# Implementation of the ATLAS Run 2 event data model

Scott Snyder

Brookhaven National Laboratory, Upton, NY, USA

Apr 13, 2015
CHEP 2015

# Introduction

Limitations of Run 1 event data model

- Overly-complicated; relied on expensive features like virtual derivation.
- Transient EDM converted to/from simplified persistent form during I/O. Schema evolution also handled this way. Adds overhead and makes it difficult to read data outside the full ATLAS release.
- Led to offline EDM generally not being used for analysis.

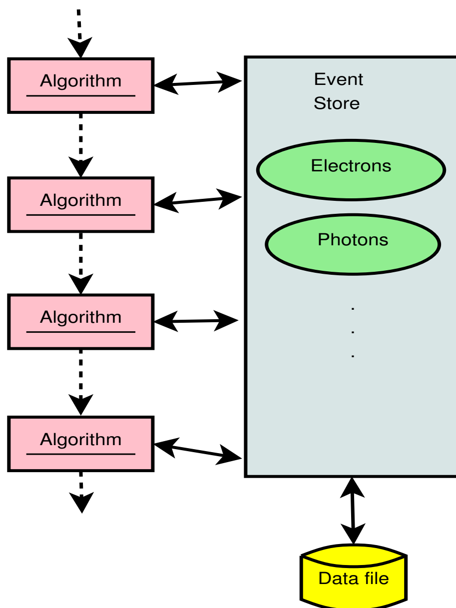
ATLAS analysis event data model was completely redesigned for Run 2.

- Simplify.
- Improve performance.
- Make directly readable from ROOT.

Implementation based on concept of "auxiliary store."

This talk: summarize the implementation of the auxiliary store underlying the Run 2 data model.

# ATLAS offline overview




"Whiteboard" pattern

- Sequence of Algorithms communicating via event store.

Any type of object in event store, but containers usually `DataVector<T>`.

Like `vector<T*>` except:

- Optional strict-ownership semantics.
- "Container covariance".

New for Run 2

- Auxiliary data.

# Desires for Run 2

Several Run 1 types supported adding extra named pieces of data — “decorations” — to elements of containers.

- Originally done to be able to separate pieces of the structure for I/O.
- Several different, incompatible implementations.
- Can we unify this?

Data stored as, essentially, “array of structures.”

- Poor locality of reference.
- “Structure of arrays” might be better.
- Can we still make it look like a collection of structures?

Make data easily and efficiently readable from ROOT.

- Avoid copies.
- Partial object reading / writing.
- User extensibility for analysis.

# Run 2 event data model

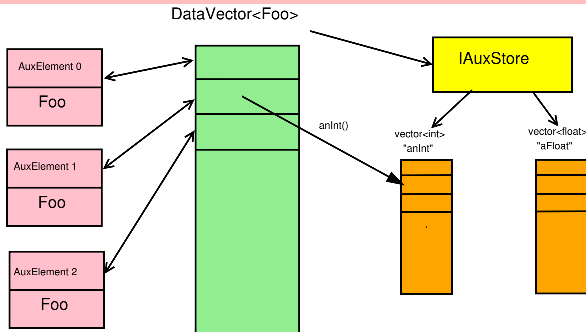
Analysis-level event data model redesigned for Run 2. (“xAOD”)

- Simplify, and make more directly usable with ROOT.

Based on new “auxiliary data” feature of `DataVector`.

- Attach data of arbitrary type to elements of `DataVector`.
- Data are stored as vectors, managed via separate “auxiliary store” object via abstract interface.
- Object data stored as auxiliary data rather than in the object itself.

Almost all object data stored as auxiliary data.



# Usage example 1

```
struct C : public AuxElement {
  int anInt() const {
    static Accessor<int> acc("anInt"); return acc(*this);
  }
  int setAnInt (int x) {
    static Accessor<int> acc("anInt"); acc(*this) = x;
  }
};
...
DataVector<C> vc;

// Set an aux data store.
CAuxContainer store;
vc.setStore (&store);
```

# Usage example 2

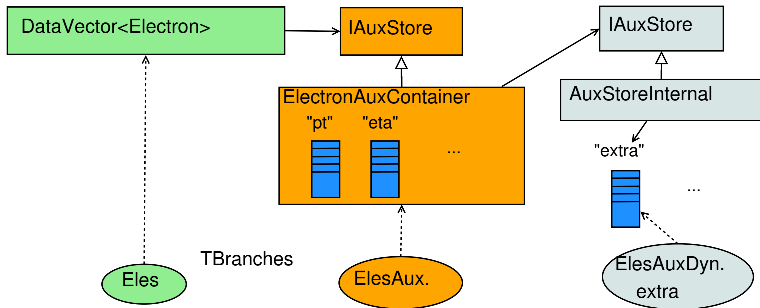
```
vc.push_back (new C);  vc.push_back (new C);

// Set/get auxiliary data through class members.
vc[0]->setAnInt (3);
std::cout << vc[0]->anInt() << "\n";

// Attach additional auxiliary data to objects.
// The Accessor object caches the lookup of an internal
// identifier from the data item name.
static C::Accessor<int> myInt ("myInt");
myInt(vc[0]) = 2;
myInt(vc, 1) = myInt(vc[0]) + 1;

// Alternate interface that does not cache the lookup.
v1[1]->auxdata<float> ("myFloat") = 1.5;
```

# I/O interaction

`DataVector<T>` itself saved to ROOT like `vector<T>`

The aux store for xAOD types is a “static” class containing the aux variable vectors as members. Saved and loaded as a single object.

The static store references a “dynamic” store, which can manage arbitrary aux data. New aux data items are added here. Items can be saved and loaded individually from ROOT. Branch objects are set to the vectors managed here.



# IAuxStore

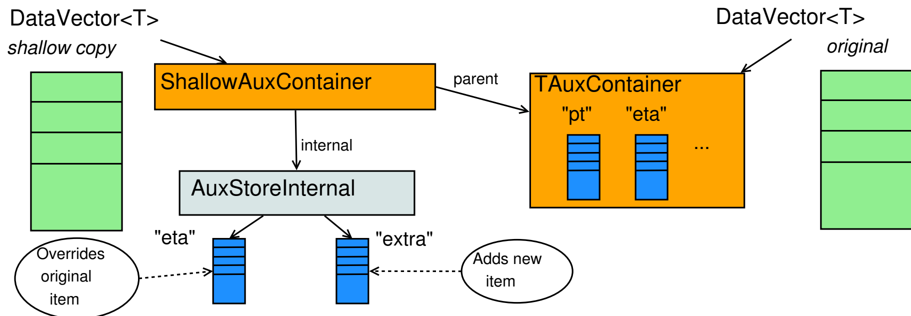
A key feature is the ability to change the auxiliary store implementation through the abstract interface. Some types used:

Each xAOD type has a static auxiliary store chained to a dynamic store.

In trigger: implementation specialized for storage in raw data stream.

On input: implementation allowing on-demand reading of items.

"Shallow copy" store: records writes, forwards reads for unknown items to another store.

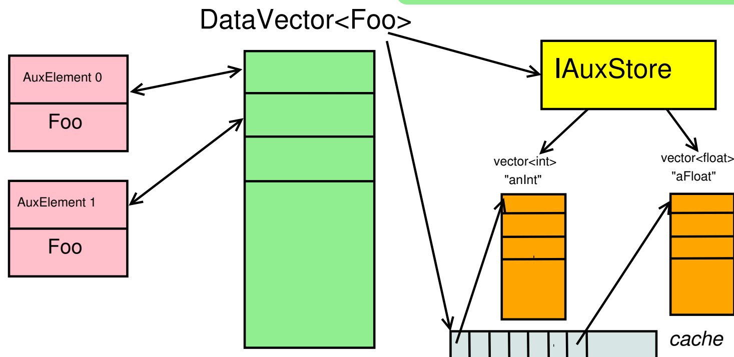


# Implementation notes

Types with aux data derive from `AuxElement`: holds a reference to the container and the element index.

- Updated by vector operations.
- No overhead for types that do not derive from `AuxElement`.

Aux data items identified by small integers via a registry. `DataVector` holds cache of pointers to start of data for each item. If pointer is in cache, access is entirely inlined. Otherwise, calls to out-of-line code.



# Implementation notes: aux data dereferencing

Actual aux data dereference written something like this (simplified a bit):

```
template <class T>
T& getData (size_t auxid, size_t i)
{
  return reinterpret_cast<T*>(getDataArray(auxid))[i];
}
void* getDataArray (size_t auxid)
{
  if (auxid >= m_cache_len || m_cache[auxid] == 0) {
    getDataArrayOol (auxid);
    // Inform compiler of postcondition
    if (auxid >= m_cache_len || m_cache[auxid] == 0)
      __builtin_unreachable();
  }
  return m_cache[auxid];
}
```

# Implementation notes

If the same item is retrieved twice, as in

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

then can elide the test on the cache vector in the second access. The postcondition allows the compiler to do this.

Compilers currently can't do this for loops or for more complicated access patterns — for further work.

Thread safety:

Operations changing the `DataVector` must be externally locked (like `std::vector`). Only need to worry about managing the aux data cache.

Only really need to worry about what happens if the vector changes size.

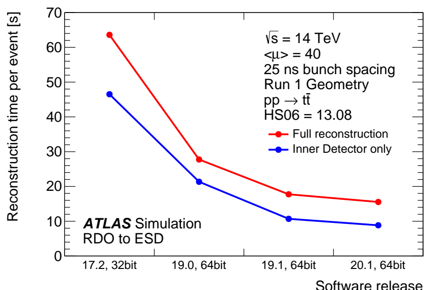
Take cue from RCU: when cache vector is reallocated, don't delete the old copy. Inlined portion is then lockless; only need locking in out-of-line code.

# Performance

The new event data model is one of the contributors to the improvements in reconstruction time in version 19.0 and later.

Dedicated tests show that the new auxiliary store implementation is anywhere from 25% to 3$\times$ faster than the Run 1 "auxiliary store" implementations.

Expect that this new design will provide more opportunities for vectorization in the future.




Event size budget for analysis data ("AOD") is 250kB/event ($\mu = 25$ interactions/crossing).

Current measured event size from simulation: 240kB/event. Comparable to Run 1.

# Summary

ATLAS redesigned the analysis event data model for Run 2.

Based on a unified “auxiliary store” design, providing

- Data stored internally as a set of vectors.
- Traditional interfaces are supported.
- Data are directly readable from ROOT.
- Individual data items can be removed from the persistent store.
- User analysis code can add arbitrary new data items to objects.

Separating data representation from interface via abstract interface allows:

- On-demand reading.
- Shallow copies.

Foundation for the ATLAS reconstruction moving forwards!

# Additional material

# Standalone objects

The auxiliary data for an object are associated with the container. But what if you want to have an object that is not part of a container?

Call `makePrivateStore` on the object to create a private store associated with the object itself. If the object is later added to a container, the aux data will be copied and the private store deleted. If it is later removed from the container, the private store will be recreated and repopulated.

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

Can also call `setStore` directly on object for externally-managed store.