Journal of Physics: Conference Series

PAPER • OPEN ACCESS

Implementation of the ATLAS Run 2 event data model

To cite this article: A Buckley *et al* 2015 *J. Phys.: Conf. Ser.* **664** 072045

View the article online for updates and enhancements.

Related content

- Implementation of the ATLAS trigger within the multi-threaded software framework AthenaMT
  Ben Wynne and ATLAS Collaboration
- ATLAS analysis model
  A Farbin
- New developments in file-based infrastructure for ATLAS event selection
  P van Gemmeren, D M Malon and M Nowak

Recent citations

- Extreme I/O on HPC for HEP using the Burst Buffer at NERSC
  Wahid Bhimji *et al*
- Event visualization in ATLAS
  R M Bianchi *et al*

IOP ebooks™

Bringing you innovative digital publishing with leading voices to create your essential collection of books in STEM research.

Start exploring the collection - download the first chapter of every title for free.

This content was downloaded from IP address 131.169.5.251 on 02/01/2019 at 23:01

21st International Conference on Computing in High Energy and Nuclear Physics (CHEP2015) IOP Publishing
Journal of Physics: Conference Series **664** (2015) 072045 doi:10.1088/1742-6596/664/7/072045

Implementation of the ATLAS Run 2 event data model

A Buckley[1], T Eifert[2], M Elsing[2], D Gillberg[2], K Koeneke[3], A Krasznahorkay[2], E Moyse[4], M Nowak[5], S Snyder[5], and P van Gemmeren[6]

(For the ATLAS Collaboration)

[1] University of Glasgow, Department of Physics and Astronomy, Glasgow G12 8QQ, United Kingdom

[2] CERN, CH-1211 Geneva 23, Switzerland

[3] Albert-Ludwigs-Universität, Fakultät für Mathematik und Physik, Hermann-Herder Str. 3, D-79104 Freiburg i.Br., Germany

[4] University of Massachusetts, Department of Physics, 710 North Pleasant Street, Amherst, MA 01003, USA

[5] Brookhaven National Laboratory, Physics Department, Bldg. 510A, Upton NY 11973, USA

[6] Argonne National Laboratory, High Energy Physics Division, 9700 S. Cass Avenue, Argonne IL 60439, USA

E-mail: `snyder@bnl.gov`

Abstract.

During the 2013–2014 shutdown of the Large Hadron Collider, ATLAS switched to a new event data model for analysis, called the xAOD. A key feature of this model is the separation of the object data from the objects themselves (the 'auxiliary store'). Rather than being stored as member variables of the analysis classes, all object data are stored separately, as vectors of simple values. Thus, the data are stored in a 'structure of arrays' format, while the user still can access it as an 'array of structures'. This organization allows for on-demand partial reading of objects, the selective removal of object properties, and the addition of arbitrary user-defined properties in a uniform manner. It also improves performance by increasing the locality of memory references in typical analysis code. The resulting data structures can be written to ROOT files with data properties represented as simple ROOT tree branches. This paper focuses on the design and implementation of the auxiliary store and its interaction with ROOT.

1. Introduction

The first data-taking run (Run 1) of the ATLAS experiment [1] was very successful. However, it became apparent that the design of the event data model (EDM) used for this run [2] had some limitations. In particular, the data structures used were quite complicated and relied on expensive C++ features such as virtual inheritance.

The structure of the event data was in fact sufficiently complicated that it could not be reasonably written directly in ROOT [3] format. To deal with this, ATLAS converted on output the complex transient data model to a simpler persistent data model which could be written to ROOT directly. Besides adding overhead, this made it difficult to use files written with the ATLAS EDM without having the full ATLAS release available.

Content from this work may be used under the terms of the Creative Commons Attribution 3.0 licence. Any further distribution of this work must maintain attribution to the author(s) and the title of the work, journal citation and DOI.
Published under licence by IOP Publishing Ltd

In response to this, ATLAS physicists would often convert the data samples using the full EDM to ones using a simplified data format directly readable from ROOT [4]. While this was originally intended to be used for only the much-reduced data sets used for the final steps of a data analysis, eventually each physics group was converting essentially all the data to ROOT format. This not only led to excessive duplication of data, but it made it difficult to maintain analysis tools that could be used both for the full EDM and for the ROOT-friendly formats.

Therefore, for Run 2, ATLAS decided to redesign the part of the EDM used for analysis. Several considerations went into this, as outlined below.

First, several types in the Run 1 EDM supported adding extra named pieces of data, called 'decorations', to elements of containers. This was originally implemented to allow separating pieces of the structure for I/O. However, this was implemented independently for different EDM types, resulting in several versions of decoration code all doing essentially the same thing, but in different ways. So one desire for the new design was to unify this functionality across all types.

Second, each container element was a separate C++ object; the data were essentially stored as an 'array of structures', which can give poor locality of reference. Might a 'structure of arrays' work better, and can the code still have an interface that looks like a collection of structures?

Third, the data format should be easily and efficiently readable from ROOT, using at most only a small part of the ATLAS code base. It is desirable to be able to read only parts of objects, as needed, and also to allow user analysis code to extend the data stored for an object.

These considerations led to an EDM design for Run 2 based on the concept of an 'auxiliary data store'. The rest of this paper describes the design of this store and how it is used by the offline software, and finally gives some notes on its implementation and performance.

2. Auxiliary store design overview

The ATLAS offline software uses the usual 'whiteboard' pattern. Event processing is specified as a list of algorithms, executed in sequence. These do not communicate directly with each other, but rather read and write event data from a central event store, which contains objects of arbitrary type, identified by string keys. I/O operations also work from the event store.

While types in the event store are arbitrary, containers of objects are usually represented by the type `DataVector<T>`, which provides the interface to the auxiliary store. This is much like `std::vector<T*>`, but with several additions.

Optional ownership A `DataVector` may own the elements to which it points; these elements are deleted when they are erased from the vector. Whether a `DataVector` owns its elements or not is specified by an optional argument to the `DataVector` constructor.

Container covariance Given these declarations:

```
1 class FourVector {};
2 class Particle : public FourVector {};
3 DATAVECTOR_BASE (Particle, FourVector);
```

then `DataVector<Particle>` will derive from `DataVector<FourVector>`, making it easier to write, for example, a generic algorithm operating on a container of `FourVector` objects.

Auxiliary data Data of arbitrary type can be attached to elements of a `DataVector`, identified by name. These data are stored as vectors and managed by a separate 'auxiliary store' object accessed via an abstract interface.

The general design of the auxiliary data store is illustrated in Fig. 1. The `DataVector` object contains pointers to its elements; the elements, in turn, contain a pointer back to the container and their index within the container. The `DataVector` also has a pointer to an auxiliary store object, implementing the abstract interface `IAuxStore`. This store object manages a set of

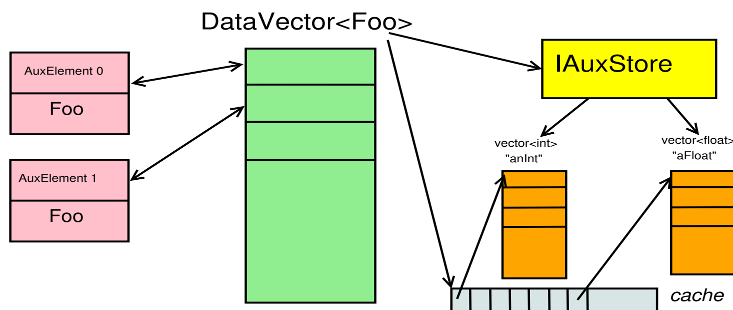

Figure 1. Summary of the design of the auxiliary store.

vector objects, one per variable. When a request is made for an auxiliary variable associated with a given element of the container, the `DataVector` retrieves from the store object the vector for that variable, then uses the element's index to find the proper element in the variable vector.

For Run 2, almost all object data are stored as auxiliary data rather than as class members.

3. Usage of auxiliary data

Here is an example of how a class using auxiliary data may be defined.

```
1  struct C : public AuxElement {
2    int anInt() const {
3      static Accessor<int> acc("anInt"); return acc(*this);
4    }
5    int setAnInt (int x) {
6      static Accessor<int> acc("anInt"); acc(*this) = x;
7    }
8    ...
9  };
```

As shown in line 1, classes that have associated auxiliary data should derive from the base class `AuxElement`. (Other types may still be used with `DataVector`; they just cannot have any auxiliary data.) Auxiliary variables that are considered part of the class will usually have getter and setter methods (the ATLAS EDM provides macros to streamline this). User code identifies variables with a string, while they are represented internally by small integers. The `Accessor` class seen in line 3 serves to cache this lookup, so it does not need to be done each time the variable is accessed. (`Accessor` has no other internal state, so it can safely be made `static`.)

The following example shows some of the ways in which this class may be used.

```
1  // Create container, set an aux data store, and add elements.
2  DataVector<C>* vc = new DataVector<C>;
3  CAuxContainer* store = new CAuxContainer;
4  vc->setStore (store);
5  vc->push_back (new C);  vc->push_back (new C);
6
7  // Set/get auxiliary data through class members.
8  (*vc)[0]->setAnInt (3);
9  std::cout << (*vc)[0]->anInt() << "\n";
10
11 // Attach additional auxiliary data to objects.
12 static C::Accessor<int> myInt ("myInt");
13 myInt((*vc)[0]) = 2;
```

```
14    myInt(*vc, 1) = myInt((*vc)[0]) + 1;
15
16    // Alternate interface that does not cache the lookup.
17    (*vc)[1]->auxdata<float> ("myFloat") = 1.5;
18
19    // Like Accessor, except can add data to const objects.
20    static C::Decorator<float> myFloat ("myFloat");
21    const DataVector<C>& cvc = ...;   myFloat (cvc[0]) = 10;
```

Before any auxiliary variables can be used, the auxiliary store object must be associated with the container, as in line 4. By convention, each EDM class has a corresponding auxiliary store class. Auxiliary variables may then be accessed through the getter and setter, as in line 8. Note that the vector does not own the store object; typically, they will both be registered with the event store (not shown in this example). One may associate arbitrary additional auxiliary data with the container, as shown starting at line 12. Here, a new `Accessor` object is declared and used to read and write the variable. The variable may be accessed either through an element of the container [`myInt((*vc)[0])`] or directly from the container itself, given the element index [`myInt(*vc, 0)`]. The latter form avoids a memory read from the element itself. It is also possible to access a variable directly, without using the `Accessor` class, as shown in line 17; however, this form will be slower as the variable name needs to be looked up each time.

The `Accessor` class will only allow adding or changing a variable in a non-`const` container. However, one important use case involves reading data from a file, 'decorating' it with new variables, and saving the results. Objects read from the input file will be `const`, so this cannot be done through the `Accessor` interface. Instead, a similar class is provided called `Decorator`. This does allow adding new data to a `const` container; however, an attempt to change existing data will fail with a runtime error.

As mentioned above, an auxiliary store must be associated with a container before any auxiliary data may be associated with the container's elements. To add auxiliary data to an object that is not part of a container, first call `makePrivateStore()` on the object. This will create a new, private, auxiliary store associated with the object itself. If the object is later added to a container, the auxiliary data will be copied from the private store to the container's store, and the private store deleted. If the object is later removed from the container, the private store will be recreated and repopulated. For example:

```
1  C* c = new C;    c->makePrivateStore();
2  static C::Accessor<int> anInt ("anInt");
3  anInt(*c) = 5;  // In c's private store.
4  DataVector<C>& vc = ...;
5  vc.push_back (c);  // c's private store deleted, data copied to vc's store.
6  ...
7  vc.swapElement (vc.begin(), 0, c); // vc gives up ownership.
8                                     // c's private store recreated.
```

The `setStore` method may also be called directly on an object to associate it with an externally-managed store. This is used for standalone objects that are to be written to the output file, such as the `EventInfo` object which records run and event numbers.

A key feature of the design is the isolation of the auxiliary store from the `DataVector` via an abstract interface. This allows for multiple implementations of the auxiliary store; for example:

- Each EDM class has a corresponding 'static' auxiliary store class, containing the variables regarded as class members. The static store references another 'dynamic' store, managing

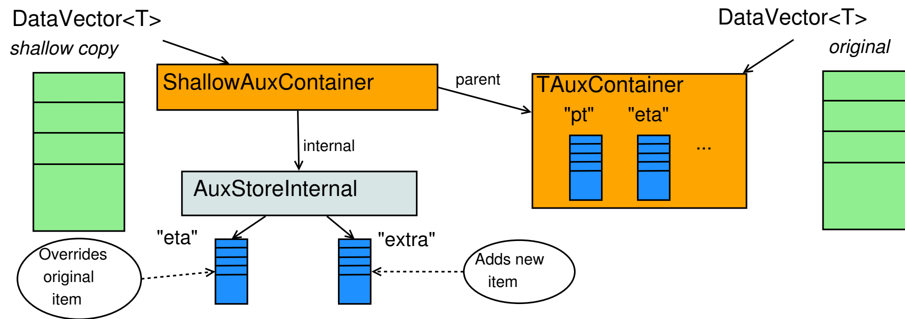

Figure 2. A 'shallow copy' of a `DataVector` allows writes to go into a new auxiliary store, while reads are forwarded to the original store.

arbitrary data. New auxiliary variables are added to the dynamic store, while the static store forwards any requests for variables that it does not manage to the dynamic store.

- EDM objects produced by the trigger can use an auxiliary store implementation specialized for storage in the raw data stream.
- For objects which are read from a data file, a special auxiliary store implementation is used that allows the reading of variables to be deferred until they are actually referenced.
- A special auxiliary store implementation is used to implement 'shallow copies'. When a shallow copy is made of a `DataVector` with auxiliary data, the `DataVector` itself is copied. The auxiliary store for the new vector is of the type `ShallowAuxContainer`, and it maintains a reference to the original store. Any requests to write a variable will be carried out in the `ShallowAuxContainer`, while read requests for variables not in the `ShallowAuxContainer` will be forwarded to the original store (see Fig. 2). This allows one to make a copy of a container and change a few variables, but still share the storage for most of the data.

4. I/O of auxiliary data

Types with auxiliary data can be written to and read from ROOT files. The `DataVector` itself is saved as if it were a `std::vector<T>` by means of a custom ROOT collection proxy. For most EDM types, the elements themselves (`T`) do not actually contain any data, so this effectively just records the length of the vector.

Object data are instead stored in the separate auxiliary store object. The association between the `DataVector` container and its auxiliary store object is made by name: if the `DataVector` is saved to a ROOT branch called '`Electrons`', then then auxiliary store will be saved to the branch '`ElectronsAux.`'. (The trailing period is required by ROOT in order to be able to split the object into separate branches.) When a container is read in, the I/O system is responsible for calling `setStore` on the `DataVector` to associate it with the proper auxiliary store. As mentioned above, this is usually an instance of the corresponding 'static' auxiliary store class, containing the data which are considered to be part of the EDM object proper. Instances of this static store are saved and restored as a single object. The vectors held by the dynamic store referenced by the static store are associated directly with the ROOT `TBranch` objects, eliminating the need to copy these data. Branches for dynamic data are distinguished by having names of the form '`ElectronsAuxDyn.`'. This organization is illustrated in Fig. 3.

5. Implementation notes

If a type is to have associated auxiliary data, it must derive from the base class `AuxElement`, which contains the element's index within the container and a pointer back to the container itself.

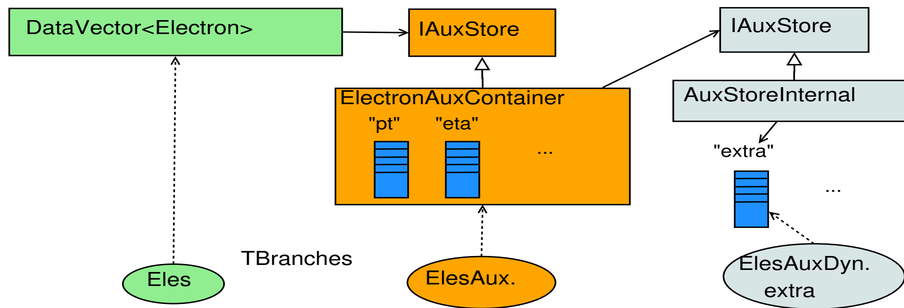

Figure 3. Illustration of static and dynamic stores as used for I/O.

This allows retrieving auxiliary data from the container given only a pointer to the element, which is important for compatibility with Run 1 code. All operations which can modify the `DataVector` are extended so as to correctly maintain the container reference and index within the elements. In the case of element types that do not derive from `AuxElement`, template techniques are used to completely elide these extra operations, ensuring that the auxiliary data feature does not add extra overhead for existing types that do not use auxiliary data.

User code references variables by name; however, internally they are represented by small integer identifiers. These identifiers are purely internal; in particular, in data files, variables are only identified by name, not by identifier. The mapping between names and integers is managed by a global registry class. The registry keeps track of the type of each variable, and will raise an exception if an attempt is made to access a variable with a different type than was used for the first access. The registry provides factory functions to create the vectors managed by the auxiliary stores; the actual vectors are embedded in wrapper objects that provide a common interface for operations such as resizing the vector. The registry also provides operations such as swapping two elements given `void` pointers to them and their auxiliary data identifier. These generic operations are implemented by template code which is instantiated when a variable is accessed with a specific type (e.g., '`Accessor<int> v ("myVar");`'). In some cases, one needs to be able to manipulate auxiliary data vectors read from an input file before user code has made any explicit references to variables of that type. In that case, a generic implementation is used, based on ROOT's type reflection facilities.

In order to make access to auxiliary data efficient, the container maintains a cache of pointers to the data for each vector (see Fig. 1); this cache is simply a vector of pointers, indexed by the auxiliary data identifiers. If the pointer for a given variable is present in the cache, then access to that variable can be done entirely with inlined code. Otherwise, an out-of-line method is called that retrieves the pointer via the `IAuxStore` abstract interface.

A simplified version of the code that actually accesses an auxiliary variable is shown below.

```
1  template <class T>
2  T& getData (size_t auxid, size_t i) {
3    return reinterpret_cast<T*>(getDataArray(auxid))[i];
4  }
5  void* getDataArray (size_t auxid) {
6    if (auxid >= m_cache_len || m_cache[auxid] == 0) {
7      getDataArrayOol (auxid);
8      // Inform compiler of postcondition
9      if (auxid >= m_cache_len || m_cache[auxid] == 0)
10       __builtin_unreachable();
11   }
12   return m_cache[auxid];
13 }
```

The key operation here is `getDataArray`, which returns a pointer to the start of the data for a given variable. This first checks at line 6 if the cache entry for this variable is valid. If not, then the out-of-line method is called at line 7 to fetch the pointer from the auxiliary store using the `IAuxStore` interface. Finally, the pointer is retrieved from the cache at line 12.

If the same variable is retrieved multiple times for a given container, then it is possible to remove the check on the cache validity for accesses after the first. In fact, this is a necessary requirement for being able to vectorize accesses using this interface. This is the purpose of the use of `__builtin_unreachable` at line 10. After the call to `getDataArrayOol(auxid)`, the cache entry for `auxid` must be valid, as a postcondition of the call. However, as this is in out-of-line code, the compiler has no way of knowing this to be true. The `__builtin_unreachable` construction functions to inform the compiler of this postcondition.

Recent versions of gcc (tested with 4.9) are able to take advantage of this, compiling this:

```
1    return v.getData<float> (auxid, 0) + v.getData<float> (auxid, 1);
```

into

```
1          movq        72(%rdi), %rax  ; m_cache_len
2          cmpq        %rax, %rsi      ; compare against auxid (in rsi)
3          jb       .L118
4   .L114:  <out-of-line code here>
5   .L118:
6          andl        $1, %eax
7          movq        56(%rdi,%rax,8), %rax  ; fetch m_cache
8          movq        (%rax,%rsi,8), %rax    ; fetch m_cache[auxid]
9          testq        %rax, %rax
10         je       .L114
11         movss        (%rax), %xmm0          ; first aux var
12         addss        4(%rax), %xmm0         ; add second aux var
```

(The extra operations at line 7 are related to thread-safety.) The Intel compiler makes a similar optimization, while clang as of version 3.4 does not. No compiler tested so far, however, will fully take advantage of this optimization within a loop. This is an area for further work.

Auxiliary-data operations on a `DataVector` are thread-safe in the same manner as `std::vector`; that is, operations that can change the vector need to be locked externally, but it is possible to read auxiliary data from multiple threads simultaneously. Further, the thread-safety should not significantly affect the performance of auxiliary data access, which implies that there should be no locking in the inline code. For the common case, where the length of the cache vector doesn't change, there is no problem: if the cache entry is invalid, the out-of-line code is called to update the cache, and the result of this will be the same in all threads. The out-of-line code itself is protected with a lock. The case where the cache vector needs to grow is handled with a strategy inspired by read-copy-update. In the out-of-line code, within the lock, a new cache vector is allocated and copied from the old cache vector. Then the pointer to the cache vector is updated, followed by the cache length. As long as these fields are updated in this order, other threads will always have a consistent view of the cache vector, even if they do not acquire a lock. The old cache vectors are then simply saved until the `DataVector` object is deleted. This strategy is illustrated by the (simplified) code fragment below.

```
1   const void* AuxVectorData::getDataArrayOol (SG::auxid_t auxid) {
2     guard_t guard (m_mutex);
3     const void* ptr = m_store->getData (auxid); // Retrieve from IAuxStore.
4     if (!ptr) throw SG::ExcBadAuxVar (auxid);
```

```
5    if (auxid >= m_cache_len) {        // Does the cache vector need to grow?
6      // Allocate new cache vector and grow.
7      size_t newlen = (auxid+1) * 2;
8      void** newcache = new void*[newlen];
9      m_allcache.push_back (newcache);
10     std::copy (m_cache, m_cache + m_cache_len, newcache);
11     std::fill (newcache + m_cache_len, newcache + newlen, nullptr);
12     // Update the cache pointer/size.
13     m_cache = newcache;
14     fence_seq_cst();  // Prevent write reorders; not needed on x86(_64)
15     m_cache_len = newlen;
16   }
17   m_cache[auxid] = ptr;   // Store the pointer in the cache.
18 }
```

6. Performance

The new EDM is an important contributor to the overall $\sim 3\times$ speedup of the ATLAS reconstruction achieved between Runs 1 and 2. Dedicated tests show that the new auxiliary store implementation is from 25% to $3\times$ faster than the 'auxiliary store' implementations used in Run 1. Going forward, the new data organization of this design should provide more opportunities for vectorization. The size for analysis data with 25 interactions per crossing is about 240 kB/event, similar to that of Run 1 and within the budget of 250 kB/event.

7. Summary

ATLAS redesigned the event data model for Run 2 to simplify it and make it more easily readable directly from ROOT. The implementation is based on the concept of an 'auxiliary store', which stores object data in a set of vectors separate from the objects themselves. Additional features include allowing user analysis code to add additional information to objects, on-demand reading of parts of objects, and shallow copies. Performance of the new event data model has been good, and it forms a good foundation for future work in ATLAS offline software.

Acknowledgments

This work is supported in part by the U.S. Department of Energy under contract DE-AC02-98CH10886 with Brookhaven National Laboratory.

References

[1] ATLAS Collaboration 2008 *JINST* **3** S08003
[2] van Gemmeren P and Malon D 2009 *IEEE Int. Conf. on Cluster Computing and Workshops, 2009, New Orleans, USA*
[3] Brun R and Rademakers F 1997 *Nucl. Inst. Meth.* A **389** 81–86 URL `http://root.cern.ch`
[4] Snyder S and Krasznahorkay A 2012 *J. Phys. Conf. Ser.* **396** 022047