CHEP'07 VICTORIA, BC
International Conference on Computing in High Energy and Nuclear Physics
2-7 Sept 2007 Victoria BC Canada

Explicit State Representation and the ATLAS Event Data Model

Theory and Practice

Marcin Nowak (Brookhaven National Laboratory)
David Malon, Arthur Schaffer, Peter van Gemmeren,
Scott Snyder, Sebastien Binet, Kyle Cranmer

ATLAS Database, EMB and Core Software groups

Introduction

- The Scope: Athena – ATLAS offline software framework
  - Management of data objects – transient and persistent
  - Event data persistency based on POOL and ROOT technologies
- The Problem: Schema Evolution
  - Unavoidable for production data
  - Must be possible without forfeiting the ability to read old data
- The Solution: Persistent state representation
  - Implementation
  - Performance implications
  - Direct access from ROOT

Schema Evolution

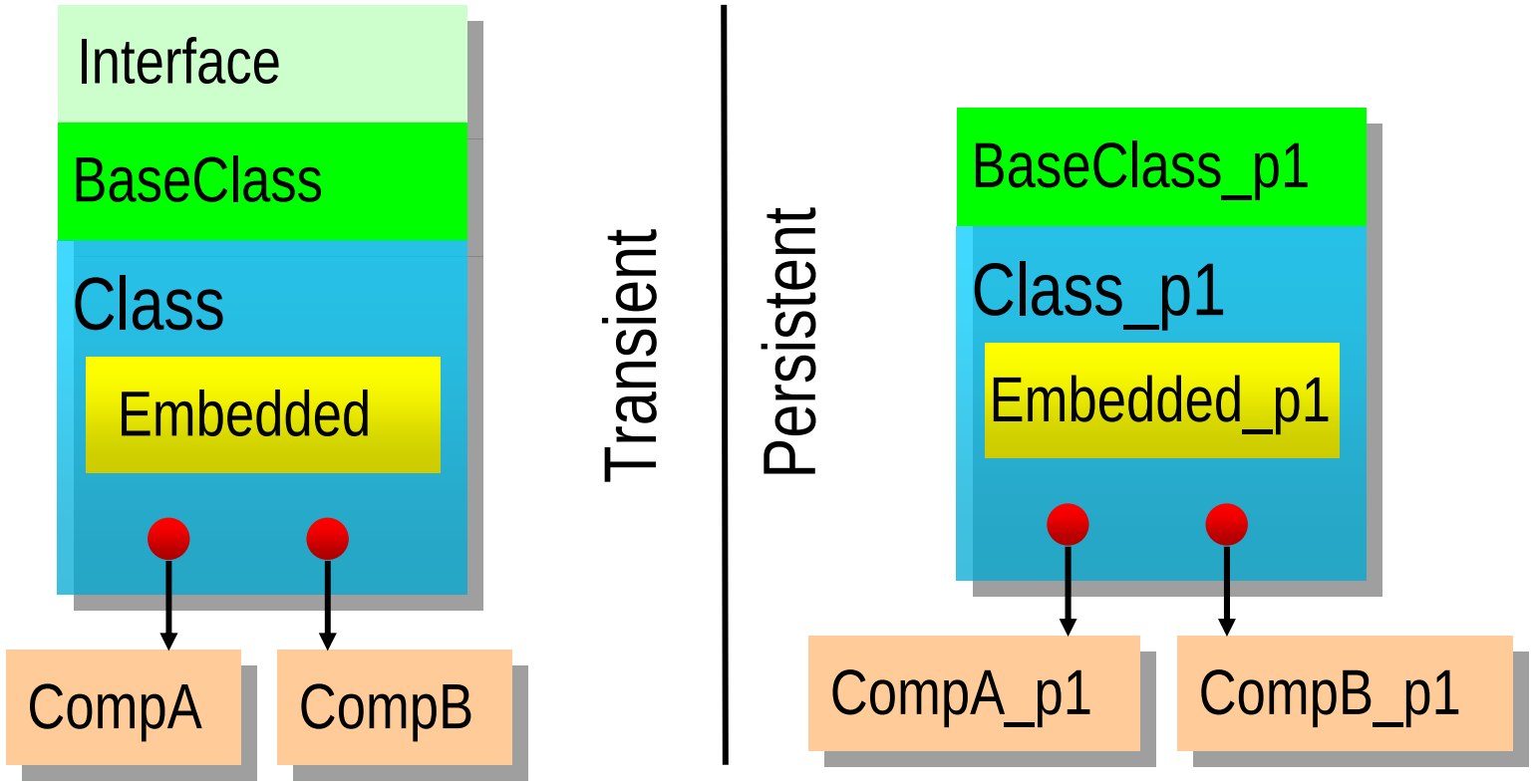
- Schema evolution is happening all the time
  - Often unnoticed on the application (transient) side
    - Class names do not change
    - Data representation is hidden behind interfaces
    - No ‘memory’ of the previous class shape
  - Quite visible on the persistent side
    - Data is more exposed
    - History of different class shapes, reaching years in the past
    - Class names still do not change
      How can we keep track of the different versions?
- Occasional redesigns of large portions of the data model
- Ideally schema evolution should be supported by the persistency layer
  - ROOT automatic schema evolution
    - Works in simple cases, fails in more complicated ones
  - ROOT custom streamers
    - Work in streamed mode, do not work in the split mode
- No satisfactory solution – maybe ATLAS needs to provide their own

Supporting Schema Evolution in Athena

- Starting requirements:
  - Preserve old versions of the schema
    - Maintain a library of conversion methods for reading the old data
  - Prevent ROOT automatic schema evolution in cases where it would fail
- Solution: create a separate, independent persistent data model
  - Typenames in the persistent data model are different from their transient equivalents
  - Schema version is part of the persistent typename
    - All versions are stored in the data dictionary
  - By introducing new versions we can ensure that persistent classes do not change beyond the limits of ROOT automatic schema evolution
  - Data objects need to be written as persistent types
    - Not possible in ROOT to write one type and read it as another type
    - Conversion has to happen both when reading and writing

The Persistent Data Model

- Example of a transient class and its persistent representation:

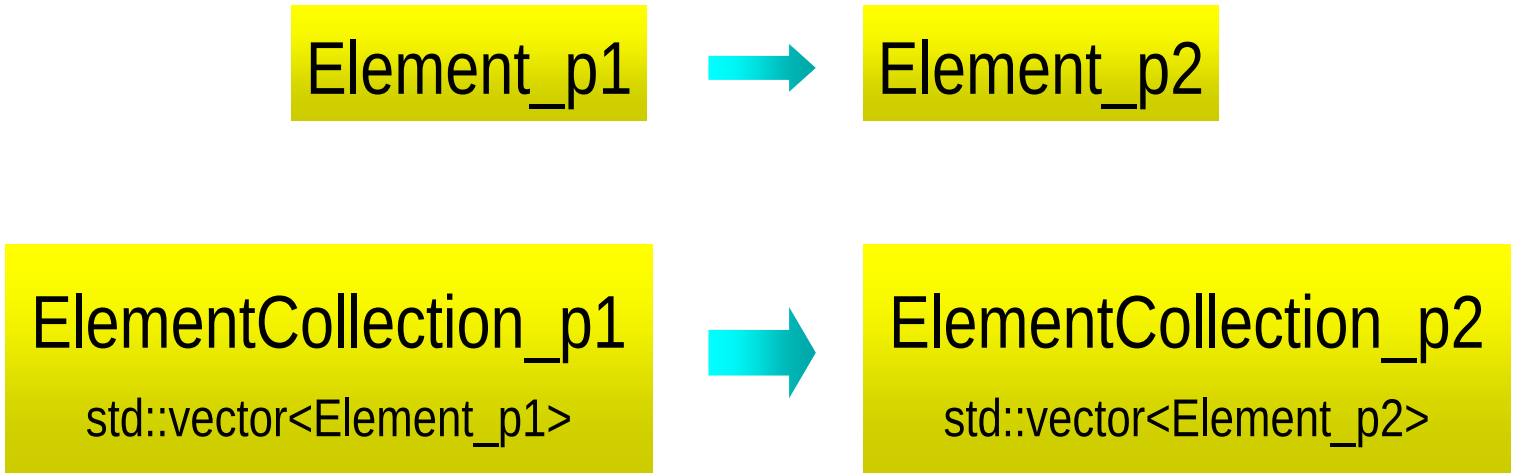


T/P Converters

- T/P Converters provide methods to convert data objects between their transient and persistent representations
  - One converter per each persistent representation
    - Old versions provide only persistent->transient conversion for reading
- T/P Converters are separate classes from persistent representations
  - Conversion methods need to operate on transient types – better to keep them apart to maintain data model separation
  - To make implementation easier, they are created by specialized base converter template classes
    - Simple and elegant on transient side, ugly in the data dictionary
    - Originally T/P converters were strictly transient objects
      - Now for direct ROOT access they need to be in the data dictionary
  - Abstract converter API makes handling converters easy even if the object type is not known

Chain Effect in Schema Evolution

- „Chain effect” is a result of introducing version numbers in the type names of the persistent data model. Changing the version of a type referenced by another object is causing a schema change for the referencing object, and the effect propagates

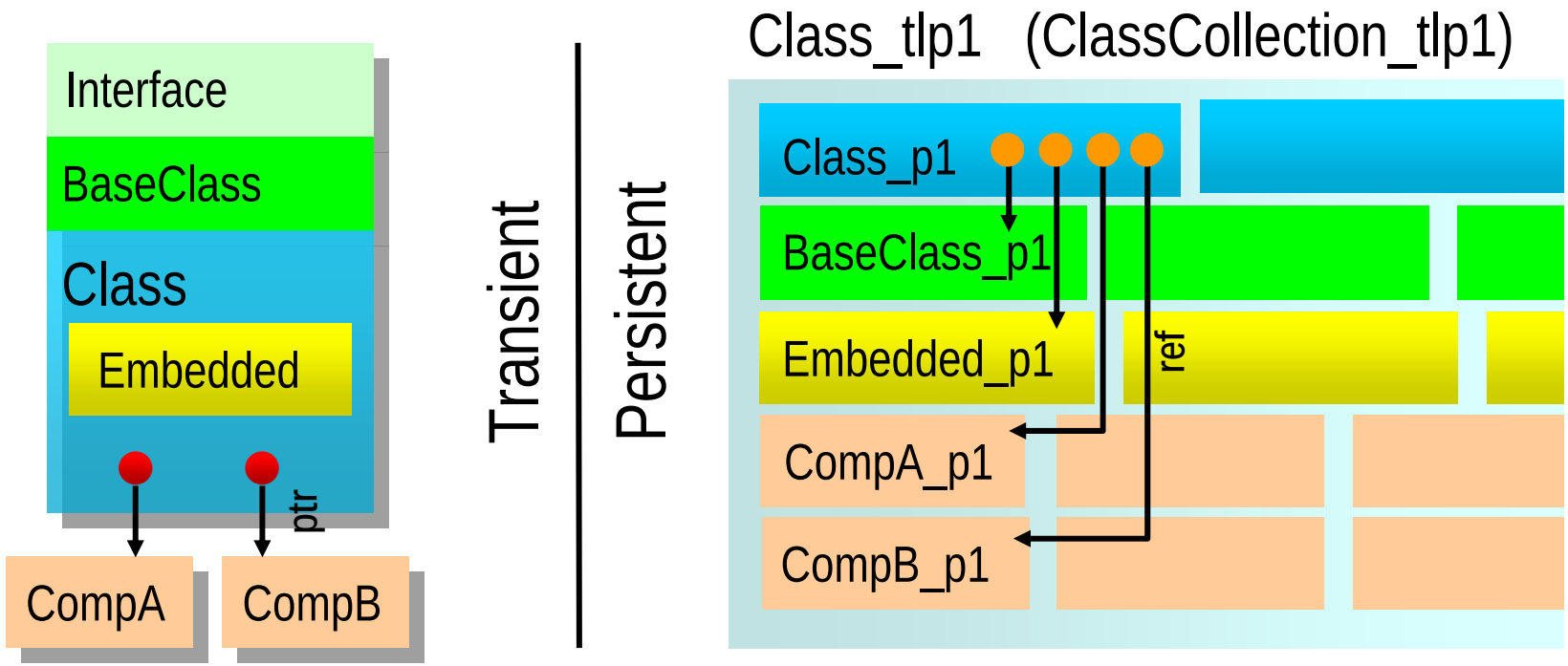


Typeless Persistent References

- To avoid the chain effect, we decided to replace direct C++ type dependencies in the persistent data model by typeless persistent references (TPObjRef)
  - TPObjRef contains only integers that have meaning to the TP converters
  - The new persistent data model does not any more (in general) contain pointers, inheritance or embedding
- Object components referenced by TPObjRef are not part of the same object in the C++ sense, so ROOT will not write them out automatically
  - To assure all elemental object components are stored together, they are gathered in special container objects, called top-level persistent objects

Top-Level Storage Objects

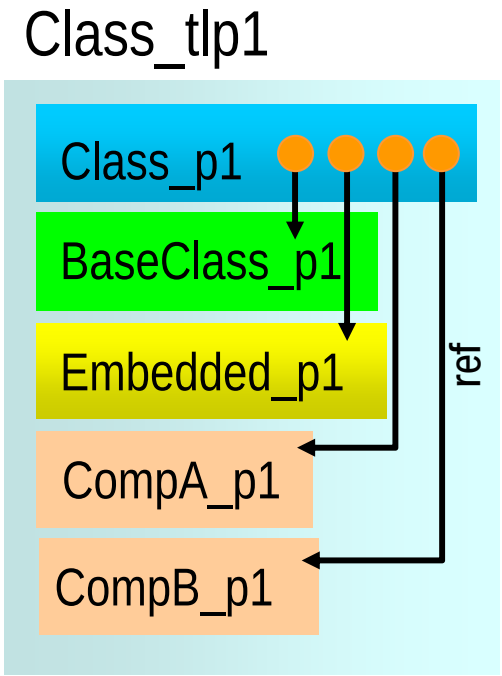
- Top-level persistent objects consist of vectors of component objects



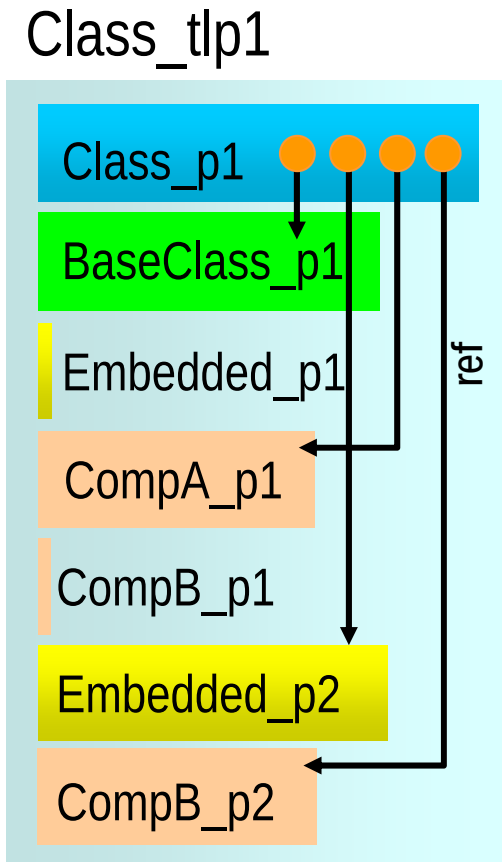
- Top-level objects are managed by top-level TP converters
  - Top-level converters own all TP converters for component objects
  - They take care of placing objects in the storage vectors

Schema Evolution with Top-Level Objects

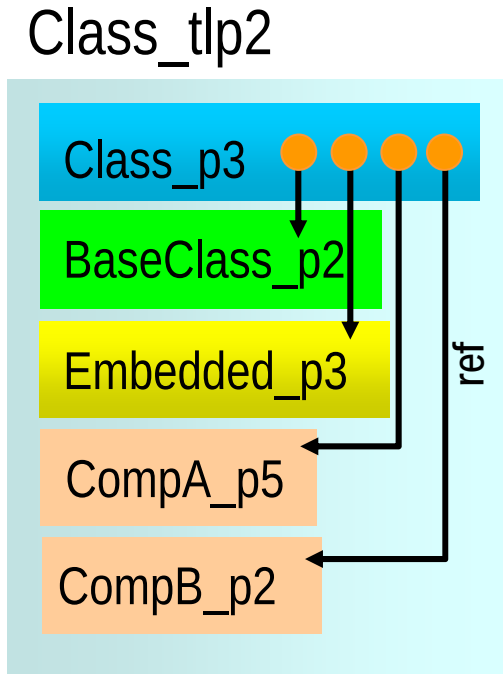
Automatic schema evolution – no changes



New versions of the component objects



New version of the Top-level object

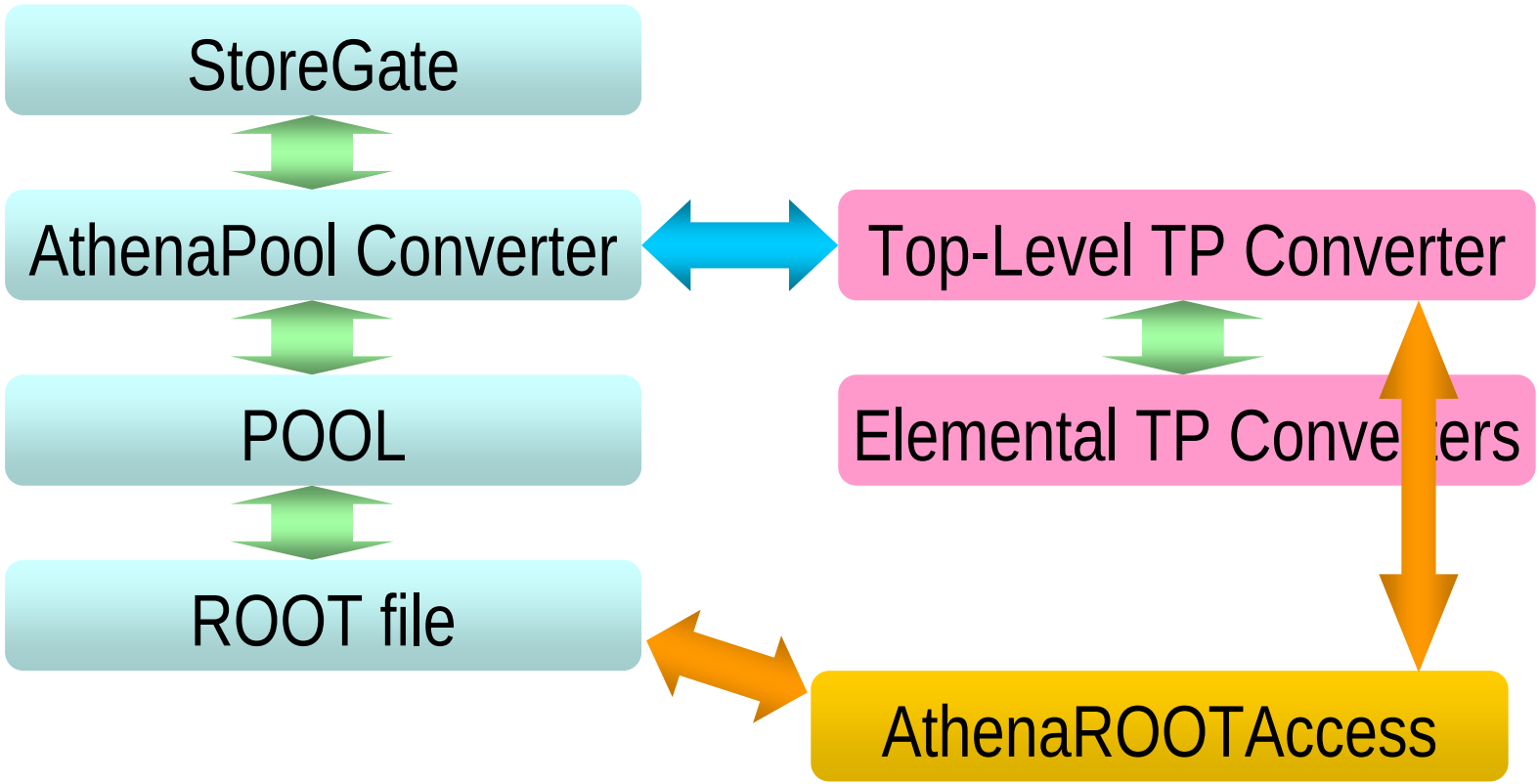


T/P Separation and AthenaPool Converters

- The T/P separation framework is inserted between Athena and the POOL persistency layer at the level of AthenaPool converters
  - Each StoreGate object type has an AthenaPool converter
    - The converters are used for writing to and reading from POOL
  - By default they are generated automatically by CMT and they do no conversion
    - AthenaPool converters can be customised
- A custom AthenaPool converter is needed to use T/P separation
  - Custom AthenaPool converter should own instances all versions of the top-level T/P converters for a given StoreGate object type
    - The newest version is always used when writing
  - After writing an object, AthenaPool converter returns POOL Token
    - All Tokens are stored as strings in the Event DataHeader
  - When reading, the correct version of a top-level TP converter is determined using type information from the Token

Complete Persistency Stack in AthenaPool

Original persistency layers in Athena
Persistency layers in Athena with T/P separation



Performance

- Improving I/O performance was not the original goal
  - Main focus was on schema evolution support
  - The solution was only supposed not to seriously degrade performance
    - Conversion between transient and persistent state had to introduce additional CPU load
- The first prototype based on the tracking data model showed:
  - Decrease in disk space usage by about 25%, caused mainly by:
    - Simpler data model (but not by much)
    - Better control over what data is written out
  - Slight improvement in I/O performance – the time spent in data conversion was recuperated by:
    - Less time spent in ROOT due to somewhat simpler data model
    - Smaller file size
  - Performance requirement was met

Performance with Top-Level Objects

- The new persistent state representation (with top-level objects, typeless references and object decomposition) improved read performance by a surprising factor
  - The Tracking data read speed improved 4 times
  - Various packages read speed improved up to 10 times
  - Winner: LArRawChannel data - **20 times faster**
    - DataVector with 183'000 small elements (~20 bytes each)
- File size reduction remained in the 20%-40% region
- We believe the reason for performance improvement is the difference in ROOT streamed and split writing mode
  - Streamed mode writes type information for every object and sub-object
    - Requires more processing per object
  - Removing pointers from ATLAS persistent data model allows ROOT writing in the split mode

Direct Access From ROOT

- End users may want to work with data files directly from ROOT
  - Without the Athena framework overhead
  - But using the transient classes, not the persistent state representations
- AthenaROOTAccess
  - Package enabling automatic use of T/P converters without Athena
  - Creates a proxy TTree with branches corresponding to the transient schema
    - The tree contains branches of special new type: TBranchTPConvert
    - Friend of the original TTree with persistent data
  - Accessing the 'transient' TTree automatically triggers reading from the 'persistent' TTree and T/P conversion
    - Standard tools like TTree::Draw() work as well
    - The end-user is entirely insulated from the persistent representations
  - Locating the correct T/P converters is done via the data dictionary
    - T/P converters must be entered into the dictionary now

Conclusions

- T/P separation is being widely adopted in ATLAS EDM
  - Persistent classes are being added in Athena release 13
  - With the goal to have the entire EDM T/P separated in release 14
- Explicit persistent state representation of data objects allows support for schema evolution and dramatically improves read performance
  - ATLAS expects to be able to achieve 10-15MB/s read speed
- Creating persistent classes by hand resulted in optimized and better thought through data model
  - Reduced file size
  - Quite a few old errors found and fixed
  - The extra effort required from the developers is hopefully justified
- The effects of explicit state representation on ATLAS ability to evolve EDM while maintaining backward compatibility will be visible in the future
  - Time will tell