usage of a Merger topology.

Also, overheads of some libraries and frameworks might relate to the number of supported actor inputs. If the performance decreases with a higher amount of handled data sources, a smaller reduction factor might be needed. If the performance is stable across the input number range, one can omit this aspect, otherwise it be taken into account when modelling this piece of software.

6.3.1. Modelling one Merger process

The queueing theory was used to model Mergers and its message queues. If we assume that arrivals of input objects are determined by a Poisson process, the merging time is constant and the Merger can process one message at the same time, then it can be represented by the M/D/1 queue. In this case, the worker utilisation factor can take the following two forms:

$$\rho_i = \frac{\lambda}{\mu(R_i)} = \frac{R_i}{T\mu(R_i)} \tag{6.4}$$

where: λ - total object arrival rate, i - layer index, R_i - reduction factor in i-th layer (number of input channels per Merger), $\mu(R)$ - objects merging rate with respect to reduction factor, T - publication interval of a single data source. If the Merger performance does not depend on the number of input channels, then the $\mu(R)$ component becomes constant. If ρ_i becomes greater than 1, which corresponds to Merger not being able to sustain input data throughput, then the input message queue grows infinitely and the following formulas cannot be applied.

The average number of messages in a queue, which is related to the average memory usage, can be calculated with Eq. (6.5).

$$\xi_i = \frac{\rho_i^2}{2(1-\rho_i)} \tag{6.5}$$

The approximate amount of objects contained by one Merger and its queue can be estimated with Eq. (6.6) for the entire objects Merger and with Eq. (6.7) the delta Merger.

$$I_i = \xi_i + \rho_i + R_i \tag{6.6}$$

$$I_i = \xi_i + \rho_i + 1 \tag{6.7}$$

The ρ_i component corresponds to the average processing time of an object after it finishes waiting in the queue. When merging deltas, one complete object is stored in memory, while the entire object Mergers require R_i objects cached. One can estimate amount of used memory by multiplying I_i with the fixed object size.

6.3.2. Modelling Merger topologies

Having the model of one Merger and its queues, one can proceed with understanding how the reduction factor may influence the total performance of a Merger topology. The following approximate cost functions with respect to R are proposed, assuming that only fixed-size objects are merged:

$$C_{Mp}(R) = c_p \sum_{i=1}^{L} M_i \rho_i \tag{6.8}$$

$$C_{Mm}(R) = c_m \cdot o_{size} \sum_{i=1}^{L} M_i I_i \tag{6.9}$$

where $M_0 = P$, C_{Mp} - cost of the processing power, C_{Mm} - cost of memory.

To find the optimal cost of a Merger setup C_M with respect to the reduction factor R, the minimum of Eq. (6.10) should be found.

$$C_M(R) = C_{Mp}(R) + C_{Mm}(R) \tag{6.10}$$

The proposed model can be employed in analogous processing topologies, where a process regularly receives messages on a number of input channels and produces some data as its output. Merging might be very well replaced with other kind of computation.

6.3.3. Example

The following example scenario is discussed. The process topology consists of 2500 data sources, each producing a 500 MB delta object every minute. The Merger performance can be approximated in the range [2, 2500] with the linear function $\mu(R_i) = -0.002R_i + 24$. The cost of one CPU core is 118 \$, while memory costs 6.25 \$/GB.

Fig. 6.3 illustrates the CPU, memory and total cost estimations with respect to the reduction factor. The discontinuities and the flat regions in the cost functions are the consequence of the number of layers and Mergers being integer values. A range of reduction factor values may correspond to the same topology arrangements.

With small R values, the memory usage becomes high, since the number of processes in the Merger topology is very large. It falls with larger reduction factors until R is greater than 1250. Then the message queues of Mergers become longer. The CPU cost grows for $R > 14$, because the Merger performance decreases proportionally to the number of input channels. The total cost minimum is located in the range $[500, 624]$, which corresponds to a topology with 2 layers, consisting of 5 and 1 Mergers.

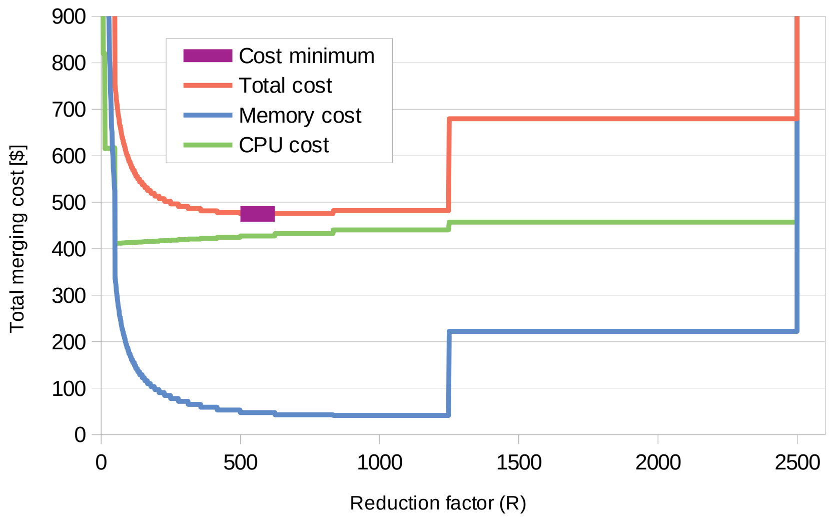


Fig. 6.3. An example of the cost functions Eq. (6.8), Eq. (6.9) and Eq. (6.10). The total cost minimum lies within the range $[500, 624]$.

6.4. QC Tasks localisation

Having the optimal process arrangement of Mergers and their total cost, one can compare the two ways of running QC Tasks - locally and remotely.

6.4.1. Modelling a QC Task

QC Task is modelled similarly to Merger. It contains a message queue, whose size (thus occupied memory) depends on the performance of a particular QC Task. In case of Mergers, one could estimate how much memory does the process requires by counting the amount of internally stored incomplete and complete objects. However, QC Tasks might also need RAM not only for storing messages or objects they publish, but also for their processing needs (e.g. information about detector geometry).

As assumed at the beginning of the chapter, process utilisation is proportional to the amount of received data. The average CPU utilisation of a QC Task can be defined with Eq. (6.11) if it runs locally and with Eq. (6.12) in the alternative case.

$$\rho_q = D \cdot Q_p \tag{6.11}$$

$$\rho_q = P \cdot D \cdot Q_p \tag{6.12}$$

where ρ_q should make less than 1, so the process can sustain the input data throughput. While for Mergers we assumed that processing time is constant for any incoming object, here we extend this assumption, as input data may differ in size. Then, the messages sizes are defined by the normal distribution with mean τ_d and variance σ_d^2. Hence, the M/G/1 model may represent this problem - messages arrive according to a Poisson process, processing time distribution is arbitrary (general) and there is one worker. The queue length can be described as follows:

$$\xi_q = \frac{{\rho_q}^2}{2(1-\rho_q)}\left(1+\frac{{\sigma_d}^2}{{\tau_d}^2}\right) \tag{6.13}$$

One should note, that an M/G/1 queue becomes an M/D/1 queue if the processing time is invariant, so the this QC Task model can be easily simplified.

The total memory used for message buffers and processed data can be estimated in the following way:

$$Q_{mI} = \tau_d \cdot (\xi_q + \rho_q) \tag{6.14}$$

With the presented equations, one can find the total cost of a QC Task with Eq. (6.15). The processing cost rises proportionally to the CPU utilisation, thus to the amount of data. The memory cost has two components, one corresponding to input buffers and processed data, the second accountable for all permanently allocated resources. One might consider omitting the Q_{mI} component if the CPU usage is small, so message queue does not occupy much memory.

$$C_Q = c_p \cdot \rho_q + c_m \cdot (Q_{mI} + Q_m) \tag{6.15}$$

6.4.2. Comparing the total cost of local and remote QC Tasks.

The total cost of a given setup is defined in Eq. (6.16). It consists of the infrastructure cost which runs locally (C_L), the cost of data transfer between the two clusters (C_T) and the cost of remote infrastructure (C_R). In both alternatives, one can omit costs of sampling data and running Checks, as it stays the same in both cases.

$$C = C_L + C_T + C_R \tag{6.16}$$

If QC Tasks run locally, the three components take the following forms. Only the local QC Tasks are executed on the main processing machines (Eq. (6.17)). The bandwidth costs results from all Monitor Objects sent by parallel QC Tasks (Eq. (6.18)). QC servers carry the cost of merging incomplete Monitor Objects (Eq. (6.19)), computed as in Eq. (6.10).

$$C_L = P \cdot C_Q \tag{6.17}$$

$$C_T = c_b \cdot P \cdot \frac{o_{size}}{T} \tag{6.18}$$

$$C_R = C_M \tag{6.19}$$

When QC Tasks are executed on remote QC machines, the same three components can be found in the following way. Locally, nothing is executed aside from Data Sampling, which cost is omitted

(Eq. (6.20)). Transfer cost includes all sampled data from the main computing nodes (Eq. (6.21)). Finally, one instance of the QC Task runs on the remote servers (Eq. (6.22))

$$C_L = 0 \tag{6.20}$$

$$C_T = c_b \cdot P \cdot D \tag{6.21}$$

$$C_R = C_Q \tag{6.22}$$

By calculating the total setup cost, the three kinds of computational resources are combined into one, more abstract resource. If the c_p, c_m, c_b factors use a currency as the unit and are derived from the hardware price, then this resource becomes money. However, one could choose any other approach to determine the availability of the processing power, memory and bandwidth. Still, when choosing if a QC Task should run locally or remotely, other, unmeasurable factors might appear. For example, a process which is prone to memory leaks could be executed on remote servers to avoid impeding the data acquisition, even if then the calculated setup cost is higher.

6.4.3. Exploring the model's parameter space

In this section, the model's parameter space is explored. Several scenarios are investigated, each having certain parameters changed within the expected range in the QC system. Tab. 6.3. contains the complete constant and variable parameter list. The changes in the CPU and RAM costs were not investigated, as their price differences might have influence on the parameters related to the performance (such as τ_d, Q_p, $\mu(R)$). For example, buying a more expensive processor might decrease the amount of CPU time needed by a QC Task. Therefore, since the model does not cover these correlations, investigating the influence of these parameters in isolation could lead to wrong conclusions.

The cost of bandwidth may differ e.g. depending on the technology used to connect computing sites and the distances between them. The ALICE QC nodes are placed in the same counting room as the FLPs. They both communicate with the EPN site via an InfiniBand network. Fig. 6.4 shows how the setup costs change with respect to the price of a bandwidth unit. In both cases, the transport cost rises proportionally to the amount of data transferred to the remote QC servers. Obviously, the influence is much less visible when less data is transferred, such as when the first step of data reduction is performed locally.

The number of main processing nodes has a proportional relationship with the C_L and C_T components of the local QC Tasks cost (Fig. 6.5), since it translates to running more instances of QC Tasks and publishing more Monitor Objects. The C_R component has less clear correlation with P, but it is not well visible in the presented example. The cost of remote QC Tasks also grows with the number of parallel nodes. At certain value, the workers cannot sustain the amount of input data, which is expressed with the cost rising to infinity. Then, running QC Tasks locally becomes the only feasible alternative. Increasing

Tab. 6.3. The complete set of parameters used for the model examples presented in Sec. 6.4.3. The '**x**' symbol denotes the variable parameters of a given scenario.

Parameter		Scenario								
Symbol	Unit	A	B	C	D	E	F	G	H	I
c_p	\$/CPU	118	118	118	118	118	118	118	118	118
c_b	\$/MB/s	**x**	0.09072	0.09072	0.09072	0.09072	0.09072	0.09072	0.09072	0.09072
c_m	\$/MB	0.00625	0.00625	0.00625	0.00625	0.00625	0.00625	0.00625	0.00625	0.00625
P	1	50	**x**	50	50	17	25	25	20	80
D	MB/s	15	2	**x**	10	23	15	5	5	1
τ_d	MB	2	2	2	**x**	1	1	1	1	1
σ_d	MB	0.5	0.5	0.5	1	0.1	0.1	0.1	0.1	0.1
o_{size}	MB	10	100	100	100	**x**	200	200	200	100
T	s	60	60	60	60	60	**x**	60	60	60
Q_p	CPU/MB/s	0.001	0.002	0.001	0.00199	0.0025	0.0015	**x**	0.005	0.005
Q_m	MB	250	500	500	500	300	500	500	**x**	500
$\mu(R)$	1	10	10	10	10	**512/x**	10	10	10	**x**

the amount of data needed from each processing node significantly raises the transfer costs for remote QC Tasks, as shown in Fig. 6.6. When running them locally, the larger input data throughput increases their CPU and RAM usage. The granularity of data messages has influence on the input message queue sizes of QC Tasks. In this context, they will grow if the process utilisation is close to 100% and if the messages size rises, as in Eq. (6.13) and Eq. (6.14). This effect can be observed in Fig. 6.7 for the remote setup cost, as the QC Task is highly occupied due to large input data throughput. Since the computations can be split among several local QC Tasks, their input queues do not contain as many waiting messages, so the local setup cost does not vary significantly. Also, if the proportion of the average message size to its standard deviation is larger, the additional burden of variable message sizes on the queue length becomes less significant. However, this effect is not visible in Fig. 6.7.

If the Mergers performance per input data throughput is kept steady while modifying the size of Monitor Objects produced by each instance of a QC Task, one may notice a large negative effect on local QC Task setups cost (Fig. 6.8). The cost increase is related to the higher bandwidth usage and heavier load on the Mergers. However, the cost can be decreased by choosing a longer QC Task cycle duration, thus publishing the Monitor Objects less frequently (Fig. 6.9). Still, the cost of keeping the QC Tasks running remains constant. The QC Task CPU usage per input data throughput has similar effect on the setup cost in both alternatives, as illustrated in Fig. 6.10. In both cases, the same total amount of data is processed by either a group of local QC Tasks or one remote actor. However, running only one, remote copy of the QC Task greatly limits the amount of data which the system can sustain. If it is exceeded, a local setup becomes the only alternative available. On the other hand, one could also consider executing several QC Tasks remotely and merging their results afterwards in order to alleviate this limitation. Fig. 6.11 illustrates how the total setup cost rises with the idle QC Task RAM usage increase. Naturally,

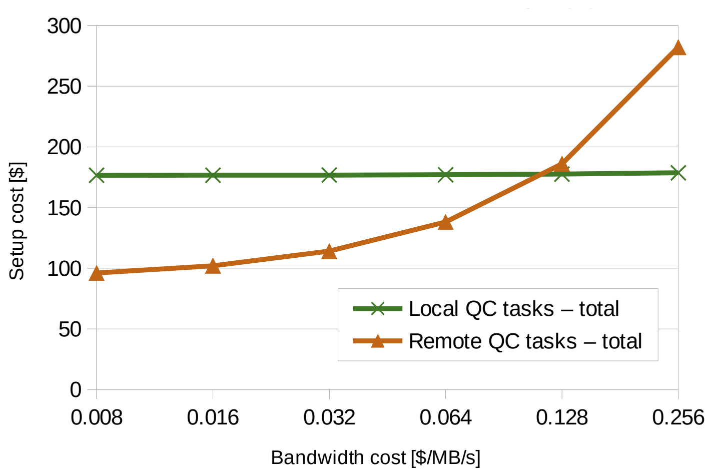


Fig. 6.4. The cost of local and remote QC setups as a function of the bandwidth cost (scenario A).

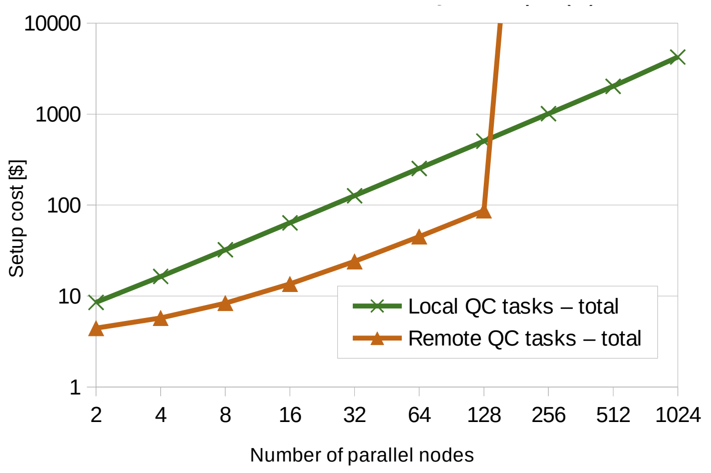


Fig. 6.5. The cost of local and remote QC setups as a function of the number of parallel nodes (scenario B).

the total RAM cost is also proportional to the amount of QC Task instances, thus the relationship is more visible for local, parallel setups.

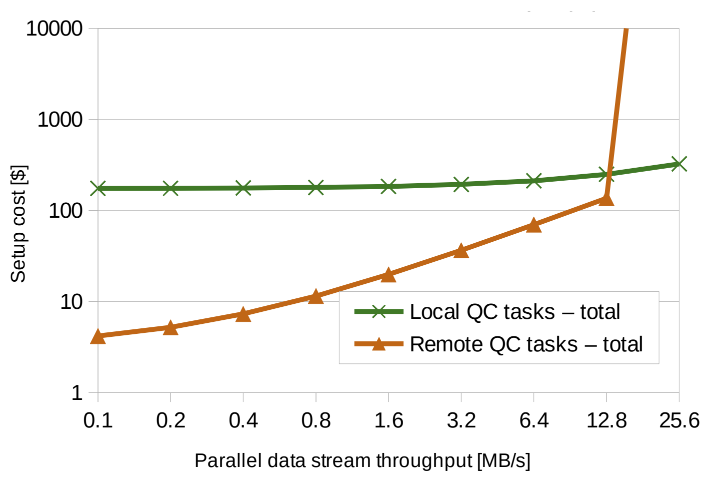


Fig. 6.6. The cost of local and remote QC setups as a function of the parallel input data throughput (scenario C).

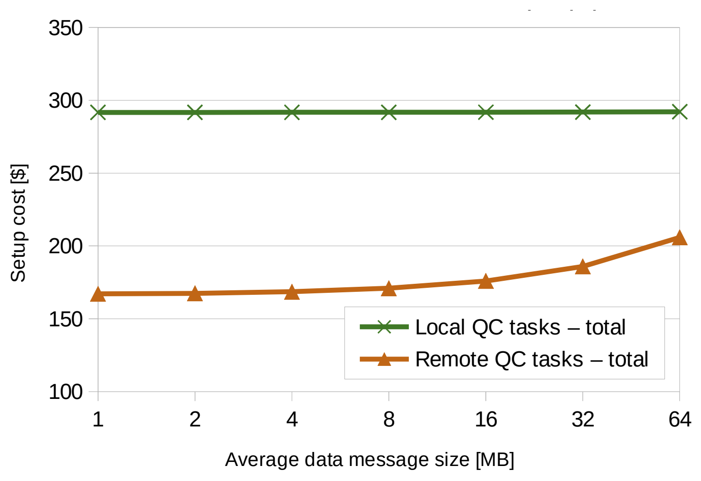


Fig. 6.7. The cost of local and remote QC setups as a function of the average data message size (scenario D).

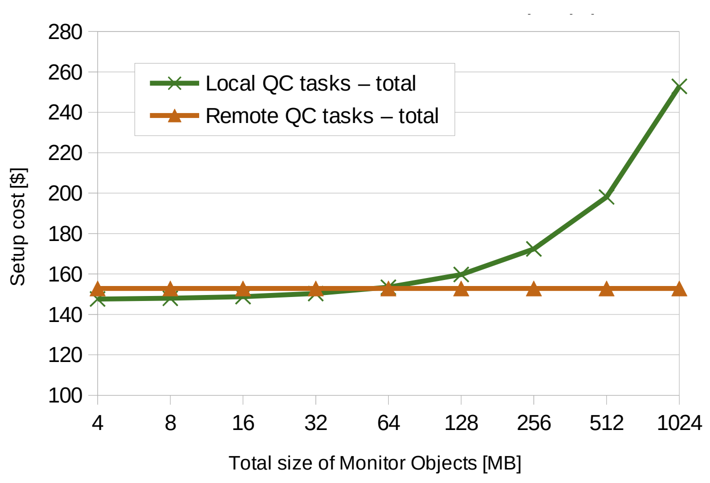


Fig. 6.8. The cost of local and remote QC setups as a function of the total size of Monitor Objects produced by one QC Task (scenario E). The Merger performance was adjusted accordingly to the relationship $\mu(R) \cdot o_{size} = 512\ MB/s$.

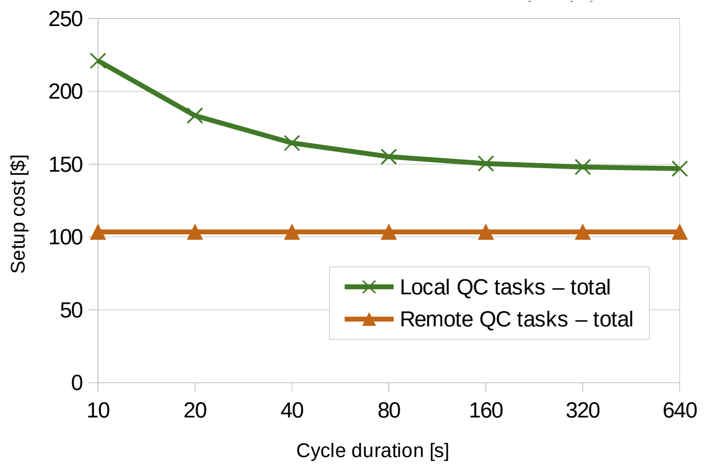


Fig. 6.9. The cost of local and remote QC setups as a function of the cycle duration (scenario F).

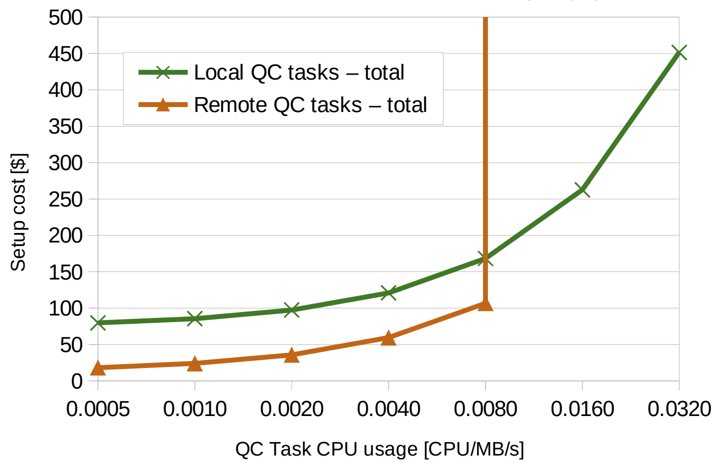


Fig. 6.10. The cost of local and remote QC setups as a function of the QC Task CPU usage per input data throughput (scenario G).

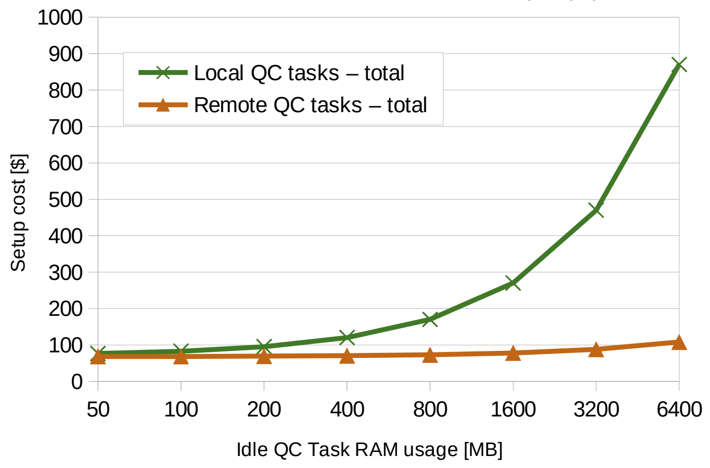


Fig. 6.11. The cost of local and remote QC setups as a function of the idle QC Task RAM usage (scenario H).

Finally, Fig. 6.12 depicts the modelled cost as a function of the Merger performance. It becomes significantly higher when merging more complex Monitor Objects. Otherwise, multiple instances of local QC Tasks dominate in the total cost.

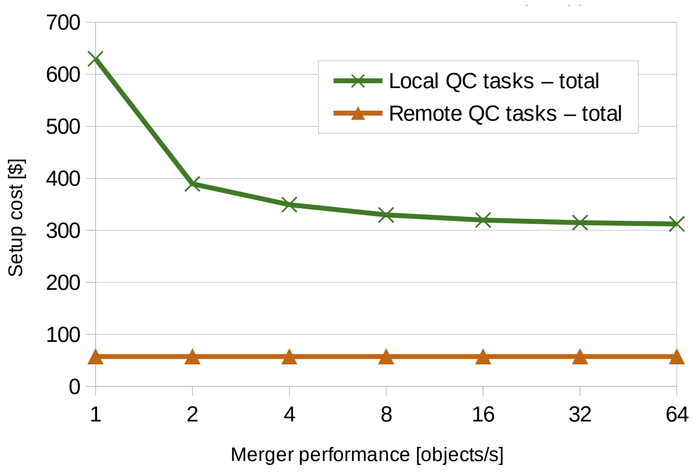


Fig. 6.12. The cost of local and remote QC setups as a function of the Merger performance (scenario I).

6.5. Summary

Supervising over 100 distinct QC Tasks requires a considerable amount of effort. Improving their performance and decreasing their impact on the computing system allows to carry out more comprehensive data quality control with the limited resources. By defining the mathematical models of the selected components of the QC system, it is possible to find better performing process topologies in a semi-automatic way. Besides, the exploration of their parameter space gave the author and other QC coordinators better understanding of the roles that different factors have in the system performance and cost. It also demonstrated how the parallel message-passing QC topologies can scale in large computing systems such as the O^2.

7. Benchmarking the Quality Control framework

This chapter contains benchmark results of each component of the Quality Control system. The clear responsibility division between the actors allows to test them in isolation by running mock-ups of the components they interact with. However, measuring performance of message-passing software is not always straightforward due to the stochastic nature of the communication between the actors and limited computational resources. There are many traps and intricacies which might lead to false or imprecise results. Descriptions of benchmarking methods explain potential problems which were alleviated, thus they might help designers and developers of similar systems to find the best way to test their software. Finally, the measurements can serve as a reference for other message-passing systems. All of the presented results were achieved on servers with Dual Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60 GHz and 128 GB RAM. Tab. 7.1 contains a list of the performed measurements.

Tab. 7.1. The list of performed benchmarks described in Chapter 7.

Component	Measured values	Variable
Data Sampling	message processing rate	number of data producers
Data Sampling	message processing rate, input data throughput	message payload size
Task Runners	message processing rate, input data throughput	number of data producers
Task Runners	message processing rate, input data throughput	message payload size
Task Runners	object publication rate, output data throughput	object size
Task Runners	object publication rate, output data throughput	number of objects
Check Runners	Check execution rate	number of objects
Check Runners	Check execution rate	number of Checks
Check Runners	Check execution rate	object size
Mergers	object processing rate	number of data producers
QC repository	input data throughput	object size

7.1. Data Sampling

The Data Sampling is the QC component which will have to cope with the highest amounts of data. Since it interacts with the main data processing flow, it may not slow down its performance nor cause any data loss. Therefore, it was essential to obtain a good understanding of the Dispatcher's performance in various working conditions. The Data Sampling benchmarks results were also published in [9].

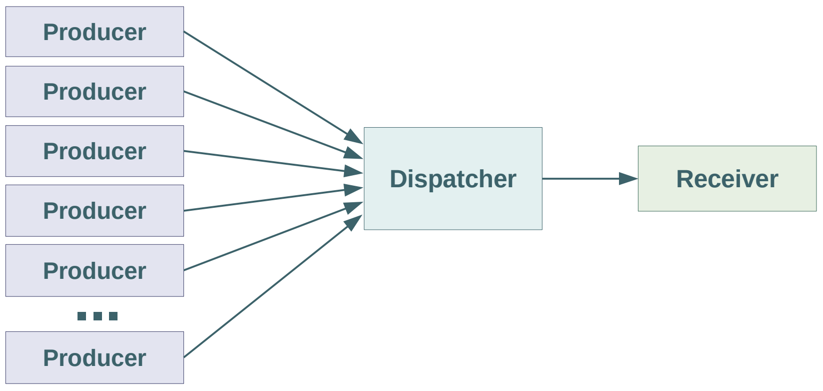


Fig. 7.1. The performance benchmark's process topology.

7.1.1. Benchmarking method

Fig. 7.1 depicts the topology of processes used to benchmark the Data Sampling. A group of data producers send messages at the highest possible rate. Their number is configurable and is treated as one of the test variables. The Dispatcher receives messages, evaluates them and passes forward if the selection criteria are met. All messages published by the Dispatcher are sent to the receiver, which serves only as an endpoint of the process topology.

The aim of the benchmark was to measure how fast Dispatcher can process incoming messages in different conditions. However, with big enough values of message rates and payload sizes, the cost of allocating the memory becomes the bottleneck, not the Dispatcher performance. Luckily, Linux has a memory overcommitment mechanism - when a process requests a certain amount of memory, it is actually allocated by the system when it is first accessed, either by a read or write operation. Thus, if data producers do not set the content of generated message payloads and the Dispatcher does not inspect them, then the costly allocation is avoided. Of course, when the Dispatcher copies messages, the memory is allocated both for input and output messages. While this effect does not cause any problems when measuring the performance with small payload sizes, the memory usage grows rapidly when bigger payloads are used, eventually hitting the available RAM limit. In such case the operating system starts to include memory swap files, which decreases the observed message rates or it kills the Dispatcher process with the Out-Of-Memory killer mechanism. When the shared memory is used, the Dispatcher might get stuck in a memory deadlock. If it tried to copy a message without enough space in the shared segment, it would wait until enough memory is freed, but that would only happen if more messages were processed by the same Dispatcher.

Because of these reasons, a dedicated benchmarking procedure was developed. First, the data producers send a number of messages which corresponds to a certain upper limit of acceptable memory usage (e.g. 80% of the shared memory segment). If the memory usage has not significantly increased after a brief time (e.g. to more than 20%), the data producers are allowed to produce messages as fast as they can, since there is little risk of saturating the memory. Otherwise, the data producers generate

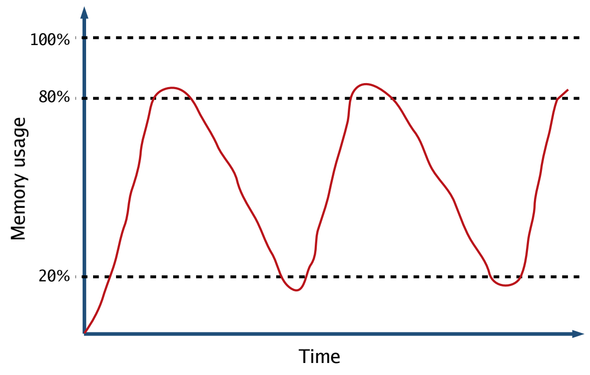


Fig. 7.2. An illustration of memory usage during the Data Sampling benchmark with the mechanism to avoid memory saturation.

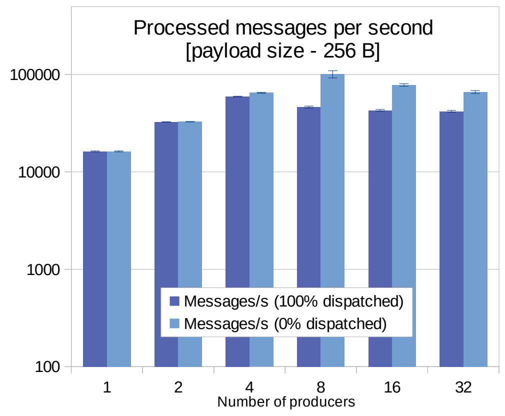
a limited amount of messages which is enough to reach the upper acceptable memory usage limit and then they wait until it drops enough to start sending messages again (see Fig. 7.2). It should be noted that this mechanism applies well only when memory usage is not too inertial with respect to the number of produced messages, which is the case for shared memory, but not for the normally allocated memory. In the latter case, the inertia was decreased by writing data into messages payload contents.

Each configuration was ran 5 times for 5 minutes to observe variations of the results, which could occur due to e.g. changing affiliation between processes and CPU cores. Moreover, as the performance of Dispatcher is largely influenced by the desired fraction of messages passed forward, the benchmark was executed for two extreme values (100% and 0%) to develop a better understanding of the performance range.

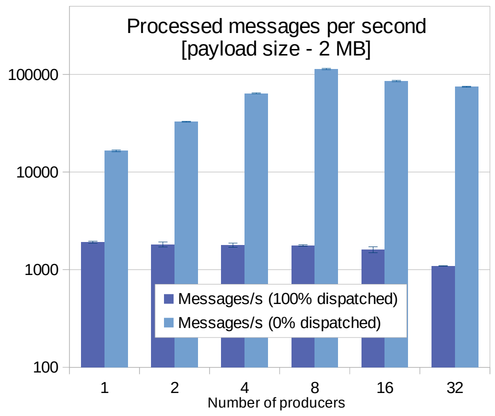
7.1.2. Benchmark results overview

The benchmark results in Fig. 7.3 indicate how the number of input channels served influences the performance of a single Dispatcher. On both figures one can see that if the Dispatcher does not dispatch data, it can sustain up to 4 data producers sending messages at the maximum rate possible. For very small payload sizes, 4 producers are needed to reach the highest message dispatching rate of around 59000 per second. When Dispatcher should reject all data, it can receive around 100000 messages per second, regardless of the data size. As expected, copying 2 MB messages requires more time and the performance becomes limited by how fast the system can copy memory. The message processing rate slightly decreases for more than 8 producers, thus one could consider assigning one Dispatcher per 8 sources at most to improve the performance.

As DPL workflows running on the same machine use a common shared memory segment, they should pass only pointers to the data. Therefore, receiving and rejecting messages should not be determined by their size. The benchmark results illustrated in Fig. 7.4 confirm these expectations with the result of around 114000 messages per second when none are passed further. On the other hand, if data are always



(a) The benchmarks results for the 256 B payload size [9].



(b) The benchmarks results for the 2 MB payload size [9].

Fig. 7.3. Dispatcher's performance with respect to the number of data producers.

copied, then the message rate decreases for larger payload sizes. Still, the total data throughput rises, since it is more efficient to copy data organised in larger chunks. The data throughput over 2 GB/s is seen for the payload range of [256 kB, 1 GB], with the peak of around 3480 MB/s for 256 kB messages. When copying all data, the highest message rate is around 58000 messages per second, obtained for the smallest possible payloads.

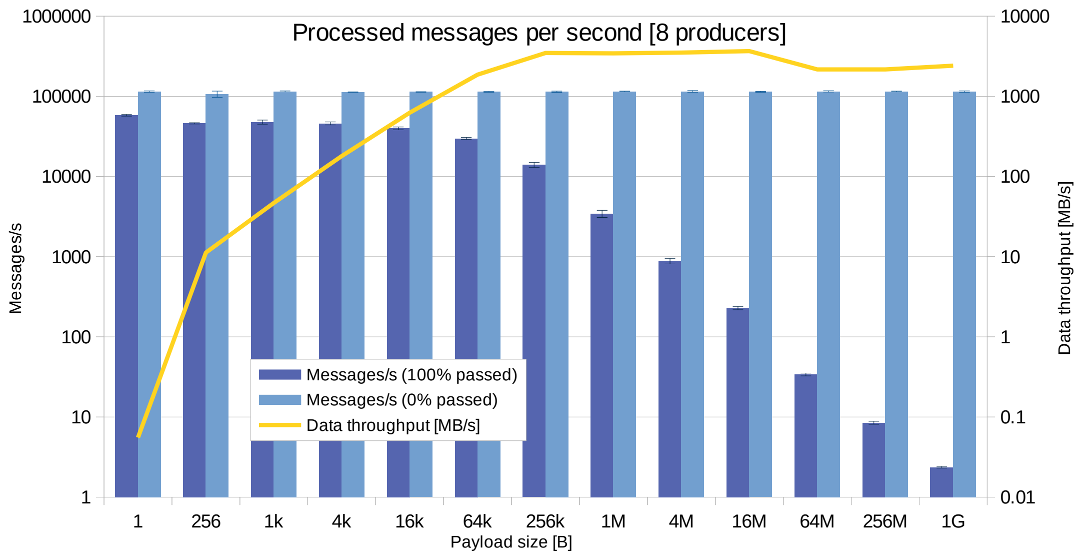


Fig. 7.4. Dispatcher's performance depending on the message payload size [9].

7.2. Task Runners

Thanks to the shared memory communication, the framework can avoid performing expensive copies of data when it is possible. Thus, the input data throughput which one QC Task can process is determined by two factors. The first one is the QC Task performance, which is dependent on the particular algorithm needed to assess the quality of data. The performance of message-passing or, in other words, the number of messages per second that can be provided to a QC Task by the QC framework makes the second factor. In this section, the framework performance benchmarks results are presented and discussed.

The benchmark uses a similar process topology as the previously described Data Sampling benchmark (Fig. 7.1). The test subject, Task Runner, receives data from one or more data producers. They generate empty messages with a rate which, in this case, does not exceed a specified throughput limit. Monitor Objects generated by the QC Task are sent to a Check Runner and discarded afterwards. The QC Task performs the minimal amount of processing possible. It involves generating non-empty standard ROOT histograms and accessing the message payloads. Check Runner executes one Check, which indicates always good quality given the set of received objects. 5 measurements were taken in each configuration, then an average and a standard deviation were calculated. Each test run lasted 5 minutes.

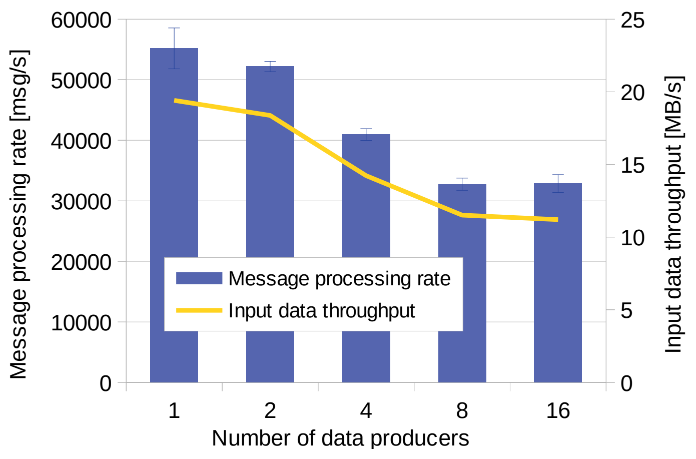


Fig. 7.5. The total message processing rate of a QC Task with respect to the number of data producers used. The message payload size was set to 256 B.

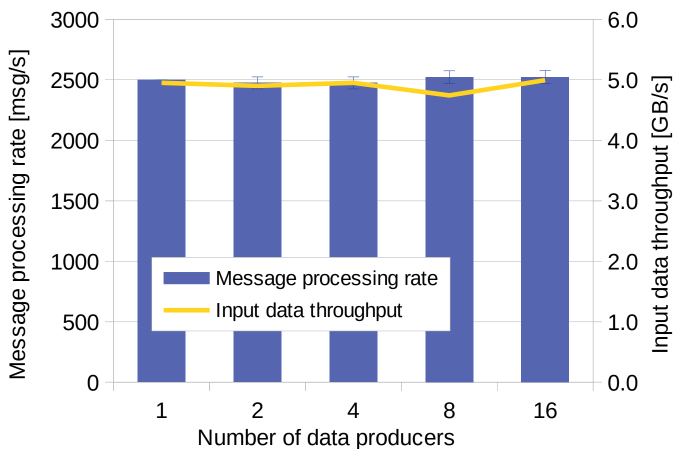


Fig. 7.6. The total message processing rate of a QC Task with respect to the number of data producers used. The message payload size was set to 2 MB.

Fig. 7.5 illustrates how the number of data producers influences the message processing rate of QC Tasks, assuming 256 B message payload size. In such case, QC Tasks sustain around 55000 messages per second when receiving data from one producer, which is the most popular case expected. It corresponds to around 18 µs needed by the QC framework and the simplest possible QC Task to handle one message. The message rate drops if more data sources are used. When 2 MB payload size is used (Fig. 7.6), the described setup can sustain the input data throughput of 5 GB/s for any number of producers in the measured range.

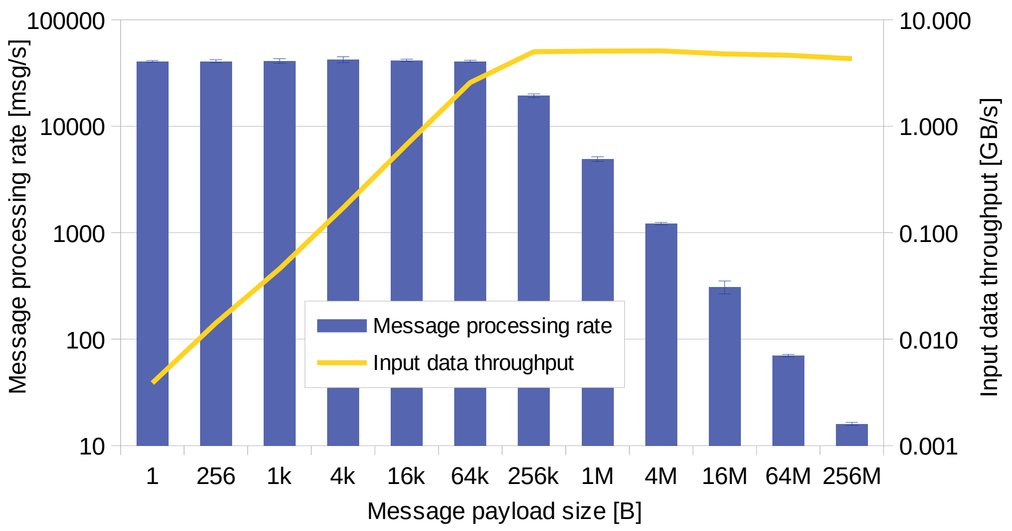


Fig. 7.7. The message processing rate of a QC Task with respect to the message payload size. The messages were generated by 4 data producers in parallel.

The relationship between the processing rate and the granularity of messages is presented in Fig. 7.7. Similarly to the Data Sampling, the data processing rate is limited by the message passing speed if small payloads are used. The maximum input data throughput was set to 5 GB/s and it can be met with payload sizes in the range of [256 kB, 4 MB] in the described setup. It slightly decreases to around 4.3 GB/s for 256 MB payload sizes. In general, the benchmark results indicate that the framework overhead is very small compared to the expected input message rates of QC Tasks (a few hundred at most).

The other activity common to all QC Tasks is publication of Monitor Objects, which is also handled by the framework. In another benchmark (Fig. 7.8), the QC Task's cycle duration was set to 1 second and the object publication rate was measured with different Monitor Objects sizes. The framework was able to sustain the output data throughput up to around 400 MB/s, which was achieved with the QC Task publishing collections of 100 objects, each requiring 4 MB. Reaching higher values was limited by the ROOT message serialisation mechanism, which would crash when processing too big object collections. Fig. 7.9 illustrates the influence of the number of objects on the publication rate. The framework consistently allowed to publish between 1 and 1024 objects of size 250 kB every second. The highest achieved output data throughput in this configuration was around 260 MB/s. Again, obtaining greater values was not possible due to mentioned ROOT serialisation problems.

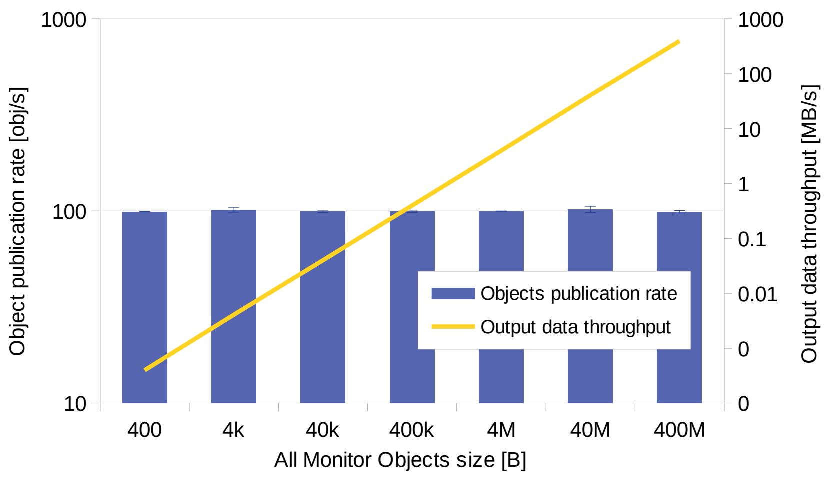


Fig. 7.8. The object publication rate of a QC Task with respect to the total size of all Monitor Objects. The QC Task was configured to generate 100 standard ROOT histograms with 4 B bin counters (`TH1I`) every second. The number of bins was modified to reach different total Monitor Object sizes.

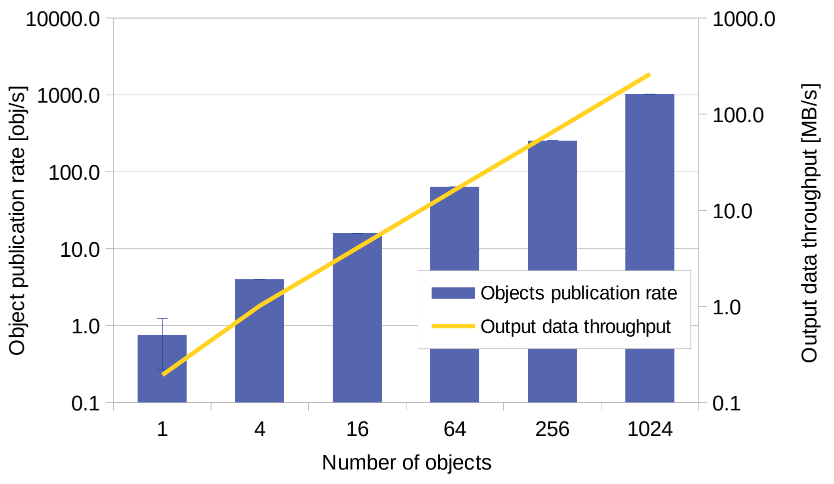


Fig. 7.9. The object publication rate of a QC Task with respect to the number of created objects. The QC Task was configured to generate standard ROOT histograms (`TH1I`) of size 250 kB every second.

One should note that the presented results were achieved with standard ROOT histograms. Thus, the performance might vary when using other data structures, although it is not expected to differ significantly. Finally, since an average QC Task may produce around 200 objects of size 250 kB per minute according to the survey, it is concluded that the cost publishing objects is relatively small.

7.3. Check Runners

The Check Runners should support various associations between Monitor Objects and Quality Objects. One Check may expect multiple Monitor Objects and likewise, one Monitor Object can be requested by many Checks. In this section, the performance of Check Runners is evaluated, including extreme ratios of Checks and Monitor Objects. In all benchmarks, a Check Runner executed a number of Checks on objects sent by one QC Task in one second intervals. Checks always returned the same quality flag regardless of the data. Similarly to the Task Runner benchmarks, there were 5 measurements taken in each configuration and each test ran for 5 minutes. The figures contain averaged results with standard deviations as error bars.

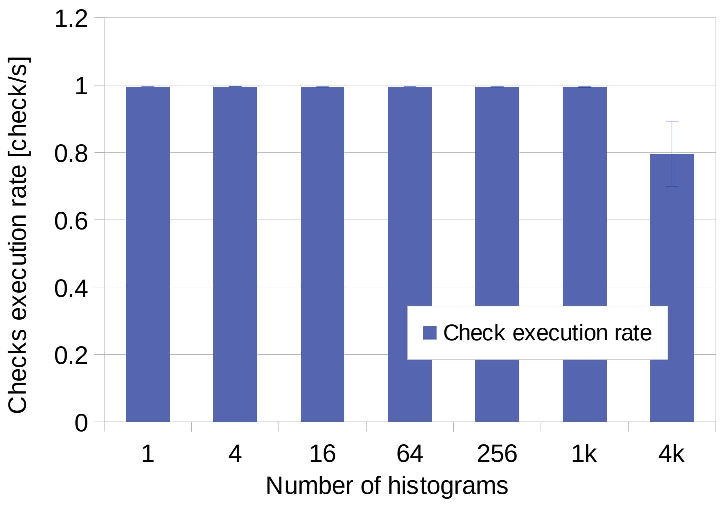


Fig. 7.10. The Check execution rate with respect to the number of objects requested by one Check. Standard ROOT histograms (TH1I) of size 250 kB were published every second by a QC Task.

In the first scenario, the Check Runner contained one Check, which expected a configurable number of objects. Fig. 7.10 contains the benchmark results obtained for objects of size 250 kB. The Check execution rate is steady across the range of [0, 1024] and falls for the larger number of objects. The highest input data throughput of around 815 MB/s was achieved for 4096 objects needed by one Check.

In another scenario, a constant number of objects needed by one Check was assumed, while the number of Checks on the same set of objects was modified. Fig. 7.11 illustrates the benchmark results for 100 objects per Check. In the measured range of 1 to 256 Checks, the Check Runner could sustain the anticipated Check execution rate.

In the last benchmark, the size of each input object was modified, keeping the constant association of 100 Checks requiring the same set of 100 objects. The framework was able to sustain the expected Check execution rate in the full measured range [400 B, 400 MB] of the total size of Monitor Objects.

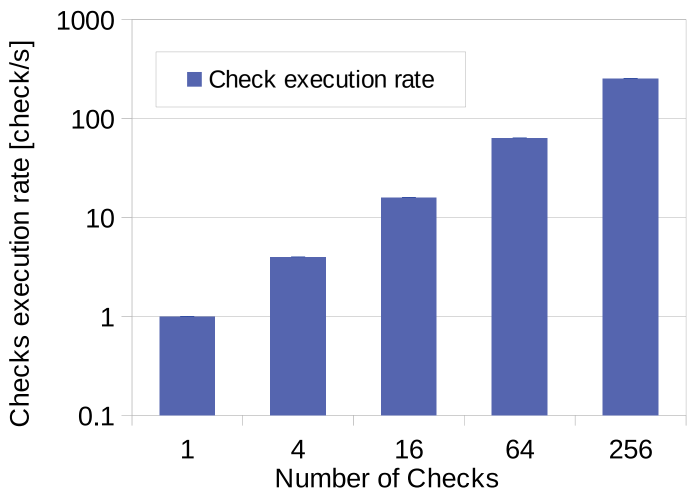


Fig. 7.11. The Check execution rate with respect to the number of Checks in one Check Runner. Each Check evaluated 100 objects (250 kB each) published by a QC Task every second.

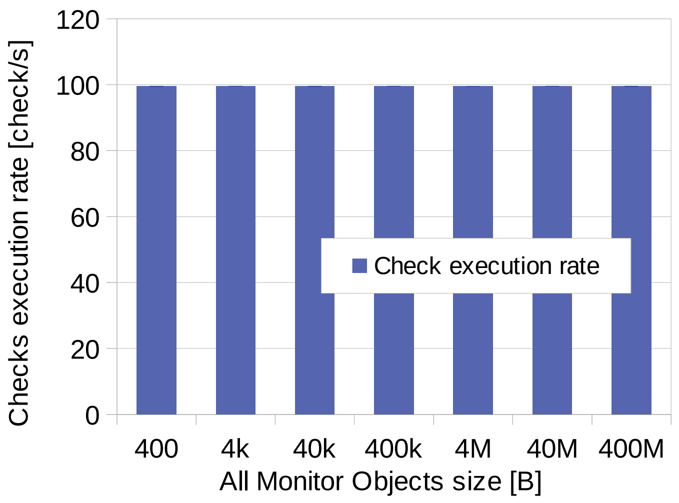


Fig. 7.12. The Check execution rate with respect to the total size of Monitor Objects needed by one Check. Check Runner was provided with 100 objects every second. There were 100 Checks declared, each requested the same 100 objects.

According to the performed surveys, the QC system will generate around 1700 Quality Objects every second. The presented benchmark results indicate that Check Runners can sustain high input data throughputs and are able to execute large numbers of Checks per second. With the minimal framework overhead and by running many instances of Check Runners in parallel, the system will be able to cope with the expected data rates.

7.4. Mergers

The performance of Mergers was measured in a laboratory setup with three servers connected with the Dell Force10 S4810 switch via 10Gbps Ethernet ports. Two nodes hosted up to 250 data sources each. Achieving higher numbers of processes on one server was limited by the DPL framework. Each produced one-dimensional histograms (`TH1I`) of size 250 kB with a configurable time interval. The third node was used to receive generated data and run one Merger process. Its performance was measured by increasing the incomplete object production rate in steps of 100 and evaluating if the Merger is able to sustain the input data throughput (i.e. it reports the same object rates as data sources and its memory does not rise drastically). The highest stable object rate was treated as the correct result. Thus, the performance measurement error of 100 objects per second is assumed. Additionally, to detect potential bottlenecks in the data sources and the communication, the object production rate and the network bandwidth were monitored.

When benchmarking the Merger implemented for entire objects, the publication rate of the complete object was set equal to the object generation rate of each data source. This way, the merging was performed when all input objects got updated. The delta Merger processed each object upon receiving it and published the complete result every 10 seconds.

Fig. 7.13 illustrates the benchmark results with respect to the amount of data sources. Both modes of merging are characterised by similar performance, which varies between 2600 and 2900 objects per second within the measured range and slightly decreases for larger numbers of producers. In the worst case, one CPU core was enough to sustain a data stream of 650 MB/s.

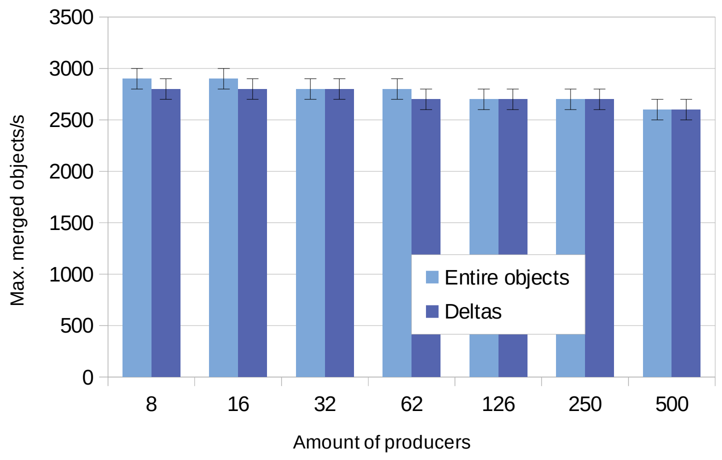


Fig. 7.13. Performance of merging 250 kB 1D histograms (TH1I) in the three-node benchmark.

7.5. QC repository

The feasibility of using the CCDB as the QC repository back-end was evaluated in 2018 by Barthélémy von Haller. According to the surveys performed with the sub-detector teams, the QC system will generate around 25000 Monitor Objects per minute. The objects will have an average size of 250 kB. This translates to a data stream of around 100 MB/s, or 400 objects per second.

The setup consisted of 6 servers connected via a 10 Gb/s switch. One server was used to host a CCDB instance. It had 56 Hyper-Threading CPU cores, 256 GB of RAM and a 1.8 TB solid-state drive (SSD). The other 5 nodes hosted 20 QC Tasks in total, each sending 20 objects per second. The object size was modified across time. According to the benchmark results (Fig. 7.14), one instance of the CCDB could sustain a stream of 400 objects of size 1 MB per second, which is 4 times more than the expected rate.

Given the benchmark results, the QC Objects storage facility should withstand the anticipated data rates. In case that higher performance is required, many small objects can be concatenated into singular messages, thus reducing the amount of requests received by the CCDB. Another possibility includes deploying several CCDB instances which would be assigned to specific ALICE sub-detectors. Thus, the total object stream could be split among several nodes without a significant impact on the user experience.

7.6. Summary

According to the survey performed in 2017, the QC system should allow to execute more than 100 distinct QC Tasks, most of them running in parallel on multiple computing nodes. In total, they would generate 25000 complete objects every minute, requiring 250 kB of memory on average. Around 10000 of them would have to be merged beforehand. Moreover, the objects would undergo extensive automatic evaluation, resulting in around 100000 quality flags produced each minute.

The QC framework was extensively benchmarked to confirm that it will be able to sustain the unprecedented data throughputs and amounts of generated objects. By testing each kind of actor separately, it was possible to measure how different factors influence the QC framework performance. Thanks to the Data Processing Layer, the FairMQ library and the underlying software, which implement the message-passing within the actor model, the high input data throughput can be split among multiple Task Runners and Check Runners executed in parallel. This allows the system to scale well across hundreds or thousands of computing nodes and thus, the software will fulfil the requirements for the ALICE data quality control during the next data-taking periods.

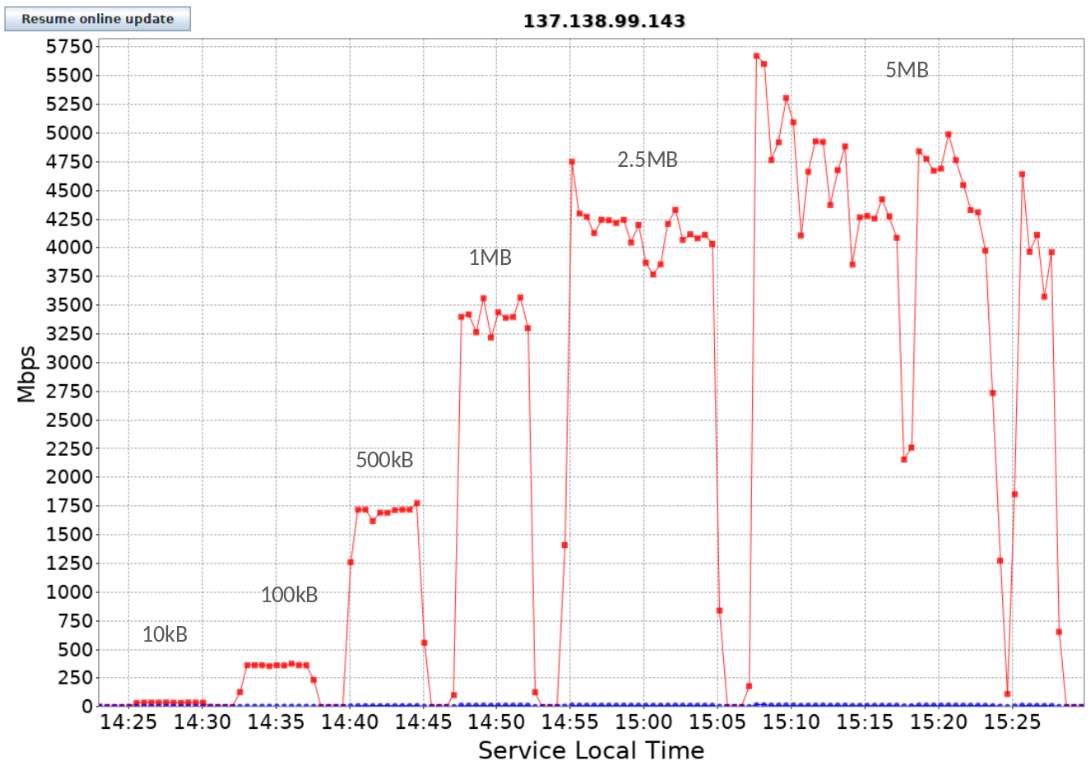


Fig. 7.14. The input data throughput of a CCDB instance with different object sizes being sent. The results indicate that the CCDB could sustain a stream of 400 objects per second, 1 MB each, which corresponds to an input data throughput of 3200 Mb/s. When 2.5 MB objects were sent, the database reported lower input data rate than expected (around 4250 Mb/s instead of 8000 Mb/s), thus it could not sustain the throughput. The benchmark results were provided by Barthélémy von Haller.

8. Applications of the Quality Control framework

This chapter contains a summary of the recent developments of the QC modules. Typical usage examples and the most up-to-date statistics are presented. Additionally, other studies which use the QC framework are outlined.

In 2021, the ALICE detector is being reassembled and commissioned together with the new computing system. The O^2 will be tested with simulated or random data and by measuring the detector noise, particles coming from space and eventually collisions obtained from low-intensity test beams in the LHC. Firstly, the sub-detectors will be commissioned in isolation, but at later stage, they will run at the same time. This is also the time when a lot of effort will be put to the development of QC modules, so they can be used during the commissioning and from the first days of data taking.

8.1. Current status of the QC modules development

As of January 2021, the Quality Control software repository [115] contains 11 detector-specific modules, which are developed by the detector expert teams. In total, the modules include 24 QC Tasks and 25 Checks. The Trending Task was adopted twice so far. There are also custom 4 Post-processing Tasks, 3 of them being extended versions of the Trending Task. The detector module developers also implemented 4 Reductors to adopt them in their Post-Processing Tasks. At this stage, a lot of QC Tasks do not use the Data Sampling software, since the detector experts require all data to commission their apparatus. However, there is an increasing trend to randomly select acquired data when it is applicable and other, more advanced usages are also expected in the final system. The Data Sampling was also incorporated outside of the QC system in the commissioning setup of the MCH detector. It was used to randomly downscale the amount of data sent to the MCH decoding workflow.

The QC includes also a set of special modules. The Common module contains a set of Reductors and Checks which can be applied in any setup. Beginner users can base their code on the Skeleton module. It contains minimal examples of different QC components, which lets the users to copy and paste the code to their own modules and develop it further. Additionally, more advanced usage examples are available in the Example module. Finally, the DAQ modules contains a dedicated Task and Checks used to evaluate the aspects of raw data which are independent of the sub-detector.

While the detector commissioning is usually performed with setups consisting of one server, at least 3 detector teams have already managed to configure and run multi-node setups. For example, the ITS commissioning setup involved eight servers receiving data from different parts of the detector and each running 2 QC Tasks locally. The ninth node hosted the Mergers, Checkers and two Post-Processing Tasks.

The detector teams are encouraged to list all their planned QC Tasks in an online sheet. As of January 2021, the table includes 108 QC tasks. 50 of these are proposed to be run on the FLPs, another 50 on the EPNs, while the rest should be executed on the QC servers. It is very likely that more QC Tasks will be executed on the remote servers, especially those which will not require much input data. In total, the QC Tasks will generate around 52230 objects with various update rates. On average, 29165 objects will be published each minute.

8.2. Data-agnostic quality control

Data quality algorithms are usually written for concrete detectors, as they use different physics phenomena to detect particles and they have distinct structures. Also, data received from one detector take several forms during the event reconstruction. If possible, each step should undergo some kind of quality monitoring. However, during early development or commissioning of a detector one could benefit from a general quality control task which would calculate some features of binary data blobs. For example, it could compute randomness indicators of received message payloads, find patterns inside data, calculate the distribution of byte values and find correlations between two consecutive payloads. Moreover, the data contained in headers, such as payload size and timestamps, could be aggregated in suitable data structures. This information could allow to perform basic Checks, which would assess some basic features of the data, e.g. non-emptiness and variability across time.

Such a QC module should provide data quality control information complementary to the tailor-made algorithms and facilitate debugging problems both by deduction and induction [119]. Additionally, it may measure some properties of the data acquisition and processing software, thus helping to investigate potential issues in the system during data-taking. The project proposed by the author is currently under development by Mateusz Knapik and Kacper Janda.

8.3. Machine Learning

During the recent years, Machine Learning (ML) is becoming increasingly popular in the HEP community [120]. Experimental applications of ML cover many aspects of data acquisition and analysis. Among others, ML is applied in: collision simulations to reduce the computational requirements, triggering data acquisition and performing real-time data analysis, event reconstruction, searches for new physics, monitoring of detectors, detecting hardware anomalies and scheduling pre-emptive maintenance. The last three domains are especially important in the context of data quality control. Thus, in parallel to the development of conventional methods to evaluate the quality of data, the application of

ML is also strongly considered. In 3rd-4th December 2018, the "Machine Learning and Quality Control in ALICE" workshop was held at CERN. The presentations and discussion covered the planned applications for the QC in Run 3, available data sets and existing usages of ML in other experiments.

One of the efforts includes the application of semi-supervised anomaly detection to automatically assess the quality of data [121]. The researcher focused on finding abnormalities in the set of 200 parameters related to working conditions of the TPC detector and features of the reconstructed particle tracks. An autoencoder network with 5 hidden layers was trained with data from 5 acquisition periods in 2018, divided in 15 minutes chunks. The model could recreate almost the same data quality classification as with the conventional, manual methods. A follow-up study was described in [122]. The set of 200 input parameters was reduced to 97, which were corresponding to physical attributes of the TPC detector. An autoencoder with 2 hidden layers was applied. For lead-lead collisions, the proposed method found 1.9% outliers in the data, while 2.7% were identified with standard approaches. The author suggests that the excess of outliers found with the conventional methods could have been tagged falsely, which could mean that the machine learning methods may actually improve the separation of anomalies.

The machine learning methods are also evaluated with the statistics collected by the data-agnostic module, described in the previous section. The developers aim to automatically discover unusual or unexpected patterns in the message headers and payloads.

Exercising the machine learning methods on the Run 1 and 2 data shows that the ALICE experiment may benefit from replacing manual checks with classification performed by trained models. However, the upgraded detector will produce different data and the derived input parameters might not be compatible with previously gathered sets. It means that it is unlikely that ML will be used to classify data quality since the first days of operations during Run 3. However, the models will be trained on the newly acquired data as soon as possible.

9. Summary and Conclusions

High energy physics experiments, such these at the LHC, have always pushed the latest technology to its limits in order to conduct high-quality research. Increasing the amounts of registered data lets the physicists perform previously impossible discoveries and more precise measurements. Now, the ALICE collaboration is preparing its detector to generate a raw data stream of 3.5 TB/s, which will be handled by the new computing system with new software. As it will perform the event reconstruction on the fly and discard the raw data, a reliable and advanced data quality control will play a crucial role in quickly identifying potential problems during data-taking and providing necessary feedback for final quality assurance. Preparing a software framework which will fulfil these requirements was the topic of this dissertation.

The main achievements of this dissertation are:

– a comprehensive review of the existing data quality control frameworks in the High Energy Physics community,

– an implementation of the first data quality control framework that is fully based on the message passing approach and the actor model,

– development of new data sampling methods which were not available in similar systems,

– a mathematical model of the most resource-heavy components of the framework.

The new data Quality Control framework is a highly parallel system, which can split computations among hundreds or thousands of nodes. The author took the leading role in its design and development of its most crucial parts: actors which run user-defined QC Tasks, Checks and Post-processing Tasks, as well as the two backbones of the framework, namely Data Sampling and Mergers. The choice of Data Processing Layer, the FairMQ library and other underlying technologies allowed the author to implement the QC framework as a message-passing system which applies the actor model and the zero-copy approach. Consequently, different responsibilities could be divided into independently running processes. They can share data without unnecessary copies if they are located in the same server, which significantly reduces the cost of communication between them. If suitable, the actors can be replicated to sustain higher input data throughputs and their partial results can be merged afterwards. Thanks to the possibility of using multi-layer Merger topologies, the maximum number of supported parallel workers is not limited to the performance of one process, since they can be arranged in a hierarchical structure. Moreover, the Mergers support both entire and delta objects, which brings greater flexibility to the detector modules developers when choosing the most appropriate data structures to store their quality control results. The author also proposed and implemented a mechanism to juxtapose pseudo-randomly selected data corresponding to the same events, yet split among many parallel computing nodes. The amount of data transferred to quality control algorithms may not only be reduced by pseudo-random sampling, but also by performing custom message selection, which was not possible in similar systems. Combining several sampling methods together gives interesting, previously unavailable possibilities for advanced data quality control during the acquisition. Due to the fine-grained distribution of responsibilities among the processing actors, a temporary failure of one component does not immediately stop the full system from acquiring and processing data. The processes can be separately started, stopped and reconfigured during the run-time, so the experiment shift crew can e.g. enable certain demanding QC Tasks only on request. The Data Processing Layer greatly facilitates extending the existing process topologies with new components. According to the reviewed literature, there are no other data quality control frameworks in HEP which rely fully on the message-passing approach and the actor model.

Managing more than 100 QC Tasks requires a lot of effort and understanding of how different factors affect their performance. To facilitate that, a mathematical model of QC Tasks and Mergers was proposed in the dissertation. It provides methods to comprehend the influence of various aspects on the usage of hardware resources and it allows to find the optimal configuration with the smallest resources usage. The benchmark results of the QC framework can serve as a reference to the developers of similar systems. The discussed process topologies and their models are not specific only to data quality control, but may also find their applications in other domains.

The dissertation also contains a thorough summary of large data quality control systems in physics experiments. The general concept behind each of the described pieces of software is similar - all of them analyse some portion of acquired data and optionally perform automatic checks. However, they use different naming conventions which makes the knowledge exchange within the data quality control community more difficult. With the aim of improving it, these distinct terms were assembled in one table, which can serve as a dictionary between the collaborations.

The QC framework is being adopted by the detector teams. As of January 2021, around 25% of the planned QC Tasks are already implemented and their number is steadily growing. The software was already used to test and commission several ALICE sub-detectors. This included larger setups with QC Tasks running on multiple servers in parallel, Mergers combining their results, Checks evaluating them and Post-processing Tasks trending selected observables.

9.1. Discussion

Implementing the Quality Control as a message-passing system brought up some great advantages - easy parallelisation, clear division of responsibilities and extensibility. However, the presence of 100,000s independently running processes in the O^2 [30] puts a great burden on tools which are supposed to control and configure them, as well as gather resource usage statistics and log messages. They need to scale proportionally to the data processing software and also allow the experiment control crew to use them without being overwhelmed with the system complexity. Also, designing, implementing and managing over 100 unique QC Tasks and the associated Checks requires a significant amount of manpower. Any of these components may stop functioning or cause memory leaks due to errors in their code. While this is unavoidable to some extent due to the complexity of the detector itself, the problem remains. The increasing interest in the machine learning methods may partially mitigate this problem, e.g. by replacing some QC Tasks and Checks with common, data-agnostic modules. Finally, the notion to unify the ALICE software running synchronously and asynchronously to data-taking allowed for greater code reuse. However, it also increased the total build time of its components due to large number of direct and indirect software dependencies. The Quality Control has almost 50 dependencies in total, while the AMORE (the ALICE Run 1&2 DQM software) had only 6.

The QC relies heavily on the ROOT framework, a powerful tool for statistical data analysis. Its data structures, such as histograms and columnar data storage types, can be used to perform most of the data quality control tasks. However, the QC framework requires that any Monitor Object inherits the ROOT's common interface. Moreover, the Quality Control GUI may visualise only objects supported by JSROOT. In effect, using data types available in other libraries is made difficult and sometimes impossible. However, the future work plans include removing this constraint, so one may use data types outside of the ROOT's ecosystem.

9.2. Future work

During the year 2021 and early 2022, the ALICE detector and its new data acquisition and processing system will be deployed and commissioned. The remaining QC Tasks and Checks will be implemented by the detector teams, and then tested with the detector setup. According to the latest LHC schedule [123], the proton-proton collisions will start in early 2022. The first heavy-ion run is planned in December 2022. Then, the ALICE O^2 system will have to withstand the long-awaited data rates.

In the meanwhile, the QC framework will be further improved, extended and exercised in bigger data acquisition setups. Also, following the ALICE software unification efforts, the QC will be further integrated with the data simulation software and the data analysis framework. One of the extensions will include an alarm system, which will inform the concerned experts about recording bad quality data via selected communication channels, such as the Mattermost platform, the e-mail system or the GSM network.

The author will put major research efforts into development of automatic procedures to let the physicists analyse only the parts of data acquisition runs which consist of good quality data. This has never been done in similar systems – data quality flags used to be assigned on run basis or even larger periods. The process of tagging good quality data will require additional post-processing steps to combine quality flags issued both automatically and manually, and then derive a global quality for a given data set. Also, the author will investigate how to organise collecting data quality feedback from detector experts in a systematic and efficient manner.

In parallel, the author will continue researching the possible arrangements of actors in the QC framework to allow for more flexibility and to further reduce the usage of available computing resources. One of such topologies could consist of parallel QC tasks and Mergers both running on servers dedicated for QC.

Finally, following the investigations about the machine learning methods usage, selected observables will be gathered by another group of researchers as soon as data acquisition starts. They will be used to train ML models, which are expected to enhance the traditional methods of controlling the data quality.

List of Figures

List of Tables

A. Glossary

The dissertation contains a large number of abbreviations and terms which might not be familiar to every reader. Below, the list of such abbreviations and terms in the alphabetical order is presented:

– ALICE - A Large Ion Collider Experiment, one of the experiments at LHC

– AliECS - ALICE Experiment Control System

– API - Application Programming Interface

– CCDB - Condition and Calibration DataBase, the database which will store information about changing conditions of the detector and derived calibration values in the O^2

– CERN - fr. *Conseil Européen pour la Recherche Nucléaire*, now *Organisation Européenne pour la Recherche Nucléaire*, The European Organization for Nuclear Research

– CPU - Central Processing Unit

– CRU - Common Read-Out, the FPGA card used to receive raw data from detectors and transfer them into the memory of FLPs

– DAQ - Data AcQuisition System

– DCS - Detector Control System

– DPL - Data Processing Layer, the software framework of the O^2 system

– DPL workflow, DPL topology - a group of independently running actors which can process and exchange messages, implemented using the Data Processing Layer

– DQM - Data Quality Monitoring

– EPN - Event Processing Node - a server in the second computing farm of the O^2 system, responsible for aggregating data from all sub-detectors and their reconstruction

– FIFO - First In First Out queue

– FLP - First Level Processor, a server in the first computing farm of the O^2 system, responsible for data aggregation from a certain geometric part of the detector and detector-dependent processing

- FPGA - Field Programmable Gate Array
- GPU - Graphical Processing Unit
- GUI - Graphical User Interface
- HEP - High Energy Physics
- LHC - the Large Hadron Collider, the biggest particle accelerator located at CERN
- ML - Machine Learning
- MO - Monitor Object, any object in the ALICE QC system which contains some kind of statistics used for data quality control
- O^2 - The Online-Offline computing system, the new processing system of the ALICE experiment for Run 3
- QC - Quality Control, the data quality control software for the ALICE experiment in Run 3
- QC object - an alternative name for Monitor Object
- QCG - Quality Control GUI, the web interface for inspection of objects stored in the QCDB
- QO - Quality Object - an object which stores results of data quality checks
- ROOT - an object-oriented framework for processing and analysis of large data volumes, developed at CERN
- SCTP - Stream Control Transmission Protocol
- TCP - Transmission Control Protocol
- UDP - User Datagram Protocol
- WLCG - Worldwide LHC Computing Grid

B. Benchmarks of mergeable data types

The user algorithms in the Quality Control system prevalently use the ROOT data types. Due to this fact, performance of these data structures has a big influence on the total needs for computing resources. Thus, it is crucial that the maintainers of the software and infrastructure have good understanding of their general performance and how it changes with respect to related parameters. In this appendix, a thorough analysis of popular data types in the QC is held. Also, recently released `boost` histograms are compared with standard histogram types in the ROOT framework.

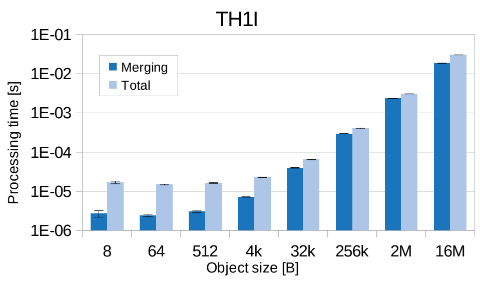
All the presented performance measurements have been obtained on servers with a dual Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60 GHz and 128 GB of RAM. The software was compiled with GCC v7.3.0 and the optimisation settings set to `-O2`. Each data point was measured 10 times. The first result was always discarded, as the processing time was usually longer than in later trials, probably due to ROOT dynamically loading or initializing certain components. An average and standard deviation were obtained from the remaining 9 results.

5 parameters were measured or estimated, if measurements were not possible: object size in RAM, object size after serialisation, deserialisation time, merging time, serialisation time. Size of sparse histograms in memory was particularly difficult to estimate, as there is no method in their interface, which provides the size of a singular bin. Instead it was estimated accordingly to the information in the documentation.

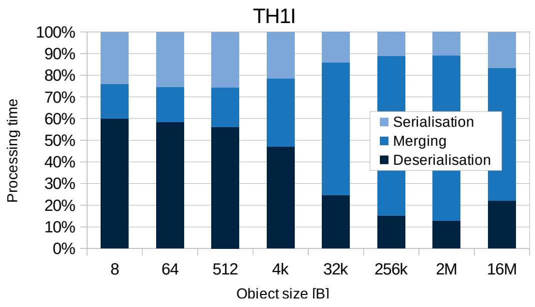
B.1. Standard histogram types

Standard ROOT histograms (`TH1`, `TH2`, `TH3`, `THn`) always allocate memory for each possible bin. Therefore, the object size should scale proportionally to the amount of bins and the bin counter size. Merging mostly involves accessing the bins and adding the number to each other, which also is dependent on amount of bins.

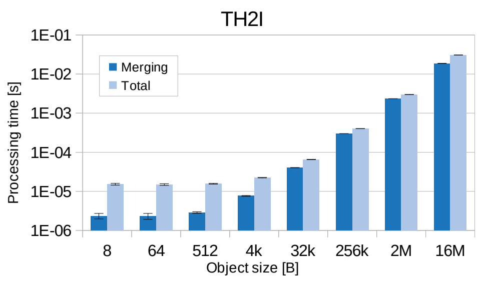
Fig. B.1 shows the benchmark results of handling singular 1, 2 and 3 dimensional histograms. Each object was filled with 50000 entries. Serialisation and deserialisation was performed with the `TMessage` facility. The presented measurements are aligned with the predictions about proportionality of the performance. However, one can also notice an existence of some processing time which is independent of the number of bins, best visible for the small object size. After reaching certain threshold, around 4 kB, this overhead becomes less apparent.



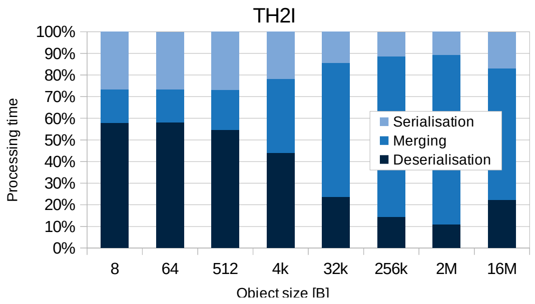
(a) The performance of `TH1I` with respect to its size.



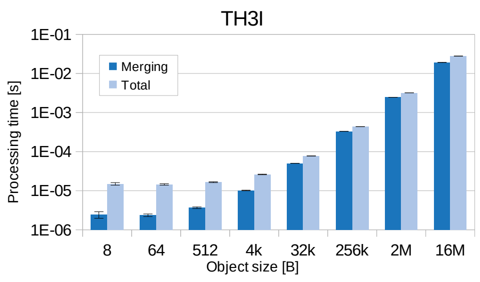
(b) Proportions of the three processing stages of `TH1I` with respect to its size.



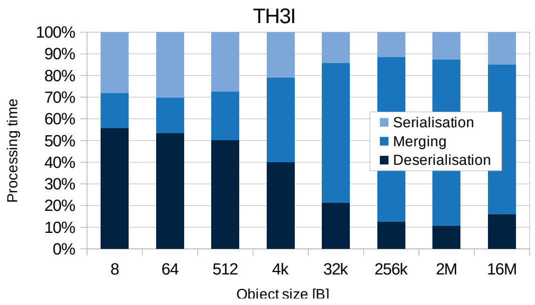
(c) The performance of `TH2I` with respect to its size.



(d) Proportions of the three processing stages of `TH2I` with respect to its size.



(e) The performance of `TH3I` with respect to its size.



(f) Proportions of the three processing stages of `TH3I` with respect to its size.

Fig. B.1. Performance of 1, 2 and 3 dimensional standard ROOT histograms with respect to their size.

Similarly, serialisation and deserialisation time is especially visible in the smaller objects, then it grows with the increase of the number of bins. Thus, if when having a choice between multiple, small histograms and less, but bigger ones, in order to cover the contain the same information, one should prefer the latter option, as it adds less total overhead.

Performance of 1, 2 and 3 dimensional histograms is very coherent across the object size spectrum. It shows that dimensionality does not play a significant role here, while the number of bins does.

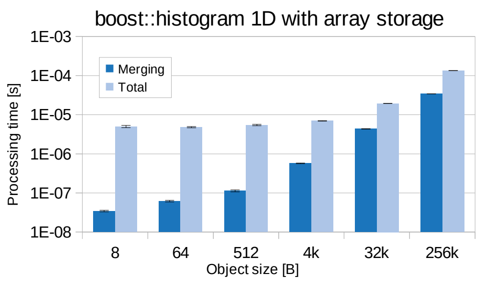
Assuming an average QC object size of 256 kB, which needs around 400 µs to be processed, one-threaded process cannot cope with more than 2500 objects per second if they are sent one by one (no benefits from common (de)serialisation). Also, as the processing time scales proportionally to the object size (excluding small values), on can easily estimate total amount of computing resources to merge all standard histograms in the system by calculating the total object size and reading the processing time from extrapolated benchmark results.

In 2019, the `boost` library [43] was extended with a histogram package. The authors claim that it was designed with high performance in mind and it is one of the fastest libraries on the market. As the QC system heavily relies on histogram types, an evaluation was held in order to find potential performance improvements.

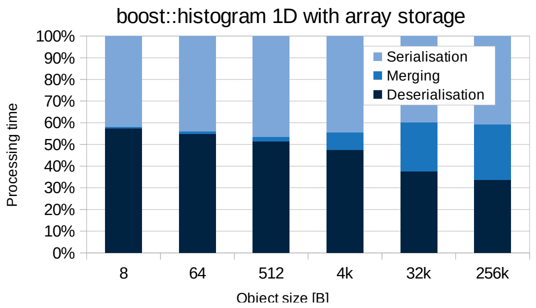
The `boost::histogram` library allows to select the underlying storage for bin counters. By default, it is the unlimited storage - it optimizes memory usage by using the shortest bin counters necessary to contain the histogram values without overflowing. As soon as one of the counters is due to reach the maximum possible value, the whole storage container is reallocated with a data type supporting larger value range. Also custom containers are supported, with many STL types working out of the box. One can use `std::array` to allocate a histogram on stack, which is supposed to improve processing performance in some cases, but limits histogram size to smaller values, which can be contained inside a stack. `std::vector` is also supported - then data is allocated on the heap.

Fig. B.2 compares serialisation, deserialisation and merging performance of singular instances of 1 and 2 dimensional `boost` histograms as in the version v.1.72. Standard arrays and vectors were used as the underlying storage. Each object was filled with 50000 entries. (De)serialisation was performed with `binary_iarchive` and `binary_oarchive` - they are faster than `text_archives`, but do not compress data, as opposed to the former. After each test, data were accessed to avoid any excessive optimisation by the compiler. Measurements of the array-based histograms includes smaller size range, as greater object sizes would cause stack overflows.

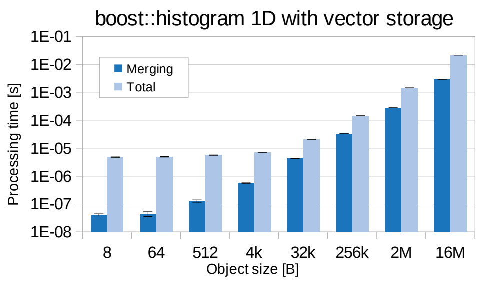
The histograms based on arrays and vectors show a very similar performance, which undermines the expectations to see an advantage of using stack-based storage. The performance difference was visible with code optimisation turned off, however, it would disappear when the benchmark was compiled with the `-O2` flag. Similarly to the ROOT histograms, processing time grows proportionally to the object size after certain threshold, which is better visible in case of heap-based types, as the benchmarks can cover larger objects. In the case of small objects, merging time is miniscule compared to the serialisation and deserialisation costs.



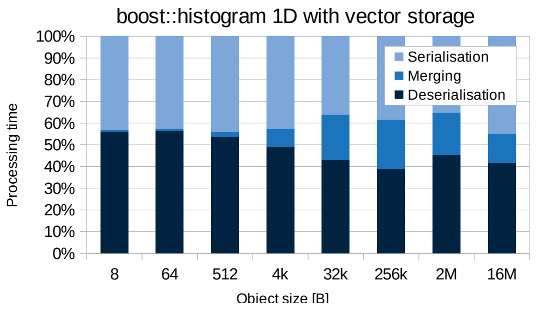
(a) The performance of 1D array-based `boost` histogram with respect to its size.



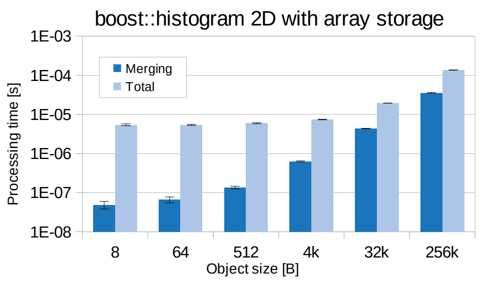
(b) Proportions of three processing stages of 1D array-based `boost` histogram with respect to its size.



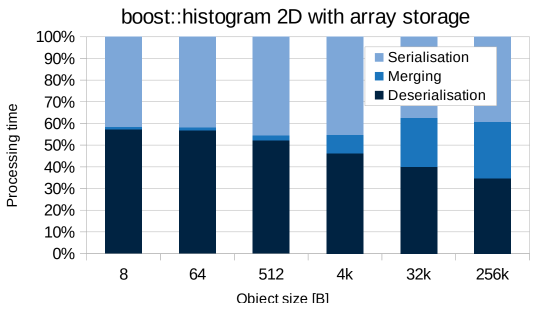
(c) The performance of 1D vector-based `boost` histogram with respect to its size.



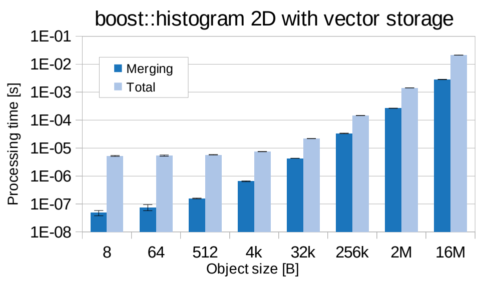
(d) Proportions of three processing stages of 1D vector-based `boost` histogram with respect to its size.



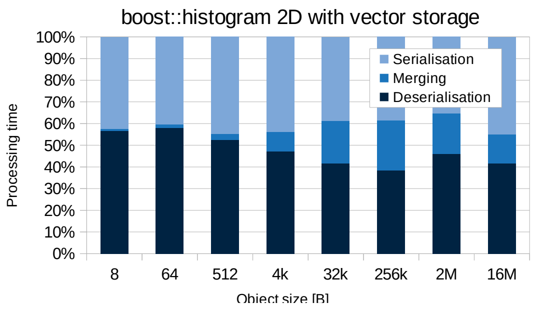
(e) The performance of 2D array-based `boost` histogram with respect to its size.



(f) Proportions of three processing stages of 2D array-based `boost` histogram with respect to its size.

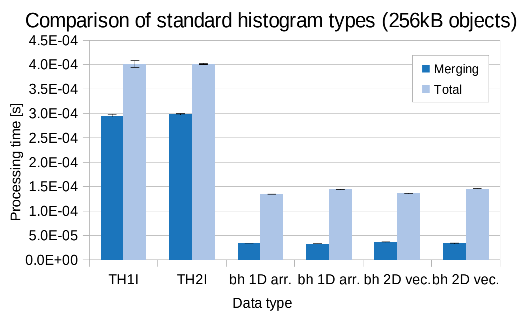


(g) The performance of 2D vector-based `boost` histogram with respect to its size.

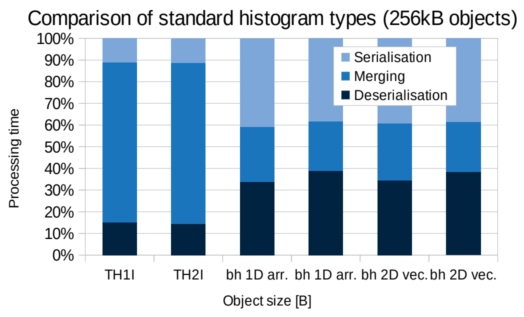


(h) Proportions of three processing stages of 2D vector-based `boost` histogram with respect to its size.

Fig. B.2. Performance of 1 and 2 dimensional `boost` histograms with `std::array` and `std::vector` storage, as a function of their size.



(a) Performance of the benchmarked types.



(b) Proportions of three processing stages of the benchmarked types.

Fig. B.3. Comparison of the 1 and 2 dimensional ROOT and `boost` histograms of size 256kB, based on 32 bit integer bin counters.

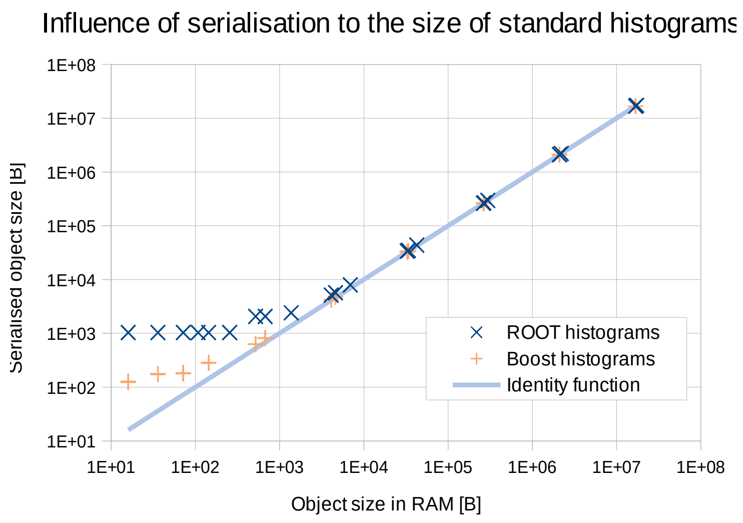


Fig. B.4. Comparison of the standard `boost` and ROOT histograms' size before and after serialisation.

Fig. B.3 contains a performance comparison between the ROOT and `boost` histograms. `Boost` histograms have a significant performance advantage in merging compared to ROOT histogramming types - they are almost an order of magnitude faster (almost 300 µs opposed to around 35 µs). While, serialisation and deserialisation take similar time for all types, total processing time is still much lower in case of `boost` histograms. Fig. B.4 shows how serialisation influences ROOT and `boost` histogram sizes, covering the same configuration points as the aforementioned results concerning performance. While for higher numbers of bins the proportion between packed and unpacked objects is close to 1, there is an overhead present in small serialised histograms, especially visible for the ROOT types.

The measurements show a clear supremacy of `boost` over ROOT histograms. However, there are disadvantages of `boost` types which concern the ease of use. While the library provides a clear factory interface to create new histograms, the classes themselves are highly templated. It is safe to say, that the real type declarations are actually long and complicated, as indicated in the List. B.1. The authors assume a heavy usage of type deduction methods in the modern C++, which might require some workarounds when deserialising a message - the object type has to be known upfront then. ROOT, while it is also known for its steep learning curve, it allows for type recognition of serialised objects, thus adds a possibility to perform cross-checks on the received object's types.

```
#include <boost/histogram.hpp>
namespace bh = boost::histogram;
...
const double min = 0.0;
const double max = 1000000.0;
const size_t bins = 62500;
const std::string axisName = "x";

auto histogram1 = bh::make_histogram(bh::axis::regular<>(bins, min, max, axisName));
auto histogram2 = bh::make_histogram(bh::axis::regular<>(bins, min, max, axisName));

// filling the histograms
histogram1(42);
histogram2(4400);

// merging two histograms into another one
auto histogram3 = histogram1 + histogram2;

// the underlying histogram type is:
// boost::histogram::histogram<std::tuple<boost::histogram::axis::regular<double, boost::use_default, boost::use_default,
      boost::use_default> >, boost::histogram::unlimited_storage<> >
```

Listing B.1. An exemplary usage of `boost` histograms.

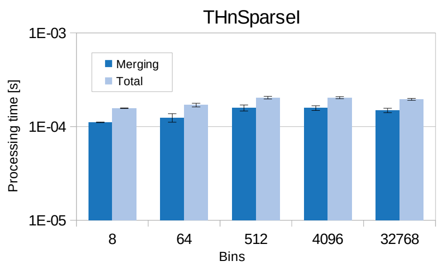
The described version of the QC system supports only ROOT types. However, the plan of future work includes allowing any types in the framework, so also `boost` histograms may find their use with their promising performance.

B.2. Sparse histograms

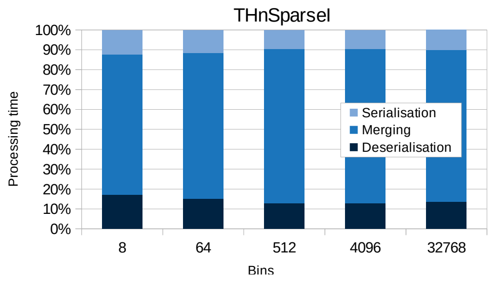
The `THnSparse` type in the ROOT framework is designed to allow for multi-dimensional histograms, while consuming less memory than standard histograms. It may indeed come with great benefits if the histogram entries are distributed sparsely (thus the name), so most of the bin counters are equal to zero. `THnSparse` decreases the RAM consumption by allocating only bin counters which are actually used, as opposed to `THn`, which books the memory for all bins during its initialisation.

Sparse histograms have more than one factor which might influence the efficiency of merging, with four suspected the most - amount of dimensions, number of bins per dimension, number of entries and their distribution. Fig. B.5 shows how different parameters influence the performance of the discussed data type. The histograms were filled with random entries, which were highly unlike to fall into the same bin twice if the total number of bins was much larger than the number of entries. Therefore, one can make an assumption that these two parameters are almost equal with exception to the last two plots (Fig. B.5g and B.5h).

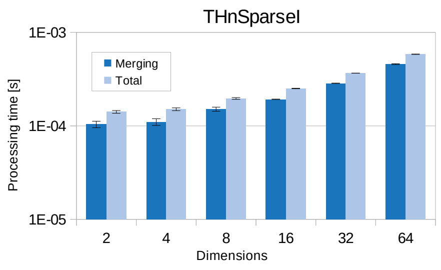
Fig. B.5a and B.5b show that the number of bins has little influence on the performance, including the serialisation times. As `THnSparse` stores only non-zero bins, having to use longer numbers as bin



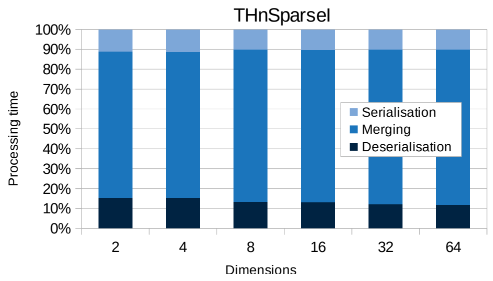
(a) Performance of `THnSparseI` with respect to the number of bins per dimension, assuming 8 dimensions and 512 entries.



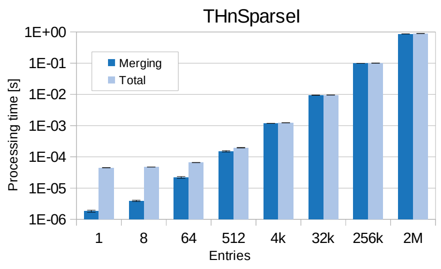
(b) Proportions of three processing stages of `THnSparseI` with respect to the number of bins per dimension, assuming 8 dimensions and 512 entries.



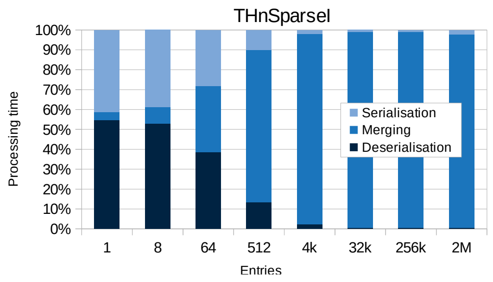
(c) Performance of `THnSparseI` with respect to the number of dimensions, assuming 512 bins per dimension and 512 entries.



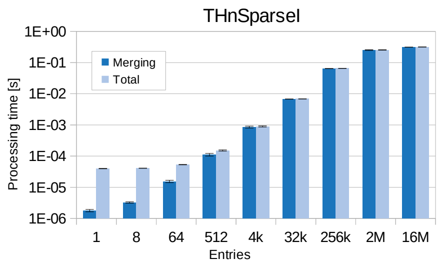
(d) Proportions of three processing stages of `THnSparseI` with respect to the number of bins per dimension, assuming 512 bins per dimension and 512 entries.



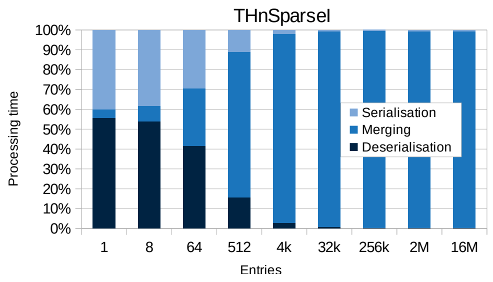
(e) Performance of `THnSparseI` with respect to the number of entries, assuming 512 bins per each of the 8 dimensions.



(f) Proportions of three processing stages of `THnSparseI` with respect to the number of entries, assuming 512 bins per each of the 8 dimensions.



(g) Performance of `THnSparseI` with respect to the number of entries, assuming 32 bins per each of the 4 dimensions. With higher number of entries, they start to occupy the same bins.



(h) Proportions of three processing stages of `THnSparseI` with respect to the number of entries, assuming 32 bins per each of the 4 dimensions.

Fig. B.5. Performance of `THnSparseI` with respect to different parameters.

coordinates does not imply much larger processing time. The number of dimensions has slightly higher influence on the performance in the usually used range, as shown in Fig. B.5c. However, the change from 2 to 64 dimensions does not cover the full order of magnitude. The proportions of different stages of processing are constant (Fig. B.5d). The number of uniquely allocated bins has the biggest impact on the merging time, making it grow proportionally (Fig. B.5e). Converting objects has a significant overhead when the amount of entries is small. For higher amounts, it grows, but becomes less apparent compared to the merging time.

Fig. B.5g illustrates an example where the number of entries becomes larger than the number of bins, so new entries are added into existing counters. The flattening seen for the higher amount of entries confirms that it is the number of uniquely allocated bins which has direct influence on the performance. Thus, the total size of sparse histograms and their processing efficiency highly depends on the specific patterns in collected data.

The interface of `THnSparse` does not provide a reliable way of estimating its size in memory, thus no measurements of the influence of serialisation are presented.

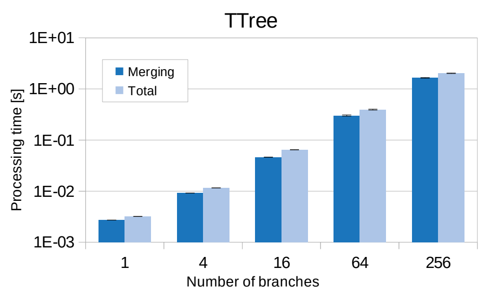
B.3. Columnar data storage

The ROOT framework includes a columnar data storage facility, called `TTree`. It facilitates its visualisation and statistical analysis, and optimizes random access. One tree can contain many branches (columns), which may use distinct storage types - booleans, characters, integers, floating-point numbers, as well as structures and arrays of those. Horizontally, `TTree` stores entries (rows), which contain values for each declared branch.

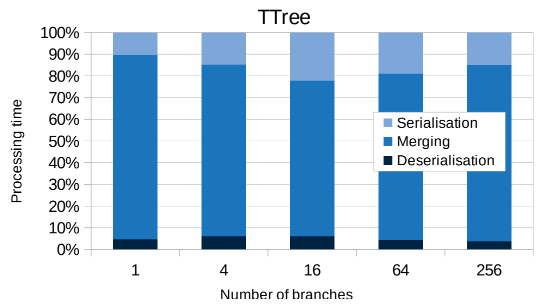
The performance of `TTrees` was evaluated with respect to three parameters - number of branches, size of branches and number of entries. Each branch was an array of 8 B integers with adjustable size. The content of each entry was generated randomly.

The processing time scales proportionally to the number of branches within the range [1, 256], as shown in Fig. B.6a. The serialisation and deserialisation time does not follow a clear pattern in this domain, however it is not negligible (Fig. B.6b). Processing time grows exponentially when the branch size is increased, with the serialisation time becoming particularly long (Fig. B.6c and B.6d). The number of stored entries has a proportional relationship with the processing time above a certain threshold (Fig. B.6e). The proportions of the three processing stages change with respect to the amount of entries and the overhead is particularly significant when `TTree` does not contain much data (Fig. B.6f).

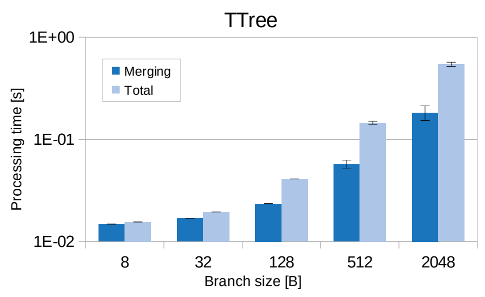
Fig. B.7 shows how serialisation impacts `TTree` size. The serialised object size was computed by iterating on stored branches and accumulating values returned by the method `TBranch::GetTotalSize()`. It has to be noted that obtained values were sometimes smaller than expected, therefore this approach is either invalid or the described class performs some kind of compression of data in memory. The plot indicates that `TTree` grows after serialisation, sometimes doubling in size. However, the serialized object take similar amount of space as it was estimated by multiplying the



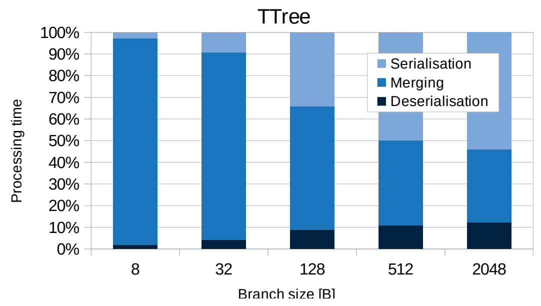
(a) Performance of `TTree` with respect to the number of branches.



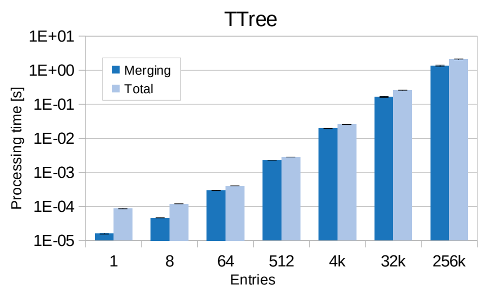
(b) Proportions of three processing stages of `TTree` with respect to the number of branches.



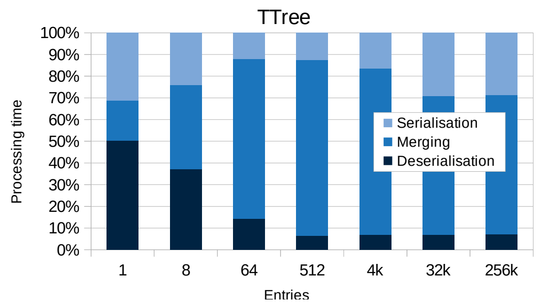
(c) Performance of `TTree` with respect to the branch size.



(d) Proportions of three processing stages of `TTree` with respect to the branch size.



(e) Performance of `TTree` with respect to the number of entries.



(f) Proportions of three processing stages of `TTree` with respect to the number of entries

Fig. B.6. Performance of `TTree` (ROOT's columnar data storage) with respect its number of branches (columns), branch size (column width) and number of entries (rows).

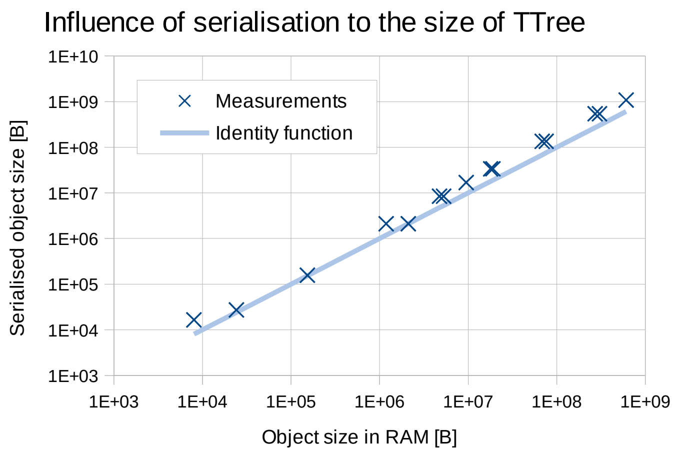


Fig. B.7. `TTree` size before and after serialisation.

number of entries, branches and branch size with each other. For this reason, one should approach these results with caution.

B.4. Relationship between total processing time and object size

Estimations which concern merging objects can be simplified if processing time grows proportionally to object size. However, before assuming it, one confirm validity of this statement or at least find a range which is applicable. Fig. B.8 shows how the object size of the 5 evaluated ROOT types influences their total processing time. The size of `THnSparseI` was estimated according to the available documentation. The measurements indicate that processing time generally grows proportionally to object size for objects larger than around 200 kB. While standard histograms and `TTrees` do not deviate significantly down to 10 kB, the sparse histograms have a noticeable minimal size, even if they did not allocate many bins.

Clearly, the decision about the accepted accuracy of such estimations depends on particular use-case. In the QC, where the standard histogram types are used mostly, we assume that processing time is proportional to object size when modelling Mergers.

B.5. General remarks

The benchmarks results greatly contributed in the estimations of required computing resources for merging incomplete results coming from parallel QC Tasks. They allowed to pinpoint the most influential factors contributing to performance of different data types and served as a base for extrapolation in order to obtain estimated load in any configuration.

The results indicate that dividing data structures into smaller components containing the same information adds a noticeable performance and storage overhead, therefore it should be avoided.

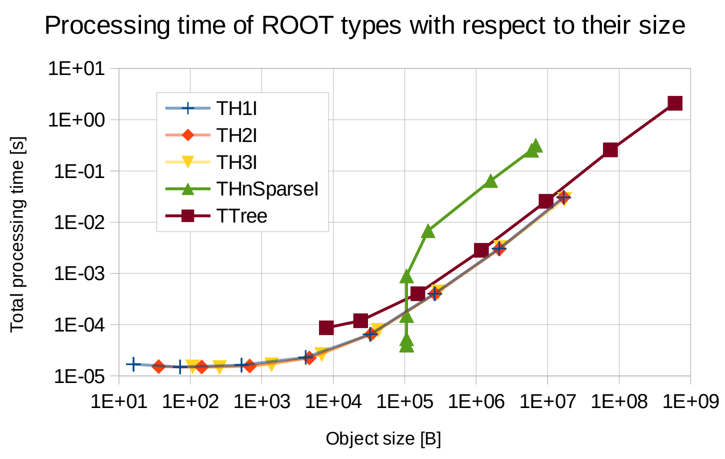


Fig. B.8. The influence of object size on total processing time for the five evaluated ROOT types.

Histogram types available in the `boost` library show great potential in reducing the processing time of QC data based on ROOT types, thus necessary steps will be performed to enable their presence in the QC system. On the other hand, ROOT is well established in the HEP experiments software, as it offers a variety of ready-to-use tools for statistical analysis and visualisation of its data types. Almost any experienced physicist in the related field is used to this framework, which justifies the presence of this toolbox in the QC system.

Bibliography

[1] E. Ozcesmeci and CERN. *LHC: pushing computing to the limits*. 2019. URL: *https://home.cern/news/news/computing/lhc-pushing-computing-limits* (visited on 2021-01-08).

[2] L. R. Evans and P. Bryant. "LHC Machine". In: *JINST* 3 (2008). This report is an abridged version of the LHC Design Report (CERN-2004-003), S08001. 164 p.

[3] The ALICE Collaboration. *The ALICE software repository*. 2020. URL: *https://github.com/alisw*.

[4] The ALICE Collaboration. "The ALICE experiment at the CERN LHC". In: *JINST* 3 (2008), S08002. DOI: *10.1088/1748-0221/3/08/S08002*.

[5] The ALICE Collaboration. "Upgrade of the ALICE Experiment: Letter Of Intent". In: *Journal of Physics G: Nuclear and Particle Physics* 41.8 (2014), p. 087001.

[6] B. von Haller et al. "Design of the data quality control system for the ALICE O2". In: *Journal of Physics: Conference Series* 898.3 (2017), p. 032001.

[7] P. Lesiak. "Development of the data quality assurance and visualization system for the Time Projection Chamber in ALICE experiment at the LHC". master. AGH University of Science and Technology in Cracow, 2016.

[8] G. Eulisse et al. "Evolution of the ALICE Software Framework for Run 3". In: *23rd International Conference on Computing in High Energy and Nuclear Physics (CHEP 2018)*. Vol. 214. Jan. 2019, p. 05010. DOI: *10.1051/epjconf/201921405010*.

[9] P. Konopka and B. von Haller. "Data Sampling methods in the ALICE O2 distributed processing system". In: *Computer Physics Communications* 258 (2021), p. 107581. DOI: *10.1016/j.cpc.2020.107581*.

[10] P. Konopka and B. von Haller. "The ALICE O2 data quality control system". In: *EPJ Web Conf.* 245 (2020), p. 01027. DOI: *10.1051/epjconf/202024501027*.

[11] The ATLAS Collaboration. "The ATLAS Experiment at the CERN Large Hadron Collider". In: *Journal of Instrumentation* 3.08 (2008), S08003.

[12] S. Chatrchyan et al. "The CMS experiment at the CERN LHC". In: *Journal of Instrumentation* 3 (Jan. 2008), S08004.

[13] The LHCb Collaboration. “The LHCb Detector at the LHC”. In: *Journal of Instrumentation* 3.08 (2008), S08005.

[14] G. Aad et al. “Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC”. In: *Physics Letters B* 716 (Sept. 2012), 1–29.

[15] S. Chatrchyan et al. “Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC”. In: *Physics Letters B* 716 (Sept. 2012), pp. 30–61.

[16] W. N. Cottingham and D. A. Greenwood. *An Introduction to the Standard Model of Particle Physics*. 2nd ed. Cambridge University Press, 2007. DOI: *10.1017/CBO9780511791406*.

[17] J. Rafelski. “Melting Hadrons, Boiling Quarks”. In: *The European Physical Journal A* 51 (Aug. 2015). DOI: *10.1140/epja/i2015-15114-0*.

[18] U. Heinz and M. Jacob. “Evidence for a New State of Matter: An Assessment of the Results from the CERN Lead Beam Programme”. In: *arXiv e-prints*, nucl-th/0002042 (Feb. 2000), nucl–th/0002042. arXiv: *nucl-th/0002042* `[nucl-th]`.

[19] The ALICE Collaboration et al. “ALICE: Physics Performance Report, Volume II”. In: *Journal of Physics G: Nuclear and Particle Physics* 32.10 (2006), pp. 1295–2040. DOI: *10.1088/0954-3899/32/10/001*.

[20] The ALICE Data Preparation Group. “ALICE data flow”. Internal presentation. 2019.

[21] The ALICE Collaboration. “Real-time data processing in the ALICE High Level Trigger at the LHC”. In: *Computer Physics Communications* 242 (2019), pp. 25 –48. ISSN: 0010-4655. DOI: *https://doi.org/10.1016/j.cpc.2019.04.011*.

[22] J. Shiers. “The Worldwide LHC Computing Grid (worldwide LCG)”. In: *Computer Physics Communications* 177.1 (2007). Proceedings of the Conference on Computational Physics 2006, pp. 219 –223. ISSN: 0010-4655. DOI: *https://doi.org/10.1016/j.cpc.2007.02.021*.

[23] K. Binder and D. W. Heermann. *Monte Carlo simulation in statistical physics: an introduction; 2nd ed.* Springer Series in Solid-State Sciences. Berlin: Springer, 1992. DOI: *10.1007/978-3-662-30273-6*.

[24] M. Bernardini and K. Foraz. “Long Shutdown 2 @ LHC”. In: *CERN Yellow Reports* 2.00 (2016), p. 290.

[25] B. Abelev et al. “Technical Design Report for the Upgrade of the ALICE Inner Tracking System”. In: *Journal of Physics G: Nuclear and Particle Physics* 41.8 (2014), p. 087002. DOI: *10.1088/0954-3899/41/8/087002*.

[26] The ALICE Collaboration. *Technical Design Report for the Muon Forward Tracker*. Tech. rep. CERN-LHCC-2015-001. ALICE-TDR-018. 2015.

[27] The ALICE Collaboration. *Upgrade of the ALICE Time Projection Chamber*. Tech. rep. CERN-LHCC-2013-020. ALICE-TDR-016. 2013.

[28] W. H. Trzaska. “New Fast Interaction Trigger for ALICE”. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 845 (2017). Proceedings of the Vienna Conference on Instrumentation 2016, pp. 463 –466. ISSN: 0168-9002. DOI: *https://doi.org/10.1016/j.nima.2016.06.029*.

[29] The ALICE Collaboration. *Technical Design Report for the Upgrade of the Online–Offline Computing System*. Tech. rep. CERN, 2015.

[30] T. Mrnjavac et al. “AliECS: a New Experiment Control System for the ALICE Experiment”. In: *EPJ Web Conf.* 245 (2020), p. 01033. DOI: *10.1051/epjconf/202024501033*.

[31] A. Wegrzynek and G. Vino. “The evolution of the ALICE O2 monitoring system”. In: *EPJ Web Conf.* 245 (2020), p. 01042. DOI: *10.1051/epjconf/202024501042*.

[32] S. Chapeland et al. “The ALICE DAQ infoLogger”. In: *Journal of Physics: Conference Series* 513 (June 2014), p. 012005. DOI: *10.1088/1742-6596/513/1/012005*.

[33] C. Hewitt, P. Bishop, and R. Steiger. “A Universal Modular ACTOR Formalism for Artificial Intelligence”. In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. IJCAI’73. Stanford, USA: Morgan Kaufmann Publishers Inc., 1973, 235–245.

[34] L. Torvalds, ed. *The Linux kernel repository*. URL: *https://github.com/torvalds/linux*.

[35] The TOP500 project. *The list of 500 most powerful commercially available computer systems*. URL: *https://www.top500.org*.

[36] *The CERN CentOS official website*. URL: *https://linux.web.cern.ch/centos/*.

[37] W. Stevens. *UNIX network programming*. Upper Saddle River, NJ: Prentice Hall PTR, 1998. ISBN: 0130810819.

[38] *The ZeroMQ library main website*. URL: *https://zeromq.org/*.

[39] *The ZeroMQ library online documentation*. URL: *http://zguide.zeromq.org/*.

[40] M. Boretto et al. “DAQling: an open-source data acquisition framework”. In: *EPJ Web Conf.* 245 (2020), p. 01026. DOI: *10.1051/epjconf/202024501026*.

[41] M. Al-Turany et al. “Extending the FairRoot framework to allow for simulation and reconstruction of free streaming data”. In: *Journal of Physics: Conference Series* 513.2 (2014), p. 022001. DOI: *10.1088/1742-6596/513/2/022001*.

[42] The FairRoot Group at GSI. *The FairMQ library online repository*. URL: *https://github.com/FairRootGroup/FairMQ*.

[43] *The boost library main webpage*. URL: *https://www.boost.org/*.

[44] J. Harvey. “Data Acquisition in High Energy Physics”. In: *Techniques and Concepts of High-Energy Physics V*. Ed. by T. Ferbel. Boston, MA: Springer US, 1990, pp. 347–406. ISBN: 978-1-4615-8001-0. DOI: *10.1007/978-1-4615-8001-0_8*.

[45] G. Eulisse et al. “Data Analysis using ALICE Run 3 Framework”. In: *EPJ Web Conf.* 245 (2020), p. 06032. DOI: *10.1051/epjconf/202024506032*.

[46] P. Cortese. *ALICE Time-Of-Flight system (TOF): addendum to the Technical Design Report*. Technical Design Report ALICE. Geneva: CERN, 2002.

[47] A. Akindinov et al. “Data quality monitor as the final quality assurance procedure for the ALICE-TOF detector”. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 602.3 (2009). Proceedings of the 9th International Workshop on Resistive Plate Chambers and Related Detectors, pp. 821 –824. ISSN: 0168-9002. DOI: *https://doi.org/10.1016/j.nima.2008.12.138*.

[48] F. Roukoutakis, S. Chapeland, and O. Cobanoglu. “The ALICE-LHC Online data quality monitoring framework: Present and future”. In: *Nuclear Science, IEEE Transactions on* 55 (Mar. 2008), pp. 379 –385.

[49] N. Abgrall et al. “NA61/SHINE facility at the CERN SPS: beams and detector system”. In: *JINST* 9 (2014), P06005. DOI: *10.1088/1748-0221/9/06/P06005*. arXiv: *1401.4699* `[physics.ins-det]`.

[50] W. R. Leo. “Ionization Detectors”. In: *Techniques for Nuclear and Particle Physics Experiments: A How-to Approach*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 127–156. ISBN: 978-3-642-57920-2. DOI: *10.1007/978-3-642-57920-2_6*.

[51] *ALICE Inner Tracking System (ITS): Technical Design Report*. Technical design report. ALICE. Geneva: CERN, 1999.

[52] R. Brun and F. Rademakers. “ROOT - An Object Oriented Data Analysis Framework”. In: *Proceedings AIHENP’96 Workshop* (Sept. 1996), pp. 81–86.

[53] B. Bellenot and S. Linev. “JavaScript ROOT”. In: *Journal of Physics: Conference Series* 664.6 (2015), p. 062033. DOI: *10.1088/1742-6596/664/6/062033*.

[54] A. Buckley. *The problem with ROOT (a.k.a. The ROOT of all Evil)*. Aug. 2016. URL: *http://insectnation.org/articles/problems-with-root.html*.

[55] F. Carena et al. “The ALICE data acquisition system”. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 741 (2014), pp. 130 –162. ISSN: 0168-9002. DOI: *https://doi.org/10.1016/j.nima.2013.12.015*.

[56] *MySQL 8.0 Reference Manual. What is MySQL?* Oracle corporation. 2020. URL: *https://dev.mysql.com/doc/refman/8.0/en/what-is-mysql.html*.

[57] C. Gaspar, M. Dönszelmann, and Ph. Charpentier. “DIM, a portable, light weight package for information publishing, data transfer and inter-process communication”. In: *Computer Physics Communications* 140.1 (2001). CHEP2000, pp. 102 –109. ISSN: 0010-4655. DOI: *https://doi.org/10.1016/S0010-4655(01)00260-0*.

[58] B. von Haller et al. “The ALICE Data Quality Monitoring: qualitative and quantitative review of three years of operations”. In: *Journal of Physics: Conference Series* 513.1 (2014), p. 012038.

[59] B. von Haller et al. “The new ALICE DQM client: a web access to ROOT-based objects”. In: *Journal of Physics: Conference Series* 664.6 (2015), p. 062064.

[60] R. Ehlers and J. Mulligan. “ALICE Overwatch: Online monitoring and data quality assurance using HLT data”. In: *EPJ Web Conf.* 214.arXiv:1812.00791 (2018). 8 pages, 4 figures, Proceedings of the 23rd International Conference on Computing in High Energy and Nuclear Physics (CHEP 2018), 9-13 July 2018, 01038. 8 p. DOI: *10.1051/epjconf/201921401038*.

[61] The ATLAS experiment. *Trigger and Data Acquisition System | ATLAS experiment at CERN*. 2021. URL: *https://atlas.cern/discover/detector/trigger-daq*.

[62] A. Corso-Radu et al. “Data quality monitoring framework for the ATLAS experiment at the LHC”. In: *Nuclear Science, IEEE Transactions on* 55 (Mar. 2008), pp. 417–420.

[63] C. Cuenca Almenar et al. “ATLAS Online Data Quality Monitoring”. In: *Nuclear Physics B - Proceedings Supplements* 215.1 (2011). Proceedings of the 12th Topical Seminar on Innovative Particle and Radiation Detectors (IPRD10), pp. 304 –306. ISSN: 0920-5632. DOI: *https://doi.org/10.1016/j.nuclphysbps.2011.04.038*.

[64] J. Adelman et al. “ATLAS offline data quality monitoring”. In: *Journal of Physics: Conference Series* 219.4 (2010), p. 042018.

[65] S. Kolos et al. “Experience with CORBA communication middleware in the ATLAS DAQ.” In: ATL-DAQ-2005-001. ATL-COM-DAQ-2004-019 (2005), 6 p. DOI: *10.5170/CERN-2005-002.105*.

[66] P. Conde Muino. “Portable gathering system for monitoring and online calibration at ATLAS”. In: *14th International Conference on Computing in High-Energy and Nuclear Physics*. 2005, pp. 111–114.

[67] P. Renkel and the ATLAS Collaboration. “The Gatherer – a mechanism for integration of monitoring data in ATLAS”. In: *Journal of Physics: Conference Series* 219.2 (2010), p. 022043. DOI: *10.1088/1742-6596/219/2/022043*.

[68] N. C. Benekos. “ATLAS Muon data quality with 2010 LHC”. In: *2011 2nd International Conference on Advancements in Nuclear Instrumentation, Measurement Methods and their Applications*. 2011, pp. 1–6. DOI: *10.1109/ANIMMA.2011.6172857*.

[69] The ATLAS Collaboration. “Monitoring and data quality assessment of the ATLAS liquid argon calorimeter”. In: *Journal of Instrumentation* 9.07 (2014), P07024.

[70] Y. Ilchenko et al. “Data Quality Monitoring Display for ATLAS experiment at the LHC”. In: *Journal of Physics: Conference Series* 219.2 (2010), p. 022035.

[71] V. Adler. “Data Quality Monitoring of the CMS Tracker”. In: *2009 IEEE Nuclear Science Symposium Conference Record (NSS/MIC)*. 2009, pp. 609–612. DOI: *10.1109/NSSMIC.2009.5401987*.

[72] S. Dutta. “The Data Quality Monitoring of the CMS Experiment: the Tracker Case”. In: *Nuclear Physics B - Proceedings Supplements* 197.1 (2009). 11th Topical Seminar on Innovative Particle and Radiation Detectors (IPRD08), pp. 271 –274. ISSN: 0920-5632. DOI: *https://doi.org/10.1016/j.nuclphysbps.2009.10.083*.

[73] F. M. Palmonari and S. Dutta. “The CMS tracker Data Quality Monitoring expert GUI”. In: *2009 IEEE Nuclear Science Symposium Conference Record (NSS/MIC)*. 2009, pp. 602–604. DOI: *10.1109/NSSMIC.2009.5401981*.

[74] The CMS Collaboration. *CMS Physics: Technical Design Report Volume 1: Detector Performance and Software*. Technical design report. CMS. Geneva: CERN, 2006.

[75] A. Batinkov et al. “The CMS Data Quality Monitoring Software: Experience and future improvements”. In: *2013 IEEE Nuclear Science Symposium and Medical Imaging Conference (2013 NSS/MIC)*. 2013, pp. 1–5. DOI: *10.1109/NSSMIC.2013.6829716*.

[76] M. Santa Mennea, G. Zito, and N. De Filippis. “Test of distributed data quality monitoring of CMS tracker”. In: *IEEE Nuclear Science Symposium conference record. Nuclear Science Symposium*. Vol. 2. Nov. 2005, pp. 852 –855. ISBN: 0-7803-9221-3.

[77] L. Borrello. “The Data Quality Monitoring Software for the CMS experiment at the LHC”. In: *2014 IEEE Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC)*. 2014, pp. 1–3. DOI: *10.1109/NSSMIC.2014.7431053*.

[78] M. Borisyak et al. “Towards automation of data quality system for CERN CMS experiment”. In: *Journal of Physics: Conference Series* 898 (2017), p. 092041. ISSN: 1742-6596. DOI: *10.1088/1742-6596/898/9/092041*.

[79] O. Callot et al. “Online data quality monitoring tools of LHCb”. In: *IEEE Nuclear Science Symposium conference record. Nuclear Science Symposium* (Oct. 2008).

[80] E. van Herwijnen et al. *Control and monitoring of on-line trigger algorithms using a SCADA system*. Tech. rep. LHCb-PROC-2006-006. CERN-LHCb-PROC-2006-006. Proceedings published on CD: Computing in High Energy and Nuclear Physics (CHEP-2006), Volume I and II, Editor Sananda Banerjee, MacMillan Advanced Research Series. Geneva: CERN, 2006.

[81] M Adinolfi et al. “LHCb data quality monitoring”. In: *Journal of Physics: Conference Series* 898.9 (2017), p. 092027.

[82] G. Barrand et al. “GAUDI — A software architecture and framework for building HEP data processing applications”. In: *Computer Physics Communications* 140.1 (2001). CHEP2000, pp. 45 –55. ISSN: 0010-4655. DOI: *https://doi.org/10.1016/S0010-4655(01)00254-5*.

[83] S. Petrucci, R. Matev, and R. Aaij. “Scalable monitoring data processing for the LHCb software trigger”. In: *EPJ Web Conf.* 245 (2020), p. 01039. DOI: *10.1051/epjconf/202024501039*.

[84] F. Abe et al. “The CDF Detector: An Overview”. In: *Nucl. Instrum. Meth. A* 271 (1988), pp. 387–403. DOI: *10.1016/0168-9002(88)90298-7*.

[85] S. Abachi et al. “The D0 detector D0 Collaboration”. In: *Nuclear Instruments and Methods in Physics Research A* 338.2-3 (July 1993).

[86] “Is it the Top Quark? Yes!!!” In: *FermiNews* 18.4 (May 1995).

[87] A. Brenner et al. “The Fermilab Collider Detector Facility Data Acquisition System”. In: *Nuclear Science, IEEE Transactions on* 29 (Mar. 1982), pp. 105 –110. DOI: *10.1109/TNS.1982.4335805*.

[88] W. Wagner et al. “Online Monitoring in the CDF II experiment”. In: *Proceedings of International Europhysics Conference on High Energy Physics — PoS(hep2001)*. Sissa Medialab, 2001. DOI: *10.22323/1.007.0273*.

[89] F. Scuri. “Online Monitoring for the CDF Run II Experiment and the remote operation facilities”. In: *PoS* ACAT (2007), p. 027.

[90] The DØ Collaboration. *DØ’s Data Quality Co-ordination*. 2007. URL: *https://www-d0.fnal.gov/computing/data_quality/*.

[91] V. Shary. “Data quality monitoring for the D0 calorimeter”. In: *International Conference on Calorimetry in High Energy Physics - CALOR2004 11* (Mar. 2004), pp. 205–209.

[92] L. K. Nuttall et al. “Improving the Data Quality of Advanced LIGO Based on Early Engineering Run Results”. In: *Classical and Quantum Gravity* 32 (Aug. 2015).

[93] R. Balasubramanian et al. “GEO 600 online detector characterization system”. In: *Classical and Quantum Gravity* 22.23 (2005), p. 4973.

[94] A. Irles et al. *DQM4HEP - A Generic Online Monitor for Particle Physics Experiments*. Tech. rep. AIDA-2020-CONF-2017-008. IEEE NSS/MIC 2017 Conference Record. Geneva: CERN, 2017.

[95] The CALICE Collaboration. “Construction and commissioning of the CALICE analog hadron calorimeter prototype”. In: *Journal of Instrumentation* 5.05 (2010), P05004–P05004. DOI: *10.1088/1748-0221/5/05/p05004*.

[96] The CALICE Collaboration. “First results of the CALICE SDHCAL technological prototype”. In: *Journal of Instrumentation* 11.04 (2016), P04001–P04001. DOI: *10.1088/1748-0221/11/04/p04001*.

[97] J. Gärtner. “Analysis of Entropy Usage in Random Number Generators.” Master dissertation. Stockholm: KTH, School of Computer Science and Communication (CSC)., 2017.

[98] M. E. O’Neill. *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation*. Tech. rep. HMC-CS-2014-0905. Claremont, CA: Harvey Mudd College, Sept. 2014.

[99] S. Vigna. *A fixed-increment version of Java 8's Splittable Random generator*. 2015. URL: *http://xorshift.di.unimi.it/splitmix64.c*.

[100] L. Mueller and T. Mueller. *What integer hash function are good that accepts an integer hash key?* Stack Overflow. URL: *https://stackoverflow.com/questions/664014/what-integer-hash-function-are-good-that-accepts-an-integer-hash-key/12996028\#12996028*.

[101] D. James. *Combining hash values*. URL: *https://www.boost.org/doc/libs/1_70_0/doc/html/hash/combine.html*.

[102] R. G. Brown, D. Eddelbuettel, and D. Bauer. *Dieharder: A Random Number Test Suite*. 2019. URL: *http://webhome.phy.duke.edu/~rgb/General/dieharder.php*.

[103] R. G. Brown. *dieharder(1) - Linux man page*. 2020. URL: *https://linux.die.net/man/1/dieharder*.

[104] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), pp. 1–84. DOI: *10.1109/IEEESTD.2019.8766229*.

[105] J. Lockhart, K. Rawashdeh, and C. Purdy. "Verification of Random Number Generators for Embedded Machine Learning". In: July 2018, pp. 411–416. DOI: *10.1109/NAECON.2018.8556780*.

[106] D. Lemire. *Testing non-cryptographic random number generators: my results*. 2017. URL: *https://lemire.me/blog/2017/08/22/testing-non-cryptographic-random-number-generators-my-results/*.

[107] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. .Template Method. Addison-Wesley Professional. Chap. Template Method, pp. 325–330. ISBN: 0-201-63361-2.

[108] R. Pachołek. „Implementacja systemu kontroli jakości do wzbudzania alarmów w ALICE O2". Master's thesis. AGH University of Science i Technology in Cracow, 2019.

[109] M. Krzewicki. *Real-time ROOT object merging in the HLT*. CERN ALICE internal QA tools meeting. URL: *https://indico.cern.ch/event/674811/contributions/2761002/attachments/1544407/2423716/2017-10-20-HLT-ROOT-Mergers.pdf*.

[110] R. T. Fielding. "Architectural Styles and the Design of Network-based Software Architectures." Doctoral dissertation. Irvine: University of California, 2000.

[111] *Project Jupyter*. 2020. URL: *https://jupyter.org/*.

[112] D. Piparo et al. "SWAN: a Service for Interactive Analysis in the Cloud". In: *Future Gener. Comput. Syst.* 78.CERN-OPEN-2016-005 (2016), 1071–1078. 17 p. DOI: *10.1016/j.future.2016.11.035*.

[113] *The ALICE O2 Web UIs repository*. The ALICE Collaboration. 2020. URL: *https://github.com/AliceO2Group/WebUi*.

[114] *The main Consul website*. HashiCorp. 2020. URL: *https://www.consul.io/*.

[115] The ALICE Collaboration. *The repository of the data quality control (QC) software for the ALICE O2 system.* 2020. URL: *https://github.com/AliceO2Group/QualityControl*.

[116] *GNU General Public License*. Version Version 3. Free Software Foundation, Inc. URL: *https://www.gnu.org/licenses/gpl-3.0.html*.

[117] *The main website of git*. 2020. URL: *https://git-scm.com/about*.

[118] R. B. Cooper. *Introduction to Queueing Theory*. 2nd. North Holland, 1981, p. 189. ISBN: 0-444-00379-7.

[119] G. J. Myers et al. *The Art of Software Testing*. Business Data Processing: A Wiley Series. Wiley, 2004. ISBN: 9780471469124.

[120] K. Albertsson et al. "Machine Learning in High Energy Physics Community White Paper". In: *J. Phys. : Conf. Ser.* 1085.arXiv:1807.02876. 2 (2018). Editors: Sergei Gleyzer, Paul Seyfert and Steven Schramm, 022008. 27 p. DOI: *10.1088/1742-6596/1085/2/022008*.

[121] K. R. Deja. "Using Machine Learning techniques for Data Quality Monitoring in CMS and ALICE experiments". In: *PoS* LHCP2019 (2019), 236. 11 p. DOI: *10.22323/1.350.0236*.

[122] P. W. Nowak. "Anomalies detection with autoencoders". In: Machine Learning and the Physical Sciences Workshop at the 33rd Conference on Neural Information Processing Systems (NeurIPS) (2019).

[123] CERN. *Longer term LHC schedule*. URL: *https://lhc-commissioning.web.cern.ch/schedule/LHC-long-term.htm*.