TDataFrame: a declarative, parallel interface for ROOT's data analyses

Enrico Guiraud for the ROOT Team

DIANA/HEP, 11 December 2017

https://root.cern

Who needs new analysis interfaces?

ROOT's mission is to get physicists from collision to publication quickly and correctly

→ strive for a simple programming model

→ allow to effectively write efficient code

→ allow to easily express parallelism

Improving on current interfaces

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*

Improving on current interfaces

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

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
- ✔ easy to parallelize event-loop over entries

TDataFrame: design goals

simple and powerful programming model

TDataFrame: design goals

simple and powerful programming model

---

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

---

TDataFrame: design goals

simple and powerful programming model

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

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

TDataFrame: design goals

simple and powerful programming model

---

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

---

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

Available since ROOT v6.10, many new features added in v6.12

DOI 10.5281/zenodo.260230

the user guide can be found at root.cern.ch/doc/master

TDataFrame: an overview

Analyses as computation graphs

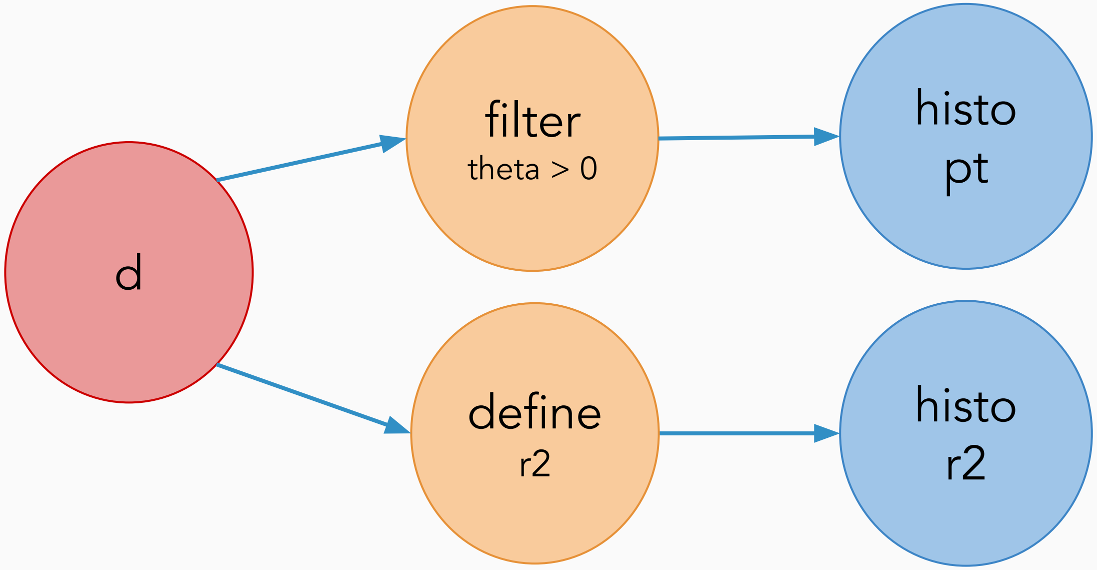
```
TDataFrame d("tree","file.root");
auto h2 = d.Filter("theta > 0").Histo1D("pt");
auto h1 = d.Define("r2","x*x + y*y").Histo1D("r2");
```

TDF: analyses as computation graphs

```
TDataFrame d("tree","file.root");
auto h2 = d.Filter("theta > 0").Histo1D("pt");
auto h1 = d.Define("r2","x*x + y*y").Histo1D("r2");
```

transform the data: filters, definition of new columns, ...

leaf nodes produce a result: histograms, profiles, sums, counts, ...

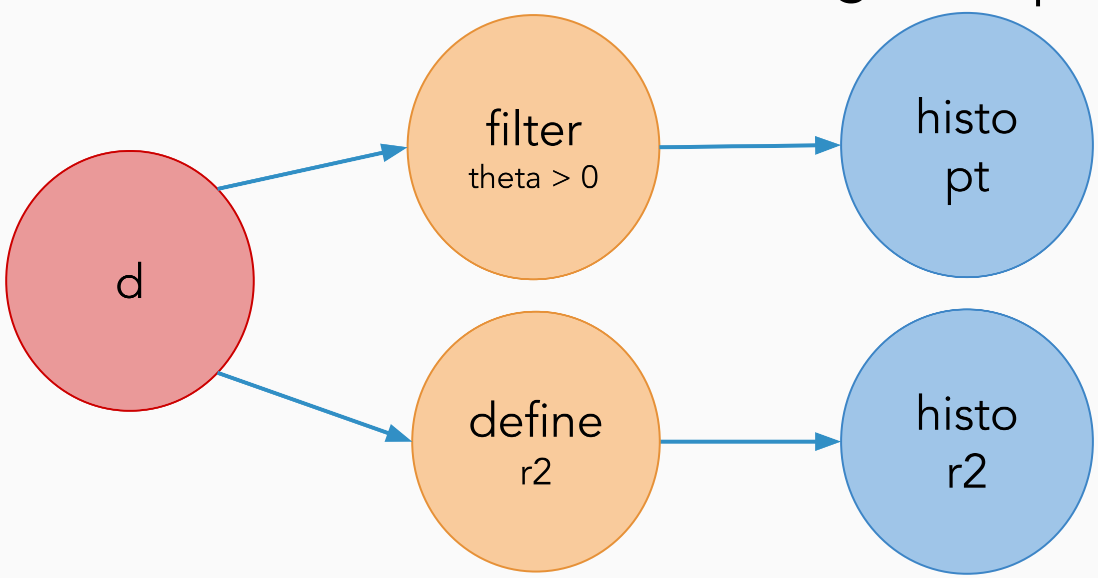


TDF: analyses as computation graphs

```
TDataFrame d("tree","file.root");
auto h2 = d.Filter("theta > 0").Histo1D("pt");
auto h1 = d.Define("r2","x*x + y*y").Histo1D("r2");
```

transform the data: filters, definition of new columns, ...

leaf nodes produce a result: histograms, profiles, sums, counts, ...



Graph is evaluated lazily, upon first access to a result

One evaluation of the graph corresponds to one loop over the data. It fills all pending results.

C++ -> JIT -> pyROOT

Pure C++

```
d.Filter([](double t) { return t > 0.; }, {"th"})
```

C++ -> JIT -> pyROOT

Pure C++

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

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*");
```

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*");
```

---

pyROOT

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

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*");
```

pyROOT -- just leave out the ;

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

Transformations and actions

Transformations

return a new graph node

- Define
- DefineSlot
- DefineSlotEntry
- Filter
- Range

Actions

return a result proxy

- Count
- Min
- Max
- Mean
- Sum
- Histo{1,2,3}D
- Profile{1,2}D
- Fill
- Reduce
- Foreach
- Take
- Snapshot
- Accumulate
- Graph
- StdDev

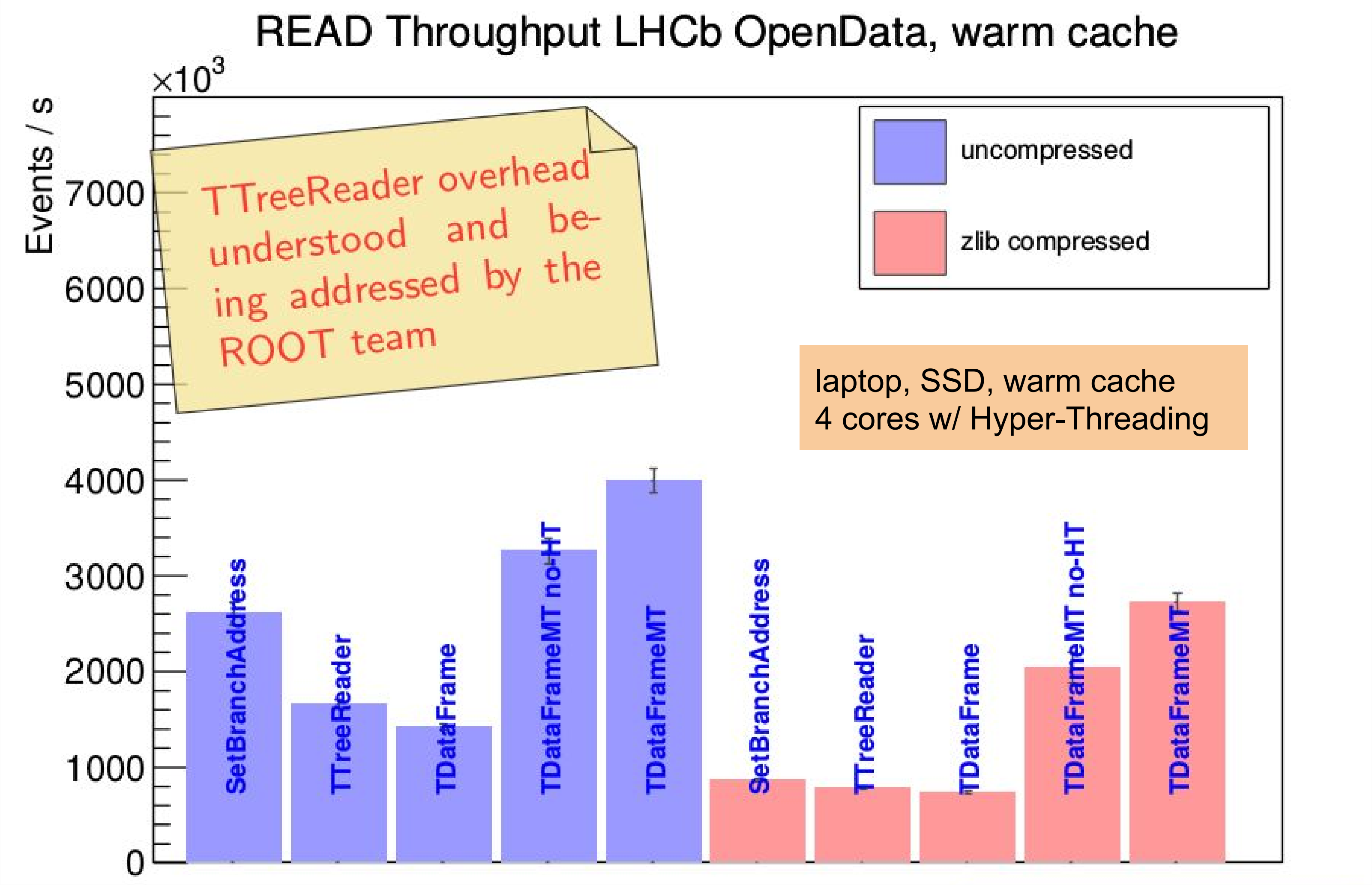
Putting everything together

Producing a skimmed, thinned TTree
and a histogram
in the same event loop
running on a CSV file
with multiple threads

```
ROOT::EnableImplicitMT();
auto tdf = MakeCsvDataFrame("data.csv");
auto zHist = tdf.Histo1D("z");
tdf.Snapshot("outT", "out.root", {"x","y"});
```

Performance and scaling

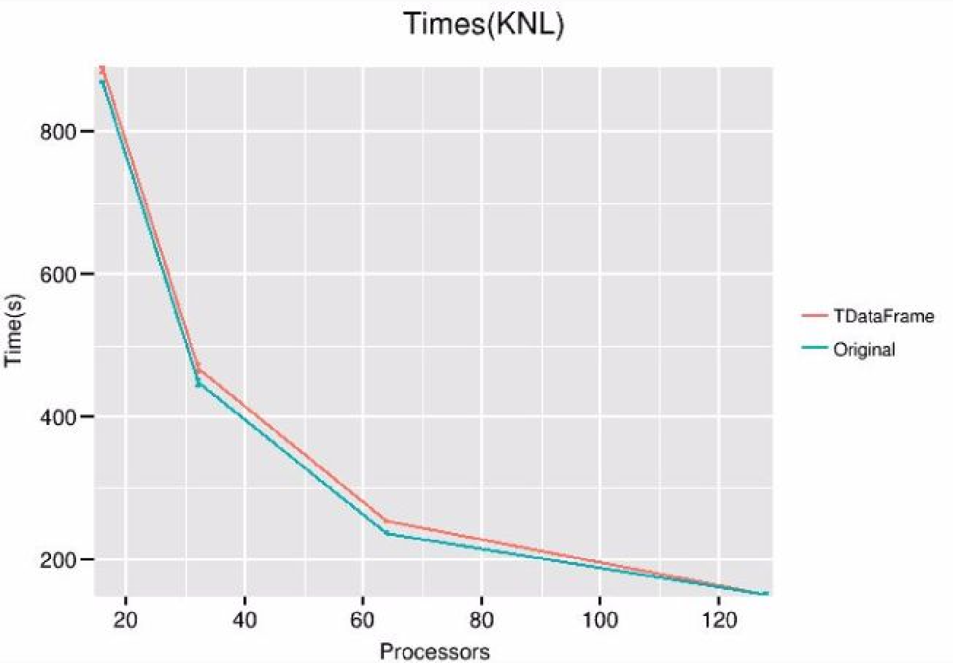
TDataFrame: performance

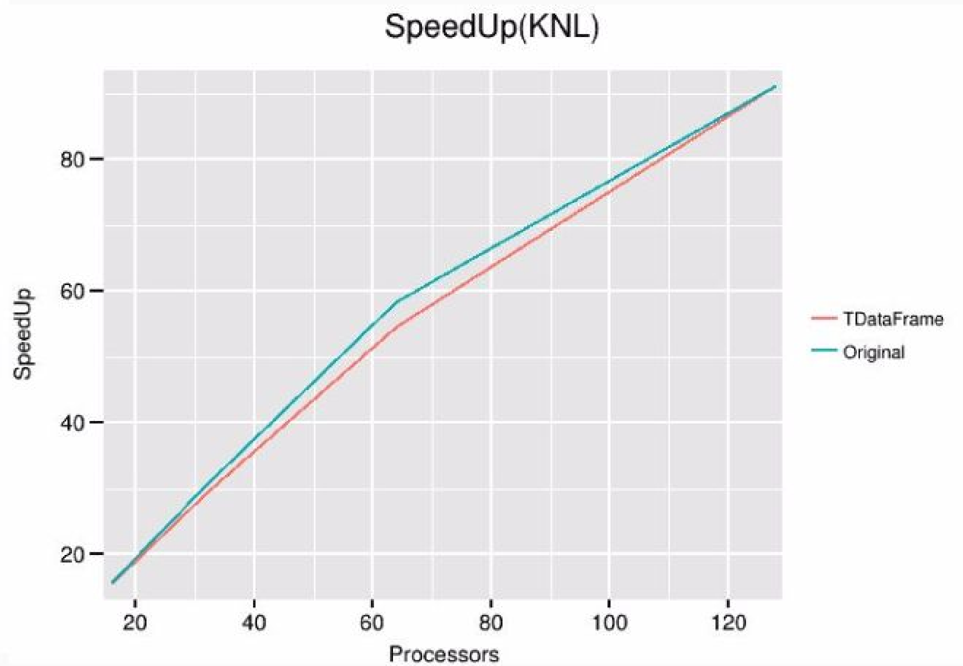


source: [A quantitative review of data formats for HEP](), Jakob Blomer, ACAT 2017

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)

source: Xavier Valls Pla, ROOT team

A few more features

TDataSource: a format adaptor for TDF

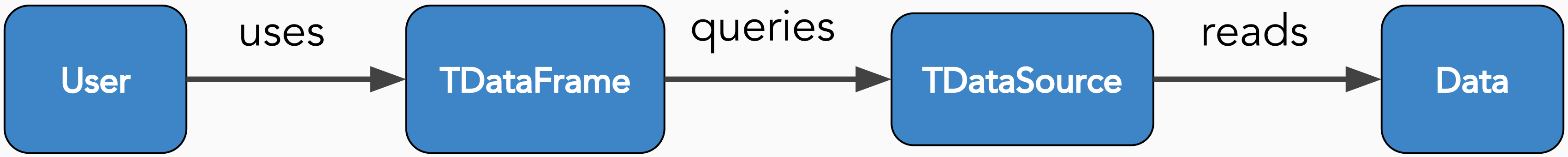
➔ TDataFrame can read data through TDataSource objects

TDataSource: a format adaptor for TDF

➔ TDataFrame can read data through TDataSource objects

➔ third-parties can implement and seamlessly integrate specific TDataSources for their format of choice

TDataSource: a format adaptor for TDF



➔ TDataFrame can read data through TDataSource objects

➔ third-parties can implement and seamlessly integrate specific TDataSources for their format of choice

➔ we currently support CSV through this mechanism:

```
auto tdf = MakeCsvDataFrame("data.csv"); // use as usual
```

➔ proof-of-concept implementations for ROOT and LHCb's binary MDF format

Event-loop callbacks

Users can register callbacks to be executed every N entries, in one thread or in all threads

Callbacks act on analysis results, e.g. a partially-filled histogram

```
auto h = tdf.Histo1D("x");
TCanvas c;
auto drawH = [&c](TH1D &h_) {
   c.cd();
   h_.Draw();
   c.Update();
};
// register callback
h.OnPartialResult(100, drawH);
```

Creating datasets with TDataFrame

```
ROOT::EnableImplicitMT();
TDataFrame d(100);
auto d2 = d.Define("x", []() { return rand(); })
           .Define("y", [](double x) { return x + noise(); }, {"x"})
           .Snapshot("tree", "newfile.root");
```

➔ this creates a TDF with 100 (empty) entries, defines some columns, saves them to file -- in parallel

➔ easiest way to create a new TTree

➔ proof of concept: TDF has been used to write events generated by Pythia8 to a TTree, in parallel

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 %
```

- ➔ calling `Report` on the head node:
  prints statistics for all filters *with a name*
- ➔ calling `Report` on other nodes,
  prints statistics for all *upstream* filters with a name

Summary

ROOT provides a modern, declarative, type-safe, parallelised interface for data analysis: TDataFrame

Summary

ROOT provides a modern, declarative, type-safe, parallelised interface for data analysis: TDataFrame

TDataFrame is <u>performant</u>, <u>scales</u> to many-core architectures, and aims to offer all HEP physicists need for their analyses

Summary

ROOT provides a modern, declarative, type-safe, parallelised interface for data analysis: TDataFrame

TDataFrame is performant, scales to many-core architectures, and aims to offer all HEP physicists need for their analyses

Use it in ROOT macros, compiled code, or in a notebook

Summary

ROOT provides a modern, declarative, type-safe, parallelised interface for data analysis: TDataFrame

TDataFrame is performant, scales to many-core architectures, and aims to offer all HEP physicists need for their analyses

Use it in ROOT macros, compiled code, or in a notebook

Future plans

- ➔ distributed execution
- ➔ more syntactic sugar for common operations on arrays
- ➔ a *fast path* for reading files containing simple data structures (integrating bulk I/O?)
- ➔ low-level performance optimization (analysis @100 cores)

EOF

More details on Jakob's data-set

"Fruit Fly" Data Set: The LHCb OpenData Sample

Starting point: "What if I had my data set in format X?"

Example Analysis

- 8.5 million LHC run 1 events $B \to KKK$ Link
- Flat *n*-tuple, 26 branches, mostly floating point numbers
- 21 branches needed for the toy analysis
- 2.4 million events can be skipped because one of the kaon candidates is flagged as a muon
- Toy analysis: sum over all branches from non-cut Kaons

```
struct BDecay {
  double h1_px;
  double h2_px;
  double h3_px;
  double h1_py;
  ...
};
```

On the simple end of the spectrum, helps to understand performance base case

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

TDataFrame's nuke bomb: Foreach

```
ROOT::EnableImplicitMT();
auto tdf = TDataFrame("tree","f*.root");
tdf.Filter(IsGood, {"x"})
   .Foreach(DoStuff, {"y","z"});
```

`Foreach` provides complete freedom of implementation while TDataFrame still provides transparent parallelization