Big Data Software in Particle Physics

Jim Pivarski

Princeton University – DIANA-HEP

August 2, 2018

What software should you use in your analysis?

Sometimes considered as a vacuous question, like "What color is your hammer?"

What software should you use in your analysis?

Sometimes considered as a vacuous question, like "What color is your hammer?"

Anything to get the job done!

What software should you use in your analysis?

Sometimes considered as a vacuous question, like "What color is your hammer?"

Anything to get the job done!

But there are real differences and sometimes it matters:

What software should you use in your analysis?

Sometimes considered as a vacuous question, like "What color is your hammer?"

Anything to get the job done!

But there are real differences and sometimes it matters:

- Tool for the wrong problem (hammer vs. screwdriver)

What software should you use in your analysis?

Sometimes considered as a vacuous question, like "What color is your hammer?"

Anything to get the job done!

But there are real differences and sometimes it matters:

- Tool for the wrong problem (hammer vs. screwdriver)
- Ease of use/data analyst productivity (ergonomics of handle)

What software should you use in your analysis?

Sometimes considered as a vacuous question, like "What color is your hammer?"

Anything to get the job done!

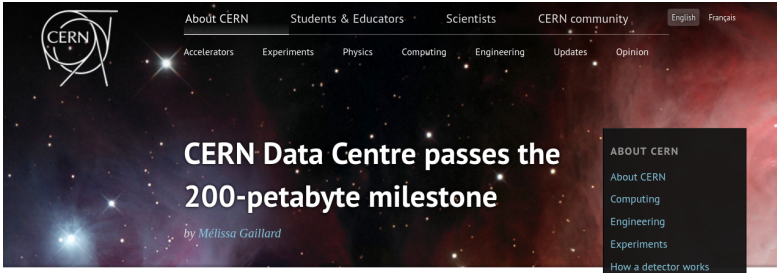
But there are real differences and sometimes it matters:

- Tool for the wrong problem (hammer vs. screwdriver)
- Ease of use/data analyst productivity (ergonomics of handle)
- Computational performance (head weight)

Why this matters now:

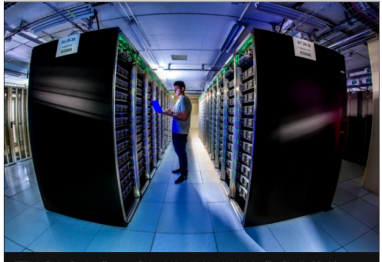
we're not the only ones analyzing big datasets anymore: "web scale analytics"

We measure globally distributed data in hundreds of PB

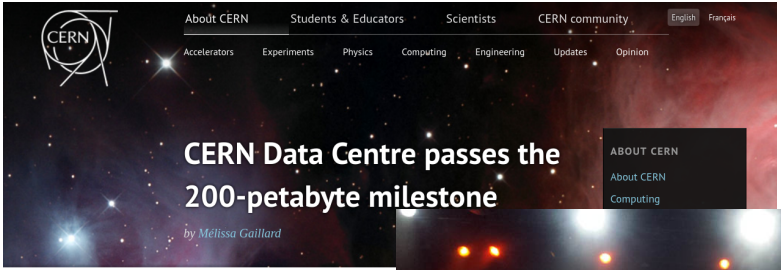


Posted by Stefania Pandolfi on 6 Jul 2017. Last updated 7 Jul 2017, 11.18.
Voir en français

This content is archived on the CERN Document Server


CERN's Data Centre (Image: Robert Hradil, Monika Majer/ProStudio22.ch)

But for "web scale" companies, 100 PB = 1 truck

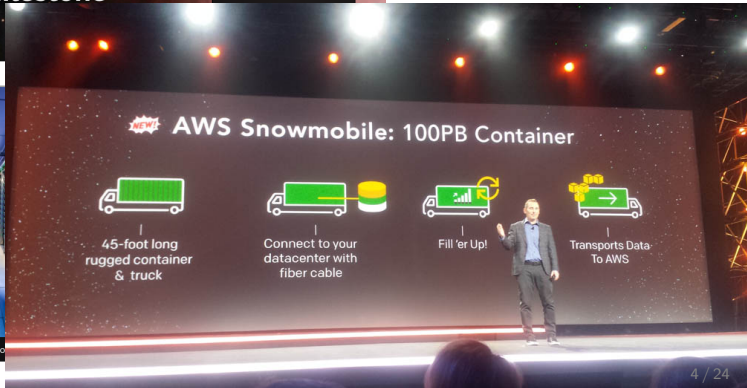


Posted by Stefania Pandolfi on 6 Jul 2017. Last updated 7 Jul 2017, 11.18.

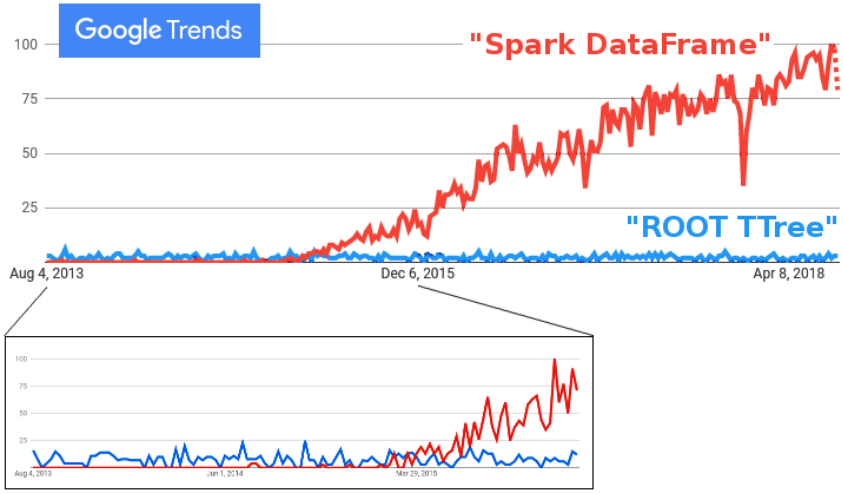
Voir en français

This content is archived on the CERN Document Server

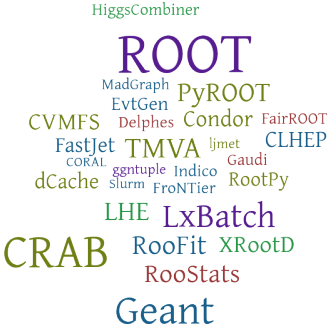
CERN's Data Centre (Image: Robert Hradil, Mo

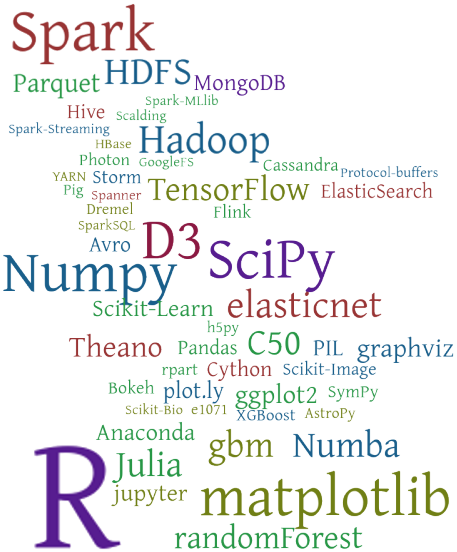


Their tools are new, but widely used (many eyes on the code)



Their ecosystem was mostly developed independently from ours





Ideally, we should use the strengths of both

What we can find off the shelf

- distributed processing, scale-out
- C++/Python interoperability
- special functions, matrix math
- fitting/minimization, integration, differentiation, interpolation
- symbolic algebra
- advanced statistics
- machine learning
- graphics, advanced plotting
- graphical interfaces, user workflows

What we still must develop in-house

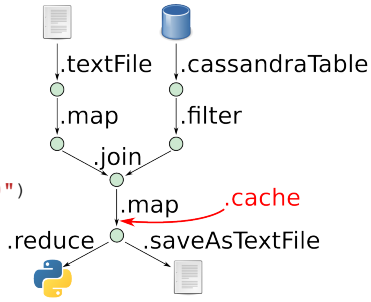
- reading/writing ROOT files
- collaboration frameworks, triggers, event reconstruction
- advanced histogramming and fitting
- efficient variable-length lists (our kind of non-relational data)
- domain-specific functions

Functional programming, managed data

http://www.cs.sfu.ca/CourseCentral/732/ggbaker/content/spark.html

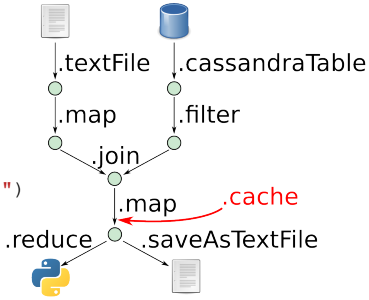
- Big, distributed dataset is a local variable:

```
lookup = spark.textFile("lookup.txt").map(int)
table  = spark.cassandraTable("T").filter("col > 0")
both   = table.join(lookup).where("n == N")
cached = both.map("x**2 + y**2").cache()
hist   = cached.reduce(histogram)
cached.saveAsTextFile("tmp.txt")
```



Functional programming, managed data

http://www.cs.sfu.ca/CourseCentral/732/ggbaker/content/spark.html



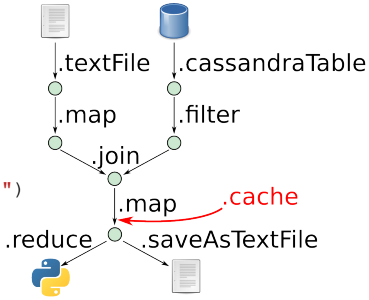
- Big, distributed dataset is a local variable:

```
lookup = spark.textFile("lookup.txt").map(int)
table  = spark.cassandraTable("T").filter("col > 0")
both   = table.join(lookup).where("n == N")
cached = both.map("x**2 + y**2").cache()
hist   = cached.reduce(histogram)
cached.saveAsTextFile("tmp.txt")
```

- Distributed job starts when you ask for its output

Functional programming, managed data

http://www.cs.sfu.ca/CourseCentral/732/ggbaker/content/spark.html



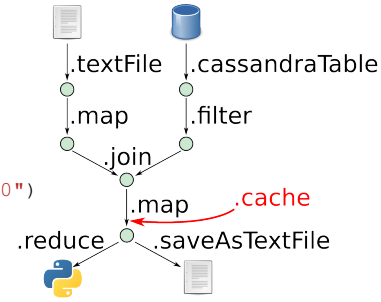
- Big, distributed dataset is a local variable:

```
lookup = spark.textFile("lookup.txt").map(int)
table  = spark.cassandraTable("T").filter("col > 0")
both   = table.join(lookup).where("n == N")
cached = both.map("x**2 + y**2").cache()
hist   = cached.reduce(histogram)
cached.saveAsTextFile("tmp.txt")
```

- Distributed job starts when you ask for its output
- Spark manages files, task dependencies, retry-on-failure

Functional programming, managed data

http://www.cs.sfu.ca/CourseCentral/732/ggbaker/content/spark.html



- Big, distributed dataset is a local variable:

```
lookup = spark.textFile("lookup.txt").map(int)
table  = spark.cassandraTable("T").filter("col > 0")
both   = table.join(lookup).where("n == N")
cached = both.map("x**2 + y**2").cache()
hist   = cached.reduce(histogram)
cached.saveAsTextFile("tmp.txt")
```

- Distributed job starts when you ask for its output
- Spark manages files, task dependencies, retry-on-failure
- Distributed dataset may be in RAM (fast), disk (big), or both (spill-over)

Accessing particle physics data in Spark

Spark-ROOT: presents a large set of ROOT files as a Spark DataFrame

```
mydata = (spark.read.format("org.dianahep.sparkroot")
                    .option("tree", "Events")
                    .load("many-files/*.root"))
```

https://github.com/diana-hep/spark-root

XRootD-HDFS: presents an XRootD service (e.g. EOS) as HDFS for Spark

```
...load("hdfs://eos.cern.ch/many-files/*.root")
```

https://github.com/opensciencegrid/xrootd-hdfs

Accessing particle physics data in Spark

Spark-ROOT: presents a large set of ROOT files as a Spark DataFrame

```
mydata = (spark.read.format("org.dianahep.sparkroot")
                    .option("tree", "Events")
                    .load("many-files/*.root"))
```

https://github.com/diana-hep/spark-root

XRootD-HDFS: presents an XRootD service (e.g. EOS) as HDFS for Spark

```
...load("hdfs://eos.cern.ch/many-files/*.root")
```

https://github.com/opensciencegrid/xrootd-hdfs

ROOT's new RDataFrame adds a Spark-like interface to local ROOT files; potential for driving Spark from ROOT in the future.

Spark runs on Java Virtual Machines (ubiquitous in business).

Spark runs on Java Virtual Machines (ubiquitous in business).

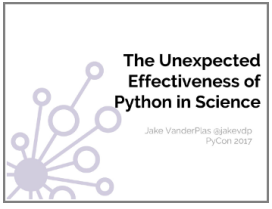
Hard to interface C++ (especially ROOT) with Java;
PySpark performance is limited by its Java-Python tunnel.

Spark runs on Java Virtual Machines (ubiquitous in business).

Hard to interface C++ (especially ROOT) with Java;
PySpark performance is limited by its Java-Python tunnel.

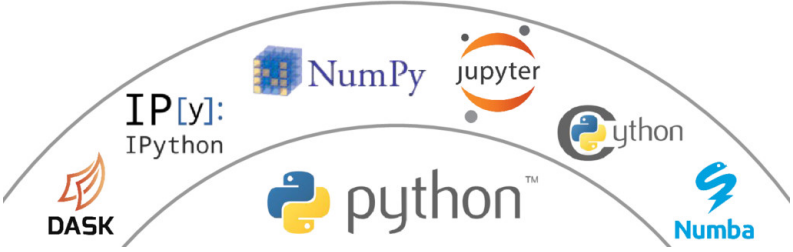
Python itself is much more open to interoperability with C++,
has similar distributed computing projects: Dask, Joblib, Parsl...

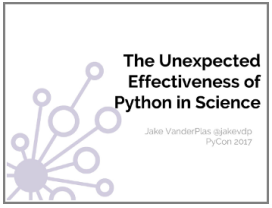
Python's Scientific Stack



Data analysis ecosystem has grown around Python

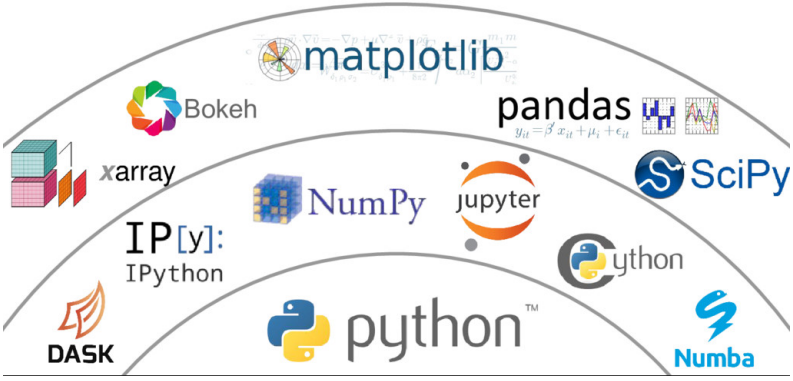
Python's Scientific Stack

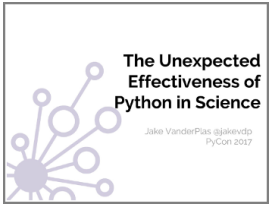




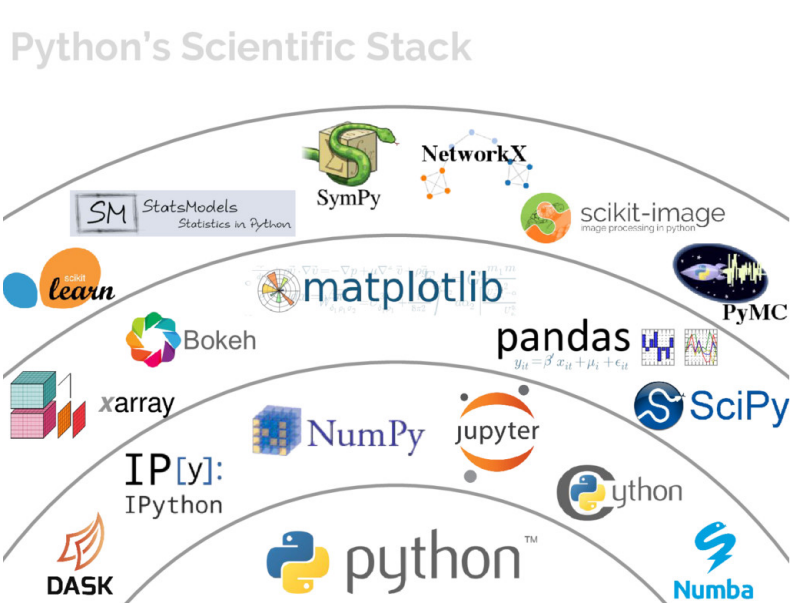
Data analysis ecosystem has grown around Python

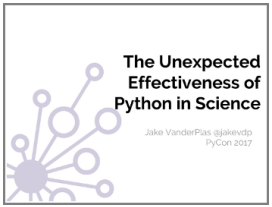
Python's Scientific Stack



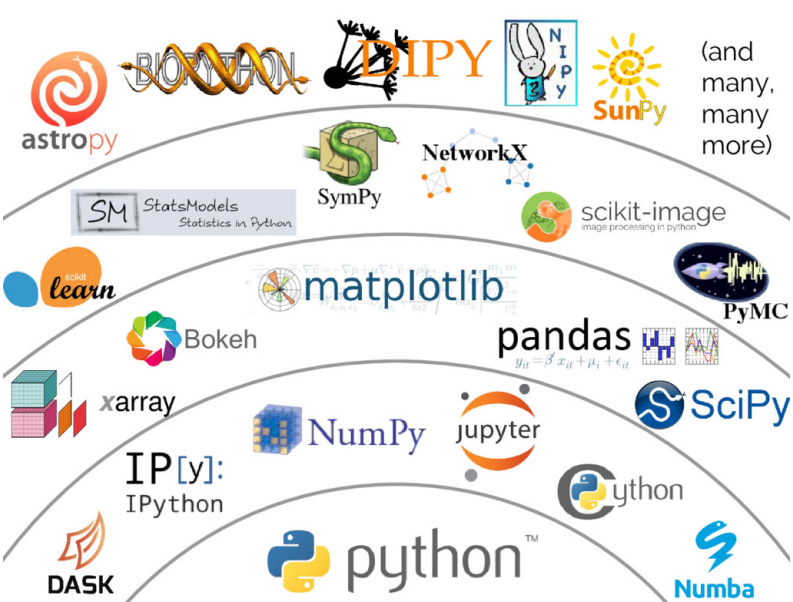


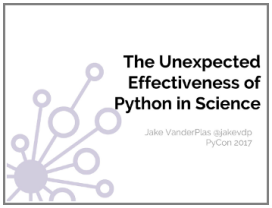
Data analysis ecosystem has grown around Python





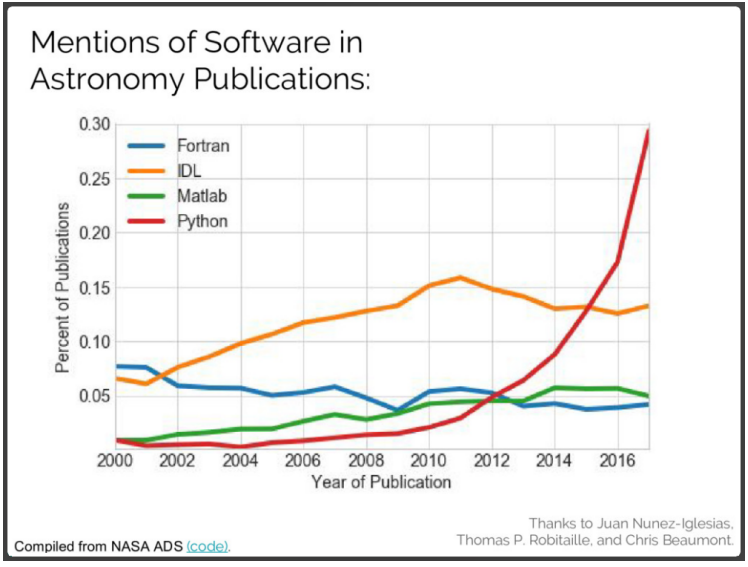
Data analysis ecosystem has grown around Python





Particularly machine learning

In some fields, adoption is astronomical. . .



Accessing ROOT data in Python

PyROOT: general solution, but slow if called in a loop over events; new ROOT features like `TTree::AsMatrix()` fill Numpy arrays directly, but only for primitive types.

Accessing ROOT data in Python

PyROOT: general solution, but slow if called in a loop over events; new ROOT features like `TTree::AsMatrix()` fill Numpy arrays directly, but only for primitive types.

root_numpy: good solution for many years, except slow for nested data (e.g. `std::vector<float>`) and compiles against one version of ROOT; changing ROOT version breaks binary compatibility.

Accessing ROOT data in Python

PyROOT: general solution, but slow if called in a loop over events; new ROOT features like `TTree::AsMatrix()` fill Numpy arrays directly, but only for primitive types.

root_numpy: good solution for many years, except slow for nested data (e.g. `std::vector<float>`) and compiles against one version of ROOT; changing ROOT version breaks binary compatibility.

uproot: reimplementation of ROOT I/O in Python+Numpy with good performance, skipping a step that is unnecessary for filling arrays:

bytes of ROOT file $\rightarrow$ C++ objects $\rightarrow$ Numpy arrays

Accessing ROOT data in Python

PyROOT: general solution, but slow if called in a loop over events; new ROOT features like `TTree::AsMatrix()` fill Numpy arrays directly, but only for primitive types.

root_numpy: good solution for many years, except slow for nested data (e.g. `std::vector<float>`) and compiles against one version of ROOT; changing ROOT version breaks binary compatibility.

uproot: reimplementation of ROOT I/O in Python+Numpy with good performance, skipping a step that is unnecessary for filling arrays:

bytes of ROOT file $\rightarrow$ ~~C++ objects~~ $\rightarrow$ Numpy arrays

Accessing ROOT data in Python

PyROOT: general solution, but slow if called in a loop over events; new ROOT features like `TTree::AsMatrix()` fill Numpy arrays directly, but only for primitive types.

root_numpy: good solution for many years, except slow for nested data (e.g. `std::vector<float>`) and compiles against one version of ROOT; changing ROOT version breaks binary compatibility.

uproot: reimplementation of ROOT I/O in Python+Numpy with good performance, skipping a step that is unnecessary for filling arrays:

bytes of ROOT file $\rightarrow$ ~~C++ objects~~ $\rightarrow$ Numpy arrays

(disclosure: I'm the author)

Numpy arrays and jagged arrays

Installs without ROOT (or any particular version of ROOT):

uproot

```
$ pip install uproot --user
```

Numpy arrays and jagged arrays

Installs without ROOT (or any particular version of ROOT):

uproot

```
$ pip install uproot --user
```

One-per-event values become Numpy arrays

```
>>> import uproot
>>> events = uproot.open("NanoAOD.root")["Events"]
>>> events.array("MET_pt")
array([17.244043, 42.480724, 29.10839 , ..., 12.228868, 44.115654,
       33.511974], dtype=float32)
```

Numpy arrays and jagged arrays

Installs without ROOT (or any particular version of ROOT):


```
$ pip install uproot --user
```

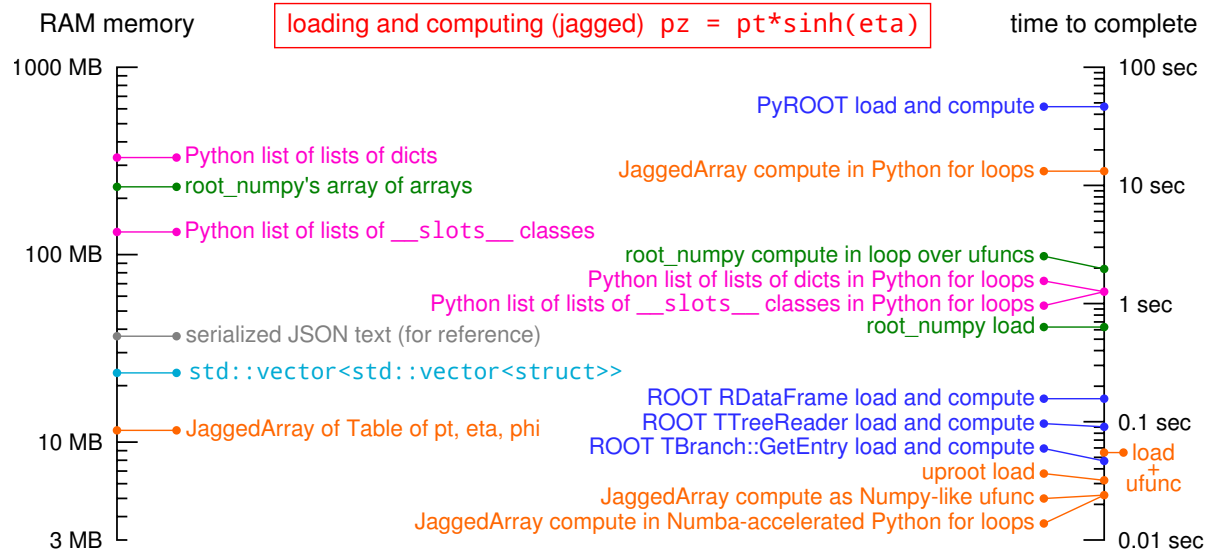
One-per-event values become Numpy arrays

```
>>> import uproot
>>> events = uproot.open("NanoAOD.root")["Events"]
>>> events.array("MET_pt")
array([17.244043, 42.480724, 29.10839 , ..., 12.228868, 44.115654,
       33.511974], dtype=float32)
```

Multi-per-event become "jagged arrays," variable-length sublists simulated by indexes

```
>>> events.array("Jet_pt")
jaggedarray([[41.46875  27.625    22.       18.734375],
             [23.75    23.64062 18.9687  ... 16.7812 16.2812 16.25  ],
             ...,
             [],
             [34.03125  18.828125 18.359375]])
```

Loading and computing columnar arrays is fast



Work in progress by summer students

- Writing data with uproot (TH* and hopefully TTree) — Pratyush Das (DIANA-HEP fellow)
- Jagged array algorithms (vectorized on GPU) — Jaydeep Nandi (Google Summer of Code)

Work in progress by summer students

- Writing data with uproot (TH* and hopefully TTree) — Pratyush Das (DIANA-HEP fellow)
- Jagged array algorithms (vectorized on GPU) — Jaydeep Nandi (Google Summer of Code)

Logical extensions of Numpy for non-flat data:

$$\underbrace{\texttt{metphi}}_{\text{flat}} - \underbrace{\texttt{jetphi}}_{\text{jagged}} \rightarrow \underbrace{\texttt{phidiff}}_{\text{jagged}}$$

Work in progress by summer students

- Writing data with uproot (TH* and hopefully TTree) — Pratyush Das (DIANA-HEP fellow)
- Jagged array algorithms (vectorized on GPU) — Jaydeep Nandi (Google Summer of Code)

Logical extensions of Numpy for non-flat data:

`metphi` − `jetphi` → `phidiff`

flat jagged jagged

`events`[`selection`] → `fewer_events`

jagged flat booleans jagged

Work in progress by summer students

- Writing data with uproot (TH* and hopefully TTree) — Pratyush Das (DIANA-HEP fellow)
- Jagged array algorithms (vectorized on GPU) — Jaydeep Nandi (Google Summer of Code)

Logical extensions of Numpy for non-flat data:

`metphi` − `jetphi` → `phidiff`

(`metphi`: flat; `jetphi`: jagged; `phidiff`: jagged)

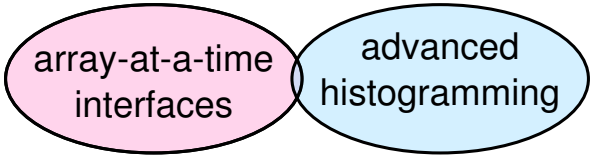
`events`[`selection`] → `fewer_events`

(`events`: jagged; `selection`: flat booleans; `fewer_events`: jagged)

`particle_attrs`.`max()` → `event_attrs`

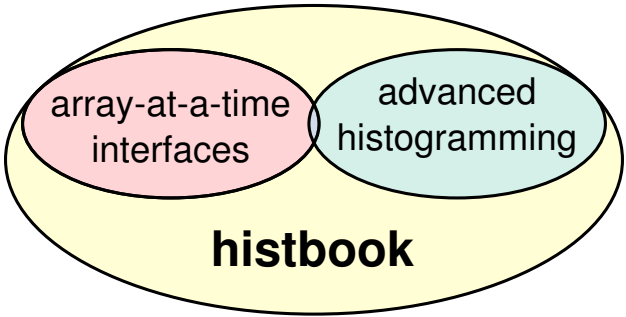
(`particle_attrs`: jagged; `max()`: reducer; `event_attrs`: flat)

Nearly all histogramming packages designed for Numpy arrays are feature-poor (from a particle physicist's perspective).



Access bin values, errors, weighted events, profile plots, efficiencies...

Nearly all histogramming packages designed for Numpy arrays are feature-poor (from a particle physicist's perspective).



Access bin values, errors, weighted events, profile plots, efficiencies...

Histogramming in histbook, plotting in Vega-Lite

Another two Python+Numpy packages (again, I'm the author)

```
$ pip install histbook vegascope --user
```

histbook Vega Scope

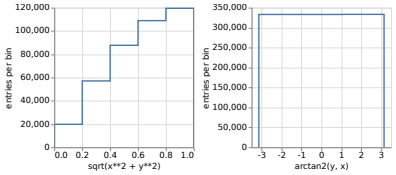
Histogramming in histbook, plotting in Vega-Lite

Another two Python+Numpy packages (again, I'm the author)

```
$ pip install histbook vegascope --user
```

All histograms are N-dimensional and expressions to compute are next to their binning.

```
>>> from histbook import *
>>> hist = Hist(
...     bin("sqrt(x**2 + y**2)", 5, 0, 1),
...     bin("arctan2(y, x)", 3, -pi, pi))
>>> hist.fill(dataframe)
>>> beside(hist.step("sqrt(y**2 + x**2)"),
...        hist.step("arctan2(y,x)"))
```



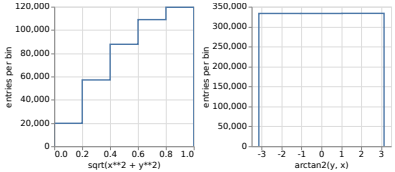
Histogramming in histbook, plotting in Vega-Lite

Another two Python+Numpy packages (again, I'm the author)

```
$ pip install histbook vegascope --user
```

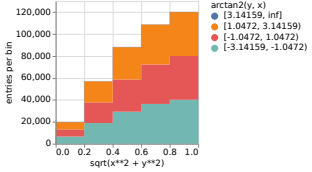
All histograms are N-dimensional and expressions to compute are next to their binning.

```
>>> from histbook import *
>>> hist = Hist(
...     bin("sqrt(x**2 + y**2)", 5, 0, 1),
...     bin("arctan2(y, x)", 3, -pi, pi))
>>> hist.fill(dataframe)
>>> beside(hist.step("sqrt(y**2 + x**2)"),
...        hist.step("arctan2(y,x)"))
```



Projections, rebinning, binwise selections, can all be performed after filling.

```
>>> (hist.select("arctan2(y, x) >= -pi")
...       .stack("arctan2(y, x)")
...       .area("sqrt(x**2+y**2)"))
```



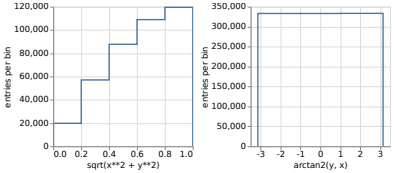
Histogramming in histbook, plotting in Vega-Lite

Another two Python+Numpy packages (again, I'm the author)

```
$ pip install histbook vegascope --user
```

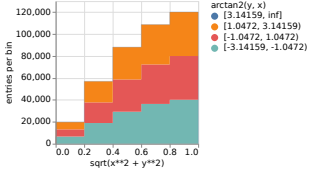
All histograms are N-dimensional and expressions to compute are next to their binning.

```
>>> from histbook import *
>>> hist = Hist(
...     bin("sqrt(x**2 + y**2)", 5, 0, 1),
...     bin("arctan2(y, x)", 3, -pi, pi))
>>> hist.fill(dataframe)
>>> beside(hist.step("sqrt(y**2 + x**2)"),
...        hist.step("arctan2(y,x)"))
```



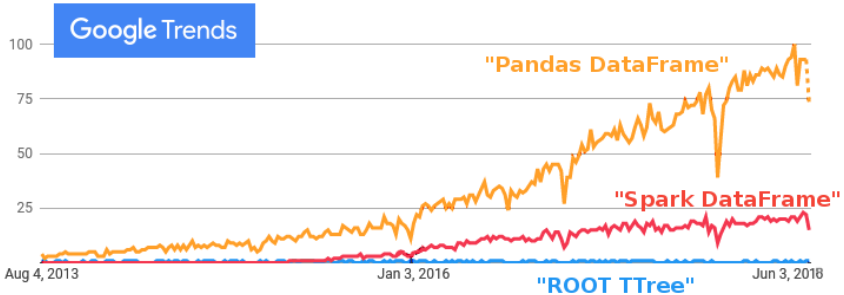
Projections, rebinning, binwise selections, can all be performed after filling.

```
>>> (hist.select("arctan2(y, x) >= -pi")
...       .stack("arctan2(y, x)")
...       .area("sqrt(x**2+y**2)"))
```



Plots appear inline in JupyterLab or can be viewed with VegaScope, a TCanvas-clone for Vega-Lite plots.

Pandas is a bigger thing than Spark



pandas

$$y_i t = \beta' x_{it} + \mu_i + \epsilon_{it}$$

Pandas is an in-memory table with advanced indexing

Our in-memory, indexed data are histograms. Multidimensional binning is a multi-tiered index, which has the same structure whether dense or sparse.

```
hist.pandas()
```

		count()	err(count())
sqrt(x**2 + y**2)	arctan2(y, x)		
[-inf, 0.0)	[-inf, -3.14159265359)	0.0	0.000000
	[-3.14159265359, -1.0471975512)	0.0	0.000000
	[-1.0471975512, 1.0471975512)	0.0	0.000000
	[1.0471975512, 3.14159265359)	0.0	0.000000
	[3.14159265359, inf)	0.0	0.000000
	{NaN}	0.0	0.000000
[0.0, 0.2)	[-inf, -3.14159265359)	0.0	0.000000
	[-3.14159265359, -1.0471975512)	6630.0	81.424812
	[-1.0471975512, 1.0471975512)	6757.0	82.200973
	[1.0471975512, 3.14159265359)	6517.0	80.727938
	[3.14159265359, inf)	0.0	0.000000
	{NaN}	0.0	0.000000
[0.2, 0.4)	[-inf, -3.14159265359)	0.0	0.000000
	[-3.14159265359, -1.0471975512)	19255.0	138.762387
	[-1.0471975512, 1.0471975512)	18953.0	137.669895
	[1.0471975512, 3.14159265359)	19061.0	138.061580
	[3.14159265359, inf)	0.0	0.000000
	{NaN}	0.0	0.000000

```
df = hist.pandas()
df[df["count()"] != 0]
```

		count()	err(count())
sqrt(x**2 + y**2)	arctan2(y, x)		
[0.0, 0.2)	[-3.14159265359, -1.0471975512)	6630.0	81.424812
	[-1.0471975512, 1.0471975512)	6757.0	82.200973
	[1.0471975512, 3.14159265359)	6517.0	80.727938
[0.2, 0.4)	[-3.14159265359, -1.0471975512)	19255.0	138.762387
	[-1.0471975512, 1.0471975512)	18953.0	137.669895
	[1.0471975512, 3.14159265359)	19061.0	138.061580
[0.4, 0.6)	[-3.14159265359, -1.0471975512)	29200.0	170.880075
	[-1.0471975512, 1.0471975512)	29332.0	171.265875
	[1.0471975512, 3.14159265359)	29545.0	171.886591
[0.6, 0.8)	[-3.14159265359, -1.0471975512)	36385.0	190.748526
	[-1.0471975512, 1.0471975512)	36615.0	191.350464
	[1.0471975512, 3.14159265359)	36130.0	190.078931
[0.8, 1.0)	[-3.14159265359, -1.0471975512)	39716.0	199.288735
	[-1.0471975512, 1.0471975512)	39987.0	199.967497
	[1.0471975512, 3.14159265359)	39975.0	199.937490
[1.0, inf)	[-3.14159265359, -1.0471975512)	201645.0	449.048995
	[-1.0471975512, 1.0471975512)	202372.0	449.857755

Multi-tier indexing can also be used to represent jaggedness

```
import uproot
events = uproot.open("NanoAOD.root")["Events"]
```

```
df = events.pandas.df(["MET_*", "Muon_*"])
df
```

		MET_pt	MET_phi	Muon_pt	Muon_eta	Muon_phi	Muon_dxy	Muon_dz	Muon_charge	...	Muon_pdgId
entry	subentry										
0	0	17.244043	1.035400	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN
1	0	42.480724	0.128357	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN
2	0	29.108391	0.932129	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN
3	0	29.060146	1.775635	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN
4	0	39.994896	-1.418701	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN
5	0	47.382843	2.964844	4.662842	0.839111	1.189697	-0.001610	0.994629	-1.0	...	13.0
	1	NaN	NaN	3.446648	-2.194824	-2.062500	-0.011620	-2.531250	-1.0	...	13.0
6	0	15.256557	1.333740	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN
7	0	17.382278	-2.448730	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN
8	0	25.201349	-1.796387	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN

Multi-tier indexing can also be used to represent jaggedness

```
df.pivot_table(index="entry", columns="subentry")
```

	MET_phi	MET_pt	Muon_charge			Muon_dxy			Muon_dz			Muon_eta	
subentry	0	0	0	1	2	0	1	2	0	1	2	0	1
entry													
0	1.035400	17.244043	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	0.128357	42.480724	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	0.932129	29.108391	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	1.775635	29.060146	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	-1.418701	39.994896	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
5	2.964844	47.382843	-1.0	-1.0	NaN	-0.001610	-0.011620	NaN	0.994629	-2.531250	NaN	0.839111	-2.194824
6	1.333740	15.256557	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
7	-2.448730	17.382278	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
8	-1.796387	25.201349	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
9	1.400391	24.369278	-1.0	NaN	NaN	-0.004219	NaN	NaN	0.025299	NaN	NaN	-1.405762	NaN
10	-0.028637	37.702976	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
11	-3.036133	17.890409	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Collecting Pythonic HEP packages under one roof

Scikit-HEP is a GitHub organization to build a Pythonic HEP ecosystem: `scikit-hep.org`

- uproot, histbook, vegascope
- root_numpy, root_pandas
- numpythia: bindings from Pythia to Numpy
- pyjet: bindings from FASTJET to Numpy
- formulate: convert between `TTree::Draw` syntax and numexpr (for histbook)
- decaylanguage: express complex B decay chains

Onboarding or in development...

- hepvector: LorentzVector operations as Numpy ufuncs
- root_ufunc: use any ROOT function as a Numpy ufunc

Conclusions

The world beyond particle physics has a lot of useful software, but it's not a perfect match to our needs.

To gain the benefits without giving up our own tools, we're developing software to connect the two worlds.