A Brief and Gentle Overview of ATLAS I/O and Data Persistence

David Malon
malon@anl.gov

US ATLAS / BNL CSI Workshop
25 July 2018, Brookhaven

Introduction

- High-level overview of the major components that support ATLAS I/O and data persistence
- Experts will notice nontrivial omissions and (over)simplifications
  - We can add detail if we discover it matters for this workshop's purposes
- Will try to convey the basic ideas, but dedicate more time to areas that may be candidates for collaborative development
- Have proposed a few areas for possible joint efforts in our collective planning document
  - Admittedly without knowledge of BNL CSI interests and expertise
- Perhaps others will see additional possibilities in this description of ATLAS I/O and persistence

ATLAS and its event data

- By far the preponderance of ATLAS processing is event processing
- Our framework is (principally) an event processing framework
- Most data read and most data written are event data
- ATLAS currently has between 200 and 300 petabytes of event data
  - Including replicated datasets
- ATLAS stores the bulk of its event data using ROOT as its persistence technology
  - Though raw readout data from the detector is in another format
  - Much more on ROOT later
- The ROOT team estimates more than 1 exabyte of Large Hadron Collider (LHC) data reside in ROOT files

Auxiliary data

- While most ATLAS processing reads events and writes events, auxiliary non-event data are generally also needed for such processing
- Examples: Detector geometry, alignments, detector calibrations and conditions, detector and beam status, trigger menus, metadata, …
- These may be time-varying, not all on the same time scale
  - ATLAS divides its data-taking into small time intervals over which conditions may be treated as approximately constant
- This is one of the factors that complicates (adds interest to?) ATLAS processing as just another SPMD application
- At ATLAS (and HPC) scales (but functionally at any scale) this matters
  - Different nodes processing events from different time intervals may require different conditions data
  - And even a single node may find that the next event to be processed requires different conditions data
  - So one cannot quite initialize all nodes with exactly the same auxiliary data and be done
- Getting the metadata and bookkeeping and its propagation and associations right is also a nontrivial consideration
  - Not just "did we process all of the events"
  - ATLAS is reconsidering how to handle this better (we can talk more about this later if it seems worthwhile to do so)

High-level I/O components

- Event Selector: the means through which events are read
  - Connects a job to an event source
- Outstream: the means through which events are written
  - Connects a job to an event sink
- (Less visible to users) Converters and conversion services provide support for reading and writing objects of specific types

Event Selector

- Connects the event-processing framework to an event source
  - Source is generally but not necessarily a file or list of files
- In event-loop-based processing, its primary role is to implement next():
  - makes the contents of (objects in) the next input event available to the algorithms that will process them
  - "makes available" can mean triggering actual retrieval of input event data objects
  - Or it can more simply mean populating the transient store with "pointers" to the relevant data in support of something closer to on-demand retrieval
- Event selectors can select (filter), though often they simply iterate through the attached input
  - Most commonly filtering, if any, is done only on run and event numbers, or by choosing the Mth through Nth events in a file
  - But in principle next() may mean "next event that satisfies a filter predicate"
- Event Selector has other duties, too
  - Managing what happens on file boundaries (processing, metadata, …)
- And their are many ancillary services not described here

Outstreams

- Outstreams connect a job to an event sink
  - Most often a file
- Configured with a wide range of information related to persistence, either for the outstream's own use or to pass to the underlying persistence technology
  - File names, compression choices, persistent data layout hints, commit intervals, high-water marks, …
- Configured as well with the list of names of transient data objects to be written
- May have multiple outstreams in a job
  - Events meeting different criteria may be sent to different outstreams
  - Possibly with output content that may differ by stream

Converters and conversion services

- The ATLAS transient event data model is defined in C++
- The role of conversion services and their converters is to provide a means to write C++ objects to storage and read them back
- General model is that converters are type-specific
  - Though there are ways to deal with commonalities
  - And one may want to do this for objects of different derived types that share one or more base classes, as just one example
- Converters are how one chooses a persistent representation, and conversely how one builds a transient data object from such a representation
  - Clearly there is more than one way to serialize a given object's state, particularly for non-trivially-structured objects, and there are storage and performance reasons why one might care
  - And even for serialization of state, one can imagine choosing different representations for different purposes or different technologies
- Converters are the locus for schema evolution
  - A place to put code to read old versions of persistent data and create the latest versions of transient event data objects, and more

Conceptual model: converters and their roles

- Can imagine an outstream simply iterating over the list of objects to be written and calling their (type-dependent) converters to do the writing
  - Potentially in two steps: one to write, the other to handle inter-object pointers, though ATLAS has other ways to handle references as well
- Converter finds the object in the transient store, writes it, and returns a "reference" to the persistent data
  - "Reference" contains sufficient information to locate and describe the data in persistent storage for later retrieval
- What one does with such references Is a basis for an event store navigational model
- Similarly on input: can imagine an event selector ensuring that such references to input data are "registered" in a transient store so that retrieval of an object with a given name/key* is possible

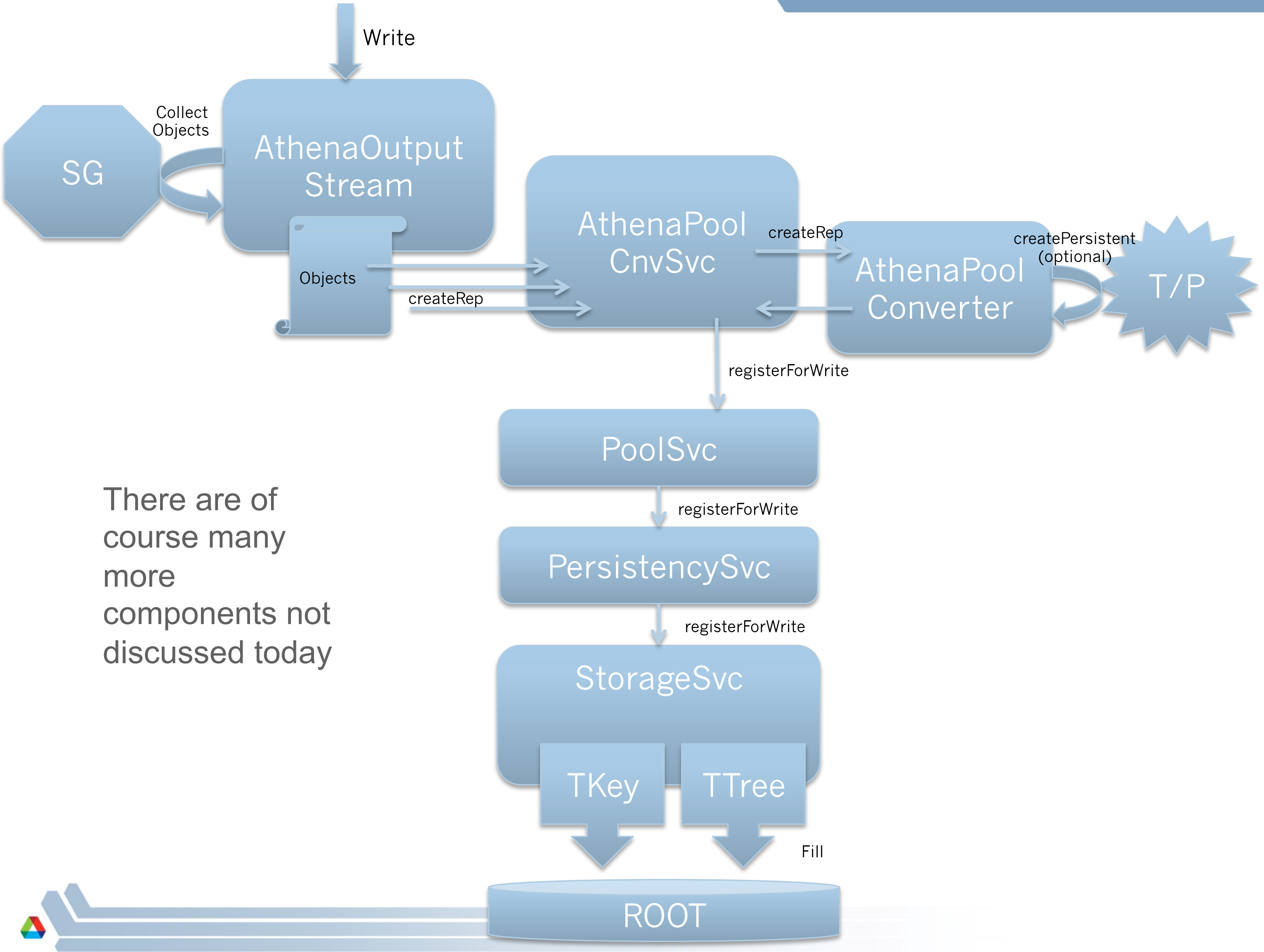
  * (or even a different name/key, with a valid reference to persistent data of a suitable type)

Event store navigation

- Recall that objects in the transient store also have a name (key)
- After storing all objects, ATLAS "remembers" where it put them by recording their names/keys and their locations (via the returned references) in a DataHeader object
  - Which also maintains a provenance record that points to upstream processsing stage
- A reference to this DataHeader object is used in turn in event selection metadata systems to identify where within the ATLAS distributed data store (hundreds of petabytes) to find this event
- Current navigational model is more general than this, with more capabilities than ATLAS uses (or needs today)
  - Example: putting event data objects needed by only a small number of downstream users into a different file than most of the data
    - Replicating such data this less frequently, and reducing transfer of unneeded data to a site or job
  - Example: real-time back navigation and access to upstream data
- Under review: we may simplify

Persistent event data model

- Design of current conversion services foresaw the need to support the full expressive power of C++ object definition
  - Arbitrarily complex and heterogeneous objects could be part of the event data model, and they were: in Run 1
- ATLAS greatly reduced the complexity of its data model for Run 2 (the current run), and adopted a much more homogeneous strategy for event data object state implementation
- Concurrently, ROOT's ability to support C++ data object persistence developed and matured
  - Lessening the need for ATLAS-specific smart converters
- In practice, current converters create persistent representations highly aligned with the xAOD and its AuxStore-based transient data model
- Efforts toward simplification and more direct leveraging of ROOT I/O are in progress

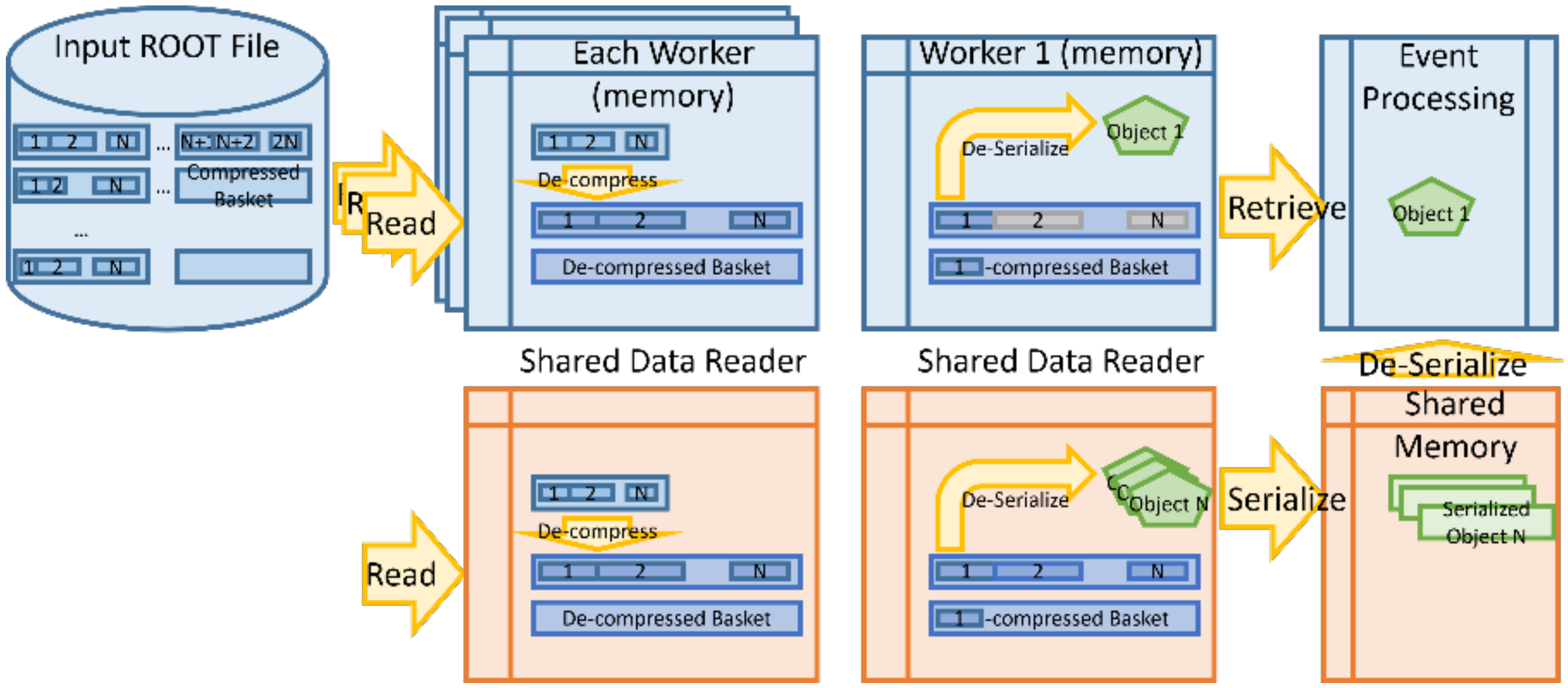


There are of course many more components not discussed today

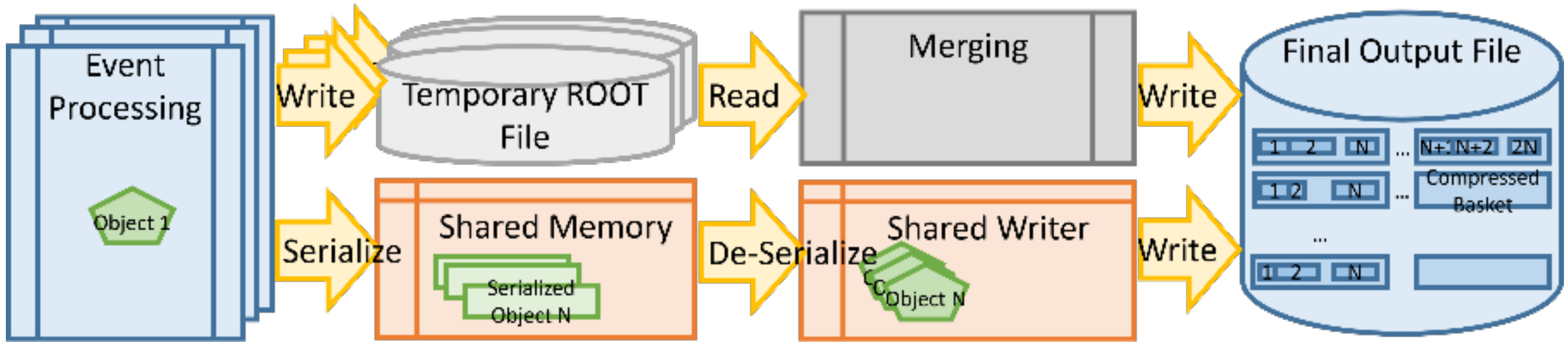
Shared I/O components

- The Athena framework may be run in multiprocessing mode (AthenaMP)
- Fork N identical workers after initialization or later
  - As late as possible to maximize memory sharing
- Each process *could* read its own file
  - If there are enough input files (and there are tradeoffs here)
- Each process *could* read from the same file
  - Contention and needless work (all processes decompressing the same buffers, etc.)
- Shared reader: one process handles reading and distribution of input
- Each process could write its own file
  - Result: lots of memory consumption, lots of small files that must be merged later, …
- Each process could write to the same file
  - Ouch. Seriously?
- Shared writer: one process handles writing

Unshared and shared reading



Unshared and shared writing



Content thanks to the ROOT team, and especially Philippe Canal

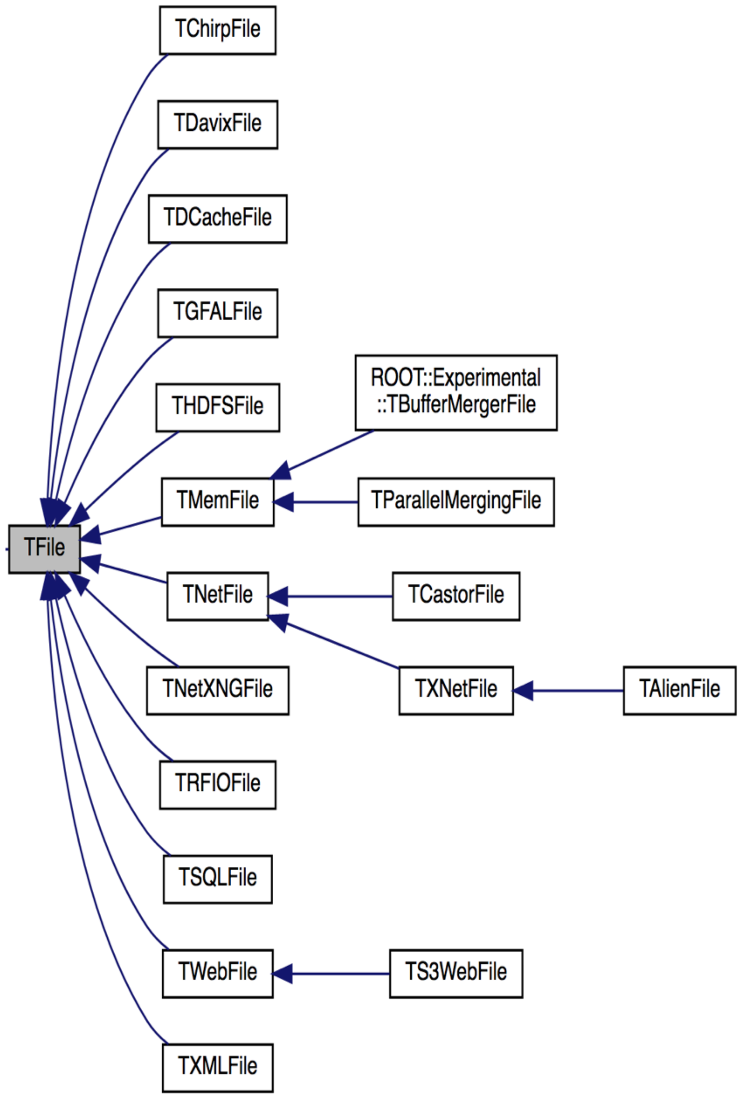
Begin interlude: ROOT

The ROOT File

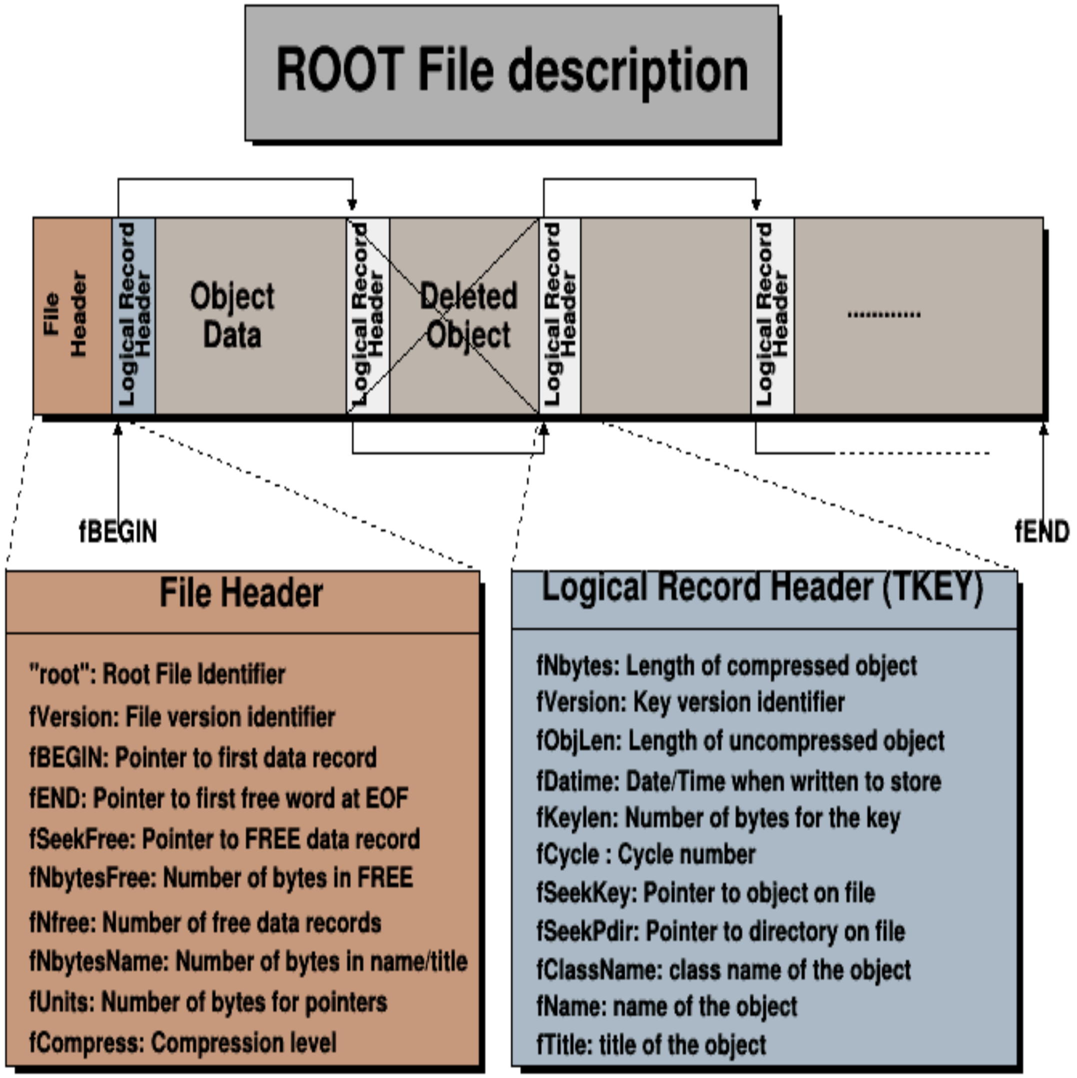
- In ROOT, objects are written in files*
- ROOT provides its file class: the **TFile**
- TFiles are *binary* and have: a *header*, *records* and can be compressed (transparently for the user)
- TFiles have a logical “file system like” structure
  - e.g. directory hierarchy
- **TFiles are self-descriptive**:
  - Can be read without the code of the objects streamed into them
  - E.g. can be read from JavaScript

* this is an understatement

Flavour of TFiles



ROOT File Description



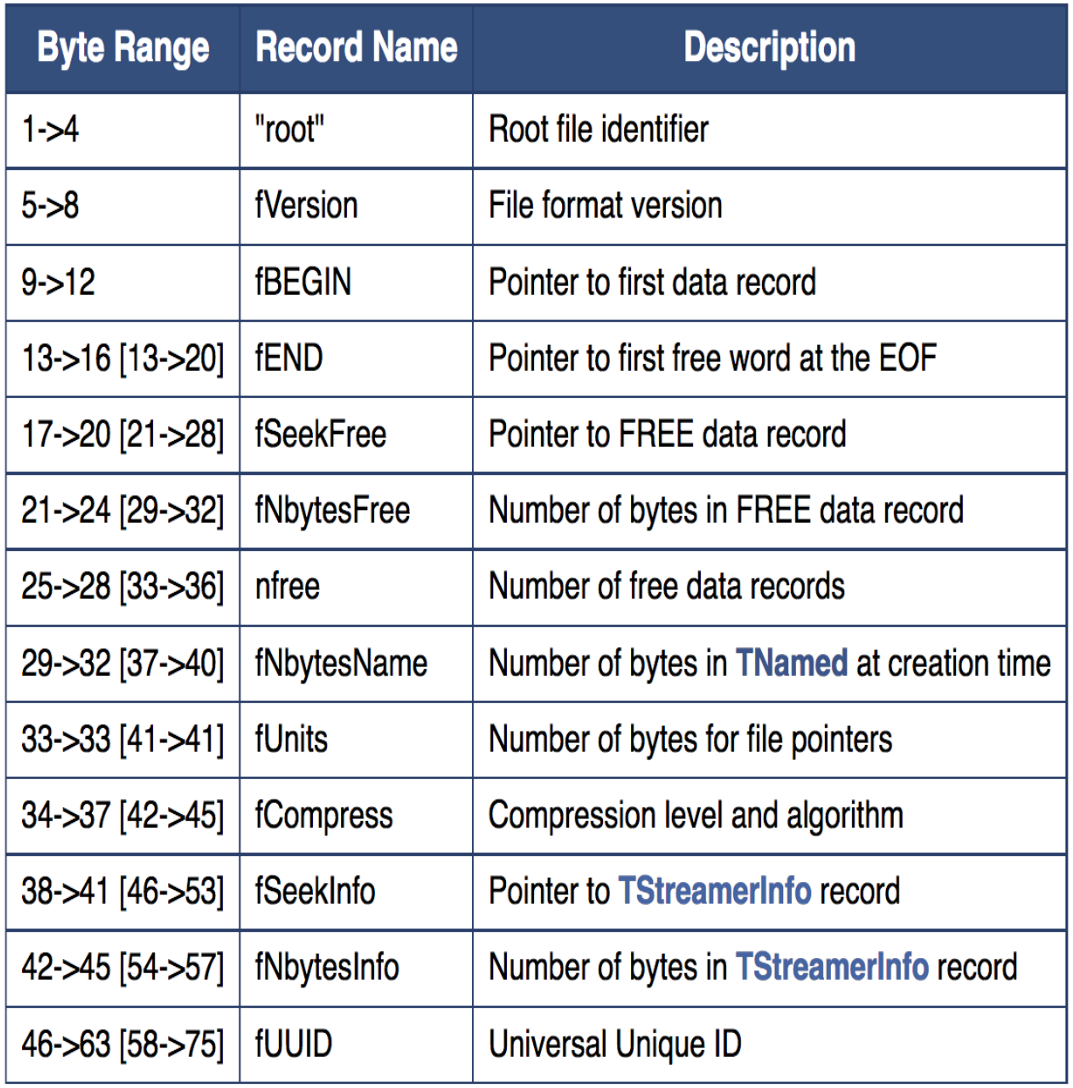
File Header

"root": Root File Identifier
fVersion: File version identifier
fBEGIN: Pointer to first data record
fEND: Pointer to first free word at EOF
fSeekFree: Pointer to FREE data record
fNbytesFree: Number of bytes in FREE
fNfree: Number of free data records
fNbytesName: Number of bytes in name/title
fUnits: Number of bytes for pointers
fCompress: Compression level

Logical Record Header (TKEY)

fNbytes: Length of compressed object
fVersion: Key version identifier
fObjLen: Length of uncompressed object
fDatime: Date/Time when written to store
fKeylen: Number of bytes for the key
fCycle : Cycle number
fSeekKey: Pointer to object on file
fSeekPdir: Pointer to directory on file
fClassName: class name of the object
fName: name of the object
fTitle: title of the object

A Well Documented File Format



Byte Range	Record Name	Description
1->4	"root"	Root file identifier
5->8	fVersion	File format version
9->12	fBEGIN	Pointer to first data record
13->16 [13->20]	fEND	Pointer to first free word at the EOF
17->20 [21->28]	fSeekFree	Pointer to FREE data record
21->24 [29->32]	fNbytesFree	Number of bytes in FREE data record
25->28 [33->36]	nfree	Number of free data records
29->32 [37->40]	fNbytesName	Number of bytes in TNamed at creation time
33->33 [41->41]	fUnits	Number of bytes for file pointers
34->37 [42->45]	fCompress	Compression level and algorithm
38->41 [46->53]	fSeekInfo	Pointer to TStreamerInfo record
42->45 [54->57]	fNbytesInfo	Number of bytes in TStreamerInfo record
46->63 [58->75]	fUUID	Universal Unique ID

How Does it Work in a Nutshell?

- **C++ does not support native I/O** of its objects
- Key ingredient: reflection information - **Provided by ROOT**
  - What are the data members of the class of which this object is instance? I.e. How does the object look in memory?
- The steps, from memory to disk:

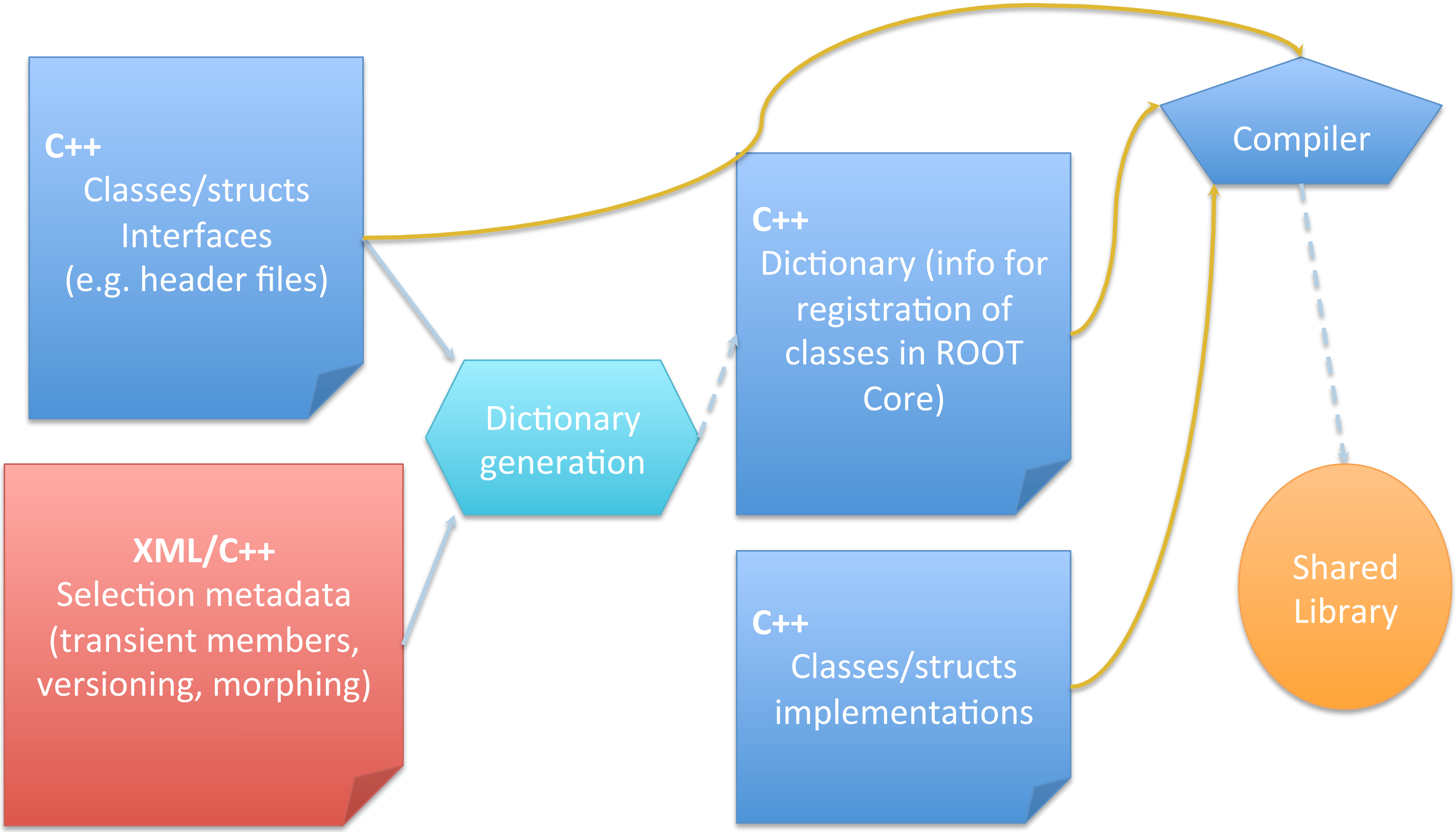
1. Serialisation: from an object in memory to a blob of bytes
2. Compression: use an algorithm to reduce size of the blob (e.g. zip, lzma, lz4)
3. "Real" writing via OS primitives

Serialisation: not a trivial task

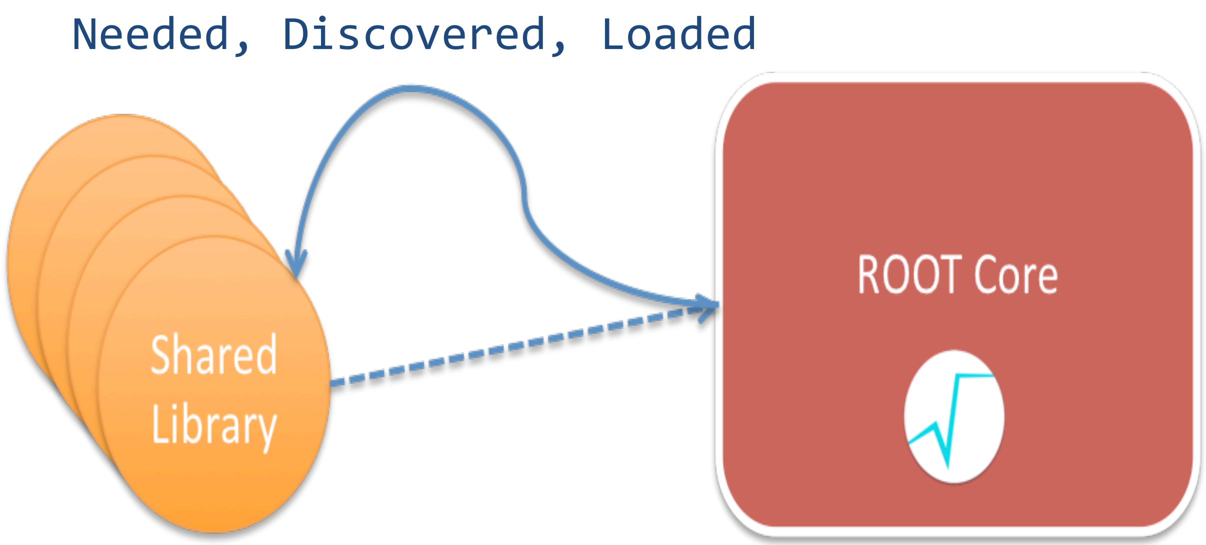
For example:

- Must be platform independent: e.g. 32bits, 64bits
    - Remove padding if present, little endian/big endian
- Must follow pointers correctly
    - And avoid loops ;)
- Must treat stl constructs
- Must take into account customisations by the user
    - E.g. skip “transient data members”

Persistency



Injection of Reflection Information

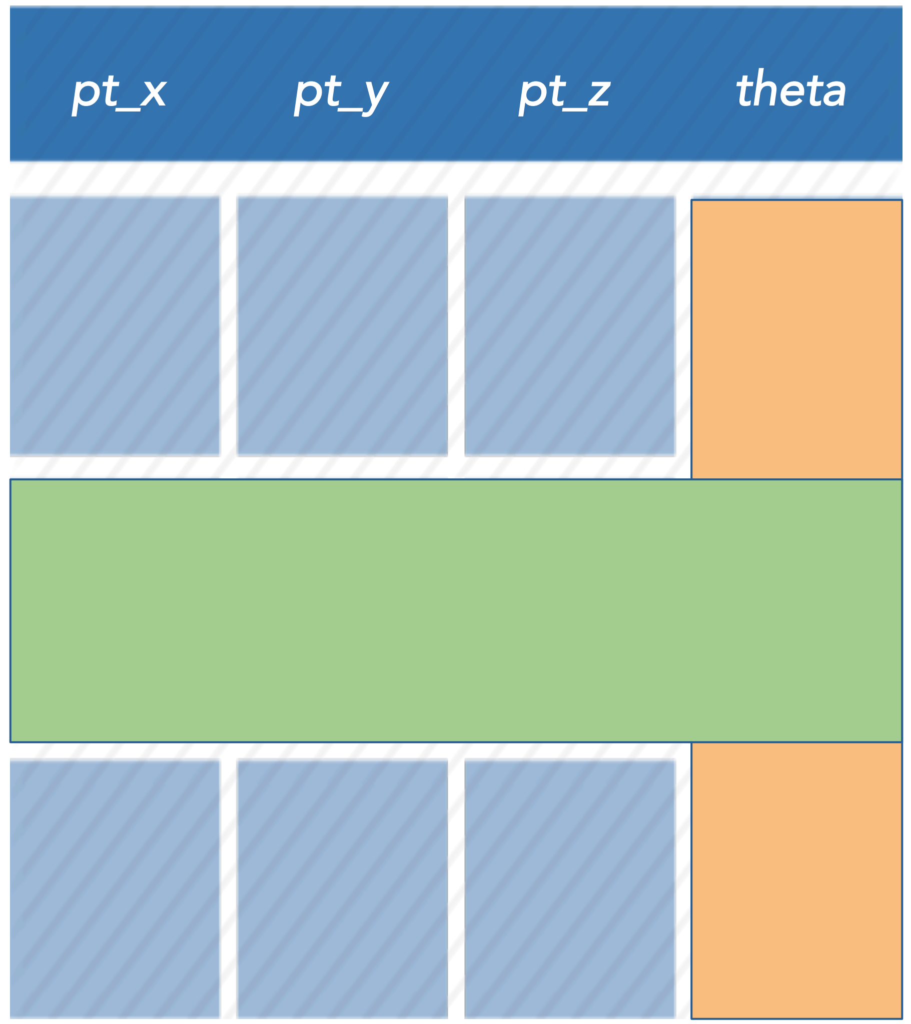


Now ROOT “knows” how to serialise the instances implemented in the library (series of data members, type, transiency) and to write them on disk in row or column format.

Columns and Rows

- High Energy Physics: many statistically independent *collision events*
- Create an event class, serialise and write out N instances on a file? No. Very inefficient!
- Organise the dataset in **columns**

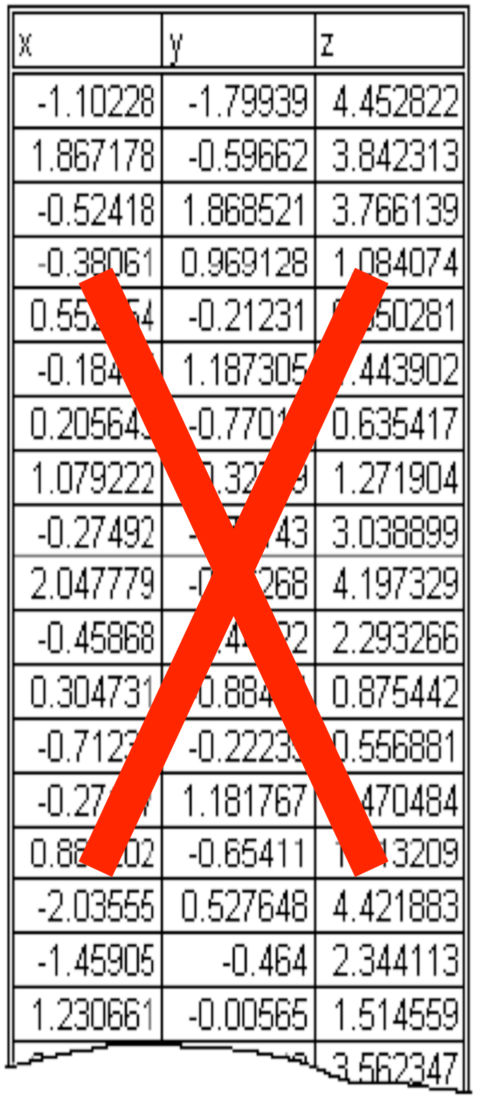
Columnar Representation



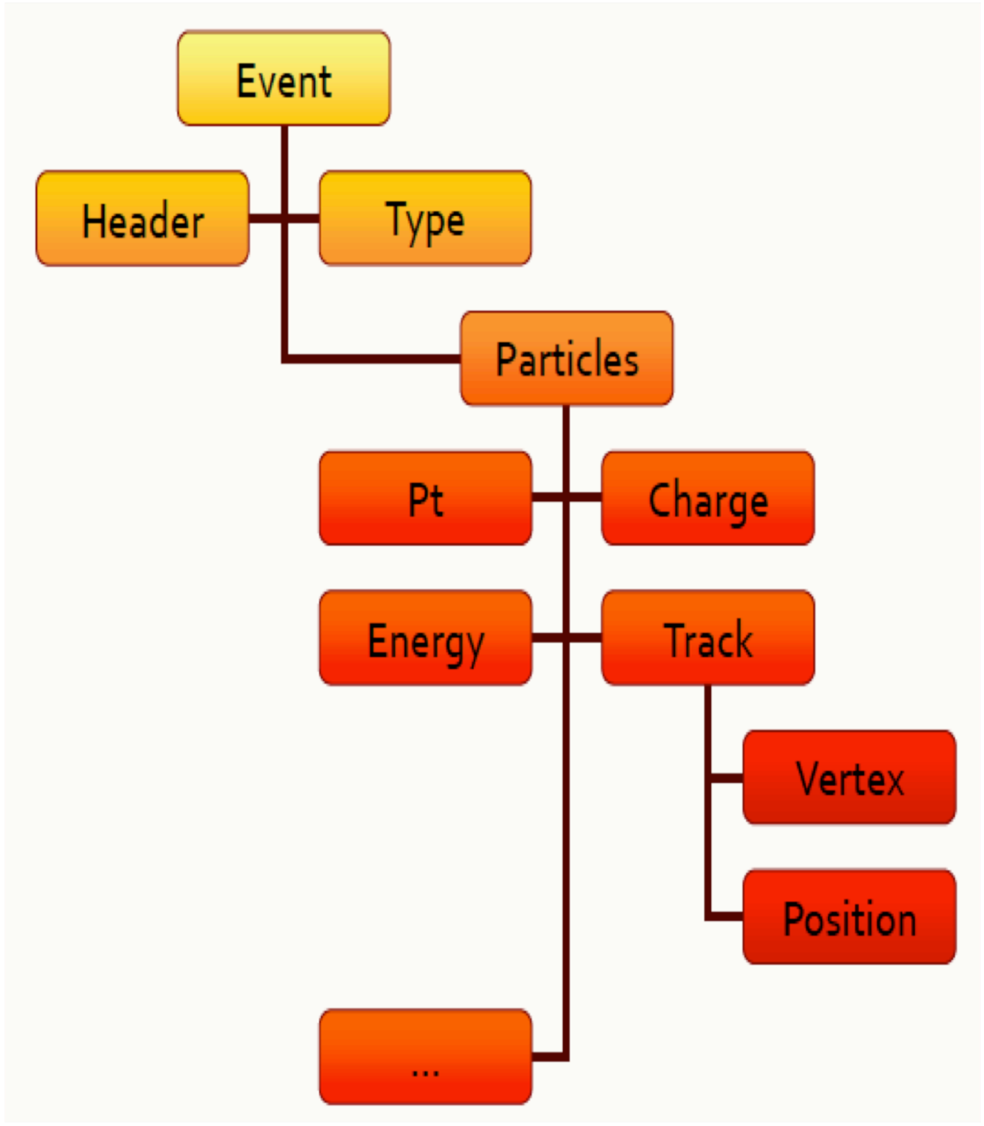
entries
or events
or rows

columns
or "branches"
can contain any
kind
of c++ object

Relations Among Columns



x	y	z
-1.10228	-1.79939	4.452822
1.867178	-0.59662	3.842313
-0.52418	1.868521	3.766139
-0.38061	0.969128	1.084074
[illegible]	-0.21231	[illegible]
[illegible]	1.187305	[illegible]
[illegible]	[illegible]	0.635417
1.079222	[illegible]	1.271904
-0.27492	[illegible]	3.038899
2.047779	[illegible]	4.197329
-0.45868	[illegible]	2.293266
0.304731	[illegible]	0.875442
[illegible]	[illegible]	[illegible]
[illegible]	1.181767	[illegible]
[illegible]	-0.65411	[illegible]
-2.03555	0.527648	4.421883
-1.45905	-0.464	2.344113
1.230661	-0.00565	1.514559
[illegible]	[illegible]	3.562347



Optimal Runtime and Storage Usage

Runtime:

- Can decide what columns to read
- Prefetching, read-ahead optimisations possible

Storage Usage:

- Run-length Encoding (RLE). Compression of individual columns values is very efficient
  - Physics values: potentially all "similar", e.g. within a few orders of magnitude - position, momentum, charge, index

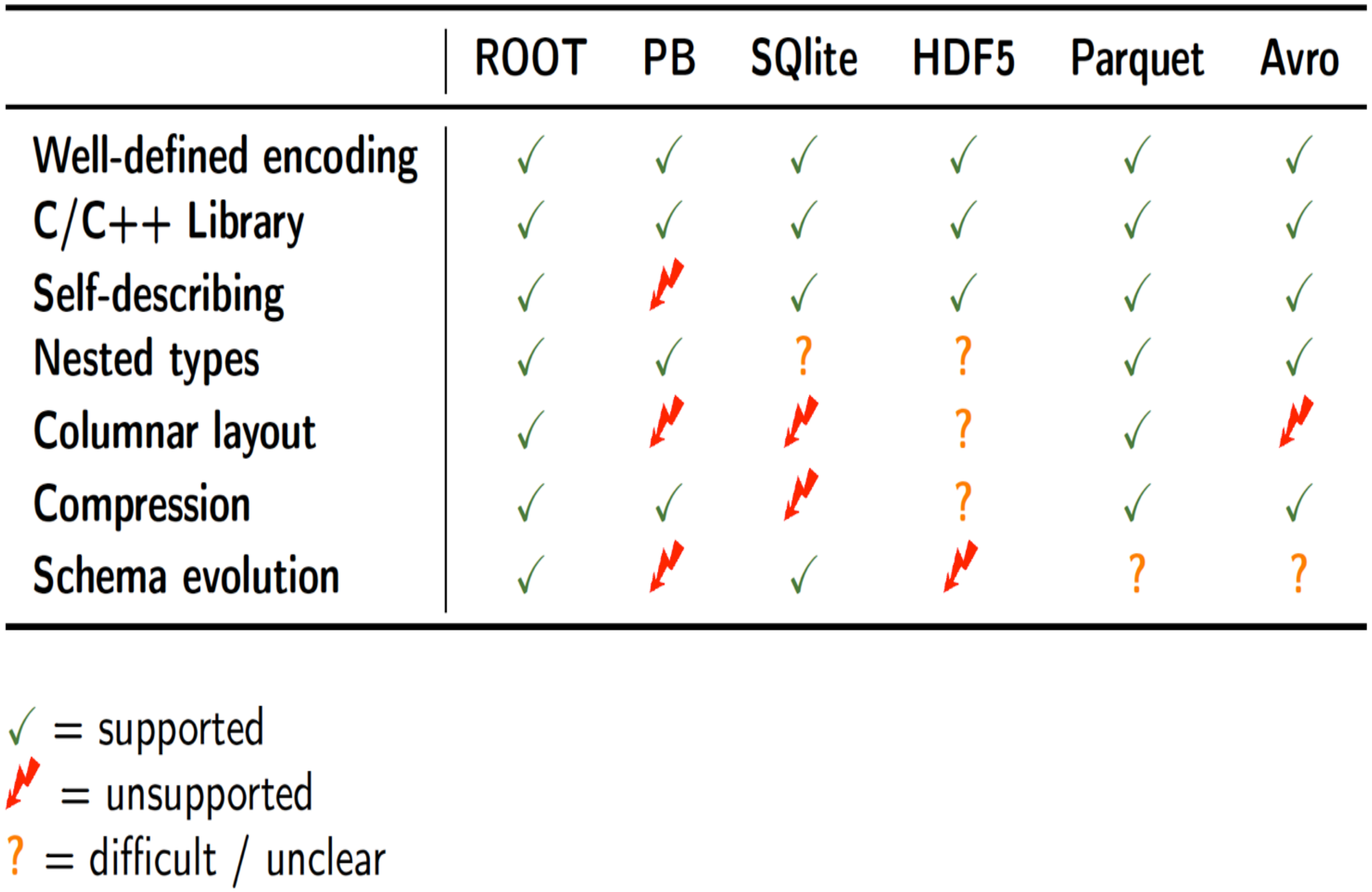
The TTree

A columnar dataset in ROOT is represented by **TTree**:

- Also called *tree*, columns also called *branches*
- An object type per column, **any type of object**
- One row per *entry* (or, in collider physics, *event*)

If just a **single number** per column is required, the simpler **TNtuple** *can* be used.

Comparison With Other I/O Systems



	ROOT	PB	SQlite	HDF5	Parquet	Avro
Well-defined encoding	✓	✓	✓	✓	✓	✓
C/C++ Library	✓	✓	✓	✓	✓	✓
Self-describing	✓	⚡	✓	✓	✓	✓
Nested types	✓	✓	?	?	✓	✓
Columnar layout	✓	⚡	⚡	?	✓	⚡
Compression	✓	✓	⚡	?	✓	✓
Schema evolution	✓	⚡	✓	⚡	?	?

✓ = supported
⚡ = unsupported
? = difficult / unclear

More on technology comparisons

- See the same paper for performance comparisons of ROOT to other systems
- J. Blomer, A quantitative review of data formats for HEP analyses ACAT 2017

End interlude: ROOT

Some starting points for possible collaborations?

- **Getting data onto and off of a large number of** high-performance computational **nodes** efficiently will be essential to effective exploitation of HPC architectures.
- There are a number of facets to such work that could be undertaken separately or together. .
- There are a variety of ways to support such models
  - I/O concentrators on output, for example, to which individual worker nodes would send their data to be merged and later written to storage by a much smaller number of workers.
  - Input is comparable in principle, e.g., with a relatively small number of workers ("shared readers") reading from persistent storage and distributing input data to many more workers
- Far from groundbreaking conceptually, but
  - There are many design choices to be made, and nontrivial development to be done
  - Possibility of collaboration between components that operate inside and outside an event processing framework (or not)

Efficient I/O for HPCs

- A number of possible implementation strategies to be investigated
  - e.g., integrating message passing with serialization strategies and extensions of current shared readers and writers
  - With merging provided by ROOT or by our own code
- Shared I/O components already in production (e.g., in AthenaMP (ATLAS framework in multiprocessing mode) and the I/O components already supporting multithreaded processing (AthenaMT) provide a solid foundation for such work
- A look at integrating current ATLAS shared writer code with MPI underway at LBNL
  - Buffers filled by workers are sent to writer via message passing rather than shared memory
  - But this is just one possible strategy
- Related work (TMPIFile with synchronization across MPI ranks) by a summer student at Argonne ongoing
- Efficient access on HPCs to time-varying conditions and other non-event data needed for event processing is another area where work is needed
  - May differ from node to node and vary over the course of a job

Persistent data organization

- Can we optimize persistent data organization for HPC use cases? Should we?
- Current ATLAS data organization is designed principally for serial processing, with columnar data access
- For other processing models there are more efficient ways to organize data on storage, both for feeding discrete chunks to many independent processes and for receiving data from many independent processes with greatly reduced contention on the storage side
- Other scenarios: may want to get data off of processors as fast as possible and worry about storage footprint optimization only later
- The promise of smart server-side data access and filtering (with the event streaming service as just one example) and the data reorganization capabilities present in "data lake" models will allow ATLAS to expand its range of potential persistent data organizations and representations, and to tune them to a range of workflows

Event-level, object-level, attribute-level serialization

- ATLAS already employs a serialization infrastructure
  - for example, to write high-level trigger (HLT) results
  - and for communication within a shared writer implementation
- A unified approach to serialization that supports, not only event streaming, but data object streaming to coprocessors, to GPUS, and to other nodes, would be a boon.
- ATLAS takes advantage of ROOT-based streaming (which, importantly, supports schema description for streamed data).
- An integrated but lighter-weight approach for streaming data more directly (structs of floats contiguous in memory, for example, as a block of bytes) would allow us to exploit co-processing more efficiently
  - Note that "ROOT-based" and "lighter-weight" may eventually prove not to be mutually exclusive (cf. the zero-copy work of the DIANA project)
- Could imagine integrating an approach to streaming transient data to coprocessors with a pipeline from persistent storage, conceptually speaking
  - In any case, probably do not want two different approaches to serialization that are unaware of each other and reuse nothing

An I/O component could make sense in many possible collaborative efforts

- One attractive feature of any I/O and I/O optimization work is that these areas of development or any subset of them could be one useful and important component of a program of work no matter what "algorithmic hot spot" we eventually select