The Run 2 ATLAS Analysis Event Data Model

Marcin Nowak, BNL

On behalf of the ATLAS Analysis Software Group and Event Store Group

16th International workshop on Advanced Computing and Analysis Techniques in physics research (ACAT)

September 2014 Prague

Overview

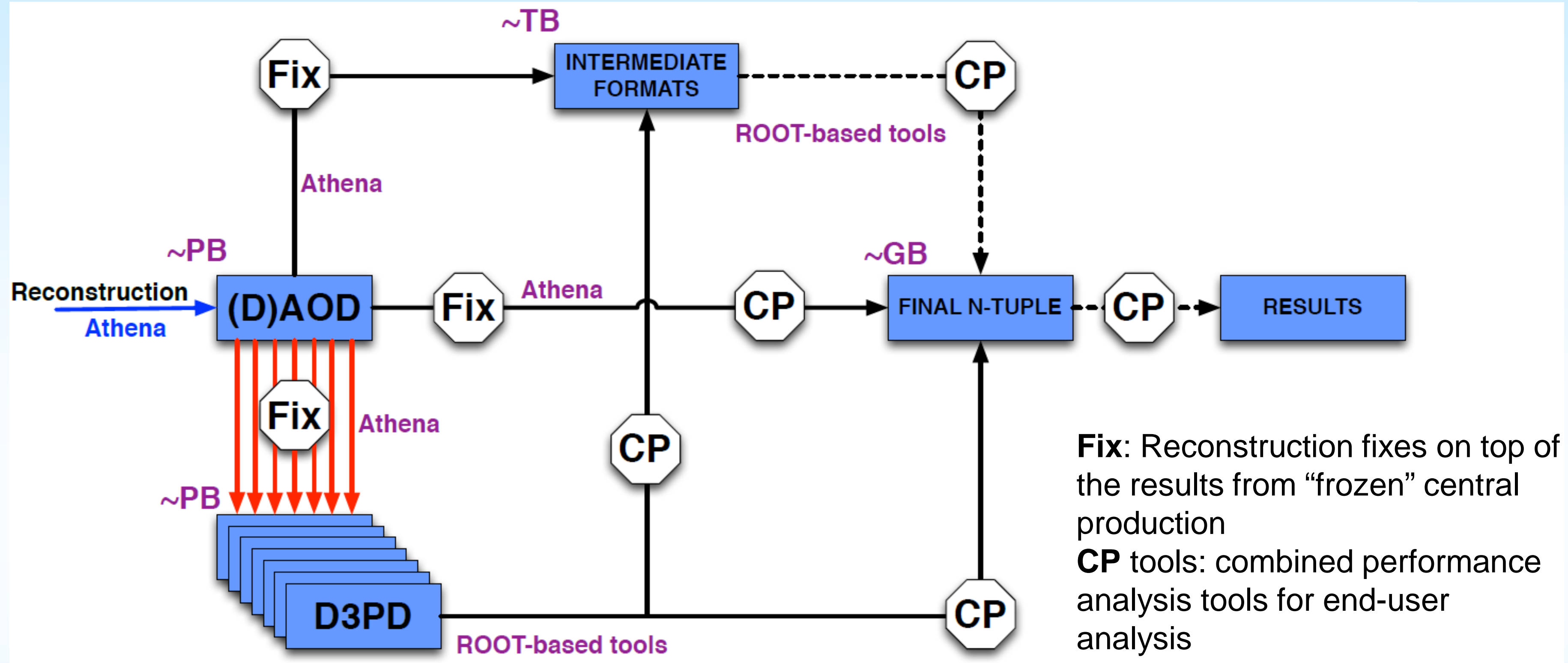
- ATLAS Run 1 analysis data model
  - The original design
  - The actual analysis model during Run 1
  - Problem areas and things we wanted to improve

- The new model for Run 2
  - Design
  - Implementation
  - Persistency
    - ROOT file structure
    - Simple code example
  - Schema evolution
  - Performance

The ATLAS Run 1 Event Data Model

- Event reconstruction process produces data in the AOD format (Analysis Object Data): official ATLAS-wide event representation with reduced information for physics analysis
  - Fully object-oriented (complex) EDM, part of Athena: ATLAS offline software framework
  - Size = 350-400KB
  - Persistency: object based
    - Using a different persistent data model to be able to freely evolve the transient EDM without compromising backward compatibility
    - Statically defined persistent object shape (schema)
  - Persistent data format required Athena (or at least its persistency layer) to read AOD
    - Even though the files were in ROOT format
    - Quite a lot of libraries needed (dictionaries, converters)
  - Frozen Tier0 policy
    - Reconstruction fixes not part of original AOD – need to be redone every time
- AOD reading too slow for many physicists
  - Athena startup, object reading and AODfix overheads
- Majority of the users turned to intermediate data formats (DPD)
  - Working groups started to produce their own private Derived Physics Data datasets – readable directly from ROOT

ATLAS Analysis Model During Run 1



- DPDs produced on request only – delay in respect to the central AOD production
- Data format different than in Athena causing duplication of software tools
- DPD-based tools also different between groups
  - Hard to share code and compare results
- Combined DPD size ~3x the size of corresponding AOD

Coming up with a Better Model for Run 2

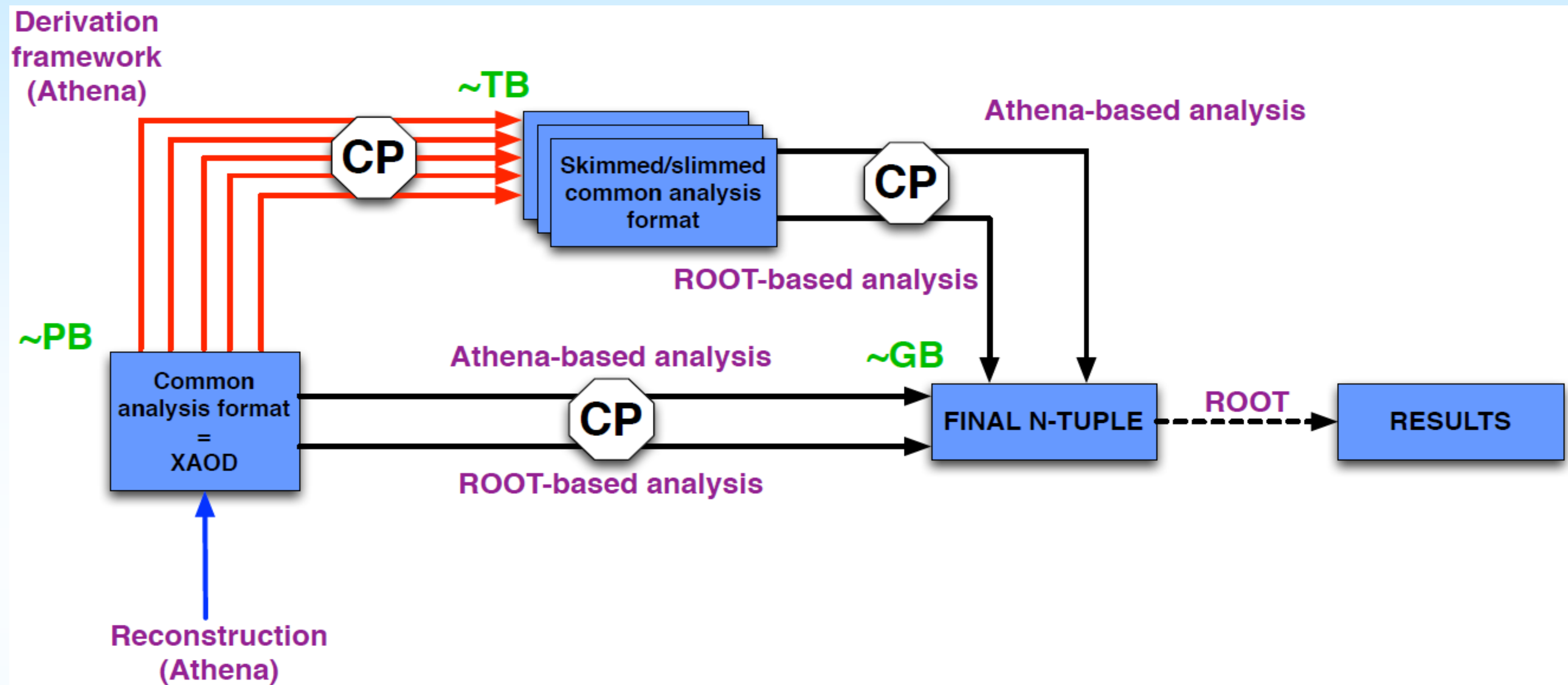
- The first Long Shutdown of LHC created an opportunity to rethink and redesign the analysis data model, based on the experience from Run 1
- New model design requirements:
  - Prepare for increased data rate in Run 2 (~2x that of Run 1)
    - Flexibility in balancing CPU vs disk space requirements
  - Maintain the I/O performance of standalone ROOT: >1kHz
    - Enable reading of single attributes
    - Data directly analyzable in ROOT
  - Reduce the latency of delivering data to the end users
  - Make code sharing between groups and between Athena and DPDs possible
    - Promote collaboration between groups
  - Maintain the ability to read full AOD in Athena environment
    - Access to calibration databases

- Proposal: merge AOD and DPD into a new format called **xAOD**

Introducing the xAOD Format

- Replacement for both AOD and DPD data for Run 2
  - Produced as the end result of the Athena-based reconstruction
    - Full xAOD data available without delay
  - Used as both input and output for physics group productions
    - xAOD allows reduction of content without changing the format
  - Can be created and read in standalone ROOT
    - Lightweight – number of libraries limited to minimum
- Single, object-oriented API
  - From the user point of view just like the old AOD
  - Special implementation with respect to class data members
  - Software tools using single common API can function in both frameworks
- Dynamic xAOD object shape
  - Data members added at runtime or removed during copying
- Single transient/persistent representation
  - No longer fixed class shape like before
  - No separate persistent data model
  - Ability to read single attributes

The New Analysis Model for Run 2

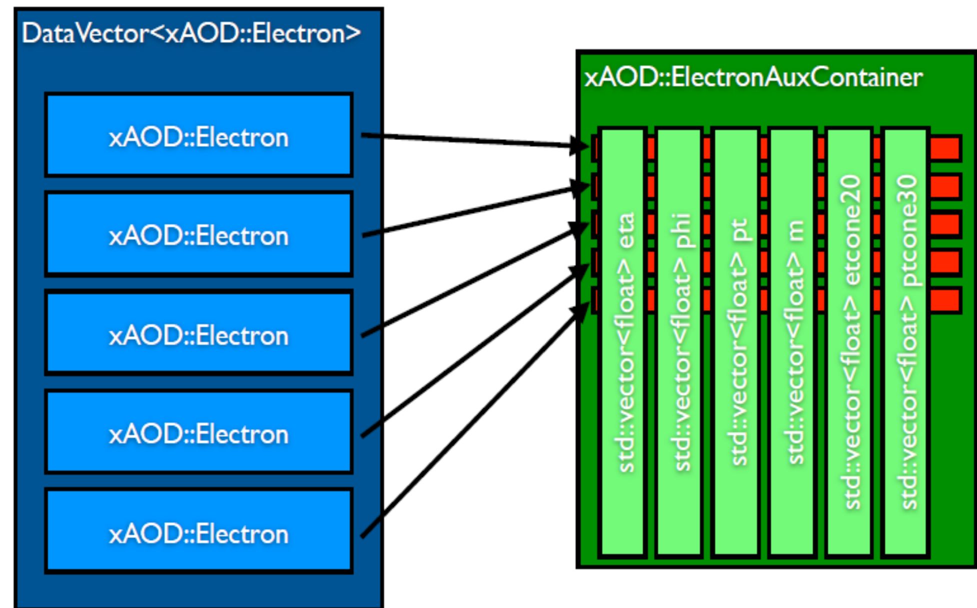


- xAOD data format delivered by central production, directly usable for analysis in Athena and ROOT (lower path)
- Reduction Framework producing reduced-size data samples for analysis groups (upper path)
- CP tools: combined performance analysis tools for end-user analysis, both in Athena and ROOT
- Final analysis stage done in pure ROOT, primarily on local resources

xAOD Format: Basic Design

- xAOD objects consist of an interface object and a storage container
  - Not to be confused with class interface, more like a proxy
- From the user point of view, the interface object is the only visible object, but usually it does not have any data members itself
  - Data member are stored in the storage container
  - There is one storage container per collection of objects
    - ATLAS collections in Athena are implemented using DataVector class

- Storage containers keep arrays of attributes
- An apparent array-of-structs is actually represented in memory as a struct-of-arrays
  - Interesting implications for vectorization and I/O



Single objects have their dedicated storage container with 1-element arrays

- Can be added or removed from collections

xAOD Implementation: Data Stores

- xAOD objects have a set of pre-defined (static) attributes
  - Storage containers assigned to these types have data arrays to store their static attributes:

```
class JetAuxContainer_v1 : public AuxContainerBase {
  public:  JetAuxContainer_v1();
  private: std::vector<float>  pz;       ← storage array for PZ attribute
}
JetAuxContainer_v1::JetAuxContainer_v1() {
   AUX_VARIABLE( pz );
}
```

    - The constructor uses a macro to automatically register data arrays in Registry
      - Arrays can be later looked up by their names

- Additional (dynamic) object attributes can be added at any time
  - They are kept in a storage container extension that allocates storage arrays as needed
- Type-specific storage containers are only an optimization!
  - Technically all different xAOD types could use just the dynamic store
- Dynamic attributes may be selectively dropped when writing to file
  - 3 level selection lists in Athena: by object type / name / attribute
  - Static store may be converted to dynamic in order to drop static attributes

xAOD Implementation: Data Store Access

- xAOD object data is stored in the storage container
- The interface object uses getter and setter methods to access static attributes
- Accessors are provided to make these methods fast:

```
float Jet_v1::pz() const {
   static Accessor<float>  pz_acc("pz");
   return pz_acc(*this);
}
void Jet_v1::setPz(float pz) {
   static Accessor<float>  pz_acc("pz");
   pz_acc(*this) = pz;
   return;
}
```

  - Accessors use attribute type and name for initialization (lookup in Registry)
    - C++ static storage can be used to ensure the (slow) identifier lookup is done only once
    - After initialization the accessor provides direct access to the storage array
  - Accessors are attribute-specific, not object-specific
    - Object they access needs to be specified for every use (still fast)
- Accessors for dynamic attributes can be declared anywhere in the user code
    - Also C++ static!

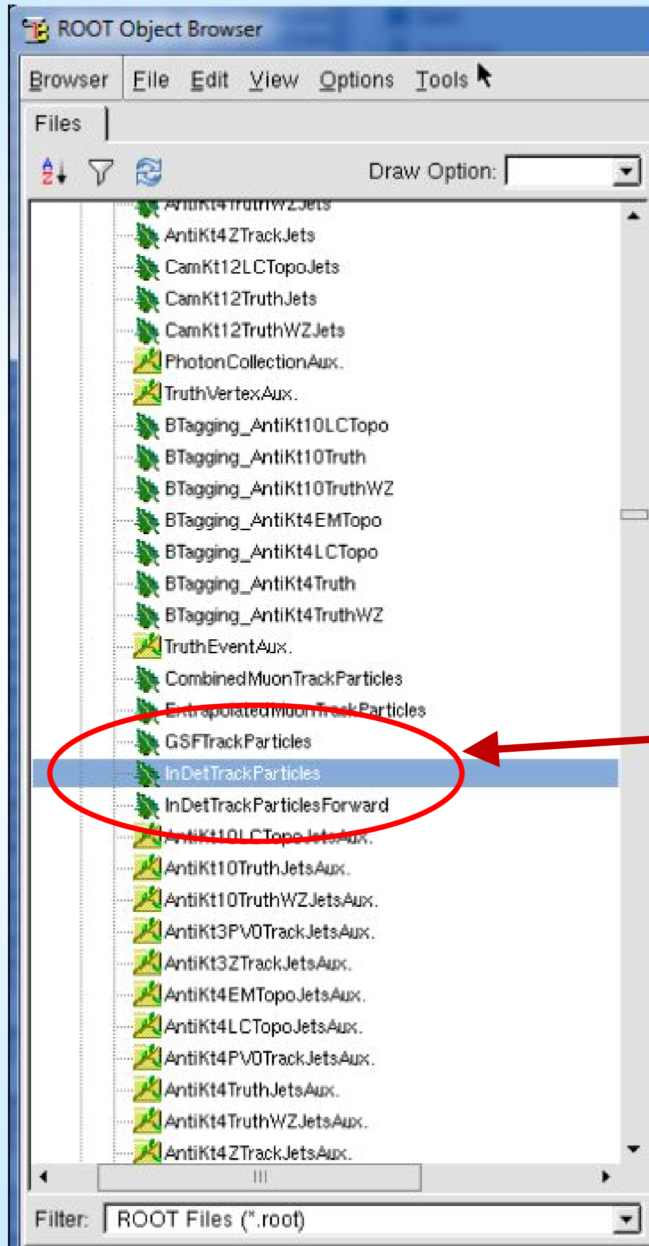
xAOD: Persistency

- xAOD data files are created both by Athena and standalone ROOT
  - Files coming from both sources need to be readable by ROOT and in particular allow single attribute reading
- Athena persistency layer had to be modified:
  - Historically Athena used object-based I/O with fixed class schema defined in dictionaries – not possible for the dynamic store!
  - Static store single attribute reading needed tuning of ROOT split level
- Solution: writing xAOD collections by components:
  - Collection (DataVector) of interface objects: stored as an object
  - Static part of the storage container: stored as an object
    (object-based storage requires (ROOT) class dictionary)
  - Dynamic attributes: stored in dedicated TTree branches created as needed
- Storage container provides uniform API for accessing storage of both static and dynamic attribute storage
  - For both attribute types the I/O API delivers storage array plus the type information
  - Opens interesting options for conversion of object shape during writing
- Reading of dynamic attributes is implemented with a dedicated storage container
  - Empty in the beginning, with attributes read transparently when accessed

Note: dynamic attributes can make files with the same data types have different TTree structure

- Can be a surprise when trying to merge files!

xAOD File in the TBrowser



Inspecting an xAOD file produced during ATLAS Data Challenge 2014 – using ROOT TBrowser

InDetTrackParticles collection – the interface object with no attributes

xAOD File in the TBrowser (2)

ROOT Object Browser

Browser File Edit View Options Tools Help

Files | Draw Option:

- ExtrapolatedMuonTrackParticlesAux.
- GSFTrackParticlesAux.
- InDetTrackParticlesAux.
  - InDetTrackParticlesAux.xAOD::AuxContainerBase
  - InDetTrackParticlesAux.d0
  - InDetTrackParticlesAux.z0
  - InDetTrackParticlesAux.phi
  - InDetTrackParticlesAux.theta
  - InDetTrackParticlesAux.qOverP
  - InDetTrackParticlesAux.definingParametersCovMatr
  - InDetTrackParticlesAux.vx
  - InDetTrackParticlesAux.vy
  - InDetTrackParticlesAux.vz
  - InDetTrackParticlesAux.parameterX
  - InDetTrackParticlesAux.parameterY
  - InDetTrackParticlesAux.parameterZ
  - InDetTrackParticlesAux.parameterPX
  - InDetTrackParticlesAux.parameterPY
  - InDetTrackParticlesAux.parameterPZ
  - InDetTrackParticlesAux.trackParameterCovarianceM
  - InDetTrackParticlesAux.parameterPosition
  - InDetTrackParticlesAux.chiSquared
  - InDetTrackParticlesAux.numberDoF
  - InDetTrackParticlesAux.trackFitter
  - InDetTrackParticlesAux.particleHypothesis
  - InDetTrackParticlesAux.trackProperties
  - InDetTrackParticlesAux.patternRecoInfo
  - InDetTrackParticlesAux.numberOfContribPixelLayer
  - InDetTrackParticlesAux.numberOfBLayerHits
  - InDetTrackParticlesAux.numberOfBLayerOutliers
  - InDetTrackParticlesAux.numberOfBLayerSharedHits

Filter: ROOT Files (*.root)

Canvas_1 | Editor 1

InDetTrackParticlesAux.chiSquared

htemp	
Entries	1603207
Mean	34.2
RMS	19.8

$\times 10^3$

0, 20, 40, 60, 80, 100, 120, 140

0, 50, 100, 150, 200, 250, 300, 350, 400

InDetTrackParticlesAux.chiSquared

Command

Command (local):

Static attributes in the InDetTrackParticles storage container's TBranches

xAOD File in the TBrowser (3)

ROOT Object Browser

Browser File Edit View Options Tools Help

Files

Draw Option:

- GSFConversionVerticesAuxDyn.py
- GSFConversionVerticesAuxDyn.pz
- GSFTrackParticlesAuxDyn.truthParticleLink
- GSFTrackParticlesAuxDyn.trackLink
- GSFTrackParticlesAuxDyn.originalTrackParticle
- InDetTrackParticlesAuxDyn.truthType
- InDetTrackParticlesAuxDyn.truthMatchProbability
- InDetTrackParticlesAuxDyn.truthParticleLink
- InDetTrackParticlesAuxDyn.trackLink
- InDetTrackParticlesAuxDyn.truthOrigin
- InDetTrackParticlesForwardAuxDyn.truthType
- InDetTrackParticlesForwardAuxDyn.truthMatchProbability
- InDetTrackParticlesForwardAuxDyn.truthParticleLink
- InDetTrackParticlesForwardAuxDyn.trackLink
- InDetTrackParticlesForwardAuxDyn.truthOrigin
- LArClusterEMFrwdAuxDyn.BAD_CELLS_CORR_E
- LArClusterEMFrwdAuxDyn.CELL_SIGNIFICANCE
- LArClusterEMFrwdAuxDyn.CELL_SIG_SAMPLING
- LArClusterEMFrwdAuxDyn.CENTER_LAMBDA
- LArClusterEMFrwdAuxDyn.CENTER_MAG
- LArClusterEMFrwdAuxDyn.CENTER_X
- LArClusterEMFrwdAuxDyn.CENTER_Y
- LArClusterEMFrwdAuxDyn.CENTER_Z
- LArClusterEMFrwdAuxDyn.DELTA_ALPHA
- LArClusterEMFrwdAuxDyn.DELTA_PHI
- LArClusterEMFrwdAuxDyn.DELTA_THETA
- LArClusterEMFrwdAuxDyn.DM_WEIGHT
- LArClusterEMFrwdAuxDyn.EM_PROBABILITY
- LArClusterEMFrwdAuxDyn.ENG_BAD_CELLS
- LArClusterEMFrwdAuxDyn.ENG_FRAC_CORE

Filter: ROOT Files (*.root)

Canvas_1 | Editor 1

InDetTrackParticlesAuxDyn.truthMatchProbability.InDetTrackParticlesAuxDyn.truthMatchProbability

htemp	
Entries	1603207
Mean	0.9758
RMS	0.07446

$\times 10^3$

1000, 800, 600, 400, 200, 0

0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1

etTrackParticlesAuxDyn.truthMatchProbability.InDetTrackParticlesAuxDyn.truthMatchProbability

InDetTrackParticles dynamic attributes in standalone TBranches

Command

Command (local):

Standalone ROOT xAOD Code Example

- Lightweight and simple access to xAOD from user code “in ROOT”:

```
#include “xAODRootAccess/Init.h”
#include “xAODRootAccess/TEvent.h”
#include “xAODMuon/MuonContainer.h”

int main() {
   xAOD::Init();
   TFile* file = TFile::Open(“xAOD.root”, “READ”);

   xAOD::TEvent event;
   event.readFrom(file);

   for( Long64_t entry=0; entry < event.getEntries(); ++entry ) {
          event.getEntry(entry);

          const xAOD::MuonContainer* muons = 0;
          event.retrieve(muons, “Muons”);

          std::cout << “1st muon pT = “ << muons->at(0)->pt() << std::endl;
   }
   return 0;
}
```

xAOD: Schema Evolution

- In Run 1, support for schema evolution in Athena had a big impact on the persistent data format
  - Design tailored specifically to Athena, operating on a whole object at a time
  - Maintaining 2 separate data models and necessary converters required effort and expertise
  - A model fitting better to reconstruction than to analysis
- For Run 2, the xAOD is both the transient and the persistent EDM
  - xAOD objects have version number in the class name: e.g. Jet_v1
    - Serious changes in class schema will increase the version number
    - Athena can read old version if the class converter support is (like in Run 1)
  - **The end user sees the class name without the version (typedef)**
  - For standalone ROOT, no support for schema evolution is foreseen
    - Except what we can get from ROOT
    - Always working with the "current" EDM, no backward compatibility
  - Athena will continue to use its conversion layer
    - Used in general not only for schema evolution
    - Can be used for schema evolution but only when reading
- ROOT support for schema evolution is much better now than 10 years ago
    - It's class-based, so dynamic attributes have limited schema evolution support

xAOD: Performance

- Performance gains:
  - No conversion to/from persistent EDM during I/O
  - Data members arranged “column-wise”
  - Dynamic attributes read only on-demand
- Potential trouble areas:
  - Large numbers of top-level branches in the TTree
    - One per each dynamic attribute
  - Read-everything mode has more overhead because of the dynamic attributes
    - Main reason for not storing all attributes in dynamic format

Observed results (ATLAS Data Challenge 2014):

- xAOD files are larger than Run 1 files by ~20%
  - But there will be no duplication between AOD and DPD
  - Size increase depends on the data type
    - worst case almost 2x larger but also seen some types become smaller
  - Difference attributed to absence of T/P converters that were compressing data
- In ROOT reading selected attributes >1KHz
  - Interactive ROOT very responsive
- In Athena the development is still ongoing (changes to EventInfo)
  - Not much reliable performance data yet

Summary

- We implemented a new data format that allows in a flexible way to add and remove object properties at runtime
  - In collaboration with the ROOT team
  - We hope to use the model for vectorization
- The full reconstruction code was rewritten to use xAOD
  - Currently teaching the collaboration members to use the new data format giving a series of tutorials
  - First response is positive
- Files can be accessed without the full ATLAS offline software
  - The format is readable with ROOT using only ~100MB of xAOD libraries
- ATLAS Data Challenge 2014 is under way with the new data format