Describing the ROOT format with a DSL

Jim Pivarski

Princeton University – DIANA

October 16, 2017

ROOT is a file format.

- It's like HDF5 in that it organizes data objects in a filesystem-like structure.
- It's like Avro in that it defines the structure of the classes it stores.
- It's like Parquet in that it can split classes into columns for efficient access.
- Although it's more like Arrow/Feather in the way that it implements splitting.
- It's like Pickle in that its data model encompasses an entire language (C++ rather than Python).
- It's like FITS in that it was developed by a scientific community for that community.
- It's unlike most of the above in that it doesn't have a formal specification.

Specifications and implementations: what I could find

dianahep

	inception	specification	implementations
FITS	1981	https://fits.gsfc.nasa.gov/standard30/fits_standard30aa.pdf	38
netCDF,HDF4/5	1992	https://support.hdfgroup.org/HDF5/doc/H5.format.html	35
ROOT	1995	some class headers like TFile and TKey; not enough info to read a file	6
Pickle	1996	*implementation* changes: 1→2 PEP-307, 3→4 PEP-3154; not a real spec	4
Protocol buffers	2001	https://developers.google.com/protocol-buffers/docs/encoding	20
Thrift	2007	**UNOFFICIAL:** https://erikvanoosten.github.io/thrift-missing-specification/	15
Avro	2009	http://avro.apache.org/docs/current/spec.html	13
Parquet	2013	http://parquet.apache.org/documentation/latest/	5
Arrow/Feather	2016	https://arrow.apache.org/docs/memory_layout.html	7

Why not specify?

- inhibits development
- human-readable documents get out of date
- personnel already limited
- streamer info already specifies most classes dynamically
- ROOT C++ implementation must be primary

Why specify?

- better data preservation
- clarifies invariants that are hard to express or not local in code
- formal process for adding I/O features
- allows alternate I/O projects to maintain themselves
- may be descriptive, rather than prescriptive

What alternate implementations?

project	language	purpose	maintainer
ROOT	C++	main project	the ROOT Team (Philippe Canal)
JsRoot	Javascript	interacting with ROOT in the browser or standalone	the ROOT Team (Sergey Linev)
RIO	C++	embedded in GEANT-4	Guy Barrand?
root4j	Java	Spark and other Big Data	Viktor Khristenko
rootio	Go	go-hep ecosystem in Go	Sebastien Binet
uproot	Python	BulkIO-style Numpy access, pip-installable root_numpy, understanding ROOT I/O, prototyping	Jim Pivarski (me)

Sample code from uproot

```
fNbytes, fVersion, fObjlen, fDatime, fKeylen, fCycle = \
                                         file.readfields("!ihiIhh")

if fVersion > 1000:
    fSeekKey, fSeekPdir = file.readfields("!qq")    # 64-bit
else:
    fSeekKey, fSeekPdir = file.readfields("!ii")    # 32-bit

fClassName = file.readstring()  # byte or int32 size prefix
fName      = file.readstring()
fTitle     = file.readstring()
```

Sample code from uproot

```
fNbytes, fVersion, fObjlen, fDatime, fKeylen, fCycle = \
                                            file.readfields("!ihiIhh")

if fVersion > 1000:
    fSeekKey, fSeekPdir = file.readfields("!qq")    # 64-bit
else:
    fSeekKey, fSeekPdir = file.readfields("!ii")    # 32-bit

fClassName = file.readstring()  # byte or int32 size prefix
fName      = file.readstring()
fTitle     = file.readstring()
```

This is imperative Python code, but it doesn't need to be.

As a declarative specification

```yaml
TKey:
  assert:
    - $size == fKeylen  # check!
  members:
    - fNbytes: int32
    - fVersion: int16
    - fObjlen: int32
    - fDatime: int32
    - fKeylen: int16
    - fCycle: int16
    - if:
        - case: version > 1000
          then:
            - fSeekKey: int64  # big
            - fSeekPdir: int64
        - else:
            - fSeekKey: int32  # small
            - fSeekPdir: int32
    - fClassName: string
    - fName: string
    - fTitle: string
```

YAML is a declarative data language like JSON, but optimized for human input, often used for configuration files.

With additional interpretation, we can use it as a DSL for describing ROOT layout.

- An *executable* specification!
- Says nothing about eagerness vs. laziness of reading, leaving that to the implementation.
- Resembles streamer info, apart from if-then branches.
- Can add features as needed, but syntax is fixed.

As a declarative specification

diana hep

```yaml
TDirectory:
  members:
    - version: int16
    - ctime: int32
    - mtime: int32
    - nbyteskeys: int32
    - nbytesname: int32
    - if:
        - case: version <= 1000
          then:
            - seekdir: int32
            - seekparent: int32
            - seekkeys: int32
        - else:
            - seekdir: int64
            - seekparent: int64
            - seekkeys: int64
    - keys:
        type: TKeys
        at: seekkeys # seek to this value
```

YAML is a declarative data language like JSON, but optimized for human input, often used for configuration files.

With additional interpretation, we can use it as a DSL for describing ROOT layout.

- An *executable* specification!
- Says nothing about eagerness vs. laziness of reading, leaving that to the implementation.
- Resembles streamer info, apart from if-then branches.
- Can add features as needed, but syntax is fixed.

As a declarative specification

```yaml
TKeys:
  doc: |
    There is no ROOT class named
    "TKeys," but it's useful to define
    one here to represent a header
    TKey followed by an arbitrary
    number of TKeys.
  members:
    - header: TKey
    - nkeys:
        type: int32
        at: $pos + header.keylen
    - keys: {array: TKey, size: nkeys}

TBuffer_ReadVersion:
  doc: What TBuffer::ReadVersion does.
  members:
    - bytecount:
        # needs to be transformed first
        type: uint32
        postprocess: |
          bytecount & ~uint32(0x40000000)
    - version: uint16
```

YAML is a declarative data language like JSON, but optimized for human input, often used for configuration files.

With additional interpretation, we can use it as a DSL for describing ROOT layout.

- An *executable* specification!
- Says nothing about eagerness vs. laziness of reading, leaving that to the implementation.
- Resembles streamer info, apart from if-then branches.
- Can add features as needed, but syntax is fixed.

Eagerness vs. laziness

Selective reading is one of the most important features of ROOT I/O.

- In C++, ROOT reads TKey and TBasket data in response to user requests.
- In Python, I additionally want to avoid creating strings and other non-primitives before they're accessed (as Python "properties").

Eagerness vs. laziness

Selective reading is one of the most important features of ROOT I/O.

- In C++, ROOT reads TKey and TBasket data in response to user requests.
- In Python, I additionally want to avoid creating strings and other non-primitives before they're accessed (as Python "properties").

These decisions are very implementation-dependent, and shouldn't be a part of a specification.

- Although the YAML file describes the order of fields in the bytestream, they don't have to be read in that order.

Eagerness vs. laziness

Selective reading is one of the most important features of ROOT I/O.

- In C++, ROOT reads TKey and TBasket data in response to user requests.
- In Python, I additionally want to avoid creating strings and other non-primitives before they're accessed (as Python "properties").

These decisions are very implementation-dependent, and shouldn't be a part of a specification.

- Although the YAML file describes the order of fields in the bytestream, they don't have to be read in that order.
- As a demonstration, I implemented a *purely* lazy ROOT TH1F reader (*nothing* is read until explicitly referenced), configured by a YAML file:

https://github.com/jpivarski/rootspec

Streamer info

Most ROOT classes are already specified to this degree, not in a document, but in the ROOT files themselves, as TStreamerInfo.

Includes some very basic classes, like TTree, TList, TNamed, TObjArray...

No reason to duplicate this (and the version dependencies would be complicated to express in branches, anyway).

How far can this kind of documentation go?

Brian's wish list:

- Container classes to "bootstrap" to the point where we can read streamers: TFile, TKey, TBasket, TDirectory, the streamer classes themselves...
- How STL classes are streamed (may rewrite STL documentation in our format).
- ROOT's custom framing for compressed blocks (9 bytes before ZLIB, LZMA, and 17 before LZ4).
- How streamers are generated from classes.
- How classes are split into branches.
- How cross-references are keyed by byte positions (relative to what origin).

The last three could be hard to express declaratively...

Conclusions

A non-prescriptive specification of how ROOT I/O works would benefit data preservation, the process of changing ROOT I/O, and alternate implementations.

Conclusions

A non-prescriptive specification of how ROOT I/O works would benefit data preservation, the process of changing ROOT I/O, and alternate implementations.

Such a document can be an unofficial description (as Erik van Oosten did for Thrift).

Conclusions

A non-prescriptive specification of how ROOT I/O works would benefit data preservation, the process of changing ROOT I/O, and alternate implementations.

Such a document can be an unofficial description (as Erik van Oosten did for Thrift).

It can also be executable, to verify that it's correct on test samples and to ensure that it doesn't get out-of-date.

Conclusions

diana hep

A non-prescriptive specification of how ROOT I/O works would benefit data preservation, the process of changing ROOT I/O, and alternate implementations.

Such a document can be an unofficial description (as Erik van Oosten did for Thrift).

It can also be executable, to verify that it's correct on test samples and to ensure that it doesn't get out-of-date.

I began a demonstration-level project documenting some core ROOT classes with YAML, which can configure readers anywhere on the eager-to-lazy spectrum (by implementing purely lazy; eager is much more straightforward).

https://github.com/jpivarski/rootspec