ROOT Files for Computer Scientists

Reza NasiriGerdeh*, Jim Pivarski†, Michael A. Sevilla*, Carlos Maltzahn*
*University of California, Santa Cruz {*rnasirig, msevilla, carlosm*}*@ucsc.edu*
†Princeton University *pivarski@princeton.edu*

***Abstract*—ROOT is an object-oriented framework to process, analyze, visualize, and store large scale data. There are many publications on introducing ROOT framework to physicists because they are the main users of the framework. However, physicists are not the only people who are interested in ROOT framework. The framework is also interesting for computer scientists who perform joint research projects with the physicists. In this paper, we present ROOT framework from a viewpoint more familiar to computer scientists. We focus on the ROOT files, which is the input/output part of the framework. We provide an overview of ROOT files and their logical and physical structure. Moreover, we introduce ROOT files with trees, which are one of the most commonly used types of ROOT files and are leveraged to store tabular data (tuples).**

***Index Terms*—ROOT Framework, ROOT File, Computer Science, Storage Systems, Physics Data, Tree, Branch, Basket**

I. Introduction

$ROOT$ [1], [2] is a modular and object-oriented framework for large scale data processing, analysis, and storage [1]. Daily, thousands of physicists around the world leverage ROOT framework based applications to process and visualize their data [3]. ROOT is written in C++ language and it consists of around 3000 classes, categorized into around 110 packages and plugins [3]. One of the most important parts of the framework is the Input/Output part (focus of this paper), which is used to store and retrieve physics data. The framework employs the $ROOT\ file$ format to write the data into and read the data from the underlying storage.

There are many useful documents and tutorials on the Internet to help scientists use the the ROOT framework [4], [5]. Many of these documents are written for physicists because they are the main users of ROOT framework. However, physicists are not the only people who are interested in the framework. The ROOT framework is also interesting for computer scientists who are responsible for extending the framework and developing ROOT based applications. Most importantly, there are lots of joint research opportunities for computer scientists on computation and storage aspects of physics data. The initial step toward carrying out these joint researches is to get to understand the ROOT framework.

In this paper, we present the ROOT framework from the viewpoint more familiar and understandable for computer scientists. We focus on ROOT files, which is the input/output part of the framework. We begin with a brief introduction of ROOT files along with a code to create a simple ROOT file in section II. In section III, we describe the logical and physical structure of a ROOT file and elaborates on the different components comprising the physical structure of the ROOT files including $header$ (subsection III-A), $directory$ (subsection III-B), key (subsection III-C), data objects, and $trailer$ (subsection III-D). We introduce the ROOT files containing $trees$ and describe how data (tabular) can be organized as trees and stored in ROOT file(s) (section IV). We conclude the paper with a brief conclusion (section V).

II. Brief Overview of ROOT Files

$ROOT$ file is similar to UNIX directory structure [5]. ROOT file has a $root\ directory$ which is akin to "/" directory in UNIX. Inside the $root\ directory$, there exists $directories$ or $data\ objects$. Directories in turn contain other directories or data objects. Data objects can only exist inside a directory. Directories and data objects in ROOT file are equivalent to directories and files in UNIX file system. A ROOT file is created by instantiating an instance of $TFile$ class. A directory is created through instantiating an instance of $TDirectoryFile$ class. Data object is an instance of any class defined to contain user data. All classes in ROOT framework including $TFile$, $TDirectoryFile$, and classes for data objects inherit directly or indirectly from the base class $TObject$.

To make the concepts of ROOT file, root directory, directories, and data objects more clear, lets create a simple ROOT file called "my-root-file.root" with the directory structure shown in Figure 1. The file has a root directory; it contains another directory called "directory1" and the data object "histogram1". The directory "directory1" contains the data object "histogram2".

The code shown in Listing 1 creates the ROOT file "my-root-file.root". Line 6 opens the ROOT file in write mode, which creates the ROOT file header and root directory. Line 11 creates the data object "histogram1", which assigns the data object to the current directory and line 13 writes the data object in the current directory. Please notice that after opening the file, the current directory is set to the root directory. Line 17 creates the directory "directory1" inside the root directory and line 18 changes the current directory from root directory to directory "directory1". Line 23 creates the data object "histogram2" and line 25 writes the data object to the directory "directory1" (current directory). Finally, line 28 closes the ROOT file.

To execute the code, first install the ROOT framework. The installation instructions for ROOT can be found at [6]. After installing ROOT framework, execute the command $root$ and

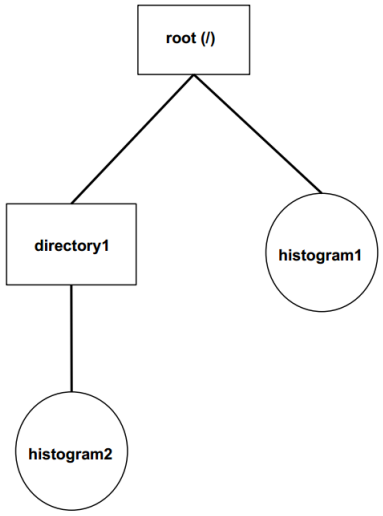


Fig. 1: Directory hierarchy of the simple ROOT file "my-root-file.root" containing one directory and two data objects

enter the ROOT interactive environment. Finally, type and execute $.x\ simpleRootFile.cpp()$.

```
1  int simpleRootFile(){
2
3    // Open a ROOT file in write mode
4    char * rootFilePath = "my-root-file.root";
5    char * openMode = "RECREATE";
6    TFile * myRootFile = new TFile (rootFilePath,
       openMode);
7
8    // First data object
9    char * firstHistName = "histogram1";
10   char * firstHistTitle = "First Histogram";
11   TH1F *hist1 = new TH1F(firstHistName,
       firstHistTitle,100,0,10);
12   hist1->FillRandom("gaus", 1000);
13   hist1->Write();
14
15   // Create a directory inside the root directory
16   char * directoryName = "directory1";
17   TDirectoryFile * dir1 = (TDirectoryFile *)
       myRootFile->mkdir(directoryName);
18   dir1->cd();
19
20   // Second data object
21   char * secondHistName = "histogram2";
22   char * secondHistTitle = "Second Histogram";
23   TH1F *hist2 = new TH1F(secondHistName,
       secondHistTitle,100,0,10);
24   hist2->FillRandom("gaus", 2000);
25   hist2->Write();
26
27   // Close the ROOT file
28   myRootFile->Close();
29
30   return 0;
31
32 }
```

Listing 1: Code (simpleRootFile.cpp file) to create the ROOT file "my-root-file.root" which has a directory and two data objects

III. Structure of ROOT Files

As mentioned in section II, the *logical structure* of a ROOT file is similar to UNIX directory structure (Fig. 2 - bottom). The root directory is the top directory (level 1) of the hierarchy. The directories or data objects inside the root directory are at level 2 of the hierarchy. The directories at level 2 can have other directories or data objects (level 3) and so on and so forth.

The *physical structure* of a ROOT file is illustrated in Figure 2-top. As shown in the figure, physical structure of a ROOT file consists of a *header*, *root directory* a collection of *directories* or *data objects* with their associated keys, and a *trailer*. The *header* maintains the metadata describing the whole ROOT file (such as the total size of the ROOT file). The header is created when the ROOT file is created (by instantiating an instance of $TFile$ class). The *root directory* always immediately follows the header and like header is created when the ROOT file is created. This is because $TFile$ class inherits from the $TDirectoryFile$ class, and as a result, if an instance of $TFile$ is created, the constructor of the $TDirectoryFile$ is called, which creates the root directory. The remaining parts of the ROOT file (except trailer) can be a *directory* or *data object*. Directories do not contain any data and they mainly keep metadata about the directories and data objects they contain. A directory is created through instantiating an instance of $TDirectoryFile$ class. Data objects contain the actual data. A Data object is created by instantiating an instance of a class which is defined to store user data, e.g. an instance of $TH1F$ class. Moreover, an instance of $TKey$ class is used to define the metadata of the directories or data objects and is written along with the associated directory or data object in the ROOT file. *mkdir* can be used to create and write the directory in the file and $Write$ function can be used to write the data objects in the ROOT file (see Listing 1). It is worth mentioning that the only directory that has no key is the root directory. The last part of the ROOT file is the trailer, which is used to add data to the file. trailer includes the list of the keys of directories or data objects inside the root directory, $FreeSegments$ record, and etc.

Fig. 2 also shows the mapping from the *logical structure* to the *physical structure* of the ROOT file (dashed arrows). The metadata in the header, directories, keys, and trailer of the ROOT file defines this mapping. Notice that directories are written in the ROOT file in the order they are created (by instantiating an instance of $TDirectoryFile$ class). On the other hand, the data objects are written in the ROOT file in the order that they call their "Write" function. For example, consider the ROOT file "my-root-file.root" created by the code of Listing 1 again. The physical structure of the file is shown in Fig. 3. The header and root directory are at the beginning of the file since they are created first (line 6, Listing 1 which creates the ROOT file by instantiating an instance of $TFile$ class). Next is the data object "histogram1" which is written in the ROOT file before the directory $directory1$ and data object "histogram2" (line 13, Listing 1). Directory

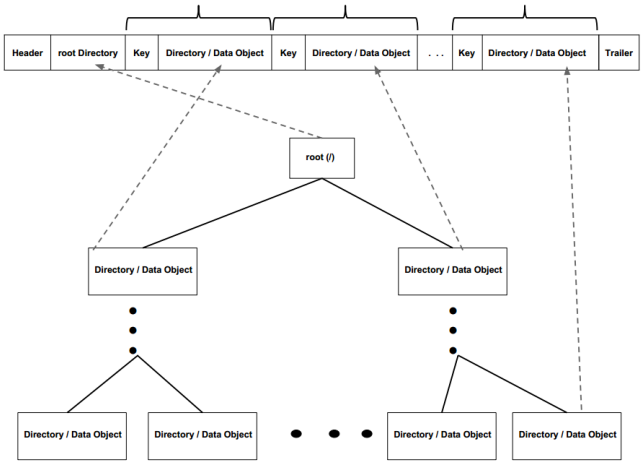


Fig. 2: Physical structure (top) and logical structure (bottom) of a ROOT file

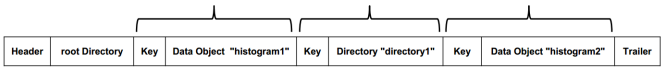


Fig. 3: Physical structure of the ROOT file "my-root-file.root"

"directory1" (created in line 17, Listing 1) follows the data object "histogram1". Then, data object "histogram2" (line 25, Listing 1) is written into the ROOT file. Finally, the trailer is written to the file. In the following subsections, we provide much more details about the header, directory, data objects, keys, and trailer of the ROOT file.

A. ROOT File Header

ROOT file header maintains general information about the file including the total size of the file, the size of the header itself, the start offset of the root directory, and etc. Figures 4a and 4b show the header of a small (size less than or equal to 2 GB) and large (size greater than 2GB) ROOT file, respectively. The numbers on top are the size (in terms of byte) of the fields and the numbers in the bottom are the start and end offsets of the fields. Notice that both small and large ROOT files have the same fields in the header. The difference is only in the size of the fields $fEND$, $fSeekFree$, and $fSeekInfo$, and as a result, in the start and end offsets of the fields. For a small file, the mentioned fields are of size 4 bytes while for a large file, they are of size 8 bytes. Table I lists the brief description of each field of the header.

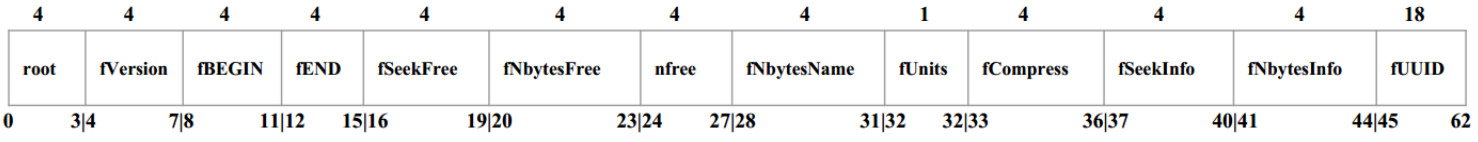
The first 4 bytes of the header (bytes 0-3) known as $magic\ header$ always contain "root" to specify that the file is of type ROOT. The value of $fVersion$ determines whether or not the file size is greater than 2GB. If the value of $fVersion$ is greater than 1000000 (1 million), the size of ROOT file is greater than 2GB and the fields $fEND$, $fSeekFree$, and $fSeekInfo$ are of size 8 bytes. Otherwise, the size of the file is less than or equal to 2GB and the fields $fEND$, $fSeekFree$, and $fSeekInfo$ are of size 4. $fBEGIN$ indicates the size of the header, which is also the offset that the $root\ directory$ of the ROOT file begins. $fEND$ contains the size of the ROOT file, which is also the offset of the last byte of the ROOT file.

$fSeekFree$ contains the beginning offset of the free data objects (is a pointer to the list of the free data objects). If a data object is deleted, the space occupied with that object is released and it is considered as the free data object. The released space can be used by the newly written data objects. $fNbytesFree$ specifies the total size (in terms of byte) of the list of the free data objects. $nfree$ is the number of free data objects. The fields $fSeekFree$, $fNbytesFree$, and $nfree$ manage the free data objects in the ROOT file. $fUnits$ specifies the size of the fields that have different size in small and large ROOT files; that is, the fields $fEND$, $fSeekFree$, and $fSeekInfo$. $fUnits$ value is 8 if the the size of the ROOT file is greater than 2 GB ($fVersion$ value is greater than 1 million) and it is 4 otherwise. $fNbytesName$ is the size of the $TNamed$ instance at the time of the ROOT file creation. $TNamed$ is the base class for all named ROOT classes. Named class means the class that has a name and title. $TFile$, $TDirectoryFile$, and $TKey$ are examples of named classes because they inherit from $TName$ class, and therefore, have name and title. $fCompress$ specifies the compression algorithm and compression level to be used to compress data objects. The compression algorithms include ZLIB and ZLMA and the compression levels are from 1 to 9, which higher levels indicates more compression ratios. Level 0 means no compression. $fSeekInfo$ contains the beginning offset of the $fStreamerInfo$ instance (is a pointer to $fStreamerInfo$ instance). $fStreamerInfo$ is an instance of $TObjArray*$ and is used to serialize and deserialize the directories and data objects. $fNbytesInfo$ indicates the size (in terms of bytes) of the $fStreamerInfo$ instance. $UUID$ is the universal unique identifier of the ROOT file.

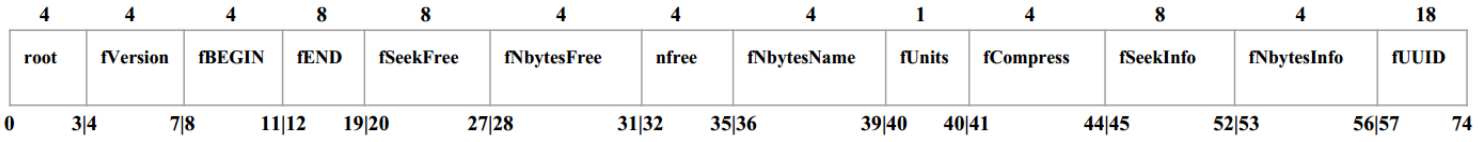
The third column of Table I shows the value of the header fields for the ROOT file "my-root-file.root". The first four bytes of the header contains the value "root" indicating that the file "my-root-file.root" is of type ROOT. The value of 61008 for $fVersion$ and 4 for $fUnits$ indicate that the file is less than 2GB. The value of $fBEGIN$ is 100 showing that the size of the header is 100 bytes and the $root\ directory$ starts from the offset of 100. $fEND$ value is 4468, which indicates that file size is 4468 bytes. $nfree$ is zero meaning that no data object has been deleted from the file. However, the value of $fNbytesFree$ indicates there are 61 free bytes at the end of the file. Notice that the value of $fSeekFree$, which is 4407, indicates the offset that the free space begins. The value of $fSeekFree$ + $fNbytesFree$ equals the value of $fEND$, the size of the file.

B. Directory in a ROOT File

A directory contains metadata about the data objects or directories it contains. It has no data associated with it. The directory is created by creating an instance of class $TDirectoryFile$. Table II shows the description of each field of a directory as well as the value of the fields for



(a) Small ROOT file (Size $\leq$ $2GB$)



(b) Large Root file (Size $>$ $2GB$)

Fig. 4: ROOT File Header

Filed Name	Field Description	Value for "my-root-file.root"
root	ROOT file identifier	"root"
fVersion	File Format Version, indicating whether or not the ROOT file size is greater than 2GB	61008
fBEGIN	Size of the header; start offset of the root directory	100
fEND	Size of the file	4468
fSeekFree	Start offset of the list of the free data objects (pointer to the list of the free data objects)	4407
fNbytesFree	size of the free data objects	61
nfree	number of the free data objects	0
fNbytesName	size of the TNamed at the creation time	70
fUnits	size of the fields fEND, fSeekFree, and fSeekInfo (whether 8 or 4)	4
fCompress	Compression algorithm and compression level	1
fSeekInfo	Start offset of the fStreamerInfo instance (pointer to fStreamerInfo instance)	1113
fNbytesInfo	size of the fStreamerInfo instance	3000
fUUID	Universal Unique Identifier of the file	963e63a4-4d1e-11e8-b0a6-0100007fbeef

TABLE I: Header fields of a ROOT file (adapted from [1] with some modifications) and their values for "my-root-file.root"

root directory and "directory1" of the ROOT file "my-root-file.root". The descriptions are self-explanatory. $fName$ is the name, $fTitle$ is the title, and $fUUID$ is the universal unique identifier of the directory. $fDatimeC$ and $fDatimeM$ indicate the data and time when the directory is created and last modified, respectively. $fWritable$ determines whether or not the directory is writable and $fModified$ indicates whether the directory has been modified. $fSeekDir$ specify the start offset of this directory in the ROOT file. $fSeekParent$ indicates the start offset of the parent directory of this directory. $fSeekKeys$ specify the start offset of the keys of the directories or data objects the directory contains and $fNbytesKeys$ is the total size of the keys in the directory.

As shown in the third and forth columns of the Table II, the name of the root directory is the same as the name of the ROOT file but it has no title. The parent directory of the root directory is the ROOT file header which starts from offset 0 (the value of $fSeekParent$ for root directory). The parent directory of "directory1" is the root directory which starts from offset 100 (the value of $fSeekDir$ for the root directory is equal to the value of $fSeekParent$ for the "directory1").

C. Key in a ROOT File

As mentioned before, a directory/data object is assigned a key (instance of $TKey$ class) before it is written to the file. After being assigned the key, associated key and the directory/data object is written to the ROOT file. Key can be considered as the header of the directory/data object. It is

Filed Name	Field Description	Value for root directory	Value for "directory1"
fName	Name of the directory	my-root-file.root	directory1
fTitle	Title of the directory		directory1
fUUID	Universal Unique Identifier of the directory	963e63a4-4d1e-11e8-b0a6-0100007fbeef	9659042a-4d1e-11e8-b0a6-0100007fbeef
fDatimeC	Date and time the directory was created	Tue May 1 02:03:59 2018	Tue May 1 02:04:00 2018
fDatimeM	Date and time the directory was last modified	Tue May 1 02:04:00 2018	Tue May 1 02:04:00 2018
fModified	Indicates whether or not the directory modified	True	True
fWritable	Determines whether or not the directory is writable	False	False
fSeekDir	start offset of this directory in the file	100	606
fSeekParent	start offset of the parent directory of this directory in the file	0	100
fSeekKeys	start offset of the list of keys associated with this directory	4113	4285
fNbytesKeys	size of the list of keys	172	122

TABLE II: Fields of a directory (adapted from [1] with some modifications)

used to retrieve the directory/data object from the file. Table III lists each field of the key, a short description of the field, and the value of the fields for directory "directory1" and data objects "histogram1" from "my-root-file.root". $fName$ and $fTitle$ are the name and the title of the directory/data object associated with the key. $fClassName$ is the name of the class of the directory/data object. $fKeylen$ indicates the size of the key, $fObjlen$ shows the size of the directory/data object before compression, and $fNbytes$ shows the size of the directory/data object after compression + the size of the key (The size of directory/data object after compression is $fNbytes$ - $fKeylen$). Notice that key is never compressed. $fSeekKey$ is the start offset of the directory/data object associated with the key. $fSeekPdir$ contains the start offset of the directory that this directory/data object belongs to. $fCycle$ is the cycle number of the directory/data object; each time the directory/data object modified, the cycle number increments. $fDatime$ is the date and time when the directory/data object was written in the file.

D. Trailer in a ROOT File

The last part of a ROOT file contains the trailer (Fig. 5 and Table IV). As mentioned before, trailer in general used to add data to the ROOT file. $fStreamerInfo$ field includes the metadata associated with the serialization and deserialization of directories or data objects in the file; $KeysList$ is the list of the keys of the directories or data objects inside the root directory. $FreeSegments$ contains the list of the free data objects. END is the last byte of the ROOT file. The fields $fEND$, $fSeekFree$, $fSeekInfo$ from the header contain the start offset of the fields END, $FreeSegments$, and $fStreamerInfo$ of the trailer, respectively. Moreover, $fSeekKeys$ field from root directory includes the start offset of the $KeysList$ of the trailer (Fig. 5).

IV. Root Files With Trees

In section II, we described how ROOT files can be used to store histogram data objects in different directories. In this section, we show how tabular data can be stored in ROOT files using $trees$ (instances of $TTree$ class). Consider a small table shown in Table V, which contains the age and GPA of four students . The code in Listing 2 stores the data of the table in the ROOT file using a $tree$.

Line 7 of the code creates "tree-root-file.root" ROOT file. Line 12 creates a tree with name "tree1" and title "A simple tree". Line 19 creates the first $branch$ and adds the branch to tree "tree1". The $Branch$ function takes three arguments describing the branch. First argument is the name of the branch ("branch1"), the second one is the address of the variable associated with the branch ($\&studentAge$), and the third one is the maximum size of the $baskets$ of the branch (120 bytes). Line 25 creates the second branch and adds it to the tree. This branch has name "branch2". It is associated with the variable $studentGPA$ and has the maximum size of baskets of 100 bytes.

As you may notice, we define a tree to store data contained in a table (we map a table to a tree). A tree is created by instantiating an instance of $TTree$ class. Each tree must have a name (e.g. "tree1") and might have a title (e.g. "A simple tree"). For each column of the table, we define a variable with type appropriate to the data in that column. For example, we define the variable $studentAge$ which is of type int (size

Filed Name	Field Description	Value for Directory "directory1"	Value for Data Object "histogram1"
fName	Name of the directory/data object	directory1	Histogram1
fTitle	Title of the directory/data object	directory1	First Histogram
fClassName	Class name of the directory/data object	TDirectoryFile	TH1F
fNbytes	Size of the compressed directory/data object + size of the key	119	376
fObjlen	Size of the uncompressed directory/data object	60	953
fDatime	Date and time the directory/data object was written to the file	Tue May 1 02:04:00 2018	Tue May 1 02:04:00 2018
fKeylen	Size of the key	59	58
fCycle	cycle number of the directory/data object	1	1
fSeekKey	start offset of the directory/data object associated with this key	606	230
fSeekPdir	start offset of the directory that this directory/data object belongs to	100	100

TABLE III: Fields of a key (adapted from [1] with some modifications) and the values for directory "directory1" and data object "histogram1" from "my-root-file.root"

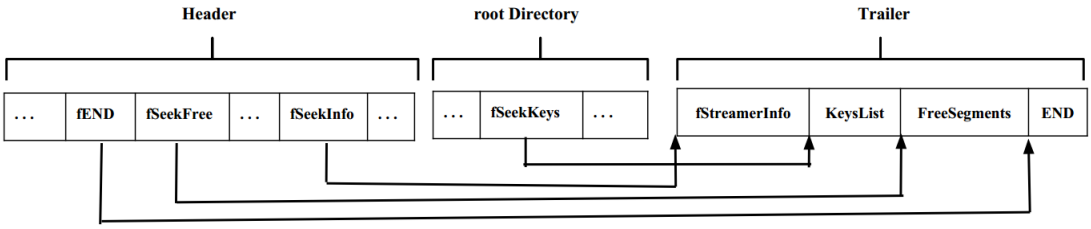


Fig. 5: Trailer of a ROOT file (right) with pointers from header and root directory

Field Name	Field Description
fStreamerInfo	metadata related to serialization and deserialization of directories/data objects
KeysList	List of the keys of directories/data objects in the root directory
FreeSegments	The list of the free data objects
END	end byte of the file

TABLE IV: Fields of the trailer

of 4 bytes) for the first column of the table containing the age of students and the variable $studentGPA$ which is of type $double$ (size of 8 bytes) for the second column of the table containing the GPA of the students. Similarly, we define a branch for each column and add it to the tree. A branch is defined by instantiating an instance of $TBranch$ class ($Branch$ function of $TTree$ returns an instance of $TBranch$ class). For example, we define the branch "branch1" for the first column of the table and branch "branch2" for the second column of the table.

Finally, we associate the variable defined for each column to the branch defined for that column. To this end, we pass the address of the variable as the second argument to the $Branch$ function defining the branch. For instance, line 19 associates the variable $studentAge$ to the branch "branch1" by passing the address of the variable as the second argument to the Branch function defining the branch "branch1". After creating a branch and associating a variable for that branch, a $buffer$ is created for that branch (Fig. 6). When the $Fill$ function of the tree is called, it copies the values of the variables to the buffers of the branches associated with those variables. For example, when the line 30 is executed, the value of variable $studentAge$, which is 18 is copied to the "branch1" buffer and the value of variable $studentGPA$, which is 3.7 is copied to "branch2" buffer (Fig. 6). Lines 34, 38, and 42 of the code copy the current values of the variable $studentAge$ (20,19, and 23) to "branch1" buffer and current values of the variable $studentGPA$ (3.8, 3.2, and 4.0) to "branch2" buffer.

Line 45 writes the tree to the ROOT file. When "Write"

function of the $TTree$ class is called, it writes the values of the buffer of each branch to the ROOT file independent of the buffers of other branches. The values in the buffer of each branch first grouped into a $basket$, then the basket is compressed and is written to the file. A basket is an instance of $TBasket$ class. Remember that we can pass the maximum size of the baskets of the branch as the third argument of the $Branch$ function. That is, we can set a limit on the size of the baskets containing the values of the branch buffers. In our example, the first three values of the "branch2" buffer (3.7, 3.8, and 3.2) are grouped into a basket. Notice that we set the limit of 100 bytes for the size of "branch2" baskets. The size of the actual data in the basket is (8+8+8=24) bytes but each basket also contains metadata. Because adding the forth value of the "branch2" buffer makes the basket size larger than 100 bytes, the first three values of the buffer are put in the first basket and the forth value of the "branch2" buffer (4.0) is put into the second basket. Similarly, the maximum size of the baskets of the "branch1" is 120 bytes and The values of the "branch1" buffer are of size 4 bytes. A single basket can accommodate all four buffer values of "branch1" (18, 20, 19, 23). Therefore, they are grouped into a basket and the basket is written to the file after compression.

According to Fig. 6, the baskets of the same branch are not necessarily consecutive in the ROOT file. For instance, the first basket of the "branch2" is first written into the file. Next, the first basket of "branch1" and the second basket of "branch2" are written into the file. Moreover, $tree1$ object(instance) is the last object written to the file. In general, tree is more similar to a directory than data object because tree only contains metadata including the branches it has, number of branch values it contains (rows of the table), and etc.

Student Age	Student GPA
18	3.7
20	3.8
19	3.2
23	4.0

TABLE V: Small table containing the age and GPA of four students

```
int simpleTreeExample(){

  // Create a ROOT file in write mode
  char * rootFilePath = "tree-root-file.root";
  char * openMode = "RECREATE";
  TFile * treeRootFile = new TFile (rootFilePath,
    openMode);

  // create tree1
  char * tree1Name = "tree1";
  char* tree1Title = "A simple tree";
  TTree * tree1 = new TTree(tree1Name, tree1Title);

  // create branch1 and add it to tree1
  char * branch1Name = "branch1";
  int studentAge;
  int branch1MaxBasketSize = 120;
  tree1->Branch(branch1Name, &studentAge,
    branch1MaxBasketSize);

  // create branch2 and add it to tree1
  char * branch2Name = "branch2";
  double studentGPA;
  int branch2MaxBasketSize = 100;
  tree1->Branch(branch2Name, &studentGPA,
    branch2MaxBasketSize);

  // fill the tree (branches of the tree)
  studentAge = 18;
  studentGPA =  3.7;
  tree1->Fill();

  studentAge = 20;
  studentGPA = 3.8;
  tree1->Fill();

  studentAge = 19;
  studentGPA = 3.2;
  tree1->Fill();

  studentAge = 23;
  studentGPA = 4.0;
  tree1->Fill();

  // write tree in the ROOT file
  tree1->Write();

  return 0;
}
```

Listing 2: Code (simpleTreeExample.cpp file) to create the ROOT file "tree-root-file.root" to store tabular data

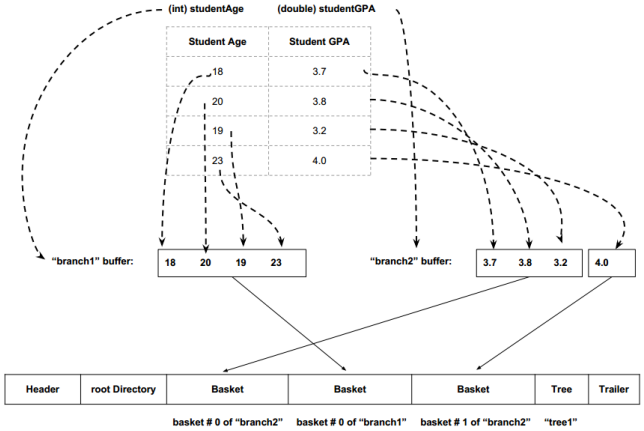


Fig. 6: Physical structure of the ROOT file storing the simple Table V

It is worth mentioning that here we used a simple table to show the functionality of trees, branches, and baskets. The columns of the table can be another table or the column can have a vector of values instead of a scalar value. In other words, they can be complex user-defined classes which contains other user-defined classes or they can be a vector or array of the user-defined or primitive types. We can consider each class as a

Fig. 7: Physical structure of a ROOT file containing trees

column or each field of the class as a column. Moreover, each value of the buffer of a branch can have different size. For example, if a table contains a column of string values, the size of each string can be different, and as a result, the related branch consists of baskets containing values of different sizes.

In general, the logical structure of a ROOT file consists of a set of trees and each tree has a set of branches. Each branch consists of a set of baskets and each basket contains a set of branch values (entries) the basket is associated with. A tree in the ROOT file is akin to a table. Each branch of the tree is similar to a column of the table. Each entry of the branch can be considered as a column value. A basket of a branch groups the set of entries of that branch (the values of the column), compress them and store them in the ROOT file. In other words, we can say that each branch of a tree consists of a set of baskets and those baskets contain a set of entries of the branch.

The physical structure of the ROOT file with trees consists of a $header$, $root\ directory$, set of $trees$, set of $baskets$ associated with the trees (branches of the trees), and $trailer$ (Fig. 7); The header, root directory, and tailer are the same as those for general ROOT files (see Fig 2). Each tree (an instance of $TTree$ class) consists a metadata describing the tree such as the branches it contains and the number of entries it has. We use the term $entry$ or $tree\ entry$ to refer a row in the table and $branch\ entry$ to refer to a value of the branch (cell in the table). Notice that the number of entries of the tree is always equal to the number of the branch entries in the branches of that tree. The baskets contain the actual data of the table (branch entries). Each basket also have some metadata describing the data it contains including the branch it is associated with, the basket number, the start and end entry numbers it contains (e.g the first basket contains the branch entries from 0 to 10, the second basket contains branch entries 11 to 18, and so on), the size of the basket, and etc.

It is worth mentioning that although branches (instance of $TBranch$ class) are not shown in the physical structure of a ROOT file, they are written as part of the associated tree in the file. Moreover, the baskets of the same branch are not consecutively stored in the ROOT file (baskets with the same color in Fig. 7 belongs to the same branch). Finally, the tree objects are written after all baskets are written in the file. They are at the end of the ROOT file (before trailer).

V. Conclusion

In this paper, we presented the basics of ROOT files in a different viewpoint which was more familiar to computer scientists. We provided an overview of the ROOT files and show ed the concepts of ROOT file, directory, and data objects using a simple code example. Moreover, we described the logical structure of a ROOT file, which is akin to UNIX directory hierarchy and physical structure which consists of a header, root directory, set of directories or data objects, and trailer. We elaborated each component of the physical structure through describing the functionality of each field of the component. Finally, we presented the ROOT files with trees that used to store tuples. Finally, we described the concepts of tree, branch, and basket and presented the physical structure of ROOT files containing trees.

We think the concepts we presented in this paper will help computer scientists to better understand the ROOT framework, especially ROOT files, the input/output part of the framework. The next step to make ROOT files more graspable for computer science is to design and implement a ROOT based library based on the fundamental concepts of the ROOT files introduced in this paper.

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] "Users guide for root i/o," https://root.cern.ch/root/htmldoc/guides/users-guide/InputOutput.html, (accessed May 01, 2018).

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

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