Data Models and Data Structures for Data-intensive Computing

Paolo Calafiura LBNL
ESC09, Bertinoro

BERKELEY LAB

Office of Science
U.S. DEPARTMENT OF ENERGY

Overview

Data Models:

- Event-centric

Data Structures:

- The Role of Containers
  - STL, polymorphism, and memory layout

Persistency:

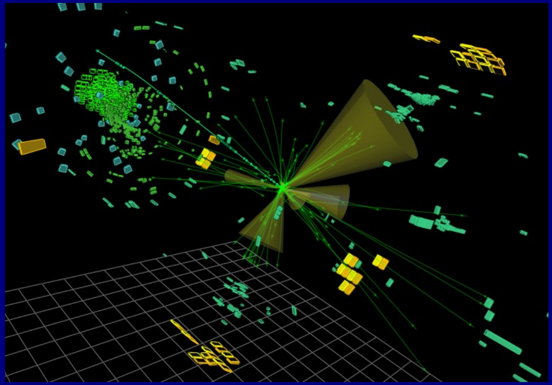
- Constraints on Data Model Design
  - Object relationships
  - Schema evolution, T/P separation

Personal Biases

- C++/Linux/gcc
- HEP Computing, ATLAS, Gaudi, ROOT
- Transient/Persistent Separation
- Performance-oriented designs
- KISS rule

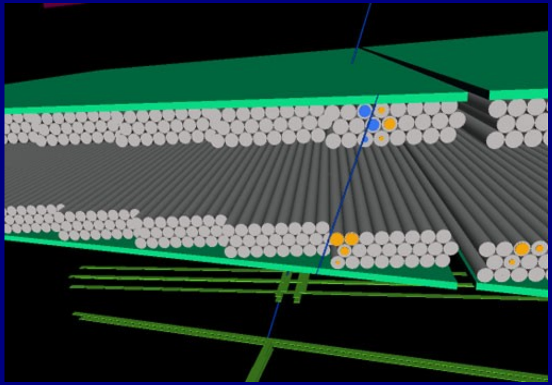
Data Models

- Event-centric
  - Collect data from all channels for a given trigger



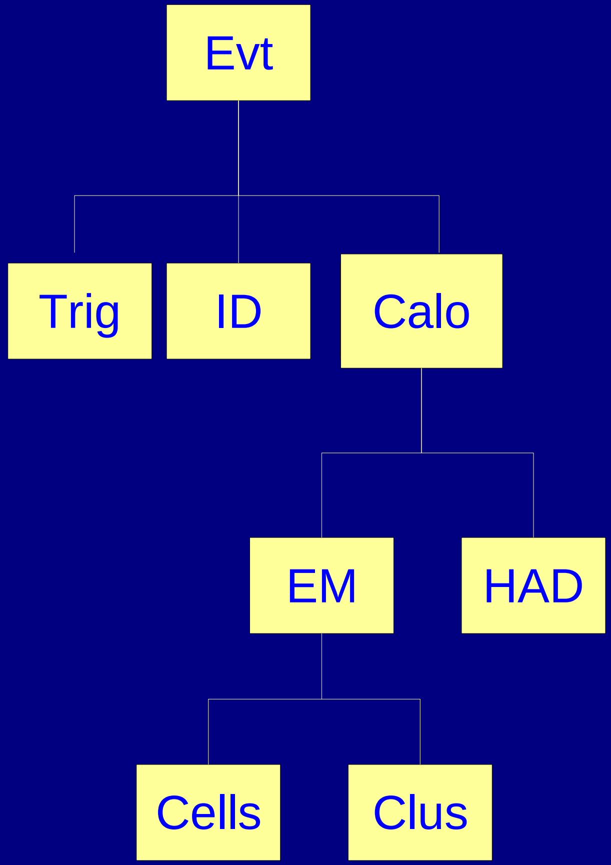
  - Reconstruction, Analysis

- Detector-centric
  - Collect data from all triggers for a given channel



  - Monitoring, Calibration

Event-centric Data Model



Typical HENP event is a tree (or table) of Primary Data Objects (PDO)

- Usually the EDM is static (all events contain the same PDOs)

Primary Data Objects

- PDOs are direct-accessible using the event structure API. In ATLAS StoreGate

```
McClusterCollection *pClusters;
eventStore()->retrieve(pClusters, “G4Clusters”);
```

or, in ROOT

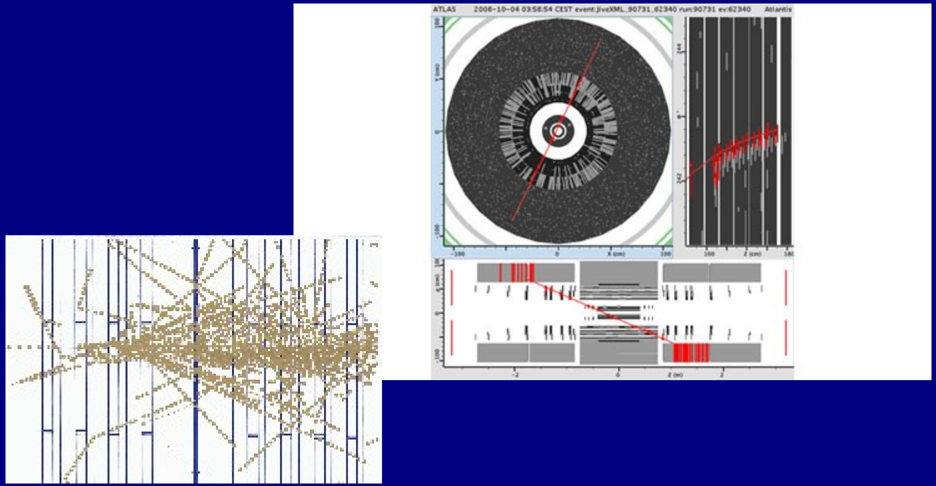
```
McClusterCollection clusters;
pEvTree->SetBranchAddress(“G4Clusters”,&clusters);
```

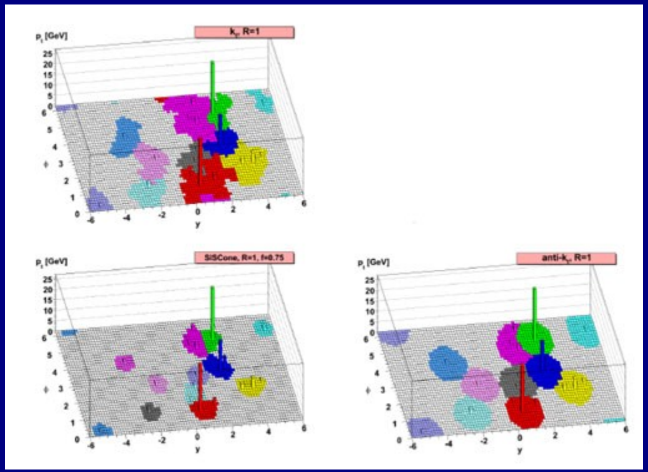
Secondary Data Objects

- Most PDOs are collections
  - We call their elements SDOs
    - only accessible navigating to the parent PDO and using its API
  - Persistable references among SDOs challenging to implement particularly when elements are accessed via an interface

Data Producers and Consumers

- PDOs can look very different to producers (adding data to the event) and consumers (retrieving it)
- Example Jet Reco using Tracks/Clusters even Calorimeter cells as “particles”





PDO Containers: implementation examples

- std:: containers, vector, string and map
- Ad-hoc containers
  - ROOT TClonesArray
  - Athena DataVector
  - Gaudi VectorMap (sorted vector)

(ROOT TClonesArray and Athena DataVector: manage SDO memory)

STL Containers

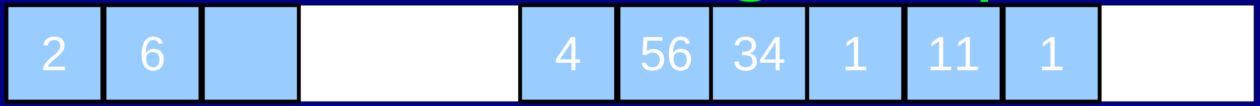
- Powerful, easy to use
- Too easy to use

  map<int,Track>

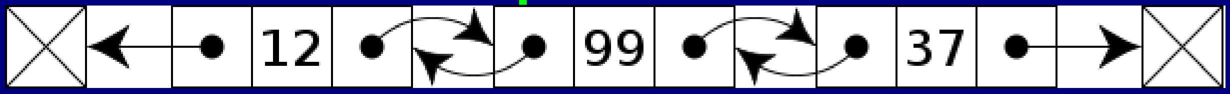
  - Do you know its memory layout?
  - What happens when you insert a new element?
- Used appropriately solid foundation for any C++ data model

Container Memory Layout

- Array-based (vector, string, deque)



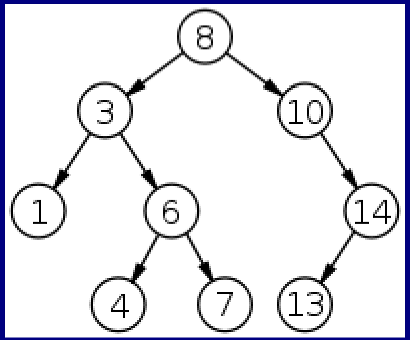
  - Most efficient memory-wise (contiguous chunk allocations)
  - Easy to access from C, python, java etc
- Node-based (list, map,…)



  - Fragmented memory
  - Fast insertion/erasures

Stick to Containers of Basic Types

```
map<int, LArHit> hm;
hm.insert(make_pair(6,aHit));
```



- aHit is copied upon insertion
- May be copied many more times to rebalance the tree on later insertions
- Vectors are even worse (think about sorting)
- CPU efficiency aside, there is a problem of correctness since so many classes have broken copy constructors

PDO as Containers: Requirements

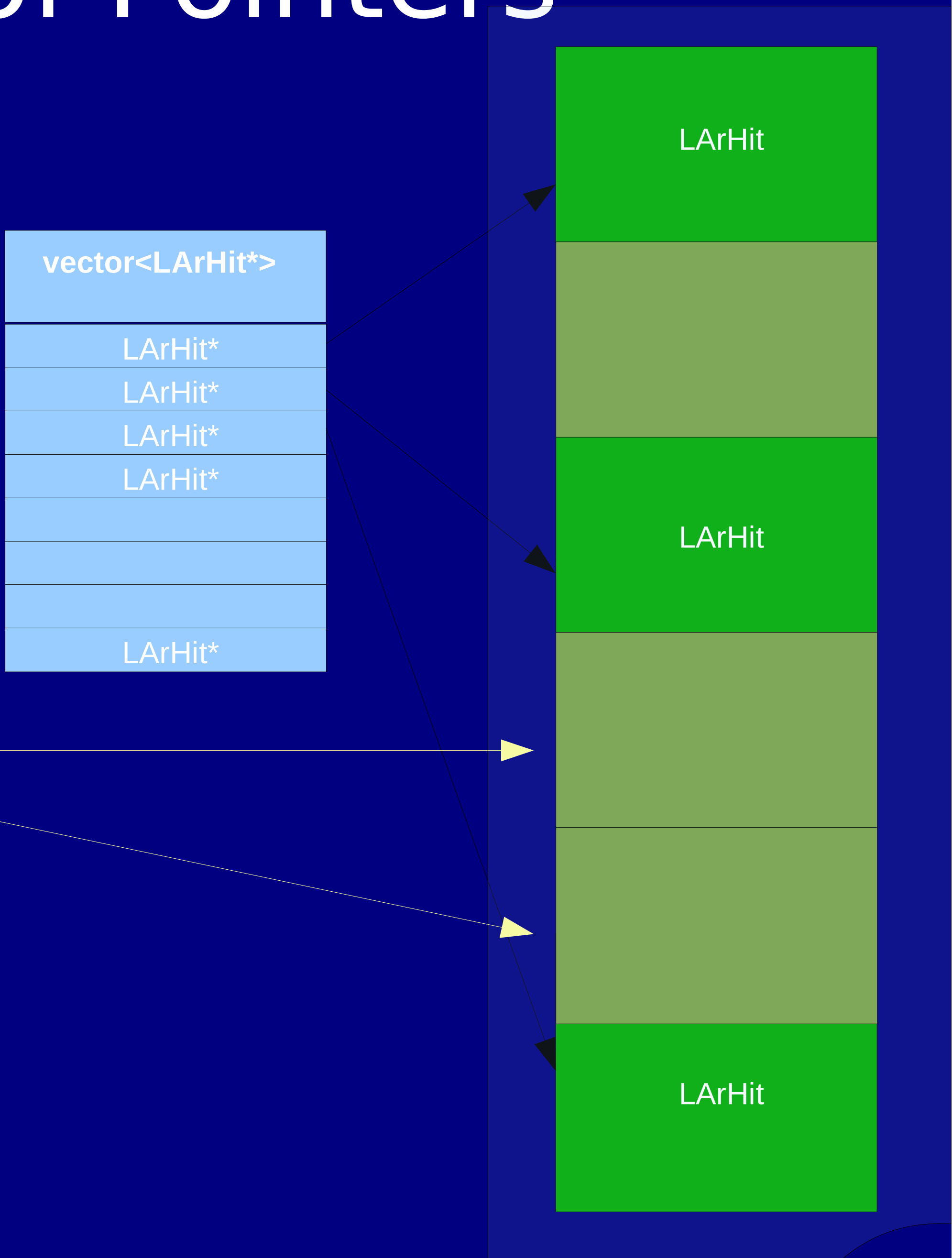
- Variable size, possibly empty
- Direct access to container elements (SDOs)
- Polymorphic
  - SDOs of various types, share an interface
- Manage SDOs memory
- Persistable

Containers of Pointers

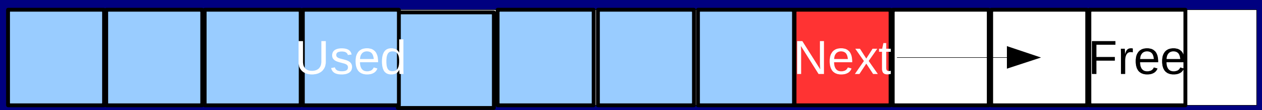
- For class or struct elements use containers of pointers:

```
vector<LArHit*> hv;
```

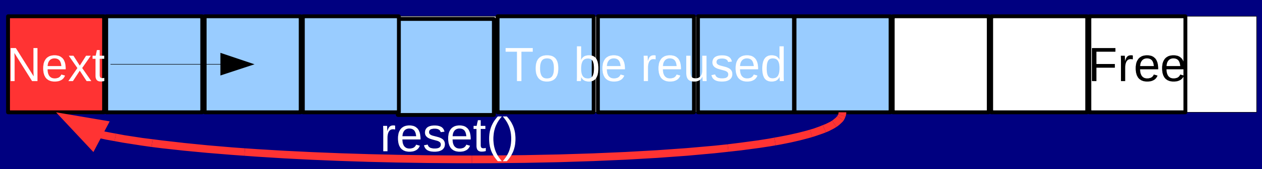
- Beware of
  - Memory holes (next slide + Lassi lectures)
  - Element ownership
  - Persistency



Memory Pools



- Basically an array of reusable objects
  - You decide how many to preallocate and when to start reusing them (@ EndEvent)



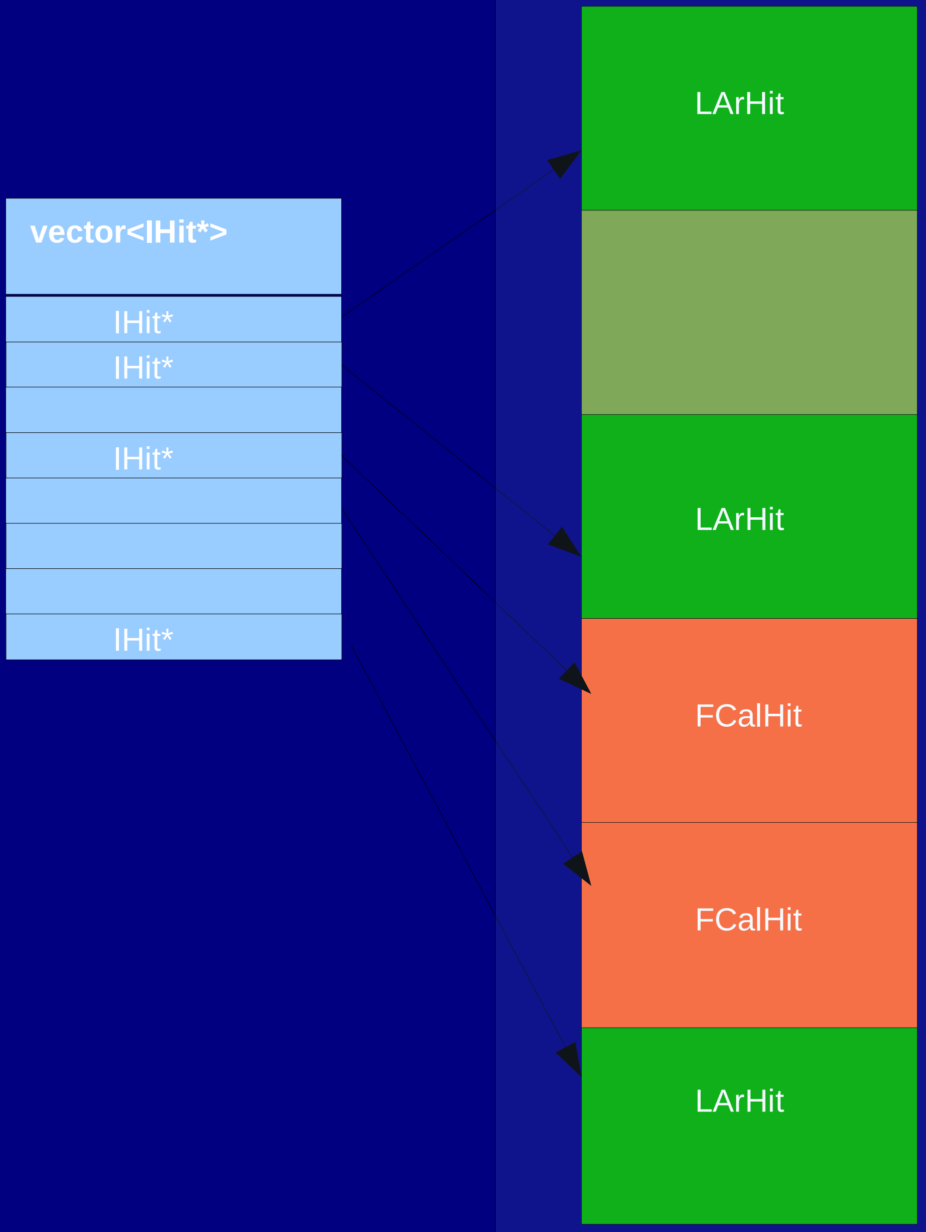

```
boost::object_pool<LArHit> hitP(10000);
LarHit* pHit= new(hitP.malloc()) LArHit(x,y,z);
...
hitP.purge_memory();
```

Polymorphic Containers

- Containers of pointers to interface class

```
vector<IHit*> hv;
```

- Main tool to address producer/consumer dichotomy



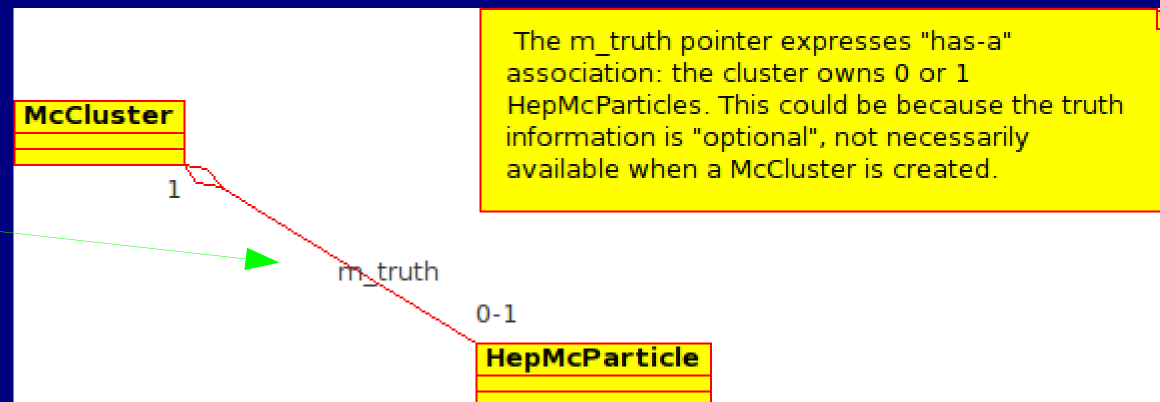
Pointer Quiz

```
Class McCluster {
     ...
   private:
     HepMcParticle* m_truth;
     vector<IHit*> m_hits;
};
```

Who owns m_truth and and the hits in m_hits?

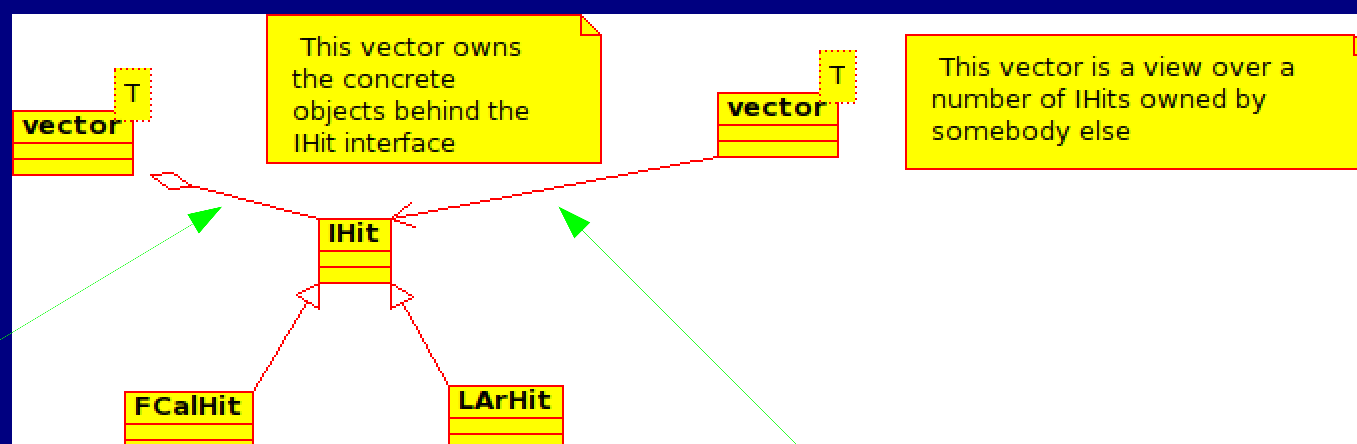
Pointer Roles

Optional Data



Polymorphic Containers:

Aggregation



Association (references)

Disambiguating Pointers: Expressing (Shared) Ownership

- boost::shared_ptr<T>
  - a copyable, ref-counted, smart pointer that provides shared ownership of a T

```
vector<boost::shared_ptr<IHit> > m_hits;
```

    - IHits owned by m_hits (+possibly others)

- boost::scoped_ptr<T>
  - Non-copyable, single ownership of T

```
boost::scoped_ptr<HepMcParticle> > m_truth;
```

    - Defines m_truth intent (optional aggregation)

Container-based Memory Management

DataVector: a vector<T*> owning its elements

```
DataVector<IHit> > m_hits;
```

- More compact than vector<shared_ptr<T> >
  - No reference counting
  - Central control of ownership
- Persistency easier (single owner)
- Not a std::vector (duplicated functionality)

`http://twiki.cern.ch/twiki/bin/view/Atlas/DataVector`

Another Container-based Solution

- ROOT TClonesArray
  - Owning container of pointers like DataVector
  - Integrates object pool functionality
  - Extremely efficient: less allocations, less con/destructors calls
    - Special constraints on elements (need to set/reset internal state)
  - Not polymorphic: all elements must have same type and size

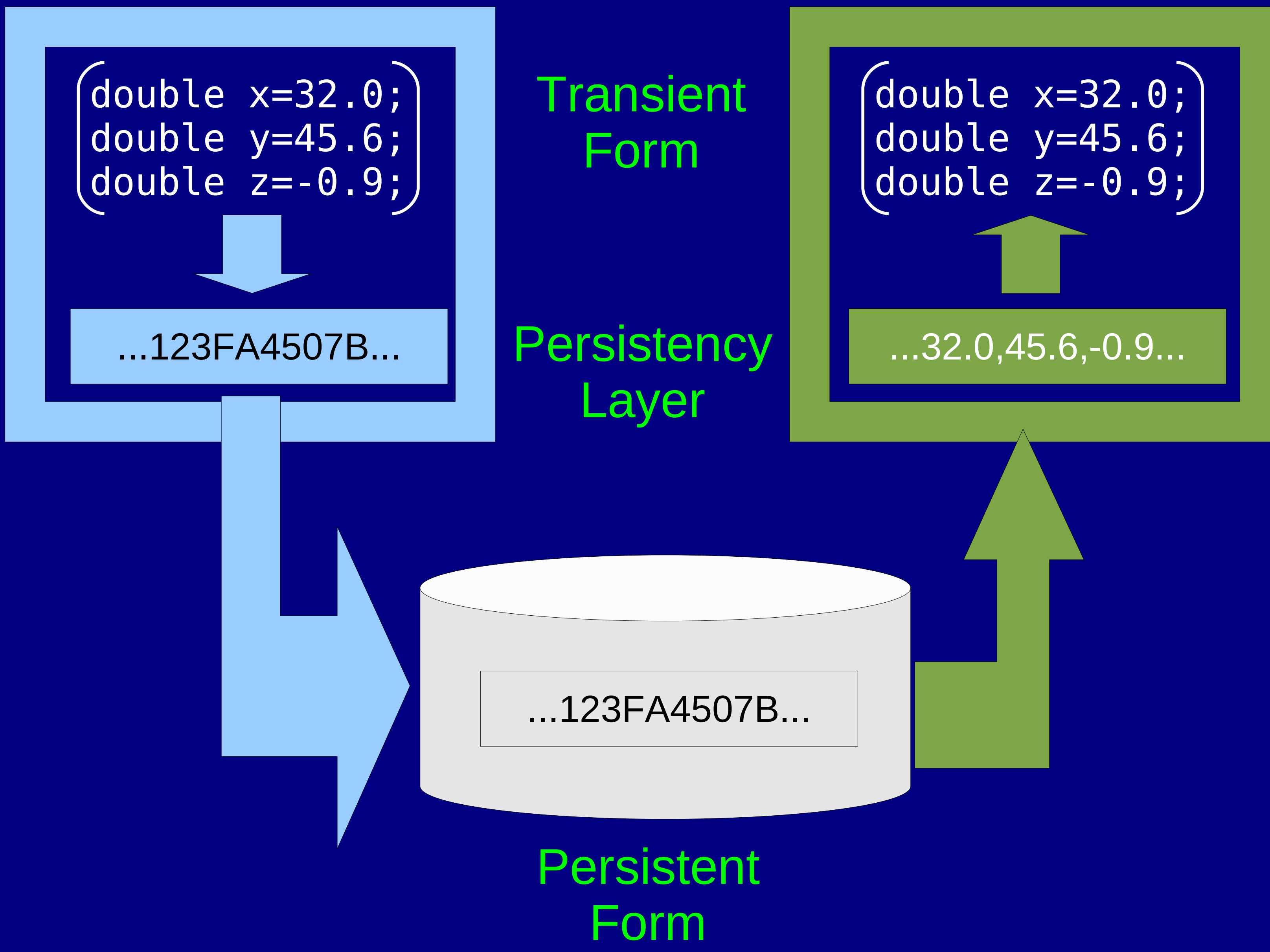
Recap: Event Data Models

- PDOs/SDOs
- Containers of pointers
- Object ownership
- STL and Custom Containers

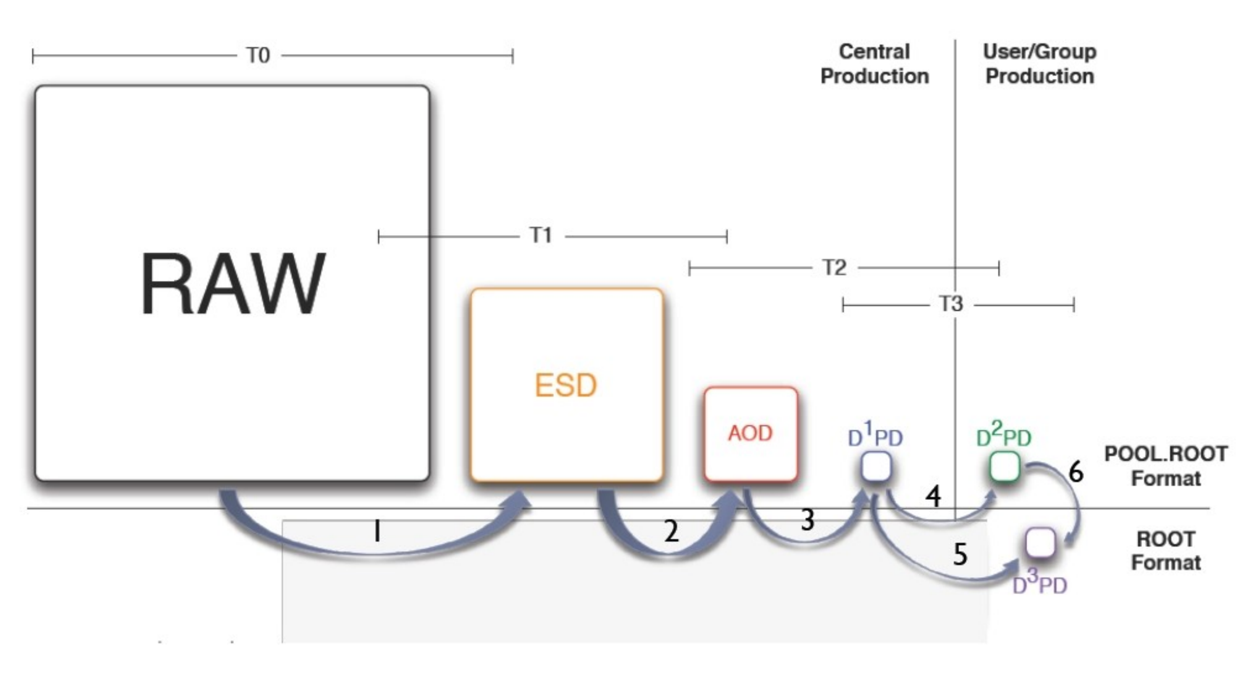
Persistency and Data Models

- Basics
- Data Streaming and Clustering
- Schema Evolution
- Persistency Mechanisms
  - Streamers, Dictionaries, T/P Separation
- Persistable References

Persistency Basics



Event Data Streams and Processing Stages

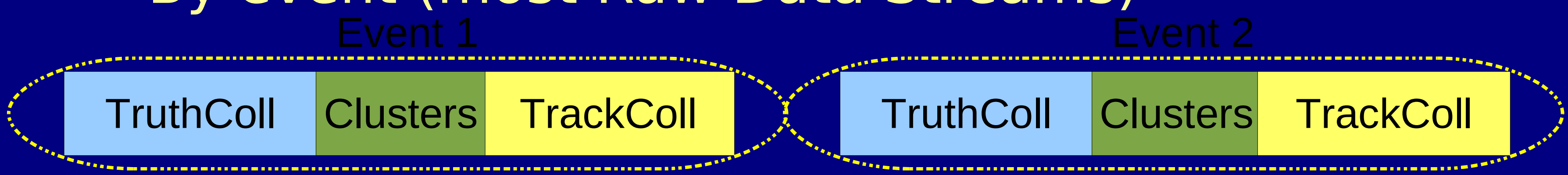


- Streaming dictated by hardware necessities
- Tension disk I/O-efficiency/usability
- Abstracting level of detail in EDM allows to use same algorithmic code at different stages

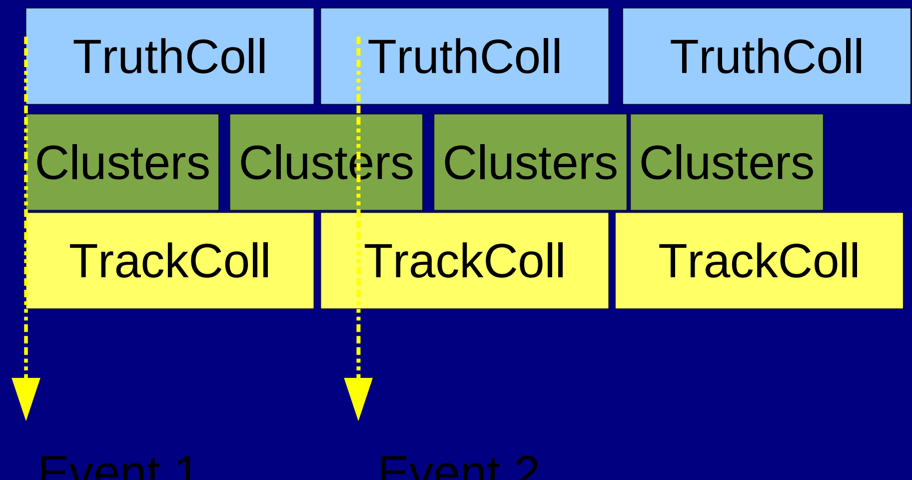
Data Clustering

How are data objects written to disk

- By event (most Raw Data Streams)



- By object, splitting events (most ROOT files)
    - Allows to read subset of event data



Schema Evolution

Fact #1: data models evolve

Fact #2: (Peta)bytes already on disk don't

Solution:

- Read old data using current Data Model
  - Easy to handle automagically for basic types
  - Harder when (pointers to) objects are involved
  - Even harder when classes are split or merged

Persistency Mechanisms

- Fundamental types (int,float,...)
  - Built-in (machine dependent!)
- Structs and Objects
  - Streamer-based (manual)
  - Dictionary-based (automatic)
  - Object-mediated (hybrid)

Our Example Class

```
class McCluster {
 public:
    McCluster(); //usually required for persistency

     ...

   private:
    double m_x;
    double m_y;
    double m_z;
    HepMcParticle* m_truth;
    vector<IHit*> m_hits;

 };

 CLASS_DEF(McCluster, 3405700781);
```

Streamer-based Persistency

A classic C++ streamer

```
streamer_t& operator <<(McCluster& o, streamer_t& s) {
     s >> o.m_x >> o.m_y >> o.m_z
       >>  ???     //m_truth
       >>  m_hits; //vector streamer loop elements

}
```

or the ROOT version

```
TObject::Streamer(TBuffer&);
```

- 1st issue: reading back, what object to build?
- How to invoke the streamer?
- The other issue are of course pointers...

Pointer Quiz #2

How to write the pointers in our MCCluster to disk, and read them back?

- How to write a HepMcCluster*?
- Can you write an IHi?t?
- How do you handle two pointers to the same IHit?
    - Don't forget you may have a LArHit* and an IHit* pointing to the same object...
- Hint:
    - Assume an object read back from disk is read-only

Reading back from Disk

Choosing the right streamer

- Assign Class Identifier (CLID) to type

  `CLASS_DEF(McCluster, 0xCAFEDEAD)`

- Register streamer with CLID
- Write CLID alongside data object

  ...CAFEDEAD123F...

Invoking streamer

- Generic wrapper for data objects (with CLID)
- Base class method (`TObject::Streamer`)
  - Simpler, but more intrusive

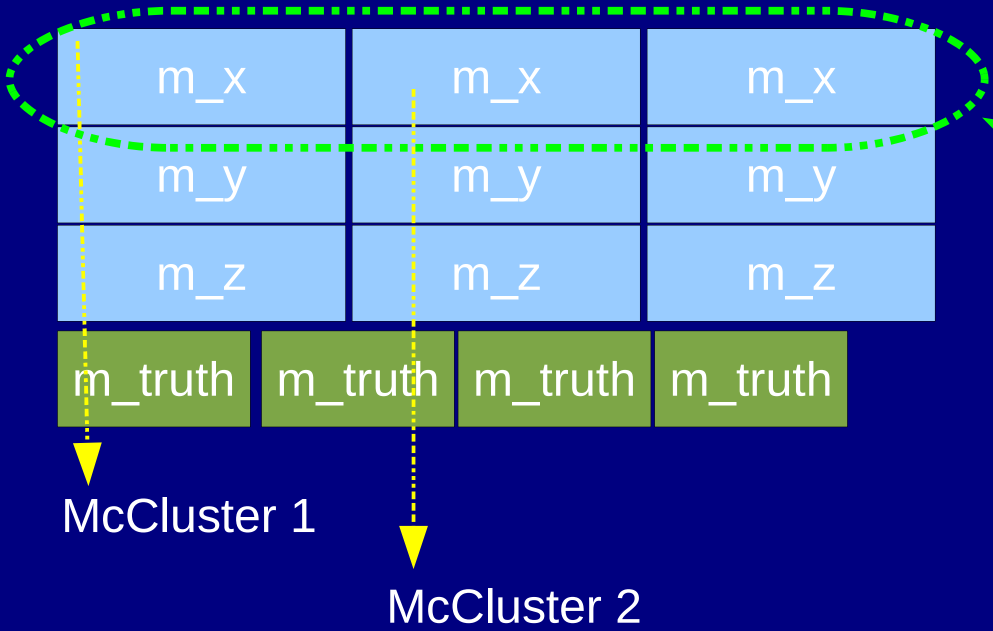
Dictionary-based Persistency

- Generate class reflection dictionary
  - Shape (data members)
  - Factory methods (default constructor req'd)
- Use dictionary to auto-generate streamers
  - Pioneered by ROOT/CINT
- Automatic persistency, but
  - Efficient persistency constrains EDM design
    - C-like simplicity, probably for the best

Data Clustering in ROOT

Full Split Mode

- Like an n-tuple

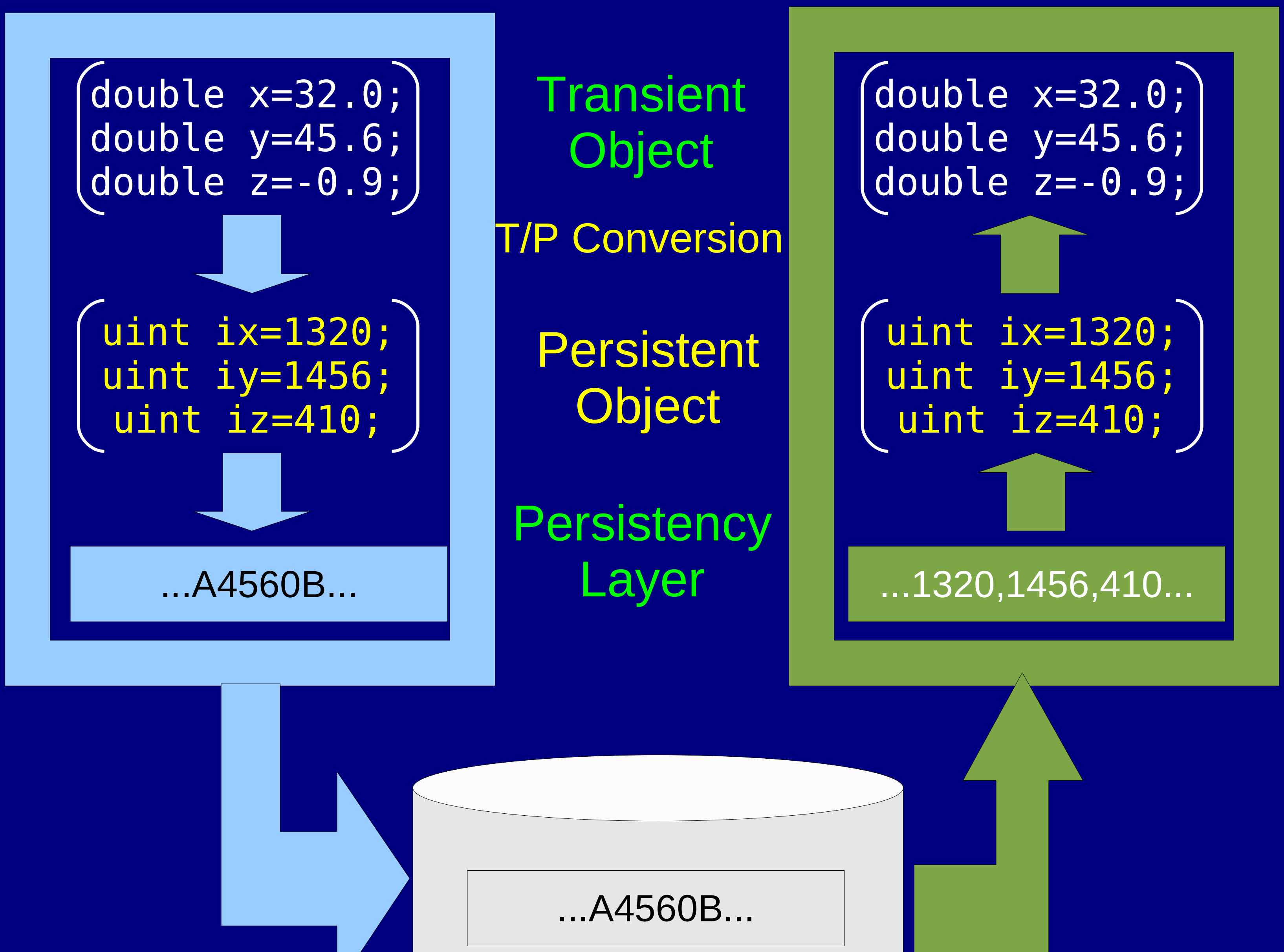


- Use dictionary to split objects and cluster data members

Enables maximal data compression
Gains size up to x2

- Allow to read subset of event data (or object data, usually bad idea)

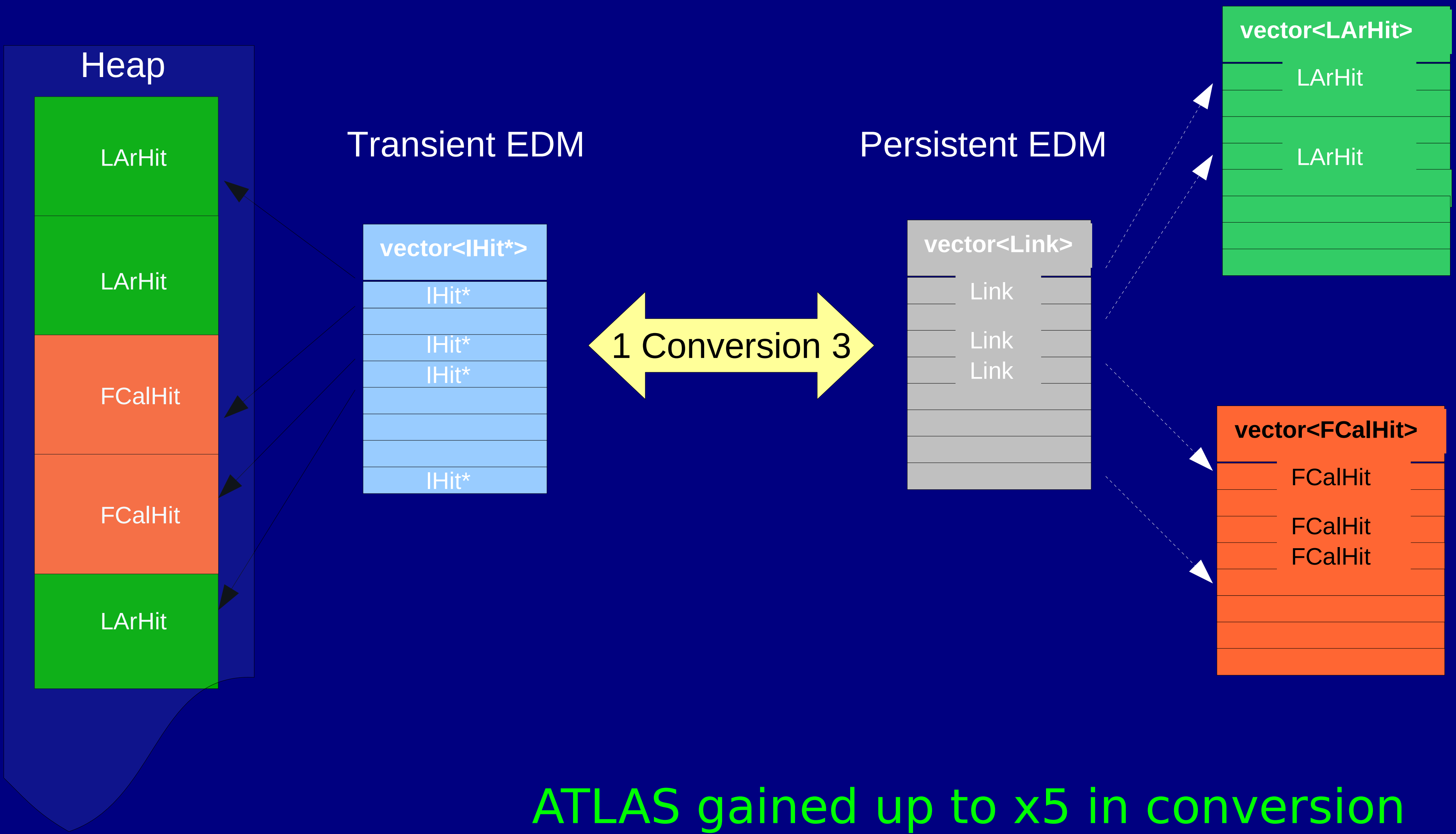
Object-mediated Conversion



Transient-Persistent Separation

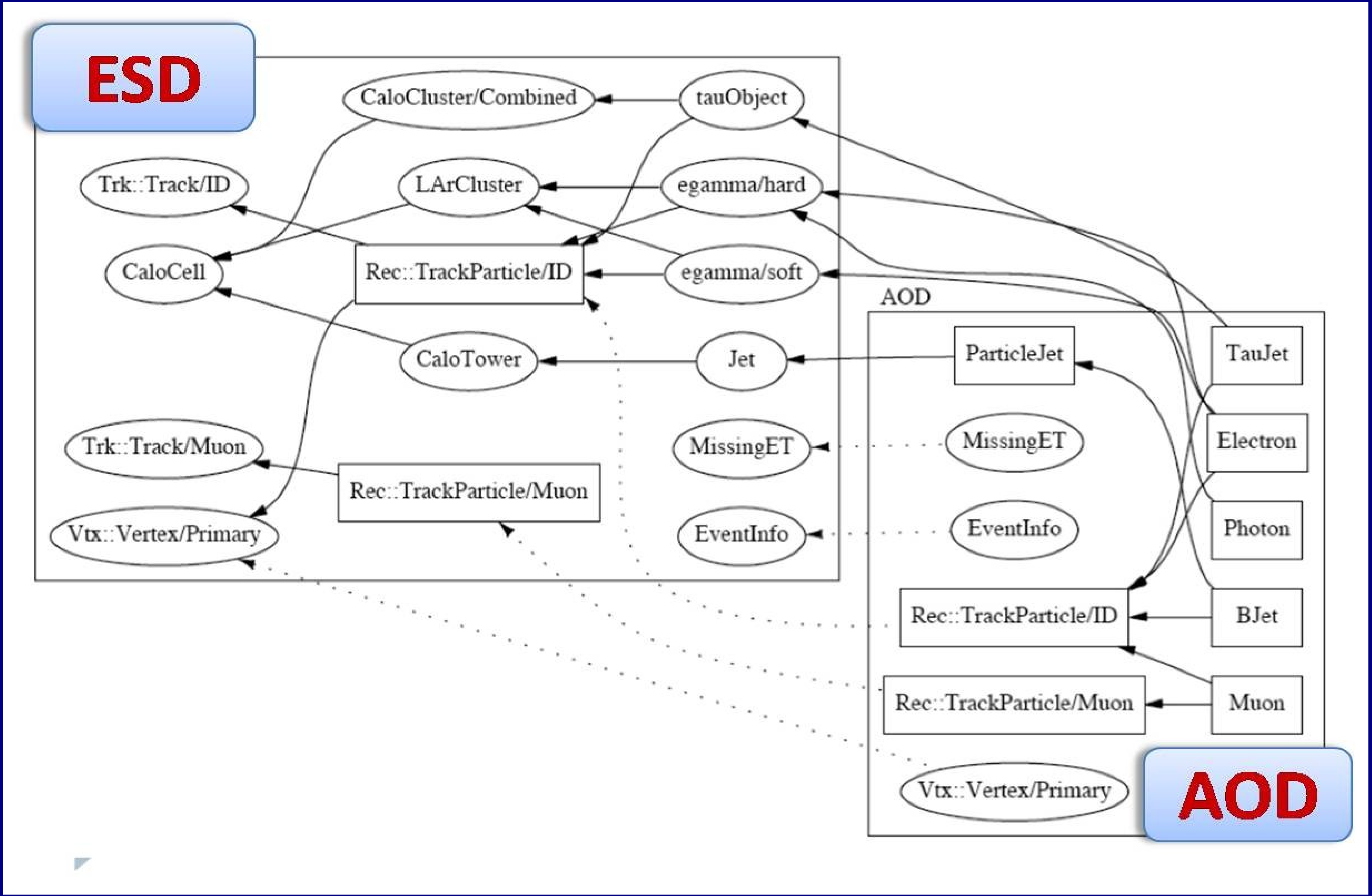
- Transient EDM, Technology-independent
  - Full power of the language
  - Free(r) to evolve
- Persistent EDM technology-optimized
  - To optimize ROOT persistency:
    - Avoid polymorphism, pointers in general
    - Avoid strings, node-based containers
    - Use basic types, and arrays thereof
- Overhead from separated T/P models and conversions between the two

Power of T/P Separation



ATLAS gained up to x5 in conversion speed using non-trivial mappings like this

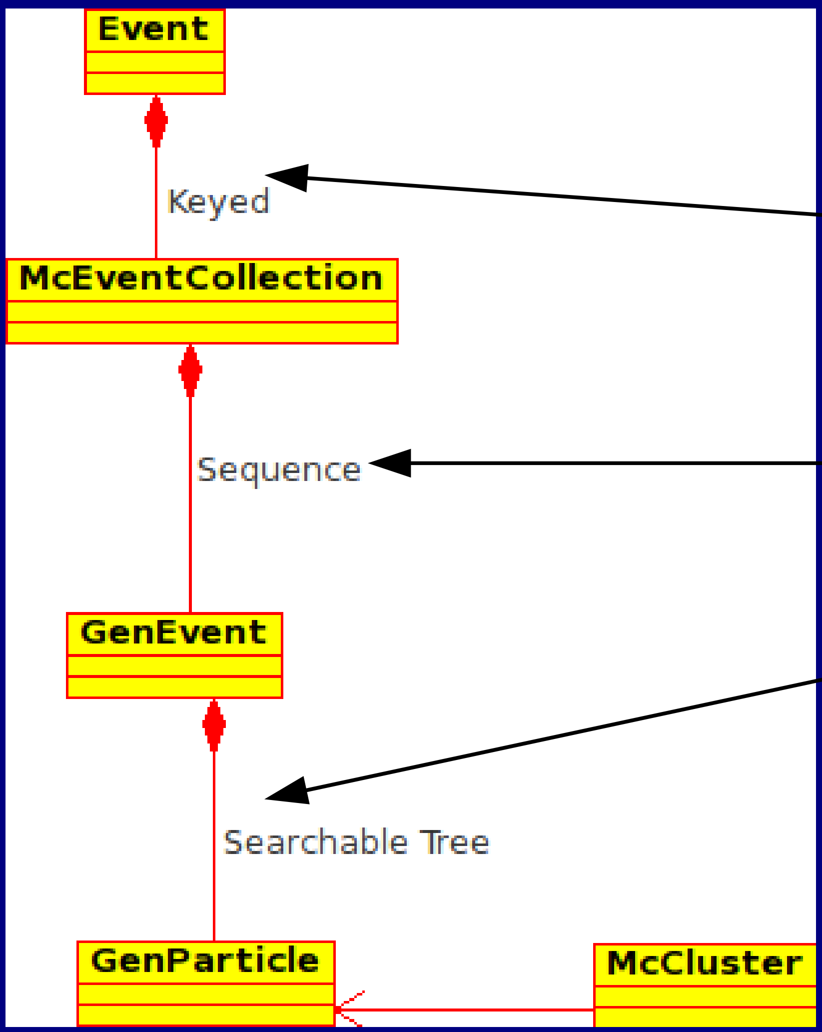
Persistable References



Persistable References

- Pointer value meaningful only within program address space
- Replace with persistent object identifier
  - ROOT TRef, POOL::Ref
- Replace with logical object identifier
  - Gaudi SmartRef, ATLAS Data/ElementLink
  - Technology (even language) independent
  - Only works for PDOs and SDOs

Logical Reference Example



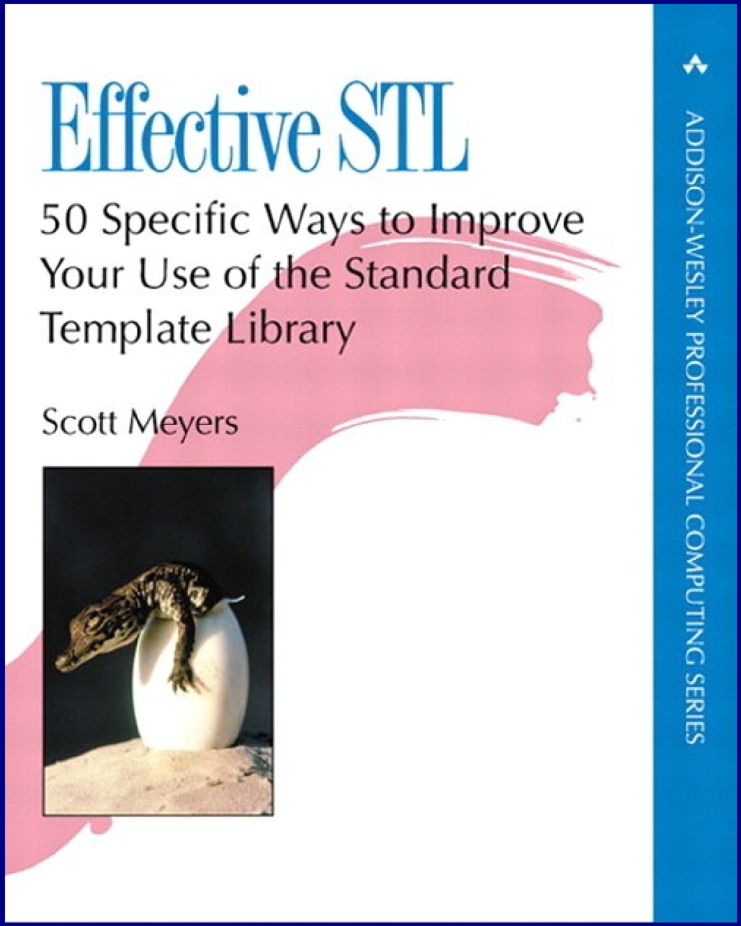
Follow link to GenParticle:

1. Get McEventCollection using its PDO ID ("key")
2. Find GenEvent using McEventCollection index
3. Search GenParticle in GenEvent using barcode

In Summary

- Event Data Models are pulled in opposite directions
  - Abstract, flexible designs: by physics code
  - Concrete, compact implementations: by persistency
- T/P separation may help satisfy both
- When in doubt follow the KISS rule

References



www.boost.org

root.cern.ch

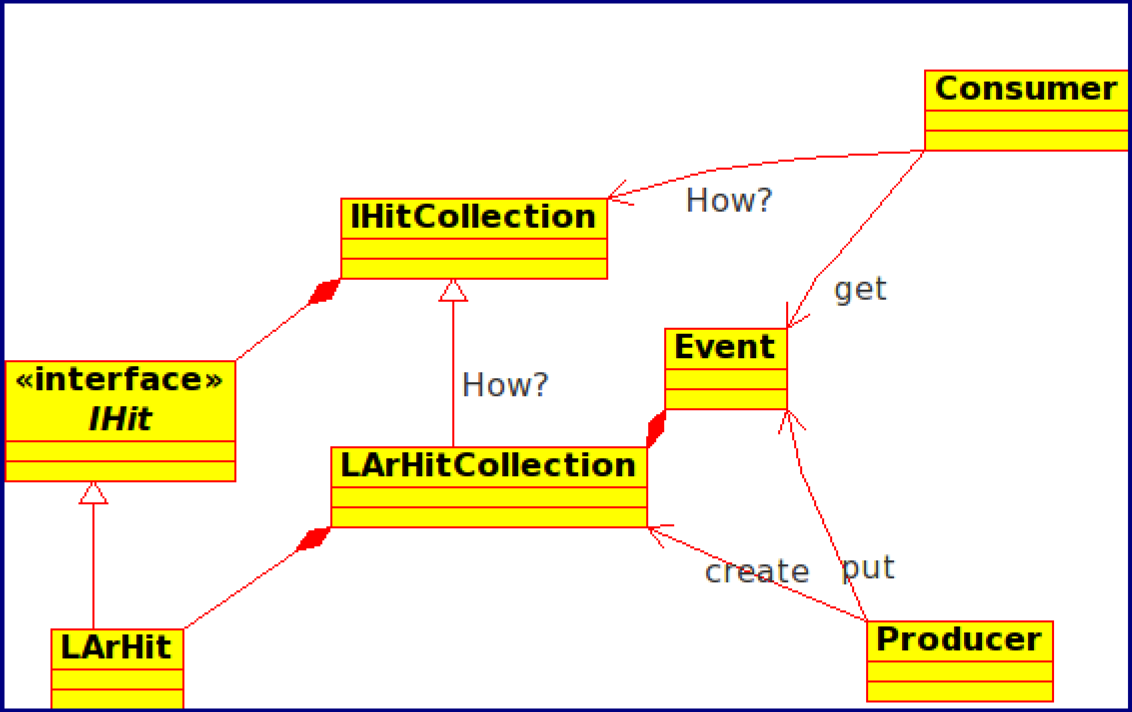
www.cern.ch/gaudi

twiki.cern.ch/twiki/bin/view/Atlas/CoreSoftware#Data_Model_Foundation_Classes

twiki.cern.ch/twiki/bin/view/Atlas/TransientPersistentSeparation

Extras

Consumer/Producer and Container “Inheritance”



- Consumer would like to get the LArHitCollection as IHitCollection
  - Just like you can use a LarHit* as an IHit*

DataVector “Inheritance”

- Describe element inheritance relation to event store (set of macros)
  - Store creates “views” of the collection for each base class declared via macro

Data Clustering Performance Trade-offs

Event Parameters	File Size (Comp. factor)	Total time to write (Mbytes/s)	Effective time to write (Mbytes/s)	Total time to read All (Mbytes/s)	Total time to read Sample (Mbytes/s)
Comp = 0 Split = 0	48.7 Mbytes (1.0)	19.85s (2.44 MB/s)	7.35s (6.62 MB/s)	7.87s (6.17 MB/s)	not possible
Comp = 0 Split = 1	47.15 Mbytes (1.0)	21.25s (2.12 MB/s)	8.75s (5.39 MB/s)	8.39s (5.38 MB/s)	1.40s (31.77 MB/s)
Comp = 1 Split = 1	36.85 Mbytes (1.30)	24.97s (1.81 MB/s)	12.47s (3.78 MB/s)	8.69s (5.19 MB/s)	1.69s (26.3 MB/s)
Comp = 1 Split = 0	32.01 Mbytes (1.52)	67.02s (0.72 MB/s)	54.52s (0.89 MB/s)	17.07s (2.84 MB/s)	not possible
Comp = 2 Split = 1	27.20 Mbytes (1.73)	65.4s (0.69 MB/s)	52.9s (0.89 MB/s)	16.26s (2.78 MB/s)	4.2s (10.6 MB/s)

Comp=1
gzip ints

Comp=2
gzip everything

Split=1 one branch
per member

- Read all vs sample
- Write speed vs file size

http://root.cern.ch/root/Ebench.html

1997 results!