Bridging the Particle Physics and Big Data Worlds

Jim Pivarski

Princeton University – DIANA-HEP

October 25, 2017

Particle physics: the original Big Data

For decades, our computing needs were unique:

- large datasets (too big for one computer: a moving definition!),
- complex structure (nested data, web of relationships within each event),
- has to be reduced (aggregated, by histogramming, usually)
- to be modeled (fitting to extract physics results).

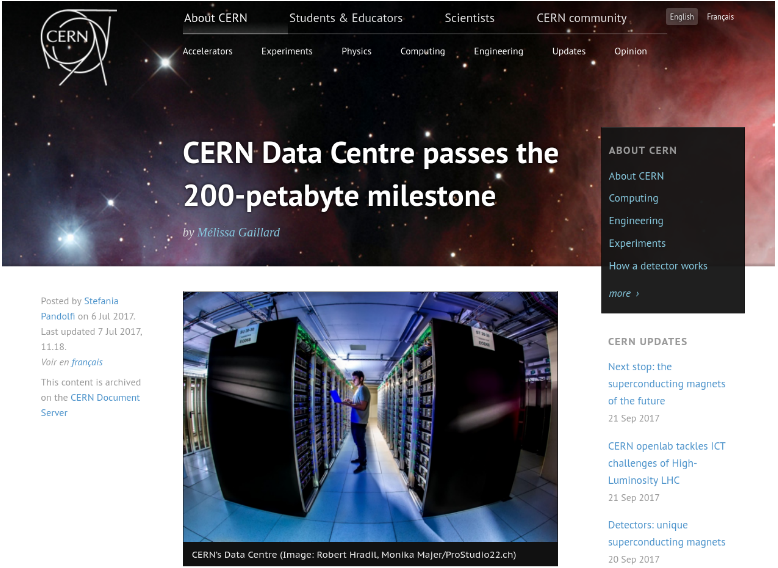
Particle physics: the original Big Data

For decades, our computing needs were unique:

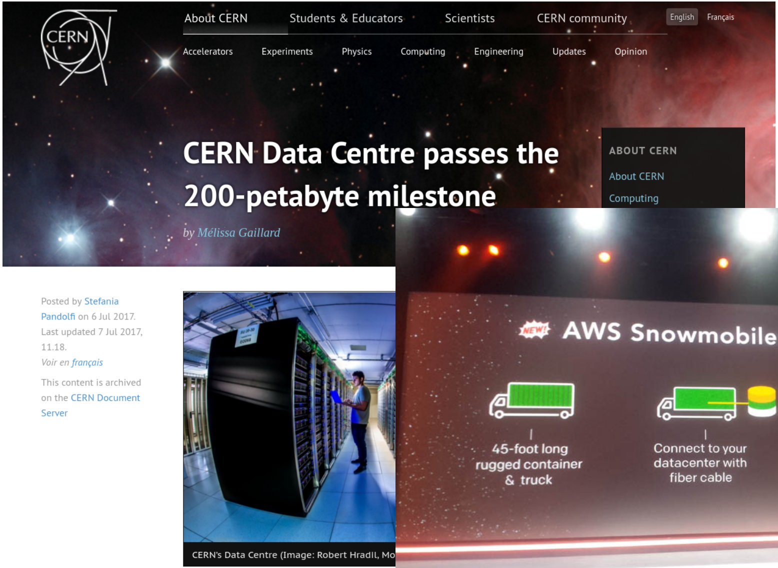
▸ large datasets	(too big for one computer: a moving definition!),
▸ complex structure	(nested data, web of relationships within each event),
▸ has to be reduced	(aggregated, by histogramming, usually)
▸ to be modeled	(fitting to extract physics results).

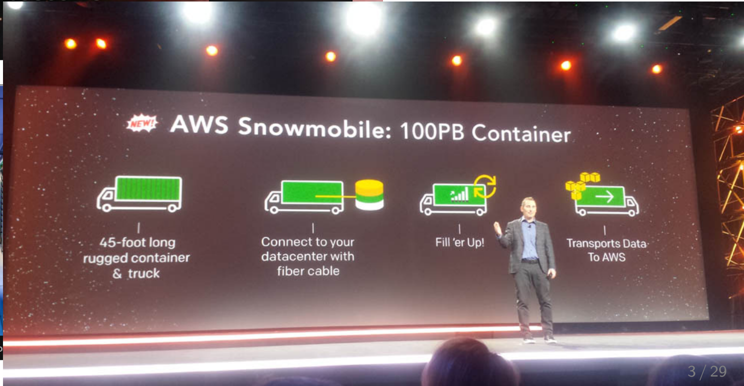
Today these criteria apply equally, or more so, to "web scale data."

200 PB is a lot of data



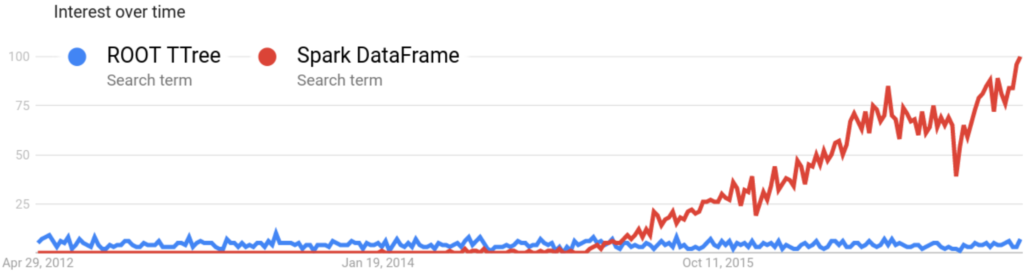
200 PB is a lot of data, but for Amazon, it's two truckloads





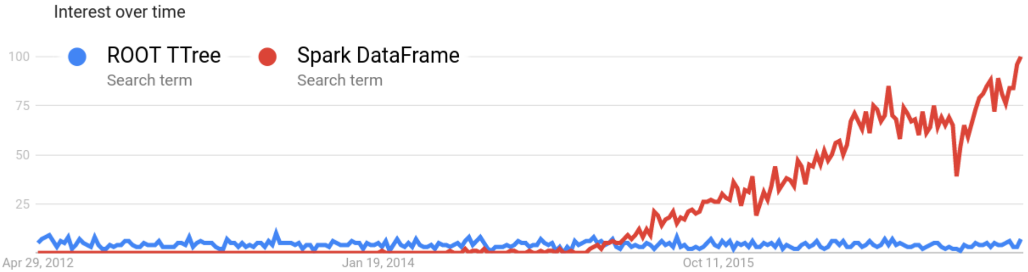
Also a much larger community

Rate of web searches for "ROOT TTree" vs. "Spark DataFrame" (Google Trends):



Also a much larger community

Rate of web searches for “ROOT TTree” vs. “Spark DataFrame” (Google Trends):

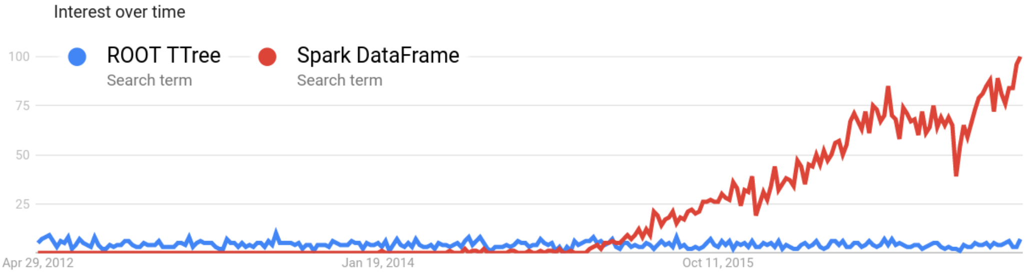


Similarly for question-and-answer sites:

- RootTalk: 14,399 threads in 1997–2012 (15 years)
- StackOverflow questions tagged `#spark`: 26,155 in the 3.3 years the tag has existed. (Not counting CrossValidated, Spark Developer and User mailing lists. . .)

Also a much larger community

Rate of web searches for "ROOT TTree" vs. "Spark DataFrame" (Google Trends):



Similarly for question-and-answer sites:

- RootTalk: 14,399 threads in 1997–2012 (15 years)
- StackOverflow questions tagged `#spark`: 26,155 in the 3.3 years the tag has existed. (Not counting CrossValidated, Spark Developer and User mailing lists...)

More users to talk to; more developers adding features/fixing bugs.

Particle physics is a special case

Particle physics

- Events (modulo cosmics vetos or time-dependent calibrations) may be processed in isolation; embarrassingly parallel.

Big Data

- All-to-all problems are common, such as matching a customer's purchases with all other purchases to make a recommendation.

Particle physics is a special case

Particle physics

- Events (modulo cosmics vetos or time-dependent calibrations) may be processed in isolation; embarrassingly parallel.
- Once collected, physics datasets are immutable (with revisions).

Big Data

- All-to-all problems are common, such as matching a customer's purchases with all other purchases to make a recommendation.
- Transactions accumulate in the database during analysis.

Particle physics is a special case

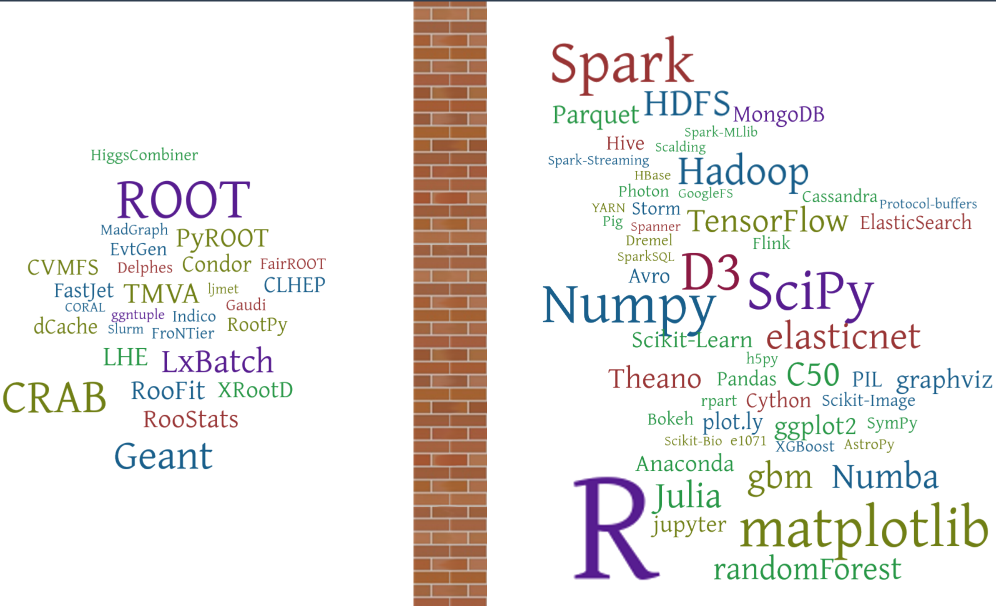
Particle physics

- Events (modulo cosmics vetos or time-dependent calibrations) may be processed in isolation; embarrassingly parallel.
- Once collected, physics datasets are immutable (with revisions).
- Often fitting a model with a small number of parameters.

Big Data

- All-to-all problems are common, such as matching a customer's purchases with all other purchases to make a recommendation.
- Transactions accumulate in the database during analysis.
- Modeling human behavior, more interested in predictions than description, so models may have thousands of free parameters.

Our software is largely isolated from these developments



Who am I? Why am I giving this talk?

Jim Pivarski 

- 5 years CLEO (9 GeV e^+e^-)
- 5 years CMS (7 TeV pp)
- 5 years Open Data Group
- 2 years Project DIANA-HEP

Who am I? Why am I giving this talk?

Jim Pivarski

- 5 years CLEO (9 GeV e^+e^-)
- 5 years CMS (7 TeV pp)
- 5 years Open Data Group
- 2 years Project DIANA-HEP

$\longrightarrow$

hyperspectral imagery
automobile traffic
network security
Twitter sentiment
Google n-grams
DNA sequence analysis
credit card fraud detection
and "Big Data" tools

Who am I? Why am I giving this talk?

Jim Pivarski

- 5 years CLEO (9 GeV e^+e^-)
- 5 years CMS (7 TeV pp)
- 5 years Open Data Group
- 2 years Project DIANA-HEP

$\longrightarrow$

hyperspectral imagery
automobile traffic
network security
Twitter sentiment
Google n-grams
DNA sequence analysis
credit card fraud detection
and "Big Data" tools

My goal within DIANA-HEP is to make it easier for physicists to use Big Data tools in their analyses, particularly for interactive, exploratory analysis.

dianahep Team Activities/Products DIANA Fellows Blog

Collaborative Analyses

Establish infrastructure for a higher-level of collaborative analysis, building on the successful patterns used for the Higgs boson discovery and enabling a deeper communication between the theoretical community and the experimental community

Reproducible Analyses

Streamline efforts associated to reproducibility, analysis preservation, and data preservation by making these native concepts in the tools

Interoperability

Improve the interoperability of HEP tools with the larger scientific software ecosystem, incorporating best practices and algorithms from other disciplines into HEP

Faster Processing

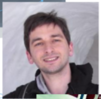
Increase the CPU and IO performance needed to reduce the iteration time so crucial to exploring new ideas

Better Software

Develop software to effectively exploit emerging many- and multi-core hardware.
Promote the concept of software as a research product.

Training

Provide training for students in all of our core research topics.





Collabora Repro rability

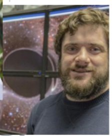
Establish infrastructure f collaborative analysis, buil ful patterns used for the Higgs nd enabling a deeper commun theoretical community and the experiment

S a nd dat conce

prove the interoperability of HEP tools with software ecosystem, actices and algorithms iplines into HEP










Faster Processing

Increase the CPU and IO performance needed to reduce the iteration time so crucial to exploring new ideas

re

Deve xploit emerging many- and multi-core hardware. Promote the concept of software as a research product.

aining

Pr students in all of our core research topics.

What to do with physics software: three cases

Case I: Physics software that serves *the same function* as software in the Big Data community.

Case II: Domain-specific software for our analyses. Example: "HiggsCombiner."

Case III: Physics software or concepts that would benefit the Big Data community.

What to do with physics software: three cases

Case I: Physics software that serves *the same function* as software in the Big Data community.

Big Data community has better resources for

- maintaining code
- catching bugs
- revising bad designs.

Case II: Domain-specific software for our analyses. Example: "HiggsCombiner."

Case III: Physics software or concepts that would benefit the Big Data community.

What to do with physics software: three cases

Case I: Physics software that serves *the same function* as software in the Big Data community.

Big Data community has better resources for

- maintaining code
- catching bugs
- revising bad designs.

Case II: Domain-specific software for our analyses. Example: "HiggsCombiner."

Obviously. This really is a unique problem.

Case III: Physics software or concepts that would benefit the Big Data community.

What to do with physics software: three cases

Case I: Physics software that serves *the same function* as software in the Big Data community.

Big Data community has better resources for

- maintaining code
- catching bugs
- revising bad designs.

Case II: Domain-specific software for our analyses. Example: "HiggsCombiner."

Obviously. This really is a unique problem.

Case III: Physics software or concepts that would benefit the Big Data community.

Cultural exchange goes in both directions.

All three cases in a single story: porting an analysis from ROOT to Spark.

CMS Big Data Project

- Oliver Gutsche, Matteo Cremonesi, Cristina Suárez (Fermilab) wanted to try their CMS dark matter search on Spark.
- My first DIANA-HEP project: I joined to plow through technical issues before the analysts hit them.

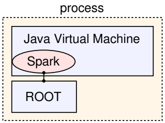
https://cms-big-data.github.io/

How to get the data into Spark (which runs in Java)

A year of trial-and-error in one slide

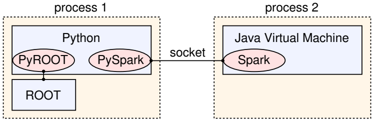
1. Java Native Interface (JNI)

   No! This ought to be the right solution, but Java and ROOT are both large, complex applications with their own memory management: couldn't keep them from interfering (segmentation faults).



2. Python as glue: PyROOT and PySpark in the same process

   PySpark is a low-performance solution: all data must be passed over a text-based socket and interpreted by Python.



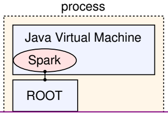
3. Convert to a Spark-friendly format, like Apache Avro

   We used this for most of the year. Efficient after conversion, but conversion step is awkward. Avro's C library is difficult to deploy.

A year of trial-and-error in one slide

1. Java Native Interface (JNI)

   No! This ought to be the right solution, but Java and ROOT are both large, complex applications with their own memory management; couldn't keep



This problem is incidental, not essential. Industry-standard formats like Avro and Parquet can store complex physics events; we just happen to have a lot of data in ROOT files.

   over a text-based socket and interpreted by Python.

3. Convert to a Spark-friendly format, like Apache Avro

   We used this for most of the year. Efficient after conversion, but conversion step is awkward. Avro's C library is difficult to deploy.

This was a missed opportunity for exporting physics solutions!

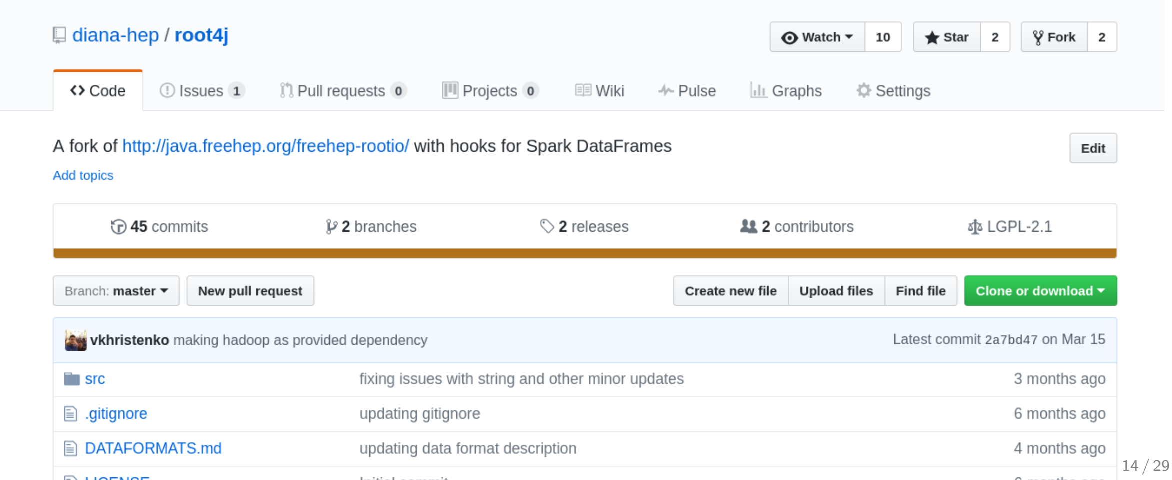
ROOT was storing nested data structures in a columnar format (for faster access) over a decade before it was reinvented at Google.

Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis. *Dremel: Interactive Analysis of Web-Scale Datasets* (2010).

> storage and reduce CPU cost due to cheaper compression. Column stores have been adopted for analyzing relational data [1] but to the best of our knowledge have not been extended to nested data models. The columnar storage format that we present is supported by many data processing tools at Google, including MR, Sawzall [20], and FlumeJava [7].
>
> In this paper we make the following contributions:
>
> - We describe a novel columnar storage format for nested data. We present algorithms for dissecting nested records

Easiest solution: reimplement ROOT I/O in Java

root4j/ spark-root	Java/Scala	For Spark and other Big Data projects that run on Java.	Started by Tony Johnson in 2001, updated by Viktor Khristenko.



Easiest solution: reimplement ROOT I/O in Java, JS

root4j/ spark-root	Java/Scala	For Spark and other Big Data projects that run on Java.	Started by Tony Johnson in 2001, updated by Viktor Khristenko.
JsRoot	Javascript	For interacting with ROOT in web browsers or standalone.	Sergey Linev

Easiest solution: reimplement ROOT I/O in Java, JS, Go

root4j/ spark-root	Java/Scala	For Spark and other Big Data projects that run on Java.	Started by Tony Johnson in 2001, updated by Viktor Khristenko.
JsRoot	Javascript	For interacting with ROOT in web browsers or standalone.	Sergey Linev
rootio	Go	go-hep ecosystem in Go.	Sebastien Binet

Easiest solution: reimplement ROOT I/O in Java, JS, Go, Python

root4j/ spark-root	Java/Scala	For Spark and other Big Data projects that run on Java.	Started by Tony Johnson in 2001, updated by Viktor Khristenko.
JsRoot	Javascript	For interacting with ROOT in web browsers or standalone.	Sergey Linev
rootio	Go	go-hep ecosystem in Go.	Sebastien Binet
uproot	Python	For quickly getting ROOT data into Numpy and Pandas for machine learning.	Jim Pivarski (me)

Easiest solution: reimplement ROOT I/O in Java, JS, Go, Python

root4j/ spark-root	Java/Scala	For Spark and other Big Data projects that run on Java.	Started by Tony Johnson in 2001, updated by Viktor Khristenko.
JsRoot	Javascript	For interacting with ROOT in web browsers or standalone.	Sergey Linev
rootio	Go	go-hep ecosystem in Go.	Sebastien Binet
uproot	Python	For quickly getting ROOT data into Numpy and Pandas for machine learning.	Jim Pivarski (me)
	Rust?		

Example session (native Spark, which is Scala)

Launch Spark with packages from Maven Central.

```
spark-shell --packages org.diana-hep:spark-root_2.11:x.y.z, \
                       org.diana-hep:histogrammar_2.11:1.0.4
```

Read ROOT files like any other DataFrame input source.

```
val df = spark.sqlContext.read.root(
                              "hdfs://path/to/files/*.root")

df.printSchema()
root
 |-- met: float (nullable = false)
 |-- muons: array (nullable = false)
 |    |-- element: struct (containsNull = false)
 |    |    |-- pt: float (nullable = false)
 |    |    |-- eta: float (nullable = false)
 |    |    |-- phi: float (nullable = false)
 |-- jets: array (nullable = false)
```

Example session (PySpark in Python)

Launch Spark with packages from Maven Central.

```
pyspark     --packages org.diana-hep:spark-root_2.11:x.y.z, \
                       org.diana-hep:histogrammar_2.11:1.0.4
```

Read ROOT files like any other DataFrame input source.

```
df = sqlContext.read.format("org.dianahep.sparkroot") \
                    .load("hdfs://path/to/files/*.root")

df.printSchema()
root
 |-- met: float (nullable = false)
 |-- muons: array (nullable = false)
 |    |-- element: struct (containsNull = false)
 |    |    |-- pt: float (nullable = false)
 |    |    |-- eta: float (nullable = false)
 |    |    |-- phi: float (nullable = false)
 |-- jets: array (nullable = false)
```

Example session (native Spark and PySpark)

```
df.show()
+---------+--------------------+--------------------+
|      met|               muons|                jets|
+---------+--------------------+--------------------+
| 55.59374|[[28.07075,-1.331...|[[194.19714,-2.65...|
|39.440292|                  []|[[93.64958,-0.273...|
|2.1817229|[[5.523367,-0.375...|[[96.09923,0.7058...|
|  80.5822|[[48.910114,-0.17...|[[165.2686,0.2623...|
| 84.43806|                  []|[[51.87823,1.6442...|
| 84.63146|[[33.84279,-0.062...|[[137.74776,-0.45...|
| 393.8167|[[25.402626,-0.66...|[[481.8268,-1.115...|
|  75.0873|                  []|[[144.62373,-2.21...|
|2.6512942|[[6.851382,2.3145...|[[72.08256,-1.713...|
|36.753353|                  []|[[72.7172,-1.3265...|
+---------+--------------------+--------------------+
only showing top 10 rows
```

Example session (Spark)

```scala
// Bring dollar-sign notation into scope.
import spark.sqlContext.implicits._


// Compute event weight with columns and constants.
df.select(($"lumi"*xsec/nGen) * $"LHE_weight"(309)).show()

// Pre-defined function (notation's a little weird).
val isGoodEvent = (
    ($"evtHasGoodVtx" === 1)  &&
    ($"evtHasTrg" === 1)      &&
    ($"tkmet" >= 25.0)        &&
    ($"Mu_pt" >= 30.0)        &&
    ($"W_mt" >= 30.0))

// Use it.
println("%d events pass".format(
                    df.where(isGoodEvent).count()))
```

Example session (PySpark)

```
# Python trick: make columns Python variables.
for name in df.schema.names:
    exec("{0} = df['{0}']".format(name))

# Look at a few event weights.
df.select((lumi*xsec/nGen) * LHE_weight[309]).show()

# Pre-defined function (notation's a little different).
isGoodEvent = (
    (evtHasGoodVtx == 1) &
    (evtHasTrg == 1)     &
    (tkmet >= 25.0)      &
    (Mu_pt >= 30.0)      &
    (W_mt >= 30.0))

# Use it.
print "{} events pass".format(
                  df.where(isGoodEvent).count())
```

Example session (Spark)

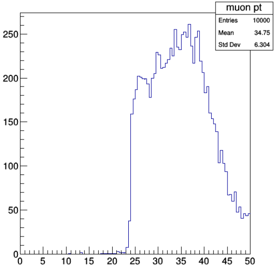
```scala
// Use Histogrammar to make histograms.
import org.dianahep.histogrammar._
import org.dianahep.histogrammar.sparksql._
import org.dianahep.histogrammar.bokeh._

// Define histogram functions with SparkSQL Columns.
val h = df.Label(
          "muon pt" -> Bin(100, 0.0, 50.0, $"Mu_pt"),
          "W mt" -> Bin(100, 0.0, 120.0, $"W_mt"))

// Plot the histograms with Bokeh.
val bokehhist = h.get("muon pt").bokeh()
plot(bokehhist)
val bokehhist2 = h.get("W mt").bokeh()
plot(bokehhist2)
```

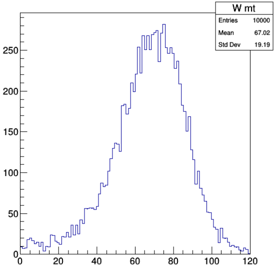
Example session (PySpark)

```python
# Use Histogrammar to make histograms.
from histogrammar import *
import histogrammar.sparksql
histogrammar.sparksql.addMethods(df)

# Define histogram functions with SparkSQL Columns.
h = df.Label(
      muon_pt = Bin(100, 0.0, 50.0, Mu_pt),
      W_mt = Bin(100, 0.0, 120.0, W_mt))

# Plot the histograms with PyROOT.
roothist = h.get("muon_pt").plot.root("muon pt")
roothist.Draw()
roothist2 = h.get("W_mt").plot.root("W mt")
roothist2.Draw()
```

Example session (PySpark)

```
# Use Histogrammar to make histograms.
from histogrammar import *
import histogrammar.sparksql
histogrammar.sparksql.addMethods(df)

# Define histogram functions with SparkSQL Columns.
h = df.Label(
      muon_pt = Bin(100, 0.0, 50.0, Mu_pt),
      W_mt = Bin(100, 0.0, 120.0, W_mt))

# Plot the histograms with PyR(
roothist = h.get("muon_pt").pl(
roothist.Draw()
roothist2 = h.get("W_mt").plot
roothist2.Draw()
```





Spark and Big Data in general are weak in plotting.

Spark and Big Data in general are weak in plotting.

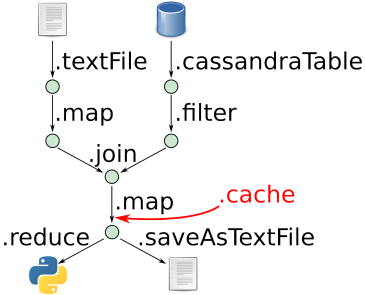
They have fancy visualizations (d3), but lack convenient workaday routines for quick histograms, profiles, heatmaps, lego plots, etc.

Spark and Big Data in general are weak in plotting.

They have fancy visualizations (d3), but lack convenient workaday routines for quick histograms, profiles, heatmaps, lego plots, etc.

(Exception: Python and R have good interactive graphics for in-memory analytics.)

Analysis in Spark is a chain of higher-order functions



In Spark, you submit work by passing functions in a chain:

```
one   = source1.textFile("some.txt")
                .map(x => x.upper())
two   = source2.cassandraTable
                .filter(x => x.field > 3)
three = one.join(two)
four  = three.map((x, y) => (y, x)).cache()
five  = four.reduce((x, y) => x + y)
six   = five.saveAsTextFile("other.txt")
```

Thus, the code doesn't depend on whether or not it's parallelized
(so it can be massively parallelized).

ROOT histogram API is cumbersome in this setting

ROOT (or any HBOOK-style) histograms

```
x = Histogram(100, -5.0, 5.0)

for event in events:
    x.fill(event.calcX())

x.plot()
```

Using them in Spark

```
x = events.aggregate(
    Histogram(100, -5.0, 5.0),
    lambda h, event: (
        h.fill(event.calcX())),
    lambda h1, h2: (
        h1 + h2))

x.plot()
```

ROOT histogram API is cumbersome in this setting

ROOT (or any HBOOK-style) histograms

```
x = Histogram(100, -5.0, 5.0)
y = Histogram(100, -5.0, 5.0)

for event in events:
    x.fill(event.calcX())
    y.fill(event.calcY())

x.plot()
y.plot()
```

Using them in Spark

```
x, y = events.aggregate(
    (Histogram(100, -5.0, 5.0),
     Histogram(100, -5.0, 5.0)),
    lambda hs, event: (
        hs[0].fill(event.calcX()),
        hs[1].fill(event.calcY())),
    lambda hs1, hs2: (
        hs1[0] + hs2[0],
        hs1[1] + hs2[1]))

x.plot()
y.plot()
```

ROOT histogram API is cumbersome in this setting

ROOT (or any HBOOK-style) histograms

```
x = Histogram(100, -5.0, 5.0)
y = Histogram(100, -5.0, 5.0)
z = Histogram(100, -5.0, 5.0)

for event in events:
    x.fill(event.calcX())
    y.fill(event.calcY())
    z.fill(event.calcZ())

x.plot()
y.plot()
z.plot()
```

Using them in Spark

```
x, y, z = events.aggregate(
    (Histogram(100, -5.0, 5.0),
     Histogram(100, -5.0, 5.0),
     Histogram(100, -5.0, 5.0)),
    lambda hs, event: (
        hs[0].fill(event.calcX()),
        hs[1].fill(event.calcY()),
        hs[2].fill(event.calcZ())),
    lambda hs1, hs2: (
        hs1[0] + hs2[0],
        hs1[1] + hs2[1],
        hs1[2] + hs2[2]))

x.plot()
y.plot()
z.plot()
```

Solution: make the histograms functional, like the rest of Spark

Histogram constructor as a higher-order function:

```
h = Histogram(numBins, lowEdge, highEdge, fillRule)
```

where **`fillRule`** is a function : *data* $\to \mathbb{R}$ that determines which bin an element of *data* increments.

Solution: make the histograms functional, like the rest of Spark

Histogram constructor as a higher-order function:

```
h = Histogram(numBins, lowEdge, highEdge, fillRule)
```

where **fillRule** is a function : *data* $\rightarrow \mathbb{R}$ that determines which bin an element of *data* increments.

All domain-specific knowledge is in the constructor. The filling function may now be generic (and automated).

```
h.fill(datum)       # calls fillRule(datum) internally
```

Familiar histogram types are now generated by combinators

Histograms:

```
Bin(num, low, high, fillRule,
  Count())
```

Two-dimensional histograms:

```
Bin(xnum, xlow, xhigh, xfill,
  Bin(ynum, ylow, yhigh, yfill,
    Count()))
```

Profile plots:

```
Bin(xnum, xlow, xhigh, xfill,
  Deviate(yfill))
```

where `Deviate` aggregates a mean and standard deviation.

Mix and match binning methods:

```
IrregularlyBin([-2.4, -2.1, -1.5,
    0.0, 1.5, 2.1, 2.4],
  filleta,
  Bin(314, -3.14, 3.14, fillphi,
    Count()))
```

```
SparselyBin(0.01, filleta,
  Bin(314, -3.14, 3.14, fillphi,
    Count()))
```

```
Categorize(fillByName,
  Bin(314, -3.14, 3.14, fillphi,
    Count()))
```

It all got mathematical pretty fast. . .

For transparent parallelization, combinators must

be additive:

independent of *whether* datasets are partitioned.

$$\mathrm{fill}(\mathrm{data}_1 + \mathrm{data}_2) = \mathrm{fill}(\mathrm{data}_1) + \mathrm{fill}(\mathrm{data}_2)$$

be homogeneous in the weights:

fill weight 0.0 corresponds to no fill, 1.0 to simple fill, 2.0 to double-fill, . . .

$$\mathrm{fill}(\mathrm{data}, \mathrm{weight}) = \mathrm{fill}(\mathrm{data}) \cdot \mathrm{weight}$$

linear

be associative:

independent of *where* datasets get partitioned.

$$(h_1 + h_2) + h_3 = h_1 + (h_2 + h_3)$$

have an identity:

for both the fill and $+$ methods.

$$h + 0 = h, \quad 0 + h = h, \quad \mathrm{fill}(\mathrm{data}, 0) = 0$$

monoid

histo·*grammar*

/histōˌˈɡɹæm.ər/

http://histogrammar.org

(Get it?)

Other projects in development

- **uproot**: fast reading of ROOT files into Numpy/Pandas/Apache Arrow.
- **Arrowed**: transpiling complex analysis functions to skip object materialization (like SQL term rewriting, but for objects in Arrow format).
- Extending ROOT to use an object store database instead of seek points in a file (the "petabyte ROOT file" project).
- Speeding up analysis cuts with database-style indexing.
- **Femtocode**: Domain Specific Language (DSL) for particle physics queries.

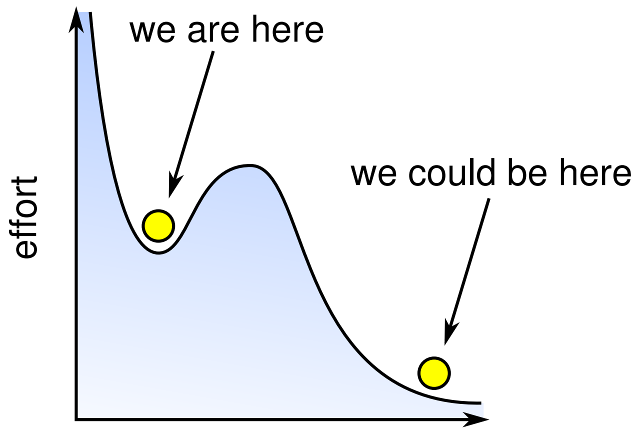
Other projects in development

- **uproot**: fast reading of ROOT files into Numpy/Pandas/Apache Arrow.
- **Arrowed**: transpiling complex analysis functions to skip object materialization (like SQL term rewriting, but for objects in Arrow format).
- Extending ROOT to use an object store database instead of seek points in a file (the "petabyte ROOT file" project).
- Speeding up analysis cuts with database-style indexing.
- **Femtocode**: Domain Specific Language (DSL) for particle physics queries.

All of the above are parts of the following:

- To develop a centralized query service that is as responsive as a private skim: to eliminate the need to copy data just to analyze it.

Conclusions



Conclusions