https://doi.org/10.1051/epjconf/202429506013

Integration of RNTuple in ATLAS Athena

Florine de Geus[1,*], *Javier* López-Gómez[1], *Jakob* Blomer[1], *Marcin* Nowak[2], and *Peter* van Gemmeren[3]

[1]CERN
[2]Brookhaven National Laboratory
[3]Argonne National Laboratory

Abstract. After using ROOT's TTree I/O subsystem for over two decades and storing more than an exabyte of compressed High Energy Physics (HEP) data, advances in technology have motivated a complete redesign, RNTuple, which breaks backward-compatibility to take better advantage of these storage options. The RNTuple I/O subsystem has been designed to address performance bottlenecks and other shortcomings of TTree. Specifically, RNTuple comes with an updated, more compact binary data format that can be stored both in ROOT files and natively in object stores. It is designed for modern storage hardware (e.g. high-throughput low-latency NVMe SSDs), and provides robust and easy to use interfaces. The binary format of RNTuple is scheduled to become production grade in 2024, and recently has become mature enough to start exploring the integration into software used by HEP experiments. In this contribution, we discuss the developments to support the features as required by the ATLAS analysis Event Data Model (EDM) in RNTuple, which will enable its integration into the Athena software framework. With these developments in place, we evaluate the performance of the current most recent versions of RNTuple-based ATLAS data sets and compare this to that of TTree.

1 Introduction

With the High-Luminosity LHC (HL-LHC), ATLAS expects to collect up to ten times more data than it did during the first three runs [1]. Major efforts are required to ensure that compute resources are able to keep up with this increase in data volume. Currently, physics data from all four LHC experiments, including ATLAS, is stored and accessed using ROOT's TTree I/O subsystem [2]. The TTree data format is specifically optimised for High Energy Physics (HEP) data and allows for columnar storage of both plain data (i.e. numerical values) as well as complex (custom) C++ objects and collections. However, it is not designed to make optimal use of modern data storage systems. In addition, both its implementation and API show shortcomings when it comes to safe multithreaded and GPU-driven analysis. Driven by the experiences with TTree and inspired by concepts from state-of-the-art industry standards, RNTuple [3] is currently being developed as a backwards-incompatible successor to TTree, and is anticipated to be ready for production use in the HL-LHC era. Its binary data format is designed to be compatible with ROOT's existing I/O facilities, as well as be able to natively target object stores such as Intel DAOS [4] or Amazon S3. In addition, RNTuple will come

[*]e-mail: florine.de.geus@cern.ch

with robust interfaces that should to be easy to use correctly, and support multithreaded and GPU-driven usage.

Initial performance evaluations have shown promising results compared to both TTree and industry standards [3, 5]. The logical next step of development involves ensuring RNTuple can be integrated into existing experiment workflows as seamlessly as possible, without significantly compromising on the previously observed performance gains. In this contribution, we focus on the integration of RNTuple into the ATLAS software framework, Athena [6]. In section 2, we present the features that have been added to RNTuple to support the Analysis Object Data (AOD) format and the Derived AOD formats, DAOD_PHYS and DAOD_PHYSLITE [7] as part of the analysis Event Data Model (EDM) used by ATLAS, xAOD [8]. With these features in place, we compare RNTuple with TTree for DAOD_PHYS (which is the current recommended analysis format in ATLAS) and report on these results in section 3. Finally, we discuss these results and potential next steps in section 4.

2 Supporting the ATLAS EDM in RNTuple

To make RNTuple compatible with the ATLAS EDM and to be able to integrate RNTuple into Athena, a number of new features have been added, which will be discussed in this section.

2.1 Custom collections

xAOD is in large part based on the `DataVector` class and types derived from it. `DataVector` is an Athena-specific custom container behaving like a vector of objects, but implemented as a vector of pointers to these objects, which is advantageous when representing a collection of objects as a collection of their base type. To be able to perform I/O operations on such user-defined collections, in particular in-memory object creation when reading from storage, ROOT provides an abstract interface that allows for creation of so-called *collection proxies*. With TTree, every (custom) collection class has to have an associated collection proxy.

So far, RNTuple has only supported collections where the elements are contiguous in memory, such as `std::vector` and ROOT's own `ROOT::RVec` [9], which makes it possible to avoid the use of collection proxies. To properly be able to (de)serialize custom collections for which we cannot predict the memory location of their elements, support for collection proxies in RNTuple has been added. This enables the reading and writing of `DataVector`s, but more generally any non-associative collection that provides a collection proxy. The on-disk format of collections with an associated collection proxy is identical to that of `std::vector` fields, with a mother field that stores the index offsets of each collection and a child field that holds the data elements. As a consequence, currently only proxies for custom *non-associative* collections are supported.

2.2 Support for `std::set` fields

While not present DAOD_PHYS(LITE), the full ATLAS AOD (i.e. before derivation, used as input to produce DAOD_PHYS(LITE)) also contains objects of the `std::set` type. To ensure that the EDM at any point in the production workflow can be written to and read from by RNTuple, support for `std::set`-type fields (and nestings thereof) has been added to RNTuple. While the C++ standard defines the `std::set` as an associative container, the keys in a set are equal to their value. This allows for using the same on-disk layout as for non-associative collections, which in turn means that the previously discussed support for collection proxies in RNTuple can be leveraged to get access to a given set's data members when writing to disk and to reconstruct the set when reading from disk. ROOT already provides collection proxies for most STL collections (including for `std::set`), which means these can be accessed directly by the internal implementation and users don't have to explicitly provide them (which is required for custom collections as discussed in the previous section). In turn, this allows us to add `std::set` to te RNTuple as a separate, specialized field type.

2.3 Post-read callbacks

A number of classes present in xAOD, the most prevalent among them being the `ElementLink` class, require special *I/O customization rules*, also known as *read rules*, in order to be properly initialized after reading. These customization rules may contain C++ code snippets that are able to access and – if desired – modify the data members. The rules are invoked for every entry read, after deserializing the objects from the relevant branch or field[1]. Use cases of such read rules include the initialization of transient data members and custom schema evolution. The RNTuple implementation has been extended to load and process such read rules, currently limited to modifying transient class members only. This is sufficient for reading `ElementLink` objects.

2.4 Late model extensions

Another feature specific to xAOD are types referred to as *dynamic attributes*. Dynamic attributes are data members whose presence in an xAOD object is optional and determined only at runtime by the derivation application, based on the data content of each processed event independently, and can be found in particular in DAOD_PHYS(LITE). This implies that at the start of writing, when defining the RNTuple model, it is not yet known exactly which dynamic attribute fields should be present.

To accommodate for the addition of dynamic attributes after the model has been defined, we introduce *late model extension* to RNTuple. Contrary to the corresponding feature in TTree, extending an existing RNTuple model only updates the on-disk metadata, i.e. previously written entries are left untouched. Instead, a default zero-initialized value for the added field will be used as a placeholder when reading these entries. The interface for late model extension is demonstrated in Figure 1.

3 Evaluation of RNTuple for DAOD_PHYS

In this section, we evaluate the storage efficiency and read throughput of TTree and RNTuple for DAOD_PHYS. We evaluate a number of different compression methods and storage media, presented in Table 1. We use two sets of data samples. The first sample contains 180k Run 2 Monte Carlo (MC) events, and 1789 branches/top-level fields. The second sample contains around 210k Run 2 collision data events and 1312 branches/top-level fields.

The TTree-based benchmark data samples are prepared using ROOT's `hadd`, which allows for recompressing the original data samples with the different compression methods mentioned in Table 1. We use `hadd`'s "`-O`" flag to ensure that the basket size of each recompressed TTree is optimized for the specified compression method. The RNTuple-based benchmark data samples are prepared by converting the previously prepared TTree samples into RNTuples using the `RNTupleImporter` utility that will be available alongside RNTuple

[1] One way to define such read rules is by defining a `#pragma read` directive in the `LinkDef` header of the class, which specifies the data members to read from and write to.

```
auto model = RNTupleModel::Create();
auto fieldFloat = model->MakeField<float>("myFloat");

auto ntuple = RNTupleWriter::Recreate(std::move(model), "myNTuple", "myNTuple.root");
*fieldFloat = 42.0;
ntuple->Fill();

auto modelUpdater = ntuple->CreateModelUpdater();
modelUpdater->BeginUpdate();
auto fieldVec = std::make_shared<std::vector<float>>();
modelUpdater->AddField<std::vector<float>>("myVector", fieldVec.get());
modelUpdater->CommitUpdate();

*fieldFloat = 48.0;
*fieldVec = {1., 2., 3.};
ntuple->Fill();
```

Figure 1. RNTuple interface for late model extension. The contents of the `myVector` field will be empty in the first entry.

Table 1. Overview of the DAOD_PHYS benchmarking phase space.

Storage format	TTree, RNTuple
Compression method	LZ4, zstd[1], LZMA (level 1), LZMA (level 7)
Storage medium	SSD (NVMe), HDD, RAM (tmpfs), XRootD

[1] Current ATLAS default compression method

in ROOT7. The contents of the converted RNTuples are validated against the original TTrees by verifying that the all RNTuple fields have a corresponding TTree branch and vice versa, and by verifying that the histograms of a semi-randomly[2] selected set of fields are equivalent between TTree and RNTuple, as well as across compression methods.

3.1 Storage efficiency

Figure 2 shows the average event size for the different compression methods mentioned in Table 1, both for the MC samples and the Run 2 data samples, respectively. The average event size is determined by dividing the total number of compressed bytes (excluding bytes not linked to any specific event, such as the header and footer in the case for RNTuple) by the number of events in the data sample. We see that the average event size for the MC samples is slightly larger than for the data samples. This can be explained by the additional MC information present in these events, as well as potentially more pile-up. However, we also observe that the RNTuple-based MC samples have a higher storage efficiency ratio with regard to TTree, which in turn results in the average event size approaching that of the RNTuple-based data samples. Further evaluation with more DAOD_PHYS (as well as DAOD_PHYSLITE) samples is required to better understand which factors affect the RNTuple storage efficiency for DAOD_PHYS files, but based on the current measurements, the relative storage reduction achieved by RNTuple for the different compression methods appears to correspond with earlier observations using different benchmark data sets [3].

[2]For compatibility reasons, we only consider `std::vector<float>` for this part of the validation.

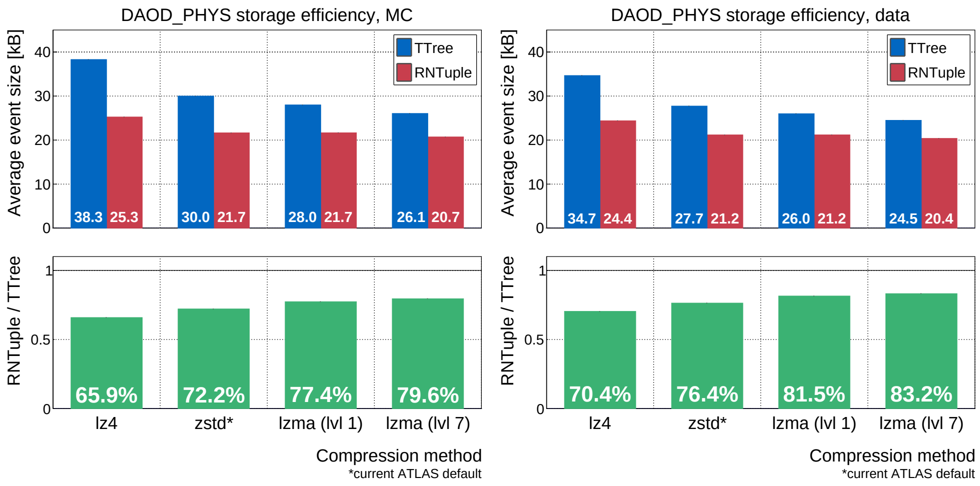


Figure 2. RNTuple and TTree-based DAOD_PHYS storage efficiency for different compression methods. Left shows the average event size for the Monte Carlo samples, right for the data samples.

3.2 Read throughput

We evaluate the read throughput using an artificial event loop written in ROOT's RDataFrame [10]. This event loop reads 32 branches or fields containing particle data, all of which are of type `std::vector<float>`, and fills a histogram for each branch or field. No other processing of the data takes place, with the goal of minimising the time spent by the CPU on non-I/O related tasks. The setup used for performing the benchmarks as described here is shown in Table 2. We use a release build (compiler flag `-O3`) of ROOT[3], with asynchronous I/O through `io_uring` [11] enabled.

Table 2. Overview of the read throughput benchmarking setup.

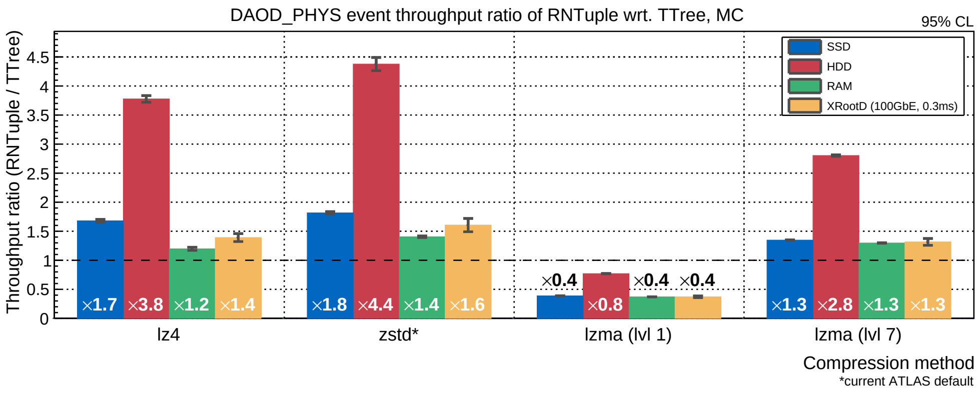
CPU	AMD APYC 7702P, 2GHz
RAM	128GB DDR4 RDIMM, 3200MHz
SSD	Samsung MZWLJ3T8HBLS-00007
HDD	TOSHIBA MG07ACA14TE SATA, 7200 RPM
Network	100 Gbit/s Ethernet[1]
OS	AlmaLinux 9.1, Linux kernel 6.3 from ELRepo[2]

[1] XRootD benchmarks use the `projects.cern.ch` EOS instance, in the same datacenter.
[2] RHEL9's kernel currently does not enable `io_uring`.

The event loop as described above is used to measure the performance of different compression methods on the storage media described in Table 1, for both TTree and RNTuple. Performance is measured both in terms of the number of uncompressed bytes read from the given storage medium per second (*raw I/O throughput*), and the number of events processed per second (*event throughput*). The I/O throughput will be independent of the compression method, while for the event throughput, we expect to observe different results for the different

[3]https://github.com/enirolf/root/releases/tag/chep23

methods of compression, since part of the event processing involves decompression of data. Taking into consideration both kinds of throughput moreover allows for determining whether the benchmark runs are CPU bound or I/O bound. For SSD, HDD and RAM, each benchmark is repeated ten times, with the page cache being cleared between each repetition for SSD and HDD to ensure that the data is actually read from disk each time. Due to a non-negligible variation in latency in XRootD, the benchmarks in this case are repeated twenty times and outliers are removed (again with the page cache cleared between runs).



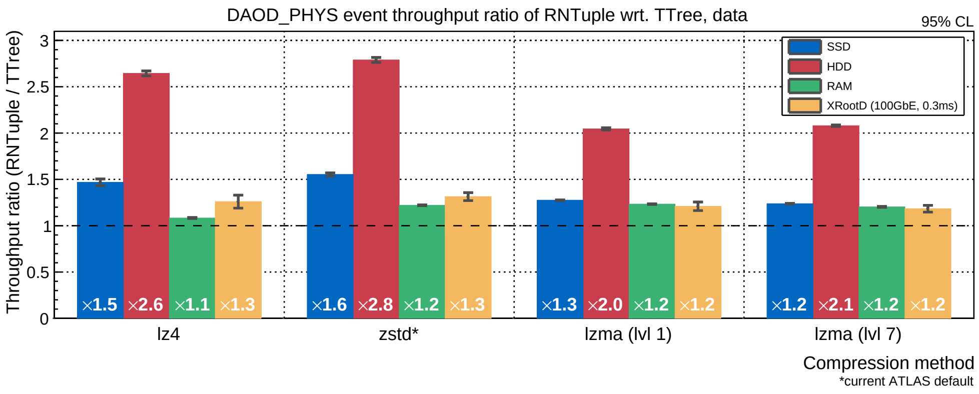


Figure 3. RNTuple-based DAOD_PHYS event throughput speed-up compared to TTree-based DAOD_PHYS, using different compression methods and storage media. Above shows the average throughput speedup for the Monte Carlo samples, below for the data samples.

Figure 3 shows the event throughput ratio of RNTuple compared to TTree for the compression methods and storage media included in the benchmarks. The difference in relative speedup when reading from SSD compared to HDD can be explained by the fact that performance is CPU bound for the benchmarks run using SSD, likely caused by decompression. This is further demonstrated in Figure 4 and Figure 5, which show that for the MC sample, the raw I/O throughput ratio between TTree and RNTuple is reasonably comparable between both media However the event throughput ratio between TTree and RNTuple for all compression methods (except for LZMA with level 1, which will be further investigated) is significantly higher when reading from HDD than it is when reading from SSD. Similar observations are made for the data sample.

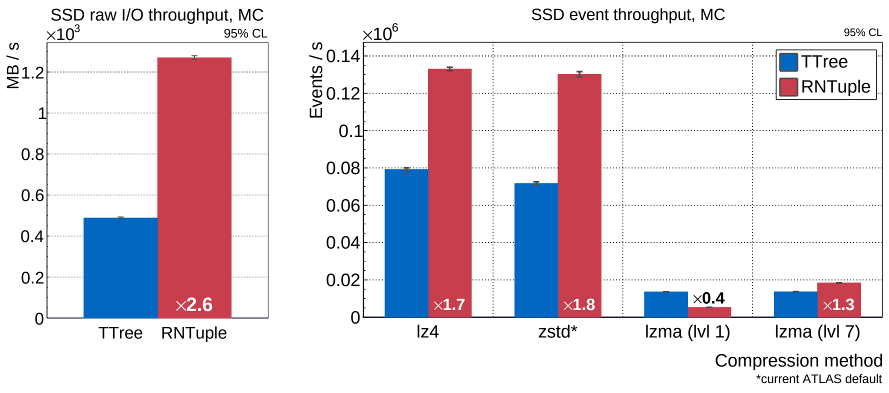


Figure 4. DAOD_PHYS SSD throughput. Left shows the raw I/O throughput for TTree and RNTuple. Right shows the event throughput for TTree and RNTuple using different compression methods.

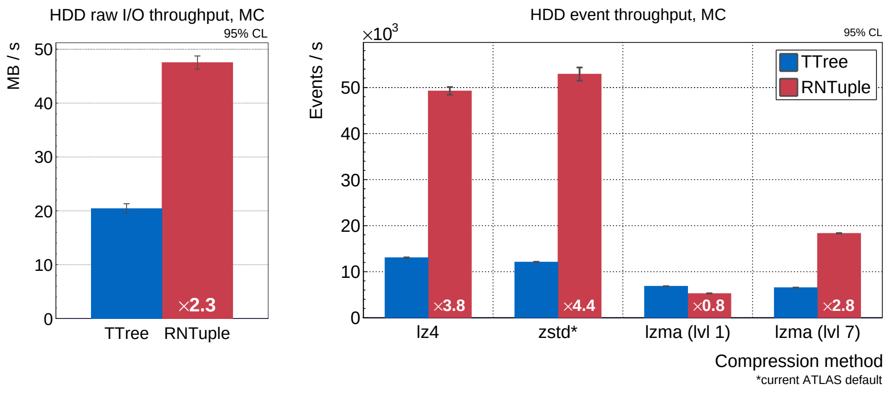


Figure 5. DAOD_PHYS HDD throughput. Left shows the raw I/O throughput for TTree and RNTuple. Right shows the event throughput for TTree and RNTuple using different compression methods.

4 Discussion

In this contribution, we presented the efforts to support the ATLAS analysis EDM, xAOD, in ROOT's new I/O subsystem, RNTuple. These efforts entail adding support for fields containing custom collection-type and `std::set` objects, partial support for I/O customization rules and late model extensions. With these additions, the reading and writing of AOD, DAOD_PHYS and DAOD_PHYSLITE data sets in the offline analysis framework used by ATLAS, Athena becomes possible.

We have compared the performance of RNTuple with respect to ROOT's TTree I/O subsystem that is currently used in Athena using DAOD_PHYS samples originating from both Monte Carlo simulation and detector data. In general, we see that RNTuple outperforms TTree both in terms of storage efficiency and read throughput. These results are in line with previous observations using different data formats. When taking into consideration both storage efficiency and read throughput, the current default compression method used by ATLAS for TTree-based DAOD_PHYS(LITE) data files, zstd, also performs best with RNTuple.

Future work includes the evaluation of more data samples using both Monte Carlo and detector data, in order to further understand the how the composition of these samples affects performance. One observation to further investigate is the decreased event throughput for LZMA level 1 of RNTuple compared to TTree for the MC. Moreover, we want to study the performance of different access patterns using more (representative) analysis workflows. In addition to DAOD_PHYS, we intend on evaluating DAOD_PHYSLITE in anticipation for its official release and use in the HL-LHC era.

In conclusion, the results shown here make the benefits of using RNTuple for xAOD apparent. However, we will need to continuously monitor how any future addition or change will affect performance, be it positively and negatively, as RNTuple evolves further into a production-ready state.

References

[1] ATLAS Collaboration, Tech. Rep. CERN-LHCC-2022-005, LHCC-G-182, CERN, Geneva (2022), `http://cds.cern.ch/record/2802918`
[2] R. Brun, F. Rademakers, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **389**, 8186 (1997)
[3] J. Blomer, P. Canal, A. Naumann, D. Piparo, EPJ Web of Conferences **245**, 02030 (2020)
[4] J. López-Gómez, J. Blomer, EPJ Web of Conferences **251**, 02066 (2021)
[5] J. Lopez-Gomez, J. Blomer, Journal of Physics: Conference Series **2438**, 012118 (2023)
[6] ATLAS Collaboration, *Athena* (2021), "athena [software]", `https://zenodo.org/record/4772550`
[7] J. Elmsheuser, C. Anastopoulos, J. Boyd, J. Catmore, H. Gray, A. Krasznahorkay, J. McFayden, C.J. Meyer, A. Sfyrla, J. Strandberg et al., EPJ Web of Conferences **245**, 06014 (2020)
[8] A. Buckley, T. Eifert, M. Elsing, D. Gillberg, K. Koeneke, A. Krasznahorkay, E. Moyse, M. Nowak, S. Snyder, P.v. Gemmeren et al., Journal of Physics: Conference Series **664**, 072045 (2015)
[9] ROOT, *RNTuple reference specification* (2023), `https://github.com/root-project/root/blob/44efaba459d269497c6e02e1a9fefaebe018a922/tree/ntuple/v7/doc/specifications.md`
[10] D. Piparo, P. Canal, E. Guiraud, X.V. Pla, G. Ganis, G. Amadio, A. Naumann, E. Tejedor, EPJ Web of Conferences **214**, 06029 (2019)
[11] J. Axboe, *liburing library for io_uring* (2023), "liburing" [software], `https://github.com/axboe/liburing`