EPJ Web of Conferences **251**, 03063 (2021) https://doi.org/10.1051/epjconf/202125103063
CHEP 2021

ALICE Run 3 Analysis Framework

Anton Alkin[1,*], *Giulio* Eulisse[1], *Jan Fiete* Grosse-Oetringhaus[1], *Peter* Hristov[1], and *Maja* Kabus[2]

[1]Organisation Européenne pour la Recherche Nucléaire, CERN, Meyrin, Switzerland
[2]Faculty of Mathematics and Information Science, Warsaw University of Technology, Warsaw, Poland

Abstract. In LHC Run 3 the ALICE Collaboration will have to cope with an increase of lead-lead collision data of two orders of magnitude compared to the Run 1 and 2 data-taking periods. The Online-Offline (O^2) software framework has been developed to allow for distributed and efficient processing of this unprecedented amount of data. Its design, which is based on a message-passing back end, required the development of a dedicated Analysis Framework that uses the columnar data format provided by Apache Arrow. The O^2 Analysis Framework provides a user-friendly high-level interface and hides the complexity of the underlying distributed framework. It allows the users to access and manipulate the data in the new format both in the traditional "event loop" and a declarative approach using bulk processing operations based on Arrow's Gandiva sub-project. Building on the well-tested system of analysis trains developed by ALICE in Run 1 and 2, the AliHyperloop infrastructure is being developed. It provides a fast and intuitive user interface for running demanding analysis workflows in the GRID environment and on the dedicated Analysis Facility. In this document, we report on the current state and ongoing developments of the Analysis Framework and of AliHyperloop, highlighting the design choices and the benefits of the new system.

1 ALICE software framework

To cope with the challenges of LHC Run 3 and to ensure a unified and coherent computing environment from data taking up to analysis, the ALICE Collaboration is developing a new computing software framework [1, 2]. The new framework is named O^2 [3], from *Online-Offline*, as it integrates both the domain of data readout and its subsequent processing. It derives originally from ALICE High Level Trigger architecture [4], a message-passing multiprocess system, used in Run 1 and 2, and consists of three major components, the transport layer, the data model and the data processing layer. The transport layer of the system, a component that provides inter-process communication, is implemented using the FairMQ [5] message passing toolkit, defining the core blocks of the architecture in terms of FairMQDevices (devices) and their communication in a medium-agnostic way. A shared-memory-backed message passing transport for devices running on the same computing node is available. On top of the transport layer, ALICE implements the O^2 data model, a programming-language-agnostic extensible description of messages being exchanged, allowing efficient transport and mapping to data objects stored in shared or GPU memory, where required. Various

*e-mail: anton.alkin@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/).

data formats and serialization methods are provided, including specialized formats used by subdetectors. In particular, the Apache Arrow [6] columnar format is supported, as are ROOT-serializable [7] objects, such as histograms, that play a central role in physics analysis. Apache Arrow provides an efficient contiguous in-memory layout for large amounts of data and libraries that implement zero-copy streaming and inter-process communication. The Arrow format is widely used in opensource and commercial data analysis tools, which, together with its technical merits, makes it a natural choice for the analysis framework, ensuring interoperability. Finally, the Data Processing Layer (DPL) [8], developed since 2017, has been introduced to hide the complexity of the abstract transport layer and low-level data model. It allows concise logical descriptions of a set of data processors and their interconnections, which defines a certain data analysis workflow. The DPL automatically builds the workflow topology based on interdependencies between data processors, including user-defined ones. It hides the underlying complexity of connecting data producers to data consumers thus ensuring correct multiplexing and pipelining where needed.

ALICE in Run 3 will run in so-called continuous data-taking mode, with the unit of information being a snapshot of data in a 10 ms-long time window, dubbed timeframe. The timeframe represents the minimal processing unit at all stages of the data processing, from the data taking, synchronous reconstruction, asynchronous reconstruction phase phase, and up to the final analysis. Each timeframe is processed independently. The result of asynchronous reconstruction is the Analysis Object Data (AOD) format, with the best calibration available. During processing the content of an AOD timeframe is kept contiguously in shared memory allowing efficient application of both parallel execution and pipelining, sharing I/O cost between the devices in the workflow. AODs are stored as ROOT trees, which is a natural choice as they map directly on columnar in-memory layout. ROOT I/O also allows for bulk reading and writing, which is essential for pipelining. Because of the continuous nature of the data taking in Run 3, vertex-to-track association, for example, is no longer unambiguous, and thus collisions and tracks are represented as separate tables, connected by an index. This maps naturally on a "flattened" structure-of-arrays data format, which was chosen for its efficiency in leveraging modern hardware capabilities. Thus the analysis data model differs considerably from the hierarchical "event contents" of Runs 1 and 2.

The DPL hides most of the underlying complexity of the framework. In fact, the DPL is used almost everywhere in ALICE computing system, including the simulation and reconstruction, as it is a very convenient tool for building and extending generic data processing pipelines. In order to fully exploit the potential of the DPL also for physics analyses, additional developments were required, resulting in the creation of the Analysis Framework as an extension of the DPL. In Run 1 and 2, ALICE developed a system of analysis *trains* to optimize the usage of computing resources [9]. The train infrastructure relied on interactive web tools that allowed the user to configure analysis tasks (or wagons) that are expected to be run on the same data sets. A similar system, based on the React framework [10] (front end) and integrated in the Monalisa monitoring and control system [11] (back end), will be used for Run 3. The system, called AliHyperloop, can benchmark each analysis in terms of functionality and resource consumption and then compose trains that are later submitted to the GRID or a dedicated Analysis Facility - a specialized Grid site with CPU and disk resources adjusted for analysis needs and small fraction of data pre-staged locally. AODs will be collected on specialised Analysis Facilities which shall be capable of processing 10 PB of data within a time scale of one day, which corresponds to an analysis throughput of about 115 GB/s [12]. The system assures bookkeeping of analysis configurations. It also interacts with the Grid submission infrastructure to provide job monitoring and resubmission, as well as merging the final output files. The production of derived filtered data sets will also rely on the the same system.

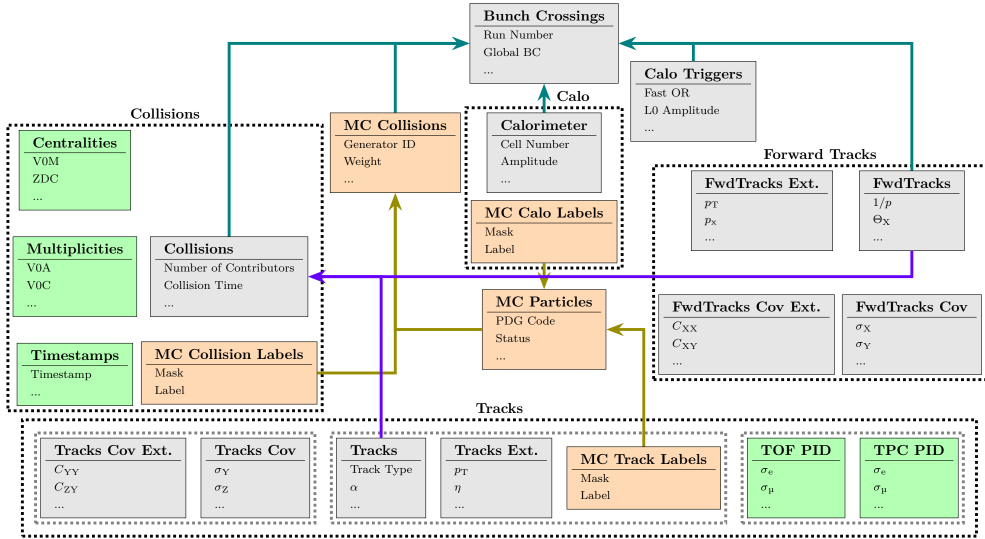

Figure 1. Schematic (partial) of AOD tables connections, including both new Run 3 and Run 2 data converted to Run 3 format representations. Tables in gray are present on disk or can be calculated from those present on disk. Tables in orange are tables present in simulated data. Tables in green can be calculated from those on disk using calibration data. Directed arrows indicate index connections.

2 DPL Analysis Framework

2.1 Core design

The analysis data model in Run 3 is a collection of flat tables, arranged in a relational database-like structure using index connections. The definition of a particular table includes all the necessary metadata needed by the framework to handle it through the DPL. Using the same tools, end users can define their derived data tables that will behave in the same way. The preliminary structure of the data model is schematically presented in Fig. 1. Any self-contained cluster of data, like the one containing the information about collisions or the tracks, is represented as a single table. The information stored in separate tables can be combined by performing join operations (similar to that in SQL). By adopting this approach, one can limit the amount of the data read into memory based on the workflow requirements.

Individual tables are represented as distinct C++ types, defined by their list of *columns* backed by Arrow `ChunkedArrays` in memory. The table class provides an *iterator* interface similar to standard C++ containers, that allows users to access table rows sequentially, essentially iterating over all columns simultaneously, and also provides an OOP-like accessor to the row contents. A database-like structure allows implementation of various zero-copy operations such as joining (merging column lists of compatible tables) and grouping (extracting a subset of rows that have a particular value in a certain column). Index columns define pre-existing connections between tables that can be followed to access the information from related tables. Additionally, index tables can be dynamically created to define new connections required by a particular physics analysis, such as, for example a table for rare decay candidates, containing indices referring to a pair or triplet of tracks. This also allows the Framework to create indices in opposite directions or cross-indices on the fly. For example tables for triggered detectors data have index to the bunch crossings (BCs) table, as triggers are tied to bunch crossings, but not specific collisions. The collisions table has index to the

BCs table as well. These connections make it possible to automatically construct on demand an index table relating triggered detectors tables to collisions directly.

The analysis workflow is a modular and flexible collection of interconnected DPL data processors. Continuing from the Run 1 and 2 train system individual worfklows that are combined into a train are known as *wagons*. It expands on the previous approach with sequential Analysis Tasks [13] adopted in Run 1 and 2 by adding multi-process capability. Thanks to the possibility of processing input data in a sequential manner through pipelines, common tasks (e.g. file reader, writer, calibration and quality filtering tasks) can be shared among different analyses. The DPL workflow merging feature is particularly useful in this regard, as different analysis workflows can be dynamically built based on user input and deployed in any available computing environment. The final merged workflow can be serialized using JSON.

The particular data processors, known as *tasks* for similarity with the previous analysis framework, are created by the end users and provide a C++ structure with conventionally defined callbacks and declarations of inputs/outputs. Any table known by DPL can be used as input, including user-defined tables, allowing to build complicated multi-staged analysis algorithms from self-contained blocks. Outputs can be transient (intended for further processing) or persistent (intended for storage, like skims, i.e. small filtered subsets of data kept for further processing) derived data tables, as well as analysis objects, like histograms. A dedicated `HistogramRegistry` output template is provided to easily define collections of histograms of various dimensions produced by a given task, accompanied by a more generic `OutputObj` that allows to wrap any ROOT-serializable object. Arguments of the main callback are parsed as data subscriptions, and can be used to declare various in-place operations such as joining, grouping and filtering. Joining merges column lists of compatible tables, to allow accessing, for example track covariance matrix information in addition to the basic track information. Grouping allows users, by relying on pre-defined index, to access parts of the grouped table (for example the track information table) that correspond to certain rows in the grouping table (for example the collisions table). This is possible for any tables, related by index, if the index column of the table is sorted. Filtering is an operation that exposes to the user only those rows in a table that pass certain criteria. It leverages the bulk operations mechanism, described in the next section. All these operations are zero-copy, only the user-facing wrapper object is modified in each case.

A `Configurable` template is provided to define externally-configurable parameters, ranging from single values of simple types, to one- and two-dimensional arrays and even C++ structures. AliHyperloop provides a visual interface for building the workflow configurations, and means to compare it between different train runs. Naturally, workflow configurations can be serialized using JSON.

The user analysis of input data is defined in stages, using individual tasks as building blocks. Each stage consumes input tables and produces derived data for the next stage in form of tables and analysis objects. For a workflow to be correctly formed, all input requirements need to be satisfied. This approach provides natural pipelining of data processing. The web-based analysis tools automatically adds service tasks at certain stages, that provide certain common information like particle identification (PID) decisions or track selection flags, when deploying a workflow.

2.2 Bulk operations and declarative analysis features

In LHC Run 3, one of the big challenges for ALICE computing is represented by the constraints on disk-space resources for storage. The new AOD data format provides a significant reduction in required storage in a number of ways. Run 3 AODs are based on the ROOT

```
1  namespace o2::aod {
2  namespace track {
3  // declaration of expression-based column - this is defined internally and created automatically,
4  // it is only present in the listing to illustrate the expression DSL
5  DECLARE_SOA_EXPRESSION_COLUMN(Eta, eta, float, //! Pseudorapidity
6          -1.f * nlog(ntan(0.25f * static_cast<float>(M_PI) - 0.5f * natan(aod::track::tgl))));
7  }
8  using TracksWithCovariance = soa::Join<aod::Tracks, aod::TracksCov>;
9  }
10
11 // declaration of analysis task
12 struct Task {
13   // configurable parameters can be provided, for example, via command line or a JSON file
14   Configurable<float> ptlow{"ptlow",0.5f,""};
15   Configurable<float> ptup{"ptup",2.0f,""};
16   // filters, that refer to the same table, will be automatically logically combined with &&
17   Filter ptFilter_a = aod::track::pt > ptlow;
18   Filter ptFilter_b = aod::track::pt < ptup;
19   Filter posZfilter = nabs(aod::collision::posZ) < 10.0f;
20   // HistogramRegistry allows to define framework-managed output histograms
21   // Configurable parameters can be used to define histogram axes binning
22   HistogramRegistry registry{
23     "registry",
24     {
25       {"eta_vs_Z", ";#eta;Z_{vtx};entries",
26                           {HistType::kTH2F, {{101, -2.01, 2.01}, {101, -10.1, 10.1}}}}
27     }
28   };
29   // this process signature requests filtered collisions, one by one
30   // and corresponding tracks, joined with extra information, and grouped by collisions,
31   // tracks are filtered as well - the filters, defined above, are automatically applied
32   // to compatible tables marked with soa::Filtered<>
33   void process(soa::Filtered<aod::Collisions>::iterator const& collision,
34                soa::Filtered<TracksWithCovariance> const& tracks,
35                aod::BCs const&)
36   {
37     // get bunch crossing information through index connection
38     LOGF(INFO,"BC = %ll", collision.bc().globalBC());
39     for (auto& track : tracks) {
40         // 'track' can be used to access different track properties, available in Tracks
41         // and TracksCov tables
42         // registry provides fast access to managed histogramss
43         registry.fill(HIST("eta_vs_Z"), track.eta(), collision.posZ());
44     }
45   }
46 };
```

Listing 1: An example of an analysis task definition, showcasing various declarative and imperative analysis features: data subscription with join and grouping, definition of expression-based table extensions, definition of configurable parameters, definition of expression-based filters, using table row accessors in a for loop, accessing connected table information through index. More detailed examples are available in O^2 tutorials, under `Analysis/Tutorials/src` in main repository[3].

tree format that provides a good compression ratio. Additionally, floating point values are truncated to their respective uncertainties. Finally, only a minimal set of values is stored, while the others are calculated on-the-fly based on the analysis needs. This leads to reduction in the AOD size in terms of disk space by a factor of five or better, as measured with converted Run 1/2 data. Still, it is estimated that one month of Pb–Pb data-taking would lead to about 4 PB of AODs. This means that every analysis will require additional steps to "unpack" the AOD, and extend it by including the additional variables. Two approaches are used in this regard. Tables allow one to define so-called *dynamic columns* that are C++ lambda functions tied to a table row. Their argument list can be fully or partially bound to other material columns defined in the same table. From the end-user perspective, dynamic

columns are accessed in exactly the same way as material (i.e. with values stored in memory) columns and their definition can also include external arguments as function parameters. A Domain Specific Language (DSL), based on C++ expressions, is defined and used to declare recipes for materializing new columns calculated from pre-existing ones using Arrow's Gandiva sub-project [14]. From the user perspective, it provides a way to define new columns directly with formulas written in C++, as shown in lines 4-5 in Listing 1. Dynamic columns do not correspond to any memory and their values are calculated on demand, by calling a user-defined function, and thus have only CPU cost. In contrast, expression columns are pre-calculated once per timeframe and become available to the full workflow in shared memory. CPU cost is thus shared between all the tasks, however memory usage may be quite significant, especially for long tables such as the track table. This approach is preferable in cases where the calculation involves high CPU cost functions such as trigonometrical functions.

For some values that are commonly used by most analyses, such as track momentum components, these recipes are predefined in the data model. At workflow deployment, the DPL provides a hook to insert automatically common data processors (e.g. file readers), that may be required by a particular workflow. A special data processor dedicated to applying the materialization recipes is also added if the corresponding tables are requested. These recipes, dubbed expression columns, can also be defined by users and combined into table extensions. These will not be created automatically, unless a user specifically requests their materialization. Such extensions can either be local for the particular user task or made available to the whole workflow through shared memory, in which case they need to be predefined. Another application of the expression engine is related to operations of filtering and partitioning of the tables. Boolean-valued expressions can be used to define conditions that are either applied to input data upfront for prefiltering, or to define subtables, which are available as separate entities in addition to full input collection. Internally, expression-based recipes are translated into a tree representation, which also allows the Framework to inject values of configuration parameters when they become available at run time. These representations are later translated to Gandiva, which converts them into executable code using just-in-time compilation. This code is then applied in a single pass to a given table, one time per timeframe. Users are encouraged to offload common operations to expression-based mechanisms as much as possible to guarantee efficient vectorized processing. A code example, taken from the O^2 tutorials, that showcases various declarative and traditional analysis tools is provided in Listing 1.

A dedicated declarative tool to iterate over *combinations* of analysis entities, such as tracks, has been developed to limit the use of CPU-consuming double or triple "loops" and avoid the memory cost of creating a combinations table. Another important benefit of using Apache Arrow as an in-memory storage format for the data is its interoperability. Arrow is used in many modern data analysis tools and machine learning packages and its usage opens a lot of possibilities in terms of post-processing of the O^2 outputs and integration of those external tools in user analysis code.

3 Future developments and conclusions

Since it appearance in 2019, the DPL-based O^2 Analysis Framework underwent a rapid development during 2020 and it now provides the tools to reproduce a large fraction of analyses using Run 1 and 2 data converted into Run 3 AOD format. By the end of 2021, the O^2 framework is expected to be used also for new Run 1 and 2 analyses using converted data, as it will guarantee a more efficient use of disk space and CPU resources. Most of the legacy analysis code already can be easily ported in O^2 with minor changes, with exception of some special cases. This will also encourage the development of new analysis code that exploits

the declarative capabilities of the framework. The data model will be further expanded, introducing more advanced grouping and search operations, and additional index functionality like self-indexing tables. More column types and more ways to express complex relations between tables will be also added. The expression DSL will be extended with more mathematical functions and conditional expressions will be added. Thanks to these developments, complex algorithms will be implemented with few lines of code, ensuring that the hardware capabilities will be leveraged. The combinations engine is also being extended, to allow, for example, efficient event mixing required by correlation analyses.

Additionally, the utilities that automatically attach tasks/wagons will be expanded. User-defined bulk data processors for common analysis tasks like PID, centrality estimation and others will be declared as service tasks, ensuring they are added to the workflow when needed, and that they are dynamically configured such that only required outputs are produced.

In the future, the activities will focus on the optimization and development of new declarative analysis features based on analysis requirements. The integration of machine learning tools in the framework, for both training and application of models within a user-defined analysis task, is already underway.

Last but not least, optimization for multi-core computing environments is is ongoing, in order to fully exploit available hardware, in particular the dedicated Analysis Facility.

References

[1] The ALICE Collaboration, Journal of Instrumentation **3**, S08002 (2008)
[2] P. Buncic, M. Krzewicki and P. Vande Vyvre, CERN-LHCC-2015-006.
[3] *ALICE Online-Offline software framework, O^2, main repository*, https://github.com/AliceO2Group/AliceO2 (2021), accessed: 2021-05-10
[4] Mikolaj Krzewicki et al, J. Phys.: Conf. Ser. **898** 032055 (2017)
[5] M Al-Turany et al, J. Phys.: Conf. Ser. **513** 022001 (2014)
[6] *Apache Arrow, a cross-language development platform for in-memory analytics*, https://arrow.apache.org/ (2021), accessed: 2021-02-24
[7] R. Brun, F. Rademakers, Nucl. Inst. Meth. **A389**, 81 (1997)
[8] G. Eulisse, P. Konopka, M. Krzewicki, M. Richter, D. Rohr, S. Wenzel, EPJ Web of Conferences **214**, 05010 (2019)
[9] M. Zimmermann and J.F. Grosse-Oetringhaus (for the ALICE collaboration), J. Phys.: Conf. Ser. **608**, 012019 (2015)
[10] *React: A JavaScript library for building user interfaces*, https://reactjs.org/ (2021), accessed: 2021-02-24
[11] J. Balcas, D. Kcira, A. Mughal, H. Newman, M. Spiropulu and J. R. Vlimant, J. Phys. Conf. Ser. **898** (2017) no.9, 092055 doi:10.1088/1742-6596/898/9/092055
[12] K.Schwarz, EPJ Web Conf. **214**, 08027 (2019) doi:10.1051/epjconf/201921408027
[13] A. Gheata, PoS **ACAT08**, 028 (2009) doi:10.22323/1.070.0028
[14] *Gandiva: A LLVM-based Analytical Expression Compiler for Apache Arrow*, https://arrow.apache.org/blog/2018/12/05/gandiva-donation/ (2018), accessed: 2021-02-24