Analyzing-Trees-Section1

June 26, 2018

1 Analyzing Data in a ROOT Tree

1.1 Why Trees?

ROOT trees are optimized for the storage of the kind of data produced in high-energy and nuclear physics experiments:

- Very large numbers of *events* with essentially the same data structure
- Variable length containers holding identical data/objects
- Tree-like structures of collections of objects, similar to databases

Access patterns to data in ROOT trees are also optimized for the kinds we typically need for our analyses:

- "Column-wise" reading of individual data elements—only the element(s) of interest are read, not entire events
- Only one event (or part of it) in memory at a time (modulo buffer size)
- Buffered to disk, some degree of integrity protection during writing ("cycles")

Without structures like ROOT trees, the efficient analysis of petabyte-size data sets from CERN and elsewhere would be nearly impossible. ROOT trees

- minimize memory requirements and I/O
- greatly enhance access speed

In addition to simple n-tuple-like data, ROOT lets you write C++ objects to file. This is impossible in native C++ and is achieved through class dictionaries generated with the *Cling* C++ interpreter/compiler. The ability to write entire object trees is critical for storing the very complex event data from HEP experiments. This is also knows as "C++ object persistency". Some programming languages offer object persistency natively, but not C++.

1.2 Hall A Analyzer trees

ROOT trees produced by the Hall A and Hall C analyzers have a rather simple structure. Each "global variable" (analysis result) selected for output to the tree is written to its own branch with only a single leaf. The name of the branch is identical to the name of the global variable, for example `L.tr.p`. At this time, data branches always have the type `Double_t`. We are planning to support other data types, in particular integers, starting with analyzer version 1.7.

In the case of arrays, a second branch is written to the tree whose name is `Ndata.` followed by the name of the corresponding array, *e.g.* `Ndata.L.tr.p`. These branches always have type `Int_t` and hold the number of elements in the array for the current event. There is a lot of redundancy in `Ndata` variables for various elements of a single object, such as the collection of tracks. We may reduce or even eliminate this redundancy in a future version of the analyzer.

Here is an example of a typical Hall A analyzer output tree where all the Left-HRS track data have been written out with a `block L.tr.*` statement in the output defintion file:

To explore this exact tree yourself, start an interactive `analyzer` or ROOT session and type:

```
analyzer [1] f = TFile::Open("/data/ROOTfiles/g2p_3132.root","READ");
analyzer [2] b = new TBrowser;
```

or, if you didn't download the file yet,

```
analyzer [1] f = TFile::Open("http://hallaweb.jlab.org/data_reduc/AnaWork2018/ROOTfiles/g2p_3132
analyzer [2] b = new TBrowser;
```

1.3 How to work with tree data

1.3.1 Inspecting text output

ROOT offers a number of ways to work with data in trees. First, there are two commands that produce text output. They are essentially legacy commands from the days of PAW, but nevertheless they work well with the n-tuples in our trees:

Scan: Prints a table where each row corresponds to an event and and each column, to the branch data. If there are multiple entries in a variable-sized array, multiple rows are printed for a single event, each row corresponding to the array index, called "Instance". Allows quick comparison of a number of columns (often faster and clearer than plotting).

Show: Prints all data for a single event. The output can be large. Helps with understanding the data structure. Often used for inspecting unusual events such as misreconstructed tracks.

Let's try these

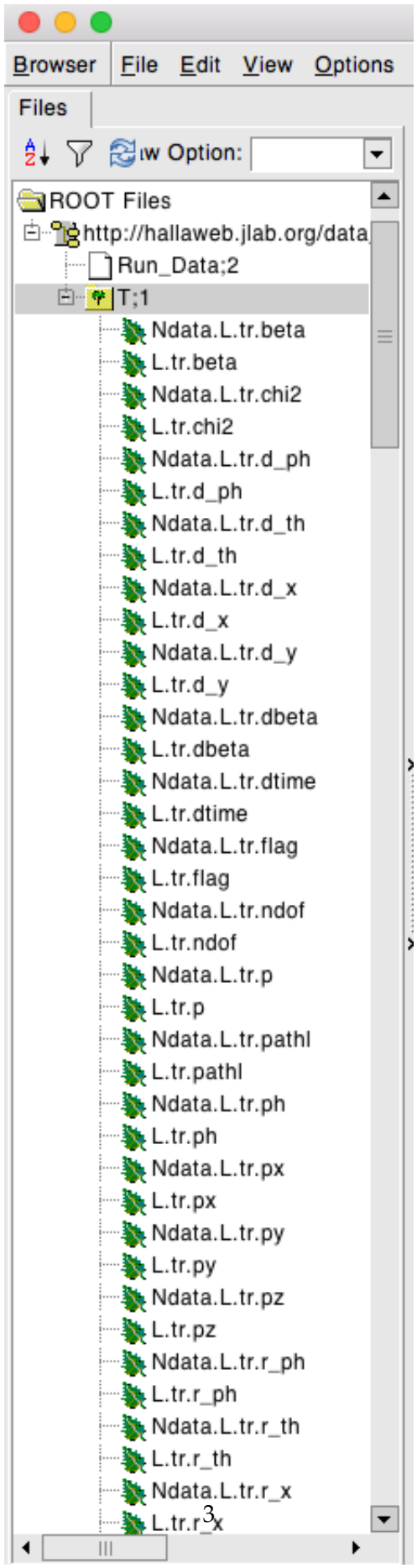
TTree::Scan

```
In [1]: // Open ROOT file
        f = TFile::Open("/data/ROOTfiles/g2p_3132.root","READ");
```

```
Warning in <TClass::Init>: no dictionary for class THaEvent is available
Warning in <TClass::Init>: no dictionary for class THaEventHeader is available
Warning in <TClass::Init>: no dictionary for class THaRun is available
Warning in <TClass::Init>: no dictionary for class THaCodaRun is available
Warning in <TClass::Init>: no dictionary for class THaRunBase is available
Warning in <TClass::Init>: no dictionary for class THaRunParameters is available
```

Ignore the warnings about `"no dictionary for class ..."` These occur because we are running plain ROOT and not the `analyzer`.

```
In [2]: // Look at the contents of the file
        f->ls();
```


Browser
File
Edit
View
Options
Files
w Option:
ROOT Files
http://hallaweb.jlab.org/data
Run_Data;2
T;1
Ndata.L.tr.beta
L.tr.beta
Ndata.L.tr.chi2
L.tr.chi2
Ndata.L.tr.d_ph
L.tr.d_ph
Ndata.L.tr.d_th
L.tr.d_th
Ndata.L.tr.d_x
L.tr.d_x
Ndata.L.tr.d_y
L.tr.d_y
Ndata.L.tr.dbeta
L.tr.dbeta
Ndata.L.tr.dtime
L.tr.dtime
Ndata.L.tr.flag
L.tr.flag
Ndata.L.tr.ndof
L.tr.ndof
Ndata.L.tr.p
L.tr.p
Ndata.L.tr.pathl
L.tr.pathl
Ndata.L.tr.ph
L.tr.ph
Ndata.L.tr.px
L.tr.px
Ndata.L.tr.py
L.tr.py
Ndata.L.tr.pz
L.tr.pz
Ndata.L.tr.r_ph
L.tr.r_ph
Ndata.L.tr.r_th
L.tr.r_th
Ndata.L.tr.r_x
L.tr.r_x

TreeView

```
TFile**                /data/ROOTfiles/g2p_3132.root
 TFile*                /data/ROOTfiles/g2p_3132.root
  KEY: THaRun        Run_Data;2        g2p run 3132
  KEY: TTree         T;1        Hall A Analyzer Output DST
```

Note the tree named **T**

```
In [3]: // Print the value of the "L.tr.p" variable (momentum of reconstructed track in GeV)
        // and "L.tr.vz" (vertex z-coordinate in m) for the first 10 events.
        T->Scan("L.tr.p:L.tr.vz","","",10);
```

```
************************************************
*    Row   * Instance *    L.tr.p *   L.tr.vz *
************************************************
*        0 *        0 * 2.2508660 * 0.0016883 *
*        1 *        0 * 2.2000959 * 0.0075512 *
*        2 *        0 * 2.2489658 * 0.0085386 *
*        3 *        0 * 2.2487922 * 0.0482533 *
*        4 *        0 * 2.2428897 * 0.0239013 *
*        5 *        0 * 2.2513184 * 0.0304236 *
*        6 *        0 *  2.206881 * 0.0085658 *
*        7 *        0 * 2.2273119 * 0.0132412 *
*        8 *        0 * 2.1854466 * 0.0153844 *
*        9 *        0 * 2.2493795 * 0.0435411 *
************************************************
```

Let's scan again, this time selecting events with multiple tracks. Because these are rare (at the level of a few percent), let's scan the first 500 events.

```
In [4]: T->Scan("L.tr.p:L.tr.vz","L.tr.n>1","",500);
```

```
************************************************
*    Row   * Instance *    L.tr.p *   L.tr.vz *
************************************************
*       38 *        0 * 3.0418453 * 2.3118241 *
*       38 *        1 * 2.8860089 * 2.6425863 *
*       44 *        0 * 52.622658 * 0.2657139 *
*       44 *        1 * 2.5149214 * -1.542678 *
*       44 *        2 * 2.9936564 * -1.244326 *
*       44 *        3 * 2.5968255 * -0.199140 *
*      156 *        0 * 2.5946465 * 3.0075283 *
*      156 *        1 * 2.3305075 * -0.080241 *
*      174 *        0 * 2.8993665 * 1.4813918 *
*      174 *        1 * 2.0904357 * 1.4947630 *
*      218 *        0 * 4.3860716 * 3.1441395 *
*      218 *        1 * 7.6736062 * 1.8077999 *
*      470 *        0 * 3.7749093 * 1.9965279 *
```

```
*      470 *        1 * 22.422996 * 3.7932473 *
*      470 *        2 * 3.7986176 * 2.7120001 *
*      480 *        0 * 1.9938786 * 1.9214297 *
*      480 *        1 * 113.13986 * 10.898452 *
*      484 *        0 * 2.3361050 * 4.6470851 *
*      484 *        1 * 2.4774303 * 1.2389566 *
*      484 *        2 * 23.240640 * -1.556557 *
************************************************
==> 20 selected entries
```

Modify the command above to scan the first 1000 events or more. Once the output fills a good screenful, ROOT will prompt you if you wish to continue or quit. (The prompt is ignored in the notebook environment.)

As you can see, there are now multiple instances per event (=row number). The `L.tr.p` and `L.tr.vz` arrays are parallel, *i.e.* for both arrays the index has the same meaning. If you are unsure if arrays are parallel, you can plot the corresponding `Ndata` elements against each other. To do so, we can use `T->Draw()`, which we'll discuss a little later.

TTree::Show

The second text-based command to inspect trees is `TTree::Show(entry)`. It simply prints the entire contents of the given entry (event). Let's give it a try:

```
In [5]: T->Show(5);
```

```
======> EVENT:5
 Ndata.L.tr.beta = 1
 L.tr.beta       = 1e+38
 Ndata.L.tr.chi2 = 1
 L.tr.chi2       = 0.00160361
 Ndata.L.tr.d_ph = 1
 L.tr.d_ph       = 0.00480143
 Ndata.L.tr.d_th = 1
 L.tr.d_th       = 1.04106
 Ndata.L.tr.d_x  = 1
 L.tr.d_x        = 0.198577
 Ndata.L.tr.d_y  = 1
 L.tr.d_y        = 0.00333265
 Ndata.L.tr.dbeta = 1
 L.tr.dbeta      = 1e+38
 Ndata.L.tr.dtime = 1
 L.tr.dtime      = 1e+38
 Ndata.L.tr.flag = 1
 L.tr.flag       = 16384
 Ndata.L.tr.ndof = 1
 L.tr.ndof       = 17
 Ndata.L.tr.p    = 1
 L.tr.p          = 2.25132
 Ndata.L.tr.pathl = 1
```

```
L.tr.pathl      = 25.7707
Ndata.L.tr.ph   = 1
L.tr.ph         = 0.00332678
Ndata.L.tr.px   = 1
L.tr.px         = 0.200856
Ndata.L.tr.py   = 1
L.tr.py         = -0.0314901
Ndata.L.tr.pz   = 1
L.tr.pz         = 2.24212
Ndata.L.tr.r_ph = 1
L.tr.r_ph       = 0.00542946
Ndata.L.tr.r_th = 1
L.tr.r_th       = -0.00309156
Ndata.L.tr.r_x  = 1
L.tr.r_x        = 0.137589
Ndata.L.tr.r_y  = 1
L.tr.r_y        = 0.0107624
Ndata.L.tr.tg_dp = 1
L.tr.tg_dp      = 0.0104525
Ndata.L.tr.tg_ph = 1
L.tr.tg_ph      = -0.00996509
Ndata.L.tr.tg_th = 1
L.tr.tg_th      = 0.0139894
Ndata.L.tr.tg_y = 1
L.tr.tg_y       = -0.00271471
Ndata.L.tr.th   = 1
L.tr.th         = 0.01955
Ndata.L.tr.time = 1
L.tr.time       = 1e+38
Ndata.L.tr.vx   = 1
L.tr.vx         = 0
Ndata.L.tr.vy   = 1
L.tr.vy         = 0
Ndata.L.tr.vz   = 1
L.tr.vz         = 0.0304237
Ndata.L.tr.x    = 1
L.tr.x          = 0.137589
Ndata.L.tr.y    = 1
L.tr.y          = 0.00286525
L.cer.asum_c    = 1140.82
L.ekine.Q2      = 0.0414565
L.ekine.W2      = 124.951
L.ekine.angle   = 0.0904299
L.ekine.epsilon = 0.995922
L.ekine.nu      = 0.00202145
L.ekine.omega   = 0.00202145
L.ekine.ph_q    = 2.98608
L.ekine.q3m     = 0.203619
```

```
L.ekine.q_x     = -0.200856
L.ekine.q_y     = 0.0314901
L.ekine.q_z     = 0.0112203
L.ekine.th_q    = 1.51566
L.ekine.x_bj    = 10.9319
L.gold.beta     = 1e+38
L.gold.dp       = 0.0104525
L.gold.index    = 0
L.gold.ok       = 1
L.gold.p        = 2.25132
L.gold.ph       = -0.00996509
L.gold.px       = 0.200856
L.gold.py       = -0.0314901
L.gold.pz       = 2.24212
L.gold.th       = 0.0139894
L.gold.x        = 0
L.gold.y        = -0.00271471
L.tr.n          = 1
L.vx.z          = 0.0304237
Event_Branch    = (THaEvent*)0x7f31f4d58a50
fEvtHdr.fEvtTime = 1331594504000000
fEvtHdr.fEvtNum = 8
fEvtHdr.fEvtType = 3
fEvtHdr.fEvtLen = 447
fEvtHdr.fHelicity = 0
fEvtHdr.fTargetPol = 0
fEvtHdr.fRun    = 3132
```

As you can guess, this is mostly useful for understanding odd outlier-type events. The output can be very lengthy and is often best redirected to a file using `.> filename` at the ROOT command prompt.

You may notice that a few values in this event are set to `1e+38`. This is a special value, `kBig`, that the analyzer uses to indicate that this value was never set. In the above analysis, no beta calculation was done, so the corresponding data equal `kBig`. While this is convenient for distinguishing, say, a zero reading from the "never read" case, unfortunately this convention also makes it difficult to plot any tree variables with at least a trivial cut like `abs(L.tr.x)<10` because there will always be some events that weren't fully computed, e.g. because of missing track data etc.

1.3.2 Graphical & Programmatic Processing

Here is a summary of the common methods for graphical and programmatic analysis of trees.

Draw: Extremely useful command for plotting tree data. Produces 1-D and 2-D histograms as well as 3-D graphs with various selection and drawing options.

MakeClass: Legacy function to create skeleton C++ code for a *specific* TTree. An event loop is provided for you in the source file, which you need to fill in.

MakeSelector: Improved method to create skeleton C++ code. Provides a processing class deriving from TSelector with various callback functions. ROOT does the looping over events for

you, and you have to specify what you would like to do for each event. Compared to MakeClass code, it is better organized, faster, and suitable for multithreading. Available as of ROOT 6.

TDataFrame: Experimental processing framework featuring implicit multithreading and on-demand execution. Relies heavily on C++11. Currently still quite buggy and slow. Available as of ROOT 6.10 with a mainstream appearance in ROOT 6.14/00. Essentially a preview of what to expect in ROOT 7.

In the following two sections, we will walk through examples of `TTree::Draw` and `TTree::MakeSelector`. Click on "Next Section" below.

Next: Section 2, TTree::Draw >>