Exploring Object Stores for High-Energy Physics Data Storage

Javier López-Gómez[1,*] and *Jakob* Blomer[1,**]

[1]CERN, Geneva, Switzerland

Abstract. Over the last two decades, ROOT TTree has been used for storing over one exabyte of High-Energy Physics (HEP) events. The TTree columnar on-disk layout has been proved to be ideal for analyses of HEP data that typically require access to many events, but only a subset of the information stored for each of them. Future colliders, and particularly HL-LHC, will bring an increase of at least one order of magnitude in the volume of generated data. Therefore, the use of modern storage hardware, such as low-latency high-bandwidth NVMe devices and distributed object stores, becomes more important. However, TTree was not designed to optimally exploit modern hardware and may become a bottleneck for data retrieval. The ROOT RNTuple I/O system aims at overcoming TTree's limitations and at providing improved efficiency for modern storage systems. In this paper, we extend RNTuple with a backend that uses Intel DAOS as the underlying storage, demonstrating that the RNTuple architecture can accommodate high-performance object stores. From the user perspective, data can be accessed with minimal changes to the code, that is by replacing a filesystem path by a DAOS URI. Our performance evaluation shows that the new backend can be used for realistic analyses, while outperforming the compatibility solution provided by the DAOS project.

1 Introduction

In High-Energy Physics (HEP), an event is encoded as a record that may contain a number of variable-length collections or properties. For instance, an event might contain a collection of particles, each including a set of scalar properties (e.g., momentum, energy, etc.), a collection of tracks, jets, and any other data collected by a particle detector. Most analysis of HEP data require access to many events, but only require a subset of the properties stored for each instance. In this scenario, using conventional row-oriented storage systems is suboptimal, as they incur in an overhead due to reading a high volume of unneeded data.

ROOT TTree [1] avoids this overhead by using a columnar layout, i.e., consecutively storing values of the same property for a range of rows. This encoding not only avoids the aforementioned overhead, but also contributes to improve data compression as similar values are stored together. While TTree has been used to efficiently store more than 1 EB of HEP data during the last two decades, it was not designed to fully exploit modern hardware and/or storage systems, e.g. NVMe devices and object stores. In a previous publication [2] we

---

*e-mail: javier.lopez.gomez@cern.ch
**e-mail: jblomer@cern.ch

presented RNTuple, the ROOT's new, experimental columnar I/O subsystem. RNTuple is a backwards-incompatible redesign of TTree that overcomes its limitations and takes advantage of state-of-the-art hardware and storage systems. The layered design of RNTuple permits the separation of encoding and storage of pages (groups of values belonging to the same data column). This separation is crucial for supporting different storage systems, such as POSIX files or object stores.

In this paper, we extend RNTuple with a new backend that leverages Intel DAOS [3] as the underlying storage. Given the importance of object stores in next-generation HPC centers, e.g. the Argonne's Aurora [4] supercomputer, we aim to bridge the gap between HEP and the upcoming supercomputing facilities. In particular, we contribute with the following:

- We provide the design and prototype implementation of a RNTuple backend that can leverage an existing DAOS deployment to store HEP data. In that regard, we propose and evaluate two different page to object mappings.

- We evaluate the read/write performance of our backend with a realistic LHCb analysis in several conditions, and compare it to the DAOS dfuse compatibility layer and a local filesystem.

The rest of this paper is organized as follows. Section 2 briefly reviews other related work found in the literature. Section 3 provides an introduction to RNTuple and Intel DAOS. Section 4 describes the design of the new RNTuple DAOS backend. In Section 5, we evaluate its performance compared to the DAOS dfuse compatibility layer and a local filesystem. Section 6 summarizes the contribution and enumerates future work.

2 Related work

Due to the aforementioned advantages, columnar storage systems are popular not only in high-energy physics but also in other areas. During the last few years, many solutions for columnar storage of large datasets such as Amazon Redshift [5], Apache Parquet [6] and Apache Arrow [7] have emerged. Oftentimes, users need to store nested complex data types, which is especially relevant for the HEP area. A well-known portable, cross-platform solution for this is HDF5 [8]; however, it currently does not support columnar storage.

At the time of this writing, we found few publications in the literature targeting DAOS integration for next-generation exascale HPC centers. In particular, HDF5 already implements a connector for DAOS [9]. Also, there is work in progress to integrate DAOS with Apache Parquet/Arrow, and the Lustre [10] parallel filesystem.

3 Background

In this section, we briefly describe the ROOT project and its experimental, new columnar I/O subsystem (RNTuple). Additionally, we introduce Intel DAOS (Distributed Asynchronous Object Store) [3], a contemporary object store relevant for next-generation HPC centers.

3.1 The ROOT project and RNTuple

ROOT [1] is an open-source data analysis framework that is part of the software stack of many HEP experiments, e.g. CERN's ATLAS and CMS. ROOT provides components for statistics analysis, data visualization, convenient data storage/retrieval, and is bundled with an interactive C++ interpreter that can be used for fast prototyping of analysis code. ROOT also includes dynamic Python bindings that provide access to C++ types and objects. The traditional columnar HEP data storage is provided by TTree.

RNTuple is the new columnar I/O subsystem that shall replace TTree in the long term. RNTuple has been designed to take advantage of state-of-the-art hardware and storage systems. Its implementation makes thorough use of template metaprogramming and features available in recent C++ standards, which contributes to a lower run-time overhead and a more maintainable and simpler code. Several design decisions were taken to improve the effective read/write bandwidth and decrease the size of the data stored on a persistent medium. For instance, integer and floating-point numbers use little-endian encoding so that they match the endianness of most popular architectures, which allows for direct memory-mapping of pages; booleans are packed into octets, and more compression schemes are under investigation. RNTuple was previously described in [2]. For the sake of space, we will only summarize the high-level architecture.

The design of RNTuple comprises four layers: storage layer, primitives layer, logical layer, and event iteration layer (see Figure 1). The storage layer abstracts the details of the underlying storage class (e.g. a POSIX filesystem, an object store, etc.), optionally providing additional packing and/or compression; therefore, this layer is responsible for storing/retrieving byte ranges organized in clusters and pages. Pages contain a range of values for a given column, whereas a cluster contains pages that store data for a specific row range. The primitives layer groups a range of elements of a fundamental type, i.e. a horizontal partition, into pages and clusters. The logical layer maps complex C++ types to columns of fundamental types, e.g. `std::vector<float>` is mapped to an index column and a value column. Finally, the event iteration layer provides a convenient interface for iterating over events.
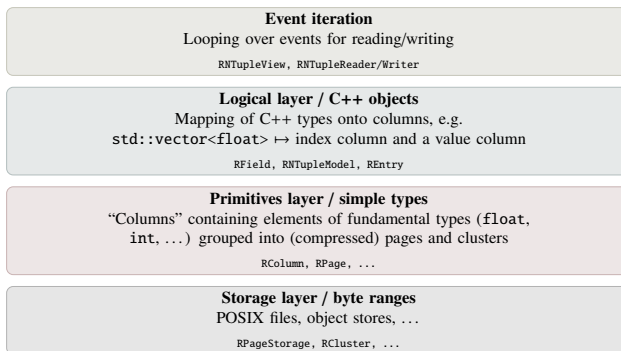

> **Event iteration**
> Looping over events for reading/writing
> `RNTupleView, RNTupleReader/Writer`
>
> **Logical layer / C++ objects**
> Mapping of C++ types onto columns, e.g.
> `std::vector<float>` ↦ index column and a value column
> `RField, RNTupleModel, REntry`
>
> **Primitives layer / simple types**
> "Columns" containing elements of fundamental types (`float`, `int`, …) grouped into (compressed) pages and clusters
> `RColumn, RPage, ...`
>
> **Storage layer / byte ranges**
> POSIX files, object stores, …
> `RPageStorage, RCluster, ...`

Figure 1. Layers of the RNTuple subsystem.

Given this layer organization, support for a new storage system can be added by extending the storage layer via a backend that is able to handle read/write operations for pages and clusters. RNTuple already includes a file backend that allows reading and writing ROOT files. The relevant on-disk layout is depicted in Figure 2. This layout comprises an anchor (that includes the offsets and size of the header and footer sections, among other information), a header (that describes the schema), a footer (that encodes locations of pages/clusters), and a number of pages grouped into clusters.

3.2 Intel DAOS

Traditional storage systems and kernel I/O schedulers were designed for high-latency rotating disks that could handle a few hundred IOPS (I/O operations per second). On the one hand, optimizations applied in the block layer (coalescing, buffering, etc.) are less relevant for
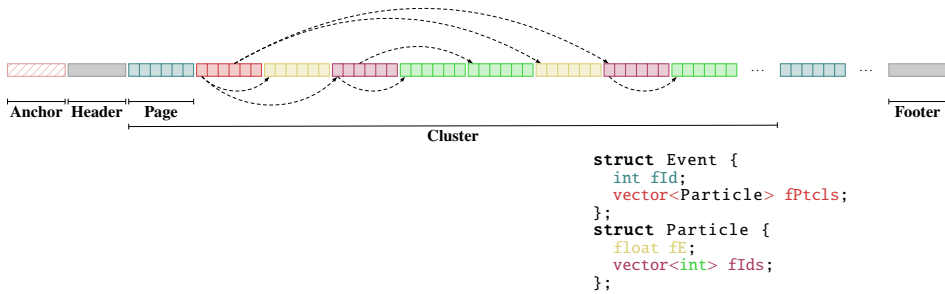
```
struct Event {
  int fId;
  vector<Particle> fPtcls;
};
struct Particle {
  float fE;
  vector<int> fIds;
};
```

Figure 2. RNTuple on-disk physical layout. A page stores a range of values for the corresponding struct member (same color). Dotted arrows symbolize a reference to a different page.

modern storage devices, such as NVMe SSDs and Intel Optane DC persistent memory, and they add overhead if used. On the other hand, the POSIX I/O semantics and particularly its strong consistency model, have been identified as a major problem in parallel filesystem scalability. To tackle this issue, Intel DAOS provides a fault-tolerant object store optimized for high bandwidth, low latency, and high IOPS, that targets next-generation exascale HPC centers [4] and aims at fully exploiting modern hardware. Traditionally, object stores provide at least the *get* and *put* primitives (*fetch* and *update* in DAOS, respectively) to access object content.

In DAOS, a system is comprised of a number of servers that run a Linux daemon exporting local NVM storage. DAOS servers listen on one management interface and many fabric endpoints (for data transport). Storage resources are partitioned into *targets* that can be accessed independently to avoid contention. The system administrator can create pools that may span a number of servers. Its storage is distributed among the available targets; such a division is called a *shard*. Containers can be created by users using the space assigned to a pool. Objects can then be placed into a container. Pools and containers are identified by a UUID (Universally Unique IDentifier), while objects can be referenced using a 128-bit object identifier (OID). An object is a key-value store with locality and redundancy/replication (as determined by the object class). The key is split into distribution key (*dkey*) and attribute key (*akey*), where values for the same *dkey* will be co-located in the same target. Object contents can be accessed through three different APIs: multi-level key-array (native), key-value, and one-dimensional array. The relationship between the different entities can be seen in Figure 3.
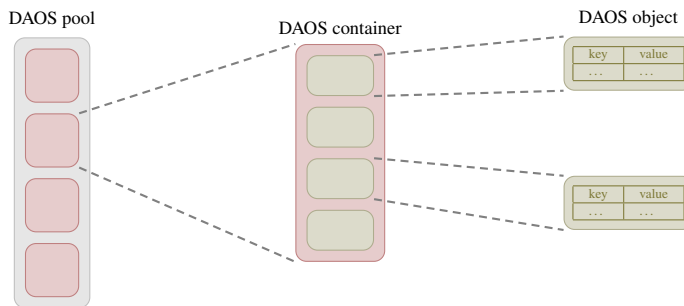

Figure 3. Graphical representation of different DAOS entities.

DAOS also provides components that permit to port applications. Specifically, it implements a POSIX filesystem that can be used either through `libioil` (I/O interception library) or `dfuse` (a FUSE filesystem), a ROMIO driver for applications using MPI-IO, and connectors for other systems such as HDF5 and Apache Spark.

4 Design of the RNTuple DAOS backend

This section explains in detail the new RNTuple DAOS backend. In Section 4.1, we give an overview of the design and engineering decisions. In Section 4.2, we describe how clusters and pages map to DAOS objects. Finally, Section 4.3 illustrates required changes to the user code.

4.1 Overview

RNTuple relies on the `RPageSource` and `RPageSink` classes for reading and writing pages, respectively. Therefore, new backends can be implemented by inheriting from these classes. The `RPageSource` class defines two member functions that can be overridden in a subclass: `PopulatePage()`, responsible for reading a single page, and `LoadCluster()`, that is called to read all pages belonging to a given cluster. The latter can optionally implement additional optimizations, e.g. vector reads and parallel decompression. The `RPageSink` class provides the `CommitPage()` and `CommitCluster()` methods that are responsible for writing data. Like their read counterparts, they may include optimizations to coalesce write requests.

The C DAOS API is provided by libdaos. While it can be directly consumed in a C++ program, it involves writing boilerplate code and requires manual resource management. To hide away complexity and manage shared resources, we wrapped a subset of libdaos functionality into three C++ classes: `RDaosPool`, `RDaosContainer`, and `RDaosObject`. `RDaosPool` handles the connection to a DAOS pool. This connection is kept open at least until all the containers have been closed. `RDaosContainer` gives basic access to objects inside a container; additionally, it provides vector read/write of multiple objects via `ReadV()` and `WriteV()`. Internally, these functions rely on libdaos event queues and asynchronous fetch/update operations. Finally, low-level access to objects is provided by the `RDaosObject` class.

In summary, `PopulatePage()` and `CommitPage()` execute a synchronous read or write operation through the simple interface defined by `RDaosContainer`. Additionally, reading all the pages in a cluster is optimized using a vector read. At the time of this writing, vector writes are not used. The scheme for assigning OIDs, *dkey*, and *akey* is described in the next section.

Finally, we provide mocks for the libdaos subset used by the previous classes that makes it possible to test the backend in those environments lacking a real DAOS deployment.

4.2 Mapping RNTuple clusters and pages to objects

Given the richness of the DAOS interface, several mappings of RNTuple pages and clusters to DAOS entities are conceivable.

Our proposal defines two possible mappings for pages (and clusters, where applicable) which are described in the following.

One OID per page. A sequential OID is assigned for each committed page. We make no specific use of *dkey* and *akey*, i.e., both are kept constant. This naïve association does not fully exploit DAOS capabilities.

One OID per cluster. The OID equals the RNTuple clusterId; the *dkey* is a sequential number that is used for addressing pages in the cluster; *akey* has no specific use and is kept constant. This impacts how pages are distributed among targets in DAOS servers, which might improve both read and write bandwidth.

Regardless of the mapping, independent OIDs and constant *dkey*/*akey* pair are used to store the RNTuple header, footer, and anchor. We compared the read/write throughput measured for both cases as part of our experimental evaluation in Section 5.

4.3 User interface

The `RNTupleReader` and `RNTupleWriter` classes provide a convenient interface to store/retrieve data. The second and third arguments of the constructor specify, respectively, the ntuple name and the location. This location is typically a filesystem path; however, a DAOS container may be referenced through the use of a specific URI format. These URIs contain, in order, the string '`daos://`', the pool UUID followed by '`:`', the list of service replica ranks separated by '`_`', and the container UUID. As can be seen in Figure 4, user code only requires minor changes to read or write data in a DAOS container.

The previous schema allows a user to access a DAOS container transparently. Nevertheless, it requires the manual specification of the pool and container UUIDs. While this is not user-friendly, we consider finding the physical path of a dataset a task of the data management system and outside the scope of RNTuple.

```cpp
auto ntuple = RNTupleReader::Open(
    "DecayTree",
    "./B2HHH~zstd.ntuple");

auto viewH1IsMuon = ntuple->GetView<int>("H1_isMuon");
...
```

(a) File-backed storage.

```cpp
auto ntuple = RNTupleReader::Open(
    "DecayTree",
    "daos://4b614f30-f476-4831-84ba-a51197600020:1/f1b0a25a-7fbb-
        4fba-b7d2-9a9f4e10e8f4");

auto viewH1IsMuon = ntuple->GetView<int>("H1_isMuon");
...
```

(b) DAOS-backed storage.

Figure 4. Changes required to the user code. Note the use of a `daos://` URI in sub-figure (b).

5 Evaluation

In this section, we present the results of the experimental evaluation of the RNTuple DAOS backend we carried out. Specifically, we measured performance differences w.r.t. file-backed storage in a variety of situations. All the experiments in this performance evaluation employed the following hardware and software environment.

Hardware platform. We used the CERN Openlab DAOS testbed to run the experiments. This cluster comprises three servers and two client nodes, connected to an Omni-Path Edge 100 Series 24-port switch. Note that we only made use of one client node, i.e., all tests ran on the same host. The hardware specifications for the cluster are detailed in the following paragraphs.

Server nodes. These nodes have 2× Intel Xeon Platinum 8260 CPU (24 physical cores) running at 2.4 GHz, 35.75 MB of L3 cache, 384 GB of DDR4 RAM. Note that HyperThreading was enabled (total of 96 logical cores per node). The nodes are equipped with 12× 128 GB DDR4 Optane DCPMM memory, 4× 1 TB NVMe SSDs (Intel SSDPE2KX010T8), and an

Intel Omni-Path HFI Silicon 100 Series Host Fabric Adapter. The OS is CentOS Linux 7.9 (kernel 3.10.0-1127).

Client node. The client node is equipped with 2× Intel Xeon Platinum 8160 CPU (24 physical cores) running at 2.1 GHz, and 192 GB of DDR4 RAM; HyperThreading is enabled for this node (total of 96 logical cores). High-bandwidth interconnection is provided by a Intel Omni-Path HFI Silicon 100 Series Host Fabric Adapter. The OS is CentOS Linux 7.9 (kernel 3.10.0-1160).

Software. The evaluation was carried out using DAOS 0.9.4, libfabric 1.7.2, libpsm2 11.2.78, and ROOT revision `1f1e9b8`. DAOS was configured to use PSM2 transport (`ofi+psm2`) and flow control was disabled (`CRT_CREDIT_EP_CTX=0`) in order to avoid limiting the bandwidth of in-flight asynchronous requests.

Test cases. In order to evaluate the performance of our proposal, we used the `gen_lhcb` and `lhcb` programs from the RNTuple benchmark repository [11]. These programs are used to import an existing CERN LHCb dataset (in a ROOT file) into RNTuple, and to perform a typical LHCb analysis, respectively. In the following, we will use `gen_lhcb` to measure the write throughput. Similarly, `lhcb` provides a measurement of the read throughput.

We evaluated the performance of the RNTuple libdaos-based backend w.r.t. the use of a local filesystem and the dfuse FUSE filesystem. In the subsequent plots, the data point and error bars correspond, respectively, to the average and minimum/maximum value over 20 executions of the same test. Specifically, we analyzed the performance in the following scenarios.

Constant page size, increasing cluster size. The number of elements per page is kept constant at 10 000, while the number of elements per cluster is doubled on each iteration (starting at 20 000). A page size of 10 000 roughly equates the TTree default basket size of 32 kB. Because RNTuple loads in parallel pages that belong to the same cluster, this scenario evaluates the effect of queuing many small read operations.

Increasing page size, constant cluster size. The number of elements per page is doubled on each iteration (starting at 10 000), while the number of elements per cluster is kept constant at 320 000. In this case, we aim at measuring the impact of the I/O request size on the throughput.

In the following, we provide an analysis of the read/write performance for the previous test cases in a variety of configurations. Specifically, we measured differences in using different DAOS object classes (SX and RP_XSF) and compression algorithms (none and zstd). The SX object class forces spreading across all the targets in a pool. Alternatively, RP_XSF uses many replicas so that it is highly scalable for fetching data. The local filesystem used for the tests is XFS on an Intel Optane NVMe SSD.

5.1 Performance analysis for one OID per page

In this section, we evaluate the read/write performance for the one OID per page case. The size of the LHCb dataset for these tests is 1.5 GB. Figure 5 shows the throughput obtained for different cluster sizes while the page size is kept constant, i.e., the number of pages per cluster that can be read in parallel is increasingly larger. As can be seen in Fig. 5.b and Fig. 5.d, the libdaos-based backend peaks at 1.66 GB/s for 160 000 elements per cluster. Given a small page size of less than 80 kB, a larger cluster size is detrimental for reading, i.e., issuing many small I/O requests negatively impacts the read throughput. Also, using the RP_XSF object class results in noticeable benefits, as it increases the read throughput at the cost of a slightly degraded write performance (see Fig. 5.a and Fig. 5.b).

Figure 6 shows the measured throughput in case of increasing the page size while keeping the cluster size constant at 320 000. As can be observed, the DAOS backend benefits from using larger I/O sizes which contributes to improve the read/write throughput or even surpass local SSDs. It is also worth mentioning that the proposed backend outperformed dfuse in all situations. Despite the promising results (best case is 2.22 GB/s), the read throughput is still far below the 12.3 GB/s measured by the IOR [12] benchmark tool using a transfer size of 4 MB and 16 processes.
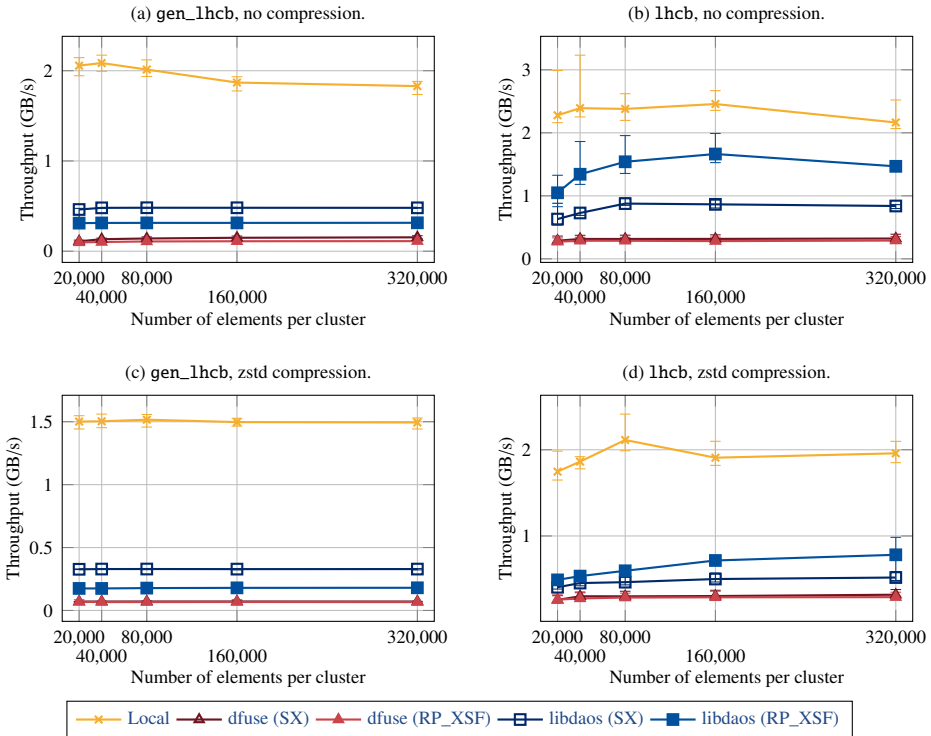
Figure 5. One OID per page, constant page size and increasing cluster size. The lhcb and gen_lhcb programs measure the read or write throughput, respectively.

5.2 Performance analysis for one OID per cluster

In this section, we compare the performance for one OID per cluster w.r.t. one OID per page. Figure 6 shows the impact of larger reads on the performance. As of 160 000 elements per page, i.e. around 1 MB reads, the remote I/O to DAOS matches the speed of a local NVMe device. As can be seen in Figure 7, using one OID per cluster slightly improves the read throughput for a given object class. An explanation for this is that having a non-constant distribution key leads to a more even distribution of data among targets. However, the benefits seen in the write throughput are negligible in this case. Note that measurements for "OID/cluster (RP_XSF)" were prohibited by a known problem in the libpsm2 library.
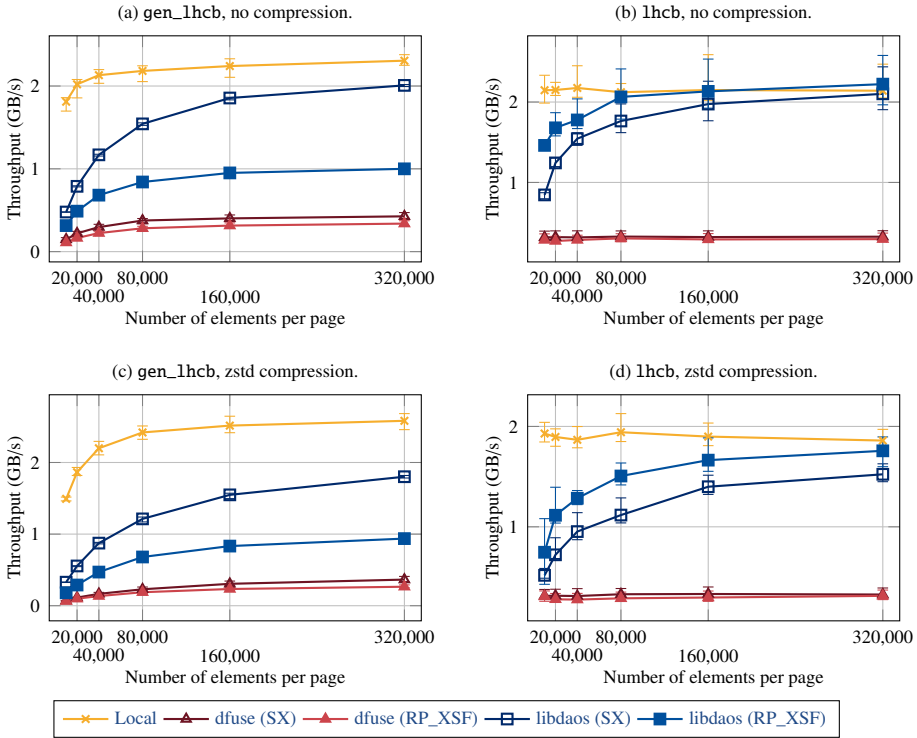
Figure 6. One OID per page, increasing page size and constant cluster size. The `lhcb` and `gen_lhcb` programs measure the read or write throughput, respectively.
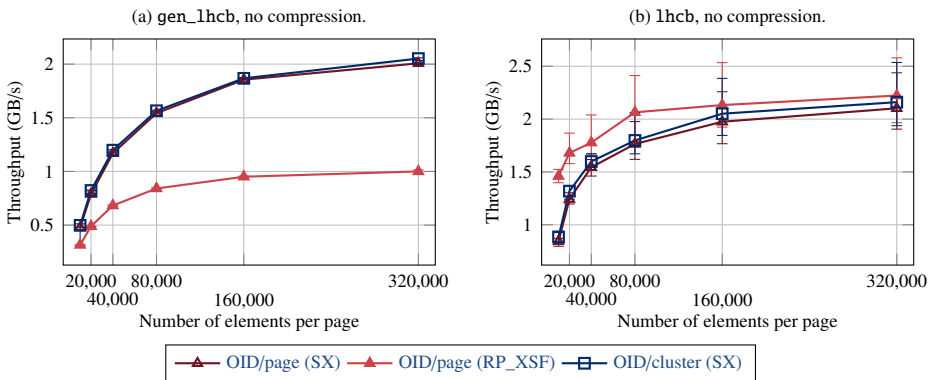

Figure 7. Comparison of OID/page and OID/cluster, increasing page size and constant cluster size (320 000). The `lhcb` and `gen_lhcb` programs measure the read or write throughput, respectively.

6 Conclusion

In this paper, we present an extension to RNTuple which allows users to store HEP data in a DAOS container, therefore closing the gap between the HEP community and the next-generation HPC centers. From the user point of view, switching to a different storage backend can be performed with minimal efforts, only requiring to change a filesystem path to an URI.

As demonstrated in the evaluation, applications benefit of using the dedicated backend, which outperforms the DAOS dfuse filesystem in all cases. Nevertheless, we were unable to reach the expected peak performance as reported by the IOR benchmark.

As a future work, we aim at investigating the performance issues found in the experimental evaluation. Also, we plan to analyze the impact of an alternative mapping that makes use of the attribute key, i.e., the OID and distribution key shall be derived from the RNTuple cluster index and column identifier, respectively, while the attribute key is used to address individual pages. Finally, we plan to provide an efficient mechanism to transfer data from HEP storage to DAOS.

Acknowledgements

References

[1] R. Brun, F. Rademakers, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **389**, 81 (1997)

[2] J. Blomer, P. Canal, A. Naumann, D. Piparo, *Evolution of the ROOT Tree I/O*, in *EPJ Web of Conferences* (EDP Sciences, 2020), Vol. **245**, p. 02030

[3] *DAOS: Revolutionizing High-Performance Storage with Intel(R) Optane Technology*, https://www.intel.com/content/www/us/en/high-performance-computing/daos-high-performance-storage-brief.html (2019), accessed: 2021-02-28

[4] *Aurora | Argonne Leadership Computing Facility*, https://alcf.anl.gov/aurora (2021), accessed: 2021-06-16

[5] *Amazon Redshift - Cloud Data Warehouse*, https://aws.amazon.com/redshift/ (2021), accessed: 2021-02-28

[6] D. Vohra, *Apache Parquet* (Apress, Berkeley, CA, 2016), pp. 325–335, ISBN 978-1-4842-2199-0

[7] *Apache Arrow*, https://arrow.apache.org/ (2021), accessed: 2021-02-28

[8] M. Folk, G. Heber, Q. Koziol, E. Pourmal, D. Robinson, *An Overview of the HDF5 Technology Suite and Its Applications*, in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases* (Association for Computing Machinery, New York, NY, USA, 2011), AD '11, p. 3647, ISBN 9781450306140

[9] *HDF5 DAOS VOL connector*, https://github.com/HDFGroup/vol-daos (2021), accessed: 2021-02-28

[10] P. Braam, *The Lustre storage architecture* (2019), arXiv preprint arXiv:1903.01955

[11] *ROOT RNTuple Virtual Probe Station*, https://github.com/jblomer/iotools/tree/master (2021), accessed: 2021-02-28

[12] H. Shan, J. Shalf, *Using IOR to analyze the I/O performance for HPC platforms* (2007), Ernest Orlando Lawrence Berkeley National Laboratory, Berkeley, CA (US)