The xAOD Event Data Model

Alexander Lory
LMU Munich

ATLAS-D
17.09.2019

The xAOD Event Data Model

- Event Data Model (EDM):
  - Collection of classes and interfaces that represent an event and ease its manipulation by the analysis developer
  - How electrons, jets, … are stored and how to use them

- xAOD (Analysis Object Data):
  - EDM of ATLAS for run2 which was designed with following goals:
    - Handle trigger, reconstruction, performance and physics analysis use-cases
    - Fast partial reading
    - Flexibility: remove/add/modify objects
    - Readable with ROOT and ATHENA
    - Simplicity
    - …

Outlook

- Examples: xAOD::EventInfo, xAOD::Iparticle
- Objects and object data
- Containers
- ElementLinks
- Copying containers
- xAOD: how does it look like?
- Objects are versioned

xAOD::EventInfo

- It contains information about the given event, e.g.:
  - What was the pile-up for this event?
  - What is the current run number, event number, luminosity block number?

```
const xAOD::EventInfo *myEvtInfo = …
const uint32_t myRunNumber = myEvtInfo->runNumber();
```

- There is one and only one object of this type for a given event in the (D)AOD

Basic event information

uint32_t	**runNumber** () const The current event's run number.
void	**setRunNumber** (uint32_t value) Set the current event's run number.
unsigned long long	**eventNumber** () const The current event's event number.
void	**setEventNumber** (unsigned long long value) Set the current event's event number.
uint32_t	**lumiBlock** () const The current event's luminosity block number.
void	**setLumiBlock** (uint32_t value) Set the current event's luminosity block number.
uint32_t	**timeStamp** () const POSIX time in seconds from 1970. January 1st.
void	**setTimeStamp** (uint32_t value) Set the POSIX time of the event.
uint32_t	**timeStampNSOffset** () const Nanosecond time offset wrt. the time stamp.
void	**setTimeStampNSOffset** (uint32_t value) Set the nanosecond offset wrt. the time stamp.
uint32_t	**bcid** () const The bunch crossing ID of the event.
void	**setBCID** (uint32_t value) Set the bunch crossing ID of the event.

https://atlas-sw-doxygen.web.cern.ch/atlas-sw-doxygen/atlas_22.0.X-DOX/docs/html/df/df8/classxAOD_1_1EventInfo__v1.html

xAOD::IParticle

- Base class for all 4-vector physics objects (muons, jets, clustered energy deposits in calorimeters, tracks, …)
- Pure interface (= all abstract → methods are implemented separately for all classes that inherit from it)

Functions describing the 4-momentum of the object

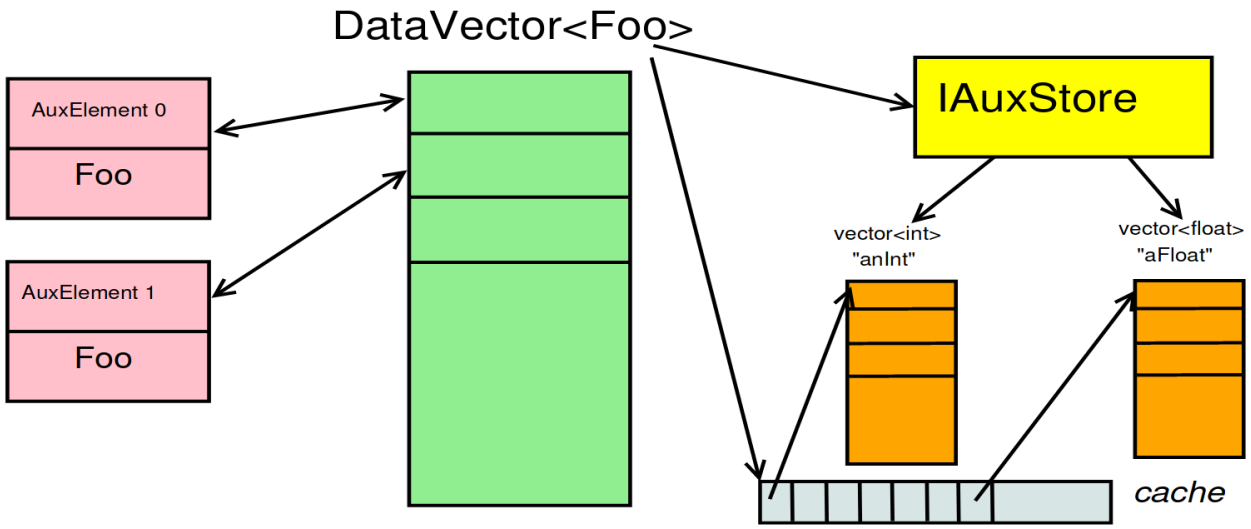
typedef TLorentzVector	**FourMom_t** Definition of the 4-momentum type.
virtual double	**pt** () const =0 The transverse momentum (p_T) of the particle.
virtual double	**eta** () const =0 The pseudorapidity (η) of the particle.
virtual double	**phi** () const =0 The azimuthal angle (ϕ) of the particle.
virtual double	**m** () const =0 The invariant mass of the particle.
virtual double	**e** () const =0 The total energy of the particle.
virtual double	**rapidity** () const =0 The true rapidity (y) of the particle.
virtual const **FourMom_t** &	**p4** () const =0 The full 4-momentum of the particle.

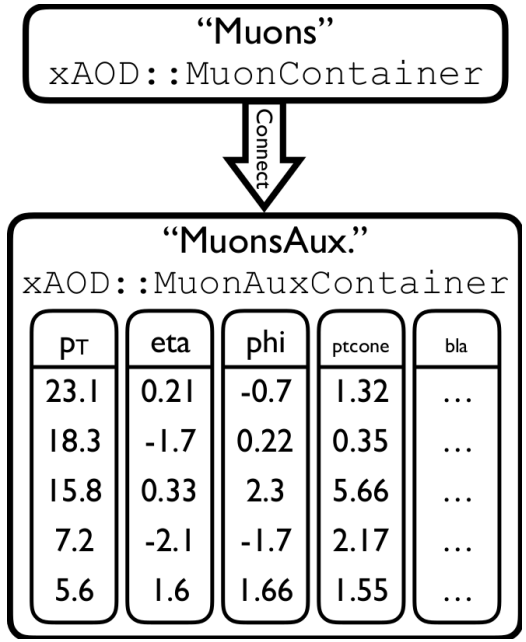
- Example:

```
const xAOD::IParticle *myPart = …
const double myPartPt = myPart->pt();
```

Objects & Object Data

- Objects (e.g. xAOD::Muon) we manipulate are separate from the storage of the payload
  - Data is not member variables of objects but stored separately as vectors of simple types
  - We manipulate „vectors of structures“, but the data is stored as „structures of vectores“
  - Data is accessed with the “auxiliary data store”





→ allows fast partial read

→ each DataVector (container) (e.g. Muons) has an auxiliary store associated (e.g. MuonsAux.)

- **Always use interfaces to the objects. Do not access auxiliary store directly**

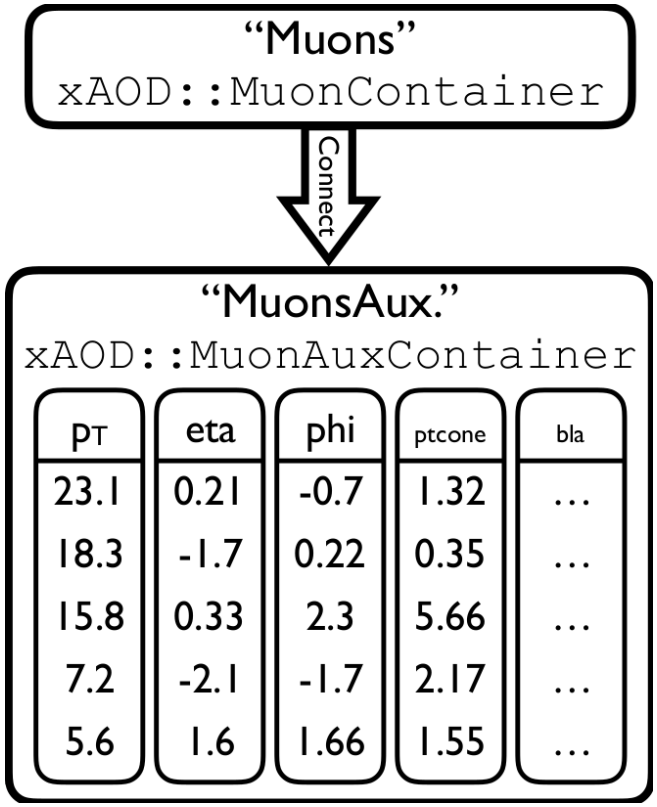
Containers

- Objects in an event are stored in containers
- Containers are std::vectors, with some additions (e.g. auxiliary store)
- Consequence of the auxiliary store design:

  → all objects in a container have the same variables associated

  → adding data to one implies adding data to all

  → new variables/branches need to be initialised on every object



xAOD::IParticle

- Provides standardized ways to access auxiliary store

Functions for getting and setting user properties (from Doxygen):

template<class T >
T & **auxdata** (const std::string &name, const std::string &clsname="")
Fetch an aux data variable, as a non-const reference.

template<class T >
const T & **auxdata** (const std::string &name, const std::string &clsname="") const
Fetch an aux data variable, as a const reference.

template<class T >
bool **isAvailable** (const std::string &name, const std::string &clsname="") const
Check if a user property is available for reading or not.

template<class T >
bool **isAvailableWritable** (const std::string &name, const std::string &clsname="") const
Check if a user property is available for writing or not.

- Example: (if outside loop)

```
const xAOD::IParticle *myPart = …
const float myVar = myPart->isAvailable<float>(“NameOfVar”) ?
                        myPart->auxdataConst<float>(“NameOfVar”) : 0.0;
…
if ( myPart->isAvailableWritableAsDecoration<int>(“NameOfNewVar”) ){
  myPart->auxdecor<int>(“NameOfNewVar”) = 42;
}
```

xAOD::IParticle

- Iparticle->auxdataConst<type>(“var”) & IParticle->auxdecor<type>(“var”) we are looking up „var“ in aux store every time
- Use accessors to avoid this → faster
- Decorator: allows adding data to a const „container“ (= vector of objects)

- Example: (if inside loop)

```
for ( const xAOD::IParticle *myPart : *myPartContainer ){
  static const SG::AuxElement::ConstAccessor<float> myVarAcc(“NameOfVar”);
  const float myVar = myVarAcc.isAvailable(*myPart) ? myVarAcc(*myPart) : 0.0;
  …
  static const SG::AuxElement::Decorator<int> myNewVarDeco(“NameOfNewVar”);
  if ( myNewVarDeco.isAvailableWritableAsDecoration(*myPart) ){
    myNewVarDeco(*myPart) = 42;
  }
}
```

ElementLinks

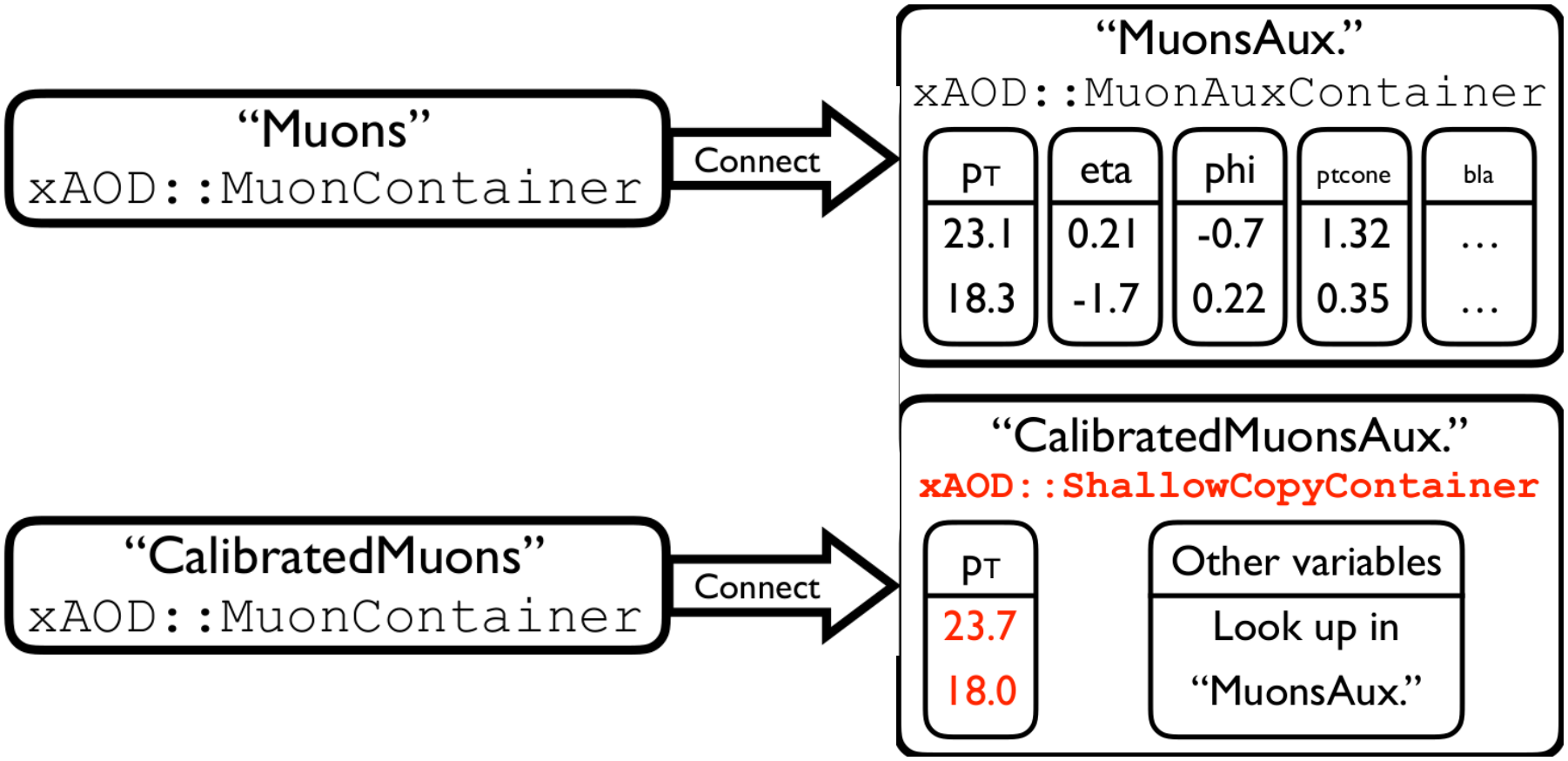
- Objects can have references to other objects, e.g.:
  - xAOD::Electron has an ElementLink to e.g. its xAOD::TrackParticle or its xAOD::CaloCluster
  - XAOD::Muon has an ElementLink to e.g. its xAOD::TrackParticle, which has a decoration to its truth type:

```
const xAOD::Muon *myMuon = …
const ElementLink<xAOD::TrackParticleContainer>& inDetTrkPartLink =
  myMuon->inDetTrackParticleLink();
int type = -1;
static const SG::AuxElement::ConstAccessor<int> truthTypeConstAcc("truthType");
if ( inDetTrkPartLink.isValid() ){
  ATH_MSG_VERBOSE("Got a valid muon inner-detector TrackParticleLink");
  const xAOD::TrackParticle* inDetTrkPart = *inDetTrkPartLink;
  if ( truthTypeConstAcc.isAvailable(*inDetTrkPart) ){
    type = truthTypeConstAcc(*inDetTrkPart);
  }
}
```

Copying containers

- Deep copies: full copy of the containers with the same variables as the original, but only a subset of the objects.

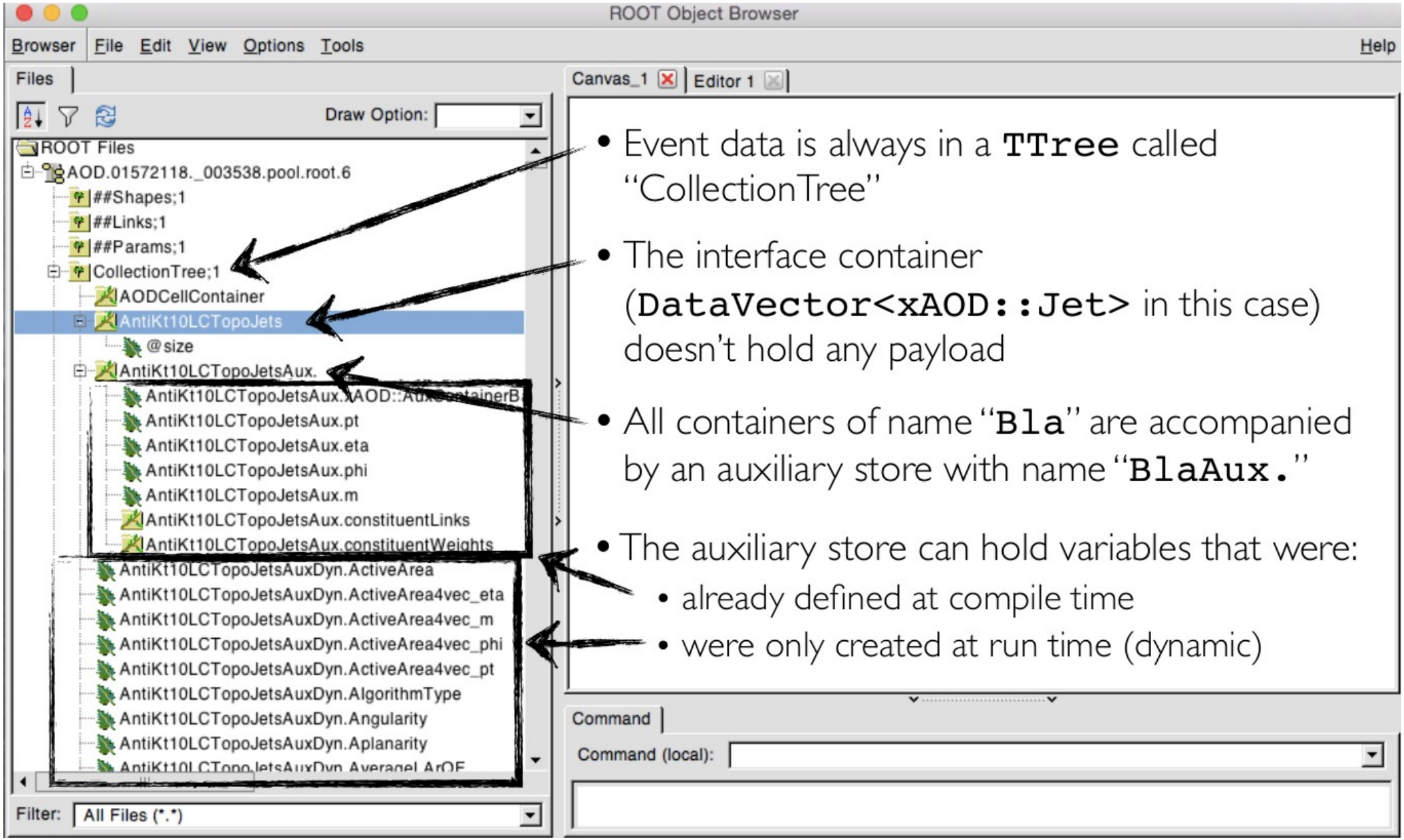
- Shallow copies: is a light weight copy that only updates some variables. Useful for e.g. calibrating muons.



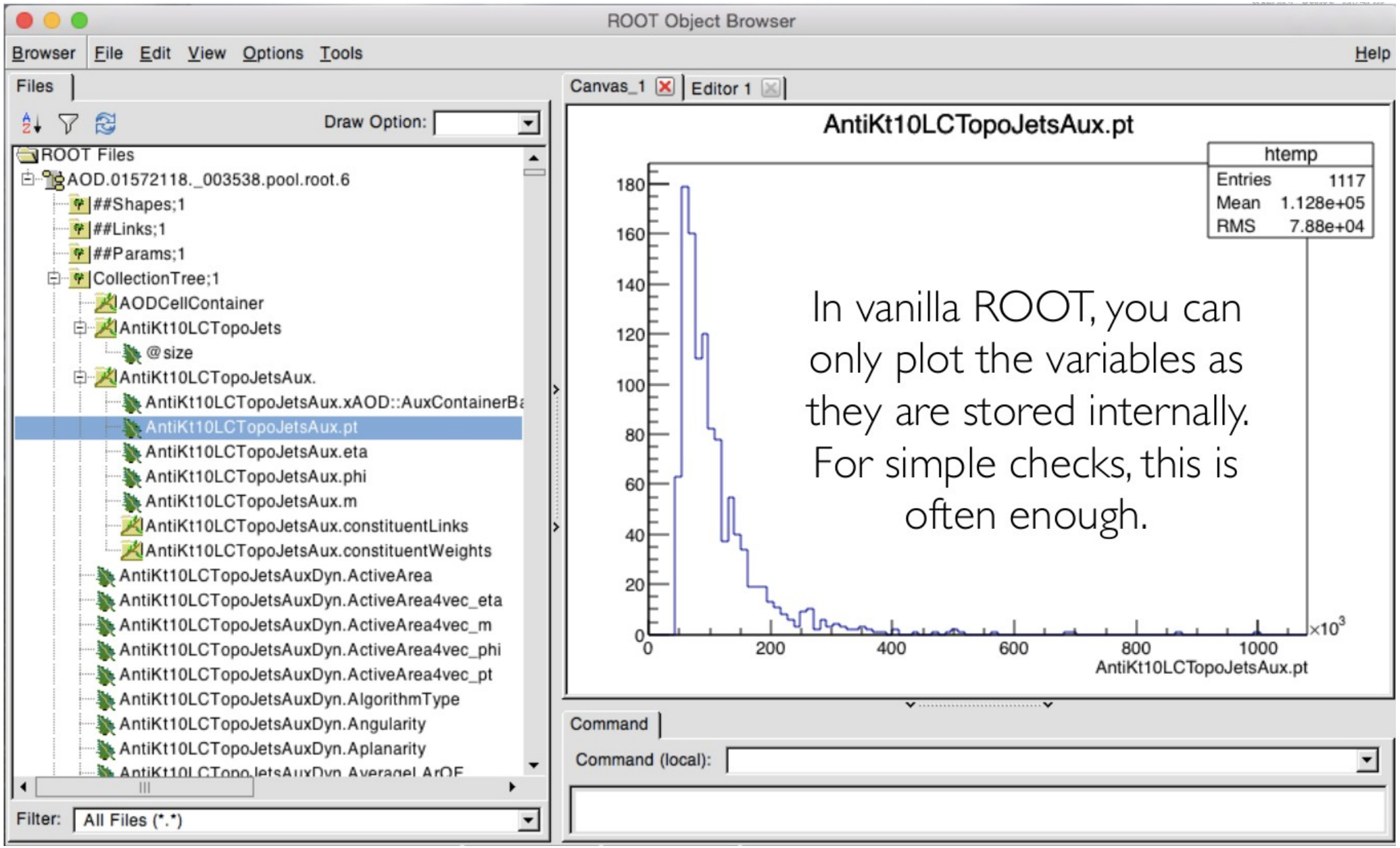
ShallowCopy example: muon systematics

```
// Load input muons
const xAOD::MuonContainer *inCont = nullptr;
ATH_CHECK( evtStore()->retrieve( inCont, "Muons" ));

// Put the following inside a loop over muon momentum systematics:

// Create shallow copy of the const input container (auto=std::pair<xAOD::MuonContainer*,xAOD::ShallowAuxContainer*>)
auto muonContShallowCopy = xAOD::shallowCopyContainer( *inCont );
ATH_CHECK( evtStore()->record( muonContShallowCopy.first,  "Muons___sysA__1up" ) );
ATH_CHECK( evtStore()->record( muonContShallowCopy.second, "Muons___sysA__1upAux." ) );

// Set the tool state to apply a systematic variation.
// Event if it is an empty variation, i.e, the nominal case, set the state of the tool to that
// to avoid that the tool is still in a systematic state from the previous event.
if( m_muCalibSmearTool->applySystematicVariation( systSet ) != CP::SystematicCode::Ok ) {
  ATH_MSG_ERROR("Cannot configure MuonCalibrationAndSmearingTool for systematic variation " << systSet.name() );
  return StatusCode::FAILURE;
}

// Loop over all Muons in the shallow-copy container
for ( xAOD::Muon* muon : *(muonContShallowCopy.first) ) {
  ATH_MSG_VERBOSE("Now iterating over the muon shallow copy container... at index=" << muon->index() );
  if ( m_muCalibSmearTool->applyCorrection(*muon) == CP::CorrectionCode::Error ) {
    ATH_MSG_ERROR("MuonCalibrationAndSmearingTool reported a CP::CorrectionCode::Error");
    return StatusCode::FAILURE;
  }
} // End: loop over Muons
```

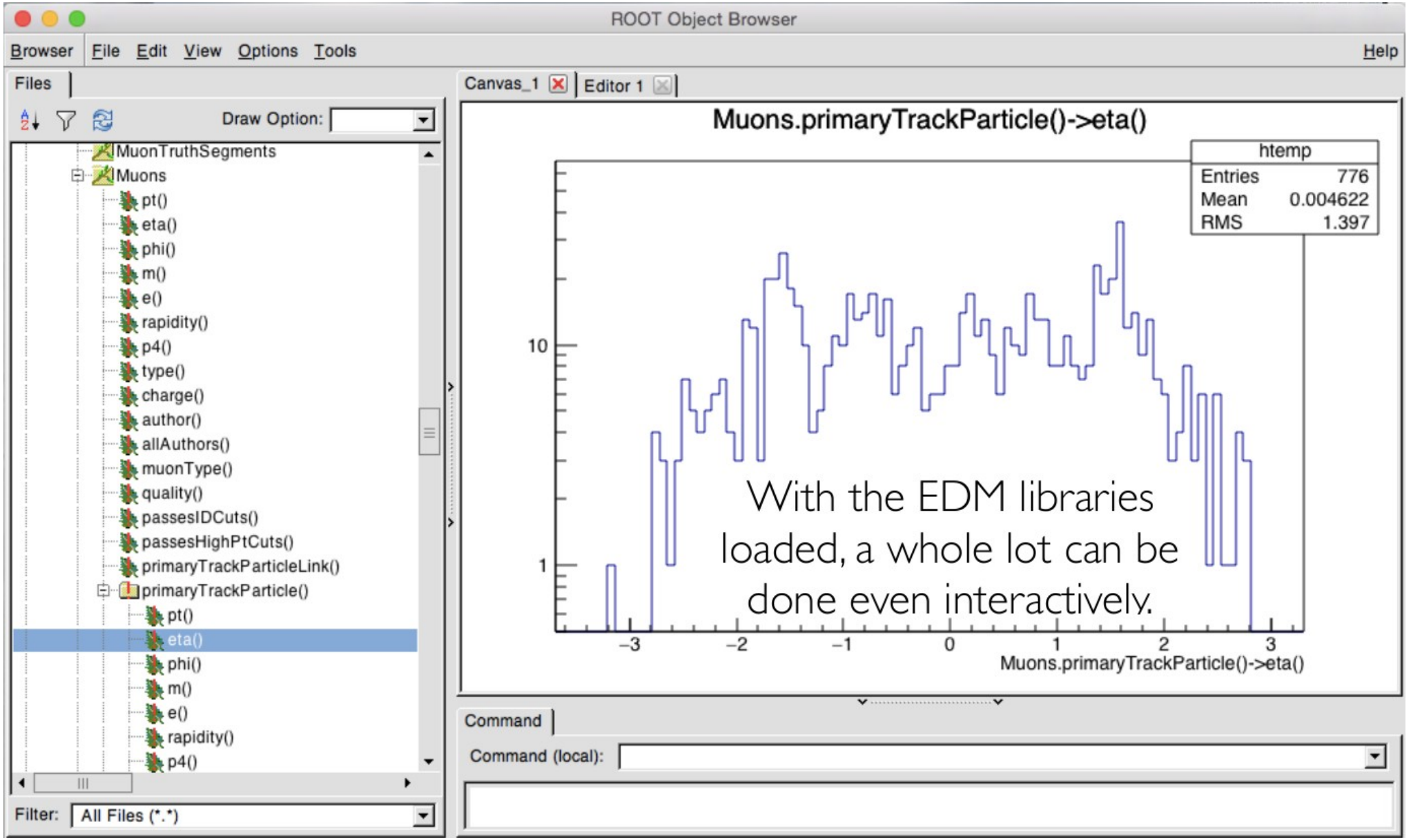
XAOD: How does it look like?



ShallowCopy example



ShallowCopy example



Objects are versioned

- Classes are versioned (e.g. xAOD::Muon_v1)
- xAOD::Muon is a typedef to the most recent version (e.g. to xAOD::Muon_v1)
- Reason: schema evolution
  - when a significant evolution of a class happens, a new class is created (e.g. xAOD::Muon_v2)
- Users and other classes only interact with the “versionless” xAOD::Muon
- When reading an old file with ROOT, where we loaded a newer version of the class, it will try to stream the old data into the new object and should (hopefully) fail.
- When reading an old file in Athena, data is converted automatically to the newest version

→ Path of the implementation of xAOD::Muon is

Event/xAOD/xAODMuon/Root/Muon_v1.cxx

References/Links

- Report of the xAOD design group:

  https://cds.cern.ch/record/1598793

- Implementation of the ATLAS run 2 data model:

  https://iopscience.iop.org/article/10.1088/1742-6596/664/7/072045

- Doxygen documentation:

  https://atlas-sw-doxygen.web.cern.ch/atlas-sw-doxygen/atlas_22.0.X-DOX/docs/html/classes.html

- Search code with lxr:

  https://acode-browser2.usatlas.bnl.gov/lxr/source/r21

References/Links

- Report of the xAOD design group:

  https://cds.cern.ch/record/1598793

- Implementation of the ATLAS run 2 data model:

  https://iopscience.iop.org/article/10.1088/1742-6596/664/7/072045

- Doxygen documentation:

  https://atlas-sw-doxygen.web.cern.ch/atlas-sw-doxygen/atlas_22.0.X-DOX/docs/html/classes.html

- Search code with lxr:

  https://acode-browser2.usatlas.bnl.gov/lxr/source/r21