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

The ATLAS experiment's Software Performance Optimization Team has efforts in developing the Athena software framework that is scalable in performance and ready for widespread 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.

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 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

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

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

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

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 the smallest interactions at high energies. It has roots in electromagnetism, with the discovery of the electron and other particles, and quantum mechanics, that include descriptions of atoms, particles, and their interactions both relativistic and non-relativistic speeds. There have been many efforts in experimentally probing for unique interactions, and the experiments at the Large Hadron Collider (LHC) at CERN has been at the forefront in revealing new insights. This thesis looks into software performance optimization of the ATLAS experiment at CERN. Specifically, ways to modernize and optimize 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 The ATLAS Detector

The LHC is a 26.7-kilometer ring that crosses between the France-Switzerland border at a depth between 50 and 175 meters underground.[8] The ATLAS experiment 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.[10] 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.[6] 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. [3] 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 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). [9]

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

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. 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 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 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. [10] 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.[4] 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.[5]

ROOT is an open-source software framework used for high-energy physics analysis at CERN.[11] 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 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. [12]

2.3 Derivation Production Jobs

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

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 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 effient 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 attatched to the object, this data could gain complexity the more it was used. Persistent data needed to be simplified before it could be persistified 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 usuable 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 and Monte Carlo 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 would give us an effective model that would resemble how current derivation jobs would act on real and MC simulated data. These toy model mixtures would 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 Introducing the Toy Model

There were a number of iterations to the toy model, but the first was constructed by filling up a TTree with branches that each have vectors with varying number of random floats written and read back. 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. Specifics are illustrated in Appendix A.1.

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. To reiterate, the goal was to create branches with data similar to what could be found in real post-reconstructed data samples. This would mean creating for branches with compression ratios on the order $\mathcal{O}(5)$, as is typical for such real data.

3.2 Toy Model Compression

3.2.1 Random Float Branches

Upon reading back the ROOT file, the user would be able to see 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.

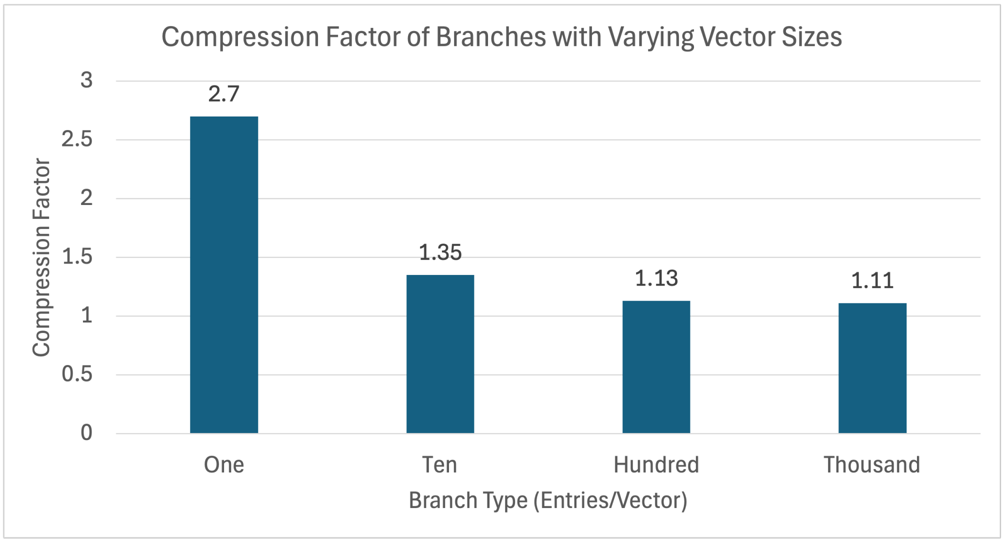

Figure 3.1: Compression factors of $N = 1000$ entries per branch with random-valued vectors of varying size.

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.

3.2.2 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, see code in Appendix A.2. The first attempt creating branches with mixed amounts of randomization was having a mixture of 1/2 random data at $N = 10^6$ events. However that led to the larger branches still seeing poor compression, so a new mixture of 1/4 random data was introduced.

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.

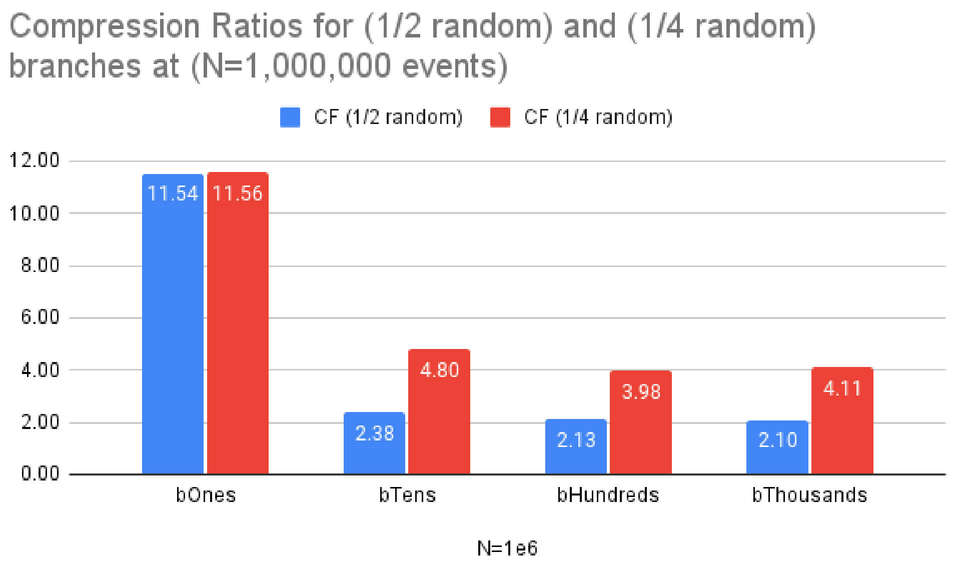

Figure 3.2: Compression Ratios for ($\frac{1}{2}$ random) and ($\frac{1}{4}$ random) branches at ($N = 10^6$ events)

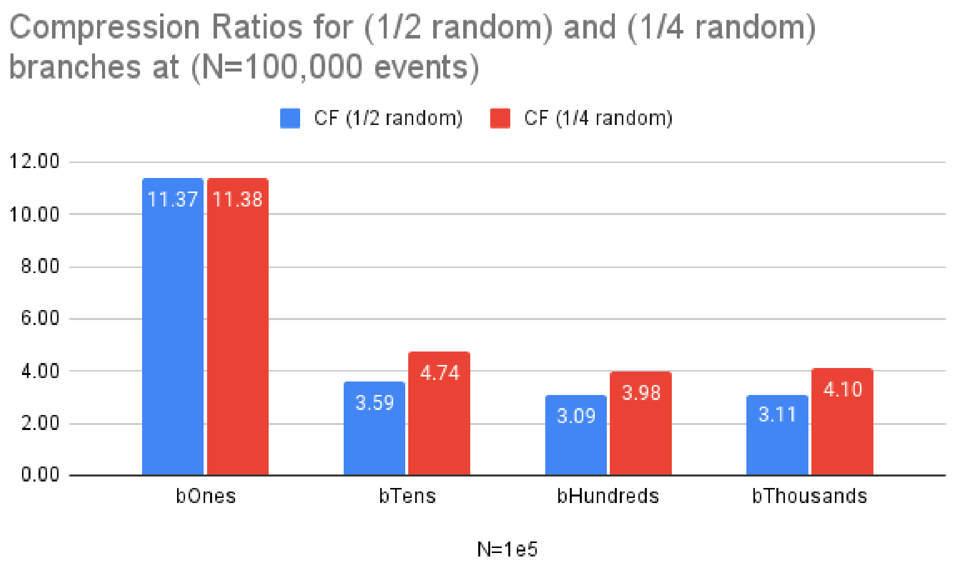

Figure 3.3: Compression Ratios for ($\frac{1}{2}$ random) and ($\frac{1}{4}$ random) branches at ($N = 10^5$ events)

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. It was still worth testing the $N = 10^6$ events for stress testing purposes. Here is where tuning the basket size can begin to start.

3.3 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. 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 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

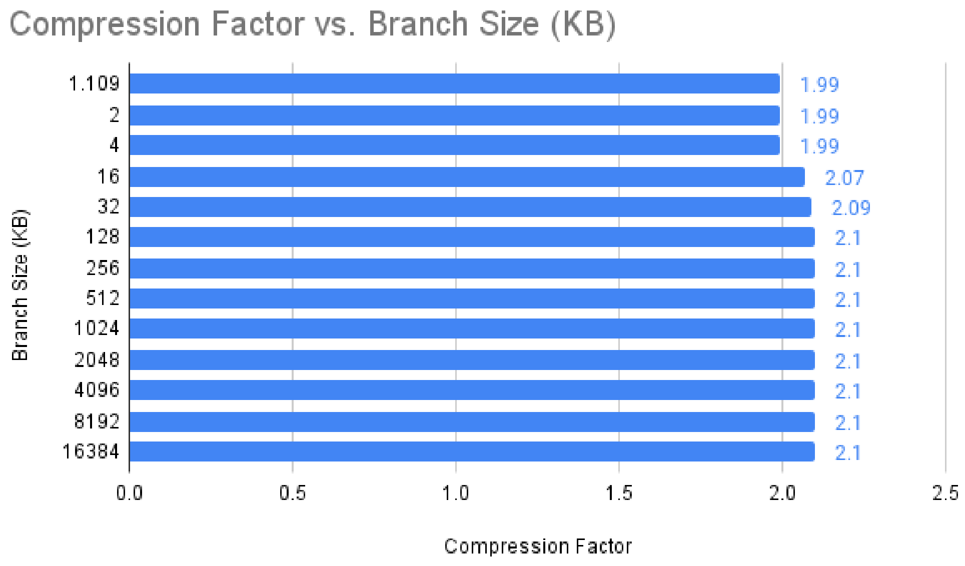

Figure 3.4: Compression Factors vs Branch Size (1/2 Mixture $N = 10^6$ events)

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.

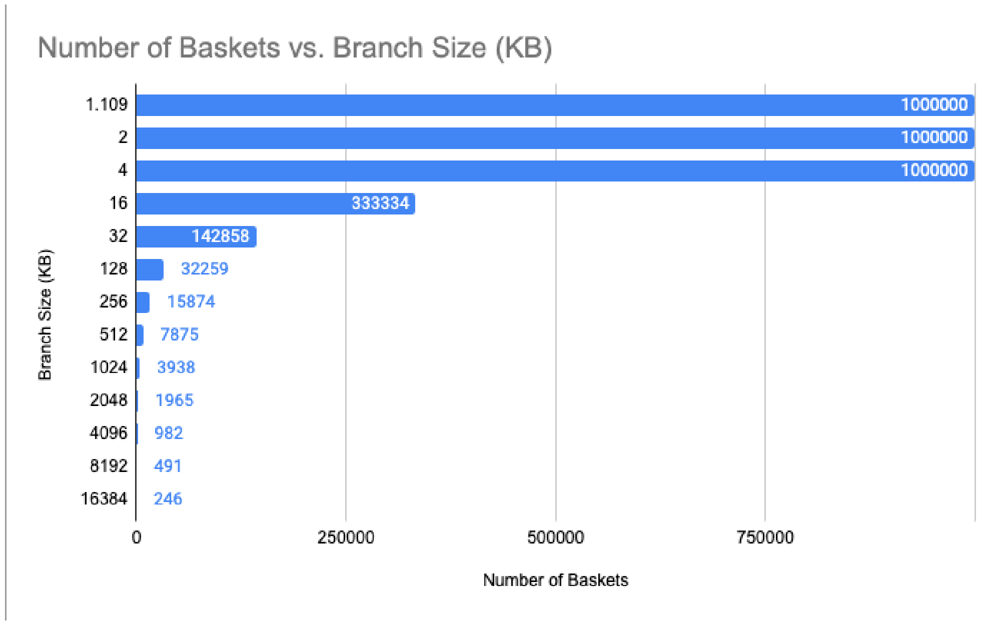

Figure 3.5: Number of Baskets vs Branch Size (1/2 Mixture $N = 10^6$ events)

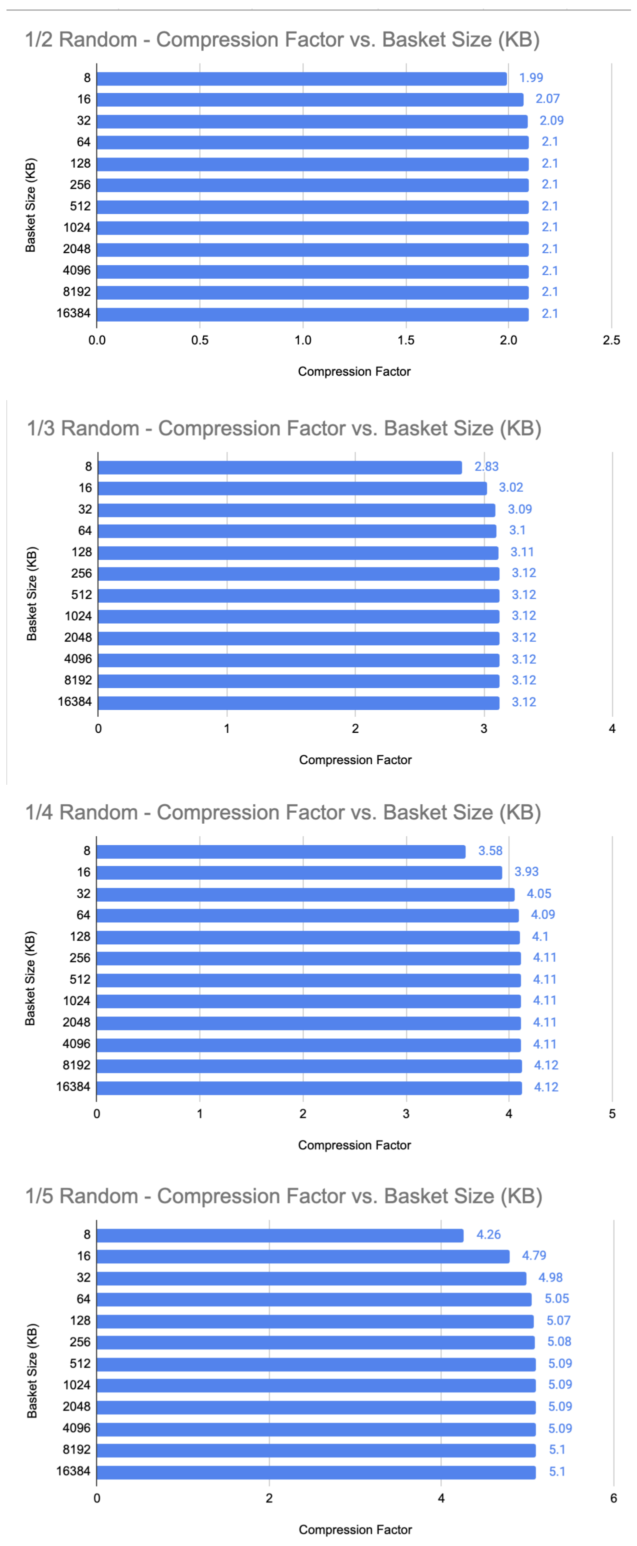

Figure 3.6: Varying Mixtures in 8 point precision - Number of Baskets vs Branch Size ($N = 10^6$ events)

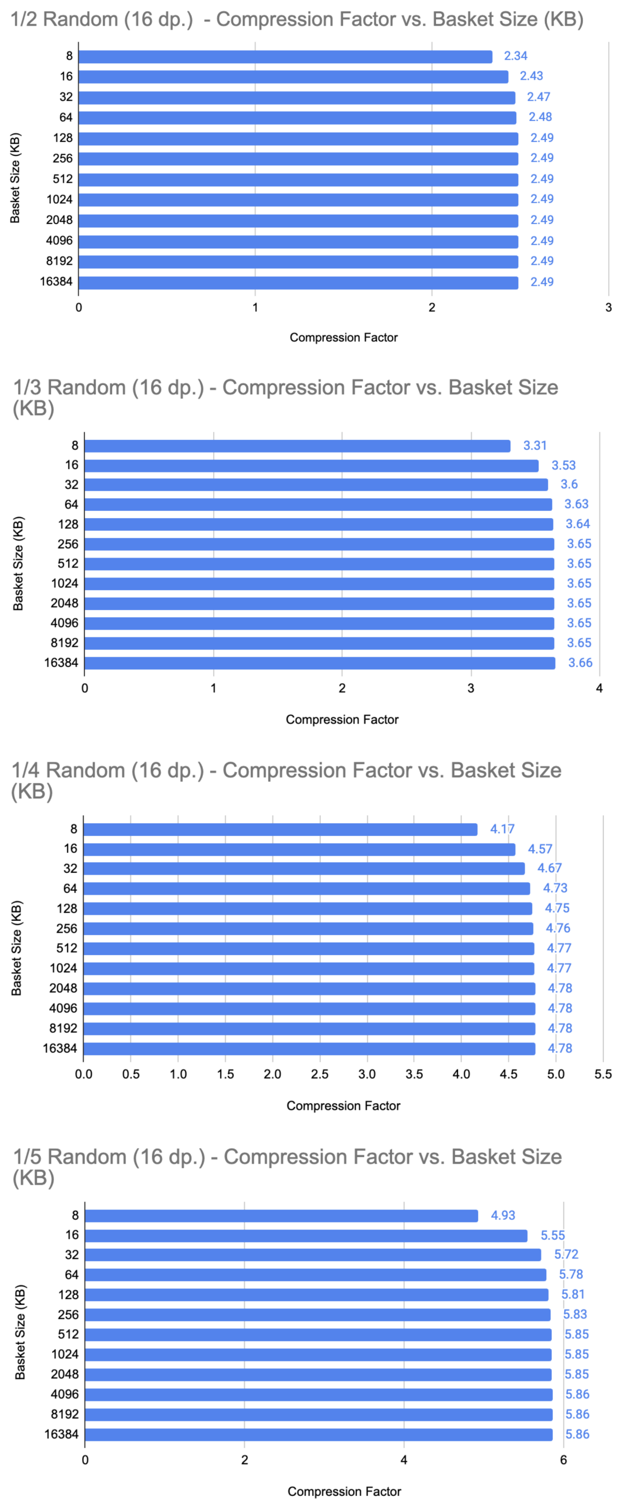

Figure 3.7: Varying Mixtures in 16 point precision - Number of Baskets vs Branch Size ($N = 10^6$ events)

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 the following section, 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 B.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. [7]

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 entires per branch. PHYSLITE MC derivation production, from Table 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 4.3 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) (Δ% default)	PHYS outFS (GB) (Δ% default)	PHYSLITE outFS (GB) (Δ% 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) (Δ% default)	PHYS outFS (GB) (Δ% default)	PHYSLITE outFS (GB) (Δ% 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 3 through Table 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 v24.0.16 configurations (Data)	Max PSS (MB) (Δ% default)	PHYS outFS (GB) (Δ% default)	PHYSLITE outFS (GB) (Δ% default)
With basket-cap and min-num-entries (default)	27.8591 (+ 0.00 %)	3.2571 (+ 0.00 %)	1.0334 (+ 0.00 %)
Without both basket-cap and min-num-entries	28.6432 (+ 2.74 %)	3.2552 (- 0.06 %)	1.0302 (- 0.31 %)
Without basket-cap but with min-num-entries	28.2166 (+ 1.27 %)	3.2553 (- 0.05 %)	1.0303 (- 0.30 %)
With basket-cap but without min-num-entries	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.

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?)

[Table 5 from the int note]

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 optimzation

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.

BIBLIOGRAPHY

[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] 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.

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

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

[6] F. Hugging. *The ATLAS pixel detector*. June 2006. DOI: 10.1109/tns.2006.871506. URL: http://dx.doi.org/10.1109/TNS.2006.871506.

[7] A.C. Kraus. *GitHub Repository: building-athena*. https://github.com/arthurkraus3/building-athena.git. 2023.

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

[9] 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.

[10] 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`.

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

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

APPENDIX A

TOY MODEL AOD CODE

A.1 Random floats into branches

This is a part of the code that was run on ROOT to fill up branches with entirely random float values into branches of sizes, 1, 10, 100, and 1000 entries.

```
#include <iostream>
#include <memory>
#include <ostream>
#include <vector>

#include "TBranch.h"
#include "TCanvas.h"
#include "TFile.h"
#include "TH1.h"
#include "TRandom.h"
#include "TStyle.h"
#include "TTree.h"

void VectorTree() {
    /*
    Goal: separately load up four different branches
    with different vector sizes
        Size 1: 1 entry
        Size 2: 10 entries
        Size 3: 100 entries
        Size 4: 1000 entries
```

```
*/

std::cout << "Initializing..." << std::endl;

// CREATING FILE AND TREE
std::cout << "creating file and tree..." << std::endl;

std::unique_ptr<TFile> myFile =
    std::make_unique<TFile>("VectorTreeFile.root", "RECREATE");
TTree *tree = new TTree("myTree", "myTree");

tree->SetAutoFlush(0);
tree->SetAutoSave(0);

if (!myFile) {
    std::cout << "Error: file creation failed" << std::endl;
}

// INITIALIZING VARIABLES
std::cout << "initializing variables and vector" << std::endl;

const int NEvents = 1e6; // N = 1,000,000
// Set Number of Entries with 10^# of random floats
int NEntries0 = 1;
int NEntries1 = 10;
int NEntries2 = 100;
int NEntries3 = 1000;

// vectors
std::vector<float> vten0; // 10^0 = 1 entry
std::vector<float> vten1; // 10^1 = 10 entries
std::vector<float> vten2; // 10^2 = 100 entries
std::vector<float> vten3; // 10^3 = 1000 entries

// variables
float f0;
float f1;
float f2;
float f3;

// INITIALIZING BRANCHES
std::cout << "creating branches" << std::endl;

tree->Branch("bOnes", &vten0);
tree->Branch("bTens", &vten1);
tree->Branch("bHundreds", &vten2);
tree->Branch("bThousands", &vten3);

int bSize = 4000;
```

```
tree->SetBasketSize("*", bSize);

// Events Loop
std::cout << "generating events..." << std::endl;
for (int j = 0; j < NEvents; j++) {
    // Clearing entries from previous iteration
    vten0.clear();
    vten1.clear();
    vten2.clear();
    vten3.clear();

    // Generating vector elements, filling vectors
    // Fill vten0
    for (int m = 0, m < NEntries0; m++) {
    f0 = gRandom->Rndm() * 10; // Create random float value
    vten0.emplace_back(f0);    // Emplace that float into vector
    }
    // Fill vten1
    for (int n = 0, n < NEntries1; n++) {
    f1 = gRandom->Rndm() * 10;
    vten1.emplace_back(f1);
    }
    // Fill vten2
    for (int a = 0, a < NEntries2; a++) {
    f2 = gRandom->Rndm() * 10;
```

```
        vten2.emplace_back(f2);
        }
        // Fill vten3
        for (int b = 0, b < NEntries3; b++) {
        f3 = gRandom->Rndm() * 10;
        vten3.emplace_back(f3);
        }
        tree->Fill(); // Fill our TTree with all the new branches
    }
    // Saving tree and file
    tree->Write();

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

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

  int main() {
    VectorTree();
    return 0;
  }
```

A.2 Mixed compressible data in branches

The mixture of floats stored in each vector needs to fill up both the vector and branch to resemble that of real or Monte Carlo simulated data to reach a comparable compression factor. The way this was done for the toy model stage of derivation job optimzation studies was by using code from Appendix A.1 and modifying the parts that filled in each vector.

```
#include <iostream>
#include <memory>
#include <ostream>
#include <vector>

#include "TBranch.h"
#include "TCanvas.h"
#include "TFile.h"
#include "TH1.h"
#include "TRandom.h"
#include "TStyle.h"
#include "TTree.h"

void VectorTree() {
  /*
  Goal: separately load up four different branches
    with different vector sizes
      Size 1: 1 entry
      Size 2: 10 entries
    Size 3: 100 entries
    Size 4: 1000 entries
*/

std::cout << "Initializing..." << std::endl;

// CREATING FILE AND TREE
std::cout << "creating file and tree..." << std::endl;

std::unique_ptr<TFile> myFile =
    std::make_unique<TFile>("VectorTreeFile.root", "RECREATE");
TTree *tree = new TTree("myTree", "myTree");

tree->SetAutoFlush(0);
tree->SetAutoSave(0);

if (!myFile) {
  std::cout << "Error: file creation failed" << std::endl;
}

// INITIALIZING VARIABLES
std::cout << "initializing variables and vector" << std::endl;

const int NEvents = 1000000; // N = 1,000,000
int NEntries0, NEntries1, NEntries2, NEntries3;
```

```
// Vectors
std::vector<float> vten0; // 10^0 = 1 entry
std::vector<float> vten1; // 10^1 = 10 entries
std::vector<float> vten2; // 10^2 = 100 entries
std::vector<float> vten3; // 10^3 = 1000 entries

// Variables
float f0;
float f1;
float f2;
float f3;

// INITIALIZING BRANCHES
std::cout << "creating branches" << std::endl;

tree->Branch("bOnes", &vten0);
tree->Branch("bTens", &vten1);
tree->Branch("bHundreds", &vten2);
tree->Branch("bThousands", &vten3);

int bSize = 4000;

tree->SetBasketSize("*", bSize);
```

```
// EVENTS LOOP
std::cout << "generating events..." << std::endl;

for (int j = 0; j < NEvents; j++) {

  // Randomizing number of entries
  int NEntries0 = 1;
  int NEntries1 = 10;
  int NEntries2 = 100;
  int NEntries3 = 1000;

  // clearing events
  vten0.clear();
  vten1.clear();
  vten2.clear();
  vten3.clear();

  // Generating vector elements, filling vectors

  // Generating vten0
  for (int m = 0; m < NEntries0; m++) {
    if (m < (NEntries0 / 2)) {
      f0 = gRandom->Gaus(0, 1) * gRandom->Rndm();
      vten0.emplace_back(f0);
    } else {
```

```
    f0 = m;
    vten0.emplace_back(f0);
  }
}

// Generating vten1
for (int n = 0; n < NEntries1; n++) {
  if (n < NEntries1 / 2) {
    f1 = gRandom->Rndm() * gRandom->Gaus(0, 1);
    vten1.emplace_back(f1);
  } else {
    f1 = 1;
    vten1.emplace_back(f1);
  }
}

// Generating vten3
for (int a = 0; a < NEntries3; a++) {
  if (a < NEntries3 / 2) {
    f3 = gRandom->Rndm() * gRandom->Gaus(0, 1);
    vten3.emplace_back(f3);
  } else {
    f3 = 1;
    vten3.emplace_back(f3);
  }
```

```
        }

        tree->Fill();
    }

    // Saving tree and file
    tree->Write();

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

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

int main() {
    VectorTree();
    return 0;
}
```

APPENDIX B

DERIVATION PRODUCTION DATA

B.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
```