Improving Performance Statistics for ROOT Input/Output

Roger Xiao

July 30, 2014

Argonne National Laboratory

Department of High Energy Physics, the Atlas Collaboration, Lemont, IL, 60439, Unites States

Abstract:

The ALTAS detector at CERN's Large Hadron Collider gathers 10 petabytes of data annually. With so much information being stored and processed, it's essential that data storage, retrieval, and manipulation is efficient, with as little strain to memory, processing time, and data usage as possible.

ROOT is a software tool used to store, display, and manipulate ATLAS data. ROOT stores data in organized collections known as trees and stores these trees in "root files."

ROOT provides a method of monitoring the "performance statistics" of a tree's input and output. This data pertains to memory, data, and time usage during input/output operations and can be very useful for monitoring input/output efficiency. Currently, however, performance statistics can only be monitored for one tree at a time per file, and files in ROOT often contain multiple trees. Therefore, functionality was added to ROOT to allow for simultaneous performance statistics monitoring.

I. Overview

A. ATLAS and ROOT:

ATLAS, short for "A Toroidal LHC Apparatus," is one of seven particle detector experiments at CERN's Large Hadron Collider, the world's largest particle accelerator. It was one of the two experiments at CERN that were involved in finding the Higgs Boson, and is currently being used to research topics including micro black holes, extra dimensions, and dark matter.

ATLAS is one of the biggest data storages in the world, with over 100 Petabytes (100,000,000 gigabytes) of accumulated data about particle collisions, and is continuously increasing in size. As of now, it is producing about 10 Petabytes annually. ROOT is an interactive data analysis framework developed by CERN that is primarily used to manage and manipulate this data.

Not only ATLAS, but all seven of the detector experiments at CERN'S LHC, as well as experiments outside of CERN, such as Fermilab's Tevatron, and Brookhaven National Lab's PHENIX detector, use ROOT for data analysis. ROOT is also being used in experiments in astrophysics and computational neuroscience and is the basis for developing software at Fermilab's NOVA neutrino experiment and Long Baseline Neutrino Experiment. As of now, ROOT's user base is estimated to be 20,000 people.

For more general information about ROOT, see the articles *ROOT-A C++ framework for petabyte data storage, statistical analysis and visualization* by Rene Brun and *Introduction to ROOT* by Jane Fiete Grosse-Oetringhaus.

B. Data Storage:

With so much data being processed by ROOT, it is very important that its input/output functionality is as efficient as possible i.e. data can easily and effectively stored and accessed using ROOT with minimal strain on memory, CPU, and processing speed.

Data in ROOT is stored in containers-organized collections related to a common subject. The containers in ROOT are known as trees. Tree is split into branches, each containing data regarding a specific subcategory of the tree. If a tree were to represent a particle collision event, there could be branches designated to the x, y, and z coordinates of the particles before and after the collision, as well as branches designated to velocity and energy. Each basic variable, structure, and object value inside a branch is stored in what is known as a leaf. Trees are created by direct user commands through the ROOT interface and through program scripts written by users and executed by ROOT.

These trees are saved in what are known as ROOT files. From these files, data from trees can be accessed using ROOT and displayed using graphs and histograms, compared to similar data, and used for regression analysis, 3D visualization, and mathematic operations. ROOT provides a variety of parameters which users can adjust to optimize the I/O of their applications.

For more information about how data is gathered and stored in ROOT, see the articles *The Event Data Store and I/O Framework for the ALTAS Experiment at the Large Hadron Collider* and *Supporting High-Performance I/O at the Petascale: The Event Data Store for ATLAS at the LHC* by Peter van Gemmeren and David Malon.

II. The Problem:

ROOT uses an object known as TTreePerfStats to monitor what are known as the "performance statistics" of input and output regarding trees. These statistics include the number of leaves in the tree, the number of bytes being read, the number of times files are accessed, and the size of the data on a disc in bytes. TTreePerfStats is a valuable tool for monitoring input/output efficiency for specific trees and programs, and is especially useful for monitoring changes and progress when efforts are made to make input/output more efficient.

The problem with TTreePerfStats, however, is that it is only able to monitor input/output for one tree at a time per file. Often, the files that ATLAS uses contain multiple trees. Event Summary Data (EDS), event data written as the reconstruction of raw data from ATLAS's event filter, is stored in ROOT files eight trees at a time, with 700 kilobytes per event. Analysis Object Data (AOD) is a derived, reduced version of ESD suitable for physics analysis, with 200 kilobytes per event and stored eight trees at a time (van Gemmeren and Malon, 2010). However, if multiple TTreePerfStats objects are initialized and used in the same operation for these multiple trees, only the TTreePerfStats object initialized last will be functional, and will end up reading performance statistics data for the file as a whole instead of for the individual tree it was assigned to, which is often not desirable. Therefore changes were made to ROOT to allow for the use of simultaneous TTreePerfStats objects within the same function in order to monitor trees individually.

III. Modifications and Solutions:

A. Adjusting Pointers

gPerfStats, the global object in ROOT that controls all performance statistics operations, points to a single TTreePerfStats object at a time after one has been defined and assigned to a tree. If you are working with multiple trees at a time, gPerfStats will only point to the last TreePerfStats object that was defined, and only that object will record data. It will have the performance statistics data of all of the trees in the file combined and will be the only TTreePerfStats object with displayable data. Figures A and B give a visual representation of this pointer flaw.

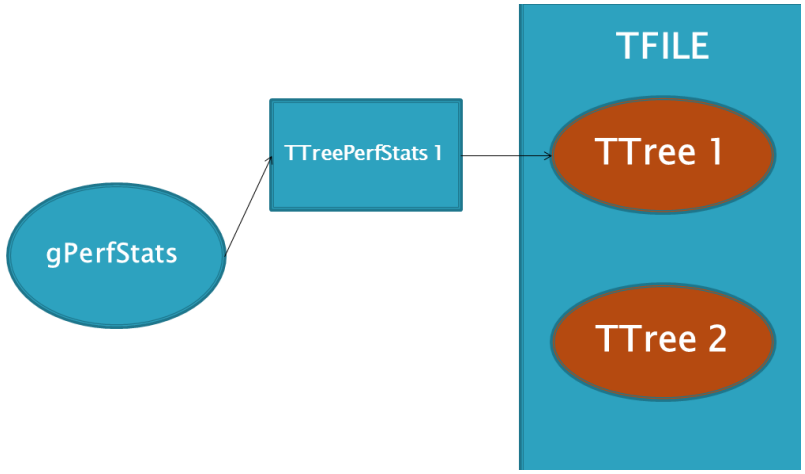


Figure A: Consider a file with two trees. If a TTreePerfStats object is assigned to tree 1, gPerfStats will point to TTreePerfStats 1. All performance statistics data from tree 1 will be stored in TTreePerfStats 1. However, all performance statistics data from tree 2 will also be stored in TTreePerfStats 1

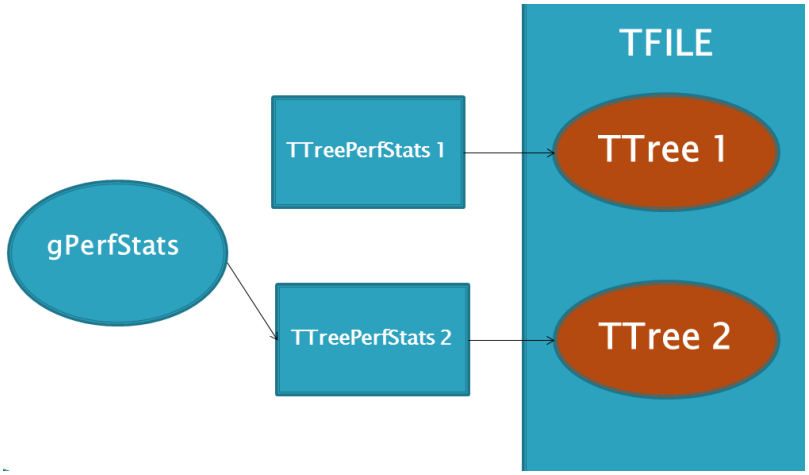


Figure B: If for the same file as in figure A a TTreePerfStats object is defined for the second tree, gPerfStats will now point to the second TTreePerfStats object. All performance statistics data will be recorded by the second object regardless of whether said data comes from tree1 or tree2. TTreePerfStats 2 now has the combined performance statistics of both trees and TTreePerfStats 1 will no longer be updated

The solution was to allow gPerfStats to adjust and point to different TTreePerfStats objects whenever said objects needed to record performance statistics i.e. whenever their respective trees were being operated on and therefore producing performance statistics data. To change the gPerfStats to point to specific TTreePerfStats objects accordingly, I first created a performance statistics object called fPerfStats. This object is a defined in TTree.h, the main header file in charge of the functionality of and operations of trees. fPerfStats is intended to be set to the address value of a tree's corresponding TTreePerfStats object.

A function called SetPerfStats was then defined in TTree.cxx as a function that equates the current fPerfStats value to that of an imported parameter value. In the constructor of TTreePerfStats

in TTreePerfStats.cxx, the line "fTree->SetPerfStats(this)" is used to set the current tree's fPerfStats value to the value of the current TTreePerfStats object's address.

This is done to allow trees to point back to their designated TTreePerfStats objects. TTreePerfStats objects point to their respective trees since the tree address value is a parameter of the TTreePerfStats constructor. Essentially, TTreePerfStats objects "know" which tree they are assigned to, and such information can be obtained from TTreePerfStats objects. Trees, however, do not point to their TTreePerfStats objects and have no knowledge of which TTreePerfStats objects are assigned to them. "fTree->SetPerfStats(this)" sets fPerfStats, an object of the tree, equal to the address of the tree's TTreePerfStats object. Now trees point back to their respective TTreePerfStats objects so that a tree knows which object is assigned to it. This allows for the functionality of a function I implemented in TTree.cxx called GetPerfStats, which returns the value of fPerfStats and of the current TTreePerfStats' address whenever it is called.

Functionality in TBasket.cxx, the main C++ file in charge of the functionality for reading from and writing to files, is then changed. gPerfStats has to be adjusted before a function called ReadBuffer is called, since ReadBuffer contains the functionality for updating performance statistics using gPerfStats, as well as the functionality for reading files. The function GetPerfStats is used to check the current tree for an fPerfStats value and then retrieve it. gPerfStats is then set equal to said fPerfStats, which at this point is equal to the value of the address of the current TTreePerfStats object. gPerfStats now points to the current TTreePerfStats object, allowing the TTreePerfStats object to draw performance statistics data from its tree. This is now done automatically whenever performance statistics must be drawn from a tree so that no performance statistics data is sent to the wrong TTreePerfStats object.

With an adjustable gPerfStats object, ROOT is now able to monitor the performance statistics of multiple trees at once using TTreePerfStats objects. The data for each TTreePerfStat is displayable through graphs and histograms-see appendix B for a comparison of performance statistics data from before and after this change was implemented. See figure C for a visual demonstration of the new pointing implementation.

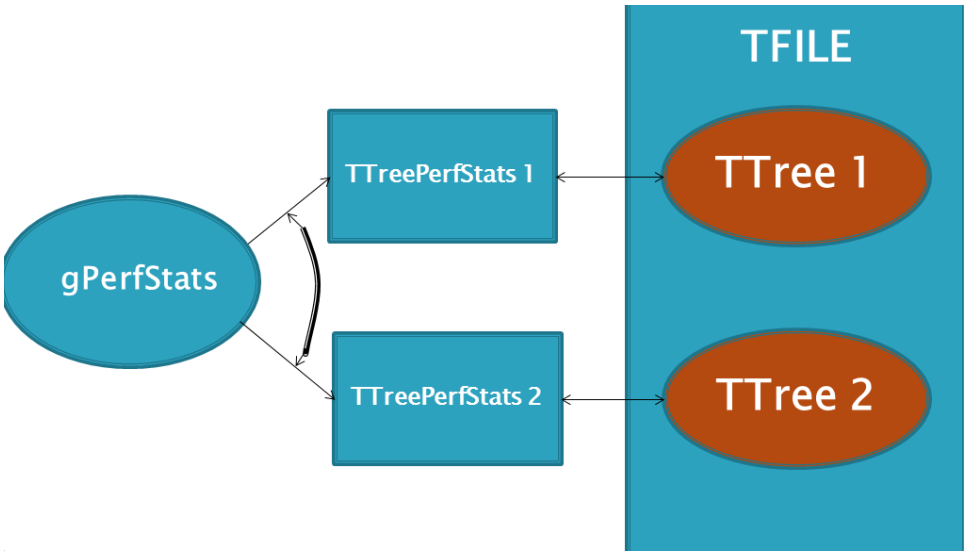


Figure C: Through the use of GetPerfStats and SetPerfStats, gPerfStats can now be adjusted to point to different TTreePerfStats object address values in accordance to which object currently needs to gather data. This allows for accurate, simultaneous performance statistics reporting. The double arrows between the TTreePerfStats objects and the trees represent how the TTreePerfStats objects now point to their assigned trees and vice versa

B. Modifying Counter Algorithms

However, even though gPerfStats is able to adjust to multiple TTreePerfStats objects, there are still two major inaccuracies in the statistics TTreePerfStats reports. The reported number of times the file that the tree comes from is called for reading (ReadCalls) and the number of bytes read for each tree (BytesRead) is still shown as the sum for all of the trees in the file combined.

The number of read calls is monitored by a counter called fReadCalls. Originally, fReadCalls is incremented every time ReadBuffer is called. TTreePerfStats then retrieves the total count from fReadCalls after TTreePerfStats is finished with its functionality. The number of bytes read is calculated the same way, with a counter called fBytesRead incrementing by the number of bytes currently being read every time ReadBuffer is called. The problem is that this method has no way of discriminating between which tree and which TTreePerfStats object ReadBuffer is being called for. Every call to ReadBuffer increments the same ReadCalls and BytesRead counter and the total fReadCalls and BytesRead numbers retrieved by each TTreePerfStats object at the end of its execution is thus the same combined value.

The code has been modified so that fReadCalls and fBytesRead are incremented within the FileReadEvent function, a function called by ReadBuffer that gathers statistical data about file read events, instead of by the ReadBuffer function itself. FileReadEvent is called using gPerfStats, which has previously been modified to adjust its value to that of the address of the current TTreePerfStats object. Thus FileReadEvent, like gPerfStats, can discriminate between different trees and TTreePerfStats objects, unlike ReadBuffer, which is not a function of gPerfStats. fReadCalls is now incremented by one and fBytesRead is incremented by the number of bytes read for that specific call every time FileReadEvent is called.

This method also fixes another inherent algorithm flaw that was present in TTreePerfStats functionality for counting ReadCalls and BytesRead. The original method of counting ReadCalls and BytesRead in the function ReadBuffer does not discriminate between calls to ReadBuffer during which data is actually being read and calls where it is not. There several calls to ReadBuffer in the beginning of every TTreePerfStats object's functionality where no data is being read. These are from various constructor and initialization functions. During these calls, gPerfStats has a value of 0, and thus is not pointing to any TTreePerfStats object. Since there is no TTreePerfStats object to collect performance statistics data, no performance statistics data is read, yet fReadCalls and fBytesRead are still incremented. These calls to ReadBuffer also do not trigger a call to FileReadEvent, since FileReadEvent, as a function of gPerfStats, is only called on if gPerfStats has a nonzero value. By incrementing fReadCalls and fBytesRead inside FileReadEvent, it is assured that only for calls to ReadBuffer during which gPerfStats has a nonzero value and something is being read are ReadCalls and BytesRead incremented.

Now performance statistics for ReadCalls and BytesRead are unique to the separate trees they are monitoring and only represent the calls where something is actually being read. TTreePerfStats can now effectively read performance statistics from multiple trees simultaneously and independently.

IV. Implications and Applications:

A. Split Level and Basket Size

ROOT users often have the task of optimizing split levels and basket sizes of trees. A ROOT tree branch can have its own sub-branches, and split level pertains to the number of sub-branch layers that are present in a tree. Split level can be manually adjusted to any value between 0 and 99 based on organization preferences, but certain split levels come with a cost.

Every branch in a ROOT tree has a basket-an area to temporarily store data that is being tranfered to and from files. Baskets have a default size of 32000 bytes. However, if split level is high, resulting in a large quantity of branches, basket size needs to be decreased in order to save memory. Baskets too small, however, may be insufficient for storing required data for transfer. To optimize basket size and split level, performance statistics are used to analyze data about leaf numbers, read sizes, CPU time, and disk I/O, factors that affect the optimization of I/O efficiency that are directly affected by factors such as split level and basket size.

B. Caching:

A method known as caching is used to make input/output with trees and branches more efficient. Caching is the act of allocating blocks of memory for temporary storage of data more likely to be used. If data required can be found in the cache, it can be quickly retrieved from the cache instead of from the main storage location where that data originally came from. Caching takes advantage of patterns in memory access to predict which data will be accessed in the future.

ROOT is able to cache tree branches together. Caching speeds-up performance considerably, especially when the tree is accessed remotely via a high latency network, where reading many branches individually would cause large buildups of delays as opposed to reading them all at the same time from a single location.

The tuning and adjusting of caches in ROOT requires close monitoring of input/output statistics, and often branches have to be cached from multiple trees at a time. With the new functionality of performance statistics, it is now possible to observe cache size, read calls and read size, disk and CPU time, and disk IO from multiple trees at a time in order to monitor caching.

V. Conclusion:

The methods of recording and reporting performance statistics in ROOT have been improved in order to contribute to input/output efficiency and will be included in the next version of ROOT. By allowing the global object responsible for performance statistics functionality to be able to adjust to different trees and TTreePerfStats objects, multiple trees from the same file can now be monitored simultaneously and independently, and performance statistics are no longer only read by file. Modifications to performance statistics counting algorithms have allowed for more accurate reporting of read call and data reading values for performance statistics in general. This all allows for more powerful and more accurate analysis and works towards the goal of better organizing the hundreds of petabytes of data collected by the LHC detectors as scientists continue to use them for groundbreaking work.

References

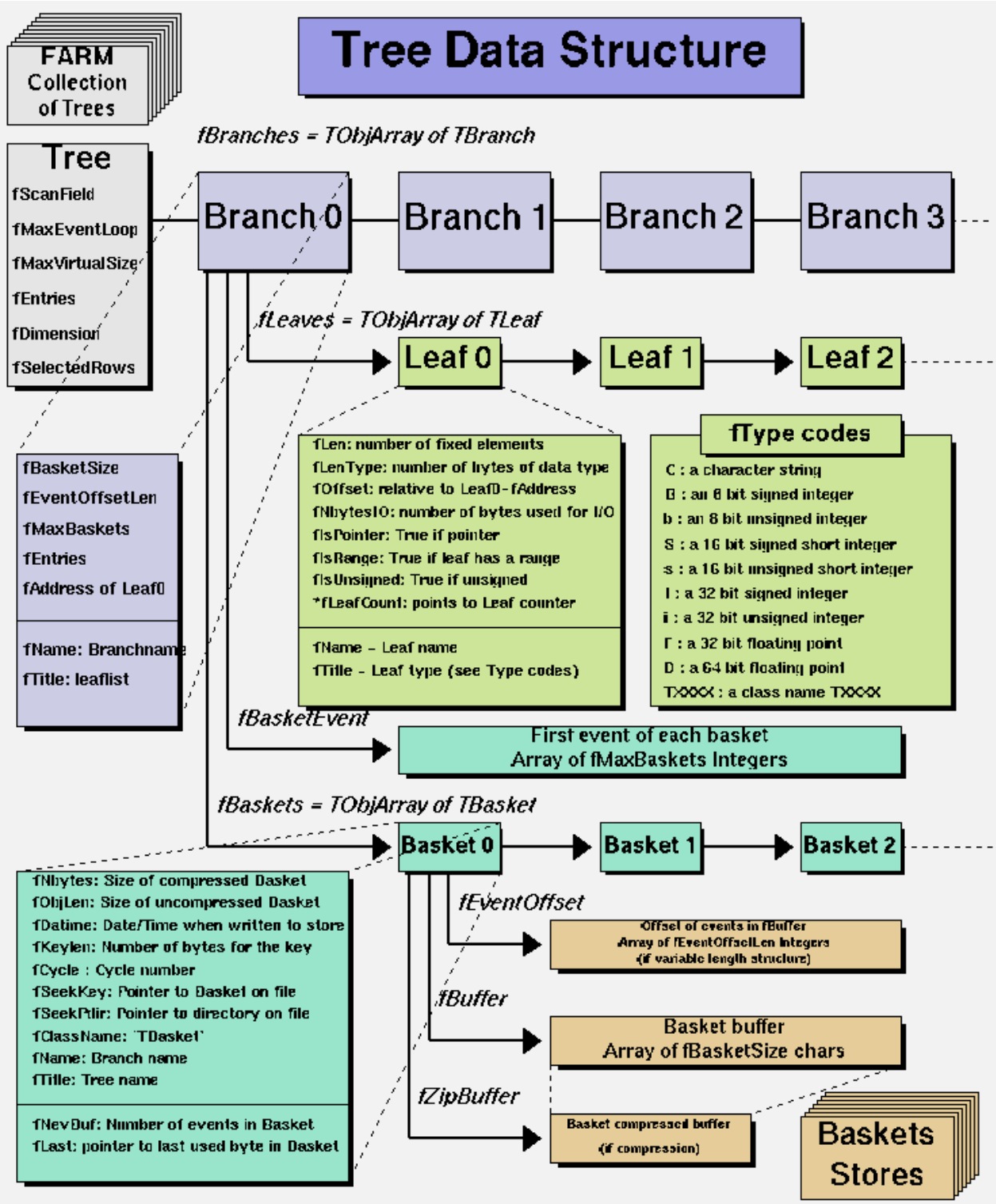
Grosse-Oetringhaus, Jan F. *Introuction to ROOT*. Indico.cern.ch. 11 July, 2011. Web. 18 July, 2014

Malon, David, and van Gemmeren, Peter. *The Event Data Store and I/O Framework for the ALTAS Experiment at the Large Hadron Collider*. ieee.org. Web. 5 May, 2014

Malon, David, and van Gemmeren, Peter. *Supporting High-Performance I/O at the Petascale: The Event Data Store for ATLAS at the LHC*. ieee.org. Web. 5 May, 2014

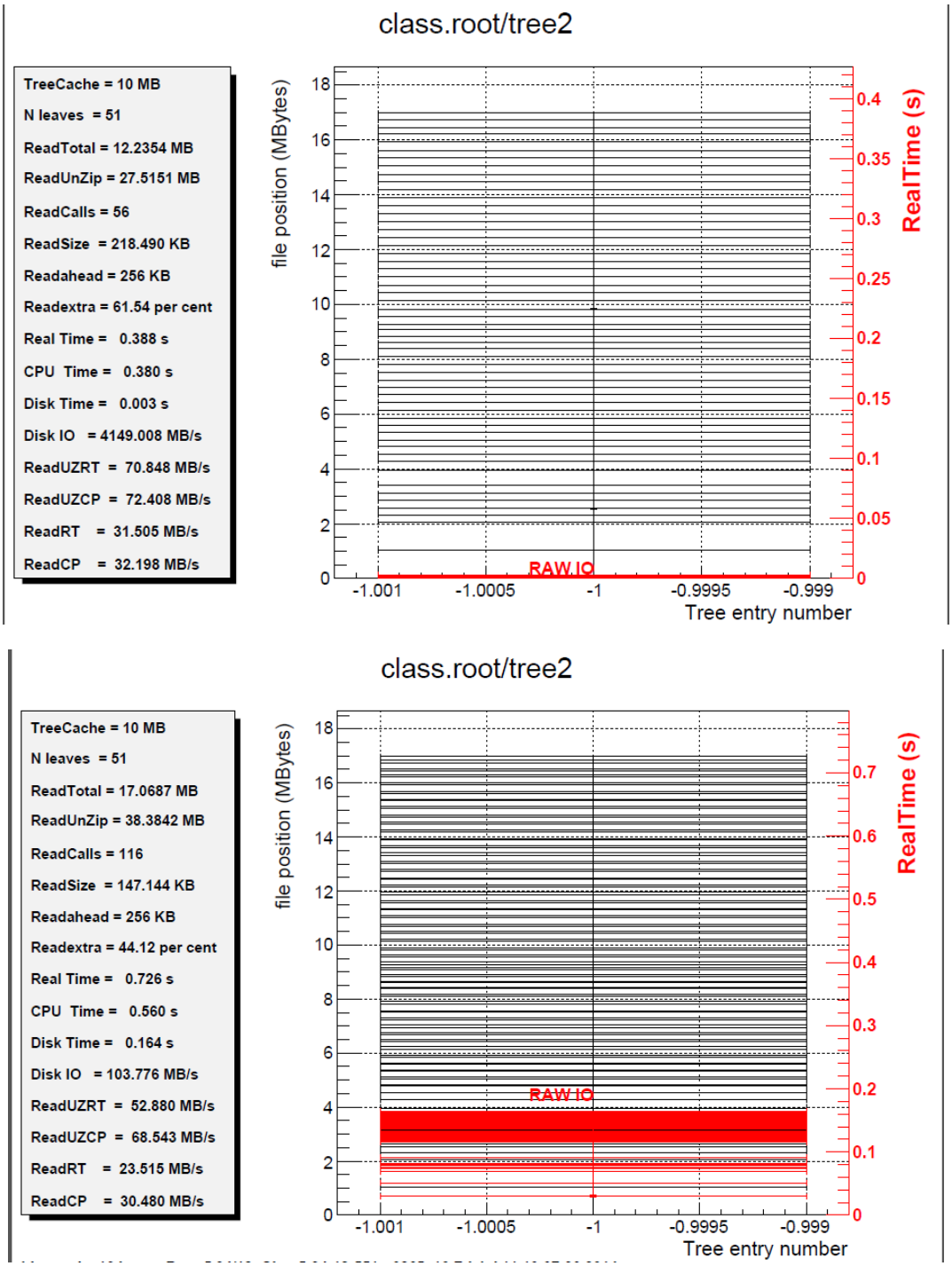
R. Brun, F. Rademakers, *et al*. *ROOT-A C++ framework for peabyte data storage, statistical analysis and visualization*. Sciencedirect.com. 28 April, 2009. Web. 18 July, 2014

Appendix A



The TTree Class and Data Structure

Appendix B



The displayed statistics given by a TTreePerfStats objects for one of two trees in a file run by the modified code (top) vs. displayed statistics given by the same TTreePerfStats object run by the original code (bottom). The TTreePerfStats object on the top has data for a whole file while the one on the bottom only reads for its designated tree