Making your analysis more efficient with ROOT

Axel Naumann and Stefan Wunsch for the ROOT team

ROOT
Data Analysis Framework
https://root.cern

This Tutorial

- **ROOT new functionalities to get you to your results faster**
  - An incomplete selection, a sort of “Survival Kit”
- **Focus mainly on the treatment of datasets**
  - Mention also other areas, e.g. graphics
- **Discuss functionalities along two lines:**
  - **Parallelism and performance**
  - **Programming model**

Start with a presentation, then dive into a hands-on session (based on SWAN - use ROOT on the web)

Reference ROOT Release 6.16/00

Important Preliminary Step

If you don't have a **CERNBox**, the CERN "DropBox-like" service, connect now to

cernbox.cern.ch

This is needed to carry out the hands-on on SWAN

Talk and work with us!

- **Mattermost: https://mattermost.web.cern.ch/root**
- Have a question about ROOT? https://root-forum.cern.ch
- Have an idea about evolving ROOT? https://root-forum.cern.ch/c/my-root-app-and-ideas
- Have a bug to report? https://root.cern/guidelines-submitting-bug
- Have some code ready to go in the next ROOT release? https://github.com/root-project/root/pulls
  - *Github pull requests are always welcome*: simple (and not so simple) bug fixes, typos, missing documentation, tutorials...

New and Hot Features for Data Analysis

ROOT is now available on conda!

Given a working conda installation (one-line instructions here):

Install ROOT and its dependencies:

▶️ `conda create --name my-root-env --channel conda-forge python=3 root`

Activate the environment with:

▶️ `conda activate my-root-env`

Deactivate with:

▶️ `conda deactivate`

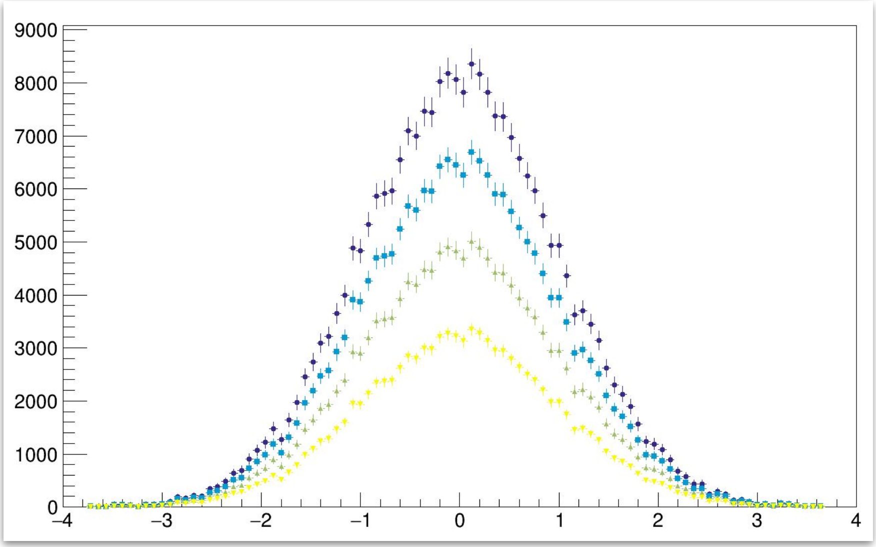
`root` and `root-*` commands work out of the box, as well as PyROOT.
To compile your C++ source code, use `$(root-config --cxx)` as the compiler.

Currently available on Linux, Mac support underway. Please report any problems you might encounter.

Automatic Colouring of Primitives

- Just draw, **ROOT picks an adequate set of colours for you**
- Accessible via a draw option (`myHisto.Draw("XXX")`)
  - **PLC**: Automatic line colour
  - **PMC**: Automatic marker colour
  - **PFC**: Automatic fill colour

```
h1->Draw("PLC PMC");
h2->Draw("SAME PLC PMC");
h3->Draw("SAME PLC PMC");
h4->Draw("SAME PLC PMC");
h5->Draw("SAME PLC PMC");
```



- **Automatic legend placement**, too:
  - `TPad::BuildLegend()`
  - E.g. `mycanvas.BuildLegend()`

Inspect ROOT Files

`TBrowser`, but also command line:

- `rootbrowse`: open a ROOT file and a `TBrowser`
- **`rootls`**: list file content, tree branches, objects’ stats
- **`rootcp`**: copy objects within a file or between files
- `rootdrawtree`: simple analyses, from command line!
- **`rooteventselector`**: select branches, events, compression algorithms and extract slimmer trees
- `rootmkdir`: creates a directory in a `TFile`
- `rootmv`: move objects between files
- `rootprint`: print objects in plots on files
- `rootrm`: remove objects from files

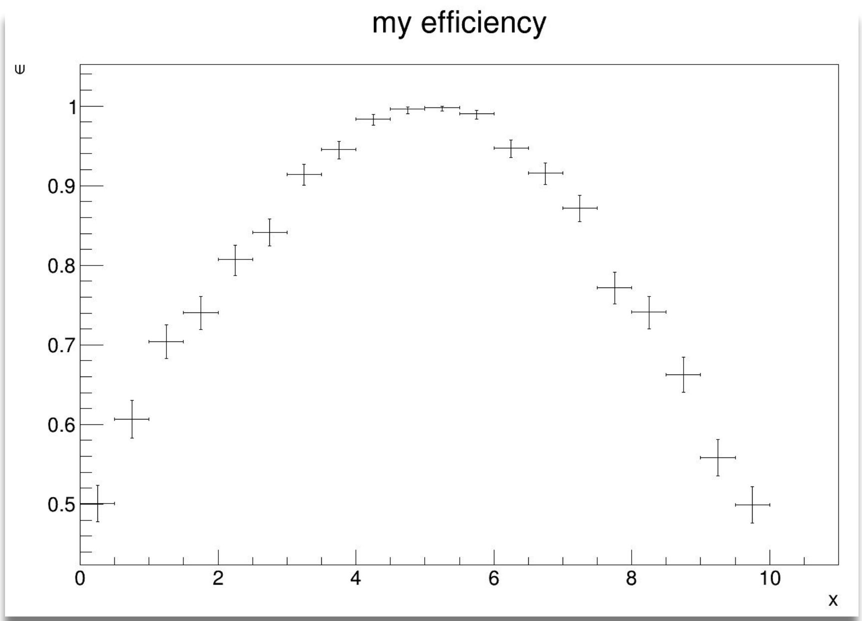
easy-to-find usage and options:

$ rootls --help

TEfficiency

- A class representing efficiency histograms
  - And get the uncertainties right...
  - 1,2,3 dimensions + weights
- Behaves like a histogram
  - `Fill(passFlag, value)`

[TEfficiency Documentation]



Parallelism in ROOT

- ***Explicit***: users manage parallelism (e.g. create threads)
  - **ROOT::EnableThreadSafety()**
  - TThreadExecutor and TProcessExecutor, TSpinLock
- ***Implicit***: ROOT manages parallelism internally
  - **ROOT::EnableImplicitMT() / root -t**
  - TTree I/O, fitting, RDataFrame

Parallelism is a requirement to tackle Run3 and HL-LH data analysis

Declarative Analyses with RDataFrame

Improving on current interfaces

```
ttree->Draw("pt", "eta > 2")

ttree->Draw("Muon_pt","Sum$(Muon_pt*(Muon_eta > 1)) > 30")
```

Improving on current interfaces

```
ttree->Draw("pt", "eta > 2")
```

```
ttree->Draw("Muon_pt","Sum$(Muon_pt*(Muon_eta > 1)) > 30")
```

- ad-hoc language allows to quickly specify queries
- can only produce histograms/graphs
- one event loop per histogram
- parallelisation is not possible
- relies on ROOT memory management of the histogram

Improving on current interfaces

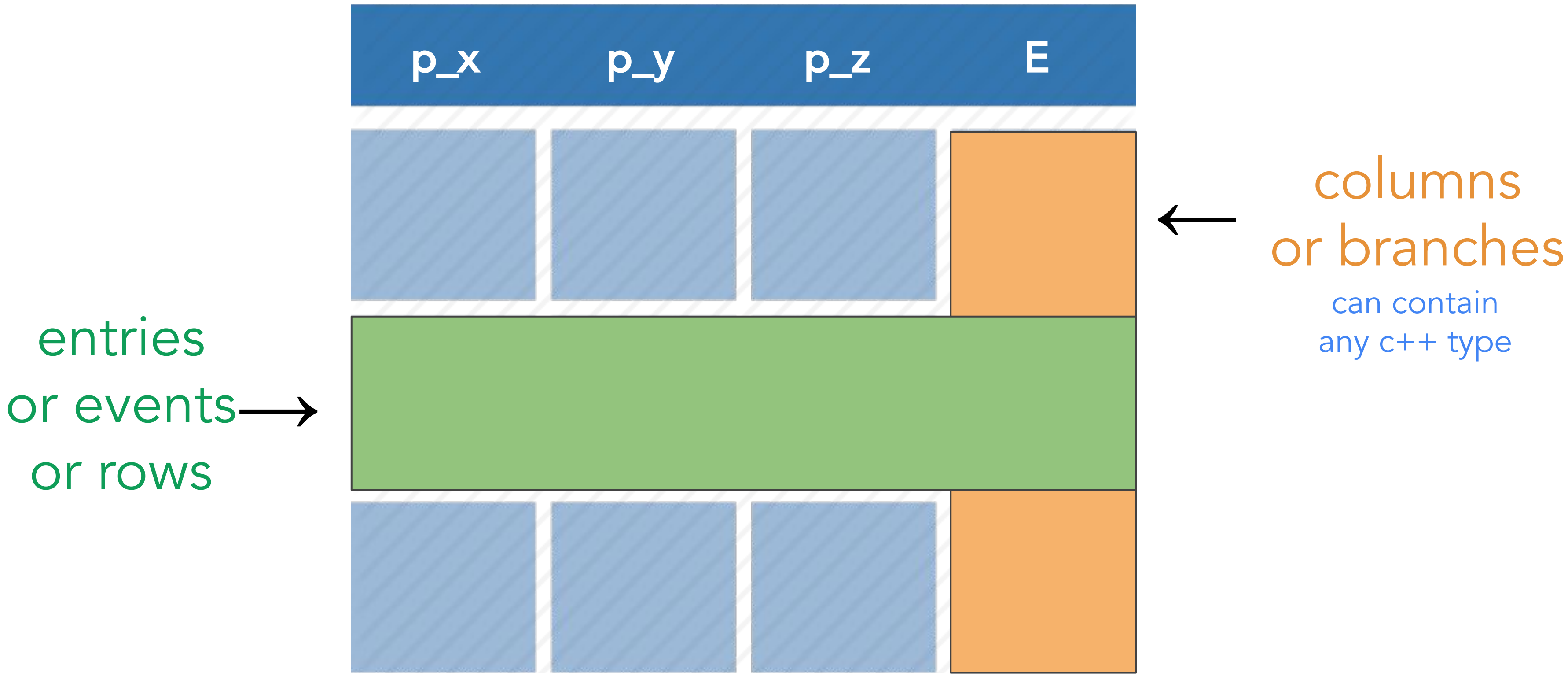
```
ttree->Draw("pt", "eta > 2")
```

```
ttree->Draw("Muon_pt","Sum$(Muon_pt*(Muon_eta > 1)) > 30")
```

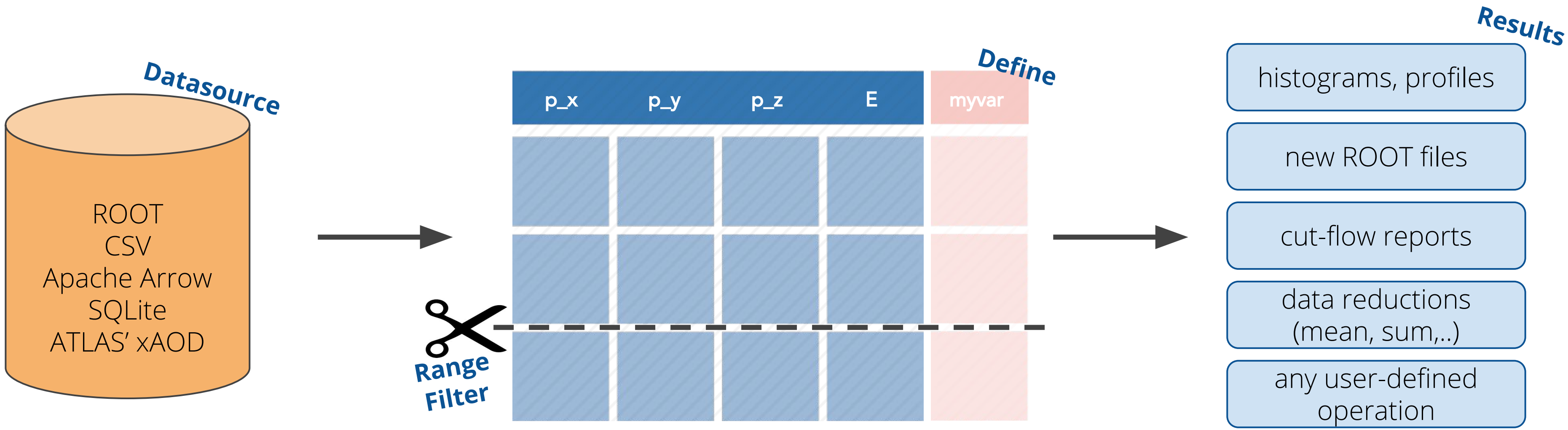
- ad-hoc language allows to quickly specify queries
- can only produce histograms/graphs
- one event loop per histogram
- parallelisation is not possible
- relies on ROOT memory management of the histogram

can we address these limitations without losing expressivity?

Columnar representation



RDataFrame in a nutshell



An ergonomic, fast C++ dataframe

```
ROOT::RDataFrame df("tree", "file.root");
```

on this (ROOT, CSV, ...) dataset

An ergonomic, fast C++ dataframe

```
ROOT::RDataFrame df("tree", "file.root");
```
on this (ROOT, CSV, ...) dataset

```
auto df2 = df.Filter("pt > 0")
```
only accept events for which pt > 0

An ergonomic, fast C++ dataframe

```
ROOT::RDataFrame df("tree", "file.root");        on this (ROOT, CSV, ...) dataset
auto df2 = df.Filter("pt > 0")                   only accept events for which pt > 0
             .Define("r", "sqrt(eta*eta + phi*phi)"); define r = sqrt(eta² + phi²)
```

An ergonomic, fast C++ dataframe

`ROOT::RDataFrame df("tree", "file.root");`	on this (ROOT, CSV, ...) dataset
`auto df2 = df.Filter("pt > 0")`	only accept events for which pt > 0
`.Define("r", "sqrt(eta*eta + phi*phi)");`	define r = sqrt(eta^2 + phi^2)
`auto rHist = df2.Histo1D("r");`	plot r for events that pass the cut

An ergonomic, fast C++ dataframe

```
ROOT::EnableImplicitMT();                               Run a parallel analysis
ROOT::RDataFrame df("tree", "file.root");               on this (ROOT, CSV, ...) dataset
auto df2 = df.Filter("pt > 0")                          only accept events for which pt > 0
             .Define("r", "sqrt(eta*eta + phi*phi)");   define r = sqrt(eta² + phi²)
auto rHist = df2.Histo1D("r");                          plot r for events that pass the cut
```

An ergonomic, fast C++ dataframe

```
ROOT::EnableImplicitMT();                              Run a parallel analysis
ROOT::RDataFrame df("tree", "file.root");              on this (ROOT, CSV, ...) dataset
auto df2 = df.Filter("pt > 0")                         only accept events for which pt > 0
             .Define("r", "sqrt(eta*eta + phi*phi)");  define r = sqrt(eta² + phi²)
auto rHist = df2.Histo1D("r");                         plot r for events that pass the cut
```

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

Quick RDF how-to

1. build a RDataFrame object by specifying your dataset
2. apply a series of transformations to your data
   - filter (e.g. apply some cuts) or
   - define new columns
3. apply actions to the transformed data to produce results (e.g. fill a histogram)

RDataFrame: Design Goals

simple and powerful interface

RDataFrame: Design Goals

simple and powerful interface

---

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

---

RDataFrame: Design Goals

simple and powerful interface

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

RDataFrame: Design Goals

simple and powerful interface

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

[RDF docs] [RDF tutorials]

RDataFrame feature overview

Lazy triggering of the event loop

```
ROOT::RDataFrame d("tree", "file.root");
auto histoCut = d.Filter("eta > 0").Histo1D("pt");
histoCut->Draw(); // event loop is run here, when you
                  // access a result for the first time
```

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

Lazy triggering of the event loop

```
ROOT::RDataFrame d("tree", "file.root");
auto histoCut = d.Filter("eta > 0").Histo1D("pt");
auto histoAll = d.Histo1D("pt");
...
// event loop is run here!
histoCut->Draw();
```

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

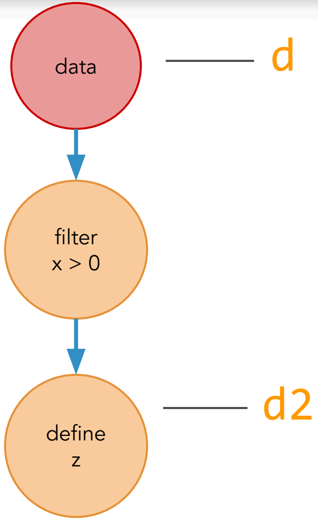
Define a new column

```
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 *(downstream)* you can use the new column as if it were present in your data.

Think of your analysis as data-flow

```
// d2 is a new data-frame, a transformed version of d
auto d2 = d.Filter("x > 0")
           .Define("z", "x*x + y*y");
```

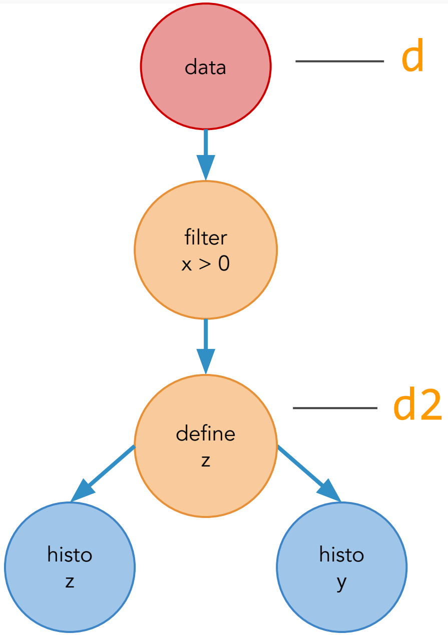


You can store transformed dataframes in variables, then use them as a RDataFrame.

Think of your analysis as data-flow

```
// select events, define a new column...
auto d2 = d.Filter("x > 0")
           .Define("z", "x*x + y*y");

// ...and make some histograms
auto hy = d2.Histo1D("y");
auto hz = d2.Histo1D("z");
```



Event loop will be run upon first access to a result:

hy-> or *hy

Weighted and multi-dim histograms

```
// 1D, x weighted with y, automatic range deduction
d2.Histo1D("x","y");
// 2D, x vs y, range is explicitly specified
d2.Histo2D({"hxy","hxy",
            100,-10.,10., 100,-10.10.},"x","y");
```

Model: parameters for the constructor of TH2F

1D histograms *may* take a model (title and axis range)
2D and 3D histograms always require a model

Passing C++ callables to RDF

```
d.Filter("eta > 0").Histo1D("pt")
```

=

```
auto IsPos = [](double x) { return x > 0; };
d.Filter(IsPos, {"eta"}).Histo1D<double>("pt");
```

You can pass the body of a C++ function as a string.
Or directly pass free functions, functor classes, etc...
avoiding some runtime penalty

Cutflow reports

```
auto dd = d.Filter("x > 0", "xcut").Filter("y < 2", "ycut");
dd.Report()->Print();
```

Thanks to idea from previous ATLAS tutorial!

Output

```
xcut    : pass=49 all=100 -- eff=49.00 % cumulative eff=49.00 %
ycut    : pass=24 all=49  -- eff=48.98 % cumulative eff=24.00 %
```

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

Saving data to ROOT files

```
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 (TTree, TFile) code.

RVec: numpy-like C++ collections

```
RVec<double> v = CreateMyRVec();
auto v2 = v[v > 3];
auto v3 = v[sin(v) < 0.5];
```

Easy filtering and transformations

Already integrated with RDataFrame

```
df.Define("pts", "sqrt(pxs*pxs + pys*pys)")
  .Define("good_pts", "pts[E > 100]")
  .Histo1D("good_pts");
```

https://root.cern/doc/master/classROOT_1_1VecOps_1_1RVec.html

No templates: C++ → JIT → Python

C++

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

No templates: C++ → JIT → Python

C++

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

C++ with cling's just-in-time compilation

```
d.Filter("theta > 0").Snapshot("tree","file.root","pt_x");
```

No templates: C++ → JIT → Python

C++

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

C++ with cling's just-in-time compilation

```
d.Filter("theta > 0").Snapshot("tree","file.root","pt_x");
```

PyROOT, automatically generated Python bindings

```
d.Filter("theta > 0").Snapshot("tree","file.root","pt_x")
```

RDataFrame and pandas

- similar concepts, some overlap in features
- different target applications:
  - large on-disk/remote datasets vs in-memory computation
  - potentially complex C++ objects vs numpy arrays
  - integration with ROOT vs integration with python libraries

Pick the right tool for your problem

RDataFrame: C++/PyROOT, GB+ of events, cuts and histograms, r/w ROOT files

pandas: “flat ntuple” that fits in memory, group-bys, sorts, ...

RDataFrame *to* pandas

```python
# Run input pipeline with C++ performance that can process TBs of data, reads from remote, ...
import ROOT
df = ROOT.RDataFrame("tree", "file.root")
         .Filter("Any(pt>30)",      "Trigger requirement")
         .Filter("All(tight_iso)", "Quality cut")
         .Define("r", "sqrt(eta*eta + phi*phi)")

# Read out final selection with defined variables as numpy arrays
col_dict = df.AsNumpy(["r", "eta", "phi"])

# Wrap data with pandas
import pandas
p = pandas.DataFrame(col_dict)
print(p)

   r      eta    phi
0  0.26   0.1   -0.5
1  1.0   -1.0    0.0
2  4.45   2.1    0.2
...
```

Available in v6.18

RDF transformations

Transformation	Description
Define	Creates a new column in the dataset.
DefineSlot	Same as `Define`, but the user-defined function must take an extra `unsigned int slot` as its first parameter. `slot` will take a different value, `0` to `nThreads - 1`, for each thread of execution. This is meant as a helper in writing thread-safe `Define` transformation when using `RDataFrame` after `ROOT::EnableImplicitMT()`. `DefineSlot` works just as well with single-thread execution: in that case `slot` will always be `0`.
DefineSlotEntry	Same as `DefineSlot`, but the entry number is passed in addition to the slot number. This is meant as a helper in case some dependency on the entry number needs to be honoured.
Filter	Filter the rows of the dataset.
Range	Creates a node that filters entries based on range of entries

RDF Actions

Lazy action	Description
Aggregate	Execute a user-defined accumulation operation on the processed column values.
Book	Book execution of a custom action using a user-defined helper object.
Cache	Caches in contiguous memory columns' entries. Custom columns can be cached as well, filtered entries are not cached. Users can specify which columns to save (default is all).
Count	Return the number of events processed.
Display	Obtains the events in the dataset for the requested columns. The method returns a RDisplay instance which can be queried to get a compressed tabular representation on the standard output or a complete representation as a string.
Fill	Fill a user-defined object with the values of the specified branches, as if by calling `Obj.Fill(branch1, branch2, ...).
Graph	Fills a **TGraph** with the two columns provided. If Multithread is enabled, the order of the points may not be the one expected, it is therefore suggested to sort if before drawing.
Histo{1D,2D,3D}	Fill a {one,two,three}-dimensional histogram with the processed branch values.
Max	Return the maximum of processed branch values. If the type of the column is inferred, the return type is `double`, the type of the column otherwise.
Mean	Return the mean of processed branch values.
Min	Return the minimum of processed branch values. If the type of the column is inferred, the return type is `double`, the type of the column otherwise.
Profile{1D,2D}	Fill a {one,two}-dimensional profile with the branch values that passed all filters.
Reduce	Reduce (e.g. sum, merge) entries using the function (lambda, functor...) passed as argument. The function must have signature `T(T,T)` where T is the type of the branch. Return the final result of the reduction operation. An optional parameter allows initialization of the result object to non-default values.
Report	Obtains statistics on how many entries have been accepted and rejected by the filters. See the section on named filters for a more detailed explanation. The method returns a RCutFlowReport instance which can be queried programmatically to get information about the effects of the individual cuts.
StdDev	Return the unbiased standard deviation of the processed branch values.
Sum	Return the sum of the values in the column. If the type of the column is inferred, the return type is `double`, the type of the column otherwise.
Take	Extract a column from the dataset as a collection of values. If the type of the column is a C-style array, the type stored in the return container is a **`ROOT::VecOps::RVec`**<T> to guarantee the lifetime of the data involved.

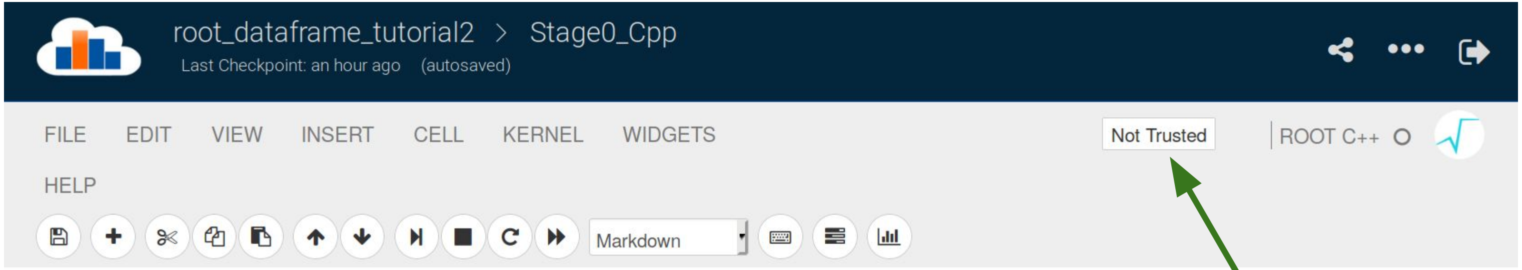
RDF Actions and Other Operations

Instant action	Description
Foreach	Execute a user-defined function on each entry. Users are responsible for the thread-safety of this lambda when executing with implicit multi-threading enabled.
ForeachSlot	Same as `Foreach`, but the user-defined function must take an extra `unsigned int slot` as its first parameter. `slot` will take a different value, `0` to `nThreads - 1`, for each thread of execution. This is meant as a helper in writing thread-safe `Foreach` actions when using `RDataFrame` after `ROOT::EnableImplicitMT()`. `ForeachSlot` works just as well with single-thread execution: in that case `slot` will always be `0`.
Snapshot	Writes processed data-set to disk, in a new `TTree` and `TFile`. Custom columns can be saved as well, filtered entries are not saved. Users can specify which columns to save (default is all). Snapshot, by default, overwrites the output file if it already exists. `Snapshot` can be made *lazy* setting the appropriate flage in the snapshot options.

Other Operations

Operation	Description
Alias	Introduce an alias for a particular column name
GetColumnNames	Get all the available columns of the dataset
GetColumnType	Return the type of a given column as a string.
GetFilterNames	Get all the filters defined. If called on a root node, all filters will be returned. For any other node, only the filters upstream of that node.
Display	Provides an ASCII representation of the columns types and contents of the dataset printable by the user.
SaveGraph	Store the computation graph of an RDataFrame in graphviz format for easy inspection.

Let’s Get Our Hands Dirty!



click this button to execute a cell

click this button to restart the notebook

click this button to “trust” the notebook

More on RDF

Event Loop Callbacks

Callbacks can be used to inspect partial results of the analysis while the event loop is running, or execute a function at constant intervals.
E.g. one can draw an up-to-date version of a result histogram every 100 entries:

```
auto h = df.Histo1D("x");
TCanvas c("c","x hist");
h.OnPartialResult(100, [&c](TH1D &h_) {
    c.cd(); h_.Draw(); c.Update();
});
// event loop runs here
// `Draw` is executed after the event loop is finished
h->Draw();
```

Reading CSV files with RDataFrame

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

```
ROOT::EnableImplicitMT();
auto df = ROOT::MakeCsvDataFrame("data.csv");
auto zHist = df.Filter(cutFun, {"x","y","z"})
               .Histo1D("z");
df.Snapshot("tree", "output.root","[xyz]");
```

RDataFrame's nuke bomb: Foreach

```
ROOT::EnableImplicitMT();
auto df = RDataFrame("tree","f.root",{"x","y"});
df.Filter(IsGood).Foreach(DoStuff);
```

Full control over what happens during the (parallel) event-loop:
execute `DoStuff(x,y)`
for all events that pass `IsGood(x,y)`

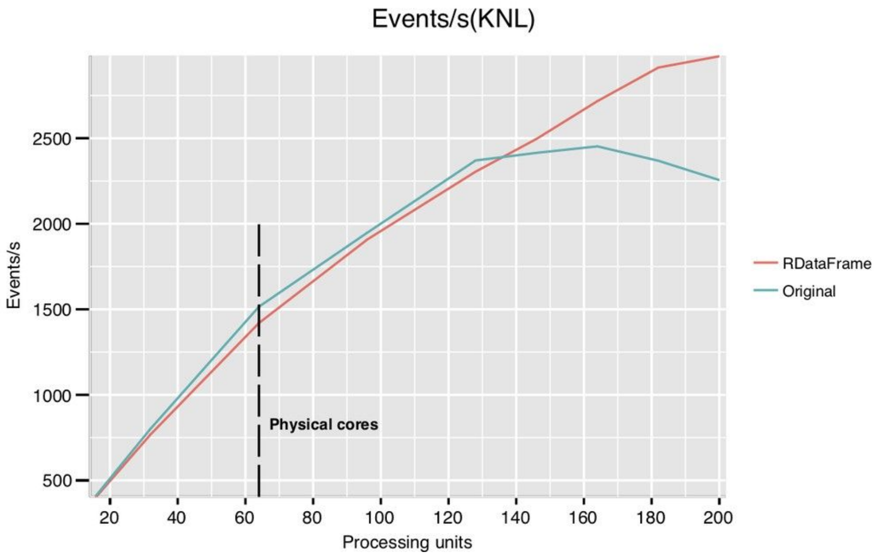
Creating a ROOT dataset from scratch

```
ROOT::EnableImplicitMT();
auto df = RDataFrame(10000);
tdf.Define("x", randomDouble)
   .Snapshot("tree", "f.root");
```

Full control over what happens during the (parallel) event-loop:
execute `DoStuff(x,y)`
for all events that pass `IsGood(x,y)`

RDataFrame: does it scale?

RDF was benchmarked on a many-core KNL machine against the same multi-thread analysis written in a patched ROOT5



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

source: Xavier Valls Pla, ROOT team

High-level customization points: RDataSource

User $\xrightarrow{\text{uses}}$ RDataFrame $\xrightarrow{\text{queries}}$ RDataSource $\xrightarrow{\text{reads}}$ Data

- ➔ RDataFrame **can read non-ROOT data** through RDataSource objects
- ➔ **third parties** can implement and **seamlessly integrate** RDataSource implementations for their format of choice

- ➔ CSV and Apache Arrow currently supported via RDataSource
- ➔ prototypes for LHCb's MDF binary data format and ATLAS' xAOD event model

DOI 10.5281/zenodo.1303038

More PyROOT

Contiguous Memory to np array

Zero-copy C++ to NumPy array conversion

- Objects with contiguous data (`std::vector`, `RVec`)
- Pythonization tells NumPy about data and shape

```python
import ROOT
import numpy as np

vec = FunctionThatReturnsStdVector()
arr = np.asarray(vec) # zero-copy operation
vec[0], vec[1] = 1, 2

assert arr[0] == 1 and arr[1] == 2
```

Memory adopted!