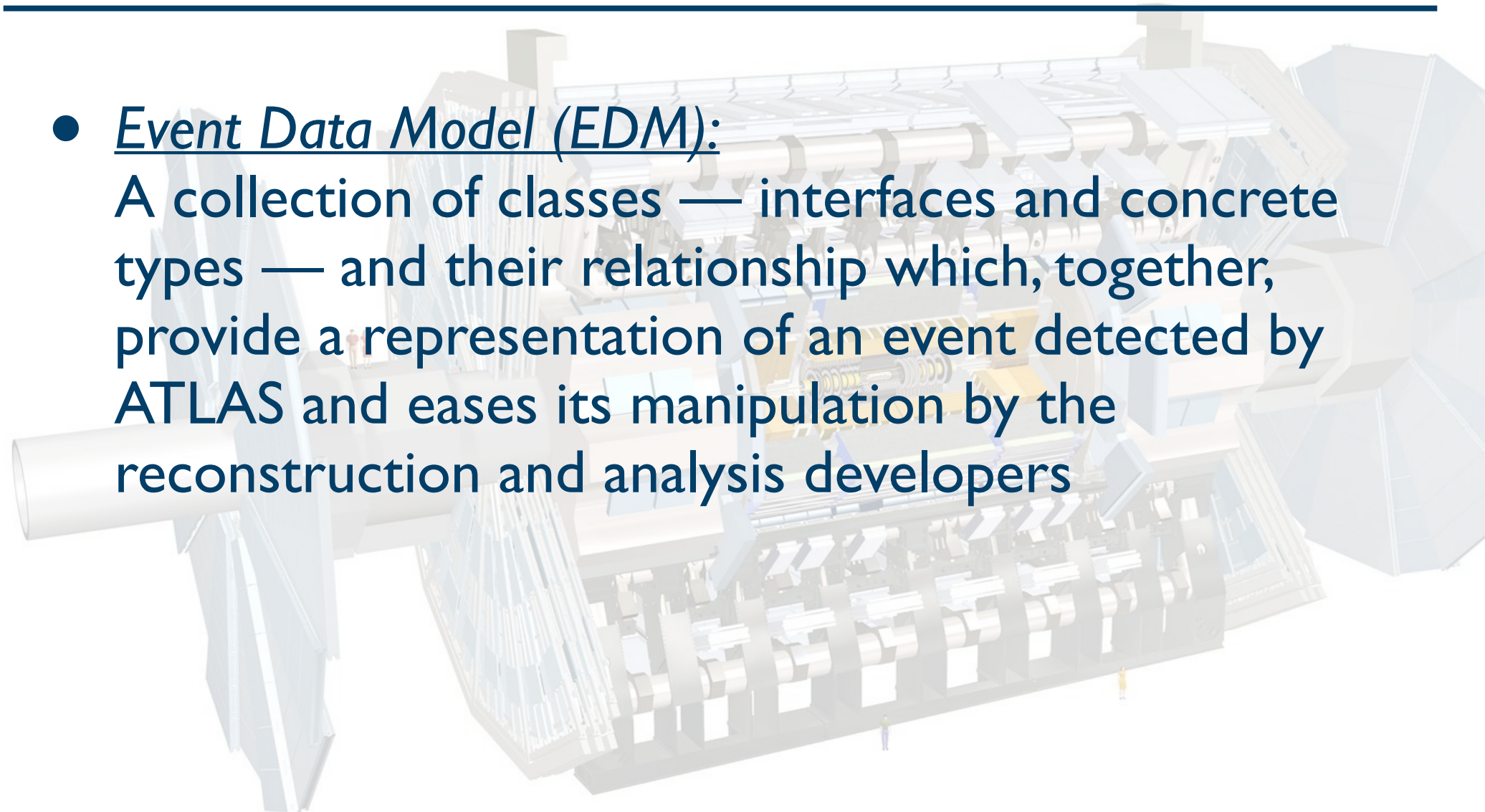
The xAOD Event Data Model

Attila Krasznahorkay

ATLAS EXPERIMENT

What is the Event Data Model?



- *Event Data Model (EDM):*
  A collection of classes — interfaces and concrete types — and their relationship which, together, provide a representation of an event detected by ATLAS and eases its manipulation by the reconstruction and analysis developers

What is the Event Data Model?

- *Event Data Model (EDM):*
  A collection of classes — interfaces and concrete types — and their relationship which, together, provide a representation of an event detected by ATLAS and eases its manipulation by the reconstruction and analysis developers

- *Or in plain words:*
  How "electrons", "muons", "jets", etc. are represented in memory, how they are stored on disk, and how we use them

Why Have an Event Data Model?

Using the same, coherent classes throughout the software has many advantages:

- *Improved commonality and coherence across*
  - detector subsystems and subgroups
  - reconstruction groups
  - physics analysis groups
- *Improved use of common software (code re-use)*
  e.g. Filtering based on 4-momentum, overlap removal, etc.
- *Improved software quality*
  Everybody using the same software results in bugs being found more quickly
- *Common definition of objects*
  Both physics- and software wise

What It Normally Looks Like

- Object oriented programming makes it very easy to construct a simple EDM
  - In fact, many people did (and still do) this exercise when they start writing physics analysis code
- The OO design principles align very naturally with the requirements for an EDM
  - *Inheritance and Polymorphism*
    One can quickly construct an `Electron` class that inherits from `Lepton`, which inherits from `Particle`
  - *Encapsulation*
    An `Electron` is normally constructed out of a `Cluster` and a `Track`, it makes sense on first order to give it such member variables

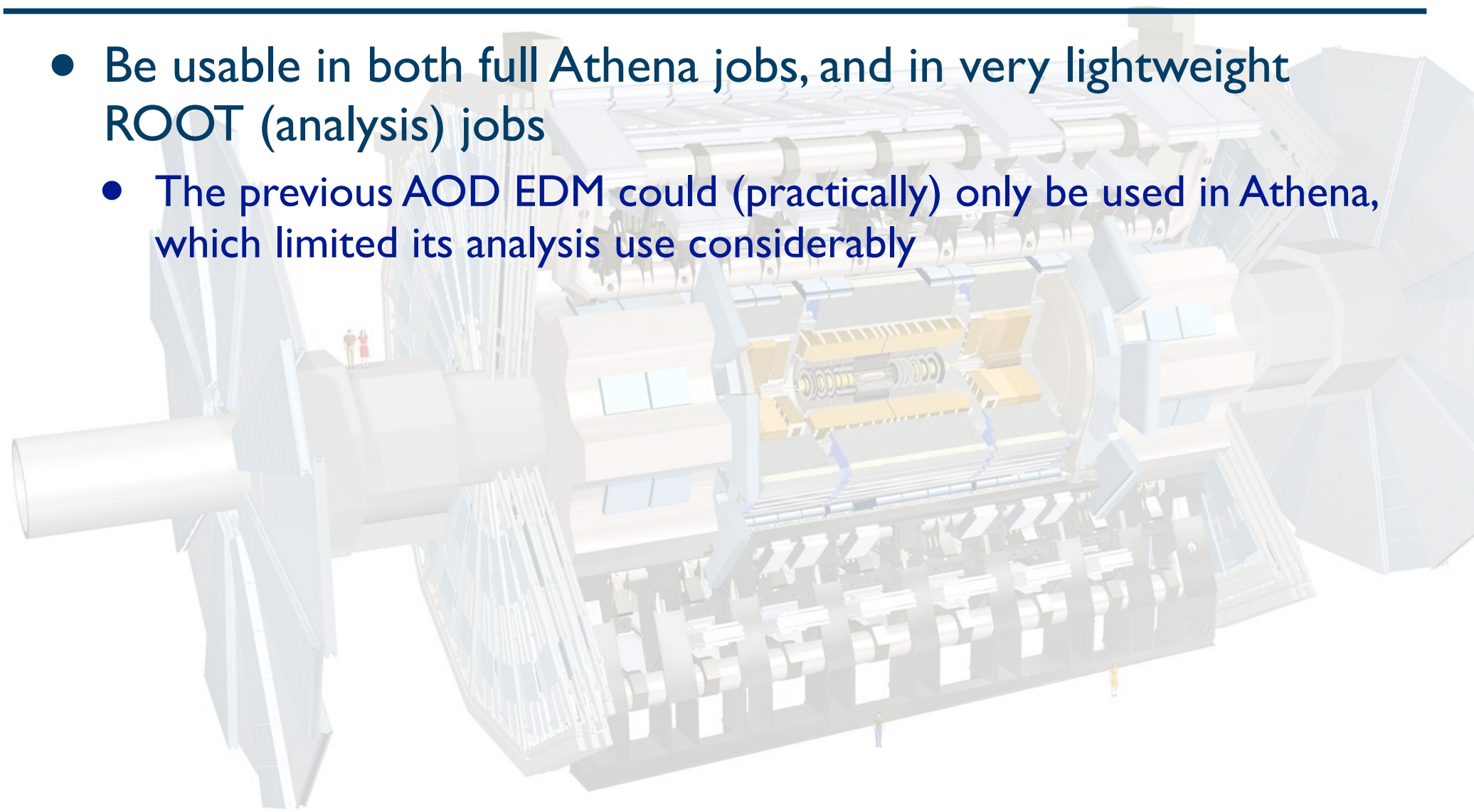
What It Normally Looks Like

- Object oriented programming makes it very ea… construct a simple EDM
  - In fact, many people did (and still do) … they start writing physics anal…
- The OO design princi… …ly with the requirements fo…
  - *Inherita…* … an `Electron` class that inherits … inherits from `Particle`

Constructing a simple EDM is very easy.
Constructing a good one is what's tricky.

…`ectron` is normally constructed out of a `Cluster` and a `Track`, it makes sense on first order to give it such member variables

xAOD EDM Requirements



- Be usable in both full Athena jobs, and in very lightweight ROOT (analysis) jobs
  - The previous AOD EDM could (practically) only be used in Athena, which limited its analysis use considerably

xAOD EDM Requirements

- Be usable in both full Athena jobs, and in very lightweight ROOT (analysis) jobs
  - The previous AOD EDM could (practically) only be used in Athena, which limited its analysis use considerably
- Provide an as easy as possible interface, to make it easy to start using
  - The EDM is mainly designed for analysis use. Clever but hard-to-understand designs were dropped in favour of simpler ones.

xAOD EDM Requirements

- Be usable in both full Athena jobs, and in very lightweight ROOT (analysis) jobs
  - The previous AOD EDM could (practically) only be used in Athena, which limited its analysis use considerably
- Provide an as easy as possible interface, to make it easy to start using
  - The EDM is mainly designed for analysis use. Clever but hard-to-understand designs were dropped in favour of simpler ones.
- Be as fast as possible
  - Simple ROOT ntuples will always be the fastest, but it was designed to be as close to them as possible.

xAOD EDM Requirements

- Be usable in both full Athena jobs, and in very lightweight ROOT (analysis) jobs
  - The previous AOD EDM could (practically) only be used in Athena, which limited its analysis use considerably
- Provide an as easy as possible interface, to make it easy to start using
  - The EDM is mainly designed for analysis use. Clever but hard-to-understand designs were dropped in favour of simpler ones.
- Be as fast as possible
  - Simple ROOT ntuples will always be the fastest, but it was designed to be as close to them as possible.
- Be as flexible as possible
  - The user is allowed to add/remove variables to/from objects (containers) at runtime

General xAOD Features

- Objects are split in two:
  - *Interface objects*
    These are objects that provide a usable UI for the type, but don't hold on to any persistent data
  - *Payload objects ("auxiliary data store")*
    These are relatively dumb objects that just allocate continuous memory for the data, and allow the interface objects to access this data.
- This split means that:
  - Data storage technology is independent of the interface that we provide for instance for electrons.
  - We can partially read information about objects during analysis
  - Any "regular" code should only ever create auxiliary objects (when creating new xAOD objects), but never manipulate them directly
- All EDM code is put into the `xAOD::` namespace

xAOD::EventInfo

- Contains general information about the current event. Like:
  - What's the pileup of the event?
  - What's the run and event number of the event?
  - Only one object, with the key "EventInfo" is in a given event
- Doxygen link: here

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.

xAOD::IParticle

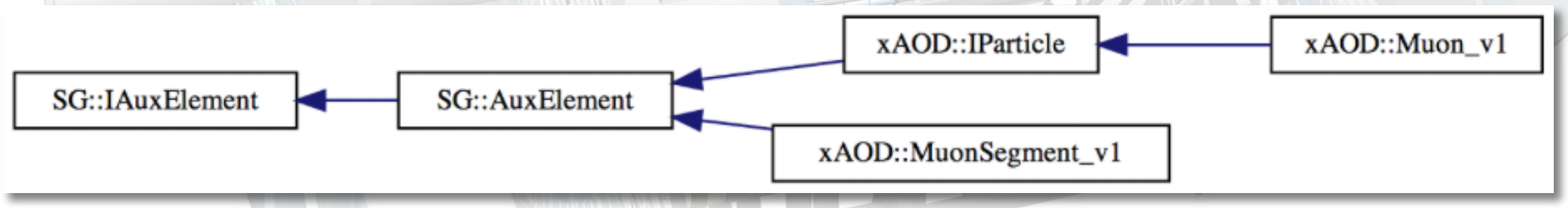
- A pure virtual interface for:
  - All particle types (electron, muon, jet, …)
  - Clustered energy deposits in the calorimeter
  - Charged and neutral particle trajectories (tracks)
- Provides access to the basic particle properties, like their 4-momentum (Doxygen link)

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.

xAOD Class Structure

- `xAOD::IParticle` is not at the base of the inheritance structure
  - Most xAOD types inherit from it, but some (like `xAOD::EventInfo`) don't
- The class that all xAOD interface classes inherit from is `SG::AuxElement`



- It's this class that provides much of the *infrastructure* for the EDM implementation

SG::AuxElement

- Provides standardised access to all variables stored in the object's auxiliary store
- Allows the user to add new variables to an object at runtime
- Regular users only really need to know a few of its functions…
- Doxygen link: here

template<class T >

AuxDataTraits< T >::reference_type **auxdata** (const std::string &name)

Fetch an aux data variable, as a non-const reference.

template<class T >

AuxDataTraits< T >

::const_reference_type **auxdata** (const std::string &name) const

Fetch an aux data variable, as a const reference.

template<class T >

AuxDataTraits< T >::reference_type **auxdecor** (const std::string &name) const

Fetch an aux decoration, as a non-const reference.

Versioning

- As you should've noticed by now, concrete xAOD types always have a name ending in `_vX`
  - These classes are under "xAOD<Bla>/versions/" in the xAOD packages
  - But it's important that in your code you never use the *versioned names* directly. Instead, always use the *versionless* typedefs.
- A versionless typedef is provided for each EDM class under "xAOD<Bla>/", and points at the most recent version of the class
- When you read an old xAOD file in Athena, the framework converts the `vX` version to the latest `vY` version automatically for you, allowing you to read different files with the same offline software version
  - But in ROOT you can only read xAOD files with the appropriate analysis release versions, which use the same version of the classes as the file

DataVector

- Almost all EDM objects are stored in containers
  - Showing that we have a variable number of particles event by event
  - Only objects describing the entire event (like `xAOD::EventInfo`) are stored on their own
- We use a custom vector type, `DataVector<T>` to create such containers
  - For all *simple* intents and purposes `DataVector<T>` behaves just like `std::vector<T*>`
    - Needed so we can use polymorphism inside the containers
  - But also includes code for implementing the interface <-> auxiliary store separation
  - And most importantly allows us to define inheritance relationships between the types
    - Such that since `xAOD::Muon` inherits from `xAOD::IParticle`, `DataVector<xAOD::Muon>` also inherits from `DataVector<xAOD::IParticle>`

ElementLink

- Objects can point to each other
  - Since objects can get large both in memory and on disk, we don't save multiple instances of them.
  - Instead we put all calorimeter clusters and all inner detector track particles into their own containers, and electrons just have smart pointers to a cluster and a track particle in those containers
- In memory a bare C/C++ pointer would be enough, but to write these links to disk in a smart way, we use a smart pointer type, `ElementLink<T>`
  - There's also a `DataLink<T>` type, but that's not used in the analysis EDM at the moment
- Practically an `ElementLink` is two numbers:
  - The first one identifying the container of the target object
  - The second one storing the index of the target in its container

ElementLink

- Objects can point to each other
  - Since objects can get large both in memory a… save multiple instances of them.
  - Instead we put all calorimeter c… particles into their own c… pointers to a cluste…
- In memory … …gh, but to write … use a smart

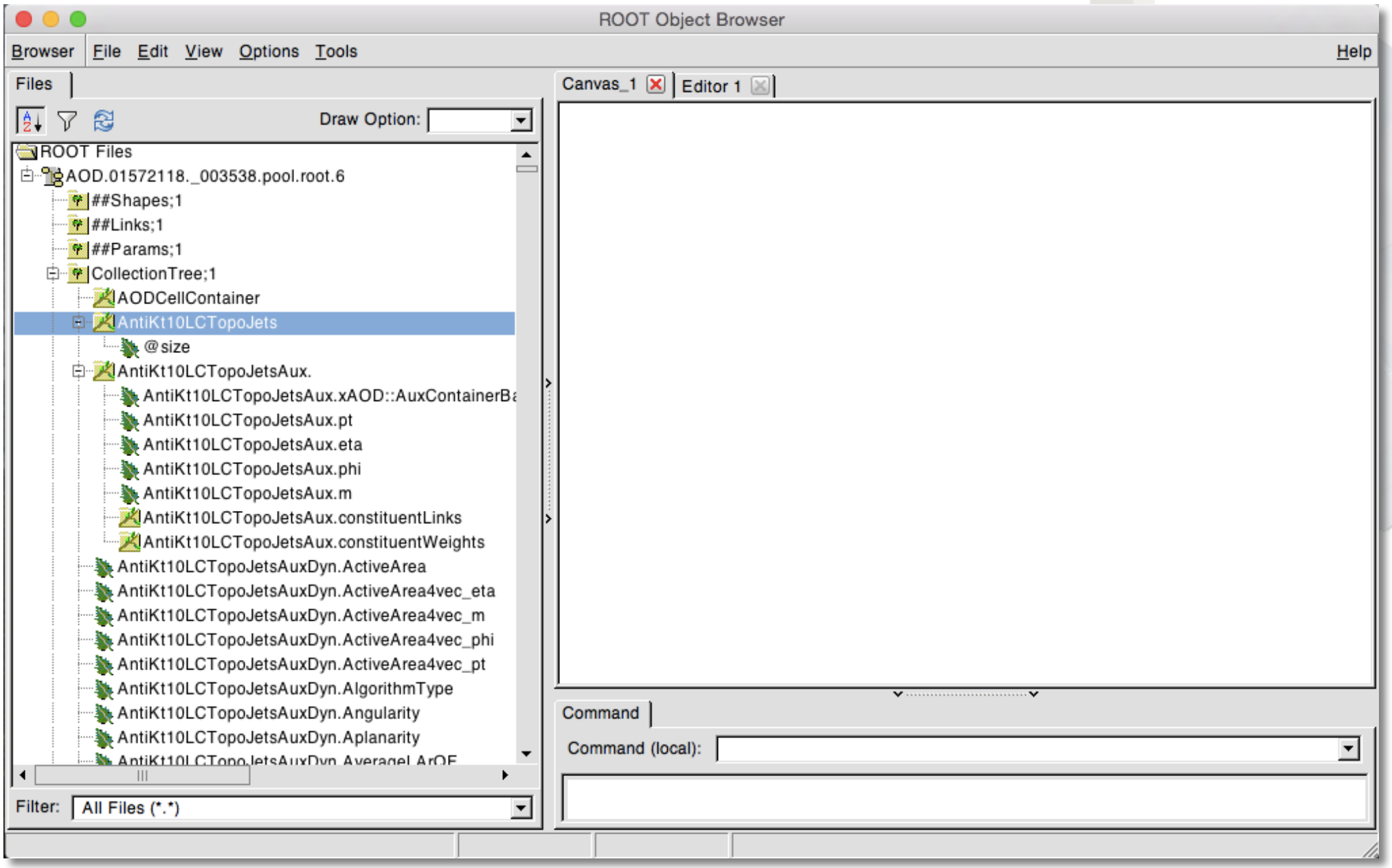
…ype, but that's not used in the analysis

…ementLink is two numbers:

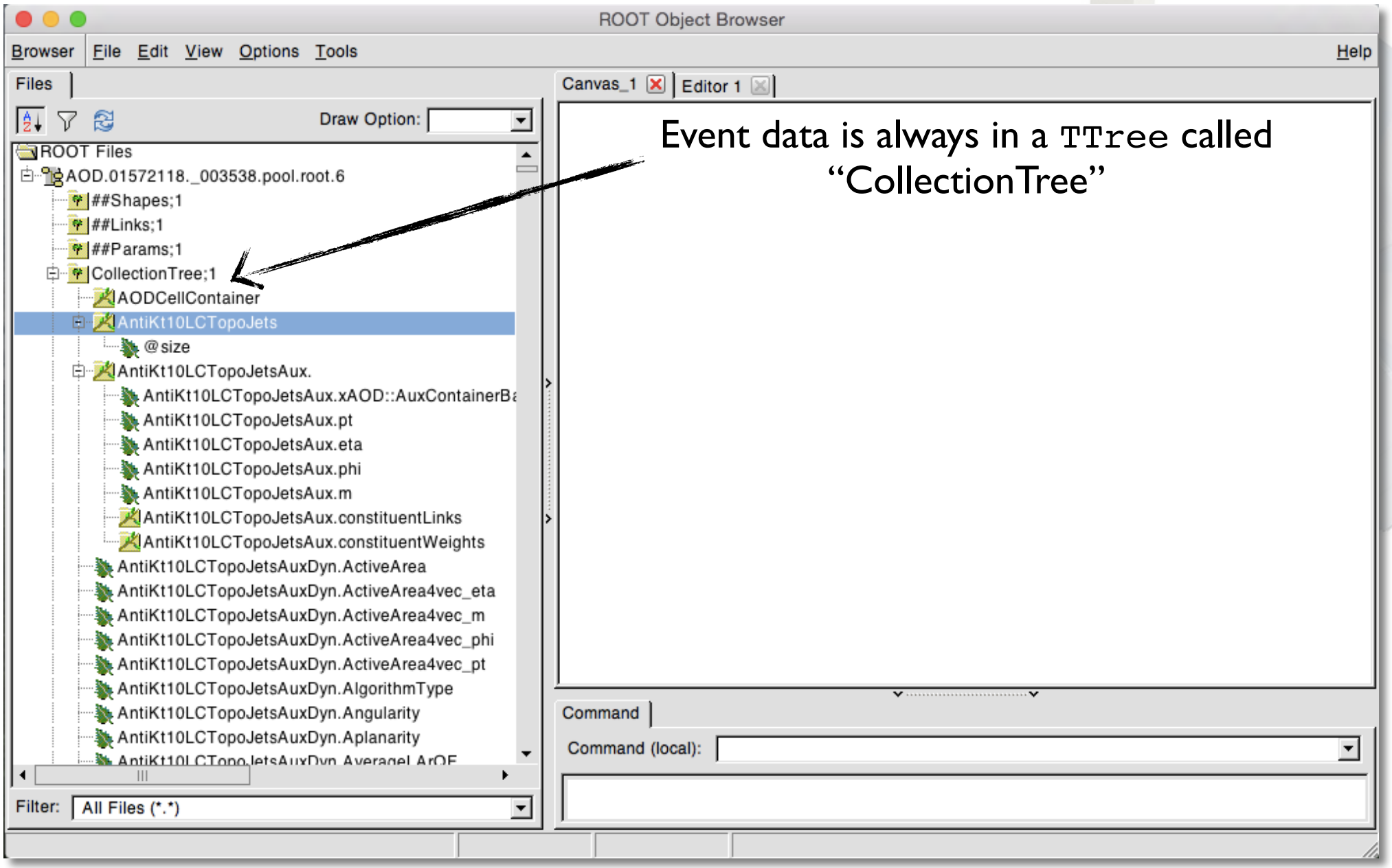
- …one identifying the container of the target object
- …he second one storing the index of the target in its container

For simple operations you don't use this type directly, but it's used throughout the EDM behind the scenes.

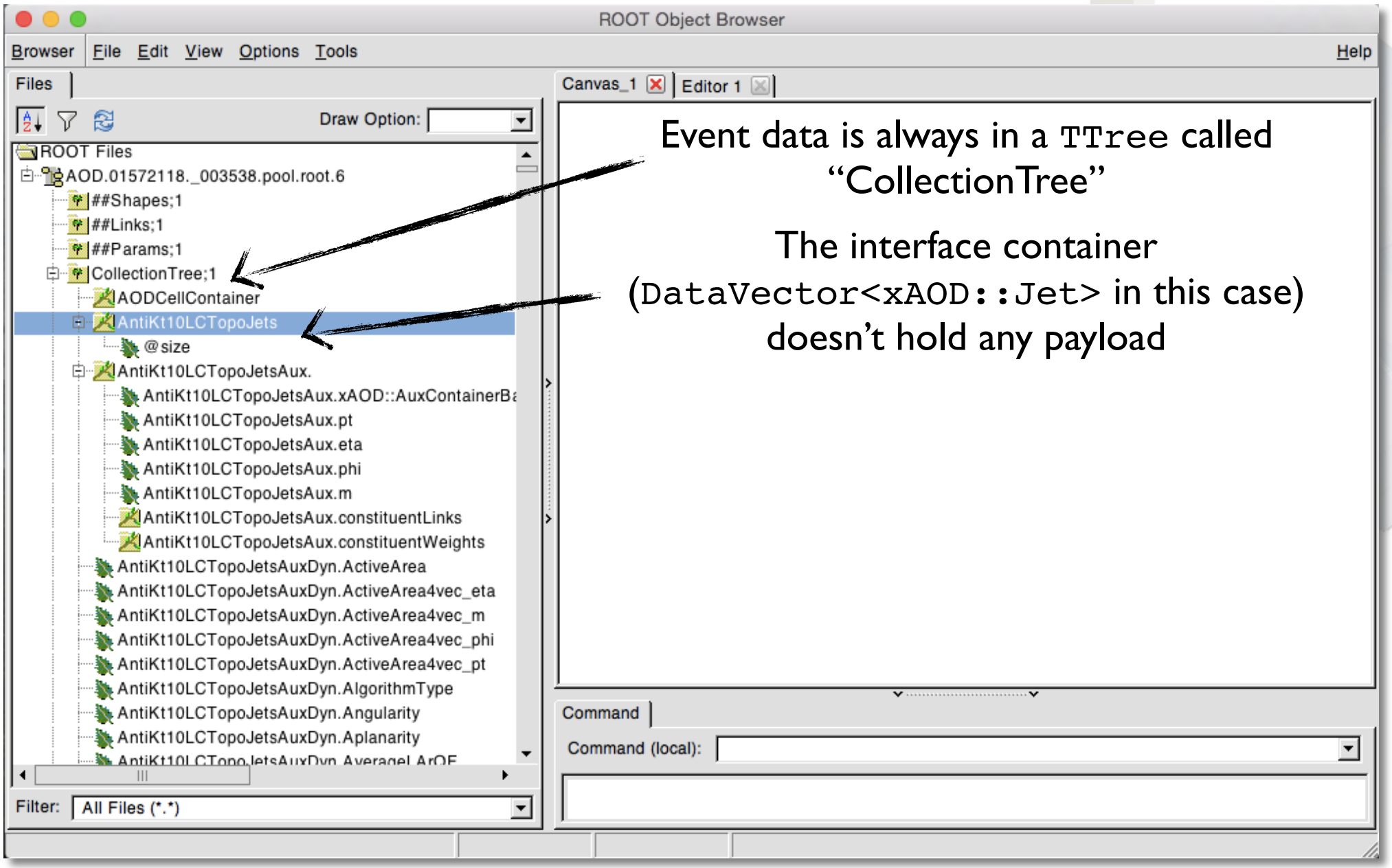
What It Looks Like (1)



What It Looks Like (1)



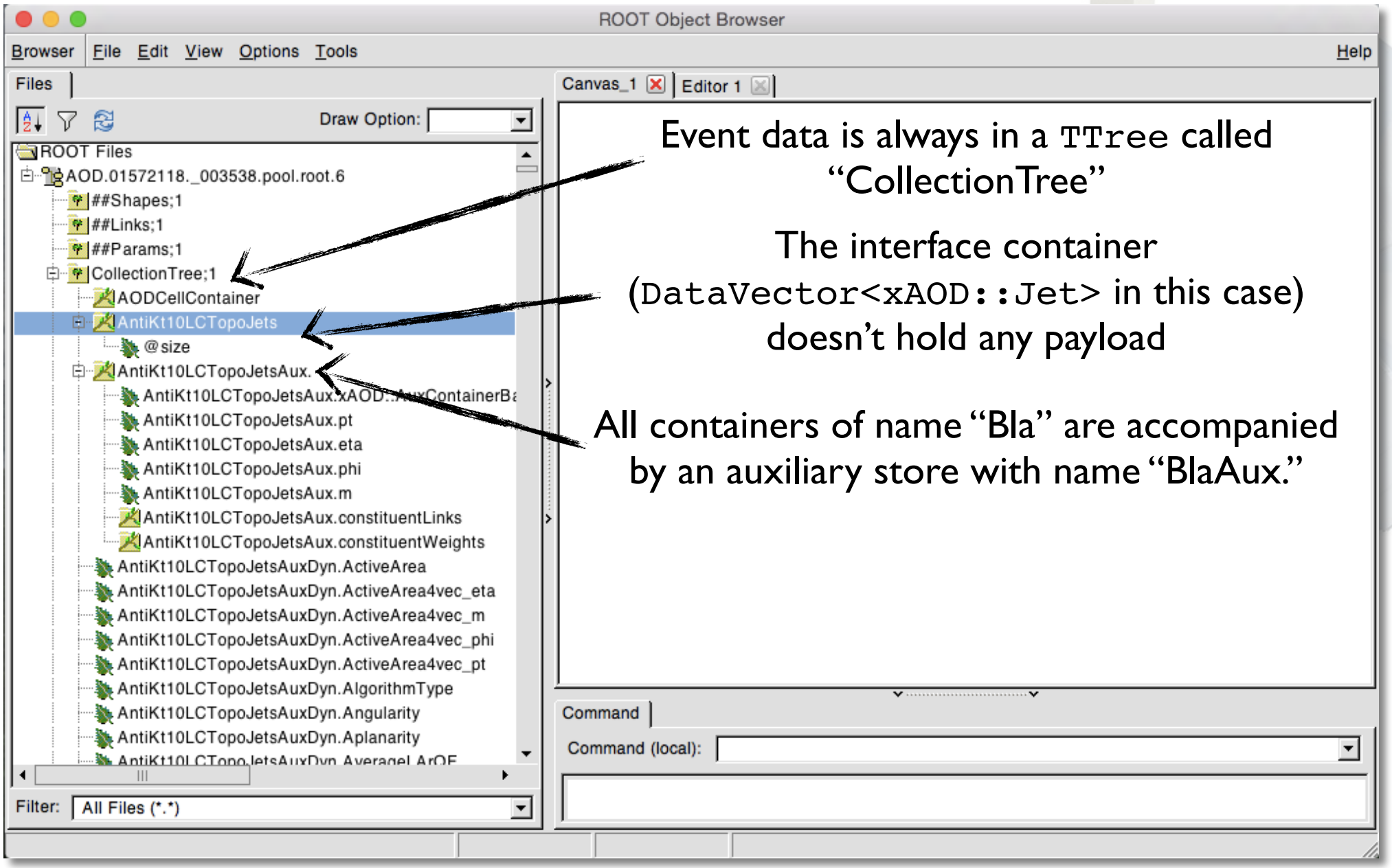
What It Looks Like (1)



Event data is always in a `TTree` called "CollectionTree"

The interface container (`DataVector<xAOD::Jet>` in this case) doesn't hold any payload

What It Looks Like (1)

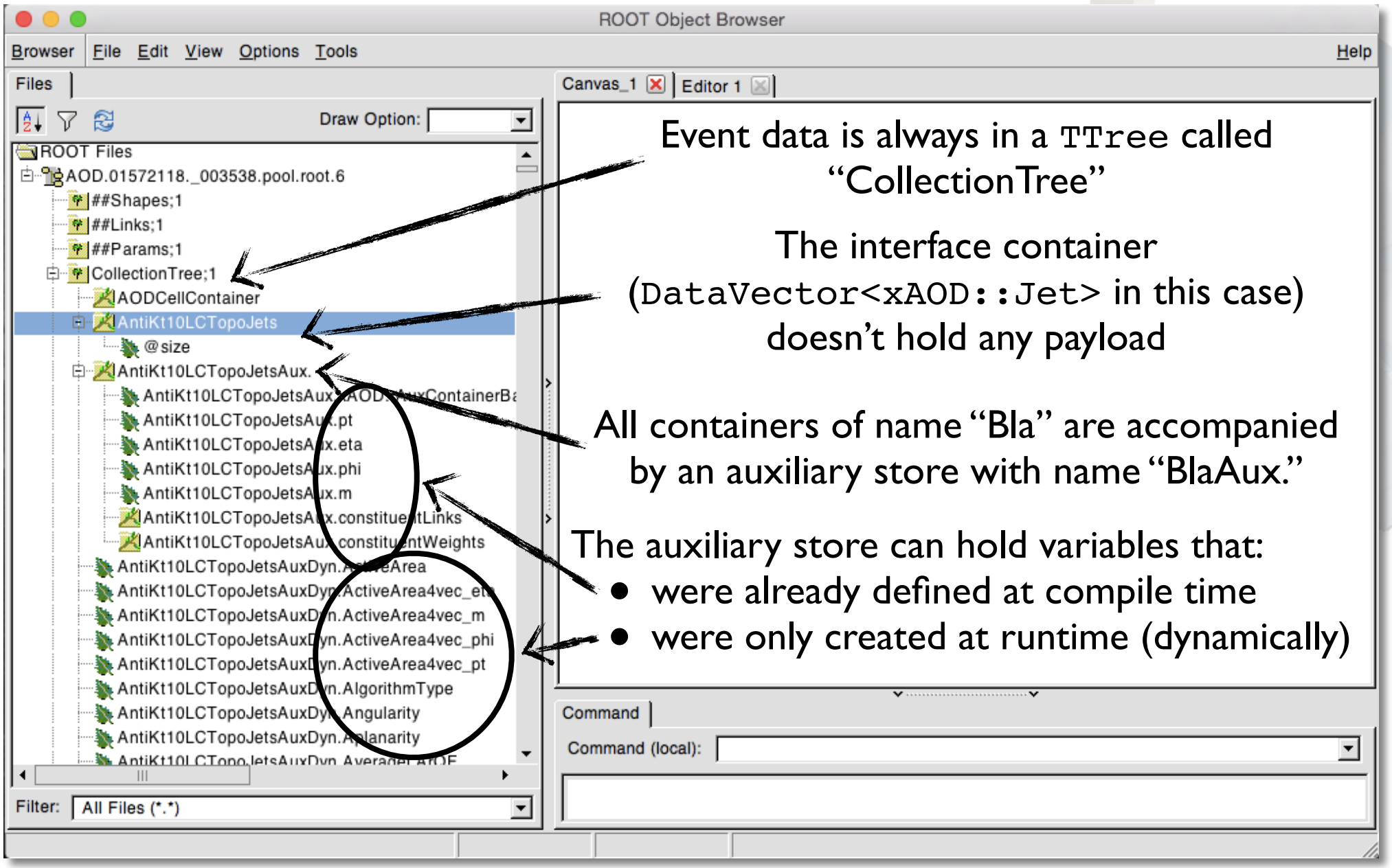


Event data is always in a `TTree` called "CollectionTree"

The interface container (`DataVector<xAOD::Jet>` in this case) doesn't hold any payload

All containers of name "Bla" are accompanied by an auxiliary store with name "BlaAux."

What It Looks Like (1)



Event data is always in a `TTree` called "CollectionTree"

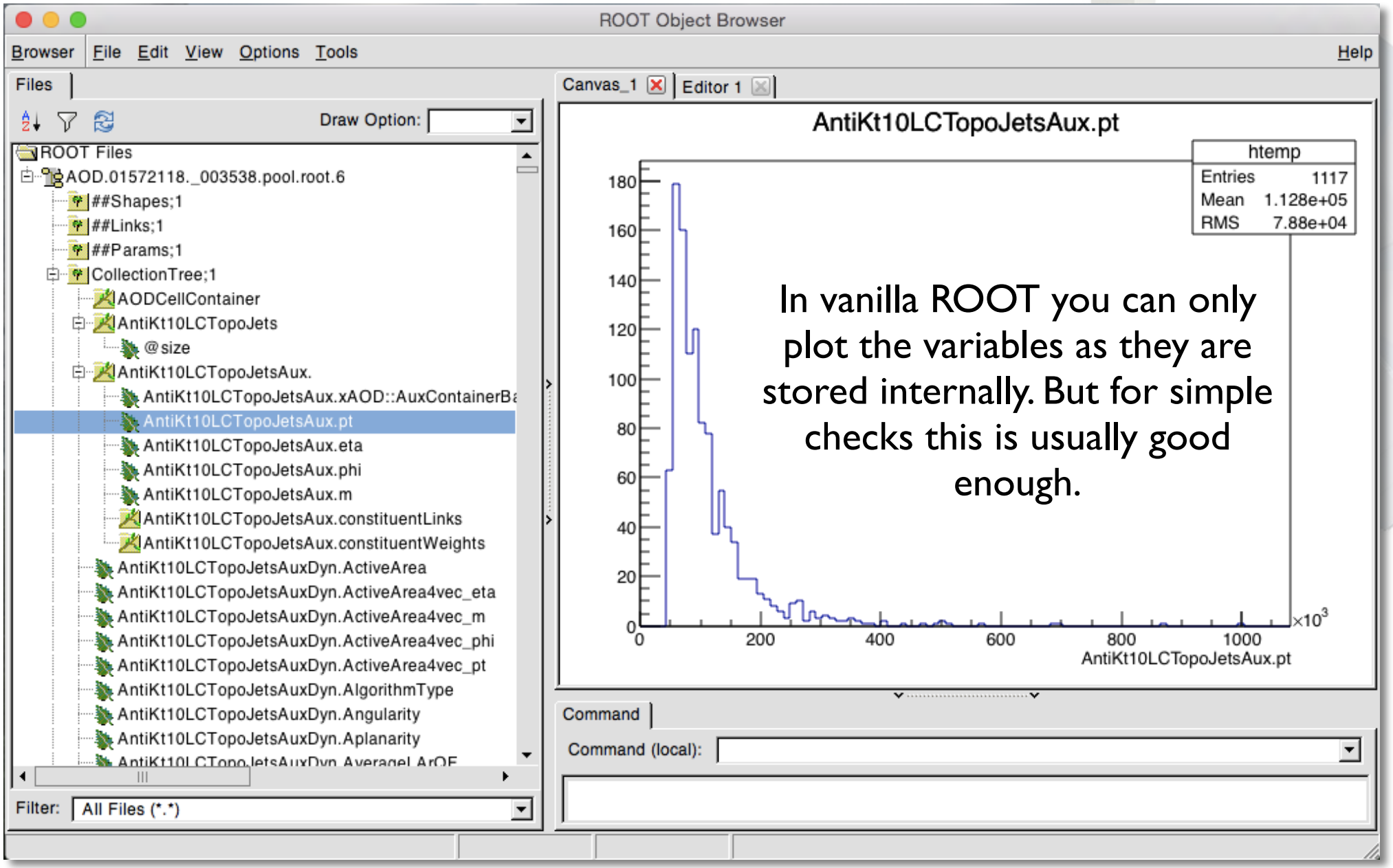
The interface container (`DataVector<xAOD::Jet>` in this case) doesn't hold any payload

All containers of name "Bla" are accompanied by an auxiliary store with name "BlaAux."

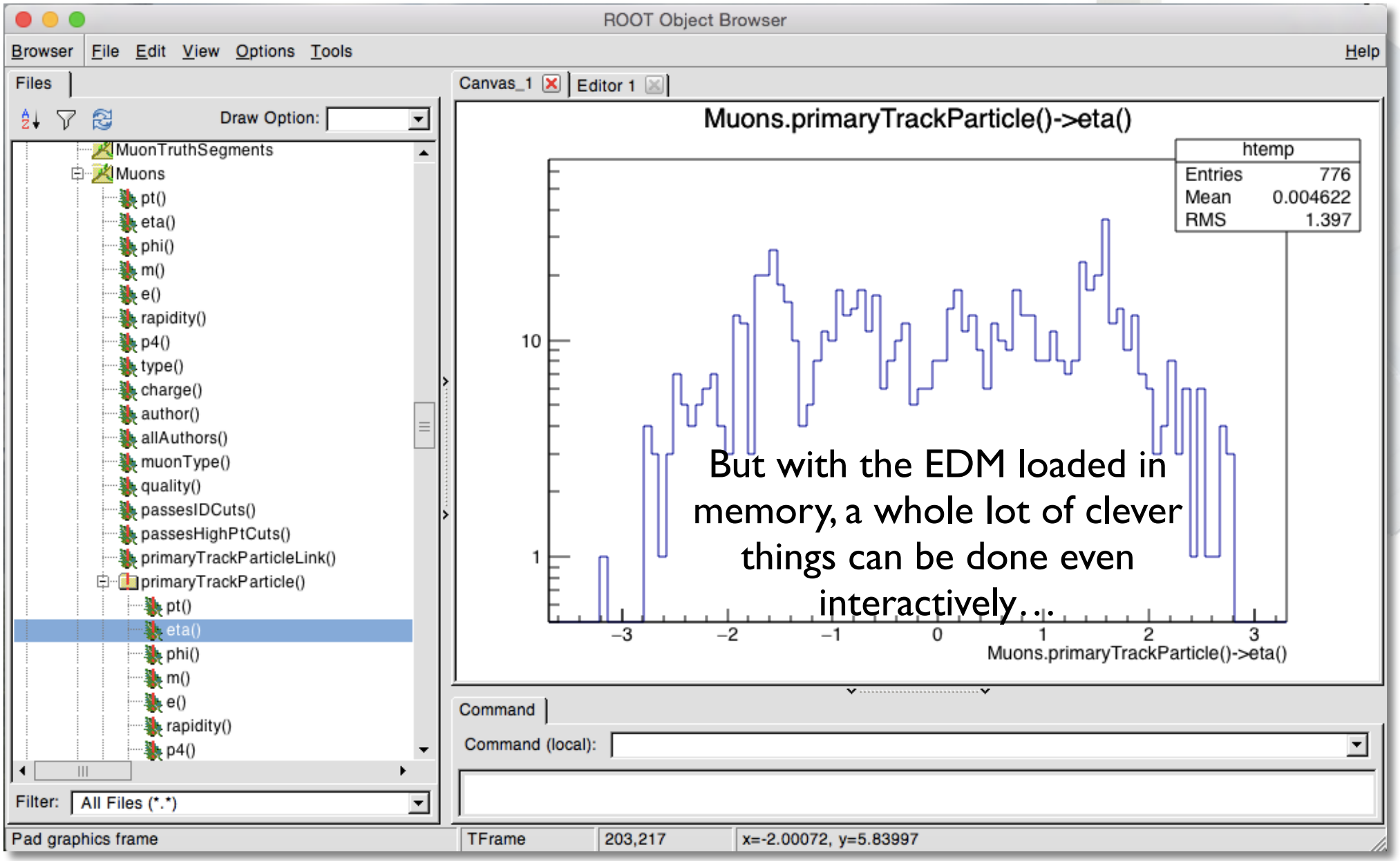
The auxiliary store can hold variables that:

- were already defined at compile time
- were only created at runtime (dynamically)

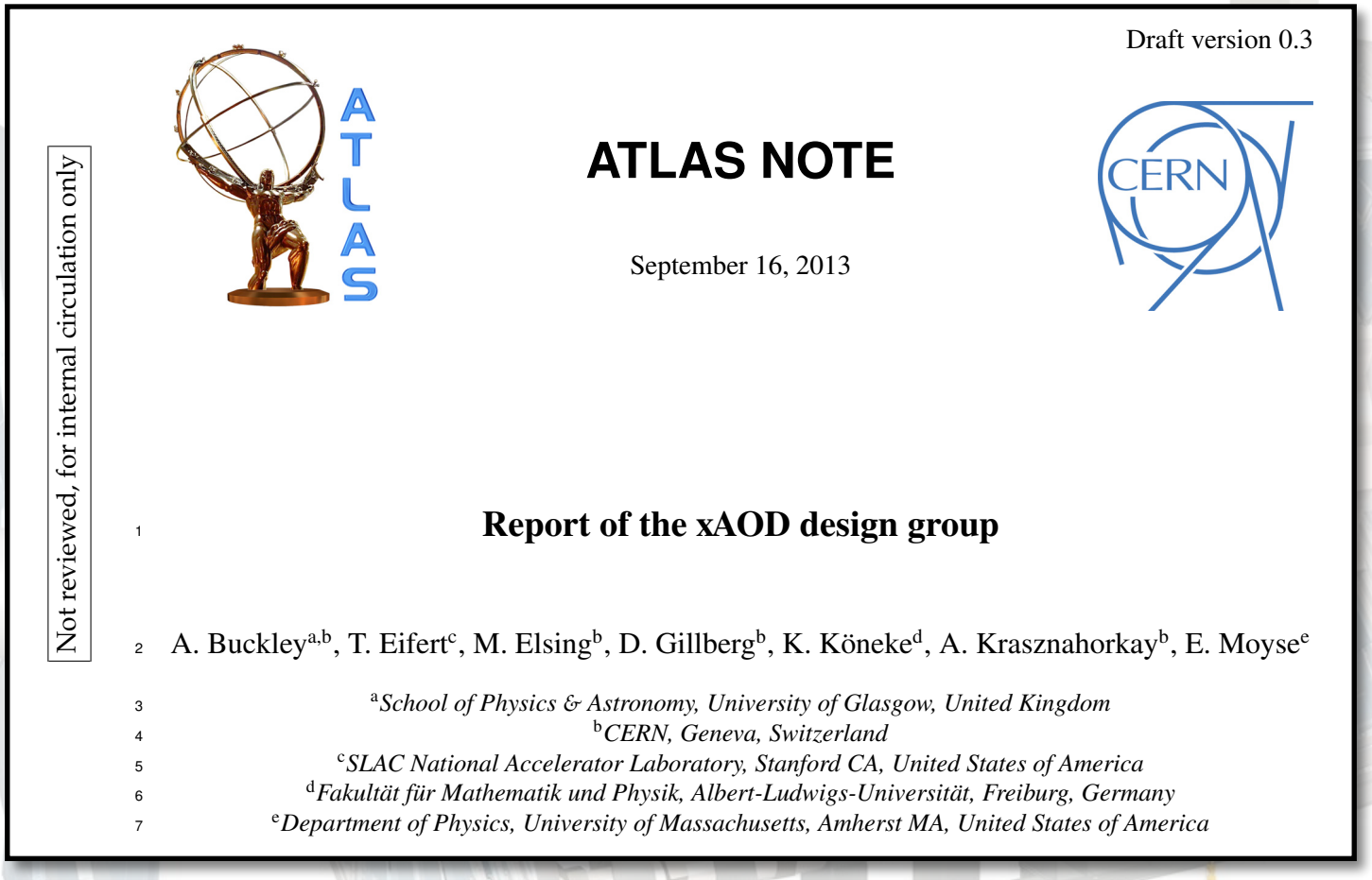
What It Looks Like (2)



What It Looks Like (3)



xAOD EDM Design Details



Draft version 0.3

Not reviewed, for internal circulation only

ATLAS NOTE

September 16, 2013

Report of the xAOD design group

A. Buckley[a,b], T. Eifert[c], M. Elsing[b], D. Gillberg[b], K. Köneke[d], A. Krasznahorkay[b], E. Moyse[e]

[a] *School of Physics & Astronomy, University of Glasgow, United Kingdom*
[b] *CERN, Geneva, Switzerland*
[c] *SLAC National Accelerator Laboratory, Stanford CA, United States of America*
[d] *Fakultät für Mathematik und Physik, Albert-Ludwigs-Universität, Freiburg, Germany*
[e] *Department of Physics, University of Massachusetts, Amherst MA, United States of America*

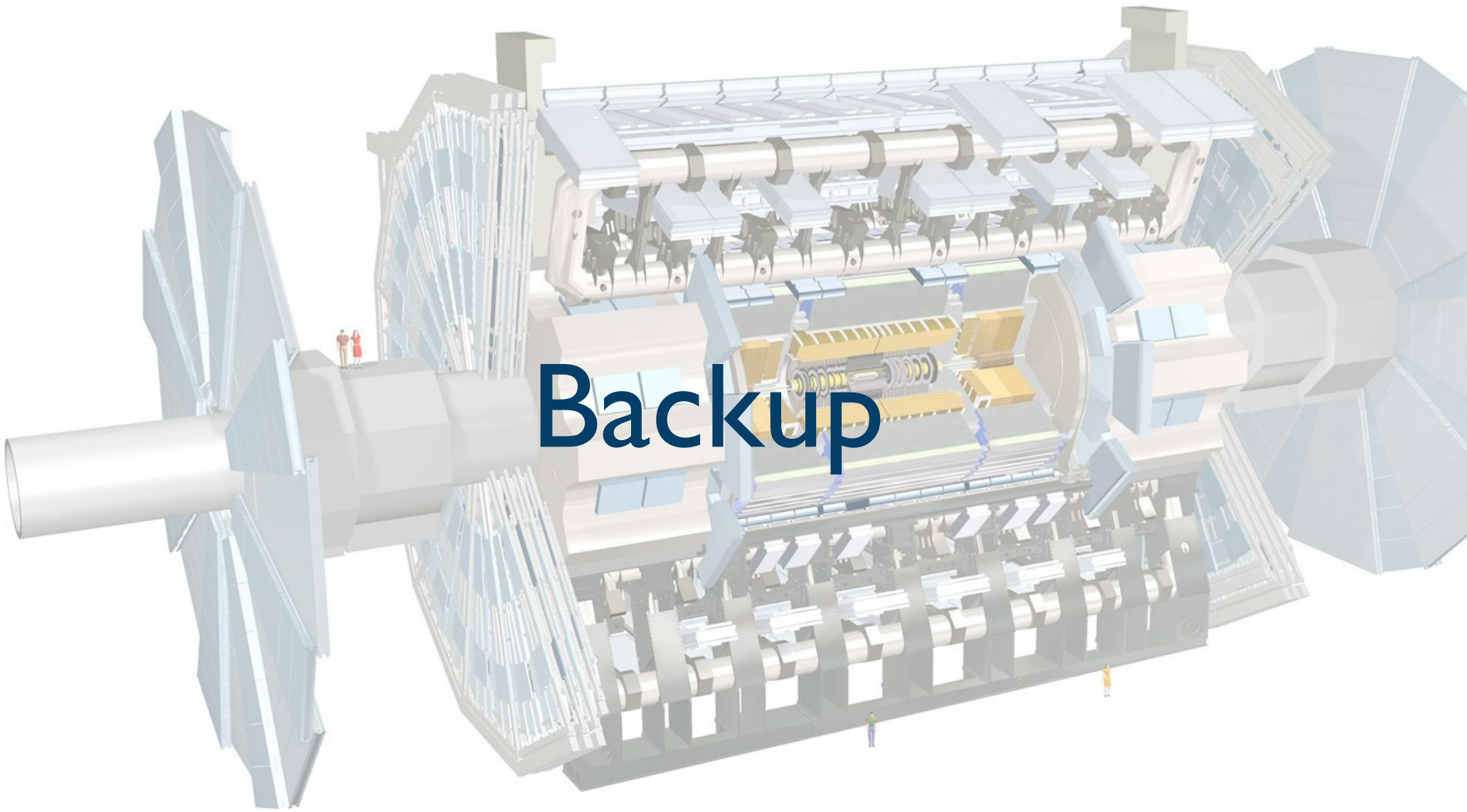
- The xAOD EDM design considerations are described in https://cds.cern.ch/record/1598793
- Includes some extra details that I didn't discuss in the talk

Additional Sources of Information

- We strive to make the Doxygen documentation as usable as possible, and for most types it's very functional already
  - http://atlas-computing.web.cern.ch/atlas-computing/links/nightlyDocDirectory/allpackages.html
- But if you completely run out of ideas:
  - Try looking at the classes themselves. All xAOD packages are under `Event/xAOD` in the offline repository.
  - Send a question to atlas-sw-analysis-forum@cern.ch

Summary

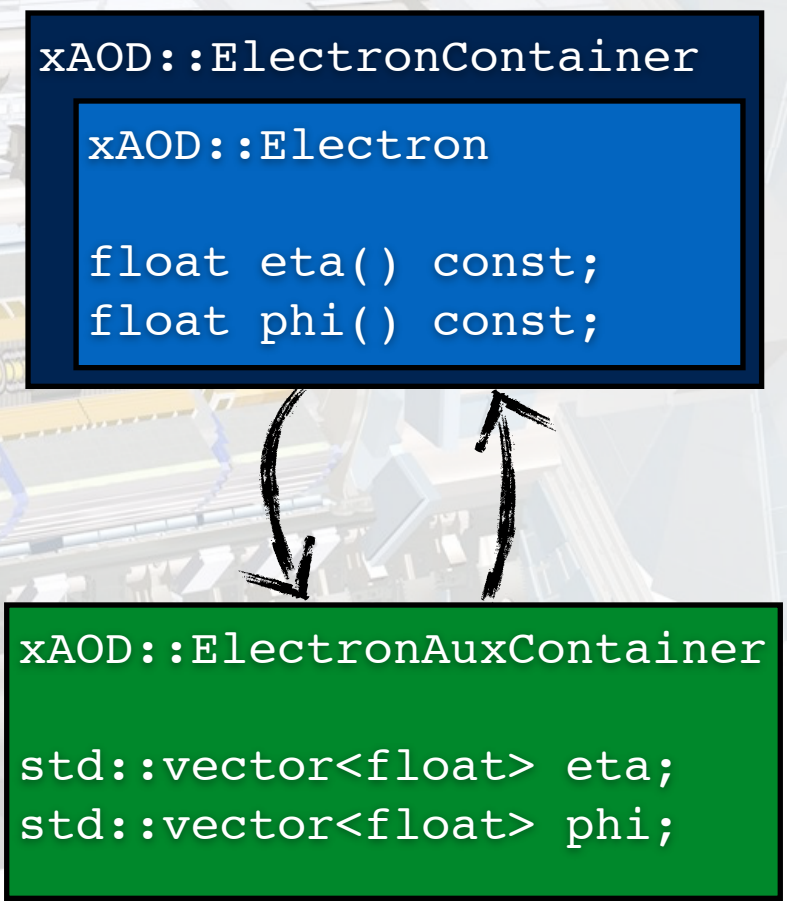
- Can only scratch the surface in such a short time
  - You will (hopefully) get familiar with the code as you use it
- Things to remember:
  - You should always only manipulate the objects using the versionless interface types, but you should be aware that the data payload is stored in a separate object in memory
  - All particle types inherit from `xAOD::IParticle`
  - Doxygen is a very useful tool for learning about the classes
  - If you have questions, send them to atlas-sw-analysis-forum@cern.ch



Backup

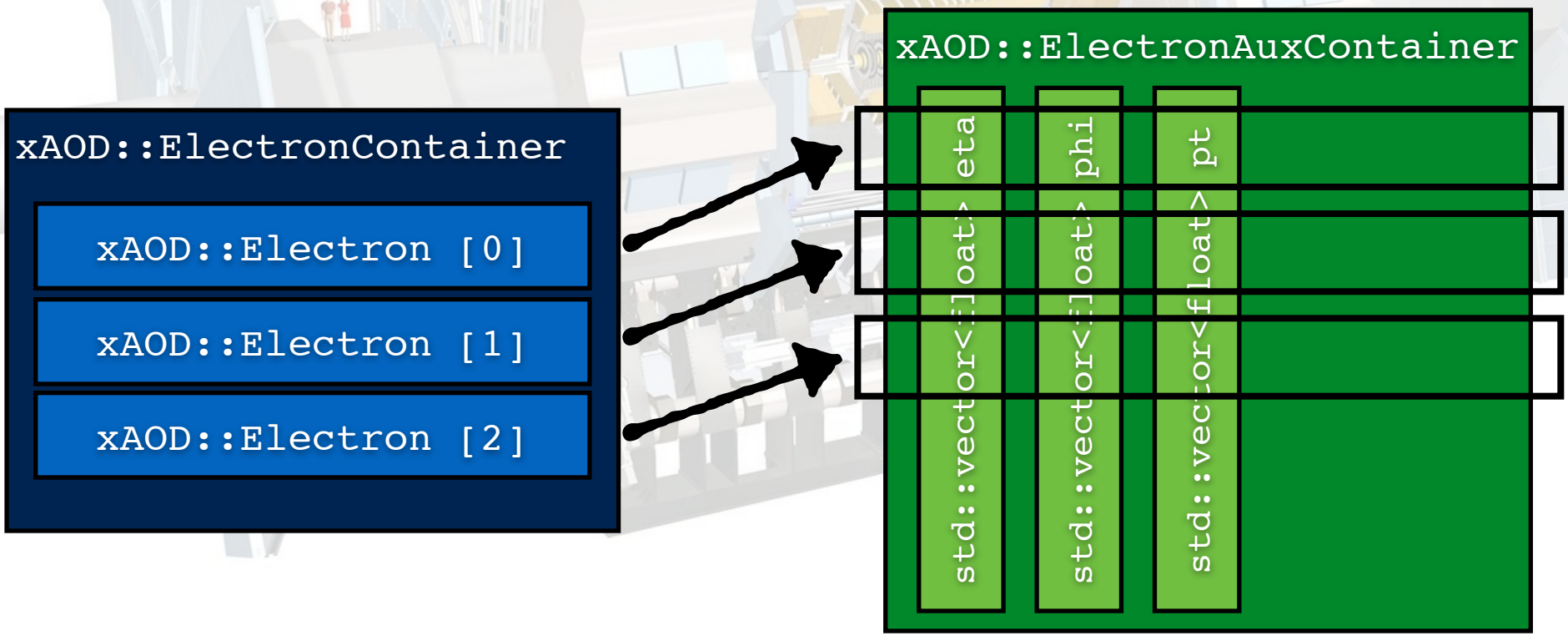
The xAOD Format (1)

- Merging the good properties of D3PD and AOD files
  - Using some pretty advanced code, hidden in the background
- Classes that the user interacts with are just "interfaces" to data stored in an "auxiliary store"
  - Allows us to freely remove/add properties from/to objects during the analysis process, while still using the same xAOD interface class in the code

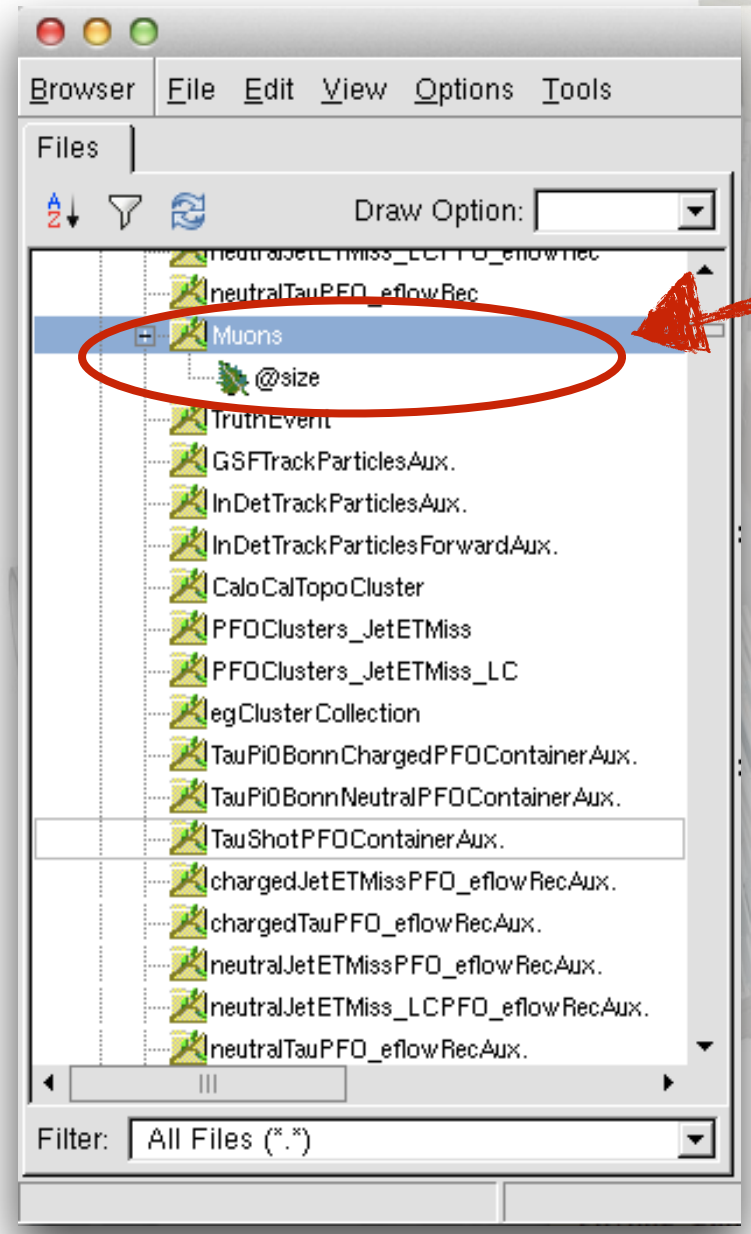


The xAOD Format (2)

- Is technically quite smart code…
  - Provides an "array of structs" interface to data held as "struct of arrays" in memory
  - This "struct of arrays" layout allows us to write files that can be browsed similar to D3PD files

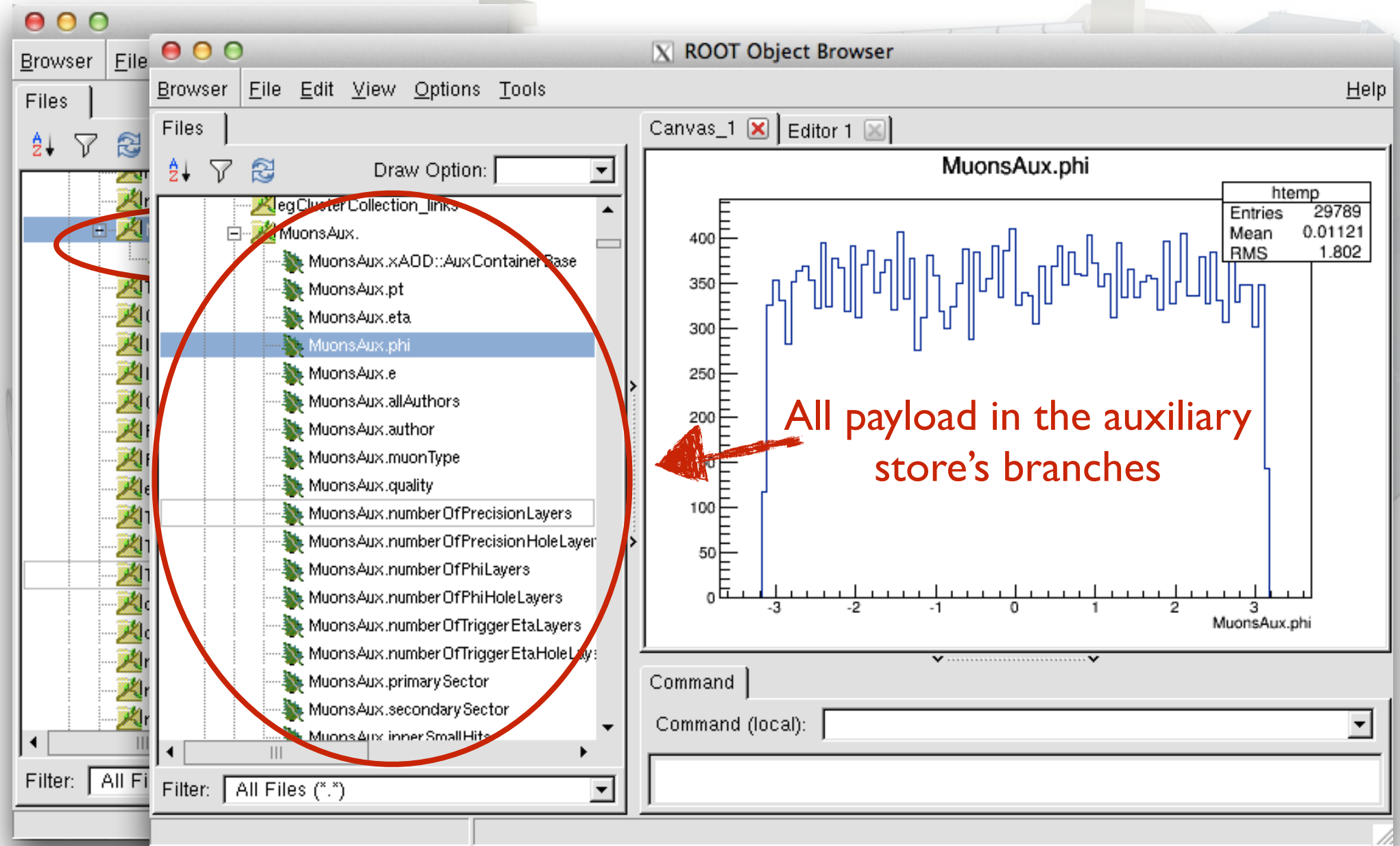


The xAOD Files (I)

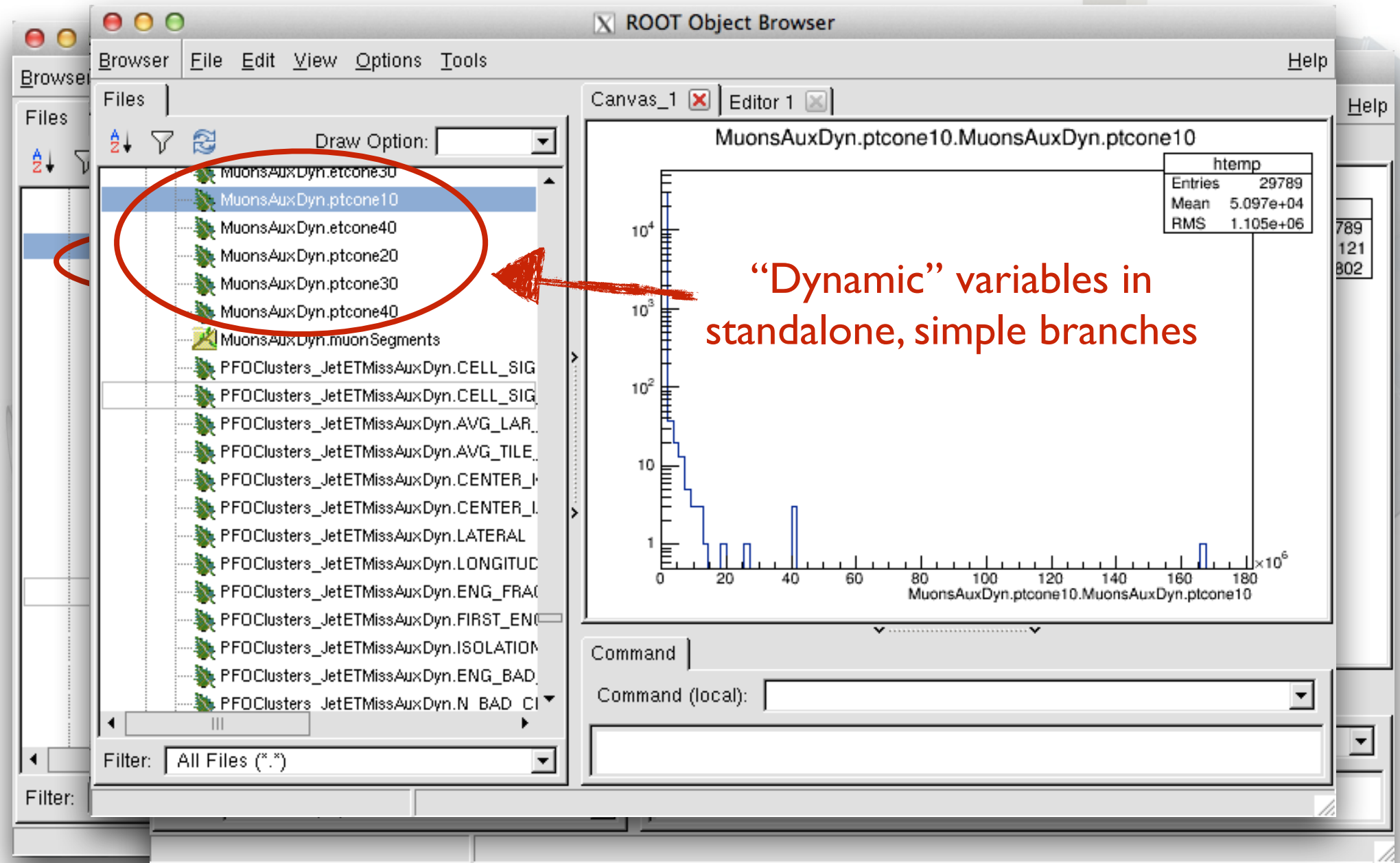


No payload in the interface container's branch

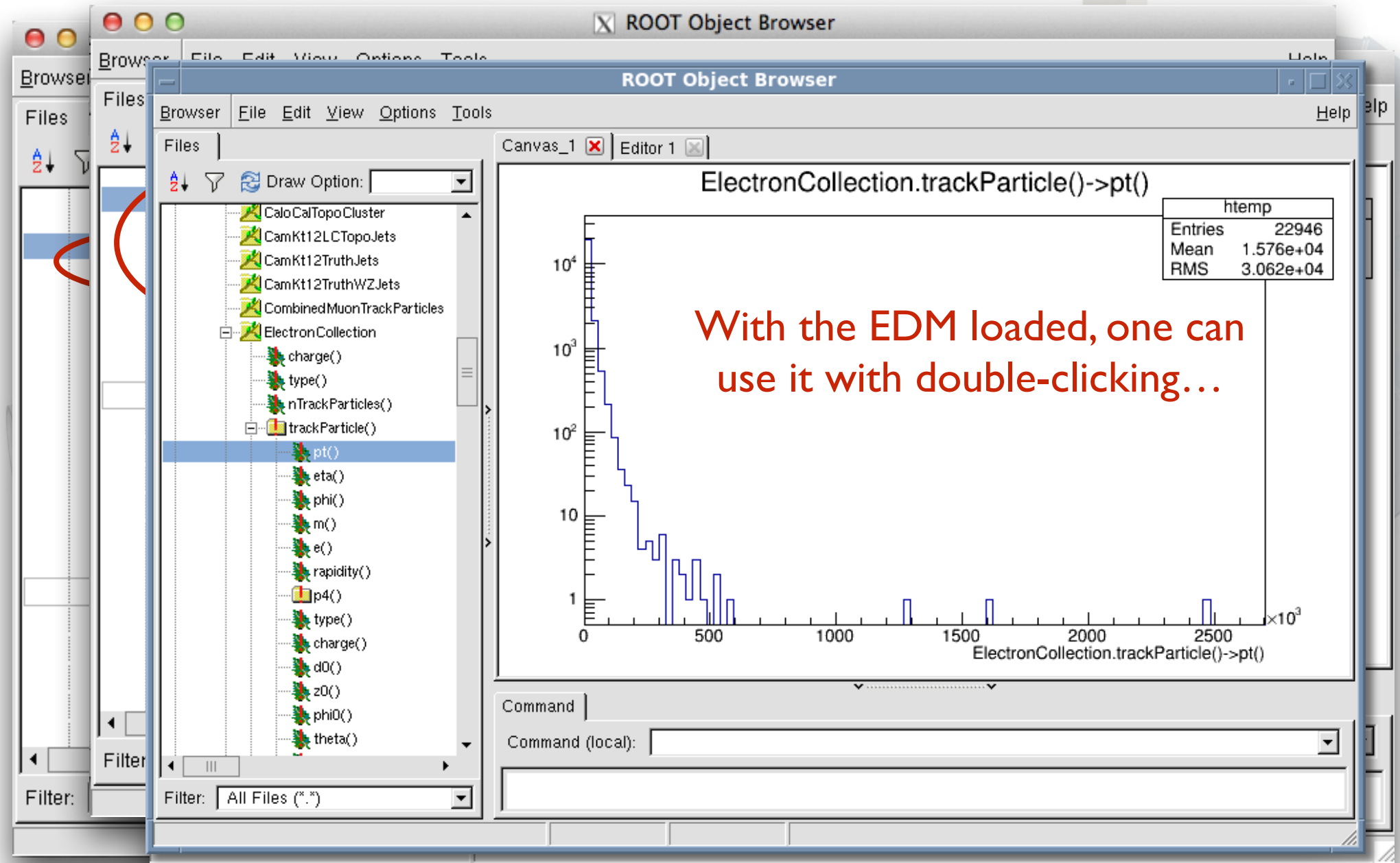
The xAOD Files (I)



The xAOD Files (1)



The xAOD Files (1)



The xAOD Files (2)

- Because the data is stored in such a simple format, you can make some basic plots in interactive ROOT without loading any additional code:

```
root [1] CollectionTree->Draw( “MuonsAux.pt” );
root [2] CollectionTree->Draw( “MuonsAuxDyn.ptcone20” );
```

- But when you load the xAOD EDM code into interactive ROOT, you become able to do even plots like:

```
root [1] CollectionTree->Draw( “Muons.pt()” );
root [2] CollectionTree->Draw( “ElectronCollection.pt() -
                                ElectronCollection.trackParticle().pt()” );
```