Persistifying the Complex Event Data Model of the ATLAS Experiment in RNTuple

Alaettin Serhan Mete[1], Marcin Nowak[2], Peter Van Gemmeren[1]

[1]Argonne National Laboratory, [2]Brookhaven National Laboratory

22nd International Workshop on Advanced Computing and Analysis Techniques in Physics Research
11-15 March 2024

A Brief Introduction to ATLAS and Athena

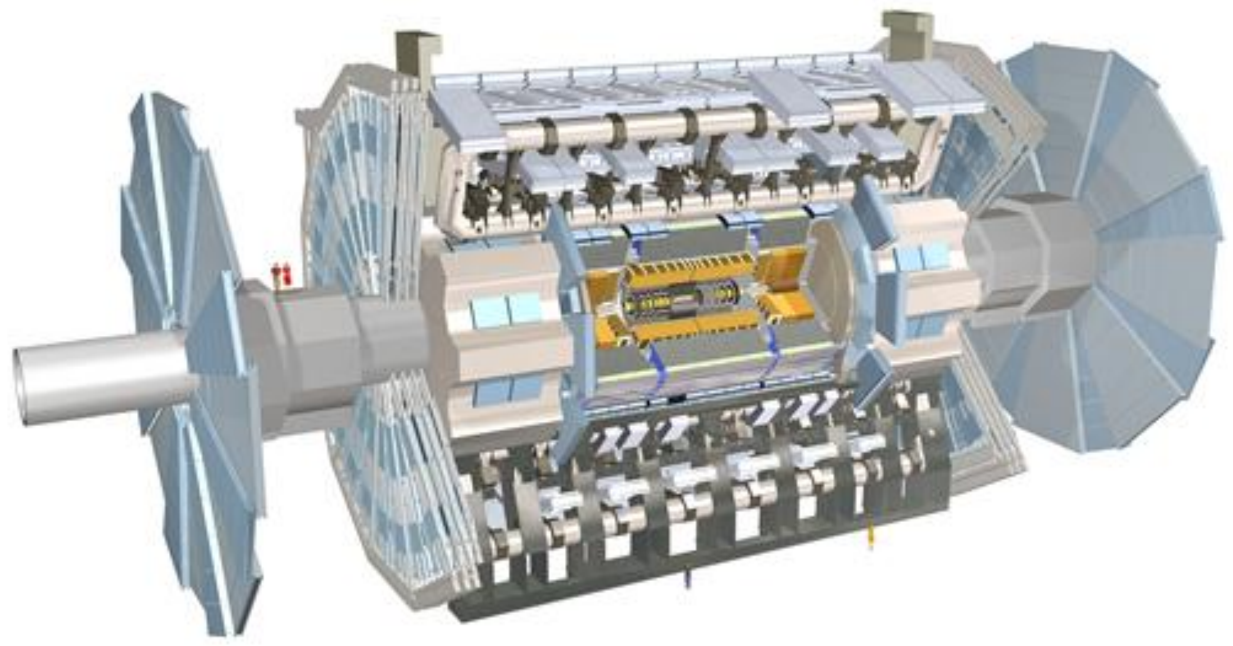
- **ATLAS is a general-purpose detector at the Large Hadron Collider (LHC)**
- **Athena is the open-source software framework of ATLAS**
  - Based on the Gaudi framework, jointly managed by the ATLAS and the LHCb experiments
  - It consists of about 4 (1.5) million lines of C++ (python) code
    - ☐ CMake for *building*, python for *job configuration*, and C++ for *the framework and the algorithms*

Typical ATLAS Data Processing Chain

Name (Output Format)

Monte Carlo: Event Generation (EVNT) → Simulation (HITS) → Digitization/ Overlay (RDO)

Collision Data: (RAW)



Reconstruction (ESD, AOD) → Derivation (DAOD) → Analysis

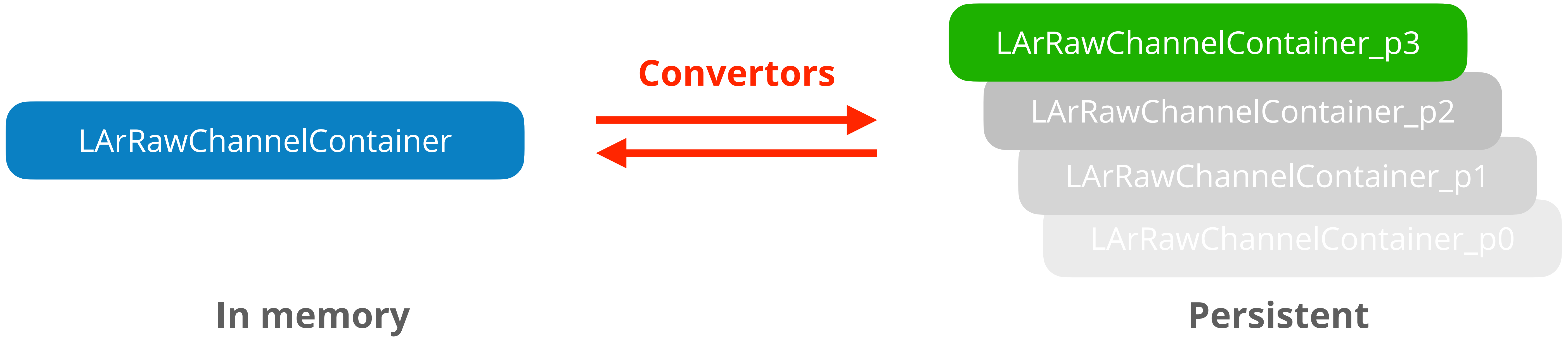
As one moves down the processing chain the complexity and event sizes go down

A Few Words on the ATLAS Event Data Model (EDM)

- **The ATLAS detector consists of many complex sub-systems/groups**
  - For successful data processing all of them have to perform in unison
- **Thus it is extremely important to ensure that:**
  - We have common interfaces and data objects across the experiment
    - Same objects can be used in trigger, event reconstruction, physics analysis etc.
  - We have coherent software that is easy to maintain over many decades
  - We can read/write data in a consistent manner over the lifetime of the experiment
- **In a nutshell, the ATLAS Event Data Model (EDM):**
  - Provides a collection of C++ classes that define detector/physics objects
  - Allows streamlined and efficient processing of highly complex algorithms
    - Rely on advanced C++ concepts and data structures to accomplish these
- **E.g. Tracks (ID and/or MS), Clusters (LAr and/or Tile), Electrons, Photons**

The Transient/Persistent Data Model Separation

- **The transient data model is used during data processing (in-memory)**
  - Athena uses data stores, e.g., EventStore, to maintain the transient objects (cleaned at every event)
  - This can get arbitrarily complex to take advantage of complex programming concepts
  - Not every aspect of the transient model needs to be preserved indefinitely
  - Definitions can change over the course of the experiment
- **The persistent data model is used for storing the data permanently**
  - The definition can be different than the transient model, almost always simpler
  - One can:
    - Prune the data to minimize the storage footprint
    - Have multiple versions of the persistent data (schema evolution)



- **The main GOOD:**
  - Flexibility and performance
- **The main NOT-AS-GOOD:**
  - More code to write and maintain

... or not

- **During Run 1 (2011-2013) the ATLAS EDM used a fully T/P separated model**
  - Allowed to hide the C++ complexity from the storage side and schema evolution
- **Parts of the EDM were re-written during Long Shutdown 1 (2013-2015)**
  - Now called the xAOD EDM, the primary target was reconstruction and analysis formats
- **The xAOD EDM is generally simpler and does not have T/P separation**
  - There is, however, a separation between the interface (user) and the payload (data)
  - Most of the underlying data are stored in fundamental types of std::vector (of std::vector)
    - Some data are part of the class definitions (static)
    - Others can be added **on-the-fly on-demand at any time during processing** (dynamic)
- **The xAOD EDM is primarily (but not exclusively) used in the (D)AOD data**
  - Some are used in upstream formats, e.g., xAODEventInfo (run number, event number, ...)
- **The upstream formats, e.g., HITS, RDO, still use T/P separated EDM**

The ATLAS Input/Output (I/O) System

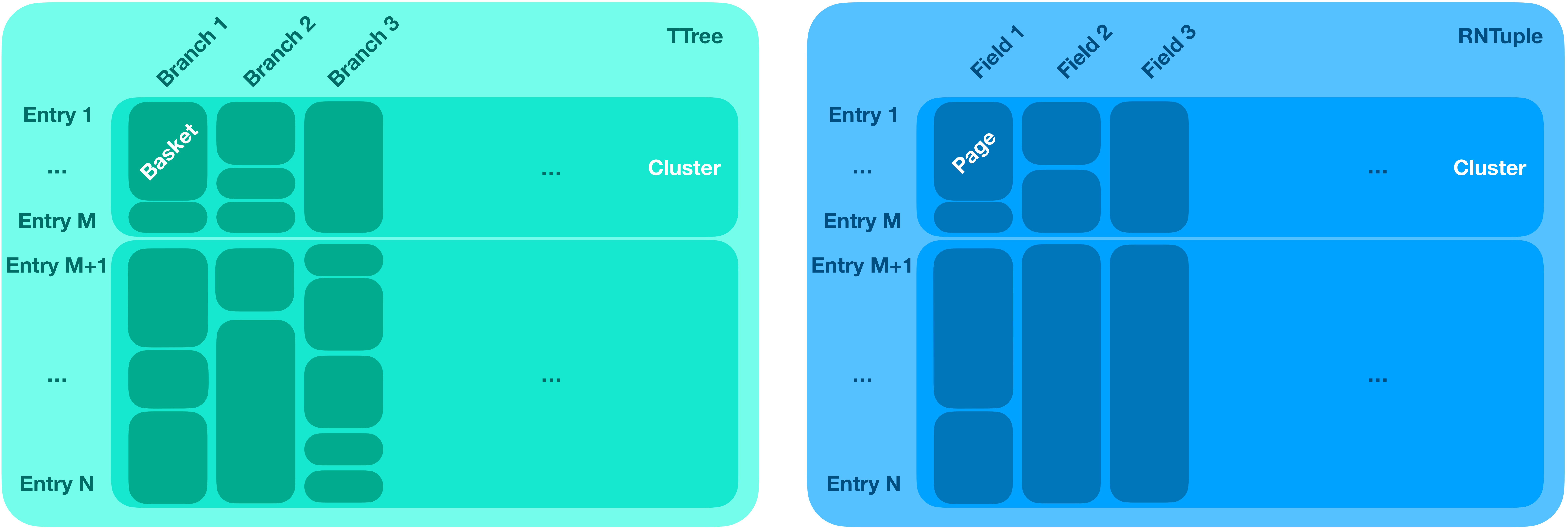
- **ATLAS' I/O system is based on the primary LCG POOL concepts**
  - The data storage broken down into a structured hierarchy:

  - In a nutshell, objects are stored in containers that reside in databases
  - The API hides the technology specific implementation of the storage service
- **Since the beginning of data taking, ATLAS has used ROOT's TTree**
  - This basically means having a ROOT storage service that contains and implements:
    - RootDatabase, i.e., ROOT file-level operations, opening/closing TFile etc.
    - TreeContainer, i.e., ROOT TTree-level operations, creating, filling TTree/TBranch etc.
- **The most important aspect is that ROOT API is isolated from the EDM**
  - The framework/EDM is not glued to the ROOT API apart from the storage service

RNTuple: The Future of ROOT I/O

- **Starting with Run 4 (2029) ROOT's primary I/O sub-system will be RNTuple**
  - TTree will be available in ROOT as legacy support, i.e., functional but no new features
- **For better or worse TTree grew organically over the last two decades**
  - Well pre-dates recent C++ language standards, shifts in programming paradigms etc.
- **In a nutshell, RNTuple is a more modern and efficient approach**
  - Adopts some of the latest C++ language features, cleaner memory management etc.
  - It has a codified specifications (that is not yet finalized but getting there)
  - However, one important point is that it is **not a drop-in replacement of TTree**
    - For example, **it does not support raw pointers, polymorphism**, etc.
- **Therefore, experiments need to (possibly) embrace some changes**
  - This can vary from simply adopting the new API to rewriting (parts of) the EDM

A Simplified Crash Course: TTree to RNTuple



- **An approximate translation from TTree to RNTuple**
  - The internals are completely different, which we don't dive in here
- **As far as the user is concerned, the real difference lies in the API and the philosophy**
  - Gains due to internal workings of RNTuple are all bonuses!

RNTuple: ATLAS Requirements

- **ATLAS needs a set features:**
  - Plain Old Data (POD), STL vectors (nested), user defined classes/enums
    - These are fairly experiment independent requirements
    - As an extension, we need some stdlib types, e.g., std::map etc.
  - User-defined collection proxies and late model extensions
    - These features are needed primarily by the xAOD EDM
  - Type-based user code execution when reading data a.k.a. Read Rules
    - This feature is needed for initializing (some) data for transient objects and schema evolution
  - A void* based interface to bind the I/O layer with the rest of the framework

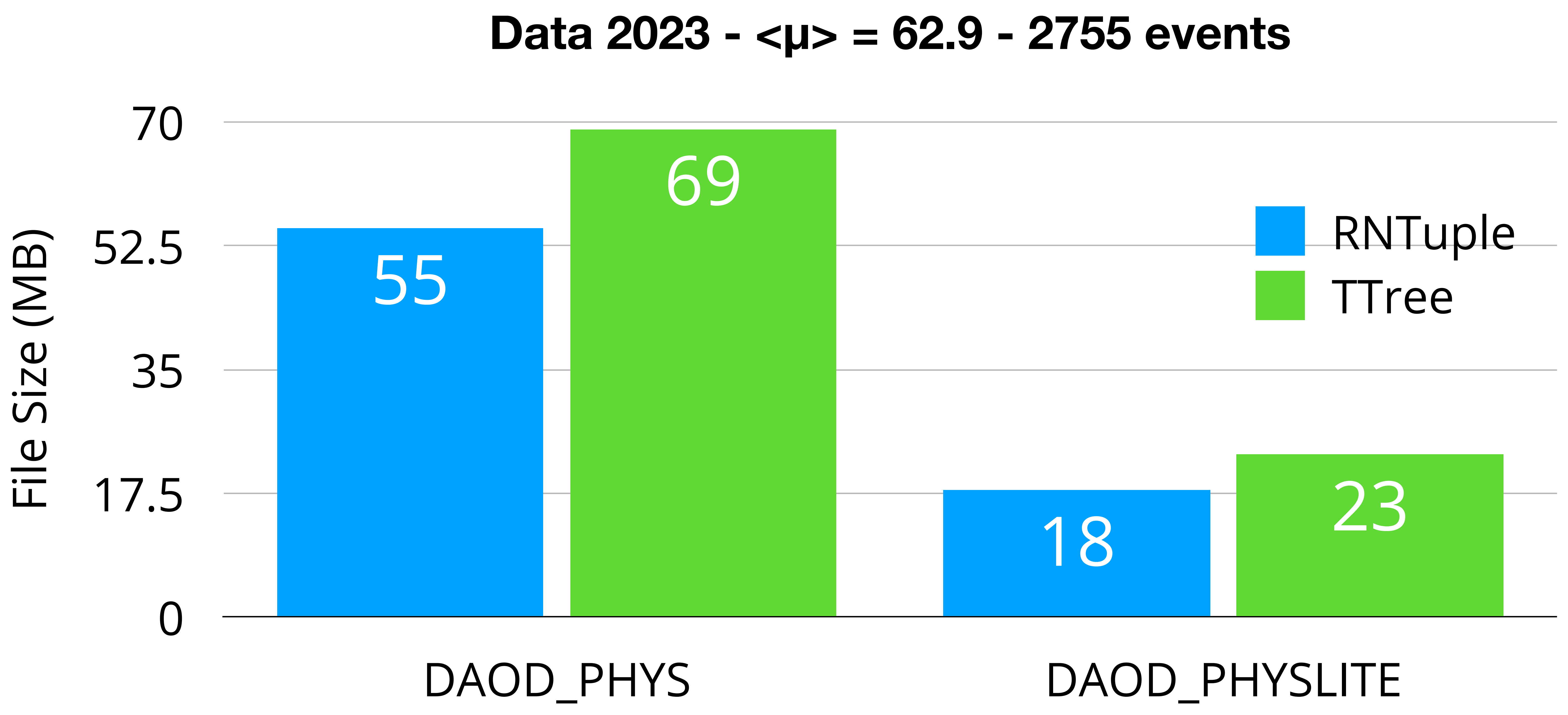
- **Current RNTuple implementation supports all these features**

RNTuple: ATLAS' Perspective on Adoption

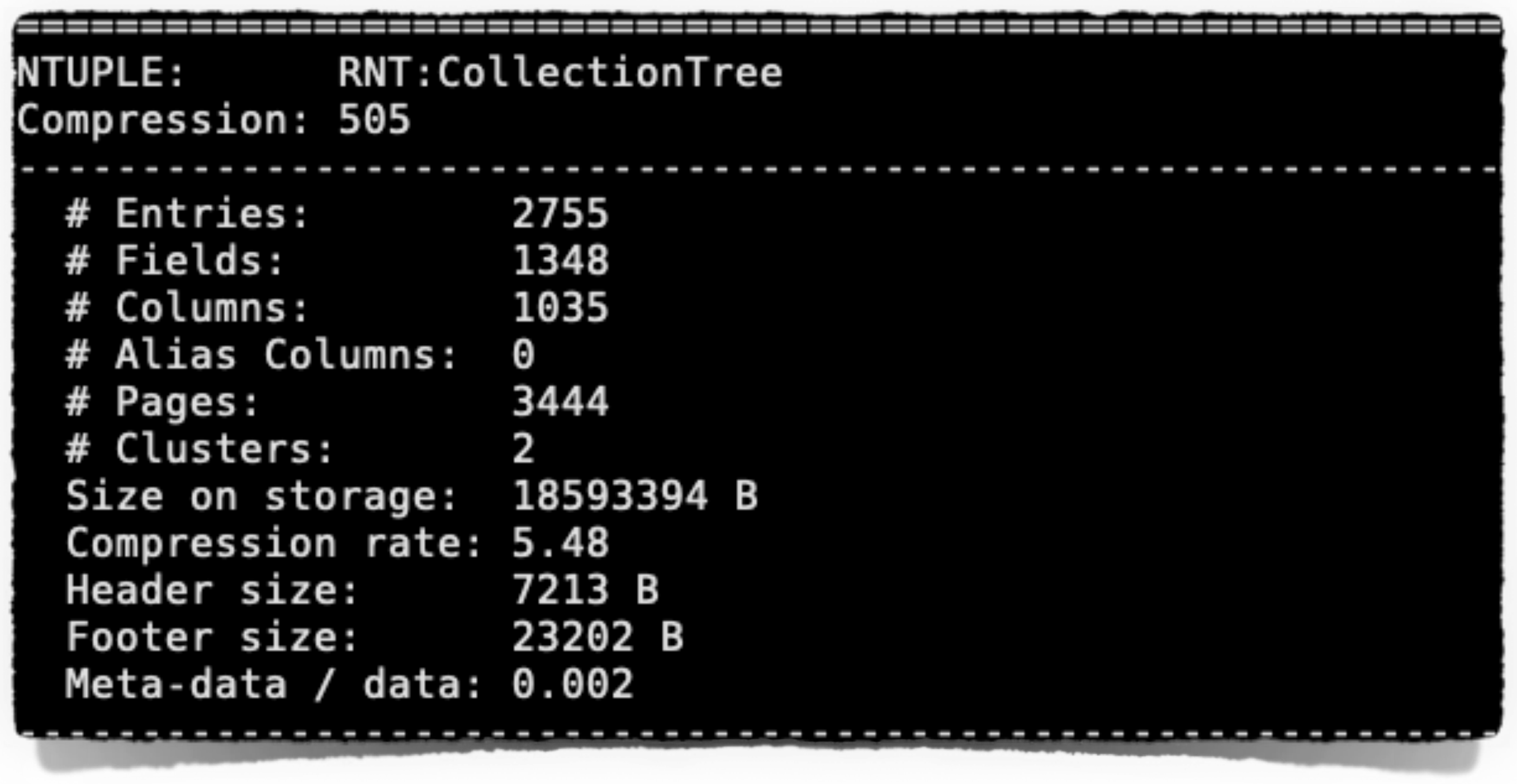
- **Two main earlier design choices that eased our adoption process:**
  - A good chunk of the reconstruction EDM was already simplified, i.e., xAOD EDM
  - More complex parts of the EDM largely adopts the T/P separation that hides the complexity form the storage layer
  - But more importantly ROOT (API) was kept disjoint from the EDM and only used in the storage layer
- **For the most part, the work was focused on introducing a new POOL technology layer**
  - Introducing a new RNTupleContainer technology and introducing the new Reader/Writer code that goes with it
    - No changes to the user code, everything is handled in the I/O software...
  - Possibly the most complex logic was to support on-the-fly dynamic attributes with the xAOD EDM
    - Once the model extension support was introduced to RNTupleWriter, this became fairly transparent, too
- **There were only a few cases of special ATLAS specific issues, e.g.:**
  - Most of the transient ATLAS data are stored in a specialized class called DataVector<T>
    - In short, DataVector<T> is a class that acts like std::vector<T*> (note: DataVector pre-dates the latest C++ language extensions)
    - RNTuple not supporting raw pointers makes storing these as they are a bit problematic
  - This was one of the few special problems we encountered migrating the upstream formats from TTree to RNTuple
  - (One) Solution: T/P separate the relevant class and store std::vector<T> instead
    - This means introducing a new persistent type and a convertor
      - If reading an old file, simply return the object, if reading a new file, convert the persistent type to transient and return that (or vice versa if writing)

RNTuple: A Quick Look at DAOD Performance

- **Current studies indicate about 20+% storage savings is possible in DAODs**
  - It's important to note TTree is heavily optimized over the last 20 years
  - Similar optimization studies will be carried out for RNTuple prior to production



Sample DAOD_PHYSLITE in RNTuple



```
NTUPLE:      RNT:CollectionTree
Compression: 505
------------------------------------------------------------
 # Entries:          2755
 # Fields:           1348
 # Columns:          1035
 # Alias Columns:    0
 # Pages:            3444
 # Clusters:         2
 Size on storage:    18593394 B
 Compression rate:   5.48
 Header size:        7213 B
 Footer size:        23202 B
 Meta-data / data:   0.002
```

Towards Getting Production Ready

- **Being able to read/write our data in RNTuple is a great start!**
  - **ATLAS can read/write all data formats, i.e., HITS, RDO, ESD, AOD, and DAOD in RNTuple!**
- **However, there are many other features that are needed for production**
  - Fast merging of RNTuple objects on-the-fly and custom entry/event indexing
    - These are primarily needed for the DAOD production workflows
    - These jobs run in multi-process Athena where a dedicated process merges worker outputs on-the-fly
  - Having various utilities/tools to peek into, compare, validate, ... RNTuples
    - These are needed for job configuration, input/output validation etc.
  - Relational RNTuples, a.k.a. *friendship*
    - This allows us to use event sample augmentation
- **In addition, detailed optimizations/stress-testing studies need to be done**
  - We need to make sure RNTuple works reliably/efficient in all official ATLAS workflows
  - We also need to make sure that the data products and the jobs meet production limitations

Conclusions and Outlook

- **A Rough RNTuple timeline from ATLAS' perspective:**

LHC Run 3			Long Shutdown 3			LHC Run 4 (HL-LHC)
… 2023	2024	2025	2026	2027	2028	2029 onward
Experimental Development			Adoption, Testing, Validation			Production

★ We are here!

- **The current plan is to adopt RNTuple for (at least) the Event Data for Run 4**
  - Discussions on how to handle in-file Meta Data is currently ongoing
- **All in all we're in a very good position but there is much work ahead of us!**
  - All aspects need to be rigorously tested and validated well in advance of Run 4
    - Multi-process/thread Athena jobs, complementary tools, benchmarking, and optimizations
- **We're looking forward to all of the fun ahead!**


Argonne
NATIONAL LABORATORY
DEPARTMENT OF ENERGY
UNITED STATES OF AMERICA
U.S. DEPARTMENT OF
ENERGY