SCHEMA EVOLUTION AND THE ATLAS EVENT STORE

D. Malon, P. van Gemmeren, Argonne National Laboratory, Argonne, IL 60439, USA
M. Nowak, Brookhaven National Laboratory, Upton, NY 11973, USA
A. Schaffer, LAL et Univ. de Paris-Sud 11, Orsay, France

Abstract

The ATLAS event data model will almost certainly change over time. ATLAS must retain the ability to read both old and new data after such a change, regulate the introduction of such changes, minimize the need to run massive data conversion jobs when such changes are introduced, and maintain the machinery to support data conversions when they are unavoidable. In database literature, such changes to the layout of persistent data structures are known as schema evolution. This paper describes the approach being taken by ATLAS to deploy an infrastructure that will robustly support schema evolution.

OVERVIEW

The ATLAS software framework Athena, based upon Gaudi [1], supports a clear separation between the transient objects manipulated by user-defined algorithms and persistent objects written out. This "conversion" layer incorporates a converter specific to each type read and written. For the experiment's event data model (EDM), ATLAS uses POOL/ROOT as its storage technology. In practice, while the EDM has been under development, data have been written via generic converters that employ automatic streamer generation capabilities provided by POOL/ROOT, wherein streamers are derived from transient class descriptions generated with the SEAL Reflection dictionary [3]. This approach has spared developers a great deal of effort—the "converter" layer is auto-generated—but it couples transient class definitions to persistent representations. Over the past few years, ATLAS has largely converged upon an initial definition of its event data model [2] from event generation through simulation to reconstruction and analysis objects, and currently is on its second iteration. In order to manage and oversee EDM evolution, ATLAS has put into place an Event (content) Management Board (EMB). The database group has developed a model that supports quite general schema evolution, described in the following sections.

The overall schema evolution strategy introduces an intermediate state representation layer between transient objects and persistent storage. This allows the transient EDM to evolve in a managed fashion, because legacy data impose no restrictions on transient classes seen by users—legacy data are coupled only to legacy state representations. Decoupling the persistent model from the transient serves a second purpose: it allows control and optimization of persistent event data organization and storage. One can design the persistent store, rather than treating it as a core dump of whatever was in transient memory when events were written. The approach introduces an extra copy step in memory (transient←→state representation←→persistence), but measurements show a net gain in speed and disk space. For example, the current ATLAS transient EDM cannot take advantage of a ROOT streaming mode known as "split mode" (more on this later), which would improve the overall performance, because the model employs collections of pointers to objects and other constructs that preclude split mode in ROOT. Further, employing simple persistent state objects would better exploit ROOT storage optimization capabilities, and would allow further compression by utilizing explicit knowledge of the data model.

Because ATLAS already has a large investment in data written without an intermediate state representation layer, the plan for introduction of schema evolution capabilities provides a means to maintain readability of existing data written with purely generic streaming technologies.

The Gaudi/Athena conversion layer notwithstanding, the use of generic streaming has coupled the persistent and transient models in terms of type names. For example, once a transient object of type Track has been streamed as a persistent object of type Track, it can in general be read back only into a transient Track object. (The exceptions are few, but important, and ATLAS takes advantage of them.) If the Track class changes in only minor ways, then ROOT's automatic schema evolution support can cope with the changes; however, this generic streaming fails when the transient class evolves beyond the capabilities of ROOT's automatic schema evolution support. We describe below how the ATLAS schema evolution strategy will address access to such legacy data.

TOWARD A TRANSIENT/PERSISTENT DATA MODEL SEPARATION

In order to allow flexible evolution of the transient event data model, ATLAS is introducing an intermediate state model for its EDM. This intermediate state will be written and read with generic ROOT streamers derived from class dictionary descriptions, taking advantage of performance improvements of split mode streaming. Subsequent evolution will be handled in the Athena conversion layer, which will manage the persistent-transient transformations. In the case where one wants to be able to read old data written without an intermediate layer and the transient type has evolved beyond automatic schema evolution capabilities, ROOT custom streamers

will be introduced to stream in this old data. The following sections describe the approach in a sequence of stages of increasing complexity: first, how persistence is handled and intermediate state representations are used in the absence of legacy data; second, how schema evolution is supported by this state representation model; third, how ATLAS expects to be able to handle legacy data--data written without an intermediate state representation—once this transient/persistent separation strategy has been deployed.

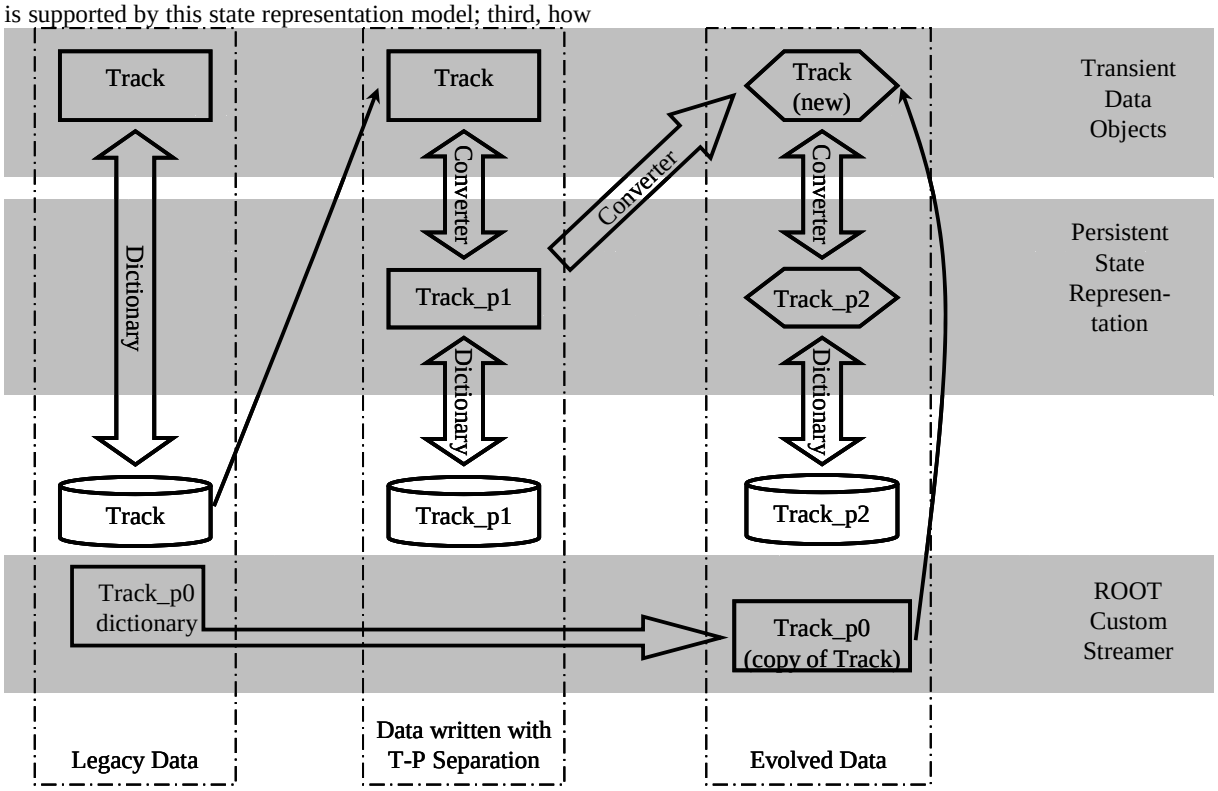

Figure 1: Schema evolution as intermediate state representations are introduced, maintaining readability of legacy data

Basic Model (No Legacy Data)

In the simplest case, the model is as follows. For a transient class such as Track, introduce a representation of Track's state, say, Track_p1. On output, a converter fills Track_p1 with the state of Track, and POOL/ROOT streamers, in general automatically generated, persistify Track_p1. Track_p1 is designed explicitly with the eventual persistent layout and potential storage and I/O performance optimizations in mind. On input, POOL/ROOT streamers fill Track_p1, and the converter builds a transient Track from Track_p1.

Basic Model: Schema Evolution (No Legacy Data)

When the definition of the transient Track changes, a new state object, Track_p2, is introduced. All new Tracks are handled as in the basic model, with I/O passing through the new intermediate state object Track_p2. When old data are read, POOL/ROOT successfully fills state objects of type Track_p1—the definition of Track_p1 has not changed. A new converter, though, is introduced, one that can build new transient Tracks from Track_p1 objects. This converter may be hand coded—after all, the changes to Track could be quite complex—but no restrictions on changes to Track are imposed by old data, since old data are coupled only to a state class, Track_p1, that can be preserved unchanged, and that physicists never see. The new Track_p1-to-Track converter is invoked only on input—new data are written via the Track_p2 path.

Reading Legacy Data

The subtlest issues arise in preserving the readability of data already written without benefit of an intermediate state representation while still allowing schema evolution. While the principles are straightforward, understanding how the approach is implemented in practice requires some explanation of ROOT streamers and how they work. To aid in this understanding, we will describe in the

following sections the various ways to stream objects with ROOT, the planned use of custom ROOT streamers, and finally some of the transformations to be employed in the Athena conversion layer. First, though, we outline the legacy data support strategy.

The first step in retaining readability of legacy data is to introduce a new type, Track_p0, which is in fact simply a clone of the transient Track class, or more precisely, of the Track class's data members (its state)—Track's methods are irrelevant. If, when an "old" Track is encountered as input, it can be streamed into the isomorphic class Track_p0, then a Gaudi/Athena converter can build an evolved Track from Track_p0. Coercing the underlying technology to build a Track_p0 when it encounters a Track (a Foo when it finds a Bar) turns out to be non-trivial, and sometimes impossible, given current POOL/ROOT limitations.

ROOT Object Data Storage

ROOT writes objects by streaming them to output buffers in one of two modes: object mode and split mode. In object mode, the data members of each object are converted into a stream of bytes written to a single buffer, while in split mode each data member is written into its own output buffer. In general, split mode provides more efficient read access in two ways: first, the generic buffer compression produces smaller output files, because buffer contents are more regular; second, at the ROOT level individual data member buffers may be read in separately when only selected data members are needed (e.g., for histogramming).

In the ATLAS EDM, one typically encounters vectors of pointers to objects, rather than objects by value. When the transient EDM is streamed directly, the ROOT-level storage layer can only utilize object mode rather than split mode, because pointers may refer to polymorphic types, which are difficult to split. With the separation of the transient and persistent models, ATLAS will be able to ensure that its persistent model can take advantage of the split mode.

ROOT Custom Streamers

A ROOT custom streamer is a user-defined procedure, invoked for specialized streaming of objects of a particular type. A custom streamer can fully control the streaming of data members, and can execute code that transforms their values as well. To read legacy data, ATLAS will use custom streamers for a somewhat different purpose: to fill an object different than but equivalent to the originally streamed object. Consider, for example, the case that a Track has been directly streamed to persistent storage without passing through an intermediate state representation, and that the definition of Track has since changed beyond the limited capabilities of ROOT automatic schema evolution support. When such a Track is encountered as input, a custom Track streamer will be called, and will build an object of an altogether different but isomorphic class—Track_p0. Note that this custom streamer need not be written from scratch; in fact, it can simply invoke a streamer that was automatically generated for Track_p0 from Track_p0's dictionary, because of the isomorphism between Track and Track_p0.

In fact, there can be more than one version of the Track class in any ROOT file. They are distinguished by a checksum of the persistent shape. The custom streamer must use the checksum to select the correct streamer for reading. Thus a custom streamer can be used for multiple iterations of an evolving Track class, where Track is always written in the standard way using the generated streamer for Track and is read back as Track_p0, Track_p1, etc. Note that at any one time the reflection dictionary contains all the needed versions of Track: Track, Track_p0, Track_p1, etc. However, as we see next, this capability breaks down for split mode.

ROOT Custom Streamers and Split Mode

ROOT custom streamers might appear at first to be a good solution for treating schema evolution: it is sufficient to implement them only when changes actually occur, without any preparation in advance, they work transparently to the end user, and they do not propagate class schema changes to their embedding types (see the chain reaction effect described below). They have been proven to treat well CLHEP schema changes when going from CLHEP v1.8 to v1.9.

Unfortunately, ROOT custom streamers can be used only for data written in object (non-split) mode. The current ATLAS EDM is sufficiently complex that very little of it has been successfully split by ROOT, so ATLAS will be able to use the custom streamer strategy described above. Custom streamers will not, however, provide a long-term solution: ATLAS needs to begin to take advantage of the improved performance of split mode. The use of custom streamers must therefore be limited to only the first step in the sequence of an evolving transient schema, when one needs to read old data corresponding to transient objects that were streamed directly and unsplit.

Data Model Transition

It is natural to ask at what point one should introduce an intermediate state object like Track_p0 that is, essentially, simply the transient Track class renamed. It is needed, strictly speaking, only when the transient class has evolved beyond the capabilities of automatic schema evolution. Introducing it at an earlier stage, however, offers the advantage that data written after its introduction will be coupled only to Track_p0, rather than to (the equivalent) Track, freeing Track for schema evolution without a later need to provide custom streamers. ATLAS hopes to move to a genuine intermediate state representation model—one in which the state representations are defined explicitly with the intention of organizing persistent data for improved performance—in a time frame that will render this question moot.

Transient/Persistent Data Model Separation

As the ATLAS EDM evolves, custom ROOT streamers will maintain the readability of existing data where the transient types have been streamed out directly. As we have seen, this involves as a first step the creation of a persistent state class for the old version of the transient class. This will be the starting point of fully separated the transient and persistent classes. This will allow the use of ROOT split mode, and one can take advantage of the opportunity to explicitly compress the persistent model. For example, four-vectors can be saved as floats rather than doubles, allowing for required precision for the mass. This separation will accommodate large-scale redesign of the transient model while retaining the capability to read old data.

Providing an intermediate state object comes at a cost: it requires more effort on the part of developers. In early stages of software development, while classes are continually changing, developers should not need to think unduly about how to write and read their objects. Before producing petabytes of data, though, it makes sense to think about and perhaps even design the layout of events in the persistent store, rather than simply relying upon a core dump of the state of the transient whiteboard.

An Athena converter will be responsible for the transformation between the persistent and transient shapes. In Athena, a user algorithm creates one, or more, top-level objects and registers them in a (event) data store in memory. These objects are available to other algorithms and may be streamed out by converters via POOL. For each object written by POOL a token, or persistent reference, is returned and saved in a DataHeader object, which in turn is streamed out with each set of written objects. On input a DataHeader is first read in, and then its set of tokens determines which objects may be subsequently read in. Note that the token contains both information regarding an object's placement and its type in terms of a unique id.

Typically, the top-level object written by a converter is a collection of objects, which may in turn contain other embedded objects and possibly smart pointers to objects in other collections written out separately. These smart pointers are in ATLAS known as ElementLinks (or DataLinks, for objects not in a container). On output, the converters will copy the data members of the transient objects into persistent objects, which have entries in the Reflection dictionary. The top level persistent class is assigned a GUID when creating its dictionary entry, and this must be changed whenever the persistent class evolves beyond the automatic schema capabilities of ROOT. Then on read back the converter can use the GUID in the token to choose which persistent class to stream in and create the transient class.

One delicate point is that a change in an embedded class requiring manual schema evolution will force all of the top-level collections using this class to move to a new version – a sort of chain reaction[*]. This will certainly have the tendency to simplify the persistent model. The storage of the inter-object references, i.e., ElementLinks, does not pose any problem since they work as references inside the Athena framework and not at the persistence level. Infrastructure to facilitate developers' work, including templated converters and code and dictionary entry generation, is provided by the database group.

SUMMARY AND CURRENT STATUS

ATLAS has developed a strategy and an infrastructure to support quite general schema evolution of its event data model. Exploration of realistic test cases has demonstrated a 20-30% net increase in read performance. An Event Management Board has been put into place to oversee the migration, and ATLAS is currently working on transforming its event data model over the next several months, starting with the simulation and raw data, in preparation for computing system commissioning in 2006, and data-taking in 2007.

ACKNOWLEDGMENT

The submitted manuscript has been created by The University of Chicago as Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. W-31-109-Eng-38. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.

REFERENCES

[1] Gaudi Project Portal: http://proj-gaudi.web.cern.ch/proj-gaudi/welcome.html

[2] Final report of the ATLAS AOD/ESD Definition Task Force. ATL-SOFT-2004-006; http://doc.cern.ch//archive/electronic/cern/others/atlnot/Note/soft/soft-2004-006.pdf.

[3] Currently ATLAS is using the dictionary services of Reflection, but will be migrating to Reflex in the near term. See the SEAL portal for more details: http://seal.web.cern.ch/seal/

[*] A ROOT custom streamer is an ideal solution for the case where an embedded class used in a number of places has changed beyond automatic schema evolution, but one that only works today for classes in non-split mode.