ROOT6: History, Performance, Functionality and a Look to the Future

D. Piparo (CERN, EP-SFT) for the ROOT Team

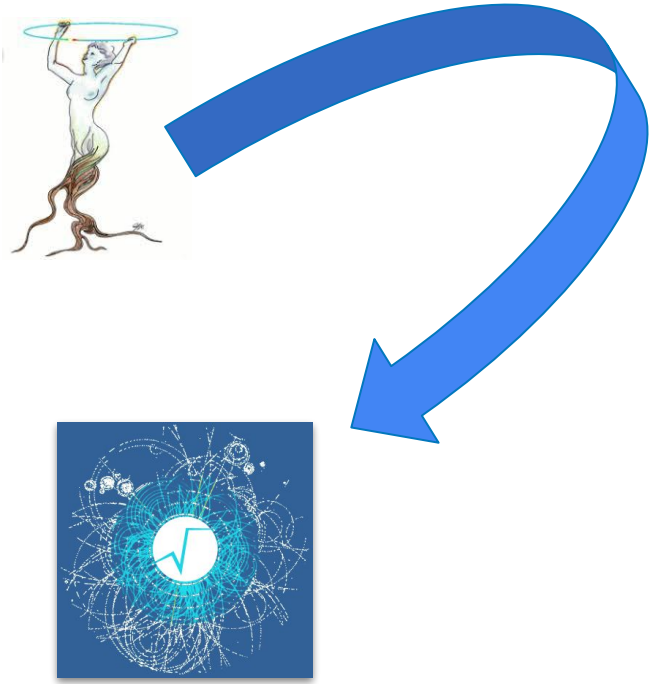
ROOT

Data Analysis Framework

https://root.cern

This talk

- Introduction
- Motivation, Some History
- Architecture
- Most Important Features
- Future Evolution



Thanks to G. Amadio, E. Tejedor, X. Valls, B. Bellenot and E. Guiraud for the contributed slides!

Introduction

The Team and the POW

Kim Albertsson
Guilherme Amadio
Bertrand Bellenot
Philippe Canal
Olivier Couet
Enrico Guiraud
Lorenzo Moneta
Axel Naumann (Project Leader)
Danilo Piparo
Oksana Shadura
Yuka Takahashi
Enric Tejedor
Xavier Valls
Vassil Vassilev

Main External Contributors

Brian Bockelman
Sergey Linev
Marsupial
Zhe Zhang

- *Not 18 ftes, not everybody working 100% on ROOT!*
- *2018 Program of Work:*
  https://docs.google.com/spreadsheets/d/1od9X5HpqcQ7SIvyloqfnEpkzav4wTB-AIfwDxBnSnhg/edit#gid=0

Communicate More, Communicate Often

- Thanks for the invitation to the ALICE Offline Week!
- Engaging with experiments is crucial for us
  - Core software, central processing, triggering and *analysis*…
- Let's continue to talk: forum, email, in person!
- Collaborate with us: https://github.com/root-project/root

ROOT Release Schedule 2018

- ~now: **6.13** -Development release
- May: **6.14** - Production release
- September: **6.15** - Development release
- November: **6.16** - Production release

How to Use the “Latest ROOT”

Provided by EP-SFT

- With LCG releases on CVMFS!
  - ~400 packages coherently built, a full ecosystem, also for analysis
  - http://lcginfo.cern.ch for the full list of packages!
- LCG Soft + “ROOT of yesterday” (nightly builds):

```
source /cvmfs/sft.cern.ch/lcg/views/dev3/latest/<your  architecture>/setup.sh
```

- LCG Soft + Released ROOT (XY being 92, 93, …):

```
source /cvmfs/sft.cern.ch/lcg/views/LCG_XY/<your  architecture>/setup.sh
```

LCG releases also available in SWAN. You can use ROOT in your browser: https://swan.cern.ch

Motivation and History

ROOT6: A Big Change

Problem: ROOT5 interpreter Cint

- C parser, with some C++ capabilities
- Reflection, I/O: no support for the new C++ standard at the time - C++11
- Interactivity (not only I/O) required dictionaries
  - Select classes, functions…
- Visible cracks in the infrastructure: e.g. support for gccxml on OSX

Solution: Replace Cint with Cling

- Cling: a C++ interpreter based on Clang/LLVM technology

Side effect: a lot of work!

- The vision: “the benefits will outweigh the cost”

A production quality compiler toolkit!

Investments are needed for sustainability

Challenges Involved

Push forward software technology

- Cling: first of its kind (JIT compilation of C++!)
- Rewrite of entire ROOT Core components
  - Including layer between ROOT and its interpreter (“ROOT Meta”)

Compared with CINT, optimised during 20y!

Existing features to support, a rich set of new ones

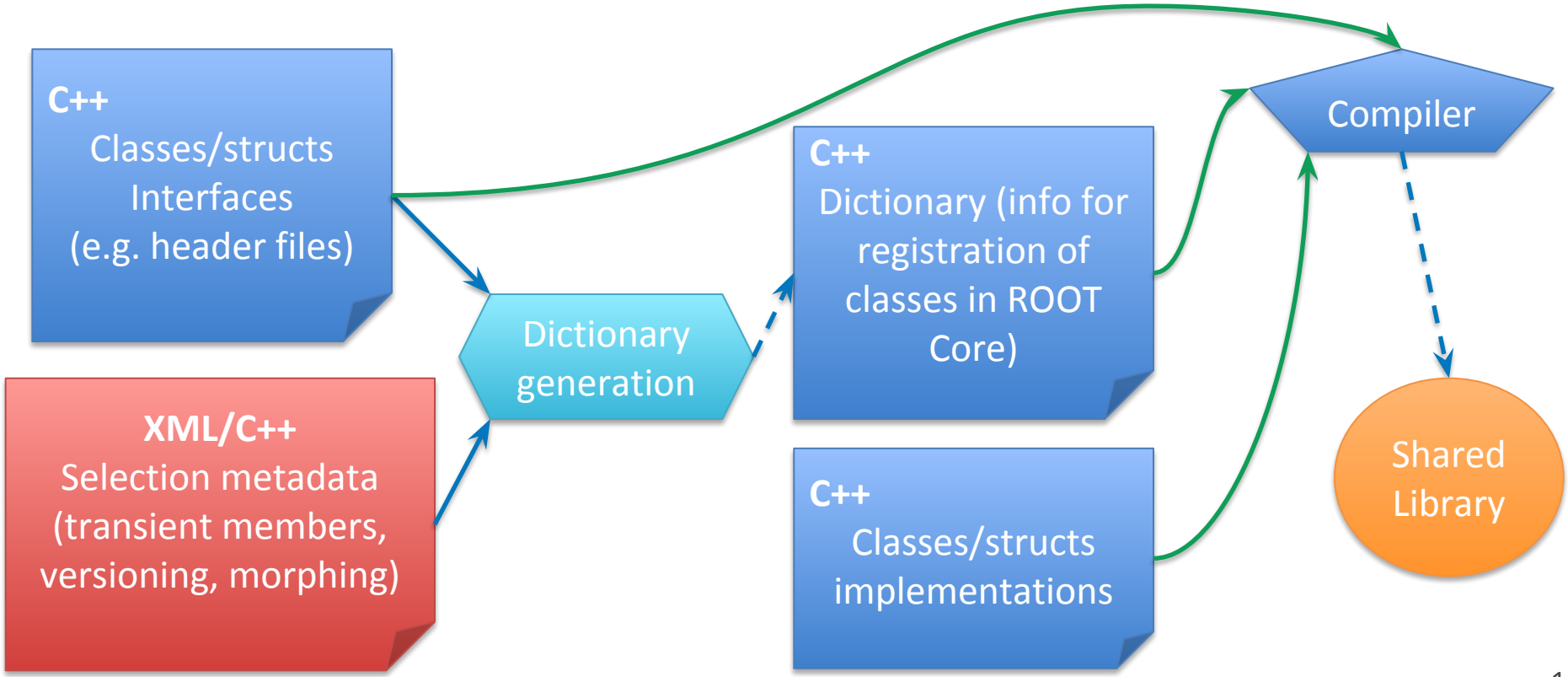
- Many end users: O(10^4) - Backward compatibility needed
- Experiments stacks: multi-MLOC software system

A real quest but an opportunity!

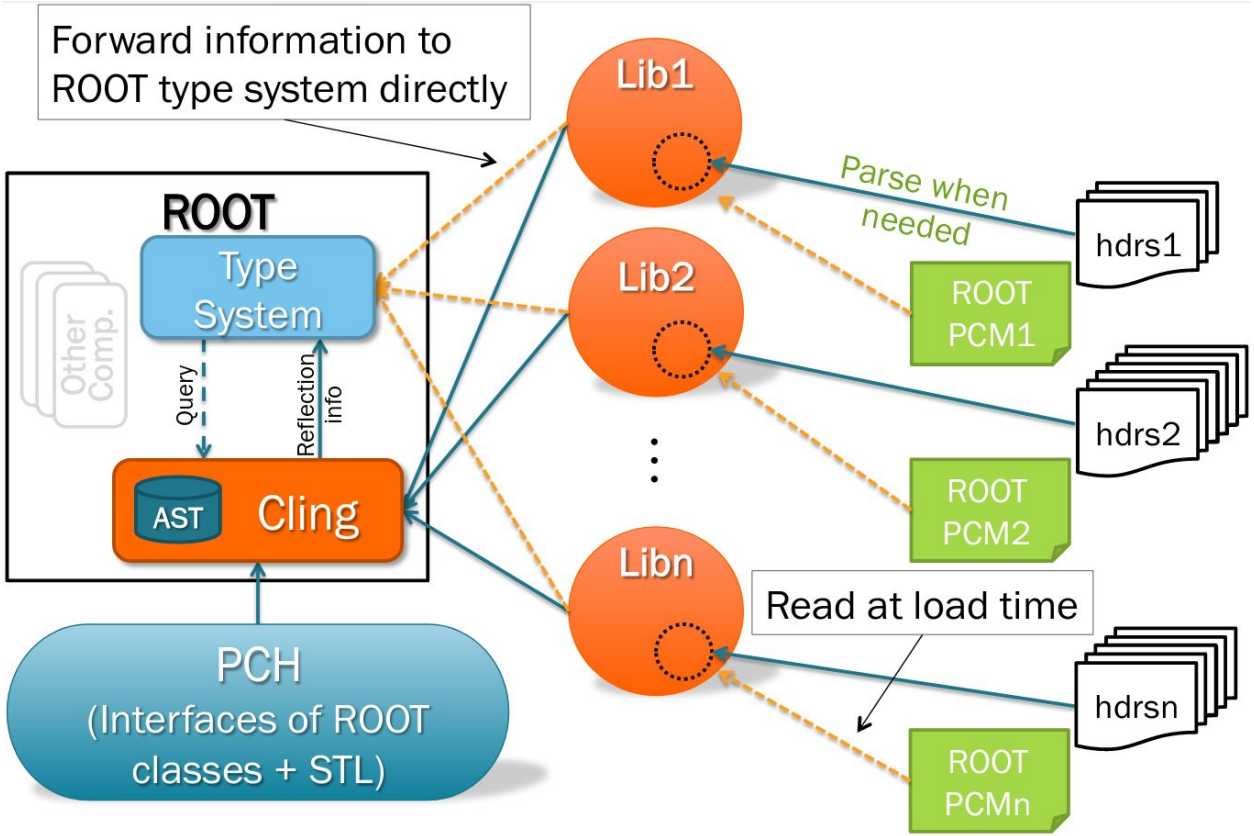
- Such radical changes happen rarely in core software

Improve to evolve our sw, e.g. with agile techniques

A Glimpse of the Complexity: Dictionaries

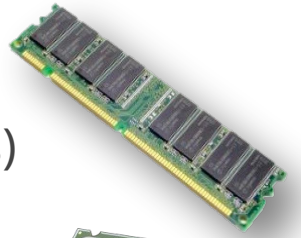


A Glimpse of the Complexity: Type System

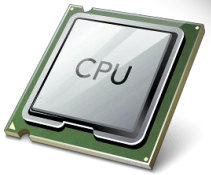


… Hard Work, Great Results

- First measurements of CMS workflows based on ROOT6 (Q1 2014)
  - +1 GB RSS extra :-(
  - Heavy runtime penalty (also associated to the many allocations)



Hard work: optimise, redesign, profile, communicate



- Recovered ROOT5 runtime performance and memory footprint
  - And even beat it: ROOT6 faster than ROOT5, using less memory (Q2 2015)!

Today, CMS, LHCb, Belle2, ATLAS, Neutrino Platrform and others use ROOT6 in production

Also due to experiments' flexibility and willingness to make it happen

… And to P. Hristov, for profiling, porting and giving feedback throughout the entire process!!

Ultimate Interactive Experience

```
root [0] std::map<int, std::string> m {{1,"one"}, {2,"two"}}
(std::map<int, std::string> &) { 1 => "one", 2 => "two" }
root [1] auto t = std::make_tuple<int, double, std::string>(1,42.,"hello");
root [2] t
(std::tuple<int, double, basic_string<char> > &) { 1, 42.000000, "hello" }
```

```
root [0] auto f = [](int i){return i*i;}
((lambda) &) @0x10d45a3c0
root [1] f("foo")
ROOT_prompt_1:1:1: error: no matching function for call to object of type '(lambda at ROOT_prompt_0:1:10)'
f("foo")
^
ROOT_prompt_0:1:10: note: candidate function not viable: no known conversion from 'const char [4]' to 'int' for 1st argument
auto f = [](int i){return i*i;}
         ^
ROOT_prompt_0:1:10: note: conversion candidate of type 'int (*)(int)'
```

Ultimate Interactive Experience

```
root [0] std::map<int, std::string> m {{1,"one"}, {2,"two"}}
(std::map<int, std::string> &) { 1 => "one", 2 => "two" }
```

```
root [0] .! cat a.h
#ifndef __MyClass__
#define __MyClass__
class MyClass {
public:
  MyClass() {std::cout << "Hello!\n";}
};
#endif
root [1] #include "a.h"
root [2] MyClass m;
Hello!
```

Interactivity, without dictionaries, with an interpreter powered by a real compiler

Changes in User Code

We did our best, but a few things had to be changed

- Headers can be parsed at runtime (minor issue in general)
  - Dictionaries keep information about the headers, *autoparsed* by ROOT (no action by the users)
- CINT was parsing code line by line: Cling is based on a compiler, it “sees all the code at once”
  - E.g. No runtime load of library in a macro to resolve symbols of the macro itself - “Too late” :-)

Best Practices to Cope with Change

- Do not load manually libraries (`gSystem->Load("mylib.so")`)
  - Let ROOT autoload them for you!
- Always generate a rootmap file together with a dictionary
  - Allows auto{loading,parsing}: resolve symbols automatically
  - See `-rml`, `-rmf` switches of rootcling (`--rootmap`, `--rootmap-lib` of genreflex)
- Take advantage of the ROOT_INCLUDE_PATH env variable
  - Analogous to LD_LIBRARY_PATH, allows ROOT to find headers
- If autoloading at runtime is not enough, force loading at parsing time
  - `R__LOAD_LIBRARY(mylib)`
  - E.g. libraries outside the library path…

Parallelism

Parallelism in ROOT

- Renovation of parallelism support started ~3y ago
  - Important results achieved
- Support of *explicit* and *implicit* parallelism
  - *Explicit*: the user expresses parallelism with building blocks provided (also by) ROOT
  - *Implicit*: ROOT parallelises expensive operations without requiring actions from the user (task based model)
  - Multi-thread and multi-process paradigms

Parallel Execution

- ROOT::TProcessExecutor and ROOT::TThreadExecutor
  - Same interface, ROOT::TExecutor
  - Inspired by Python *concurrent.features.Executor*
- Map, Reduce, MapReduce patterns provided

```
ROOT::TProcessExecutor pe(Nworkers);
auto myNewColl = pe.Map(myLambda, myColl);
```

The Runtime

- Multiprocessing runtime: ROOT provides its own utilities
- Threading runtime: adopted Threading Building Blocks (TBB)
  - Optional installation, but necessary for implicit parallelism
  - Task based parallelism
  - Build system fetches, builds and installs it if requested and not available
  - TBB interface not exposed directly
    - Ready to complement with other runtimes

https://www.threadingbuildingblocks.org

Implicit Parallelism

- ROOT parallelises common, expensive operations behind the scenes: implicit parallelism
  - TBB runtime exploited, not visible to the user
  - No explicit action requested from the user, except...

```
ROOT::EnableImplicitMT();
or
root -t
```

Also thread safety and parallelism would have been impossible with CINT...

CMS & ATLAS: ROOT implicit MT is ON by default

What is Implicitly Parallel?

- **TTree::GetEntry** reads multiple branches in parallel
- **TTree::FlushBaskets** writes baskets to disk in parallel
- **TTreeCacheUnzip** uncompresses baskets in parallel

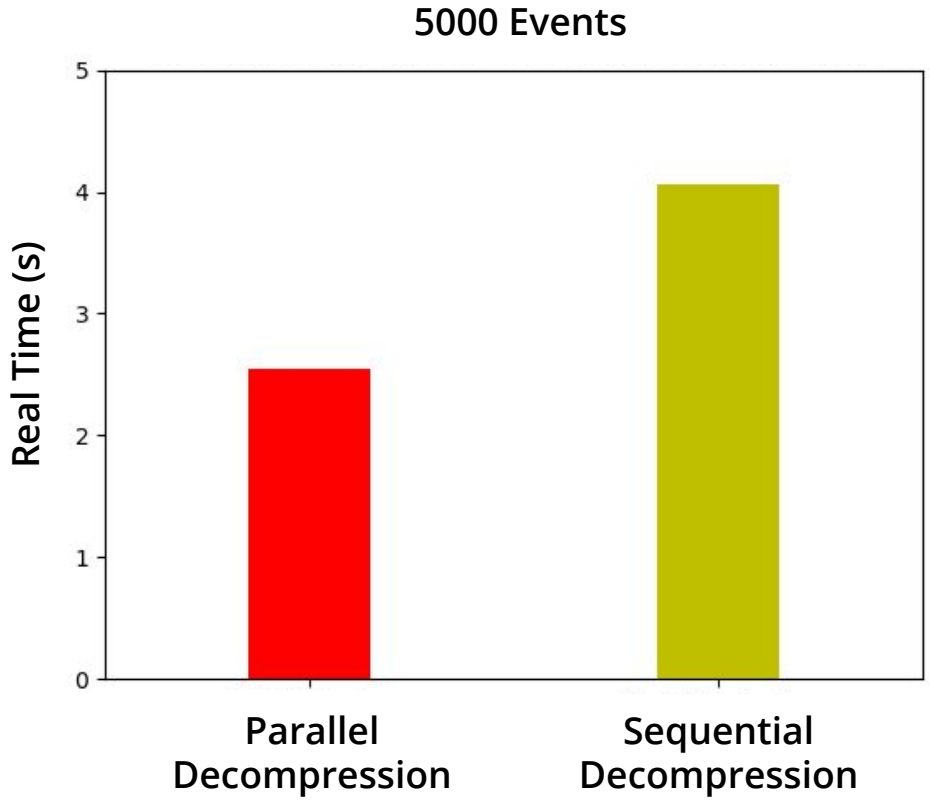
Speed up Reading

- **TH*::Fit** performs in parallel the evaluation of the objective function
- **TMVA::DNN** trains the deep neural network in parallel
- **TDataFrame** parallelises event loop over ranges of entries

hadd also provides a -j option to specify the number of workers!

Benchmark: Parallel Basked Decompression

- ROOT Event Data
- Fully split dataset
- Tested on an Intel® Core i5 3330 (6M cache, 3.00 GHz)



G. Amadio

Speed Up Writing: TBufferMerger

G. Amadio

Sequential usage of `TFile`

```
void Fill(TTree &tree, int init, int count)
{
   int n = 0;

   tree->Branch("n", &n, "n/I");

   for (int i = 0; i < count; ++i) {
      n = init + i;
      tree.Fill();
   }
}

int WriteTree(size_t nEntries)
{

   TFile f("myfile.root");
   TTree t("mytree","mytree");

   Fill(&t, 0, nEntries);

   t.Write();

   return 0;
}
```

Parallel usage of `TFile` with `TBufferMerger`

```
void Fill(TTree *t, int init, int count); // same as on the left

int WriteTree(size_t nEntries, size_t nWorkers)
{
   size_t nEntriesPerWorker = nEntries/nWorkers;

   ROOT::EnableThreadSafety();
   ROOT::Experimental::TBufferMerger merger("myfile.root");

   std::vector<std::thread> workers;

   auto workItem = [&](int i) {
         auto f = merger.GetFile();
         TTree t("mytree", "mytree");
         Fill(t, i * nEntriesPerWorker, nEntriesPerWorker);
         f->Write(); // Send remaining content over the wire
      };

   for (size_t i = 0; i < nWorkers; ++i)
      workers.emplace_back(workItem,i);

   for (auto&& worker : workers) worker.join();

   return 0;
}
```

Speed Up Writing: TBufferMerger

G. Amadio

Sequential usage of `TFile`

```
void Fill(TTree &tree, int init, int count)
{
   int n = 0;

   tree->Branch(

   for (int i =
      n = init +
      tree.Fill(
   }
}

int WriteTree(si
{

   TFile f("myfile.root");
   TTree t("mytree","mytree");

   Fill(&t, 0, nEntries);

   t.Write();

   return 0;
}
```

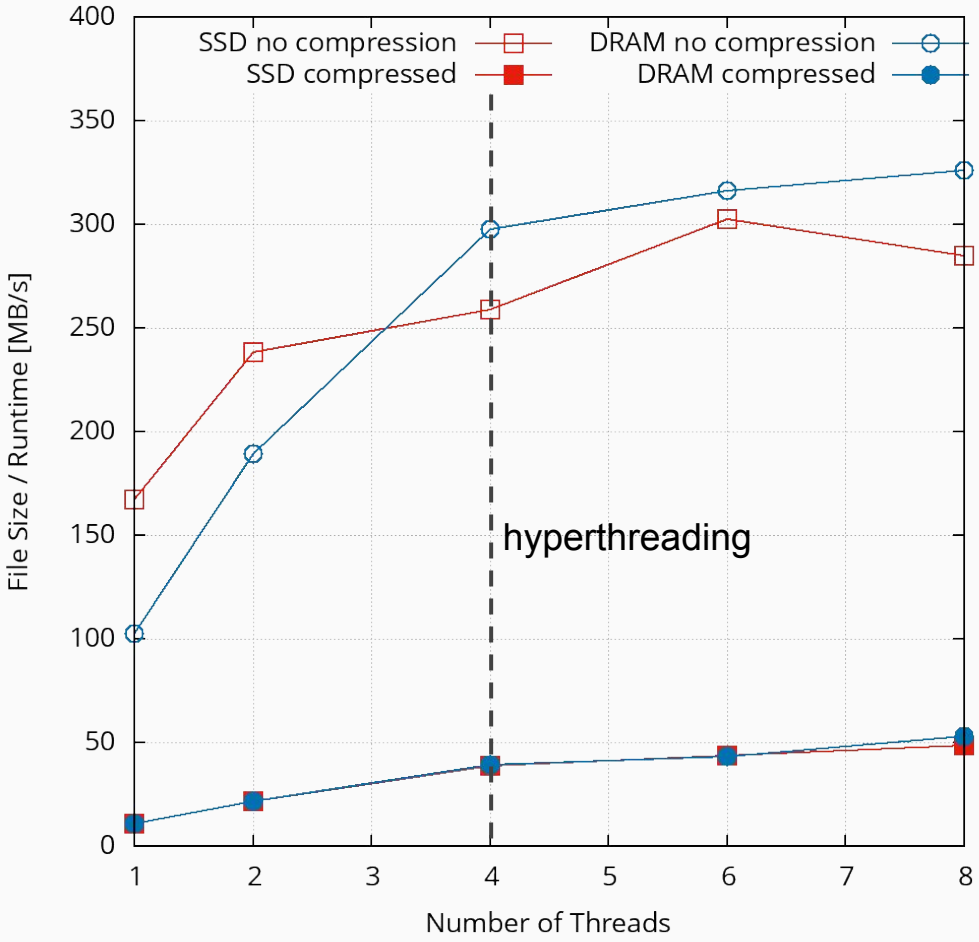
Parallel usage of `TFile` with `TBufferMerger`

```
void Fill(TTree *t, int init, int count); // same as on the left

                                          ers)

                                          kers;

                                          ("myfile.root");

                                          nEntriesPerWorker);
      f->Write(); // Send remaining content over the wire
   };

   for (size_t i = 0; i < nWorkers; ++i)
      workers.emplace_back(workItem,i);

   for (auto&& worker : workers) worker.join();

   return 0;
}
```

The effort expended on achieving high parallel processing rates is wasted unless accompanied by outstanding achievements in the programming model.

Benchmark: TBufferMerger with Random Data

- Fill a tree with one branch with random numbers
- Synthetic benchmark that exacerbates the role of I/O by doing only lighweight computations
- Create ~1GB of data and write out to different media (SSD and DRAM)
- Quad core laptop Intel® Core i7 4710HQ (2.5GHz, 6M cache)



Declarative Analysis

A rich collection of code examples:
https://root.cern/doc/master/group__tutorial__tdataframe.html

Filling a Tree with Objects

```
TRandom3 R;
using trivial4Vectors =
std::vector<std::vector<double>>;

TFile f("vectorCollection.root",
        "RECREATE");
TTree t("t","Pseudo particles");

trivial4Vectors  parts;
auto partsPtr = &parts;

t1.Branch("tracks", &partsPtr);
// pi+/pi- mass
constexpr double M = 0.13957;
```

```
for (int i = 0; i < 128; ++i) {
   auto nPart = R.Poisson(20);
   particles.clear(); parts.reserve(nPart);
   for (int j = 0; j < nPart; ++j) {
      auto pt = R.Exp(10);
      auto eta = R.Uniform(-3,3);
      auto phi = R.Uniform(0, 2*TMath::Pi() );
      parts.emplace_back({pt, eta, phi, M});
    }
    t.Fill();
  }
  t.Write();
}
```

Reading Objects from a TTree

what we write

```
TTreeReader reader(data);
TTreeReaderValue<A> x(reader,"x");
TTreeReaderValue<B> y(reader,"y");
TTreeReaderValue<C> z(reader,"z");
while (reader.Next()) {
   if (IsGoodEntry(*x, *y, *z))
      h->Fill(*x);
}
```

what we *mean*

- full control over the event loop
- requires some boilerplate
- users implement common tasks again and again
- parallelisation is not trivial

Can we do Better?

simple yet powerful way to analyse data with modern C++

---

provide high-level features, e.g.
less typing, better expressivity, abstraction of complex operations

---

allow transparent optimisations, e.g.
multi-thread parallelisation and caching

The Vision: CHEP 2016

Functional Chains R&D

- We are constantly looking for opportunities to apply implicit parallelism in ROOT
- "Functional Chains" R&D being carried out
  - Functional programming principles: no global states, no for/if/else/break
  - Analogy with tools like ReactiveX*, R dataframe, Spark
  - Gives room for optimising operations internally

Can this be a successful model for our physicists?

```
import ROOT
f = ROOT.TFile("aliDataset.root")
aliTree = f.Events
dataFrame = TDataFrame(aliTree)

dataFrame.filter(sel1).map(func2).cache().filter(sel3).histo('var1:var2').Draw('LEGO')
```

Express analysis as a chain of functional primitives.

https://indico.cern.ch/event/505613/contributions/2228338/

TDataFrame: declarative analyses

```
TDataFrame d(data);
auto h = d.Filter(IsGoodEntry, {"x","y","z"})
          .Histo1D("x");
```

- full control over *the analysis*
- no boilerplate
- common tasks are already implemented

? parallelization is not trivial?

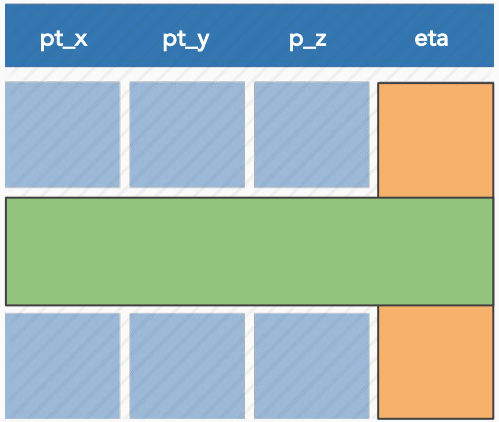
TDataFrame: declarative analyses

```
ROOT::EnableImplicitMT();
TDataFrame d(data);
auto h = d.Filter(IsGoodEntry, {"x","y","z"})
          .Histo1D("x");
```

- full control over *the analysis*
- no boilerplate
- common tasks are already implemented

How to Imagine TDataFrame

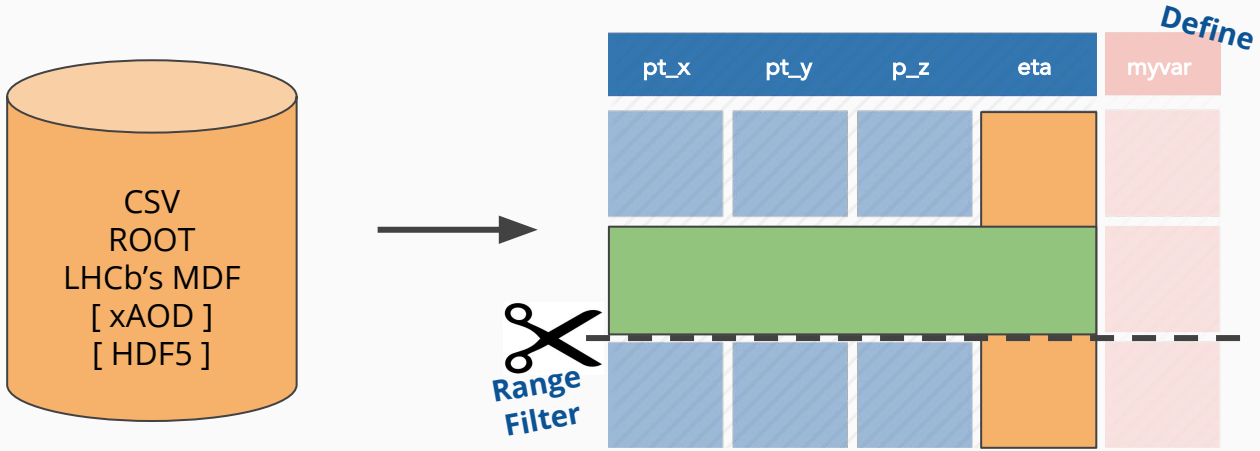
CSV
ROOT
LHCb's MDF
[xAOD]
[HDF5]



columns
or "branches"

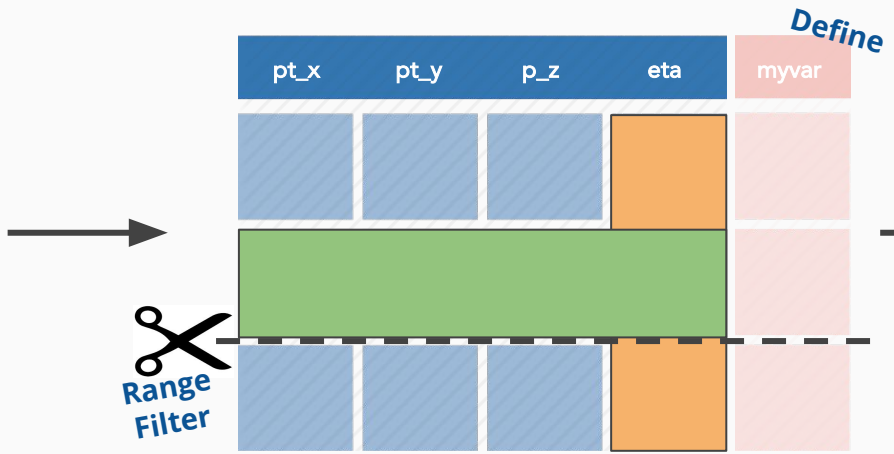
can contain any kind
of c++ object

How to Imagine TDataFrame



How to Imagine TDataFrame

CSV
ROOT
LHCb's MDF
[xAOD]
[HDF5]



- histograms, profiles
- new ROOT files
- cut-flow reports
- data aggregations (mean, sum, ...)
- custom side-effects

Creating a TDataFrame - 1 file

```
TDataFrame d1("treename", "file.root");

TDataFrame d2("treename", filePtr);

TDataFrame d3(myTree); // by reference!
```

Three ways to create a TDataFrame that reads tree "treename" from file "file.root"

Creating a TDataFrame - more files

```
TDataFrame d1("treename", "file*.root");
TDataFrame d2("treename", {"file1.root","file2.root"});

std::vector<std::string> files = {"file1.root","file2.root"};
TDataFrame d3("treename", files);

TChain chain("treename");
chain.Add("file1.root); chain.Add("file2.root);
TDataFrame d4(chain); // passed by reference, not pointer!
```

Here TDataFrame reads tree “treename” from files “file1.root” and “file2.root”

Not Only ROOT Datasets

- TDataSource: Plug *any columnar* format in TDataFrame
- Keep the programming model identical!
- ROOT provides CSV data source
- More to come
  - TDataSource is a programmable interface! E.g. https://github.com/bluehood/mdfds LHCb raw format
  - **TArrowDS (G. Eulisse)**: https://github.com/root-project/root/pull/1712
  - **TXAodDS (U. Dharmaji)**: interface to ATLAS analysis ntuples

ROOT analysis of non-ROOT data made easy

Not Only ROOT Datasets

```
// Let's first create a TDF that will read from the CSV file.
// The types of the columns will be automatically inferred.
auto fileName = "tdf014_CsvDataSource_MuRun2010B.csv";
auto tdf = ROOT::Experimental::TDF::MakeCsvDataFrame(fileName);

// Now we will apply a first filter based on two columns of the CSV,
// and we will define a new column that will contain the invariant mass.
// Note how the new invariant mass column is defined from several other
// columns that already existed in the CSV file.
auto filteredEvents =
   tdf.Filter("Q1 * Q2 == -1")
      .Define("m", "sqrt(pow(E1 + E2, 2) - (pow(px1 + px2, 2) + pow(py1 + py2, 2) +
    pow(pz1 + pz2, 2)))");
```

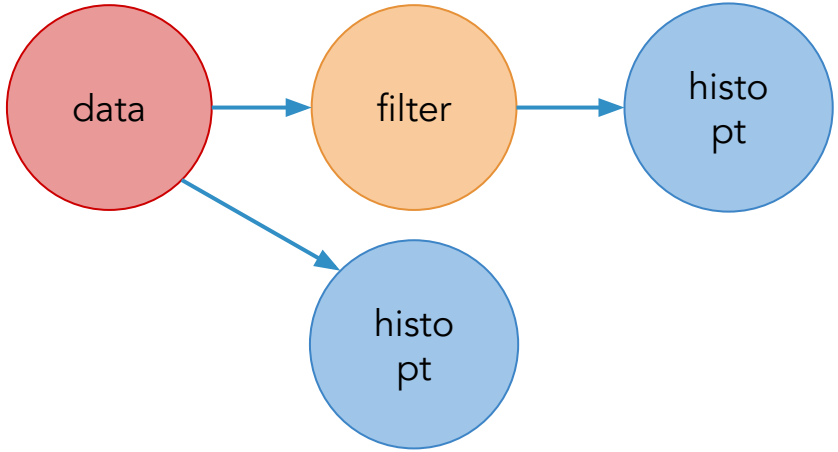
https://root.cern/doc/master/tdf014__CSVDataSource_8C.html

Cut on theta, fill histogram with pt

```
TDataFrame d("t", "f.root");
auto h = d.Filter("theta > 0").Histo1D("pt");
h->Draw(); // event loop is run here, when you access a result
           // for the first time

TFile f("f.root");
TTree *t; f.GetObject(t ,"t");
t.Draw("pt >> h", "theta > 0");
auto h = (TH1F*) gDirectory->Get("h");
h->Draw();
```

event-loop is run *lazily*, upon first access to the results

Think of your analysis as data-flow




```
auto h2 = d.Filter("theta > 0").Histo1D("pt");
auto h1 = d.Histo1D("pt");
```

Using callables instead of strings

```
// define a lambda - an inline function - that checks “x>0”
auto IsPos = [](double x) { return x > 0.; };
// pass it to the filter together with a list of branch names
auto h = d.Filter(IsPos, {"theta"}).Histo1D("pt");
h->Draw();
```

any callable (function, lambda, functor class) can be used as a filter, as long as it returns a boolean

Filling multiple histograms

```
auto h1 = d.Filter("theta > 0").Histo1D("pt");
auto h2 = d.Filter("theta < 0").Histo1D("pt");
h1->Draw();          // event loop is run once here
h2->Draw("SAME"); // no need to run loop again here
```

Book all your actions upfront. The first time a result is accessed, TDataFrame will fill all booked results.

Define a new column

```
double m = d.Filter("x > y")
            .Define("z", "sqrt(x*x + y*y)")
            .Mean("z");
```

`Define` takes the name of the new column and its expression. Later you can use the new column as if it was present in your data.

Programming Model: Jitting

- TDF is heavily templated: type safety! performance!
  - Easier programming model, types of columns can be inferred at runtime
  - Type safety ensured, but at runtime: thanks Cling!
  - Crucial with many template parameters (e.g. when writing out)
  - Jitting used also for filters and adding columns to the dataset

```
d.Histo1D<float>("myCol");
```

JIT

```
d.Histo1D("myCol");
```

```
d.Define("v1v2",
[](T& v1, T& v2){return v1*v2;},
{"v1","v2"});
```

JIT

```
d.Define("v1v2","v1*v2");
```

Ideal for PyROOT!

A string to replace a callable, no DSL but C++ (jitted!)

Cutflow reports

```
d.Filter("x > 0", "xcut")
 .Filter("y < 2", "ycut");
d.Report();
```

```
// output
xcut       : pass=49         all=100        --   49.000 %
ycut       : pass=22         all=49         --   44.898 %
```

When called on the main TDF object, `Report` prints statistics for all filters *with a name*

Running on a range of entries

```
// stop after 100 entries have been processed
auto hz = d.Range(100).Histo1D("x");

// skip the first 10 entries, then process one every two until the end
auto hz = d.Range(10, 0, 2).Histo1D("x");
```

Ranges are only available in single-thread executions.
They are useful for quick initial data explorations.

Saving data to file

```
auto new_df = df.Filter("x > 0")
                .Define("z", "sqrt(x*x + y*y)")
                .Snapshot("tree", "newfile.root");
```

We filter the data, add a new column, and then save everything to file. No boilerplate code at all.

Saving data to file

```
ROOT::EnableImplicitMT();
auto new_df = df.Filter("x > 0")
                .Define("z", "sqrt(x*x + y*y)")
                .Snapshot("tree", "newfile.root");
```

Writing datasets (in parallel) made easy!

We filte…en save everything to file. No boilerplate code at all.

Creating a new data-set

```
TDataFrame d(100);
auto new_d = d.Define("x", []() { return gRandom->Uniform(); })
              .Define("y", []() { return gRandom->Uniform(); })
              .Snapshot("tree", "newfile.root");
```

We create a special TDF with 100 (empty) entries, define some columns, save it to file

C++ / JIT / PyROOT

Pure C++

```
d.Filter([](double t) { return t > 0.; }, {"th”})
 .Snapshot<vector<float>>("t","f.root",{"pt_x"});
```

C++ and JIT-ing with CLING

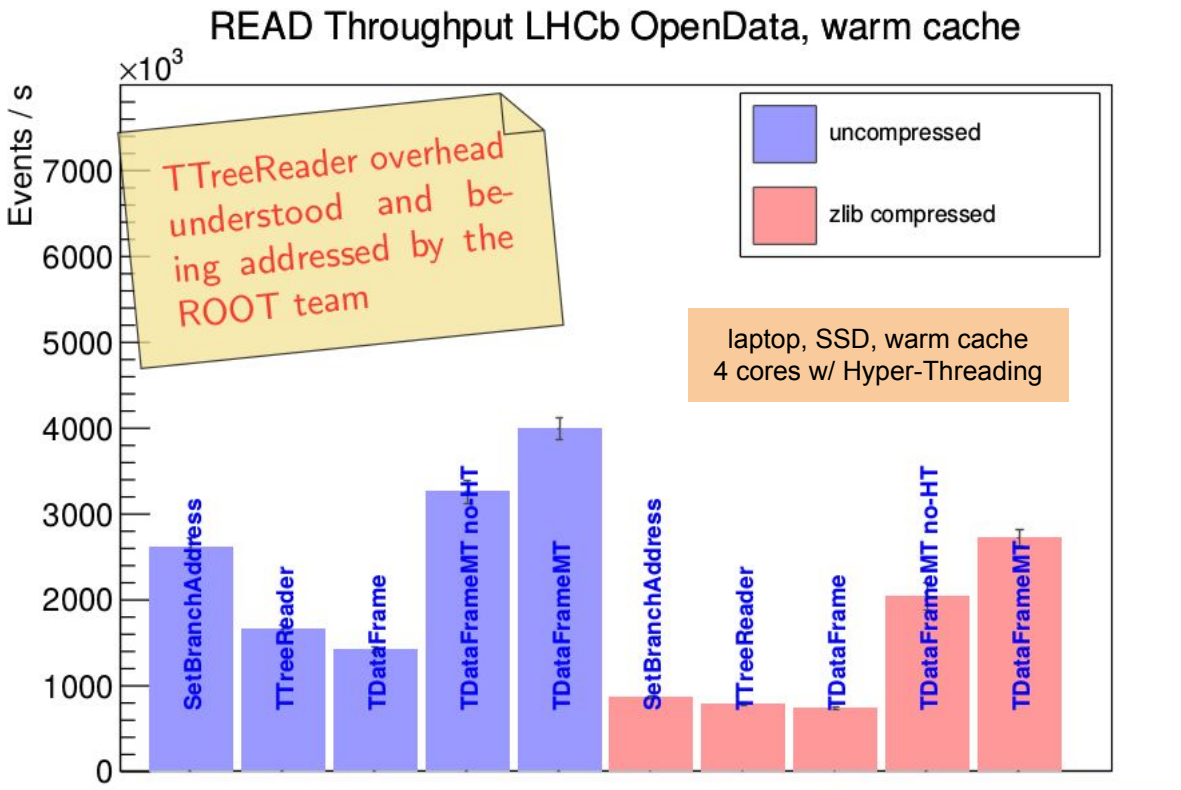
```
d.Filter("th > 0").Snapshot("t","f.root","pt_x");
```

---

pyROOT -- just leave out the ;

```
d.Filter("th > 0").Snapshot("t","f.root","pt_x")
```

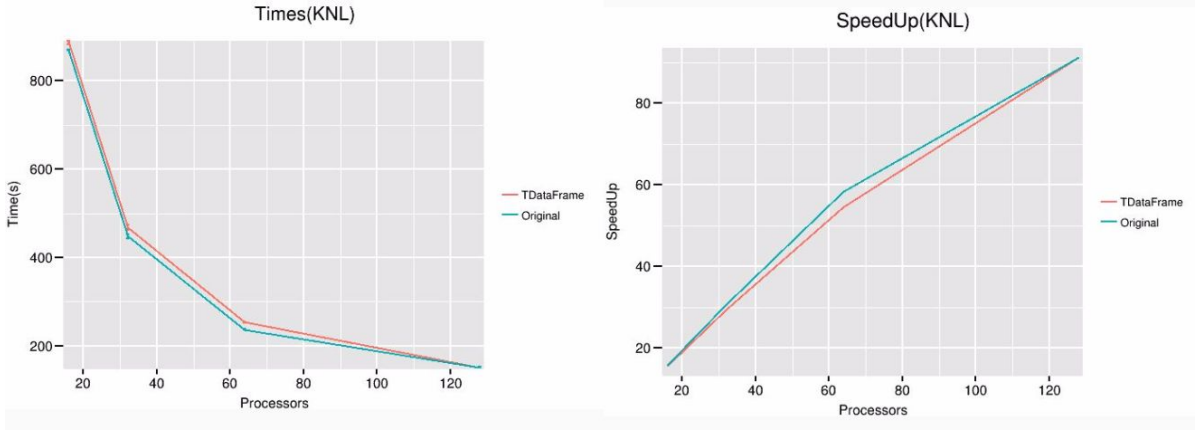
TDataFrame: performance



source: A quantitative review of data formats for HEP, Jakob Blomer, ACAT 2017

TDataFrame: does it scale?

TDF was benchmarked on a many-core KNL machine against the same multi-thread analysis written in ROOT5:
Monte Carlo QCD Low-Pt events generation + analysis on the fly



(n.b. the analysis generates data on-the-fly, does not perform I/O)

What About Windows?

ROOT6 on Windows

- For very long time, clang was not fully compatible with Visual Studio (ABI incompatibility).
  - Work-around (falling-back on cl.exe): inappropriate for our use-case!
- ROOT6 now builds and partially works on Windows (with VS 2017)
- Next step is to make ROOT passing most of the tests

Current Status (Private Branch)

57% of ROOT integration/functionality tests pass

```
57% tests passed, 263 tests failed out of 608

Label Time Summary:
cling          = 120.90 sec*proc (96 tests)
longtest       =  27.86 sec*proc (23 tests)
matrix         =   1.15 sec*proc (1 test)
regression     = 119.60 sec*proc (95 tests)
roottest       = 122.04 sec*proc (97 tests)

Total Test time (real) = 280.61 sec
```

ROOT 6.14:

Windows "Alpha Release"

Most ROOT tests pass

```
99% tests passed, 4 tests failed out of 403

Label Time Summary:
longtest     = 497.31 sec*proc (12 tests)
tutorial     = 2298.58 sec*proc (316 tests)

Total Test time (real) = 636.66 sec
```

Vectorisation

Vectorisation Already Available: Fitting

- Integration of VecCore in ROOT as the common vector abstraction in HEP
  - Definition of new ROOT SIMD types: ROOT::Double_v, ROOT::Float_v.
- Adaptation of TF1 to evaluate functions evaluating over vector types.
- Adaptation of the fitting classes and interfaces in ROOT to accept the new SIMD types and functions implementing them.
- Parallelization of the fitting objective functions (Max. Likelihood, Least Squares)

Vectorisation Already Available: Fitting

```
//Example Fit: Implementation of the vectorized function
ROOT::Double_v func(const ROOT::Double_v *data, const double *params)
{
   return params[0] * exp(-(*data + (-130.)) * (*data + (-130.)) / 2) +
          params[1] * exp(-(params[2] * (*data * (0.01)) - params[3] *
          ((*data) * (0.01)) * ((*data) * (0.01))));
}

// Enable implicit parallelization
ROOT::EnableImplicitMT();

//This code is totally backwards compatible
auto f = TF1("fvCore", func, 100, 200, 4);
f.SetParameters(1, 1000, 7.5, 1.5);
TH1D h1f("h1f", "Test random numbers", 12800, 100, 200);
h1f.FillRandom("fvCore", 1000000);
h1f.Fit(&f);
```

Future Support of Vectorisation in ROOT

- Math functions used frequently in fitting to be vectorised
  - Leverage VecCore library
  - Integration also with Vdt library of vectorised functions
- Build ROOT with runtime detection - enable “right” SIMD instruction set (“fat libraries”)
  - Challenge: binaries usable on many platforms, e.g. AVX2, SSE4, AVX512...
  - Solution: detect instruction set and enable the right one at runtime
  - Fitting, compression, TMVA, etc: take advantage of this feature
- Potentially, optimize some expensive operations internally
  - E.g. Matrices - vectorised implementation, no changes in the user interface

Python and Notebook Interfaces

An Even More Powerful PyROOT

- New interpreter: PyROOT “knows more”
  - Enhance user experience!
- Many new features added: “brace initialisation”, r-value reference support…
- Many coming: C++ lambdas, numpy interoperability

An Even More Powerful PyROOT

```
root [0] .! cat a.h
#ifndef __MyClass__
#define __MyClass__
class MyClass {
public:
  MyClass() {std::cout << "Hello!\n";}
};
#endif
root [1] #include "a.h"
root [2] MyClass
Hello!
```

A unique feature of ROOT

```
>>> import ROOT
>>> ROOT.gInterpreter.ProcessLine('#include "a.h"')
0L
>>> a = ROOT.MyClass()
Hello!
>>> v = ROOT.vector('MyClass')()
```

Interlude: Notebooks

Simple ROOTbook (Python)

This simple ROOTbook shows how to create a histogram, fill it and draw it.

Let's start importing the ROOT module, which gives us access to all ROOT classes.

In [2]:
```
import ROOT
```

```
Welcome to JupyROOT 6.07/07
```

In order to activate the interactive visualisation we can use the JSROOT magic:

In [3]:
```
%jsroot on
```

Now we will create a histogram specifying its title and axes titles:

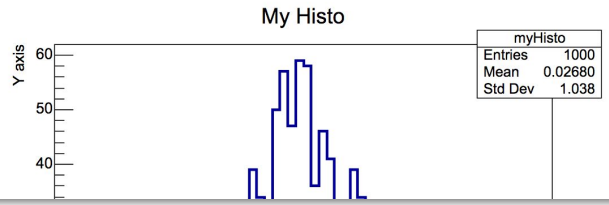
In [5]:
```
h = ROOT.TH1F("myHisto","My Histo;X axis;Y axis",64, -4, 4)
```

Time to create a random generator and fill our histogram:

In [6]:
```
rndmGenerator = ROOT.TRandom3()
for i in xrange(1000):
    rndm = rndmGenerator.Gaus()
    h.Fill(rndm)
```

We can now draw the histogram. We will at first create a canvas, the entity which in ROOT holds graphics primitives. Note that thanks to JSROOT, thi
plot but an interactive visualisation. Try to play with it and save it as image when you are satisfied!

In [7]:
```
c = ROOT.TCanvas()
h.Draw()
c.Draw()
```



Simple ROOTbook (C++)

This simple ROOTbook shows how to create a histogram, fill it and draw it. The language chosen is C++.

In order to activate the interactive visualisation we can use the JSROOT magic:

In [1]:
```
%jsroot on
```

Now we will create a histogram specifying its title and axes titles:

In [2]:
```
TH1F h("myHisto","My Histo;X axis;Y axis",64, -4, 4)
```

```
(TH1F &) Name: myHisto Title: My Histo NbinsX: 64
```

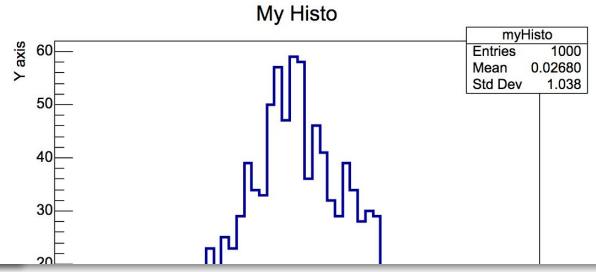
If you are wondering what this output represents, it is what we call a "printed value". The ROOT interpreter can indeed be instructed to "print" according to certain rules instances of a particular class.

Time to create a random generator and fill our histogram:

In [3]:
```
TRandom3 rndmGenerator;
for (auto i : ROOT::TSeqI(1000)){
    auto rndm = rndmGenerator.Gaus();
    h.Fill(rndm);
}
```

We can now draw the histogram. We will at first create a canvas, the entity which in ROOT holds graphics primitives.

In [4]:
```
TCanvas c;
h.Draw();
c.Draw();
```



Interlude: Notebooks

Simple ROOTbook (Python)

This simple ROOTbook shows how to create a histogram, fill it and draw it.

Let's start importing the ROOT module, which gives us access to all ROOT classes.

In [2]:
```
import ROOT
```

```
Welcome to JupyROOT 6.07/07
```

In order to activate the interactive visualisation w

In [3]:
```
%jsroot on
```

Now we will ...eate a histogram specifying its titl

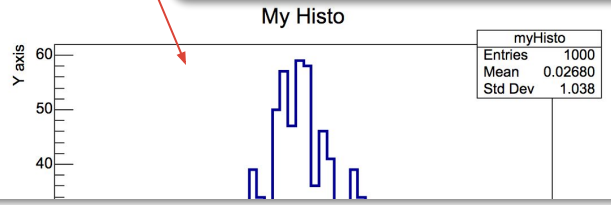
In [5]:
```
h = ROOT.TH1F("myHisto","My Histo;X axis;Y axis",64, -4, 4)
```

Time to create a random generator and fill our histogram:

In [6]:
```
rndmGenerator = ROOT.TRandom3()
for i in xrange(1000):
    rndm = rndmGenerator.Gaus()
    h.Fill(rndm)
```

We can now draw the histogram. We will at first create a canvas, the entity which in ROOT holds graphics primitives. Note that thanks to JSROOT, thi
plot but an interactive visualisati

In [7]:
```
c = ROOT.TCanvas()
h.Draw()
c.Draw()
```



Both C++ and Python interfaces of ROOT are fully integrated with Jupyter notebooks

This is an interactive JSROOT visualisation!

Simple ROOTbook (C++)

This simple ROOTbook shows how to create a histogram, fill it and draw it. The language chosen is C++.

In order to activate the interactive visualisation we can use the JSROOT magic:

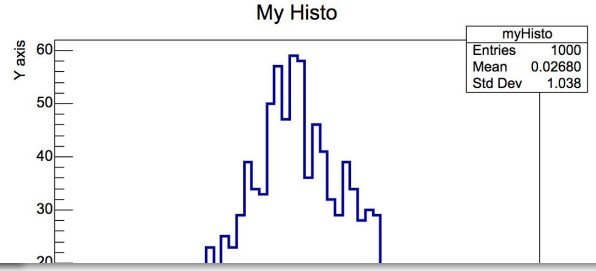
...OT interpreter can indeed be instructed to "print" according to certain

Time to create a random generator and fill our histogram:

In [3]:
```
TRandom3 rndmGenerator;
for (auto i : ROOT::TSeqI(1000)){
    auto rndm = rndmGenerator.Gaus();
    h.Fill(rndm);
}
```

We can now draw the histogram. We will at first create a canvas, the entity which in ROOT holds graphics primitives.

In [4]:
```
TCanvas c;
h.Draw();
c.Draw();
```



Compression

Columnar I/O in ROOT

- Much more than reading/writing data from/to disk
- Read / inflate / deserialize ↔ serialize / deflate / write out
  - Granularity imposed onto files (e.g., clusters of entries)
- Interactions with other parts of ROOT
  - Dynamic library loading
  - Bulk reading of data (TTreeCache)
  - Queries to the type system to determine how objects are represented
- Partial reads possible, remote reading, data dependencies
  - e.g. pointers, array sizes

Profiling ROOT I/O

- A delicate procedure: CPU, memory, storage - keep everything under control
- Sophisticated, multidimensional metrics
  - Size on disk, read speed, write speed, CPU usage, memory footprint
  - No approach is best in all dimensions
  - To be parallelised carefully: no task shall wait!
- Strongly depends on the usecase
  - Reading final ntuples? Writing reco data? What data model? What kind of storage device?

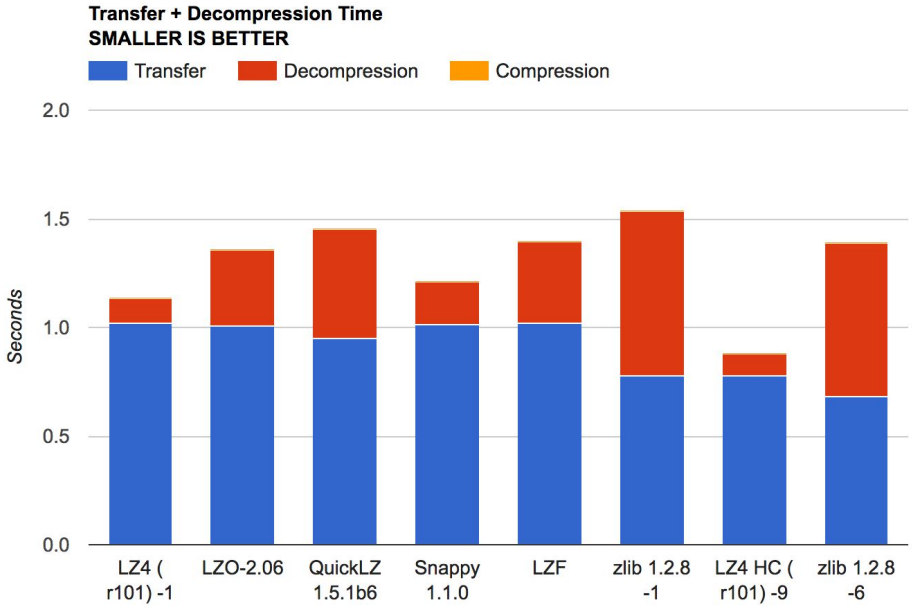
Compression algorithms/levels in ROOT

- Algorithm is accompanied by a *compression level*
- Algorithms available in ROOT:
  - Now: ZLIB (default, level 1), LZMA - since a while
  - **6.13: LZ4 - will be the new default (level 4)**
  - 6.1{3 or 4 or 5}: Cloudflare zlib
  - 2019: zstd
  - Others being tested and benchmarked: e.g. zlib-ng
- Can be set in the TFile ctor: `TFile(`*`name, option, title,`* **`compress`**`)`
- Obtain **compression setting code** with function
  `ROOT::CompressionSettings(`*`algorithm, compr level`*`)`

Priorities *can* be shuffled depending on needs of experiments

```
root [0] ROOT::CompressionSettings(ROOT::ECompressionAlgorithm::kZLIB,  6)
(int) 106
```

Why LZ4?



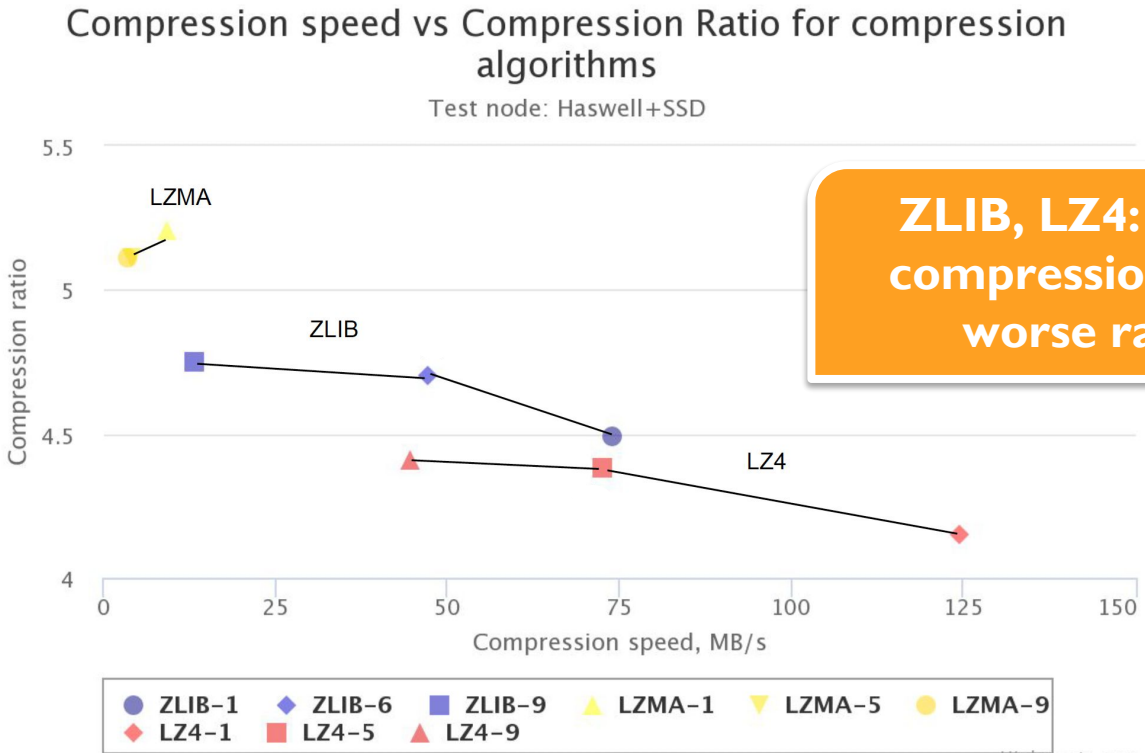
Source: http://www.lz4.org

LZ4 VS Zlib

- Faster decompression
- … At the price of worse compression ratio / runtime

Compromise is an advantage for the analysis usecase

Write Speed: Benchmarks

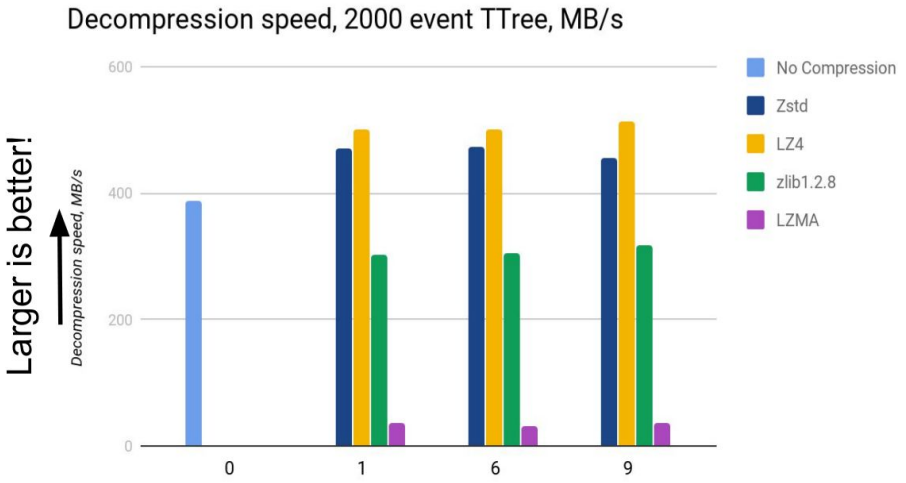


ZLIB, LZ4: Same compression rate, worse ratio

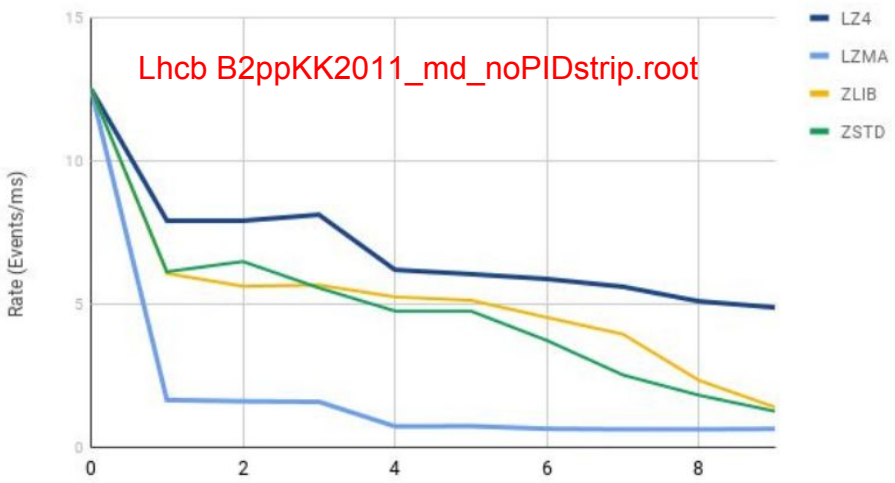
Synthetic benchmark

Oksana Shadura

Read Speed: Benchmarks



Synthetic benchmark **Oksana Shadura**



Open Data LHCb **Oksana Shadura**

- LZ4: fastest reading, bigger size on disk
- LZMA: smallest files, CPU investment
- Datamodel plays crucial role

Evolution of ROOT

EP R&D Program

ROOT contributions:

- “Filesystem-less high performance I/O of HEP Data”
- “High throughput data analysis on future heterogeneous platforms“
- “Future Distributed Analysis of HEP Data”

Don’t miss the workshop this Friday!

Modernising ROOT

Modernising ROOT not only with TDataFrame, Cling, parallelisation or PyROOT - We are working on:

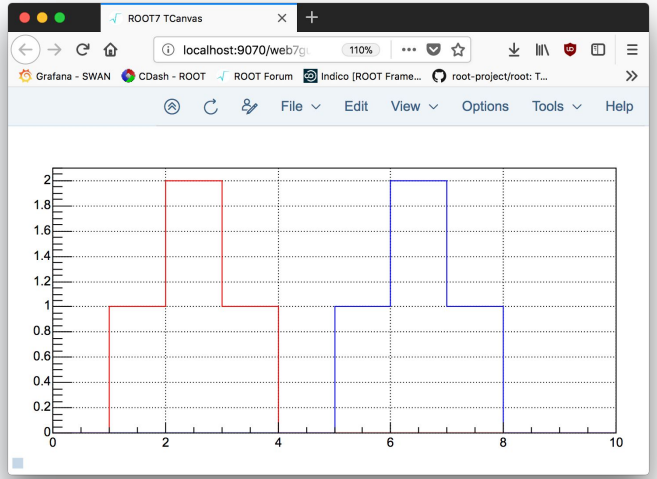
- A new histogramming package
- A new, JS based graphic infrastructure
- A new TFile and TTree

Codename “v7 interfaces”, meaning “Your New ROOT”

Can be tried now: cmake -Dcxx14=ON [...]

Modernising ROOT: an Example

```
using namespace ROOT;

// Create the histogram.
Experimental::TAxisConfig xaxis(10, 0., 10.);
auto pHist = std::make_shared<Experimental::TH1D>(xaxis);
auto pHist2 = std::make_shared<Experimental::TH1D>(xaxis);

// Fill a few points.
pHist->Fill(1);
pHist->Fill(2);
pHist->Fill(2);
pHist->Fill(3);

pHist2->Fill(5);
pHist2->Fill(6);
pHist2->Fill(6);
pHist2->Fill(7);

// Create a canvas to be displayed.
auto canvas = Experimental::TCanvas::Create("Canvas Title");
canvas->Draw(pHist)->SetLineColor(Experimental::TColor::kRed);
canvas->Draw(pHist2)->SetLineColor(Experimental::TColor::kBlue);

canvas->Show();
```



Stay tuned, more to come!!

https://root.cern/doc/master/group__tutorial__v7.html

Wrap Up

Conclusions

- **ROOT6 is in production since years**
  - **Gratifying** user experience: prompt, macros, PyROOT, Notebooks
  - **Performance** for centralised experiments workflows e.g. at the LHC
- **Support for parallelism**: IO, analysis, math, user driven
- **New declarative, parallel analysis approach: TDataFrame**
  - Read columnar data from different formats
  - Easy dataset modification/writing/caching
  - Powerful tool to get scientific results, faster
- **ROOT continues to evolve**
  - Modernisation of interfaces: TTree, TFile, Graphics …
  - R&D lines

Backup

More on histograms #1

```
auto h = d.Histo1D("x","w");
```

TDF can produce *weighted* TH1D, TH2D and TH3D.
Just pass the extra column name.

More on histograms #2

```
auto h = d.Histo1D({"h","h",10,0.,1.},"x", "w");
```

You can specify a model histogram with a set axis range, a name and a title (optional for TH1D, mandatory for TH2D and TH3D)

Running on a range of entries #2

```
// ranges can be concatenated with other transformations
auto c = d.Filter("x > 0")
          .Range(100)
          .Count();
```

This `Range` will process the first 100 entries
that pass the filter

Caching

```
TDataFrame d("mytree", "myFile.root");
auto cached_d = d.Cache();
```

All the content of the TDF is now in (contiguous) memory.
Analysis as fast as it can be (vectorisation possible too).

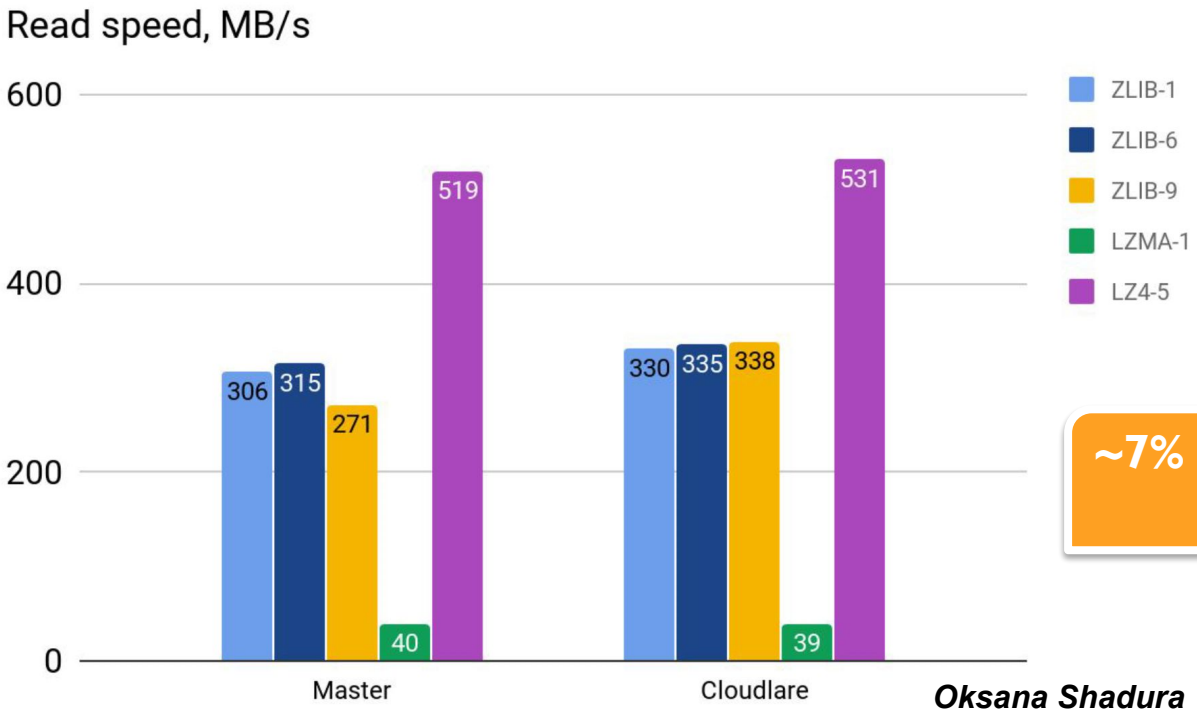
N.B. It is always possible to selectively cache columns to save some memory!

Filling histograms with arrays

```
auto h = d.Histo1D("pt_array", "x_array");
```

If `pt_array` and `x_array` are an array or an STL container (e.g. std::vector), TDF fills histograms with all of their elements. `pt_array` and `x_array` are required to have equal size for each event.

Cloudflare ZLIB



~7% improvement wrt zlib

Oksana Shadura

A PR already exists: https://github.com/root-project/root/pull/1527