CERN Summer Student Programme
2016

Oulu University of Applied Sciences

Designing new interfaces for ROOT data processing

Kalle Elmer Vuorinen

September 2, 2016

Supervisors:
Gerardo Ganis
Pere Mato Vila

Institute: CERN
Department: EP-SFT

Contents

1 Introduction **2**

2 ROOT **3**

3 Development and design **3**
3.1 Chains of functional primitives 3
3.2 DataFrame 4
3.3 Implementation details 5
3.3.1 Python 6
3.3.2 JIT compiled C++ 6

4 Results **6**
4.1 Functional Chains 6
4.2 PyTreeReader 8

5 Future steps **9**

6 Acknowledgments **9**

Abstract

ROOT is a C++ framework for data analysis provided with a Python interface (PyRoot). ROOT is used in every Large Hadron Collider experiment. This project presents a way of reading ROOT TTree by using a new class called DataFrame, which allows the usage of cache and functional chains. Reading TTrees in Python has been quite slow compared to the C++ way of doing it and for this reason we also bring the possibility to read them with just-in-time (JIT) compiled C++ code, using another new Python class called TreeReader.

1 Introduction

CERN or *The European Organization for Nuclear Research* operates the largest particle physics laboratory in the world. It houses the largest and most powerful particle collider to date: the Large Hadron Collider (LHC). From the CERN Control Centre the proton beams inside the LHC are made to collide inside four particle detectors, positioned at various locations along the accelerator ring: ATLAS, CMS, ALICE and LHCb. As a Summer Student I was a part of EP-SFT group.

The abbreviation EP-SFT stands for Experimental Physics – SoFTware development for Experiments. The main responsibility of the group is to develop and maintain the common scientific software of the experiments in a close collaboration with the EP groups, IT department and external HEP institutes. The majority of the group works with projects that are part of the Applications Area of the LHC Computing Grid also known as LCG [1].

SFT projects have four different core areas which are: Simulation, Libraries and Frameworks, Distributed Systems and Collaborations with Experiments. For analyses in Large Hadron Collider experiments a data structure is used called tree. More specifically these trees are abstracted data types. The events from the collisions are stored into a structure called tree.In ROOT we have a similar class called TTree [2]. TTree is a list of independent branches and each one of these branches has its own definition and list of buffers. In an analysis the user can specify whether the analysis needs only one branch or multiple ones from a tree. This way the analysis can speed up by reading the needed parts of the tree instead of the whole structure. This is where the chains of functional primitives come in handy.

Chains of functional primitives or functional chains are a way of using ROOT in a more sophisticated way. One can think of a time line on which events occur. Each event effects the result itself. Apache Spark offers us a *Resilient Distributed Data* set or RDD that has a similar concept behind it [3].

The performance of reading trees has also been an issue. Looping over the trees and reading the entries is much slower in Python compared to C++. This project presents possibilities to improve the speed of reading TTrees and the ways of using functional chains with a class called DataFrame.

2 ROOT

ROOT is an object-oriented data-analysis framework that is mainly build for High Energy Physics, but it's also used elsewhere. It offers tools for effectively handling data and various toolboxes, for instance multivariate analysis. ROOT is used in all of the major LHC experiments at CERN. It is a modular scientific framework which is written mainly with C++ but also has integrations with Python and R. C++ is the main language of ROOT because of the performance it provides. More thorough information about ROOT is found in [4].

3 Development and design

3.1 Chains of functional primitives

The idea behind the chains of functional primitives came from the Apache Sparks Resilient Distributed Data set (RDD). More specifically the idea was to create a similar way of using transformations and actions in ROOT [5]. A new class DataFrame is introduce describing a dataset and providing the relevant tools

Transformations are lazy functions, in the sense that they do not compute their results right away. Instead, they remember the transformations applied to some base data set, a DataFrame object in this instance.

Examples of transformation functions in DataFrame:

- Filter(): Filters out all the elements of the data set according to the function
- Map(): Returns a new data set with the elements changed by the function
- FlatMap(): Maps a function over a data set and flattens the result with one level

Actions are functions that return a value to the driver program after running a computation on the data set. In functional chains they also apply all the transformations before they compute their own function.

Some examples of action functions in DataFrame:

- Draw(): Draws the histogram according to its options
- Histo(): Creates a histogram and fills it
- Cache(): Caches the result for later use
- Head(): Prints the result to a table

3.2 DataFrame

DataFrame is a new Python class in ROOT. It works as a data set class and it describes a tree. When a DataFrame object is created it needs a tree for initialization.

The features of functional chains are provided when a user uses DataFrame. The simplest usage of functional chains is adding one filter to the data set. For example, the following can be computed:

```
dataFrame = DataFrame(testTree)
```

```
dataFrame.filter(lambda e : e.Children() > 4).cache().head(5)
```

Category	Flag	Age	Service	Children	Grade	Step	Hrweek	Cost	Division	Nation
300	13	49	24	5	9	9	40	10039	L	F
201	15	58	26	5	11	9	40	14390	P	G
560	14	47	19	5	5	13	20	4624	E	F
202	15	47	19	6	11	4	40	13574	P	D
102	7	36	5	5	10	1	40	11026	E	F

After initialization of the DataFrame it can start to receive transformations to the data set. Transformations are lazy functions, which do not perform any computing. The filter will be added to two lists inside the class, which are called transformations and filters. When an actual action is called, the program uses the correct order from transformations list. When the function is called from the transformations list it will also compare it to the contents of filters. This is how the program identifies transformations and uses them in the right way. In this example the program uses the function as a filter. The same pattern works for each type of transformation. It first adds a function to the transformations and to the type list of the function (for example filters list for filters).

When we call an action, for example histo(), the program will create a histogram according to the quantity of the variables. After creating a histogram it will loop over the entries and use the transformations to the entries if any are given.

```
dataFrame.filter(lambda e : e.Age() > 45).cache().histo('Age:Cost').Draw('COLZ')
ROOT.gPad.Draw()
```

DataFrame brings the possibility to cache results into a TEntryList [6]. By caching results the user will first loose some time but if the user is constantly using the same computing chain they will actually save a lot of time since the computing has already been made.

```
(dataFrame.filter(lambda e : e.Age() > 45).cache()
.filter(lambda e: e.Cost() > 8500).histo('Age:Cost').Draw('COLZ'))
ROOT.gPad.Draw()
```

3.3 Implementation details

DataFrame class uses PyTreeReader for reading the entries of the trees. PyTreeReader is initialized inside the constructor of DataFrame with a tree. It has the same idea as the TTreeReader, however it uses Python to iterate through the entries [7].

3.3.1 Python

The python side of the PyTreeReader is rather simple: It has a string of class templates, which work to generate C++ code from Python. Python works as a getter and setter of the variables,which it can get from the given tree. It will look for branch types and names for the C++ generated class. It uses ROOT's gInterpreter to declare the generated class. After the class is generated with Python the tree can be iterated over using the speed of C++ but still using the iterator from Python.

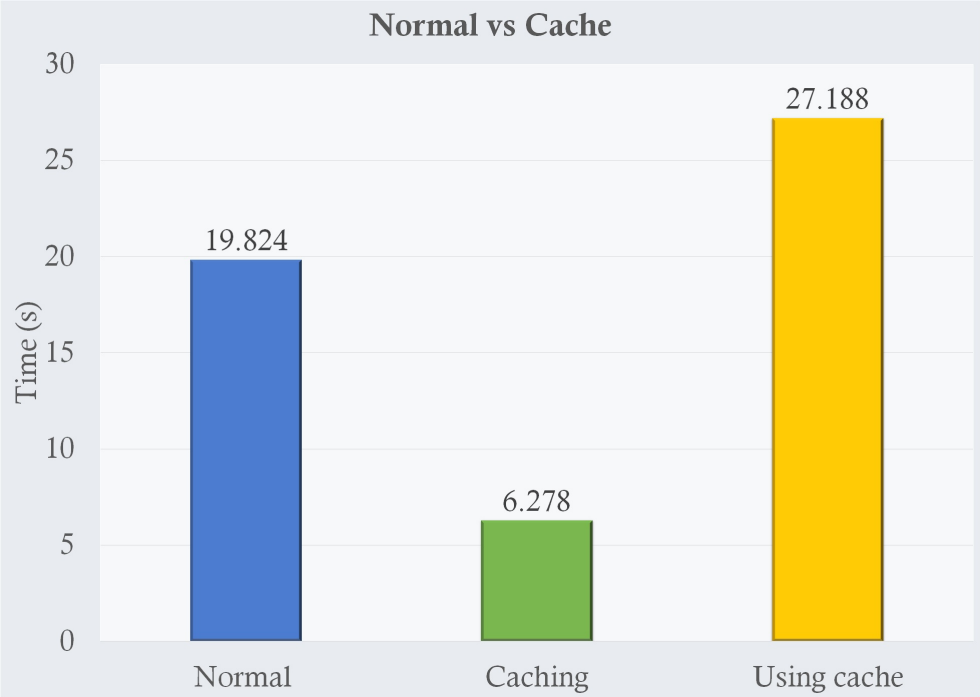
3.3.2 JIT compiled C++

The speed and reasoning of PyTreeReader comes from using JIT compiled C++ code with which we can generate a TreeReader class in C++. The word JIT stands for just-in-time, which means that we will compile and declare the C++ class before using it.

4 Results

4.1 Functional Chains

To prototype the class we use the tree which is available in ROOT under tutorials and tree called cernstaff.root. File was multiplied by 1300 to get over 4 million entries to get better demonstration on caching and usage of functional chains. Tests were computed using SWAN notebook service [8] and before every test the files cache was cleared and the notebook was restarted to get cold reads .

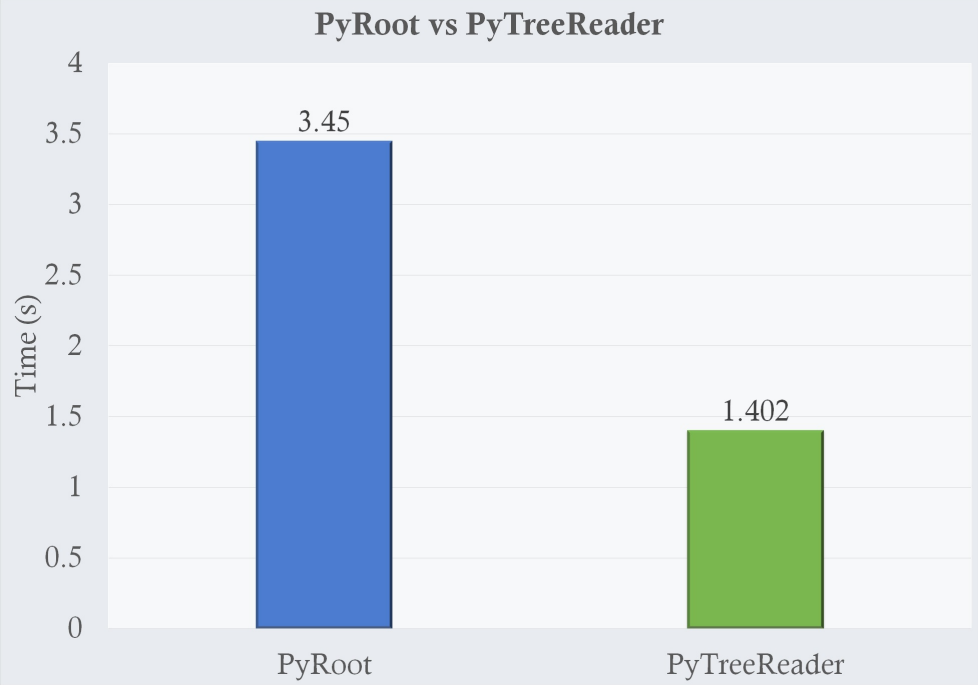


From the bar chart above we can see that the results are self-explanatory. When analysis is done in a normal way it will take approximately 19.8 seconds. If the result are cached it will take longer, in this case 7.4 seconds longer. The most important idea to notice here is that the user will lose time if caching if used in every case. The user should only use cache when result are going to be reused.

Caching should be used when similar analyses are being computed twice or more. For example, if a normal analysis is computed twice according to the test results it would 39.6 seconds. If the same is done with cache and cached values, we will have two different values. With caching the program will take 27.2 seconds but on the second run it will only take 6.3 seconds. In total using cache and cached values it would take 33.5 seconds, which is already faster than the original way. Computing a similar analysis over and over again will make caching more useful.

4.2 PyTreeReader

PyTreeReader tests were done with CMS data. The structure of the tree in this case is very complex but it equates extremely well to a real analysis. This CMS.root file had 650000 entries and the file size was 20 GB. First the test was ran by using the original way of Python looping with setting the branch before looping. The second test performed the same test with PyTreeReader. With PyTreeReader a branch does not need to be specified before looping over the entries.



Looking at the bar chart above we can see that the original Python looping took on average 3.45 seconds, while using PyTreeReader it took 1.402 seconds. In this case PyTreeReader takes on average a third of the time that the original way of looping over the TTree in pyRoot takes. This is an agreement with the results of other tests [7].

5 Future steps

DataFrame is a working skeleton class that can be used to develop the primitives of functional chains further. Functional chains and caching work perfectly with adding filters, but there are still some issues regarding multiple filters before and after the cache. Those issues are more or less design challenges and not technological issues.

Functional chains is a feature that can ease a new user of ROOT to learn to read trees more efficiently. This RDD type feature is a step closer to integrating ROOT with Apache Spark.

The next step in this project would be to develop DataFrame under C++ and to develop more transformations and actions to cover Sparks functionalities. Currently there are some technical difficulties for caching functions. Next CLANG version may allow to overcome these problems [9].

6 Acknowledgments

Finally, I would like to express my gratitude towards my both supervisors: Gerardo Ganis and Pere Mato Vila. They have been the best supervisors I could have hoped for. They brought a lot of insight to the project. I have learned more than I would have hoped for during the summer and without my supervisors this summer would not have been nearly as amazing as it has been.

References

[1] EP-SFT.
https://ep-dep-sft.web.cern.ch/

[2] ROOT, TTree Class Reference.
https://root.cern.ch/doc/master/classTTree.html

[3] Apache Spark.
http://spark.apache.org/docs/latest/programming-guide.html

[4] ROOT Data Analysis Framework.
https://root.cern.ch/

[5] Jim Pivarski, Chains of functional primitives, 2016.
https://indico.cern.ch/event/535391/contributions/2175710/attachments/1277373/1895792/main.pdf

[6] ROOT, TEntryList Class Reference.
https://root.cern.ch/doc/master/classTEntryList.html

[7] Danilo Piparo, PyTreeReader, 2016.
https://github.com/dpiparo/pytreereader

[8] The Swan Service.
https://swan.web.cern.ch/

[9] Vasil Vassilev, Native Language Integrated Queries with CppLINQ in C++, 2014.
https://indico.cern.ch/event/258092/contributions/1588604/attachments/454240/629656/cpplinq.pdf