ABSTRACT

OPTIMIZATION OF DERIVATION JOBS AND MODERNIZATION OF NIGHTLY CI BUILD I/O TESTS FOR THE ATLAS EXPERIMENT

Arthur C. Kraus, M.S.
Department of Physics
Northern Illinois University, 2025
Dr. Jahred Adelman, Director

High-Luminosity LHC (HL-LHC) is a phase of the LHC that is expected to run toward the end of the decade. With this comes an increase in data taken per year that current software and computing infrastructure, including I/O, is not prepared to handle. The ATLAS experiment's Software Performance Optimization Team has efforts in developing the Athena software framework that is scalable in performance and ready for wide-spread use during Run-3 and HL-LHC data ready to be used for Run-4. It's been shown that the storage bias for TTree's during derivation production jobs can be improved upon compression and stored to disk by about 4-5% by eliminating the basket capping, with a simultaneous increase in memory usage by about 11%. Additionally, job configuration allows opportunity to improve many facets of the ATLAS I/O framework.

Athena and software it depends on are updated frequently, and to synthesize changes cohesively there are scripts, unit tests, that run which test core I/O functionality. This thesis also addresses a project to add a handful of I/O unit tests that exercise features exclusive to the new xAOD Event Data Model (EDM) such as writing and reading object data from the previous EDM using transient and persistent data. These new unit tests also include and omit select dynamic attributes to object data.

NORTHERN ILLINOIS UNIVERSITY
DE KALB, ILLINOIS

MAY 2025

OPTIMIZATION OF DERIVATION JOBS AND MODERNIZATION OF NIGHTLY CI BUILD I/O TESTS FOR THE ATLAS EXPERIMENT

BY

ARTHUR C. KRAUS

A THESIS SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE
MASTER OF SCIENCE

DEPARTMENT OF PHYSICS

Thesis Director:
Dr. Jahred Adelman

ACKNOWLEDGEMENTS

Here's where you acknowledge folks who helped. Here's where you acknowledge folks who helped. Here's where you acknowledge folks who helped. Here's where you acknowledge folks who helped.

DEDICATION

28

To all of the fluffy kitties. To all of the fluffy kitties. To all of the fluffy kitties. To all of the fluffy kitties.

TABLE OF CONTENTS

Chapter | Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Particle Physics and the Large Hadron Collider

Particle physics is the branch of physics that explores the fundamental constituents of matter and the forces governing their interactions. The field started as studies in electromagnetism, radiation, and further developed with the discovery of the electron. What followed was more experiments to search for new particles, new models to describe the results, and new search techniques which demanded more data. The balance in resources for an experiment bottlenecks how much data can be taken, so steps need to be taken to identify interesting interactions and optimize the storage and processing of this data. This thesis investigates software performance optimization of the ATLAS experiment at CERN. Specifically, ways to modernize and optimize areas of the software framework, Athena, to improve input/output (I/O) during derivation production and create new tests that catch when specific core I/O functionality is broken.

1.2 LHC and The ATLAS Detector

The Large Hadron Collider (LHC), shown in Figure 1.1, is a particle accelerator spanning a 26.7-kilometer ring that crosses between the France-Switzerland border at a depth between 50 and 175 meters underground.[14] The ATLAS experiment, shown in Figure 1.2, is the largest LHC general purpose detector, and the largest detector ever made for particle collision experiments. It's 46 meters long, 25 meters high and 25 meters wide.[16] The ATLAS

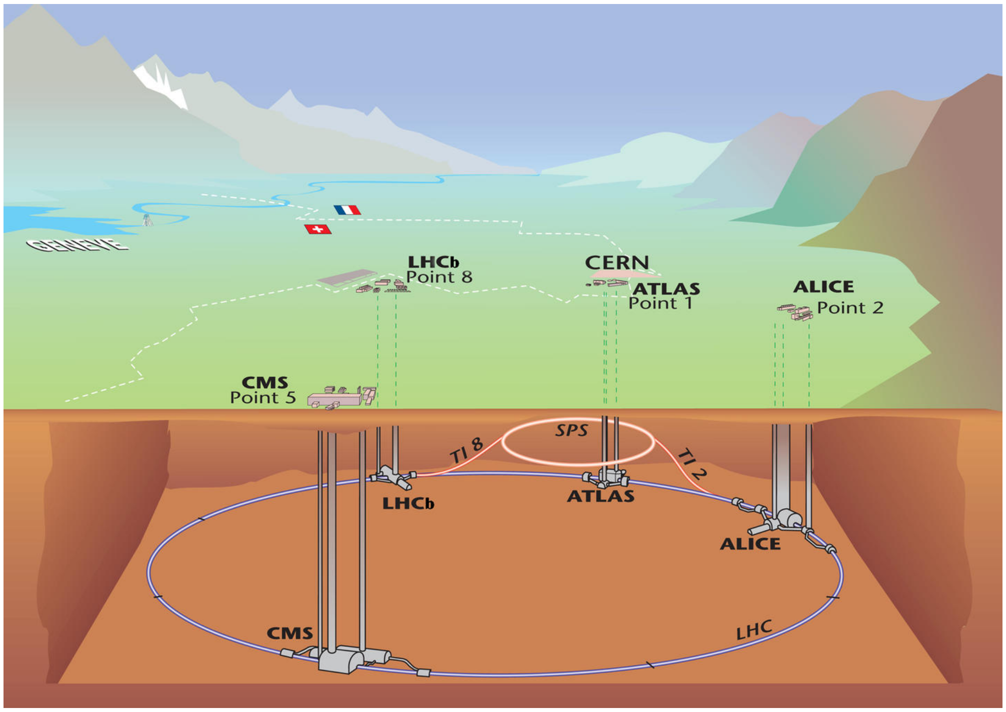

Figure 1.1: Illustration of the LHC experiment sites on the France-Switzerland border. [7]

detector is comprised of three main sections, the inner detector, calorimeters and the muon detector system.

The inner detector measures the direction, momentum and charge of electrically charged particles. Its main function is to measure the track of the charged particles without destroying the particle itself. The first point of contact for particles emerging from *pp*-collisions from the center of the ATLAS detector is the pixel detector.[1] It has over 92 million pixels and is radiation hard to aid in particle track and vertex reconstruction. When charged particles pass through a pixel sensor, it ionizes the one-sided doped-silicon wafer to produce an excited electron will then occupy the conduction band of the semiconductor producing an electron-hole pair, leaving the valence band empty.[12] This hole in the valence band together with the excited electron in the conduction band is called an electron-hole pair. The electron-hole pair is in the presence of an electric field, which will induce drifting of the electron-hole pair, drifting that will generate the electric current to be measured.

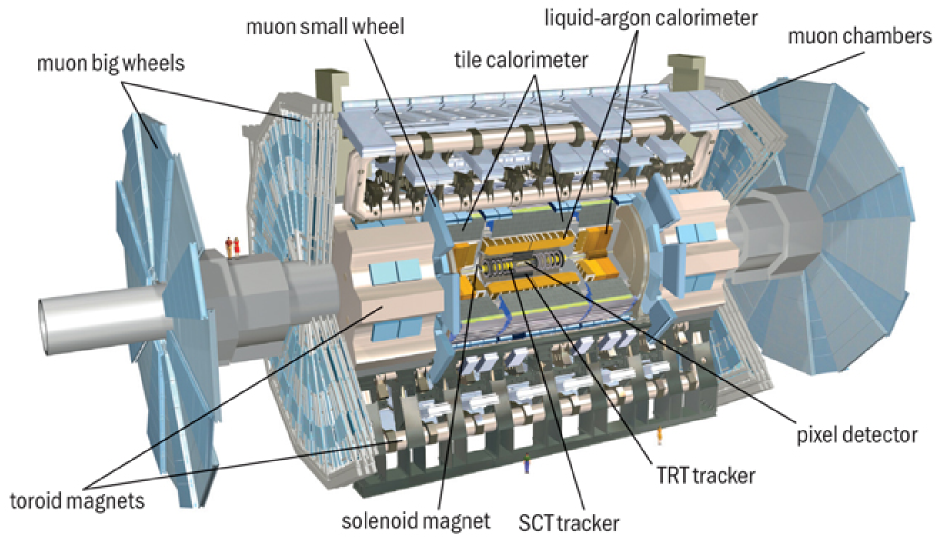

Figure 1.2: Overview of the ATLAS detectors main components. [9]

Surrounding the pixel detector is the SemiConductor Tracker (SCT), which uses 4,088 modules of 6 million implanted silicon readout strips.[2] Both the pixel detector and SCT measure the path particles take, called tracks. While the pixel detector has measurement precision up to $10\mu m$, the SCT has precision up to $25\mu m$.

The final layer of the inner detector is the transition radiation tracker (TRT). The TRT is made of a collection of tubes made with many layers of different materials with varying indices of refraction. The TRT's straw walls are made of two $35\mu m$ layers comprised of $6\mu m$ carbon-polymide, $0.20\mu m$ aluminum, and a $25\mu m$ Kapton film reflected back.[8] The straws are filled with a gas mixture of $70\%\mathrm{Xe} + 27\%\mathrm{CO_2} + 3\%\mathrm{O_2}$. Its measurement precision is around $170\mu m$. Particles with relativistic velocities have higher Lorentz γ-factors (see Eq. (1.1)). The TRT uses varying materials to discriminate between heavier particles, which have low γ and radiate less, and lighter particles, which have higher γ and radiate more. [15]

$$\gamma = \frac{1}{\sqrt{1 - \frac{v^2}{c^2}}} \tag{1.1}$$

There are two main calorimeters for ATLAS, the Liquid Argon (LAr) calorimeter and the Tile Hadronic calorimeter. The LAr calorimeter surrounds the inner detector and measures the energy deposits of electrons, photons and hadrons (quark bound states, such as baryons qqq and mesons $q\bar{q}$). It layers various metals to intercept the incoming particles to produce a shower of lower energy particles. The lower energy particles then ionize the liquid argon that fill the barrier in between the metal layers to produce a current that can be read out. The Tile calorimeter surrounds the LAr calorimeter and is the largest part of the ATLAS detector weighing in around 2900 tons. Particles then traverse through the layers of steel and plastic scintillating tiles. When a particle hits the steel, a cascade of secondary particles is generated, and the plastic scintillators will produce photons whose current can be measured.

1.3 ATLAS Trigger and Data Acquisition

The LHC produces pp-collisions at a rate of 40 MHz, each collision is an "event". The ATLAS Trigger system is responsible for quickly deciding what events are interesting for physics analysis. The Trigger system is divided into the first- and second-level triggers and when a particle activates a trigger, the trigger makes a decision to tell the Data Acquistion System (DAQ) to save the data produced by the detector. The first-level trigger is a hardware trigger that decides, within $2.5\mu s$ after the event, if it's a good event to put into a storage buffer for the second-level trigger. The second-level trigger is a software trigger that decides within $200\mu s$ and uses around 40,000 CPU-cores and analyses the event to decide if it is worth keeping. The second-level trigger selects about 1000 events per second to keep and store long-term. [6] The data taken by this Trigger/DAQ system is raw and not yet in a state that is ready for analysis, but it is ready for the reconstruction stage.

The amount of data taken at ATLAS is substantial. ATLAS sees more than 3.2 PB of raw data each year, each individual event being around 1.6 MB. [16] All of the data produced by LHC experiments, especially ATLAS, has to be sent to the LHC Computing Grid (LCG). The increase in data means more resources from the Grid will be needed, so optimization is an essential part of ensuring scalability of the data able to be taken in by the experiment. Reconstructed AOD are then processed through derivation jobs that reduced AODs from $\mathcal{O}(1)$ MB per event to $\mathcal{O}(10)$ kB per event, creating Derived AOD (DAOD).

1.4 HL-LHC and Future Needs in Computation

The High-Luminosity LHC (HL-LHC) is the upgrade to LHC that anticipates more events and more data taken than ever before. The goal is to reach $350 fb^{-1}$ in luminosity, which is forecasted to be gradually reached by around 2040.[3] The HL-LHC era is projected to demand anywhere from 6-10 times as much yearly storage, so any attempt to save on disk storage will help.

CHAPTER 2

I/O TOOLS

The Trigger/DAQ system sends and saves data from the detector to a persistent data storage solution. It's at this stage where the data isn't yet ready for an effective analysis, so what needs to happen is the data needs to be reconstructed and consolidated into physics objects, or Analysis Object Data (AOD) files. Creating AODs from data requires significant computation power and Athena is the software framework that plays a significant role in this process. This chapter will cover the software tools used by ATLAS

2.1 Athena and ROOT

Athena is the open-source software framework for the ATLAS experiment.[10] It uses on other software such as ROOT, Geant4 and other software as part of the LCG software stack. Athena manages ATLAS production workflows which include event generation, simulation of data, reconstruction from hits, and derivation of reconstructed hits.[11] It also provides some in-house based analysis tools as well as tools for specifically ROOT based analysis.

CMake and Make are open-source software that is used to build Athena, ROOT, and other software. A sparse build is a way to make changes to an individual package of code without having to recompile the entire framework at once, which saves time and resources. A user can create a text file identifying the path to the package modified, and the sparse build for Athena will proceed upon issuing the following commands:

```
cmake -DATLAS_PACKAGE_FILTER_FILE=../package_filters.txt ../athena/
Projects/WorkDir/
```

```
make -j
```

Where `../package_filters.txt` is the text file containing the path to the package modified, and `../athena/Projects/WorkDir/` is the path to the Athena source.

AthenaPOOL is data storage architecture suite of packages within Athena that provide conversion services. It originated as a separate project to serve as a layer between the transient data used by the software framework and the data stored permanently, persistent. The transient/persistent style of representing event data will be further explained in § 2.4.

These unit tests are component accumulator (CA) scripts written in Python and call upon algorithms written in C++. The python scripts are also used to set the job options for the algorithms added to the component accumulator, job options like flag definitions, input and output file names, and other algorithm specific options. A CA script written in pseudocode would take the form:

```python
# Import Packages
from AthenaConfiguration.AllConfigFlags import initConfigFlags
from AthenaConfiguration.ComponentFactory import CompFactory
from OutputStreamAthenaPool.OutputStreamConfig import OutputStreamCfg,
outputStreamName

# Set Job Options
outputStreamName = "StreamA"
outputFileName = "output.root"

# Setup flags
flags = initConfigFlags()
flags.Input.Files = ["input.root"]
flags.addFlag(f"Output.{streamName}FileName", outputFileName)
# Other flags
flags.lock()
```

```python

    # Main services
    from AthenaConfiguration.MainServicesConfig import MainServicesCfg
    acc = MainServicesCfg( flags )

    # Add algorithms
    acc.addEventAlgo( CompFactory.MyAlgorithm(MyParameters) )

    # Run
    import sys
    sc = acc.run(flags.Exec.MaxEvents)
    sys.exit(sc.isFailure())
```

ROOT is an open-source software framework used for high-energy physics analysis at CERN.[18] It uses C++ objects to save, access, and process data brought in by the various experiments based at the LHC, the ATLAS experiment uses it in conjunction with Athena. ROOT largely revolves around organization and manipulation of TFiles and TTrees into ROOT files. A TTree represents a columnar dataset, and the list of columns are called branches. The branches have memory buffers that are automatically allocated by ROOT. These memory buffers are divided into corresponding baskets, whose size is designated during memory allocation. More detail on branch baskets are explored in Chapter 3 and 4.

2.1.1 Continuous Integration (CI) and Development

CI is a software development practice where new code is tested and validated upon each merge to the main branch of a repository. Every commit to the main branch is automatically built and tested for specific core features that are required to work with the codebase. This helps to ensure that the codebase is working as intended and that the new code is compatible with the existing codebase.

Athena is hosted on GitLab and developed using CI with an instance of Jenkins, called ATLAS Robot, to build and test the new changes within a merge request interface. ATLAS Robot will then provide a report of the build and test results. If the build or test fail, ATLAS Robot will provide a report of which steps failed and why. This allows for early detection of issues before the nightly build is compiled and tested.

2.2 TTree Object

A TTree is a ROOT object that organizes physically distinct types of event data into branches. Branches hold data into dedicated contiguous memory buffers, and those memory buffers, upon compression, become baskets. These baskets can have a limited size and a set minimum number of entries. The Athena default basket size at present is 128 kB, and the default minimum number of entries is 10.

One function relevant to TTree is `Fill()`. `Fill()` will loop over all of the branches in the TTree and compresses the baskets that make up the branch. This removes the basket from memory as it is then compressed and written to disk. It makes reading back branches faster as all of the baskets are stored near each other on the same disk region. [19]

`AutoFlush` is a function that tells the `Fill()` function after a designated number of entries of the branch, in this case vectors, to flush all branch buffers from memory and save them to disk.

2.3 Derivation Production Jobs

A derivation production job takes AODs, which comes from the reconstruction step at $\mathcal{O}(1\text{ MB})$ per event, and creates a derived AOD (DAOD) which sits at $\mathcal{O}(10\text{ kB})$ per event. Derivation production is a necessary step to make all data accessible for physicists doing analysis as well as reducing the amount of data that needs to be processed. While derivations are reduced AODs, they often contain additional information useful for analysis, such as jet collections and high-level discriminants.[17] Athena provides two types of output files from a derivation job, PHYS and PHYSLITE. Figure 2.1 shows the object composition of a PHYS and PHYSLITE $t\bar{t}$ sample. PHYS output files, at 40.0 kB per event, is prodomi-

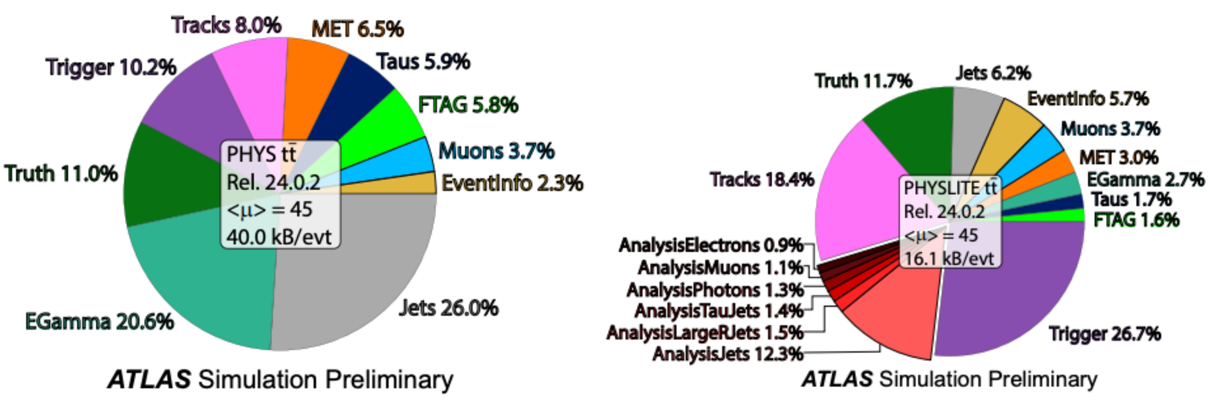

Figure 2.1: Object composition of a PHYS and PHYSLITE $t\bar{t}$ sample from Run 3.

nantly made of jet collections, while PHYSLITE, at 16.1 kB per event, has more trigger and track information. There is ongoing work to reduce the amount of Trigger information in PHYSLITE which would help to reduce the file size.

PHYSLITE, being the smallest file of the two, sees the largest effect upon attempts of optimization. These jobs can demand heavy resource usage on the GRID, so optimization of the AOD/DAODs for derivation jobs can be vital.

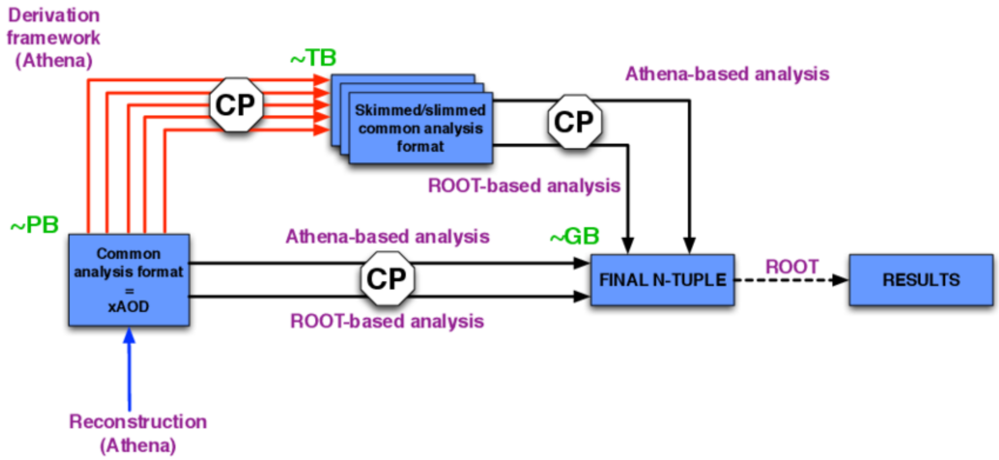

Figure 2.2: Derivation production from Reconstruction to Final N-Tuple[13]

The derivation framework is sequence of steps that are performed on the AODs to create the DAODs. Skimming is the first step in the derivation framework, and it's responsible for removing whole events based on pre-defined criteria. Thinning is the second step, and it removes whole objects based on pre-defined criteria. Lastly slimming removes variables from objects uniformly across events.

2.4 Event Data Models

An Event Data Model (EDM) is a collection of classes and their relationships to each other that provide a representation of an event detected with the goal of making it easier to use and manipulate by developers. An EDM is how particles and jets are represented in memory, stored to disk, and manipulated in analysis. It's useful to have an EDM because it brings a commonality to the code, which is useful when developers reside in different groups with various backgrounds. An EDM allows those developers to more easily debug and communicate issues when they arise.

2.4.1 Transient/Persistent (T/P) EDM

One of the previous EDM schemas used by ATLAS concerned a dual transient/persistent status of AOD. With this EDM, the AOD was converted into an ntuple based format called D3PDs. While this conversion allowed for fast readability and partial read for effient analysis in ROOT, it left the files disconnected from the reconstruction tools found in Athena.[5] When transient data was present in memory, it could have information attatched to the object and gain in complexity the more it was used. Transient data needed to be simplified before it could become persistent into long-term storage (sent to disk). ROOT had trouble handling the complex inheritance models that would come up the more developers used this EDM. Before the successor to the T/P EDM was created, ATLAS physicists would convert data samples using the full EDM to a simpler one that would be directly readable by ROOT. This would lead to duplication of data and made it challenging to develop and maintain the analysis tools to be used on both the full EDM and the reduced ones. Additionally, converting from transient to persistent data was an excessive step which was eventually removed by the adoption of using an EDM that blends the two stages of data together, this was dubbed the xAOD EDM.

2.4.2 xAOD EDM

The xAOD EDM is the successor to the T/P EDM and brings a number of improvements. This EDM, unlike T/P, is usuable both on Athena and ROOT. It's easier to pick up for analysis and reconstruction. The xAOD EDM has the ability to add and remove variables at runtime, these variables are called "decorations."

The xAOD EDM use two types of objects handle data, interface objects and payload objects. Interfaces act as an interface for the user to access the object but without its stored data. This differs from T/P where the user would have to load an object into memory to access the object. If the user wanted to delay the loading of data into memory, they could use the interface object to do so. The payload object contains the data for the interface object and is allocating contiguous blocks of memory. Payload classes are often referred to as auxiliary storage.

The specific data structure used by ATLAS is the ROOT TTree, but the EDM is agnostic to the type of data structure used. ATLAS specific libraries are not required to handle files written in the xAOD format since the payload can be read directly from the contiguous allocation of memory, a central tenent of the xAOD EDM. This allows for the separation of ATLAS specific analysis frameworks and the preferred analysis tool of the user. More information on how the xAOD EDM is deployed into unit tests in § 5.1.

CHAPTER 3

TOY MODEL BRANCHES

Building a toy model for derivation production jobs offers a simplified framework to effectively simulate and analyze the behavior of real and Monte Carlo (MC) data under techniques of optimization aimed to study. One commonality between both data and MC is the data types stored in branches for both is made of a mixture between repeated integer-like data and randomized floating-point data. Intergers are easier to compress than floating-point numbers, so adjusting the mixture of each will yield compression ratios closer to real and MC data. Replicating this mixture in a branch give us an effective model that resemble how current derivation jobs act on real and MC simulated data. These toy model mixtures provide an avenue to test opportunities for optimizing the memory and storage demands of the GRID by first looking at limiting basket sizes and their effects on compression of branches.

3.1 Toy Model Compression

3.1.1 Random Float Branches

There were a number of iterations to the toy model, but the first was constructed by filling a TTree with branches that each have vectors with varying number of random floats to write and read. Vectors are used in this toy model, as opposed to arrays, because vectors are dynamically allocated and deallocated, which allows for more flexibility when synthesizing AOD. This original model had four distinct branches, each with a set number of events

(`N=1000`), and each event having a number of entries, vectors with 1, 10, 100, and 1000 floats each.

The script can be compiled with `gcc` or `g++` and it requires all of the dependencies that come with ROOT. Alternatively, the script can be run direclty within ROOT.

The following function `VectorTree()` is the main function in this code. What is needed first is an output file, which will be called `VectorTreeFile.root`, and the name of the tree can simply be `myTree`. Initializing variables start with the total number of events in the branch, i.e. the number of times a branch is filled with the specified numbers per vectors, `N`. Additionally the branches have a number of floats per vector, this size will need to be defined as `size_vec_0`, `size_vec_1`, etc. The actual vectors that are being stored into each branch need to be defined as well as the temporary placeholder variable for our randomized floats, `vec_tenX` and `float_X` respectively.

```
void VectorTree() {
  ...
  const int N = 1e4; // N = 10000, number of events
  // Set size of vectors with 10^# of random floats
  int size_vec_0 = 1;
  int size_vec_1 = 10;
  int size_vec_2 = 100;
  int size_vec_3 = 1000;

  // vectors
  std::vector<float> vec_ten0; // 10^0 = 1 entry
  std::vector<float> vec_ten1; // 10^1 = 10 entries
  std::vector<float> vec_ten2; // 10^2 = 100 entries
  std::vector<float> vec_ten3; // 10^3 = 1000 entries

  // variables
```

```
  float float_0;
  float float_1;
  float float_2;
  float float_3;
  ...
}
```

From here, initialize the branches so each one knows where its vector pair resides in memory.

```
void VectorTree() {
  ...
  // Initializing branches
  std::cout << "creating branches" << std::endl;
  tree->Branch("branch_of_vectors_size_one", &vec_ten0);
  tree->Branch("branch_of_vectors_size_ten", &vec_ten1);
  tree->Branch("branch_of_vectors_size_hundred", &vec_ten2);
  tree->Branch("branch_of_vectors_size_thousand", &vec_ten3);
  ...
}
```

One extra step taken during this phase of testing is the disabling of `AutoFlush`.

```
void VectorTree() {
  ...
  tree->SetAutoFlush(0);
  ...
```

Disabling `AutoFlush` allows for more consistent compression across the various sizes of branch baskets. The toy model needed this consistency more than the later tests as these early tests were solely focused on mimicking data procured by the detector and event simulation. The derivation production jobs tested in Chapter 4 were tested with `AutoFlush` enabled because those tests are not as concerned with compression as they are with memory and disk usage. Following branch initialization comes the event loop where data is generated and emplaced into vectors.

```
void VectorTree() {
  ...
  // Events Loop
  std::cout << "generating events..." << std::endl;
  for (int j = 0; j < N; j++) {
      // Clearing entries from previous iteration
      vec_ten0.clear();
      vec_ten1.clear();
      vec_ten2.clear();
      vec_ten3.clear();

      // Generating vector elements, filling vectors
      // Fill vec_ten0
      // Contents of the vector:
      //    {float_0}
      //    Only one float of random value
      float_0 = gRandom->Rndm() * 10; // Create random float value
      vec_ten0.emplace_back(float_0); // Emplace float into vector

      // Fill vec_ten1
      // Contents of the vector:
      //    {float_1_0, ... , float_1_10}
      //    Ten floats, each float is random
      for (int n = 0, n < size_vec_1; n++) {
          float_1 = gRandom->Rndm() * 10;
          vec_ten1.emplace_back(float_1);
      }
```

```
        // Do the same with vec_ten2 and vec_ten3, except for
        //     vectors with size 100 and 1000 respectively.

        // After all branches are filled, fill the TTree with
        //     new branches
        tree->Fill();
    }
    // Saving tree and file
    tree->Write();
    ...
}
```

Once the branches were filled, ROOT then will loop over each of the branches in the TTree and at regular intervals will remove the baskets from memory, compress, and write the baskets to disk (flushed), as was discussed in Section §2.2.

As illustrated, the `TTree` is written to the file which allows for the last steps within this script.

```
void VectorTree() {
    ...

    // Look in the tree
    tree->Scan();
    tree->Print();

    myFile->Save();
    myFile->Close();
}

int main() {
```

```
    VectorTree();
    return 0;
}
```

Upon reading back the ROOT file, the user can view the original size of the file (Total-file-size), the compressed file size (File-size), the ratio between Total-file-size and File-size (Compression Factor), the number of baskets per branch, the basket size, and other information. Filling vectors with entirely random values was believed to yield compression ratios close to real data, but the results in Figure 3.1 show changes needed to be made to bring the branches closer to a compression ratio of $\mathcal{O}(5)$. It is evident that branches containing vectors with purely random floats are more difficult to compress due to the high level of randomization.

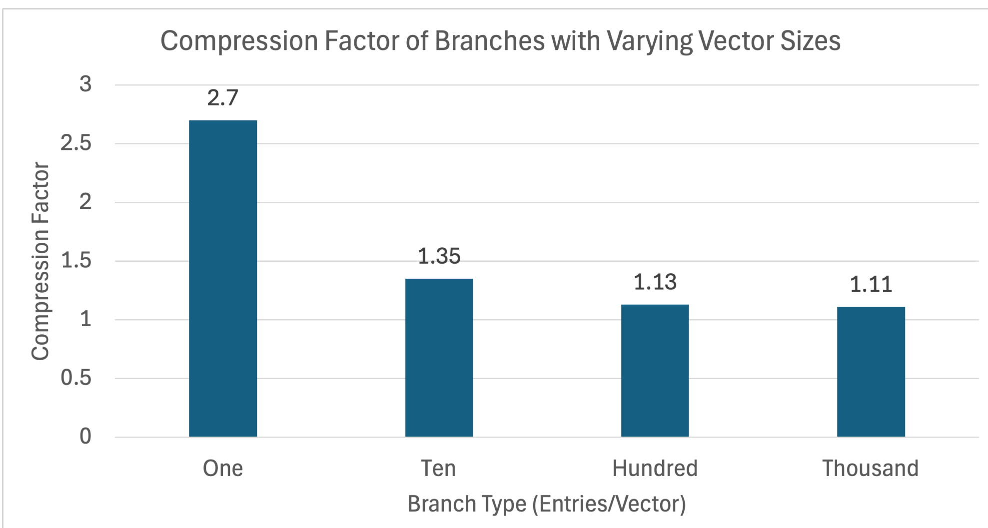

Figure 3.1: Compression factors of $N = 1000$ entries per branch with random-valued vectors of varying size.

Figure 3.1 shows compression drop-off as the branches with more randomized floats per vector were present. This is the leading indication that there needs to be more compressible data within the branches.

3.1.2 Mixed-Random Float Branches

The branches needed to have some balance between compressible and incompressible data to mimic the compression ratio found in real data. How this was achieved was by filling each vector with different ratios of random floats and repeating integers, which will now be described in detail.

The first change was increasing the total number of events per branch from $N = 10^4$ to $N = 10^5$, or from 10,000 to 100,000. Mixing of random floats and repeated integer values takes the same script structure as Section § 3.1.1 but adjusts the event generation loop.

```
void VectorTree() {
  ...
  // Events Loop
  for (int j = 0; j < N; j++) {
      // Clearing entries from previous iteration
      vec_ten0.clear();
      vec_ten1.clear();
      vec_ten2.clear();
      vec_ten3.clear();

      // Generating vector elements, filling vectors
      // Generating vec_ten0
      // Contents of the vector:
      //    {float_0}
      //    Only one float of random value
      // And since there's only one entry, we don't mix the entries.
      float_0 = gRandom->Gaus(0, 1) * gRandom->Rndm();
      vec_ten0.emplace_back(float_0);


```

```
21          // Generating vec_ten1
22          // Contents of the vector:
23          //     {float_1_0, float_1_1, float_1_2, float_1_3, float_1_4, 1,
    1, 1, 1, 1}
24          //     5 floats of random values, 5 integers of value 1.
25          for (int b = 0; b < size_vec_1; b++) {
26              if (b < size_vec_1 / 2) {
27                float_1 = gRandom->Rndm() * gRandom->Gaus(0, 1);
28                vec_ten1.emplace_back(float_1);
29              } else {
30                float_1 = 1;
31                vec_ten1.emplace_back(float_1);
32              }
33          }
34
35          // Do the same with vec_ten2 and vec_ten3, except for
36          //      vectors with size 100 and 1000 respectively.
37
38
39          // After all branches are filled, fill the TTree with
40          //      new branches
41          tree->Fill();
42      }
43      // Saving tree and file
44      tree->Write();
45      ...
46  }
```

As shown in the `if`-statements in lines `14`, `25`, `36` and `47`, if the iterator was less than half of the total number of entries in the vector then that entry had a randomized float put in that spot in the vector, otherwise it would be filled with the integer `1`. Having a mixture of half random floats and half integer 1 led to the larger branches still seeing poor compression, so a new mixture of 1/4 random data was introduced. Even though $N = 10^5$ had the larger branches closer to the desired compression ratio, testing at $N = 10^6$ events improves the accuracy of the overall file size to more closely resemble real data.

Figure 3.2 shows the difference between compression between the two mixtures at $N = 10^6$ events. When the number of events is increased from $N = 10^5$ to $N = 10^6$, at the 1/2 random-mixture, the branches with more than one entry per vector see their compression factor worsen. Figure 3.3 shows a compression ratio hovering around 3 for the larger branches, whereas Figure 3.2 shows the same branches hovering around 2.

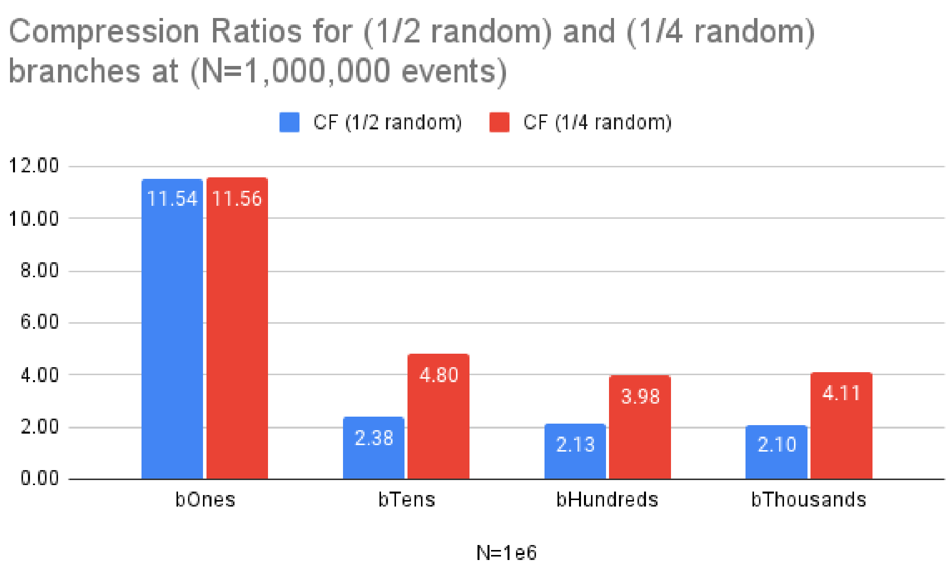

Figure 3.2: Compression Ratios for ($\frac{1}{2}$ random) and ($\frac{1}{4}$ random) branches at ($N = 10^6$ events)

Unlike the mixture of branches having 1/2 random data, the 1/4 mixture does not see the same compression effect, but with this mixture we see a compression ratio that is in-line with real data. This is inline with expectation, more repeated integers within the mixture makes the branch more compressible, and the more random floats in the mixture will make

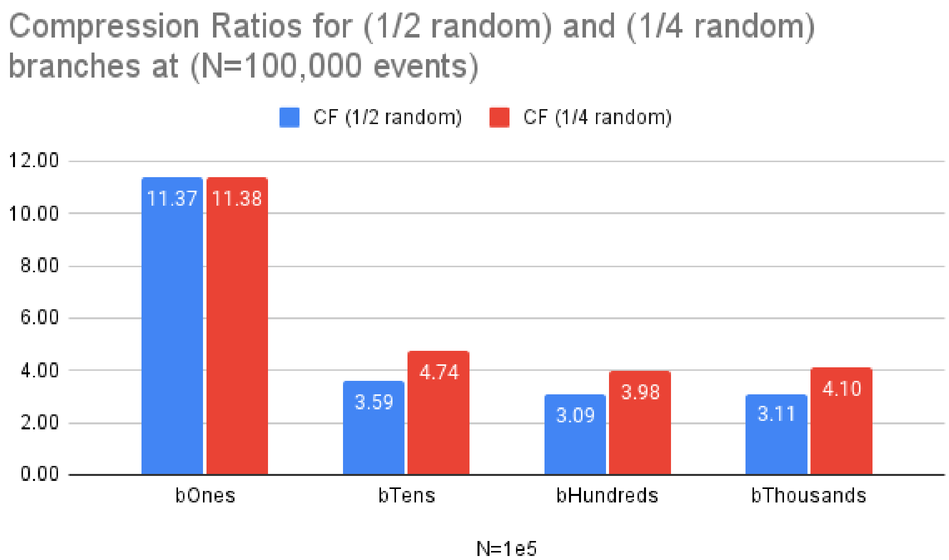

Figure 3.3: Compression Ratios for ($\frac{1}{2}$ random) and ($\frac{1}{4}$ random) branches at ($N = 10^5$ events)

the branch more difficult to compress. With these mixtures added to the toy model, we can start looking at varying the basket sizes to see how they affect compression.

3.2 Basket-Size Investigation

Investigating how compression is affected by the basket size requires us to change the basket size, refill the branch and read it out. Changing the basket sizes was done at the script level with a simple setting after the branch initialization and before the event loop the following code:

```
int basketSize = 8192000;
tree->SetBasketSize("*",basketSize);
```

This ROOT-level setting was sufficient for the case of the toy model; testing of the basket size setting both at the ROOT- and Athena-level would be done later using derivation production jobs in Section §??. The lower bound set for the basket size was 1 kB and the upper bound was 16 MB. The first branch looked at closely was the branch with a thousand vectors with half of them being random floats, see Figure 3.4.

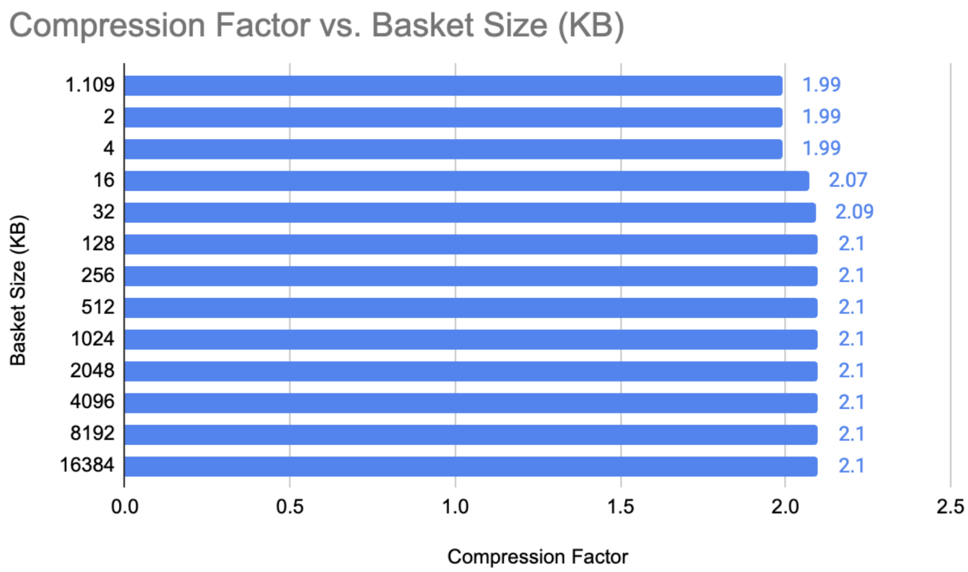

Figure 3.4: Compression Factors vs Branch Size (1000 entries per vector, 1/2 Mixture $N = 10^6$ events)

Figure 3.4 and Figure 3.5 are the first indication that the lower basket sizes are too small to effectively compress the data. For baskets smaller than 16 kB, it is necessary to have as many baskets as events to store all the data effectively. For a mixed-content vector with one thousand entries, containing 500 floats and 500 integers (both are 4 bytes each), its size is approximately 4 kB. ROOT creates baskets of at least the size of the smallest branch entry, in this case the size of a single vector. So even though the basket size was set to 1 or 2 kB, ROOT created baskets of 4 kB. These baskets $\leq$ 4kB have a significantly worse compression than the baskets $\geq$ 4kB in size, so the focus was shifted toward baskets Once the basket size is larger than the size of a single vector, more than one vector can be stored in a single basket and the total number of baskets is reduced.

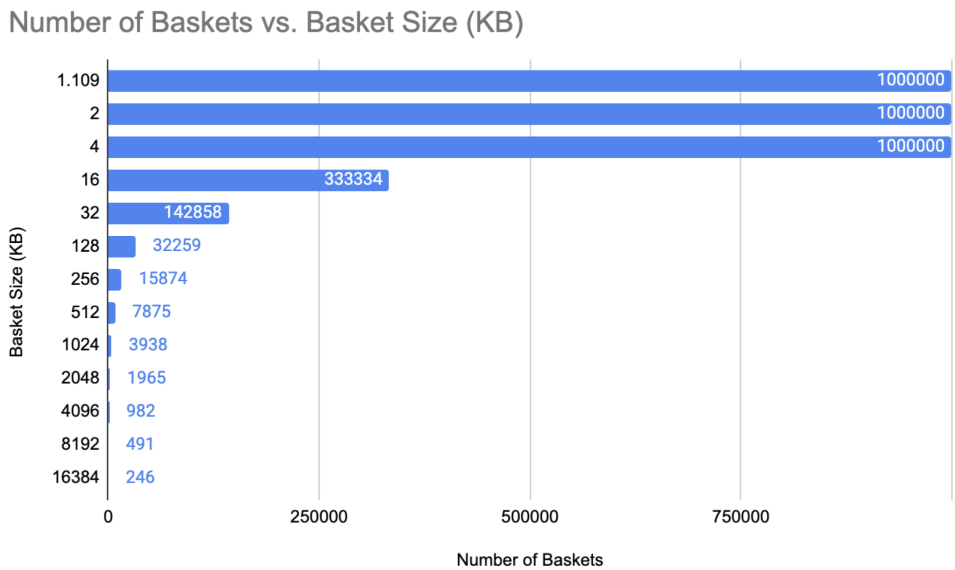

Figure 3.5: Number of Baskets vs Branch Size (1000 entries per vector, 1/2 Mixture $N = 10^6$ events)

There were different types of configuration to the toy model investigated by this study. Looking further into the types of mixtures and how they would affect compression are shown in Figure 3.6 and 3.7. Here the same mixtures were used but the precision of the floating point numbers was decreased from the standard 32 floating-point precision to 16 and 8, making compression easier.

Each of these sets of tests indicate that after a certain basket size, i.e. 128 kB, there is no significant increase in compression. Having an effective compression at 128 kB, it's useful to stick to that basket size to keep memory usage down. Knowing that increasing the basket size beyond 128 kB yields diminishing returns, it's worth moving onto the next phase of testing with actual derivation production jobs.

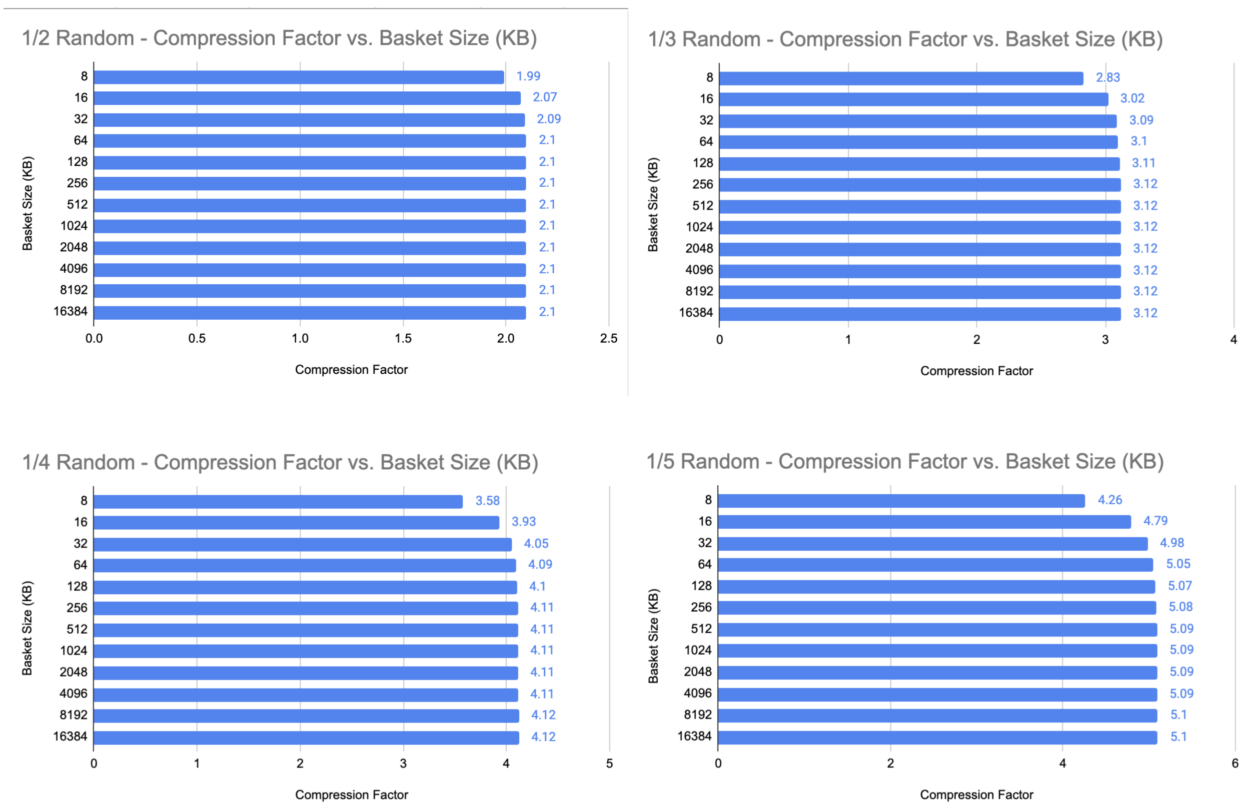

Figure 3.6: Varying Mixtures in 8 point precision - Number of Baskets vs Branch Size ($N = 10^6$ events)

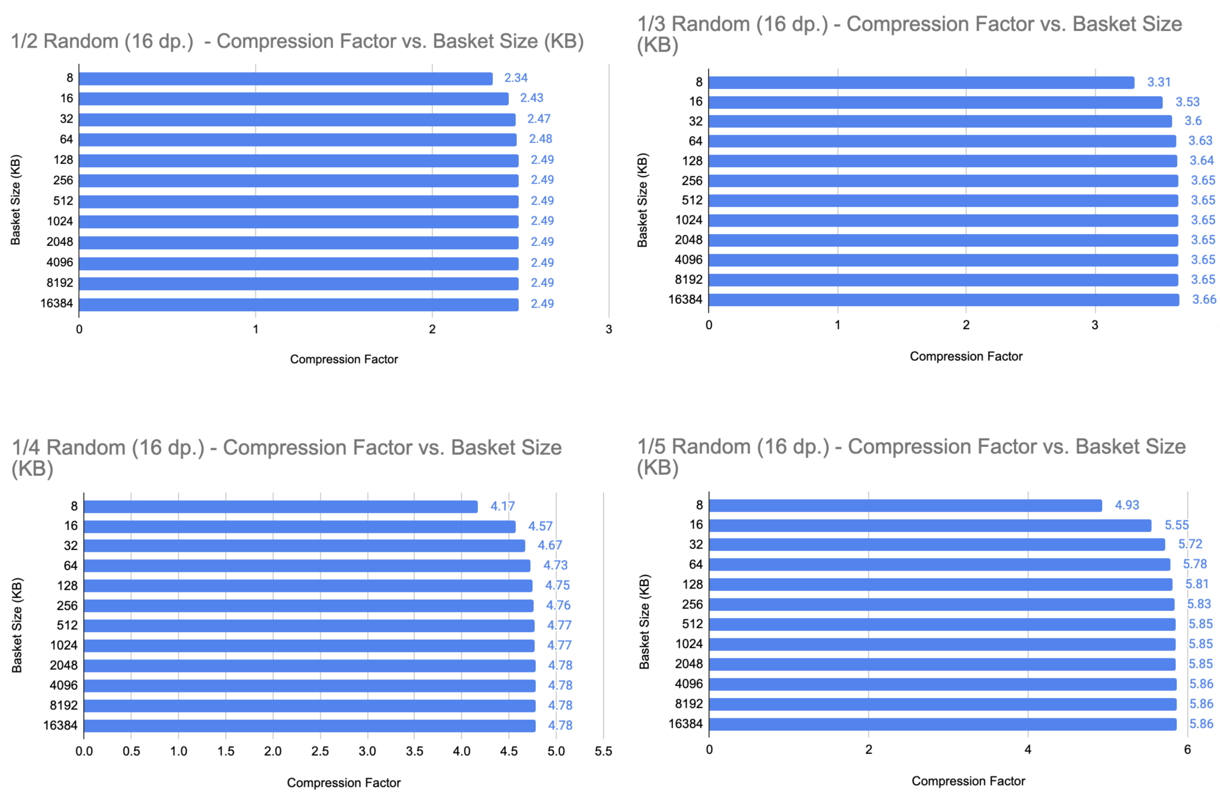

Figure 3.7: Varying Mixtures in 16 point precision - Number of Baskets vs Branch Size ($N = 10^6$ events)

CHAPTER 4

DATA AND MONTE CARLO DERIVATION PRODUCTION

Derivation production demands high memory usage, and DAODs make up a bulk of disk-space usage. DAODs are used in physics analyses and ought to be optimized to alleviate stress on the GRID and to lower disk-space usage. Optimizing both disk-space and memory usage is a tricky balance as they are typically at odds with one another. For example, increasing memory output memory buffers results in lower disk-space usage due to better compression but the memory usage will increase since the user will load a larger buffer into memory. This project opted to take is by optimizing for disk-space and memory by testing various basket limits and viewing the effects of the branches on both data and Monte Carlo (MC) simulated analysis object data (AODs).

4.1 Basket-size Configuration

As the toy model ruled out, the focus here was on optimizing Athena and not ROOTs contribution for optimization. The initial focus was on the inclusion of a minimum number of entries per buffer and the maximum basket buffer limit. The AthenaPOOL script directly involved with these buffer settings is the `PoolWriteConfig.py` found in the path `athena/Database/AthenaPOOL/AthenaPoolCnvSvc/python/`. As discussed in Section §4.2, further testing opted to keep the minimum number of entries set to its default setting, 10 entries per buffer.

Throughout the duration of this testing, the results of compression or file size are independent of any changes to the release or the nightly version of Athena. The data derivation job comes from a 2022 dataset with four input files and 160,327 events. The MC job comes from a 2023 $t\bar{t}$ standard sample simulation job with six input files and 140,000 events. The datasets are noted in Appendix A.1.

4.1.1 Derivation Job Command

To run a derivation job, AODs need to be downloaded by data-management service, such as Rucio, to a users local machine. Rucio is the data-management solution used for this project to procure the various AOD input files used for the derivation jobs. The machine running the Rucio client will need to have a valid proxy added for Rucio to run correctly. A sample command would look like:

```
rucio download data22_13p6TeV:AOD.31407809._000898.pool.root.1
```

This downloads the AOD file from Rucio and saves it to the user's local directory.

The command used by Athena to run a derivation job takes the form of the following example:

```
ATHENA_CORE_NUMBER=4 Derivation_tf.py \
--CA True \
--inputAODFile mc23_13p6TeV.601229.PhPy8EG_A14_ttbar_hdamp258p75_SingleLep
    .merge.AOD.e8514_e8528_s4162_s4114_r14622_r14663/AOD.33799166._001224.
    pool.root.1 \
--outputDAODFile art.pool.root \
--formats PHYSLITE \
--maxEvents 2000 \
--sharedWriter True \
--multiprocess True ;
```

Where Athena allows one to specify the number of cores to use with the `ATHENA_CORE_NUMBER` environment variable. `Derivation_tf.py` is a script that runs the derivation job and is part of the Athena release. The `--inputAODFile` is the input file for the derivation job, in this case an AOD file. The user can specify multiple input files at a time by enclosing the input files in quotes and separating each file with a comma, like the following:

```
--inputAODFile="AOD.A.pool.root.1,AOD.B.pool.root.1,AOD.C.pool.root.1,
AOD.D.pool.root.1"
```

The `--outputDAODFile` is the output file for the derivation job, in this case a DAOD file. The `--formats PHYSLITE` flag allows the job to use the PHYSLITE format for the DAOD. Here is where the user may choose to include PHYS or PHYSLITE simply by inlusion of one or both. The `--maxEvents` flag allows one to specify the maximum number of events to run the job on. The `--sharedWriter True` flag allows the job to utilize SharedWriter. The `--multiprocess True` flag allows the job to use AthenaMP tools.

The input files for both data and MC jobs were ran with various configurations of Athena by modifying the basket buffer limit. The four configurations tested all kept minimum number of basket buffer entries at 10 and modified the basket limitation in the following ways:

1. "*default*" - Athena's default setting, and basket limit of 128×1024 bytes
2. "*256k*" - Limit basket buffer to 256×1024 bytes
3. "*512k*" - Limit basket buffer to 512×1024 bytes
4. "*no-lim*" - Removing the Athena basket limit, the ROOT imposed 1.3 MB limit still remains

4.2 Results

4.2.1 Presence of basket-cap and presence of minimum number of entries

First batch testing was for data and MC simulation derivation production jobs with and without presence of an upper limit to the basket size and presence of the minimum number of basket buffer entries. PHYSLITE MC derivation production, from Table 4.2, sees a 9.9% increase in output file size when compared to the default Athena configuration. Since this configuration only differs by the omission of the “min-number-entries” requirement, we assume the minimum number of basket buffer entries should be kept at 10 and left alone. Table 4.2 also shows the potential for a PHYSLITE MC DAOD output file size reduction by eliminating our upper basket buffer limit altogether.

Presence of features (Data)	Max PSS (MB) (Δ% default)	PHYS outFS (GB) (Δ%)	PHYSLITE outFS (GB) (Δ%)
basket-cap, min-num-entries (default)	27.1 (+ 0.0 %)	3.22 (+ 0.0 %)	1.03 (+ 0.0 %)
~~basket-cap~~ ~~min-num-entries~~	27.8 (+ 2.5 %)	3.22 (+ 0.2 %)	1.04 (+ 0.2 %)
~~basket-cap~~ min-num-entries	27.8 (+ 2.5 %)	3.22 (- 0.0 %)	1.03 (- 0.4 %)
basket-cap, ~~min-num-entries~~	27.3 (+ 0.7 %)	3.22 (+ 0.2 %)	1.04 (+ 0.7 %)

Table 4.1: Comparing the maximum proportional set size (PSS) and PHYS/PHYSLITE output file sizes (outFS) for data jobs while varying the presence of features in Athena PoolWriteConfig.py for 160327 entries.

Presence of features (MC)	Max PSS (MB) (Δ% default)	PHYS outFS (GB) (Δ%)	PHYSLITE outFS (GB) (Δ%)
basket-cap, min-num-entries (default)	14.1 (+ 0.0 %)	5.8 (+ 0.0 %)	2.6 (+ 0.0 %)
~~basket-cap~~ ~~min-num-entries~~	16.1 (+ 12.1 %)	6.0 (+ 2.9 %)	2.7 (+ 5.1 %)
~~basket-cap~~ min-num-entries	16.0 (+ 11.5 %)	5.7 (- 2.8 %)	2.5 (- 5.6 %)
basket-cap, ~~min-num-entries~~	14.2 (+ 0.4 %)	6.2 (+ 5.4 %)	2.9 (+ 9.9 %)

Table 4.2: Comparing the maximum proportional set size (PSS) and PHYS/PHYSLITE output file sizes (outFS) for MC jobs while varying the presence of features in Athena PoolWriteConfig.py for 140000 entries.

4.2.2 Comparing different basket sizes

Pre-existing derivation jobs were ran for data and MC simulations to compare between configurations of differing basket sizes limits. The results for this set of testing are found from Table 4.3 through Table 4.10. The following tables are the DAOD output-file sizes of the various Athena configurations for PHYS/PHYSLITE over their respective data/MC AOD input files.

Athena Configs (Data)	Max PSS (MB) (Δ% default)	PHYS outFS (GB) (Δ%)	PHYSLITE outFS (GB) (Δ%)
(default)	27.9 (+ 0.0 %)	3.3 (+ 0.0 %)	1.0 (+ 0.0 %)
256k_basket	28.2 (+ 1.3 %)	3.3 (- 0.1 %)	1.0 (- 0.3 %)
512k_basket	28.5 (+ 2.2 %)	3.3 (+ 0.0 %)	1.0 (- 0.3 %)
1.3 MB (ROOT MAX)	28.6 (+ 2.7 %)	3.3 (- 0.1 %)	1.0 (- 0.3 %)

Table 4.3: Comparing the maximum proportional set size (PSS) and PHYS/PHYSLITE output file sizes (outFS) for Data jobs over various Athena configurations for 160327 entries.

Athena Configs (Data)	Max PSS (MB) (Δ% default)	PHYS outFS (GB) (Δ%)	PHYSLITE outFS (GB) (Δ%)
(default)	15.0 (+ 0.0 %)	5.9 (+ 0.0 %)	2.6 (+ 0.0 %)
256k_basket	15.3 (+ 1.9 %)	5.8 (- 1.4 %)	2.5 (- 3.1 %)
512k_basket	16.4 (+ 8.6 %)	5.7 (- 2.5 %)	2.5 (- 5.1 %)
1.3 MB (ROOT MAX)	16.9 (+ 11.3 %)	5.7 (- 2.8 %)	2.5 (- 5.6 %)

Table 4.4: Comparing the maximum proportional set size (PSS) and PHYS/PHYSLITE output file sizes (outFS) for MC jobs over various Athena configurations for 140000 entries.

"Max PSS" refers to the maximum proportional set size, which is the maximum memory usage of the job. Table 4.3 tells us that with this $t\bar{t}$ data sample, there are marginal changes in both the memory usage for the job and the output file size of the DAODs. Whereas Table 4.4 shows a much more drastic change, with a 5.6% reduction in output file size for the MC PHYSLITE DAOD when compared to the default Athena configuration. While there's a 5.6% reduction in output file size for the MC PHYSLITE DAOD, there's also a 11.3% increase in memory usage.

4.2.3 Monte Carlo PHYSLITE branch comparison

Derivation production jobs work with initially large, memory-consuming branches, compressing them to a reduced size. These derivation jobs are memory intensive because they first have to load the uncompressed branches into readily-accessed memory. Once they're loaded, only then are they able to be compressed. The compression factor is the ratio of pre-derivation branch size (Total-file-size) to post-derivation branch size (Compressed-file-size). The compressed file size is the size of the branch that is permanently saved into the DAOD.

Branches with highly repetitive data are better compressed than non-repetitive data, leading to high compression factors–the initial size of the branch contains more data than it needs pre-derivation. If pre-derivation branches are larger than necessary, there should be an opportunity to save memory usage during the derivation job.

The following tables look into some highly compressible branches and might lead to areas where simulation might save some space. (AOD pre compression?)

Athena v24.0.16 (default) MC branch	Branch size (kB)	Total-file-size (MB)	Compressed-file-size (MB)	Compression factor
PrimaryVerticesAuxDyn.trackParticleLinks	128	2146.2	24.0	89.4
HardScatterVerticesAuxDyn.incomingParticleLinks	128	118.5	1.7	71.6
HLTNav_Summary_DAODSlimmedAuxDyn.linkColNames	128	784.0	11.9	65.7
HardScatterVerticesAuxDyn.outgoingParticleLinks	128	108.6	1.9	58.7
TruthBosonsWithDecayVerticesAuxDyn.incomingParticleLinks	96	31.6	0.7	43.5
HLTNav_Summary_DAODSlimmedAuxDyn.linkColClids	128	390.6	10.7	36.6
AnalysisTauJetsAuxDyn.tauTrackLinks	128	75.0	2.0	36.6
HLTNav_Summary_DAODSlimmedAuxDyn.linkColKeys	128	390.6	11.7	33.4
AnalysisJetsAuxDyn.GhostTrack	128	413.8	13.1	31.5
TruthBosonsWithDecayVerticesAuxDyn.outgoingParticleLinks	83.5	27.3	0.9	31.0

Table 4.5: Top 10 branches sorted by compression factor, MC PHYSLITE [Athena v24.0.16 default configuration.]

An immediate observation: with the omission of the Athena basket limit (solely relying on ROOTs $1.3MB$ basket limit), the compression factor increases. This is inline with the original expectation that an increased buffer size limit correlate to better compression. *PrimaryVerticesAuxDyn.trackParticleLinks* is a branch where, among each configuration of Athena MC derivation, has the highest compression factor of any branch in this dataset.

Athena v24.0.16 (no-lim) MC branch	Branch size (kB)	Total-file-size (MB)	Compressed-file-size (MB)	Compression factor
PrimaryVerticesAuxDyn.trackParticleLinks	1293.5	2145.5	22.9	93.5
HardScatterVerticesAuxDyn.incomingParticleLinks	693.0	118.5	1.3	90.1
HardScatterVerticesAuxDyn.outgoingParticleLinks	635.5	108.5	1.5	74.0
HLTNav_Summary_DAODSlimmedAuxDyn.linkColNames	1293.5	783.5	11.9	65.8
TruthBosonsWithDecayVerticesAuxDyn.incomingParticleLinks	96.0	31.6	0.7	43.5
AnalysisTauJetsAuxDyn.tauTrackLinks	447.0	74.9	1.9	39.2
HLTNav_Summary_DAODSlimmedAuxDyn.linkColClids	1293.5	390.3	11.0	35.5
HLTNav_Summary_DAODSlimmedAuxDyn.linkColKeys	1293.5	390.3	11.3	34.5
AnalysisJetsAuxDyn.GhostTrack	1293.5	413.5	13.0	31.9
TruthBosonsWithDecayVerticesAuxDyn.outgoingParticleLinks	83.5	27.3	0.9	31.0

Table 4.6: Top 10 branches sorted by compression factor, MC PHYSLITE [Athena v24.0.16 without limit to the basket buffer.]

Athena v24.0.16 (default) MC branch	Branch size (kB)	Total-file-size (MB)	Compressed-file-size (MB)	Compression factor
PrimaryVerticesAuxDyn.trackParticleLinks	128	2146.2	24.0	89.4
HLTNav_Summary_DAODSlimmedAuxDyn.linkColNames	128	784.0	11.9	65.7
AnalysisJetsAuxDyn.GhostTrack	128	413.8	13.1	31.5
HLTNav_Summary_DAODSlimmedAuxDyn.linkColClids	128	390.6	10.7	36.6
HLTNav_Summary_DAODSlimmedAuxDyn.linkColKeys	128	390.6	11.7	33.4
AnalysisJetsAuxDyn.SumPtChargedPFOPt500	128	148.9	7.3	20.5
AnalysisJetsAuxDyn.NumTrkPt1000	128	148.8	8.7	17.2
AnalysisJetsAuxDyn.NumTrkPt500	128	148.8	11.9	12.5
HardScatterVerticesAuxDyn.incomingParticleLinks	128	118.5	1.7	71.6
AnalysisLargeRJetsAuxDyn.constituentLinks	128	111.5	7.1	15.8

Table 4.7: Top 10 branches sorted by total file size in bytes, MC PHYSLITE [Athena v24.0.16 default configuration.]

Athena v24.0.16 (no-lim) MC branch	Branch size (kB)	Total-file-size (MB)	Compressed-file-size (MB)	Compression factor
PrimaryVerticesAuxDyn.trackParticleLinks	1293.5	2145.5	22.9	93.6
HLTNav_Summary_DAODSlimmedAuxDyn.linkColNames	1293.5	783.5	11.9	65.8
AnalysisJetsAuxDyn.GhostTrack	1293.5	413.5	13.0	31.9
HLTNav_Summary_DAODSlimmedAuxDyn.linkColClids	1293.5	390.3	11.0	35.5
HLTNav_Summary_DAODSlimmedAuxDyn.linkColKeys	1293.5	390.3	11.3	34.5
AnalysisJetsAuxDyn.SumPtChargedPFOPt500	905.5	148.8	6.8	21.9
AnalysisJetsAuxDyn.NumTrkPt1000	905	148.8	8.5	17.6
AnalysisJetsAuxDyn.NumTrkPt500	905	148.8	11.8	12.6
HardScatterVerticesAuxDyn.incomingParticleLinks	693	118.5	1.3	90.2
AnalysisLargeRJetsAuxDyn.constituentLinks	950.5	111.4	6.4	17.4

Table 4.8: Top 10 branches sorted by total file size in bytes, MC PHYSLITE [Athena v24.0.16 without limit to the basket buffer.]

Athena v24.0.16 (default) MC branch	Branch size (kB)	Total-file-size (MB)	Compressed-file-size (MB)	Compression factor
PrimaryVerticesAuxDyn.trackParticleLinks	128	2146.2	24.0	89.4
AnalysisJetsAuxDyn.GhostTrack	128	413.8	13.1	31.5
AnalysisJetsAuxDyn.NumTrkPt500	128	148.8	11.9	12.5
HLTNav_Summary_DAODSlimmedAuxDyn.linkColNames	128	784.0	11.9	65.7
HLTNav_Summary_DAODSlimmedAuxDyn.linkColKeys	128	390.6	11.7	33.4
HLTNav_Summary_DAODSlimmedAuxDyn.linkColClids	128	390.6	10.7	36.6
AnalysisJetsAuxDyn.NumTrkPt1000	128	148.8	8.7	17.2
AnalysisJetsAuxDyn.SumPtChargedPFOPt500	128	148.9	7.3	20.5
AnalysisLargeRJetsAuxDyn.constituentLinks	128	111.5	7.1	15.8
HLTNav_Summary_DAODSlimmedAuxDyn.name	128	80.8	4.4	18.4

Table 4.9: Top 10 branches sorted by compressed file size in bytes, MC PHYSLITE [Athena v24.0.16 default configuration.]

Athena v24.0.16 (no-lim) MC branch	Branch size (kB)	Total-file-size (MB)	Compressed-file-size (MB)	Compression factor
PrimaryVerticesAuxDyn.trackParticleLinks	1293.5	2145.5	22.9	93.5
AnalysisJetsAuxDyn.GhostTrack	1293.5	413.5	13.0	31.9
HLTNav_Summary_DAODSlimmedAuxDyn.linkColNames	1293.5	783.5	11.9	65.8
AnalysisJetsAuxDyn.NumTrkPt500	905	148.8	11.8	12.6
HLTNav_Summary_DAODSlimmedAuxDyn.linkColKeys	1293.5	390.3	11.3	34.5
HLTNav_Summary_DAODSlimmedAuxDyn.linkColClids	1293.5	390.3	11.0	35.5
AnalysisJetsAuxDyn.NumTrkPt1000	905	148.8	8.5	17.6
AnalysisJetsAuxDyn.SumPtChargedPFOPt500	905.5	148.8	6.8	21.9
AnalysisLargeRJetsAuxDyn.constituentLinks	950.5	111.4	6.4	17.4
HLTNav_Summary_DAODSlimmedAuxDyn.name	242	80.8	4.5	18.0

Table 4.10: Top 10 branches sorted by compressed file size in bytes, MC PHYSLITE [Athena v24.0.16 without limit to the basket buffer.]

Some branches, like *HLTNav Summary DAODSlimmedAuxDyn.linkColNames* show highly compressible behavior and are consistent with the other job configurations (data, MC, PHYS, and PHYSLITE). Further work could investigate these branches for further optimization of derivation jobs.

4.2.4 Conclusion to derivation job optimization

Initially, limiting the basket buffer size looked appealing; after the 128 kB basket buffer size limit was set, the compression ratio would begin to plateau, increasing the memory-usage without saving much in disk-usage. The optimal balance is met with the setting of 128 kB basket buffers for derivation production.

Instead, by removing the upper limit of the basket size, a greater decrease in DAOD output file size is achieved. The largest decrease in file size came from the PHYSLITE MC derivation jobs without setting an upper limit to the basket buffer size. While similar decreases in file size appear for derivation jobs using data, it is not as apparent for data as it is for MC jobs. With the removal of an upper-limit to the basket size, ATLAS stands to gain a 5% decrease for PHYSLITE MC DAOD output file sizes, but an $11-12\%$ increase in memory usage could prove a heavy burden (See Tables 2 and 4).

By looking at the branches per configuration, specifically in MC PHYSLITE output DAOD, highly compressible branches emerge. The branches inside the MC PHYSLITE DAOD are suboptimal as they do not conserve disk space; instead, they consume memory inefficiently. As seen from (Table 5) through (Table 10), we have plenty of branches in MC PHYSLITE that are seemingly empty–as indicated by the compression factor being $\mathcal{O}(10)$. Reviewing and optimizing the branch data could further reduce GRID load during DAOD production by reducing the increased memory-usage while keeping the effects of decreased disk-space.

CHAPTER 5

MODERNIZING I/O CI UNIT-TESTS

Athena uses a number of unit tests during the development lifecycle to ensure core I/O functionality does not break. Many of the I/O tests were originally created for the old EDM and haven't been updated to test the xAOD EDMs core I/O functions. This project took in track information from a unit test using the T/P EDM, writes the data into an example xAOD object to file and reads it back.

5.1 xAOD Test Object

The object used to employ the new unit test is the `xAOD::ExampleElectron` object, where the `xAOD::` is a declaration of the namespace and simply identifies the object as an xAOD object. An individual `ExampleElectron` object only has a few parameters for sake of testing, its transvese momentum, `pt`, and its charge, `charge`. A collection of `ExampleElectron` objects are stored in the `ExampleElectronContainer` object, which is just a `DataVector` of `ExampleElectron` objects.[4] This `DataVector<xAOD::ExampleElectron>` acts similar to a `std::vector<xAOD::ExampleElectron>`, but has additional code to handle the separation of interface and auxiliary data storage.

The xAOD EDM uses an abstract interface connecting between the `DataVector` and the auxiliary data, this is the `IAuxStore`. The function `setStore` is responsible for ensuring the auxillary data store is matched with it's corresponding `DataVector`. Another feature to the xAOD EDM is the ability to have a dynamic store of auxiliary data. This separates the auxiliary data between static and dynamic data stores. Where the static dta stores comprise of known variables and the dynamic counterpart stores data of variables not declared but still might be needed by the user. Figure 5.1 illustrates how a simple setup of storing a `DataVector` of electrons that hold some specific parameters into one `IAuxStore` while also having a separate `IAuxStore` specifically for the dynamic attributes.

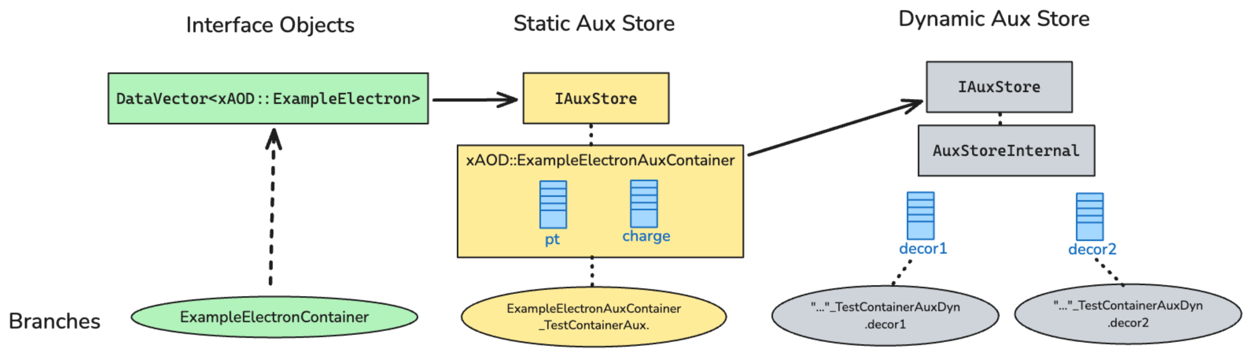

Figure 5.1: The framework between interface objects and the static/dynamic auxiliary data store for a collection of xAOD::ExampleElectrons.

5.2 Unit Tests

Unit tests are programs that act as a catch during the continuous integration of a codebase and exhaust features that need to remain functional. Athena has a number of unit tests which check every merge request and nightly build for issues in the new code that could break core functionality, either at the level of Athena, ROOT, or any other software in the LCG stack. With the adoption of the xAOD EDM, there were no unit tests to cover core I/O functionality related to this new EDM.

Specifically there were no unit tests to handle selection of dynamic attributes, or decorations, on xAOD objects created during writing and read back. To address this, a new xAOD test object needed to be created and written during a new unit test that fit into the existing unit tests. The list of AthenaPoolExample unit tests that are currently executed during a nightly build can be found in Table 5.1. These tests are executed in this order, as the objects created in one might be used in proceeding test.

Unit Test	**Employed Algorithms**
Write	WriteData
ReadWrite	ReadData
Read	ReadData
Copy	None
ReadWriteNext	ReadData, ReWriteData
WritexAODElectron	ReadData, WriteExampleElectron
ReadxAODElectron	ReadExampleElectron
ReadAgain	ReadData
WriteConcat	WriteData, ReWriteData
ReadConcat	ReadData
WriteCond	ReadData, WriteCond
ReadCond	ReadData, ReadCond
WriteMeta	WriteData, WriteCond
ReadMeta	ReadData

Table 5.1: List of unit tests in the AthenaPoolExample package that are currently executed during a nightly build.

The mechanism for passing a unit test is done automatically by building the framework, running the unit tests, and comparing the diff of the output file to the unit test with a reference file associated with that particular unit test. If the unit test passes, then the diff, a product of the `git diff` command, will be empty and the unit test will be marked as passing. Conversely, if the unit test fails, then the diff will be non-empty and the unit test will be marked as failing.

5.2.1 WritexAODElectron.py

The two new tests added to the package were `WritexAODElectron` and `ReadxAODElectron`. During this first unit test, the first algorithm called is to `ReadData` which reads off all of the `ExampleTrack` objects stored in one of the files produced by the `ReadWrite` unit-test.

Within the python script of the first unit test, the user is able to decide what decorations to have written to file. This is a part of the OutputStreamCfg parameter, ItemList, wherein the user specifies the object and its name in the format shown in Figure 5.2.

```
ItemList = [ "ExampleTrackContainer#MyTracks",
"xAOD::ExampleElectronContainer#TestContainer",
"xAOD::ExampleElectronAuxContainer#TestContainerAux.-decor2"] )
```

Figure 5.2: WritexAODElectron header file setup

Much of the code in Athena practice header/source-file separation, the header acting as an interface for the whole object and the source file containing the core functionality of the algorithm. The header file includes various packages needed by the algorithm, such as data objects, `Write/ReadHandleKeys`, base algorithms that give consistent structure to the algorithm, and whatever else is required. In the write-algorithm, there are `ReadHandleKeys` for `ExampleTrack` objects saved by a prior unit test. For the `WriteHandleKeys`, there is one for the `ExampleElectronContainer` and the name given to it is “TestContainer”. This “TestContainer” name will be needed for the `ReadExampleElectron` algorithm as the name is how it’s able to refer to the correct `ExampleElectronContainer` present in the input file. Additionally, a `WriteHandleDecorKey` for the decoration objects is needed for appending each decoration onto each `ExampleElectron` object. Figure 5.3 shows the syntax for how these keys would be presently defined.

Then the `WriteExampleElectron` algorithm is called and takes `ExampleTracks`, creates an `ExampleElectron` object and sets the electrons `pt` to the tracks `pt`. As shown in Figure 5.4, the `ExampleElectronContainer` and `ExampleElectronAuxContainer` are created and set to the `elecCont` and `elecStore` respectively. The `elecCont` has an associated aux store, so the `setStore` function is called with the `elecStore` pointer. The track container is accessed by using StoreGate’s `ReadHandle`, which associates the `m_exampleTrackKey` with the `ExampleTrackContainer` specified in the header file. This is then looped over all elements

```cpp
// Read key ExampleTracks
SG::ReadHandleKey<ExampleTrackContainer> m_exampleTrackKey{
    this, "ExampleTrackKey", "MyTracks"};

// Write key for the ExampleElectronContainer
SG::WriteHandleKey<xAOD::ExampleElectronContainer>
    m_exampleElectronContainerKey{this, "ExampleElectronContainerName",
                                        "TestContainer"};

// Decoration keys
SG::WriteDecorHandleKey<xAOD::ExampleElectronContainer> m_decor1Key{
    this, "ExampleElectronContainerDecorKey1", "TestContainer.decor1",
    "decorator1 key"};
SG::WriteDecorHandleKey<xAOD::ExampleElectronContainer> m_decor2Key{
    this, "ExampleElectronContainerDecorKey2", "TestContainer.decor2",
    "decorator2 key"};
```

Figure 5.3: WritexAODElectron header file setup

in the container and the `pt` of each track is set to the `pt` of the electron. A `WriteHandle`, called `objs`, is then created for the container of `ExampleElectrons` which is then recorded.

Within the same algorithm, the next step is to loop over each of the newly produced `ExampleElectrons`, accessing the decorations `decor1` and `decor2`, and setting each to an arbitrary float value that are easily identifiable later. Figure 5.5 shows how this is done using two handles for each decoration. Note the difference here using the `WriteDecorHandle`, where the prior handle type was `WriteHandle`.

5.2.2 ReadxAODElectron.py

The only algorithm called in this test is `ReadExampleElectron`. The header file for the `ReadExampleElectron` only creates `ReadHandleKey` for the container of ExampleElectrons, with the same name from the header of the `WriteExampleElectron` algorithm header, syntax shown in Figure 5.6. Following this, the `ReadxAODElectron` unit test will demonstrate the

```
auto elecCont = std::make_unique<xAOD::ExampleElectronContainer>();
auto elecStore = std::make_unique<xAOD::ExampleElectronAuxContainer>();
elecCont->setStore(elecStore.get());

SG::ReadHandle<ExampleTrackContainer> trackCont(m_exampleTrackKey, ctx);
elecCont->push_back(std::make_unique<xAOD::ExampleElectron>());

for (const ExampleTrack* track : *trackCont) {
    // Take on the pT of the track
    elecCont->back()->setPt(track->getPT());
}

SG::WriteHandle<xAOD::ExampleElectronContainer> objs(
    m_exampleElectronContainerKey, ctx);
ATH_CHECK(objs.record(std::move(elecCont), std::move(elecStore)));
```

Figure 5.4: Algorithm to initialize and write T/P data (ExampleTracks) to an xAOD object container (ExampleElectronContainer).

```
SG::WriteDecorHandle<xAOD::ExampleElectronContainer, float> hdl1(
    m_decor1Key,ctx);
SG::WriteDecorHandle<xAOD::ExampleElectronContainer, float> hdl2(
    m_decor2Key,ctx);

for (const xAOD::ExampleElectron* obj : *objs) {
    hdl1(objs) = 123.;
    hdl2(objs) = 456.;
}
```

Figure 5.5: Writing of dynamic variables for each of the ExampleElectron objects.

xAOD function of accessing one dynamic variable, in this case `decor1`, while not accessing the other, `decor2`. This is done by

5.3 Results

```
SG::ReadHandleKey<xAOD::ExampleElectronContainer>
m_exampleElectronContainerKey{this, "ExampleElectronContainerName",
                                   "TestContainer"};
```

Figure 5.6: ReadHandleKey for the container of ExampleElectrons

CHAPTER 6

CONCLUSION

The toy model testing allowed us to create branches with data similar compression ratios to real and simulated data, allowing to investigate the hypothesis that modifying the basket buffer limit had an effect on disk and memory usage. It led to the conclusion that, upon investigating with real data and real MC simulation, that there might be an avenue to look at both ROOT and Athena to limit basket sizes. Modifying the basket buffer sizes at the Athena level shows there was a balance struck

This study also illuminated the possibilty at a class of unoptimized branches in MC simulated data, from which it was not clear

The xAOD EDM comes with a number of new additions to bring about optimization the future of analysis work at the ATLAS experiment. Integrating the new features into a few comprehensive unit tests allow for the nightly CI builds to catch any issues that break core I/O functionality as it pertains to the xAOD EDM, which has not been done before. These new unit-tests exercise reading and writing select decorations ontop of the already existing data structures attacted to an example object called `ExampleElectron`.

BIBLIOGRAPHY

[1] A. Abdesselam et al. “ATLAS pixel detector electronics and sensors”. In: *Journal of Instrumentation* 3.07 (July 2008), P07007. DOI: 10.1088/1748-0221/3/07/P07007. URL: https://dx.doi.org/10.1088/1748-0221/3/07/P07007.

[2] A. Abdesselam et al. “The barrel modules of the ATLAS semiconductor tracker”. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 568.2 (2006), pp. 642–671. ISSN: 0168-9002. DOI: https://doi.org/10.1016/j.nima.2006.08.036. URL: https://www.sciencedirect.com/science/article/pii/S016890020601388X.

[3] I. Bejar Alonso et al. “High-Luminosity Large Hadron Collider (HL-LHC): Technical design report”. In: 10 (2020), p. 390. DOI: https://doi.org/10.23731/CYRM-2020-0010. URL: https://e-publishing.cern.ch/index.php/CYRM/issue/view/127.

[4] A Buckley et al. “Implementation of the ATLAS Run 2 event data model”. In: *Journal of Physics: Conference Series* 664.7 (Dec. 2015), p. 072045. DOI: 10.1088/1742-6596/664/7/072045. URL: https://dx.doi.org/10.1088/1742-6596/664/7/072045.

[5] A. Buckley et al. *Report of the xAOD Design Group*. 2013. URL: https://cds.cern.ch/record/1598793/files/ATL-COM-SOFT-2013-022.pdf.

[6] ATLAS Experiment at CERN. *Trigger and Data Acquisition*. URL: https://atlas.cern/Discover/Detector/Trigger-DAQ.

[7] Jean-Luc Caron for CERN. *LHC Illustration showing underground locations of detectors*. 1998. URL: https://research.princeton.edu/news/princeton-led-group-prepares-large-hadron-collider-bright-future.

[8] The ATLAS TRT collaboration et al. "The ATLAS Transition Radiation Tracker (TRT) proportional drift tube: design and performance". In: *Journal of Instrumentation* 3.02 (Feb. 2008), P02013. DOI: 10.1088/1748-0221/3/02/P02013. URL: https://dx.doi.org/10.1088/1748-0221/3/02/P02013.

[9] Beniamino Di Girolamo and Marzio Nessi. *ATLAS undergoes some delicate gymnastics*. 2013. URL: https://cerncourier.com/a/atlas-undergoes-some-delicate-gymnastics/.

[10] ATLAS software group. *Athena*. URL: https://doi.org/10.5281/zenodo.2641997.

[11] ATLAS software group. *Athena Software Documentation*. URL: https://atlassoftwaredocs.web.cern.ch/athena/.

[12] Glenn F. Knoll. *Radiation Detection and Measurement*. New York: John Wiley & Sons, Inc., 2010.

[13] P J Laycock et al. "Derived Physics Data Production in ATLAS: Experience with Run 1 and Looking Ahead". In: *Journal of Physics: Conference Series* 513.3 (June 2014), p. 032052. DOI: 10.1088/1742-6596/513/3/032052. URL: https://dx.doi.org/10.1088/1742-6596/513/3/032052.

[14] Ana Lopes and Melissa Loyse Perry. *FAQ-LHC The guide*. 2022. URL: https://home.cern/resources/brochure/knowledge-sharing/lhc-facts-and-figures.

[15] Bartosz Mindur. *ATLAS Transition Radiation Tracker (TRT): Straw tubes for tracking and particle identification at the Large Hadron Collider*. Geneva, 2017. DOI: 10.1016/j.nima.2016.04.026. URL: https://cds.cern.ch/record/2139567.

[16] ATLAS Outreach. "ATLAS Fact Sheet : To raise awareness of the ATLAS detector and collaboration on the LHC". 2010. DOI: 10.17181/CERN.1LN2.J772. URL: https://cds.cern.ch/record/1457044.

[17] Schaarschmidt, Jana et al. “PHYSLITE - A new reduced common data format for ATLAS”. In: *EPJ Web of Conf.* 295 (2024), p. 06017. DOI: `10.1051/epjconf/202429506017`. URL: `https://doi.org/10.1051/epjconf/202429506017`.

[18] ROOT Team. *ROOT, About*. URL: `https://root.cern/about/`.

[19] ROOT Team. *ROOT, TTree Class*. 2024. URL: `https://root.cern.ch/doc/master/classTTree.html`.

APPENDIX

DERIVATION PRODUCTION DATA

A.1 Derivation production datasets

For both the nightly and the release testing, the data derivation job, which comes from the dataset

```
data22_13p6TeV:data22_13p6TeV.00428855.physics_Main.merge.AOD.
    r14190_p5449_tid31407809_00
```

was ran with the input files

```
AOD.31407809._000894.pool.root.1
AOD.31407809._000895.pool.root.1
AOD.31407809._000896.pool.root.1
AOD.31407809._000898.pool.root.1
```

Similarly, the MC derivation job, comes from the dataset

```
mc23_13p6TeV:mc23_13p6TeV.601229.PhPy8EG_A14_ttbar_hdamp258p75_
    SingleLep.merge.AOD.e8514_e8528_s4162_s4114_r14622_r14663_
    tid33799166_00
```

was ran with input files

```
AOD.33799166._000303.pool.root.1
AOD.33799166._000304.pool.root.1
AOD.33799166._000305.pool.root.1
AOD.33799166._000306.pool.root.1
AOD.33799166._000307.pool.root.1
AOD.33799166._000308.pool.root.1
```