

# Evolution of the ROOT Tree I/O

Jakob Blomer<sup>1,\*</sup>, Philippe Canal<sup>2</sup>, Axel Naumann<sup>1</sup>, and Danilo Piparo<sup>1</sup>

<sup>1</sup>CERN, Geneva, Switzerland

<sup>2</sup>Fermilab, Chicago, U.S.

**Abstract.** The ROOT TTree data format encodes hundreds of petabytes of High Energy and Nuclear Physics events. Its columnar layout drives rapid analyses, as only those parts (“branches”) that are really used in a given analysis need to be read from storage. Its unique feature is the seamless C++ integration, which allows users to directly store their event classes without explicitly defining data schemas. In this contribution, we present the status and plans of the future ROOT 7 event I/O. Along with the ROOT 7 interface modernization, we aim for robust, where possible compile-time safe C++ interfaces to read and write event data. On the performance side, we show first benchmarks using ROOT’s new experimental I/O subsystem that combines the best of TTrees with recent advances in columnar data formats. A core ingredient is a strong separation of the high-level logical data layout (C++ classes) from the low-level physical data layout (storage backed nested vectors of simple types). We show how the new, optimized physical data layout speeds up serialization and deserialization and facilitates parallel, vectorized and bulk operations. This lets ROOT I/O run optimally on the upcoming ultra-fast NVRAM storage devices, as well as file-less storage systems such as object stores.

## 1 Introduction

The data describing a High Energy Physics (HEP) event is typically represented by a record containing variable-length collections of sub records. An event can, for instance, contain a collection of particles with certain scalar properties ( $p_t$ ,  $E$ , etc.), another collection of jets, a collection of tracks, and so on. A typical physics analysis uses a large number of events but processes only a subset of the available properties. Therefore, ROOT’s TTree storage format support a *columnar* physical data layout for nested sub records and collections [1]. Values of a single property of many events (e.g.,  $p_t$  for events 1 to 1000) are stored consecutively on disk. Thus, only those parts that are required for an analysis need to be read. Similar values are likely to be grouped together, which is beneficial for compression.

More than 1 EB of data is stored in the TTree format. For HEP use cases, the TTree I/O speed and storage efficiency has shown to be significantly better than many industry products [2]. Furthermore, ROOT provides the unique feature of seamless C++ and Python integration where users do not need to write or generate a data schema. Yet, the TTree implementation limits the optimal use of new storage systems and storage device classes, such as object stores and flash memory, and it shows shortcomings when it comes to multi-threaded and GPU supported analysis tasks and fail-safe APIs.

---

\*e-mail: jblomer@cern.ch

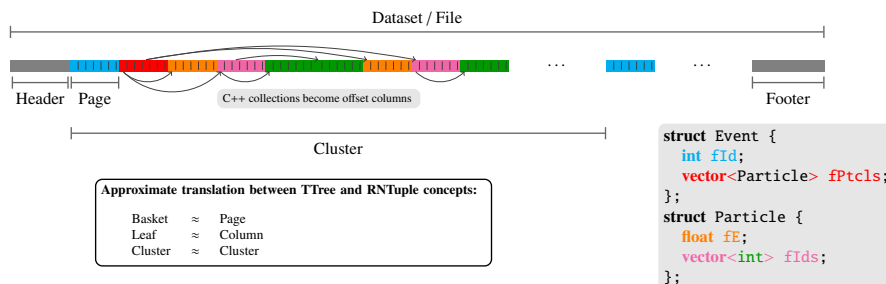
In this contribution, we present the design and first benchmarks of the *RNTuple* set of classes. The *RNTuple* classes provide a new, experimental columnar event I/O system that is backwards-incompatible to *TTree* both on the file format level and on the API level. Breaking backwards compatibility allows us to use contemporary nomenclature and to design the ROOT event I/O from the ground up for next-generation devices and the increased data rates expected from HL-LHC.

## 2 Design of the *RNTuple* I/O subsystem

This section describes key design choices of the *RNTuple* data format and of class design and the interfaces.

### 2.1 Data layout

Compared to the *TTree* binary data layout, the *RNTuple* data layout is modestly modernized and borrows some ideas from Apache Arrow [3] (see Figure 1). Data is stored in *columns* of fundamental types supporting arbitrarily deeply nested collections (*TTree* drops the “columnar-ness” for deeply nested collections).<sup>1</sup> Columns are partitioned in compressed *pages*, of typically a few tens of kilobytes in size. Like in *TTree*, *clusters* are a set of pages that contain all the data of a certain event range. They are typically a few tens of megabytes in size and a natural unit of processing for a single thread or task.



**Figure 1.** Breakdown of the *RNTuple* data layout. Each scalar field of the event struct is stored in a separate column.

A collection’s representation contains an offset column whose elements indicate the start index within the columns that store the collection content; this allows for random-access of individual events. The indexing is local to the cluster such that clusters can be written in parallel and freely concatenated to a larger data set. This also allows for “fast merging”, where several *RNTuple* files can be concatenated by only adjusting the header and footer. In contrast to *TTree*, offset pages and value pages are always separated, which should improve the compression ratio (to be confirmed). Integers and floating point numbers in columns are stored in little-endian format (*TTree*: big-endian) in order to allow for memory mapping of pages on most contemporary architectures. Boolean values, such as trigger bits, are stored as bitmaps (*TTree*: byte arrays), which improves the compression.

The *RNTuple* meta-data are stored in a header and a footer. The header contains the schema of the *RNTuple*; the footer contains the locations of the pages. At a later point,

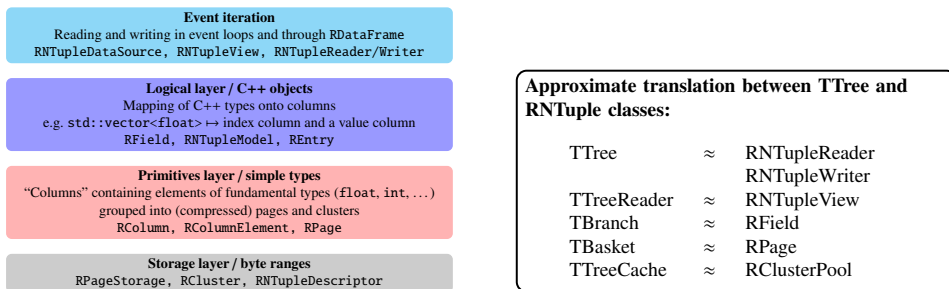
<sup>1</sup>In contrast to *TTree*, *RNTuple* currently does not support row-wise storage.

we will extend the meta-data with a regularly written *checkpoint footer* (e. g. every 100 MB) in order to allow for data recovery in case of an application crash during data taking. We will also extend the meta-data with a user-accessible, namespace-scoped map of key-value pairs, such that the experiment data management systems can maintain relevant information (checksums, replica locations, etc.) together with the data.

The pages, header and footer do not necessarily need to be written consecutively in a single file. The container for pages, header and footer can be a ROOT file where data is interleaved with other objects such as histograms. The container can also be an RNTuple bare file or an object store. It is also conceivable to store header and footer in a different file than the pages to avoid backward seeks.

## 2.2 Class design

The RNTuple class design comprises four layers (see Figure 2). The RNTuple classes make use of templates, such that for simple types (e.g., vectors of floats) that are known at compile time, the compiler can inline a fast path from the highest to the lowest layer without additional value copies or virtual calls.



**Figure 2.** Layers and key class of RNTuple and their approximate counterparts in TTree.

The *event iteration layer* provides the user-facing interfaces to read and write events, either through RDataFrame [4] or as hand-written event loops. The user interface is presented in more detail in Section 3.

The *logical layer* splits C++ objects into columns of fundamental types. Its central class is the *RField* that provides a C++ template specialization for reading and writing of an I/O supported type. Currently there is support for boolean, integer and floating point, `std::vector` and `std::array` containers, `std::string`, `std::variant`, and user-defined classes with a ROOT dictionary. In the future, we will provide support for additional types (e.g., `std::map`, `std::chrono`) and possibly for intra-event object references as a limited form of pointers. While RNTuple limits I/O support to an explicit subset of C++ types, those types are fully composable (e.g., a user-defined class containing a vector of arrays of another user-defined class).

The *primitives layer* governs the pool of uncompressed and deserialized pages in memory and the representation of fundamental types on disk. For most fundamental types, the memory layout equals the RNTuple on-disk layout. In some circumstances, pages need to be *packed* and *unpacked*, for instance in order to store booleans as bitmaps or in order to store floating point values with reduced precision.

The *storage layer* provides access to the byte ranges containing a page on a physical or virtual device. The storage layer manages compression and reads and writes to and from the

I/O device. It also allocates memory for pages in order to allow for direct page mappings. Currently there is support for a storage layer that uses a ROOT file as an RNTuple data container and a storage layer that uses a bare file for comparison and testing. We plan to add another implementation that uses an object store. We will also add virtual storage layers that combine RNTuple data sets similar to TTree's *chains* and *friend trees*.

An RNTuple *cluster pool* provides I/O scheduling capabilities. The cluster pool spawns an I/O thread that asynchronously preloads upcoming pages of active columns. The cluster pool can linearize, merge and split requests to optimize the read pattern for the storage device at hand (e. g. spinning disk, flash memory, remote server).

### 3 RNTuple user interfaces

The RNTuple user-facing API is supposed to be easy to use correctly as to minimize the likelihood of application crashes and wrong results. To this end, RNTuple provides an RDataFrame data source so that RDataFrame analyses code can be use unmodified with RNTuple data.

The RNTuple interface for implementing hand-written event loops uses modern standard techniques, including smart pointers, event traversal by C++ iterators and compile-time safety through templated interfaces (see Figure 3). For the type-unsafe interface, a runtime check verifies that the the on-disk type and the in-memory type of fields match.

```

auto ntpl =
  RNTupleReader::Open("Events", "f.root");
auto viewPt = ntpl->GetView<float>("pt");

for (auto i : ntpl->GetEntryRange()) {
  hist.Fill(viewPt(i));
}

auto model = RNTupleModel::Create();
auto fldPt = model->MakeField<float>("pt");
// Note: there is also a void* based,
//       runtime type-safe API

auto ntpl = RNTupleReader::Open(
  std::move(model), "Events", "f.root");

for (auto entryId : *ntpl) {
  ntuple->LoadEntry(entryId);
  hist.Fill(*fldPt);
}

```

**Figure 3.** RNTuple interface sketch for reading data. Left-hand side: zero-copy interface where the memory pointed to by *views* is managed by RNTuple. Right-hand side: interface that copies values into generated or user-provided memory locations.

The RNTuple classes are thread-friendly, i. e. multiple threads can safely use their own copy of RNTuple classes to read the same data concurrently. In the future, we envision support for multi-threaded writing (one cluster per thread or task) as well as support for multiple threads reading concurrently from the same range of clusters of an RNTuple. In single-threaded analyses, available idle cores should be used for decompression. We believe that these changes will require very little changes to the user-facing API.

Error handling, for instance in case of device faults or malformed input data, is an important aspect of I/O interfaces. While it is often difficult to recover gracefully from I/O errors, the I/O layer should reliably detect errors and produce an error report as close as possible to the root cause. To this end, RNTuple throws C++ exceptions for I/O errors.

At a later point, we intend to add a limited C API for RNTuple in order to facilitate ROOT data being transferred to 3rd party consumers, such as numpy arrays or machine learning toolkits. To this end, most of RNTuple is implemented not to depend on core ROOT classes,

**Table 1.** Sample analyses for performance evaluation. Main differences are the data model (flat or with collections) and the number of required branches (dense or sparse reading).

LHCb run 1 open data B2HHH	H1 micro dst [ $\times 10$ ]	CMS nanoAOD June 2019
18/26 branches (>75 %)	16/152 branches ( $\sim 10$ %)	6/1479 branches (<1 %)
fully flat data model	event sub collections	event sub collections
8.5 million events	2.8 million events	1.6 million events
24 k selected events	75 k selected events	141 k selected events

**Table 2.** Overview of the benchmarking hardware.

Hardware	Machine 1	Machine 2
CPU	Xeon Platinum 8260 @ 2.4 GHz	Xeon E5-2630v3 @ 2.4 GHz
Memory	DDR4 RDIMM 2933 MHz	DDR4 RDIMM 2133 MHz
Optane (NVRAM)	Optane DC 2666 MHz (ext4/DAX)	—
SSD (flash)	Intel DC P4510, PCIe 3.1 $\times 4$	—
HDD (spinning)	—	2 $\times$ SAS 7200 RPM (RAID1)
Network	—	1 GbE

such that a minimal, stand-alone RNTuple I/O library can be built. The functionality of this library will initially be limited to reading simple numerical type fields and vectors thereof.

## 4 Performance evaluation

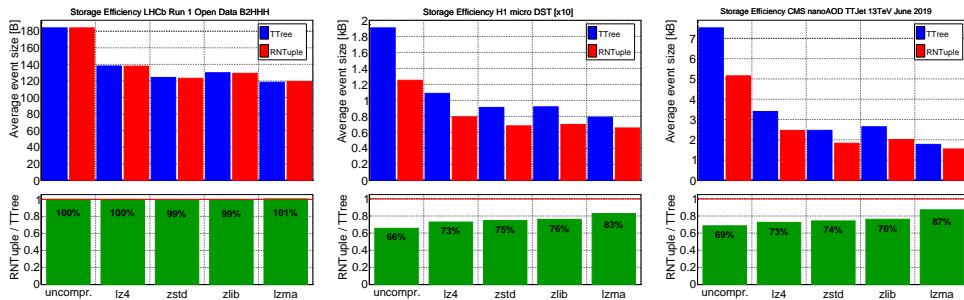
In this section, we analyze the RNTuple performance in terms of read throughput and file size for typical, single threaded analysis tasks. We use three sample analyses for the benchmarks (see Table 1). Each analysis requires a subset of the available event properties, uses some properties to filter events, and calculates an invariant mass from the selected events. The analyses were implemented using both TTree and RNTuple, each variant optimized for best performance with hand-written event loops<sup>2</sup>. Basket/Page sizes and cluster sizes are comparable between TTree and RNTuple files. The “LHCb” sample is derived from an LHCb Open Data course [5]. The “H1” sample is derived from the ROOT “H1 analysis” tutorial with the original data cloned ten times. The “CMS” sample is derived from the ROOT “dimuon” tutorial using the 2019 nanoAOD format [6] with simulated data. Two dedicated physical nodes, “machine 1” and “machine 2” are used for running the benchmarks (see Table 2). Both machines run CentOS 7 and have ROOT<sup>3</sup> compiled with gcc 7.3. A third dedicated node runs XRootD in version 4.10 and is configured to hold the data on a RAM disk.

### 4.1 Storage efficiency

Figure 4 shows the file format efficiency for the input data of the sample analysis. As expected, the TTree and RNTuple efficiency is very similar on the “LHCb” flat data model. For “H1” and “CMS”, RNTuple shows significantly better efficiency due to the more efficient storage of collections and boolean values. (Approximately half of the difference in file size could be eliminated by using TTree’s experimental `kGenerateOffsetMap` I/O flag.) Space savings of RNTuple remain even after compression.

<sup>2</sup>For the implementation, see <https://github.com/jblomer/iotools/tree/ntuple-chep-2019>

<sup>3</sup>ROOT branch <https://github.com/jblomer/root/tree/ntuple-chep-2019>



**Figure 4.** File size comparison of the sample analysis input data in TTree and RNTuple format with different compression algorithms.

## 4.2 Read performance

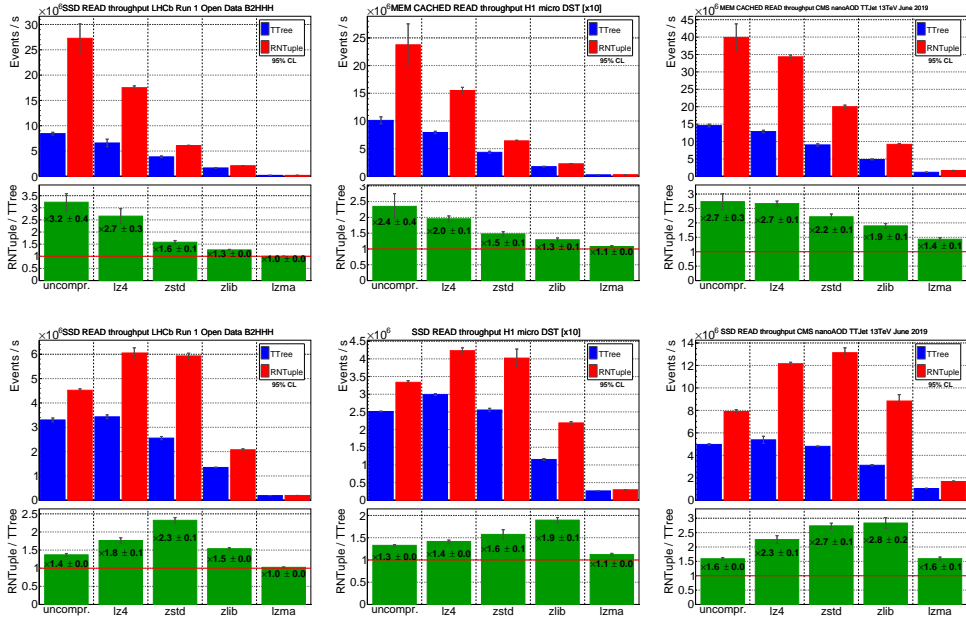
Figure 5 shows the event throughput for running the sample analyses. When reading from warm file system buffers, the performance is dominated by deserialization and decompression. The data deserialization in RNTuple is significantly faster compared to TTree. With stronger compression algorithms, the performance is more dominated by decompression than by deserialization. Still, even for LZMA compressed data reading RNTuple data is faster for the H1 and CMS samples.

When reading with cold file system buffers, as shown in the lower half of Figure 5, the performance depends not only on the deserialization and decompression speed but also on the I/O throughput of the device. The additional CPU time spent on strong compression can be more than compensated by a smaller transfer volume. For RNTuple, there is a sweet spot for the recent zstd compression algorithm, in particular if taking into account the smaller file size as compared to zlib and lz4.

Figure 6 compares cold cache read performance for different, frequently used physical data sources. For the slow devices HDD and 10 GbE, the performance is dominated by the I/O scheduler, i. e. by TTreeCache resp. RClusterPool. The I/O scheduler linearizes requests, merges nearby requests, and issues vector reads in order to minimize the overall number of requests sent to a device and the total transfer volume. In these benchmarks the RNTuple's I/O scheduler shows a performance at least as good as the TTreeCache.

## 4.3 SSD optimizations

In contrast to spinning disks, SSDs are inherently parallel devices that benefit from a large queue depth so that they can read from multiple flash cells concurrently. Figure 7 shows the effect of reading with multiple concurrent streams. To this end, we extend the RNTuple I/O scheduler to read with multiple threads (1 stream/thread). Where the read performance is limited by I/O and not by decompression and deserialization, increasing the number of streams can yield another speed improvement of around a factor of 2.5. The gains max out at around 16 streams. The lower gains for uncompressed LHCb and CMS samples are due to a limitation in the current RNTuple implementation that only preloads a single cluster. It therefore does not provide enough concurrent requests to fill the parallel streams. With implementation of multi-cluster read-ahead, this limitation is going to be removed. An interesting topic of future work is investigating automatic ways of the I/O scheduler to adjust to the underlying physical hardware.



**Figure 5.** Read throughput in events per second on machine 1. Upper half shows the results for warm file system buffers. Lower half shows the results for reading from SSD with a cold cache. See Figure 7 for further improvements for SSDs.

#### 4.4 Optane DC NV-RAM evaluation

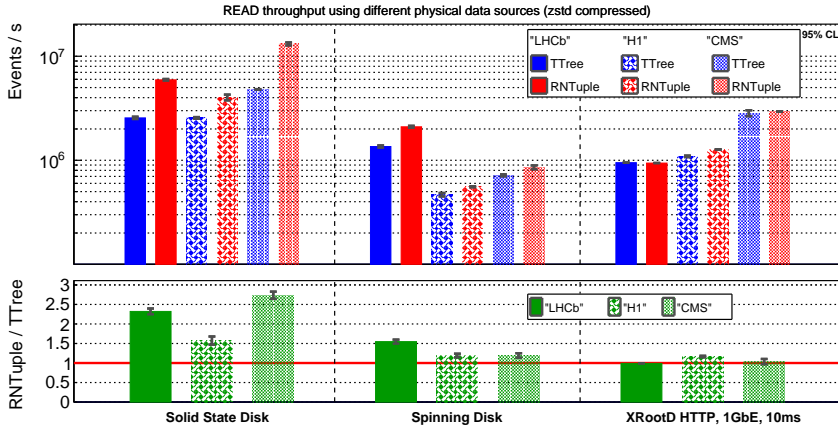
Figure 8 shows the performance when reading RNTuple data from Optane DC NV-RAMs. The performance characteristics of NV-RAMs are in-between RAM and SSDs (here, we are not exploiting the non-volatility). In the future, they might become a more widespread additional cache layer or installed as a dedicated performance storage tier, e. g. in analysis facilities.

The results show no significant difference between reading from warm file system caches and reading from NV-RAM. As we also do not reach the peak throughput of the NV-RAM modules, the results suggest a bottleneck in the I/O deserialization or plotting part of the analysis run. Further optimizations of the RNTuple I/O path are subject of future work.

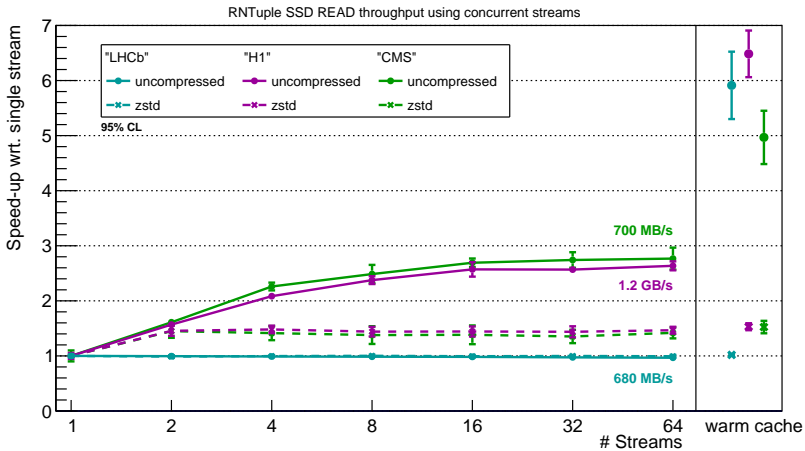
Due to the fact that the RNTuple on-disk layout matches the in-memory layout, we can compare reading data explicitly with POSIX `read()` and implicitly by memory mapping. On (byte-addressable) NV-RAM and warm file system buffers, both mechanisms yield comparable results. When reading sparsely from SSDs, the RNTuple I/O scheduler optimizations bring a significant performance gain. Further investigation reveals that the I/O scheduling that underpins the memory mapping in Linux issues an order of magnitude more requests to the device than the RNTuple scheduler.

## 5 Conclusion

In this contribution we presented the design and a first performance evaluation of RNTuple, ROOT's new experimental event I/O system. The RNTuple I/O system is a backwards-



**Figure 6.** Read speed with different bandwidth and latency profiles. SSD benchmarks from machine 1, HDD and HTTP benchmarks from machine 2 connected to a dedicated, third XRootD server. Note that the SSD results on the left hand side are identical to the SSD/zstd results in Figure 5.

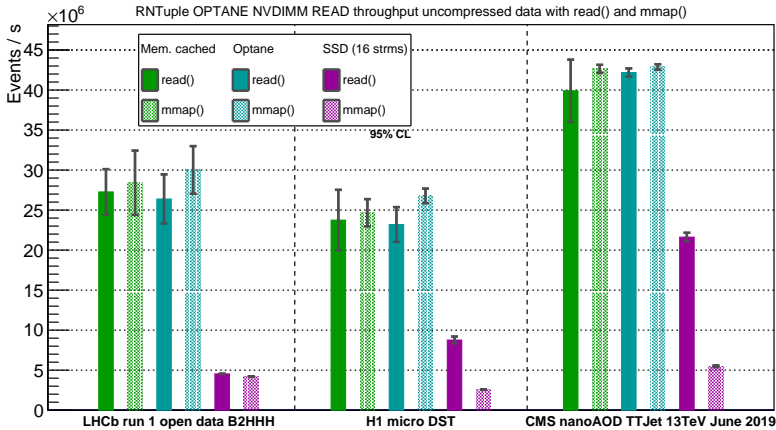


**Figure 7.** Full exploitation of SSDs by concurrent streams on machine 1. For comparison, the right hand side repeats the warm cache results from Figure 5. Single stream performance is identical to the SSD/zstd results in Figure 5.

incompatible redesign of TTree, based on the many years of experience of the TTree development. It is from the ground up designed to work well in concurrent environments and to optimally support modern storage hardware and systems, such as SSDs, NV-RAM, and object stores.

Our benchmarks suggest that compared to TTree RNTuple can yield read speed improvements between a factor of 1.5 to 5 in realistic analysis scenarios, while at the same time reducing data sizes by 10 % to 20 %. We will gradually move the RNTuple code from a pro-





**Figure 8.** Read performance using Optane DC NV-RAM in “App Direct” mode on machine 1. The NV-RAM block device is formatted with ext4 with DAX optimization. Uncompressed input data is used in order to allow for comparison between POSIX `read()` and `mmap()`. Note that the SSD `read()` results are identical to the 16 streams result in Figure 7.

totype to a ROOT production component. The RNTuple classes are already available in the `ROOT::Experimental::RNTuple` namespace if ROOT is compiled with the `root7` cmake option. Tutorials are available to demonstrate the RNTuple functionality. We consider these developments and the associated future R&D topics essential building blocks for coping with data rates at the HL-LHC.

## Acknowledgements

We would like to thank Fons Rademakers and Luca Atzori from CERN openlab for giving us access to NV-RAM devices. We would like to thank Dirk Düllmann and Michal Simon from CERN IT for providing us an XRootD test node. We would like to thank Oksana Shadura, Brian Bockelman, and Jim Pivarski for many fruitful discussions and suggestions.

## References

- [1] R. Brun, F. Rademakers, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment A **389**, 81 (1997)
- [2] J. Blomer, Journal of Physics: Conference Series **1085** (2018)
- [3] The Apache Software Foundation, *Apache Arrow* (2019), <https://arrow.apache.org>
- [4] G. Amadio, J. Blomer, P. Canal, G. Ganis, E. Guiraud, P.M. Vila, L. Moneta, D. Piparo, E. Tejedor, X.V. Pla, Journal of Physics: Conference Series **1085** (2018)
- [5] A. Rogozhnikov, A. Ustuzhanin, C. Parkes, D. Derkach, M. Litwinski, M. Gersabeck, S. Amerio, S. Dallmeier-Tiessen, T. Head, G. Gilliver (2016), talk at the 22nd Int. Conf. on Computing in High Energy Physics (CHEP’16)
- [6] A. Rizzi, G. Petrucciani, M. Peruzzi, EPJ Web Conf **214** (2019)