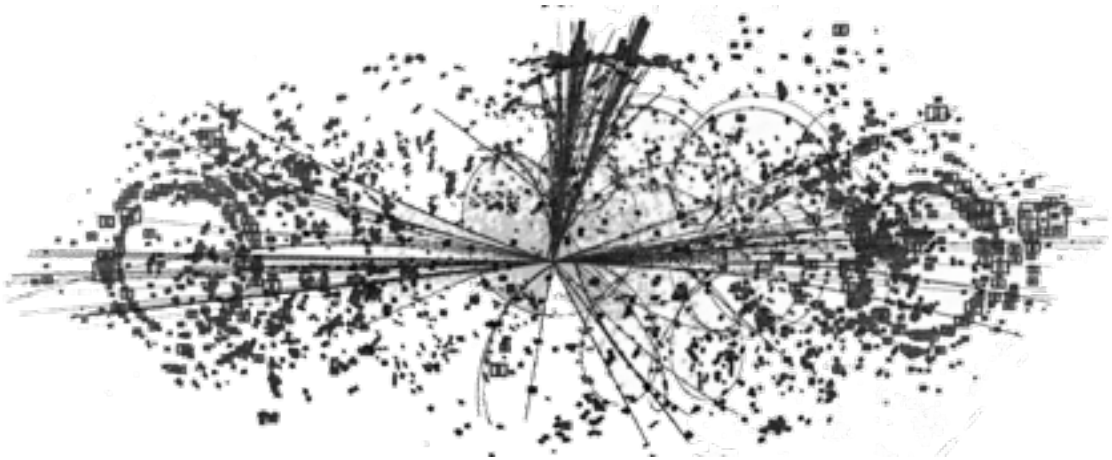


Experience with the CMS EDM

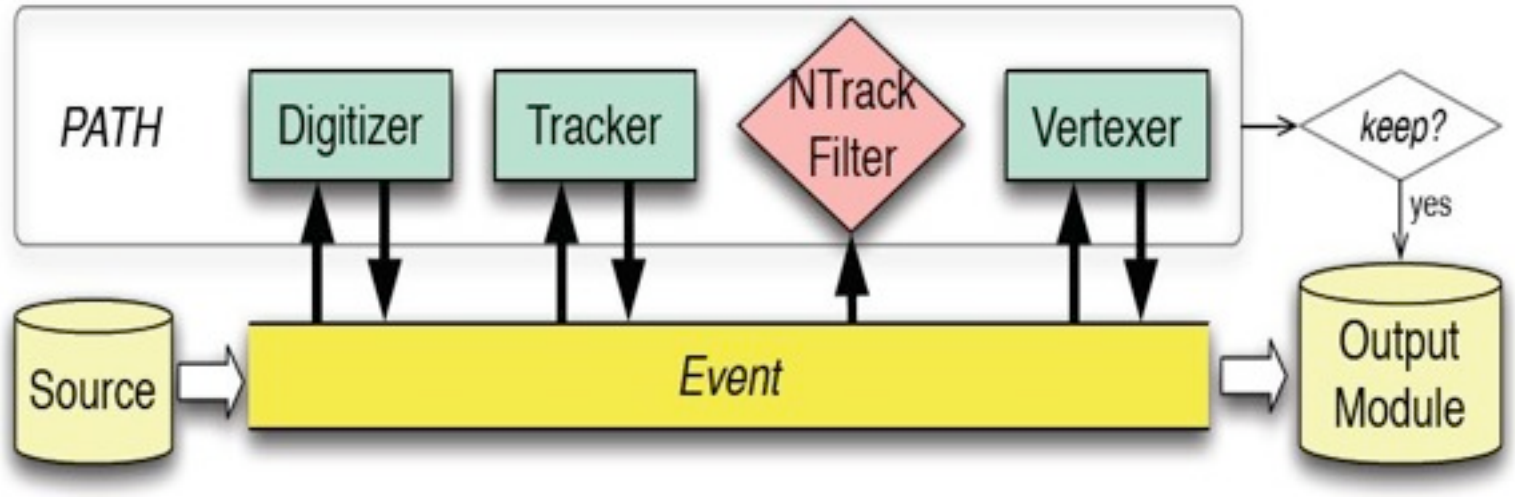
Peter Elmer (Princeton U.), Benedikt Hegner (CERN),
Liz Sexton-Kennedy (FNAL)

on behalf of the CMS offline and computing projects

Basic Concepts

- One executable: *cmsRun*
- The Full Framework uses a modular architecture concept
  - *Module:* component to be plugged into cmsRun via shared object libraries
  - It's a unit of clearly defined event-processing functionality
- Event Data Model (EDM) based on the event:
  - Single entity in memory: edm::Event container
  - Modular content
- Applications are steered and defined via Python job configurations

Flow of the Data



Software Bus Model

- Communication among modules **only** via the event
- Objects (products) in the event, once stored, are **read-only**

Data stored

Single Event data

- At the end of the job individual products can be kept or dropped
- Products can be defined as transient only and will be dropped automatically

Non-event data

- Support LumiBlocks and RunBlocks which span longer time periods
- Can act as temporary storage/cache for objects with the appropriate interval of validity
- Provides a store for offline DQM information
- Caveat: late in the processing chain (user skims) this information can become larger than the actual event data.

Object Storage Implementation

Our goals for storage

- Hide type complexity from the users

```
CompositeRefCandidateT<edm::RefVector<std::vector<reco::LeafCandidate > > >
```

- Avoid writing tons of streamer code
- Good performance, in FWLite track collections can be read at a rate of 3300 events/sec. In full framework, the rate is ~300 events/sec.

⇒ make transient and persistent representation identical so that no time is spent in translating from one to the other

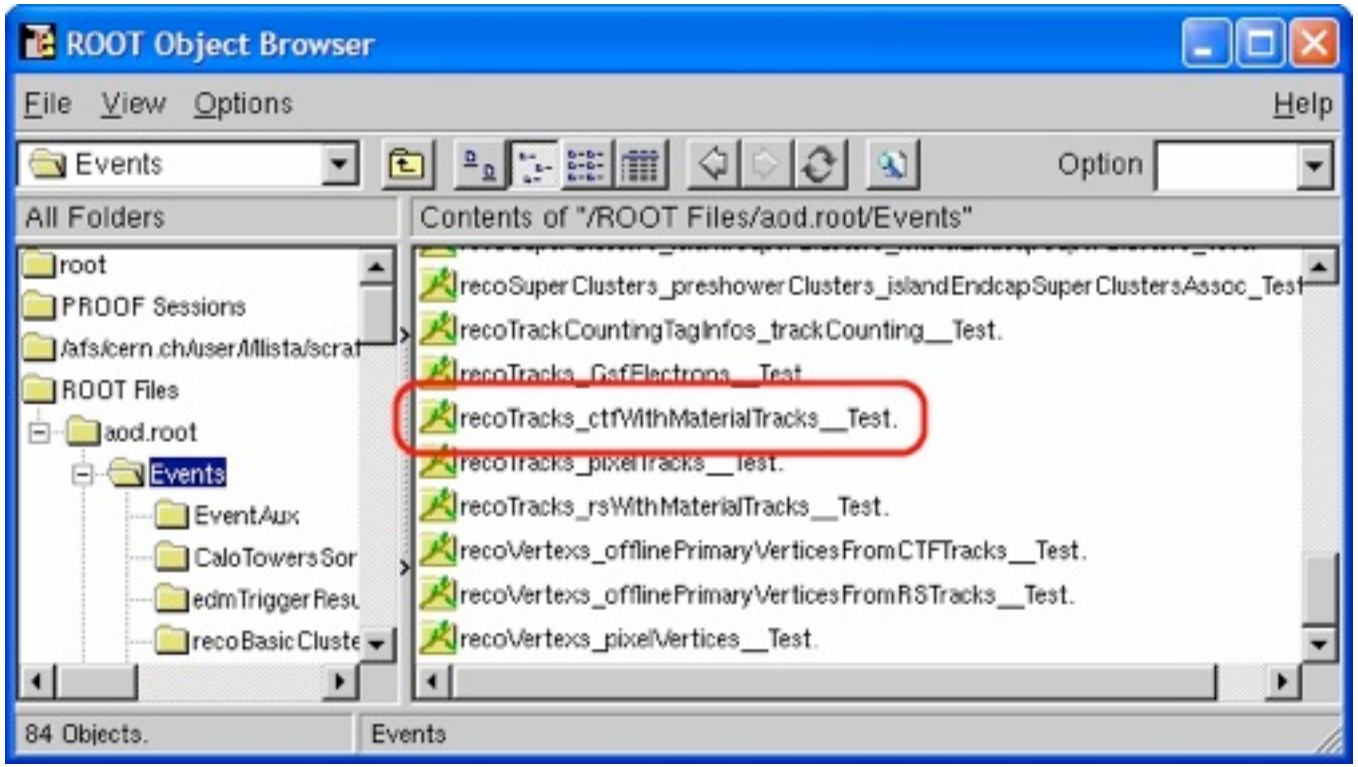
- Avoid n-tuples / make data files accessible with well known tools

⇒ use ROOT for implementation

- Trace origin and history of objects via provenance (see talk by C.D. Jones)

Inspecting Files

```
root.exe
[] TFile f("aod.root")
[] new TBrowser()
```



How to Persistify

classes.h

```
#include "DataFormats/Candidate/interface/Particle.h"
#include "DataFormats/Common/interface/Wrapper.h"

namespace {
  struct dictionary {
    std::vector<reco::Particle> v1;
    edm::Wrapper<std::vector<reco::Particle> > w1;
   };
}
```

classes_def.xml

```
<lcgdict>
<selection>
  <class name="reco::Particle">
    <field name="p4Polar_" transient="true" />
    <field name="p4Cartesian_" transient="true" />
  </class>
</lcgdict>
</selection>
```

Extending Objects

CMS data objects are read-only

- Once placed objects cannot be changed any more
- Vectors cannot get new entries
- From C++ point of view objects are just const

Associations

- Additional data (e.g. btag information) is attached to object in a map like way using persistent references as keys
- Gives flexibility on the RECO level to extend objects with information without making C++ types ‘unstable’
- Adds complexity on the end user side though
- Final analysis layer provides easier interface and can be remade more frequently (see PAT talk by G. Petrucciani)

Net result

What's the result of our choice?

- CMS is one of the few 'power users' of ROOT/Reflex
- Indeed ROOT/Reflex works for our use cases
- It allows us to store even complex data types
- Unfortunately complex things are what people tend to do, and this degrades performance
- Our ROOT files consisted of 12,000 fully split branches last year
- Too long branch names
- That all triggered central intervention -> code policing

Split of Event Data

Event Splitting

- Some event data are complementary
- We have the capability to synchronously read two event TTrees based on EventID, so the content of each tree can be customized to it's purpose.
- Major use case: RECO + RAW data sets

Root Object/Branch splitting

- Improves compression, and access to individual members of an object
- Sets boundary conditions on data types (e.g. polymorphism in members)
- Doing a full split of object read/write results in *many buffers* and *huge memory footprint*
- Often full split not really needed
- We decided to switch from full split to a 'case-by-case' object mode
- We split modes based on known access patterns
  (e.g. low-level RECO objects are unsplit)

Split of Object Data

One problem case with fine splitting - skimming

- In computing challenge 07 (a.k.a. CSA07)
- We exercised the full offline chain from HLT to final analysis data set

- It was the first time we used a large number of output skims from one process
- Each output has its *own copy* of buffers.
- We observed that memory footprint scaled with number of outputs

⇒ restricted to on average only 4 skims/outputs per job

- Following challenge in 08 (iCSA08) still faced a problem due to the huge number of Alignment & Calibration streams.

Merging files

- By taking advantage of fast cloning of TTrees EDM file merging is reasonably fast, 4min. to merge a typical 1Gbyte file. It is IO bound.
- Major use case is in data set creation so that files are the appropriate size for tape storage.
- The input order of events to the merge is important since we read in EventID order.

Merging block information

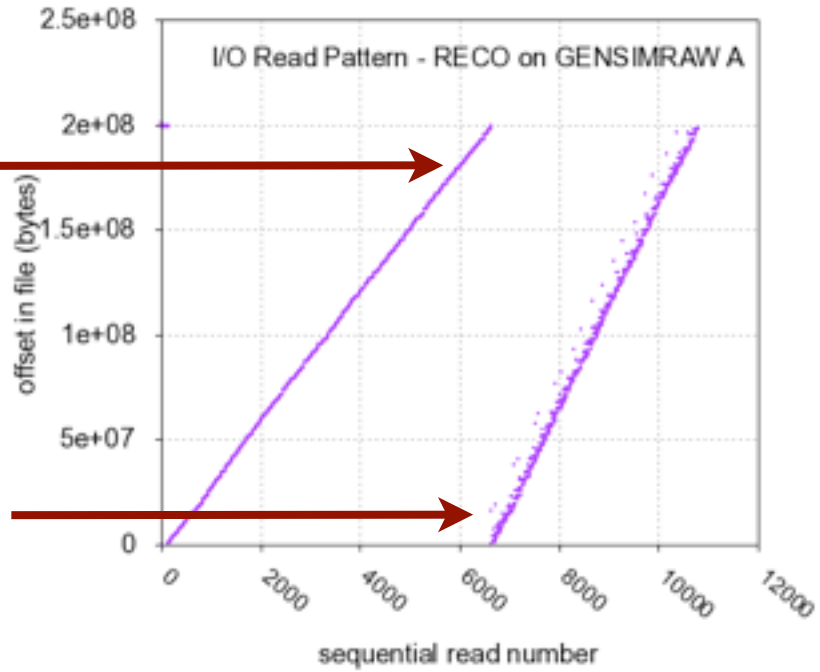
- Lumi- and RunBlock objects get merged automatically (given the type defines a `mergeProduct` method)
- Example: monitoring histograms
- Level of desired merging steered by job option

Read Pattern I

Standard RECO job on MC RAW

The first line happens quickly at the beginning of the job. This is caused by fast cloning of selected branches from input

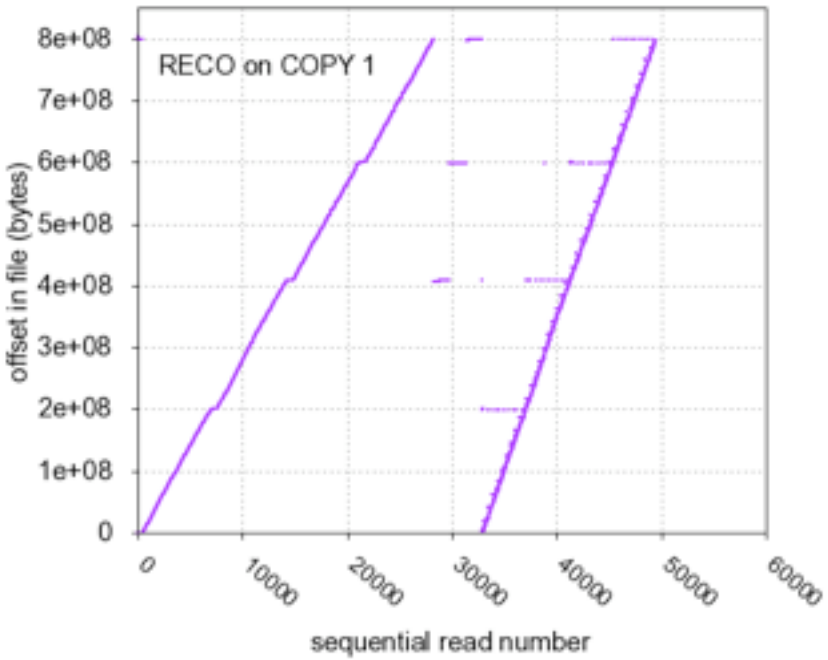
The second line represents reading events for RECO processing. Different objects have different fill rates so different pieces of a single event gets written to different places in the file.



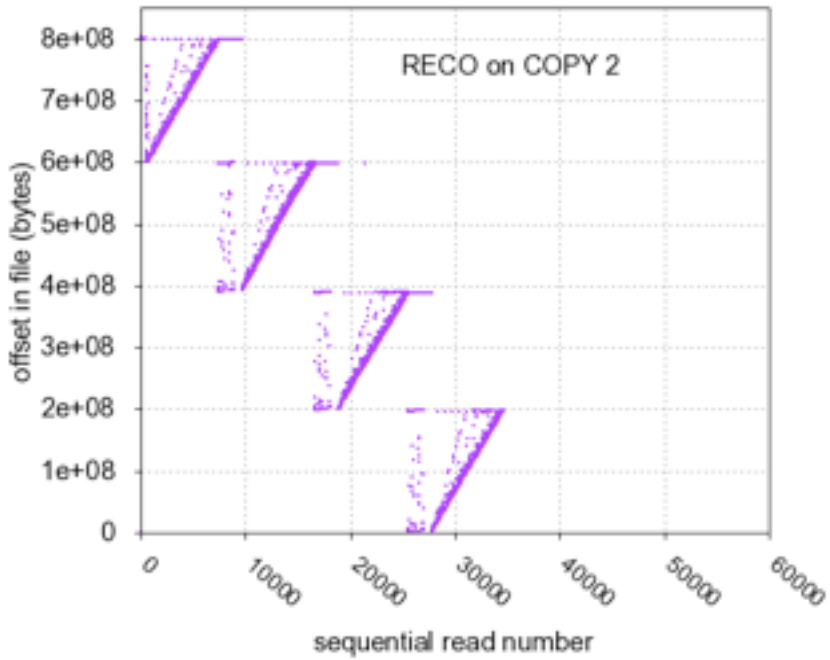
Read Pattern II

Read pattern for two different files. The only difference is how they were written.

After ordered merge



After unordered merge



File Access with ROOT

Opening EDM files with ROOT very well received!

- Bare ROOT access
- ROOT + auto-loading libraries
- ROOT + libs + FWLite
- PyROOT + libs
- Most of them come just for free

FWLite

- Small event loop which mimics full FW
- A negligible overhead w.r.t. ROOT alone
- Used in light-weight event display Fireworks
  (see talk by D. Kovalskyi)

Open items

Further optimization of I/O performance

- Improvements on provenance storage
- In an effort to improve the read patterns discussed on the previous pages we have worked with the root team to add two new features.

1. Merge files in read-order instead of write order. In the process which reads the smaller output files of parallel processing jobs that then needs to be merged to tape appropriate sizes, we can tell the tree cloner to write the new merged tree in read order instead of write order. This fits with our general strategy of optimizing for read since most HEP data is write once read many.
2. Use 'flush buffers' when writing TTrees. In root releases earlier then 5.22, all of the unfinished buffers would be cached on the tree data structure itself. For the large number of branches used in CMS this caused large memory allocations each time a new tree was opened

- These are currently under test

Open items

Schema evolution

- Another major item before data taking is to deploy and commission ROOT schema evolution in order to have old data compatible with new software
- We will collect many years of data and cannot reprocess all data every release. Even in the beginning there will be a rapid release cycle that only a fraction of the data will be able to keep up with.
- Schema evolution will likely penalize IO performance so it should be used with care. It should only be used to “hold us over” until we have enough code changes to justify a reprocessing of the data.
- In other words schema evolution should be used as an insurance policy not as common practice.

Summary

- CMS data model is based on ROOT/Reflex
- CMS is one of few ‘power users’ of the tools
- Letting users store every type they want is dangerous
- Working closely with ROOT team on I/O performance
- Opening data files without full framework is a very successful approach

Thank you very much
for your attention!