UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

GIOVANNA LAZZARI MIOTTO

Leveraging Object Stores for Particle Physics Analysis with RNTuple

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Science

Advisor: Prof. Dr. Claudio Resin Geyer
Coadvisor: Dr. Javier López–Gómez

Meyrin, Switzerland
September 2023

It isn't that life ashore is distasteful to me. But life at sea is better.
(Sir Francis Drake)

ACKNOWLEDGMENTS

This work took a village. First and foremost, it is dedicated to the villagers who persisted in protecting higher education and science in Brazil and abroad against adversity. I can only name a subset of those who have shared their time, knowledge and laughter with me along my trajectory and motivated me toward my goals.

Ai miei genitori – è impossibili ringraziarvi per tutta una vita di amore e sostegno incondizionati. Quest'opera è tanto vostra quanto mia.

To my advisor Claudio Geyer, thank you for your support and unyielding encouragement over the years, without which this thesis would not have been possible.

To Javier López and Jakob Blomer for the perplexing depth of your knowledge and enthusiasm to share it, which has unequivocally elevated this work. You, along with Axel Naumann and Philippe Canal, made ROOT I/O my second family.

To Laura Promberger, who pushed me beyond my comfort zone and brought invaluable advice to this thesis (rarely solicited but always astonishingly pertinent).

Aos professores e mentores que guiaram minha trajetória acadêmica: Ana Bazzan, Bruno Castro, Mariana Kolberg, Lucas Schnorr. A Marcelo Walter, Rodrigo Machado, Sergio Cechin e Danielle Rosa pelo apoio incessante da COMGRAD. Aos meus amigos, pela *camaraderie* incondicional: Arthur, Eduardo, Giovane, Lucas A., Lucas R. e Maria Flávia.

To the brightest and funnest in CERN–EP: Andrea, Bernhard, Elliott, Enrico, Eugenio, Dante, Florine, Harald, Jakob E., Jonas, Jolly, Marta, Vincenzo. Thank you for the coffee breaks, concerts, hikes, swims, Wizard rounds and evenings at Ô'B, in R1, Bari and Norfolk. To my talented partners–in–crime, Baidyanath, Garima, Piyush and Ivan, for the worst judgment and the best stories.

To HPE's Lance Evans, Kevan Rahm and Sridhar Balachandriah and CERN openlab's Krzysztof Mastyna, for the pivotal help with this work's experiments.

This page is incomplete without those who cheered me on through the finish line: Diptarko and Niccolò, for the co–working through all-nighters and summer weekends in Bdg. 40; my inspiring classmates at CSC Tartu - Andreas, Beth, Elias, Eva, Ksenia, Mark and Valeria; and Tom and Emilio, for your faith in me.

This work benefited from support by the CERN Strategic R&D Programme on Technologies for Future Experiments CERN-OPEN-2018-006 and the Intel–CERN openlab collaboration. Access to the hardware for the experimental evaluation was provided by Hewlett-Packard Enterprise and CERN openlab.

ABSTRACT

The advent of the HL-LHC is projected to increase the volume of data generated by LHC experiments for Particle Physics, or High Energy Physics (HEP), by at least an order of magnitude, overwhelming current storage and analysis tools in the field such as CERN'S ROOT. RNTuple is ROOT's new I/O subsystem engineered to leverage modern storage technologies. Object stores are an emerging asset in scalable data storage, with widespread use in cloud and HPC applications. We propose to integrate performant object store backends into RNTuple through two systems with distinct use cases – DAOS and S3 –, in order to exploit, respectively, exascale supercomputing facilities for analyses and a vast storage topology for disseminating data globally and granularly; in each case, the backend should cater to concerns in scalability, efficiency and latency. We introduced to an experimental RNTuple–DAOS backend a set of features designed to capitalize on bulk transfer, target co-locality and our domain knowledge of HEP analysis patterns, with the goal of optimizing throughput for data ingestion and retrieval. Informed by preliminary results regarding data volume, we further propose a zero–copy concatenation method based on scatter–gather I/O that improves transfer speed. A similar approach guided our proof–of–concept RNTuple–S3 backend, adapted to consider latency limitations. We evaluated the two contributions for single–node analyses on performant clusters over InfiniBand RDMA and Ethernet, respectively. RNTuple–DAOS demonstrated high throughput of over 10 GB/s (write) and 4.5 GB/s (read), corroborating our approach. The concatenation mechanism reached double the original write speed and attained the same read speed as the targeted throughput, partially emancipating transfer rates from the layout of the dataset. Furthermore, we validated RNTuple–S3 as a backend for the cloud and developed next strategies for better performance. Equipped with a production–grade DAOS backend for exascale supercomputers and an S3 backend to access a global storage infrastructure, RNTuple is positioning itself as the data format for the next era of HEP research at the HL–LHC and beyond.

Keywords: Particle physics. ROOT. DAOS. S3. high performance computing. distributed systems.

Utilizando Object Stores para Análise em Física de Partículas com RNTuple

RESUMO

A chegada do HL-LHC deve aumentar o volume de dados gerados por experimentos do LHC para Física de Partículas, ou Física de Alta Energia (HEP), em pelo menos uma ordem de magnitude, sobrecarregando atuais ferramentas de armazenamento e análise da área, como ROOT. RNTuple é o novo subsistema de E/S do ROOT, projetado para usufruir de tecnologias modernas de armazenamento. *Object stores* são um recurso para armazenamento escalável de dados usado para nuvem e computação de alto desempenho (HPC). Propomos integrar ao RNTuple *backends* à *object stores* por meio de dois sistemas de usos distintos – DAOS e S3 –, a fim de explorar, respectivamente, centros de supercomputação exaescala para análise e uma vasta topologia para disseminação global e granular de dados; ambos devem atentar para escalabilidade, eficiência e latência. Introduzimos ao *backend* RNTuple-DAOS experimental melhorias que capitalizam transferência em massa, co-localidade e nosso conhecimento de padrões de análise para otimizar a vazão na ingestão e releitura de dados. Informados por resultados preliminares, propomos um método de concatenação sem cópia baseado em E/S *scatter–gather*. Uma abordagem semelhante orientou nosso *backend* prova de conceito, RNTuple–S3, com adaptações para considerar limitações de latência. Avaliamos as duas contribuições em análises nodo–único e em *clusters* de alto desempenho conectados por InfiniBand e Ethernet, respectivamente. RNTuple–DAOS apresentou alta vazão, com picos acima de 10 GB/s (escrita) e 4,5 GB/s (leitura), o que corrobora nossa abordagem. A concatenação atingiu o dobro da velocidade de escrita original e a mesma velocidade de leitura que a vazão–alvo, parcialmente emancipando o desempenho do *layout* de dados. Além disso, validamos o RNTuple–S3 como *backend* para a nuvem e desenvolvemos estratégias para atingir alto desempenho. Equipado com um *backend* DAOS maduro para supercomputadores exaescala e um *backend* S3 para acessar uma infraestrutura de armazenamento global, RNTuple se posiciona como o formato de dados para a próxima era de pesquisa em HEP, no HL–LHC e além.

Palavras-chave: física de partículas. ROOT. DAOS. S3. computação de alto desempenho, sistemas distribuídos.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ACID	Atomicity, Consistency, Isolation, Durability
AGC	Analysis Grand Challenge
ALICE	A Large Ion Collider Experiment
ANL	Argonne National Laboratory
AOD	Analysis Object Data
API	Application Programming Interface
ATLAS	A Toroidal LHC ApparatuS
AWS	Amazon Web Services
BASE	Basically available, Soft-state, Eventual consistency
CAP	Consistency, Availability, Partition tolerance
CERN	European Organization for Nuclear Research
CMS	Compact Muon Solenoid
DAOS	Distributed Asynchronous Object Store
DBMS	Database Management System
DFS	DAOS File System
DMA	Direct Memory Access
DRAM	Dynamic Random Acces Memory
FUSE	Filesystem In Userspace
HDD	Hard Disk Drive
HDF5	Hierarchical Data Format version 5
HDFS	Hadoop Distributed File System
HEP	High Energy Physics
HL-LHC	High Luminosity Large Hadron Collider
HPC	High Performance Computing
HTTP	Hypertext Transfer Protocol
IOPS	I/O operations per second
IOR	Interleaved or Random
KVS	Key-Value Store
LHCb	LHC beauty
LHC	Large Hadron Collider
MPI	Message Passing Interface
NIC	Network Interface Card

NUMA	Non-Uniform Memory Access
NVMe-oF	Non-Volatile Memory express over Fabrics
NVMe	Non-Volatile Memory express
NVM	Non-Volatile Memory
OFI	Open Fabrics Interfaces
PCIe	Peripheral Component Interconnect Express
PMem	Persistent Memory
POSIX	Portable Operating System Interface
RDMA	Remote Direct Memory Access
S3	Amazon Simple Storage Service
SCM	Storage-Class Memory
SSD	Solid State Drive
TBB	Threading Building Blocks
UUID	Universally Unique Identifier
VOL	Virtual Object Layer
WLCG	Worldwide LHC Computing Grid
WORM	Write Once, Read Many
I/O	Input/Output
DTR	Data Transfer Rate

CONTENTS

1 INTRODUCTION

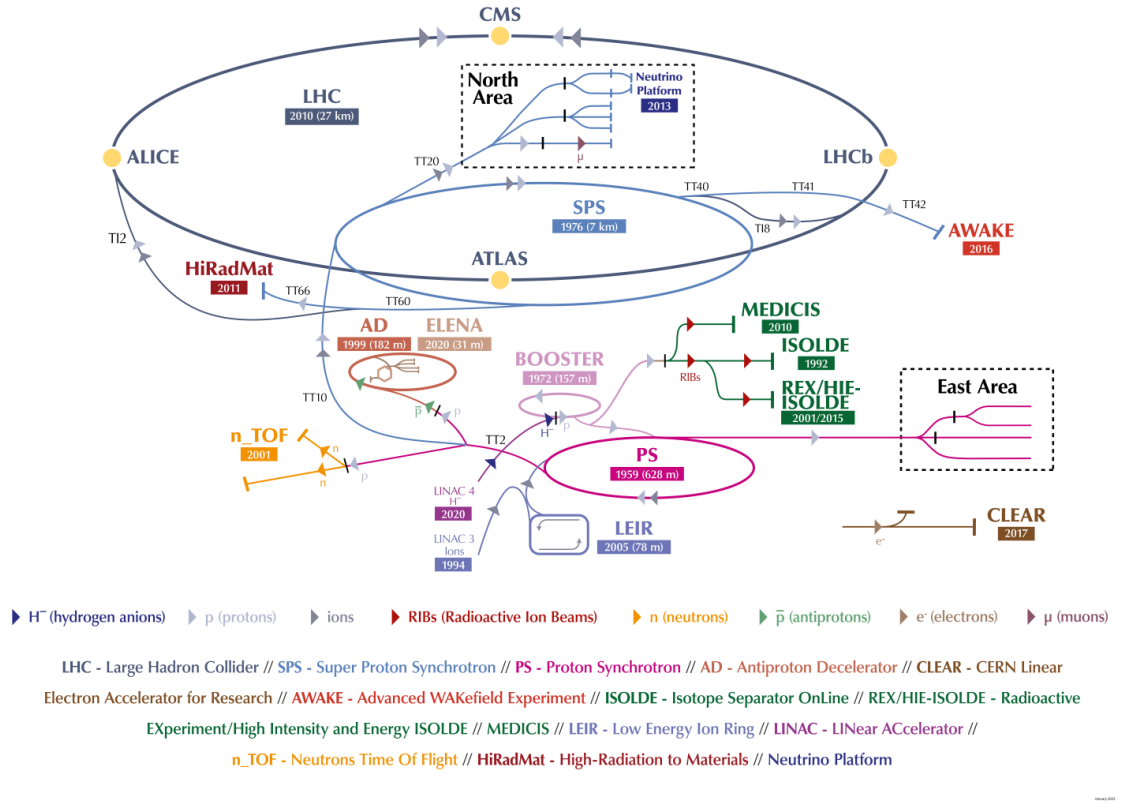
The field of High Energy Physics is both a beneficiary and a benefactor of groundbreaking technological progress in Computer Science. This chapter provides context on some of its current and future challenges that motivated our work and presents our guiding hypotheses. Following that is a brief description of the structure of this thesis.

Context

High Energy Physics (HEP), or particle physics, studies elementary particles and their interactions at the subatomic level in order to further our theoretical understanding of matter at the smallest and largest scales of the universe.

For half a century, the most coherent set of equations to describe the fundamental forces behind particle interaction has been the Standard Model. The

Figure 1.1: The CERN Accelerator Complex in 2022. The points indicate where each experiment's detector is stationed.



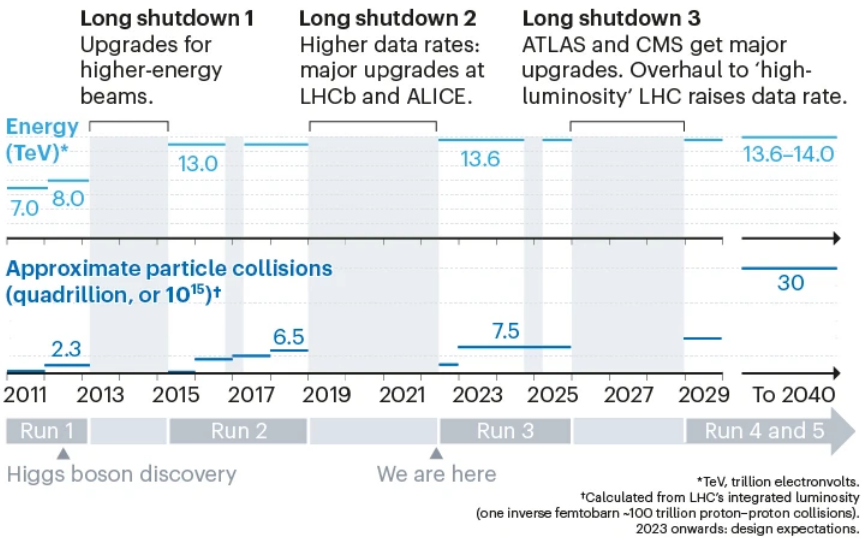
Source: Ewa Lopienska, February 2022 (adapted).

prevailing way to gather experimental data on the Standard Model is through colliding subatomic particles at high speeds. Such an impact decays the protons into more basic subatomic particles, e.g., quarks and gluons, from whose interaction a gamut of short-lived, composite particles may form. The higher the acceleration between the initial particles, the more energy involved in the collision, increasing the chances that heavier particles, such as bosons, be produced in the aftermath.

Currently, the largest particle collider in the world is the Large Hadron Collider (LHC), with a circumference just shy of 27 kilometers intersected by the French-Swiss border near Geneva. The LHC is maintained by the European Organization for Nuclear Research (CERN). Collisions on the LHC are studied by four main experiments, namely, A Large Ion Collider Experiment (ALICE), A Toroidal LHC ApparatuS (ATLAS), Compact Muon Solenoid (CMS) and LHC beauty (LHCb), as pinpointed in figure 1.1. The much-publicized observation of the Higgs' boson, whose existence was experimentally validated in 2012 (AAD et al., 2012), was a result of CMS efforts during the first run of the LHC.

Such findings come from analyzing data picked up by sensors in the detector, which is then digitized, filtered and stored in tape and disk. Figure 1.2 presents a timeline of LHC operations, or *runs* interspersed with *long shutdowns*, showcasing the increase in energy, and thus particle collisions, since its inaugural startup. The larger capacity for collisions, along with more powerful detector technology, directly

Figure 1.2: LHC timeline (2011–2040), including runs, shutdowns and capacity in energy and number of particle collisions



Source: Gibney (2022).

leads to more data that can be analyzed by researchers. Estimates place the volume of data stored by all the LHC experiments at 1 petabyte of raw data per day of operation during its second run (2015–2018); the third and fourth runs (respectively, 2022–2025 and 2029–2032) are expected to handily surpass these numbers (CERN, 2023; BOCKELMAN; ELMER; WATTS, 2023).

For decades, CERN has conceived in-house software solutions for its computing needs, designed and optimized for the usability and performance requirements of particle physics data management and research. ROOT (BRUN; RADEMAKERS, 1997) is the foremost data analysis framework for the HEP community, with hundreds of contributors and thousands of daily users worldwide (ROOT Project, 2023a).

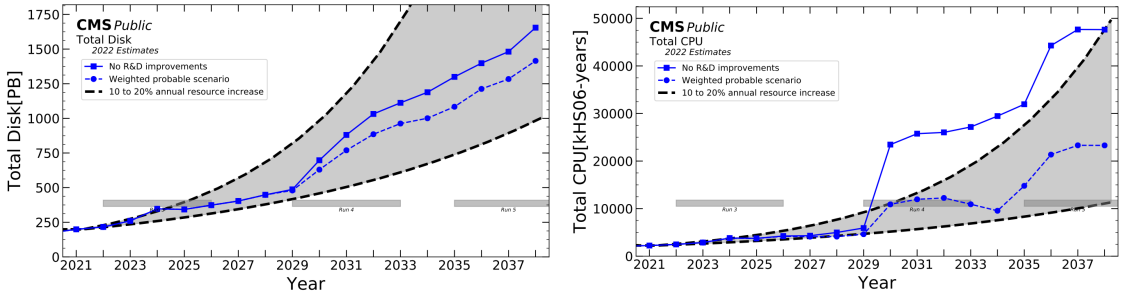
The pattern and scale of HEP data drove the development of ROOT's own data format and Input/Output (I/O) subsystem TTree. After a quarter–century of evolution and over one exabyte of data cumulatively stored in the format, TTree has established itself as the de facto standard format in the HEP community, outperforming other formats for typical HEP analysis workflows (BLOMER, 2018).

Motivation

As figure 1.3a shows, the High Luminosity Large Hadron Collider (HL-LHC) is projected to generate at least ten times as much collision data for end-user analyses as past experiment cycles in the LHC, reaching the hundreds of petabytes per year of operation. This context has motivated significant efforts by the HEP computing community to adequate existing tools and leverage cutting-edge technologies in software and hardware. One such push is IRIS–HEP's Analysis Grand Challenge (AGC), comprising performance objectives in critical steps of HEP analysis, e.g., infrastructure, data access, particle collision selection, and statistical model building. ROOT figures among those projects contemplated by the AGC (BOCKELMAN; ELMER; WATTS, 2023).

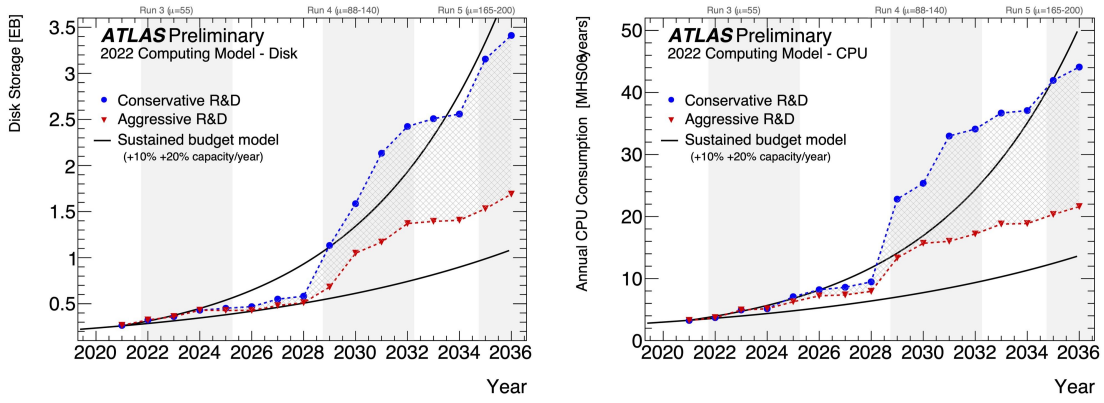
In order to store and analyze data at such a scale, comparable to the biggest players in big data (1.4), it is necessary that distributing computing become the native theater for HEP analysis. Although researchers have benefited since Run 1 from the scale-out orchestration of the Worldwide LHC Computing Grid (WLCG), currently servicing over two million jobs daily (WLCG, 2023), usability roadblocks com-

Figure 1.3: Projected landscape on disk storage and compute needs by LHC experiments and their estimated capacity in different funding scenarios



(a) CMS disk storage in petabytes (left) and CPU use in kHS06 [1](right).

Source: CMS Offline Software and Computing (2022).



(b) ATLAS disk storage in exabytes (left) and CPU use in MHS06, or 10^3 kHS06 (right).

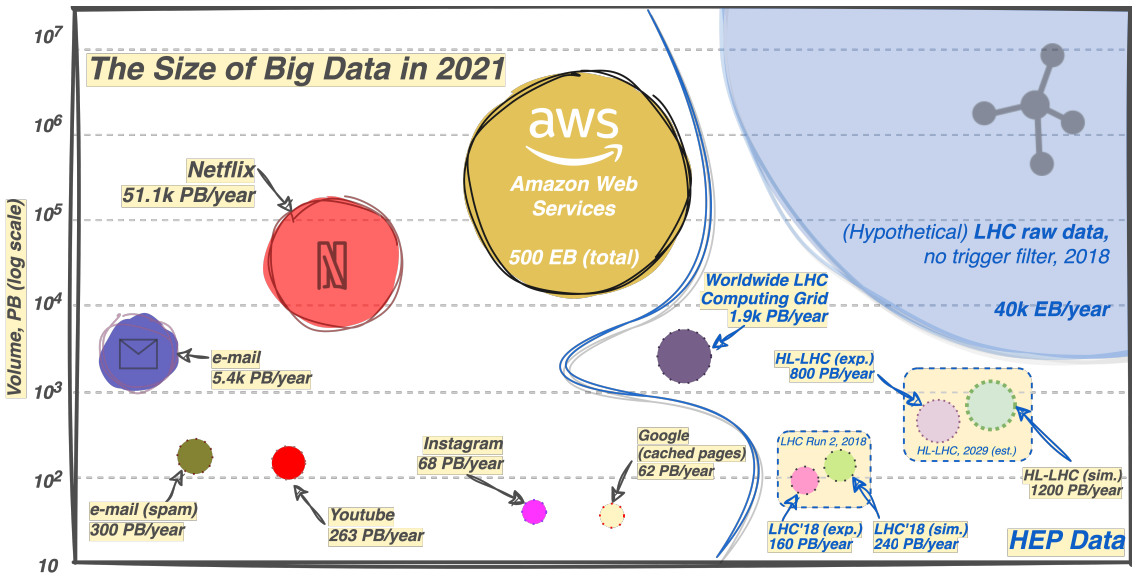
Source: ATLAS Collaboration (2022).

mon to grid computing limit its effectiveness in facilitating HEP research. Such obstacles come from the researcher's need to manually divide-and-conquer large tasks by splitting datasets, designating resources, and aggregating the results through hand-crafted, shell-based scripts (LüTTGAU et al., 2018; PADULANO et al., 2022).

At the same time, throughout the last decade, the logistical challenges of big data management have evoked paradigm-shifting developments in distributed and parallel processing, e.g., object store-based cloud and High Performance Computing (HPC) data centers and Storage-Class Memory (SCM). Two notable technologies that embody such a paradigm change are the ubiquitous Amazon Web Services (AWS) Amazon Simple Storage Service (S3) cloud topology and the Distributed

[1]HS06 is a customary unit in high energy physics to represent computation resources, stemming from the SPEC06 benchmark; 1 kHS06, or 10^3 HS06, corresponds approximately to the compute power of 100 CPU cores as of 2012 and has been equated to 1 teraflop (BERGHöFER et al., 2015).

Figure 1.4: The size of Big Data in 2021. Note the real and projected data volumes for LHC projects, such as the WLCG and the HL-LHC.)



Source: The Authors (adapted from Clissa (2022)).

Asynchronous Object Store (DAOS) (DAOS Project, 2023a; LIANG et al., 2020). DAOS is a low-latency, high-throughput, high I/O operations per second (IOPS) object storage system that is the basis of Intel's exascale stack for HPC applications, leveraging Non-Volatile Memory express (NVMe) devices and SCM (also referred to as persistent memory). DAOS is present in a plurality of the top performing HPC clusters in the IO500 list (IO500 Foundation, 2022) and is the underlying filesystem for Argonne's Aurora Exascale Supercomputer (Argonne National Laboratory, 2023).

However, TTree was not designed to facilitate native, fine-grained integration with object stores. To address this and other concerns ahead of the HL-LHC, ROOT has been developing RNTuple, an experimental I/O subsystem designed from scratch with modern storage technologies and principles in mind. RNTuple's architecture is modular, allowing uncomplicated extensions for, e.g., new data types and storage backends. With this backend agnosticism comes an unprecedented opportunity to explore object store support for HEP analyses and tap into a widespread infrastructure of opportunist and specialized facilities. Through RNTuple, it becomes possible to leverage both a global infrastructure of cloud storage facilities and a number of exascale HPC data centers for efficient, distributed analysis of HL-LHC data.

Thus, extending RNTuple to support a production-grade native DAOS back-

end could be the first step to enlist the power of exascale supercomputers toward efficient, distributed ROOT analyses, while entering the cloud scene will provide researchers with a scalable, fine–grained access to HEP datasets for the HL-LHC era and beyond.

1.1 Hypotheses

In light of the requirement to attain a much higher I/O throughput for HEP analysis to fully benefit from HPC data centers, we identify the following research questions:

Hypothesis 1. Domain-specific knowledge of HEP analysis access patterns and the DAOS storage model can improve RNTuple's single-core I/O performance and throughput to DAOS object stores.

Hypothesis 2. Applying a similar approach, RNTuple can intercommunicate with other object stores developed for different use cases, such as AWS S3 and other cloud-based storage providers.

1.2 Objectives and Methodology

Objective 1. Identify approaches that improve DAOS integration into RNTuple with regards to throughput.

- **1a.** Implement a more informed data mapping based on domain knowledge of HEP analysis, RNTuple and DAOS.
- **1b.** Evaluate the role of certain parameters under our control: page and cluster sizes, replication and sharding levels.
- **1c.** Develop a page concatenation mechanism for RNTuple inter-backend portability.

Objective 2. Based on the experience from the above objective, propose an RNTuple backend for the cloud (AWS S3).

- **2a.** Implement a proof-of-concept S3 backend that addresses cloud concerns, subject to evaluation.

Methodology. The above objectives **1a–c** and **2a** are met by means of raw read and write throughput (defined in 2.5) in the context of an I/O-intensive, realistic benchmark for HEP analysis, as detailed in 5.2.

Main Contributions. We identify two especially noteworthy contributions in this work:

1. An RNTuple-DAOS data mapping that exploits data co-locality in a storage node's target, storing together content that is typically fetched together by HEP analyses;
2. A scatter-gather concatenation mechanism that enables transfer between backends at higher speeds, irrespective of the ntuple's native layout.

1.3 Structure

This thesis is structured as follows. Chapter 2 introduces the necessary foundations to grasp the proposed method and ensuing evaluation. Chapter 3 provides a brief survey on the state of the art that, to the best of our knowledge, eclipses the body of related work at the time of writing. Chapter 4 presents a conceptual model of our proposed solutions custom–made for DAOS and S3. Chapter 5 shows the conducted experiments and contains an evaluation of our findings. Chapter 6 summarizes the findings and contrasts them to the objectives and hypotheses enumerated in the introduction. The bibliography, which lists publications and materials that informed this work, follows suit. Finally, a glossary of domain-specific terms closes the thesis.

2 BACKGROUND

This chapter explains useful concepts for the remainder of this thesis. Section 2.1 introduces the ROOT framework in the context of HEP analysis. Section 2.4 lays out the foundational elements of POSIX-compliant filesystems and object stores, with an emphasis on their importance for scalability in distributed systems. In section 2.3, we introduce DAOS and its properties. In section 2.5, we cover basic concepts of High Performance Computing and its technologies, such as distributed memory and interconnection networks.

2.1 High Energy Physics and its Analysis Patterns

High Energy Physics is a computationally intensive field with unique performance and usability requirements. Given shortcomings of generic solutions found in industry, physicists have long depended on domain-specific analysis tools fine-tuned for their use cases to accelerate their research.

This section introduces useful concepts of Particle Physics and presents insights into how data from Large Hadron Collider experiments is generated and analyzed. Subsection 2.1.1 offers an abridged description of the lifetime of a collision event, from beam acceleration to data point. Subsection 2.1.2 discusses the particularities of HEP analysis and its prevailing data access patterns.

2.1.1 Large Hadron Collider Data

The object of study for HEP analysis is the event. Henceforth, we refer to an event as a collision between accelerated groupings of protons launched together – proton bunches –, described in terms of certain properties and the collateral particles that decayed from the collective impact.

Experiments at the LHC can now yield billions of collisions every second. These originate from accelerating pairs of proton bunches in opposing directions at a frequency of 40 MHz. As each event generates around 1 MB of signal data, current LHC experiments can spawn information in the order of petabytes per second (PB/s).

However, a majority of this information reiterates well-understood phenomena that do not advance the field of particle physics; thus, a complex filtering and selection pipeline exists in order to keep analysis data tractable.

The first step in the pipeline, *data acquisition*, involves highly selective, online triggers that filter out sensor readings corresponding to ordinary events. These trigger mechanisms are implemented as layers in hardware and software to keep storage latency as low as possible (ARDINO et al., 2023). After culling event data by five orders of magnitude, the *event reconstruction* step pieces out particle trajectories from various sensor readings to describe particle interactions throughout their lifetimes post-collision.

Reconstruction-originated datasets are organized in custom data formats, with degrees of compactness optimized for the analysis needs of each experiment. Since the LHC's first run, CMS analyses have been based on the Analysis Object Data (AOD) format, an extract from reconstruction data. Each AOD event occupies about 400 kB in memory. On subsequent runs, more compact formats, designed for the use cases of a majority of analysis workflows, have spawned from the AOD: the MiniAOD (35–60 kB per event) and NanoAOD (1–2 kB per event) layouts (PERUZZI et al., 2020).

In parallel, artificial datasets are painstakingly generated through *Monte Carlo simulation* frameworks such as Geant4 (AGOSTINELLI et al., 2003). Statistical processes are informed by theoretical formulae to be validated or rejected by real-world experiments.

Each entry in LHC datasets lists a collision event and its noteworthy aftereffects. Since each event relates to a separate proton bunch crossing, entries are treated as statistically independent data points, conditioned on experimental context.

Two events can have different outcomes in terms of the particles they generate and the features of interest to the researcher. Thus, HEP datasets have a mutable schema and store data in jagged arrays, leading to support of these less common mechanisms in HEP-fomented tools, e.g., ROOT.

2.1.2 Data Analysis for High Energy Physics

HEP Analysis Workflows

In HEP analyses, physicists are interested in validating or falsifying theories on particle interaction. To that end, analysis workflows typically compare ground truth observations reconstructed from the experiment detectors to data generated from Monte Carlo simulation processes that follow such theoretical models.

As with any data-intensive statistical analysis, it is critical to establish features of interest to filter out irrelevant information. These *cuts* may target entire feature columns or data points failing a given threshold, leaving the physicist with only the interesting subset of events from the dataset. These events' features of interest are then processed to generate custom observables in a step akin to feature engineering. Finally, the physics observables are aggregated and summarized, commonly in binned histograms, so that their distribution patterns can be interpreted through statistical inference methods. The column-oriented design of HEP data formats facilitates such analyses centered around features of interest.

LHC Analysis at Scale

With a volume of LHC data in the exabytes, HEP data analysis requires vast computation and storage resources that surpass the capacity of any single node. On the other hand, since an experiment's events are statistically independent, its workloads fall in with other computationally–intensive tasks called "embarrassingly parallel" that are ideal targets for divide–and–conquer strategies, e.g., for distributed computing across nodes working in unison.

Before the LHC's first run, the Grid (FOSTER; KESSELMAN; TUECKE, 2001) was already under development to overcome the logistical challenges ushered by data outflows of its projected scale. The Grid defines a virtual, resource–sharing organization able to orchestrate a heterogeneous architecture of computing resources across collaborating facilities globally. The WLCG (WLCG, 2023) is the Grid's implementation that concerts the combined efforts of HEP research contributors toward more efficient LHC analyses. Currently, the WLCG counts over 1.4 million cores and 1.5 exabytes of storage around the globe.

2.2 The ROOT Framework

This section introduces the ROOT framework (ROOT Project, 2023a) and its role for HEP data analysis. For brevity, we give more meticulous attention only to its I/O subsystems TTree and RNTuple, despite the toolkit's wide range of features.

Overview

ROOT (BRUN; RADEMAKERS, 1997; ANTCHEVA et al., 2009) is a general-purpose software framework developed at CERN to provide the petabyte-scale storage and efficient data processing required for high-volume scientific analyses in various fields of study, including HEP (KHACHATRYAN et al., 2014), genomics (GIANNUZZI et al., 2011) and medicine (GREVILLOT et al., 2011). The four main experiments at the LHC (ATLAS, ALICE, CMS and LHCb) are among ROOT's most prominent users; as such, the framework's design is heavily influenced by their particular requirements.

As seen in chapter 1, the scale at which LHC data generation occurs calls for specialized software. Over a quarter-century, ROOT has cemented itself as the main language of HEP storage and analysis.

2.2.1 ROOT I/O

HEP analyses typically target only a subset of arbitrarily-formed observables across the entire range of events derived from an experiment's collision (HARTMANN; ELMSHEUSER; DUCKECK, 2021). As such, ROOT's I/O subsystem subscribes to a columnar data storage paradigm, in addition to its support for arbitrary types and collections in C++.

Though TTree I/O speed and storage performance has been demonstrated to topple other formats like HDF5 and Apache Parquet in the context of HEP analyses (BLOMER, 2018), its near three decade long design carries shortcomings. The most critical of them are optimization choices intending to hide *seek* latency in spinning disks - pointless in an age of modern, NVMe flash storage systems. The underlying, Portable Operating System Interface (POSIX)-compliant storage backend is built

monolithically, taking major rewriting to adapt to other backends.

TTree's performance bottlenecks may hinder analysis workflows starting with LHC's Run 4, once the planned HL-LHC upgrade is completed (High Luminosity LHC Project, 2022). This occasion is expected to usher an increase of at least an order of magnitude in the volume of data the LHC generates. Such an influx rate would be untenable for TTree to handle, instigating its gradual retirement in favor of a new, canonical data format for the future of HEP storage.

??, named for *nested tuple*, is the backwards-incompatible evolution of TTree touted to address the latter's shortcomings. In addition to modernized software design principles that promise adaptability to shifting requirements, it brings a smaller on-disk representation, more robust interfaces and type-safety by default. To accelerate early adoption, there are ongoing developments to expand ??'s feature set and eclipse TTree's, e.g., integration with ROOT RDataFrame and a planned, minimal I/O API in C, christened RNTuple Lite, which will enable ntuple migration to existing machine learning stacks (BLOMER et al., 2020).

?? subscribes to a columnar layout that is reminiscent of design characteristics seen in TTree and Apache's Arrow (The Apache Software Foundation, 2023a) and Parquet (The Apache Software Foundation, 2023b).

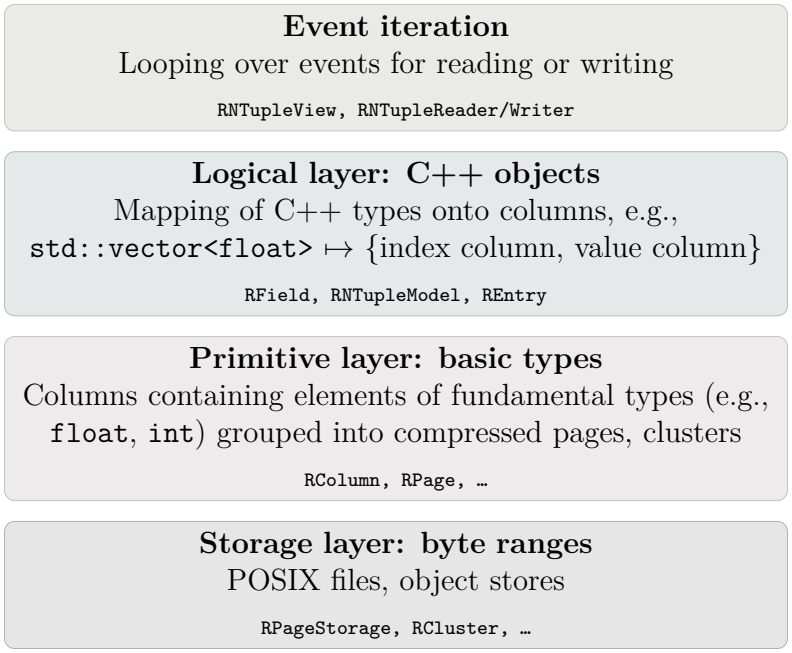
2.2.2 RNTuple Architecture

As described in subsection 2.2.1, ?? is a proposed, modernized format built for the next generation of LHC experiments, while inheriting decades of domain knowledge from TTree.

RNTuple's architecture comprises four separate and functionally distinct layers. The event layer offers a user interface to read and write collision event entries, either through hand-crafted event loops or through ROOT's declarative interface for analysis RDataFrame. The logical layer maps C++ objects to their corresponding columns of fundamental types. Objects may be custom structures that are fully composable and that include arbitrarily–nested collections, e.g., dictionaries and vectors of vectors. The primitives layer coalesces ranges of values of the same fundamental type into RNTuple pages. Lastly, the storage layer implements the concrete backend that handles I/O operations for pages, clusters and their metadata (LOPEZ-GOMEZ; BLOMER, 2021).

Figure 2.1 illustrates RNTuple's high-level architecture. The layered design makes it simple to cater to shifting requirements and technologies, e.g., including support for new data types and storage backends.

Figure 2.1: Layers of the RNTuple subsystem



Source: The Authors (adapted from Lopez-Gomez and Blomer (2021)).

2.3 DAOS

The Distributed Asynchronous Object Store (DAOS) is an open source, high performance object store designed for massively distributed Non-Volatile Memory (NVM) devices (LIANG et al., 2020; DAOS Project, 2023c). It is the foundation of Intel's exascale storage stack, running on Argonne National Laboratory (ANL)'s Aurora supercomputer.

In this section, we provide an overview of DAOS' architecture, storage and data models, with emphasis on key design choices that make DAOS different compared to opportunist object stores, and particularly suited for HPC applications.

2.3.1 System Architecture

DAOS is designed from the ground up to exploit HPC architectures based on modern storage systems with low latency, byte-granular access and high throughput interconnects. As such, it breaks away from traditional I/O models by not supporting high latency, block-based disks.

Its architecture is defined by three "building blocks": Persistent Memory (PMem), NVMe and `libfabric`. PMem, as a form of SCM, is significantly faster than Solid State Drives (SSDs) (see section 2.5). PMem is the storage destination for all internal metadata and as a stage for critical small I/O operations, where they undergo aggregation before being committed. For this reason, PMem must account for at least 6 % of total storage as of DAOS 2.2 (DAOS Project, 2023d). For bulk transfers and noncritical small I/O, NVMe SSDs are supported (and recommended, due to PMem storage costs). Lastly, `libfabric`, or Open Fabrics Interfaces (OFI) (OFIWG, 2023), is a low-level communication library for HPC. In the context of DAOS, it provides low-latency integration with underlying fabric communication hardware.

The absence of system calls after spinup contributes to DAOS' lower latencies. This is achieved by memory mapping PMem storage and the employment of user-level libraries and kernel-bypassing remote transfer protocols like Non-Volatile Memory express over Fabrics (NVMe-oF). Though this makes DAOS very lightweight, applications that expect the POSIX I/O API must go through DAOS File System (DFS) compatibility middleware, e.g., `dfuse` (user space mount) and `libioil` (I/O intercept).

2.3.2 Storage and Data Models

The data organization in DAOS is unique to object stores. In this subsection, we detail how data and metadata are managed in DAOS and how its rich interfaces grant the user a distinctive degree of control on data locality, distribution and integrity, contrasting to other popular solutions in the field.

The schematic in figure 2.3 exemplifies DAOS' stratified storage model, from the pool of physical storage targets, to its potential hundreds of independent, namespace containers, to their potential billions of objects, each capable of holding an

extremely large number of data values.

Pool

The pool represents a reservation of storage distributed across a subset of the physical targets that make up a server's storage nodes. Once allocated, the pool's storage space in a given target are known as one of the pool's shards.

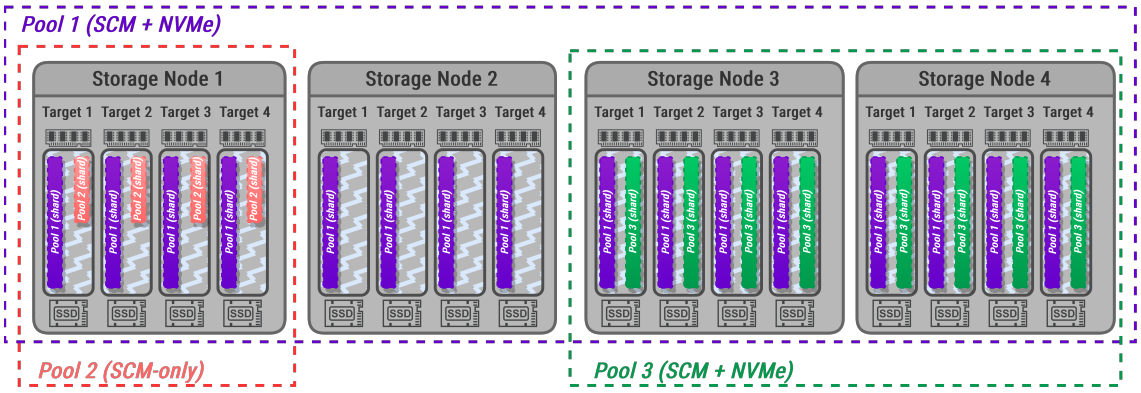
In DAOS, a pool can sustain hundreds of object stores - containers - which operate independently of each other. Data pertaining to a container can be spread across pool targets to achieve a particular level of resilience and I/O performance.

Container

A container denotes a private address space within the pool that is able to hold in the order of 10^{28} unique data-containing objects. Containers must be *open*ed by applications connected to the DAOS pool before I/O requests can be issued. Container types, denoting different data layouts, can be implemented on top of the DAOS Application Programming Interface (API) provided through `libdaos`, such as the POSIX-based DFS or ROOT's own middleware featured in this work.

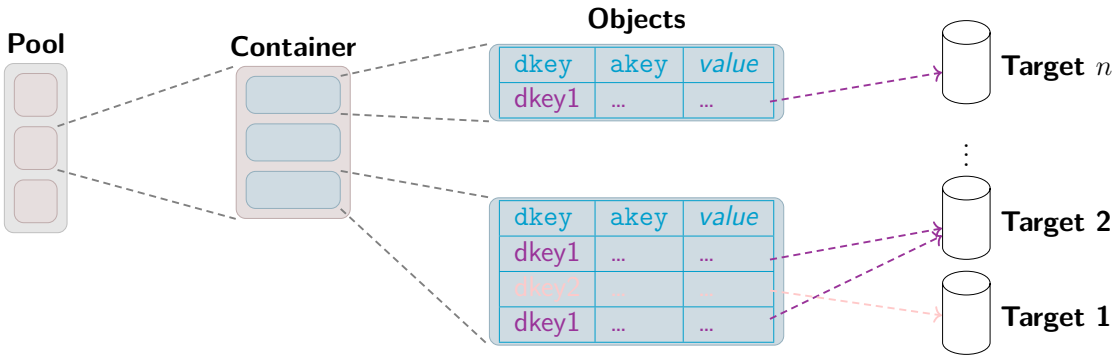
The container is responsible for data versioning and transactional consistency based on epoch timestamps. For permanent references to consistent dataset states, DAOS supports timestamped and immutable snapshots, allowing the entire container to be rolled back. To cope with distributed settings and mitigate contention, multiple I/O operations are combined into a transaction with Atomicity, Consistency, Isolation, Durability (ACID) properties, which is subjected to optimistic,

Figure 2.2: DAOS storage model: the pool abstraction.



Source: The Authors (adapted from DAOS Project (2023b)).

Figure 2.3: A simplified view of the storage model and data organization with the multi-level Key-Value Store API in DAOS.



Source: The Authors.

multi-version concurrency control; versioned updates are eventually aggregated to reclaim space. Concurrency conflicts are settled on the basis of timestamp ordering.

Each container is associated with a Universally Unique Identifier (UUID). Since DAOS 2.0, string labels have become the default interface for container identifiers, matching S3.

Object

A DAOS object is a data partition capable of containing multiple blobs, unlike the most ubiquitous object stores. This entity has two supported paradigms, depending on whether the stored data is structured.

1. The array API turns objects into arrays. Array elements have fixed size, are accessed through an index and can be overwritten individually. Both flat and multi-dimensional arrays are supported.
2. The Key-Value Store (KVS) API turns objects into full-blown key-value stores. Elements, or blobs, have variable size. A key is assigned to access each blob. The complexity of this key depends on the particular KVS interface chosen by the user:

    2a Single-level KVS, with pairs $\langle$`akey`, `value`$\rangle$. This is akin to a traditional object store interface: an attribute key (`akey`) maps to the value.

    2b Multi-level KVS: with pairs $\langle\langle$`akey, dkey`$\rangle$, `value`$\rangle$. Here, the mapping interface has a composite key, where the addition of a distribution key (`dkey`) impact data locality in hardware.

Figure 2.3 exemplifies how data is assigned to storage under the multi-level KVS model. The `dkey` directly determines where data is physically stored in the server. Under the same object, two values sharing the same `dkey` are guaranteed to be mapped to the same node target. This guarantee does not extend to the same `dkey` under different objects, even in the same container namespace. The `akey` completes the composite key to form a unique identifier among object values.

Whatever the underlying paradigm, each object is uniquely identifiable within its container namespace by the object id (`oid`), a 128-bit numerical value. The first 32 bits of the `oid` are reserved for DAOS metadata. The remaining 96 bits are the user's to define.

Object Operations DAOS operations are concentrated on the object. The values within an object are *immutable*, *replaceable* blobs. DAOS is optimized to fetch or update these values as a single request unit. Thus, at this time, partial, byte-range requests are unsupported, either for reading or writing.

Object updates and fetches are triggered by API calls to `daos_obj_update()` and `daos_obj_fetch()`, which take vectors with an arbitrary number of I/O Vectors (IOVs) and Scatter-Gather Lists (SGLs). As such, these operations support Linux' scatter-gather I/O – multiple buffers on the user's machine may be combined into one single data Binary Large Object (blob) server-side. Each blob is sequentially described by a corresponding `akey` stored in the list of IOVs, which inform the total size of each blob.

Table 2.1: A subset of pre-defined object classes in DAOS.

Object Class	Replication Factor	Sharding Factor
`TINY`	1	1
`MAX`	1	"Maximum"
`SX`	1	All
`RP_2G1`	2	1
`RP_TINY`	1	All
`RP_MAX`	"Maximum"	All
`XSF`	12..128	1

Object Classes: DAOS object classes are a shorthand for properties relating to object data redundancy and layout across pool shards. There exist pre-defined object classes optimal for common object store use cases. Custom object class schema can be defined at any time. Table 2.1 contains examples of pre-defined object classes relevant to this work, along with their data protection factor (i.e., replication count) and sharding level (i.e., distribution between the available physical devices).

Metadata Management: Metadata is a known source of contention and scalability concerns in distributed systems (BOITO et al., 2018). DAOS' storage model addresses these concerns by keeping metadata lightweight and low-latency. For this reason, metadata I/O is exclusive to PMem devices, a form of SCM significantly faster than NVMe.

DAOS sacrifices object-level metadata granularity for many details traditionally maintained by file systems. Only object type and schema - regulating striping and replication, for example -, are kept on a per-object basis (DAOS Project, 2023b).

2.4 Data Storage Paradigms

Storage systems comprise the infrastructure and logical abstractions developed for organizing, storing and protecting the correctness of data. Throughout the years, file and database systems have been proposed for efficient data I/O in local and distributed settings. In this section, we provide an overview of traditional and emerging systems and their efforts to manage stored content coherently.

2.4.1 Traditional Approaches

In this work, we refer to traditional data storage systems as the conventional paradigms that predate modern approaches targeting distributed and cloud storage. We briefly discuss three important concepts and their approaches to data management: POSIX files, block storage and transactional databases.

File-Based Storage

File-based systems are organized in a tree-like hierarchy of directories and data-containing files. Its elements are typically saddled with metadata, permissions, timestamps and ownership credentials. Data is accessible by a path location stemming from the root directory. As file-based systems grow in size, managing the hierarchy's metadata becomes cumbersome, which hurts the scalability of such systems. One source of contention in file-based systems is the adherence to the POSIX standards.

Definition (POSIX I/O) A POSIX standard which describes well-defined semantics and interfaces for I/O in file systems across platforms, stipulating stateful file descriptors, atomic operations, inter-process sequential consistency and prescriptive metadata for owner, group permissions and timestamps, among others. It also regulates aspects like buffering and flushing of data to storage, blocking and asynchronous I/O.

Though POSIX I/O semantics provides systems with data integrity and consistency, it comes at the cost of limited scalability in data and concurrency. Some concerns include file-level consistency through locking, blocking I/O by default, centralized metadata management and hierarchical traversal and data placement.

Block-Based Storage

Block-based storage splits content into granular chunks of equal size, called blocks. Each block is stored independently, which enables better hardware utilization and access through block-level addressing.

In contrast to file-based storage, block storage eschews a hierarchical structure and metadata bookkeeping, which can lead to better scalability for applications managing their own data structuring, such as databases. In particular, block storage provides a robust foundation for transactional databases, as block granularity mitigates the effects of contention caused by transactions. However, management and scalability concerns may arise from the need for lookup tables to keep track of the storage blocks to which data are assigned.

Transactional Databases

Transactional databases are designed to ensure reliable and consistent data management. Such databases are based on transactions, or sequences of one or more database operations that are executed reliably in the presence of concurrency or unexpected crashes. Transactions offer guarantees described by four properties better known for their shorthand ACID.

Definition (ACID). A shorthand for four properties of database transactions – Atomicity, Consistency, Isolation, Durability –, whose definitions follow suit. **Atomicity**: a transaction's statements are unitary and indivisible. **Consistency**: the database transitions between consistent states. **Isolation**: concurrent operations act on the system's state as if they were sequential. **Durability**: committed transactions are not lost in case of crashes.

2.4.2 Modern and Distributed Storage Systems

Emerging storage systems propose advanced and specialized concepts in data management and access, such as parallel file systems, columnar databases and object stores that are better suited to distributed settings. Such systems are geared toward scalability and reliability in contexts from big data analytics to HPC.

Parallel File Systems

Parallel file systems distribute files through striping across multiple, block-based storage nodes in a cluster, seeking to achieve fast and concurrent access to data. They represent an especially-designed data-sharing solution to mitigate I/O bottlenecks in parallel data processing. Once requested, data is served concurrently and transparently through separate I/O paths to saturate bandwidth. HPC applications with high data volume typically resort to parallel file systems for these benefits; a prominent example is Lustre (BRAAM, 2019).

Distributed Systems

Distributed systems span multiple networked nodes working in tandem toward common tasks as if a single computing system. Together, these systems can

tackle much larger problems than any single node. Examples of distributed systems include cloud platforms like AWS, Ceph (JEONG et al., 2019) and the Hadoop Distributed File System (HDFS). The latter deploys scale-out file systems consisting of commodity nodes to distribute work, though its tree hierarchies at the node level impact the system's manageability at scale.

As can be seen, for their benefits, distributed systems introduce many challenges to maintain a globally synchronized and consistent state against data concurrency and component failures. For this, modern storage solutions can apply distributed transactions and consistency control, as well as fault tolerance through the replication of the same data across several backup nodes.

Definition (Availability, Partition Tolerance, Consistency): Availability is kept if requests always receive a response within an acceptable timeframe. High availability rates are referred by their number of "nines" (i.e., 99.99 % as "four nines"). **Partition tolerance** means system operability despite arbitrary network latency between nodes. **Consistency** models impose an ordering of operations that maintains a coherent global state between nodes. Some guarantees include "strong consistency" (a read always accesses the most globally recent content) and its relaxation, "eventual consistency" (updates are eventually propagated to every node).

The strong consistency expectations of POSIX-compliant file systems cause I/O bottlenecks for distributed applications. Not only does this cripple scalability for databases where structure is not critical, but it affects concurrent access due to frequent locking. This is supported by the acronymous Consistency, Availability, Partition tolerance (CAP) theorem, presented below.

Theorem 2.4.1 (Brewer's CAP Theorem) *A distributed system may attain at most two out of the three following properties to a rigorous degree at any given time: (strong) consistency, availability and tolerance to networking partitioning.*

Naturally, CAP is only applicable in networked systems. Depending on their purpose, services may offer CAP trade-offs like consistency degradation (e.g., from strong to eventual) in order to keep availability rates high (MUñOZ-ESCOí et al., 2019).

A weaker set of guarantees ubiquitous to distributed systems is BASE ("Basically Available, Soft-state, Eventually consistent"), prioritizing scalable availability

and fault tolerance over strict consistency. Its properties affirm that nodes may temporarily be unresponsive or have inconsistent data views due to concurrent operations and network latency before converging to a consistent state.

Columnar Storage

In column-oriented Database Management System (DBMS), data of a given column is stored sequentially; a column spans several entry records consecutively on disk. This is in contrast with row-oriented DBMS, wherein each entry is an individual, heterogeneous record spanning all columns in the database. Apache Parquet (The Apache Software Foundation, 2023b) is a highly popular and performant columnar DBMS.

According to Abadi, Boncz and Harizopoulos (2009), columnar databases tend to be more compact than row-based approaches in terms of storage footprint, due to the effectiveness of compression algorithms when applied to homogeneous data with low information entropy, e.g., values from the same column.

The on-disk data layout of column stores makes them suitable for large volume, read-intensive applications such as scientific and business analytical workloads: only the columns corresponding to features of interest are accessed, yielding higher effective I/O rates. However, columnar layouts lead to high seek latency by spinning media when serving scattered I/O requests from different columns (HARTMANN; ELMSHEUSER; DUCKECK, 2021; ABADI; MADDEN; HACHEM, 2008).

Object Storage

The reliance of traditional file systems on file hierarchy and lookup tables to access blocks in storage globally slows the system down as the database expands. A more scalable approach is to do away with structure altogether and keep a flat collection of data, each piece associated to a unique key. This is known as a KVS.

Key-value object stores (or "object stores") segment data as self-contained and independent units ("objects") bundled with a custom set of embedded metadata. This benefits navigation and access to objects. The latter is provided through a set of intentionally-simple operations, e.g., `GET`, `PUT` and `DELETE`. As a downside of this limited interface and the lack of block granularity, object stores are better suited for predominantly static, Write Once, Read Many (WORM) databases instead of those

with regularly changing data.

According to Liu et al. (2018a), a key factor behind the scalability of object stores is the self-describing nature of the data; unlike with POSIX I/O schemes, neither a directory hierarchy nor a set of prescribed properties are enforced on objects. Metadata exists as custom tags on objects, favoring flexibility over usability Transactions are kept simple to prevent concurrency locks: objects are immutable and their access is stateless and not descriptor-based. Additionally, strong consistency is often downgraded to eventual consistency guarantees.

Myriad object store solutions exist in the market, such as Amazon Dynamo, Apache Cassandra, Google Spanner, AWS S3, and DAOS. The latter two have been validated in HPC contexts (GADBAN; KUNKEL, 2021; LIU et al., 2018a).

2.5 High Performance Computing

High Performance Computing (HPC) engages powerful, interconnected nodes to tackle intensive tasks efficiently. At the scale of thousands or millions of cores, these nodes make up a supercomputer or HPC cluster that leverages cutting edge components in processing, storage and networking, tuned to operate in parallel with minimal latency and exceptionally high throughput.

2.5.1 Persistent Storage Technologies

Non-Volatile Memory

NVM describes a persistent storage system that retains its contents if power is interrupted. These systems are typically designed for long term, secondary storage, such that latency is a lesser concern than cost and access is block-granular. As such, even NVMs based on NAND flash, such as SSDs, are multiple orders of magnitude slower than Dynamic Random Acces Memory (DRAM), a type of volatile memory.

Storage-Class Memory and Persistent Memory

Despite latency reductions of an order of magnitude introduced by SSDs compared to Hard Disk Drives (HDDs), I/O remains the performance bottleneck in

storage devices. To address this divide between cutting-edge processors and NAND-based storage, a new tier of storage technology, called SCM, has emerged with the potential to make modern storage as fast as the rest of its system (HADY et al., 2017).

SCM denotes solid state storage that share features with both NAND and DRAM devices, serving as a compromise between the two. It is nonvolatile; as such, it is also referred to as PMem. On the other hand, it offers an order of magnitude lower latency than SSDs, while remaining more cost-effective than DRAM (THOMASIAN, 2022).

3D XPoint Storage (Optane)

3D XPoint ("cross-point") is a type of NVM jointly developed by Intel and Micron Technology that is speculated to have a phase-change memory (PCM), transistor-less architecture with a dense layout of stacked memory cells (HADY et al., 2017). The technology has been commercialized under the Intel Optane brand for different use cases. Among them, we note the Optane SSD - a POSIX-compliant storage for block-abstracted, asynchronous I/O interfaces -, and Optane PMem.

2.5.2 Communication Technologies

Remote Direct Memory Access

Remote Direct Memory Access (RDMA) is a network communication technique that generalizes Direct Memory Access (DMA) for networked nodes. In DMA, subsystems can bypass the CPU and directly access the system's storage, as opposed to programmed, memory-mapped and TCP socket-based I/O, where the CPU has more active and computationally expensive roles. Instead, for RDMA and DMA both, the CPU only steps in to grant initial access to the data region and to handle the end of the transfer signaled by the driver. Particularly, the RDMA-enabled host grants its Network Interface Card (NIC) access to application memory so that guest nodes may read and write data without going through the I/O stack at either endpoint (Nvidia, Inc., 2023). Though overhead from setting up endpoint access is significant, its impact is attenuated by much faster transfer rates for bulk data.

InfiniBand

InfiniBand is a high-throughput and low-latency standard that implements RDMA to achieve reliable communication between interconnected nodes. A switched fabric topology enables point-to-point data transfer across multiple channels in parallel. As of 2022, InfiniBand could reach a theoretical effective data rate of 100 Gbps per link (InfiniBand Trade Association, 2023).

NVMe

Non-Volatile Memory express (NVMe) is a high-bandwidth communication protocol specification to enable access of SSDs through Peripheral Component Interconnect Express (PCIe) buses and better capitalize on the parallelism capabilities of their hosts' storage. Non-Volatile Memory express over Fabrics (NVMe-oF) is an extension of the above protocol that encapsulates it through transport protocols, e.g., TCP, RDMA and InfiniBand, allowing NVMe commands to be tunneled between remote nodes.

The `verbs` API is available through the `libfabric` library and provides functions for applications to access NICs. It supports direct read and write access between remote nodes from the application layer with guarantees against packet losses.

2.5.3 HPC Clusters and Exascale Computing

Above the computational capability for 10^18 64-bit operations per second, supercomputers are considered exascale (KOGGE et al., 2008). This barrier was only recently crossed with a growing class of exascale supercomputers, like 2022's Frontier (Oak Ridge National Laboratory), and the upcoming Aurora (Argonne National Laboratory) and JUPITER (Forschungszentrum Jülich).

The IO500 benchmark suite (IO500 Foundation, 2022) is a well known test battery comprising five workloads to evaluate the I/O performance of HPC systems. The tests measure bulk, small and metadata I/O for both random and pattern-based data access, to which a single score is assigned. Ranked listings with the world's top-scoring systems are unveiled twice a year. A "research" listing is currently headed by Pengcheng Laboratory's Cloudbrain-II on Atlas 900, based on their SuperFS

filesystem. In the "production" listing, the headliner is the Leibniz Supercomputing Centre's LRZ, running DAOS.

The most important metric in I/O benchmarks is the Data Transfer Rate (DTR), or throughput, which is capped by the link medium's capacity, or bandwidth. We define these terms below.

Definition (Throughput). Throughput, alternatively Data Transfer Rate (DTR), is the transmission speed R_m of successfully sending d units of information between connected devices over a link M in the time t_d. It is usually measured in bytes per second.

$$R_m(d, t_d) = \frac{d}{t_d} \tag{2.1}$$

Definition (Bandwidth). Bandwidth refers to the theoretical capacity C_m for data transfer over a connecting link m. It corresponds to the peak or maximum throughput observable over any period of time on that medium, i.e., $C_m \geq R_m, \forall m$.

Exascale clusters require modernized software stacks that fully exploit the capabilities of their hardware components.

Among parallel storage solutions, the prevalent use of parallel file systems is a known chokepoint to scalability in applications that do not need the consistency semantics or hierarchical structure of POSIX I/O, such as computationally-intensive analysis workflows with WORM data.

Therefore, HPC storage systems are designed around object-based semantics, despite being traditionally exposed to middleware (e.g., Hierarchical Data Format version 5 (HDF5)) through POSIX. Recently, HPC and cloud-based applications have started to exploit object interfaces natively and efficiently (GADBAN; KUNKEL, 2021; LIU et al., 2018a).

3 RELATED WORK

TTree and RNTuple as Data Formats for Scientific Analysis

Though not yet production–ready, ROOT RNTuple is already present in the literature for HEP storage and analysis applications. The format has been extensively shown to outperform its predecessor ROOT TTree in quantitative and qualitative criteria. RNTuple boasts a compact representation of nested collections, which has led to 15–25 % smaller LHC experiment files post–compression. It has also demonstrated finer-grained parallelism and improved memory management and I/O performance in SSDs, with over 500 MB/s/core throughput (BLOMER et al., 2020; BLOMER, 2018; LOPEZ-GOMEZ; BLOMER, 2022).

In addition to promising metrics on local SSDs, claimed to be partly due to forgoing spinning disk optimizations, RNTuple has been adapted for use in tandem with remote storage solutions, such as Hypertext Transfer Protocol (HTTP) through libdavix (DEVRESSE; FURANO, 2014), XRootD (DORIGO et al., 2005) and object stores (NAUMANN et al., 2022; LOPEZ-GOMEZ; BLOMER, 2021).

In particular, Lopez-Gomez and Blomer (2021) introduced an experimental DAOS backend for ROOT's RNTuple as a probe towards first-class support of object stores. The work extended RNTuple's generic storage layer with a concrete implementation for DAOS without altering the user API. The authors defined a *naïve* data mapping of one RNTuple data chunk per DAOS object. The native DAOS backend was evaluated over sockets and compared against both a local POSIX filesystem and a Filesystem In Userspace (FUSE) filesystem, accessible through a compatibility layer. Experimentally, the proposed backend outperformed FUSE in throughput by wide margins, while falling short of RNTuple's file backend on local SSDs. The results suggested that a native solution with a more thoughtful data mapping was needed to fully exploit DAOS.

As in-house solutions, TTree and RNTuple have both been subjected to comparisons against industry-standard data formats for analysis. In Blomer (2018), TTree had overall better performance than a wide range of popular I/O libraries like Parquet (The Apache Software Foundation, 2023b), HDF5 (SOUMAGNE et al., 2022), and SQLite for the HEP use case, i.e., partial, columnar and repeated reading of the dataset, at which columnar formats excel. In Lopez-Gomez and Blomer (2022), RNTuple was similarly compared to Parquet and HDF5. A qualitative study

showed RNTuple's efforts to cover features critical to HEP analysis, such as schema evolution and creation from C++ classes. Furthermore, RNTuple dominated the aforementioned formats throughput–wise when accessing from CephFS, HDD, and particularly SSD.

With each upgrade cycle, the LHC experiments generate larger volumes of data. For HEP analysis to cope with this increased data production, has been a push to modernize HEP workflows and its specialized software to make full use of cloud computing, HPC clusters and distributed data centers. There has been considerable effort to adapt ROOT for distributed computing settings (SEHRISH; KOWALKOWSKI; PATERNO, 2017). Particularly, its declarative analysis interface RDataFrame has been adapted to leverage modern and scalable analytics engines like Spark and Dask and enable distributed performance transparently to ROOT end–users without the complexity of grid computing (PADULANO et al., 2020; PADULANO et al., 2023; PADULANO, 2023).

To speed up RDataFrame analyses with repeated data access, Padulano et al. (2022) proposed a caching system for RNTuple that exploits available fast storage locations, e.g., SSDs and remote object stores, in a backend-agnostic way. While compressed chunks are read from slow, mass storage, the mechanism writes a copy of the data in parallel to the cache location, interleaving the I/O operations with CPU-bound decompression and analysis. Subsequent workflows by any user can access the cache directly. Using DAOS as cache over the conceptual backend introduced in Lopez-Gomez and Blomer (2021), with transfer chunks of 4 MiB, the authors evaluated the proposal on one and seven client nodes armed with an InfiniBand interface and spawning 16 threads each. They observed processing throughputs of 8 GB/s and 37 GB/s, equivalent to 46% and 74% of the theoretical maximum reading throughput, respectively.

Storage Models for High Performance Computing Applications

In recent literature, many data-intensive applications have turned to HPC-targeted storage engines and experimented with object stores, e.g., in the context of research and scientific analysis, seeking to avoid performance and scalability bottlenecks associated with POSIX I/O.

Both in generic benchmarks and realistic contexts such as numerical weather prediction, simulation and deep learning, the performance of object stores has been

stress-evaluated and compared with that of parallel filesystems geared toward HPC applications, e.g., Lustre, OrangeFS and BeeFS. Through emulation, object stores were shown to boost scalability for the HDF5 library in intensive HPC workloads (CHIEN et al., 2018). For the past years, a plurality of the top-performing storage stacks on the IO500 benchmark suite have utilized DAOS as their underlying system (IO500 Foundation, 2022), including Argonne's Aurora exascale supercomputer (Argonne National Laboratory, 2023). In studies, DAOS was shown to handily outperform kernel-dependent storage stacks on small and average-sized clusters of up to dozens of nodes equipped with persistent memory and NVMe devices. These demonstrated DAOS as particularly performant for small I/O transfers of 10 MiB or less, and an equivalent contender for bulk data to the order of several tens of MiB, suggesting that it can better sustain a high throughput of metadata transactions. In contrast, the same studies suggested limitations at larger scales having more complex network topologies – i.e., hundreds or thousands of nodes –, especially with regards to metadata bottlenecks for a high count of thousands of Message Passing Interface (MPI) tasks. Furthermore, as big data clusters typically rely on HDDs as primary storage, there are concerns that NVM-based storage systems like DAOS will be limited to mid-sized clusters or kept to caching layers at best (MANUBENS et al., 2022; MANUBENS et al., 2023; LOGAN et al., 2023; HENNECKE, 2023). At the same time, other object stores have been explored for HPC. Ceph is a versatile and popular storage system based on an underlying object store layer, RADOS (WEIL et al., 2007). Jeong et al. (2019) revealed challenges in applying Ceph to HPC workloads because large files are transcribed to small objects, throttling performance.

For existing parallel filesystem libraries with established interfaces and implementations built around POSIX I/O semantics that impair their scalability to larger clusters, a common solution is to develop connectors to more scalable backends that have been promoted for HPC, e.g., DAOS, OpenStack Swift, Ceph RADOS, Ceph BlueStore. This approach, based on the concept of a Virtual Object Layer (VOL), bypasses block storage backends and operates in users-space. Though performance tends to lag in comparison to native backends, it avoids significant changes in middleware and application codebases. Through evaluation, these VOLs have consistently shown better scalability than parallel filesystems, though object stores still lack the degree of optimization the former have attained in the past decades (SOUMAGNE et al., 2022; DUWE; KUHN, 2021; LIU et al., 2018b).

Though less focused on performance, the use of HTTP-based managers can bridge the gap between heavy data processing and object stores. One example is CERN's Davix, which offers a single interface to manage different object stores APIs remotely over HTTP, e.g., Amazon S3, Google Cloud, Microsoft Azure and WebDAV (DEVRESSE; FURANO, 2014). Despite optimizations such as range coalescing, as a file manager, integration with object stores is limited to converting operations at the POSIX I/O level rather than the application's. Even so, integration of object stores to grid computing infrastructure through Davix was proposed by Ayllon et al. (2017), as a promising alternative to distribution of HEP data through FTS (FTS, 2023).

4 INTEGRATING OBJECT STORES INTO ROOT RNTUPLE

In this chapter, we propose the integration of two object store backends into RNTuple. The first is a production–grade, DAOS–based I/O layer to allow HEP analysis workflows to be efficiently deployed to HPC facilities. A proof–of–concept was already in place before our work; therefore, we focus on the added features to the framework. The second contribution is an experimental approach to cloud–based object stores through an AWS S3 backend.

The chapter layout is as follows. Section 4.1 presents RNTuple as a data format and introduces key terminology to understand the proposed approaches. Section 4.2 verses on the challenges of adapting file–based ntuples to object storage. Section 4.3 explains the added features to the RNTuple–DAOS backend. Section 4.4 introduces the experimental RNTuple–S3 backend, focusing on the differences to the DAOS backend. Finally, section 4.5 describes the tools and technologies used in the development of this project.

4.1 The RNTuple Data Format

In this section, we introduce RNTuple's data organization as it is serialized on disk and the modifications made toward interoperability with the object store paradigm.

Layout and Serialization

The RNTuple binary format describes the layout of an ntuple in its serialized and on-disk representation, i.e., in terms of its *pages* (with data) and *envelopes* (with metadata). Figure 4.1 gives a complete example of a serialized ntuple.

Data Building Blocks

On disk, ntuples are stored in horizontal splits called clusters ("`RCluster`"), equivalent to the TTree counterpart. These are self-contained blocks holding a range of sequential entries ("`REntry`") and its contents for the scheme's features. Sized at $\mathcal{O}(100\,\mathrm{MB})$, the cluster is RNTuple's unit for efficient I/O and serves as a recovery

checkpoint in the case of crashes while taking in data.

Each feature of an entry represents a homogeneously-typed field ("`RField`") that is internally mapped to one or more columns, depending on the complexity of the type. For instance, to attend expectations of HEP analysis mentioned in subsection 2.1.2, RNTuple supports arbitrarily-deep nested collections. These are projected on disk as multiple columns: the *offset column* indexes the start of each entry in the *value column*, which can be another collection.

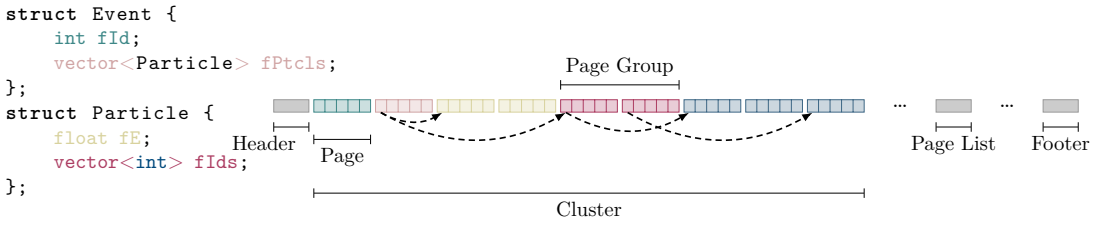
This nested approach gives "ntuple" its name. Its benefits include random access of each entry (after one indirection step to retrieve the index) and the fast vertical merging of two datasets under the same schema. Data compression can also be more efficient, as type-appropriate compression algorithms are applicable on a per-column basis.

Each column ("`RColumn`") has a fundamental type associated with it. For data positioned in the same cluster, the contents of a column are serialized contiguously on disk, as expected of a columnar system.

Columns are further broken into pages ("`RPage`"). Pages are the building blocks of RNTuple and the smallest data unit in the subsystem, occupying sizes in the order of $\mathcal{O}(64\,\mathrm{kB})$ before compression. Since pages are the compounds of columns and thus share the same fundamental type, it is at this scale that compression algorithms are applied. It is roughly equivalent to TTree's basket.

Finally, a page group ("`RPageGroup`") denotes the set of pages that belong in the same column and cluster; their contents stem from nearby entries, share the same fundamental type. Page groups can be seen as corresponding to the "unit of analysis", as their data is typically accessed together in such workflows.

Figure 4.1: RNTuple on-disk format.



Source: The Authors (adapted from Blomer et al. (2020)).

Metadata Envelopes

The point of entry for a serialized ntuple is the *anchor* envelope. It is a minimal record that specifies the format version, total dataset size and location and length of the remaining envelopes that describe how to read the ntuple.

The *header* envelope delineates the RNTuple schema. It contains the ntuple name, along with identifiers and types for the data model, indicating how to interpret the fields and columns.

The *footer* envelope is integral to traversing the data stored in the ntuple. It is the point of entry for nested lists called *pagelists*, which indirectly stores the location and size of the pages through double indirection. Each pagelist covers a group of consecutive clusters, providing an on-disk descriptor for each cluster. These descriptors contain the columns descriptors, which finally index the column's pages on disk.

RNTuple I/O

A well designed I/O subsystem must competently orchestrate computational resources, e.g., CPUs, memory and storage, and minimize the impact of communication latency. I.e., efficiently storing and retrieving data from external devices should consider the computer system holistically while operating on a bounded memory budget.

RNTuple provides a mature interface that implements efficient techniques for I/O operations at the page and cluster scale. The specific implementations for writing and reading are known as the *data sink* and *data source*, respectively.

Data Sink

RNTuple's data sink is the mechanism responsible for the writing of data (in the form of pages) into storage. The functions of the data sink are to manage data ingestion efficiently, apply suitable data compression according to type and scale, and keep metadata updated through regular transactions.

As the granular data unit of compression and I/O access, pages can be committed individually or through a vector write mechanism. With vector writes, the writing of pages is deferred until all belonging to the current cluster are ready for

I/O, i.e., have been compressed and buffered. The main advantage of vector writes is the potential for asynchronous compression in parallel and throughput saturation by increasing the volume of data being transferred.

When the option for vector writes, or "buffered writes", is not available, e.g., when the backend does not support this ingestion mode or when only a partial range of the column's data is being committed, RNTuple defaults to synchronous, individual commits for each page. This approach has been shown in the literature to collapse performance in the face of large amounts of data, especially for backends that might deal with significant network latency.

Data Source

RNTuple's data source manages data retrieval from storage to the user application. Since reading data is more common than updating it in data analysis, a pattern reminiscent of the WORM storage paradigm, the efficiency of the data source is seen as a more pressing concern than the sink's.

Analogously to the sink, the source provides the functionality to fetch pages individually or in bulk; unlike the sink, only pages from fields needed by ongoing analyses are requested. When performing bulk fetching, the source is able to include columns from clusters within a given read-ahead window (e.g., the current cluster and the next two), provided enough memory is available. Reading ahead is useful to increase throughput when the set of columns requested for analysis is sparse and to engage the link layer while decompression utilizes the CPU.

Compression

RNTuple supports compression at the page level with various built-in algorithms, including, in approximate increasing order of compression rate, `zlib`, `lz4`, `zstd` and `lzma`. Because of RNTuple's columnar format, different compression configurations can be applied to each column according to its type. Furthermore, the separation of composite fields, e.g., collections, into separate offset and value columns leads to higher compression rates due to lower data entropy.

Task Scheduler

RNTuple relies on a thread pool model implemented through Threading Building Blocks (TBB) (Intel Corporation, 2023) to schedule its I/O operations and achieve task concurrency. Thread pools are a resource-bounded, efficient approach to concurrent programming, as they consist of a limited group of preallocated worker threads that consume tasks submitted to a queue. To enable parallelism, the implicit multi-threading option `ROOT::EnableImplicitMT()` must be signaled by the user.

In RNTuple, this is applied at the cluster bunch level, i.e., a range of clusters fetched at once for parallel decompression and analysis. The cluster pool ("`RClusterPool`") spawns an asynchronous thread responsible for preloading pages whose columns are on demand by the analysis at hand. With parallelism, comes the potential for optimizing the data source's fetching pattern. Specifically, queued requests can be merged, linearized or otherwise manipulated when supported by the storage backend (BLOMER et al., 2020).

4.2 Adapting RNTuple for Object Stores

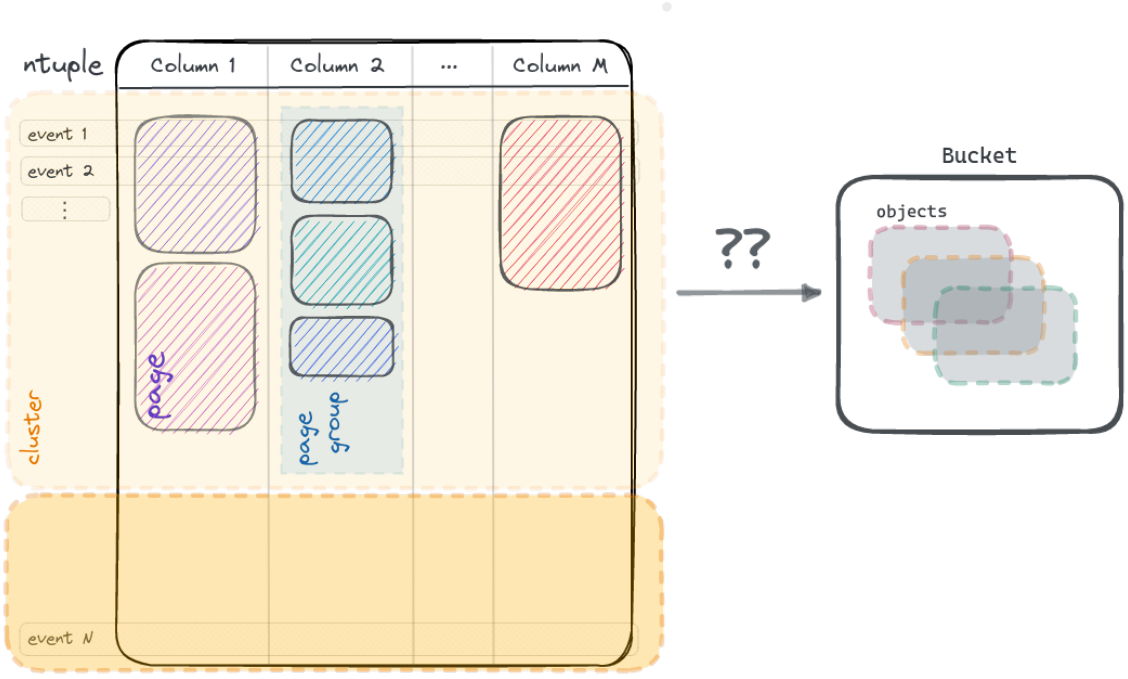
Emerging object store solutions provide invaluable capabilities for future analysis. In particular, their relevance for HEP research stems from their ubiquity in cloud topologies, horizontal scalability and cost-effectiveness (section 2.4).

Contrary to ROOT TTree's tight integration with file systems (subsection 2.2.1), RNTuple imposes a separation between higher-level abstractions and the lower-level storage layer. The process of extending support to object stores is simplified as a result. RNTuple presence in object stores is envisioned as two-fold:

Cloud: massively distributed object storage as an intermediate and transient stage between workflow extremities, leveraging existing global infrastructure to provide granular ntuple access to researchers worldwide. For this use case, we chose AWS S3. S3 is among the most popular object stores in the industry and its API has become the standard for other cloud providers. Therefore, extending support to its API allows the future leveraging of multiple others, e.g., Microsoft Azure.

HPC: cluster-local distributed object storage for HPC data centers as a stage for

Figure 4.2: The RNTuple-to-object mapping problem



Source: The Authors.

big data analysis. While object stores are still gaining a foothold into the HPC space lately, their scalability has been successfully demonstrated (LIU et al., 2018b). For the past several years, Intel DAOS has figured among the best performing stores in the category; that, along with its open-source status, made it a natural choice for ROOT.

Guiding Principles

The development of this project sought to adhere to the following design and performance principles:

Principle 1 Efficient Resource Management. Sober memory allocation and minimal use of system calls.

Principle 2 Bulk Transfer and Deferred Engagement of the Link Layer. Launching requests individually for small data chunks hinders transfer scalability and negatively affects throughput. By waiting to issue data requests in bulk, effective I/O rates may increase; in parallel, the processing unit is freed from interruptions to tackle other threads.

Principle 3 Minimal New Metadata. No additional descriptors unless strictly necessary, minimizing format specification overhead and metadata latency.

Principle 4 Thoughtful Data Mapping. Preserve RNTuple's column-based pattern across storage paradigms. Exploit maximally the backend's mapping interface to achieve fine-tuned, backend-specific results. Also, HEP reading patterns should be taken into account ("data read together is stored together").

Principle 5 Granular Data Access. When possible, new backends should preserve a reading granularity closer to RNTuple's file backend, e.g., by supporting byte range requests.

Principle 6 Artificial RNTuple Layout. When possible, object store data should not be bound by an existing ntuple's native layout boundaries, which may be optimized for a different backend.

Principle 7 Coexistence of ntuples. Bucket-like namespaces should handle distinct ntuples that share the same underlying resources, but kept distinguished via an implicit hierarchy or partitioning transparent to the user.

4.3 RNTuple-DAOS: Design and Implementation

In this section, we provide an overview of our proposed mechanisms and strategies to redesign RNTuple's DAOS backend into a high-performance alternative for HEP analyses.

Operation Management

In order to saturate the bandwidth capacity, our approach hinges on DAOS' support for scalable and non-blocking bulk transfer. For that, the I/O pipeline must avoid superfluous system calls and simplify operation polling.

Operation Queue

Creating DAOS operation queues ("event queues") incurs significant overhead, as they are instantiated in tandem with a communication endpoint for data

transfer. In the case of RDMA-enabled interfaces, such as InfiniBand through `libfabric`'s `verbs` API, spawning a fabric endpoint requires a prohibitively costly system call.

Therefore, optimal I/O performance takes a persistent operation queue throughout the program's execution. Our approach ties the lifetime of the queue to that of the programmatic container manager. When creating and tearing down the instance of RNTuple's class for DAOS containers, so is the queue constructed and destroyed. With that, endpoint costs are paid only once while spinning up the backend, before the first I/O request.

Asynchronous Calls

In RNTuple, I/O operations act as transactions at the cluster level: each cluster is guaranteed to be committed before progressing to the next one. For reading, a cluster bunch with multiple clusters may be optionally fetched at once, provided the data volume fits the application's memory budget.

In lockstep, the DAOS backend operates within this transaction window to issue all calls to remote storage based on the cluster(s) at hand. Operations are launched asynchronously until all pages requests in the cluster have been made; at this point, the backend reaches a blocking barrier which waits for any pending operations. This asynchronous scheduling is especially useful for large enough clusters, e.g., with 500 MiB or more in size, where the link might go underutilized for too long as requests are processed and issued.

Grouping Operations

For each cluster, many DAOS requests may be issued asynchronously, contingent on the pages needed by the analysis and on the mapping function selected for RNTuple-to-DAOS data ingestion.

To avoid polling several operations on synchronization points, we instantiate a symbolic parent operation corresponding to the batch of operations presently in-flight and for whose completion the barrier should wait. In DAOS, this parent may be launched after the actual operations, and its completion is subservient to that of its children.

The uses of symbolic parents and the blocking barrier are demonstrated in

Algorithm 1; line 2 instantiates the parent "event" (in DAOS nomenclature), which is conditioned to the success of all child operations instantiated in line 5. These child operations are sent in-flight by the calls to `FetchByObjDkey()` or `UpdateByObjDkey()`, lines 9 or 11 respectively, depending on the mode being reading or writing. Finally, a blocking call waits only for the parent operation in line 13; this raises an operation barrier in DAOS that prevents new children from being instantiated until the current batch is no longer in–flight. We return the success or failure of the parent operation in line 14.

If parent operations were not used, the implementation would be more complex; the blocking call would have to poll every child operation in the list, popping them as they are concluded.

Pseudocode 1: RNTuple-DAOS Container Vector Read/Write Operations

```
1  function Container::VectorReadWrite(batches : MultiObjectRWBatch,
     mode : WRITE ∪ READ)[1]
2      parentOp ←daos_event_t{}, childOps ←List
3                  ▷ Iterate over coalesced batches, instantiate operation handles.
4      for ⟨⟨oid, dkey⟩, batch : RWBatch ⟩ in batches do
5          childOps.Append(daos_event_t{})
6          daosPool.queue.InitializeOperation(oid, parentOp)
7              ▷ Launch fetch or update operation asynchronously. Tie IODs and
           SGLs to child operation.
8          if mode is READ then
9              FetchByObjDkey(oid, dkey, batch.dataRequests.iods,
                batch.dataRequests.sgls, childOps[-1])
10         else
11             UpdateByObjDkey(oid, dkey, batch.dataRequests.iods,
                batch.dataRequests.sgls, childOps[-1])
12
13                                 ▷ Blocking call waits for the symbolic operation.
       daosPool.queue.Wait(parentOp)
14     return daosPool.queue.IsSuccess(parentOp)
```

4.3.1 Co-Locality Mapping Function

The typical access pattern of a HEP analysis is columnar, retrieving row-sequential values for a feature of interest (subsection 2.1.2). However, migration to

[1] `VectorReadWrite()` on Github: `RDaos.cxx#L231`

an object store dissolves the ntuple's columnar schema by virtue of its key-value paradigm (subsection 2.4.2).

From section 2.3, the co-locality of a DAOS object's data blob in the server is determined by the distribution key, i.e., an object's data is stored together if they share an identical `dkey`. This interface, uncommon in object stores, enables the otherwise-unstructured key-value frame to be imbued with columnar semantics through an especially-crafted function between RNTuple pages and DAOS values.

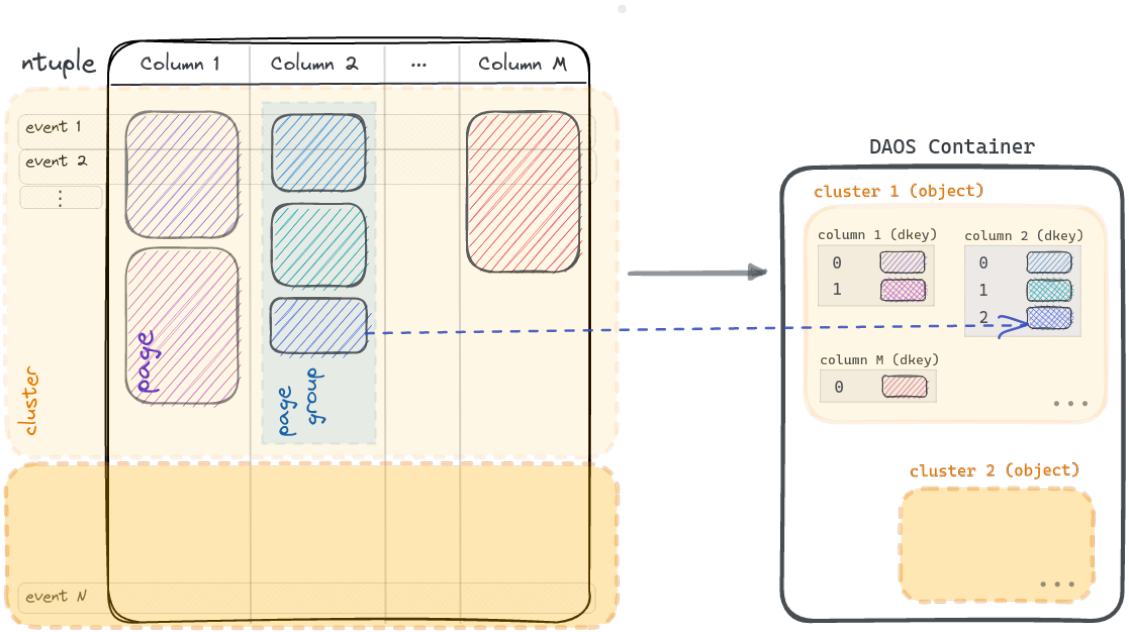
Below, we formally define two mappings, $\phi_{obj-per-page}$ and $\phi_{co-locality}$, between RNTuple pages and DAOS objects. Both project the k^{th} page in the j^{th} column and i^{th} cluster onto a unique object store locator of the form $\langle \texttt{oid}, \texttt{dkey}, \texttt{akey} \rangle$.

$$\phi\colon \langle cluster_i, column_j, page_k \rangle \rightarrow \langle \texttt{oid}, \texttt{dkey}, \texttt{akey} \rangle$$

$$\phi_{obj-per-page}(cluster_i, column_j, page_k) \mapsto \langle page_k, \alpha_{\texttt{dkey}}, \alpha_{\texttt{akey}} \rangle \tag{4.1}$$

$$\phi_{co-locality}(cluster_i, column_j, page_k) \mapsto \langle cluster_i, column_j, page_k \rangle \tag{4.2}$$

Figure 4.3: A visualization of the RNTuple-to-DAOS mapping based on target co-locality



Source: The Authors.

The proposed mapping function ensures that the pages of a page group remain associated throughout the data's lifetime. Coupled with the request coalescing pre-processing step introduced in 4.3.2, this mapping offers an opportunity to treat

a page group as a single transfer unit, speeding up an analysis' reading stage.

4.3.2 Request Coalescing

The mapping based on co-locality, proposed in subsection 4.3.1, ensures a page group's physical coincidence in DAOS servers. This offers an opportunity to write and read data from all pages in a page group in parallel, as if they were a single transfer unit.

To benefit from this parallel optimization, pages belonging to the same page group should be requested simultaneously, i.e., in the same call. From subsection 2.3.2, the DAOS API can issue multiple operations together through the `daos_obj_fetch` and `daos_obj_update` calls. These operate on multiple attribute keys for the same `oid` and `dkey`.

The DAOS calls above are represented in Algorithm 1 through the evocation of `FetchByObjDkey()` and `UpdateByObjDkey()` in lines 9 and 11, respectively. Internally, they conform to the DAOS API by specifying the open handle associated with the object `oid`, the `dkey`, contiguous arrays with the I/O Descriptors (IODs), SGLs and their total sizes in bytes.

The DAOS IOD contains a description of the element as it pertains to DAOS storage, such as the `akey`, total size, and whether the element is a simple blob or follows the array API (see background section 2.3). The SGL contains the actual buffer pointers in size as they are disposed in application memory, i.e., the transfer IOVs which will be copied to DAOS through scatter–gather I/O.

The fetch and update calls also specify a pointer to the child operation, instantiated in line 5, so that its completion can be later polled, as introduced in the above subsection.

Provided all page operations in a cluster are known ahead of time, we can effectively coalesce requests by co-locality via a `<oid, dkey>` pair. With the appropriate redundancy levels, we attain parallel and bulk transfers for page groups.

Algorithms 2 and 3 present our approach to coalesce requests based on a common root of the DAOS mapping that is aligned with the required arguments of the provided DAOS API calls (the `<oid, dkey>`). This is evidenced by lines 12–14 and 23–24, respective to both algorithms. Note that the tuple – and hence the coalescing potential – are sensitive to the mapping strategy (lines 13 or 23).

This tuple serves as index for the (`requests`) dictionary, passed to the `VectorReadWrite()` call (in lines 21 and 25, respectively). The method, presented in Algorithm 1, is part of our backend's "`RDaosContainer`" class.

Batch I/O

Executing I/O operations in batches can amortize network fabric overhead costs and exploit parallel transfer, thus maximizing bandwidth usage. Shortening the data import and export stages is critical for HEP analyses on larger datasets.

Pseudocode 2: RNTuple-DAOS Vector Write (With Caging Support)

```
1  function Sink::CommitPages (ntupleId, clusterId, pageGroups:List) [2]
2      requests ←Map[⟨int, int⟩ → List], locators ←List
3                                          ▷ Coalesce requests by ⟨oid, dkey⟩
4      for ⟨columnId, columnPages : List ⟩ in pageGroups do
5          offset ← 0, itemCount ← 0, index ← itemCount++
6          for page : columnPages do
7              iov ←⟨page.buffer, page.size⟩
8              if offset + page.size > MAX_CAGE_SIZE then        ▷ New cage
9                  offset ← 0, index ← itemCount++
10                                                  ▷ Advance cage index
11
12                                     ▷ Map RNTuple page location to DAOS
13             ⟨oid, dkey, akey⟩ ← RNTuple2DaosMapping(ntupleId,
                clusterId, columnId, index)
14             requests[⟨oid, dkey⟩].Append(iov)
15                                  ▷ Encode page location within DAOS cage
16             pageLocator.position ← EncodePosition(index, offset)
17             pageLocator.size ← page.size
18             locators.Append(pageLocator)
19             offset ← offset + page.size
20                                     ▷ Issue sorted requests to backend in bulk
21     daosContainer.VectorReadWrite(requests, WRITE)
22     return locators
```

In order to speed up data transfer, RNTuple implements both vector reads and vector writes. In ROOT analyses, the range of events and fields to be accessed, e.g., in a for-loop, triggers the fetching of their corresponding pages in storage. When vector reads are enabled, multiple clusters are fetched at once (by the

[2] `CommitSealedPageVImpl()` on Github: `RPageStorageDaos.cxx#L307`.

`LoadClusters()` method in Algorithm 3), scheduled for decompression and eventually accessed.

Analogously, vector writes are available by means of the `CommitSealedPages()` method in Algorithm 2. This method is activated by buffering event data after page compression. If enough compressed pages have accumulated to fill one or more clusters, including every column thereof, then the committing is done in bulk.

Pseudocode 3: RNTuple-DAOS Vector Read (With Caging Support)

```
1  function Source::LoadClusters(clusterDescriptors : List) [3]
2      // Initialize collections.
3      requests ←MultiObjectRWBatch
4      clusterPages ←Map[int →⟨PageType,int⟩]
5      pagemaps ←List
6      for cluster : clusterDescriptors do  ▷ Clusters in read-ahead window
7          for column : cluster.columns do
8              for page : column.pages do
9                  ⟨ cagePosition, cageOffset⟩ ← DecodePosition(page)
10                 ▷ Collect page requests under their cages to prevent duplicate
                    requests.
11                 clusterPages[cagePosition].Append(⟨page, cageOffset⟩)
12         Allocate clusterPayload ∀ cage s.t. cage ∈ cluster
13                 ▷ Pagemaps know the page metadata and own their data buffers
14         Instantiate cluster's pagemap
15
16                                          ▷ Coalesce IOVs under ⟨oid, dkey⟩
17         for ⟨cagePosition, sortedPages⟩ : clusterPages do
18             for page : sortedPages do
19                                              ▷ Tie memory area to logical page
20                 pagemap.RegisterPage(⟨page.columnId,
                    page.indexInColumn⟩, ⟨clusterPayload +
                    page.payloadOffset, page.size⟩)
21                 pagemaps.Append(pagemap)
22
23             iov ← ⟨clusterPayload, cageOffset, cage.size⟩
24             ⟨oid, dkey, akey⟩ ← RNTuple2DaosMapping(page.ntupleId,
                page.clusterId, page.columnId, cagePosition)
25             requests[⟨oid, dkey⟩].Append(iov)
26     daosContainer.VectorReadWrite(requests, READ)      ▷ Request to read
       from DAOS
27     return pagemaps
```

[3] `LoadClusters()` on Github: `RPageStorageDaos.cxx#L688`

4.3.3 Scatter-Gather Concatenation (Caging)

During (de-)compression and I/O, pages are stored in individual buffers managed in a memory pool by RNTuple. In general, considering that page groups are limited by the cluster barrier, and that the page size attribute applies ntuple-wide, the page size attribute cannot be too large: this can regularly lead to wasteful memory allocation for columns with types that have a small footprint (even byte-packed, like booleans). Page sizes cannot be too small, either, as that can cripple a compression algorithm's effectiveness.

From inception, RNTuple was designed around the partitioning column data in building blocks of around $\mathcal{O}(100\,\text{kB})$ in size, before compression. For the file backend, keeping chunks at that scale showed a balanced compromise between data granularity, memory consumption and compression rates. Therefore, RNTuple's default, uncompressed page size was set to 64 kB.

However, one page size is unlikely to fit all backends. With object stores come considerations on remote communication, both in endpoint overhead costs and network latency, e.g., over RDMA or sockets (FREY; ALONSO, 2009). It remains to be seen if the scale used for the file backend is adequate for transfers over the network, as throughput is sensitive to a number of often stochastic factors, the size of transfer buffers especially.

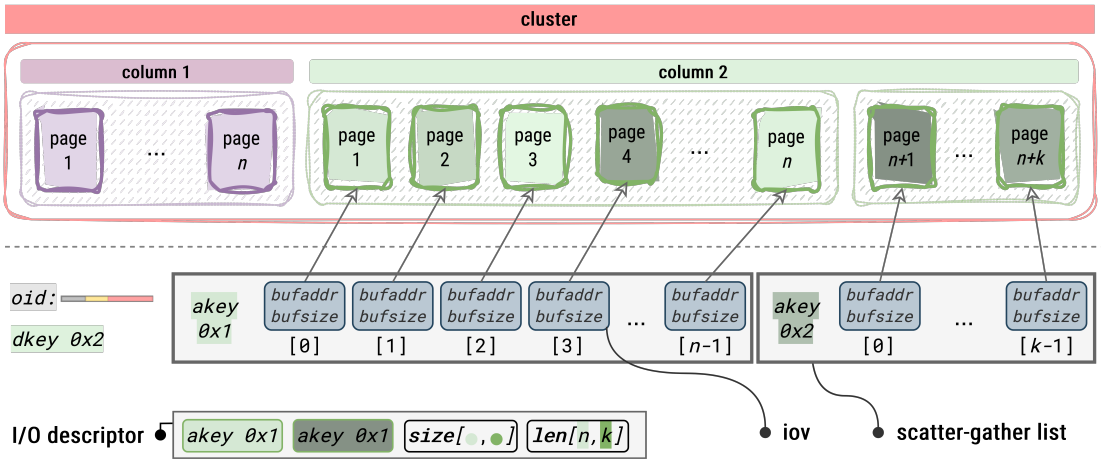
Thus, in order to emancipate an ntuple file's migration to object stores from its native page size, we propose a mechanism that logically concatenates neighboring pages in RNTuple's data sink.

The term *cage* is a *portmanteau* of *concatenated page*. It underlines that the constituting pages are written and read back together, always sharing the same I/O request throughout their lifetime. Note that grouping is incidental and driven by workflow efficiency, as values in sequential rows are statistically independent in an ntuple. In other words, there is no semantic meaning behind which event' data are caged together.

All pages in the same cage share the DAOS identifiers, i.e., `oid`, `dkey` and `akey`. In effect, a cage is a single, contiguous blob in the DAOS namespace.

Sequential pages from the same page groups are concatenated into a cage until (a) no more pages fit within a given maximum cage size; or (b) there are no additional pages in the page group to commit. The maximum cage size is user-

Figure 4.4: Scheme for scatter-gather concatenation (caging). An update request contains a scatter-gather list of IOVs. Each IOV describes the memory region of a buffered page. I/O descriptors denote that the memory regions are to be stored as a blob.



Source: The Authors.

defined with an empirically–adequate default value of 1 MiB, as seen in subsection 5.3.3. Algorithms 2 and 3 present our caging–supported implementation.

In Alg. 2, pages from the same page group are sequentially assigned the same `akey` in DAOS until the limit in `MAX_CAGE_SIZE` in line 8 is reached. If that happens, a new cage is inaugurated by progressing the index that serves as the `akey`. In other words, pages from the same cage are filed under the same ?? in the update call from Algorithm 1, even though data is located in separate buffers and thus separate IOVs of the corresponding SGL. Caging can be disabled by simply assigning `MAX_CAGE_SIZE` a value of zero, such that every page will be its own cage with an independent blob.

As for Alg. 3, the potential multiple requests to pages belonging to the same cage unit is an issue to circumvent if we are to avoid the same blob being fetched repeatedly by RNTuple. We tackle this with a dictionary structure, which is populated in lines 8–11: the indexing value is the same `cagePosition` later used as the `akey` to perform a single fetch operation to object storage.

We note that page location metadata is kept minimal at 64-bit values; to accurately retrieve the page data, we pack together into those bits both the cage position and the offset within the cage, in bytes, where the data for that specific page starts. Thus, the backend needs to `EncodePosition` and `DecodePosition` in lines 16 (`CommitPages()`) and 9 (`LoadClusters()`), respectively.

Once written, DAOS blobs are only opaquely–accessible to the user, i.e., without support for partial and byte–range read (subsection 2.3.2). Thus, regardless of how many pages are requested in the page group, the cage is read back from storage as a whole – though only the requested pages within it are decompressed. At worst, any request for a single page forcefully requires the backend to request dozens to hundreds of pages, depending on the cage size and compression factor.

While this is a clear limitation of the caging mechanism for analysis, HEP use cases are unlikely to subscribe to such a reading pattern. Instead, sequential ntuple field ranges are requested in bulk for analysis over the statistically–independent events (i.e., rows).

In light of this, the interface for page requests is disabled for reading ntuples stored as cages, i.e., the 'cluster caching' option must remain activated. This simplifies the implementation and prevents users from misguidedly and silently executing inefficient analyses.

Multiple NTuples per Container

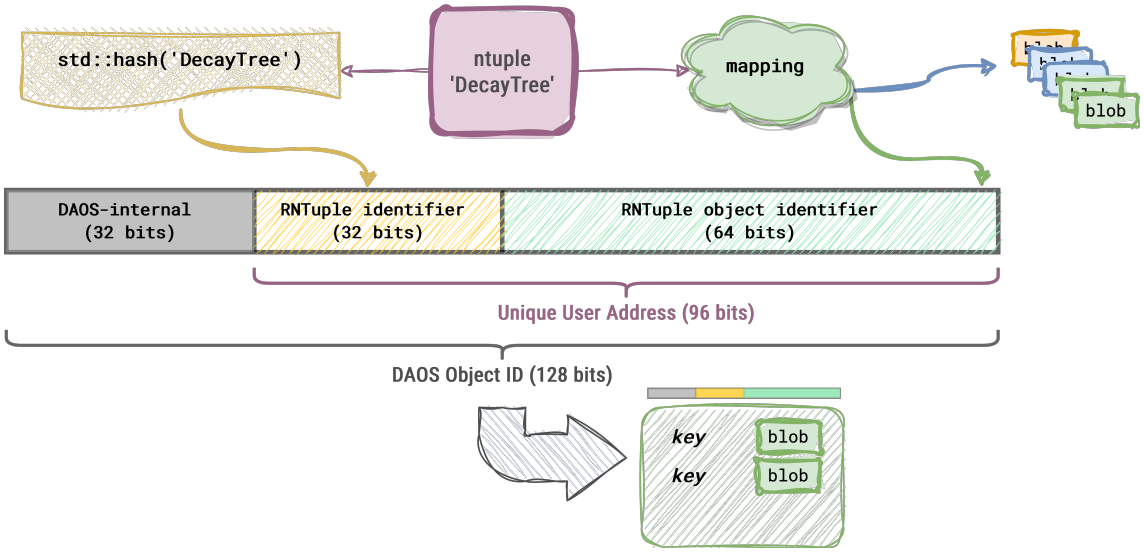
From subsection 2.3.2, a DAOS pool can host hundreds of containers, each with billions of objects. A one-ntuple-per-container approach is likely to (a) waste the mapping image, and (b) be impractical, as the container limit would saturate quickly, as dozens of sibling datasets can spawn from the same raw experiment data.

We tackle that by enabling multiple (billions of) ntuples to populate the same container, assigning segments of the addressable object space to separate datasets without an explicit hierarchy.

For that, we designate the 32 most-significant, non-reserved bits of the object ID to specify the dataset. This 32-bit value is derived from the ntuple's name as outputted by the `std::hash` function implementation for `std::string`s. With the zeroth ntuple reserved, this allows for $2^{32} - 1 = 4294967295$ different ntuples per user namespace, which is likely sufficient for most projects and experiments.

There are no plans to support the handling of hashing collisions for differently-named ntuples, as empirical tests with randomized inputs exhibited a 0.244% ntuple name collision rate. This decision simplifies RNTuple index resolution by eschewing linked lists of RNTuple headers or an index table. More importantly, it keeps the solution metadata-less, observing Principle 3 in 4.2.

Figure 4.5: Composition of the DAOS object ID with support for multiple ntuples.



Source: The Authors.

Table 4.1: Differences and similarities between DAOS and S3 w.r.t. properties relevant to RNTuple.

Property	**DAOS**	**S3**
Storage Structure	Flat	
Namespace Access	$\langle$`server:string`, `id:string`$\rangle \mapsto$ namespace	
Namespace Terminology	Container	Bucket
Blob Access	$\mathbb{N}^3 \mapsto$ blob	`string` $\mapsto$ blob
Data Locality Control	Explicit (`dkey`)	Induced by Naming
Object Structure	array or KVS	blob
Buffer Management	IOV	IOV/Stream
Latency	Very Low	Average to High
Protection, Acceleration	Custom (Object Classes)	Service Tier (Region, Batch)

4.4 RNTuple-S3: Backend for the Cloud

Developing RNTuple's DAOS backend to a mature and production-ready state provided knowledge applicable to other object stores. We sought to leverage it toward supporting a new use case: the cloud.

Whereas HPC exploits distributed processing by spreading data cluster-

locally, cloud-based solutions bring data to the users at the edge of a global topology. This comes with the cost of dealing with high latency from, e.g., the TCP/IP protocols, as well as heterogeneous service depending on the facility's support of S3 features and infrastructure.

For a first RNTuple backend for the cloud use case, AWS S3 was our provider of choice. S3 has become ubiquitous to the point that its API can be considered the *de facto* standard for the industry. By adhering to its interface and *modus operandi*, we intend to more easily adapt to other cloud providers, such as Microsoft Azure and Google Cloud.

The cloud standard, however, is dissimilar to DAOS. On the one hand, S3 its data organization relatively black–boxed to ensure cloud storage remains simple and scalable. On the other hand, this approach limits our ability to propose an informed mapping like the one in section 4.3. We refer to Table 4.1 for a summary of the differences - and similarities - between S3 and DAOS.

Cloud Development Concerns

Based on our past experience, we identified the following main concerns when developing an I/O approach targeting cloud storage:

Network latency: Provider-side, a multi-tier topology of data centers and edge locations may be available to reduce network latency, which is a known liability of solutions based on cloud storage. To counteract the impact of latency on the user side, the framework can adapt its I/O behavior and schedule operations in a cloud-optimized manner. Examples of such strategies include larger transfer buffers, asynchronous scheduling of operations and R/W models that do not require forthwith consistency among replica nodes.

Non-standardized coverage of features: Different S3 servers offer a distinct subset of the features. Some features, like byte-range requests, impact the data scale at which objects are concocted.

Choice of API: There exist several interfaces used to communicate with S3 services due to HTTP request support. Examples include the AWS C++ SDK and HTTP managers like Davix. The choice of interface has impact on performance, feature support, maintenance effort and library dependencies.

4.4.1 Mapping Function

Compared to DAOS, S3 offers a more opaque mapping interface. Firstly, the flat hierarchy in the namespace ("bucket") means that its objects are simple blobs, as seen in Table 4.1. Secondly, its storage model does not explicitly allow for user input on data co-locality, though it is known that an object's label plays a deterministic part in mapping objects or their shards to specific hardware targets (Amazon, Inc., 2023).

The mapping $\phi_{blob-per-page}$ between RNTuple pages and S3 object blobs is defined below. It casts the k^{th} page in the j^{th} column and i^{th} cluster onto a unique `string` label in the S3 bucket namespace, which displays this information separated by slashes ("`/`").

$$\frac{\phi\colon \langle cluster_i, column_j, page_k\rangle \rightarrow \texttt{object\_identifier : string}}{\phi_{blob-per-page}(cluster_i, column_j, page_k) \mapsto \texttt{string(}cluster_i\texttt{/}column_j\texttt{/}page_k\texttt{)}} \quad (4.3)$$

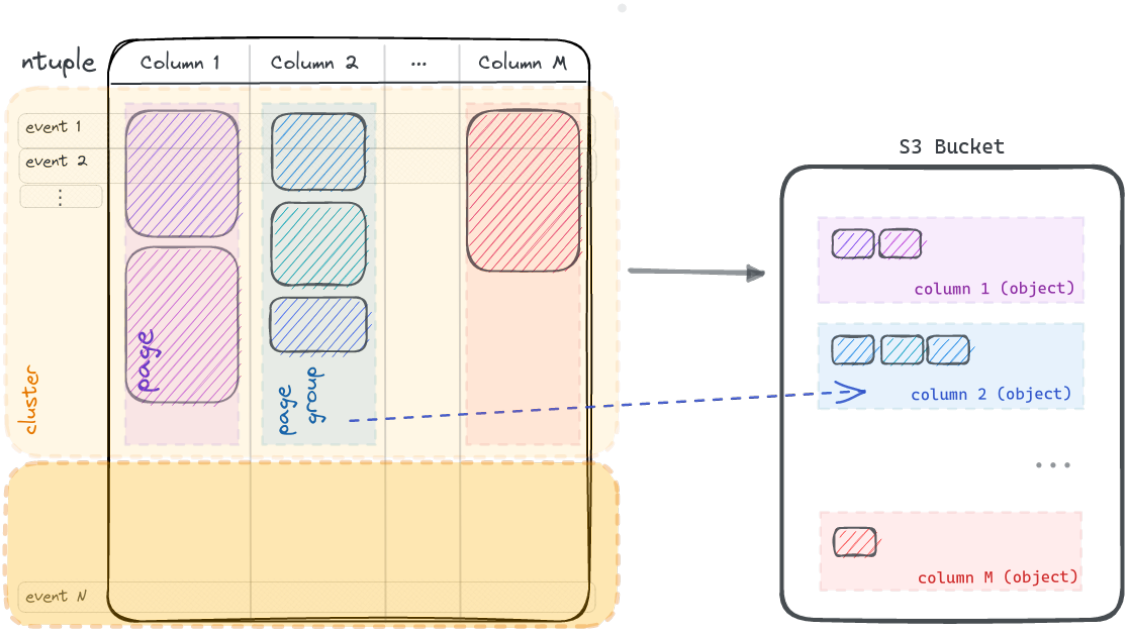
More elaborate mappings can be considered based on efficiency concerns in lieu of an apparent loss of access granularity; one such example is pictured in Figure 4.6, storing entire page groups together, as these are expected to be fetched together under normal HEP workflows.

4.4.2 Davix–based Implementation

Multiple interfaces are available to enable interoperability between RNTuple and AWS S3, including several APIs provided by Amazon, e.g., the AWS C++ SDK. [4] For this work, we chose to leverage CERN's own Davix. [5] Davix is already a ROOT dependency for supporting file transfers with FTS. Furthermore, Davix provides support for byte–range requests, which the evaluation in section 5.4 will prove to be a crucial feature for a mature, cloud backend for RNTuple. Finally, Davix implements the same IOV–based interface for its transfer buffers, making the development of a connecting layer straightforward.

[4] Documentation on AWS: https://aws.amazon.com/sdk-for-cpp.
[5] Repository on Github: cern-fts/davix.

Figure 4.6: A visualization of a proposed RNTuple-to-S3 mapping (page groups as S3 objects) retaining columnar access and projected to amortize latency concerns.



Source: The Authors.

Our S3 backend does not include any request coalescing in its sink or source layers, once again differing from the DAOS backend. While it still processes read and write requests in batches on a cluster bunch basis for similarity with the DAOS backend, there is no support for vector writes in our connector to Davix.

Pseudocode 4: RNTuple-S3 (Davix) Iterated Write

```
1 function Bucket::VectorWrite(operation : MultiObjectRWBatch)[6]
2                                          ▷ Iterate over single–buffer operations
3     for operation : RWBatch in operations do
4         davixURI ← GetDavixLocator(bucket, operation.akey)
5         davixObjects ← Davix::DavFile(davixUri)
6                                          ▷ Blocking call to write a single object
          davixObjects.put(operation.buffer, operation.size)
7     return
```

Algorithm 4 presents a simple implementation that writes RNTuple pages as S3 objects through Davix API calls. We reuse the `MultiObjectRWBatch` structure for managing cage requests, though the simpler mapping interface limited us from exploiting it fully. The procedure iterates over each (unique) identifier, generating a string label from the RNTuple locators (i.e., the cluster, column and page identifiers),

as defined by the mapping $\phi_{blob-per-page}$ in this section.

Pseudocode 5: RNTuple-S3 (Davix) Vector Read

```
1 function Bucket::VectorRead(batches : MultiObjectRWBatch)[7]
2     ▷ Iterate over batches (cages) sequentially, instantiating lightweight Davix
      object handles based on the derived URIs.
3     for batch : RWBatch in batches do
4         davixURI ← GetDavixLocator(bucket, batch.akey)
5         davixObject ← Davix::DavFile(davixUri)
6                         ▷ Blocking call to read batch under the same cage index.
          davixObject.readPartialBufferVec(batch.iov, batch.sgls,
          batch.sizes)
7     return
```

The implementation of vector reads in RNTuple–S3 is detailed in Algorithm 5. While Davix provides the means to request multiple blobs through the `DavFile` interface, these are meant to stitch together different remote objects as a file unit in memory. Due to the single writes limitation discussed above, we do not exploit this capability in the current proof–of–concept, though we retain the structures that would support it in the future. In particular, we envisage this interface for partially reading segments of much larger blobs using byte–range requests. This is discussed in detail in the evaluation section 5.4.

4.5 Tools and Technologies

ROOT is mostly written in the C++ and Python programming languages. As such, C++ was the language used in the development of our approaches. Given C++20 support limitations in ROOT, our particular flavor of C++ consists of modern programming practices based on the C++17 standard's feature set, with the inclusion of back-ported libraries only standardized in C++20, such as `std::span`.

Being an open source framework hosted on Github, some of the technical contributions in this work have made their way to the main branch (ROOT Project, 2023b), leveraging Git version control, a continuous integration pipeline and code review. Other tools used throughout development include `gdb` (debugging), valgrind (memory management) and perf (statistical profiling).

[6] `VectorWrite()` on Github: `RS3Davix.cxx#L156`
[7] `VectorRead()` on Github: `RS3Davix.cxx#L196`

5 EVALUATION

With the proposed contributions introduced in chapter 4, we evaluate our approach and the hypotheses in section 1.1.

This chapter is organized as follows: first, in section 5.1, we set out our evaluation objectives. In section 5.2, we describe the hardware and software configurations and explain our chosen benchmark. Section 5.3 presents our results and analyses on the proposed DAOS backend, including the concatenation feature. Section 5.4 shows preliminary results obtained with the experimental S3 backend.

5.1 Evaluation Objectives

In this evaluation, we seek to:

1. **Validate DAOS as a high throughput object store for HEP.**
   - **1a.** Compare the features presented in chapter 4 with the experimental baseline.
   - **1b.** Investigate the impact of user-defined parameters in RNTuple and DAOS.
   - **1c.** Understand which features bring the most significant improvements.
2. **Confirm the viability of the proposed, experimental S3 backend.**
3. **Compare the two object store backends in RNTuple.**
   - **3a.** Identify key differences between the Cloud and HPC use cases for HEP.
   - **3b.** Extract common patterns as a blueprint for a future, generic object store layer for RNTuple.

Our evaluation methodology is based on the effective I/O throughput as measured in both the writing and reading stages. We consider throughput as a metric of I/O-boundness and saturation of the link layer to be maximize, under the assumption that CPU workload can be parallelized around it in future. As such, this evaluation will guide future developments in RNTuple. In our throughput measurement, we include any and all preparation steps introduced by our proposal in Chapter 4, such as request coalescing.

5.2 Experimental Setup

The evaluations were conducted on two different platforms, named `DAOS-Setup` and `S3-Setup`, which are specified below. The former includes a modest HPC cluster meant to benchmark production-grade software. The latter is an ad-hoc setting for the purpose of testing software under development.

5.2.1 Platforms

The `DAOS-Setup` Platform:

Hardware. We were granted access to Hewlett-Packard Enterprise's Delphi cluster, consisting of two servers and six client nodes interconnected by an InfiniBand fabric. This is how the nodes were configured:

Server nodes. 4× Intel Xeon Gold 6240M CPU sockets, each with 18 physical cores, running at 2.60 GHz. Hyper-threading SMT enabled. Each server was equipped with 24.75 MB of level 3 (L3) cache, 185 GB of DDR4 RAM and a Mellanox MT28908 ConnectX-6 InfiniBand network adapter.

Client nodes. 2× Intel Xeon E5-2640 v3 CPU sockets, each with 8 physical cores, running at 2.60 GHz. Hyper-threading SMT enabled. Each client node had 20 MB of L3 cache and 131 GB of DDR4 RAM. High-speed interconnection was available through a Mellanox MT27800 ConnectX-5 InfiniBand adapter. Each client had two Non-Uniform Memory Access (NUMA) topologies, one of which was associated with the adapter; thus, we pinned the experiment jobs with `taskset` to the range of logical CPUs local to the interface for optimal RDMA transfer.

IOR Benchmark. Interleaved or Random (IOR) is an I/O benchmark suite designed to measure the performance of parallel storage systems in various access patterns. For this reason, IOR tests are frequently used to arrive at a practical throughput limit against which to compare I/O applications.

We ran the benchmark on the Delphi cluster to measure the working bandwidth as a reference point for our results. For the `SX` DAOS object class, default

Table 5.1: Results, IOR Benchmark, HPE Delphi cluster.

Transfer Size (bytes)	Mean Tput, Write (GB/s)	Std. Tput, Write	Mean Tput, Read (GB/s)	Std. Tput, Read
65536	0.695	0.003	0.462	0.006
131072	1.012	0.009	0.780	0.005
262144	1.311	0.21	1.191	0.003
524288	1.601	0.046	1.260	0.008
1048576	1.755	0.034	1.428	0.023
2097152	2.789	0.115	2.316	0.002

SCM-SSD ratio of 6 / 94 % through the DFS interface, 100 GiB block size and buffer sizes ranging from 64 kB to 2 MiB, the measured bandwidth values over InfiniBand interconnect are presented in Table 5.1. IOR reached a throughput of 2.8 GB/s (write) and 2.3 GB/s (read) with 2 MiB transfer buffer size on our single client, two-server `DAOS-Setup` running DAOS 2.2.

Software. The operating system was Red Hat Enterprise Linux 8.4 (kernel 4.18.0-305). The DAOS deployment was based on `daos-2.2.0` (`ofi+verbs` provider) and `libfabric` 1.15.1. The project was compiled with g++ 8.5.0 and O2-optimization.

Project Versions. Below, we specify the versions of ROOT we evaluated, along with their short-hand and covered features introduced in chapter 4. The revisions are taken from the main branch of ROOT Project's repository on Github [1], at points in time that coincide with ROOT release versions v6.26 and v6.28, with a minor patch that moves atomic timers to the start of each request function, in order to cover the entire request preparation:

`v0-BASE`: ROOT revision `#d8de5d0`, containing a basal implementation of a DAOS backend with synchronous, single-page requests and a flat data mapping (one page per KVS object).

`v1-PERS`: ROOT revision `#cda2281`. Incorporates a persistent operation queue and symbolic operations to simplify operation polling (see subsection 4.3).

`v2-COAL`: ROOT revision `#2e38273`. Adds support for vector writes ("Batch I/O"

[1] ROOT Project on Github, main branch: https://github.com/root-project/root/tree/master

Table 5.2: Feature matrix for named versions of RNTuple-DAOS under evaluation.

Feature	v0-BASE	v1-PERS	v2-COAL	v3-COLO	v4-CAGE
Persistent Queues	✗	✓	✓	✓	✓
Batch I/O	✗	✗	✓	✓	✓
Request Coalescing	✗	✗	✓	✓	✓
Co-Locality Mapping	✗	✗	✗	✓	✓
Page Concatenation	✗	✗	✗	✗	✓

in subsection 4.1) and request coalescing. With `kDefaultDaosMapping` := `kOidPerPage`.

`v3-COLO`: ROOT revision `#2e38273`. Adds the proposed mapping based on target co-locality, i.e., `kDefaultDaosMapping` := `kOidPerCluster`.

`v4-CAGE`: ROOT revision `#eee4c8e`. Atop previous features, enables page concatenation from subsection 4.3.3, given a positive concatenation target size, in bytes. For the purposes of this evaluation, we fix `MAX_CAGE_SIZE` := 1048576.

The `S3-Setup` Platform:

In order to evaluate the RNTuple-S3 backend, we sought to observe the framework's behavior in a controlled scenario. This avoids sources of latency and instability typical to realistic I/O over network, e.g., channel and server-side resource contention, inconsistent routing or replication delays. From observing I/O patterns in idealized circumstances, we hope to extract insights specific to the RNTuple-S3 integration for further development of the backend.

Simulated S3 Server. MinIO (MinIO, Inc., 2023) is an open-source object storage solution whose client-facing components are compatible with S3's APIs. MinIO can be used as a validating tool for object store middleware development, as it simplifies the management of locally set-up servers.

Hardware. For a preliminary setup, we instantiated a MinIO server on our benchmarking node `ntpl-perf-01`, while the client executed on a node from CERN open-

lab's `olsky-03` cluster.

Server node. 1× AMD EPYC 7702P CPU socket with 64 physical cores, running at 2.1 GHz Hyper-threading SMT enabled. 16 MB of L3 cache. 125 GB of RAM. Mellanox MT27800 ConnectX-5 Ethernet interface at 33 MHz with 40Gbit/s capacity, 64-bit width.

Client node. 2× Intel Xeon Platinum 8160 CPU sockets, each with 24 physical cores, running at 2.10 GHz. Hyper-threading SMT enabled. 33.7 MB of L3 cache. 187 GB of RAM. Intel Ethernet Controller X550 interface at 33 MHz with 10 Gbit/s capacity, 64-bit width.

Software. The operating system was Red Hat Enterprise Linux 8.5 (kernel 4.18.0-425). The MinIO version was RELEASE.2023-04-20T17-56-55Z. The project was compiled with g++ 8.5.0 and O2 optimization.

5.2.2 LHCb Benchmark

Table 5.3: Excerpt of the "B meson decays to 3 hadrons" (B2HHH) dataset, from CERN OpenData Run 1 for the LHCb experiment (LHCb collaboration (2017), 2017).

Entry	B Meson Data		Hadron 1 Data				…
	`FlightDist`	`Vertex`$\mathcal{X}^2$	`PX`	…	`Prob` π	`Charge`	`IsMuon?`	…
0	25.3	1.497	375.3	…	0.89	-1	false	…
1	94.7	1.38	-4985.13	…	0.04	-1	true	…
			… … …					
42	21.2	3.48	673.34	…	0.95	+1	false	…
			… … …					

The LHC beauty (LHCb) experiment investigates primarily the matter–antimatter asymmetry of the universe by observing interactions between B hadrons. For our evaluation, we use one of their findings from LHC's Run 1, the "B-meson decays to three hadrons" (B2HHH) dataset, which estimates the mass of the short-lived B-meson by tracking its decay to hadron particles. It has been made public

through CERN OpenData (LHCb collaboration (2017), 2017). The dataset has no nested collections, spans 26 columns and contains over 8.5 million events (i.e., entries), for a total uncompressed file size of 1.5 GB. Table 5.3 shows a partial excerpt of a few entries in the dataset.

The analysis of the B2HHH data is conducted by the benchmark program `lhcb.cxx` in Blomer et al. (2022). This benchmark is realistic, yet simple and well-understood. From the 26 existing columns, the analysis program iterates over all entries and 18 of the columns in B2HHH to build a histogram of the B mass spectrum, as demonstrated in Algorithm 6.

Pseudocode 6: LHCb Analysis Benchmark (B2HHH Dataset)

```
1  function LHCb(ntuple)²
2      ▷ Iterate over entries (i.e., LHC events with collision data).
3      for entry : ntuple.GetEntries() do
4          ▷ Filter out event unlikely to contain B meson.
5          if IsMuon in entry.Hadron_{1,2,3} then
6              continue
7          if ProbK in entry.Hadron_{1,2,3} < 0.5 then
8              continue
9          if ProbPi in entry.Hadron_{1,2,3} > 0.5 then
10             continue
11         ▷ Compute mass of event's B meson.
12         b_E ← 0
13         for h in entry.Hadron_{1,2,3} do
14             b_E ← b_E + GetKaonEnergy(h.PX, h.PY, h.PZ)
15         p_x ← Σ_{i=1}^{3} entry.Hadron_i.PX
16         p_y ← Σ_{i=1}^{3} entry.Hadron_i.PY
17         p_z ← Σ_{i=1}^{3} entry.Hadron_i.PZ
18         b_mass ← √(b_E² − ‖p‖)²
19         FillHistogram(b_mass)
20     return
```

The flatness of the ntuple and relatively high proportion of columns being read during the analysis make this end-to-end analysis especially I/O-intensive. Thus, it constitutes a natural candidate for the evaluation of new I/O features in RNTuple.

To simulate a high-volume data analysis, such as those typical on HPC clusters, we artificially extended the dataset through vertical concatenation. The final

[2] LHCb Analysis Benchmark on Github: `lhcb.cxx#L275`

content is equivalent to 10 identical copies of the original data, resulting in a 15 GB uncompressed dataset with over 85 million events.

Experiment Combinations

We break down the total number of runs conducted for this evaluation:

On `DAOS-Setup`, we used the following parameters for RNTuple-DAOS, for a total of **1224 combinations** before repetitions:

Versions. 5×. `v0-BASE`, `v1-PERS`, `v2-COAL`, `v3-COLO`, `v4-CAGE` (with 1 MiB cages);

Cluster Size. 2×. 50 MB, 100 MB;

Page Size. 7×. 32 kB, 64 kB, 128 kB, 256 kB, 512 kB, 1 MiB, 2 MiB, the latter excepted from version `v4-CAGE`);

Compression Algorithm. 3×. none, `zstd`, `lz4`;

Object Class. 6×. `OC_SX`, `OC_TINY`, `OC_MAX`, `OC_XSF`, `OC_RP_TINY`, `OC_RP_MAX`;

Repetitions. 5×.

Due to the number of combinations and relative lack of diversity in the results, not all are presented in this document.

On `S3-Setup`, because of the proof–of–concept nature of the RNTuple-S3 evaluation, we only executed our experiment on six page sizes (64 kB, 1 MiB, 2 MiB, 4 MiB, 8 MiB, 16 MiB), with a 500 MB cluster size and no compression algorithm or replication options. The experiment was repeated five times, for a total of **30 runs**.

5.3 Evaluation of the RNTuple-DAOS Backend

In this section, we present the findings from our evaluation of the RNTuple–DAOS backend proposed in section 4.3 on the `DAOS-Setup` platform described in section 5.2.

The DAOS object class is one of the parameters we studied in this evaluation. It regulates replication and sharding for blobs under DAOS KVS objects. A description of these properties for the studied object classes is present in the background section 2.3.

First, we compare all named versions of the backend (see Table 5.2) with a

fixed cluster size of 100 MiB and no compression, ranging over all page sizes, in order to establish which of the proposed improvements had the most impact on write and read throughput. Version `v4-CAGE`, with the caging mechanism toggled, is excluded from this study.

Then, we present the entire gamut of results for three versions: `v0-BASE` (baseline prior to our modifications), `v3-COLO` (proposed version with co-locality mapping, but no caging) and `v4-CAGE` with 1 MiB cages. Due to the cutoff at 1 MiB, the experiments for the latter version were not run for 2 MiB page size.

Lastly, we discuss observations and limitations of the backend through a statistical performance analysis on CPU utilization.

5.3.1 Version Comparison

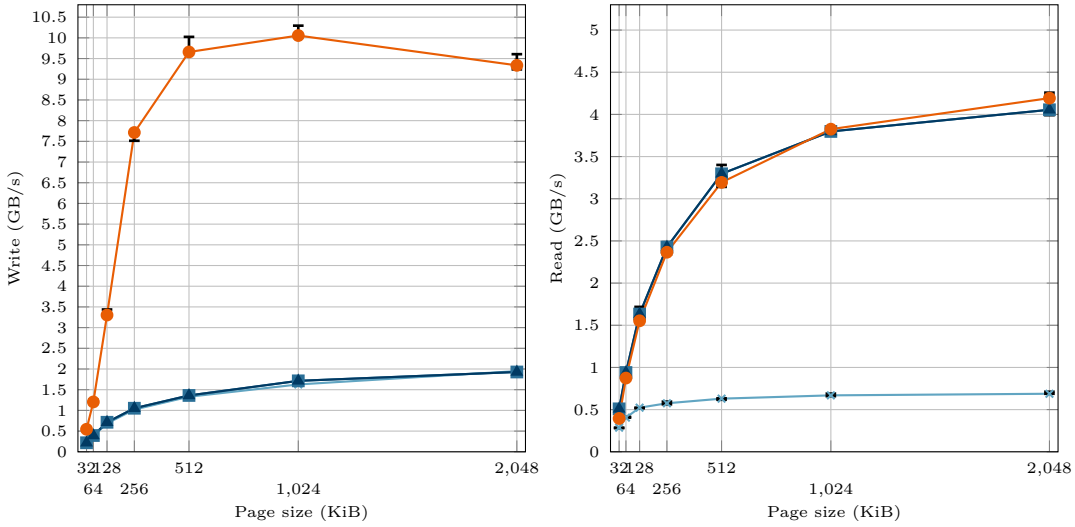
We begin by fixing the cluster size at 100 MB and using no compression in order to investigate precisely which versions introduce the greatest performance gains (or losses) among the following: `v0-BASE`, `v1-PERS`, `v2-COAL` and `v3-COLO`. From the baseline backend to the co-locality mapping proposal, each version builds on top of the previous one, as already specified in Table 5.2.

We allowed both the page size and object class parameters to vary to identify any residual effects w.r.t. the different request pipelines and mappings throughout the implementations. The results are plotted in Figures 5.1 and 5.2

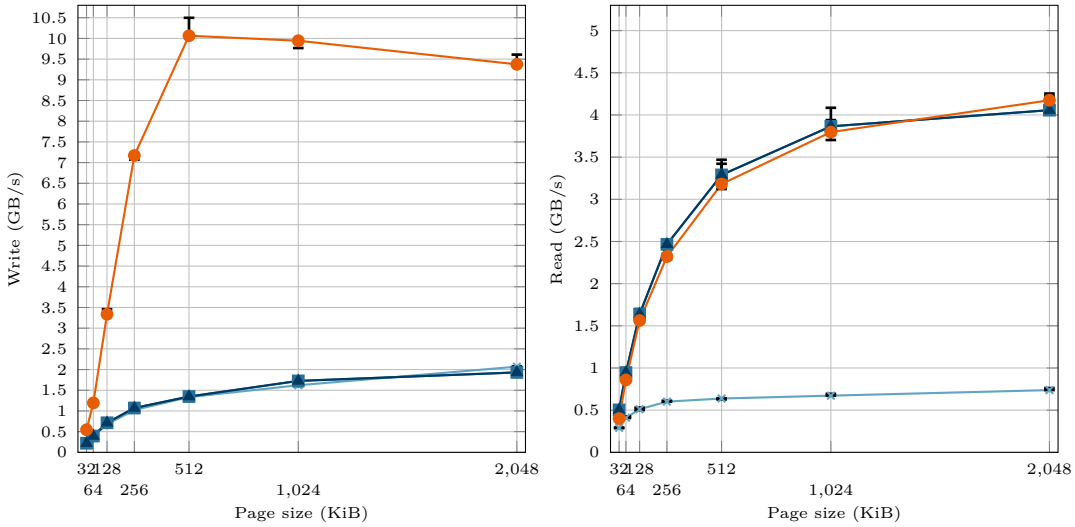
From the get–go, considering the results across object classes, three key takeaways are evident:

1. *Performance asymmetry w.r.t. implemented features.* The results suggest an asymmetry between request preparation efforts for writing and reading. Taking for example subfigure 5.1a, the entire feature pipeline is needed to achieve mensurable improvements in write throughput. Like a centerfold resting on top of the previous structures, it is only when the proposed mapping is toggled that we see performance gains with high peaks of 10 GB/s. The analysis, on the other hand, benefits immediately from a more sensible operation management that retains the persistent operation queue, achieving peaks of 4+ GB/s read speed. These results suggest that, depending on the use case – e.g., “write–once–read–thousands” –, the proposed mechanisms of request co-

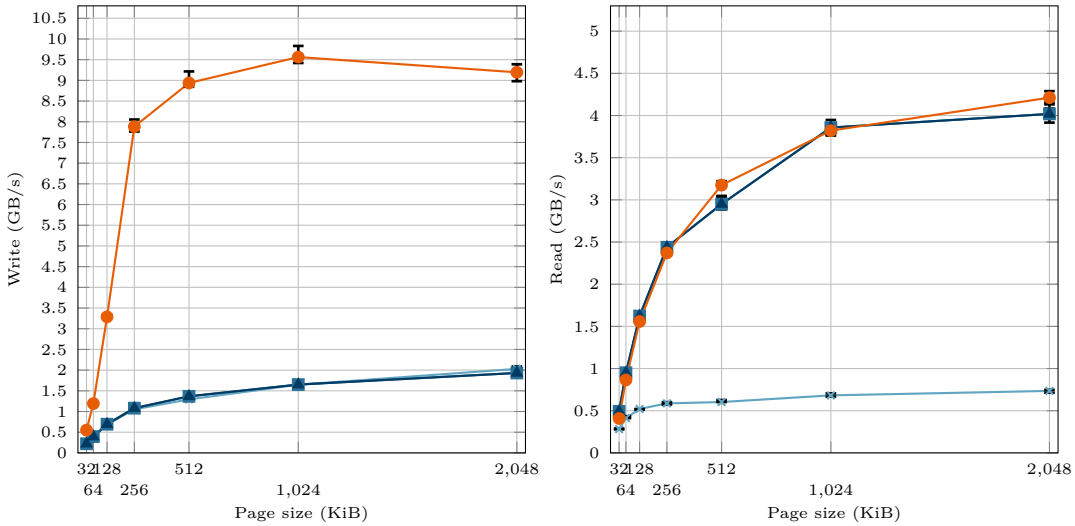
Figure 5.1: RNTuple-DAOS Version Comparison, `OC_{SX, TINY, MAX}`.



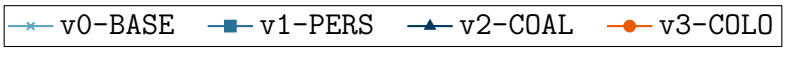
(a) 100 MB cluster size, no compression, `OC_SX` objects.



(b) 100 MB cluster size, no compression, `OC_TINY` objects.

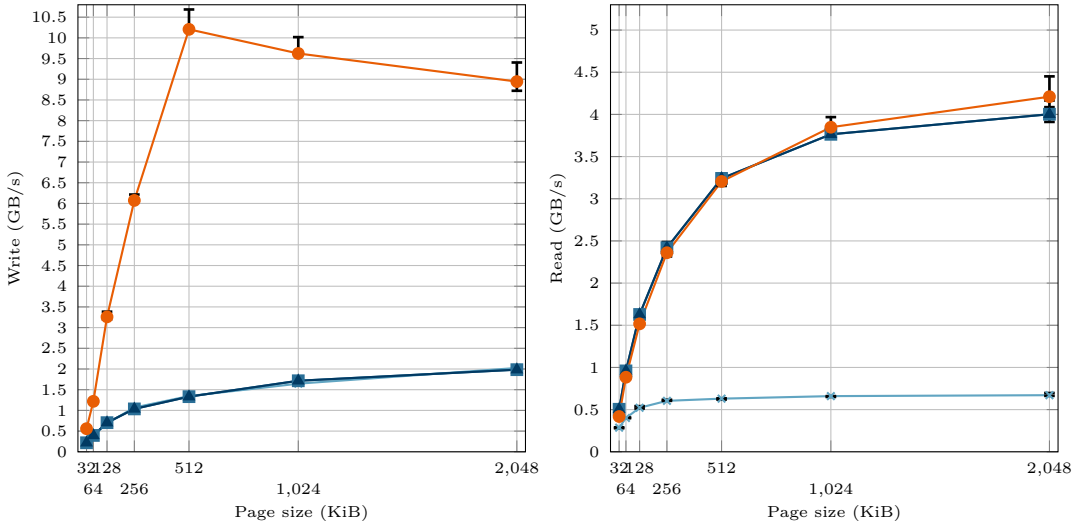


(c) 100 MB cluster size, no compression, `OC_MAX` objects.

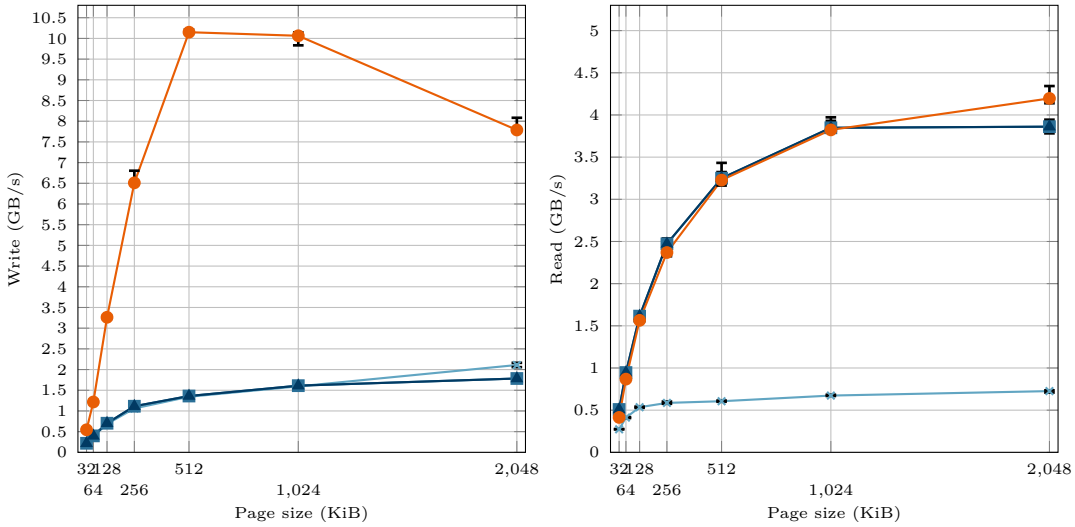


Source: The Authors.

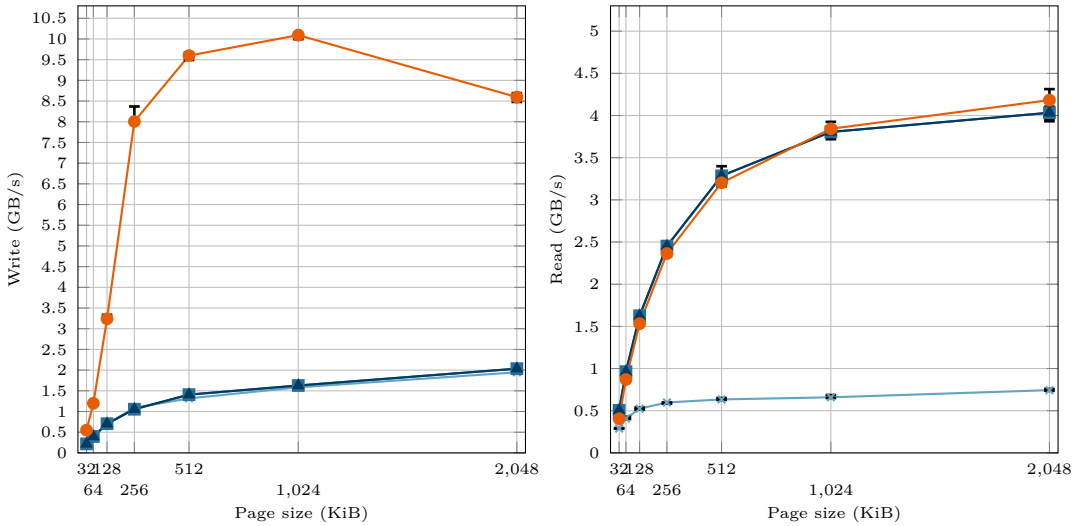
Figure 5.2: RNTuple-DAOS Version Comparison, `OC_{XSF, RP_TINY, RP_MAX}`.



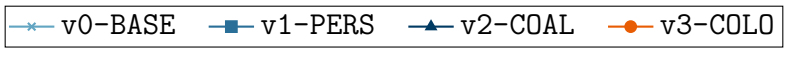
(a) 100 MB cluster size, no compression, `OC_XSF` objects.



(b) 100 MB cluster size, no compression, `OC_RP_TINY` objects.



(c) 100 MB cluster size, no compression, `OC_RP_MAX` objects.



Source: The Authors.

alescing, vector I/O and co-locality mapping may only introduce unnecessary complexity and buffering overhead without clear benefits. On the other hand, if fast population of the object store is a priority, the entire "feature ladder" is required for best results.

2. *Impact of replication factor for the setup.* Some object classes show differences of over 10 % in write throughput, e.g., the cases of 256–512 kB pages in subfigures 5.2b, 5.2c, 5.1a and 5.2a. It makes sense that the classes `OC_RP_TINY` and `OC_SX` on one side, and `OC_RP_MAX`, `OC_XSF` on the other, have the biggest throughput discrepancy on write, as they are on opposite extremes of the replication factor.
3. *Unusually high write/read throughput discrepancy.* We make note of the substantial discrepancy between the peak write and read speed, respectively around 10 GB/s and 4.5 GB/s. Usually, the write speed is lower than the read's, as exemplified in the same setup with the IOR benchmark (see results in Table 5.1). A possible explanation is that DAOS reports an update procedure as completed before data has finished moving, due to uncertainty by the application on when an RDMA transfer is done. Alternatively, the high speed on update is related to DAOS caching the update in PMem storage, so that later, the contents can be committed into slower, SSD storage (see 2.5) at the same target location.
4. *Faster throughput than the IOR benchmark.* From section 5.2 and Table 5.1), the results shown here trump those obtained in the IOR benchmark. This can be explained by the fact that operations go through DFS in IOR to measure DAOS performance, as it is designed to benchmark POSIX I/O systems. However, IOR may present inferior performance simply due to access patterns and circumstances of the experiment.
5. *Higher variance for `v3-COLO`.* As the highest performing curve, the version `v3-COLO` is the most noticeably reactive to latency variance (see error bars) and to the chosen object class. Particularly, we note the almost ritual steep decline in write throughput at the 2 MiB page size, wherein some object classes are more affected than others. It may be the case that the small cluster is causing contention, which impact more strongly classes with high replication factors, such as `OC_RP_MAX`, when dealing with large blobs.

5.3.2 Analysis of Native Parameters

We continue our evaluation by analyzing parameters native to RNTuple (page size, cluster size, compression algorithm) and DAOS (object classes governing replication and sharding strategies).

RNTuple Page Size

From subsection 4.1, pages are the basic unit of storage in RNTuple, and our background research indicates that a page size of 64 kB offers sound performance for file-based backends.

Our evaluation extensively suggests that the above assumption is not applicable for object stores, and that the appropriate transfer buffer size for fast data ingestion is orders of magnitude higher.

RNTuple Cluster Size

The cluster is a size-bound partition of sequential entries in an ntuple that serves as a checkpoint for I/O. Changing the size of the cluster impacts the average amount of data from pages accessible for transfer at any point of writing or analysis, which has ramifications on throughput.

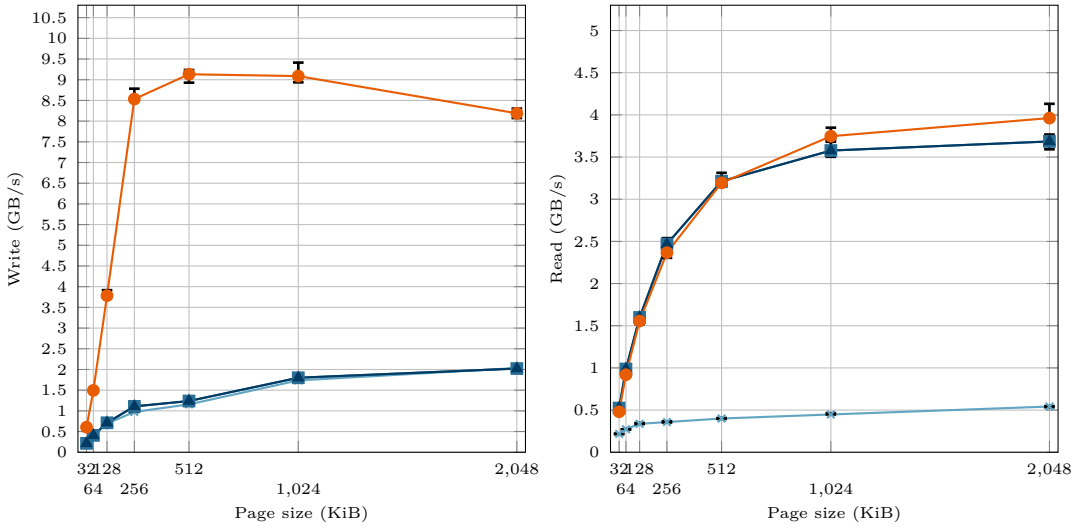
Figure 5.3 demonstrates that a larger cluster directly, albeit marginally, impacts write throughput when requests are coalesced and committed using the proposed co-locality mapping. We estimate this is due to the higher amount of pages that can be requested at once when operating with larger clusters.

RNTuple Compression Algorithm

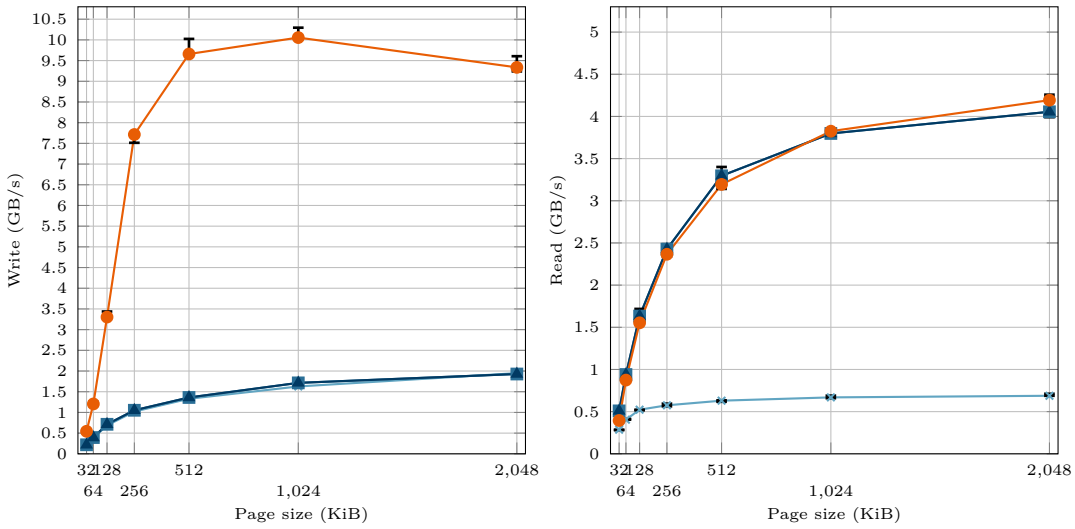
In general, as the analyses in section 5.3.2 indicate, the size of the transfer buffer has an indubitable effect on performance for our setup. Doubling the page size can have as much as a 180% improvement on write throughput and 100% gain on read speed (e.g., from 64 kB to 128 kB in Figure 5.2b). Given that, one could assume the use of compression algorithms at the page level would degrade transfer speed in accordance with their compression rates.

Some experiments demonstrated this performance degradation when compression was toggled. For example, Figure 5.4 shows a stark decrease in write

Figure 5.3: RNTuple-DAOS Cluster Size Impact.



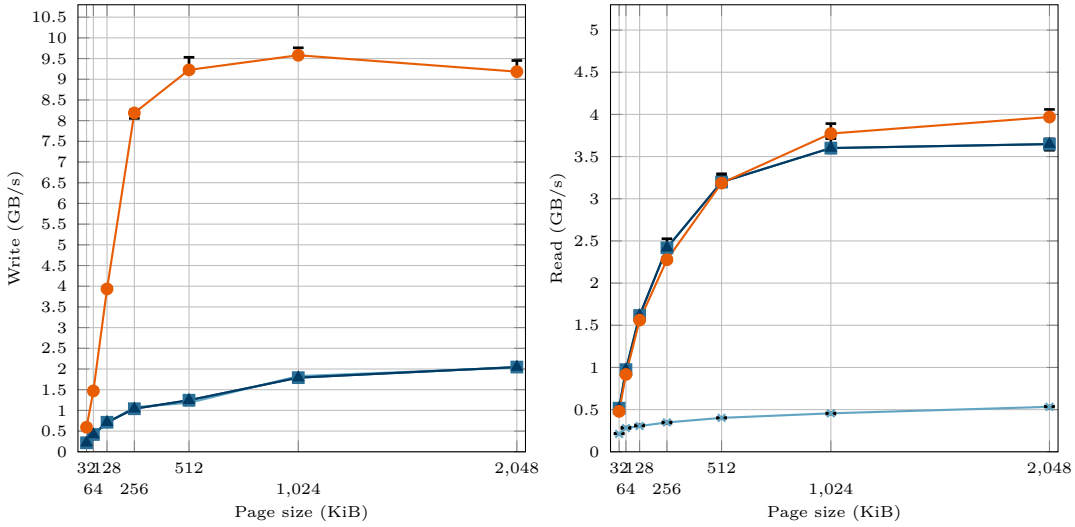
(a) 50 MB cluster size, no compression, `OC_SX` objects.



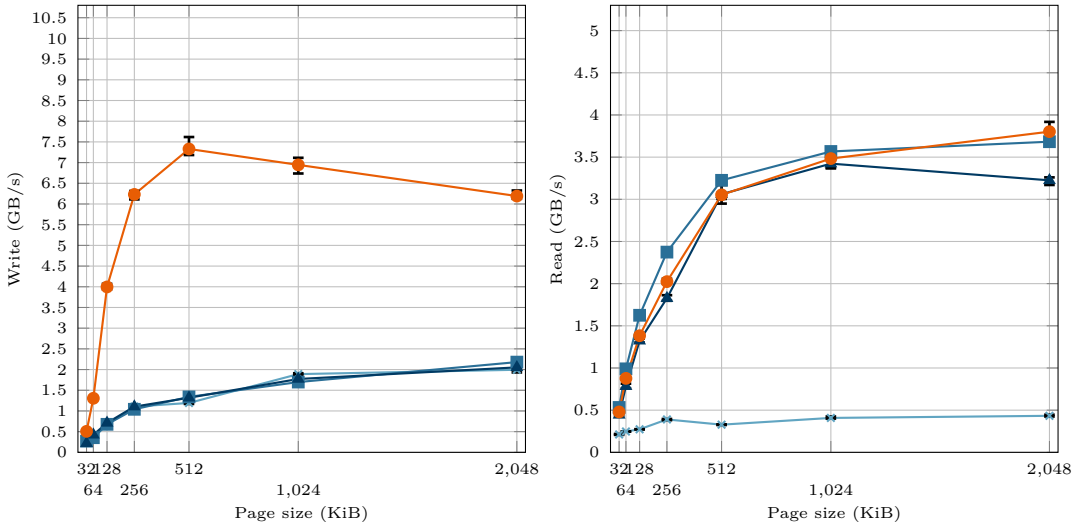
(b) 100 MB cluster size, no compression, `OC_SX` objects.

Source: The Authors.

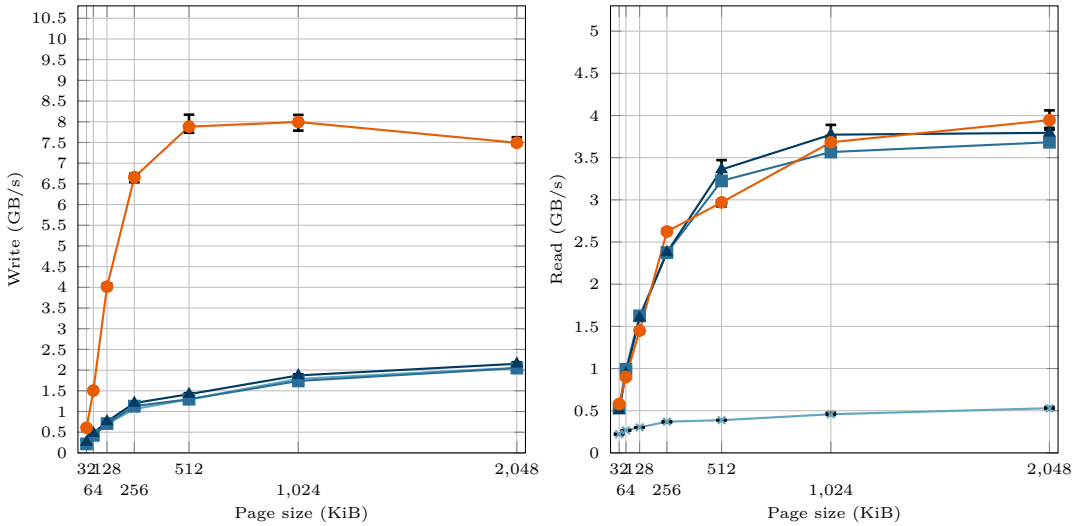
Figure 5.4: RNTuple-DAOS Compression Impact {none, `zstd`, `lz4`}.



(a) 50 MB cluster size, no compression, `OC_TINY` objects.



(b) 50 MB cluster size, `zstd` compression, `OC_TINY` objects.



(c) 50 MB cluster size, `lz4` compression, `OC_TINY` objects.

Source: The Authors.

throughput for the cases of `zstd` and `lz4` compression in comparison to no compression. This trend is however less clear on reading. The impact of compression on throughput should be investigated further.

5.3.3 Artificial Page Size (Caging)

As described in chapter 4, the caging mechanism was conceptualized after observing the impact of RNTuple's native page size on throughput seen in subsection 5.3.2, which confirm that the traditional page size of 64 kB for the file backend does not port well to object stores.

However, this insight does not affect any preexisting ntuples that were created with optimal configurations for file-based systems. To speed up HPC cluster data ingestion, the entire ntuple would first need to be reprocessed with the refactored layout before the analysis pipeline can properly start.

Subsection 4.3.3 explained that the proposed concatenation mechanism seeks to avoid the aforementioned ntuple refactor with an approach based on scatter-gather I/O. Our intent was to benefit from server-side concatenation of buffers by tuning the mapping function and submitting batched-up requests.

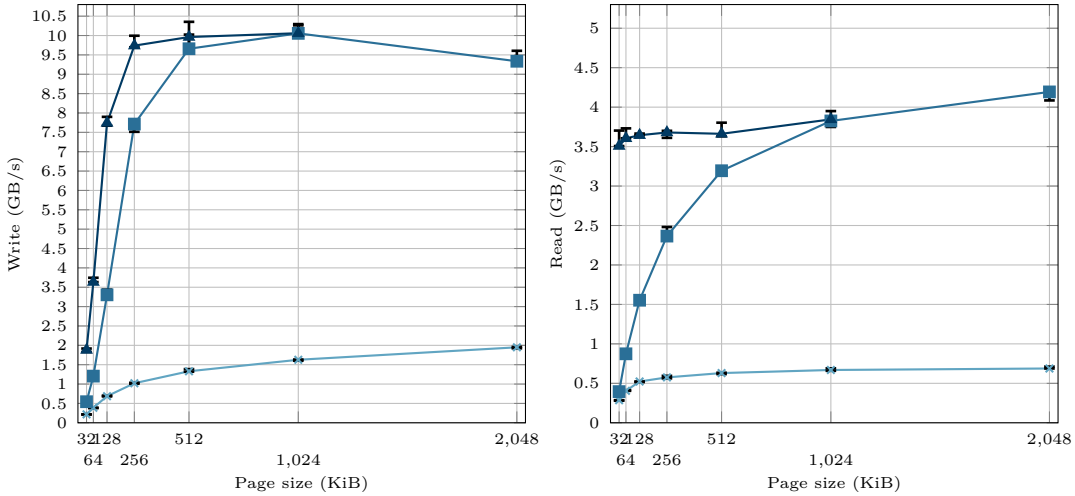
From our experiment in Figure 5.3.3, the method had the expected effects on fetching efficiency, with reading throughput reaching the artificial target throughput of the larger chunk (in our tests, 1 MiB). In short, we validated that:

a. Separate, user-side buffers are mapped to the same blob in object storage, despite being relayed through different IOV descriptors in an update request;
b. Data fetching achieves the read throughput of the targeted chunk size, i.e., the speed observed in the evaluation for the corresponding native page size, even though the request comprised multiple pages.

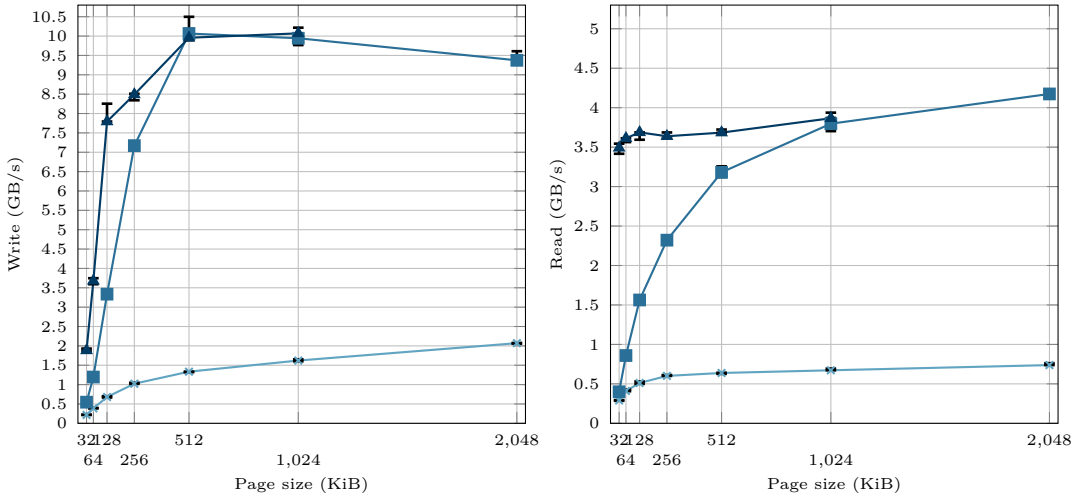
The fact that fetching achieves the target throughput is not surprising. Once in server storage, cages are no different than any contiguous, unitary blob, and so should trivially match the throughput of native page sizes at the cage size targeted.

The same cannot be said for data ingestion with the caging mechanism, as the writing throughput did not match the targeted estimates. Still, there is a tangible improvement compared to non-concatenated native pages, when looking at page sizes smaller than 1 MiB when setting the maximum cage size to that value. As

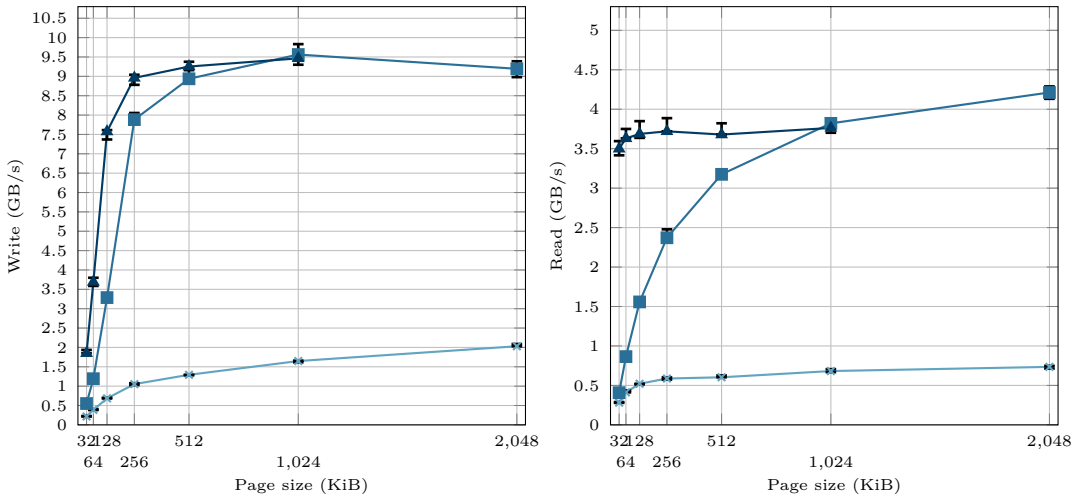
Figure 5.5: RNTuple-DAOS Caging Throughput, `OC_{SX, TINY, MAX}`



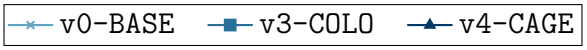
(a) 100 MB RNTuple cluster size, no compression, `OC_SX` objects.



(b) 100 MB RNTuple cluster size, no compression, `OC_TINY` objects.

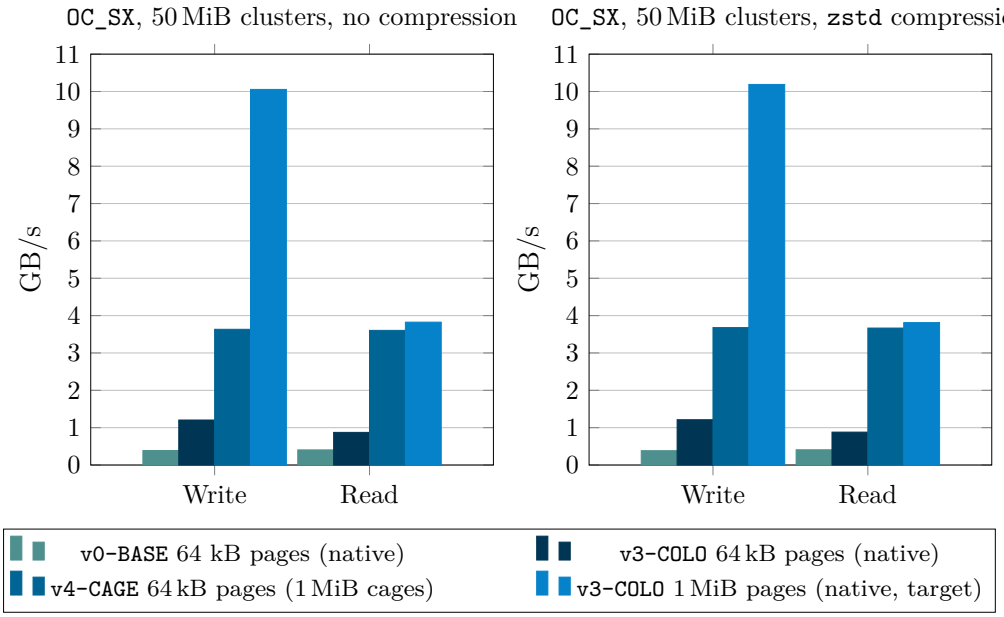


(c) 100 MB RNTuple cluster size, no compression, `OC_MAX` objects.



Source: The Authors.

Figure 5.6: Throughput comparison for native 64 KiB pages and 1 MiB concatenation under different RNTuple-DAOS versions.



Source: The Authors.

visualized in Figure 5.6, the performance of 64 kB pages concatenated into 1 MiB cages is over twice that of 64 kB pages in the native evaluation (subsection 5.3.2). As the page buffers increase, this advantage starts to saturate until it plateaus for pages sized 256 kB or bigger.

We now consider possible explanations for the observed bounded write performance which prevents the set of smaller, scattered buffers from achieving write throughput much closer to the target:

1. Throttled scalability in RDMA, inherent or caused by buffer mismanagement. This packs two possibilities. The more unlikely one is that the noncontiguous set of page buffers from the still–scattered cage prevents the transfer implementation by OFI `verbs` from scaling to the expected throughput in scatter–gather I/O. More likely and more insidiously, the higher number of buffers incurs inordinate overhead costs directly associated with RDMA transfer, and this overhead is not being amortized throughout execution; i.e., there are processing costs repeatedly paid for each buffer and for each cage sink procedure. In this case, the higher number of buffers once again would explain how smaller pages are hit more heavily.
2. DAOS server contention or limitations on the number of IOVs that DAOS

servers can withstand at once. In this case, the high number of individual buffers could be causing causing server–side contention, either by implementation or due to a partitioning of the NIC's available bandwidth. If this were the case, our experimental setup with two servers should show at least alleviated effects thereof when subscribing to object classes with high replication factors. However, no such mitigation was seen.

Due to the high impact of memory registration costs on RDMA–based transfer (see the following subsection) and due to general efficiency concerns, this evaluation has guided our development focus primarily toward improving memory buffer reuse in RNTuple, which is a work in progress.

5.3.4 Performance Analysis

In addition to the above benchmarks, we collected performance statistics of our backend's CPU usage with the help of `perf`[3], a statistical profiler for Linux systems, and the FlameGraph visualization tool[4], which agglutinates the collected stack frames in `perf` in a visually–comprehensive way.

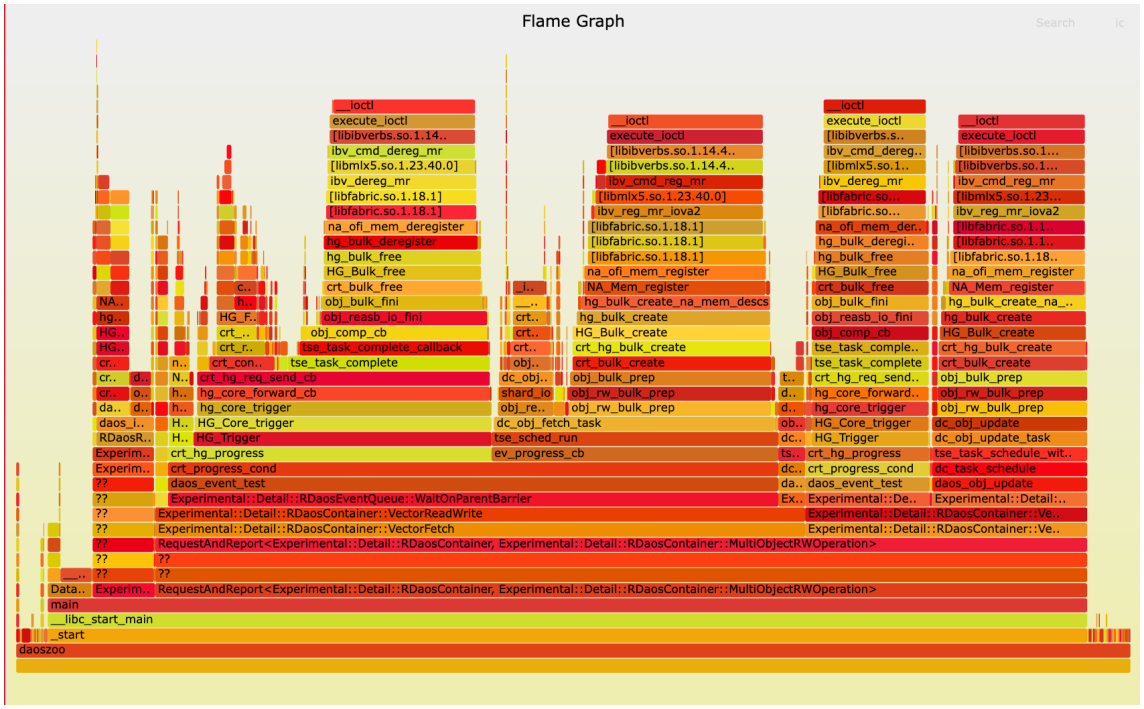
A stack trace profiler collects snapshots, or frames, of the stack trace at a uniform sampling rate during the execution of a program. It is thus a statistical tool to measure which function calls have demanded the most CPU resources in a program. Flame graphs summarize these stack traces in colorful piles; the widest the slab, the more frames contain a certain function call in its trace. Note that pile height holds no importance with regards to resource usage, and that piles are not organized in chronological order.

In Figure 5.7, we present two flame graphs which exemplify the CPU engagement during data ingestion and retrieval in the RNTuple–DAOS backend.

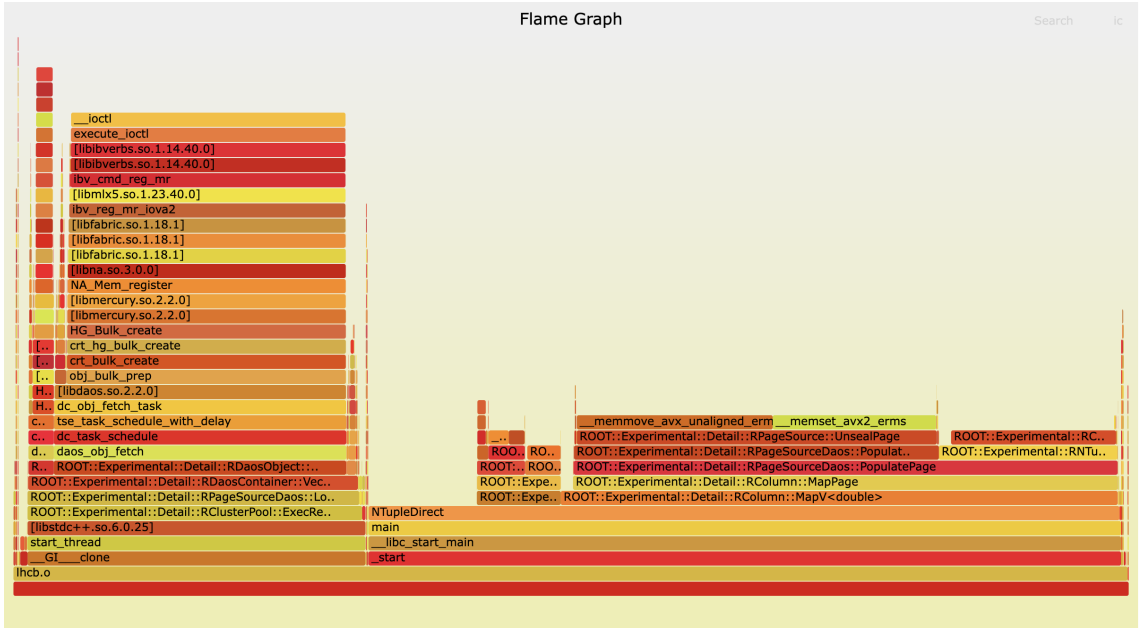
Figure 5.7a contains both the writing and reading stages of a simples standalone development tool built by us and nicknamed "RNTuple Backend Zoo". This tool generates random data to populate a given object storage location and subsequently retrieves the data, confirming its validity via a hash function. Thus, the Backend Zoo is a useful tool to debug backend behavior without the obfuscating complexity of ROOT.

[3]Online documentation: `perf` official wiki.
[4]Repository available on Github: brendangregg/FlameGraph.

Figure 5.7: Stack flame graphs, CPU usage, RNTuple–DAOS backend.



(a) Standalone tool, write/read for 64 kB buffers (31 billion frame samples).



(b) LHCb analysis with decompression, 64 kB buffers (494 billion frame samples).

Source: The Authors.

Lastly, Figure 5.7b documents a complete LHCb analysis for 64 kB pages with compression, which calls the real ROOT RNTuple. As expected, the majority of the CPU effort is spent on decompression during the actual analysis of HEP events, represented there by the call to `UnsealPage()` under the `NTupleDirect()` function in the LHCb analysis script.

Most prominently, these graphs reveal the expensive effects of memory registration in RDMA–enabled clusters. RDMA depends on given ranges of the application memory to be made visible to the adapter endpoint before remote transfer can begin. This requires a system call to register the memory region, which is a significant part of the overhead associated with such transfer methods. To amortize these costs, memory regions that will be reused for transfer buffers should remain registered throughout execution, or kept in a cache to be lazily de-registered once idle (MIETKE et al., 2006).

These steps are identifiable in the graphs by the `verbs` calls `ibv_cmd_reg_mr()` and `ibv_cmd_dereg_mr()`, implemented by `libfabric` for the Mellanox InfiniBand interface. These calls clearly dominate the performance analysis in Figure 5.7a, where the standalone procedure is simple enough and does not engage in expensive compression and decompression.

But even the LHCb analysis in Figure 5.7b suffers from memory registration almost as much as it is impacted by decompression, which should be the biggest source of CPU–boundness. This presents a problem: in the same analysis run, memory regions which should already be registered and available for the next batch of pages are incurring additional overhead costs before data is transferred. Since calls to `ibv_cmd_dereg_mr()` are not visible in the flame graph, the regions are not being forcefully de-registered before they are no longer needed. Instead, the graph suggests that memory regions that do not entirely overlap with existing ones could be undergoing memory registration because transfer buffers are not being efficiently reused by RNTuple.

RNTuple–DAOS Evaluation Summary

In this evaluation, we systematically measured the real RW throughput of multiple versions of the RNTuple-DAOS backend across the RNTuple and DAOS parameter spaces, in order to understand their impact on performance for realistic

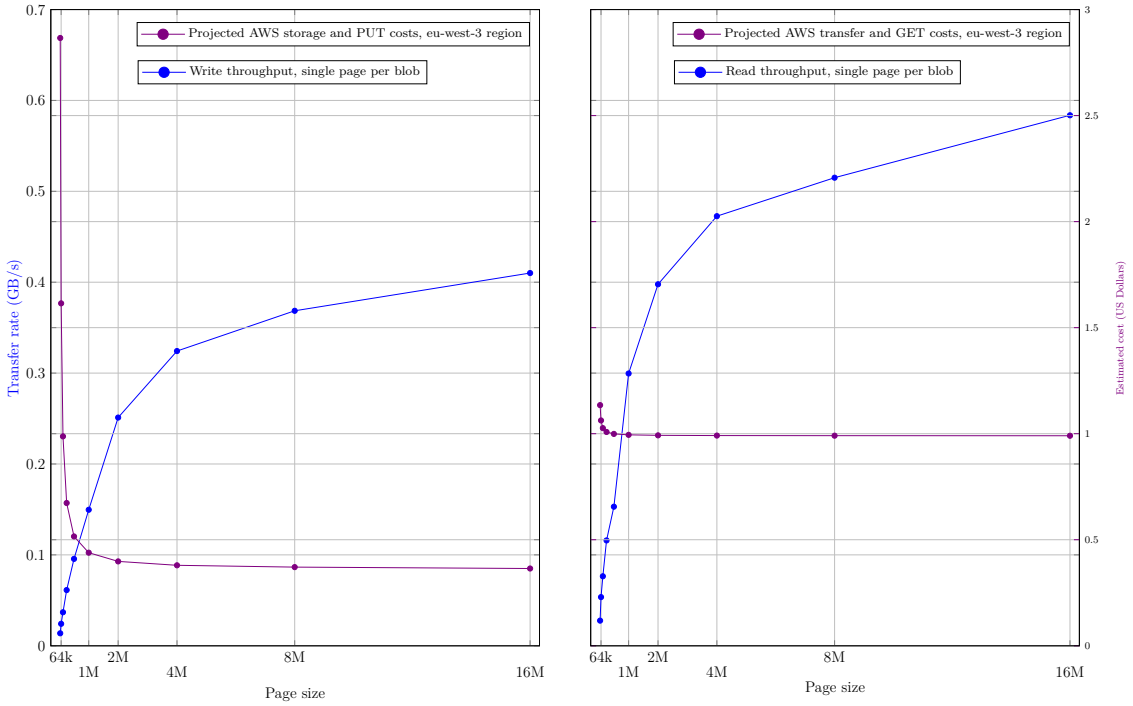
HEP analyses. We found that the introduction of the co-locality mapping function is the cornerstone for high write throughput, whereas it is inconsequential for reading in the LHCb analysis workflow; for the latter, scalability is achieved with a persistent operation queue that does not invoke system calls throughout execution, unlike the baseline. The throughput sensitivity to transfer buffer size was solidified across all scenarios; in particular, we were able to validate the caging approach as a viable way to achieve higher throughput in spite of smaller native page sizes (and thus, transfer buffers), both for reading and, to a lesser degree, for writing. We observed minimal effect of compression algorithms on throughput, even though that directly affects buffer size at the sink and source level. The evaluation revealed that bigger clusters work in the favor of the backend when using the co-locality mapping. The different DAOS object classes have a definite effect on performance; however, the limited scope of the setup, with one single client and two servers in a modest HPC cluster, hinder our ability to draw conclusions on the impact of the replication factor and sharding on analyses. For that, distributed tests should be conducted on a more powerful HPC cluster. Finally, a statistical analysis showed CPU bottlenecks associated with memory registration in RDMA–interconnected nodes, which indicates that better buffer management in RNTuple might lead to improved performance on these interfaces.

5.4 Evaluation of the RNTuple-S3 Backend

For an evaluation of our experimental RNTuple backend for cloud-based object stores, we designed a test similar to the one in subsection 5.3.2. Due to latency and scalability concerns, the test uses abnormally larger pages (natively, i.e., contiguously). For that, we fixed the cluster size at 500 MiB so that each cluster can support at least one page for each of the 26 columns in the LHCb dataset.

With that in mind, we varied the native page size and measure the resulting I/O throughput for the usual LHCb analysis without data compression. Note that page sizes at this scale are not expected to be recommended for the RNTuple Cloud use case due to loss of data granularity and a steeper page size imbalance between different data types in the same cluster.

Figure 5.8: RNTuple–S3 throughput and cost estimate for 500 MiB clusters, no compression, varying page sizes, evaluated with a simulated server in idealized network settings.



Source: The Authors.

Throughput × Native Page Size

The leftmost y-axis in Figure 5.8 records the throughput, in GB/s, as measured for the LHCb-based end-to-end analysis across page sizes ranging from 32 KiB to 16 MiB.

The curves observed for both the writing and reading stages are not much differently-shaped than the ones presented in the previous section for DAOS.

In both cases, there is a positive relation between throughput and page size, i.e., larger transfer buffers again lead to faster transfer rates. The curves for writing and reading are both monotonically increasing throughout the x-axis. In this experiment, we identify a turning point at MiB scales, which starts off a plateau trend. We did not evaluate page sizes above 16 MiB.

The key difference between this backend evaluation and the one in subsection 5.3.2 is the start of the plateau at a larger page size. Whereas the DAOS backend indicated a sensible trade-off around 512 KiB–2 MiB-sized pages, a similar plateau pattern places our desirable transfer buffer size around the 8 MiB mark here. Applying the same reasoning, we arrive at a 4-16× higher target.

Discussion on Latency-Curbing Strategies

Informed by the above results, we discuss the viability of the caging mechanism in a matured RNTuple-S3 backend.

The results suggest that the caging mechanism might not be enough for a performant RNTuple backend for the cloud. In the DAOS backend (see 5.3.3), caging is limited to pages from the same page group up to 1 MiB in uncompressed size. This can be a significant amount of data for a given page group, since the cluster size (50 MiB by default) is the limiting factor across all columns. With a reasonable data distribution, clusters with hundreds of columns are unable to scale up to the 2–8 MiB cages suggested by Figure 5.8.

Option 1: Large Clusters and Page Group → Object Mapping. For large enough clusters (e.g., 100-500 MB in size), an alternative to consistently reach higher throughputs is to map entire page groups as S3 objects. Effectively, this is what happens already through caging for large enough target sizes. However, its feasibility depends on the memory budget, as RNTuple would become more memory-hungry due to additional allocation quotas for page compression and decompression. Also, buffering data would strain memory budget constraints. As with any WORM application, the most frequent I/O operation between RNTuple and object stores is assumed to be a fetch operation. When reading data during analyses, the recommended read-ahead window (3), which pre-fetches subsequent compressed clusters in advance, by itself triples the memory budget needed to process clusters.

Option 2: Cluster → Object Mapping. Going farther, a more drastic possibility is to map each RNTuple cluster as its own S3 object. Cloud services are often advertised for large files (e.g., movies, datasets) as the higher latencies can greatly affect performance at smaller scales of transfer. Thus, a cluster-to-object mapping could potentially leverage a much higher throughput. At such volumes, certain S3 implementations automatically trigger multipart uploading – a feature for writing large S3 objects in parallel despite data being stored in a contiguous buffer, something likely to be beneficial to performance only at the scale of 100+ MB objects. This is conditioned on the hosting bucket having the feature enabled and belonging to an S3 region that supports it.

While this approach maximizes potential ingestion speed, it complicates the

fetching procedure considerably. From background sections 2.1.2 and 2.2, the HEP use case follows a selectively columnar reading pattern. Thus, pages from only a handful of columns may be requested for a given cluster. Storing clusters as a single, indivisible unit in object storage negates R/W throughput gains for any reading pattern that is not very dense (i.e., a majority of columns requested in each cluster).

For this reason, mapping clusters to S3 objects is strongly conditioned on another feature: byte-range requests. Byte-range requests break away from the simple `GET` and `PUT` object requests, traditional to cloud-based object stores like S3, in order to support partial fetch requests (and, less commonly, update requests) for an arbitrarily long and contiguous segment of one of its blobs. The S3 API does support this feature, backed by an underlying HTTP request with the `Range` header from RFC-2616 (NIELSEN et al., 1999).

Partial reads would enable a cluster-to-object mapping likely to perform ideally for writing and for sparse reading of ntuple fields. However, it is still unclear if such a solution would be as performant for column-dense analyses; instead, this scenario could lead to socket contention caused by the multiple `Range` requests issued at once, one per column. Should this problem arise, the common strategy of concatenating any contiguous range requests for neighboring columns in the same cluster would be advised.

Request Density × Page Size

Commercial cloud providers like AWS S3 and Microsoft Azure have a pricing model that charges, among others, for storage and network utilization.

The rightmost y-axis in Figure 5.8 presents an estimation of the aggregate cost associated with storing and accessing data in an S3 bucket with standard plan rates.

Cost Estimate Calculation

We based our estimate of the financial investment of each data ingestion and analysis step on AWS S3's basic pricing information (Amazon Web Services, 2023), which lists the costs to store and retrieve data across several of their service tiers.

We assumed the subscription to the "S3 Standard" service tier based on an European location (Paris region, server "`eu-west-3`"). In this tier, there are fixed monthly costs for storage per GB, as well as fees for each individual `GET` and `PUT` request issued, among others. Our calculations are as follows:

$$C_{write}(\mathcal{D}) = q_{\texttt{PUT}}\mathcal{D}_p + q_s\lceil 10^{-9}\mathcal{D}_{size}\rceil \quad \text{[i.e., PUT requests + storage volume]} \quad (5.1)$$

$$C_{fetch}(A_{\mathcal{D}}) = q_{\texttt{GET}}\mathcal{A}_p + q_t\lceil 10^{-9}\mathcal{A}_{size}\rceil \quad \text{[i.e., GET requests + transfer volume]} \quad (5.2)$$

where:

$\mathcal{D}$ = ntuple dataset.

$A_{\mathcal{D}}$ = analysis based on a given ntuple $\mathcal{D}$.

$\mathcal{D}_p$ = number of data and metadata pages in dataset (thus, single object requests).

$\mathcal{A}_p$ = number of pages requested in a given analysis $A_{\mathcal{D}}$.

$\mathcal{D}_{size}$ = sum of all page sizes in dataset, in bytes

$\mathcal{A}_{size}$ = sum of all page volume requested during analysis, in bytes.

$q_{\texttt{PUT}}$ = S3 cost, in United States Dollars (USD), per single `PUT` object request (i.e. write).

$q_{\texttt{GET}}$ = S3 cost, in USD, per single `GET` object request (i.e., fetch).

q_s = S3 storage cost per GB under first rate (i.e., first 50 TB in bucket, Standard), in USD.

q_t = S3 outbound transfer costs per GB under first rate (i.e., first 10 TB/month, Standard), in USD.

From Amazon Web Services (2023), the costs quoted in our evaluation for the named server were $q_{\texttt{PUT}} = 0.0053$, $q_s = 0.024$, $q_{\texttt{GET}} = 0.00042$, $q_t = 0.09$, all in USD.

S3 offers more elaborate storage tiers that could prove useful in future analyses. Tiers are approximately classified by the expected demand for object access, with packages for frequent, infrequent and very infrequent ("S3 Glacier") access, all

of which regulate availability via data replication, edge caching or long-term storage on hdd. An "Intelligent Tiering" option promises to adapt this tiering for each object based on recent access patterns.

However, these are commercial subscriptions which may not reflect the reality of a real-world leveraging of S3 infrastructure for LHC data analysis. In particular, the specification and deployment of a federated storage solution for HEP research falls outside the scope of this thesis and of RNTuple as a project.

Discussion on the Cost of Cloud-Based Analysis

The results demonstrate the impact of native page size on total storage and transfer costs for the writing and analysis fetching stages. Since the dataset size does not change, as the maximum uncompressed page size increases, the fewer pages are stored as S3 objects and requested via `PUT` or `GET` calls. This leads to a 6× cheaper writing stage when comparing 64 kB pages to 4–16 MiB ones, accompanied by a decrease of 30% in the analysis. Note that, as a WORM pattern dataset, the writing stage is expected to take place between once and a handful of times, while the same data may be read thousands of times by different users, analysis workflows or experiments.

This cost simulation exercise is meant to highlight and quantify the financial incentive in having a smaller number of pages, each larger in size on average. First, it shaves off costs for repeated analyses due to lesser volume of S3 object requests, e.g., by 25% from 64 kB to 2 MiB sizes for the LHCb dataset artificially extended tenfold. Second, with higher compression potential for bigger data chunks, the long-term maintenance and storage costs are also likely to be reduced, as these are based on the total stored content in a user's bucket.

Ultimately, the impact of compression on throughput and storage costs was not studied in this experiment due to scope and the minimal differences observed in the DAOS backend evaluation already presented beforehand. Likewise, the simulated setup dissuaded a realistic evaluation that considered replication on real S3 server nodes and edge locations; as a proof-of-concept, we expect our current results to lead to testbed access opportunities for the continued development of RNTuple's cloud backends.

5.5 Design Considerations for a Generic Object Store Backend

We concerted our development of RNTuple–DAOS and RNTuple–S3 to maximize the similarity between the backends. This common approach stems naturally from the task of adapting RNTuple to an object-centric data model, converging toward concepts and patterns likely to appear in the implementation of other object store backends for RNTuple in the future.

Given the proven potential of object stores for HEP analysis, and as a step toward widespread coverage of existing scalable storage infrastructure for LHC experiments, we expect an extraction of these common concepts into a generic object storage layer to follow. This layer would connect RNTuple's sink and source modules to concrete API implementations for myriad object storage providers. Thus, we identify below the key considerations for the design of such a generic backend.

→ **Layered design.** RNTuple's sink and source already offer interfaces that can be specialized by backends; notably, the file and DAOS backends override their virtual methods. An additional inheriting layer that inserts a blueprint for granular object store interoperability and management is recommended to hide backend details behind generic, opaque structures and templates.

→ **User-level namespaces.** Private namespaces are archetypal in the context of scalable, shared hardware resources such as data centers, serving as a partitioned directory for user data. Examples from this work include DAOS containers and S3 buckets. Their programmatic counterparts act as a bridge between RNTuple sink and source data structures and the underlying backend APIs.

In a generic backend, an "`RBucket`" manager structure should be encumbered with:

✓ **Connection management:** establishing the connection with remote storage, instantiating fabric endpoints and managing read and write credentials securely. For certain services, this entails parsing a server region as part of the namespace URI.

✓ **ntuple-wide management:** mapping an ntuple to an index and resolving it (when applicable), preventing mapping collisions in the process; keeping track of all ntuples in the namespace; listing and transforming

all of an ntuple's objects in storage (e.g., changing permissions and replication, dropping or duplicating data), etc.

- ✓ **Storage health and access patterns:** collecting statistics, at the level of either object, ntuple or namespace, potentially optimizing redundancy and thus availability at more impactful granularities.
- ✓ **Generalized operations:** providing a flexible interface for bulk operations that can be exploited by the underlying storage implementation, depending on support by the API and the chosen server. This interface would assume native support for either dedicated vector write and read operations or multiple, simple ones in parallel, as well as for partial, byte-range requests in fetch operations; the concrete implementation may then leverage the appropriate API calls or, if absent, implement an equivalent or simplified approach, depending on performance requirements. This interface must make as few limiting assumptions as possible, as optimizations such as request coalescing are not available before mapping is applied.

→ **Permissive mapping:** determining the exact mapping between units of RNTuple data (e.g., pages, cages, page groups, clusters or cluster bunches) and the various potential storage units (e.g., object, distribution, array element, region edge) is to be kept close to the implementation details of the backend. The generic layer should expose a virtual interface that enables rich mapping strategies, passing all of the above information for the concrete backend to exploit them.

→ **IOV-based buffer management.** The implementations of RNTuple's sink and source are based on page chunks defined by contiguous memory segments directly representable as IOV structures. Some cloud object store APIs utilize streams for relaying content, most notably in fetch operations; to circumvent that, custom stream structures can be implemented to still rely on a pre-existing buffer corresponding to the sealed page content and described by the IOV; going further, these custom implementations can take multiple IOVs in order to enable caging-type concatenation in backends that do not present the scatter-gather interface exploited for the DAOS backend in section 4.3.

→ **Redundancy shorthands.** A set of desirable configurations specifying de-

grees of replication, sharding and data protection should be generically defined by RNTuple and translated to the closest available alternative by the concrete backend layer, thus transparently to the user. This may be useful when transferring ntuples from one object store to another, or even to regulate data redundancy at the (generic) namespace level in an automatic fashion, based on access pattern metrics collected by that entity.

→ **Generalized number of namespaces.** A generic layer allows enough latitude to explore a generalization on the number of namespaces simultaneously connected. Applications for this are limited: examples would include writing out data to different services in parallel, under the assumption that the adapter interface would not be saturated due to network latency, or writing out to the same service, but different server regions, as these imply a different namespace for certain providers like S3. This is similar to the related work in (PADULANO et al., 2022), which leverages DAOS as a fast cache option for HPC clusters during fetching procedures by the traditional file backend. A generic layer could, then, retain several "bucket" namespaces open, submitting write requests to all at once; during analyses, different backends could be queried for data in parallel, avoiding network delays and bottlenecks for more reliable fetching.

The aforementioned building blocks form a blueprint for a connector layer between the logical ntuples and their practical counterparts responsible for interoperability with remote object stores. As exemplified, this generic object store layer could take up a majority of the duties and strategies for efficient throughput, eliminating redundant work when expanding this work to the myriad alternative object stores not yet explored for RNTuple.

The proposed functionality could enable RNTuple to exploit, with minimal effort, virtually every major existing cloud facility, as well as an expanding number of object-store-based HPC clusters, in service of the HEP community and LHC research.

6 CONCLUSION AND OUTLOOK

The higher luminosity from the HL-LHC is projected to increase the event data generation by 10–20×before the 2030s, compared to current runs at the LHC. Tools in HEP data analysis and storage must be adequated for this influx of data in order for the HEP field to benefit from this feat of engineering. In order to do so, it becomes critical to leverage modern storage technologies, e.g., NVMe devices, persistent memory and object stores, toward fast and scalable storage and I/O, enabling efficient and high–throughput analyses for the next generation of LHC experiments at CERN.

In this work, we proposed two backends that integrate object stores into RNTuple, ROOT's new I/O subsystem for the next generation of HEP analyses ushered by the HL-LHC. Each backend takes up an approach fit for its own use case in HEP analysis: DAOS, Intel's open–source object store, leveraging the power of HPC supercomputers and clusters for intensive analyses; and AWS S3, the dominant interface for cloud storage, as a transient stage meant to exploit existing cloud infrastructure and distribute HL-LHC data to thousands of researchers worldwide.

DAOS has risen to prominence in the last half decade as a high–performing object store for HPC. Our goal with RNTuple's DAOS backend, which predated this work in an experimental capacity, was to rework it to implement accepted techniques in the field, as seen in section 4.3. These techniques include request coalescing, supported by vector reads and writes, as well as more efficient scheduling queue management and reduced use of system calls, all of which enabled the proposal of a new data mapping between RNTuple and DAOS driven by physical target co-locality, $\phi_{co-locality}$ (defined in equation 4.2).

As the evaluation in section 5.3 shows, our implementation significantly improved I/O performance in comparison to the existing, proof–of–concept baseline. The particular reason for the improvement is unveiled in our "feature ladder" evaluation, where each partial version of the framework is benchmarked to identify the most impactful changes. For reading, the reduced use of system calls by keeping a persistent endpoint queue is necessary and sufficient for scalable fetches over RDMA. For writing, we found the opposite: only after all features are factored in, including the new data mapping, does write speed achieve high throughput. This finding unlocks a great potential for RNTuple to defer its writes by coalescing requests while,

in parallel, engaging in compression of the next batch of pages, as these are the two biggest sources of CPU utilization according to our performance analysis in section 5.3. Given the above, we consider that this work has met the **Objective 1a** set out in section 1.1.

A subsequent evaluation of the RNTuple and DAOS parameter space revealed useful information for future deployment in HPC clusters. Regarding RNTuple parameters, the weight of the page size on scalable throughput is confirmed, while the evaluation suggests that bigger RNTuple clusters favor writing speed due to larger request volumes. In spite of that, the impact of compression was negligible on the LHCb analysis and B2HHH dataset; results might differ for different datasets. Though our setup was too simplistic to extract insights on distributed analyses, we found native and caged pages to peak at around 10 GB/s for writing and 4.2 GB/s for reading. As expected, replication often caused decreases in write speed, particularly for large blobs at MiB scales. However, we cannot claim with confidence that higher replication always leads to lower write throughput and higher readbacks, or that increased sharding has benefits on performance. Given this, we estimate that **Objective 1b** was only partially satisfied; a systematic evaluation in a distributed setting, with different datasets and on a larger HPC cluster is necessary to understand the impact of the above parameters in realistic scenarios.

In light of the results, we proposed a concatenation feature ("caging") to allow ntuples with smaller data blobs to be spliced together server–side, without any additional copying. While this scatter–gather approach provided faster reading of the concatenated blobs, it did not attain the targeted throughput for writing, though an increase is observed for natively smaller pages. After further investigation, we observed a significant CPU effort tied to memory registration in our RDMA–enabled experimental setup for DAOS. Ultimately, this suggests RNTuple's sink mechanism is not optimally reusing IOV buffers. For sinking processes with many small blobs, such as when caging is toggled, the cost of memory registration is more significant. We deem **Objective 1c** reached by this method, showing promising results for the use cases that most need it and emancipating the sink layer from native parameters that are detrimental to network transfer: for data ingestion, we measured up to 2–3×better throughput when comparing 32–64 kB pages being spliced together as 1 MiB cages, whereas reading is improved 7–4×for the same parameters. As native pages grow larger, the gains in transfer rate become negligible, suggesting this

mechanism to be particularly beneficial for transfers from RNTuple's file backend, which sees default page sizes of 64 kB, to DAOS.

In order to benefit from the existing cloud infrastructure, our cloud backend approach targets the S3 service and API, which has become the unofficial standard for cloud interfacing among cloud providers. Our proposal is a proof-of-concept based on the foundations acquired with DAOS, seeking to understand the service's performance and limitatins. We began by identifying the characteristic differences and similarities between the two models (Table 4.1), such as S3's more opaque mapping interface and higher latency compared to DAOS, which we were able to counteract by increasing the transfer buffer sizes on the integration's data mapping, satisfying **Objective 2** in section 1.1.

Specifically, we were able to demonstrate that page sizes associated with a performant S3 backend should be bigger than those of the DAOS counterpart by a significant factor (4–16×). This goes hand–in–hand with our service cost estimates for ingestion and analysis, wherein larger pages directly lead to fewer requests and cost savings of up to 6-fold for writing and 30 % for reading. Note that under our WORM access pattern, the latter metric should be prioritized, and its savings are quickly the dominating factor after two dozen analyses. Given the results, we formulated two additional approaches for the S3 backend, whose implementation depends on the underlying provider's support for byte–range requests.

Through this work, as envisaged in the introductory section 1.1, RNTuple made headway into first–class support for object stores, with a production–grade DAOS backend that efficiently populates HPC data centers and a clear path to a performant S3 backend for fast HEP data distribution throughout the cloud.

Contributions

The object of this thesis has generated multiple contributions in HEP–related scientific events with dedicated computing sessions. This section lists appearances of this work in HEP conferences and a workshop organized by CERN openlab (CERN, 2023). [1]

In HEP conferences, authors customarily present their works before an arti-

[1] A public-private initiative coordinated at CERN to accelerate the development of computing technologies that present practical benefits to HEP research and test them in real-world scenarios.

cle is produced, much less peer reviewed. Such an approach stimulates discussion in a field guided by deductive reasoning and whose experiments typically require significant funding and time to conduct. The conferences that showcased our work have followed this formula. Thus, we include submissions to their corresponding proceedings, though their peer review processes are still ongoing or have not begun at the present time of writing.

ACAT 2022 (poster): *RNTuple: Towards First-Class Support for HPC data centers* (MIOTTO; LOPEZ-GOMEZ, 2022).

CERN openlab Workshop (talk): *Mapping ROOT RNTuple I/O data structures to DAOS objects* (LOPEZ-GOMEZ; MIOTTO, 2023).

CHEP 2023 (talk): *Storing LHC Data in DAOS and S3 through RNTuple* (MIOTTO; LOPEZ-GOMEZ, 2023).

There are two proposed publications about this work. The first is pending review; the second is being submitted in September 2023:

ACAT 2022 Proceedings: MIOTTO, G. L.; LOPEZ-GOMEZ, J. *RNTuple: Towards First-Class Support for HPC data centers*. 2023. Pending peer review.

CHEP 2023 Proceedings: MIOTTO, G. L.; LOPEZ-GOMEZ, J.; GEYER, C. R. *RNTuple: Efficient HEP Data I/O for Object Stores*. 2023. Submission imminent.

Outlook

The findings in this work have and will continue to guide object store support strategies for RNTuple. In this section, we list some of the directions for the backends, going forward.

This work has identified an important bottleneck in memory registration overhead costs over RDMA interconnects, as mentioned in 5.3.3. Approaches for the reuse of allocated transfer buffers in RNTuple are being considered.

The evaluation of the DAOS backend in a distributed setting was kept out of this thesis' scope; in exploratory experiments, scalability constraints prevented the link layer saturation during multi–node analysis over RDMA. In the future, this should be revisited in a larger HPC cluster, after RNTuple leaves the experimental

stage.

In its current state, the AWS S3 backend is a proof–of–concept. Its path toward production goes through the implementation of byte–range read request support, so that TCP/IP latency can be circumvented in the sink by coalescing page groups or clusters into single blobs, as described in section 5.4.

Going further, the AWS S3 backend can springboard compatibility with other object storage providers sharing similar APIs and feature sets, e.g., Microsoft Azure, Google Cloud Platform, IBM Cloud and Oracle Cloud. More generally, this hints at the implementation of a provider–agnostic connector between RNTuple's storage layer and a plethora of concrete cloud backends. The insights of this work, which identify common structures, request patterns and mappings between the two proposed backends, can be used to inform that endeavor.

REFERENCES

AAD, G. et al. Observation of a new particle in the search for the standard model higgs boson with the ATLAS detector at the LHC. **Physics Letters B**, Elsevier BV, v. 716, n. 1, p. 1–29, sep 2012. Available from Internet: <https://doi.org/10.1016%2Fj.physletb.2012.08.020>.

ABADI, D. J.; BONCZ, P. A.; HARIZOPOULOS, S. Column-Oriented Database Systems. **Proc. VLDB Endow.**, VLDB Endowment, v. 2, n. 2, p. 1664–1665, aug 2009. ISSN 2150-8097. Available from Internet: <https://doi.org/10.14778/1687553.1687625>.

ABADI, D. J.; MADDEN, S. R.; HACHEM, N. Column-stores vs. row-stores: How different are they really? In: **Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data**. New York, NY, USA: Association for Computing Machinery, 2008. (SIGMOD '08), p. 967–980. ISBN 9781605581026. Available from Internet: <https://doi.org/10.1145/1376616.1376712>.

AGOSTINELLI, S. et al. Geant4—a simulation toolkit. **Nuclear instruments and methods in physics research section A: Accelerators, Spectrometers, Detectors and Associated Equipment**, Elsevier, v. 506, n. 3, p. 250–303, 2003.

Amazon, Inc. **Best practices design patterns: optimizing Amazon S3 performance**. Amazon, Inc., 2023. Accessed: 2023-07-24. Available from Internet: <https://docs.aws.amazon.com/AmazonS3/latest/userguide/optimizing-performance.html>.

Amazon Web Services. **Amazon S3 Pricing**. Amazon Web Services, Inc., 2023. Accessed: 2023-05-03. Available from Internet: <https://aws.amazon.com/s3/pricing/>.

ANTCHEVA, I. et al. ROOT - A C++ framework for petabyte data storage, statistical analysis and visualization. **Comput. Phys. Commun.**, v. 180, p. 2499–2512, 2009.

ARDINO, R. et al. A 40 mhz level-1 trigger scouting system for the cms phase-2 upgrade. **Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment**, v. 1047, p. 167805, 2023. ISSN 0168-9002. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S016890022201097X>.

Argonne National Laboratory. **Aurora Exascale Supercomputer**. 2023. Accessed: 2023-07-27. Available from Internet: <{https://www.anl.gov/auror}.>

ATLAS Collaboration. **ATLAS Software and Computing HL-LHC Roadmap**. Geneva, 2022. Available from Internet: <https://cds.cern.ch/record/2802918>.

AYLLON, A. A. et al. Making the most of cloud storage - a toolkit for exploitation by wlcg experiments. **Journal of Physics: Conference Series**, v. 898, 2017. Available from Internet: <https://api.semanticscholar.org/CorpusID:67176643>.

BERGHöFER, T. et al. **Towards a Model for Computing in European Astroparticle Physics**. 2015.

BLOMER, J. A quantitative review of data formats for hep analyses. **Journal of Physics: Conference Series**, IOP Publishing, v. 1085, n. 3, p. 032020, sep 2018. Available from Internet: <https://dx.doi.org/10.1088/1742-6596/1085/3/032020>.

BLOMER, J. et al. Evolution of the ROOT Tree I/O. **ArXiv**, abs/2003.07669, 2020.

BLOMER, J. et al. **ROOT RNTuple Virtual Probe Station**. 2022. Accessed: 2023-06-23. Available from Internet: <https://github.com/jblomer/iotools/tree/acat22>.

BOCKELMAN, B.; ELMER, P.; WATTS, G. **IRIS-HEP Strategic Plan for the Next Phase of Software Upgrades for HL-LHC Physics**. 2023.

BOITO, F. Z. et al. A Checkpoint of Research on Parallel I/O for High-Performance Computing. **ACM Computing Surveys**, v. 51, p. 1–35, 03 2018.

BRAAM, P. **The Lustre Storage Architecture**. 2019.

BRUN, R.; RADEMAKERS, F. ROOT—An object oriented data analysis framework. **Nuclear instruments and methods in physics research section A: accelerators, spectrometers, detectors and associated equipment**, Elsevier, v. 389, n. 1-2, p. 81–86, 1997.

CERN. **CERN openlab**. 2023. Accessed: 2023-07-12. Available from Internet: <https://openlab.cern>.

CERN. **Storage: what data to record?** CERN, 2023. Accessed: 2023-05-22. Available from Internet: <https://home.cern/science/computing/storage>.

CHIEN, S. W. der et al. Exploring scientific application performance using large scale object storage. In: **Lecture Notes in Computer Science**. Springer International Publishing, 2018. p. 117–130. Available from Internet: <https://doi.org/10.1007%2F978-3-030-02465-9_8>.

CLISSA, L. Survey of Big Data sizes in 2021. **arXiv**, arXiv, 2022.

CMS Offline Software and Computing. **CMS Phase-2 Computing Model: Update Document**. Geneva, 2022. Available from Internet: <https://cds.cern.ch/record/2815292>.

DAOS Project. **DAOS Overview**. 2023. Accessed: 2023-04-25. Available from Internet: <https://docs.daos.io/v2.2/overview/architecture/>.

DAOS Project. **DAOS Overview: Storage Model**. 2023. Accessed: 2023-04-25. Available from Internet: <{https://docs.daos.io/v2.2/overview/storag}.>

DAOS Project. **daos-stack/daos: DAOS Storage Stack (client libraries, storage engine, control pane)**. Github, 2023. Accessed: 2023-04-25. Available from Internet: <https://github.com/daos-stack/daos>.

DAOS Project. **Hardware Requirements**. 2023. Accessed: 2023-04-25. Available from Internet: <{https://docs.daos.io/v2.2/admin/hardware}.>

DEVRESSE, A.; FURANO, F. **Efficient HTTP based I/O on very large datasets for high performance computing with the libdavix library**. 2014.

DORIGO, A. et al. Xrootd - a highly scalable architecture for data access. **WSEAS Transactions on Computers**, v. 4, p. 348–353, 04 2005.

DUWE, K.; KUHN, M. Using ceph's bluestore as object storage in hpc storage framework. **Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems**, 2021. Available from Internet: <https://api.semanticscholar.org/CorpusID:233384677>.

FOSTER, I. T.; KESSELMAN, C.; TUECKE, S. The Anatomy of the Grid - Enabling Scalable Virtual Organizations. **CoRR**, cs.AR/0103025, 2001. Available from Internet: <{https://arxiv.org/abs/cs/010302}.>

FREY, P. W.; ALONSO, G. Minimizing the hidden cost of rdma. In: IEEE. **2009 29th IEEE International Conference on Distributed Computing Systems**. [S.l.], 2009. p. 553–560.

FTS. **FTS Website**. 2023. Accessed: 2023-08-12. Available from Internet: <{https://fts.web.cern.ch/fts}.>

GADBAN, F.; KUNKEL, J. Analyzing the Performance of the S3 Object Storage API for HPC Workloads. **Applied Sciences**, v. 11, n. 18, 2021. ISSN 2076-3417. Available from Internet: <https://www.mdpi.com/2076-3417/11/18/8540>.

GIANNUZZI, G. et al. Analysis of high-identity segmental duplications in the grapevine genome. **BMC Genomics**, v. 12, p. 436 – 436, 2011.

GIBNEY, E. How the revamped Large Hadron Collider will hunt for new physics. Nature Research, v. 605, n. 7911, p. 604–607, May 2022. Available from Internet: <{https://www.nature.com/articles/d41586-022-01388-}.>

GREVILLOT, L. et al. A monte carlo pencil beam scanning model for proton treatment plan simulation using gate/geant4. **Physics in Medicine & Biology**, v. 56, p. 5203 – 5219, 2011.

HADY, F. T. et al. Platform storage performance with 3d xpoint technology. **Proceedings of the IEEE**, v. 105, n. 9, p. 1822–1833, Sep. 2017. ISSN 1558-2256.

HARTMANN, N.; ELMSHEUSER, J.; DUCKECK, G. Columnar data analysis with atlas analysis formats. In: EDP SCIENCES. **EPJ Web of Conferences**. [S.l.], 2021. v. 251, p. 03001.

HENNECKE, M. Understanding daos storage performance scalability. In: **Proceedings of the HPC Asia 2023 Workshops**. New York, NY, USA: Association for Computing Machinery, 2023. (HPC Asia '23 Workshops), p. 1–14. ISBN 9781450399890. Available from Internet: <https://doi.org/10.1145/3581576.3581577>.

High Luminosity LHC Project. **LS3 schedule change**. CERN, 2022. Accessed: 2023-03-07. Available from Internet: <https://hilumilhc.web.cern.ch/article/ls3-schedule-change>.

InfiniBand Trade Association. **InfiniBand Roadmap**. 2023. Accessed: 2023-03-05. Available from Internet: <{https://www.infinibandta.org/infiniband-roadmap}.>

Intel Corporation. **Intel® oneAPI Threading Building Blocks**. 2023. Accessed: 2023-08-12. Available from Internet: <https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html>.

IO500 Foundation. **IO500 SC22 List**. 2022. Accessed: 2023-03-17. Available from Internet: <https://io500.org/list/sc22/io500>.

JEONG, K. et al. Optimizing the ceph distributed file system for high performance computing. In: **2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)**. [S.l.: s.n.], 2019. p. 446–451.

KHACHATRYAN, V. et al. Search for the associated production of the higgs boson with a top-quark pair. **Journal of High Energy Physics**, v. 2014, p. 1–64, 2014.

KOGGE, P. et al. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. **Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Techinal Representative**, v. 15, 01 2008.

LHCb collaboration (2017). **Matter Antimatter Differences (B meson decays to three hadrons) - Data Files**. CERN Open Data Portal, 2017. Available from Internet: <http://opendata.cern.ch/record/4900>.

LIANG, Z. et al. DAOS: A Scale-Out High Performance Storage Stack for Storage Class Memory. In: _____. [S.l.: s.n.], 2020. p. 40–54. ISBN 978-3-030-48841-3.

LIU, J. et al. Evaluation of HPC Application I/O on Object Storage Systems. In: **2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage 'I&' Data Intensive Scalable Computing Systems (PDSW-DISCS)**. [S.l.: s.n.], 2018. p. 24–34.

LIU, J. et al. Evaluation of hpc application i/o on object storage systems. **2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)**, p. 24–34, 2018. Available from Internet: <https://api.semanticscholar.org/CorpusID:59453124>.

LOGAN, L. et al. An evaluation of daos for simulation and deep learning hpc workloads. **Proceedings of the 3rd Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems**, 2023. Available from Internet: <https://api.semanticscholar.org/CorpusID:258486751>.

LOPEZ-GOMEZ, J.; BLOMER, J. Exploring object stores for high-energy physics data storage. **EPJ Web Conf.**, v. 251, p. 02066, 2021. Available from Internet: <https://doi.org/10.1051/epjconf/202125102066>.

LOPEZ-GOMEZ, J.; BLOMER, J. RNTuple performance: Status and Outlook. arXiv, 2022. Available from Internet: <{https://arxiv.org/abs/2204.0904}.>

LOPEZ-GOMEZ, J.; MIOTTO, G. L. **Mapping ROOT RNTuple I/O data structures to DAOS objects**. 2023. Accessed: 2023-07-21. Available from Internet: <{https://indico.cern.ch/event/1225408/contributions/524384}.>

LüTTGAU, J. et al. Survey of storage systems for high-performance computing. **Supercomputing Frontiers and Innovations**, v. 5, n. 1, p. 31–58, Apr. 2018. Available from Internet: <https://superfri.org/index.php/superfri/article/view/162>.

MANUBENS, N. et al. Daos as hpc storage: a view from numerical weather prediction. In: **2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)**. [S.l.: s.n.], 2023. p. 1029–1040.

MANUBENS, N. et al. Performance comparison of daos and lustre for object data storage approaches. **2022 IEEE/ACM International Parallel Data Systems Workshop (PDSW)**, p. 7–12, 2022. Available from Internet: <https://api.semanticscholar.org/CorpusID:253581225>.

MIETKE, F. et al. Analysis of the memory registration process in the mellanox infiniband software stack. In: . [S.l.: s.n.], 2006. v. 4128, p. 124–133. ISBN 978-3-540-37783-2.

MinIO, Inc. **MinIO**. MinIO, Inc., 2023. Accessed: 2023-05-03. Available from Internet: <https://min.io/docs/minio/linux/index.html>.

MIOTTO, G. L.; LOPEZ-GOMEZ, J. **RNTuple: Towards First-Class Support for HPC data centers**. 2022. Accessed: 2023-07-21. Available from Internet: <{https://indico.cern.ch/event/1106990/contributions/499135}.>

MIOTTO, G. L.; LOPEZ-GOMEZ, J. **Storing LHC Data in DAOS and S3 through RNTuple**. 2023. Accessed: 2023-07-21. Available from Internet: <{https://indico.jlab.org/event/459/contributions/1132}.>

MUñOZ-ESCOí, F. D. et al. CAP Theorem: Revision of Its Related Consistency Models. **The Computer Journal**, v. 62, n. 6, p. 943–960, 03 2019. ISSN 0010-4620. Available from Internet: <https://doi.org/10.1093/comjnl/bxy142>.

NAUMANN, A. et al. **ROOT for the HL-LHC: data format**. arXiv, 2022. Available from Internet: <https://arxiv.org/abs/2204.04557>.

NIELSEN, H. et al. **Hypertext Transfer Protocol – HTTP/1.1**. RFC Editor, 1999. RFC 2616. (Request for Comments, 2616). Available from Internet: <https://www.rfc-editor.org/info/rfc2616>.

Nvidia, Inc. **Benefits of Remote Direct Memory Access Over Routed Fabrics**. Nvidia, Inc., 2023. Accessed: 2023-06-05. Available from Internet: <https://network.nvidia.com/pdf/solutions/benefits-of-RDMA-over-routed-fabrics.pdf>.

OFIWG. **Libfabric**. OFIWG, 2023. Accessed: 2023-07-01. Available from Internet: <https://ofiwg.github.io/libfabric/>.

PADULANO, V. E. **Distributed Computing Solutions for High Energy Physics Interactive Data Analysis**. Thesis (PhD) — Valencia, Polytechnic U., 2023.

PADULANO, V. E. et al. Leveraging state-of-the-art engines for large-scale data analysis in high energy physics. **Journal of Grid Computing**, v. 21, p. 1–21, 2023. Available from Internet: <https://api.semanticscholar.org/CorpusID:256702277>.

PADULANO, V. E. et al. A caching mechanism to exploit object store speed in high energy physics analysis. **Cluster Computing**, p. 1–16, 10 2022.

PADULANO, V. E. et al. Distributed data analysis with root rdataframe. **EPJ Web Conf.**, v. 245, p. 03009, 2020. Available from Internet: <https://doi.org/10.1051/epjconf/202024503009>.

PERUZZI, M. et al. The nanoaod event data format in cms. **Journal of Physics: Conference Series**, IOP Publishing, v. 1525, n. 1, p. 012038, apr 2020. Available from Internet: <https://dx.doi.org/10.1088/1742-6596/1525/1/012038>.

ROOT Project. **ROOT - Analyzing petabytes of data, scientifically**. CERN, 2023. Accessed: 2023-03-29. Available from Internet: <https://root.cern.ch>.

ROOT Project. **root-project/root: the official repository for ROOT. Master branch.** Github, 2023. Available from Internet: <https://github.com/root-project/root/tree/master>.

SEHRISH, S.; KOWALKOWSKI, J.; PATERNO, M. F. Spark and hpc for high energy physics data analyses. **2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)**, p. 1048–1057, 2017. Available from Internet: <https://api.semanticscholar.org/CorpusID:2357766>.

SOUMAGNE, J. et al. Accelerating HDF5 I/O for Exascale Using DAOS. **IEEE Transactions on Parallel and Distributed Systems**, v. 33, n. 4, p. 903–914, 2022.

The Apache Software Foundation. **Apache Arrow**. 2023. Accessed: 2023-08-20. Available from Internet: <{https://arrow.apache.or}.>

The Apache Software Foundation. **Apache Parquet**. 2023. Accessed: 2023-08-20. Available from Internet: <{https://parquet.apache.org}.>

THOMASIAN, A. Chapter 2 - storage technologies and their data. In: THOMASIAN, A. (Ed.). **Storage Systems**. Morgan Kaufmann, 2022. p. 89–196. ISBN 978-0-323-90796-5. Available from Internet: <https://www.sciencedirect.com/science/article/pii/B9780323907965000115>.

WEIL, S. A. et al. Rados: A scalable, reliable storage service for petabyte-scale storage clusters. In: **Proceedings of the 2nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07**. New York, NY, USA: Association for Computing Machinery, 2007. (PDSW '07), p. 35–44. ISBN 9781595938992. Available from Internet: <https://doi.org/10.1145/1374596.1374606>.

WLCG. **Worldwide LHC Computing Grid**. CERN, 2023. Accessed: 2023-07-22. Available from Internet: <{https://wlcg-public.web.cern.ch}.>

GLOSSARY

`akey` Attribute key. In DAOS, it is a 64-bit value that complements the `dkey` to comprise the blob key within the DAOS object using the KVS interface. 27, 28, 51, 52, 55, 56

`dkey` Distribution key. In DAOS, it is a 64-bit value that complements the `akey` to comprise the blob key within the DAOS object using the KVS interface. This key impacts target co-locality; within the same container and object, two blobs under the same `dkey` are guaranteed to be stored in the same target nodes on the DAOS server. 27, 28, 51, 52, 55

`oid` Object ID. In DAOS, it is a 128-bit value mapping to an object within the DAOS container. The first 32 are reserved to DAOS, e.g., to encode object class. The remaining 96 bits are freely describable by the user. 28, 51, 52, 55

blob Binary Large Object, a usually immutable entity stored as raw, binary data. 28, 51, 55–58, 60, 69, 77, 86, 93

bucket Private namespace in Cloud-based object stores, such as AWS S3, where S3 objects are allocated. 85, 89

cage Portmanteau for *concatenated page*. The result of RNTuple's *caging* in its DAOS backend. 55–57, 69, 70, 77, 79, 85, 90

caging RNTuple's scatter-gather concatenation mechanism, which combines neighboring pages from the same page group into a single blob server-side. 56, 57, 85, 90

cluster A horizontal split of an RNTuple dataset, roughly $\mathcal{O}(100)$ MB in size. RNTuple's unit of writing. 17, 42, 44, 46, 49, 52, 55, 60, 70–72, 74–76, 78, 83, 85, 86, 90

cluster bunch A typically small (1–5) range of consecutive clusters in RNTuple fetched together for parallel decompression in multi-threaded contexts. RNTuple's unit of reading. 46, 49, 61, 90

column A subset of fundamentally-typed data in RNTuple; internal component of the external-facing field. Multiple columns can be part of a field, e.g. vectors, which have an offset column and one or more columns for the comprising data. 45, 46, 60, 68

container Private namespace within pools in DAOS, where DAOS objects are al-

located. 49, 89

event The set of particle collisions resulting from a proton bunch crossing. 19, 20, 23, 53–55, 57, 69, 82

exascale Term associated with a selective but growing group of the best performing HPC clusters and supercomputers, able to achieve at least 10^18 IEEE754 64-bit operations per second. 16, 17, 24, 36, 37

IOV Basic unit for scatter-gather I/O. It denotes a structure with a memory buffer address and its corresponding length, in bytes. 28, 52, 56, 58, 60, 77, 79, 90, 93

ntuple A ROOT dataset 18, 23, 42–44, 46, 48, 51, 55, 57, 68, 74, 77, 86, 87, 89–91, 93

object In DAOS, entities with the capacity to store multiple data chunks according to a given data model, which can be array-like or akin to a key-value store. 85

page A partition of column data in RNTuple, roughly $\mathcal{O}(100)$ kB in size. It is the unit of compression and smallest scale at which I/O is performed. 17, 43–46, 49, 51, 52, 55–57, 60, 66, 70, 74, 77, 79, 82, 83, 85–88, 90, 93, 94

page group Consists of all pages sharing a given cluster and column in RNTuple. Can be described as the predominant unit of analysis. 43, 51, 52, 55, 57, 60, 85, 90

POSIX I/O Standard in the POSIX family 25, 30, 34, 37, 39–41, 73

proton bunch A batch of protons launched together in the same direction at the Large Hadron Collider. 19, 20

Request For Comments A document by the Internet Engineering Task Force (IETF) containing specifications, standards or technical memoranda on Internet-related topics. 86

SGL Scatter-Gather Lists, a flat collection of IOVs used for scatter-gather I/O. 28, 52, 56

shard A partition of a database integral to sharding, a technique for load balancing that spreads different logical data across available server targets to reduce contention for physical resources. 26

TCP/IP Transmission Control Protocol/Internet Protocol, a set of communication protocols between the nodes participating in the Internet. 59, 96

TTree ROOT's long-established row and columnar data format and I/O subsytem for HEP analysis. The predecessor to ROOT RNTuple. 14, 16, 22, 23, 38, 42, 43, 46

vector write Data transfer in bulk through data vectors that concentrate references to multiple content buffers for different objects. 44, 45