ROOT Core I/O and TTree STATUS

Philippe Canal and Jakob Blomer for the ROOT team

ROOT
Data Analysis Framework
https://root.cern

What’

- New Features in Core I/O
- Other Improvements in Core I/O
- Concurrency Improvements
- TTree Improvements
- Future Development in Core I/O

New Features in Core I/O

- Support for ***ZSTD*** and ***LZ4*** compression algorithms.
- Better interface to retrieve object from a ***TFile***:

```
auto obj = directory->Get<MyClass>("some object");
```

- Support for ***XRootD*** local redirection.
- Support for maps with string as a key in ***JSON*** output
- Creation of fully reproducible ***TFile***

```
TFile *f = TFile::Open("name.root?reproducible","RECREATE","File title");
```

  - No date info in keys and directory. No ***TUUID***.
  - But no support for ***TRef*** in such files.

Other Improvements in Core I/O

- Several deficiencies solved in I/O customization rules
- Significantly improved the scaling of hadd tear-down/cleanup-phase in the presence of large number histograms and in the presence of large number of directories.

Concurrency Improvements

- Many improvements including
  - Thread scalability of ***TRef***, ***TStreamerInfo***
  - Exclusive use of the global lock is strongly reduced or migrated to finer grained read and write locks
  - Scaling and stability of ***TBufferMerger*** feature.
  - Addition of ***TMPFile*** implement the file merging over ***MPI***.
- Streaming now scales linearly with number of threads
  - ***TFile*** has a fully scalable mode (essentially no locks)
  - ***TTree*** with ***TBufferMerger*** can be challenged by very high input rate. Reading of single ***TTree*** limited by decompression of largest buffers

TTree Improvements

- Allow creation of TTree with strictly one basket per branch per cluster:

```
tree->SetBit(TTree::kOnlyFlushAtCluster);
```

  - ***TTree*** leaflist extended to 'f' (***Float16_t***), 'd' (***Double32_t***) , ‘G’ (long) and ‘g’ (unsigned long)

- Bulk I/O
  - New interface to read whole basket of data a time
  - Currently resolving issues related to indices and content not fitting in the same number of baskets.

Future Development in Core I/O

1. Further **Thread-safety** and performance improvements
2. TBufferFile larger than **1GB**
3. Further Schema Evolution Improvement
4. Incorporate **lossy** compression engine (Accelogic)

Backup slides

- Speed-up startup, in particular in case of no or poor network accessibility

File Format Essential Properties

Robustness	Protection against media failure & API misuse
Expressiveness	Support for events with nested variable length collections
Speed	Columnar layout, merge-friendly, sophisticated I/O scheduling
Stability	Backwards and forwards compatibility, hooks for schema evolution
Usability	Accessible to novice and expert programmers
Concurrency	Facilitate concurrent reading/writing (merging) and (de-)compression
Integration	Support for HEP-specific, HPC, and Cloud storage and data mgmt systems

Facets of a full I/O system

In addition to deserializing file contents, the full I/O system has many more aspects, such as

- Parallel and distributed reading & writing
- I/O scheduling (read-ahead, request coalescing, etc)
- Beyond file system I/O: HTTP, XRootD, object stores
- Schema evolution
- Data set combinations: chains, friends, indexes, merging
- Complex object hierarchies (e.g. for ESD EDMs)
- User customizations
  - E.g. skip “transient data members”
  - I/O customization rule (transformation of data)

HEP Event Data I/O

Why invest in a **tailor-made I/O system**

TTree & RNTuple

- Capable of storing the **HEP event data model**: nested, inter-dependent collections of data points
- **Performance-tuned** for HEP analysis workflow (columnar binary layout, custom compression etc.)
- **Automatic schema** generation and evolution for C++ (via cling) and Python (via cling + PyROOT)
- Integration with **federated data management** tools (XRootD etc.)
- Long-term **maintenance** and support

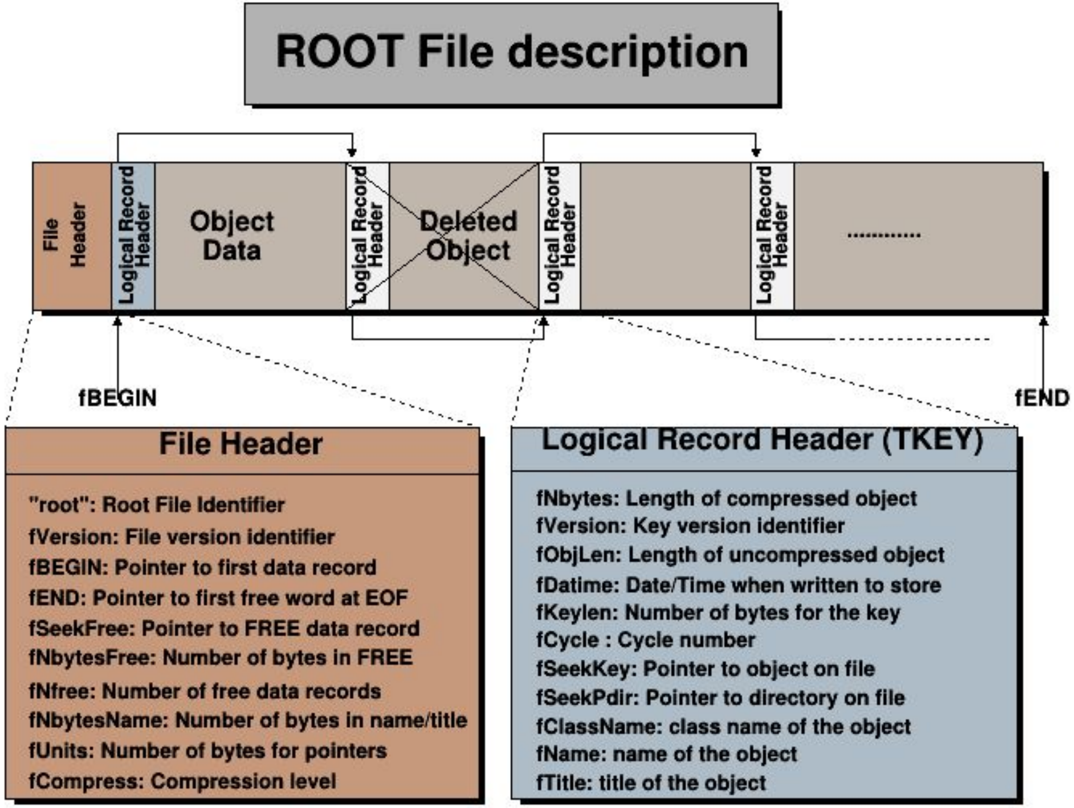
Example EDM

```
struct Event {
    std::vector<Particle> fPtcls;
    std::vector<Track> fTracks;
};

struct Particle {
    float fPt;
    Track &fTrack;
};

struct Track {
    std::vector<Hit> fHits;
};

struct Hit {
    float fX, fY, fZ;
};
```

The ROOT File

- In ROOT, objects are written in files (“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

ROOT File Description



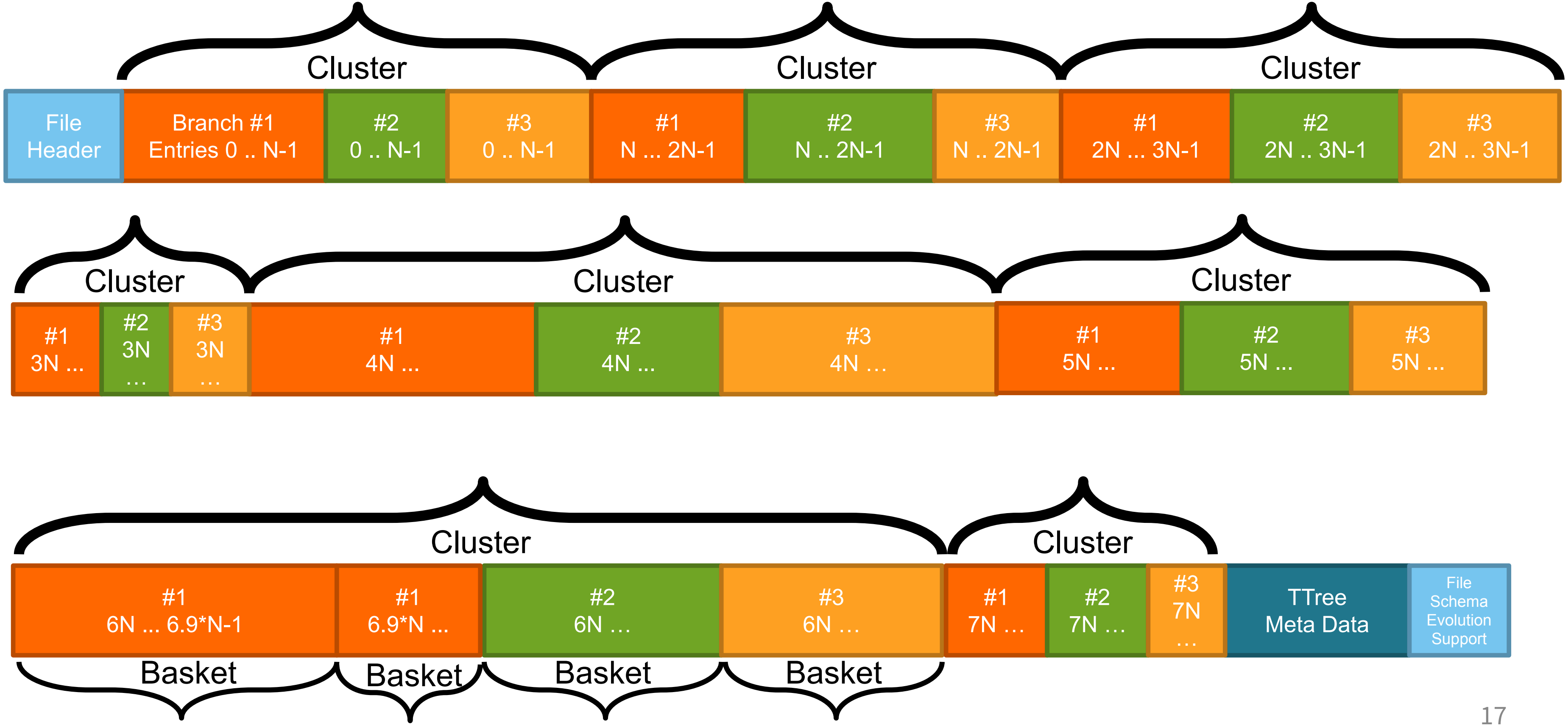
ROOT File Specification

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

Event Data and ROOT Files

- A ROOT file can be seen as a hierarchically organized container of objects
  - E.g. a file can contain directories with histograms
- In addition, ROOT files can also contain event data
  - E.g., a series of `TEvent` objects for a user-defined `TEvent` class
- Event data stored in a TTree (or RNTuple, see later) is usually written as a set of many objects
- TTree and RNTuple have a custom, internal serialization format (columnar layout)
- A binary format within the TFile binary format

Anatomy of a Tree



ROOT Data Access Options

- ROOT can read, write, and represent data in C++
- ROOT can read, write, and represent data in Python through pyROOT (dynamic binding between C++ and Python)
  - Can also export ROOT trees to numpy arrays
- ROOT can read and represent trees and the most common classes (histograms, graphs, etc.) in JavaScript with JSROOT
  - Can also export objects in JSON

3rd Party Implementations of ROOT I/O

- There are several projects that re-implement parts of the ROOT file format
  - Julia: unroot
  - Python: uproot
  - Go: hep/groot
  - Java/Scala: FreeHEP rootio
  - Rust: alice-rs/root-io

- Typically supported features: reading of simple objects (histograms) and trees with a simple structure (numerical types and vectors thereof)

ROOT I/O: Support

Full support by the ROOT Team:

- I/O through the ROOT C++ library
- pyROOT
- Conversion of simple structures to numpy arrays
- JSROOT
- JSON serialization of objects
- In the future: C API provided by RNTupleLite

Indirect support (“support the maintainers”)

- Third-party implementation of the binary format (uproot, unroot, Java, Go, ...)