CDF Event Data Management Using ROOT

Robert D. Kennedy
14 June 2001
ROOT Users Workshop 2001

Event Data Model: Manages reading/writing of events to/from data files, and manages access to objects in an event.

Data Handling System: Manages file repository, delivers files.

Framework: Configures, directs execution of software modules.

---

1) CDF EDM and ROOT: Brief Overview

2) Current Event Model: from which we are now evolving...

3) Event Data Flow: an application of fast block I/O...

4) ROOT V2 and CDF Offline: a little of our experience...

5) ROOT V3 and CDF Offline: initial impressions...

6) Requests of the ROOT Team: one operational issue, plus...

---

http://www-cdf.fnal.gov/upgrades/computing/projects/edm/edm.html

"The CDF Run II Event Data Model", talk and paper for CHEP 2000,
http://chep2000.pd.infn.it/abs/abs_c201.htm

CDF EDM and ROOT: Brief Overview

☞ CDF Offline uses ISO Standard C++. (Kai, v4)

Some legacy software in C and F77 is still used and supported. Most containers are STL, STL-based, or at least STL-oriented.

☞ Overall design driven by OOP, good physical design principles, reproducibility, ...

Use of forward declarations encouraged where possible. Objects are write-locked with history once in the event. Many classes....

☞ ROOT is used (in EDM context) solely as the implementation of event data storage.

Goal: separate storage solution details from event reconstruction and physics analysis algorithms, as much as possible. Data browsing and interaction are secondary long-term goals.

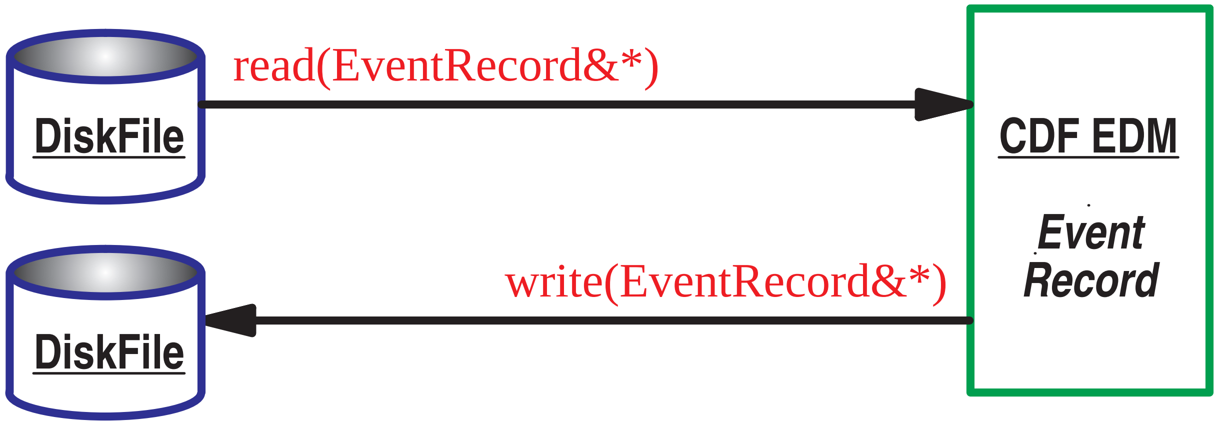
☞ Classes derived from *StorableObject* can be in the *EventRecord. StreamableObject*s have specific methods that allow them to be saved in event too.

StorableObject is derived from *TObject*, adds some protocol.
EventRecord is a collection of smart pointers to StorableObjects.
*StreamableObject*s must be contained by a StorableObject to be in the EventRecord: small concrete classes, items in a container.

☞ Offline uses all-static CDF library builds.

No CDF Infrastructure to support version control of shared libs. Tricks used to force linking of all known ROOT I/O dictionaries.

One Aspect of Current Event Model



SeqRoot format on disk
1 TTree "Sequential"
1 TBranch "Sequential"

↔

Class ***EventRecord***
In-Memory "format" =
Searchable Bag of
const "smart" StorableObject*

☞ Priority: Entire event through reconstruction.

All event data accessed in event reconstruction, so limited benefit to using multiple branches. "Seq(uential)Root" is an adequate first model from which we are evolving, *while taking data.*

☞ Add-on (F. Ratnikov): Event random access.

Using a separate branch of event selection criteria, we now have random access to events selected by (run,event) and so on.

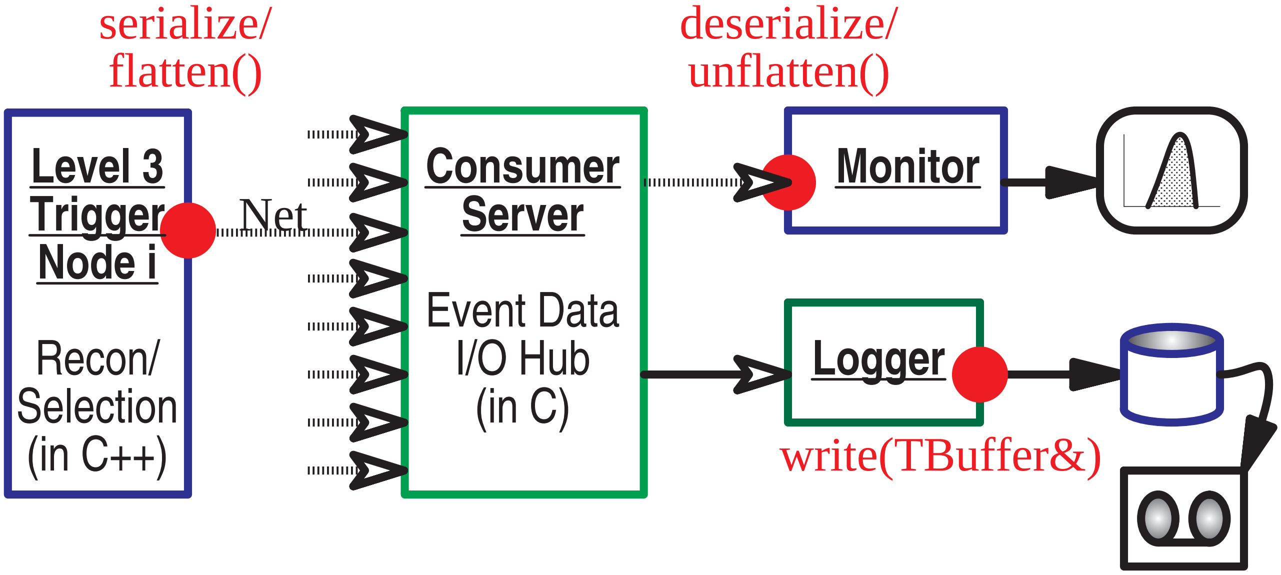
☞ Issue: I/O limited by "ambitious" Streamers

Analyses must read all event data. Some classes have CPU-intensive input methods that must be run. "Slow end-user I/O".

☞ Future: Multiple branches/trees.

**Multiple user-defined classes per branch, mapped by EDM.
Additional benefits: enables mixed block/object I/O (speed)**

Simplified On-Line Event Data Flow



EventRecord containing StorableObjects (SeqRoot) → Message Block containing a flattened event (in a TBuffer) → EventRecord containing StorableObjects (SeqRoot)

☞ **Selected events are "flattened" by L3 farm node for net tranfer, using Event's Streamer().**
Some descriptive data is copied into net message header to steer event through the I/O Hub to various requesting consumers.

☞ **Monitor "unflattens" message block, restoring original EventRecord.** **Using EventRecord::Streamer()**

☞ **Logger writes message block to disk in same format as write(Event) using minimal CPU.**
write(Event) = Event::Streamer(TBuffer) + write(TBuffer)

☞ **If event contents are not queried/altered by process, block I/O is a much faster alternative.**
Example: 7.6 vs 53 MB/cpu-sec for object vs block read, Linux, reconstruction output, 850MHz Pentium III, CPU 98/35% used.

ROOT V2 and CDF Offline

☞ CDF ROOT V2-based I/O is *SUCCESSFUL.*

Online is writing data at target rates. Common rate is 22 MB/sec on a 400 MHz R12000 IRIX machine. Production is processing that data on reconstruction farms. Users can read/write the reconstructed data. Still, there's room for improvement, from both the CDF side and from the ROOT side:

☞ Rootcint: limited preprocessor, C++ support.

Ability to use alternative preprocessor will help. Rootcint can core dump which our build system does not yet detect. CDF Offline code has many #ifndef __CINT__, accumulated over time. Incomplete support for user-defined classes, and past schema evolution recommendations, led us to write our own Streamer()s.

☞ Forward declarations break Rootcint model

Forward declarations in class headers can cause the ROOT generated dictionary code to not compile. Must use new feature #pragma extra_include in linkdef files to add #include of some headers into dict code. Template parameter constraints and improved decoupling in CDF Offline are being held up by effort required to refit CDF Offline with #pragma extra_includes.

☞ ROOT I/O: basic data types, namespaces

No "native" TBuffer support for std::string, bool, std::complex, signed char (unless you build your code with a flag), nor all STL containers of user-defined classes. No namespaces in our storable classes - supported data could not use namespaces when written.

ROOT V3 and CDF Offline

☞ Our Priorities: Multi-Branch, V3 Mainstream

Our first goal in going to ROOT v3.01 is to get back into the ROOT release mainstream. We have been using FNAL's v2.26. *Our top priority* is to implement multi-branch events, which requires overcoming little CDF EDM-ROOT model mismatches.

☞ CDF: little experience so far with ROOT V3

Philippe Canal and I have looked at building CDF Offline with ROOT v3.01, using some/none new features, respectively. No severe technical problems found (a CDF int/long mismatch), but our code would still have to be adapted to changes in V3.

☞ Automatic Schema Evolution

We write all our StorableObject streamers by hand, and most are stable now. So, there is limited immediate benefit relative to the cost of refitting existing Streamers. Selective adaptation....

☞ Fully Self-Describing Datafiles

If achieved, this would alleviate some CDF issues since missing object dicts would not be an error. As with above, what do we need to do to our Storable, Streamable Object classes to enable full self-description? Some retrofitting of code... how much?

☞ Our plan: Upgrade to v3.01 "slowly"

First, v3.01 with no new features. Study how to adapt our code to utilitize new features and review existing recommendations/doc. Adiabatically adopt new features, gain experience, then complete.

Requests of the ROOT Team

☞ Operational Issue #1: Support of Block I/O.

Recent TLeaf virtuality change and a low-level data format change in TBuffer cost CDF significant effort. CDF needs stability in the block I/O mechanism for the sake of CDF Level3/ Production I/O rate capabilities. We need in any ROOT release:

write(Event) = Event::Streamer(TBuffer) + write(TBuffer)
read(Event) = read(Cdf's TBuffer) + Event::Streamer(TBuffer)

Support for past intermediate "formats" in a TBuffer is desirable, but not a CDF requirement (unlike object I/O).

☞ Continue collaboration with Philippe Canal

Adapting CDF designs to ROOT models aided by his expertise.

☞ We request fine-tuning of error-handling.

We want to ignore errors due to missing dictionary entries, letting exes link to only those object dicts they actually use. Now, this is an error per affected object, which costs CPU. We resort to linking all known object dicts, which increases coupling.

☞ Consider managing dictionary code for I/O and non-I/O functions separately.

Breaks the static linking of I/O and non-I/O pieces which *greatly* reduces the need for "#pragma extra_include", other CDF issues.

☞ Purify: ROOT classes frequently cited

Most purify reports from CDF exes now tied to ROOT classes. None are serious, most are purify mistakes. Still, much noise....