RootUtils: a Library to Better Understand ROOT Files

Reza NasiriGerdeh*, Jim Pivarski†, Matevz Tadel ‡, Michael A. Sevilla*,
Carlos Maltzahn*, Brian Bockelman§, Frank Würthwein ‡
*University of California, Santa Cruz {*rnasirig, msevilla, carlosm*}*@ucsc.edu*
†Princeton University *pivarski@princeton.edu*
‡University of California, San Diego {*mtadel,fkw*}*@ucsd.edu*
§University of Nebraska-Lincoln *bbockelm@cse.unl.edu*

***Abstract*—We present $RootUtils$, a C++ library on top of ROOT framework. $RootUtils$ has been designed to better understand the ROOT files, the input/output part of the ROOT framework. The library provides a set of classes and functions to extract and show the different components comprising the ROOT files such as header, directories, data objects, trees, and etc. The print function associated with each component shows the value of the fields related to that component of the ROOT file. We think that $RootUtils$ library can be very helpful for computer scientists who wants to perform joint research projects for which understanding the ROOT file format is essential.**

***Index Terms*—RootUtils, ROOT Framework, ROOT File, Computer Science, Storage Systems, Physics Data**

I. Introduction

$ROOT$ [1], [2] is an object-oriented framework to process, analyze, visualize, and store large scale physics data [1]. It is written in C++ programming language and comprised of about 3000 classes, and 110 packages and plugins [3]. ROOT framework leverages $ROOT\ files$ to store the data to and retrieve the data from the underlying storage system. ROOT files have machine-independent binary format, which includes both data and its metadata [4].

In this technical report, we introduce $RootUtils$, a library on top of ROOT framework to better understand the format of ROOT files. We already presented the basics of ROOT files in detail in our previous publication [5]. Before describing the capabilities of RootUtils library, we provide a brief overview of ROOT files (from [5]).

In general, a ROOT file consists of a $header$ and a set of $serialized\ objects$ (instances of C++ classes). Some of these objects contain data and some of them contain only metadata. An object that contains data is $data\ object$ and an object including only metadata can be either $directory$ or $tree$. Logical structure of a ROOT file (with directories and data objects) is akin to a UNIX directory hierarchy. A ROOT file has a $root\ directory$ similar to "/" directory in UNIX directory structure. Directories can contain other directories or data objects. The data objects can be only inside a directory. Directories and data objects are akin to directories and files in UNIX directory hierarchy.

Physical structure of a ROOT file is shown in Fig. 1. It consists of a header, root directory, set of directories and data objects with associated keys, and trailer. Header includes the metadata describing whole ROOT file such as the size of the file. Trailer is used to append data to the existing file. Keys corresponding to directories/data objects include associated metadata such as the size of the directory/data object. See [5] for more details about the physical structure of ROOT files and the fields defining the components of ROOT files.

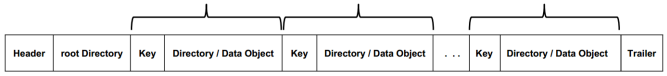

Fig. 1: Physical structure of a ROOT file with directories and data objects [5]. Each directory/data object has its own key. The only directory without key is root directory.

A ROOT file is created by instantiating an instance of class $TFile$ and passing "CREATE" or "RECREATE" option to the constructor to indicate that the file is opened in write mode. The root directory is created automatically when creating the ROOT file ($TFile$ inherits from $TDirectoryFile$). A directory is an instance of $TDirectoryFile$ and a data object is an instance of any class defined to store data. Key of each directory/data object is an instance of $TKey$ class. The base class of ROOT framework is $TObject$ and all the other classes inherit directly or indirectly from this class.

ROOT files with $trees$ is used to efficiently store tuples. The logical structure of a ROOT file containing trees consists of a set of trees in which each tree has a set of $branches$, and each branch has a set of $entries$ grouped into a set of $baskets$. A tree in the ROOT file is akin to a table; a branch of the tree is similar to a column of the table; a branch entry is akin to a column value. Notice that a branch entry is not necessarily a scalar value. It can be an array of values. In other words, tree is similar to a table in which each column of the table can also be another table. The baskets are the unit of data stored in the ROOT file. Each basket contains a set of entries of a branch. For example, the first basket of a branch might contain 0^{th} to 100^{th} entry of the branch, second basket might include $101^{th}$to 250^{th} entry of the branch, and so on and so forth.

The physical structure of a ROOT file with trees is illustrated in Fig. 2. It is comprised of a header, root directory,

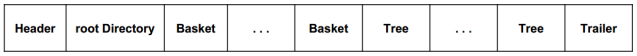

Fig. 2: Physical structure of a ROOT file with trees [5]. Baskets contain data and trees include only metadata.

set of baskets, set of trees, and a trailer. The functionality of header, root directory, and trailer are the same as ROOT files with directories. Baskets contain the actual data and the trees include only metadata. Notice that although branches are not shown in the physical structure of ROOT files, they are written as part of the tree in the file. Moreover, each basket includes some metadata (e.g size of the basket and size of the data in the basket) associated to its data. The baskets of the same branch are not consecutive in the ROOT file and can have different sizes. A tree is created by instantiating an instance of $TTree$ class. Branch and basket are instances of $TBrach$ and $TBasket$ classes, respectively.

Given this brief introduction of ROOT files, we present the capabilities and functionalities of RootUtils library in the following section.

II. RootUtils Library

As mentioned before, $RootUtils$ is an open source library on top of ROOT framework and is designed to help scientists to better grasp ROOT file format. $RootUtils$ provides a collection of classes and functions to show the value of comprising fields of the different components of the ROOT file such as header, directories, data objects, keys or trees, branches, and baskets. The ROOT framework must be installed to use the library and to execute the codes presented in this section. The installation instructions can be found at [6]. After installing ROOT framework, enter ROOT interactive environment by executing $root$ command. Type $.x\ code\_file\_name.cpp()$ in the $root$ environment to execute the code contained in $code\_file\_name.cpp()$.

To illustrate the capabilities of the library, we first create two sample ROOT files: one with directories and another with trees. Listing 1 shows the code to create a ROOT file with directories. Line 4 of the code opens the ROOT file ”first-root-file.root” in write mode (creates a new ROOT file with name ”first-root-file.root”). Lines 7, 8, and 9 create the data object ”histogram1”, fill the histogram with data, and write the data object to the root directory (which is the current directory), respectively. Line 12 creates the directory ”directory1” inside the root directory and line 13 changes the current directory to ”directory1”. Line 16 creates the data object ”histogram2”, line 17 fills it with data, and line 18 writes it into the directory ”directory1”. Line 21 close the ROOT file.

```
1  int createFirstRootFile(){
2
3    // Open ROOT file "first-root-file.root" in write
       mode
4    TFile * firstRootFile = new TFile ("first-root-
       file.root", "RECREATE");
5
6    // create histogram "histogram1" inside the root
       directory
7    TH1F *hist1 = new TH1F("histogram1","The First
       Histogram",100,0,10);
8    hist1->FillRandom("gaus", 1000);
9    hist1->Write();
10
11   // Create the directory "directory1" inside the
       root directory
12   TDirectoryFile * dir1 = (TDirectoryFile*)
       firstRootFile->mkdir("directory1");
13   dir1->cd();
14
15   // create histogram "histogram2" inside the
       directory "directory1"
16   TH1F *hist2 = new TH1F("histogram2","The Second
       Histogram",100,0,10);
17   hist2->FillRandom("gaus", 2000);
18   hist2->Write();
19
20   // Close the ROOT file
21   firstRootFile->Close();
22
23   return 0;
24 }
```

Listing 1: Code (createFirstRootFile.cpp file) to create the ROOT file ”first-root-file.root” containing a directory and two data objects

Listing 2 illustrates the code to create a ROOT file with trees. Line 7 creates a tree with name ”tree1” and title ”The First Tree”. Line 11 and 15 define the variables $column1$ and $column2$, respectively. Line 12 creates the branch ”branch1”, associates it to the variable $column1$, and adds it to tree ”tree1”. Similarly, line 16 instantiates the branch ”branch2”, associates it to the variable $column2$ and adds the branch to ”tree1”. Lines 19 to 32 fills the branch ”branch1” with values 18, 20, 19, and 23 and the branch ”branch2” with values 3.7, 3.8, 3.2, 4.0. Line 33 writes the tree ”tree1” to the file.

```
1  int createSecondRootFile(){
2
3    // Open ROOT file "second-root-file.root" in write
       mode
4    TFile * secondRootFile = new TFile ("second-root-
       file.root", "RECREATE");
5
6    // create tree1
7    TTree * tree1 = new TTree("tree1", "The First Tree
       ");
8
9
10   // create branch1 and add it to tree1
11   int column1;
12   tree1->Branch("branch1", &column1);
13
14   // create branch2 and add it to tree1
15   double column2;
16   tree1->Branch("branch2", &column2);
17
18   // fill the tree (branches of the tree)
19   column1 = 18;
20   column2 =  3.7;
21   tree1->Fill();
22
23   column1 = 20;
24   column2 = 3.8;
25   tree1->Fill();
26
```

```
27    column1 = 19;
28    column2 = 3.2;
29    tree1->Fill();
30
31    column1 = 23;
32    column2 = 4.0;
33    tree1->Fill();
34
35    // write tree in the ROOT file
36    tree1->Write();
37
38    // close the ROOT file
39    secondRootFile->Close();
40
41    return 0;
42 }
```

Listing 2: Code (createSecondRootFile.cpp file) to create the ROOT file "second-root-file.root" containing a tree with two branches

As mentioned before, the first part of every ROOT file is header. Listing 3 shows the code to print the header of a ROOT file. Lines 2 and 6 include the source code of *RootHeader* and *RootFile* classes from *RootUtils* library, respectively. RootHeader class represents the header of the ROOT file and RootFile class represents the ROOT file itself. Line 10 defines the object (instance) *rootFile* from class *RootFile*. The constructor of *RootFile* takes the name of the ROOT file as an argument. Line 12 prints the value of fields of the header in standard output.

```
1  // printHeader.cpp
2  #include "rootheader.cpp"
3  #include "key.cpp"
4  #include "directory.cpp"
5  #include "tree.cpp"
6  #include "rootfile.cpp"
7
8  void printHeader(string rootFileName){
9
10     RootFile rootFile (rootFileName);
11
12     cout << rootFile.getHeader().toString();
13
14 }
```

Listing 3: Code to print the header of a ROOT file

Lets use the code in Listing 3 to print the header of "first-root-file.root". To this end, we execute the command *.x* printHeader.cpp("first-root-file.root") in root environment (Listing 4). Please refer to [5] to the description of each file and the interpretation of the values of the header fields.

```
1  root [0] .x printHeader.cpp("first-root-file.root")
2  fVersion = 61008
3  fBEGIN = 100
4  fEND = 4509
5  fSeekFree = 4445
6  fNbytesFree = 64
7  nfree = 1634030134
8  fNbytesName = 76
9  fUnits = 4
10 fCompress = 1
11 fSeekInfo = 1140
12 fNbytesInfo = 3000
13 fUUID = ad09731e-5816-11e8-9008-0100007fbeef
```

Listing 4: Printing the header of "first-root-file.root"

Listing 5 illustrates a function to print the directories in a ROOT file. Line 3 includes the source code of *Directory* class of *RootUtils* library. Line 12 extracts the directories of the ROOT file and line 14 prints the value of the fields of the directories. Listing 6 indicates the usage of *printDirectories* function by printing the value of fields for the directories inside the "first-root-file.root".

```
1  // printDirectories.cpp
2  #include "rootheader.cpp"
3  #include "key.cpp"
4  #include "directory.cpp"
5  #include "tree.cpp"
6  #include "rootfile.cpp"
7
8  void printDirectories(string rootFileName){
9
10     RootFile rootFile (rootFileName);
11
12     vector<Directory *> allDirectories = rootFile.
       getDirectories();
13
14     for (Directory * directory : allDirectories)
15         cout <<  directory->toString();
16
17 }
```

Listing 5: A function to print the directories in the ROOT file

```
1  root [0] .x printDirectories.cpp ("first-root-file.
       root")
2  fName = first-root-file.root
3  fTitle =
4  fUUID = ad09731e-5816-11e8-9008-0100007fbeef
5  fDatimeC = Tue May 15 01:05:05 2018
6  fDatimeM = Tue May 15 01:05:05 2018
7  fModified = 1
8  fWritable = 0
9  fSeekDir = 100
10 fSeekParent = 0
11 fSeekKeys = 4140
12 fNbytesKeys = 179
13
14 fName = directory1
15 fTitle = directory1
16 fUUID = ad246d54-5816-11e8-9008-0100007fbeef
17 fDatimeC = Tue May 15 01:05:05 2018
18 fDatimeM = Tue May 15 01:05:05 2018
19 fModified = 1
20 fWritable = 0
21 fSeekDir = 623
22 fSeekParent = 100
23 fSeekKeys = 4319
24 fNbytesKeys = 126
```

Listing 6: Printing the value of the fields for directories in "first-root-file.root"

Listing 7 shows the code to print the info of data objects. The function *getDataObjects* (line 12) takes the class name of the data object we are interested in as the argument and returns a vector containing the data objects. Listing 8 employs the *printDataObjects* function and prints the histogram data objects of "first-root-file.root".

```
1  // printDataObjects.cpp
2  #include "rootheader.cpp"
3  #include "key.cpp"
4  #include "directory.cpp"
5  #include "tree.cpp"
6  #include "rootfile.cpp"
```

```
7
8  void printDataObjects(string rootFileName, string
       dataObjectClassName){
9
10     RootFile rootFile (rootFileName);
11
12     vector<Key*> dataObjects = rootFile.
       getDataObjects(dataObjectClassName);
13
14     for(Key* dataObjectKey : dataObjects)
15         cout << dataObjectKey->toString() << "\n";
16
17 }
```

Listing 7: A function to print the data objects in the ROOT file

```
root [0] .x printDataObjects.cpp ("first-root-file.
    root", "TH1F")
fName = histogram1
fTitle = The First Histogram
fClassName = TH1F
fNbytes = 387
fObjlen = 957
fDatime = Tue May 15 01:05:05 2018
fKeylen = 62
fCycle = 1
fSeekKey = 236
fSeekPdir = 100

fName = histogram2
fTitle = The Second Histogram
fClassName = TH1F
fNbytes = 398
fObjlen = 958
fDatime = Tue May 15 01:05:05 2018
fKeylen = 63
fCycle = 1
fSeekKey = 742
fSeekPdir = 623
```

Listing 8: Printing the histogram data objects of "first-root-file.root"

Listing 9 defines a function to print the trees in the ROOT file. Listing 10 shows a usage of *printTrees* function by printing the trees of "second-root-file.root".

```
1  // printTrees.cpp
2  #include "rootheader.cpp"
3  #include "key.cpp"
4  #include "directory.cpp"
5  #include "tree.cpp"
6  #include "rootfile.cpp"
7
8  void printTrees(string rootFileName){
9
10     RootFile rootFile (rootFileName);
11
12     vector<Tree*> trees = rootFile.getTrees();
13     for(Tree* tree : trees)
14         cout << tree->toString();
15
16 }
```

Listing 9: A function to print the trees in the ROOT file

```
root [0] .x printTrees.cpp ("second-root-file.root")
fName = tree1
fTitle = The First Tree
fEntries = 4
fBranches =
  branch1
  branch2
```

Listing 10: Printing the trees of "second-root-file.root"

III. Conclusion

In this technical report, we presented *RootUtils* library, which is designed to better understand the ROOT file format. We introduced the classes and functions comprising *RootUtils* and the capabilities of the library using a set of code examples. *RootUtils* can extract the metadata inside the ROOT files including header, directories, data objects, and trees. In the future versions of the library, we add the ability to extract the branches and the baskets of a given tree from the ROOT file.

References

[1] "Root data analysis framework," https://root.cern.ch, (accessed May 01, 2018).

[2] R. Brun and F. Rademakers, "ROOT: An object oriented data analysis framework," *Nucl. Instrum. Meth.*, vol. A389, pp. 81–86, 1997.

[3] I. Antcheva, M. Ballintijn, B. Bellenot, M. Biskup, R. Brun, N. Buncic, P. Canal, D. Casadei, O. Couet, V. Fine *et al.*, "Root—a c++ framework for petabyte data storage, statistical analysis and visualization," *Computer Physics Communications*, vol. 180, no. 12, pp. 2499–2512, 2009.

[4] "The root object i/o system," https://root.cern.ch/root/InputOutput.html, (accessed May 01, 2018).

[5] R. NasiriGerdeh, J. Pivarski, M. A. Sevilla, and C. Maltzahn, "Root files for computer scientists," UC Santa Cruz, Tech. Rep. UCSC-SOE-18-09, May 2018.

[6] "Building root," https://root.cern.ch/building-root, (accessed May 01, 2018).