View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UNT Digital Library

FERMILAB-CONF-09-307-CD-CMS

Experience with the CMS Event Data Model

P Elmer[1], B Hegner[2] and L Sexton-Kennedy[3]

1 Princeton University, Princeton, NJ 08544, USA
2 CERN, Geneva, Switzerland
3 FNAL, Batavia, IL 60510, USA

E-mail: `benedikt.hegner@cern.ch`

Abstract. The re-engineered CMS EDM was presented at CHEP in 2006. Since that time we have gained a lot of operational experience with the chosen model. We will present some of our findings, and attempt to evaluate how well it is meeting its goals. We will discuss some of the new features that have been added since 2006 as well as some of the problems that have been addressed. Also discussed is the level of adoption throughout CMS, which spans the trigger farm up to the final physics analysis. Future plans, in particular dealing with schema evolution and scaling, will be discussed briefly.

1. Introduction

In 2006 the re-engineered CMS EDM was presented during CHEP. Since then it has evolved to meet some of the chalanges of scale and performance caused by its wide adoption within CMS. Configuration documents have grown in size and complexity. The data model itself has grown in complexity as well. Provenance and other types of non-event data have become important. All of these things will be discussed in the next sections. [2]

2. Configuration

To use the framework, a physicist declares the name of the process to be run along with which modules should be used in the job. Module declarations specify where data should be read from, how new data should be created and where as well as which data should be saved to the resulting EDM file.

In its initial design, a framework job was configured using a dedicated CMS configuration language. Based on experience with the DZERO framework, it was designed as a declarative language in which order of the statements didn't matter; similar to a make script. However, this turned out to be an unnatural choice for our users. They expected an interpreter model with structured programming features. The addition of new, non-declarative features significantly increased the complexity of the simple parser. Due to the maintenance costs of the custom language, and anticipating further need for dynamical features, a new system based on Python was designed and deployed for version 2 of CMSSW [3]. CMSSW is the entire offline software code base which uses the re-engineered framework.

3. Data access

There are two types of data that can be stored in EDM files. Firstly event data which contains objects like hits, tracks or jet collections are stored as so called *products*. The second type of

data are *LumiBlocks* and *RunBlocks*. More will be said about them later. Event data products are uniquely identified using four pieces of information:

C++ class type Required for type safety, e.g., `std::vector<Track>` .

module label A unique string which was assigned to the module which constructed the object, e.g., `globaTrackFinder` .

product instance label If a module inserts multiple objects of the same type into the event, this string is used to differentiate the different products. If no product instance label is specified during a data access then we use an empty string as the default since that is the same default value used when data products are registered.

process name A unique string assigned to the process being run. This keeps data from different processing steps (e.g., `HLT` or `RECO`) from interfering. If no process name is given during a data access we default to the most recently run process which has a match for the other three pieces.

Inter-product links between different objects are supported via CMS specific smart pointers like `edm::Ref<>` which used when the concrete type is known and `edm::Ptr<>` used when the reference is to a base class. For example, tracks contain a list of `edm::Ref<>`s to the hits from which they are formed. A hypothesis for an event can be saved as a collection of `edm::Ptr<>`s to generic physics analysis objects.

Data retrieval from the event can happen via various methods. The simplest example is by specifying the object type and module label. Requesting a product from the Event class, it returns a smart pointer of type `edm::Handle<>`. The following code reads an instance of a collection of tracks created by a module called *tracker*. In case there is no match for this request, an exception is thrown when the handle is dereferenced. It is also possible to test for a valid handle if throwing an exception is undesirable :

```
Handle<TrackVector> tracks;
event.getByLabel(tracker,tracks):
std::cout << ''The number of tracks in the ''
              << tracker << '' collection is '' <<  tracks->size();
```

Based on the four product identifying strings explained above, individual products can be kept or dropped at the end of the job as specified by the physicist. In addition products can be defined as transient only and will then be dropped automatically.

Products within run and lumi data blocks which span longer time periods are also identified and looked up by the same set of strings as event products. They can act as cache for objects with the apropriate interval of validity and provide a store for e.g. offline DQM histograms.

This non-event data can become larger than the actual event data stored in the file, especially in the later steps of the processing chain like a highly selective event skim. Such files usually contain events from a large time window. Care must be taken to keep only the most highly summarized information needed for analysis in these data products.

4. Data Storage

The initial goals of the design of the storage system were a) have a simple interface to the physicist, b) ease of maintenance, c) good performance and d) avoid the need for doing analysis outside of the experiment infrastructure for example analysis with n-tuples. The combination of all these lead to the usage of ROOT and Reflex for the object storage. Whether all of these goals were achieved will be discussed in the following sections.

4.1. Physicist User Interface
The steps needed for persistency is only visible to the person declaring a class as persistent and for most of the types it is almost trivial. To store a collection of particles the following lines need to be declared in a *classes_def.xml* file:

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

These define the particle (and declare two of its members as transient caches), the desired collection type and a wrapper to facilitate the storage in the event. Instead of directly holding the data product, the storage actually holds an `edm::Wrapper<>` class which holds the data product. This allows additional status information to be stored as well as giving a common base class (`edm::EDProduct`) which provides a standard virtual destructor in order to properly delete objects once the event has been processed.

In addition to advertising the persistent types the template types need to be instantiated for the creation of the Reflex dictionaries.

```
#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;
   };
}
```

One of the unexpected draw backs of the ease of definition of even complex data types, is a proliferation of complex class leading to an overly complex event data model. Though each of these classes individually could be fine, their total sum caused huge performance and memory penalties. Code review and policing turned out to be essential to keep the complexity of class definitions at a reasonable level. Another lesson learned was that C++ meta-programming techniques used encoded a lot of run time behavior in the type. For example the sorting algorithm for a vector held by a class can be specified as a template argument to the class. While these techniques can be elegant in a transient only application, in the persistent world these templates will show up as different types, even though it makes no difference to the way the data in the class is stored. This is a rather specialized form of template bloat. CMS has corrected this where it can but we still need a special build of root that allows more type names then the default of 4096.

4.2. Ease of Maintenance and Good Performance
Since no custom streamer code needs to be written, the class maintenance in terms of persistency is kept minimal. Furthermore the choice of encouraging identical transient and persistent representations of objects, avoids translation between the two and the resulting computation overhead. Measurements with the full CMS framework (version 2.X.Y) on the access of, for

example the track collection, result in a throughput of 300 events/sec, and 3300 events/sec for FWLite (see further below).

4.3. Analysis Access with ROOT

One of the main goals of the EDM was to allow ROOT users direct read access to the files created by the framework. This is to allow the physicists to develop his analysis in ROOT without creating custom n-tuples. Opening event files with ROOT is a very well received feature and replaces many n-tuple use cases [4]. On top of this bare ROOT access, we provide additional functionality which we call FWLite. FWLite consists of the data format libraries, including Reflex dictionaries, an auto-loading mechanism, an event class that is similar to the full framework interface. The auto-loader is based on the CMS plugin system and takes care of finding the proper dynamic libraries needed for the products in a given EDM ROOT file. Writing a ROOT based event loop, especially if the CMS smart pointers are used, presents a series of difficulties for the physicists, as described in [4]. Therefore we provide a small event loop which mimics the full framework, called *fwlite::Event*. FWLite again only relies on the data format libraries. One application based on this lightweight framework is the *Fireworks* event display [5], which has become the main event display for the use case of physics analysis.

The minimal code to access a track collection using FWLite is given below. As mentioned it takes advantage of the automatic library loader, which loads all needed libraries to the given EDM ROOT file:

```
//Load the autoloader
gSystem->Load("libFWCoreFWLite");
AutoLibraryLoader::enable();

#include "DataFormats/FWLite/interface/Handle.h"
TFile f(...);
fwlite::Event ev(&f);
for( ev.toBegin(); !ev.atEnd(); ++ev) {
  fwlite::Handle<std::vector<Track> > tracks;
  tracks.getByLabel(ev,"trackFinder");
```

The performance overhead of these lines compared to ROOT alone turns out to be negligible. Applying the same auto-loader it is possible to write end-user analysis code in Python as well. [6]

To allow the packaging of a small distribution of only the data format libraries, all CMS data objects are by design kept independent from algorithm or other framework components. This strategy has been successful resulting in FWLite distribution rpms that are 20% of a full CMSSW distribution.

5. Provenance information

Another feature of the CMS EDM is the addition of provenance information to aid in understanding the history of processed data. Storing this within the ROOT file allows the physicist to query the file with the tools we provide and gain confidence that they understand what and how that data in that file was calculated. Already during cosmic data taking it turned out to be a feature frequently used by end users and a valuable tool for debugging. It is especially important due to the highly distributed nature of the CMS computing model. For very small skims the full provenance information can easily become larger than the actual event data. Thus, depending on the use case, the detail level of the provenance information can be customized. More details can be found in [7].

6. Associations

The design choice of making objects read only once placed into the event, requires that addition of information from further down the processing chain happens via associations. Some object data can only be computed after an object has been created. For example, a b-tagging value of a jet is attached to the jet by adding an entry for it into an association collection. An association collection is a map like structure using persistent references. For this example the b-tagging module creates a b-tagging association collection object to match its input jet collection. This allows us the possibility of extending objects without changing the C++ type of that object and thus increases the stability of our data format definitions. On the physicist's side this can lead to confusion since information related to a single object is spread among multiple event products. This is being addressed by an additional analysis layer which acts as a view on the RECO data and makes the data accessible via single entry points [8].

7. Splitting

The framework supports splitting of data at several levels. Some event data, like RAW and RECO information, are complementary and do not always need to be processed or even distributed together. To facilitate this, data can be split at event level among different files. If data from both files are required, the reading process can do a synchronous read of the two event TTrees, based on the event ID, which will reunify the event. Due to technical limitations both trees are not equal. One of them has to be considered the primary data source. The primary source controls which event IDs will be processed and can have persistent references to objects in the secondary source. The secondary can not have references into the primary. It is for this reason that the primary always has to be the output of a later step in the processing chain.

Another splitting of data happens on the product level. This improves the compression and gives access to individual members of an object. On the other hand it sets boundary conditions on the data types, e.g. if polymorphic members are used the data will not be split below that place in the object hierarchy. Initially we started with a full split mode. Together with the already mentioned complex data types, full splitting lead to a large number of branches, on the order of 12,000. This resulted in a need for many I/O buffers and a huge memory footprint.

As the full split is often not really needed, the current policy is to set the split level at a case-by-case basis. There the used split modes are based on known access patterns. For example low-level RECO objects can easily be kept unsplit.

Another use case which demonstraits the limitation of too many output buffers is a central skimming workflow. Since each output stream has its own copy of the I/O buffers, the memory consumption scales with the number of skims run in a single job. The first time production jobs with multiple outputs were run, during the 2007 Computing/Software/Analysis Challenge (*CSA07*), there were large operational problems with memory consumption. Based on these initial problems, the number of skims per job had to be limited to 4. Though we later improved the situation by adjusting the split level of various products, we still faced a problem during the following challenge *iCSA08* , where a much huger number of alignment and calibration streams were required.

8. Merging

To allow files to be of reasonable size for e.g. tape storage, multiple data files can be merged into a single file. The file merging takes advantage of the fast cloning of *TTrees* and is thus entirely I/O bound. Currently available I/O rates translate into an execution time of approximately 4 min to merge a typical set of files to a 4 GB ourput file. The order of the event files fed to the merge is important as events are read in Event ID order. The readback of an unordered merge

can be extremely inefficient since any caching scheme will be defeated by the skipping around within a file that will be needed to read a complete set of events sequentially.

The Framework will either store one copy or integrate the data accumulated, depending on the interface provided by the user object, when multiple files are merged.

During the merge block information objects from the same Lumi- and RunBlocks get merged automatically. The Framework will either store one copy or integrate the data accumulated, depending on the interface provided by the user object. For integration each mergeable type needs to implement a *mergeProduct* method. The major use case for this is the merging of monitoring histograms used in the CMS DQM framework. Types whose data is constant for the entire time interval must implement *isProductEqual* so that consistency of the files being merged together can be checked by the EDM. Depending on the access pattern and available resources the block merging can be done at various levels to control the memory usage. Especially for the case where not properly ordered event files are merged, many temporary block information objects need to be kept in memory if a full merge is done.

Delivering files in the right order sets an important requirement for the data and workflow management tools of CMS. Both event data and block data products can be dropped on input, i.e. ignored for the further processing if there are unsolvable conflicts in schema changes or memory problems during merge.

9. Schema Evolution

Another major item before data taking is to deploy and commission ROOT schema evolution in order to support old data being compatible with new software. We will collect many years of data, and we cannot reprocess all of the data taken thus far for every release. Even in the beginning there will be a rapid release cycle that only an ever decreasing fraction of the data will be able to keep up with. Schema evolution is the palative for this situation. Schema evolution will likely penalize I/O 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 all of the data taken. In other words schema evolution should be used as an insurance policy not as common practice.

10. EDM File Read Back

The organization of the data written into a file has a large effect on the performance of reading it back. Since our data files are written once and read many times, the general rule of thumb for CMS is that files should be optimized for read access, even if this will cost performance in the writer or the work load management system. As an example of the later we show what the read pattern can be after a write that was made in event ID order vs. one that was written by the merging system without regard to ordering the fragment files according to event ID. On the x axis is the read request number, and on the y axis is the offset in bytes into the file. You can see that in the unordered plot the end of the file is read first. The workload management system performance is decreased by having to wait for all of the fragment jobs to finish before it can do the ordered merge but the resulting read of the ordered file is faster. Another feature to point out in the ordered plot is the two lines with different slopes. The first line happens quickly at the beginning of the job, and is caused by the read done to fast clone selected branches from input into the output file. The second line represents 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.

11. Further I/O optimization

In an effort to improve the read patterns discussed in the previous section we have worked with the root team to add two new features to their I/O layer. Both are currently under test.

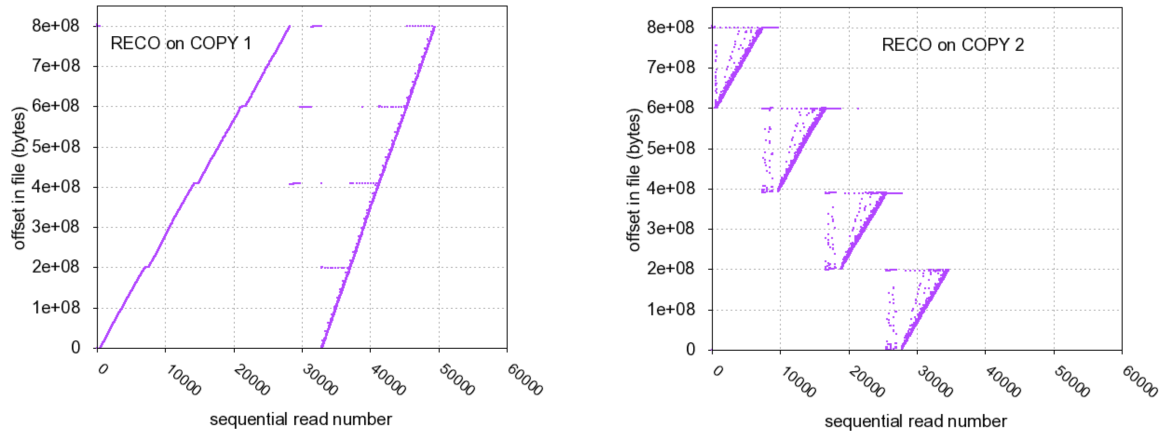

Figure 1. Offset in bytes vs. read request number for a merged file ordered by eventID (left) and a file merged without explicit ordering of events (right). More explanations in the text.

The first improvement involves a new feature of the tree cloner. The merging that we do to form the final file size in our production system, gives us an additional opportunity to reformat the data according to it's read pattern. Fitting the general CMS strategy of read optimization, we can specify and option to the tree cloner which will request that the files be merged and written back out in read- instead of write-order. The second improvement also involves the TTree class. In ROOT releases earlier than 5.22 all unfinished buffers are cached on the tree data structure itself. Due to the large number of branches used in CMS, this results in large memory allocations each time a new tree gets opened. The second added functionality allows CMS writing applications to flush these buffers to the appropriate branches on the tree just before closing the output files.

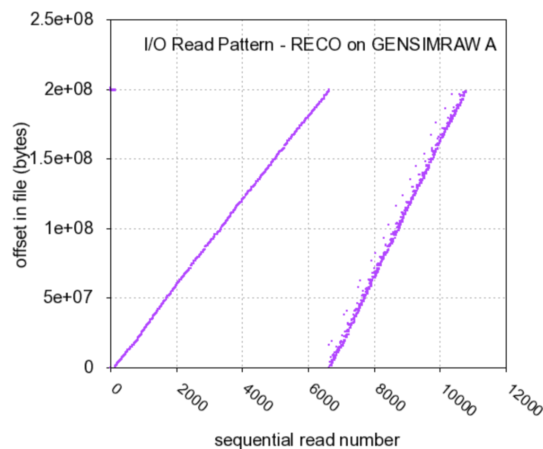

Figure 2. Offset in bytes vs. read request number for a merged file written in read-order. More explanations in the text.

12. Summary

The CMS strategy of using the same model for both serially processed, raw and reconstruction data, and direct access, analysis data has been challenging. Consequences of this are that complex transient behavior has been captured in the persistent representation. This bloats the size and complexity of types stored which negatively impacts perofrmance. However CMS physicist users like this model. They appreciate the simple user interface, the ease of defining dictionaries, the direct root access, and the automatic provenance tracking. This model has wide adoption within CMS.

13. References

[1] Rene Brun and Fons Rademakers, ROOT - An Object Oriented Data Analysis Framework, Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996, Nucl. Inst. & Meth. in Phys. Res. A 389 (1997) 81-86.

[2] C D Jones et al., The New CMS Event Data Model and Framework, *"Proc. CHEP 2006"*.

[3] R Wilkinson, B Hegner, C D Jones, Using Python For Job Configuration in CMS, *"Proc. CHEP 2009"*.

[4] F Fabozzi et al., Physics Analysis Tools for the CMS experiment at LHC, IEEE Trans.Nucl.Sci.55:3539-3543,2008.

[5] D Kovalskyi et al., Fireworks: A Physics Event Display for CMS, *"Proc. CHEP 2009"*.

[6] R Wilkinson, B Hegner , Usage of the Python Programming Language in the CMS Experiment, *"Proc. CHEP 2009"*.

[7] C D Jones, File Level Provenance Tracking in CMS, *"Proc. CHEP 2009"*.

[8] G Petrucciani et al, PAT: the CMS Physics Analysis Toolkit, *"Proc. CHEP 2009"*