# Persistency in HEP Applications: Current Models and Future Outlook

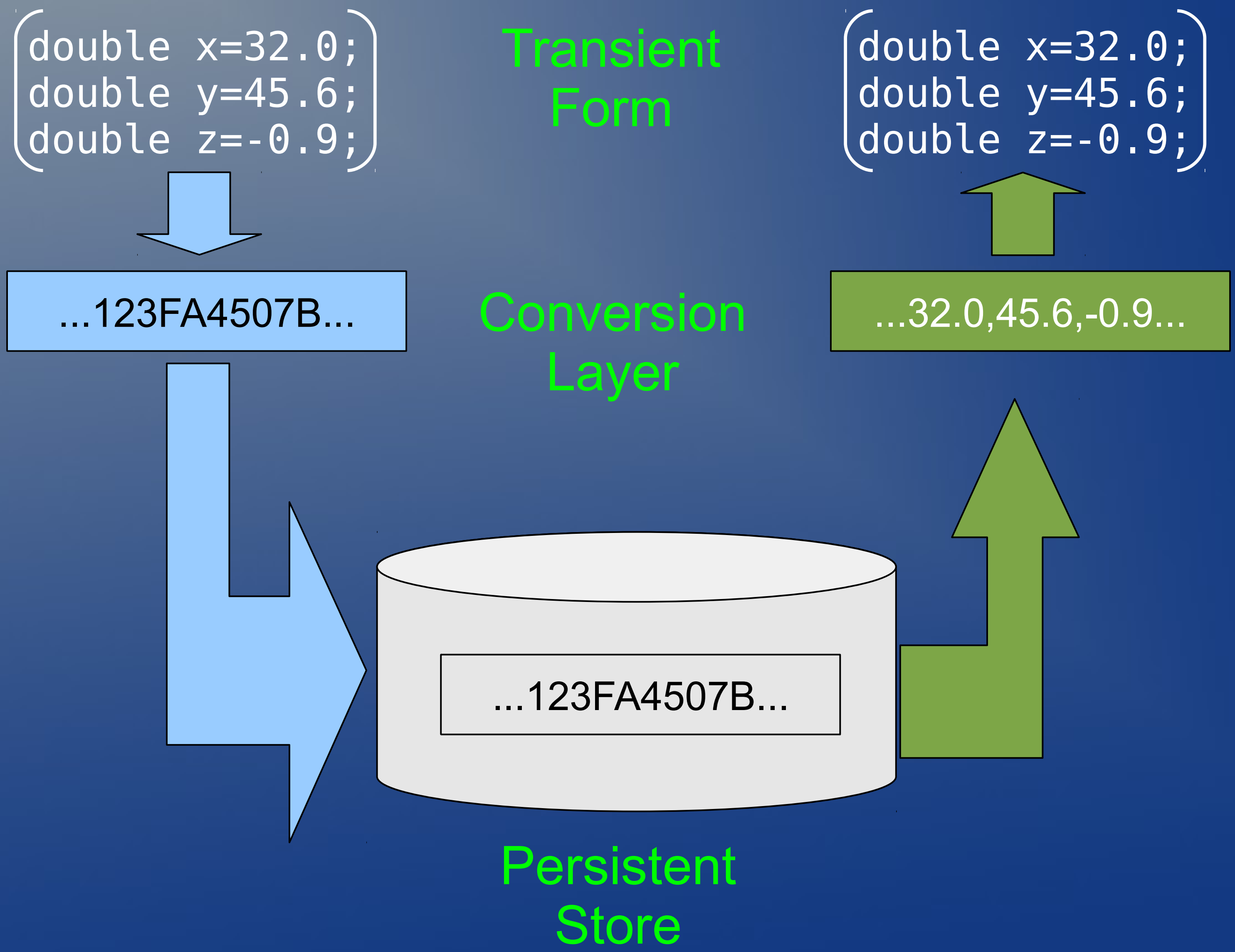
Paolo Calafiura

SuperB Workshop – March 10 2010

# Outline

- Current Models
  - Conversion Mechanisms
  - Event Store Organization
- Persistency and Parallelization
  - Limits of the Event Farm
  - Micro-streaming
- Some R&D suggestions

# Persistency Basics



# Conversion Mechanisms

- Streamer-based serialization (manual)
  - Boost serialization, ROOT TBuffer streaming
- Dictionary-based serialization (semi-automatic)
  - HDF5, LHCb GOD, Protocol Buffers
- Reflection-assisted conversion (automatic)
  - ROOT object store
- Object-mediated conversion
  - Transient/Persistent separation

# 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, 1);
```

# 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 boost version

```
template<class ARCHIVE>
void serialize(ARCHIVE& ar, McCluster& o, const unsigned int version)
{
  ar & o.m_x; a & o.m_y; a & & o.m_z;  //  & takes place of << or >>
  ar & m_truth; //pointer handled by boost serialization
  ar & m_hits; //container handled by boost serialization
}
```

# Boost Serialization Package

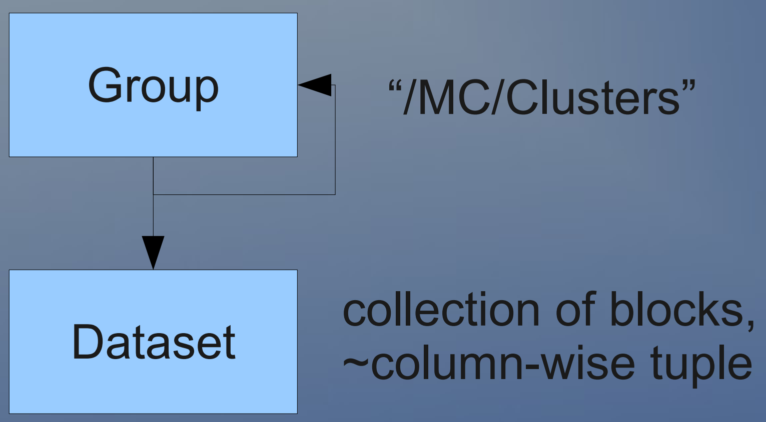
- ANSI C++ based, no dictionaries, no reflection
- Orthogonal specification of class serialization and archive format. Technology independence.
- Data Portability
- Schema evolution support
- Deep pointer save/restore. Proper handling of shared data.
- Serialization of STL containers and other templates.
- Non-intrusive serialization, can be applied to unaltered classes.

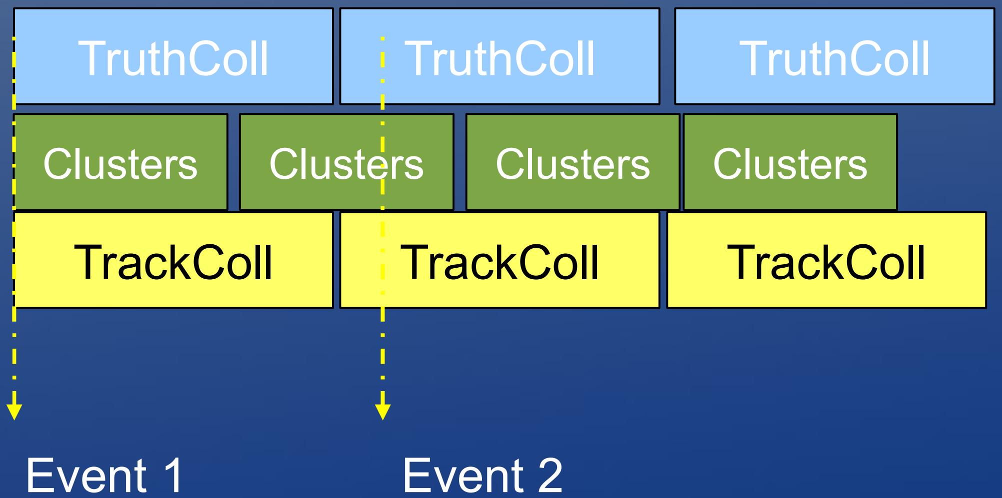
# Dictionary-based Serialization

- Describe data in a dictionary
  - XML an obvious choice (LHCb GOD)
  - Ad-hoc DDL or code annotations also popular
  - Usually  limit data types to c-like structs
    - Not necessarily a bad thing

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE gdd SYSTEM "gdd.dtd">
<gdd>
  <package name="MCEvent">
    <class name="Cluster" author="me" desc="A Calo Cluster">
      <attribute type='double' name='x' desc='X centroid'/>
      <attribute type='double' name='y' desc='Y centroid'/>
      <attribute type='double' name='z' desc='Z centroid'/>
      <relation name="truth" type="HepMCParticle" desc="Pointer to origin particl
        ....
    </class>
    <!-- more classes -->
  </package
```

# HDF5




- Hierarchical Data Store, Unix fs-like tree
- Machine Independent Data Format
- Multilanguage Data Access Library
- Extensive Toolkit:
  - Management, browsing, plotting
- At the core, H5 DDL

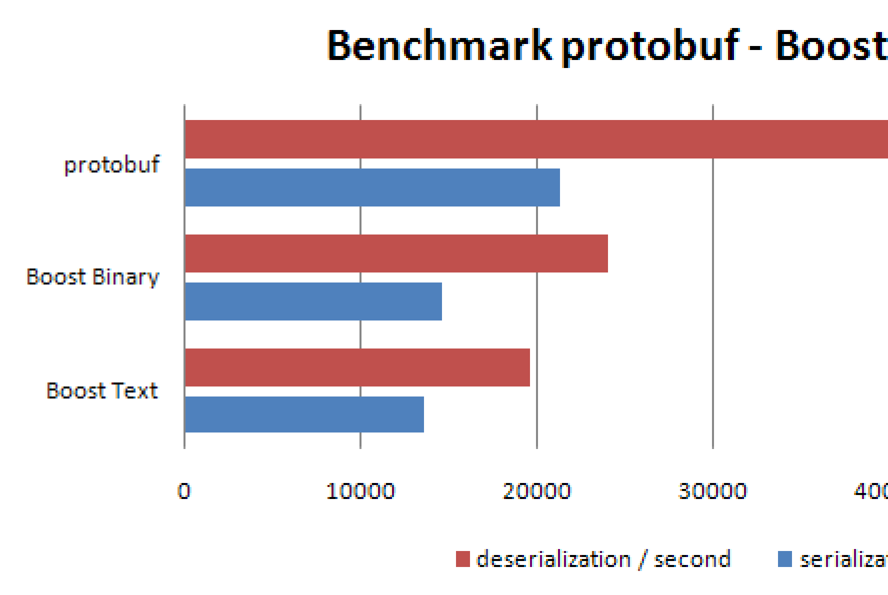
# HDF5

From NCSA, of Mosaic fame

- @ version 5, ~10 years old, mature product
  - Multilanguage serialization, IDL-based
  - Optimized for large data sizes (MB objects in TB stores)
- Parallel version (PHDF5) in production
- Used by “big iron” applications for e.g. checkpoint/restart, but also as a lingua franca for sparse collaborations (e.g. sky surveys)

# Google Protocol Buffers

- Dictionary-based automated serialization
- Multilanguage, very natural API
- Similar to HDF5, but more geared towards data exchange on the wire (RPC, map/reduce)
- Think XML-lite, even faster than boost::serialization (although more limited)




http://fz-corp.net/?p=292
Many more at protobuf site (unfortunately not ROOT)

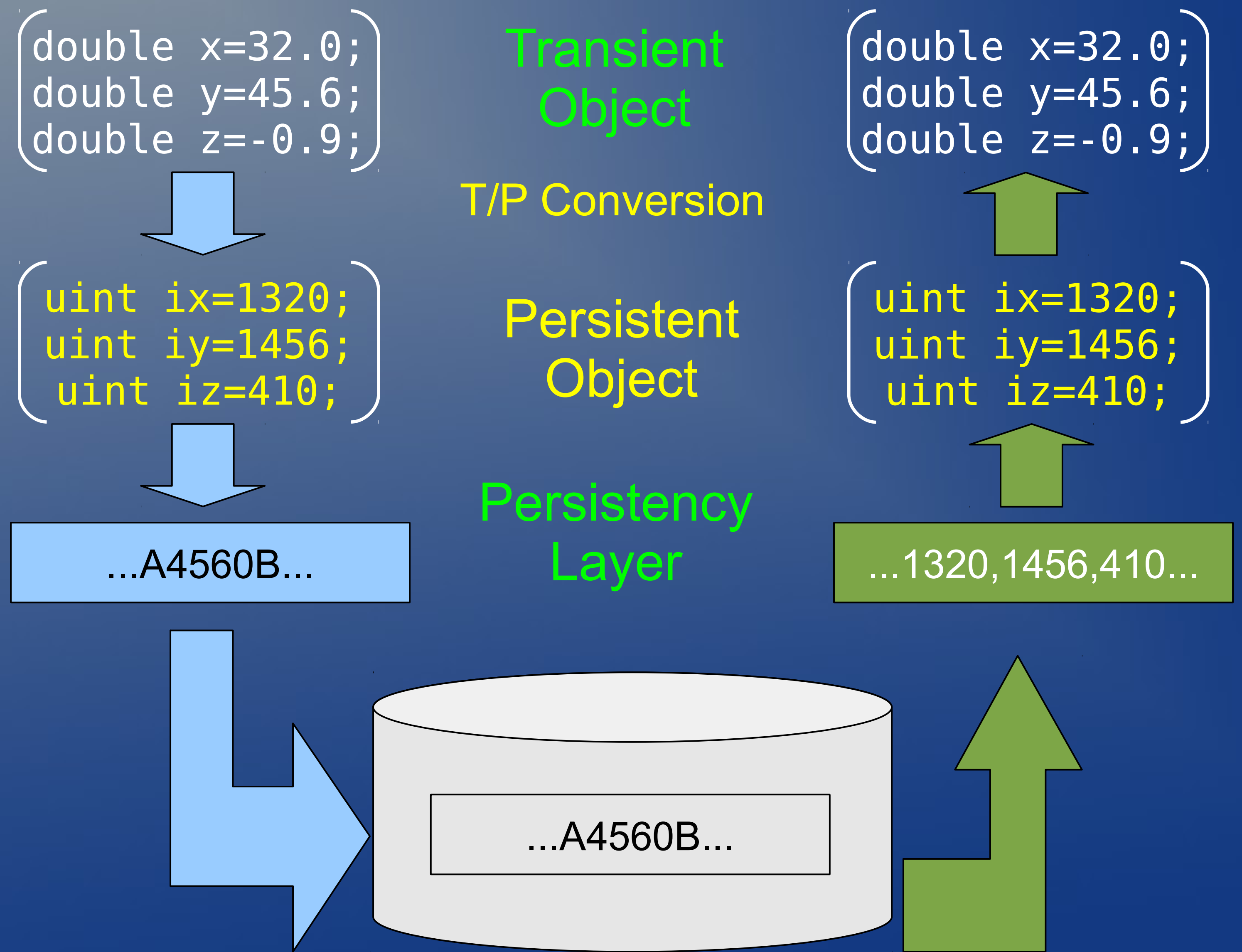
# Reflection-assisted Conversion

- Generate class reflection dictionary
  - Shape (data members)
  - Factory methods (default constructor req'd)
- Use dictionary to auto-generate streamers
  - Pioneered by ROOT/CINT, wide C++ coverage
  - Limited multilanguage support (python, C/C++)
- Automatic persistency but

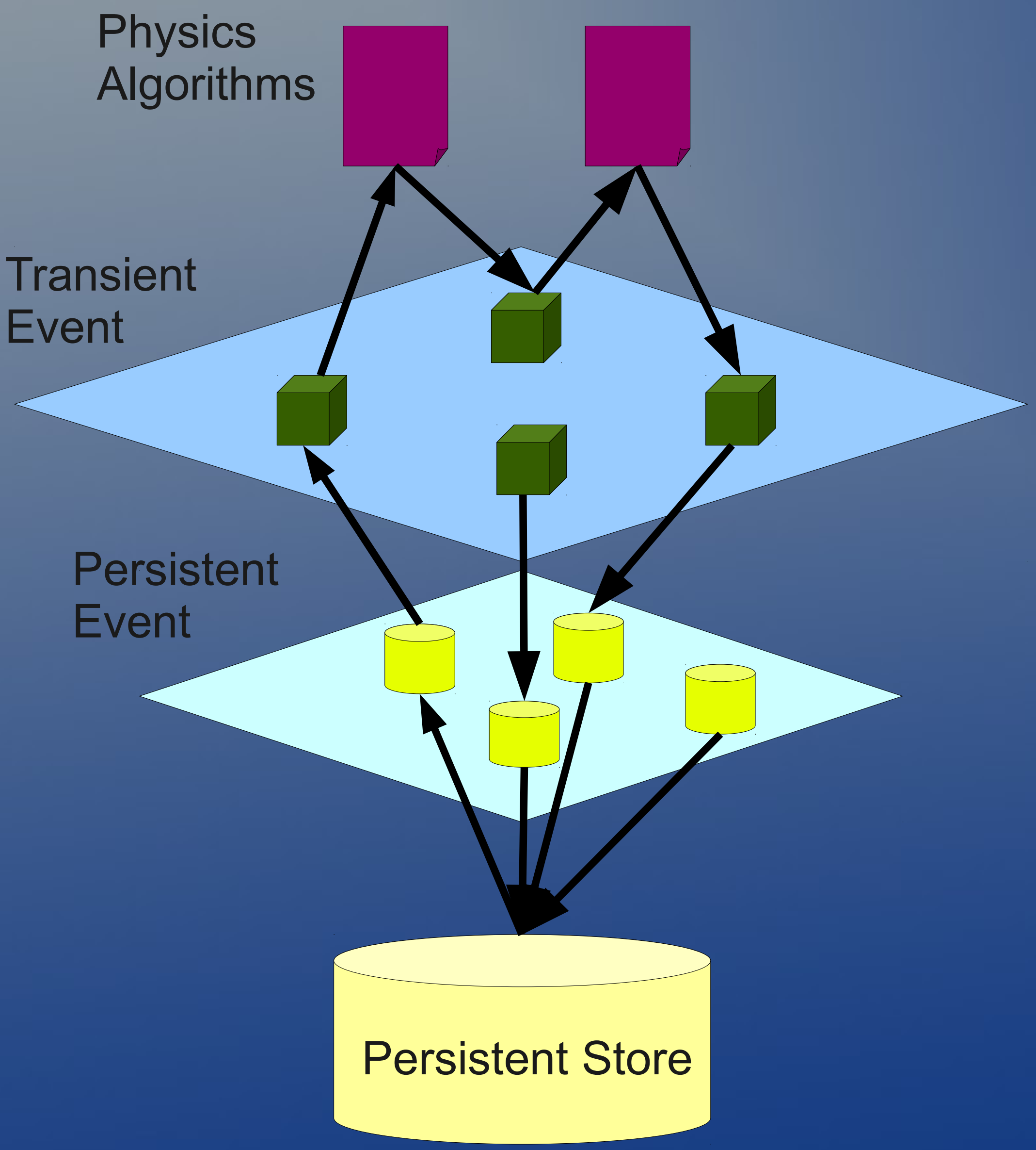
  Efficient persistency constrains EDM design

  - C-like simplicity. Again, probably for the best

# Object-mediated Conversion



# Transient-Persistent Separation




Transient EDM, Technology-independent

- Full language coverage
- Free(r) to evolve

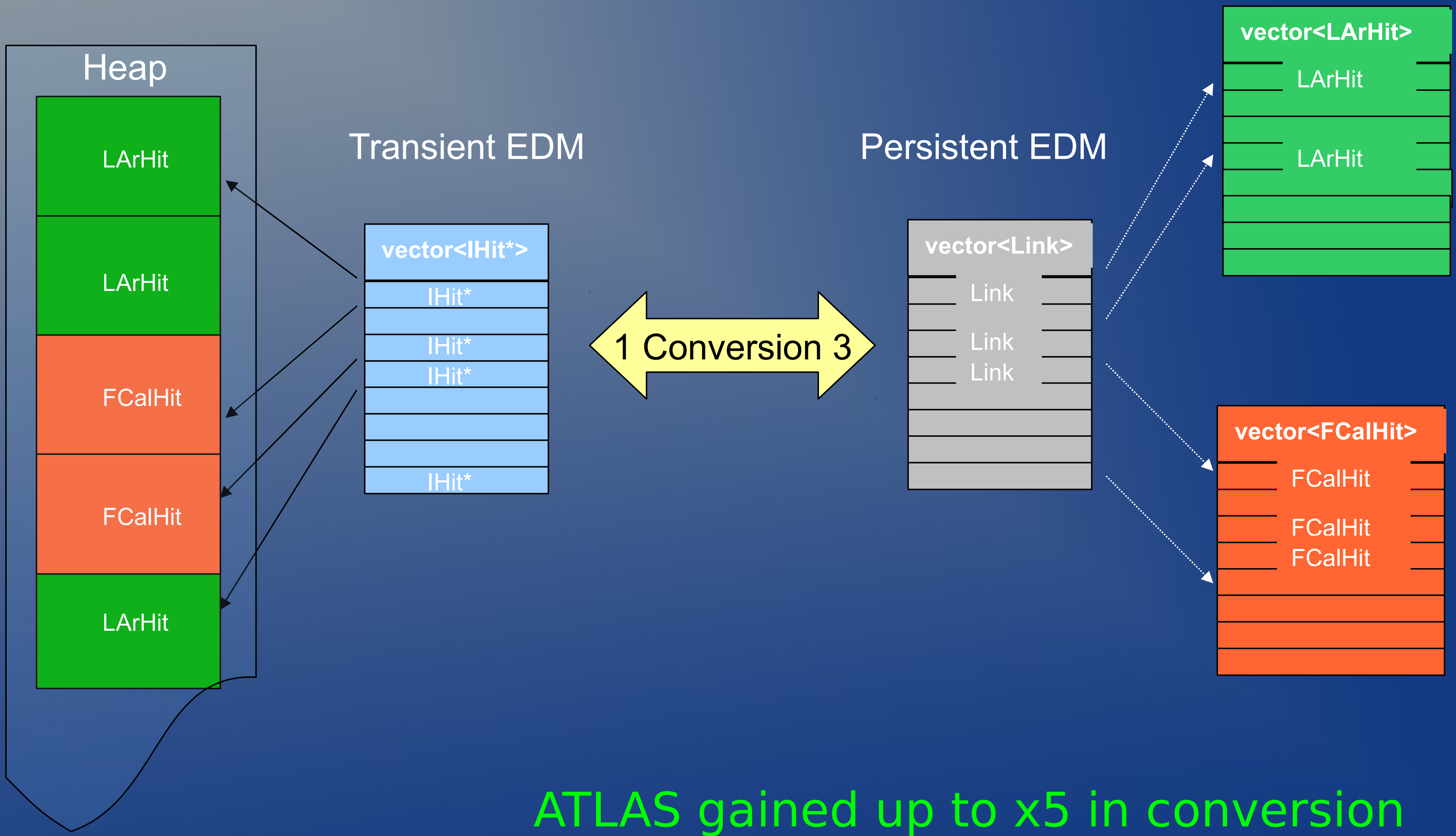
Persistent EDM technology-optimized. For ROOT

- Avoid polymorphism, pointers in general
- Avoid strings, node-based containers
- Use basic types, and arrays thereof

Event-based streaming

Overhead from separated T/P models and conversion

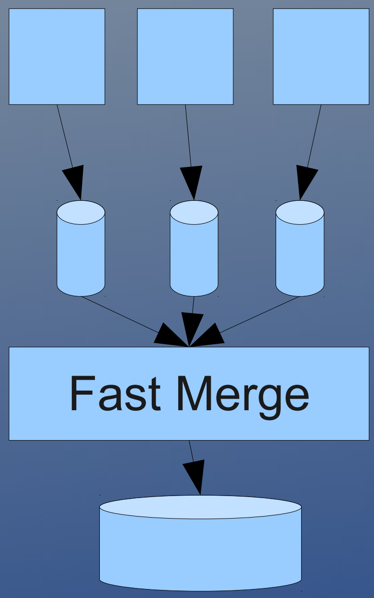
# Why T/P Separation?




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

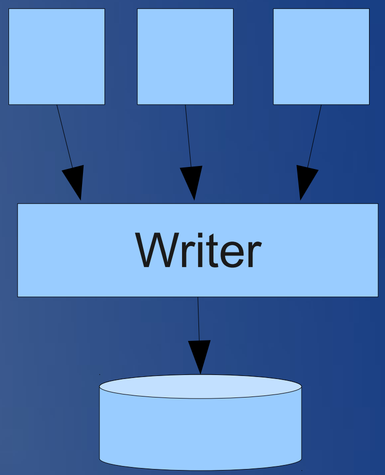
# Persistency and Event Parallelism, the Output Problem

## Write & Merge



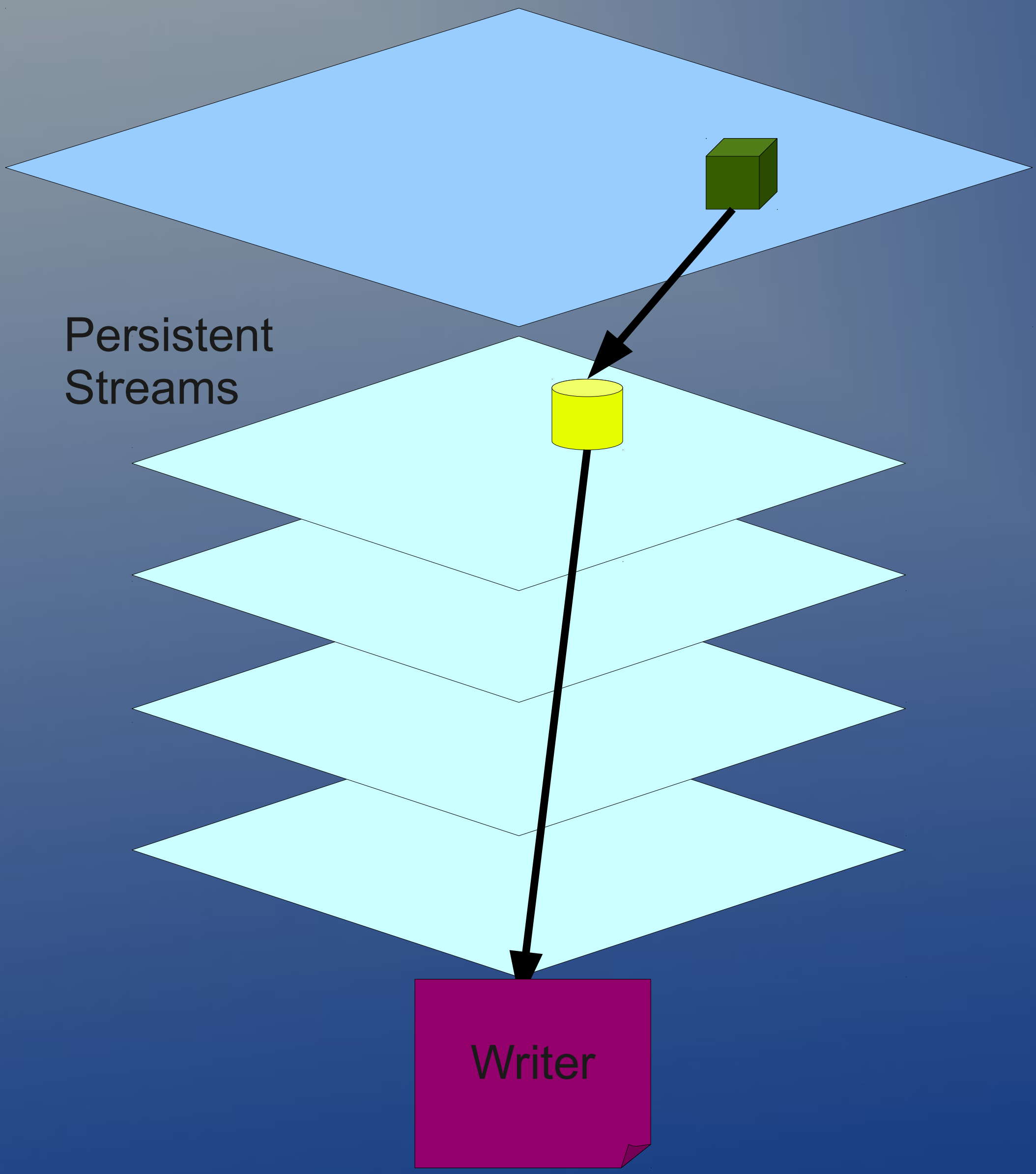

- Wasted I/O
- Metadata
- Event Ordering

## Stream-to-Write




- Serialize data to pipe to writer (structs OK)
- Output sync issues?

# Microstreaming to the Rescue?



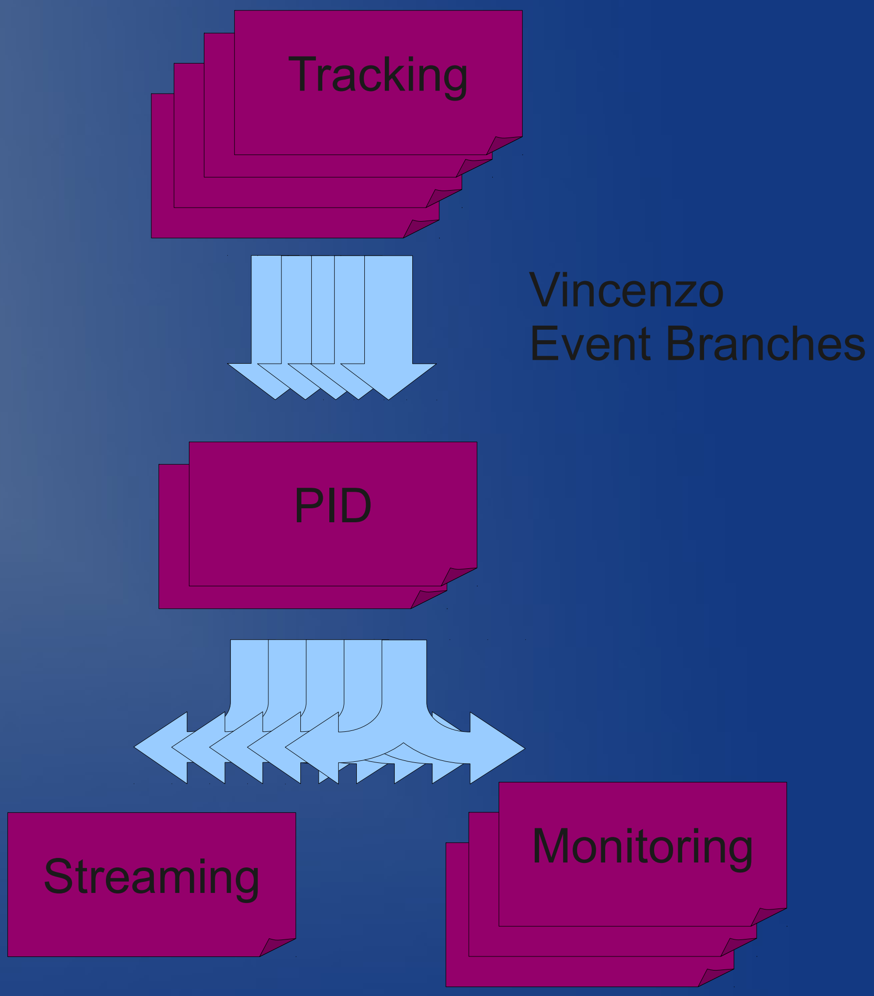

What if we immediately converted each transient data object

- Either keep results in stack of Persistent Streams
  - Write them to disk asynchronously
- Or support parallel write in ROOT
- Persistent refs potential issue

# Beyond Event Parallel

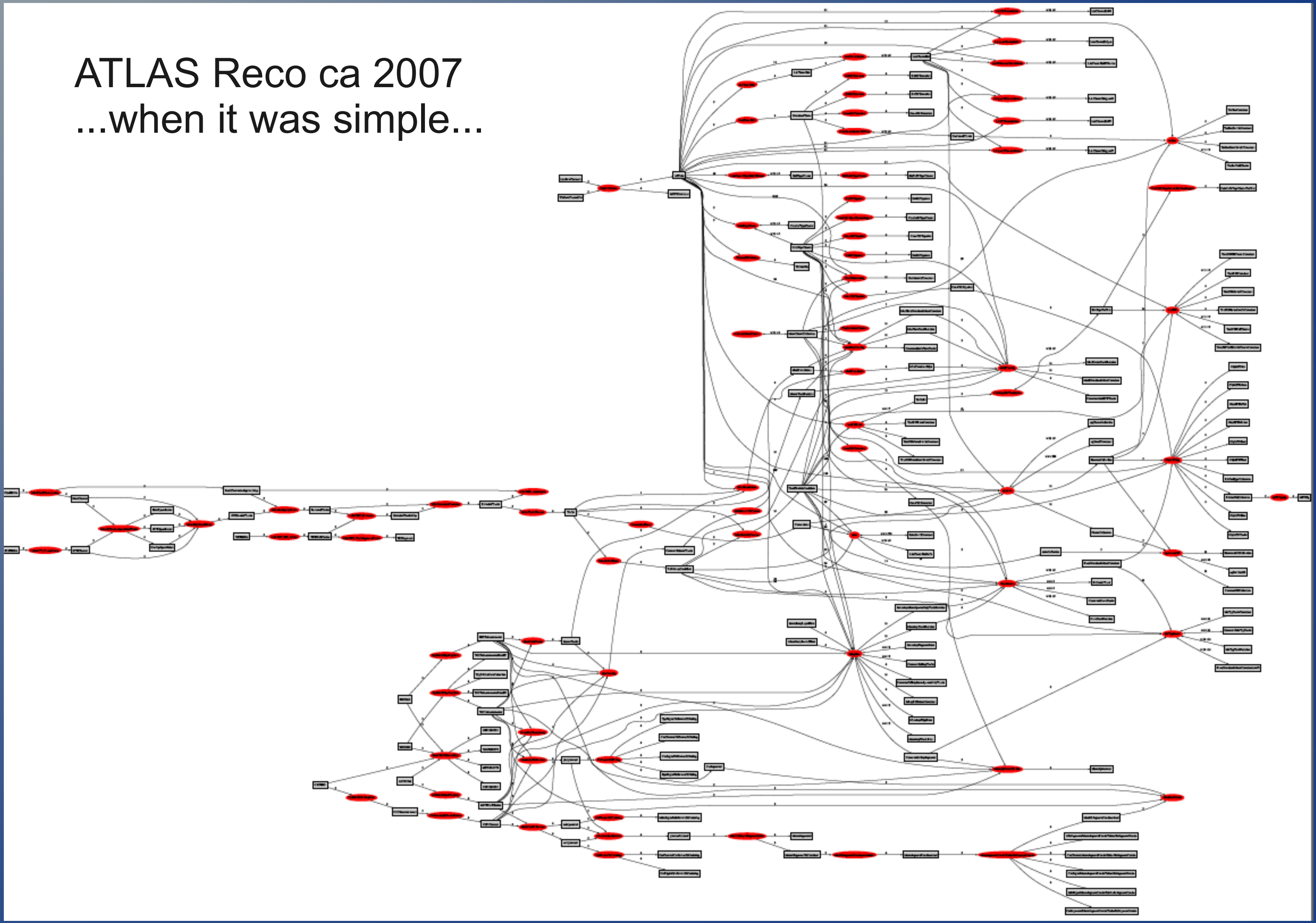
Many-core may require to go task-parallel

- Smaller processes
- Improved memory locality
- Event Branch Pipelines make processes work asynchronously



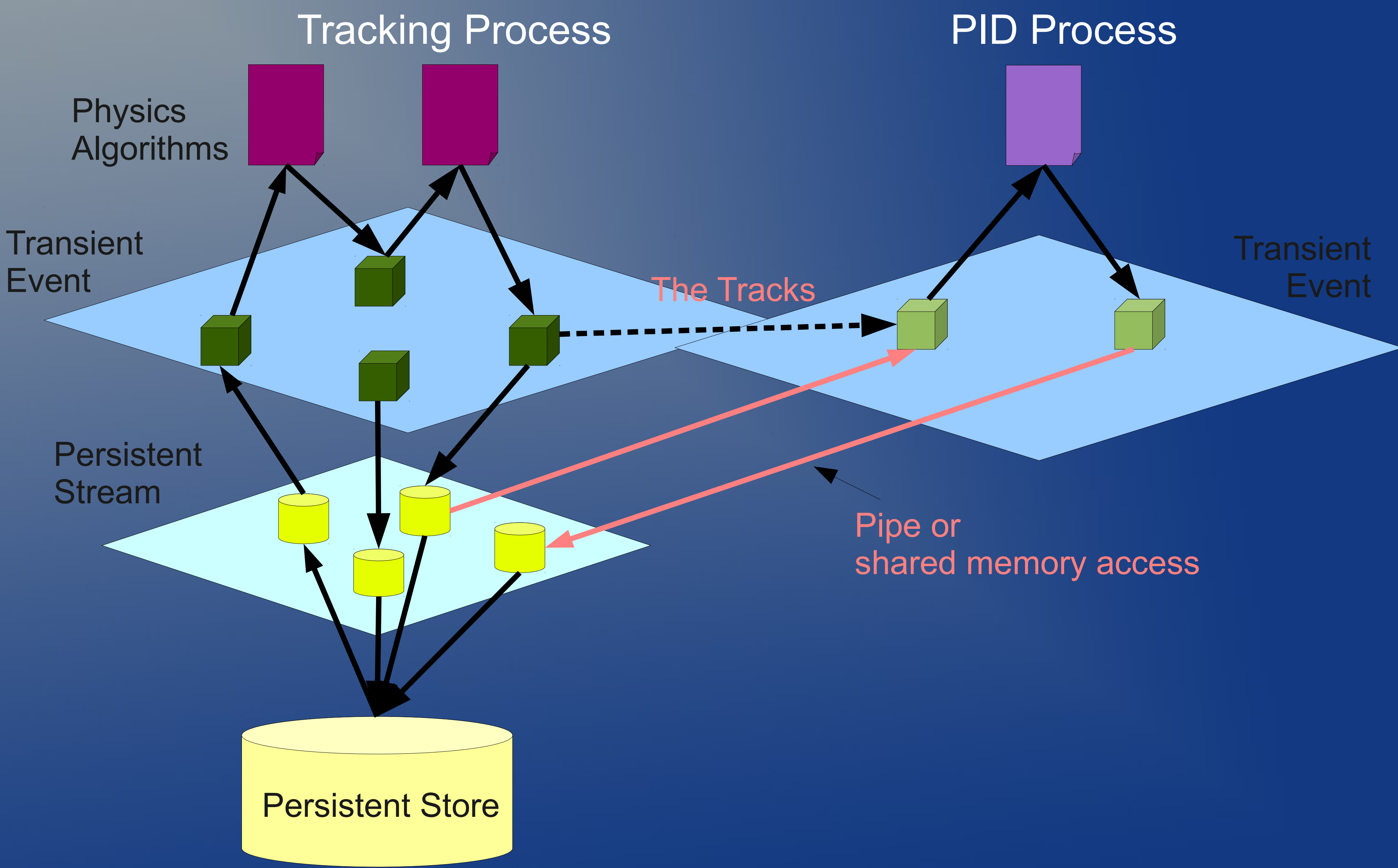
# Reality Check

ATLAS Reco ca 2007
...when it was simple...



Detailed data-flow analysis required to define and optimize event branches

# Event Branches and Microstreaming

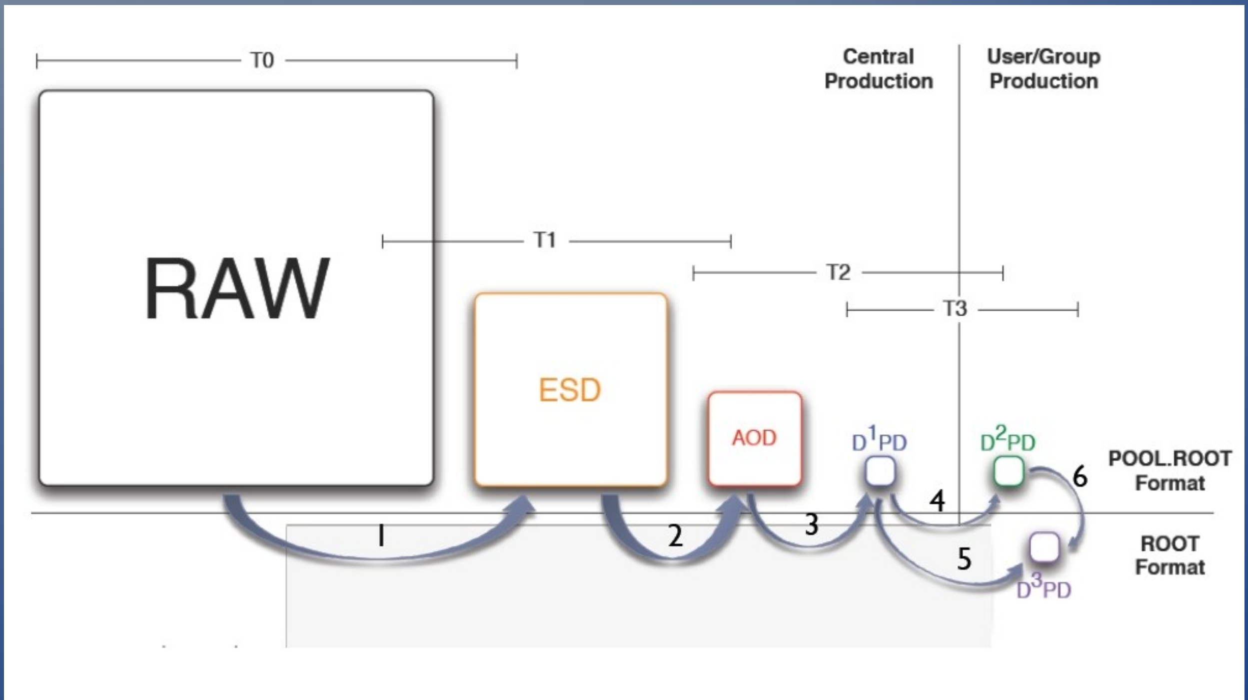


# Known Unknowns.
# R&D suggestions

- Event parallelism is not quite in our pocket
  - Need to address the output problem
    - Will ROOT support parallel streaming?
      - Recent TBasket “defragmentation” both encouraging/worrysome
- Large scale (>32) parallelism may strain write&merge and stream-to-writer approaches
  - Measure using object-level ROOT, ROOT bytestream, HDF5 and possibly protobuf
- Serialization not only for persistency
  - Investigate micro-streaming approach to sub-event parallelism

# Backup

# Event Data Streams and Processing Stages




- Streaming dictated by hardware necessities

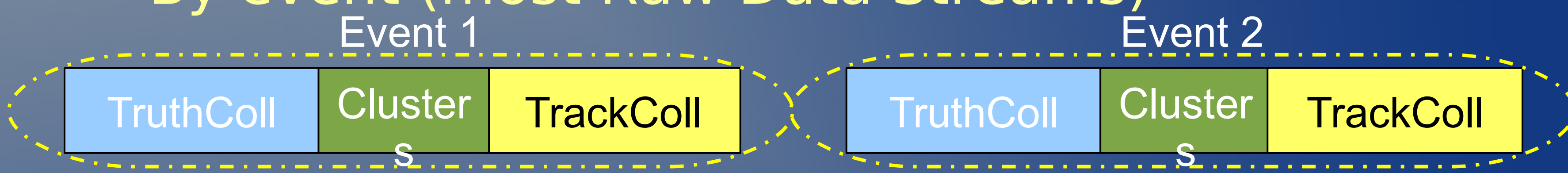
  learned from Babar!

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

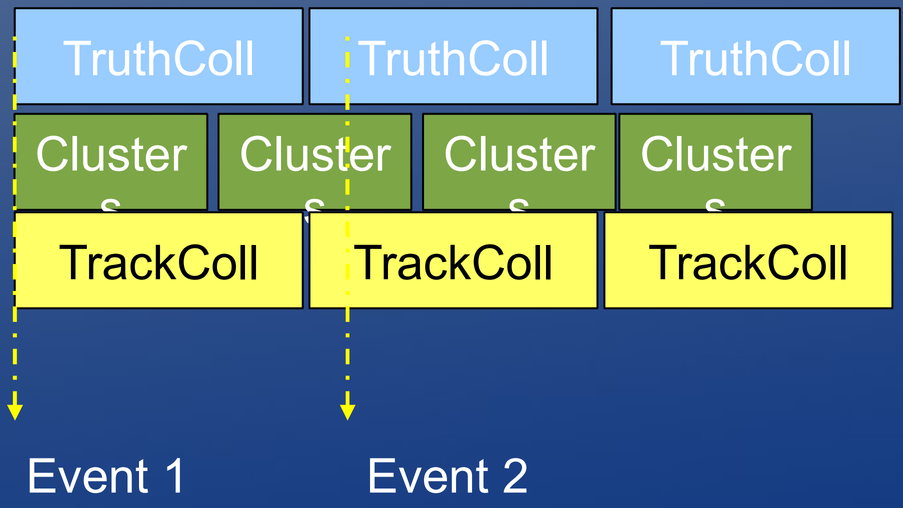
# Data Clustering

## How are data objects written to disk

- By event (most Raw Data Streams)



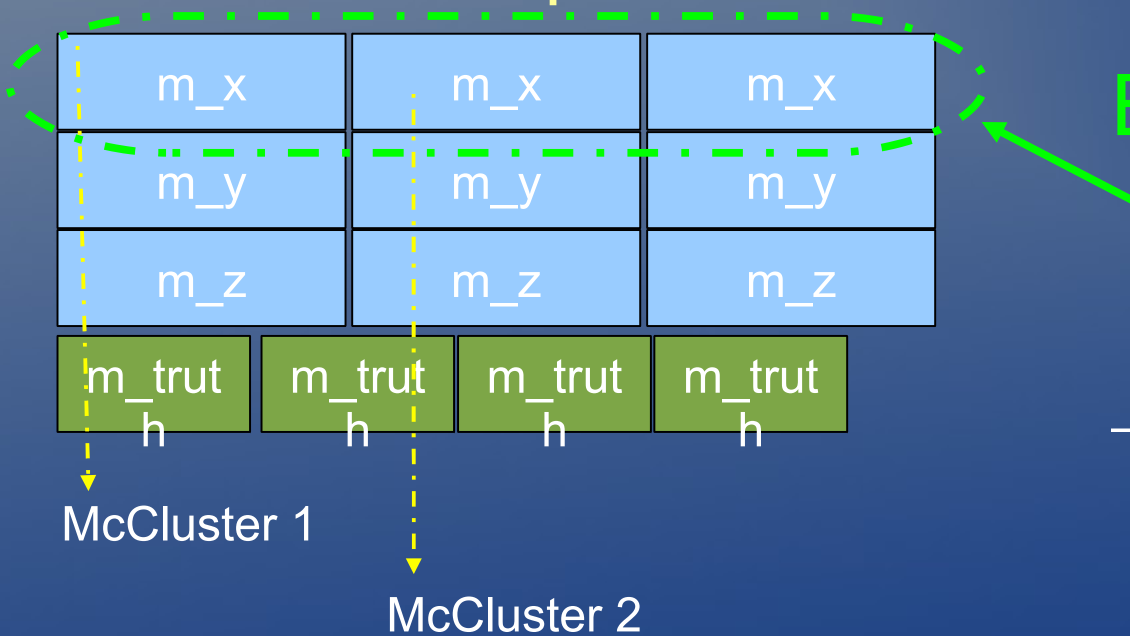

- By object, splitting events (most ROOT files)
  - Allows to read subset of event data



# 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)

# Schema Evolution

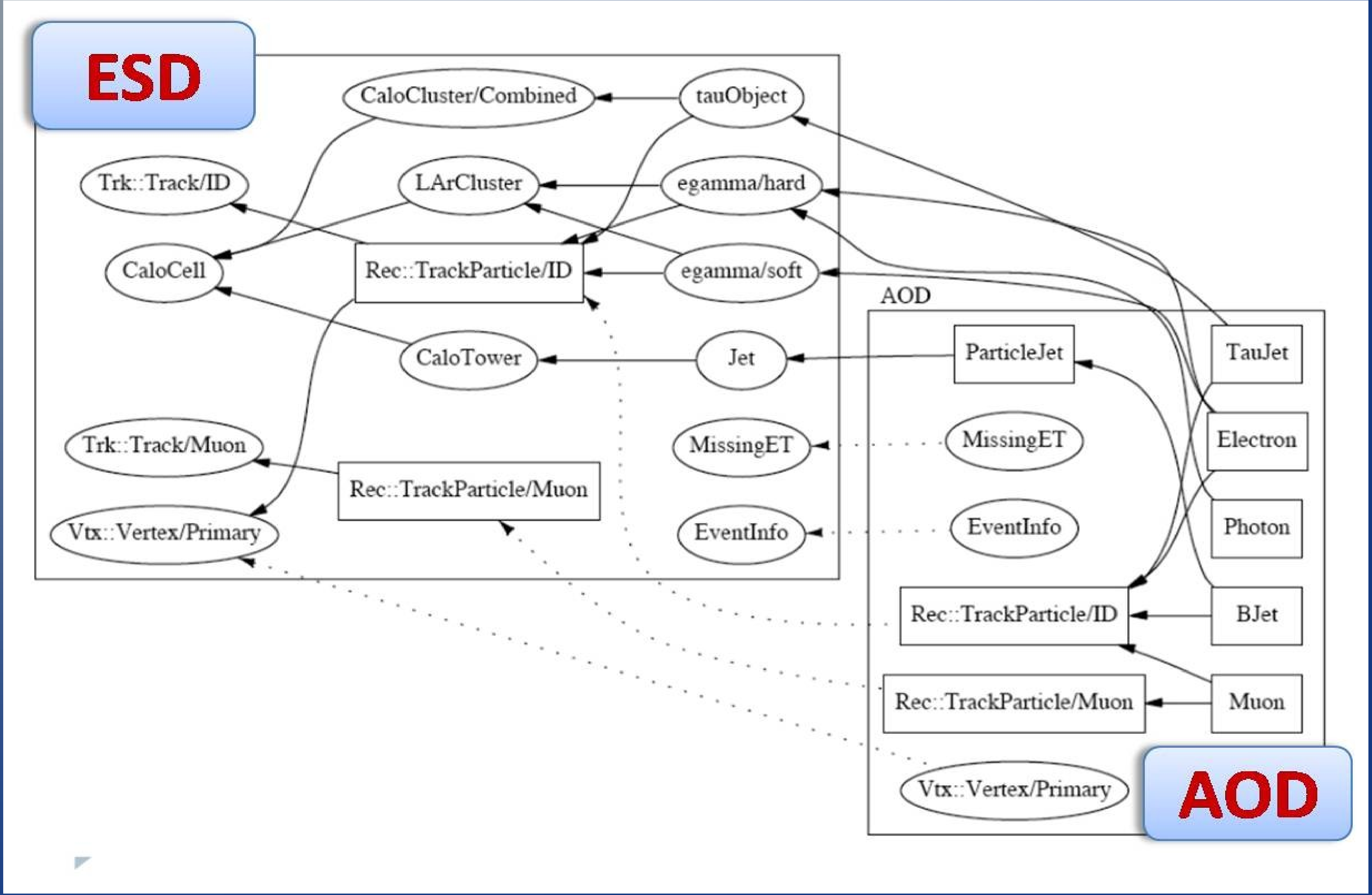
Fact #1: data models evolve

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

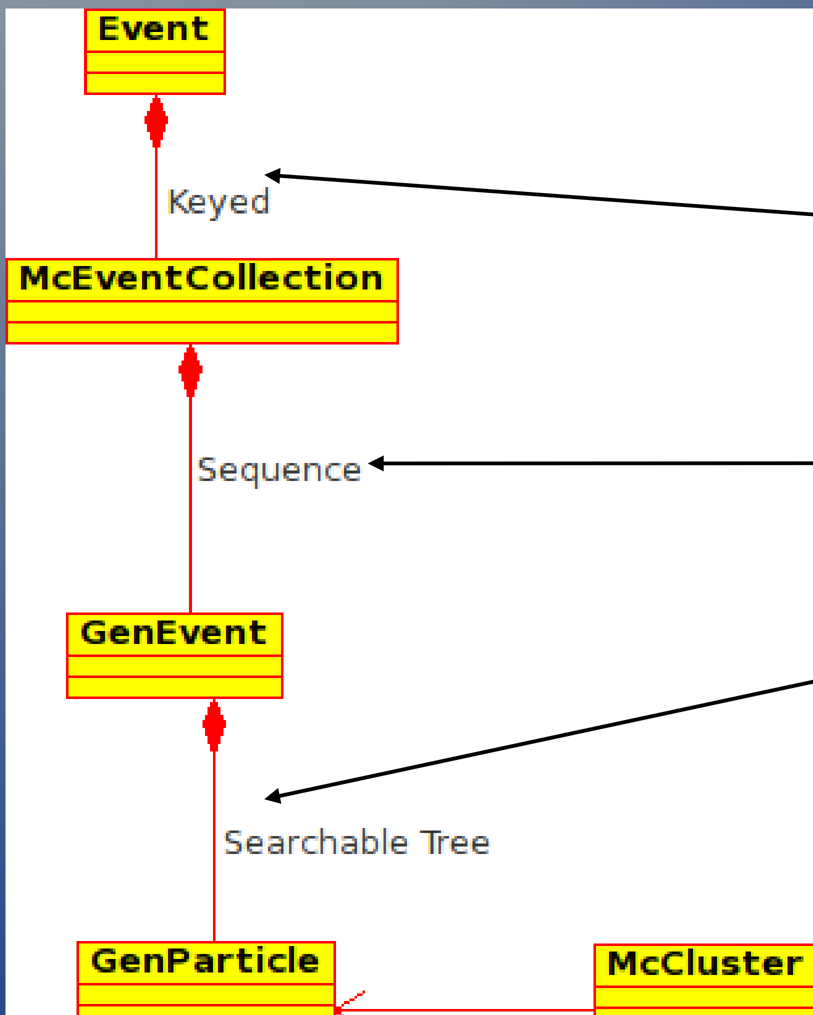
# Persistable References



# 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