Explicit state representation and the ATLAS event data model: theory and practice

Marcin Nowak[3], David Malon[1], Peter van Gemmeren[1], Arthur Schaffer[4], Scott Snyder[3], Sebastien Binet[2], Kyle Cranmer[3]

[1]Argonne National Laboratory, Argonne, Illinois 60439, USA
[2]Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA
[3]Brookhaven National Laboratory, Upton, NY 11973-5000, USA
[4]Laboratoire de l'Accelérateur Linéaire, Orsay, France

E-mail: Marcin.Nowak@cern.ch

Abstract. In anticipation of data taking, ATLAS has undertaken a program of work to develop an explicit state representation of the experiment's complex transient event data model. This effort has provided both an opportunity to consider explicitly the structure, organization, and content of the ATLAS persistent event store before writing tens of petabytes of data (replacing simple streaming, which uses the persistent store as a core dump of transient memory), and a locus for support of event data model evolution, including significant refactoring, beyond the automatic schema evolution capabilities of underlying persistence technologies. ATLAS has encountered the need for such non-trivial schema evolution on several occasions already.

This paper describes the state representation strategy (transient/persistent separation) and its implementation, including both the payoffs that ATLAS has seen (significant and sometimes surprising space and performance improvements, the extra layer notwithstanding, and extremely general schema evolution support) and the costs (additional and relatively pervasive additional infrastructure development and maintenance). The paper further discusses how these costs are mitigated, and how ATLAS is able to implement this strategy without losing the ability to take advantage of the (improving!) automatic schema evolution capabilities of underlying technology layers when appropriate.

Implications of state representations for direct ROOT browsability, and current strategies for associating physics analysis views with such state representations, are also described.

1. Introduction

The work described in this paper was done within the scope of the ATLAS experiment [1] offline software framework Athena [2]. One of the important functions of the framework is data management, which includes keeping data objects in a transient memory cache and providing storage and retrieval of these objects with the help of an independent external persistency technology. Athena allows using different persistency solutions simultaneously, but the dominant technology for event data storage is currently POOL [3] storage in ROOT [4] file format.

During the ATLAS software development phase, the problem of schema evolution of the event data model (EDM) and the ability to read old data was often deferred as something to be addressed when the EDM stabilizes. Much hope was put on the underlying persistency solutions to be able to address that issue. However, initial studies performed with POOL and ROOT showed that their ability to handle schema evolution was limited to certain cases and did not cover all situations that ATLAS foresaw would arise in the lifetime of the experiment. This discovery triggered research in the direction of how Athena could handle the issue internally. The most promising approach was to introduce a separate persistent state representation for the EDM data objects.

Shortly after the prototype solution was put in place, it was discovered that the intermediate state representation offered much-improved I/O performance. At this time, when discussing the separate state representation, the argument for performance improvements is probably as strong as the one for supporting schema evolution.

2. General issue of Schema Evolution

The data model of an experiment that will run for many years is bound to undergo many modifications. These modifications are commonly referred to as schema evolution. As long as only the transient application view changes, there is no noticeable effect. Changes to definitions of classes for which data were stored in files, though, introduce incompatibilities between the transient and persistent world, to the point where the ability to read old data with the new software is lost. Support for schema evolution means introducing mechanisms that prevent that from happening.

It should be noted that while data model changes can take many forms, including adding or removing classes, the changes to a single class are usually made without changing the class name. The ability to freely evolve class shapes without the need to change their names and without losing the ability to read old data was one of the requirements that arose continually in discussions within the ATLAS community.

2.1. Object schema handling in POOL and ROOT

POOL and ROOT use data dictionaries to describe object schema. Every type known to the application is described in detail in the dictionary under an entry corresponding to the type name. These descriptions are used when writing objects to files – a copy of the dictionary information is stored in the file as well. This information is used when reading the object later. When the current schema in the dictionary matches the one stored in the file, there is no problem. However, when the two schemata are different, some action needs to be taken to transfer the old object data into the new schema.

2.2. Support for schema evolution in ROOT

ROOT provides two mechanisms to support schema evolution:

- Automatic schema evolution
- Custom streamers

Automatic schema evolution is a ROOT feature that works automatically, without the need (or possibility) of any external action. ROOT tries to match the old schema information stored in the file with the new schema in the transient data dictionary as well as it can. The mechanism has a very good chance of coping successfully with a range of simple schema modifications – like adding new data members to a class. More complicated changes usually lead to unpredictable application behaviour.

Custom streamers are a completely different mechanism, relying on the user writing a procedure to read an object. They can be used to force ROOT to use a different streamer to read the old data – one that would match the old object schema – and then perform a data copy. See [6] for a detailed description of the process.

Custom streamers were initially seen as a good candidate to provide support for schema evolution, but at the time they were tested they were working only with data written in ROOT “streamed” mode [4]. As most performance improvements noticed came from using the other writing mode, called

"split" mode, the streamers were considered only a partial solution to be used as backup where possible.

It should be noted that since the original evaluation, ATLAS developers have been engaged in implementing improved ROOT streamers that should work with data written in either ROOT mode, streamed or split.

2.3. Introducing schema evolution support in Athena

Finding no satisfactory support for schema evolution in the POOL and ROOT persistency layer, we attempted to provide it in Athena itself. From previous attempts, we knew that to be successful, the solution had to meet two conditions:

- it should prevent ROOT from trying to perform automatic schema evolution in cases where it would fail, and
- it should maintain old class schemata for backward compatibility.

2.3.1. *Class schema versioning.* The most straightforward way to satisfy the requirements was to introduce class schema versioning. That feature is not available in the data dictionaries used by POOL and ROOT (neither is it something directly supported by C++), so it had to be implemented "by hand," by introducing a version number in the class name. By convention a class name like `Track` has its persistent state represented by `Track_p1` and later by `Track_p2` and so on. All versions of the class schema are included in the dictionary and kept as long as there is a need for backward compatibility with a given version. A new schema version is added when changes to the current class exceed ROOT's ability to do automatic schema matching, thus preventing ROOT failures and satisfying the first condition.

2.3.2. *Persistent state representation*. Due to the way ROOT I/O works, in order to be able to read an object using its schema version _pN, it had to be written as an object of exactly that type in the first place (exceptions to that rule are ROOT custom streamers, but they were not working in all situations, as explained earlier). Given the principal requirement that the class on the transient side is always called simply Track, for example, and that we need to write it out as Track_pN in order to enable schema evolution support, we need to perform conversion and copy all relevant data from Track to Track_pN. The need to maintain the conversion functions and a separate class shape for persistency is clearly an additional effort for the developers, but the benefits go beyond the schema evolution support. The persistent state can be much simpler than its transient counterpart and may be handled much better by the persistency layer, leading to surprising performance improvements, described in section 5.

3. Implementing persistent state representation for ATLAS EDM

3.1. Framework for Transient/Persistent data model separation

A persistent state representation that meets schema evolution requirements described in 2.2. can be implemented in many different ways, as preferred by the developer. However, in order to provide guidance, help avoid pitfalls, minimize effort, and maintain uniform code structure, ATLAS has developed a standard approach to implementing TP conversion based on the TP separation framework [7]. The framework consists of a set of rules regarding class creation, and a collection of support classes. These classes define uniform interfaces and attempt to provide as much automatic functionality as possible, ideally leaving the developer to implement only the actual transient/persistent conversion methods for their types.

3.2. Persistent state representation classes

Persistent state representation classes, or in short persistent classes, need to be defined for every transient class that requires schema evolution support. The exact shape of the persistent class is again

something that the developer needs to decide. In many cases the right shape is not obvious, so it is preferable that the person creating the persistent representation be the same one who designed the transient class, or at least someone who has a good understanding of it.
The process of creating the persistent representation usually begins with inspecting all data members of the transient class to determine if they need to be persistified. The process is recursive and one needs to inspect in turn all base class components, contained objects and objects referenced by pointers. Each of these can be treated in different way:

- Flattening – this is an approach wherein relevant data values are extracted from embedded structures and stored all in one object "on the same level." It offers considerable flexibility where schema evolution is concerned, and good I/O performance (due to simpler data model), but may be prohibitive in maintenance if the same object flattened in too many places needs to be changed.
- Mirroring the transient model – in this approach the persistent data model mirrors (to some extent) the structure of the transient data model, introducing distinct persistent representation classes for the embedded types as well. The advantage of this approach is that these classes and converters, once defined, can be used in many places. Creating new persistent representations and changing schema of existing ones is then easier.

The main rule when creating persistent classes is always simplicity. Simple classes are better handled by the persistency layers, easier to maintain, and easier to evolve. Pitfalls to avoid while creating persistent classes are:

- Unnecessary methods – the persistent class needs only a default constructor (for POOL);
- Complicated class definitions – the definitions need to go to the dictionary – overly complex definitions can be unwieldy to express there;
- Polymorphism – all classes should be concrete data structures without virtual methods;
- Inheritance and embedding – they introduce overhead and the objects should be either flattened or replaced by persistent references (persistent references are described in 3.5.);
- Pointers – especially pointers to polymorphic types – can force ROOT to use streamed mode for writing, and thereby seriously impact I/O performance. They should be replaced by persistent references;
- Dependencies on other transient types – every direct dependency on a transient (non-TP-separated) type introduces the danger of being affected by transient data model changes, without the possibility of recovery. Ideally all transient type references should be eliminated or replaced by persistent equivalents.

In general, the resulting persistent classes are very small and simple. They rarely require more than the implementation (.h) file. In Athena code packaging structure they are located in a separate <Package>TPCnv package and they are included in the data dictionary that is created for each such package.

3.3. Converters between transient and persistent representations

The TP separation framework performs conversion between transient and persistent representations with the help of the TP converters. The decision was taken to keep the persistent representations and their converters as disjoint classes in order to achieve full separation of the two data models. Each version of the persistent state representation has its own converter, versioned in a similar way as the persistent class. A Track class with Track_p1 representation would be by convention accompanied by TrackCnv_p1 converter.

The converters were, by initial design, transient classes so they did not need to be entered in the data dictionary, and unlike persistent representation types they could have a very complicated class structure. However, the implementation of direct ROOT access described in chapter 6 requires now their presence in the dictionary.

TP converters are implemented by specializing a base converter template provided by the framework. The template is parameterized by the transient type TRANS and a given version of the

persistent representation PERS. The templates define most of the utility methods used when invoking the converters – the developer has only to implement the two principal conversion methods:

- transToPers(const TRANS* t, PERS* p)
- persToTrans(const PERS*p, TRANS* t)

These conversion methods work on actual instances of the transient and persistent objects. Creation and handling of these objects is done automatically by the framework.

The task of the transToPers() method is to copy all persistent data members of basic type to the persistent object. Data members of complex types can be either flattened, or they can be in turn converted by invoking a subsequent TP converter. The invocation is not done using directly the transToPers() method defined by the developer, but instead by calling one of the utility methods defined by the converter template. That method returns a persistent reference that needs to be stored in place of the object. The reason for that is the cascade effect explained below.

3.4. Cascade effect of schema evolution in the early TP separation model

The early approach to TP separation assumed that the transient data model was mirrored almost exactly by the persistent data model, with the class names replaced by their versioned persistent types. This was leading to situations in which long chains of type dependencies existed on the persistent side, wherein one versioned class would contain (or point to) another versioned class and so on. Attempts to evolve schema of a class that was at the end of such a chain would by definition require changing its name due to introducing a new version. A change of name of that class would in turn constitute a schema change for the class one step earlier in the chain, and the effect would propagate throughout the entire chain.

The effort to perform such operation was unacceptable to many of the ATLAS package managers, and a different solution was requested. The new solution was based upon breaking the type dependency chains between object components by introducing typeless persistent references.

3.5. Persistent data model without type dependencies

Removing the C++ type dependencies between constituent object components was achieved by replacing all component relationships, including inheritance, embedding and pointer-based association, by typeless persistent references. The references consist only of a set of integers that have meaning to the TP converters. There is no longer a C++ type tie between the elements of the dependency chain, and every link of the chain becomes independent. Changing the schema of one class in the chain no longer affects other classes and so the developer is no longer constrained by the danger of creating a cascade of changes.

Breaking down the object structure into separate components had a strong impact on the I/O operations. Formerly, POOL/ROOT persistency was taking care of storing all object components by recursively traversing object structure, visiting base classes, handling embedded objects, and following pointers. After effectively decomposing objects into components, this became no longer possible. The TP separation framework needed to take over the responsibility for ensuring that all components are written out.

3.6. Top-level persistent objects

To provide a container for all the object components, a new category of persistent object was introduced – a top-level persistent object. Top-level persistent objects (TLP objects) contain vectors of component objects. Writing out the TLP object automatically ensures that all components are written out with it. The relationships between the components are kept in the typeless persistent references, which record in which vector and at what index a given component was stored. Placement of component objects within the vectors, and later reassembly of the transient objects, is done by the TP separation framework automatically.

Every component type, including base class components (also abstract ones), embedded types, and distinct specializations of polymorphic types, has its own storage vector in the TLP object. With a

complete decomposition of the source object, the component objects would contain only base types and persistent references.

The storage vectors are dynamic, allowing any number of component objects to be stored in them. This is an important feature when working with polymorphic types, where some specialization may or may not exist in a given data sample. It is also used to support schema evolution.

3.6.1. *Schema evolution support in top-level objects.* Every version of a persistent type has its own storage vector in the TLP object. When the TLP object is designed for the first time, all component types have one initial version. As the types evolve, new class versions are added together with new storage vectors for them. Adding a new data member to the TLP object in this manner is covered by ROOT automatic schema evolution, so the TLP object itself does not need to change its name to a new version. The same TLP object will support the new and the old data – when reading old data, only the vector of the old component type version will be filled out, the vector with the new representation will remain empty. When writing, the situation will be reversed – only the vector with the new type version will be filled out, and the old one will remain empty. Eventually, when the TLP object accumulates too many changes, or possibly when a change being introduced is very extensive, a new version of the TLP object itself can be introduced.

3.7. TP converters for top-level objects

TP converters for top-level objects (by association called top-level converters) are a more sophisticated version of the elemental TP converters. Their additional functionality is related to the task of storing object components into their corresponding storage vectors in TLP objects. To be able to do that efficiently, the top-level converters effectively own elemental TP converters for all the component types they need to handle. Each elemental TP converter is associated with one storage vector every time a new TLP object is created. The converter stores the results of TP conversion directly into the storage vector and reads them back from there in the reverse process.

3.7.1. *TP converters and the cascade effect.* It should be mentioned that the cascade effect of schema changes was also affecting the TP converters themselves, because they were templates parameterized according to the types they were handling. To prevent that from happening, an additional converter base class was introduced, parameterized only by the transient type. A pointer to that class can be used to reference all versions of TP converters for that transient type. A common API was possible because the actual persistent type is not exposed in the API – persistent objects are stored in the storage vectors internally.

3.7.2. *Converting polymorphic types.* The fact that the top-level converter knows all elemental TP converters allowed an elegant solution to the problem of converting polymorphic objects. When a polymorphic object is encountered, the top-level converter can be asked to find the right TP converter from the list of all TP converters that it knows about. The selected converter is referenced by a base class pointer in a manner similar to that which allowed avoiding the cascade effect – only this time the transient type used to parameterize the template is the common base type for the polymorphic objects family.

4. Role of AthenaPool converters in schema evolution

The Athena framework manages data objects at a certain level of granularity – each object is a separate entity with internal structure opaque to the framework. Such objects can be part of more complex data structures, but at that moment they lose their independence. Also, collections of objects (i.e., containers), even though they are a special case in some respects, are treated as a single entity when it comes to I/O operations. The general design of the framework assumes that the relationships between the objects are expressed in terms of smart pointers (links) instead of direct C++ constructs like pointers. This allows maintaining the relationship through the persistency phase and prevents

POOL and ROOT from pulling in the entire data store by following pointers when writing a single object.

Following the Gaudi [5] architecture, on which Athena is based, each of the object types that the framework can manage has a Gaudi-style I/O converter associated with it. The converters that Athena is using for the POOL storage technology are called AthenaPool converters. By default they are generated automatically and simply pass the object to the underlying POOL layer, returning a POOL Token. The converters also allow customizations and in principle any kind of transformation can be performed before writing the object out and after reading it back.

Tokens contain all information necessary to later retrieve objects from the persistent storage – including object type. Tokens of all the objects that were written out as part of the same event are gathered together in a DataHeader object.

4.1. Using Transient/Persistent conversion in Athena

The TP separation framework is mostly a standalone set of object transformation procedures – it does not depend upon external packages. To function in the Athena environment it needs to be correctly integrated. The most obvious place to accomplish this is in AthenaPool converters, which exist at the connection point between Athena transient object cache and the persistency layer.

A given AthenaPool converter is invoked for every object being written out or read back. It provides hooks that can be used by the developer to transform the input object to any type that is suitable for storage.

The TP separation framework was designed so that the single object passed to AthenaPool converter is treated as top-level object and converted into its top-level persistent representation. Each AthenaPool converter is associated with one top-level TP converter that is doing the transformation. The transient top-level object can be simple or complex (e.g. a multi-level container) – it does not affect the 1-to-1 relationship between AthenaPool converters or the way the TP conversion is done.

4.2. Supporting schema evolution in AthenaPool converters

AthenaPool converters work only with the top-level TP converters, so they are unaffected by schema evolution that does not require changes to the top-level TP converter version. For writing, AthenaPool is always using the newest version of the top-level TP converter. Reading old data on the other hand may require using older converter versions.

The essential part of the reading process is finding out which version of the persistent object is being read. The converter can get the information from the POOL Token string, which is the identifier of the object. The Token contains the class GUID (a unique identifier assigned by the developer when introducing the object to the data dictionary). Based upon the GUID, an object of the correct type is retrieved from POOL and the correct TP converter is used to produce the transient representation.

4.3. The special case of DataHeader schema evolution

The DataHeader in AthenaPool, among other responsibilities, serves as the Event entry point and stores the persistent address of all DataObjects in the form of Token strings. The DataHeader, just like the other DataObjects, is TP separated to allow schema evolution and optimize I/O performance. Here too, we use different class names (with '_p<N>) and POOL GUIDs to indicate the version.

Reading DataHeaders is different from accessing other DataObjects, because we begin without Tokens pointing to them. Instead, to ensure that the entry point to the Event can be found easily, the DataHeaders are stored in a POOL object collection in a POOL container with a well-known constant name ("POOLContainer_DataHeader"). The POOL collection iterator is used to create POOL tokens for all objects in the collection. Only from that point we can follow the standard reading procedure, detecting the actual version of the object by checking the class GUID in the Token.

Using a single POOL container to store DataHeaders restricts us to have only one version of persistent representation per file (i.e., we cannot append DataHeader_p<N> to a file with

DataHeader_p<N-1>). That restriction is not present for other DataObjects. So far the limitation has not posed any particular problems.

It should be mentioned that when following links between Events (by using back navigation [9]) the DataHeader will be accessed by a Token like all other DataObjects. That allows creation of cross-file connections between Events with different versions of DataHeader representations.

5. Performance

The original intention behind introducing a separate persistent state representation for DataObjects was primarily to add support for schema evolution. The main concern about the performance was that it should not seriously degrade – after all, the TP separation layer introduced additional conversion algorithms and memory allocations and copying, and we were not aware of any particular performance benefits we could obtain by doing that.

The results from the first prototypes, which were not using object decomposition (just straight data model mirroring), were already encouraging: the file size was reduced by up to 25% in some cases and the gains in performance from working with smaller files was covering the extra CPU usage required to do the TP conversion. However, it was only after introducing object decomposition and the use of top-level objects with vectors of simple structures contained by value that we noted surprising performance improvements.

5.1. Observed performance improvements

The I/O performance improvements strongly depend upon the transient data model structure. Simple data with large objects were already handled well by the persistency layer, so relative improvements were not large. Complex objects with pointers and small component parts profited most from the different state representation.

The TP separation of the ATLAS tracking data model improved the read speed by a factor of 4 over the reading of the directly written (not TP-separated) transient data model. In other places reading speed improvements up to 10 times were observed.

The biggest improvement achieved thus far was a factor 20 increase in reading speed of LArRawChannel data consisting of very large number (~183,000) of small objects (~20 bytes) referenced by pointers from a DataVector container.

The general tendency is that performance improvements were greatest in situations where the initial performance was poor. The changes led to more uniform reading speed across different parts of the data model. In absolute values we have achieved the desired read speed, on the order of 10MB/s.

The reduction in disk size remains typically between 20% and 40%. This is attributed mainly to reducing ROOT overheads related to type information, and improved buffer compression resulting from using simpler data structures.

5.2. ROOT writing modes

The reasons for such drastic improvements lay in the different ROOT writing modes. The streamed writing mode stores objects sequentially, recursively inspecting the object structure and introducing type information overhead on a per-object basis. In the split mode object attributes are treated like elements of a column-wise ntuple, and the type information is not replicated.

The overhead information present in the streamed format does not lead to much bigger data files, because it can be efficiently compressed. However the compression and the effort of processing the type information for every object is responsible for this mode being so much slower (at least this is the current understanding of the process in ATLAS).

ROOT is forced into streamed writing mode by the presence of pointers and polymorphic types in the data model. By eliminating these features from our persistent state representation, we enabled the possibility of using the much more efficient split writing mode.

5.3. Object pools in the TP converters

The additional cost of the memory allocations in the TP separation framework can consume about 10% to 20% of the CPU time needed to read in the objects. Since the transient model is a pointer model based on objects allocated on the heap, we have developed an “object pool” wherein objects can be allocated once per job and never deleted, if necessary reused event by event.

The use of object pools can be enabled on a per-converter basis, allowing focus on the most promising packages.

6. Accessing persistent state representation objects directly from ROOT

For purposes of end user physics analysis, it is convenient to be able to access event data directly with ROOT (without the Athena framework). It is of course possible to read the persistent form of the data from within ROOT - this simply requires loading the dictionaries for the persistent data classes (as well as a small amount of code to work around various ROOT deficiencies). However, one would really like to be able to work with the transient form of the data, without being exposed to the persistent representations with their potentially changing versioned class names. As an additional requirement, the transient data objects should also be usable with the built-in ROOT tools such as TTree::Draw.

The persistent data are stored using the ROOT TTree data structure. This can be thought of as a two-dimensional table, the rows of which are separate entries, and the columns of which are represented by TBranch objects. The TBranch objects are owned by the TTree; each one corresponds to a stored data object (or a piece of one). The TBranch class provides an interface that the standard ROOT tools use; several derived classes exist to support different ways in which the storage is organized. A TBranch maintains a memory buffer that holds the data object that it is managing; the method TBranch::GetEntry is used to seek to a given entry in the branch and read the object into the buffer.

To provide automatic TP conversion within ROOT, we introduced a new, Atlas-specific, class derived from TBranch: TBranchTPConvert. The memory buffer in this branch holds an instance of the transient data object. The branch also holds a reference to the branch of the persistent data object, as well as a reference to the transient-persistent converter. The GetEntry method of TBranchTPConvert is thus implemented by first forwarding the call to the persistent branch, then calling the conversion method of the TP converter to copy from the memory buffer of the persistent branch to that of the transient branch. All the TBranchTPConvert instances for a given tree are collected together into a new TTree, the ``transient" tree. The original, “persistent,” tree is made a friend of the transient tree. This means that a GetEntry call on the transient tree is automatically forwarded to the persistent tree, and that a branch lookup in the transient tree will search both the transient and persistent trees. In this way, the user need only deal with a single tree, and can use either the persistent or the transient data objects. Because TBranchTPConvert supplies the required TBranch interface (it actually derives from TBranchObject), all the built-in ROOT tools transparently work on the transient data.

A function is provided to automatically build the transient tree and its branches from the list of data objects contained in the DataHeader. A method is needed to find the transient class corresponding to a given persistent class, as well as the correct TP converter. Converter lookup is made possible by introducing a base TP converter template of the form T_TPCnv<T, P>, where T and P are the transient and persistent classes. Thus, given the name of the persistent class, one can find both the converter and the corresponding transient class by inspecting the list of available TP converters.

Special treatment is required to handle the object pools used by some of the transient-persistent converters, as the lifetimes of objects in separate branches are independent. The object pool was converted to use an arena-based memory allocator. In the offline reconstruction code, there is a single arena, used by all the object pools. However, for access from ROOT, a separate arena is associated with each transient branch. In this way, the object lifetimes may be managed independently. Further

special handling is needed to handle references between transient objects. These references are represented as a string key, which is resolved using information in the DataHeader.

Direct ROOT access to persistent state objects is implemented in the AthenaROOTAccess [8] package.

7. Conclusions

At the time of writing, the idea of separate persistent state representation for event data is widely accepted in the ATLAS offline software development community. It is considered the solution for schema evolution support from Athena release 14 onwards. In release 13 the event data model is already almost fully TP separated. Some classes already had their schema changed in the meantime, and the ability to read old data was tested. At the same time we are observing spectacular reading performance improvements. The overall performance that we expect to be able to achieve is 10-15 MB/sec for typical 2007 CPUs.

Implementation of the secondary data model and TP converters is undoubtedly an extra effort, so a lot of thought was devoted to minimize the amount of code that needs to be written and to make it easy to maintain. Additionally, the time spent on the analysis of the data model is not wasted – the analysis forces developers to think more carefully about the data they are writing out, instead of simply dumping the transient object state. Many errors were discovered during that phase, which also resulted in decreasing the disk size and improving performance.

As an important side effect of the schema evolution studies, ATLAS has been able to utilize the much more efficient ROOT split writing mode to a much greater extent, and is now in a position to take full advantage of that feature.

Lastly, the persistent state representations are handled internally by Athena without exposing them to the end user. With the introduction of the ROOT access mechanism, this is also true for analysis directly from ROOT. Therefore one of the important requirements, that the solution to schema evolution (or data persistency in general) does not influence or require modifications on the transient side, was met.

8. Acknowledgements

This manuscript has been authored by employees of Brookhaven Science Associates, LLC under Contract No. DE-AC02-98CH10886 with the U.S. Department of Energy.

Argonne National Laboratory's work was supported by the U.S. Department of Energy, Office of Science, Office of Basic Energy Sciences, under contract DE-AC02-06CH11357.

Notice: The publisher by accepting the manuscript for publication acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.

References

[1] http://atlas.web.cern.ch/Atlas/index.html
[2] https://twiki.cern.ch/twiki/bin/view/Atlas/AthenaFramework
[3] http://pool.cern.ch
[4] ftp://root.cern.ch/root/doc/11InputOutput.pdf
[5] http://proj-gaudi.web.cern.ch/proj-gaudi
[6] https://twiki.cern.ch/twiki/bin/view/Atlas/AthenaRootStreamerSvc
[7] https://twiki.cern.ch/twiki/bin/view/Atlas/TransientPersistentSeparation
[8] https://twiki.cern.ch/twiki/bin/view/Atlas/AthenaROOTAccess
[9] D. Malon, P. van Gemmeren, A. Schaffer, Sailing the Petabyte Sea: Navigational Infrastructure in the ATLAS Event Store, in Computing in High Energy and Nuclear Physics (CHEP-2006), Volume I, pp 312-315