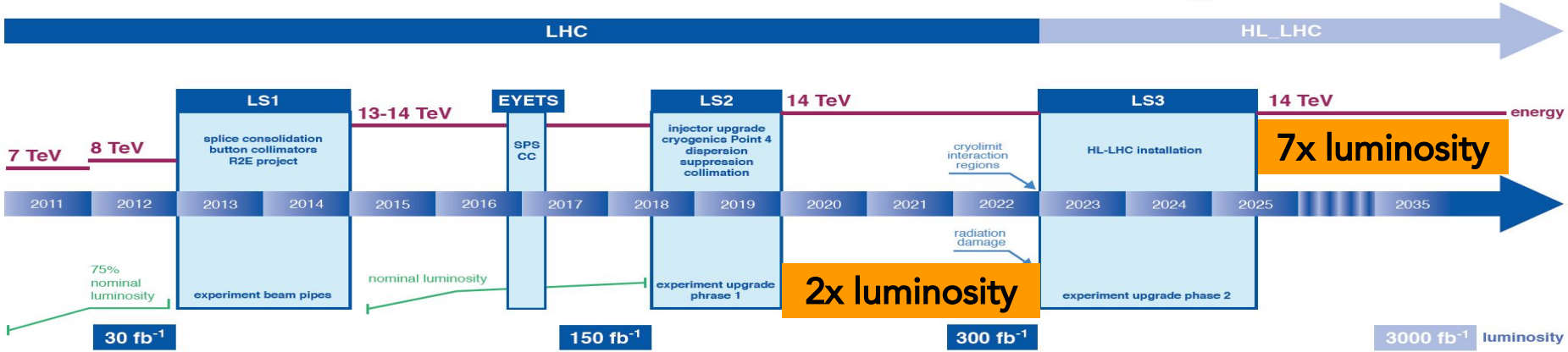
# RDataFrame

## easy parallel ROOT analysis at 100 threads

Enrico Guiraud for the ROOT team
CHEP 2018, Sofia, Bulgaria

# ROOT: a foundation library

- The amount of data produced by HEP experiments is going to increase drastically
  - e.g. at CERN: HL-LHC, FCC, ...
- ROOT's mission does not change:
  bring physicists from collision to publication as effectively as possible




source: http://acceleratingnews.web.cern.ch/content/recent-progress-hilumi-project-0

# 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
- ➔ allow to **transparently benefit from parallelism**

# 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
- ➔ allow to **transparently benefit from parallelism**

---

HEP is not alone in these challenges:
we can **learn from the data science industry**
and bring back what physicists need, in the form they need it

---

# 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
- ➔ allow to **transparently benefit from parallelism**

---

HEP is not alone in these challenges:
we can **learn from the data science industry**
and bring back what physicists need, in the form they need it

---

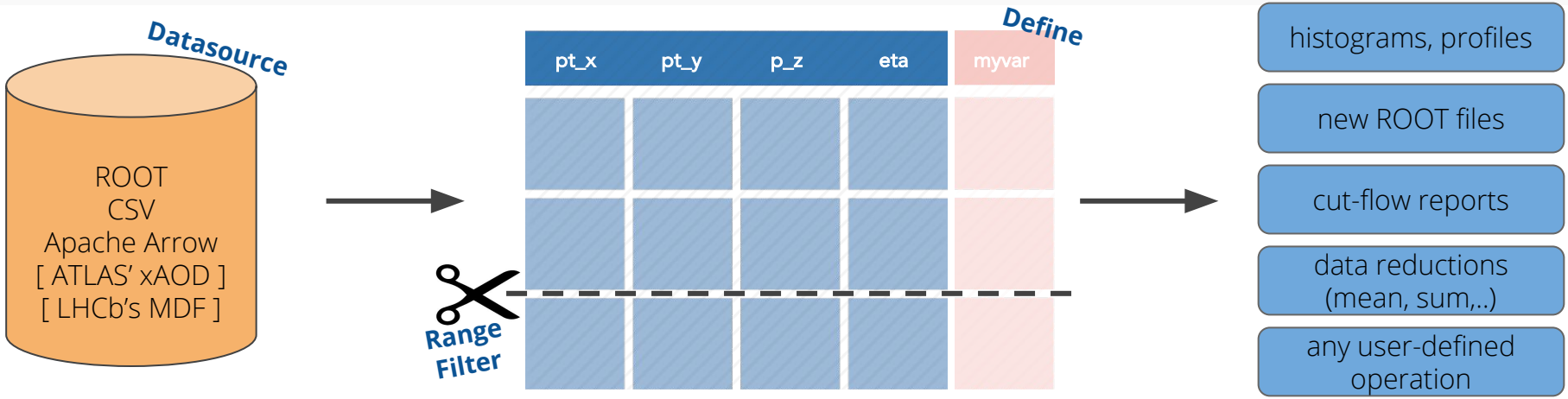
**RDataFrame**, officially part of ROOT since v6.14, tries to incarnate these ideas in the context of HEP analyses and HEP data manipulation



RDF user guide, cheat-sheet

DOI 10.5281/zenodo.260230

# RDataFrame design *goals*




- ➔ being the **fastest** way to manipulate HEP data
- ➔ being the **go-to ROOT analysis interface** from 1 to 100 cores, laptop to cluster
- ➔ full support for and consistent interfaces in both **Python and C++**

...employing elements of declarative and functional programming helped greatly

# An ergonomic, fast C++ dataframe

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

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

# An ergonomic, fast C++ dataframe

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

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

# An ergonomic, fast 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 = $x^2 + y^2$

# An ergonomic, fast C++ dataframe

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

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

# An ergonomic, fast 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

```
df2.Snapshot("newtree", "out.root");
```
write the skimmed data and r2 to a new ROOT file

# An ergonomic, fast C++ dataframe

```
ROOT::EnableImplicitMT();
```
Run a parallel analysis

```
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

```
df2.Snapshot("newtree", "out.root");
```
write the skimmed data and r2 to a new ROOT file

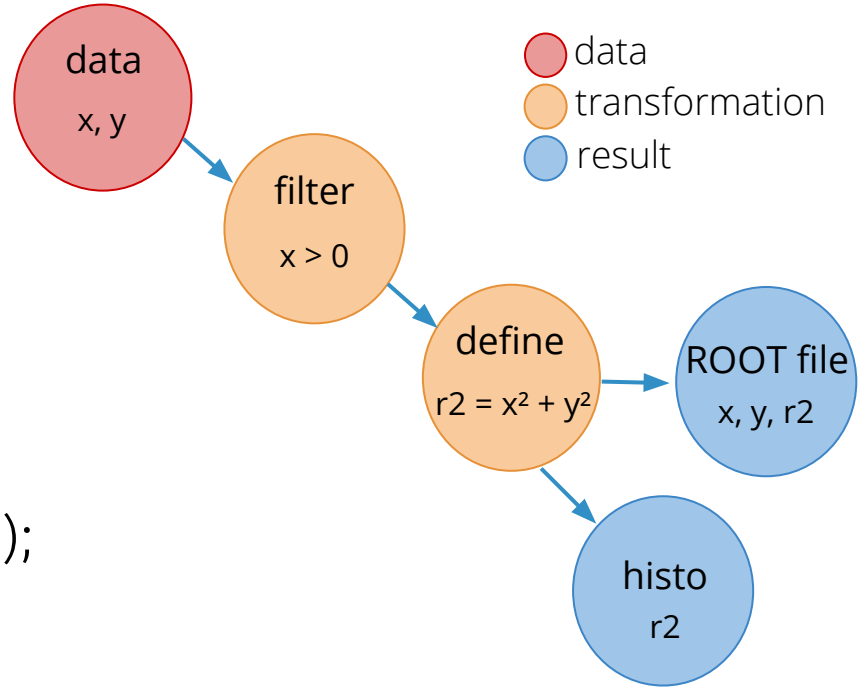
# An ergonomic, fast C++ dataframe

```
ROOT::EnableImplicitMT();                          // Run a parallel analysis
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 = x² + y²
auto rHist = df2.Histo1D("r2");                    // plot r2 for events that pass the cut
df2.Snapshot("newtree", "out.root");               // write the skimmed data and r2 to a new ROOT file
```

**Lazy execution** guarantees that all operations are performed in **one 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");
```



# No templates: C++ → JIT → Python

**C++**

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

---

# No templates: C++ → JIT → Python

**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");
```

# No templates: C++ → JIT → Python

**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")
```

# 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"});
```

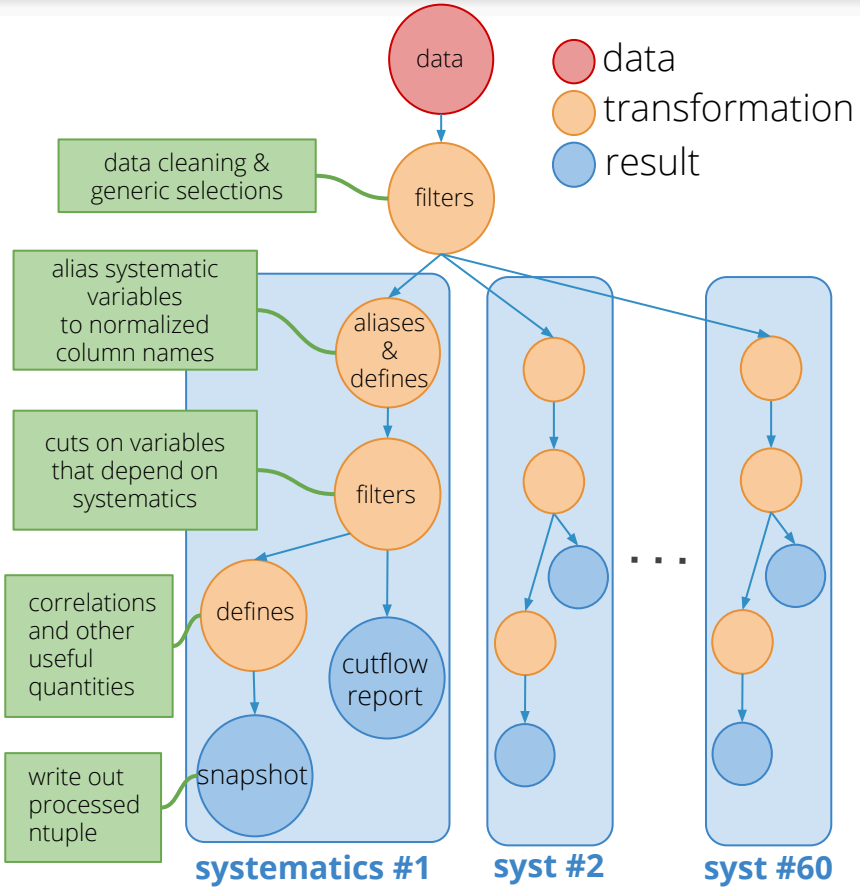
---

**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

# Case study: ATLAS SUSY ntuple → ntuple



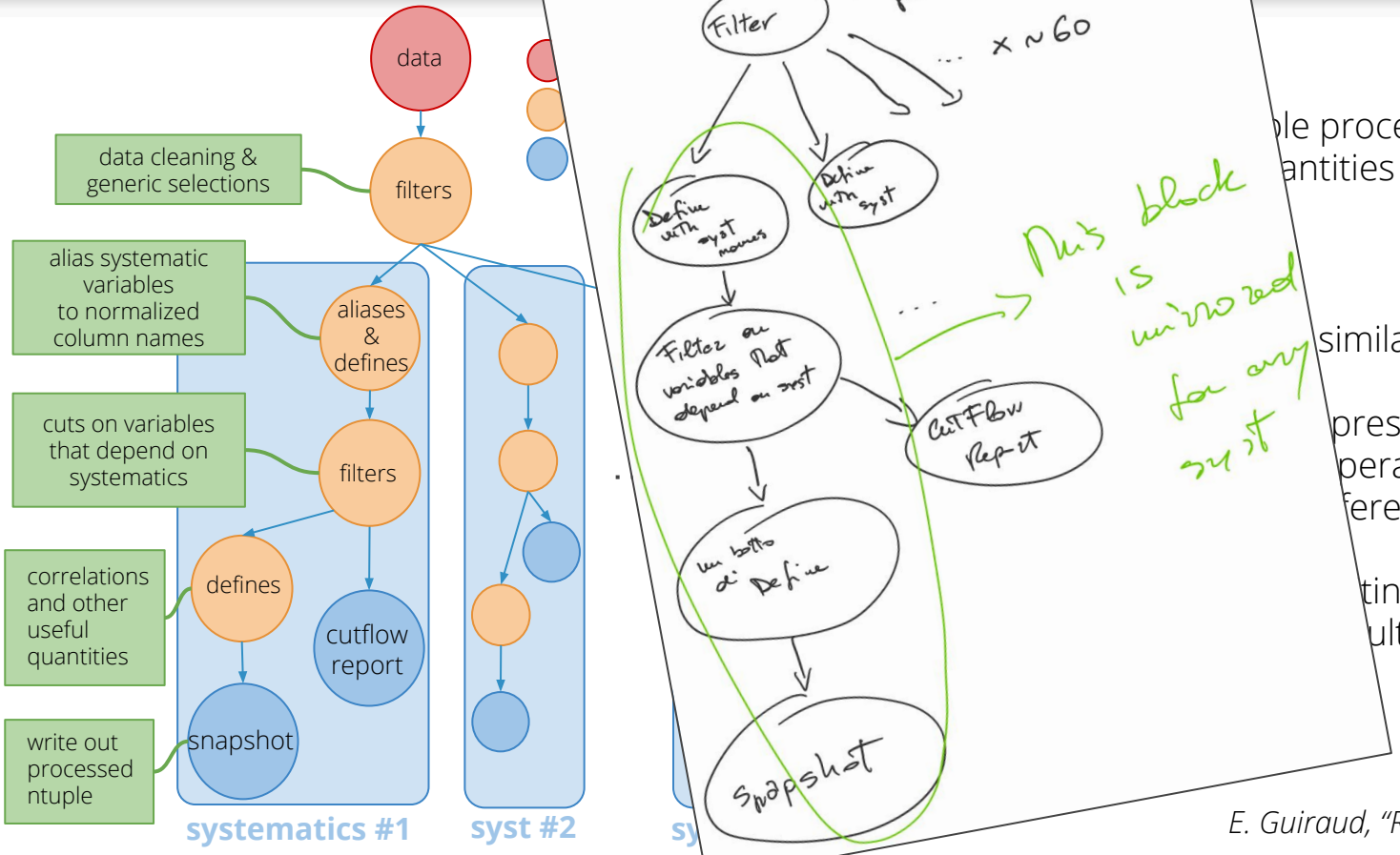

Local ntuple → ntuple processing, MC data is processed to add quantities relevant for publication

➔ 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

# 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 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

# 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 and Apache Arrow 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

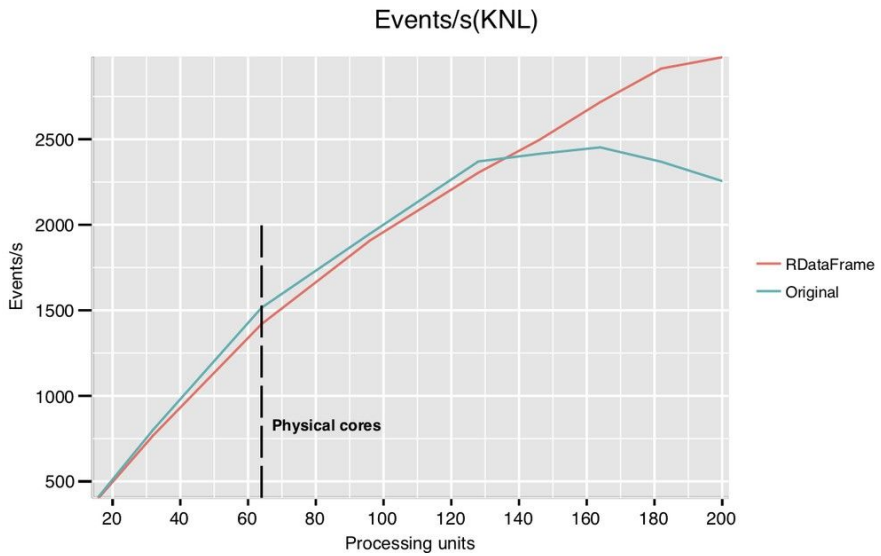
# Does it scale? Is it fast?

**No disk reads, KNL, 64 physical cores**

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

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

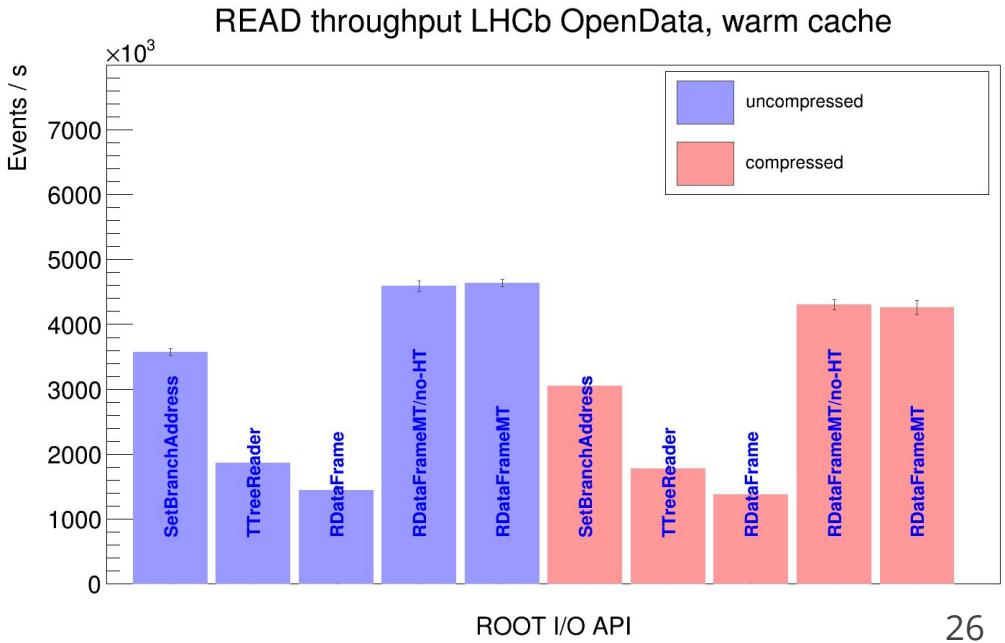
Performance analysis by X. Valls Pla




**Read speed on SSD, 4 physical cores @ 3.6GHz**

TTree+SetBranchAddress vs TTreeReader vs RDataFrame

Original results by J. Blomer




*E. Guiraud, "RDataFrame", CHEP 2018*

# Does it scale? Is it fast?

**No dis**

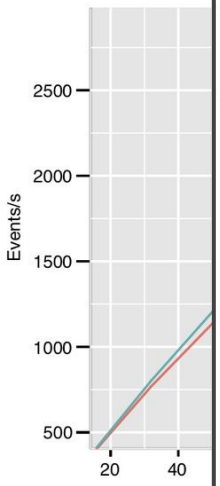
Monte Carlo C

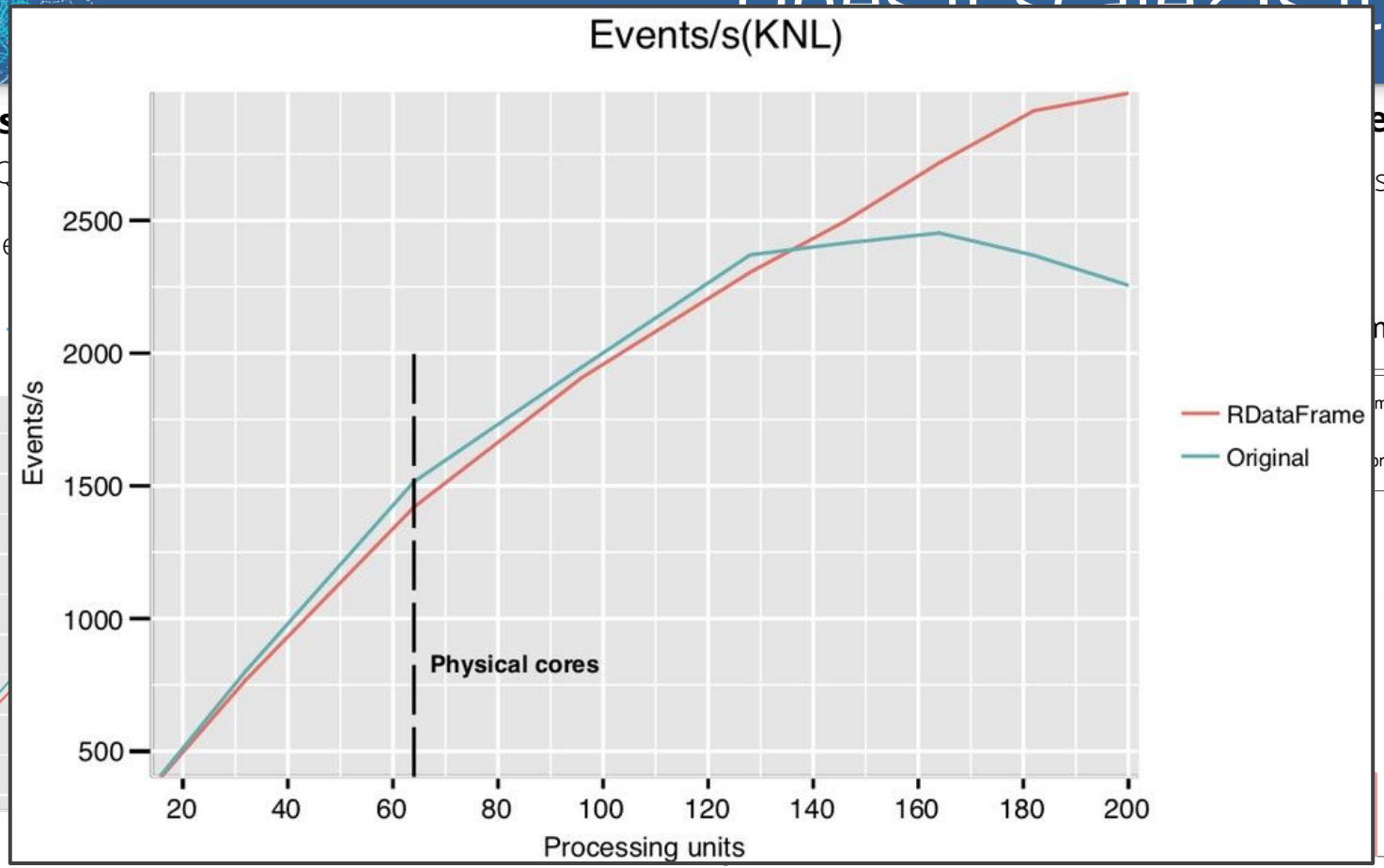
Ad-hoc impleme

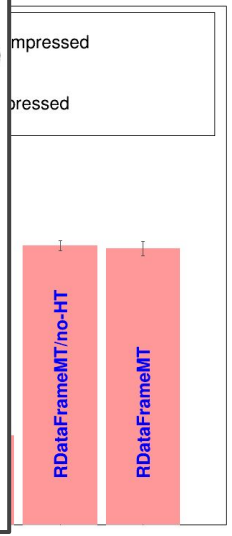
**es @ 3.6GHz**

s RDataFrame

n cache



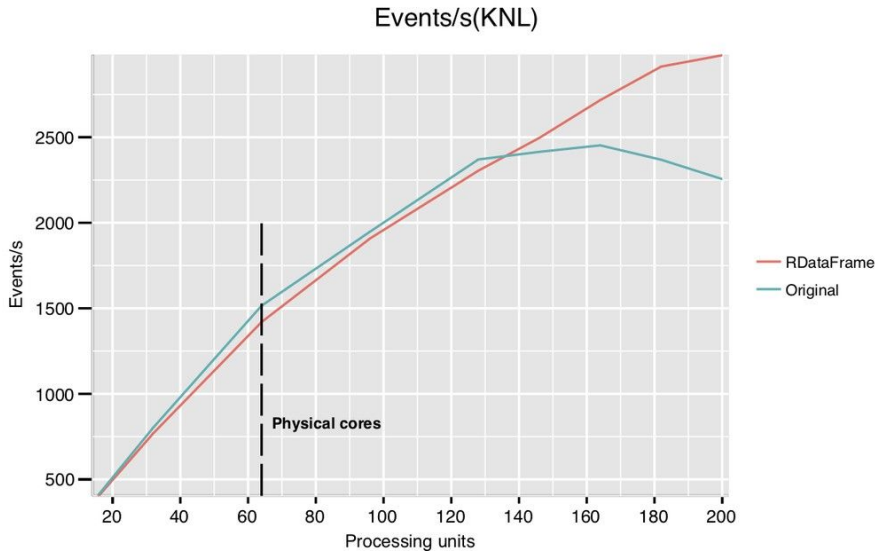
# Does it scale? Is it fast?

**No disk reads, KNL, 64 physical cores**

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

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

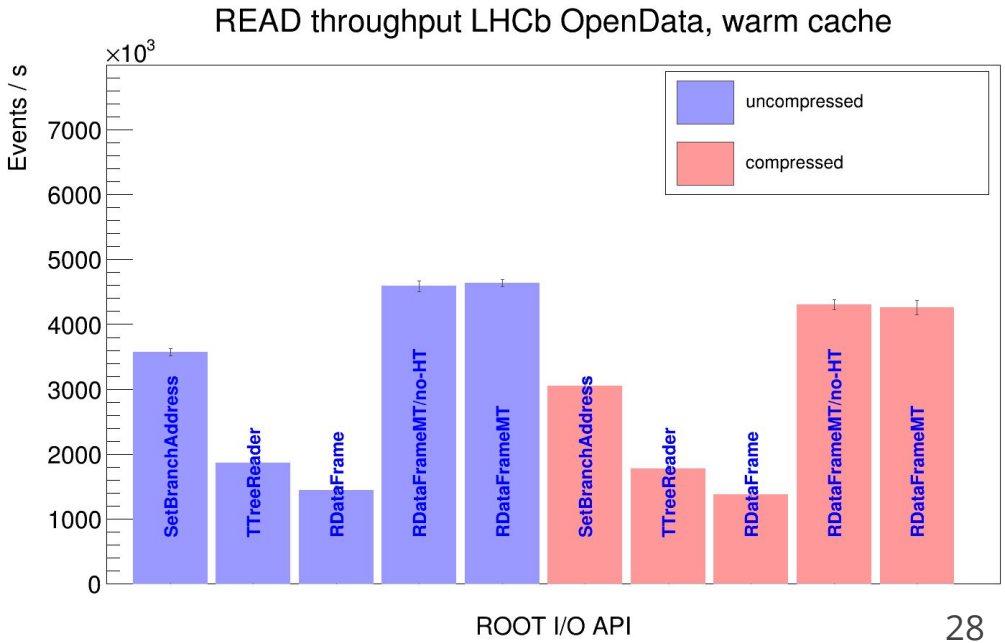
Performance analysis by X. Valls Pla



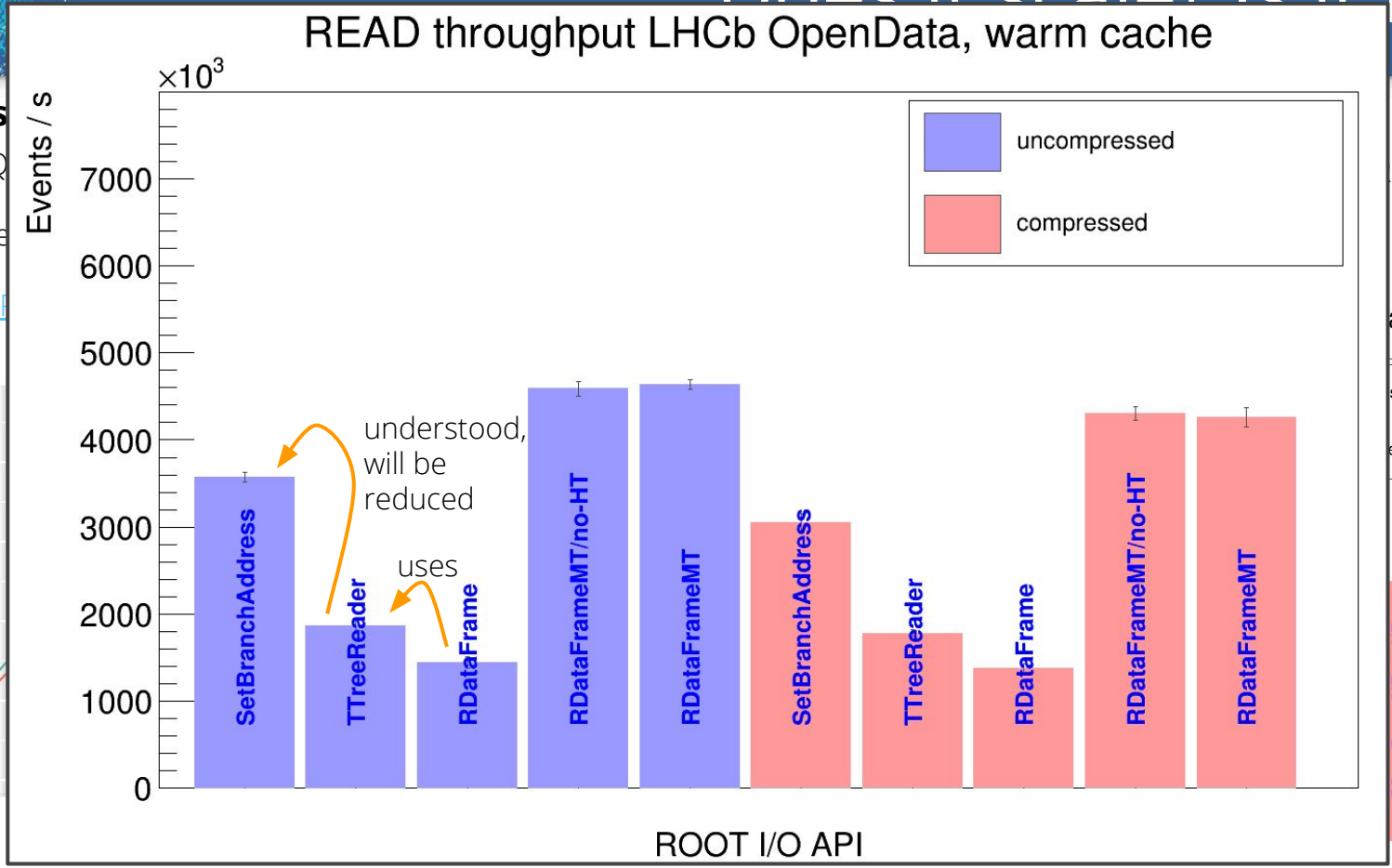

**Read speed on SSD, 4 physical cores @ 3.6GHz**

TTree+SetBranchAddress vs TTreeReader vs RDataFrame

Original results by J. Blomer



# Does it scale? Is it fast?




ROOT I/O API

# Summary, outlook

➔ ROOT provides a **modern, high-level, type-safe, parallel** interface for data analysis and manipulation

➔ **RDataFrame** is available since ROOT v6.14
- performant, scales to many-core architectures,
- has already been used successfully by physicists of major LHC experiments

# Summary, outlook

- ROOT provides a **modern, high-level, type-safe, parallel** interface for data analysis and manipulation

- **RDataFrame** is available since ROOT v6.14
  - performant, scales to many-core architectures,
  - has already been used successfully by physicists of major LHC experiments

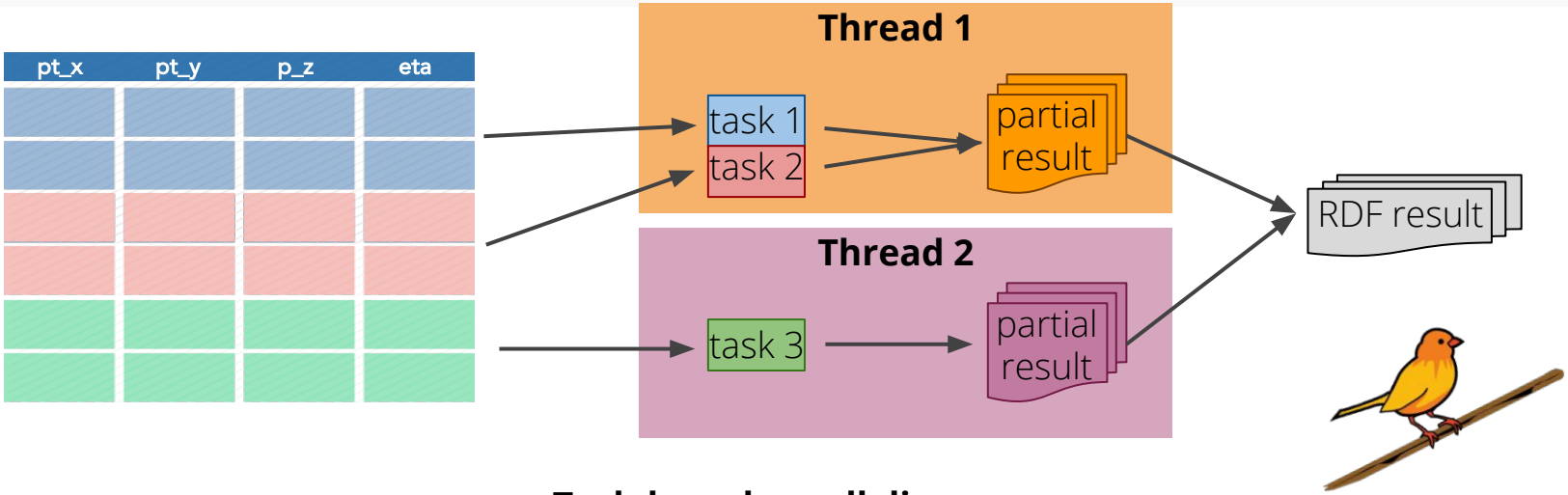
**For the future**

- more pythonic pyROOT bindings (conversion to/from numpy, python callables, …)
- **distributed execution of RDataFrame analyses**: working prototype for python+Spark
- integration with TMVA's inference layer
- low-level performance optimization

Birds of a feather session about "Present and Future of Data Analysis in 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 G. Amadio, "Writing ROOT Data in Parallel", Track 5

# 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 level functions*

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

. . . . . .

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

# 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. |
| Fill | Fill a user-defined object with the values of the specified branches, as if by calling `Obj.Fill(branch1, branch2, ...). |
| 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. |
| 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 all the available columns of the dataset |