Persistifying the complex event data model of the ATLAS Experiment in RNTuple

Alaettin Serhan Mete[1,*], Marcin Nowak[2], and Peter Van Gemmeren[1]

On Behalf of the ATLAS Computing Activity[†]

[1]Argonne National Laboratory[‡], Lemont, IL, United States
[2]Brookhaven National Laboratory, Upton, NY, United States

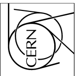
E-mail: amete@anl.gov, mnowak@bnl.gov, gemmeren@anl.gov

Abstract. The ATLAS experiment at CERN's Large Hadron Collider has been using `ROOT TTree` for over two decades to store all of its processed data. The ROOT team has developed a new I/O subsystem, called `RNTuple`, that will replace `TTree` in the near future. `RNTuple` is designed to adopt various technological advancements that happened in the last decade and be more performant from both the computational and storage perspectives. On the other hand, `RNTuple` has limited/streamlined data model support compared to `TTree`.

The ATLAS Event Data Model (EDM) must support functionality arising from the vast complexity of the underlying detector and the constraints of the computing model. It takes advantage of `C++` (object oriented) language features that allow efficient processing of highly complex algorithms that produce physics objects from various different sub-detectors. To encapsulate this complexity needed for transient processing, ATLAS had introduced a separation between the transient and the persistent (T/P) representations of the EDM. This approach simplified the adoption of `TTree` as the main event data format at the time. It also allows us to embrace different technologies and storage backends more easily while keeping the reconstruction and simulation software stack as complex as it needs to be.

In this presentation, we will discuss all the foundational work that allowed ATLAS to persistify all its processed event data, including complex simulation and reconstruction data, in the `RNTuple` format. We will discuss the key elements of ATLAS' core EDM and I/O software and how encapsulation via T/P separation can guide other (future) experiments in designing their own models and future-proofing their I/O and storage infrastructure.

ATL-SOFT-PROC-2024-002
22 July 2024



[*]Speaker

[†]Copyright 2024 CERN for the benefit of the ATLAS Collaboration. CC-BY-4.0 license

[‡]Argonne National Laboratory's work was supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357

1 Introduction

ATLAS [1] is a general-purpose experiment at CERN's Large Hadron Collider (LHC) and Athena [2] is the open-source software framework of ATLAS. Athena is based on Gaudi [3] and consists of about 4 million lines of C++ and 1.5 million lines of Python code.

The typical ATLAS data processing chain consists of multiple steps as shown in Fig. 1. All ATLAS data, with the exception of RAW, are stored in ROOT [4] files. Broadly speaking, in-file data are split into two main categories as metadata and event data. The former contains information that describes file-level information, e.g., detector description/conditions tags, while the latter contains information about the collision events, e.g., tracks, electrons, muons etc.

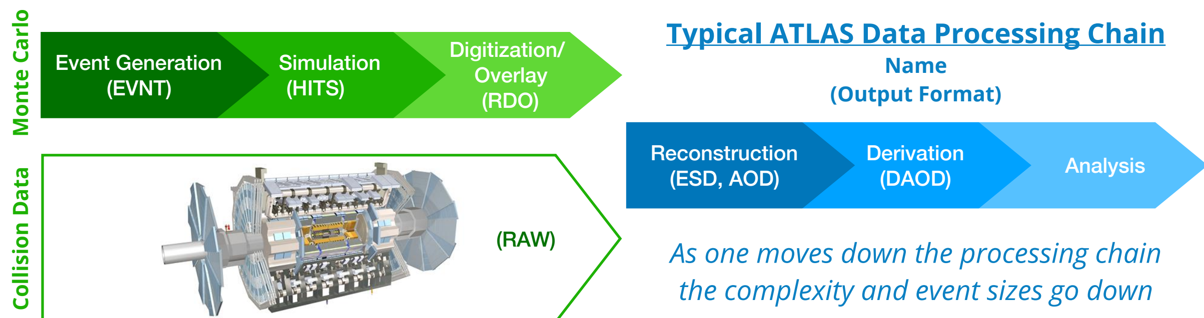


Figure 1: The typical ATLAS data processing chain.

Since the beginning of Run 1, ATLAS has been using ROOT TTree to store its in-file metadata and event data. The ROOT team is currently developing a new Input/Output (I/O) sub-system, called RNTuple , that is going to replace TTree starting with Run 4 (i.e., currently scheduled for 2029 onward). RNTuple adopts a more modern and efficient approach that utilizes the latest C++ language features and other improvements such as parallel and asynchronous I/O etc. More importantly for the experiments, it is expected to improve data throughput and minimize the file size footprint compared to TTree.

Despite its potential benefits, RNTuple also comes with certain limitations compared to TTree. For example, it does not support raw pointers, polymorphism etc. Therefore, it is not a drop-in replacement in TTree and depending on the complexity of an experiment's event data model (EDM) adopting it might come with a number of challenges.

In the next section, we will discuss ATLAS' EDM and what it would require to persistify it using any storage backend.

2 The ATLAS Event Data Model

The ATLAS detector consists of many complex sub-systems. For successful data processing all of them have to perform efficiently in unison. Therefore, it is extremely important to ensure that we have common interfaces and data objects across the experiment that are defined in coherent software that is easy to maintain over many decades. The latter is extremely important for any experiment whose lifetime spans multiple decades because writing/reading data in a consistent manner over such a long time requires not only backward compatibility but often forward compatibility as well.

To put it briefly, the ATLAS Event Data Model (EDM) provides a collection of C++ classes that define detector/physics objects to allow streamlined and efficient processing of highly complex algorithms. When necessary, it relies on advanced C++ concepts and data structures to accomplish these goals. For example, silicon hits in the inner detector and detector hits in the muon spectrometer are fitted to make Tracks, energy deposits in the calorimeter cells are grouped to make Clusters, and all of these can be combined to construct higher-level objects such as Electrons, Photons, and Muons. All these objects have their associated classes in the ATLAS EDM.

Part of the ATLAS EDM is separated into two as transient and persistent. The transient data model is used during data processing and is the in-memory representation of the objects. The persistent data model, on the other hand, is the on-disk representation of the same data and is the form that is used to store data permanently. This EDM separation is often called Transient/Persistent (T/P) separation. In almost all cases, the persistent data model is simpler than the transient one. This separation allows one to retain the complexity of the transient data that is necessary to have highly efficient data processing

while it allows one to prune the persistent data to only keep the most basic and necessary information to save storage footprint. It also allows having multiple versions of the persistent data throughout the lifetime of the experiment, a concept called schema evolution, while the data processing always uses the newest transient data. The benefit of simplicity, flexibility, and performance, however, comes at the price of having to write and maintain more code. This is because in a T/P separated EDM one has to have dedicated converters that do the translation, and possibly multiple versions of the persistent data. An example is shown in Fig. 2.

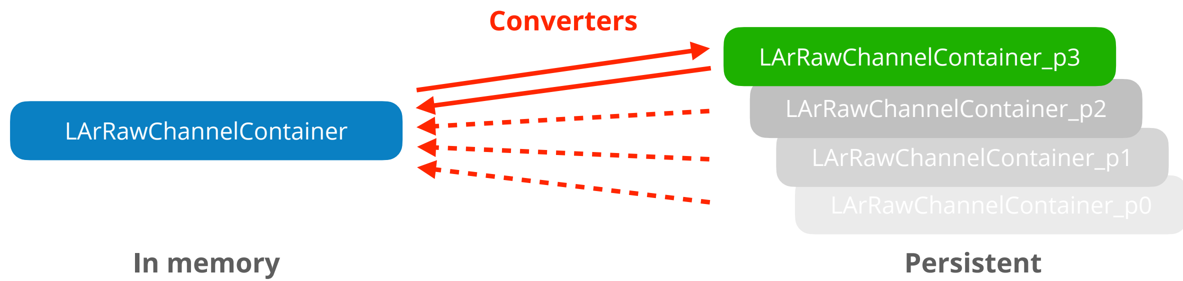


Figure 2: An example for the T/P separation.

During Run 1, when TTree's EDM support was not as well developed as it is now, the ATLAS EDM used a fully T/P separated model. This allowed us to hide the `C++` complexity from the storage side and achieve schema evolution. Parts of the EDM were re-written during the long shutdown between Run 1 and Run 2. This re-write primarily targeted unifying reconstruction and analysis data formats, and the associated EDM is now referred to as the `xAOD` EDM. The `xAOD` EDM is designed to be simpler, so that it doesn't need T/P separation, and adopts a versioning approach for schema evolution. There is, however, a separation between the interface classes (what user interacts with) and the payload (where the data are stored). Most of the underlying data are stored as (nested) vectors of fundamental types, primarily single-precision floating point numbers. Some data are part of the class definition. These are referred to as static variables. Others can be added on demand on the fly at any point during processing. These are referred to as dynamic variables. As previously mentioned, the `xAOD` EDM is primarily used for reconstruction and downstream workflows, whereas the original EDM is still used in the upstream workflows as shown in Fig. 1.

The next section describes the core concepts of the I/O system in `Athena` that glues the EDM to the underlying storage backend.

3 The ATLAS Input/Output System and ROOT

The I/O system in `Athena` , namely `Athena POOL` Replacement (`APR`), is derived from the common `LCG` persistency project `POOL` [5]. The data storage is broken down into a structured hierarchy as shown in Fig. 3.

Figure 3: The `APR` hierarchy.

For each event, objects belonging to the same category/group are placed in containers, e.g., `Electrons` in an `ElectronContainer`, which are then placed in a `Database`, e.g., a file. The application programmable interface (API) hides the technology specific implementation of the underlying storage service. So, in a nutshell, `APR` serves as an abstraction layer that sits between the EDM and the storage layer.

Since the beginning of data taking, ATLAS has used `ROOT`'s `TTree` to store its event data and in-file metadata. From a technical perspective, this means having a `ROOT` storage service that contains and implements:

- `RootDatabase` that handles `ROOT` file-level operations, i.e., opening/closing the relevant `TFile`, and

- TreeContainer that handles TTree -level operations, i.e., creating/filling TTree/TBranch etc.

The most important aspect of this implementation is that the ROOT API is completely isolated from the EDM, meaning the framework/EDM does not rely on the ROOT I/O API apart from the storage service. Therefore, it is enough to provide a new Database and Container implementation for each new storage backend, as long as said backend fulfills a set of criteria.

As mentioned earlier, the ROOT team is currently developing a new I/O sub-system, called RNTuple , that is going to replace TTree starting with Run 4 (i.e., 2029 onward). Among other improvements, RNTuple is expected to increase data throughput and minimize the file size footprint compared to TTree. For ATLAS to be able to adopt RNTuple , or any other storage layer for that matter, it has to support:

- Standard Template Library (STL) containers, mainly (nested) vectors, of Plain Old Data (POD) and user types,
- User-defined classes and enumerations (i.e., C++ classes and enums),
- User-defined collection proxies and late model extension (primarily to support xAOD EDM),
- User code execution when reading objects of a given type (a feature called Read Rules), and
- A void* based API to bind the I/O later with the rest of Athena.

After a series of iterations with the ROOT team, the current RNTuple implementation supports all these features that are needed by ATLAS.

Two main earlier design choices significantly eased ATLAS' adoption of RNTuple. Firstly, a significant portion of the reconstruction EDM was already simplified prior to Run 2, i.e., xAOD EDM. In addition, the more complex parts of the EDM that are used in the upstream worklows adopts T/P separation, which hides the data complexity from the storage layer. Secondly, and arguably more importantly, the ROOT API was kept disjoint from the EDM and used only in the storage side thanks to the APR abstraction layer. As a result, the core of the work was isolated to the introduction of a new APR technology layer, namely RNTupleContainer, that implements the RNTuple relevant code that performs the actual reading/writing using the relevant ROOT RNTuple API.

One of the more complex aspects was supporting late model extensions to accommodate dynamic attributes of the xAOD EDM. Once this functionality was introduced to RNTupleWriter, the adoption on the Athena side was relatively straightforward. During the migration, T/P separation had to be introduced in a few cases. Overall, most of the heavy-lifting went into communicating all the ATLAS requirements with the ROOT team and making sure RNTuple supported all of them.

In the next section we briefly discuss early performance benchmarks of using RNTuple for ATLAS physics analysis data.

4 Performance Benchmarks

The Derived Analysis Object Data (DAOD) is the primary data format that is used in physics analyses. In Run 3, ATLAS switched to a new physics analysis model [6] where analyses use two common/inclusive DAOD formats, namely DAOD_PHYS and DAOD_PHYSLITE. These formats include all reconstructed events and all common variables for the majority of the analyses. The DAOD_PHYS includes more high-level information and targets an event size of 50 kB, whereas DAOD_PHYSLITE includes mostly analysis-level objects after various corrections/calibrations and targets an event size of 10 kB. Each format contains more than a thousand different variables over tens of different domains.

In order test the RNTuple performance, DAOD_PHYS and DAOD_PHYSLITE files are produced for 2755 proton-proton collision events, corresponding to an average interactions per bunch crossing ($\langle \mu \rangle$) of 62.9, that are recorded in 2023. Each sample is produced twice, storing the event data in a TTree once and an RNTuple in the other case. As can be seen in Fig.4, in both formats, the files where the event data are stored in an RNTuple are a little over 20% smaller than their TTree counterparts. This is indeed very promising with the current RNTuple prototype, especially given that TTree usage has been heavily optimized over the last two decades, an exercise that is yet to be performed for RNTuple. More detailed performance tests at a larger scale are being carried on at the time of writing.

5 Conclusions and Outlook

ATLAS has a full-fledged prototype that enables reading/writing all official ATLAS data formats in RNTuple. This is a very important milestone as the experiment is getting ready to switch from TTree to RNTuple as its main storage format in Run 4. However, there is still a significant amount of work ahead to be production-ready. There are a number of missing functionality that need to be addressed/implemented, including but not limited to:

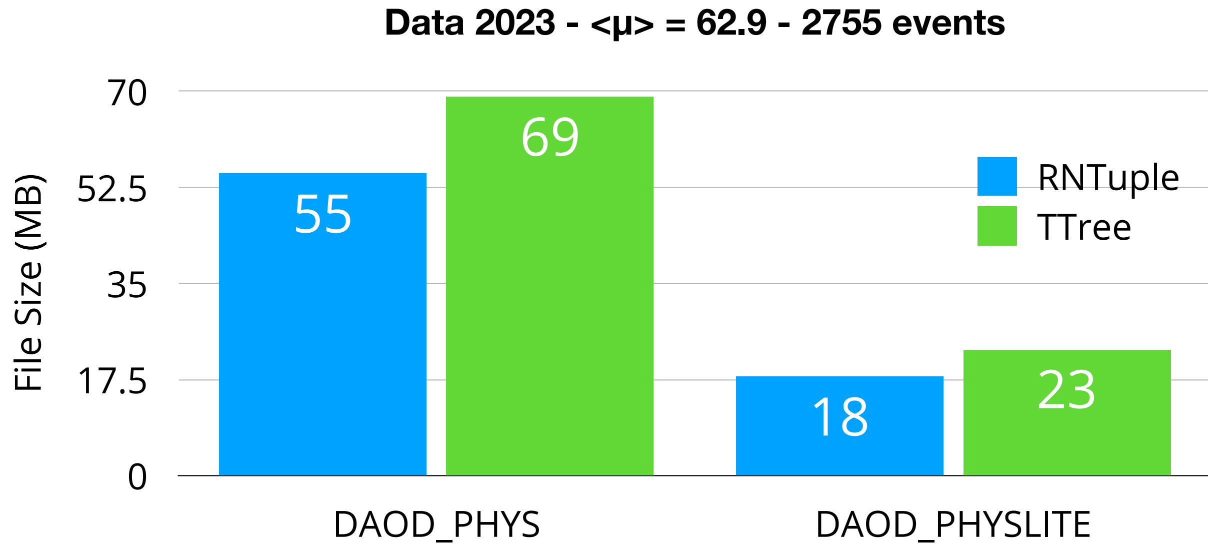


Figure 4: File size comparisons between `DAOD_PHYS(LITE)` samples with `TTree` and `RNTuple` event data.

- Fast merging of `RNTuple` objects on-the-fly and supporting custom entry/event indexing,
- Having various utility tools to peek into, compare, and validate, `RNTuple` objects,
- Having support for related `RNTuple` objects, a feature a.k.a. *friendship.*

In addition, when applicable, the storage setting parameters need to be optimized for each data format, and large-scale tests need to be performed to validate the data and ensure all official workflows producing `RNTuple` can meet the distributed computing production limitations in terms of memory and CPU usage.

All in all, ATLAS is going to adopt `RNTuple` as its main storage format beginning with Run 4. The collaboration will use the long shutdown between Run 3 and Run 4 for adopting, testing, and validating `RNTuple`. The preliminary results show that a significant reduction in file sizes can be achieved, compared to `TTree`, that is in agreement with the expectations. The ATLAS collaboration will continue working closely with the `ROOT` team to deliver a fully efficient and robust `RNTuple` in the years ahead.

References

[1] ATLAS Collaboration. (2008). The ATLAS Experiment at the CERN Large Hadron Collider. *JINST*, **3**, S08003. https://doi.org/10.1088/1748-0221/3/08/S08003

[2] ATLAS Collaboration. (2021). Athena. https://doi.org/10.5281/zenodo.4772550

[3] Barrand, G. et al. (2001). GAUDI – A software architecture and framework for building HEP data processing applications. *Comp. Phys. Comm.*, **140**, 45-55. https://doi.org/10.1016/S0010-4655(01)00254-5

[4] Brun, R. & Rademakers, F. (1997). ROOT - An Object Oriented Data Analysis Framework. *Nucl. Inst. & Meth. in Phys. Res.*, **A 389**, 81-86. https://doi.org/10.5281/zenodo.848818

[5] Duellmann, D. (2003). The LCG POOL Project, General Overview and Project Structure. arXiv:physics/0306129 [physics.comp-ph]

[6] Elmsheuser, J. et al. (2020). Evolution of the ATLAS analysis model for Run-3 and prospects for HL-LHC. *EPJ Web Conf.* **245** 06014. https://doi.org/10.1051/epjconf/202024506014