ROOT
Data Analysis Framework

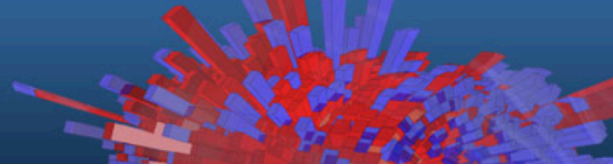
Declarative Parallel Analysis in ROOT: TDataFrame

D. Piparo For the ROOT Team

CERN EP-SFT

DOI 10.5281/zenodo.260230

Introduction



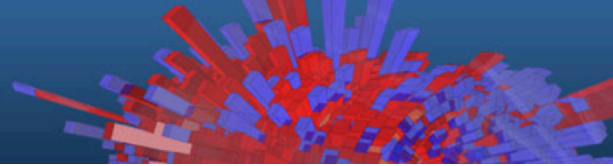
Novel way to interact with ROOT columnar format

- Inspired by tools such as **Pandas or Spark**
- Analysis expressed as a **chain of transformations and actions**
  - Transformation: filter, add a column, …
  - Actions: Fill an histo, a profile, count events, …

The **user specifies the *What* and ROOT chooses the *How***

- Streamline expression of analysis leaves room for **internal optimisation**
  - **Parallelisation**, partitioning, caching, reordering
- **Computation is only triggered at the end** of the chain having the full knowledge of what the user wants to do

Preview



```
ROOT::EnableImplicitMT();
DataFrame d("tree", "myfile_*.root");
// Filter, add a column and make a wighted histo.
// All parallelised.
auto h = d.Filter([](double a) {return a > 0.;}, {"x"})
          .Define("z", "x + y")
          .Histo1D("x", "z");
h->Draw();
```

Credit Recognised where Due

DOI 10.5281/zenodo.260230

- Developed on top of ROOT as an independent package
- Now integrated in ROOT
- Part of the portfolio of a few developers…
- … Available to everybody!

We engaged with Experiments…

Example feedback:

I think that the major issue is that **most people around here do not code in terms of pipelines, even if they think like that.** […]
As what regards any possible pipeline solution, I agree that in order to be able to profit from things like caching and automatic parallelisation, this sort of paradigm is needed.
My advice is therefore to make sure to present it together with the under-the-hood advantages it has. Otherwise it risks being dismissed as syntactic sugar.

We engaged with Experiments...

And concrete prototyping already ongoing!

TDataFrameTests

Code used in developing/testing the xAOD::TEvent <-> TDataFrame integration.

Star 1 | Fork 0 | KRB5 | https://:@gitlab.cern.ch:8443 | Global

Files (369 KB) Commits (21) Branches (3) Tags (0) Readme

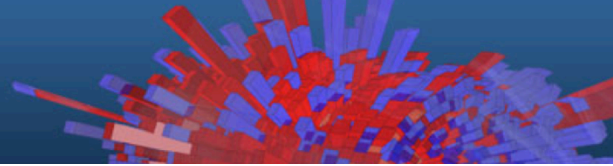
750cdd93 Some small updates to the README file · about 3 hours ago by Attila Krasznahorkay

… And Discussed Ideas …

- Parallelism, Performance and Programming Model Meeting
- Thursdays at 16:00
- No laptop, results-oriented meeting to discuss:
  - Ideas for future developments and evolution of present ROOT code in different areas
  - Planning for their implementation
  - Share encountered issues with others
  - Find the best programming model
  - Comment performance and scaling results

Mailing list: root-ppp@cern.ch

Is this ROOT7?



- Yes, absolutely!
- Well, not only
  - Available in ROOT 6

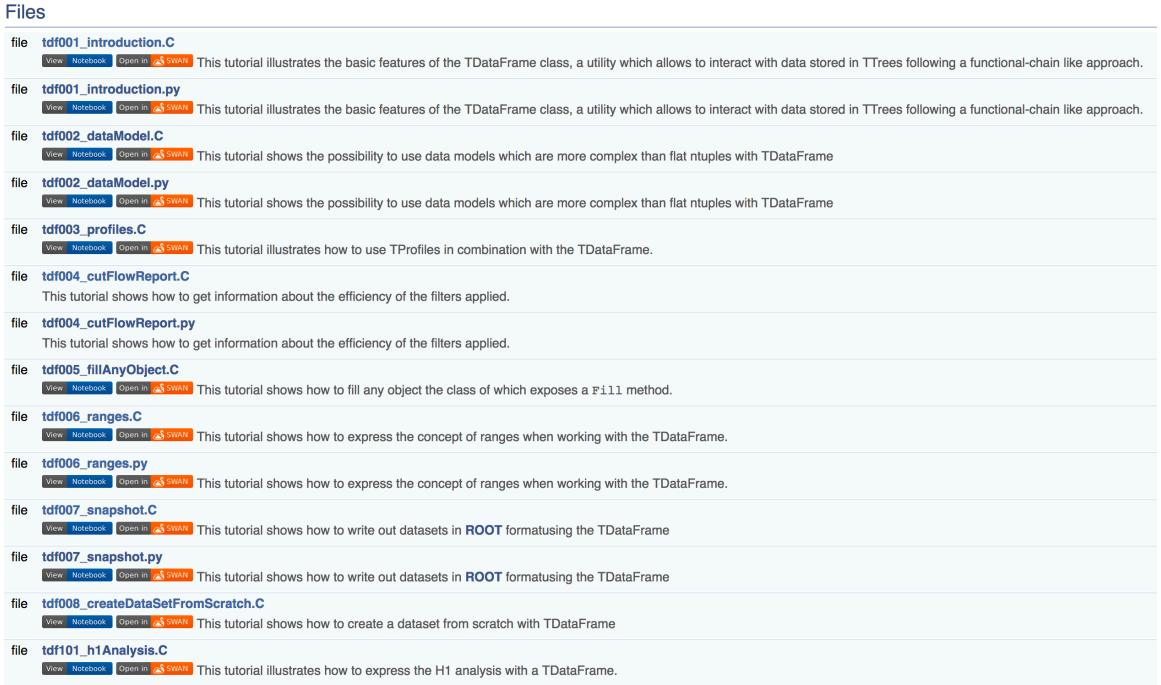
→Modern interface, plugs perfectly with other new interfaces

  - Constructed to do so, will be present in ROOT7 too
- Located in the *ROOT::Experimental* namespace
  - Minimal changes of the interface still possible

… And Prepared Doc and Examples

- TDF Manual
  https://root.cern.ch/doc/master/classROOT_1_1Experimental_1_1TDataFrame.html
- TDF Tutorials

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



Files

file **tdf001_introduction.C**
View Notebook Open in SWAN This tutorial illustrates the basic features of the TDataFrame class, a utility which allows to interact with data stored in TTrees following a functional-chain like approach.

file **tdf001_introduction.py**
View Notebook Open in SWAN This tutorial illustrates the basic features of the TDataFrame class, a utility which allows to interact with data stored in TTrees following a functional-chain like approach.

file **tdf002_dataModel.C**
View Notebook Open in SWAN This tutorial shows the possibility to use data models which are more complex than flat ntuples with TDataFrame

file **tdf002_dataModel.py**
View Notebook Open in SWAN This tutorial shows the possibility to use data models which are more complex than flat ntuples with TDataFrame

file **tdf003_profiles.C**
View Notebook Open in SWAN This tutorial illustrates how to use TProfiles in combination with the TDataFrame.

file **tdf004_cutFlowReport.C**
This tutorial shows how to get information about the efficiency of the filters applied.

file **tdf004_cutFlowReport.py**
This tutorial shows how to get information about the efficiency of the filters applied.

file **tdf005_fillAnyObject.C**
View Notebook Open in SWAN This tutorial shows how to fill any object the class of which exposes a `Fill` method.

file **tdf006_ranges.C**
View Notebook Open in SWAN This tutorial shows how to express the concept of ranges when working with the TDataFrame.

file **tdf006_ranges.py**
View Notebook Open in SWAN This tutorial shows how to express the concept of ranges when working with the TDataFrame.

file **tdf007_snapshot.C**
View Notebook Open in SWAN This tutorial shows how to write out datasets in **ROOT** formatusing the TDataFrame

file **tdf007_snapshot.py**
View Notebook Open in SWAN This tutorial shows how to write out datasets in **ROOT** formatusing the TDataFrame

file **tdf008_createDataSetFromScratch.C**
View Notebook Open in SWAN This tutorial shows how to create a dataset from scratch with TDataFrame

file **tdf101_h1Analysis.C**
View Notebook Open in SWAN This tutorial illustrates how to express the H1 analysis with a TDataFrame.

Giving up Control over the Loop

Controlling the loop implies:

- Boilerplate code
- Non-trivial parallelisation (e.g. what resources do I need to protect?)
- Operations re-implemented again and again

```
TTreeReader data(tree);
TTreeReaderValue<A> x(data, "x");
TTreeReaderValue<B> x(data, "y");
TTreeReaderValue<C> x(data, "z");

while(data.Next()) {
   if (IsGoodEvent(*x,*y,*z)) {
      DoStuff(*x,*y,*z);
   }
}
```

```
TDataFrame(tree, {"x", "y", "z"});
data.Filter(IsGoodEvent)
    .Foreach(DoStuff);
```

Giving up Control over the Loop

Controlling the loop implies:

- Boilerplate code
- Non-trivial parallelisation (e.g. what resources do I need to protect?)
- Operations re-implemented again and

```
TTreeReader data(tree);
TTreeReaderValue<A> x(data, "x");
TTreeReaderValue<B> x(data, "y");
TTreeReaderValue<C> x(data, "z");

while(data.Next()) {
   if (IsGoodEvent(*x,*y,*z)) {
      DoStuff(*x,*y,*z);
   }
}
```

Added value: 1 line to make it parallel
Not only syntactic sugar

```
ROOT::EnableImplicitMT();
TDataFrame(tree, {"x", "y", "z"});
data.Filter(IsGoodEvent)
    .Foreach(DoStuff);
```

Matches perfectly the overall modernisation and parallelisation plan of ROOT!

Intel TBB

- ROOT chose Intel TBB to express task parallelism
  - TDataFrame also exploits TBB as a consequence
- No TBB interface exposed to the user
  - Could complement TBB with other runtimes if needed

See “Expressing Parallelism in ROOT” at CHEP16
https://indico.cern.ch/event/505613/contributions/2228338

Cut and Fill Example - 1

```
auto isPos = [](double x){ return x > 0.; }
TDataFrame d("tree", "myfile_*.root");
auto h = d.Filter(isPos, {"theta"}).Histo1D<double>("pt");
h->Draw(); // event loop lazily triggered
```

- A dataset on many files containing 2 branches: *theta* and *pt*.
- Event-loop is triggered lazily, upon first access to the results (“Result pointer” returned by actions)
- Filter expressed as lambda (could be any callable but also a string JITted)

Cut and Fill Example - 2

```
auto isPos = [](double x){ return x > 0.; }
TDataFrame d("tree", "myfile_*.root");
auto h = d.Filter(isPos, {"theta"}).Histo1D<double>("pt");
h->Draw(); // event loop lazily triggered
```

```
TDataFrame d("tree", "myfile_*.root");
auto h = d.Filter("theta > 0.").Histo1D("pt");
h->Draw(); // event loop lazily triggered
```

Really trivial to read files!

- Programming model is key: every character required counts
- **Minimise effort to express what users want to do**
  - Still templated but use JIT to write the arguments
  - Simple filters and new columns can be formulated as strings
    - Language: C++, compiled just in time with cling

Fill Example: Collections

```
TDataFrame d("tree", "myfile_*.root");
auto h = d.Histo1D<std::vector<double>>("pts");
h->Draw(); // event loop lazily triggered
```

```
TDataFrame d("tree", "myfile_*.root");
auto h = d.Histo1D("pts");
h->Draw(); // event loop lazily triggered
```

- **Fill histograms with values stored in collections transparently**
- Example of "flattening"
- More to come, e.g. collection of muons, fill histogram with their Pt (obtained calling myMuon.Pt())

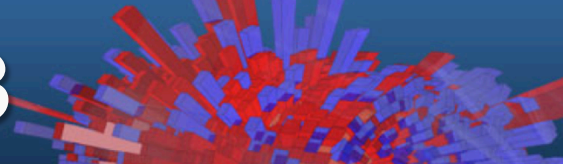
What about Python?!

```
TDataFrame d("tree", "myfile_*.root");
auto h = d.Filter("theta > 0.").Histo1D("pt");
h->Draw(); // event loop lazily triggered
```

```
d = TDataFrame("tree", "myfile_*.root")
h = d.Filter("theta > 0.").Histo1D("pt")
h.Draw() # event loop lazily triggered
```

- Most functionality already available
  - PyROOT evolved for 6.10 in order to allow that
- Items to iron out for 6.12 already identified (see ROOT-8798)
- Need to express filters, columns and Foreach arguments as Python functions and not only C++ callables and strins
  - Many options to achieve this, need to pick up / create the best possible

Cut and Fill Example - 3



```
TDataFrame d("tree", "myfile_*.root");
auto h1 = d.Filter("theta > 0.").Histo1D("pt");
auto h2 = d.Filter("theta < 0.").Histo1D("pt");
h1->Draw(); // Here the event loop takes place
h2->Draw("SAME"); // no need to re-run: histo already filled!
```

- One event loop for all actions!
  - Same price for 1,2,…, N histograms with a syntax which reminds TTree::Draw

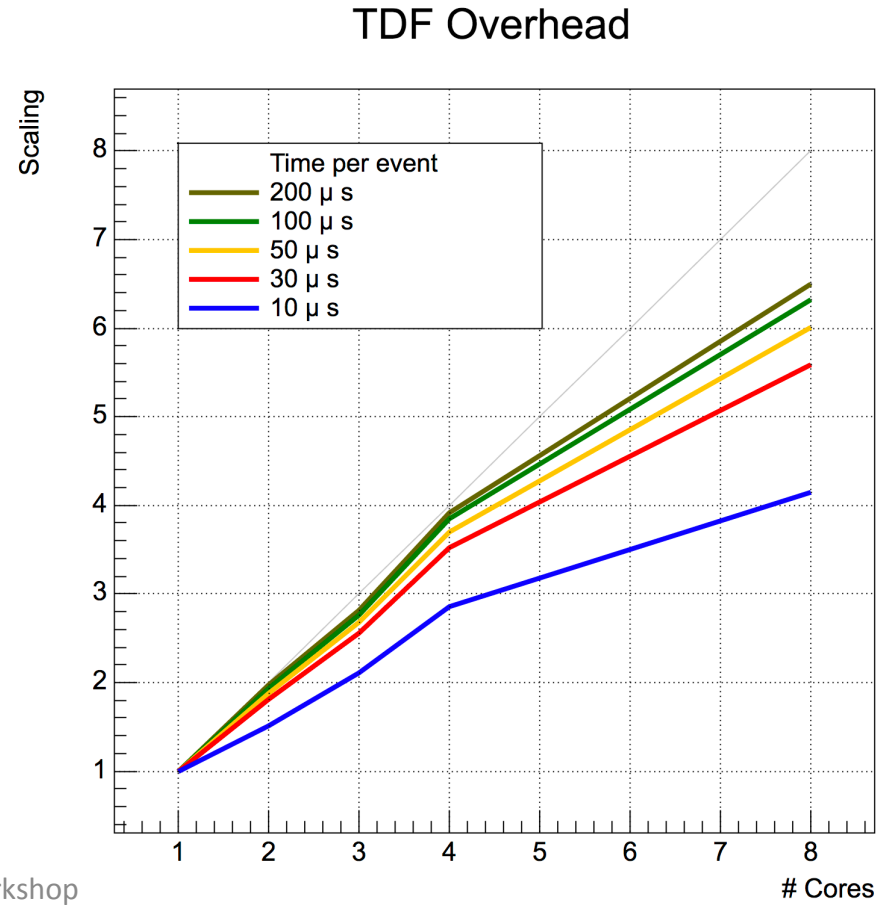
Implementation Interlude

Performance guarantee:

- Intensively rely on compiler optimisation
  - Heavily templated interface
  - Easily accessible programming model
- TTreeReader, analogous for user defined columns
  - Zero copies, cost of layer removed by the compiler
- Measured to be as fast as TTree::Draw
  - Able to manage more complexity, N histos per event loop, parallelism

Some Performance and Scaling

- No penalty wrt TTree::Draw
- 1 histo, 1 filter, 2 columns
  - **0.1s Ttree::Draw Vs 0.09s TDF**
- 3 histos, 1 filter, 2 columns:
  - **0.39s Ttree::Draw Vs 0.13s TDF**
- Scaling: excellent
  - ~ 2 ms / event needed to mask TDF overhead



A Functional Graph

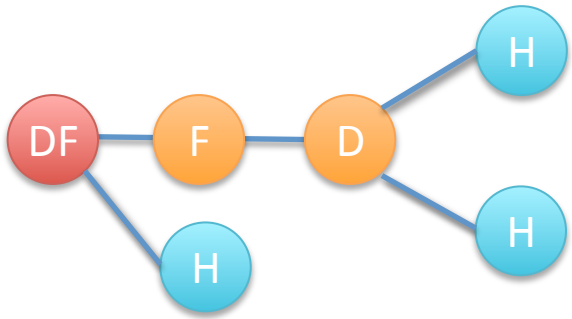
All types' names looked up, templated call jitted

```
TDataFrame d("tree", "myfile_*.root", {"x"});
// Filter and enrich the dataset
auto f = d.Filter([](double a) {return a > 0.;})
          .Define("z", "x + y");
// Hang multiple histos to it
auto hx = f.Histo1D("x");
auto hxy = f.Histo2D("x", "y");
auto pxy = f.Profile1D("x", "y");
```

Specify default column

Create new column!

- Not a "linear" chain but a veritable "graph"
- Can be extended at will, nodes saved in distinct variables



What about Writing?

```
// Start from an empty data frame
DataFrame d(1000);

// Generate MC events
auto genEvent = [](){
    thread_local unique_ptr<Pythia8::Pythia> pythia;
    if (!pythia) {
      pythia.reset(new Pythia8::Pythia)
      pythia->readString("HardQCD:all = on");
      pythia->readString("PhaseSpace:pTHatMin = 20.");
      pythia->readString("Beams:eCM = 14000.");
      pythia->init();
  }
  while (!pythia.next());
  return &pythia.event;
}

d.Define("event",genEvent);
d.Snapshot<Pythia8::Event*>("PythiaTree", "hardQCD.root");
```

Can start from an empty frame ("TDataFrame from Scratch")

Create a dataset with TDataFrame

What about Writing?

- You can write your filtered and enriched dataset with Snapshot
  - **Also in parallel: *ROOT::EnableImplicitMT()***
  - Easy access to MT writing of single tree (see Guilherme's talk)

```
DataFrame d(treeName, fileName);
auto d_cut = d.Filter("b1 % 2 == 0");
auto d2 = d_cut.Define("b1_square", "b1 * b1")
               .Define("b2_vector",
                       [](float b2) {
                          std::vector<float> v;
                          for (int i = 0; i < 3; i++) v.push_back(b2*i);
                          return v;
                       },
                       {"b2"});
auto d3 = d2.Snapshot(treeName, outFileName, {"b1", "b1_square", "b2_vector"});
// Can now re-start with the new TDF!
d3...
```

Save "Filter node"

Parallel snapshot: TBufferMerger and TBufferMergerFile classes used internally

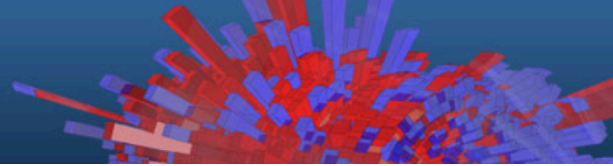
Done for Analysis - 1

```
TDataFrame d(treeName, fileName, {"b1", "b2"});
// An optional string parameter name can be passed
// to the Filter method to create a named filter.
auto filtered = d.Filter("0 == b2 % 2", "Cut2");
auto augmented = filtered.Define("b3", "b1 / b2");
auto cut3 = [](double x) { return x < .5; };
auto filtered2 = augmented.Filter("b3 < .5", "Cut3");
// Statistics are retrieved through a call to the Report method
std::cout << "Cut3 stats:" << std::endl;
filtered2.Report();
std::cout << "All stats:" << std::e
d.Report();
```

```
Cut3 stats:
Cut2      : pass=25         all=50         --    50.000 %
Cut3      : pass=23         all=25         --    92.000 %
All stats:
Cut1      : pass=24         all=50         --    48.000 %
Cut2      : pass=25         all=50         --    50.000 %
Cut3      : pass=23         all=25         --    92.000 %
```

Cutflow report: “what is the efficiency of my cuts?”

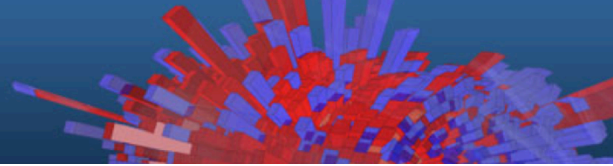
Done for Analysis - 2



```
TDataFrame d(treeName, fileName);
// ## Usage of ranges
// Now we'll count some entries using ranges
auto c_all = d.Count();
// This is how you can express a range of the first 30 entries
auto d_0_30 = d.Range(0, 30);
auto c_0_30 = d_0_30.Count();
// This is how you pick all entries from 15 onwards
auto d_15_end = d.Range(15, 0);
auto c_15_end = d_15_end.Count();
// We can use a stride too, in this case we pick an event every 3
auto d_15_end_3 = d.Range(15, 0, 3);
auto c_15_end_3 = d_15_end_3.Count();
```

Ranges: “I’d like to run to a subset of my data…”

Done for Analysis - 3



Actions supported in 6.10:

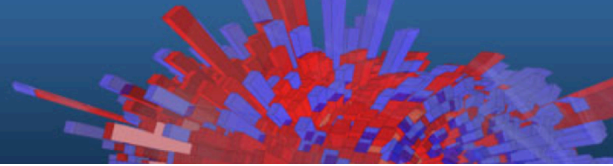
- **Count**: count events
- **Foreach**: apply function to each event
- **Foreachslot**: apply function to each event keeping track of the processing slot
- **Fill**: fill whatever object has a *Fill* method
- **Histo{1,2,3}D**: create histogram, also weighted
- **Max, Mean, Min**
- **Profile{1,2}D**: create profile
- **Reduce**
- **Take**: extract column content
- **Snapshot**

Transformations:

- **Filter, Define + Report, Range**

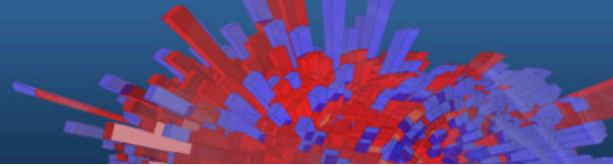
All can run in parallel
(except from range transformation, only exception)

Take Home Messages



- **Declarative analysis possible in ROOT with TDataFrame**
  - Perform analysis in terms of **transformations and actions**
  - Modern template metaprogramming
  - **Use JIT to smoothen programming model**
- **Implicit parallelism: same code runs sequential or parallel**
  - Just add "ROOT::EnableImplicitMT()" (as usual!)
  - Only exception: ranges
- IO Specific aspects:
  - Internally optimised: no copy, directly access values read
  - Can define new columns (again, no copy)
  - Can read in parallel
  - **Can write** manipulated dataset, **also in parallel**

Plans For 6.12



- Achieve full support in Python
- Add ways to “flatten” columns storing collections
  - Including invocation of methods of contained objects
- Improve scaling of MT Snapshot
- Allow to read data different from Trees
  - Data source, augment TTreeReader, act on TTrees: we’ll pick the best option
  - CSV, SQL, NumPy arrays the first candidate formats