Upcoming Storage Features in ROOT

Philippe Canal and Jakob Blomer for the ROOT team
Snowmass CompF4 Topical Group Workshop, April 2021
Major I/O upgrade of the event data file format and access API: TTree → RNTuple
ROOT Foundation Upgrade for HL-LHC

- **Major I/O upgrade** of the event data file format and access API: TTree → RNTuple
  - Target an **order of magnitude higher event throughput** (storage to compute)
  - Give access to **novel and future storage technologies**

- **Generation hand-over** of I/O experts to ensure availability of I/O expertise compatible with the HL-LHC lifetime

- Est. 50 MCHF/year on storage in WLCG
  - Strong incentive for common, highly efficient I/O layer

- **New** generation of **hardware architecture** (GPU, HPC, Object Stores, etc.)
- TTree and RNTuple: ROOT classes for **columnar storage** of event data
  - Optimized for selective reads as is typical in analysis
  - Since 25 years in ROOT, today a common standard in Big Data tools

- Support for **complex objects and nested collections** within events
- Assisted by cling: **Seamless C++ and Python integration**: no hand-coded data schema

entries or events or rows

columns
can contain complex c++ objects

pt_x pt_y pt_z theta
Overview of Foundation Components

ROOT File (local and remote): **TFile** container format hosting data (TTree, RNTuple) and summary objects (TH1 etc.)

**TTree**: $O(1\text{EB})$ Run 1 to Run 3 data

**RNTuple**: $O(10\text{EB})$ Run 4-6 data

Remote file access: **XRootD, Davix** for HTTP, X.509 and SciToken authentication

**Cling**: C++ and Python reflection for user-defined object, common AoS → SoA object mapping

**Object store adapters** for cloud and HPC (e.g., DAOS, S3)

**In-memory adapters** (e.g., numpy, Arrow)

- **Plus**: compression schemes, caching, merging, data movement
- e.g. to develop readers in Go, Julia, ...

Large-scale format transition

RNTuple PoC, first exposure to experiments
Based on 25+ years of TTree experience, redesign of the I/O subsystem for

- Less disk and CPU usage for same data content
  - 10-20% smaller files, at least x3-5 better single-core performance
  - 10 GB/s per node and 500 MB/s per core sustained end-to-end throughput (compressed data to histograms, based on current HW generation)
- Native support for HPC and cloud object stores
- Lossy compression
- Systematic use of checksumming and exceptions to prevent silent I/O errors

Full control of the I/O layer enables fast adaptation to HEP-specific needs, such as

- Tight RDataFrame integration
- Support for rich event data models (EDMs)
- Rich metadata: e.g., scale factors, data management information
- Vertical and horizontal joins (“friends”, “chains”, ...)
- Fast merging of data streams
- Good integration with multi-threaded frameworks
- Support for code & data evolution over decades

Performance and functionality unmatched by any other available data format / API
RNTuple: Current performance snapshot

RNTuple is both significantly faster and has best data compression efficiency.

**RDF Analysis Prototype:**

With Distributed RDF reached 50 GB/s using 1024 core of CERN HPC.

With DAOS reached 70% of theoretical bandwidth of the cluster (36.5GB/s out of 48 GB/s).
RNTuple Development Plan

~2018-19
Proof of concept
- Architecture
- Review on state-of-the-art
- First prototypes

~2019-20
Prototype
- Adoption in ROOT::Experimental
- I/O scheduler for local and remote access
- Performance validation

~2021-22
First exploitation
- Object store support
  - DAOS (HPC)
  - S3 (Cloud)
- RNTuple version 1 spec
- RNTupleLite
- Schema evolution
- Disk-to-disk conversion
- Virtual data sets for skims and selections
- First exposure to frameworks:
  - CMSSW nanoAOD output module
  - Prototyping by ATLAS, CMS, LHCb I/O experts

~2022-23
Pre-production
- RDataFrame bulk processing
- Debugging and inspection tools
- Metadata API
- Special use case support: e.g. backfill, in-memory adapters
- XRootD support
- Validation of feature coverage
- Training experiments’ core developers
- Large-scale experiment benchmarks
- PB scale tests
- Automatic optimization features
- Low-precision floats
- ML Training: direct GPU transfer
- End-user training
- Training and support for code and data migration

~2023-24
Production

Growing importance of coordination & collaboration with experiment I/O experts

Expecting stable, if not increasing, I/O workload well into Run 4
1. Support

2. **Thread-safety** and performance improvements

3. TBufferFile larger than **1GB**

4. Schema Evolution Improvement

5. Incorporate **lossy** compression engine (Accelogic)
1. Keeping the schedule of the RNTuple implementation plan
   ○ Risks fractured I/O landscape of ad-hoc solutions, likely resulting in increased storage needs, reduced compute efficiency, and failure in long-term data preservation
   ○ Stable support for 2.5 FTEs until 2025 on TTree, RNTuple, and experiment framework expertise
   ○ Gradual RNTuple rollout from AODs to RAW for agile adjustment of development efforts

2. Long-term retention of TTree and RNTuple I/O experts
   ○ Risks trust erosion and inefficiencies due to work-arounds
   ○ Mitigated by thorough development and documentation discipline
   ○ Mitigated by existing permanent positions in I/O

3. Design of RNTuple meeting the Run 4 hardware and software requirements
   ○ Risks limitation of HL-LHC computing workflows, in the worst case partial loss of data
   ○ Mitigated by early involvement of experiments in the RNTuple design and format specification
   ○ RNTuple designed informed by years of TTree experience
   ○ Large-scale validation tests
4. Continued support of 3rd party libraries
   ○ **Risks limitations of computing workflows involving remote I/O and AuthX**
   ○ Continued community funding for XRootD and Davix (HTTP) library developers

5. Adoption of RNTuple through experiment and analysis framework adaptations and optimized data models
   ○ **Risks mismatch between experiments’ data model and RNTuple main format and API, thus fractured landscape with significant maintenance support for both RNTuple and TTree**
   ○ Mitigated by investment on both ROOT side and experiment side for close feedback loops
   ○ Seamless analysis code migration through RDataFrame
   ○ We believe that the benefits of RNTuple warrant transition with high priority

6. Evolving ROOT reflection support (cling)
   ○ **Risks limitations in the EDMs due to lack of I/O support for language features**
   ○ Mitigated by stable positions for experts on clang/clang and llvm
Backup slides
Motivation for Investment in I/O

1. HL-LHC data challenge:
   - From $300fb^{-1}$ in run 1-3 to $3000fb^{-1}$ in run 4-6
   - 10B events/year to 100B events/year
   - Real data challenge depends on several factors: number of events, analysis complexity, number of reruns, etc.
     - As a starting point, preparing for ten times the current demand

2. Full exploitation of modern storage hardware
   - Ultra fast networks and SSDs: 10GB/s per device reachable (HDD: 250MB/s)
   - Flash storage is inherently parallel → asynchronous, parallel I/O key
   - Heterogeneous computing hardware → GPU should be able to load data directly from SSD, e.g. to feed ML pipeline
   - Distributed storage systems move from POSIX to object stores

Blurring between I/O and compute
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Robustness</td>
<td>Protection against media failure &amp; API misuse</td>
</tr>
<tr>
<td>Expressiveness</td>
<td>Support for events with nested variable length collections</td>
</tr>
<tr>
<td>Speed</td>
<td>Columnar layout, merge-friendly, sophisticated I/O scheduling</td>
</tr>
<tr>
<td>Stability</td>
<td>Backwards and forwards compatibility, hooks for schema evolution</td>
</tr>
<tr>
<td>Usability</td>
<td>Accessible to novice and expert programmers</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Facilitate concurrent reading/writing (merging) and (de-)compression</td>
</tr>
<tr>
<td>Integration</td>
<td>Support for HEP-specific, HPC, and Cloud storage and data mgmt systems</td>
</tr>
</tbody>
</table>
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)
Why invest in a **tailor-made I/O system**

- 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;
};
```
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

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

Cluster

File Header

Branch #1
Entries 0 .. N-1

#2
0 .. N-1

#3
0 .. N-1

#1
N .. 2N-1

Cluster

#2
N .. 2N-1

#3
N .. 2N-1

#1
2N .. 3N-1

Cluster

#2
2N .. 3N-1

#3
2N .. 3N-1

Cluster

#1
4N ...

#2
4N ...

#3
4N ...

Cluster

#1
3N ...

#2
3N ...

#3
3N ...

Cluster

#1
5N ...

#2
5N ...

#3
5N ...

Cluster

#1
6N ...

#2
6N ...

#3
6N ...

Cluster

#1
6.9*N-1

#2
6.9*N ...

#3
6.9*N ...

Cluster

#1
7N ...

#2
7N ...

#3
7N ...

TTree

Meta Data

File

Schema

Evolution

Support

Basket

Basket

Basket

Basket
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
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).
RNTuple Class Design

Seamless transition from TTree to RNTuple

Event iteration
Reading and writing in event loops and through RDataFrame
RNTupleDataSource, RNTupleView, RNTupleReader/Writer

Logical layer / C++ objects
Mapping of C++ types onto columns
e.g. std::vector<float> \rightarrow index column and a value column
RField, RNTupleModel, REntry

Primitives layer / simple types
“Columns” containing elements of fundamental types (float, int, …)
grouped into (compressed) pages and clusters
RColumn, RColumnElement, RPage

Storage layer / byte ranges
RPageStorage, RCluster, RNTupleDescriptor

Modular storage layer that supports files as data containers but also file-less systems (object stores)

Approximate translation between TTree and RNTuple classes:

- TTree \approx RNTupleReader
- RNTupleWriter
- TTreeReader \approx RNTupleView
- TBranch \approx RField
- TBasket \approx RPage
- TTreeCache \approx RClusterPool

→ RNTuple v1 Format Specification
RNTuple Format Evolution

- Key binary layout changes wrt. TTree
  - More efficient nested collections
  - More efficient boolean values (bitfield), interesting for trigger bits
  - Experimenting with "split floats"
  - Little-endian values (allows for mmap())

Implementation uses templates to slash memory copies and virtual function calls in common I/O paths

- Supported types
  - Boolean
  - Integers, floating point
  - std::string
  - std::vector, std::array
  - std::variant
  - User-defined classes
  - More classes planned (e.g. std::chrono timepoints)

Fully composable (including aggregation, inheritance) within the supported type system
The libRNTupleLite library is built just like any other ROOT libraries in ROOT proper (including modules, dictionaries etc).

The libRNTupleLite does not use any infrastructure from libCore but only from libROOTFoundation.

Functionality:
- RIOLite: RRawFile without support for plugins, i.e. only local files
- ROOTNTupleLite: Provide access to meta-data (schema etc.) and data pages
libRNTupleLite C API

● **C API header** and dynamic library libROOTNTupleLite.so
  ○ Header files will be in
    ■ io/iolite/inc/ROOT/IOLite.h
    ■ tree/ntuplelite/inc/ROOT/NTupleLite.h

● Provides a C wrapper to the C++ libROOTRNTupleLite.so

● Provided functionality:
  ○ Open an RNTuple that is stored in a local ROOT file
  ○ Read the schema: fields, columns, pages, and their relationships
  ○ Read pages into void * memory areas given column id and page id
    ■ Takes care of decompressing and unpacking pages along the way

● Aims at being a building block for 3rd party tool builders
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, ...)

28
RNTuple Format Breakdown

Cluster:
- Block of consecutive complete events
- Unit of thread parallelization (read & write)
- Typically tens of megabytes

Page/Basket:
- Unit of memory mapping or (de)compression
- Typically tens of kilobytes
### Comparison With Other I/O Systems

<table>
<thead>
<tr>
<th>Feature</th>
<th>ROOT</th>
<th>PB</th>
<th>SQLite</th>
<th>HDF5</th>
<th>Parquet</th>
<th>Avro</th>
</tr>
</thead>
<tbody>
<tr>
<td>Well-defined encoding</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>C/C++ Library</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Self-describing</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Nested types</td>
<td>✓</td>
<td>✓</td>
<td>?</td>
<td>?</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Columnar layout</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>?</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Compression</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>?</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Schema evolution</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

- ✓ = supported
- ✅ = unsupported
- ? = difficult / unclear

J. Blomer, [A quantitative review of data formats for HEP analyses](http://example.com) ACAT 2017