Using data frames in Julia to analyse HEP data

Enrico Guiraud[1], **Philippe Gras**[2]

Nov 5-6, 23

[1]Princeton University, USA

[2]CEA/IRFU - Saclay, France

Introduction

Background

- Event-loop HEP/LHC analysis using Julia well established and runs fast.
- Data frame, and more generally columnar, style analysis now popular in HEP and standard in other fields. Convenient when using external tools, like machine learning libraries.
- Difficult to find one's way in the rich and fast evolving Julia columnar ecosystem.
- HEP has special needs. For Python and C++, dedicated libraries needed to be developed: RDataFrame, Akward arrays, Coffea.

Aim of this talk

- Establish guidelines together to implement columnar analysis in Julia;
- Identify possible needs of core package development.

Data frame introduction

Data frame introduced for S, an interpreted programming language for statistics

Statistical model in S (1992), Chambers, John et al.

You may better know S as R, one of its implementations

- Data frame = matrix representation of events, typ. in memory, whose each row represents an event and each column an observable (or a collection of observables)
- Columns are named and can be accessed as independent vectors

events	MET	pt	eta	phi	E
1	143	$[123, 32, 3..]$	$[1.5, 1.2, 0.3, ...]$	$[2.3, 1.0, 3.3, \ldots]$	$[289.3, 55.9, 3.1, \ldots]$
2	40.5	$[100, 62, 1..]$	$[3.2, 0.3, 1.0, \ldots]$	$[3.1, 0.1, 2.0, \ldots]$	$[1229., 64.9, 1.54, \ldots]$
$\vdots$			$\vdots$		

Data frame duality

A data frame can be viewed as table and as a collection of individual columns

A Data frame can be viewed as a table (2-D)

`df[1,1]`, `df[1,:]`

Columns can be accessed individually (1-D)

Access to the column MET `df.MET .> 100.` ⇒ Vector of `true`/`false` values (equivalent to `[x > 100 for x in df.MET]`)

Combining column and table accesses

`df[df.MET .> 100., :]` ⇒ selects events with MET $> 100.$

The Table interface

Common interface for table-like structures, `Table.jl`

- Easy to implement
- No inheritance, interface can be added to an existing type with the implementation of few methods
- Ease interoperability supported by 130+ packages[1]

- Row-access, Column-access, or both access
- Column named
- We will define data frame as a table with both row and column accesses (although column-access is strictly needed for a columnar analysis).

[1] https://github.com/JuliaData/Tables.jl file INTEGRATIONS.md

Examples of data type implementing the Table interface

- Dataframe from DataFrames.jl: the counterpart of Python Pandas written in Julia. Includes both type definition and operation tools.
- Vectors of NamedTuple and NamedTuple of AbtractVectors, respectively row- and column- access tables.
  Note: Tables.columntable() and Tables.rowtable() can convert any Table to these types.
- Tables.DictRowTable and Tables.DictColumnTable
- LazyTree from UnROOT: limited to ROOT file reading, manipulations limited as columns cannot be added.
- Table and FlexTable from TypedTables.jl. NamedTuple based.
- StructArray from StructArrays.jl: array of structs stored as structs of arrays.
- Arrow.Table from Arrow.jl.
- DTable from DTables: for distributed computing with Dagger (equivalent of Dask)

Columnar tools in Julia

Language feature

- Broadcasting eases work on columns.
- `filter` function.
- `[]` with advanced indices and `df.colname` notations.

For `DataFrame` from `DataFrames.jl`

- `DataFrames.jl` includes manipulation operations: `select`, `subset`, `transform`, `combine`.
- `DataFramesMeta.jl` provides convenient and concise notation.
- `DataFramesMacro.jl`, an alternative to `DataFramesMeta.jl`.

Columnar tools in Julia

For generic tables

- TableOperations.jl
- SplitApplyCombine.jl: various tools, not specific to `Tables`, in particular lazy operations (`filterview`, `mapview`).
- MappedArray.jl: lazy transformation of arrays to use for columns, an alternative to `SplitApplyCombine.mapview`.
- Query.jl: uses its own `Table` interface definition, but supports many existing Tables. Was showing poor performance in our tests.

Machine learning

- The Julia machine learning framework `MLJ` works with `Table` compatible data

Performance comparison of row selection tools

Benchmark

Select events with two opposite charge muons from `Run2012BC_DoubleMuParked_Muons-1Mevts.root`

Technique	Time copy (ms)	Time view (ms)	
Extended indices on `DataFrame`	30	7.2	
DataFrames `subset`	30	7.1	Same or similar perf.
DataFramesMeta `@(r)subset`	30	7.1	
Query.jl on `DataFrame`	130	-	
Query.jl on `Vector{NamedTuple}`	140	-	Slow
TableOperations on a `DataFrame`	350	-	
TableOperations on a `NamedTuple{Vector}`	35	-	
TableOperations on a `Vector{NamedTuple}`	42	-	
Event loop[1] on a `DataFrame`	140	-	
Event loop[1] on a `Vector{NamedTuple}`	5.0	-	

[1] Count selected events only

Choice of the tools

Recommendations

For columnar analyses, we recommend DataFrames used together with DataFramesMeta.

Room for improvement

- Default is `copycols=true` and `view=false`: less error prone, but not ideal for large datasets
- Code depends on the data frame type. A DataFramesMeta version that works on any `Table` compatible object would be ideal.

From one to many loops

Traditional HEP data processing: each event processed in a top-level loop.

Data frame way: each statement loops over events

Single vs many loops

Data frame pros

- Speed up processing for interpreted languages: essential for PYTHON, not relevant for JULIA.
- We can see the result after the execution of each statement ⇒ Nice for interactive use.
- Facilitate declarative programming style ⇒ more concise and legible code.
- Ease interface with non-HEP Machine Learning libraries, that typically use the columnar approach.

	Data frame	Single loop
Interpreted language	✓	
Interactive usage	✓	
ML tools	✓	
Legibility	✓	
Memory footprint		✓
Evolved algorithm		✓

Single-loop pros

- Memory efficient: needs only one event at a time (for I/O performance more are actually read and put in cache)
- Free the developer's mind of one dimension when designing an algorithm: deals with objects of one event instead of objects of every event.

Single-loop vs Data-Frame Performance

At first order

As Julia is as fast as the languages used for the underlying libraries, no speed gain from a columnar approach contrary to PYTHON

Looking closer

- $\oplus$ Use of smaller loop allows better SIMD (single instruction multiple data) optimization.
  - But inner loop is often over a collection of objects within an event.
- $\ominus$ Leads to more memory allocations.

$\Rightarrow$ For an average implementation single-loop approach likely to run faster.

Non-linear analysis

With default settings, columnar approach typically loads more data into RAM $\Rightarrow$ faster for an analysis that access several times the same event.

Limited to default settings, but psychologically important.

Expressiveness: data frame vs single loop

Selecting events with two muons:

Data frame

Declarative statement

```
df = df[df.nMuons .== 2, :]
```

or[1]

```
@subset! df :nMuons .== 2
```

Single loop

Typ. imperative statement

```
nMuons == 2 || return
```

With macros, declaration style is also possible!

```
@cut nMuons == 2
```

@cut macro definition:

```
macro cut(ex) :(\$(esc(ex)) || return false); end
```

[1] uses the DataFramesMeta package.

Expressiveness: collections

Selecting two muons of opposite charges

Broadcast not supported for `[]` $\Rightarrow$ use `getindex()`

```
df = df[df.nMuon .== 2 .&& getindex.(df.Muon_charge, 1) .! getindex.(df.Muon_charge, 2),:]
```

$\rightarrow$ Expressiveness lost

DataFramesMeta.jl **becomes handy:**

```
@rsubset!(df, :nMuon == 2 && :Muon_charge[1] != :Muon_charge[2])
```

$\uparrow$
r: by-row operation

DataFramesMeta.jl less error prone than extended index notation

Find the bug!

```
df[sum.(df_2mu.Muon_charge) .== 0),:]
```

DataFramesMeta.jl

DataFramesMeta.jl provides concise and efficient operations

```
@rsubset df sum(:Muon_charge)==0
```

`@(r)subset(!)`, `@(r)transform(!)`, `@(r)select(!)`, `@chain`, `@with`, etc.

DataFrameMeta.jl macros are based on functions from DataFrames.jl. They provide conciseness and efficiency.

Zipping columns

Often convenient to build objects from elements split over several columns

- E.g., p_{T}, η, ϕ stored as different columns in CMS NanoAODs.
- Can be performed like this:

```
@rselect df :Muon_p4=StructArray(pt=:Muon_pt, eta=:Muon_eta,
                                 phi=:Muon_phi, m=:Muon_mass) :Muon_charge
```

- Preserves columnar storage of components $\rightarrow$ optimal for SIMD.

Room for development

- A tool to parse columns of data frame and zip relevant ones based on name patterns.

Uncertainty propagation

- A HEP analysis is typically rerun several times, with each independent uncertainty source varied by $+1\,\sigma$ and -1 σ.
- ROOT RDataFrame provides a convenient tool to perform the variations in a optimal manner.
- Measurements.jl provide a tool for measurement uncertainties, but it does not support multiple uncertainty sources and uses a different approach for propagation (uses derivative and linear approximation)

Room for development

Equivalent of `RDataFrame::Vary()` would be very useful, either as a new package as part of Measurements.jl.

Data frames to process large amount of data

Two approaches for data sets that do not fit within the RAM

Most common approach (Python Dask, Julia Dagger/DTables)

- Data processed in chunks made of N events loaded in memory

ROOT RDataFrame "lazy" approach

- Operations recorded and postponed until the user access to the products.
- Data of 1 event $\pm$ cache loaded in memory at a time.
- On-demand load of all events supported.
    - Interesting for interactive analysis on reduced data sets

Lazy data frames in Julia

Currently available

Lazy operation on columns can be performed using `mappedarray()` from MappedArrays.jl or `mappedview()` from SplitCombineApply.jl.

Limitations or `mappedarrays` and `mappedview`

- Eager on views.
- Cannot be used for a lazy selection of rows of a columnar table.

$\Rightarrow$ Cannot replace `RDataFrame`.

Room for development

Implementation of a lazy data frame similar to `RDataFrame`.

Distributed computing

Distributed computing in Julia

Julia has a nice support for Distributed computing, including support for HTCondor:

- Built-in DISTRIBUTED module;
- DAGGER package: aims to provide similar functionnality as DASK or SPARK;

Need for investigations and documentation

- In evolution: JULIADB which was providing support for data that does not fit in memory is no more maintained and replaced by DTABLES, which is at early development: all table operations marked as experimental, no JULIADBMETA equivalent.
  - Our first attempts with DTABLES were not conclusive. Is it the right tool?
- Easy to waste time in trying different tools
- **Needs for a "How-to" to analyse HEP data sets, on local machine, on local cluster and on the LHC computing Grid.**

Example

Let's translate the COFFEA "processor" dimuon analysis example

Example: coffea version

```python
def process(self, events):
  dataset = events.metadata['dataset']
  muons = ak.zip({
      "pt": events.Muon_pt,
      "eta": events.Muon_eta,
      "phi": events.Muon_phi,
      "mass": events.Muon_mass,
      "charge": events.Muon_charge
      },
    with_name="PtEtaPhiMCandidate",
    behavior=candidate.behavior
  )

  h_mass = (hist.Hist.new
    .StrCat(["opposite", "same"], name="sign")
    .Log(1000, 0.2, 200., name="mass",
    ↪  label="$m_{\mu\mu}$ [GeV]")
    .Int64())
```

```python
cut = (ak.num(muons) == 2) &
↪  (ak.sum(muons.charge, axis=1) == 0)
# add first and second muon in every event
↪  together
dimuon = muons[cut][:, 0] + muons[cut][:, 1]
h_mass.fill(sign="opposite", mass=dimuon.mass)

cut = (ak.num(muons) == 2) &
↪  (ak.sum(muons.charge, axis=1) != 0)
dimuon = muons[cut][:, 0] + muons[cut][:, 1]
h_mass.fill(sign="same", mass=dimuon.mass)

return { dataset: {
    "entries": len(events),
    "mass": h_mass
  }
}
```

Example: Julia data frame version

```julia
using UnROOT, DataFrames, DataFramesMeta, LorentzVectorHEP, StructArrays, FHist
LogRange(xlow, xhigh, nbins) = 10 .^ range(log10(xlow), log10(xhigh), nbins);
P4 = StructArray{LorentzVectorCyl{Float64}};
```

```julia
function process(df)
    dataset = metadata(df, "dataset")

    #Keep two-muon events only
    df = @rsubset df :nMuon==2

    #Build momenta and opposite-sign flags
    @rselect!(df,
        :Muon_charge,
        :Muon_p4=P4(pt=:Muon_pt,eta=:Muon_eta,
            phi=:Muon_phi,mass=:Muon_mass),
        :Muon_OS=(:Muon_charge[1]
                  !=:Muon_charge[2]))

    #Compute dimuon mass
    @rtransform! df :DiMuon_mass=(:Muon_p4[1] +
    ↪   :Muon_p4[2]).mass

    #Fill histograms for OS and SS categories
    bins = LogRange(0.2, 200, 1000)
    hists = @by df :Muon_OS :dataset=dataset
    ↪   :DiMuon_hMass=fit(Histogram, :DiMuon_mass, bins)

    hists
end
```

Running the example

Julia code to run the process function

```
df = LazyTree(fname, "Events", sink=DataFrame)
metadata!(df, "dataset", "Run2012BC_DoubleMuParked", style=:note)
r =  map(process, Iterators.partition(df, 10_000)) # ← pmap for a distributed computation.
rr = @combine groupby(vcat(r...), [:Muon_OS, :dataset]) :DiMuon_hMass = merge(:DiMuon_hMass...)
```

Performance comparison

	Julia DF	Julia Loop	Coffea
Execution (t/t_{fastest})	33 s (1.2)	27 s (1)	158 s (5.9)
JIT compilation	+4.9 s	+2.2 s	–
Mem. allocation	40 GiB	19 GiB	–

Conclusions

- Established some guidelines for columnar analysis in Julia. Wish to complete them with your inputs.
- Proposing to take profit of this workshop to write a How-to on out-of-core distributed columnar analysis with Julia.
- Several development projects identified.
  - Column zipping helper;
  - Uncertainty propagation tool;
  - Lazy `DataFrame` similar as ROOT `RDataFrame`.

Backup slides

Choosing a data frame type i

Two kinds

- Type-stable: type of the data frame and row structs known at compiled time
- Type-unstable: type resolved at runtime

	Type-stable	Type-unstable
Performance once compiled	🙂	🙁
JIT compilation lags	🙁	🙂
Adding columns	❌	✅

Choosing a data frame type ii

Type instability penalty: relevant for row iterations

- For column operations, dynamic dispatch is amortized by the number of rows processed one function call ⇒ **typically small for columnar analysis**.

📝 A Type-instable table be turned when needed into a type-stable table with `Tables.columntable()` (copy-less operation).

Read Why DataFrame is not type stable and when it matters

Type stability penalty: relevant for wide tables

- Lags relevant for larger number of columns, and when manipulating the data frame (adding a column creates a new data frame types).

  E.g., 21 s to load the 1698 branches of a CMS NanoAOD into a LazyTree or Typed Table, 1420 s for the first `display()` method call with current Julia release (1.7 s and 153 s with 1.10.0-beta3).

- More relevant for interactive than batch mode.