

ABSTRACT

1 OPTIMIZATION OF DERIVATION JOBS AND MODERNIZATION OF NIGHTLY CI 2 BUILD I/O TESTS FOR THE ATLAS EXPERIMENT

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

7 The ATLAS experiment's Software Performance Optimization Team has efforts in devel-
8 oping the Athena software framework that is scalable in performance and ready for wide-
9 spread use during Run-3 and HL-LHC data ready to be used for Run-4. It's been shown
10 that the storage bias for TTree's during derivation production jobs can be improved upon
11 compression and stored to disk by about 4-5% by eliminating the basket capping, with a
12 simultaneous increase in memory usage by about 11%. Additionally, job configuration allows
13 opportunity to improve many facets of the ATLAS I/O framework.

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
© 2025 Arthur C. Kraus

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

DEPARTMENT OF PHYSICS

Dissertation Director:
Dr. Jahred Adelman

ACKNOWLEDGEMENTS

16 Here's where you acknowledge folks who helped. Here's where you acknowledge folks
17 who helped. Here's where you acknowledge folks who helped. Here's where you acknowledge
18 folks who helped.

DEDICATION

19

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

Page

21	LIST OF TABLES	vi
22	LIST OF FIGURES.	vii
	Chapter	
23	1 INTRODUCTION	1
24	1.1 Particle Physics and the Large Hadron Collider	1
25	1.2 LHC and The ATLAS Detector	1
26	1.3 ATLAS Trigger and Data Acquisition	4
27	2 ATLAS I/O	5
28	2.1 Athena and ROOT	5
29	2.2 TTree Object	5
30	2.3 Derivation Production Jobs	6
31	2.4 Event Data Models	6
32	2.4.1 Transient/Persistent (T/P) EDM.	7
33	2.4.2 xAOD EDM.	7
34	3 TOY MODEL BRANCHES.	8
35	3.1 Toy Model Compression.	8
36	3.1.1 Random Float Branches	8
37	3.1.2 Mixed-Random Float Branches	14
38	3.2 Basket-Size Investigation	18
39	4 DATA AND MONTE CARLO DERIVATION PRODUCTION	23
40	4.1 Current Derivation Framework	23

41	Chapter	Page
42	4.2 Performance Metrics and Benchmarking	23
43	4.3 Results.	24
44	4.3.1 Presence of basket-cap and presence of minimum number of entries. .	24
45	4.3.2 Comparing different basket sizes	25
46	4.3.3 Monte Carlo PHYSLITE branch comparison.	26
47	4.3.4 Conclusion to derivation job optimzation	29
48	5 MODERNIZING I/O CI UNIT-TESTS	30
49	5.1 Continuous integration unit tests	30
50	APPENDIX: DERIVATION PRODUCTION DATA.	33

LIST OF TABLES

Table	Page
4.1 Athena v22.0.16: Comparing the maximum proportional set size (PSS) and PHYS/PHYSLITE output file sizes (outFS) for data jobs over various Athena configurations for 160327 entries.	25
4.2 Athena v22.0.16: Comparing the maximum proportional set size (PSS) and PHYS/PHYSLITE output file sizes (outFS) for MC jobs over various Athena configurations for 140000 entries.	25
4.3 Athena v24.0.16: Comparing the maximum proportional set size (PSS) and PHYS/PHYSLITE output file sizes (outFS) for Data jobs over various Athena configurations for 160327 entries.	26
4.4 Athena v24.0.16: Comparing the maximum proportional set size (PSS) and PHYS/PHYSLITE output file sizes (outFS) for MC jobs over various Athena configurations for 140000 entries.	26
4.5 Top 10 branches sorted by compression factor, MC PHYSLITE [Athena v24.0.16 default configuration.]	27
4.6 Top 10 branches sorted by compression factor, MC PHYSLITE [Athena v24.0.16 without limit to the basket buffer.]	27
4.7 Top 10 branches sorted by total file size in bytes, MC PHYSLITE [Athena v24.0.16 default configuration.]	27
4.8 Top 10 branches sorted by total file size in bytes, MC PHYSLITE [Athena v24.0.16 without limit to the basket buffer.]	27
4.9 Top 10 branches sorted by compressed file size in bytes, MC PHYSLITE [Athena v24.0.16 default configuration.]	28
4.10 Top 10 branches sorted by compressed file size in bytes, MC PHYSLITE [Athena v24.0.16 without limit to the basket buffer.]	28

LIST OF FIGURES

Figure		Page
77	1.1 Illustration of the LHC experiment sites on the France-Switzerland border.	
78	[3]	2
79	1.2 Overview of the ATLAS detectors main components. [5].	3
80	3.1 Compression factors of $N = 1000$ entries per branch with random-valued	
81	vectors of varying size.	14
82	3.2 Compression Ratios for ($\frac{1}{2}$ random) and ($\frac{1}{4}$ random) branches at ($N = 10^6$	
83	events)	17
84	3.3 Compression Ratios for ($\frac{1}{2}$ random) and ($\frac{1}{4}$ random) branches at ($N = 10^5$	
85	events)	18
86	3.4 Compression Factors vs Branch Size (1/2 Mixture $N = 10^6$ events)	19
87	3.5 Number of Baskets vs Branch Size (1/2 Mixture $N = 10^6$ events)	20
88	3.6 Varying Mixtures in 8 point precision - Number of Baskets vs Branch Size	
89	($N = 10^6$ events)	21
90	3.7 Varying Mixtures in 16 point precision - Number of Baskets vs Branch Size	
91	($N = 10^6$ events)	22

CHAPTER 1

INTRODUCTION

1.1 Particle Physics and the Large Hadron Collider

Particle physics is the branch of physics that seeks out the origins of the universe by probing and searching for new interactions at the highest energies. 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 LHC, shown in Figure 1.1 is a 26.7-kilometer ring that crosses between the France-Switzerland border at a depth between 50 and 175 meters underground.[10] 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

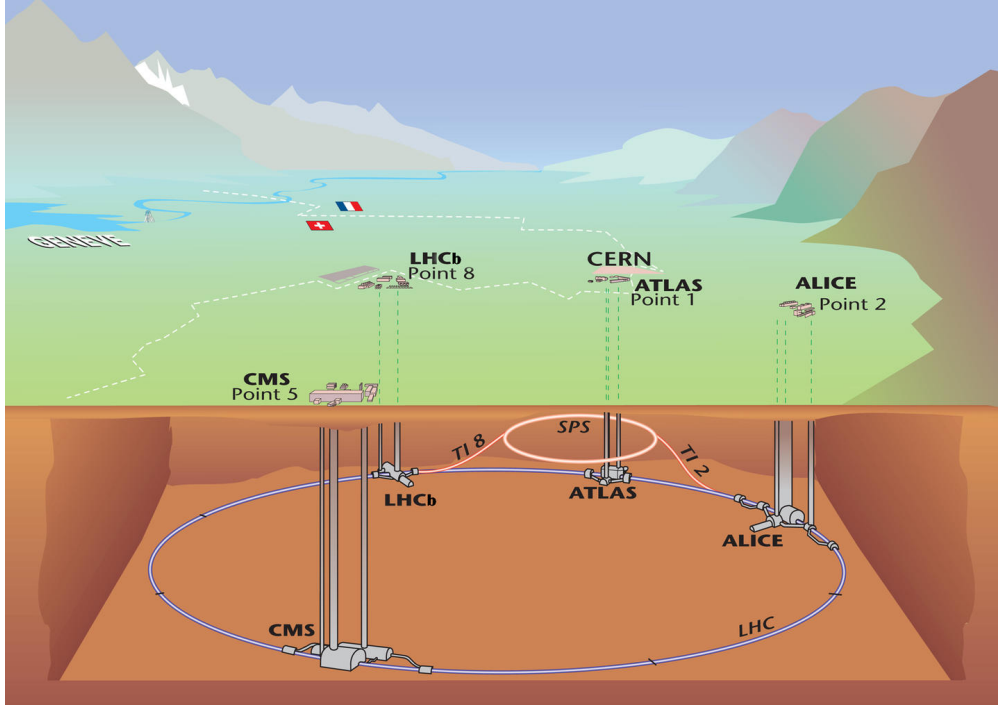


Figure 1.1: Illustration of the LHC experiment sites on the France-Switzerland border. [3]

high and 25 meters wide.[12] The ATLAS 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. It's main function is to measure the track of the charged particles without destroying the particle itself. The first point of contact for ATLAS is the pixel detector. It has over 92 million pixels and is radiation hard to aid in particle track and vertex reconstruction.[8] When charged particles pass through a pixel sensor ionizes silicon which produces an electron-hole pair, and this generates an electric current that can be measured. [4] Surrounding the pixel detector is the semiconductor tracker, which uses 4,088 modules of 6 million implanted silicon readout strips. The semiconductor tracker helps measure the path particles take, called tracks, with 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. Particles with

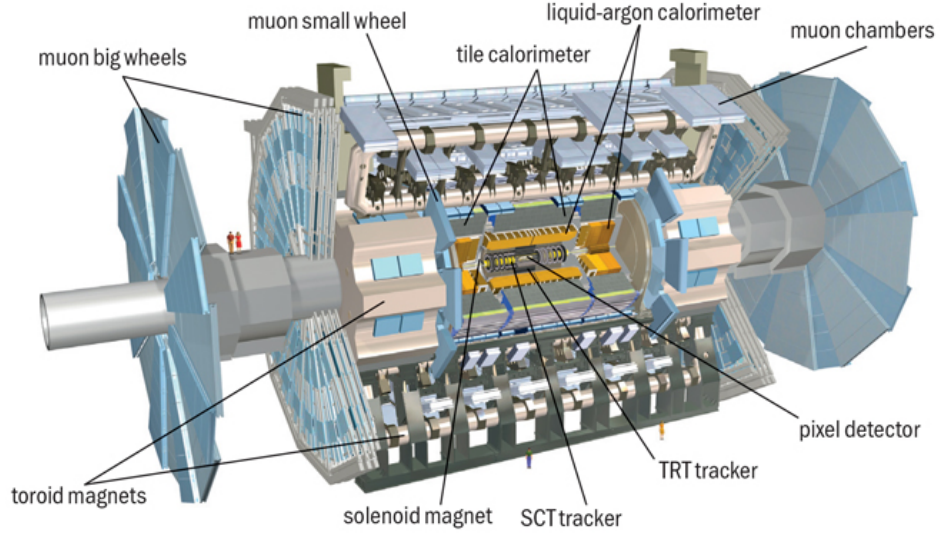


Figure 1.2: Overview of the ATLAS detectors main components. [5]

relativistic velocities have higher Lorentz γ -factors, see Eq. (1.1), the TRT uses varying materials to discriminate between heavier particles (with low γ and radiate less) and lighter particles (higher γ and radiate more). [11]

$$\gamma = \frac{1}{\sqrt{1 - \frac{v^2}{c^2}}} \quad (1.1)$$

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. If a particle hit the steel, they will then generate a new shower of particles 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 what’s 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 Acquisition 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 occurs 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. [2] 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. [12] Reconstructed data reduces the size per event to 1 MB. Reconstructed AOD are then processed through derivation jobs that reduced AODs to $\mathcal{O}(10)$ kB per event, creating Derived AOD (DAOD).

CHAPTER 2

ATLAS I/O

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.

2.1 Athena and ROOT

Athena is the open-source software framework for the ATLAS experiment.[6] 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.[7]

ROOT is an open-source software framework used for high-energy physics analysis at CERN.[13] 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.

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

172 buffers, upon compression, become baskets. These baskets can have a limited size and a set
 173 minimum number of entries. The Athena default basket size at present is 128 kB, and the
 174 default minimum number of entries is 10.

175 One function relevant to TTree is *Fill()*. *Fill()* will loop over all of the branches in the
 176 TTree and compresses the baskets that make up the branch. This removes the basket from
 177 memory as it is then compressed and written to disk. It makes reading back branches faster
 178 as all of the baskets are stored near each other on the same disk region. [14]

179 2.3 Derivation Production Jobs

180 A derivation job is that process that takes AODs, which comes from the reconstruction
 181 step at $\mathcal{O}(1 \text{ MB})$ per event, and creates a Derived Analysis Object Data (DAOD) which sits
 182 at $\mathcal{O}(10 \text{ kB})$ per event.

183 2.4 Event Data Models

184 An Event Data Model (EDM) is a collection of classes and their relationships to each
 185 other that provide a representation of an event detected with the goal of making it easier
 186 to use and manipulate by developers. An EDM is how particles and jets are represented in
 187 memory, stored to disk, and manipulated in analysis. It's useful to have an EDM because
 188 it brings a commonality to the code, which is useful when developers reside in different
 189 groups with various backgrounds. An EDM allows those developers to more easily debug
 190 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 nature of AOD. The AOD at this point was converted into an ntuple based format called D3PDs. While this conversion allowed for fast readability and partial read for efficient analysis in ROOT, it left the files disconnected from the reconstruction tools found in Athena.[1] The transient data was present in memory and could have information attached to the object, this data could gain complexity the more it was used. Persistent data needed to be simplified before it could be persisted 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. 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 usable both on Athena and ROOT. It's easier to pick up for analysis and reconstruction. xAOD EDM has the ability to add and remove variables at runtime, these variables are called “decorations.”

CHAPTER 3

TOY MODEL BRANCHES

A toy model of AOD provides a simple-to-understand representation for how real data and Monte Carlo (MC) simulated data will react under optimization conditions for derivation production jobs. One commonality between both data and MC is the branch data within both is made of a mixture between repeated integer-like data and randomized floating-point data (i.e. data that has both easily and difficult to compress.) Replicating this mixture of data 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 demand on 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. 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.

226 The script file can be compiled with `gcc` and it requires all of the dependencies that
 227 come with ROOT. To begin this script, there are a number of included ROOT and C++ standard
 228 library headers.

```

229 1 // C++ Standard Library
230 2 #include <iostream>
231 3 #include <memory>
232 4 #include <ostream>
233 5 #include <vector>
234 6
235 7 // Necessary ROOT Headers
236 8 #include "TBranch.h"
237 9 #include "TCanvas.h"
238 0 #include "TFile.h"
239 1 #include "TH1.h"
240 2 #include "TRandom.h"
241 3 #include "TStyle.h"
242 4 #include "TTree.h"
243 5 ...

```

244 The following function `VectorTree()` is the main function in this code. What is needed
 245 first is an output file, which will be called `VectorTreeFile.root`, and the name of the tree
 246 can simply be `myTree`.

```

247 1 void VectorTree() {
248 2     std::unique_ptr<TFile> myFile =
249 3     std::make_unique<TFile>("VectorTreeFile.root", "RECREATE");
250 4     TTree *tree = new TTree("myTree", "myTree");
251 5     ...
252 6 }

```

253 Initializing variables can start with the total number of events (total number of vectors)
 254 in each branch, `N`. Additionally the branches have a number of floats per vector, this size will

need to be defined as `NEntries0`, `NEntries1`, 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.

```

258 1 void VectorTree() {
259 2     ...
260 3     const int N = 1e4; // N = 1000
261 4     // Set Number of Entries with 10^# of random floats
262 5     int NEntries0 = 1;
263 6     int NEntries1 = 10;
264 7     int NEntries2 = 100;
265 8     int NEntries3 = 1000;
266 9
267 10    // vectors
268 11    std::vector<float> vec_ten0; // 10^0 = 1 entry
269 12    std::vector<float> vec_ten1; // 10^1 = 10 entries
270 13    std::vector<float> vec_ten2; // 10^2 = 100 entries
271 14    std::vector<float> vec_ten3; // 10^3 = 1000 entries
272 15
273 16    // variables
274 17    float float_0;
275 18    float float_1;
276 19    float float_2;
277 20    float float_3;
278 21    ...
279 22 }

```

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

```

282 1 void VectorTree() {
283 2     ...

```

```

284 3 // Initializing branches
285 4 std::cout << "creating branches" << std::endl;
286 5 tree->Branch("branch_of_vectors_size_one", &vec_ten0);
287 6 tree->Branch("branch_of_vectors_size_ten", &vec_ten1);
288 7 tree->Branch("branch_of_vectors_size_hundred", &vec_ten2);
289 8 tree->Branch("branch_of_vectors_size_thousand", &vec_ten3);
290 9 ...
291 10 }

```

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

```

293 1 void VectorTree() {
294 2 ...
295 3 tree->SetAutoFlush(0);
296 4 ...

```

297 `AutoFlush` is a function that tells the `Fill()` function after a designated number of entries
298 to flush all branch buffers from memory and save them to disk. Disabling `AutoFlush` allows
299 for more consistent compression across the various sizes of branches. The toy model needed
300 this consistency more than the later tests as these early tests were solely focused on mimicing
301 data procured by the detector and event simulation. Hence disabling of `AutoFlush` later on is
302 not practiced. Following branch initialization comes the event loop where data is generated
303 and emplaced into vectors.

```

304 1 void VectorTree() {
305 2 ...
306 3 // Events Loop
307 4 std::cout << "generating events..." << std::endl;
308 5 for (int j = 0; j < N; j++) {
309 6 // Clearing entries from previous iteration
310 7 vec_ten0.clear();
311 8 vec_ten1.clear();

```

```

3129      vec_ten2.clear();
3130      vec_ten3.clear();
3141
3152      // Generating vector elements, filling vectors
3163      // Fill vec_ten0
3174      for (int m = 0, m < NEntries0; m++) {
3185          float_0 = gRandom->Rndm() * 10; // Create random float value
3196          vec_ten0.emplace_back(float_0); // Emplace float into
320      vector
3217      }
3228      // Fill vec_ten1
3239      for (int n = 0, n < NEntries1; n++) {
3240          float_1 = gRandom->Rndm() * 10;
3251          vec_ten1.emplace_back(float_1);
3262      }
3273      // Fill vec_ten2
3284      for (int a = 0, a < NEntries2; a++) {
3295          float_2 = gRandom->Rndm() * 10;
3306          vec_ten2.emplace_back(float_2);
3317      }
3328      // Fill vec_ten3
3339      for (int b = 0, b < NEntries3; b++) {
3340          float_3 = gRandom->Rndm() * 10;
3351          vec_ten3.emplace_back(float_3);
3362      }
3373      tree->Fill(); // Fill our TTree with all the new branches
3384  }
3395      // Saving tree and file
3406      tree->Write();
3417      ...
3428  }

```

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. Since the branches had vectors with exclusively random floats, it becomes apparent that the more randomization in the branches the harder it is to compress. Filling vectors with entirely random values was believed to yield compression ratios close to real data, but from the results in Figure 3.1 it's clear some changes needed to be made to bring the branches closer to a compression ratio of $\mathcal{O}(5)$.

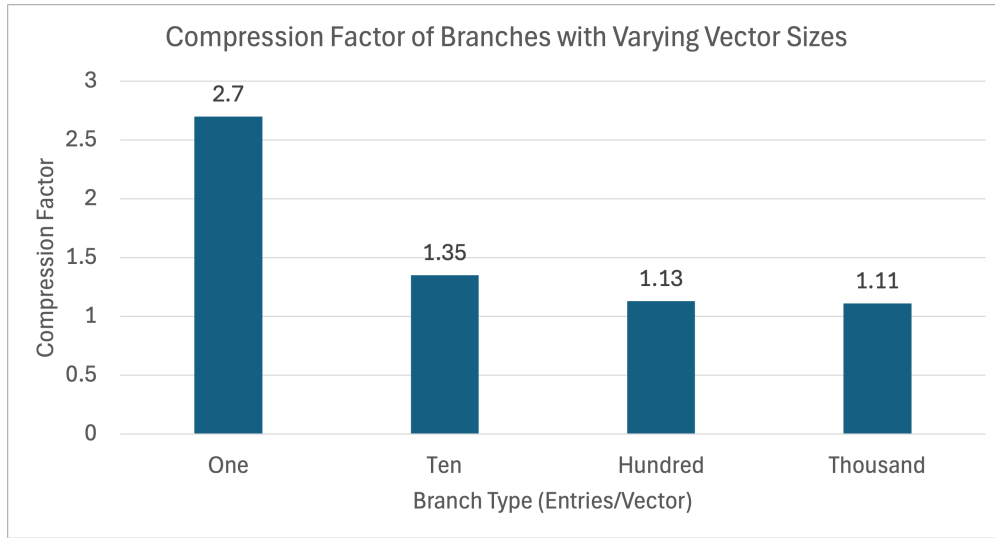


Figure 3.1: Compression factors of $N = 1000$ entries per branch with random-valued vectors of varying size.

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 $N = 1e4$ to $1e5$, or from 1000 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() {
    ...
}
```

```

384 3 // Events Loop
385 4 for (int j = 0; j < N; j++) {
386 5     // Clearing entries from previous iteration
387 6     vec_ten0.clear();
388 7     vec_ten1.clear();
389 8     vec_ten2.clear();
390 9     vec_ten3.clear();
391 10
392 11 // Generating vector elements, filling vectors
393 12 // Generating vec_ten0
394 13 for (int a = 0; a < NEntries0; a++) {
395 14     if (a < (NEntries0 / 2)) {
396 15         float_0 = gRandom->Gaus(0, 1) * gRandom->Rndm();
397 16         vec_ten0.emplace_back(float_0);
398 17     } else {
399 18         float_0 = 1;
400 19         vec_ten0.emplace_back(float_0);
401 20     }
402 21 }
403 22
404 23 // Generating vec_ten1
405 24 for (int b = 0; b < NEntries1; b++) {
406 25     if (b < NEntries1 / 2) {
407 26         float_1 = gRandom->Rndm() * gRandom->Gaus(0, 1);
408 27         vec_ten1.emplace_back(float_1);
409 28     } else {
410 29         float_1 = 1;
411 30         vec_ten1.emplace_back(float_1);
412 31     }
413 32 }
414 33

```

```

4154 // Generating vec_ten2
4165 for (int c = 0; c < NEntries2; c++) {
4176     if (c < NEntries2 / 2) {
4187         float_2 = gRandom->Rndm() * gRandom->Gaus(0, 1);
4198         vec_ten2.emplace_back(float_2);
4209     } else {
4210         float_2 = 1;
4221         vec_ten2.emplace_back(float_2);
4232     }
4243 }
4254
4265 // Generating vec_ten3
4276 for (int d = 0; d < NEntries3; d++) {
4287     if (d < NEntries3 / 2) {
4298         float_3 = gRandom->Rndm() * gRandom->Gaus(0, 1);
4309         vec_ten3.emplace_back(float_3);
4310     } else {
4321         float_3 = 1;
4332         vec_ten3.emplace_back(float_3);
4343     }
4354 }
4365 tree->Fill(); // Fill our TTree with all the new branches
4376 }
4387 // Saving tree and file
4398 tree->Write();
4409 ...
4410 }

```

442 As shown in the if-statements in lines 14, 25, 36 and 47, if the iterator was less than
 443 half of the total number of entries in the branch then that entry had a randomized float
 444 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. When the number of events is increased from $N = 10^5$ to $N = 10^6$, branches with only half of the mixture is random data become larger and the branches with more vectors per entry become more difficult to compress. 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.

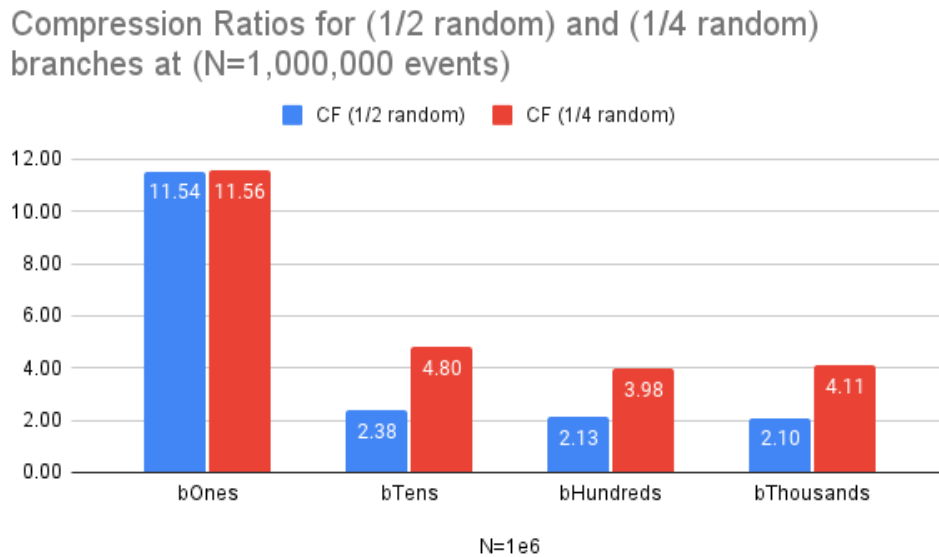


Figure 3.2: Compression Ratios for ($\frac{1}{2}$ random) and ($\frac{1}{4}$ random) branches at ($N = 10^6$ events)

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. Here is where tuning the basket size can begin to start.

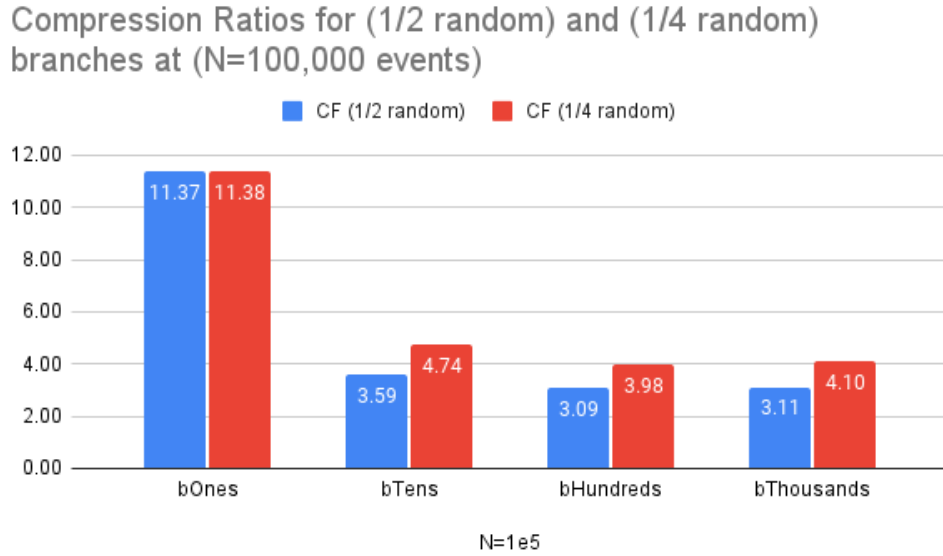


Figure 3.3: Compression Ratios for ($\frac{1}{2}$ random) and ($\frac{1}{4}$ random) branches at ($N = 10^5$ events)

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 a toy model; testing of the basket size setting both at the ROOT- and Athena-level would take later. 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.

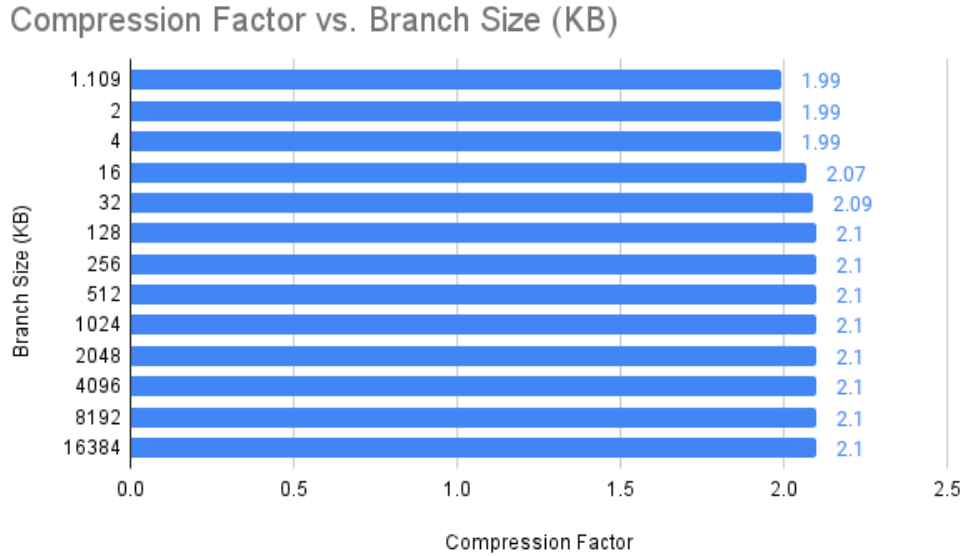


Figure 3.4: Compression Factors vs Branch Size (1/2 Mixture $N = 10^6$ events)

Figure 3.4 and Figure 3.5 is the first indication that the lower basket sizes are too small to effectively compress the data. For the baskets under 16 kB, it is required to have as many baskets as events to effectively store all the data—this will cause problems later on with memory usage so many of these basket sizes can be ignored.

There were more variations in the data that were looked at. For instance, looking further into the types of mixtures and how those mixtures would affect compression are shown in Figure 3.6. Another instance looked into the same mixtures but decreasing the precision of the floating point values that we used from the standard 32 floating-point precision to 16 and 8 which made 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.

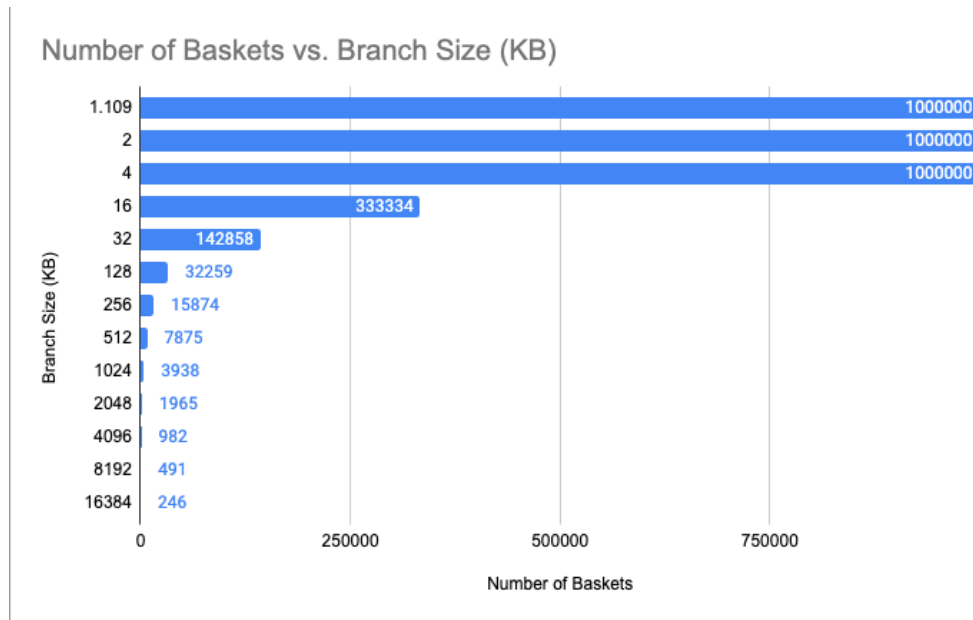


Figure 3.5: Number of Baskets vs Branch Size (1/2 Mixture $N = 10^6$ events)

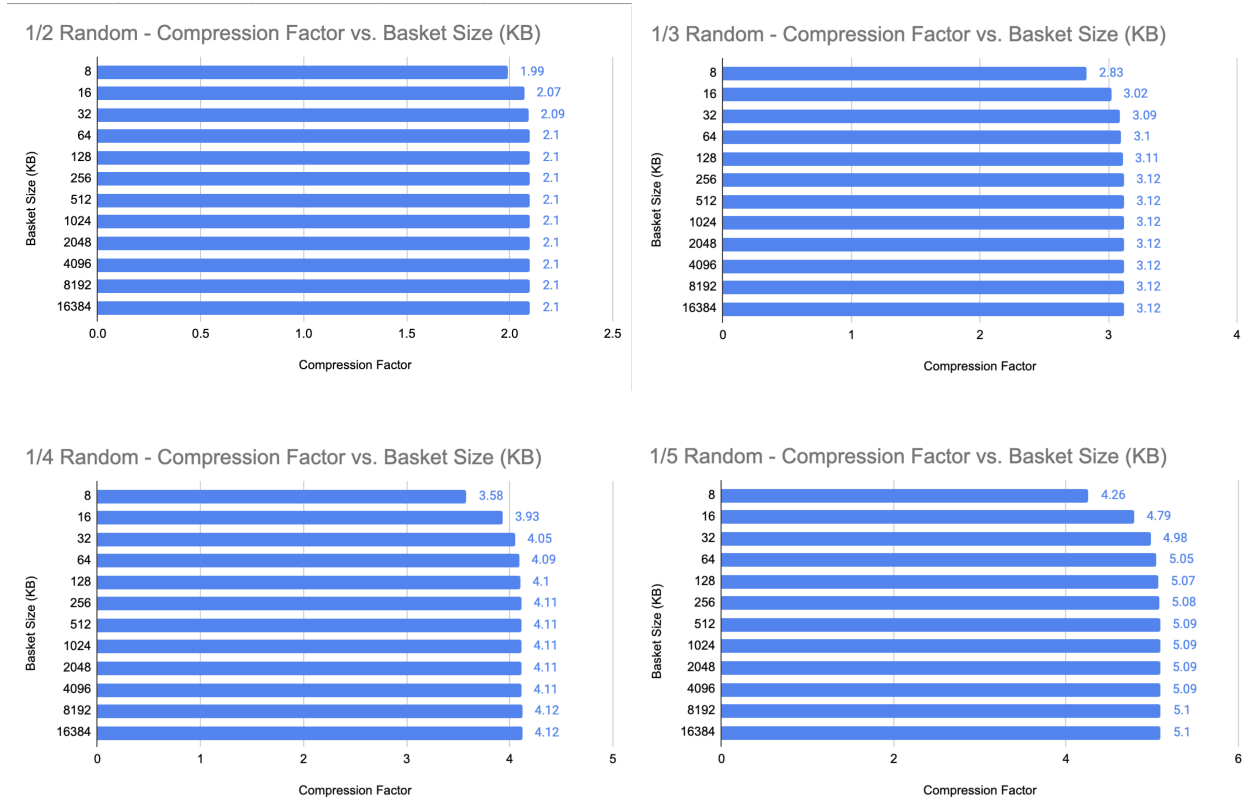


Figure 3.6: Varying Mixtures in 8 point precision - Number of Baskets vs Branch Size ($N = 10^6$ events)

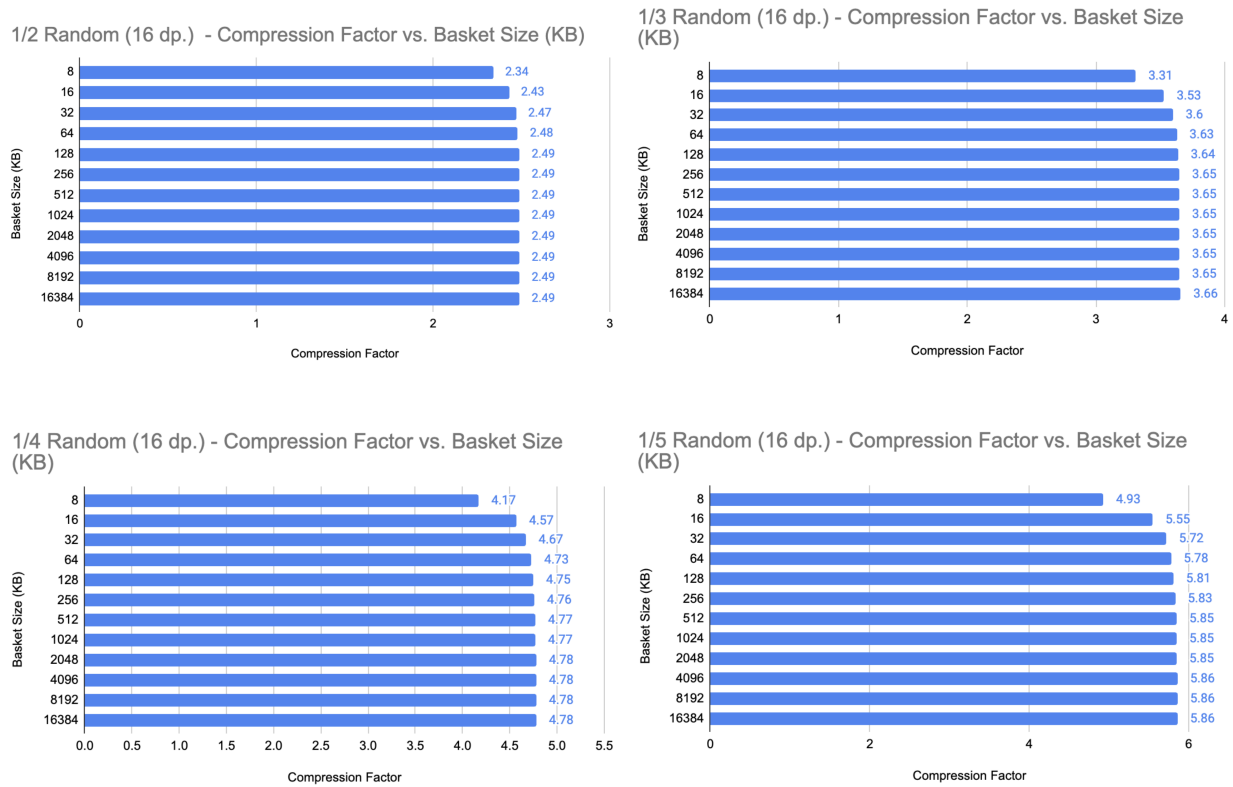


Figure 3.7: Varying Mixtures in 16 point precision - Number of Baskets vs Branch Size ($N = 10^6$ events)

CHAPTER 4

DATA AND MONTE CARLO DERIVATION PRODUCTION

4.1 Current Derivation Framework

Derivation production jobs suffer from 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 one will have to load a larger buffer into memory. The route we 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).

4.2 Performance Metrics and Benchmarking

Our initial focus was on the inclusion of a minimum number of entries per buffer and the maximum basket buffer limit. As we'll see in Section §4.3, we then opted to keep the minimum number of entries set to its default setting (10 entries per buffer).

For both the nightly and the release testing, the data derivation job comes from a 2022 dataset with four input files 160327 events. The MC job comes from a 2023 $t\bar{t}$ standard sample simulation job with six input files with 140k events. The specific datasets for both are noted in Appendix A.1.

The corresponding input files for both data and MC jobs were ran with various configurations of Athena (version 24.0.16) and its specified basket buffer limit. The four configurations tested all kept minimum 10 entries per basket and modified the basket limitation in the following ways:

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

Interesting results come from the comparison of “no-lim” and “default” configuration. The “256k” and “512k” configurations were included for completeness and provided to be a helpful sanity check throughout. Building and running these configurations of Athena are illustrated in a GitHub repository. [9]

4.3 Results

4.3.1 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 entries per branch. 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 elimination of the “min-number-entries” we assume the

minimum number of entries per branch 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.

Athena v22.0.16 configurations (Data)	Max PSS (MB) ($\Delta\%$ default)	PHYS outFS (GB) ($\Delta\%$ default)	PHYSLITE outFS (GB) ($\Delta\%$ default)
With basket-cap and min-num-entries (default)	27.109 (+ 0.00 %)	3.216 (+ 0.00 %)	1.034 (+ 0.00 %)
Without both basket-cap and min-num-entries	27.813 (+ 2.53 %)	3.222 (+ 0.20 %)	1.036 (+ 0.21 %)
Without basket-cap but with min-num-entries	27.814 (+ 2.53 %)	3.216 (- 0.00 %)	1.030 (- 0.39 %)
With basket-cap but without min-num-entries	27.298 (+ 0.69 %)	3.221 (+ 0.15 %)	1.042 (+ 0.71 %)

Table 4.1: Athena v22.0.16: 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 v22.0.16 configurations (MC)	Max PSS (MB) ($\Delta\%$ default)	PHYS outFS (GB) ($\Delta\%$ default)	PHYSLITE outFS (GB) ($\Delta\%$ default)
With basket-cap and min-num-entries (default)	14.13 (+ 0.00 %)	5.83 (+ 0.00 %)	2.59 (+ 0.00 %)
Without both basket-cap and min-num-entries	16.08 (+ 12.13 %)	6.00 (+ 2.93 %)	2.72 (+ 5.06 %)
Without basket-cap but with min-num-entries	15.97 (+ 11.51 %)	5.67 (- 2.80 %)	2.45 (- 5.58 %)
With basket-cap but without min-num-entries	14.19 (+ 0.42 %)	6.16 (+ 5.35 %)	2.87 (+ 9.90 %)

Table 4.2: Athena v22.0.16: Comparing the maximum proportional set size (PSS) and PHYS/PHYSLITE output file sizes (outFS) for MC jobs over various Athena configurations for 140000 entries.

4.3.2 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 configurations (Data)	Max PSS (MB) ($\Delta\%$ default)	PHYS outFS (GB) ($\Delta\%$)	PHYSLITE outFS (GB) ($\Delta\%$)
(default)	27.8591 (+ 0.00 %)	3.2571 (+ 0.00 %)	1.0334 (+ 0.00 %)
no_limit	28.6432 (+ 2.74 %)	3.2552 (- 0.06 %)	1.0302 (- 0.31 %)
256k_basket	28.2166 (+ 1.27 %)	3.2553 (- 0.05 %)	1.0303 (- 0.30 %)
512k_basket	28.4852 (+ 2.20 %)	3.2571 (+ 0.00 %)	1.0307 (- 0.26 %)

Table 4.3: Athena v24.0.16: 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 configurations (Data)	Max PSS (MB) ($\Delta\%$ default)	PHYS outFS (GB) ($\Delta\%$)	PHYSLITE outFS (GB) ($\Delta\%$)
(default)	15.00 (+ 0.00 %)	5.88 (+ 0.00 %)	2.59 (+ 0.00 %)
no_limit	16.90 (+ 11.27 %)	5.72 (- 2.80 %)	2.45 (- 5.55 %)
256k_basket	15.28 (+ 1.87 %)	5.80 (- 1.35 %)	2.51 (- 3.11 %)
512k_basket	16.41 (+ 8.60 %)	5.74 (- 2.46 %)	2.46 (- 5.11 %)

Table 4.4: Athena v24.0.16: Comparing the maximum proportional set size (PSS) and PHYS/PHYSLITE output file sizes (outFS) for MC jobs over various Athena configurations for 140000 entries.

4.3.3 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.]

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.SumPtChargedPFOp500	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.SumPtChargedPFOp500	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.]

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.

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.

4.3.4 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

5.1 Continuous integration 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 new merge request and nightly build for issues in the new code that could break core I/O 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.

- [1] 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>.
- [2] ATLAS Experiment at CERN. *Trigger and Data Acquisition*. URL: <https://atlas.cern/Discover/Detector/Trigger-DAQ>.
- [3] 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>.
- [4] Nico Giangiacomi. “ATLAS Pixel Detector and readout upgrades for the improved LHC performance”. Presented 18 Mar 2019. Bologna U., 2018. URL: <https://cds.cern.ch/record/2684079>.
- [5] Beniamino Di Girolamo and Marzio Nessi. *ATLAS undergoes some delicate gymnastics*. 2013. URL: <https://cerncourier.com/a/atlas-undergoes-some-delicate-gymnastics/>.
- [6] ATLAS software group. *Athena*. URL: <https://doi.org/10.5281/zenodo.2641997>.
- [7] ATLAS software group. *Athena Software Documentation*. URL: <https://atlassoftwaredocs.web.cern.ch/athena/>.
- [8] F. Hugging. *The ATLAS pixel detector*. June 2006. DOI: 10.1109/tns.2006.871506. URL: <http://dx.doi.org/10.1109/TNS.2006.871506>.
- [9] A.C. Kraus. *GitHub Repository: building-athena*. <https://github.com/arthurkraus3/building-athena.git>. 2023.

- 605 [10] Ana Lopes and Melissa Loyse Perry. *FAQ-LHC The guide*. 2022. URL: <https://home.cern/resources/brochure/knowledge-sharing/lhc-facts-and-figures>.
- 606
- 607 [11] Bartosz Mindur. *ATLAS Transition Radiation Tracker (TRT): Straw tubes for tracking*
608 *and particle identification at the Large Hadron Collider*. Geneva, 2017. DOI: 10.1016/
609 *j.nima.2016.04.026*. URL: <https://cds.cern.ch/record/2139567>.
- 610 [12] ATLAS Outreach. “ATLAS Fact Sheet : To raise awareness of the ATLAS detector
611 and collaboration on the LHC”. 2010. DOI: 10.17181/CERN.1LN2.J772. URL: <https://cds.cern.ch/record/1457044>.
- 612
- 613 [13] ROOT Team. *ROOT, About*. URL: <https://root.cern/about/>.
- 614 [14] ROOT Team. *ROOT, TTree Class*. 2024. URL: [https://root.cern.ch/doc/master/](https://root.cern.ch/doc/master/classTTree.html)
615 [classTTree.html](https://root.cern.ch/doc/master/classTTree.html).

616

APPENDIX

617

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.PyPy8EG_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
```