The ALICE Analysis Framework for LHC Run 3

Dario Berzano[1,2,*], *Roel* Deckers[1], *Costin* Grigoraş[1], *Michele* Floris[1], *Peter* Hristov[1], *Mikolaj* Krzewicki[1,3], and *Markus* Zimmermann[1]

[1]European Organization for Nuclear Research (CERN), Genève, Switzerland
[2]Istituto Nazionale di Fisica Nucleare (INFN), Italy
[3]Johann-Wolfgang-Goethe University and Frankfurt Institute for Advances Studies (FIAS), Frankfurt, Germany

Abstract. The ALICE experiment at the LHC (CERN) is currently developing a new software framework designed for Run 3: detector and software will have to cope with Pb–Pb collision rates 100 times higher than today, leading to the combination of core Online-Offline operations into a single framework called O^2. The analysis code is expected to run on a few large Analysis Facilities counting 20k cores and sustaining a 100 GB/s throughput: this requires a conjoint effort between the definition of the data format, the configuration of the Analysis Facilities and the development of the Analysis Framework. We present the prototype of a new Analysis Object Data format based on timeframes and optimized for continuous readout. Such format is designed to be extensible and transported efficiently over the network. We also present the first iteration of the Analysis Framework, based on the O^2 Data Processing Layer and leveraging message passing across a topology of processes. We will also illustrate the implementation and benchmarking of a compatibility layer designed to mitigate the transition from the current event-oriented analysis model to the new time-oriented one. Finally, we will give a status report on the integration of the Analysis Framework and Analysis Facilities for Run 3 into the current organized analysis model in ALICE.

1 Background and motivation

1.1 The ALICE analysis model in Run 2

The ALICE experiment is currently relying on an analysis framework based on several abstraction layers, and on an operational model based on organized analysis.

From the developer's perspective, the current framework is written in C++ and based on ROOT[1]. User analysis code, called *analysis task*, is also written in C++ in the form of a class overriding methods of a base abstract class. In particular, the user defines a function that will be executed for each event.

The component responsible for managing the *event loop* is the *analysis manager*. The manager features different types of backend plugins, and it is capable of running the same, unmodified user code on different runtime environment, including the user's laptop, the Grid and ROOT's PROOF-Lite[2]: the running mechanism is therefore abstracted away and transparent to the analyzer.

Another form of abstraction comes from the input data handling. Input data for analysis comes in the form of Analysis Object Data (AOD) or Event Summary Data (ESD) base files. Such formats can be integrated with extra data, using deltaAODs and ESD friends respectively. The analysis

*e-mail: dario.berzano@cern.ch

© The Authors, published by EDP Sciences. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (http://creativecommons.org/licenses/by/4.0/).

manager prepares a single event object by assembling different data sources and hands it to the event processing function of each analysis task, that remains unaware of the data provenance (one or multiple files) and source (from file, streamed over the network). The operation of writing the analysis results on an output file is abstracted as well.

The analysis manager is capable of concatenating several analysis tasks: like this, data is read only once and processed many times, and common processing results are shared across the tasks chain. This ability is leveraged at the operational level through the *analysis trains*: Grid operations are managed by a restricted number of persons, called the *train operators*. Operators collect analysis requests from the various Physics Working Groups (PWGs) and compose *trains* constituted by several analysis *wagons* processing the same data. This mechanism reduces the performance issues due to slow data access (data is accessed only once per train and processed by many wagons), and protects users from the complexity of running on the Grid.

In order for organized analysis to happen, all ALICE analysis tasks are collected in a single repository with continuous integration, and automated builds occur every day and are deployed on the Grid through CVMFS[3].

1.2 Current model's performance wall

The typical average processing rate of the current analysis model is estimated around 50 GB/s when 50 000 concurrent Grid analysis jobs are running, leading to a read throughput of 1 MB/s per job. The average analysis jobs efficiency (fraction of the total processing time used by the CPU) is 50%. Given that 98% of the data read by each job is local to the Grid site where the job is running, what contributes the most to such a small efficiency figure are I/O issues related to insufficient internal bandwidth and unscheduled site failures.

For Run 3 processing, ALICE needs to process a full 5 PB analysis dataset in 12 hours[4], leading to a required processing rate of 100 GB/s, larger than what the current model can sustain.

Storage configuration issues aside, there are other limiting factors not taken into account by the sole job efficiency. The first one is *decompression*: input data is currently stored in the form of gzip-compressed ROOT files, which are not decompressed in parallel. This is not a limiting factor for Run 2 analysis as storage performance does not allow us to appreciate any improvement in decompression, however it will be in Run 3.

The way data is structured and written to file can also be improved: at the moment, ALICE leverages ROOT to deserialize data as C++ class instances, and most of the structures are nested: this means that data needs to be heavily restructured from disk to the memory. Such restructuring comes with a large performance penalty: we will see an estimation of the decompression and deserialization cost later on in §2.1.

It has to be noted that writing analysis results to disk is not an issue: ALICE outputs are constituted by small trees or histograms, and less than 100 MB of data is produced per analysis run.

1.3 Development areas for LHC Run 3

ALICE is actively developing a new software framework for LHC Run 3 called O^2: as the name suggests, the framework is common and shared across Online and Offline processing[4]. Given the performance issues in analysis (§1.2), the whole analysis workflow is being redesigned in the context of O^2.

In the next sections we will see in more detail what are the development areas for a new ALICE analysis framework in LHC Run 3. First off, we will see how specially designed analysis facilities can be built to handle complex analysis workflows and to reduce storage performance issues (§2).

Our effort and preliminary tests on a new data format are illustrated in §3.

The inclusion of the Run 3 analysis framework in the novel, data-driven workflow handling model for O^2, called Data Processing Layer (DPL), is covered in §4.

The adoption of ROOT's `RDataFrame`, a user-facing API allowing users to write their analysis tasks in a more declarative and abstract way is explained in §5.

Finally, §6 showcases two prototypes realized by assembling all the components together.

2 Analysis facilities for Run 3

The O^2 design[4] provides for an analysis model that differs from the current distributed one: ALICE's intentions are to run organized analysis on two or three dense *analysis facilities*, counting 20 000 cores each, equipped with fast local storage and an internal network capable of sustaining high rates of data transfer from the storage and among the computing nodes.

Local storage for Run 3 analysis facilities is supposed to be non-custodial: analysis data is strictly kept for sole amount of time required for analysis; it is wiped soon afterwards in order to make room for new data. Each facility only analyzes local data, to reduce problems due to slow network connections or remote storage instabilities.

Fast internal network allows for data to be moved quickly from the storage to the nodes, and also allows for job topologies of intercommunicating jobs, which is the model used by the DPL as we will see in §4.

2.1 Analysis facility prototype at GSI

ALICE is currently running a small-scale prototype of a Run 3 analysis facility at GSI in Darmstadt, Germany[5]. The prototype allows for testing the O^2 development, and runs current Run 2 jobs when idle.

The network file system used by the facility is Lustre[6]. Different tests on the file system speed, with data uniformly distributed over the Object Storage Service (OSS) units, were performed under different load conditions. Two types of tests were run: simple byte-by-byte copy of files, and reading performance tests of the current Run 2 analysis framework.

Byte-by-byte copy tests give us an estimate of the highest transfer speed we can reach on the facility. Framework tests were run by executing the analysis manager's event loop (§1.1) without any analysis task attached. Since the framework still goes through the process of reading, uncompressing and deserializing input data, this tests allow us to identify and measure the framework's bottlenecks.

Results show that the Run 2 framework performs at 32 GB/s with 2 500 concurrent jobs; transferring raw data from a single job yields 1.2 GB/s, whereas concurrent transfers from 1 500 jobs performs at 600 GB/s. We can therefore conclude that the current hardware setup does not constitute a bottleneck for the Run 3 analysis model, as long as we keep the processing local to the analysis facility and, therefore, under stricter control and less subject to sporadic failures: the measured transfer speed is well above the required rate of 100 GB/s. It has to be noted that the Lustre model can scale by adding more OSSes. The GSI analysis facility is a good model that should be replicated to the other future analysis facilities.

We can clearly see, however, how the current framework is evidently slow *per se* by performing the sole operations of decompressing and deserializing data. We will see in the next sections how we intend to tackle the data format and framework issues.

3 Data model and format

3.1 Timeframes

The unit of information available for ALICE in Run 2 is the *event*: events are defined by the presence of specific triggers of the detector.

 https://doi.org/10.1051/epjconf/201921405045

ALICE data taking in Run 3 is continuous and triggerless: the unit of information is a snapshot of what happened during a 23 ms-long time window. This unit of information is called *timeframe*.

Down the reconstruction chain there exist different types of timeframe, each with a size that becomes smaller and smaller. Data flow, from raw data to the AOD, is represented in Figure 1. A single timeframe holds around 1 000 Pb–Pb collisions: such timeframe is 10 GB large when outputted by the First Level Processors (FLPs).

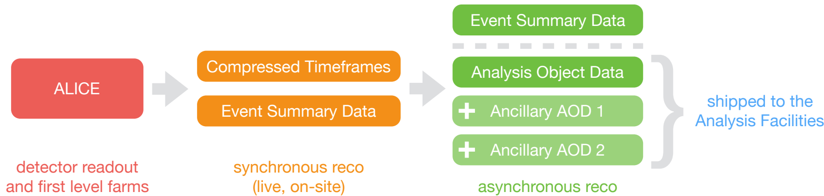

Figure 1. Data flow from the ALICE detector to AODs in O^2

FLP output is reconstructed in real time, producing timeframes of 2 GB. AODs are produced during the offline (asynchronous) reconstruction: an AOD timeframe weighs 1 GB.

The type of data available to the analyzer is no longer, as said, the event, but a collection of vertices and tracks and their features. The association between vertices and tracks can be represented by a probability. Since timeframes snapshot a relatively large amount of time, they can be safely processed independently by minimizing "border effects" such as tracks associated to vertices in the adjacent timeframes.

3.2 Run 3 analysis data format requirements

Run 2 framework results discussed in § 2.1 show the cost of reading and restructuring Run 2 data from disk to the memory. This cost should be reduced in Run 3 by designing a data format representing simpler entities, but equally extendable. Some generality will inevitably be lost to the performance's benefit.

Simpler entities, in this context, mean *flat lists of pure numbers* as opposed to serialized C++ class instances and nested collections. By dropping the nested structures we will cross-reference data from different lists through simple numeric indices, as it would be done by an SQL's `INNER JOIN` query for instance.

We have seen how the current format is extendable through extra files. We want to retain this concept by designing an *immutable* AOD format containing certain *columns*, and by reserving the possibility to create AOD extensions containing additional columns. This means that the new data format has to be *columnar*, which is essentially the ability to store columns data contiguously. Columnar formats are extendable because once an additional column is added, the memory offset between the existing columns and the additional one is fixed, making random access happen in constant time. An additional feature of a columnar format is that its memory representation is suitable for immediate vectorization.

Given an AOD estimated timeframe size of 1 GB, it is desirable to have the ability to process different parts of an existing timeframe in parallel. In other words, the chosen format needs to support *chunking*.

A simple data representation can be copied as-is, without restructuring, from disk to the memory and over the network. Using shared memory features and custom memory pools it would even be possible to not copy data at all *(zero copy)* between different buffers (such as the network buffer and the data processing one).

3.3 Data format: Apache Arrow

Our current data format prototype is based on Apache Arrow[7] as it appears to match the requirements enumerated in §3.2. Arrow is an in-memory, columnar data format designed to be memory efficient and to ease vectorization.

Arrow is widely used as a data interchange format across several popular software projects such as Pandas[8] and Spark[9] written in different programming languages. Arrow has a lively community and it is being actively developed. The presence of C++ bindings means it is easy to integrate it with the ALICE O^2 framework.

Arrow data is organized into *Tables*, each one being a collection of several immutable *Columns*. As said already, column data is stored contiguously in memory. Tables can share Columns without duplication. As we will see in §4, the O^2 data processing model is based on message passing: given Arrow's ability to define custom *Memory Pools*, it is technically feasible to share memory with the message buffers without copying data.

Finally, Arrow natively splits data into small chunks, allowing for unordered and parallel processing of a single timeframe.

4 The O^2 Data Processing Layer

The key points of ALICE O^2 are the presence of a processing framework shared between online and offline operations, and parallelism based on multiple processes exchanging messages. The rationale behind using multiple processes (called *devices*) is to make processing more immune from single unit failures. The paradigm of parallelism based on message passing is considered less error prone and more intuitive to the diverse community of developers of ALICE.

The O^2 devices and message passing model is embraced by the Data Processing Layer (DPL), that simplifies the way the single processing units are developed by allowing us to declare in a simple way how data flows between them. In other words, the DPL is an abstraction layer that makes the development of interconnected devices *data driven*.

The DPL is currently under very active development[10]. Under the hood, communication is handled through FairMQ[11], which in turn uses either ZeroMQ[12] or a shared memory backend. In particular, the DPL is capable of streaming the Apache Arrow memory format efficiently through message passing.

From the practical perspective, what the DPL user does is developing a C++ code declaring how input and output slots are connected, and what the single processing units do. This can be done in as little as a single file[13]. The communication protocol, network connections, memory initialization and sharing are never exposed to the user's task and can be constantly optimized without requiring any action from the user, whose code never changes.

Prototypes of the O^2 simulation[14] and reconstruction[10] workflows are using the DPL. We have based the analysis workflow prototype on it as well. A possible workflow for analysis is illustrated in Figure 2: in it we can see how a single AOD reader streams different columns to a number of parallel decompressors. Uncompressed timeframe columns are sent separately to each task. Some tasks may require some kind of preprocessing that adds new columns computed in real time. The single tasks subscribe to certain data columns and they will only receive those. Output data produced by the tasks is eventually collected and merged.

We can see how conceptually we can solve the problems of managing a topology of tasks and parallelizing the decompression using a single flexible tool. Such topology can work on a single node or even across multiple computing nodes in a transparent way.

The workflow is the natural evolution of the Run 2 train model (§1.1). There are two major differences though. Devices process data when received: there is no "event loop" handling any longer, but a publisher-subscriber model, in which the processing unit is called back whenever its

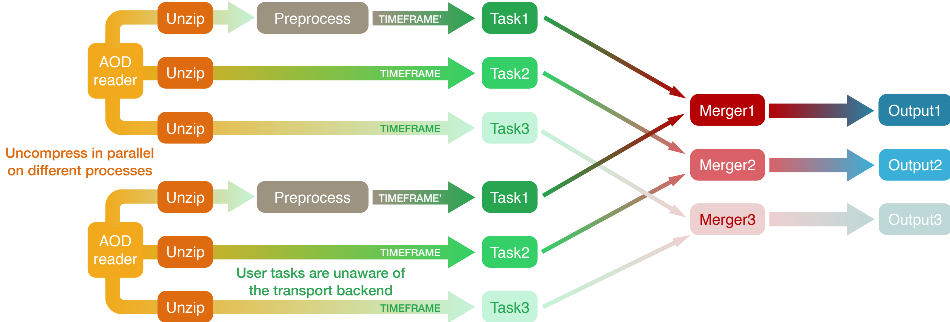

Figure 2. Example of an ALICE O^2 analysis workflow topology defined through the Data Processing Layer

data is ready. Moreover on the operational level, most of the job dependencies currently handled by the Grid middleware are performed by the DPL itself, allowing for a true full workflow replication on different runtime environments. This essentially means that the full analysis facility workflow can be run locally for easier debugging.

It is worth mentioning that one of the advantages of using a single framework makes it possible to plug the analysis into an existing running topology. For instance, Quality Assurance (QA) is currently executed in the form of analysis tasks: in Run 3 it can be a full topology being fed reconstructed data in real time without additional efforts.

5 User-facing API: RDataFrame

The current analysis framework prototype for O^2 is based on ROOT's `RDataFrame`. `RDataFrame` is a high-level interface available in ROOT for processing input data coming from multiple sources. A typical analysis processing would be expressed as:

```
RDataFrame d(i).Filter(c).Foreach(l)
```

with `i` being the input dataset, `c` some filtering criteria, and `l` a lambda function doing the heavy lifting.

What is innovative about the `RDataFrame` is the ability to concatenate several processing stages that are lazily executed: that is, `RDataFrame` takes care of optimizing the execution of each stage, running only the useful parts and only when required.

In most analysis use cases, actual processing is performed over a subset of filtered data: since filtering is a very common use case, a `Filter()` function is provided allowing as only degree of freedom the declaration of the filtering criterion. How the function actually selects data should be of no concern to the end user, and all the optimization efforts are factored out and left to the hands of the framework maintainers.

This is what can be called a *declarative* approach to analysis: strict control over processing is in part relinquished to the benefit of code readability and optimization of common code.

`RDataFrame` supports several input sources, and the processing itself is independent from the particular one used. ROOT trees are supported natively, and ALICE contributed with a plugin that feeds Apache Arrow Tables[15], a format we have selected for our first prototypes (§3.3). The fact that user tasks are unaware of the data source gives us the flexibility to change it in the future without altering existing code.

Another aspect of using RDataFrame is that implicit multithreading capabilities can be exploited depending on the input source. Our current Arrow source implementation allows RDataFrame to process the several chunks constituting a single timeframe in parallel, if desired and appropriate.

6 Prototypes

Two prototypes have been developed to test the feasibility and performances of the current design.

6.1 Arrow Column streamer

The first prototype integrates the three major components described so far: Apache Arrow (§3.3), the O^2 Data Processing Layer (§4) and ROOT's RDataFrame (§5). The example, called o2ArrowColumnStreamer[16], exploits the DPL to stream data in the Arrow format.

Every task only receives data it has subscribed to, and different Arrow columns are received on different channels: when all the columns of a timeframe reach the destination, the subscriber task puts them together dynamically in a new Arrow table. As we have seen, this operation does not copy memory.

The newly created table is supplied as input to the RDataFrame: by looking at the code[16] we can see how ROOT's "implicit multithreading" functionality has been enabled. In this case, the RDataFrame is instructed to parallelize the execution of the supplied lambda function over the different chunks in which a single timeframe is subdivided by Arrow. It has to be noted that a special ForeachSlot function is used to provide an index of the current processing slot to the user, so that we can appropriately create different output slots per parallel executor and merge the results at the end.

Concerning Arrow's chunking, the split level can be configured when producing the data. The current Arrow RDataFrame input source exploits chunking for multithreading, whereas this is not currently possible using a ROOT tree coming from a single file (implicit multithreading on trees only works by reading a TChain of input trees coming from different files, with one thread per file).

The prototype was developed using a version of DPL only supporting ZeroMQ as transmission backend: this means, in practice, that data was effectively copied when transmitted and received, and not shared as it should. The moment this feature is available in DPL it will be enabled automatically without any change to our prototype code. This constitutes a good example of how isolating different features can make most of the optimization happen behind the scenes. Since user code is forced to be independent on the data format and the running platform, the approach is beneficial to the overall code quality.

The example also shows that integrating the Arrow library into our code is a relatively easy task. The whole topology is described by a single C++ source code file. This simple example enables power users to start writing analysis tasks for Run 3 while we continue developing the framework, meeting one of the most compelling demands of our community.

6.2 Parallel decompressor

A second example was developed in order to evaluate the feasibility of exploiting a DPL-based topology in order to address the decompression bottleneck, as already theorized in Figure 2. A single reader reads data from a file: data is split and sent to different decompressors (two in the figure), and decompressed data is sent to a number of consumers.

We exploit a type of parallelism offered by the DPL, the so-called *time-based* parallelism. DPL always ensures that timeframes are processed in order: with time-based parallelism, each timeframe

produced by the reader is sent in a round-robin fashion to the different n decompressor devices:

$$\underbrace{t_{n\cdot i}}_{\text{device 0}}, \underbrace{t_{n\cdot i+1}}_{\text{device 1}}, \ldots, \underbrace{t_{n\cdot i+n-1}}_{\text{device } n} \quad (1)$$

We can see from the code[17] that decompressors and consumers are defined only once: the DPL takes care of creating the desired amount of clones. The data-driven approach makes the user code of a single device completely independent from the number of parallel executors.

The compression algorithm used is LZ4, using the "default" compression level, the recommended one as it represents a good compromise between speed and compressed size[18]. Recent ROOT versions use it by default as a replacement to gzip[19]. Tests were performed on a reference node, using first a DPL topology with a single reader and a single consumer to test the "framework speed", and then by gradually incrementing the number of compressors. Raw transfer speed allowed by the test machine and the framework is 2.2 GB/s; one decompressor performs at 0.6 GB/s, and the speed scales linearly up until 4 decompressors, where the raw transfer speed is finally hit. This in practice means that on a single reference machine, 4 cores can be fully dedicated to decompressing the input file, whereas the rest of the computing power is free for being exploited by the analysis.

The DPL makes it easy to plug a new functionality in without too much effort. As the single devices are only defined by the input data they subscribe to, the parallel decompressor can be simply integrated with the example illustrated in §6.1.

6.3 Conclusions

The design and prototypes realized so far are showing how the O^2 framework is ready for exploiting multiple dimensions of parallelism depending on the running platform and conditions. The different analysis tasks and decompressors can be expressed by a multiprocess paradigm where message passing is involved.

Within a single analysis task, thanks to the chunking ability provided by Arrow and the relative `RDataFrame` data source, a basic form of multithreading is possible without major rewriting of the analysis code.

The Arrow in-memory columnar structure makes it possible to vectorize certain operations: some optimization may even occur transparently at compile-time, or more explicitly by using certain C++ features available from C++17 onwards.

Lastly, on an analysis facility-like scenario where multiple execution nodes are involved, it is possible to repeat the same topology, on different input files, on each single node: our analysis is embarrassingly parallelizable at the input file level.

Our design effort and the development of working prototypes has made us reach the first important objective of ALICE analysis in LHC Run 3: power users can be given something tangible to work on to start developing native Run 3 analysis tasks. Given the completely different data taking model, O^2 behaves, from the analysis perspective, like a completely different experiment, and it is very difficult to programmatically convert existing analysis tasks to the new model.

The prototypes show the effectiveness of an approach based on treating user code, data format, workflow handling and runtime environment as separate entities that can be swapped without changing the rest. ROOT's `RDataFrame` was very helpful in making this separation possible by pushing a declarative approach.

The Data Processing Layer (DPL) is the cornerstone of all recent O^2 development. The presence of a common data flow model shared across different use cases is beneficial to the code robustness and it makes possible to combine the different topologies as well.

Finally, it has to be noted how the development directions and the prototypes realized so far do not neglect the years of experience achieved with the ALICE trains organized analysis model and the

Run 2 analysis framework. The current framework is considered successful from the performance and usability perspectives because it factors out critical parts and hides their complexity and optimization to the user: the Run 3 analysis framework design paradigm is pushing towards making the analysis model even more declarative, as well as data driven.

References

[1] R. Brun, F. Rademakers, *ROOT—an object oriented data analysis framework*, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **389**, 81 (1997)

[2] G. Ganis, J. Iwaszkiewicz, F. Rademakers, *Data analysis with PROOF*, in *XII Advanced Computing and Analysis Techniques in Physics Research* (SISSA Medialab, 2009), Vol. 70, p. 007

[3] J. Blomer, C. Aguado-Sánchez, P. Buncic, A. Harutyunyan, *Distributing LHC application software and conditions databases using the CernVM file system*, in *Journal of Physics: Conference Series* (IOP Publishing, 2011), Vol. 331(4), p. 042003

[4] P. Buncic, M. Krzewicki, P. Vande Vyvre, *Technical design report for the upgrade of the online-offline computing system*, CERN-LHCC-2015-006 (2015)

[5] K. Schwarz, *A prototype for the ALICE Analysis Facility at GSI*, in *Computing in High-Energy and Nuclear Physics 2018* (2018), https://indico.cern.ch/event/587955/contributions/2937941/

[6] *The Lustre network file system*, http://lustre.org/ (last accessed: March 27, 2019)

[7] *Apache Arrow*, https://arrow.apache.org/ (last accessed: March 27, 2019)

[8] *The Python Data Analysis Library*, https://pandas.pydata.org/ (last accessed: March 27, 2019)

[9] M. Zaharia, R.S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M.J. Franklin et al., *Apache spark: a unified engine for big data processing*, Communications of the ACM **59**, 56 (2016)

[10] G. Eulisse, P. Konopka, M. Krzewicki, M. Richter, D. Rohr, S.C. Wenzel, *Evolution of the ALICE Software Framework for LHC Run 3*, in *Computing in High-Energy and Nuclear Physics 2018* (2018), https://indico.cern.ch/event/587955/contributions/2938144/

[11] M. Al-Turany, A. Rybalchenko, D. Klein, T. Kollegger, A. Lebedev, A. Manafov, R. Karabowicz, F. Uhlig, D. Kresan, *ALFA: ALICE-FAIR new message queuing based framework*, in *Computing in High-Energy and Nuclear Physics 2018* (2018), https://indico.cern.ch/event/587955/contributions/2938082/

[12] *ZeroMQ*, http://zeromq.org/ (last accessed: March 27, 2019)

[13] *Data Processing Layer reference manual*, https://github.com/AliceO2Group/AliceO2/blob/dev/Framework/Core/README.md (last accessed: March 27, 2019)

[14] S.C. Wenzel, *A scalable and asynchronous detector simulation system based on ALFA*, in *Computing in High-Energy and Nuclear Physics 2018* (2018), https://indico.cern.ch/event/587955/contributions/2937621/

[15] *ALICE contribution to ROOT that adds support for Apache Arrow datasets to RDataFrame*, https://github.com/root-project/root/pull/1712 (last accessed: March 27, 2019)

[16] *O^2 Arrow Column streamer*, https://github.com/dberzano/AliceO2/blob/analysis/Framework/TestWorkflows/src/o2ArrowColumnStreamer.cxx (last accessed: March 27, 2019)

[17] *O^2 parallel decompressor*, https://github.com/dberzano/AliceO2/blob/analysis/Framework/TestWorkflows/src/o2ParallelDecompressor.cxx (last accessed: March 27, 2019)

[18] *LZ4*, https://github.com/lz4/lz4 (last accessed: March 27, 2019)

[19] B. Bockelman, Z. Zhang, J. Pivarski, *Optimizing ROOT IO For Analysis*, arXiv:1711.02659 (2018)