FCC General Software Meeting

EDM4hep.jl
Analysing EDM4hep files with Julia

Pere Mato / CERN
26 February 2024

https://github.com/peremato/EDM4hep.jl

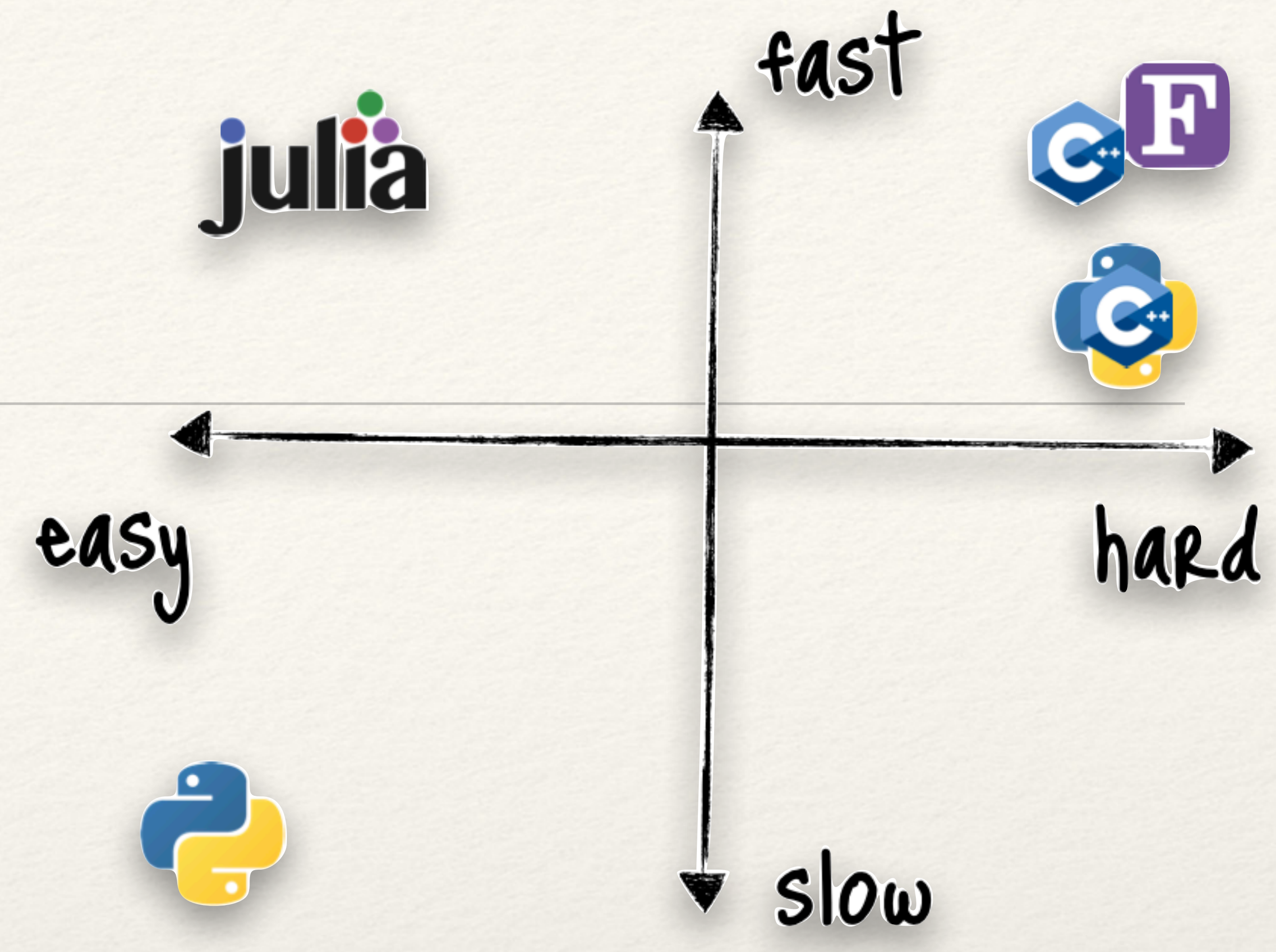
Why a new programming language?

- HEP needs a solution to the **Two Language Problem**
  - **C++** is fast but complex (and every day becoming more complex)
  - **Python** is nice and easy but very slow (mitigated if you avoid loops)
- The community has developed ways to deal with these two languages but we pay a price
  - Interoperability is not always smooth (e.g. garbage collection side effects)
  - Awkward constructions (e.g. the C++ strings in the PyRDF)

Why Julia?

- The Julia language was launched in 2012 (v1.0 in 2018) - New, but not immature!
- Modern imperative language, multi-paradigm with reflection and object orientation
- Robust **built-in tooling** (learning from earlier languages)
  - Outstanding integrated package manager and build system
  - Module system with excellent code reuse
  - Modern tooling, with built in debuggers and profilers
  - Interactive - REPL and full notebook support (it's the "Ju" in Jupyter)
- Julia has been built from the ground up to be **very fast**
  - JIT compilation via LLVM to native machine code
  - Performance is comparable to C and C++ (as a baseline, see microbenchmarks)

But, is Julia interesting for HEP?

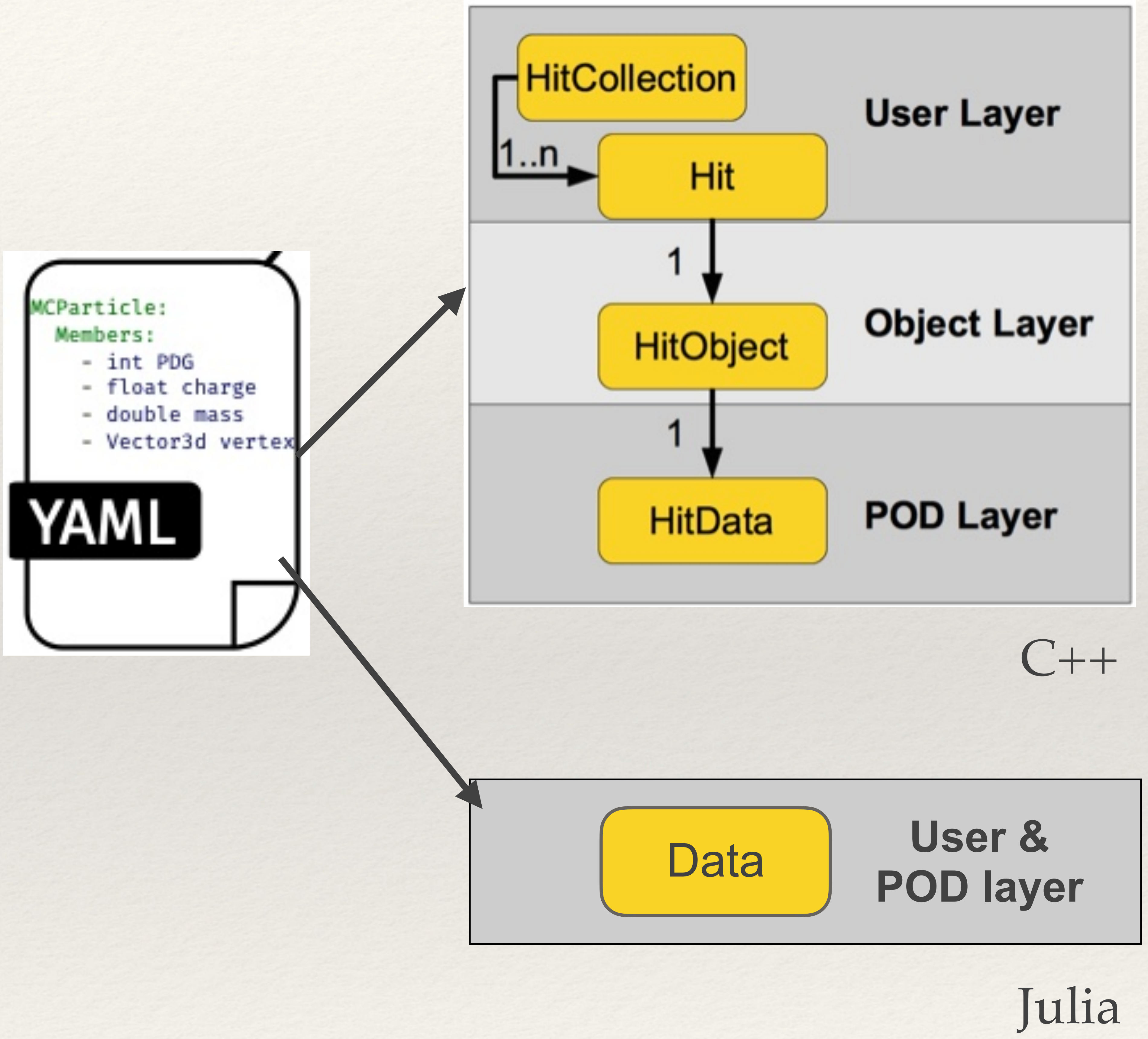


- There exist many languages in the world
  - Each has different strengths and weaknesses
- **We think the answer is yes!**
  - Julia is specifically designed for **numerical programming for science and engineering***
    - So we are the target audience and the support for our use case is strong
  - Julia is much **easier to program in than C++**
    - Experience shows that students with Python experience can be productive in Julia very quickly
  - Code written in Julia is fast, often close to peak performance
    - The first prototype can evolve naturally into the production code
  - This overcomes the two language problem that we have today
  - Wrappers allow integration with existing code in C++ and Python - vital for our existing codes

*Julia used a lot in ASML, Boeing, Pfizer among others

EDM4hep - Introduction

- Based on the PODIO edm-toolkit
  - use **yaml-files** to define EDM objects then generate C++ code via Python/Jinja scripts
  - three layers of classes (in C++)
    - POD layer - the actual data in array of structs
    - Object layer - add relations and vector members
    - User layer - thin handles and collections
- Default I/O backend: **ROOT**



Motivation for EDM4hep.jl

- Generate Julia ‘friendly’ structures for the EDM4hep data model
- Be able to read event data files (in ROOT format) written by C++ programs from Julia (using the UnROOT.jl package)
- Later, be able to also to write RNTuple files from Julia

Implementing EDM4hep in Julia is a pre-requisite for introducing the Julia language in Simulation and Reconstruction workflows

Main Design Features

- All entities are **immutable structs** for better performance, SoA, GPUs, etc.
  - POD with basic types and structs, including the relationships (one-to-one and one-to-many)
  - Objects attributes cannot be changed, new instances must be created (Accessors.jl)
- Constructors have **keyword arguments** with reasonable default values
- New objects are by default not registered, they are "**free floating**". Explicit registration or setting relationships will register them to containers.
- Note that operations like **register**, **setting relationships** will automatically create a new instances. The typical pattern is to overwrite the user variable with the new instance, e.g.:

```
p1 = MCParticle(...)
p1, d1 = add_daugther(p1, MCParticle(...))
```

- Reading EDM4hep containers from ROOT should result in **StructArrays**
  - Very efficient access by column and the same time provide convenient views as object instances

StructArrays.jl

- Package that provides tools for working with structs of arrays efficiently (SoA)
- Efficient Storage
  - Struct arrays store elements contiguously in memory, improving cache performance
- Type Stability
  - Maintains type stability even when working with arrays of structs
- Vectorized Operations
  - Enables vectorized operations on arrays of structs, similar to operations on standard arrays
- Compatibility
  - Seamlessly integrates with other Julia packages and tools

```julia
using StructArrays

# Define a custom struct
struct Point
    x::Float64
    y::Float64
end

# Create a struct array
points = StructArray([Point(1.0, 2.0),
                      Point(3.0, 4.0),
                      Point(5.0, 6.0)])

# Access elements
println(points[1])  # Output: Point(1.0, 2.0)
```

PODIO Generation

- Written small Julia script to generate Julia structs from YAML file
    - Added a **`ObjectID`** to each object to control its registration state
    - Relations implemented with **`ObjectID`** and **`Relation`** structs with just indices (isbits())
- Two files: **`genComponents.jl`**, **`genDatatypes.jl`** generated that can be complemented with utility methods

```julia
"""
struct MCParticle

    Description: The Monte Carlo particle - based on the lcio::MCParticle.
    Author: F.Gaede, DESY
"""
struct MCParticle <: POD
    index::ObjectID{MCParticle}      # ObjectID of itself

    #---Data Members
    PDG::Int32                       # PDG code of the particle
    generatorStatus::Int32           # status of the particle as defined by the ...
    simulatorStatus::Int32           # status of the particle from the simulation ...
    charge::Float32                  # particle charge
    time::Float32                    # creation time of the particle in [ns] wrt. ...
    mass::Float64                    # mass of the particle in [GeV]
    vertex::Vector3d                 # production vertex of the particle in [mm].
    endpoint::Vector3d               # endpoint of the particle in [mm]
    momentum::Vector3f               # particle 3-momentum at the production vertex..
    momentumAtEndpoint::Vector3f     # particle 3-momentum at the endpoint in [GeV]
    spin::Vector3f                   # spin (helicity) vector of the particle.
    colorFlow::Vector2i              # color flow as defined by the generator

    #---OneToManyRelations
    parents::Relation{MCParticle,1}  # The parents of this particle.
    daughters::Relation{MCParticle,2}  # The daughters this particle.
end
```

```julia
"""
struct SimTrackerHit

    Description: Simulated tracker hit
    Author: F.Gaede, DESY
"""
struct SimTrackerHit <: POD
    index::ObjectID{SimTrackerHit}   # ObjectID of itself
    #---Data Members
    cellID::UInt64                   # ID of the sensor that created this hit
    EDep::Float32                    # energy deposited in the hit [GeV].
    time::Float32                    # proper time of the hit in the lab frame in ...
    pathLength::Float32              # path length of the particle in the sensiti ...
    quality::Int32                   # quality bit flag.
    position::Vector3d               # the hit position in [mm].
    momentum::Vector3f               # the 3-momentum of the particle at the hits ...
    #---OneToOneRelations
    mcparticle_idx::ObjectID{MCParticle}  # MCParticle that caused the hit.
end
```

Building the Event Model in Memory

```
#---MCParticles----------------------------------------------------------------------------
p1 = MCParticle(PDG=2212, mass=0.938, momentum=(0.0, 0.0, 7000.0), generatorStatus=3)
p2 = MCParticle(PDG=2212, mass=0.938, momentum=(0.0, 0.0, -7000.0), generatorStatus=3)

p3 = MCParticle(PDG=1, mass=0.0, momentum=(0.750, -1.569, 32.191), generatorStatus=3)
p3, p1 = add_parent(p3, p1)

p4 = MCParticle(PDG=-2, mass=0.0, momentum=(-3.047, -19.000, -54.629), generatorStatus=3)
p4, p2 = add_parent(p4, p2)

p5 = MCParticle(PDG=-24, mass=80.799, momentum=(1.517, -20.68, -20.605), generatorStatus=3)
p5, p1 = add_parent(p5, p1)
p5, p2 = add_parent(p5, p2)

p6 = MCParticle(PDG=22, mass=0.0, momentum=(-3.813, 0.113, -1.833), generatorStatus=1)
p6, p1 = add_parent(p6, p1)
p6, p2 = add_parent(p6, p2)

p7 = MCParticle(PDG=1, mass=0.0, momentum=(-2.445, 28.816, 6.082), generatorStatus=1)
p7, p5 = add_parent(p7, p5)

p8 = MCParticle(PDG=-2, mass=0.0, momentum=(3.962, -49.498, -26.687), generatorStatus=1)
p8, p5 = add_parent(p8, p5)

#---Simulation tracking hits-----------------------------------------------------------
for j in 1:5
  sth1 = SimTrackerHit(cellID=0xabadcaffee, EDep=j*0.000001, position=(j * 10., j * 20., j * 5.), mcparticle=p7)
  sth1 = register(sth1)
  sth2 = SimTrackerHit(cellID=0xcaffeebabe, EDep=j*0.001, position=(-j * 10., -j * 20., -j * 5.), mcparticle=p8)
  sth2 = register(sth2)
end
```

Relationships and Vector members

- **`ObjectID{ED}`** - implementing 1-to-1
  - Acts as a reference to object of type ED in the EDStore
  - back and forth conversions

```
struct ObjectID{ED<:POD} <: POD
    index::Int32
    collectionID::UInt32
end

Base.convert(::Type{ED}, i::ObjectID{ED}) where ED
```

- **`Relation{ED}`** - implementing 1-to-N
  - Represents a variable size vector (realised as 3 UInt32)

```
struct Relation{ED<:POD,TD<:POD,N}
    first::UInt32    # first index (starts with 0)
    last::UInt32     # last index (starts with 0)
    collid::UInt32   # Collection ID of the data
end

Base.iterate(r::Relation{ED,TD,N}, i=1) where{ED,TD,N}
```

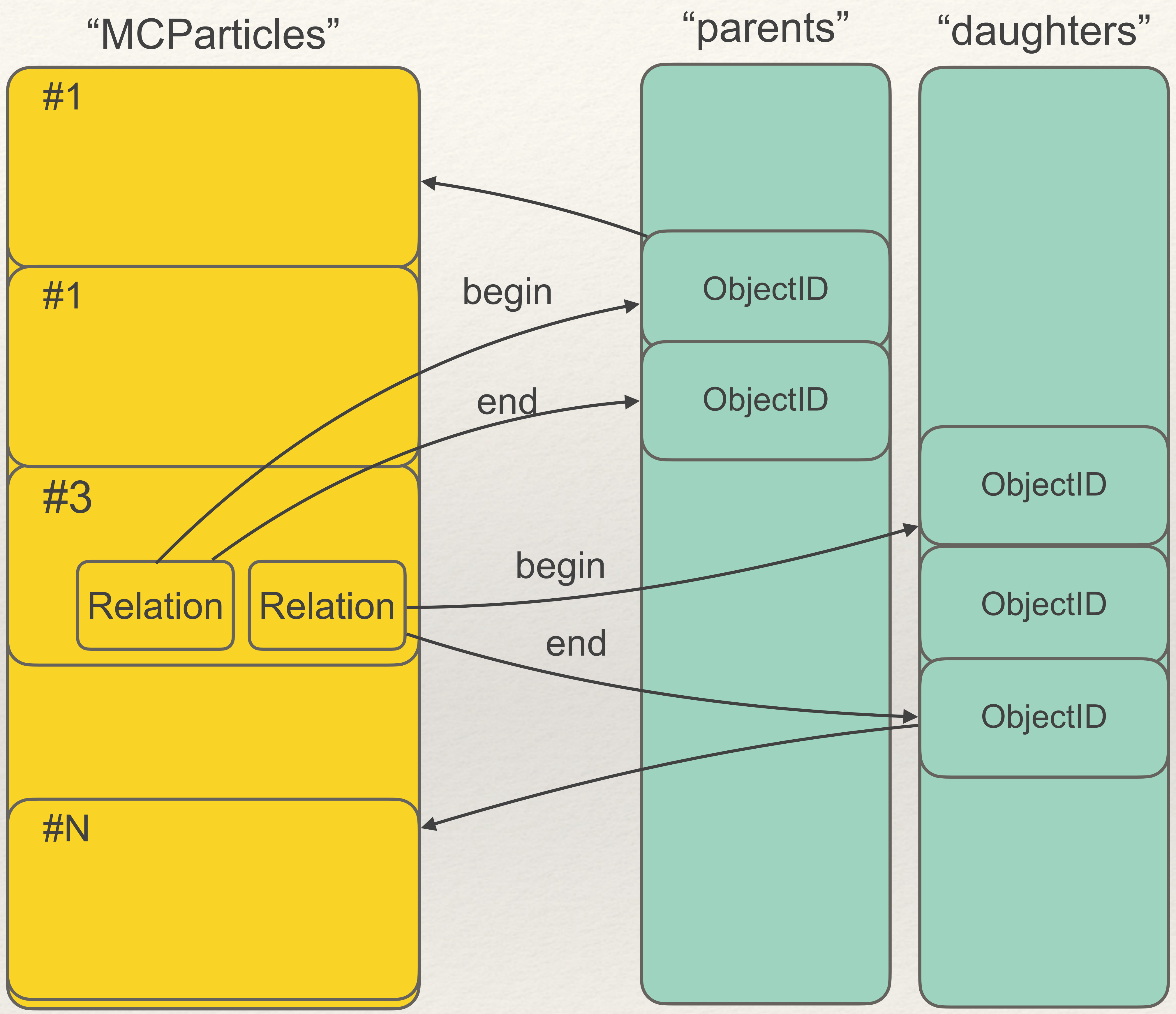
- **`PVector{T}`** - vector member
  - POD-like vector of type T
  - AbstractVector interface

```
struct PVector{ED<:POD,T, N} <: AbstractVector{T}
    first::UInt32    # first index (starts with 0)
    last::UInt32     # last index (starts with 0)
    collid::UInt32   # Collection ID of the data
end

Base.iterate(v::Relation{ED,TD,N}, i=1) where {ED,T,N}
```

Layout in Memory

- EDM objects are created free-floating
  - They are registered in containers explicitly or implicitly when setting relationships
- To keep track of the containers the struct **`EDStore{ED}`** has been introduced
  - Provided methods to control its lifetime (init!(), empty!(), etc.)



It looks complicated, but in reality is completely transparent to the User

Navigating Relationships

```
for p in getEDStore(MCParticle).objects
    println("MCParticle $(p.index) with PDG=$(p.PDG) and momentum $(p.momentum) has $(length(p.daughters)) daughters")
    for d in p.daughters
        println("   ---> $(d.index) with PDG=$(d.PDG) and momentum $(d.momentum)")
    end
end

for s in getEDStore(SimTrackerHit).objects
    println("SimTrackerHit in cellID=$(string(s.cellID, base=16)) with EDep=$(s.EDep) and position=$(s.position)
             associated to particle $(s.mcparticle.index)")
end
```

```
MCParticle #1 with PDG=1 and momentum (0.75,-1.569,32.191) has 0 daughters
MCParticle #2 with PDG=2212 and momentum (0.0,0.0,7000.0) has 3 daughters
   ---> #1 with PDG=1 and momentum (0.75,-1.569,32.191)
   ---> #5 with PDG=-24 and momentum (1.517,-20.68,-20.605)
   ---> #6 with PDG=22 and momentum (-3.813,0.113,-1.833)
MCParticle #3 with PDG=-2 and momentum (-3.047,-19.0,-54.629) has 0 daughters
MCParticle #4 with PDG=2212 and momentum (0.0,0.0,-7000.0) has 3 daughters
   ---> #3 with PDG=-2 and momentum (-3.047,-19.0,-54.629)
   ---> #5 with PDG=-24 and momentum (1.517,-20.68,-20.605)
   ---> #6 with PDG=22 and momentum (-3.813,0.113,-1.833)
MCParticle #5 with PDG=-24 and momentum (1.517,-20.68,-20.605) has 2 daughters
   ---> #7 with PDG=1 and momentum (-2.445,28.816,6.082)
   ---> #8 with PDG=-2 and momentum (3.962,-49.498,-26.687)
MCParticle #6 with PDG=22 and momentum (-3.813,0.113,-1.833) has 0 daughters
MCParticle #7 with PDG=1 and momentum (-2.445,28.816,6.082) has 0 daughters
MCParticle #8 with PDG=-2 and momentum (3.962,-49.498,-26.687) has 0 daughters
SimTrackerHit in cellID=abadcaffee with EDep=1.0e-6 and position=(10.0,20.0,5.0) associated to particle #7
SimTrackerHit in cellID=caffeebabe with EDep=0.001 and position=(-10.0,-20.0,-5.0) associated to particle #8
SimTrackerHit in cellID=abadcaffee with EDep=2.0e-6 and position=(20.0,40.0,10.0) associated to particle #7
SimTrackerHit in cellID=caffeebabe with EDep=0.002 and position=(-20.0,-40.0,-10.0) associated to particle #8
SimTrackerHit in cellID=abadcaffee with EDep=3.0e-6 and position=(30.0,60.0,15.0) associated to particle #7
SimTrackerHit in cellID=caffeebabe with EDep=0.003 and position=(-30.0,-60.0,-15.0) associated to particle #8
SimTrackerHit in cellID=abadcaffee with EDep=4.0e-6 and position=(40.0,80.0,20.0) associated to particle #7
SimTrackerHit in cellID=caffeebabe with EDep=0.004 and position=(-40.0,-80.0,-20.0) associated to particle #8
SimTrackerHit in cellID=abadcaffee with EDep=5.0e-6 and position=(50.0,100.0,25.0) associated to particle #7
SimTrackerHit in cellID=caffeebabe with EDep=0.005 and position=(-50.0,-100.0,-25.0) associated to particle #8
```

Integrated in the Julia ecosystem

- Simple structs (isbits) and vectors of them integrate well with the rest of the Julia ecosystem. Examples:
  - A container of EDM4hep datatypes can be converted to a **DataFrame** immediately
  - Very useful for GPU array programming

```
using DataFrames
df = DataFrame(getEDStore(MCParticle).objects)
```

8×15 DataFrame

Row	index ObjectID…	PDG Int32	generatorStatus Int32	simulatorStatus Int32	charge Float32	time Float32	mass Float64	vertex Vector3d	endpoint Vector3d	momentum Vector3f	momentumAtEndpoint Vector3f	spin Vector3f	colorFlow Vector2i	parents Relation…	daughters Relation…	…
1	#1	1	3	0	0.0	0.0	0.0	(0.0,0.0,0.0)	(0.0,0.0,0.0)	(0.75,-1.569,32.191)	(0.0,0.0,0.0)	(0.0,0.0,0.0)	(0,0)	MCParticle#[2]	MCParticle#[]	…
2	#2	2212	3	0	0.0	0.0	0.938	(0.0,0.0,0.0)	(0.0,0.0,0.0)	(0.0,0.0,7000.0)	(0.0,0.0,0.0)	(0.0,0.0,0.0)	(0,0)	MCParticle#[]	MCParticle#[1, 5,	
3	#3	-2	3	0	0.0	0.0	0.0	(0.0,0.0,0.0)	(0.0,0.0,0.0)	(-3.047,-19.0,-54.629)	(0.0,0.0,0.0)	(0.0,0.0,0.0)	(0,0)	MCParticle#[4]	MCParticle#[]	
4	#4	2212	3	0	0.0	0.0	0.938	(0.0,0.0,0.0)	(0.0,0.0,0.0)	(0.0,0.0,-7000.0)	(0.0,0.0,0.0)	(0.0,0.0,0.0)	(0,0)	MCParticle#[]	MCParticle#[3, 5,	
5	#5	-24	3	0	0.0	0.0	80.799	(0.0,0.0,0.0)	(0.0,0.0,0.0)	(1.517,-20.68,-20.605)	(0.0,0.0,0.0)	(0.0,0.0,0.0)	(0,0)	MCParticle#[2, 4]	MCParticle#[7, 8]	…
6	#6	22	1	0	0.0	0.0	0.0	(0.0,0.0,0.0)	(0.0,0.0,0.0)	(-3.813,0.113,-1.833)	(0.0,0.0,0.0)	(0.0,0.0,0.0)	(0,0)	MCParticle#[2, 4]	MCParticle#[]	
7	#7	1	1	0	0.0	0.0	0.0	(0.0,0.0,0.0)	(0.0,0.0,0.0)	(-2.445,28.816,6.082)	(0.0,0.0,0.0)	(0.0,0.0,0.0)	(0,0)	MCParticle#[5]	MCParticle#[]	
8	#8	-2	1	0	0.0	0.0	0.0	(0.0,0.0,0.0)	(0.0,0.0,0.0)	(3.962,-49.498,-26.687)	(0.0,0.0,0.0)	(0.0,0.0,0.0)	(0,0)	MCParticle#[5]	MCParticle#[]	

1 column omitted

ROOT I/O

- Using **UnROOT.jl** package - a really great package!
- Supports (transparently) TTree and RNTuple formats and several versions of PODIO storage
  - data files consist exclusively of ‘collections-of-datatypes’ (e.g. ReconstructedParticles, Vertices, etc.)
- The goal is to obtain a **`StructArray{DataType}`** of each collection for each event
  - The exercise consists in mapping the schema in the file (using ROOT streamer info) to the actual Julia datatype (using the Julia introspection)

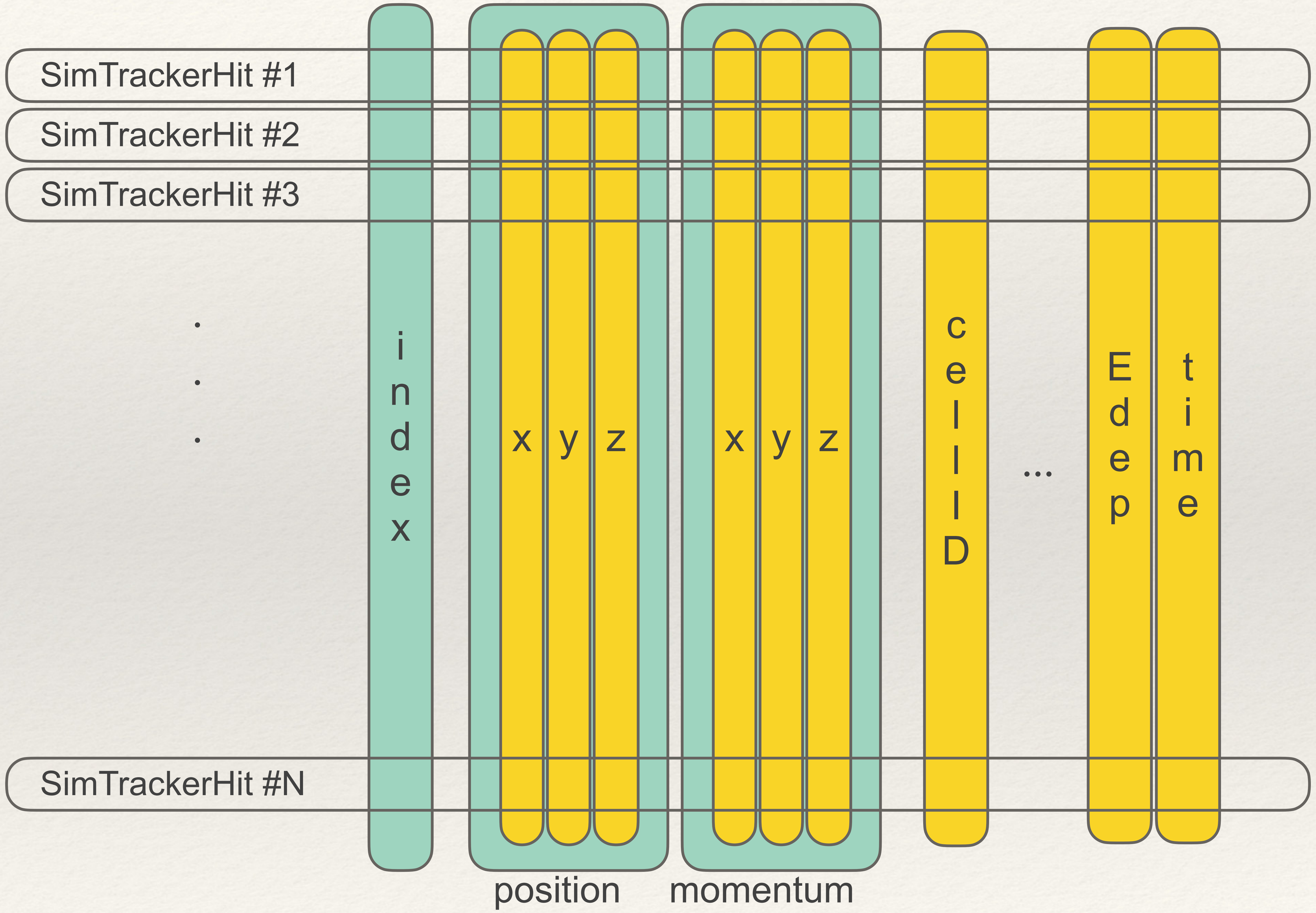
Creating SoAs from EDM4hep types

- UnROOT.jl provides the leaves arrays (in a lazy manner) and they are “mapped” to form SoA of a DataType
- Opens the possibility of schema evolution
  - filling empty attributes, type change, re-shaping, etc.

```
using StructArrays

# Create a struct array
hits = StructArray{SimTrackerHit}(Tuple(<TLeaf>...))


# Access elements
println(hits[1])  # Output: SimTrackerHit(....)
```



SoA provides a very Ergonomic interface

- Storage in memory consists of a set of column arrays
  - very fast access by column
- Materialize, when requested, object instances (usually on the stack) to be able to call user object methods
  - to achieve a user friendly access

```
julia> typeof(mcps[1])
MCParticle

julia> typeof(mcps.charge)
SubArray{Float32, 1, Vector{Float32},
Tuple{UnitRange{Int64}}, true}

julia> length(mcps.charge)
211

julia> mcps[1:2].momentum
2-element StructArray(::Vector{Float32}, ::Vector{Float32},
::Vector{Float32}) with eltype Vector3f:
 (0.5000167,0.0,50.0)
 (0.5000167,0.0,-50.0)

julia> sum(mcps[1:2].momentum)
(1.0000334,0.0,0.0)
```

Reading from a ROOT (TTree) File

```
using EDM4hep
using EDM4hep.RootIO

cd(@__DIR__)

f = "ttbar_edm4hep_digi.root"

reader = RootIO.Reader(f)
events = RootIO.get(reader, "events")

evt = events[1];

hits = RootIO.get(reader, evt, "InnerTrackerBarrelCollection")
mcps = RootIO.get(reader, evt, "MCParticle")

for hit in hits
    println("Hit $(hit.index) is related to MCParticle $(hit.mcparticle.index) with name $(hit.mcparticle.name)")
end

#---Loop over events-------------------------------------------------------------------
for (n,e) in enumerate(events)
    ps =  RootIO.get(reader, e, "MCParticle")
    println("Event #$(n) has $(length(ps)) MCParticles with a charge sum of $(sum(ps.charge))")
end
```

```
Hit #1 is related to MCParticle #65 with name pi+
Hit #2 is related to MCParticle #65 with name pi+
Hit #3 is related to MCParticle #65 with name pi+
Hit #4 is related to MCParticle #65 with name pi+
Hit #5 is related to MCParticle #66 with name pi-
Hit #6 is related to MCParticle #66 with name pi-
Hit #7 is related to MCParticle #66 with name pi-
Hit #8 is related to MCParticle #49 with name pi+
Hit #9 is related to MCParticle #49 with name pi+
Hit #10 is related to MCParticle #49 with name pi+
Hit #11 is related to MCParticle #27 with name K-
Hit #12 is related to MCParticle #27 with name K-
Hit #13 is related to MCParticle #27 with name K-
Hit #14 is related to MCParticle #95 with name e-
Hit #15 is related to MCParticle #95 with name e-
...
```

~ 1500 times faster than Python

Example Analysis (FCCee)

- Created a more complete example of a FCCee analysis (higgs/mH-recoil/mumu)
- These are the steps:
  - 1. Installation and setup. No need to install anything (except for Julia itself :-))
  - 2. Load the necessary modules (all registered!)

```julia
using EDM4hep
using EDM4hep.RootIO
using EDM4hep.SystemOfUnits
using EDM4hep.Histograms
```

Example - Creating Analysis Functions

- 3. Creating analysis functions using EDM4hep types and reusing convenient existing Julia packages (e.g. LorentzVectorHEP, Combinatorics)
  - It shows the power of software re-use of Julia

Use the EDM4hep high-level objects directly

```julia
# re-using convenient existing packages
using LorentzVectorHEP
using Combinatorics

function resonanceBuilder(rmass::AbstractFloat, legs::AbstractVector{ReconstructedParticle})
    result = ReconstructedParticle[]
    length(legs) < 2 && return result
    for (a,b) in combinations(legs, 2)
        lv = LorentzVector(a.energy, a.momentum...) + LorentzVector(b.energy, b.momentum...)
        rcharge = a.charge + b.charge
        push!(result, ReconstructedParticle(mass=mass(lv), momentum=(lv.x, lv.y, lv.z), charge=rcharge))
    end
    sort!(result, lt = (a,b) -> abs(rmass-a.mass) < abs(rmass-b.mass))
    return result[1:1]  # take the best one
end;
```

Use Julia algorithms

Example - Define Histograms

- 4. Define a custom structure with the wanted histograms
- 5. And a function to plot them

```
using Parameters
using Plots
@with_kw struct Histograms
    mz          = H1D("m_{Z} [GeV]",125,0,250, unit=:GeV)
    mz_zoom     = H1D("m_{Z} [GeV]",40,80,100, unit=:GeV)
    lr_m        = H1D("Z leptonic recoil [GeV]", 100, 0, 200, unit=:GeV)
    lr_m_zoom   = H1D("Z leptonic recoil [GeV]", 200, 80, 160, unit=:GeV)
    ...
    lr_m_zoom4  = H1D("Z leptonic recoil [GeV]", 800, 120, 140, unit=:GeV)
    lr_m_zoom5  = H1D("Z leptonic recoil [GeV]", 2000, 120, 140, unit=:GeV)
    lr_m_zoom6  = H1D("Z leptonic recoil [GeV]", 100, 130.3, 132.5, unit=:GeV)
end
function do_plot(histos::Histograms)
    img = plot(layout=(5,2), show=true, size=(1000,1500))
    for (i,fn) in enumerate(fieldnames(Histograms))
        h = getfield(histos, fn)
        plot!(subplot=i, h.hist, title=h.title, show=true, cgrad=:plasma)
    end
    return img
end
myhists = Histograms()
```

Added a thin-layer on top of FHist histograms

Example - Open data file

- 6. Using a file from the winter2023 production in EOS
  - ROOT file with a **`TTree`** called "events" with 100k events and 262 branches/leaves
  - PODIO version "0.16.2" (old layout of collections and relations)

```
f = "root://eospublic.cern.ch//eos/experiment/fcc/ee/generation/DelphesEvents/winter2023/IDEA/p8_ee_ZZ_ecm240/events_000189367.root"

reader = RootIO.Reader(f);
events = RootIO.get(reader, "events");
```

Example - The Event Loop

```
for evt in events
    #---get the collection of ReconstructedParticles and Muons
    recps = RootIO.get(reader, evt, "ReconstructedParticles");
    muons = RootIO.get(reader, evt, "Muon#0"; btype=ObjectID{ReconstructedParticle})

    sel_muons = filter(x -> pₜ(x) > 10GeV, muons)
    zed_leptonic = resonanceBuilder(91GeV, sel_muons)
    zed_leptonic_recoil = recoilBuilder(240GeV, zed_leptonic)

    if length(zed_leptonic) == 1    #  Filter to have exactly one Z candidate
        Zcand_m        = zed_leptonic[1].mass
        Zcand_recoil_m = zed_leptonic_recoil[1].mass
        Zcand_q        = zed_leptonic[1].charge
        if 80GeV <= Zcand_m <= 100GeV
            #---Fill histograms now--------------------------------------
            push!(myhists.mz, Zcand_m)
            push!(myhists.mz_zoom, Zcand_m)
            push!(myhists.lr_m, Zcand_recoil_m)
            ...
            push!(myhists.lr_m_zoom6, Zcand_recoil_m)
        end
    end
end
```

Get the needed collections. The Muon#0 is a collection of ObjectIDs (need to specify the type)

Filter and create new objects

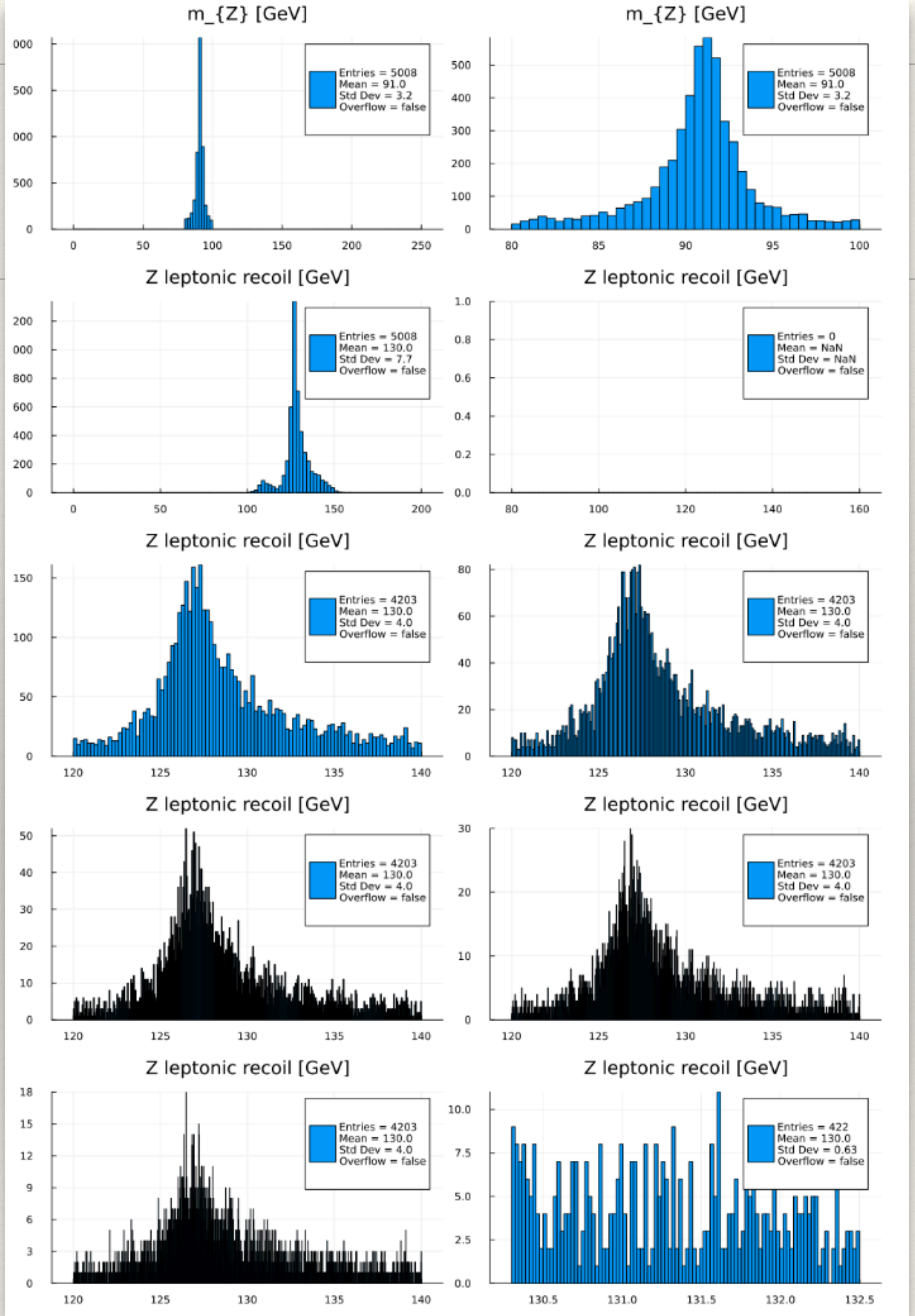
Event sections

Fill the required histograms

```
img = do_plot(myhists)
display("image/png", img)
```

Example - Results

- 8. Finally, plot the histograms
  - histograms plots not very nice
- What about the performance?
  - in this example we can process ~8200 events/s (on lcgapp-centos7-physical)
  - somehow a bit slower than FCCAnalyses framework (Python+C++) ~9500 events/s
  - further optimisation makes only sense with RNTuple



What's Next?

- Validation of RNTuple with RC2
- Optimisation
- Multi-threading support
- Multi-file support

Conclusions

- Demonstrated that Data Analysis can be done using ‘high-level objects’ instead of resigning yourself to use ‘flat n-tuples’
  - And in a single and consistent programming language :-)
  - Imagine Open Data analysis: very powerful with minimal required infrastructure
- The performance is not bad, but probably can be improved a bit further
- Missing quite a lot of HEP utilities
  - e.g. utility functions, fitting, ergonomic and good looking histograms, etc.
  - We could start building them from now
- Package EDM4hep.jl registered and ready to be used!