A Study of RAuxStore

Analyzing the Performance of RAuxStore
Through Benchmarking

Mads Lildholdt Hansen,
Nikolaj Kofod Krogh

Master's Thesis

AALBORG UNIVERSITY

STUDENT REPORT

Department of Computer Science
Software
Selma Lagerløfs Vej 300
9220 Aalborg Ø
www.aau.dk

Title:
A Study of RAuxStore

Project Period:
Spring Semester 2025

Project Group:
cs-25-ds-10-01

Participants:
Mads Lildholdt Hansen
Nikolaj Kofod Krogh

Supervisor:
Josva Kleist

Number of Pages: 51

Date of Completion:
May 15, 2025

Abstract:

This thesis uses a synthetic benchmark to analyze the performance of the `RAuxStore`, a new auxiliary store developed to replace the `TAuxStore` in the ATLAS experiment at CERN. The `RAuxStore` is the RNTuple implementation of an auxiliary store, and the RNTuple data format is designed to improve I/O performance and reduce storage size compared to TTree, the format used by the existing `TAuxStore`. Initial benchmarking revealed surprising results, with the `RAuxStore` being outperformed by the `TAuxStore`. This thesis investigates the performance bottlenecks of the `RAuxStore` with a benchmark designed according to principles of well-defined benchmarking. We successfully identified bottlenecks in the `RAuxStore`; however, due to ongoing updates to ATLAS' software, we cannot definitively conclude whether these issues still exist in the current software.

Preface

Mads Lildholdt Hansen
mlha20@student.aau.dk

Nikolaj Kofod Krogh
nkrogh20@student.aau.dk

Aalborg University, May 15, 2025

Summary

Dansk

CERN, den europæiske organisation for nuklearforskning, undersøger universets fundamentale partikler. The Large Hadron Collider (LHC), verdens største og mest kraftfulde partikelaccelerator, ligger på CERN. Ved LHC er der fire eksperimenter, der indsamler data: ATLAS, CMS, LHCb og ALICE.
I vores tidligere projekt arbejdede vi på ATLAS-eksperimentet. Ligesom de andre LHC-eksperimenter gemmer ATLAS data i et format kaldet TTree, udviklet af ROOT som er et dataanalyse framework til højenergifysik. TTree formatet blev udviklet i 1990'erne og er ikke optimeret til moderne hardware. Derfor har ROOT udviklet et nyt dataformat kaldet RNTuple, som er designet til at være mere effektivt end TTree. RNTuple forventes at være klar til produktionsbrug i ATLAS-eksperimentet i Run 4, der forventes at starte i 2030.
I denne rapport bygger vi videre på vores tidligere projekt med titlen Further Work on RNTuples in ATLAS, hvor vi udviklede `RAuxStore` som er RNtuple implementeringen af TTree's auxiliary store, `TAuxStore`. En auxiliary store er en datastruktur til at gemme auxiliary data, som er data der ikke er en del af hoveddatastrukturen, men som kan tilføjes dynamisk, hvis det er nødvendigt for en analyse. Gennem eksperimenter med `RAuxStore` i vores tidligere arbejde opdagede vi, at dens perfomance ikke levede op til forventningerne sammenlignet med `TAuxStore`.
Vores primære fokus i denne rapport har derfor været at undersøge årsagen til disse perfomance problemer i `RAuxStore` ved at gennemføre en række eksperimenter. Disse eksperimenter er udført med et benchmark, som oprindeligt blev skabt i vores tidligere projekt. I denne rapport har vi forbedret den eksisterende benchmark baseret på benchmark teori og på, hvordan man skaber et benchmark, der er både interpretable og reproducible. En benchmark er interpretable, hvis det giver tilstrækkelig information, så andre forskere kan forstå det og drage deres egne konklusioner. Det er reproducible, hvis andre forskere kan gentage resultaterne. Vores benchmark kan klassificeres som syntetisk, og vi vurderer, at den opfylder kravene til at være både interpretable og reproducible.
Med denne benchmark har vi kunnet identificere performance bottlenecks i `RAuxStore` og har løst én af dem. Vi har også lavet et kernel-benchmark for en specifik del af auxiliary stores for at undersøge den tid, det tager at indlæse en entry for en bestemt branch eller et bestemt field. Her fandt vi, at indlæsningstiden for nogle entries var op til en faktor 1000 langsommere end medianen.

Alle vores eksperimenter blev udført på et SSD, men for at teste på et andet lagermedium gennemførte vi dem også på en HDD. Ved dette fandt vi, at performance var en smule bedre på SSD end på HDD. Vi konkluderer dog, at lagermediet ikke har en meningsfuld indvirkning på resultaterne.
Vi har identificeret problemer med performance i `RAuxStore`, og de relaterer sig hovedsageligt til specifikke entries i eventdata. Vi har ikke løst alle problemer, blandt andet fordi vi ikke ved, om de stadig eksisterer. ATLAS' software er omfattende, har mange aktive bidragsydere, og koden ændres konstant. For at teste den version af `RAuxStore`, vi arbejdede med, skabte vi et snapshot af koden på et bestemt tidspunkt. Siden da er der foretaget store ændringer i auxiliary stores, og nogle af de problemer, vi opdagede, kan allerede være blevet løst.

English

CERN, the European Organization for Nuclear Research, is exploring the fundamental particles of the universe. The Large Hadron Collider (LHC), the world's largest and most powerful particle accelerator, is located at CERN. At the LHC, there are four experiments collecting data: ATLAS, CMS, LHCb, and ALICE.
In our previous project, we worked on the ATLAS experiment. Along with the other LHC experiments, ATLAS stores data in a format called TTree, which was developed by ROOT, a data analysis framework for high-energy physics. The TTree data format was developed in the 1990s and is not optimized for modern hardware. Therefore, ROOT has developed a new data format called RNTuple, which is designed to be more efficient than TTree. The RNTuple format is expected to be ready for production use in the ATLAS experiment by Run 4, which is scheduled to start in 2030.
In this thesis, we build upon our earlier project titled *Further Work on RNTuples in ATLAS*, in which we developed the `RAuxStore`, the RNTuple-based implementation of TTree's auxiliary store, `TAuxStore`. An auxiliary store is a data structure used to store auxiliary data, which is data that is not part of the main data structure but can be added dynamically, if needed for analysis. Through experimenting on the `RAuxStore` in our previous work, we discovered that it did not perform as expected in comparison to the `TAuxStore`.
Our main focus for this thesis has thus been to investigate the reason for these performance issues of the `RAuxStore` by conducting a series of experiments. These experiments have been conducted using a benchmark initially made in our previous project. In this thesis, we have improved upon the existing benchmark based on the theory of benchmarking and on how to create a benchmark that is interpretable and reproducible. A benchmark is interpretable if it provides enough information for other researchers to understand it and draw their own conclusions. It is reproducible if the results can be replicated by other

researchers. We can classify our benchmark as a synthetic benchmark, and we deem that it meets the requirements of being both interpretable and reproducible.
With this benchmark, we have been able to identify performance bottlenecks in the `RAuxStore` and have solved one of them. We have also created a kernel benchmark for a specific part of the auxiliary stores to investigate the time it takes to load an entry for a specific branch or field. Here, we found that the time taken for some entries was up to a factor of 1000 slower than the median time.
Our experiments were all performed on an SSD, but to test on a different storage medium, we also performed them on an HDD. By doing this, we found that the performance was slightly better on an SSD than on an HDD. However, we conclude that the storage medium does not have a meaningful impact on the results.
We have identified problems regarding the performance of the `RAuxStore`, and they are mainly related to specific entries in the event data. We have not solved all problems, in part because we do not know if they are still problems. ATLAS' software is huge, has many active contributors, and the code is constantly changing. To test the version of the `RAuxStore` we worked on, we created a snapshot of the code at a specific point in time. Since then, major changes have been made to the auxiliary stores, and some of the problems we discovered might already have been dealt with.

Contents

1 Introduction **1**

2 Problem Definition **6**

3 Related Work **7**
3.1 Comparing the Performance of the RNTuple to the TTree 7
3.2 Benchmarking of Computer Systems 7

4 Background **10**
4.1 TTree 10
4.2 RNTuple 11
4.3 Auxiliary Store 11
4.4 Benchmark 14
4.5 Performance Measurement Techniques 15

5 Benchmark Analysis **17**
5.1 Analyzing the Benchmark Code 17
5.2 Applying the Benchmark Theory 21

6 Performance Measuring **23**
6.1 Analyzing Performance Bottlenecks 23
6.2 Why `RAuxStore` is Faster Without a Warmup Phase 25
6.3 First `RAuxStore` Run is Faster 27
6.4 SSD vs. HDD 28
6.5 Impact of Different Seeds 29
6.6 RNTuple Throughput 31

7 Benchmark Improvements **32**
7.1 Initial Improvements 32
7.2 Redesigned Benchmark 33

8 Discussion **37**

9 Conclusion **39**

10 Future Work **40**
10.1 Application Benchmark 40
10.2 Refactoring Our Benchmark 40

Bibliography **41**

A Performance Measuring **46**

B Benchmark Tables on HDD **47**

C Comparison Tables **49**

1 — Introduction

The Conseil Européen pour la Recherche Nucléaire (CERN), or the European Council for Nuclear Research, was founded in 1954. CERN is focused on studying the fundamental particles that make up the universe [1]. This is studied by accelerating particles to high speeds and colliding them into either a fixed target or other particles using a particle accelerator [2]. CERN built its first accelerator, the 600 MeV Synchrocyclotron (SC), in 1957, which accelerated particles by repeatedly applying a moderate voltage as a particle came around in its circular orbit [3, 4]. For more than 70 years, CERN has constructed six synchrotrons, a type of circular accelerator, for High Energy Physics (HEP) research: the Proton Synchrotron (PS), the PS Booster, the Intersecting Storage Rings (ISR), the Super Proton Synchrotron (SPS), the Large Electron-Positron Collider, and the Large Hadron Collider (LHC). Additionally, CERN has built five synchrotrons for particle beam accumulation and preparation: the Antiproton Accumulator (AA), the Antiproton Cooler, the Low Energy Ion Ring (LEIR), the Antiproton Decelerator (AD), and the Extra Low ENergy Antiproton ring (ELENA). Most of these synchrotrons remain in use today, as shown in *Figure 1.1* (p. 2) [5, 4].

The largest and most powerful accelerator in the world is CERN's LHC, which uses superconducting magnets to accelerate particles to almost the speed of light [1, 6]. It is located 100 m underground, has a circumference of 27 km, and hosts four main experiments: ATLAS, CMS, ALICE, and LHCb [7]. ATLAS and CMS are general-purpose detectors, ALICE focuses on heavy-ion physics, and LHCb specializes in the study of bottom quarks [1]. ATLAS and CMS are larger, with ATLAS being the world's largest particle detector [8]. The ATLAS codebase consists of at least 4 million lines of code, developed over 15 years by more than 3000 contributors from around the world [9, 10].

Each operational year, the LHC dedicates one month to heavy-ion collisions, with the remaining time focused on proton-proton collisions [11]. To fill the LHC with protons, CERN burns hydrogen gas to extract them [1]. The LHC relies on a sequence of smaller accelerators, including the Linear Accelerator (LINAC) 4, the PS, the PS Booster, and the SPS, to provide protons at 450 GeV, which are then injected into the LHC [4]. Since these accelerators are smaller than the LHC, the process is repeated multiple times until the LHC beam is filled with proton bunches [1]. A collection of protons is called a bunch, where each bunch contains approximately 10^{11} protons. The bunches are separated by 25 ns, which allows for around 40 million collisions per second. The beam contains multiple bunches of protons, and in the LHC the beam's capacity is around 2,800 proton bunches. Once the beam reaches a peak energy of around 7 TeV, the experiments begin recording

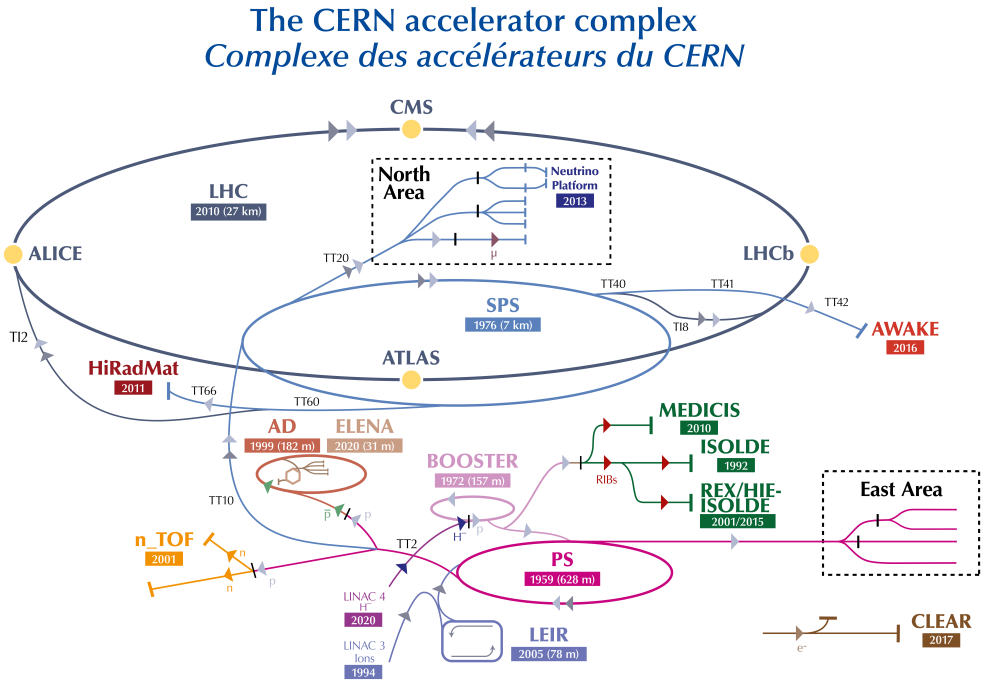


Figure 1.1: CERN's accelerator complex [5].

data, continuing until the beam has deteriorated [1].

The LHC has been operational since 2010 and has operated in phases, also known as runs. Between runs, there are long shutdown periods during which upgrades can be made to the LHC. Run 1 lasted from 2010 to 2013, Run 2 lasted from 2015 to 2018, and Run 3 began in 2022 and is expected to continue until 2026. Run 4, also known as the High Luminosity LHC (HL-LHC), is scheduled to begin in 2030 and last until 2033 [12, 13, 14].

Luminosity is a measure of a collider's performance, with higher luminosity corresponding to a greater number of collisions. It increases with higher beam intensity — that is the number of particles in the beam — which leads to higher collision frequency but can also be enhanced by squeezing the beam into a smaller cross-section, thereby raising the probability of collisions [1]. With the HL-LHC, the luminosity is anticipated to increase by a factor of 5, which means an even larger amount of data is expected to be generated by the LHC [1].

Each collision in the LHC is recorded as an event, which is stored and processed by the different experiments. However, with 40 million collisions occurring every second in the detectors, it is impossible to store the full volume of data produced. To address this challenge, a system known as the trigger has been developed, where both ATLAS and CMS have independently designed their own trigger systems [15, 16]. The ATLAS trigger system operates in three stages: Level-1 (L1), Level-2 (L2), and the event filter. The L1 is the

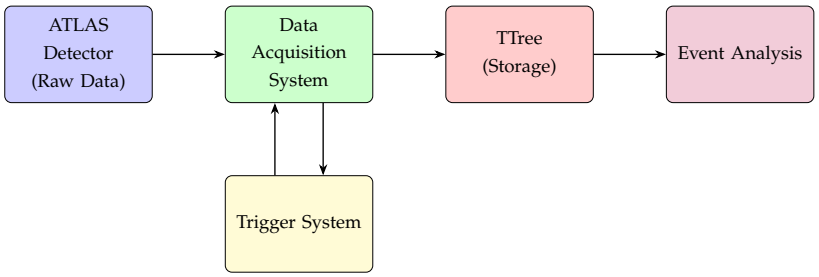


Figure 1.2: ATLAS' event data flow.

first and fastest filtering step. It uses only basic detector data to make quick decisions and can handle up to 75,000 events per second. The event processing time for the L1 must not exceed 2.5 µs, a requirement met using custom-built hardware. The L2 uses more detailed detector information to refine event selection, and it focuses on specific regions of interest identified by the L1. The L2 reduces the event rate to below 3,500 events per second with an average event processing time of 40 ms. The event filter uses the full granularity of the detector information to decide which events to store. It reduces the data rate further to about 200 events per second, with each event taking around four seconds to process [15].
The Data Acquisition System (DAQ) collects event data from the detector at the rate set by the L1. It sends specific data to the L2 when requested, focusing on regions of interest identified by the L1. If an event passes the L2 selection, the DAQ assembles the full event and transfers it to the event filter. Events that pass this final selection step are then saved for further analysis [15].
Events are stored in the TTree format, a data format for HEP developed by ROOT [17]. In HEP, data from an event is structured as a record containing multiple collections of related information. For example, a single event may include a collection of particles, each with properties such as momentum and energy; a collection of jets, which are clusters of particles moving in a similar direction; and a collection of tracks, representing the paths particles take through the detector [18]. ROOT's TTree format supports a columnar data layout for nested sub-records and collections, which is ideal for HEP analysis that often requires access to many events, but only a subset of the information stored in them [18, 19]. *Figure 1.2* (p. 3) illustrates the data flow within ATLAS, from the initial collision detection to the final storage format used for physics analysis.
To archive data, CERN uses the Worldwide LHC Computing Grid (WLCG), which is a global collaboration of computer centers and universities that provides the necessary computing resources for the experiments at CERN. The WLCG is divided into three tiers: Tier0, Tier1, and Tier2. The Tier0 center is located at CERN and is responsible for the coordination of the overall grid, where it must be able to handle the data streams coming from the experiments in the LHC. The incoming data must be archived on tape for long-term storage, and a second copy must be made at a Tier1 center. Tier1 centers receive a portion

of both raw and reconstructed data from Tier0 for long-term storage, while also providing storage for simulated data. Additionally, they provide data-intensive analysis facilities that support large-scale processing tasks. Each Tier2 center, which serves as a dedicated analysis facility for a single experiment, is associated with a Tier1. The Tier2 centers also provide computing resources for simulations, where the simulated data is sent to Tier1 for storage [20, 21].
The total global storage capacity of the WLCG is around 1.65 exabyte, with around 40% of it being on disk and 60% on tape [21]. The WLCG has multiple different storage software systems, where two of these are called EOS Open Storage (EOS) and CERN Tape Archive (CTA), and they are both used at the Tier0 site [22, 21, 23]. The EOS is a disk-only storage system and has around 650 petabytes of storage capacity from over 60,000 hard drives. It is designed to be a low-latency file system that is used for physics analysis [21]. The CTA is a tape storage system, implemented as an EOS backend. It relies on dedicated EOS instances to manage temporary disk storage and the movement of files between the disk pools and the tape library [24].
The TTree format has been a reliable way to store HEP data for over 20 years, with more than 1 exabyte of data stored in this format. However, it was developed before modern hardware and storage technologies like NVMe SSDs and object stores existed and thus does not fully exploit their capabilities [19]. With the HL-LHC upgrade and the expected increase in data, a new data format is needed. To meet this need, ROOT has developed a new data format called *RNTuple*, which is the new columnar data format that should replace the TTree format in the long term for the experiments at LHC. RNTuple is built to take full advantage of modern hardware and storage technologies. Its implementation leverages new features available in recent C++ standards like smart pointers to enforce clear ownership and lifetime management of objects. This contributes to lower runtime overhead and results in simpler, more maintainable code [19, 25].
RNTuple improves I/O performance and reduces the storage size of data on persistent storage media [19]. One of the ways it achieves this is by using little-endian encoding for integer and floating-point numbers, aligning with the byte order of most popular architectures. This enables direct memory mapping of the data, improving data compression [19]. Before the RNTuple can become the main data format in Run 4 for ATLAS, it must be implemented in the ATLAS software [26].
Athena is part of the ATLAS offline software and is designed to process events from the trigger and the DAQ, deliver the processed data to physicists for analysis, and provide the necessary tools for conducting that analysis. To achieve this, technical requirements have been established, such as constraints on processing time and memory consumption per event to remain within financial limits. Additionally, physicists' requirements encompass the ability to accurately reproduce details of the underlying processes based on data

collected by the detector [27].
Athena is responsible for nearly all production workflows in ATLAS, including event generation, digitization, simulation, and reconstruction [26, 27]. This results in a huge codebase, as well as a high level of complexity, necessitating a modular, robust, and flexible design [26, 27]. Athena follows a component-based model, where the software is constructed using plug-in components defined by configuration files. These components can be replaced or adapted as requirements evolve throughout the lifetime of the experiment [27].
To prepare for Run 4, work has been carried out towards ensuring that the `TEvent` class — an interface for handling event data — supports data in not just the TTree format but also the RNTuple format. Additionally, an `RAuxStore` has been created to provide auxiliary storage support for the RNTuple. The `RAuxStore` has been created based on the `TAuxStore` — the auxiliary store that uses the TTree format — and offers the same functionality, but specifically for RNTuples [28].

2 — Problem Definition

In our previous work, we created the `RAuxStore`, the RNTuple implementation of TTrees' `TAuxStore` [28]. Here, we conducted a benchmark experiment to show the performance of the `RAuxStore` whilst comparing its performance to that of the `TAuxStore`. Given that RNTuple leverages modern technologies to achieve better I/O performance and lower runtime overhead, we expected the `RAuxStore` to outperform the `TAuxStore`. However, the results showed that the `TAuxStore` had the best overall performance [28]. Although the implementation of the `RAuxStore` along with unit tests for it has been merged with the ATLAS repository, the benchmark code is only available in our fork of the ATLAS repository[1], along with optimizations made for the `RAuxStore`.
We, therefore, aim to investigate why `RAuxStore` did not perform as expected. Through performance measuring, we will identify performance bottlenecks and assess whether the observed differences are due to inefficiencies in the `RAuxStore` implementation itself or potential inefficiencies within ROOT's handling of RNTuples. We will then improve upon the possible inefficiencies for the `RAuxStore`. Otherwise, if the problems lie with the RNTuple, we will inform and work with ROOT to try to solve them. To ensure that our optimizations are effective, we will design a well-defined benchmark to measure and validate the impact of our improvements.
This leads us to our problem statement:

> *How can we identify bottlenecks and improve the performance of the* `RAuxStore`*, and how can we validate these improvements through a well-defined benchmark?*

[1] https://gitlab.cern.ch/mads/athena

3 — Related Work

In this chapter, we examine work related to the analysis of the `RAuxStore` and benchmarking, including comparing the performance of other parts of the RNTuple and the TTree, as well as investigating the benchmarking of computer systems.

3.1 Comparing the Performance of the RNTuple to the TTree

A multitude of papers have compared the performance pertaining to RNTuples. Most of these papers, however, compare the performance of the RNTuple to the TTree [29, 30, 26, 18], as opposed to comparing the performance of an implementation of an element that uses the RNTuple/TTree as we did in our previous research with the `RAuxStore` [28].
[26] benchmarks the storage efficiency of RNTuple and TTree in ATLAS software. They use files with event data stored in either TTree or RNTuple format, where the ones in the RNTuple format are around 20% smaller than the corresponding ones in TTree format.
[29] provides both a qualitative and an experimental comparison of RNTuple, TTree, and two other I/O libraries: HDF5 and Apache Parquet. In the qualitative comparison, they examine the support for different compression algorithms. While all libraries support some form of compression, the availability of native compression algorithms varies significantly. [29] finds that the required compression methods they need are supported by TTree and are under development for RNTuple. In the experimental comparison, they focus on throughput and compression by analyzing file size. Throughput was tested with a file that fits within the I/O library on an HDD, SSD, and CephFS — a distributed file system. The results show that the RNTuple outperforms all other libraries, with particularly high performance on SSDs. For data compression, they find that the RNTuple achieves the best results, producing the smallest file sizes among all tested libraries.
[30] and [18] both compare throughput and storage efficiency for the RNTuple and the TTree, showing significant improvements for the RNTuple. Additionally, these experiments have been conducted with realistic benchmarks mimicking actual analysis scenarios.

3.2 Benchmarking of Computer Systems

[31] looks at the importance of standardizing techniques for benchmarking computer systems. They mention reproducibility, a main principle of the scientific method, to help

validate claims. They explain that a program's performance depends on various factors, including the input, the compiler, the runtime environment, the hardware, and the measurement techniques. If any of these aspects are missing or inadequately described in an experimental design, the results cannot be reproduced. In such cases, the findings may be incorrect or even misleading as verification becomes impossible. To address this, the concept of interpretability is introduced, which is less strict than reproducibility. An experiment is considered interpretable if it provides enough information for other scientists to understand it and draw their own conclusions [31].

To evaluate the state of benchmarking in research, the authors analyzed 120 papers from three top conferences, selecting 10 random papers from each conference in the years from 2011 to 2014. Their findings indicate that while most papers report hardware details, software environments are often omitted. Additionally, they find that most papers are hard to interpret and easy to question, because measurement techniques and reporting methods are poorly described [31].

To address these shortcomings, a set of rules is proposed. If applicable, these should be followed to help researchers, reviewers, and readers in designing, conducting, and evaluating empirical studies. The rules are categorized into four key areas: state of the practice, analyzing experimental data, experimental design, and reporting results [31].

Regarding the state of the practice, they encourage authors to report units unambiguously. For instance, they recommend following the PARKBENCH [32] committee recommendations using *B* for bytes and *b* for bits. Similarly, base-2 units should follow the IEC standard to avoid confusion with base-10 units, which adds the prefix *i* e.g. *MiB* for mebibytes. Furthermore, they stress the importance of not cherry-picking results but instead reporting all of them, even if they do not support the desired conclusion [31].

When analyzing experimental data, careful consideration should be given to how results are summarized. Deterministic outcomes can be described using simple algebraic methods, whereas with nondeterministic data, statistical approaches are more appropriate. Moreover, they state that execution time is often the most meaningful metric for performance assessment due to its undebatable meaning [31].

Experimental design should prioritize reproducibility or, at the very least, interpretability. Significant setup parameters must be explicitly stated, and the measurement environment should minimize interference to reduce variability in results. Some programs establish their working state on demand, and thus, the first run of a program can be slower than the following runs, in which case, a warmup run should be considered. Additionally, if the program is small and executed repeatedly, the impact of a warm vs. cold cache should be considered. A small program's data may fit entirely in the cache, artificially accelerating execution and yielding performance results that do not reflect real-world conditions [31].

When presenting results, the goal should be to communicate system behavior clearly and

enable the reproduction of results. They encourage the use of visual representations, such as graphs. Careful thought should go into what is being shown in the graph as well as how it is presented. As a general rule, sufficient data should be included to ensure a comprehensive understanding, and lines connecting measurements should only be used when trends are evident [31].

4 — Background

4.1 TTree

The TTree data structure is specifically designed to store large collections of objects of the same class. It is optimized to minimize disk usage while maximizing data access speed because the most common task for data access in HEP is the selective, sparse scanning of data [33, 34]. In addition to TTrees, ROOT provides the TNtuple class, a specialized form of TTree that stores only floating-point values. In contrast, a TTree can accommodate a wide range of data types, including simple data types, arrays, and objects [33]. The TTree and TNtuple classes form the foundation of the tree system. The goal with the tree system was to provide a unified data model, allowing the same language and query style to be used across all experiments [17]. A visual representation of the TTree data structure can be seen in *Figure 4.1* (p. 10). A TTree consists of branches, where the structure of branches allows data to be organized in a way that optimizes access patterns based on expected usage. When two variables are independent and unlikely to be accessed together, they should be stored in separate branches. Vice versa, when variables are closely related, such as the coordinates of a point, storing them within the same branch improves efficiency. A branch contains individual data elements, known as leaves, which hold the actual values [33].

TTrees support flexible storage by allowing branches to be written to separate files. By default, each branch is written into a separate buffer within the file. This allows efficient iteration over a branch's data by requiring only the reading of its associated buffer [34].

Branches can also be split into sub-branches. If the number of branches becomes too high, adjusting the buffer size is necessary to maintain performance. In terms of reading,

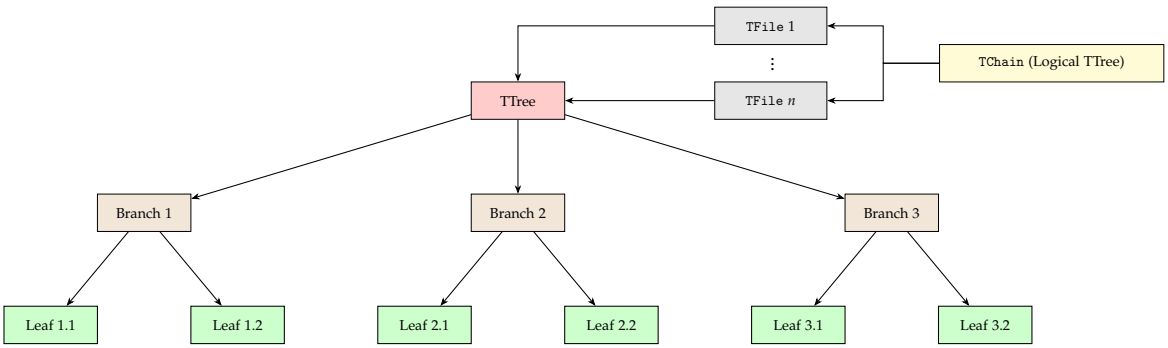


Figure 4.1: TTree data structure for a tree with a depth of two.

split branches offer faster access since variables of the same type are stored consecutively, avoiding redundant type lookups. However, for writing, split branches are slightly slower due to the additional overhead [33].
As the volume of data increases, a TTree instance can be distributed across multiple `TFile` instances. These `TFiles` can later be combined into a single logical object, known as a `TChain`, enabling seamless access to distributed data. Because a `TChain` inherits from a TTree, it maintains the same benefits in terms of optimized data access, even when spanning multiple files [34].

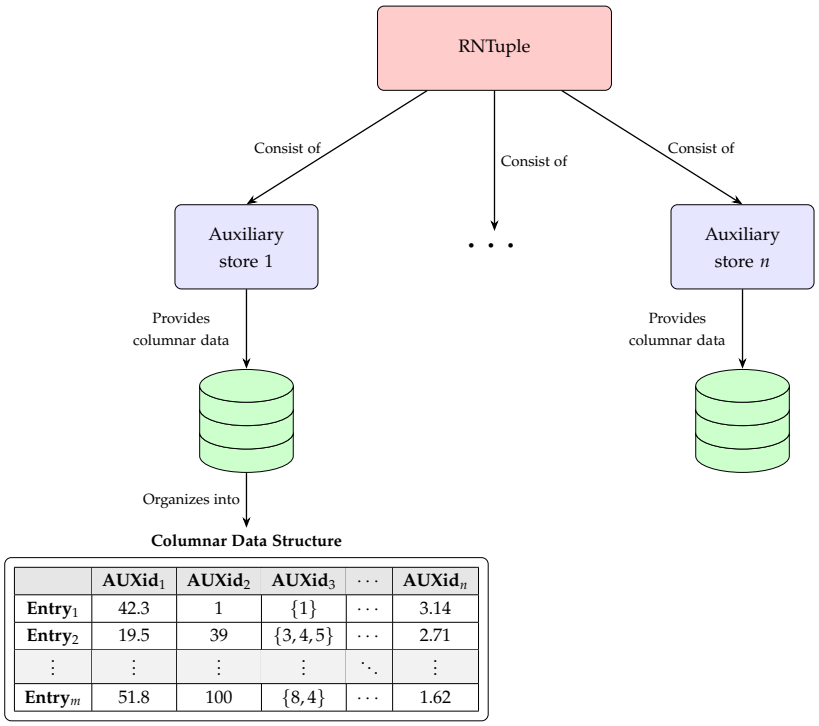
4.2 RNTuple

The RNTuple data format is designed to overcome the limitations of the TTree data format while preserving its fundamental advantages [35]. It breaks backward compatibility to provide the flexibility needed for greater space savings, increased I/O speeds, and a more robust data access API [35, 36].
The RNTuple provides the best performance for typical HEP workloads, aiming to closely match the I/O speed of modern hardware. It achieves 2-5 times better single-core performance than the TTree and is designed to scale with higher core counts. It is conditionally thread-safe, meaning it avoids static and global variables and is designed for multi-threaded environments. Compared to TTree, RNTuple produces up to 25% smaller files due to improved handling of nested collections and other optimizations [35].
RNTuple implements a four-layered architecture consisting of the storage layer, the primitives layer, the logical layer, and the event iteration layer. The storage layer abstracts underlying storage systems and, optionally, provides packing and compression functionality. It is responsible for the storage and retrieval of byte ranges organized into clusters and pages. A page contains a sequence of values for a specific column, whereas a cluster encompasses pages storing data for a defined row range. The primitives layer organizes elements of simple types into pages and clusters. The logical layer converts complex C++ types into columns of simple types. For instance, a vector of floats becomes an index column and a value column. At the lowest level, the event iteration layer provides standardized interfaces for iterating over events [19].

4.3 Auxiliary Store

The ATLAS Event Data Model (EDM) is the framework that organizes and structures event data within the ATLAS experiment. It defines how data from proton-proton collisions is represented, stored, and accessed throughout different stages of processing, from raw detector readings to high-level physics objects used in analysis. The EDM is built within

the Athena framework and follows a blackboard style architecture [37].
The blackboard architecture is a design pattern named after the metaphor of experts collaboratively solving a problem by sharing information on a common blackboard. This pattern consists of three primary components: the blackboard, which serves as a shared data structure; knowledge sources, which are independent specialized modules that interact with the blackboard performing specific tasks; and a control component that manages the knowledge sources and coordinates them. The blackboard enables asynchronous communication between knowledge sources, allowing them to interact through the shared blackboard without directly referencing each other. Each source simply reads from and writes to the shared blackboard, while simultaneously maintaining the current state of the system and providing a centralized representation of all relevant information [38].
An advantage of the blackboard architecture is its modularity, making it straightforward to add new knowledge sources without disrupting existing components. Furthermore, the architecture offers flexibility in execution flow, as it is not constrained to either top-down or bottom-up processing. Instead, knowledge sources can be activated dynamically based on the current information on the blackboard [38]. These characteristics align with Athena's principles of modularity and flexibility as described in *Chapter 1* (p. 5).
ATLAS uses a multi-tiered data format hierarchy that begins with raw data being collected, which contains direct detector readouts and trigger decisions. This is processed into Event Summary Data (ESD), the first reconstructed data format that structures detector information into more meaningful physics objects while retaining most detector details. Analysis Object Data (AOD), which is derived from ESD, provides a reduced dataset optimized for physics analysis and contains physics objects and other elements of analysis interest [37].
While Run 1 of the LHC was a success, the design of the ATLAS EDM revealed limitations [39]. The data structures used were highly complex and relied on costly C++ features. This complexity made direct writing to the ROOT format impractical, necessitating a conversion before the data could be used. While functional, this approach introduced overhead and created a dependency problem, as files written with the ATLAS EDM required access to the complete ATLAS software release. Consequently, ATLAS physicists would convert data samples using the full EDM to ones using a simplified format that could be read by ROOT. This led to duplication of data and made it difficult to maintain analysis tools, because they had to work on both the full ATLAS EDM and on ROOT. As a result of this, during the 2013-2014 LHC shutdown, ATLAS introduced the *xAOD* format as an evolution of the AOD structure to enhance usability and performance. A key feature of the xAOD is its use of an *auxiliary store*, where object properties are stored separately from the objects themselves [39].
The objects handled by the auxiliary store may be any arbitrary type and are stored as such. However, for containers of objects, the store uses a ROOT specific derivation of the



	$\mathbf{AUXid}_1$	$\mathbf{AUXid}_2$	$\mathbf{AUXid}_3$	$\cdots$	$\mathbf{AUXid}_n$
$\mathbf{Entry}_1$	42.3	1	$\{1\}$	$\cdots$	3.14
$\mathbf{Entry}_2$	19.5	39	$\{3,4,5\}$	$\cdots$	2.71
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$\mathbf{Entry}_m$	51.8	100	$\{8,4\}$	$\cdots$	1.62

Figure 4.2: The RNTuple's relationship with its auxiliary stores and how the stores provide columnar data.

`std::vector` called `DataVector`. The `DataVector` acts as a container that holds pointers to its elements, while each element maintains a reference back to the container and its position within it. The auxiliary store is then responsible for managing the corresponding object properties, which are stored as vectors of simple data types. This organization allows the system to retrieve specific properties without loading entire objects into memory [39].

Each EDM class includes a *static auxiliary store* containing variables treated as class members and a *dynamic auxiliary store* to manage additional user-defined data. If a requested variable is not found in the static store, the request is forwarded to the dynamic store, allowing users to extend objects with new variables as needed. The dynamically added variables follow a structured naming convention and are identified by having the same name as the store but with a *Dyn* suffix [39].

The TTree has an implementation of the auxiliary store, which is what the physicists are currently using when performing analysis. In our previous work, we implemented the auxiliary store for RNTuples in Athena, and it is now available in the ATLAS software release [28]. The hierarchical structure of RNTuple, consisting of auxiliary stores, and its data organization can be seen in *Figure 4.2* (p. 13).

4.4 Benchmark

Benchmarking is a structured process designed to compare the performance of one system against another in a deterministic and reproducible manner [40, 41]. Traditionally, benchmarking has been used to evaluate performance in terms of work done relative to time and resource consumption. However, its scope has expanded to evaluate additional factors such as system reliability, security, and energy efficiency. A benchmark consists of three key components: workload, metrics, and measurement methodology. The workload represents the set of tasks a system is expected to perform, serving as the basis for performance evaluation. The metrics determine which values should be extracted from measurements to generate meaningful benchmark results. The measurement methodology defines the complete process for executing the benchmark, collecting measurements, and generating results [41].
Designing an effective benchmark requires meeting certain, sometimes conflicting, qualitative criteria to ensure meaningful results. These criteria can be defined as: relevance, reproducibility, fairness, verifiability, and usability [42].
Relevance is perhaps the most important criterion for a benchmark, as even a perfectly designed workload is of little value if it does not provide meaningful insights. In benchmark design, relevance refers to ensuring that the benchmark aligns with its intended use and provides useful information for that domain. Assessing relevance and workload involves two key dimensions: the breadth of applicability and the degree to which the workload is relevant for the area. For example, an XML benchmark is highly relevant for evaluating XML processing performance but would not be suitable for assessing CPU performance. Additionally, benchmarks that are highly relevant to a specific domain tend to have narrow applicability, while those designed for broader use often sacrifice specificity and relevance for any particular area [42].
A benchmark satisfies reproducibility when repeated executions under identical test configurations yield consistent results, ensuring reliability and trust in its measurements. Here, descriptions of the environment of the benchmark, including both the software and hardware, are necessary [42].
Fairness ensures that different test configurations are evaluated based on their intrinsic performance rather than artificial constraints. One key aspect is seeking input from experts in the field to avoid unintentionally skewing results. Since benchmarks are inherently simplified representations of real-world scenarios, constraints should be introduced in the test environment to prevent results from being artificially inflated due to this simplicity. The test environment itself should be appropriately configured and include components that align with the specific requirements of the benchmark. Finally, it is important to assess whether the software executing the benchmark is also simplified, as stripped-down

versions may achieve higher performance by omitting essential features, such as security functionalities [42].
Verifiability requires the benchmark to provide clear evidence and documentation that supports the accuracy of its measurements, allowing users to confirm that the results truly reflect system performance. This includes self-validation, like the use of configuration options that allow the user to alter the settings of the benchmark to verify the results. These configuration options should be controlled by the benchmark and should preferably be included together with the results. Functional verification supports verifiability, which is about verifying the output of the benchmarks by detecting when incorrect results are produced. To improve verifiability further, more detailed results showing elements not strictly necessary to verify the result could be included. Inconsistencies within these results may show problems for the actual benchmark results [42].
Finally, usability is about increasing ease-of-use for a benchmark. This means building proper environments for executing the benchmark. It should not be difficult for the user to acquire the necessary components to execute the benchmark. Similarly, it should be clear to the user that the benchmark is running correctly and producing valid results without requiring extensive verification [42].
To evaluate the performance of a system, the system must execute a benchmark, which can be one of four types: a synthetic benchmark, a microbenchmark, a kernel benchmark, and an application benchmark. Synthetic benchmarks are artificial programs designed to simulate specific application behaviors; while flexible, they may not fully capture real-world complexities. Microbenchmarks focus on isolated system components, such as processors or memory, revealing peak performance but not overall system behavior. Kernel benchmarks extract critical code segments from applications, highlighting frequently executed operations; they are portable but may not stress all system components realistically. Application benchmarks use real-world applications, offering the most accurate performance assessment, though often with reduced input datasets for practicality [41].

4.5 Performance Measurement Techniques

To monitor changes in a system's state, different measurement techniques can be used. To measure how much time a system spends in certain states and to understand the system's behavior, performance profiling can be used [41].
There are various techniques for measuring the execution time, each involving a trade-off between resolution, accuracy, granularity, and difficulty. Resolution defines the smallest measurable time interval, while accuracy determines how closely the recorded time reflects the actual time. Granularity describes the level of detail in measurement, with coarse-grained approaches capturing execution times of larger program structures, such

as functions, and fine-grained approaches focusing on individual instructions. The difficulty, although subjective, reflects the effort required to obtain meaningful results [43].
Beyond these attributes, the software's design can significantly impact the ability to measure execution time. Although not classified as a formal attribute, since it cannot be quantified or qualified in every case, poor design choices can make profiling more challenging. Software structured in an ad-hoc manner presents particular difficulties, as inconsistent or poorly defined entry and exit points make it difficult to determine precise start and stop times, making accurate measurements nearly impossible [43].
When selecting a method for measuring execution time, thought should be given to what should be obtained from the measurement. Some of the most common reasons include code optimization, real-time performance analysis, and debugging timing errors. Depending on what we optimize for, optimizing code could use both coarse-grained and fine-grained methods. For global-scale optimizations, a coarse-grained method of timing entire routines is sufficient. However, for local optimization, where we are fine-tuning an application, a fine-grained method would be more appropriate [43].
There are different methods for measuring execution time, differing in their granularity level. One tool to perform coarse-grained measurement is to use the time command, which is available on UNIX systems by prefixing a command with `time`. This command is executed in the terminal and can thus only be used to time the entire application. However, it not only accounts for the time taken to execute the application but also includes the time consumed by the system and the user. For a fine-grained measurement technique, an embedded software timer can be used. Here, the measurements can be added directly into a function, and multiple code segments can be tested within a single execution. This method works by reading the initial value of the timer at the beginning of the code segments' execution and again after they are executed. The difference between these values represents the number of timer ticks that have elapsed. To obtain meaningful results, it is necessary to determine the duration of a single timer tick and ensure that the accuracy is sufficient [43].

5 — Benchmark Analysis

In this chapter, we analyze the benchmark code from our previous work [28] and apply the theory described in the previous chapter to suggest ways in which the benchmark could be improved.
To aid in the understanding of the analysis, *Figure 5.1* (p. 18) illustrates the relationship between the different components within our benchmark code. The benchmark code is organized into three primary components: a setup component, which prepares the environment; a warmup phase, which, if performed, is performed prior to the benchmark phase; and a benchmark phase. Both phases require the initialization of store objects. Within the benchmark phase, one or more tests are executed, where each test is defined by a combination of a specific auxiliary store type and a percentage of the auxiliary IDs to pick. Each test consists of one or more runs, each being a single timed execution. The warmup phase also consists of a single run — performed with different entries to the benchmark phase — where the timing is discarded.

5.1 Analyzing the Benchmark Code

In this section, we analyze the results from the benchmark we performed in our previous work. Here, we tested the `RAuxStore`'s performance compared to the `TAuxStore`, on an SSD, by conducting two variations of benchmark experiments — one with a warmup phase and one without. In our previous work, we used a machine, which we no longer have access to, and we thus use a different one for this project. The machine we use in this project has the following specifications:

- OS: Fedora Linux 41 (Workstation edition)
- CPU: AMD Ryzen 7 5700X
- Memory: 4×16GB DDR4-3200 MT/s
- M.2 NVMe SSD: Kingston SNV2S2000G
- HDD: Seagate ST500DM002
- Compiler: GCC 14.2.1
- ROOT Version: 6.34.06

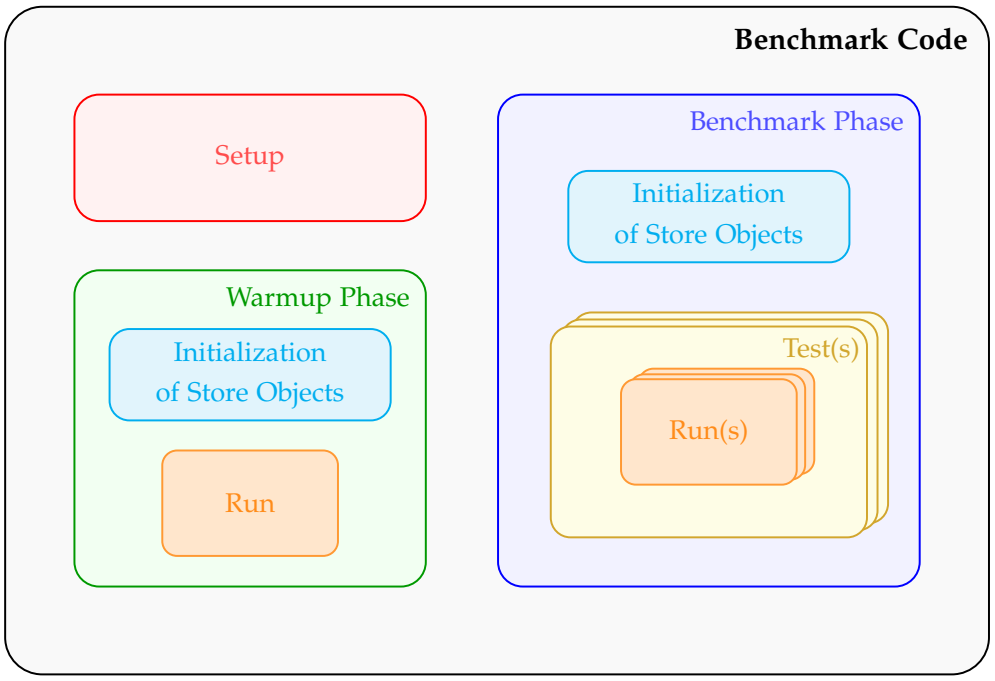


Figure 5.1: Euler diagram illustrating the relationship of the different components within our benchmark code.

Executing the same benchmark as we did in our previous work on our new machine yields the results seen in *Table 5.1* (p. 18) and *Table 5.2* (p. 19).

Benchmarking with a warmup phase

% of auxiliary IDs	`RAuxStore`	`TAuxStore`	Difference
10%	2445 ms	930 ms	1515 ms
40%	3918 ms	2009 ms	1909 ms
75%	5906 ms	3317 ms	2589 ms
100%	7407 ms	4218 ms	3189 ms
Total Time	19,676 ms	10,474 ms	9202 ms

Table 5.1: Benchmarking tests comparing the throughput of the `RAuxStore` compared to the `TAuxStore`. Each test performs 20 runs.

While not identical to the results from our previous work, the overall results are similar, where the `TAuxStore` is faster than the `RAuxStore` when a warmup phase is performed and slower when it is not.
To help us analyze these results, as well as the results from our previous work, we conducted additional experiments, the results of which can be seen in *Table 5.3* (p. 20). In our previous work, we conducted two experiments; one where we performed a warmup

Benchmarking without a warmup phase

% of auxiliary IDs	`RAuxStore`	`TAuxStore`	Difference
10%	1134 ms	916 ms	218 ms
40%	2525 ms	3862 ms	1337 ms
75%	4654 ms	5178 ms	524 ms
100%	6030 ms	6119 ms	89 ms
Total Time	14,343 ms	16,075 ms	1732 ms

Table 5.2: Benchmarking tests comparing the throughput of the `RAuxStore` compared to the `TAuxStore`. Each test is run 20 times in separate executions.

phase and one where we did not [28]. The warmup phase was performed to avoid overheads unrelated to the reading process. Performing the warmup phase resulted in the `RAuxStore` being slightly faster and the `TAuxStore` being significantly slower. However, when looking into the specifics of the slowdown of the `TAuxStore`, it appeared that an issue arose the first time we encountered a given type. A particular section of the function responsible for setting up the input data for the `TAuxStore`, specifically the part where a branch is connected to the input tree, typically takes a few microseconds. However, when encountering the type `std::vector<std::vector<ElementLink<DataVector<xAOD ::IParticle>>>>` for the first time, this operation took multiple seconds. In the benchmark files we used, an auxiliary ID of this type appeared in 176 out of the 213 total auxiliary stores.

The result for the first test configuration shows the result of executing the benchmark without a warmup phase, with 20 stores selected from the remaining 37 stores. Here, the `TAuxStore` is faster than the `RAuxStore`.

However, when executing the benchmark with a single auxiliary store that does contain an auxiliary ID of the problematic type, we get the results seen in the second test configuration. Here, the `TAuxStore` is more than 10 times slower than the `RAuxStore`. This is down to the aforementioned issue of the `TAuxStore` operation taking multiple seconds when encountering the type of `std::vector< std::vector< ElementLink< DataVector< xAOD:: IParticle>>>>`. Here, this operation accounts for almost the entirety of the benchmark for the `TAuxStore`.

If benchmarking with the same constellation, but with a warmup phase, we get the results in the third test configuration. Here, the `RAuxStore` result remains mostly the same, but the `TAuxStore` result is now very similar to the `RAuxStore` result, with the overhead of the encounter with the problematic type being in the warmup phase, and thus not timed.

In our original benchmark, which we reproduced in *Table 5.2* (p. 19), we are performing

#	Test Configuration	RAuxStore	TAuxStore
Scenario 1: 20 Stores Without Problematic Type			
1	No Warmup Phase	6060 ms	3278 ms
Scenario 2: 1 Store With Problematic Type			
2	No Warmup Phase	140 ms	2021 ms
3	With Warmup Phase	139 ms	129 ms
4	No Warmup Phase (Avg. of 20 Runs)	211 ms	224 ms

Table 5.3: Benchmark comparison of `RAuxStore` and `TAuxStore` performance across different test configurations (with around 20,000 entries and 100% of the auxiliary IDs within the auxiliary stores are used). *Problematic Type* refers to the type `std::vector<std::vector<ElementLink<DataVector<xAOD::IParticle >>>>>`. For all configurations, we perform 20 separate executions and report the average execution time. The fourth configuration additionally performs 20 runs within a single test and calculates the average time across all runs. For scenario one, all stores containing IDs with the problematic type are filtered out, and 20 stores are selected from the remaining stores. For scenario two, a store with the name: `AntiKt10TruthSoftDropBeta100Zcut10JetsAux` is picked. This store contains 10 auxiliary IDs, and one of them is of the problematic type.

a single run for each execution of the application. As the first run of the `TAuxStore` is significantly slower if the problematic type is encountered, we are taking the average of these initial slow runs, which result in the `TAuxStore` appearing much slower than it generally is. Comparing `TAuxStore` in *Table 5.2* (p. 19) with *Table 5.1* (p. 18), this slowdown only appears when using 40%, 75%, and 100% of the auxiliary IDs. For 10% of the auxiliary IDs, the time taken is more or less identical with or without a warmup phase for the `TAuxStore`. This is due to the problematic type not being present in this subset of the auxiliary IDs.

Performing the benchmark as it was performed in configuration two, but averaging the results over 20 runs within a single benchmark phase, yields the results seen in test configuration four. This is essentially the same as averaging the 19 runs of test configuration three and a single run of test configuration two. Here, the average time for the `TAuxStore` is much lower as it now contains the one run taking several seconds, but in addition to that, it also contains the faster runs that even out the time. Performing more runs would thus result in the fast runs gradually compensating for the slow run, and with enough runs, the results of `TAuxStore` would approach that of test configuration three.

While this explains the slowdown of the `TAuxStore` when not performing a warmup phase, it does not show why the `RAuxStore` is faster without a warmup phase. Executing the benchmark and looking at the individual runs, those performed without a warmup phase are all generally 500 ms faster than when performing one.

Additionally, when comparing the `RAuxStore` times in configuration two with those of configuration four, we can see that the results become slower when performing additional runs. More specifically, if we execute the benchmark with the same specification as we did in our previous work [28] (same auxiliary stores, same entries, 20 runs within one benchmark phase), we get different results for different runs. Here, the first run is about 1000 ms faster than the remaining 19 runs. This is true both with and without a warmup phase, and results in the average time being significantly slower.

5.2 Applying the Benchmark Theory

Based on the background of benchmarking described in *Section 4.4* (p. 14), we analyze the benchmark conducted in our previous work [28]. There are three necessary components for a benchmark: workload, metrics, and measurement methodology [41]. The workload is, in our case, the methods of the `RAuxStore` and `TAuxStore` that we call in order to get data from the stores. The metric we use for this benchmark is the time it takes to perform these methods. Here we perform multiple runs, where for each run we start the time at the beginning and stop it at the end, and the average of these runs is our benchmark result. Finally, our codebase, consisting of the `TAuxStore`, the benchmark code, and our altered `RAuxStore`, can be considered the measurement methodology.

Three out of the five benchmark criteria mentioned in [42] have been at least partially fulfilled: relevance, reproducibility, and usability. We consider the benchmark relevant because it tests methods that a user can encounter while reading event data. Specifically, we have tested the performance of the `RAuxStore` and the `TAuxStore`, where we deem the benchmark to have narrow applicability but provide meaningful insights into what we want to measure, making the benchmark relevant.

Modern computers introduce many complexities that can affect benchmark variability, including physical disk layout, network conditions, and user interactions. To ensure reproducibility, we have designed our benchmark to run on a stable machine with no external user interaction during execution. Additionally, we have documented the hardware and software components of the machine, allowing others to replicate the benchmark on an equivalent machine for consistent results. Our benchmark has high usability, as we have set up a simple environment that allows users to execute it with minimal effort. It can also be executed on most consumer hardware, as it requires minimal computational resources.

We deem the two remaining criteria, fairness and verifiability, not fulfilled in our case. Ensuring fairness in a benchmark requires careful consideration of multiple factors. However, fairness was not a primary focus in the design of our benchmark, as we considered it less critical for our research.

Our benchmark has limited verifiability, as we have predefined what to test and how to test

it without offering a lot of configuration options that other users might use to verify the results. For example, we have only tested with a single file, where we have hardcoded the auxiliary store names directly in the code, meaning that the results cannot be verified on different files, as they would not necessarily have the same store names. Furthermore, we only provide one number — which is the final result — for each test, which does not allow for a detailed analysis. Where more configuration options could improve verifiability, it could also decrease usability when providing too many options for the user to interact with.

There are several ways to improve the verifiability of our benchmark. We should dynamically load the store names from the file, allowing the benchmark to be executed with arbitrary files rather than relying on hardcoded values. Providing more detailed results than strictly necessary can enhance verifiability by offering greater transparency. Functional verification could also be done to handle the inconsistent results regarding the problematic type described in *Section 5.1* (p. 17).

The benchmark conducted in our previous work [28] was a synthetic benchmark. However, we attempted to represent the actual usage of the `RAuxStore`. While we do create instances of the `RAuxStore`, set it up, and read data from it, our way of accessing the data may not represent that of a realistic scenario. Doing it in this synthetic way means that the end result may not be representative of the actual usage of the application, and in this way, we are not sure if the issues we got were issues that would occur for ATLAS physicists.

6 — Performance Measuring

To understand the performance issues of the `RAuxStore`, we conducted performance measurements, comparing it with the `TAuxStore`. In our previous work [28], we identified a factor that affected this slowdown, specifically when loading an entry for a field. This occurs in the `getEntry` function when using the overloaded operator for an `RNTupleView` instance, which is implemented by ROOT.
To analyze the performance of this in both the `RAuxStore` and the `TAuxStore`, we use the C++ library function `std::chrono::high_resolution_clock` for precise timing measurements. Here, we start a clock just before loading an entry, in the `getEntry` function, and then stop it after we have loaded the entry — this results in a duration. As mentioned in *Section 4.5* (p. 15), the accuracy of such a tool must be verified to use the results from it. Here, the timing function reported a precision of 1 femtosecond, which is more than sufficient for our experiments.

6.1 Analyzing Performance Bottlenecks

For this analysis, we use six different files, representing three datasets, each available in two formats: TTree and RNTuple. The files in TTree format are obtained from the CERN Open Data Portal[1]. The corresponding RNTuple files were generated by converting the files with TTree data into files with RNTuple data using the same Python script as we did in our previous work [28]; this process preserves the event data entries. The first two datasets contain simulated event data and consist of the same auxiliary stores but differ in the number of entries, where one has 110,000 entries and the other has 150,000 entries. For these, we use all available entries. The final dataset consists of actual event data from the ATLAS detector recorded during Run 2. It features different auxiliary stores and contains slightly more than 600,000 entries, and we use the first 600,000 entries of the dataset. For all three datasets, we use a single auxiliary store containing one auxiliary ID and perform a single timed execution, preceded by a warmup phase. All runs are executed on an SSD.

[1] **File 1:** `mc20_13TeV:DAOD_PHYSLITE.37620494._000018.pool.root.1` is available at `https://opendata.cern.ch/record/80017/files/mc20_13TeV_MC_aMcAtNloPythia8EvtGen_MEN30NLO_A14N23LO_ttZnunu_file_index.json_0`
File 2: `DAOD_PHYSLITE.37620499._000012.pool.root.1` is available at `https://opendata.cern.ch/record/80017/files/mc20_13TeV_MC_aMcAtNloPythia8EvtGen_MEN30NLO_A14N23LO_ttZqq_file_index.json_0`
File 3: `DAOD_PHYSLITE.37001693._000015.pool.root.1` is available at `https://opendata.cern.ch/record/80000/files/data15_13TeV_Run_00276147_file_index.json_14`

Dataset Size	`RAuxStore`	`TAuxStore`	Difference
110,000 entries	359 ms	90 ms	269 ms
150,000 entries	639 ms	131 ms	508 ms
600,000 entries	718 ms	485 ms	233 ms

Table 6.1: Results from conducting the benchmark on datasets of varying sizes.

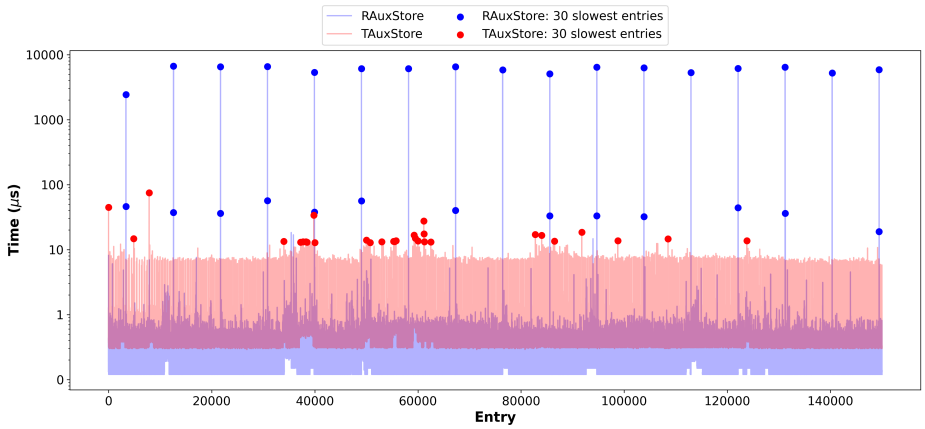


Figure 6.1: The time taken to load an entry for a field for both the `RAuxStore` and `TAuxStore` for individual event data entries. The plot uses a dataset with 150,000 entries.

The overall results from these configurations can be seen in 6.1
These results follow the same pattern as observed in our previous work [28], with the `TAuxStore` being significantly faster than the `RAuxStore` when a warmup phase is performed. However, for the dataset of actual event data, the difference between the `RAuxStore` and `TAuxStore` is less prevalent. To analyze the reason for the improved efficiency of the `TAuxStore`, we experiment on the durations obtained from the `getEntry` function. The results of these experiments can be seen in *Figure 6.1* (p. 24).
This figure shows all time durations for the `RAuxStore` and `TAuxStore`, as well as highlighting the 30 slowest entries for both auxiliary stores. Here, we observe a repeating pattern for both `RAuxStore` and `TAuxStore`. To identify this pattern, we use the *DBSCAN* algorithm from scikit-learn [44]. *DBSCAN* is a clustering algorithm that identifies dense regions based on a minimum number of neighbors, *minPts*, within a specified radius, ϵ. Clusters that satisfy this density requirement are considered core points and form part of a cluster. The goal of *DBSCAN* is to detect high-density areas separated by regions of lower density [45].
In our case, *DBSCAN* looks for clusters in the slowest 30 highlighted entries with an ϵ value of 2000 and a *minPts* value of 5, which means that we are looking for clusters that have at least five entries within a radius of 2000 µs. For the `RAuxStore`, two clusters emerged

in both datasets, where one cluster consists of times significantly slower than the other cluster.
In *Figure 6.1* (p. 24), we see that the slow cluster is repeating in a pattern approximately every 9,000 entries, from entry 12,000 to 150,000. The `getEntry` time in this cluster ranges from 5069.04 µs to 6770.77 µs, with an average of 6012.31 µs. An outlier also appears at around entry 3000, fitting with the previous pattern, but due to it having a value of only 2429.41 µs, it is not part of the cluster.
For the `TAuxStore`, all values are in the same time cluster, as all durations are under 100 µs. This cluster encapsulates the 30 slowest entry values, which range from 12.74 µs to 74.79 µs, with an average of 18.37 µs.
Results for the two other datasets were also plotted, revealing different patterns, as shown in *Figure A.1* (p. 46) and *Figure A.2* (p. 46). Despite their differences, they share a key similarity: the RNTuple forms two clusters, one with significantly slower times than the other cluster, while the TTree forms only a single cluster. These findings suggest a repeating pattern of slow entries in `RAuxStore` that does not appear in `TAuxStore`. While this pattern is pretty consistent within a single file, the pattern is different depending on the file.

6.2 Why `RAuxStore` is Faster Without a Warmup Phase

In *Table 5.2* (p. 19) and *Table 5.1* (p. 18), the `RAuxStore` is around 37% faster when not performing a warmup phase beforehand. Looking at the durations obtained from the `getEntry`, we can see that they are similar when comparing the runs with and without a warmup phase. However, the number of entries being loaded is different.
While the `RAuxStore` and `TAuxStore` are independent, they both use an auxiliary type registry, which is a singleton. Thus, updating the registry for the `TAuxStore` results in it being updated for the `RAuxStore` and vice versa. In our implementation of the `RAuxStore`, we did not implement the setup of an auxiliary field correctly, meaning if the auxiliary field did not exist in the registry, it would not be added correctly.
In the experiment, we use 20 stores, where one of them is the store with the name `EventInfoAux`, which contains 55 auxiliary IDs. Due to this incorrect implementation of setting up the auxiliary field, if we initialize the `RAuxStore` instances before initializing the `TAuxStore` instances, the registry is not updated correctly, and for that particular instance, we only find 10 auxiliary IDs. If the initialization of the `TAuxStore` instances is done first, the initialization of the `RAuxStore` instances finds the correct number of IDs as they are now available in the registry. Our benchmark code is executed in the order shown in *Figure 6.2* (p. 26), where we execute the `RAuxStore` benchmark before the `TAuxStore` one.

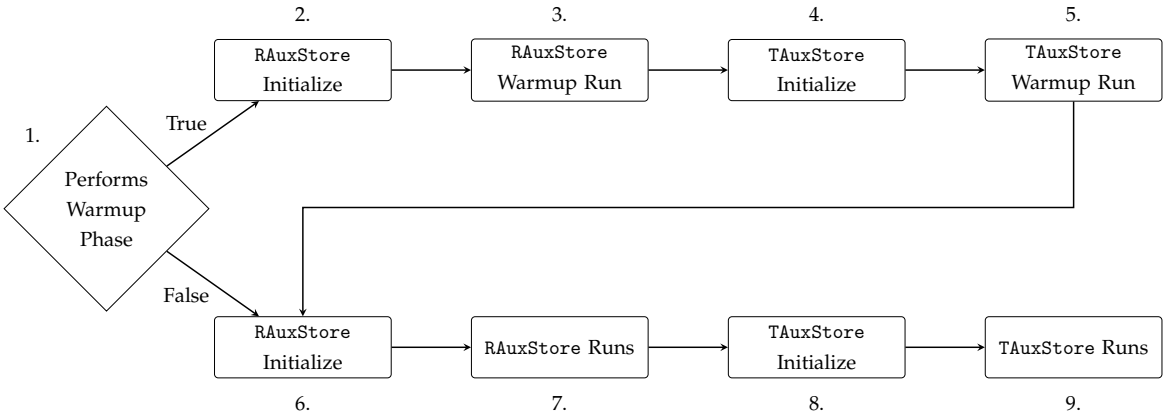


Figure 6.2: The order in which the benchmark is executed, based on whether or not we perform a warmup phase.

If we perform a warmup phase, this is done prior to the benchmark runs of both the `RAuxStore` and `TAuxStore`. Thus, when not performing the warmup phase, the `RAuxStore` instances are not correctly initialized, and the correct number of IDs for the aforementioned store is not available for the runs of the experiment. When we do perform a warmup phase, the `RAuxStore` instances are not correctly set up for the warmup phase, meaning we use the wrong number of auxiliary IDs in the warmup phase. However, after the warmup phase, we initialize the `TAuxStore` instances and update the registry. Therefore, when we initialize the `RAuxStore` instances for the benchmark phase, they can get the auxiliary fields from the registry. Overall, this means that in *Table 5.1* (p. 18), we load more entries for the fields than in *Table 5.2* (p. 19). Looking at the results when picking 100% of the auxiliary IDs, we load around 1.1 million entries for fields when benchmarking with a warmup phase. This number is reduced to around 200,000 when not performing a warmup phase. While this problem affects *Table 5.1* (p. 18) and *Table 5.2* (p. 19) as well as the results from our previous work [28], it does not affect the other experiments in this thesis. This is because we updated the benchmark code by, among other things, making it more dynamic, where instead of hardcoding the store names used, we get them dynamically. When getting them dynamically, we only add stores that contain at least one auxiliary ID. To ensure this, we initialize the `TAuxStore` instances and check how many IDs they have. Doing this initialization means that the registry is updated at the beginning of the application.
While this technically solves the problem, ideally, the `RAuxStore` should be altered to set up auxiliary fields in the registry correctly. To this end, changes have been made to the `RAuxStore`. The `auxFieldType` function should retrieve the correct type information for an auxiliary field. Our previous implementation did not do this and simply returned `null`. When initializing the `RAuxStore` in the `setupAuxField` function, we use the type known from the registry if possible, otherwise, we use the `auxFieldType` function. When using

Test Configuration	Run 1	Run 2-20 Avg.	Overall Avg.
`RAuxStore` With a Warmup Phase	7033 ms	7768 ms	7728 ms
`RAuxStore` Without a Warmup Phase	9193 ms	7902 ms	7968 ms
`TAuxStore` With a Warmup Phase	4314 ms	4241 ms	4245 ms
`TAuxStore` Without a Warmup Phase	5649 ms	4189 ms	4262 ms

Table 6.2: Benchmark results comparing execution times for `RAuxStore` and `TAuxStore` with and without a warmup phase, after modifications done to the `RAuxStore`.

this function to get the type information, we exit the `setupAuxField` early and return a success status code if the type information is not found. This meant that if the field is not in the registry, it would not be correctly set up.
Additionally, when executing the benchmark and getting the data from the stores, the `setupInputData` function is used to connect an auxiliary ID to an auxiliary field in the file. For this purpose, we need to check if the field is a primitive type or a composite type and set it up accordingly. Our previous implementation of `setupInputData` did not work correctly for some auxiliary stores, such as the store with the name `EventInfoAux`, which meant that some types were set to be neither primitive nor composite. This resulted in the store not getting the correct number of IDs associated with it.
With the changing of the `auxFieldType` function to get the correct type information and ensuring that the `setupInputData` sets up the correct number of auxiliary IDs, we get the results seen in *Table 6.2* (p. 27) when performing the same benchmark as in *Table 5.1* (p. 18), only with 100% of the auxiliary IDs.
Now, the `RAuxStore` is no longer faster without a warmup phase, and when the first run is excluded, the execution times with and without a warmup phase are similar. The results for the `RAuxStore` now resemble the results for the `TAuxStore`, where the first run without a warmup phase is significantly slower. Looking at the durations obtained from the `getEntry` function, the results are generally the same for the `RAuxStore` with and without a warmup phase. The cause of the slow time is actually the same as the one causing the slow time for the first run for the `TAuxStore` without a warmup phase as described in *Section 5.1* (p. 17).

6.3 First `RAuxStore` Run is Faster

In *Section 5.1* (p. 17), we saw that for the `RAuxStore`, the first run for each test was faster than the other ones. When testing with 100% of the auxiliary IDs and not performing a warmup phase, the median time of the durations obtained from the `getEntry` function remains nearly the same across all runs. However, the mean is significantly lower for

the first run compared to the other ones, by around 150 ns. Additionally, the other runs' maximum times are about five times higher, suggesting that some extreme values skew the result in favor of the first run.
Looking into the location of these extreme values, it seems that almost all of them appear specifically for entry 3722. Excluding all rows where the entry is 3722, the median remains unchanged, but the mean is now much more similar, differing only by about 20 ns. While the exclusion of this entry lowers the mean execution time for subsequent runs, it also leads to a slight reduction in the time observed for the first run. In *Section 6.1* (p. 23), the execution times for specific entries also appeared to be slow. While entry 3722 was not one of those slow entries there, these experiments were conducted with different auxiliary stores and entries. As such, the issue with why the first `RAuxStore` run is faster may be related to the issue discovered in that section.

6.4 SSD vs. HDD

To evaluate how the storage medium affects the performance of `RAuxStore` and `TAuxStore`, we conducted the previously performed benchmarks with the same files on the same machine on an HDD. The results from these experiments can be seen in *Appendix B* (p. 47). *Table 6.3* (p. 29) shows the results from one of these experiments performed on an HDD compared to the same experiment performed on an SSD. *Appendix C* (p. 49) presents additional results comparing experiments conducted on an HDD with those performed on an SSD.
In the experiments that were performed with a warmup phase as well as with multiple stores and a large number of entries — resulting in longer execution times — the auxiliary stores are in all but one case faster on an SSD compared to an HDD. The `RAuxStore` is more affected by this slowdown on the HDD than the `TAuxStore` is.
In *Table 6.3* (p. 29), in particular, the overall percentage decrease is 6.4% slower on the HDD for the `RAuxStore` and 1.1% slower for the `TAuxStore`, resulting in the `RAuxStore` being 5.3 percentage points slower than the `TAuxStore` on an HDD. The results from the experiments performed without the warmup phase — still with multiple stores and a large number of entries — are an exception to this trend of the HDD being slower. Here, the SSD is in most cases slower than the HDD for both auxiliary stores, particularly for the `TAuxStore`.
The results from the remaining experiments have been performed on a single store and thus result in overall lower execution times. It is unclear whether the auxiliary stores perform better on the SSD or the HDD, as their performance varies, being faster on the HDD in some cases and slower in others.
While the percentage difference might differ a lot, the absolute difference is rather low.

Benchmarking With a Warmup Phase

% of Auxiliary IDs	RAuxStore SSD	RAuxStore HDD	TAuxStore SSD	TAuxStore HDD	Percentage Decrease
10%	2445 ms	2612 ms	930 ms	947 ms	R: HDD 6.8% slower T: HDD 1.8% slower
40%	3918 ms	4223 ms	2009 ms	2026 ms	R: HDD 7.8% slower T: HDD 0.8% slower
75%	5906 ms	6266 ms	3317 ms	3350 ms	R: HDD 6.1% slower T: HDD 1.0% slower
100%	7407 ms	7841 ms	4218 ms	4269 ms	R: HDD 5.9% slower T: HDD 1.2% slower
Total Time	19,676 ms	20,942 ms	10,474 ms	10,592 ms	R: HDD 6.4% slower T: HDD 1.1% slower

Table 6.3: Performance comparison between `RAuxStore` (R) and `TAuxStore` (T) on an SSD and an HDD with a warmup phase. Created using *Table 5.1* (p. 18) and *Table B.1* (p. 47).

Thus, the difference does not seem significant and may simply be some overhead of the test.
Overall, the `RAuxStore` generally benefits more from using an SSD as a storage medium than the `TAuxStore` does. However, this trend is not entirely consistent, as in some cases, there are minimal differences between storage media. As all previous experiments in this thesis have been conducted on an SSD, we are essentially experimenting with the storage medium that benefits the `RAuxStore` the most. Despite this, the `TAuxStore` achieves better performance than the `RAuxStore`, suggesting that the performance is less dependent on the storage medium used.

6.5 Impact of Different Seeds

In the experiments performed, a seed is used to ensure randomness when choosing auxiliary stores, auxiliary IDs, and entries. We, therefore, want to test how different seeds affect the results. To this end, we have generated three random seeds between 1 and 1000 using the `std::mt19937` from C++. These three seeds are: 396, 570, and 721. As the seed determines which auxiliary stores we select, and some stores contain a single auxiliary ID, whereas others contain a multitude of them, the choice of seed can affect the overall

Seed	RAuxStore	TAuxStore	Difference	Performance Ratio
15	1776 ms	649 ms	1127 ms	2.74
396	1652 ms	557 ms	1095 ms	2.97
570	1747 ms	591 ms	1156 ms	2.96
721	1847 ms	633 ms	1214 ms	2.92
3	3053 ms	1418 ms	1635 ms	2.15

Table 6.4: Performance comparison between RAuxStore and TAuxStore across different seeds. The difference is calculated as RAuxStore time minus TAuxStore time. Performance ratio indicates how many times slower RAuxStore is compared to TAuxStore.

runtime of the benchmark. We, therefore, also run the benchmark with seed 3, where the benchmark with this seed contains the auxiliary store with the most auxiliary IDs, namely the store called AnalysisMuonsAux.

In this experiment, we compare the result of these seeds to a baseline result, obtained by executing the benchmark with seed 15; that is the seed used in previous experiments. This experiment was conducted with 10 stores, 10,000 entries, 100% of the auxiliary IDs, and 20 runs. The results from the experiment can be seen *Table 6.4* (p. 30).

We compare the value between the RAuxStore and the TAuxStore through the difference between these values as well as through a performance ratio of how slow the RAuxStore is compared to the TAuxStore, expecting these values to be similar across the selected seeds. Generally, the performance ratio is very similar for the results for the randomly picked seeds, differing only slightly from the result for the baseline seed. The difference also does not vary much, with the gap between the highest and lowest result being slightly more than 100 ms.

For the random seeds, the times for the RAuxStore and TAuxStore only vary slightly, where the TAuxStore results differ by less than 100 ms and the RAuxStore by less than around 200 ms.

For seed 3, this is not the case, as the execution times are noticeably higher compared to the execution times of the other seeds. However, the difference does not vary significantly compared to how much the actual times differ. Similarly, while the performance ratio for this result deviates from the others, the gap is not substantial. So, overall, the seed value has an impact on the results, but the relationship between the results for the RAuxStore and TAuxStore within a benchmark is generally similar.

	TTree Throughput	RNTuple Throughput
All entries	1316.73 entries/s	3007.73 entries/s
Slow entries	38.70 entries/s	4.08 entries/s
Random entries	34.08 entries/s	6.97 entries/s

Table 6.5: Throughput comparison between TTree and RNTuple for three tests. Each test is run in five separate executions.

6.6 RNTuple Throughput

In *Chapter 3* (p. 7), we saw that the throughput for reading an RNTuple should be significantly higher compared to reading a TTree. However, in our experimentation of the `RAuxStore`, it underperformed relative to the `TAuxStore`. For the RNTuple, certain entries appear to be particularly slow to read when using the `RAuxStore`. To check if these entries are only a problem for the `RAuxStore` or if the reading of the RNTuple itself is affected by them, we perform experiments on the throughput of the RNTuple on an SSD. Using ROOT's TStopwatch class, we time the throughput of the RNTuple and TTree in our dataset with 110,000 entries in three cases:

- Reading all 110,000 entries
- Reading the 8 slow entries discovered in *Section 6.1* (p. 23)
- Reading 8 random entries[2]

The results for this experiment can be seen in *Table 6.5* (p. 31).
When reading all entries of the event data, the RNTuple has double the throughput compared to TTree. When reading fewer entries, the throughput generally decreases for both TTree and RNTuple. Here, the RNTuple suffers the most, with throughput being several magnitudes lower when reading the slow entries or the random entries compared to when reading all entries. For the slow entries in particular, the RNTuple is almost 10 times slower than the TTree. Similarly, for the random entries, it is almost 5 times slower.
The RNTuple is slower at reading the *slow* entries, with its throughput being approximately 30% lower compared to when reading random entries. For the TTree, the throughput is higher with the *slow* entries compared to the random ones by around 12%.
This could help explain why the `RAuxStore` is affected by these specific entries, as the RNTuple itself also reads them more slowly.

[2]These entries were randomly generated with `std::mt19937` and with a seed of 15 and are: 93370, 84494, 19678, 65049, 5980, 12208, 39769, and 90048.

7 — Benchmark Improvements

7.1 Initial Improvements

As mentioned in *Section 6.2* (p. 25), we updated the benchmark code from its initial version to be more dynamic. Here, we altered the code to retrieve the store names dynamically with the `getStoreNames` function, which handled some issues, as explained in *Section 6.2* (p. 25). In addition to this, we made the benchmark code more intuitive by changing the way we handle entries.

Previously, we implemented an approximate 50/50 split of the entries entered into the benchmark. Thus, if a user entered n entries, around $\frac{n}{2}$ entries would be chosen. This approach was not very user-friendly, as the number of entries set by the user was not the actual number of entries used by the benchmark code. The actual number would also not be displayed to the user, and they could only assume that around half of the inputted entries would be used. This random selection was implemented using a random number generator and initialized with a fixed seed for reproducibility. Here, we iterated over the selected entries, and for each entry, we randomly assigned a value of 0 or 1. This value determines whether this entry should be included in the list of entries for which we execute the benchmark. Additionally, if the user entered more entries than were available in the file, errors would be thrown. After deciding on the entries to read, the warmup entries were selected such that they did not overlap with the benchmark entries. We used 1000 warmup entries, and if they could not be unique from the benchmark entries, the warmup phase would use all the remaining available entries. This meant that the warmup phase might not use the expected number of entries and could, if there were no unique entries available, even be executed without any entries.

The updated implementation now selects the exact number of entries requested by the user — up to the total number of entries available — providing a predictable selection process. If the user enters more entries than there are available in the file, the number of entries is set to the maximum number of entries available. These entries are selected randomly, shuffling the indices of the entries, selecting the first n entries from the shuffled list, and saving them to a new list. This new list is then sorted in ascending order, producing a final list of entries to read that mirrors our previous approach. In addition to selecting n random entries, we also added the option to select all entries up to the n^{th} entry.

The updated implementation now also verifies whether there are enough distinct entries available for the warmup phase. If not, it first uses all available distinct entries, then fills

the remaining slots by sequentially selecting entries from the beginning of the dataset.

7.2 Redesigned Benchmark

Based on the analysis of our previous benchmark implementation and the criteria outlined in *Section 5.2* (p. 21), we have redesigned our benchmark. The new version addresses some of the limitations identified in our original implementation while focusing on improving usability, verifiability, and reproducibility.
To this end, we introduce a dedicated configuration file. This allows users to set all options in one place instead of modifying the benchmark code, abstracting over the implementation details. The configuration file can be divided into five categories: flags, benchmarking parameters, store parameters, entry selection parameters, and filepaths. All configuration options and a description of them can be seen in *Table 7.1* (p. 34).

Flags

We have added several flags to the benchmark, allowing users to customize its behavior. This includes a `verbose` flag that, when enabled, causes warnings to be printed if any are encountered. Previously, it was not possible to execute the code without a warmup phase unless the user explicitly commented out code, as the warmup phase was part of the main function. We have moved the warmup code into its own function, and it is only executed if the user enables the `warmup` flag. To provide more precise information, we have introduced the `writeToCSV` flag, which means the duration of loading a specific entry is logged and the data is written to a CSV file. Similarly, we have added the `printIntermediateResults` flag. If this flag is enabled, the duration of each individual run within a test is shown to the user, and if some runs are significantly faster or slower than the median, the user is shown an info message. The `printWarmupResults` flag can be enabled, which results in the warmup times being printed in the same format as the benchmark times.
In our previous implementation, entries could only be selected randomly, but we have added the functionality to select entries sequentially, and this can be done by disabling the `selectEntriesRandomly` flag. If a user wants to perform the warmup phase with the same entries as in the benchmark phase, we have implemented this functionality, which can be toggled using the `warmupEntriesSameAsBenchmarkEntries` flag.

Benchmarking Parameters

To better control how the benchmark phase is performed, we have added several benchmarking parameters. The parameter `percentagesOfAuxIdsToPick` accepts a comma-separated list of percentages indicating the proportion of the auxiliary IDs to use in the benchmark.

Option	Description
Flags	
`verbose`	Warnings are printed.
`warmup`	A warmup phase is performed.
`writeToCSV`	Results for loading each entry are written to a CSV file.
`printIntermediateResults`	Shows duration for each benchmark run.
`printWarmupResults`	Shows the result for the warmup run.
`storeNameFiltering`	Filters out stores with problematic auxiliary IDs.
`selectEntriesRandomly`	Entries are selected randomly.
`warmupEntriesSameAs-BenchmarkEntries`	Set the warmup phase to use the same entries as the benchmark phase.
Benchmarking Parameters	
`nRuns`	Number of runs for each test.
`auxStoreToRun`	The store type to use (`RAuxStore`, `TAuxStore`, or Both).
`seed`	The seed used for randomness.
`percentagesOfAuxIdsToPick`	The different percentages of the auxiliary IDs to use for the benchmark tests.
Store Parameters	
`nStores`	Number of stores to use.
`minAuxids`	Minimum number of auxiliary IDs each store must have.
`storeNamesWithProblematic-Auxids`	Store names to exclude when filtering.
`storeNames`	Optional: list of specific stores to run with.
Entry Selection Parameters	
`nEntries`	Number of entries to use for the benchmark phase.
`nWarmupEntries`	Number of entries to use for the warmup phase.
File Paths	
`treeFilepath`	File path to the input TTree.
`ntupleFilepath`	File path to the input RNTuple.
`logDirPath`	Output directory for CSV files.

Table 7.1: Configuration parameters for the benchmark, classified by category.

Each specified percentage corresponds to a separate benchmark test, allowing performance to be evaluated across different auxiliary ID usage levels. The benchmark can be configured to perform with a specific auxiliary store type using the `auxStoreToRun` parameter, implemented as an enum, which configures the benchmark to use either the `RAuxStore` (0), the `TAuxStore` (1), or both (2). The `nRuns` parameter allows the user to specify the number of runs for each test. Finally, the user can set the `seed` parameter for the random number generator, ensuring deterministic behavior and reproducibility of the results.

Store Parameters

The store parameters allow the user to easily configure the auxiliary stores used in the benchmark. The `nStores` parameter specifies the number of stores to include. If a user requests more stores than there are available, the benchmark will use all the stores in the file, and, if the `verbose` flag is enabled, print a warning informing the user of this. When testing with different percentages of the auxiliary IDs, one might expect different auxiliary IDs to be picked. However, some of the stores contain only a single auxiliary ID, and no matter the percentages picked, this will be the only auxiliary ID used for that store by the benchmark. As such, the user can specify the minimum number of auxiliary IDs a store must contain using the `minAuxids` parameter. When `verbose` is enabled, a warning will also be printed if this value exceeds the number of IDs present in the store containing the highest number of IDs.
As mentioned in *Section 6.2* (p. 25), there is a problematic type that takes a long time to read the first time it is encountered. A majority of the auxiliary stores in the dataset contain auxiliary IDs of this problematic type, meaning there is a good chance that a test will encounter this type. To allow the user to perform the benchmark phase without that type, we have implemented a function that filters out the store names with IDs of that type. Here we have identified that store names beginning with *AnalysisTrigMatch* contain them. Additionally, depending on the files used, other stores may contain them as well. Here, we have added the `storeNamesWithProblematicAuxids` parameter, where the user can specify a list of the remaining store names that contain auxiliary IDs of this problematic type.
Instead of specifying the number of stores and the minimum number of auxiliary IDs contained in the stores, the user can specify specific names of stores to use in the benchmark with the optional `storeNames` parameter. This will override the number of stores specified by the `nStores` parameter and disregard the value assigned to the `minAuxIds`, issuing a warning to the user if either of these parameters is ignored.

Entry Selection Parameters

To make the entry selection more straightforward and intuitive, two similar parameters, `nEntries` and `nEntriesForWarmup`, have been added. This allows the user to specify the number of entries to be used in the benchmark phase and the warmup phase, respectively. If the user requests more entries than are available, the benchmark will use all available entries, and, if the `verbose` flag is enabled, display a warning. Unless the `warmupEntriesSameAsBenchmarkEntries` flag is enabled, the warmup entries should be distinct, but if there are not enough distinct entries, the remaining entries are selected. If this is the case, and if the `verbose` flag is enabled, a warning will inform the user that not enough distinct entries for the warmup are available, along with the number of non-distinct entries to use.

File paths

To enhance configurability and portability, all file path settings have been moved into the configuration file, centralizing them in a single location. For the benchmark, we use two files, one for the TTree (`treeFilePath`) and one for the RNTuple (`ntupleFilePath`). Additionally, a file path must be specified using the `logDirPath`, which is the directory where the CSV files are saved when the `writeToCSV` flag is enabled. If the user specifies a directory that does not exist, an option will be given to create that directory.

8 — Discussion

In this paper, we set out to identify the performance issues of the RAuxStore that we initially discovered in our previous work [28], and, if possible, handle these issues. These issues were discovered with a benchmark that compared the performance of the RAuxStore to that of the TAuxStore. While this benchmark did help validate these performance improvements, we set out to make a well-defined benchmark that followed standardization for benchmarks and made experimentation easier.

Extensive experimentation has allowed us to better identify the issues in the RAuxStore. In our previous work, we concluded that without a warmup phase, the RAuxStore was faster than the TAuxStore. However, we found that this conclusion was based on a flawed setup of the benchmark experiment, where we performed only a single run of the application multiple times. This made the TAuxStore appear slower due to an issue appearing the first time it encountered a specific type. We also discovered various issues regarding specific entries of the event data, which were present for the RAuxStore but not for the TAuxStore. In addition to timing the entire execution of our benchmark, we also performed a kernel benchmark on a particular part of the code for the auxiliary stores, namely, when an entry was loaded for a specific field/branch. Here, we saw that some durations for the RAuxStore were much slower than the median, by a factor of up to 1000, majorly affecting the performance. These slow durations appeared at regular intervals within different files, though the specific interval differed from file to file. Similarly, when performing multiple runs in a benchmark test, specific entries containing durations with high values in later runs were not as prevalent in the first run.

We also performed some experiments that had less of an impact on the issues we were looking at. We found that while the performance for both RAuxStore and TAuxStore was generally slightly better on an SSD, the storage medium did not have much impact on the overall performance. Similarly, we experimented with different seed values, which, though impacting the overall results, favored neither the RAuxStore nor the TAuxStore.

Overall, we identified some issues, mainly regarding specific entries of the event data. Though we discovered the existence of these issues, we did not find the cause for why they were so slow. One issue we discovered and afterwards dealt with was the fact that the RAuxStore did not set the stores up correctly and was dependent on the TAuxStore populating a singleton shared between the stores with the correct data. Solving this issue by altering the RAuxStore, however, created a new issue similar to the issue that the TAuxStore had with the first time it encountered a specific type.

In addition to updating the RAuxStore, we also updated the benchmark code itself to

better align with the theories discussed in this thesis. We improved the usability of the benchmark by handling the entries in a more intuitive way. Additionally, we improved reproducibility by adding various functions that set up elements used in the benchmark. Previously, the functionality of these functions was either not implemented or implemented in arbitrary places throughout the code, which may not be run in certain experiments. Doing this, thus, also improved the overall usability of the benchmark. To support verifiability, our main change is the addition of a configuration file, separating the configuration of the benchmark from the implementation. Additionally, we have implemented functional verification to support verifiability by alerting the user if a given run deviates significantly from the other runs.

We have also followed the recommendations from [31] as outlined in *Section 3.2* (p. 7). We believe that this has made our benchmark both interpretable and reproducible for other researchers. To ensure interpretability, we clearly describe our measurement techniques and report all results, including those that did not align with our expectations. Furthermore, reproducibility is supported by providing information about the hardware and software environment, along with a configuration file, enabling other researchers to replicate and verify our findings.

We did not create an application benchmark and instead only used a synthetic benchmark and a kernel benchmark. Creating an application benchmark where more of Athena's software is included is an interesting experiment to do, and it could tell us if the `RAuxStore` being slower than the `TAuxStore` is an actual issue for the ATLAS workflow. One way in which it would not be an issue is if the `RAuxStore` is rarely used in physics analysis, and therefore, the performance of the `RAuxStore` being slower than the `TAuxStore` is not worth the resources it takes to optimize it.

Evaluating the real-world impact is complicated by the nature of the Athena codebase itself. Athena is a huge project with a lot of active contributors changing the code daily, which meant that to make the benchmarks in this thesis, we had to make a snapshot of the code at a specific point in time. Since creating our snapshot, there have been more than 3500 commits to the Athena repository. The constant change in the codebase has also affected the `RAuxStore`, and major changes have been made since we created our snapshot. This means that the benchmarks we have performed in this thesis are not representative of the current state of the `RAuxStore`, and because of the change, we cannot just execute the benchmarks on the current `RAuxStore` to see if the performance has improved. To be able to test the `RAuxStore`, we would need to port our benchmark code to the current state of the `RAuxStore`, which would require major changes to our benchmark code and is out of the scope of this thesis.

9 — Conclusion

Our problem statement is as follows:

> *How can we identify bottlenecks and improve the performance of the* `RAuxStore`*, and how can we validate these improvements through a well-defined benchmark?*

In conclusion, this thesis identified multiple bottlenecks in the `RAuxStore`, primarily occurring from specific entries within the event data. We have identified these bottlenecks through the use of a synthetic benchmark as well as a kernel benchmark. Our main contribution is the development of the synthetic benchmark, which we refined by applying established benchmarking theory to improve upon the version from our previous project. We have thus ensured that the benchmark aligns better with this theory by making it verifiable, usable, and reproducible. Furthermore, we have made it interpretable such that other researchers can draw their own conclusions from our results.
However, as Athena is undergoing constant change, and the `RAuxStore` and the `TAuxStore` have had major changes since we started working on this thesis, there is a possibility that the problems we have identified no longer exist.

10 — Future Work

This chapter looks at elements of the thesis that should be addressed if more time were available. This includes the introduction of a new benchmark designed to more accurately reflect the actual ATLAS workflow. It also includes refactoring the benchmarks to use the updated `RAuxStore` in Athena rather than our version.

10.1 Application Benchmark

At the moment, we have a synthetic benchmark that simulates the behavior of the ATLAS workflow. While this provides valuable insight into the behavior of the implementation, it may not capture the complexity of the actual workflow. As such, an application benchmark could be made. Implementing this would require consultation with experts at ATLAS to ensure that the benchmark aligns with typical use cases.

10.2 Refactoring Our Benchmark

The version of `RAuxStore` that we experiment on in this thesis is based on a snapshot taken at a specific point in time of what the code looked like. Since then, significant changes have been made to both the `RAuxStore` as well as the `TAuxStore`. This means that the circumstances regarding the performance issues may not be present in this updated version. However, since we have only performed experiments on the snapshot we have taken of the auxiliary stores, this is not something we can confirm until experiments on the new implementation of the auxiliary stores have been performed. Thus, to ensure that the benchmark results we have achieved remain relevant, the benchmarking code should be updated to support the latest implementations of the auxiliary stores.

Bibliography

[1] V. Ziemann, *Beams*. Springer Cham, 2018, ISBN: 978-3-031-51852-2. DOI: 10.1007/978-3-031-51852-2. [Online]. Available: https://doi.org/10.1007/978-3-031-51852-2.

[2] H. Wiedemann, *Particle accelerator physics*. Springer Nature, 2015. DOI: 10.1007/978-3-319-18317-6. [Online]. Available: doi.org/10.1007/978-3-319-18317-6.

[3] E. M. McMillan, "A history of the synchrotron," *Physics Today*, vol. 37, no. 2, pp. 31–37, Feb. 1984, ISSN: 0031-9228. DOI: 10.1063/1.2916080. eprint: https://pubs.aip.org/physicstoday/article-pdf/37/2/31/8293379/31_1_online.pdf. [Online]. Available: https://doi.org/10.1063/1.2916080.

[4] O. Brüning, M. Klein, S. Myers, and L. Rossi, "70 years at the high-energy frontier with the cern accelerator complex," *Nature Reviews Physics*, vol. 6, no. 10, pp. 628–637, Oct. 2024, ISSN: 2522-5820. DOI: 10.1038/s42254-024-00758-5. [Online]. Available: https://doi.org/10.1038/s42254-024-00758-5.

[5] E. Lopienska, "The CERN accelerator complex, layout in 2022. Complexe des accélérateurs du CERN en janvier 2022," 2022, General Photo. [Online]. Available: https://cds.cern.ch/record/2800984.

[6] J. Resta-López, *Long-term future particle accelerators*, 2022. arXiv: 2206.08834 [physics.acc-ph]. [Online]. Available: https://arxiv.org/abs/2206.08834.

[7] C. L. Smith, "The large hadron collider," *Scientific American*, vol. 283, no. 1, pp. 70–77, 2000, ISSN: 00368733, 19467087. [Online]. Available: http://www.jstor.org/stable/26058791 (visited on 03/25/2025).

[8] L. Tavian, "Latest Developments in Cryogenics at CERN," CERN, Geneva, Tech. Rep., 2005. [Online]. Available: https://cds.cern.ch/record/851586.

[9] M. Boisot, "Generating knowledge in a connected world: The case of the atlas experiment at cern," *Management Learning*, vol. 42, no. 4, pp. 447–457, 2011. DOI: 10.1177/1350507611408676. [Online]. Available: https://doi.org/10.1177/1350507611408676.

[10] G. Stewart and W. Lampl, "How to review 4 million lines of ATLAS code," CERN, Geneva, Tech. Rep. 7, 2017. DOI: 10.1088/1742-6596/898/7/072013. [Online]. Available: https://cds.cern.ch/record/2248494.

[11] R. Bruce *et al.*, "First results of running the lhc with lead ions at a beam energy of 6.8 z tev," *Journal of Physics: Conference Series*, vol. 2687, no. 2, p. 022 001, Jan. 2024. DOI: 10.1088/1742-6596/2687/2/022001. [Online]. Available: https://dx.doi.org/10.1088/1742-6596/2687/2/022001.

[12] M. Lamont, *Operational Experience from LHC Run 1 & 2 and Consolidation in View of Run 3 and the HL-LHC*. World Scientific, 2024. DOI: 10.1142/9789811278952_0004. [Online]. Available: https://www.worldscientific.com/doi/abs/10.1142/9789811278952_0004.

[13] A. Borga *et al.*, "The atlas readout system for lhc runs 2 and 3," *Journal of Instrumentation*, vol. 18, no. 08, P08022, Aug. 2023. DOI: 10.1088/1748-0221/18/08/P08022. [Online]. Available: https://dx.doi.org/10.1088/1748-0221/18/08/P08022.

[14] R. Tomás *et al.*, "Operational scenario of first high luminosity lhc run," *Journal of Physics: Conference Series*, vol. 2420, no. 1, p. 012 003, Jan. 2023. DOI: 10.1088/1742-6596/2420/1/012003. [Online]. Available: https://dx.doi.org/10.1088/1742-6596/2420/1/012003.

[15] The ATLAS Collaboration and G. Aad, "The atlas experiment at the cern large hadron collider," *Journal of Instrumentation*, vol. 3, no. 08, S08003, Aug. 2008. DOI: 10.1088/1748-0221/3/08/S08003. [Online]. Available: https://dx.doi.org/10.1088/1748-0221/3/08/S08003.

[16] T. C. Collaboration and S. Chatrchyan, "The cms experiment at the cern lhc," *Journal of Instrumentation*, vol. 3, no. 08, S08004, Aug. 2008. DOI: 10.1088/1748-0221/3/08/S08004. [Online]. Available: https://dx.doi.org/10.1088/1748-0221/3/08/S08004.

[17] R. Brun and F. Rademakers, "Root — an object oriented data analysis framework," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 389, no. 1, pp. 81–86, 1997, New Computing Techniques in Physics Research V, ISSN: 0168-9002. DOI: https://doi.org/10.1016/S0168-9002(97)00048-X. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016890029700048X.

[18] Blomer, Jakob, Canal, Philippe, Naumann, Axel, and Piparo, Danilo, "Evolution of the root tree i/o," *EPJ Web Conf.*, vol. 245, p. 02 030, 2020. DOI: 10.1051/epjconf/202024502030. [Online]. Available: https://doi.org/10.1051/epjconf/202024502030.

[19] López-Gómez, Javier and Blomer, Jakob, "Exploring object stores for high-energy physics data storage," *EPJ Web Conf.*, vol. 251, p. 02 066, 2021. DOI: 10.1051/epjconf/202125102066. [Online]. Available: https://doi.org/10.1051/epjconf/202125102066.

[20] I. Bird, "Computing for the large hadron collider," *Annual Review of Nuclear and Particle Science*, vol. 61, no. Volume 61, 2011, pp. 99–118, 2011, ISSN: 1545-4134. DOI: https://doi.org/10.1146/annurev-nucl-102010-130059. [Online]. Available: https://www.annualreviews.org/content/journals/10.1146/annurev-nucl-102010-130059.

[21] EOS, *Eos description*, 2022. [Online]. Available: https://cernbox.cern.ch/pdf-viewer/public/Kl0hxpeA5bFQ4Ho/publications/eos-description-2022.pdf.

[22] Adye, T *et al.*, "Xrootd third party copy for the wlcg and hllhc," *EPJ Web Conf.*, vol. 245, p. 04 034, 2020. DOI: 10.1051/epjconf/202024504034. [Online]. Available: https://doi.org/10.1051/epjconf/202024504034.

[23] T. Mkrtchyan *et al.*, "Dcache integration with cern tape archive," *EPJ Web of Conferences*, vol. 295, May 2024. DOI: 10.1051/epjconf/202429501016.

[24] M. C. Davis, V. Bahyl, G. Cancio, E. Cano, J. Leduc, and S. Murray, "CERN Tape Archive - from development to production deployment," *EPJ Web Conf.*, vol. 214, A. Forti, L. Betev, M. Litmaath, O. Smirnova, and P. Hristov, Eds., p. 04 015, 2019. DOI: 10.1051/epjconf/201921404015.

[25] A. Naumann *et al.*, *Root for the hl-lhc: Data format*, 2022. arXiv: 2204.04557 [hep-ex]. [Online]. Available: https://arxiv.org/abs/2204.04557.

[26] A. S. Mete, M. Nowak, and P. Van Gemmeren, "Persistifying the complex event data model of the ATLAS Experiment in RNTuple," CERN, Geneva, Tech. Rep., 2024. [Online]. Available: https://cds.cern.ch/record/2905189.

[27] T. Åkesson *et al.*, "Atlas computing: Technical design report," CERN, Tech. Rep. ATLAS TDR-017; CERN-LHCC-2005-022, 2005. [Online]. Available: https://lup.lub.lu.se/search/files/5842745/941311.

[28] M. L. Hansen and N. K. Krogh, *Further work on rntuples in atlas*, Dec. 2024. [Online]. Available: https://kbdk-aub.primo.exlibrisgroup.com/permalink/45KBDK_AUB/a7me0f/alma9921964558505762.

[29] J. Lopez-Gomez and J. Blomer, "Rntuple performance: Status and outlook," *Journal of Physics: Conference Series*, vol. 2438, no. 1, p. 012 118, Feb. 2023. DOI: 10.1088/1742-6596/2438/1/012118. [Online]. Available: https://dx.doi.org/10.1088/1742-6596/2438/1/012118.

[30] Blomer, Jakob *et al.*, "Root's rntuple i/o subsystem: The path to production," *EPJ Web of Conf.*, vol. 295, p. 06 020, 2024. DOI: 10.1051/epjconf/202429506020. [Online]. Available: https://doi.org/10.1051/epjconf/202429506020.

[31] T. Hoefler and R. Belli, "Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15, Austin, Texas: Association for Computing Machinery, 2015, ISBN: 9781450337236. DOI: 10.1145/2807591.2807644. [Online]. Available: https://doi.org/10.1145/2807591.2807644.

[32] D. Bailey *et al.*, "Parkbench report - 1: Public international benchmarks for parallel computers," *Scientific Programming*, vol. 3, no. 2, pp. 101–146, 1994, ISSN: 1058-9244.

[33] T. R. team, *ROOT: an object-oriented data analysis framework: users guide 4.04*. Geneva: CERN, 2005.

[34] I. Antcheva *et al.*, "Root — a c++ framework for petabyte data storage, statistical analysis and visualization," *Computer Physics Communications*, vol. 180, no. 12, pp. 2499–2512, 2009, 40 YEARS OF CPC: A celebratory issue focused on quality software for high performance, grid and novel computing architectures, ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2009.08.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010465509002550.

[35] A. Naumann *et al.*, *Root for the hl-lhc: Data format*, 2022. arXiv: 2204.04557 [hep-ex]. [Online]. Available: https://arxiv.org/abs/2204.04557.

[36] de Geus, Florine, López-Gómez, Javier, Blomer, Jakob, Nowak, Marcin, and van Gemmeren, Peter, "Integration of rntuple in atlas athena," *EPJ Web of Conferences*, vol. 295, p. 06 013, May 2024. DOI: 10.1051/epjconf/202429506013. [Online]. Available: https://doi.org/10.1051/epjconf/202429506013.

[37] P. van Gemmeren and D. Malon, "The event data store and i/o framework for the atlas experiment at the large hadron collider," in *2009 IEEE International Conference on Cluster Computing and Workshops*, 2009, pp. 1–8. DOI: 10.1109/CLUSTR.2009.5289147.

[38] H. Velthuijsen, "The nature and applicability of the blackboard architecture," English, Ph.D. dissertation, Maastricht University, Jan. 1992, ch. 1, ISBN: 9072125347. DOI: 10.26481/dis.19920924hv.

[39] A. Buckley *et al.*, "Implementation of the atlas run 2 event data model," *Journal of Physics: Conference Series*, vol. 664, no. 7, p. 072 045, Dec. 2015. DOI: 10.1088/1742-6596/664/7/072045. [Online]. Available: https://dx.doi.org/10.1088/1742-6596/664/7/072045.

[40] K. Kanoun and L. Spainhower, *Dependability Benchmarking for Computer Systems*. Wiley-IEEE Computer Society Pr, 2008, ISBN: 978-0-470-23055-8.

[41] S. Kounev, K.-D. Lange, and J. von Kistowski, *Systems Benchmarking: For Scientists and Engineers*, 1st ed. Springer Cham, 2020, pp. 3–21, 131–147, ISBN: 978-3-030-41705-5. DOI: 10.1007/978-3-030-41705-5. [Online]. Available: https://doi.org/10.1007/978-3-030-41705-5.

[42] J. v. Kistowski, J. A. Arnold, K. Huppler, K.-D. Lange, J. L. Henning, and P. Cao, "How to build a benchmark," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '15, Austin, Texas, USA: Association for Computing Machinery, 2015, pp. 333–336, ISBN: 978-1-450-33248-4. DOI: 10.1145/2668930.2688819. [Online]. Available: https://doi.org/10.1145/2668930.2688819.

[43] D. B. Stewart, "Measuring execution time and real-time performance," *Embedded Systems Conference*, Sep. 2006. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=e255041e179f96a46f772c7959e381710a6a5a94.

[44] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, no. 85, pp. 2825–2830, 2011. [Online]. Available: http://jmlr.org/papers/v12/pedregosa11a.html.

[45] E. Schubert, J. Sander, M. Ester, H. P. Kriegel, and X. Xu, "Dbscan revisited, revisited: Why and how you should (still) use dbscan," *ACM Trans. Database Syst.*, vol. 42, no. 3, Jul. 2017, ISSN: 0362-5915. DOI: 10.1145/3068335. [Online]. Available: https://doi.org/10.1145/3068335.

A — Performance Measuring

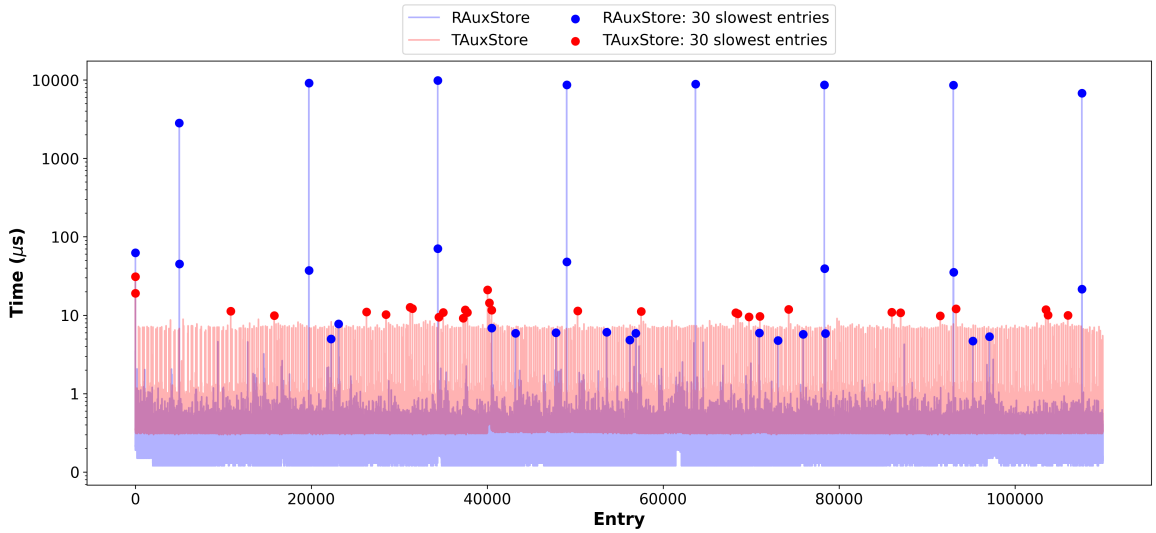


Figure A.1: The time taken to load an entry for a field for both the `RAuxStore` and `TAuxStore` for individual event data entries. The plot uses a dataset with 110,000 entries.

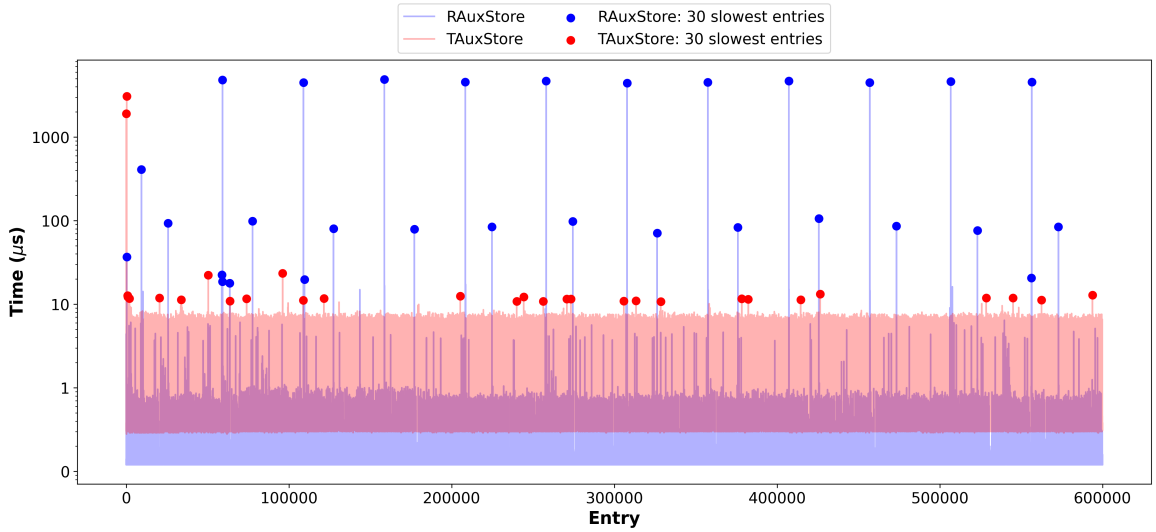


Figure A.2: The time taken to load an entry for a field for both the `RAuxStore` and `TAuxStore` for individual event data entries. The plot uses a dataset with 600,000 entries.

B — Benchmark Tables on HDD

Benchmarking With a Warmup Phase

% of Auxiliary IDs	`RAuxStore`	`TAuxStore`	Difference
10%	2612 ms	947 ms	1665 ms
40%	4223 ms	2026 ms	2197 ms
75%	6266 ms	3350 ms	2916 ms
100%	7841 ms	4269 ms	3572 ms
Total Time	20,942 ms	10,592 ms	10,350 ms

Table B.1: Benchmarking tests comparing the throughput of the `RAuxStore` compared to the `TAuxStore`. Each test performs 20 runs.

Benchmarking Without a Warmup Phase

% of Auxiliary IDs	`RAuxStore`	`TAuxStore`	Difference
10%	1081 ms	914 ms	167 ms
40%	2584 ms	3829 ms	1245 ms
75%	4607 ms	5163 ms	556 ms
100%	6166 ms	6100 ms	66 ms
Total Time	14,438 ms	16,006 ms	1568 ms

Table B.2: Benchmarking tests comparing the throughput of the `RAuxStore` compared to the `TAuxStore`. Each test is run 20 times in separate executions.

#	Test Configuration	RAuxStore	TAuxStore
Scenario 1: 20 Stores Without Problematic Type			
1	No Warmup Run	6585 ms	3345 ms
Scenario 2: 1 Store With Problematic Type			
2	No Warmup Run	137 ms	2009 ms
3	With Warmup Run	135 ms	106 ms
4	No Warmup Run (Avg. of 20 Runs)	217 ms	200 ms

Table B.3: Benchmark comparison of `RAuxStore` and `TAuxStore` performance across different test configurations (with around 20,000 entries and 100% of the auxiliary IDs within the auxiliary stores are used). *Problematic Type* refers to the type `std::vector<std::vector<ElementLink<DataVector<xAOD::IParticle >>>>>`. For all configurations, we perform 20 separate executions and report the average execution time. The fourth configuration additionally performs 20 runs within a single test and calculates the average time across all runs. For scenario one, all stores containing IDs with the problematic type are filtered out, and 20 stores are selected from the remaining stores. For scenario two, a store with the name: `AntiKt10TruthSoftDropBeta100Zcut10JetsAux` is picked. This store contains 10 auxiliary IDs, and one of them is of the problematic type.

Dataset Size	RAuxStore	TAuxStore
110,000 entries	383 ms	92 ms
150,000 entries	627 ms	124 ms
600,000 entries	719 ms	485 ms

Table B.4: Results from conducting the benchmark on datasets of varying sizes.

Test Configuration	Run 1	Run 2-20 Avg.	Overall Avg.
`RAuxStore` With a Warmup Phase	6952 ms	7979 ms	7927 ms
`RAuxStore` Without a Warmup Phase	8969 ms	7972 ms	8022 ms
`TAuxStore` Without a Warmup Phase	4338 ms	4266 ms	4269 ms
`TAuxStore` Without a Warmup Phase	5612 ms	4170 ms	4242 ms

Table B.5: Benchmark results comparing execution times for `RAuxStore` and `TAuxStore` with and without a warmup phase, after modifications done to the `RAuxStore`.

C — Comparison Tables

Benchmarking Without a Warmup Phase

% of Auxiliary IDs	`RAuxStore` SSD	`RAuxStore` HDD	`TAuxStore` SSD	`TAuxStore` HDD	Percentage Decrease
10%	1134 ms	1081 ms	916 ms	914 ms	R: SSD 4.9% slower T: SSD 0.2% slower
40%	2525 ms	2584 ms	3862 ms	3829 ms	R: HDD 2.3% slower T: SSD 0.9% slower
75%	4654 ms	4607 ms	5178 ms	5163 ms	R: SSD 1.0% slower T: SSD 0.3% slower
100%	6030 ms	6166 ms	6119 ms	6100 ms	R: HDD 2.3% slower T: SSD 0.3% slower
Total Time	14,343 ms	14,438 ms	16,075 ms	16,006 ms	R: HDD 0.7% slower T: SSD 0.4% slower

Table C.1: Performance comparison between `RAuxStore` (R) and `TAuxStore` (T) on an SSD and an HDD without a warmup phase. Created using *Table 5.2* (p. 19) and *Table B.2* (p. 47).

#	Test Configuration	RAuxStore SSD	RAuxStore HDD	TAuxStore SSD	TAuxStore HDD	Percentage Decrease
Scenario 1: 20 Stores Without Problematic Type						
1	No Warmup Phase	6060 ms	6585 ms	3278 ms	3345 ms	R: HDD 8.7% slower T: HDD 2.0% slower
Scenario 2: 1 Store With Problematic Type						
2	No Warmup Phase	140 ms	137 ms	2021 ms	2009 ms	R: SSD 2.2% slower T: SSD 0.6% slower
3	With Warmup Phase	139 ms	135 ms	129 ms	106 ms	R: SSD 3.0% slower T: SSD 21.7% slower
4	No Warmup Phase (Avg. of 20 runs)	211 ms	217 ms	224 ms	200 ms	R: HDD 2.8% slower T: SSD 12.0% slower

Table C.2: Performance comparison between `RAuxStore` (R) and `TAuxStore` (T) on SSD and HDD. Created using *Table 5.3* (p. 20) and *Table B.3* (p. 48).

Dataset Size	SSD Total Time	HDD Total Time	Difference	Percentage Decrease
`RAuxStore`				
110.000 entries	359 ms	383 ms	24 ms	HDD 6.7% slower
150.000 entries	639 ms	627 ms	12 ms	SSD 1.9% slower
600.000 entries	718 ms	719 ms	1 ms	HDD 0.1% slower
`TAuxStore`				
110.000 entries	90 ms	92 ms	2 ms	HDD 2.2% slower
150.000 entries	131 ms	124 ms	7 ms	SSD 5.7% slower
600.000 entries	485 ms	485 ms	0 ms	No difference

Table C.3: Performance comparison between an SSD and an HDD with varying dataset sizes. All experiments are performed using a single auxiliary store. Created using *Table 6.1* (p. 24) and *Table B.4* (p. 48).

Configuration	SSD Overall Avg.	HDD Overall Avg.	Difference	Percentage Decrease
`RAuxStore` With Warmup	7728 ms	7927 ms	199 ms	HDD 2.6% slower
`RAuxStore` Without Warmup	7968 ms	8022 ms	54 ms	HDD 0.7% slower
`TAuxStore` With Warmup	4245 ms	4269 ms	24 ms	HDD 0.6% slower
`TAuxStore` Without Warmup	4262 ms	4242 ms	20 ms	SSD 0.5% slower

Table C.4: Benchmark results comparing an SSD and an HDD execution times for `RAuxStore` and `TAuxStore` with and without warmup, after modifications done to the `RAuxStore`. Created using *Table 6.2* (p. 27) and *Table B.5* (p. 48).