RNTupleInspector: A storage information utility for RNTuple

Florine Willemijn de Geus[1,2], **Jakob Blomer**[1], **Philippe Canal**[3] **and Vincenzo Eduardo Padulano**[1]

[1]CERN, Geneva, Switzerland

[2]University of Twente, Enschede, The Netherlands

[3]Fermilab, Chicago (IL), U.S.A.

E-mail: florine.de.geus@cern.ch

Abstract. Inspired by over 25 years of experience with the ROOT TTree I/O subsystem and motivated by modern hardware and software developments as well as an expected tenfold data volume increase with the HL-LHC, RNTuple is currently being developed as ROOT's new I/O subsystem. Its first production release is foreseen for late 2024, and various experiments have begun working on the integration of RNTuple with their existing software frameworks and data models. To aid developers in this integration process, and to help them further understand and monitor the storage patterns of their data with RNTuple, we have developed the RNTupleInspector utility interface, which will be available with every ROOT installation that includes RNTuple. The RNTupleInspector provides storage information for full RNTuples as well as specific fields or columns, and is designed in such a way that it can be used as part of a larger monitoring tool as well as in an exploratory setting, for example through the ROOT C++ interpreter. In this contribution, we will discuss the motivation and design considerations behind the RNTupleInspector and demonstrate various use cases through examples.

1 Introduction

In anticipation of a tenfold data volume increase during the High-Luminosity LHC (HL-LHC) [1], RNTuple, ROOT's next-generation columnar data format and I/O subsystem [2], is currently in the final design and development phase before its first production release. As a consequence, experiments have started working on the integration of RNTuple into their core data production frameworks. At the time of writing, all official ATLAS experiment data products can be written and read in the RNTuple format through their data production framework, Athena [3]. A similar effort with CMS is currently ongoing.

Successful adoption of RNTuple, both into an experiment framework and on the end-users side, requires a solid understanding of its behaviour. Here, *behaviour* can be subdivided into two categories: *runtime* behaviour and *static* behaviour.
Runtime behaviour concerns the performance of RNTuple during the lifetime of a particular application execution. Relevant metrics include overall runtime, I/O performance and memory footprint. Obtaining representative results for this type of behaviour typically requires tailored, experiment-specific

benchmarks. However, some general performance metrics can be obtained from RNTuple through the `EnableMetrics` method, callable from both the `RNTupleWriter` and the `RNTupleReader`.
In contrast to runtime behaviour, static behaviour pertains to the on-disk storage of an RNTuple, and the impact of different parameters which can be specified during writing. Relevant metrics include on-disk size on various levels (total, per-field, per-column, per-cluster, per-page) and the compression factor (again, on the aforementioned levels). Obtaining these metrics in principle require less experiment-specific benchmarks, except for some potential specific measurements dictated by the experiment's event data model (EDM), which defines the structure of the data.
To help collect static behaviour measurements, we have developed the `RNTupleInspector`. It is a utility interface, available with any ROOT release starting from version 6.32.00[1]. In the subsequent sections of this paper, we will discuss the design principles of the `RNTupleInspector`, demonstrate a selection of its features, and discuss which features are still foreseen to be added.

2 Design principles of the RNTupleInspector

The primary goal behind the `RNTupleInspector` is to guide core experiment software experts towards optimal use of the RNTuple data format, and by extension, optimal use of available storage resources. The intended audience is diverse. In addition, we foresee the `RNTupleInspector` to be used by developers working on various other (auxiliary) software projects that use RNTuple. Moreover, the `RNTupleInspector` has proven to be useful in the internal development of RNTuple as well. An initial version of the `RNTupleInspector` has been used, for example, to compare the storage efficiency of the ATLAS `DAOD_PHYS` data format between RNTuple and TTree [4].

The `RNTupleInspector` is designed to be consistent with RNTuple's data format specification [5]. In brief, this data format is organized as follows: the RNTuple schema represents a (hierarchical) collection of *fields*. A field (and possibly its subfields) represents a C++ type or class, or a collection thereof. In the on-disk representation, fields are stored as *columns* of fundamental types. Each column is split into *pages*, serving as the unit for compression (a compressed page size is typically in the order of 10-100 kB). A set of pages, covering a certain entry range, belonging to a fixed set of columns is represented by a *cluster*. Clusters represent the units of data loaded into memory. In most cases, all columns can be represented by a single cluster for a given page range. In case this is not possible, however, multiple clusters may get bundled into a *cluster group*. This may happen for datasets larger than 100 GB, or in cases where the RNTuple has been created by combining two or more previously existing RNTuples (e.g., through merging). A visual representation of the on-disk data format is shown in Figure 1.

```
struct Event {
  int id;
  vector<Particle> particles;
};
struct Particle {
  vector<int> trackerIds;
  float energy;
};
```

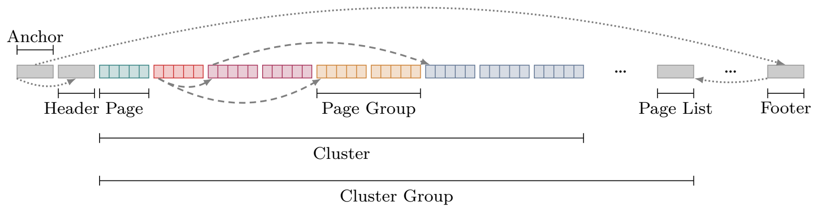


Figure 1: Visual representation of the RNTuple on-disk data format. The colour of each data member in the code snippet corresponds to an RNTuple column.

The `RNTupleInspector` provides storage metrics both in *elementary* and *aggregated* form, in order to cater to the largest number of use cases and users as possible. We explain the distinction below. Concrete examples are provided in the following sections.

Elementary metrics Elementary metrics include compression and size information for entire RNTuples, (sub)fields and columns. Here, (sub)fields and column are described by their own `RFieldTreeInspector` and `RColumnInspector` types, respectively. The decision to use the name `RFieldTreeInspector` rather than `RFieldInspector` is based on the fact that the storage information provided by this inspector will also include the field's subfields, if present. Inspector objects for fields

[1]`https://root.cern/releases/release-63200/`

and columns can be obtained from the main `RNTupleInspector` object through the field's name or ID or the column's ID. Each `*Inspector` object comes with the following base functions: `GetDescriptor`, `GetCompressedSize` and `GetUncompressedSize`. The descriptor of an RNTuple, field or column already provides a set of metadata information, which for maintainability reasons we have decided not to add directly to the inspector classes. Examples of such metadata information are the number of entries, a field name, or the ID of the field a particular column belongs to. Some information specific to the RNTuple, field or column that cannot be directly retrieved from its descriptor has been added to the respective inspector. For example, the `RNTupleInspector` provides `GetCompressionSettings`, and the `RColumnInspector` provides `GetNPages`.
The purpose of these elementary metrics is to give users fine-grained access to the storage information of their dataset. They can be used in an exploratory setting, i.e. by querying an `RNTupleInspector` object at the ROOT prompt. In addition, we foresee them being used by experiments to build their own (EDM-specific) inspection tools and scripts, for example to fine-tune and validate the various I/O parameters that can be configured during the writing of an RNTuple.

Aggregated information   Next to elementary metrics, the `RNTupleInspector` provides a handful of methods to conveniently get storage information at a glance. Currently, the following aggregated information is available:

- Storage information per column type. This information is provided in two ways: in text form, either as a table or CSV format (containing all relevant metrics), or as a histogram (containing a specific metric).
- The distribution of page sizes for one or multiple columns. This information is provided as a histogram. The column(s) can be specified by ID or by type.
- A graphical visualization of the on-disk layout of a dataset[2]. This shows a graphical representation of how pages and clusters are organized on disk.

As the `RNTupleInspector` is under active development, we foresee more aggregated information methods becoming available, in part guided by efforts from experiments in adopting RNTuple in their framework.

3 The `RNTupleInspector` interface

The interface of the `RNTupleInspector` follows the conventions of the core RNTuple interface. A new `RNTupleInspector` can be constructed via a factory method `Create`, which takes the name and a string representing the storage location of an RNTuple:

```
using ROOT::Experimental::RNTupleInspector;
auto inspector = RNTupleInspector::Create("MyNTuple", "my_ntuple.root");
```

In the current implementation, all field- and column-level inspector information will be collected and stored in memory upon construction of an `RNTupleInspector`. This is due to the fact that traversing all fields and columns is anyways necessary in order to obtain RNTuple-wide storage information. This means that for large datasets, some construction overhead is to be expected, but once the inspector has been created, storage information can be obtained instantaneously. Once an `RNTupleInspector` has been created, it can be used to obtain RNTuple-wide storage information and provides a way to get inspectors for fields or columns in this dataset. For example, assuming the dataset has a field named "muon_pt", we can obtain a `RFieldTreeInspector` as follows:

```
auto muonPtFieldInspector = inspector->GetFieldTreeInspector("muon_pt");
```

Next to methods directly providing storage information, a number of convenience methods are provided whose return value can aid the retrieval of storage information. To illustrate, suppose that our dataset also contains the fields "electron_pt", "photon_pt" and "jet_pt" in addition to the previously mentioned "muon_pt" field. Obtaining the compression factor for each field using the `RNTupleInspector` would look as follows:

[2]This feature will be available in an upcoming release of ROOT.

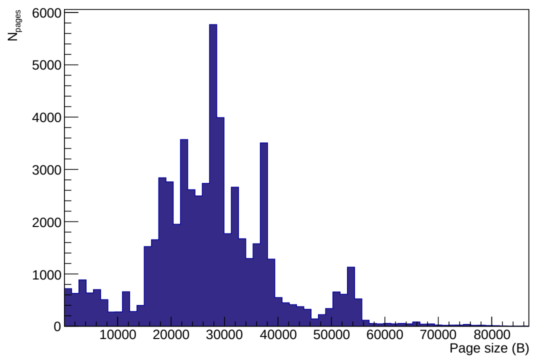
```cpp
for (const auto &fieldId : inspector->GetFieldsByName("(.*)_pt") {
  auto fieldInspector = inspector->GetFieldTreeInspector(fieldId);
  auto compressionFactor =
    fieldInspector.GetUncompressedSize() / fieldInspector.GetCompressedSize();
}
```

4 The `RNTupleInspector` by example

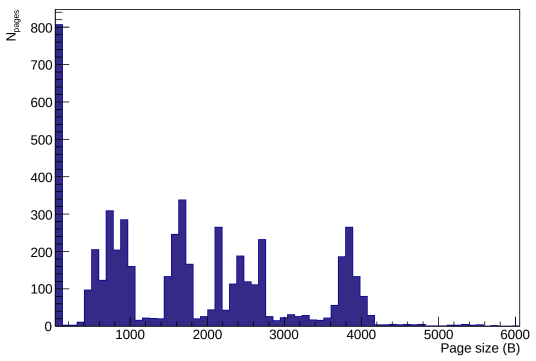
To demonstrate how the `RNTupleInspector` may be useful, we use a single file from the CMS DoubleMuon NanoAOD Open Data datasets [6], which we have converted from the original TTree format into RNTuple. We can use the `RNTupleInspector::PrintColumnTypeInfo` method to to get an indication of which data types are prevalent in a particular data sample. For this particular sample, this method provides us with the following output:

```
 column type     | count    | # elements       | compressed bytes   | uncompressed bytes
----------------|---------|-----------------|-------------------|--------------------
             Bit |      911 |       2116689026 |           35806567 |         2116689026
           UInt8 |       31 |         88488810 |           22670624 |           88488810
    SplitIndex64 |       19 |         43989237 |            9218504 |          351913896
     SplitReal32 |      349 |        937027507 |         1599353585 |         3748110028
     SplitUInt64 |        1 |          2315223 |            7028872 |           18521784
      SplitInt32 |       67 |        222633788 |           40964672 |          890535152
     SplitUInt32 |        2 |          4630446 |              16984 |           18521784
```

To understand how well this data is stored on disk, beyond the compression factor, it can be useful to understand how well individual pages are filled for a particular column type. When writing an RNTuple, pages are filled until a certain approximate size. The page gets compressed and written to the current cluster once this approximate size has been reached. The size of the compressed page highly depends on the column type and the kind of data it stores. This is well illustrated in the histograms shown in Figure 2, produced by `RNTupleInspector::GetPageSizeDistribution`. The distribution of the page sizes for float-type columns, which appears to follow some distribution, are very different from those for index columns, which are more scattered. This can be explained by the fact that float-type columns in the majority of cases contain physics data, typically following a certain distribution that dictates the compression. In contrast, index columns store the offsets of vectors and other collections, which are typically more arbitrarily distributed. In addition, the presence of empty vectors results in repeated offset values, which compresses extremely well. This is reflected by the peak close to a page size of 0 in Figure 2b.



(a) 32-bit float columns.



(b) 64-bit (integer) index columns.

Figure 2: Page size distribution for different RNTuple column types.

Finally, the storage layout of the dataset sample can be visualized through `RNTupleInspector::DrawStorageLayout`. Visualizing the layout can help in understanding the impact of some configuration options available during writing. One of these options is the use of buffered writing which, when enabled, buffers the compressed pages in memory such that pages belonging to a specific column can be grouped together in the cluster. By default, this setting is enabled. The layout of our dataset sample with buffered writing enabled is shown in Figure 3. In comparison, a version of the same dataset sample, written without buffering enabled, produces the layout visualization shown in Figure 4.

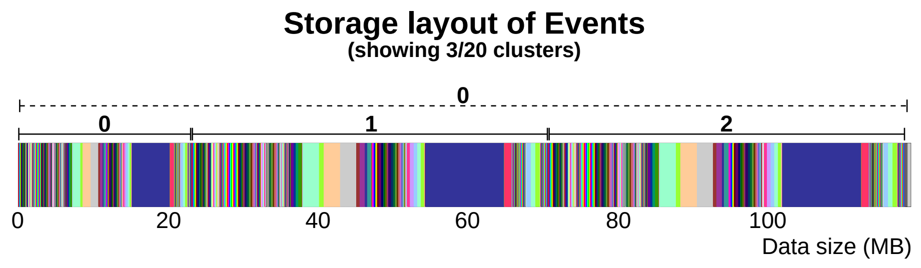


Figure 3: Storage visualization of an RNTuple "Events" with buffered writing enabled.

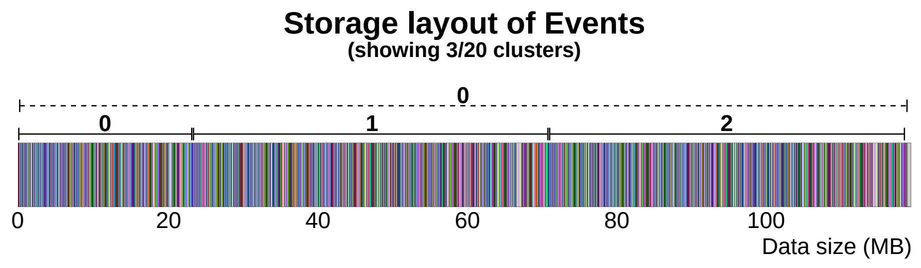


Figure 4: Storage visualization of an RNTuple "Events" with buffered writing disabled.

5 Conclusions and outlook

In this paper, we introduced the `RNTupleInspector` utility, which is designed and developed to get direct insights into the storage behaviour of a dataset stored in the RNTuple data format. The primary goal of the `RNTupleInspector` is to help developers understand the impact of different RNTuple writing settings on their data and how to potentially (further) optimize them.
The `RNTupleInspector` is available from ROOT 6.32.00 under the `Experimental` namespace, and will evolve alongside RNTuple itself. Features still foreseen to be added are inspector methods on the cluster and page level, as well as any additional features to help experiment software experts make the transition from TTree to RNTuple successful.

References

[1] O. Brüning and L. Rossi. "Chapter 1: High-Luminosity Large Hadron Collider". In: *CERN Yellow Reports: Monographs* 10 (Dec. 17, 2020), pp. 1–1. ISSN: 2519-8076. DOI: `10.23731/CYRM-2020-0010.1`.

[2] Jakob Blomer et al. "Evolution of the ROOT Tree I/O". In: *EPJ Web of Conferences* 245 (2020). ISSN: 2100-014X. DOI: `10.1051/epjconf/202024502030`.

[3] Alaettin Serhan Mete, Marcin Nowak, and Peter Van Gemmeren. "Persistifying the Complex Event Data Model of the ATLAS Experiment in RNTuple". In: *These Proceedings*. 22nd International Workshop on Advanced Computing and Analysis Techniques in Physics Research. Journal of Physics: Conference Series. Stony Brook, Long Island NY, USA, 2024. URL: `https://indico.cern.ch/event/1330797/contributions/5796492/`.

[4] Florine Willemijn de Geus et al. "Integration of RNTuple in ATLAS Athena". In: *EPJ Web of Conferences* 295 (2024), p. 06013. ISSN: 2100-014X. DOI: `10.1051/epjconf/202429506013`.

[5] The ROOT Project. *RNTuple Reference Specification.* GitHub. June 24, 2024. URL: `https://github.com/root-project/root/tree/master/tree/ntuple/v7/doc` (visited on 06/24/2024).

[6] CMS Collaboration. *DoubleMuon Primary Dataset in NANOAOD Format from RunG of 2016 (/DoubleMuon/Run2016G-UL2016_MiniAODv2_NanoAODv9-v2/NANOAOD).* 2024. DOI: `10.7483/OPENDATA.CMS.ZQS3.LGLP`.