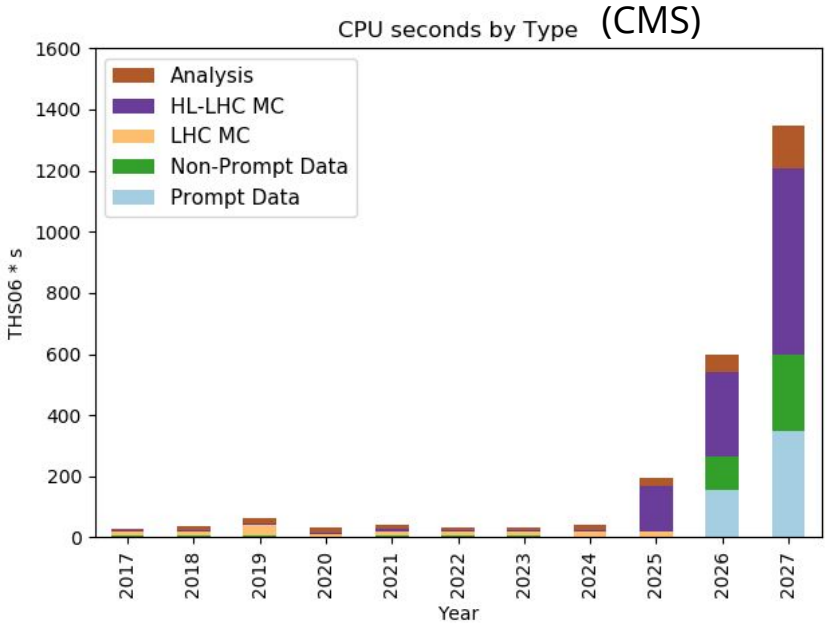
ROOT::RDataFrame

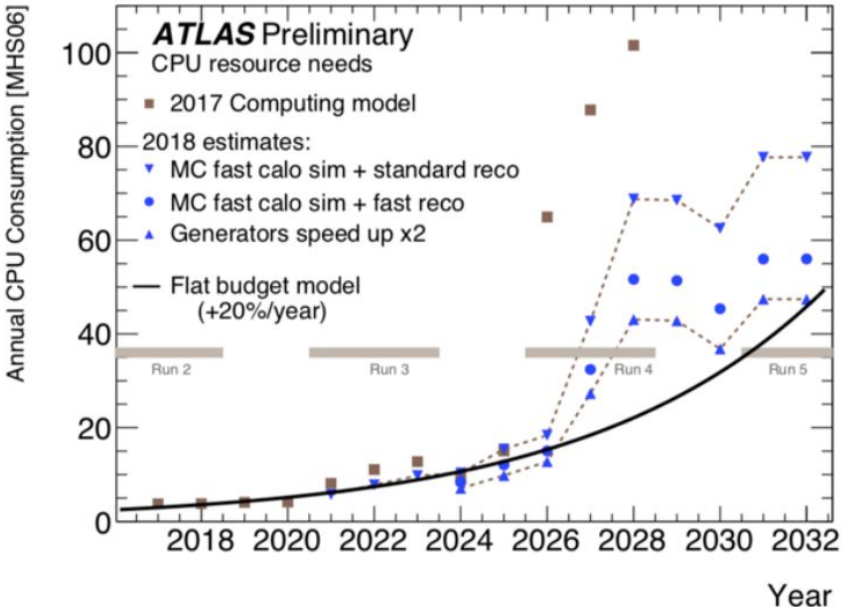
a swiss-army knife for data analysis and manipulation

Enrico Guiraud for the ROOT team
EP Software Seminar, October 2019, CERN

ROOT: a foundation library

➔ The amount of data processed by HEP scientists is going to increase drastically





sources: CMS, ATLAS

ROOT: a foundation library

- ➔ The amount of data processed by HEP scientists is going to increase drastically

- ➔ Multi-core hardware is commonplace

  many-core architectures and computing clusters are increasingly available

ROOT: a foundation library

- ➔ The amount of data processed by HEP scientists is going to increase drastically

- ➔ Multi-core hardware is commonplace

  many-core architectures and computing clusters are increasingly available

- ➔ ROOT's mission does not change:

  bring physicists from data to publication as effectively as possible

We need better, faster software that takes full advantage of the underlying hardware.

On the other hand, in the context of data analysis, we cannot expect HEP physicists to also be high-performance computing experts.

What's out there: Pandas, ReactiveX

Everything is a table, with powerful high-level transformations

Designed for in-memory data processing

Python only, **integrated with its data science ecosystem**

Strong arguments for a **functional/stateless** interface

Abstracts away **workflow parallelization**

Designed for streams of asynchronous events/values

In the community: LINQ, AwkwardArray

LINQtoROOT

first presented:
G. Watts, CHEP 2012

SQL-like query language geared towards HEP analyses

Caching of results

Tied to C#/LINQ

Awkward Array

DOI 10.5281/zenodo.3275017

Easily manipulate **arrays of (jagged) arrays**

In-memory data model fits HEP data well

Implementation is Python only (but idea is portable)

A recipe for efficient HEP analyses

- ➔ strive for a **simple programming model**
- ➔ expose modern, elegant interfaces that are **easy to use correctly** and hard to use incorrectly
- ➔ **transparently benefit from multi-core** hardware
- ➔ scale up and scale down: 1 to 100 cores to computing cluster
- ➔ make **common tasks simple, complex tasks possible**
- ➔ consistent support for HEP languages: **C++ and Python**

A recipe for efficient HEP analyses

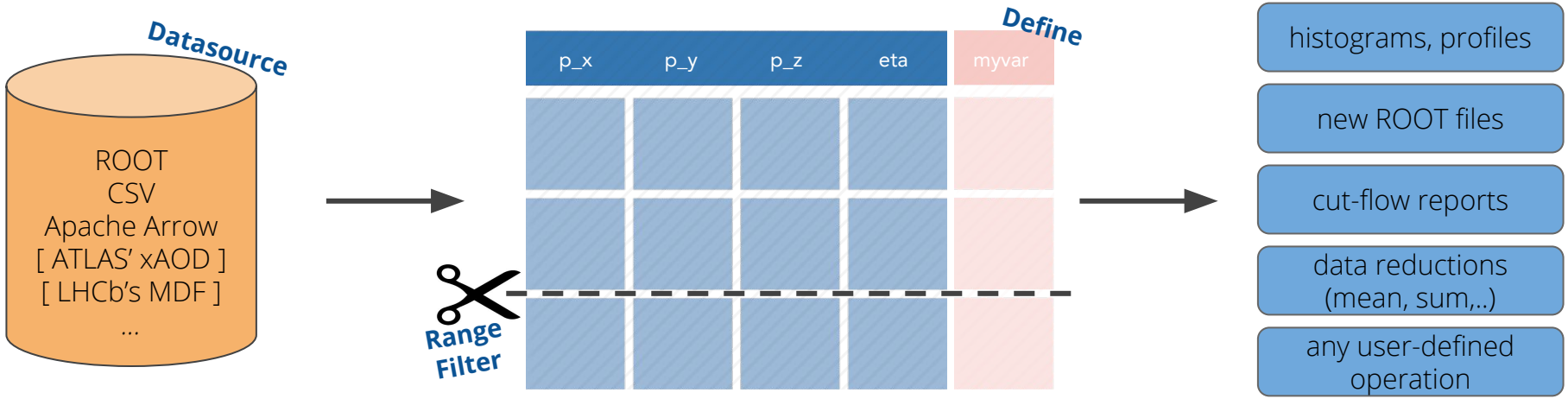
- ➔ strive for a **simple programming model**
- ➔ expose modern, elegant interfaces that are **easy to use correctly** and hard to use incorrectly
- ➔ **transparently benefit from multi-core** hardware
- ➔ scale up and scale down: 1 to 100 cores to computing cluster
- ➔ make **common tasks simple, complex tasks possible**
- ➔ consistent support for HEP languages: **C++ and Python**

RDataFrame is what you get if you follow this recipe.

RDF user guide, cheat-sheet

DOI 10.5281/zenodo.260230

RDataFrame in a nutshell



Design principles

Elements of **declarative programming**
"user says what, ROOT chooses how"

High level interfaces provide less typing, increased readability, abstraction of complex operations

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

Design principles

Elements of **declarative programming**
"user says what, ROOT chooses how"

High level interfaces provide less typing, increased readability, abstraction of complex operations

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

Elements of **functional programming**
pure functions & higher-order functions

Users code in terms of **small reusable components**

Less side-effects and less shared state increase **thread-safety and code correctness**

The result: an ergonomic C++ dataframe

```
ROOT::RDataFrame df(dataset);
```

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

The result: an ergonomic C++ dataframe

```
ROOT::RDataFrame df(dataset);
```

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

```
auto df2 = df.Filter("x > 0")
```

only accept events for which x > 0

The result: an ergonomic C++ dataframe

```
ROOT::RDataFrame df(dataset);
```
on this (ROOT, CSV, ...) dataset

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

```
            .Define("r2", "x*x + y*y");
```
define $r2 = x^2 + y^2$

The result: an ergonomic C++ dataframe

```
ROOT::RDataFrame df(dataset);
```
on this (ROOT, CSV, ...) dataset

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

```
             .Define("r2", "x*x + y*y");
```
define r2 = $x^2 + y^2$

```
auto rHist = df2.Histo1D("r2");
```
plot r2 for events that pass the cut

The result: an ergonomic C++ dataframe

```
ROOT::RDataFrame df(dataset);
auto df2 = df.Filter("x > 0")
             .Define("r2", "x*x + y*y");
auto rHist = df2.Histo1D("r2");
df2.Snapshot("newtree", "out.root");
```

- on this (ROOT, CSV, ...) dataset
- only accept events for which x > 0
- define r2 = $x^2 + y^2$
- plot r2 for events that pass the cut
- write the skimmed data and r2 to a new ROOT file

The result: an ergonomic C++ dataframe

```
ROOT::EnableImplicitMT();
ROOT::RDataFrame df(dataset);
auto df2 = df.Filter("x > 0")
             .Define("r2", "x*x + y*y");
auto rHist = df2.Histo1D("r2");
df2.Snapshot("newtree", "out.root");
```

- Run a parallel analysis
- on this (ROOT, CSV, ...) dataset
- only accept events for which x > 0
- define r2 = $x^2 + y^2$
- plot r2 for events that pass the cut
- write the skimmed data and r2 to a new ROOT file

The result: an ergonomic C++ dataframe

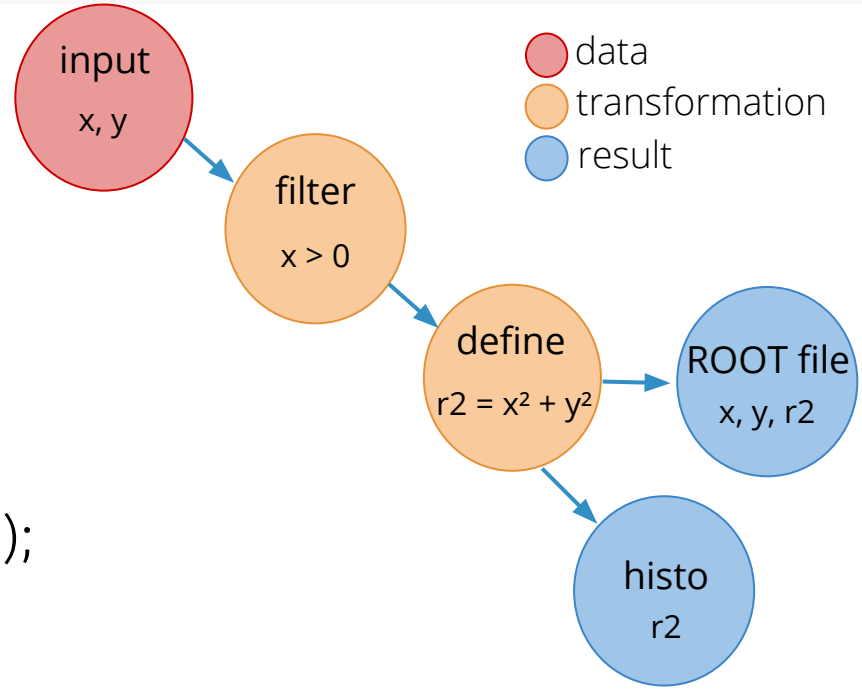
```
ROOT::EnableImplicitMT();
ROOT::RDataFrame df(dataset);
auto df2 = df.Filter("x > 0")
             .Define("r2", "x*x + y*y");
auto rHist = df2.Histo1D("r2");
df2.Snapshot("newtree", "out.root");
```

- Run a parallel analysis
- on this (ROOT, CSV, ...) dataset
- only accept events for which x > 0
- define r2 = $x^2 + y^2$
- plot r2 for events that pass the cut
- write the skimmed data and r2 to a new ROOT file

Lazy execution guarantees that all operations are performed **in a single event loop**

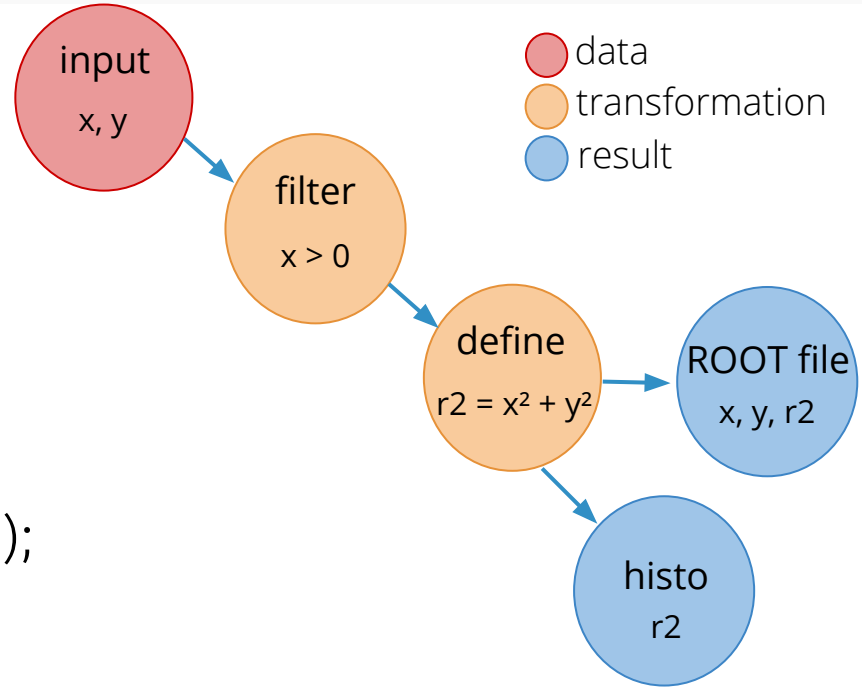
Analyses as computation graphs

```
ROOT::RDataFrame df(dataset);
auto df2 = df.Filter("x > 0")
             .Define("r2", "x*x + y*y");
auto rHist = df2.Histo1D("r2");
df2.Snapshot("newtree", "newfile.root");
```



Analyses as computation graphs

```
ROOT::RDataFrame df(dataset);
auto df2 = df.Filter("x > 0")
             .Define("r2", "x*x + y*y");
auto rHist = df2.Histo1D("r2");
df2.Snapshot("newtree", "newfile.root");
```



Re-thinking one's analysis as a computation graph is often the biggest challenge when trying out RDataFrame for the first time.

Full modern C++ support…

C++

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

Full modern C++ support...

or any type ROOT is able to read

C++

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

...with just-in-time compilation...

C++

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

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

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

...and Python bindings

C++

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

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

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

PyROOT, automatically generated Python bindings

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

...and Python bindings

simpler

C++

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

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

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

PyROOT, automatically generated Python bindings

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

faster, more flexible

Handling collections

RVec is a new type that allows operating on vectors "à la numpy"

```
RVec<double> v = ...;
auto v2 = v[v > 3];
auto v3 = v[sin(v) < .5];
```

RDataFrame reads arrays and vectors as RVecs

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

RVec reference guide

RDF in the Python world

Define your workflow in Python and run the event loop in C++

```
df = ROOT.RDataFrame(...)
ROOT.gInterpreter.Declare('#include "lib.h"')
df = df.Define('z', 'lib::myfunc(x, y)')
```

OR

Use RDF as a bridge to the python data science ecosystem, e.g. perform large cuts on remote files and then export to numpy

```
columns = ROOT.RDataFrame(...)\
  .AsNumpy(["muon_pt", "muon_eta"])
pandas_df = pandas.dataframe(columns)
```

RDF: extensible through customization points

We cannot implement all features that our large user base requires at the timescale they require.

However, we can **make RDF easily extensible** to cover one's particular needs.

Read any columnar data format by implementing your own RDataSource

Book arbitrary computations to be executed within RDF's lazy event loop

High-level customization points: RDataSource

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

High-level customization points: RDataSource

- ➔ RDataFrame **can read non-ROOT data** through RDataSource objects
- ➔ **third parties** can implement and **seamlessly integrate** RDataSource implementations for their format of choice
- ➔ CSV, Apache Arrow and RNTuple currently supported via RDataSource
- ➔ prototypes for LHCb's MDF binary data format and ATLAS' xAOD event model

DOI 10.5281/zenodo.1303038

High-level customization points: RDataSource

- ➔ RDataFrame **can read non-ROOT data** through RDataSource objects
- ➔ **third parties** can implement and **seamlessly integrate** RDataSource implementations for their format of choice
- ➔ CSV, Apache Arrow and RNTuple currently supported via RDataSource
- ➔ prototypes for LHCb's MDF binary data format and ATLAS' xAOD event model

Users can write the **same code independently of the data format** analyzed

Current R&D: RDF analyses on HPC clusters

1. user logins to SWAN, requests a connection to a Spark computing cluster
2. writes RDF analysis in python notebook
3. computation is dispatched to cluster, user can access results as usual

Thanks to the declarative paradigm,
distributed computation is transparent to the user

Published at EuroPar 2019:
Declarative Big Data Analysis for High-Energy Physics: TOTEM Use Case

github.com/JavierCVilla/PyRDF

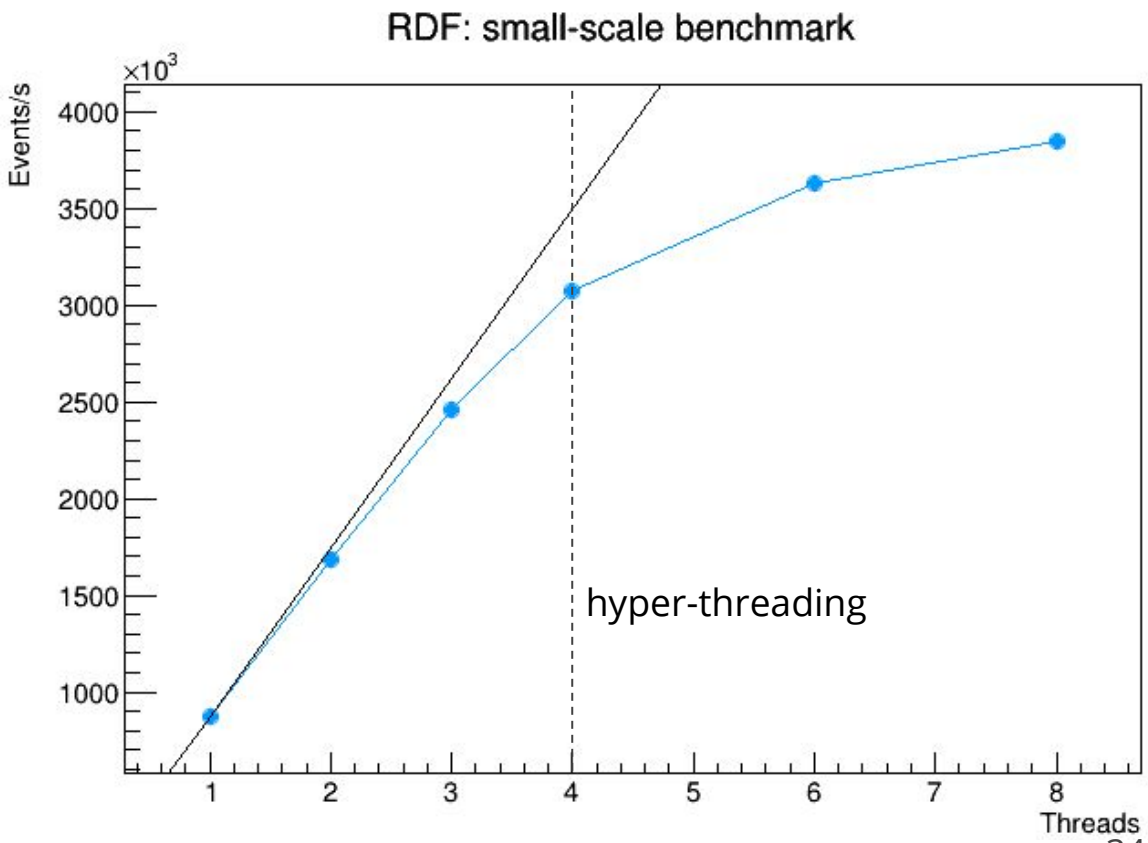
Scaling benchmark on toy analysis

Simple cut+calculation

Intel Core i7-4790 CPU
3.60GHz
4 physical cores
warm cache

CPU throttling disabled

Code and data available here

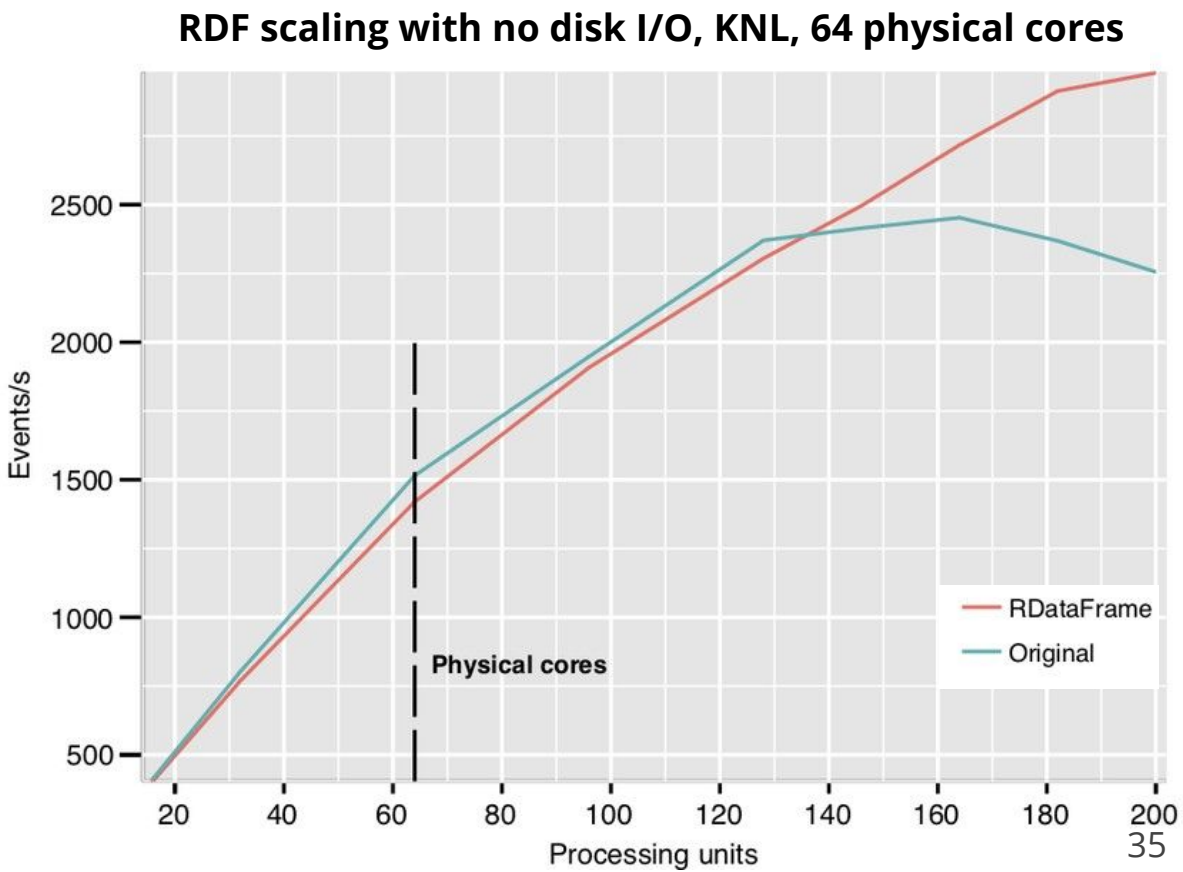


Scaling on many-core hardware (no I/O)

Monte Carlo QCD Low-Pt events generation+ analysis on the fly

Ad-hoc implementation
(patched ROOT 5 + POSIX threads)
vs
RDF

Original performance analysis
by X. Valls Pla, CERN IT Techlab

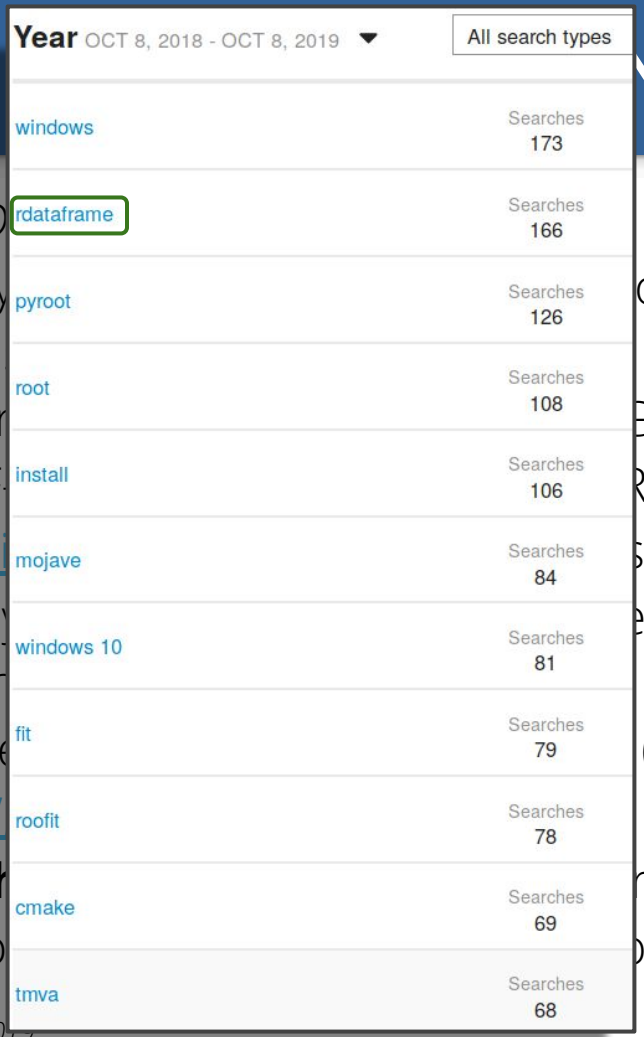


RDF and CERN experiments

- **CMS**: *Nail* and *RDFprocessor* frameworks
- **ATLAS**: LoopSusyFrame, xAOD RDataSource, software tutorials
- **LHCb**: MDF data source
- **ALICE**: Apache Arrow support contributed by G. Eulisse, currently investigating RDF-based analysis workflow for Run 3 (D. Berzano, The ALICE Analysis Framework for LHC Run 3, slides 14-17)
- **Totem**: RDF analysis on Spark clusters, see previous slides
- **SHiP**: RDF presented at software tutorials
- **FCC** plans to develop workflows based on RDF (C. Helsens, "General status and plans", FCC week 2019, slide 20)
- many users "**in the wild**": 270 posts tagged "#rdataframe" on the ROOT forum as of today, 56 out of 69 bug reports fixed

N experiments

- **CMS**: *Nail* and *RD*
- **ATLAS**: LoopSusy ... oftware tutorials
- **LHCb**: MDF data
- **ALICE**: Apache Ar ... G. Eulisse, currently investigating RDF- ... Run 3 (D. Berzano, The ALICE Analysi ... slides 14-17)
- **Totem**: RDF anal ... evious slides
- **SHiP**: RDF preser
- **FCC** plans to deve ... (C. Helsens, "General status and plans"
- many users "**in th** ... rdataframe" on the ROOT forum as o ... orts fixed



Summary, outlook

Since v6.14 (2017), ROOT provides a **modern, high-level, type-safe, parallel** interface for data analysis and manipulation

Summary, outlook

Since v6.14 (2017), ROOT provides a **modern, high-level, type-safe, parallel** interface for data analysis and manipulation

One paradigm to fit all workflows

- ➔ from quick data exploration on a laptop…
- ➔ …to large-scale studies on many-core architectures…
- ➔ …to ntuple → ntuple transformations or even data generation
- ➔ already used in production by physicists of major LHC experiments: let's hear how it worked out for Elisabetta!

Summary, outlook

Since v6.14 (2017), ROOT provides a **modern, high-level, type-safe, parallel** interface for data analysis and manipulation

One paradigm to fit all workflows

- ➔ from quick data exploration on a laptop…
- ➔ …to large-scale studies on many-core architectures…
- ➔ …to ntuple → ntuple transformations or even data generation
- ➔ already used in production by physicists of major LHC experiments: let's hear how it worked out for Elisabetta!

Coming soon

- ➔ low-level **performance optimization** (e.g. batch processing)
- ➔ more pythonic pyROOT bindings and quality of life upgrades

Getting in touch with the ROOT team

Ask for help
root-forum.cern.ch

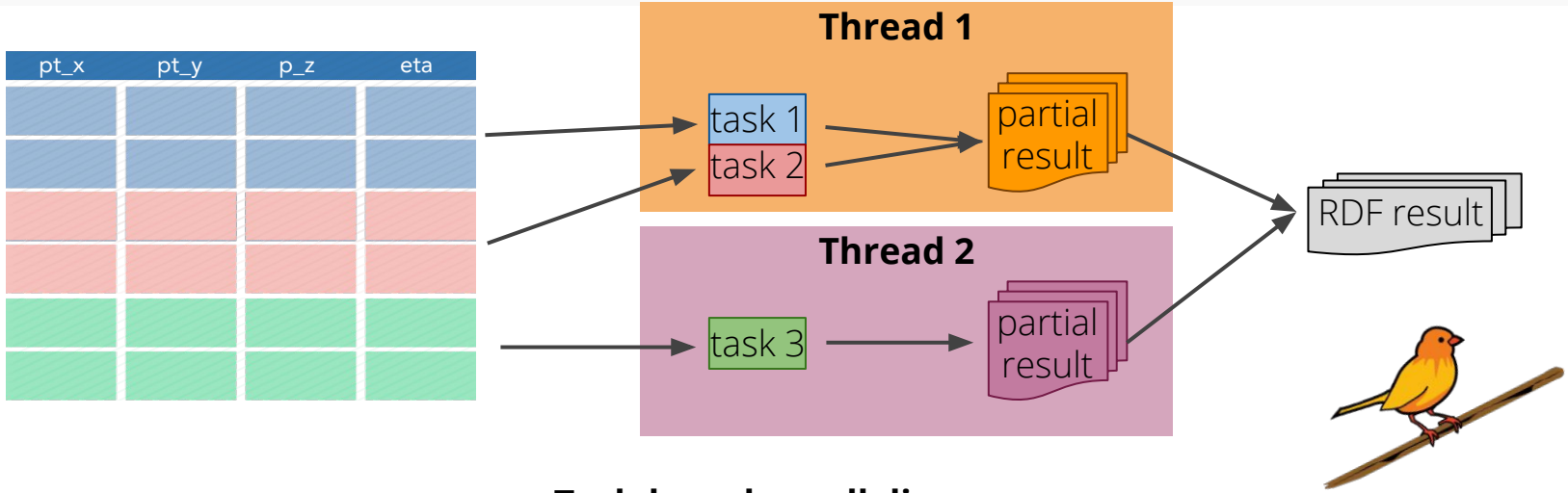
Contribute
github.com/root-project/root

Report bugs
sft.its.cern.ch/jira/projects/ROOT/issues

Discuss
mattermost.web.cern.ch/root

More stuff!

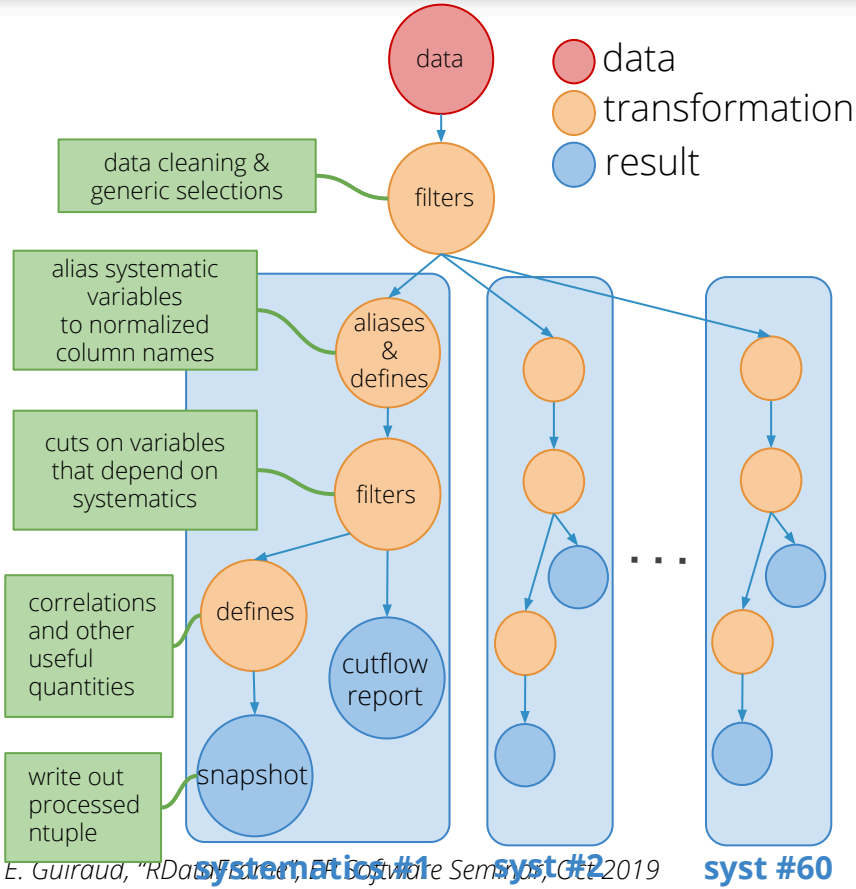
RDataFrame's parallelization scheme



Task-based parallelism

- ➔ each task processes a range of entries (thanks to inherent independence of HEP events)
- ➔ **cannot overcommit**, plays well with e.g. experiment frameworks
- ➔ range granularity is the same as TTree compression's to **avoid redundant decompressions**
- ➔ **Intel TBB** is currently ROOT's task scheduler and thread pool manager
- ➔ **RDF parallel *writing*** is also task-based, see "Writing ROOT Data in Parallel", CHEP 2018

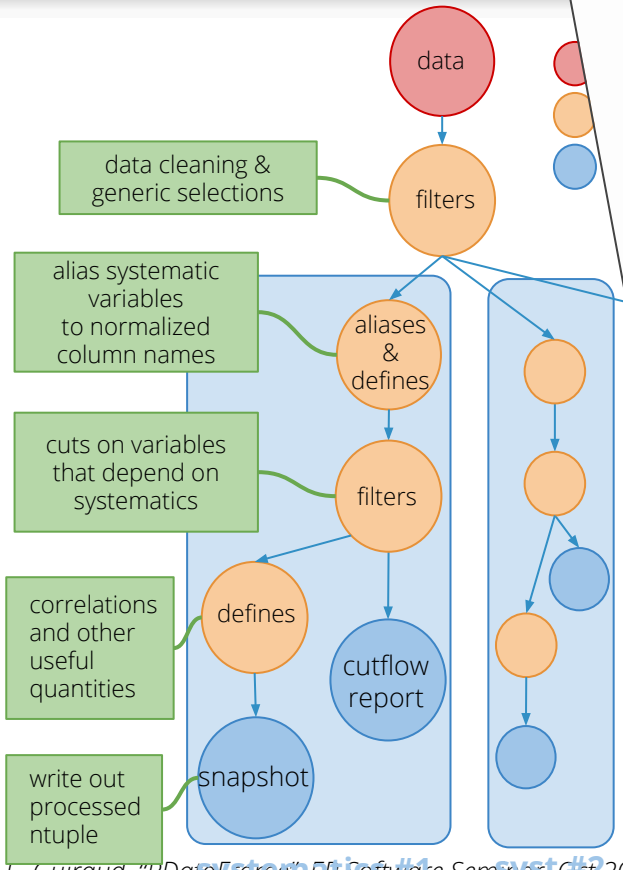
Case study: ATLAS SUSY ntuple → ntuple

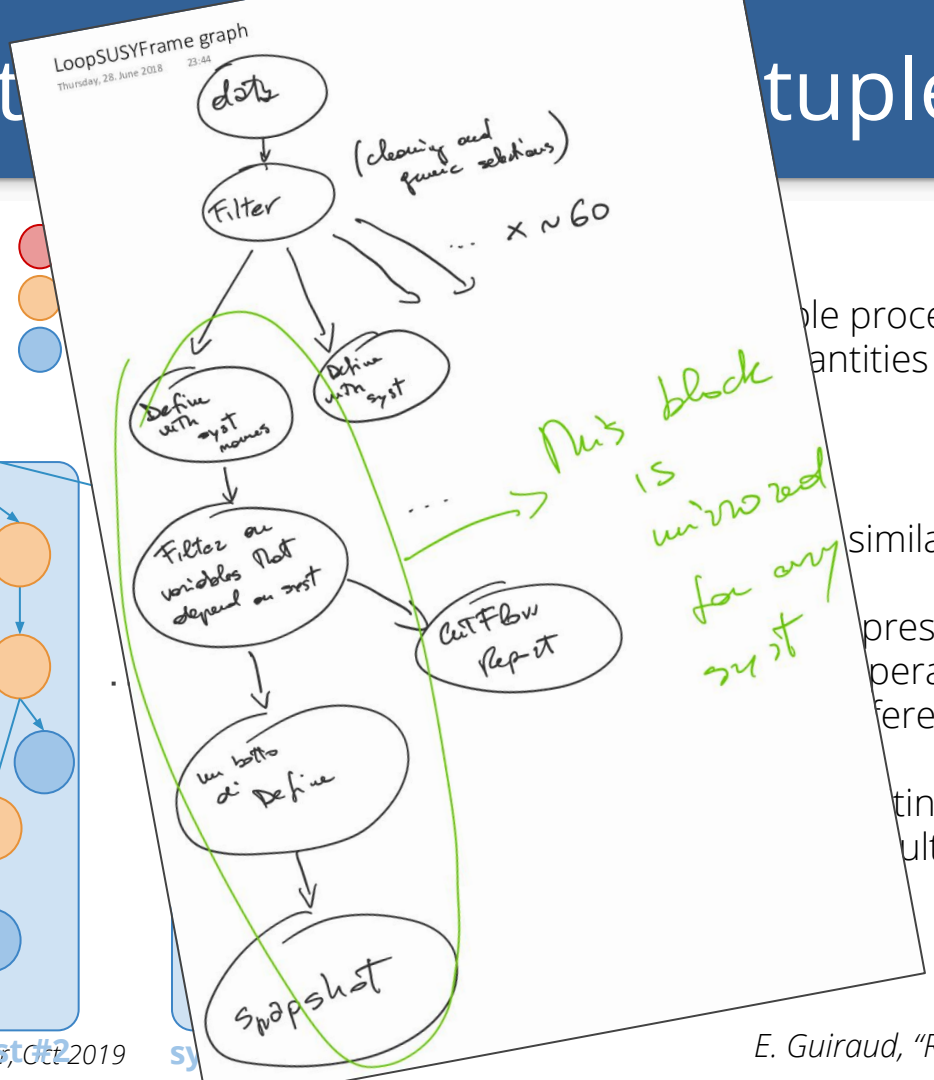


Local ntuple → ntuple processing, MC data is processed to add quantities relevant for publication

- ➔ program's **main** reads similarly to this graph
- ➔ the large blue boxes represent one single function that applies the same operations to an RDF variable and is re-used for all different systematics
- ➔ cuts, calculations and writing of the 60 output trees all happen in the same multi-thread event loop

Case st tuple → ntuple





le processing, MC data is
antities relevant for publication

similarly to this graph

present one single function
perations to an RDF variable
ferent systematics

ting of the 60 output trees
ulti-thread event loop

Collections, in-memory caching

Jitted C++ or PyROOT

```
auto inMemDF = d.Filter("All(event.muons.eta < 2.5)")
```

Collections, in-memory caching

Jitted C++ or PyROOT

```
auto inMemDF = d.Filter("All(event.muons.eta < 2.5)")
                .Cache({"event.muons.eta"});
```

Collections, in-memory caching

Jitted C++ or PyROOT

```
auto inMemDF = d.Filter("All(event.muons.eta < 2.5)")
                .Cache({"event.muons.eta"});
```

C++

```
auto cutEtas = [](RVec<float> etas) { return All(etas < 2.5); };
auto inMemDF = d.Filter(cutEtas, {"event.muons.eta"})
                .Cache<RVec<float>>({"event.muons.eta"});
```

RVec reference guide

Parallel event generation and processing

```
// The pythia generator: a “slot” corresponds to a thread
Pythia8::Pythia pythia[nSlots];

// The generator function
auto genFunc = [&](unsigned int slot) {
  return &pythias[slot].event;
};

ROOT::Experimental::TDataFrame tdf(nevents);
tdf.DefineSlot("event", genFunc)
    .Filter(...).Define(...)
    .Snapshot<Pythia8::Event*>("tree", "hardQCD.root", {"event"});
```

RDataFrame cheat sheet

Transformations apply modifications to the dataframe, return a new RDataFrame

Actions (next slide) produce results from a (possibly transformed) dataset

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

html cheat sheet

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.

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 the names of all the available columns of the dataset.
GetDefinedColumnNames	Get the names of all the defined columns
GetColumnType	Return the type of a given column as a string.
GetColumnTypeNamesList	Return the list of type names of columns in the dataset.
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.