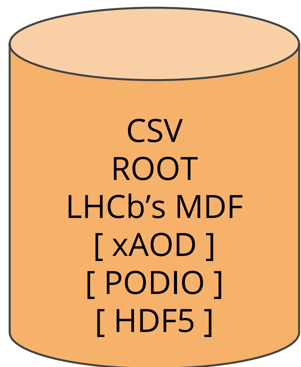
The TDataFrame rant

Thoughts on the evolution of declarative analysis in ROOT

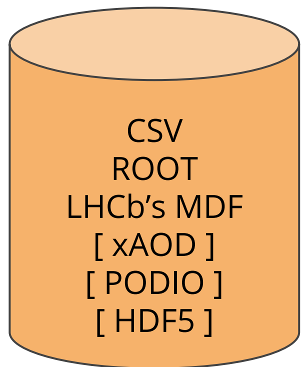
Enrico Guiraud, PPP meeting 22-02-'18

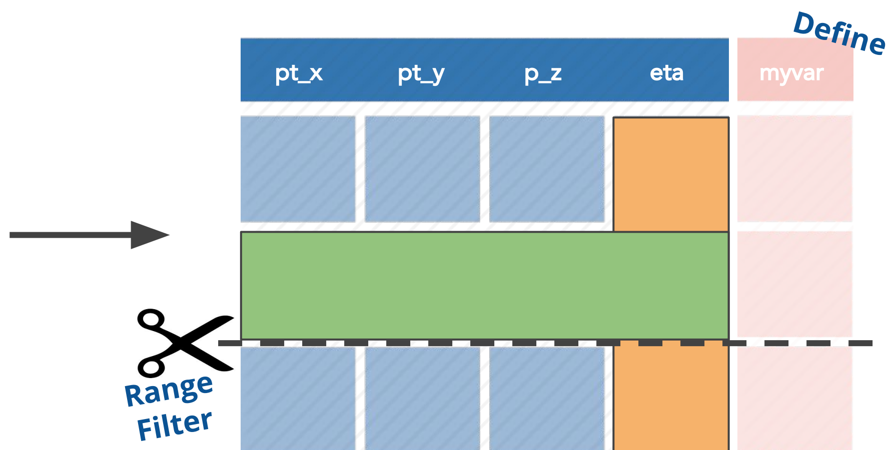
How I think of TDataFrame



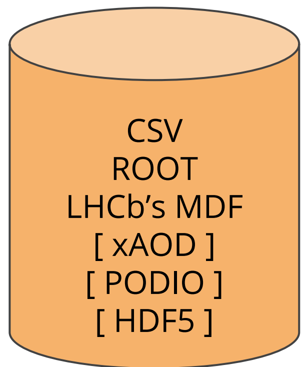
pt_x	pt_y	p_z	eta

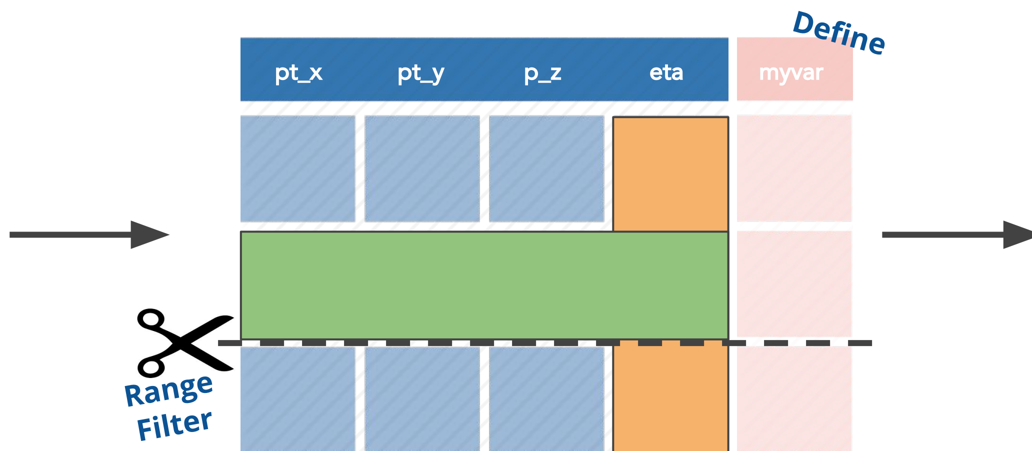
How I think of TDataFrame





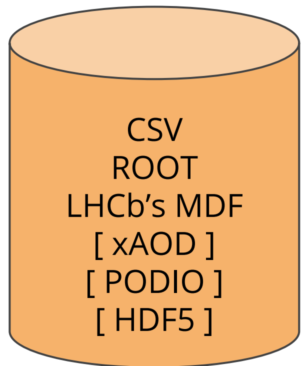
How I think of TDataFrame

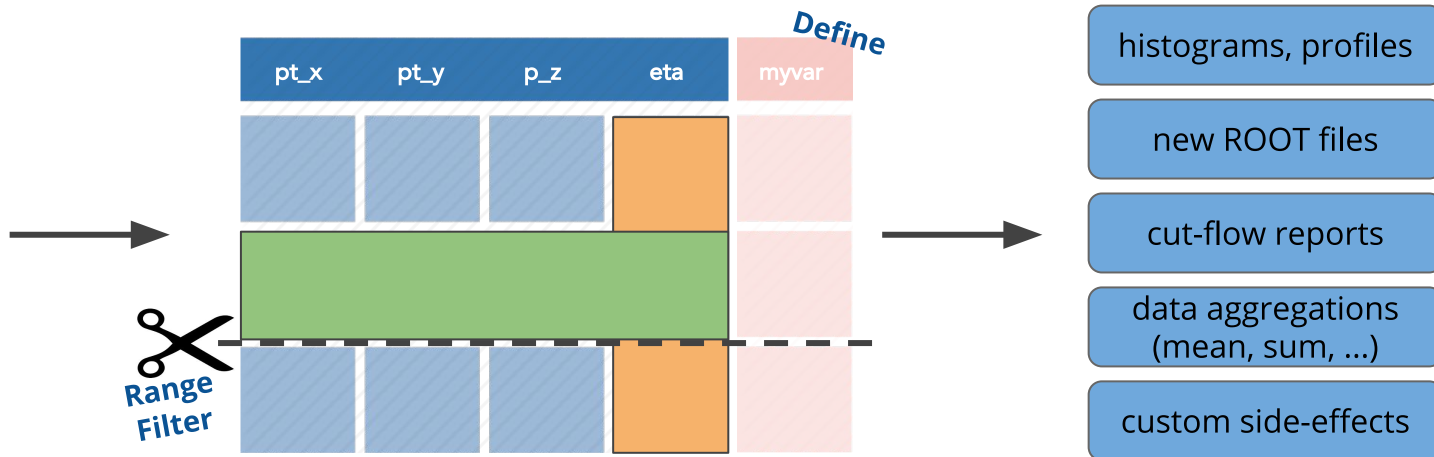




- histograms, profiles
- new ROOT files
- cut-flow reports
- data aggregations (mean, sum, ...)
- custom side-effects

How I think of TDataFrame





histograms, profiles

new ROOT files

cut-flow reports

data aggregations (mean, sum, ...)

custom side-effects

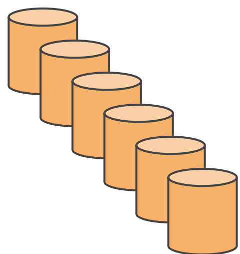
Goals

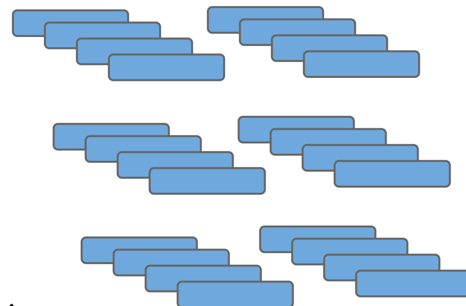
- being the fastest way to get physicists to their results
- being the go-to ROOT analysis interface from 1 to 100 cores, laptop to cluster, with little to no change in user code
- ease transition between python and c++ (in both directions)

Other interfaces (TMVA, TBrowser) could be powered by under-the-hood usage of TDF

A simple model: analysis in a can

Mapping an analysis to multiple data-sets

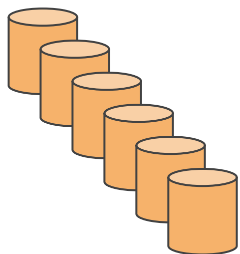


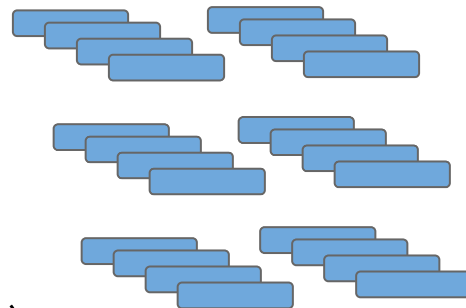


```
Results TDFAnalysis(TDataFrame &d);
auto ResultsList = MapTDF(TDFAnalysis, datasets)
```

A simple model: analysis in a can

Mapping an analysis to multiple data-sets





```
Results TDFAnalysis(TDataFrame &d);
auto ResultsList = MapTDF(TDFAnalysis, datasets)
```

Same concept can be applied for running a TDFAnalysis distributedly.
TDataFrame inherently knows how to *reduce* results, because it already does it in multi-thread runs

High-level requirements for a better TDF

pyTDF

TDF python interface is missing crucial ingredients

- ➔ integration with other common python libraries (i.e. numpy, see Stefan's presentation)
- ➔ integration with python's functional environment (lambdas, python functions)
- ➔ re-Define a column (natural to have in an interactive environment)
- ➔ more pythonic interfaces
- ➔ jitting is *slow* because of multiple calls to the interpreter, but a TDF-side fix is coming

Already in ROOT's POW, so I'm not saying anything new

Missing actions

Some useful actions are still missing (summer student project)

- ➔ StdDev
- ➔ Histo1D(“x”, preprocessingLambda)
- ➔ Histo2D/3D with autobinning (no histo model required)
- ➔ Head/Display
- ➔ Graph

but:

- ➔ we cannot satisfy all needs with TDF methods
- ➔ we want a lean interface.
- ➔ users need a way to run their custom actions

Custom actions

Booking your own action:

```
auto customResult = tdf.Book(CustomHelper("x"))
```

Custom actions

Booking your own action:

```
auto customResult = tdf.Book(CustomHelper("x"))
```

what to execute? MyCustomHelper::Exec
on what columns? CustomHelper.GetColumnNames()
of what types? CustomHelper::ColumnTypes_t

what type to return? TResultProxy<MyCustomHelper::Result_t>

Custom actions

Booking your own action:

```
auto customResult = tdf.Book(CustomHelper("x"))
```

what to execute? MyCustomHelper::Exec
on what columns? CustomHelper.GetColumnNames()
of what types? CustomHelper::ColumnTypes_t

what type to return? TResultProxy<MyCustomHelper::Result_t>

with pipes: TDF pipes gist

```
auto customResult = tdf | DefineQuantities() | ApplyFilters() | CustomAction();
```

as a side note: what would custom transformations *mean*? custom event loop manager?

Faster I/O

I/O is an analysis bottleneck in all realistic applications

Some ideas on how TDF could help:

- ➔ democratization of latest features (e.g. TBufferMerger), in a transparent manner
- ➔ “normalization” of user-facing interfaces, low-level adoption of all new advancements
- ➔ integration of bulk I/O when ready, e.g. via a TDataSource:
  for simple data models we can be much faster (see e.g. uproot)
- ➔ lobby to reduce TBranchProxy’s overhead :)
- ➔ easy explicit caching
- ➔ to be investigated: performance of multi-thread I/O over network: here be dragons
- ➔ to be investigated: implicit caching
- ➔ a serious performance optimization of TDataFrame has yet to be done:
  writing benchmarks for rootbench will help greatly

Outstanding issues
&
Short term improvements

Writing multiple files with one event loop

Problem: the following code loops over data twice, at each Snapshot call:

```
auto newTDF1 = df.Snapshot(“t”, “sig.root”, {“signal1”, “signal2”});
auto newTDF2 = df.Snapshot(“t”, “bkg.root”, {“bkg1”, “bkg2”});
```

- ➔ users will expect to call Snapshot and find their files on disk
- ➔ we don’t want users to directly trigger the event loop
- ➔ expert users should be able to tweak this behaviour to write multiple files in one loop

Proposed solution

```
TDF::TSnapshotOptions  opts;
opts.fLazy = true;
auto newTDF1 = df.Snapshot(“t”, “sig.root”, {“signal1”, “signal2”},
opts);
auto newTDF2 = df.Snapshot(“t”, “bkg.root”, {“bkg1”, “bkg2”});
```

Rename TResultProxy -> TResultPtr

Currently, TDataFrame actions return TResultProxy<T>: a smart pointer (with shared ownership) that triggers the TDF event-loop (if needed) upon access to the pointee.

Problem: users do not know what to do with a “proxy”

Rename TResultProxy -> TResultPtr

Currently, TDataFrame actions return TResultProxy<T>: a smart pointer (with shared ownership) that triggers the TDF event-loop (if needed) upon access to the pointee.

Problem: users do not know what to do with a “proxy”

Proposed solution:

- ➔ add to **TResultProxy** what is missing to behave like a **std::shared_ptr**
- ➔ change its name to **TResultPtr**

Rationale is better teachability and adherence to the principle of least surprise:

- ➔ TResultPtr instantly indicates the semantics of the object, which TResultProxy does not
- ➔ TResultPtr instantly indicates that the pointee is accessible via `->`
- ➔ C++ users won’t find it weird to declare a **vector<TResultPtr<TH1D>>**, while a **vector<TResultProxy<TH1D>>** sounds scarier (what is a proxy? how does it behave?)

Alternatives like **TResultFuture** have been considered, but in my opinion they do not convey the right information, that is “how to use” rather than “how it works”

Less surprising cut-flow reports

The “old” `Report` interface:

```
tdf.Filter("x > 0", "x_cut").Filter("y > 10", "y_cut").Report();
```

The problem: `Report` is *weird*:

- ➔ it’s not lazy nor “instant”: it triggers the event-loop if it needs to
- ➔ it looks like an action, but does not return anything
- ➔ impossible to programmatically inspect reports
- ➔ it prints on screen

Proposed solution

```
auto report = tdf.Filter("x > 0", "x_cut").Filter("y > 10", "y_cut").Report();
report->Print(); // event loop triggered here, lazily
...
std::cout << report["x_cut"].GetEff() << std::endl;
```

we are already halfway there thanks to Danilo

Improved internal ownership model

Problem:

- ➔ currently the first (head) node of a TDF computation graph owns all the other nodes
- ➔ nothing works (everything throws) if the head node goes out of scope before its children

Proposed solution: each node will keep a shared_ptr to its parent.

- all nodes that are required to stay alive automatically stay alive
- nodes that are part of “dead branches” in the graph are pruned automatically
- one-liners become possible: `auto h = TDataFrame().Define().Filter().Histo1D()`

but

- actions need to register and de-register themselves with the head node
- jitting and Reports might become more complicated with the new model
- performance impact of calling parent node’s methods via shared_ptr vs reference is to be measured: https://godbolt.org/g/pr7Pih