Data plumbing: moving data across frameworks

Jim Pivarski

Princeton University – DIANA

May 11, 2017
Why worry about moving data?

We’re all here because we think that HEP can benefit from machine learning techniques.
We’re all here because we think that HEP can benefit from machine learning techniques.

The most advanced techniques are being developed outside of HEP, using (what are becoming) industry standard tools.
Why worry about moving data?

We’re all here because we think that HEP can benefit from machine learning techniques.

The most advanced techniques are being developed outside of HEP, using (what are becoming) industry standard tools.

**Problem:** our HEP protocols and formats won’t work with them without some modification or export.
Why worry about moving data?

We’re all here because we think that HEP can benefit from machine learning techniques.

The most advanced techniques are being developed outside of HEP, using (what are becoming) industry standard tools.

Problem: our HEP protocols and formats won’t work with them without some modification or export.

This talk is about solutions to that problem.
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.
Collaboration
Establish infrastructure for collaborative analysis, building on the lessons learned from successful collective analysis and data processing and enabling a deeper community of theoretical and experimental scientists.

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

Replicability
Promote the concept of software as a research product.

Interoperability
Promote the concept of software as a research product.

Software Architecture
Develop software architecture to exploit emerging many- and multi-core hardware.

Training
Train students in all of our core research topics.
Goals of this talk

- make you aware of what’s possible
- introduce some software tools
- invite you to tell me what’s needed.
ROOT and Numpy
**ROOT** is the standard for HEP data storage and processing. **Numpy** is the standard for the scientific Python ecosystem: SciPy, Scikit-Learn, TensorFlow, Keras, PyTorch, DyNet, MinPy, MXNet…
ROOT is the standard for HEP data storage and processing.

Numpy is the standard for the scientific Python ecosystem: SciPy, Scikit-Learn, TensorFlow, Keras, PyTorch, DyNet, MinPy, MXNet…

root_numpy from Scikit-HEP provides high-level translation to/from ROOT TTrees and Numpy arrays.

http://scikit-hep.org/root_numpy/
import root_numpy

# get an array from a ROOT file named by string
filename = root_numpy.testdata.get_filepath("test.root")
array1 = root_numpy.root2array(filename, "tree")

# or get an array from a PyROOT TFile/TTree
import ROOT
rootfile = ROOT.TFile(filename)
roottree = rootfile.Get("tree")
array2 = root_numpy.tree2array(roottree, branches=["x", "y", "sqrt(y)", "TMath::Landau(x)"], selection="z > 0")
import root_numpy

# get an array from a ROOT file named by string
callname = root_numpy.testdata.get_filepath("test.root")
array1 = root_numpy.root2array(filename, "tree")

# or get an array from a PyROOT TFile/TTree
import ROOT
rootfile = ROOT.TFile(filename)
roottree = rootfile.Get("tree")
array2 = root_numpy.tree2array(roottree)

Use TTree::Draw syntax to transform branches and cut events.
array3 = root_numpy.tree2array(roottree,
    branches=['x', 'y', 'sqrt(y)', 'TMath::Landau(x)'],
    selection="z > 0")
Worth noting . . .

- Although root_numpy currently loads whole branches into memory at once, it’s possible to extend to streaming C++ APIs (hint: TensorFlow Queues).

- There are other extensions “out there,” such as this one: https://github.com/ibab/root_pandas

- Scikit-HEP is a metapackage (developers below) to try to make things like this easier to find.

  Noel Dawe (University of Melbourne), Vanya Belyaev (ITEP), Sasha Mazurov (University of Birmingham), Eduardo Rodrigues (University of Cincinnati), David Lange (Princeton University), and myself.
direct to Numpy
root_numpy is great if you have a lot of ROOT files and need to analyze them in Python.
root_numpy is great if you have a lot of ROOT files and need to analyze them in Python.

But if you don’t already have the ROOT files, generating and then converting them is awkward, especially if the dataset is large.
root_numpy is great if you have a lot of ROOT files and need to analyze them in Python.

But if you don’t already have the ROOT files, generating and then converting them is awkward, especially if the dataset is large.

Fortunately, the Numpy format is extremely simple.
root_numpy is great if you have a lot of ROOT files and need to analyze them in Python.

But if you don’t already have the ROOT files, generating and then converting them is awkward, especially if the dataset is large.

Fortunately, the Numpy format is extremely simple.

- a Numpy array is a plain C array interpreted by metadata (data type, number of elements, endianness, C vs. Fortran-style stride...)
  - you can wrap any region of memory as a Numpy array
root_numpy is great if you have a lot of ROOT files and need to analyze them in Python.

But if you don’t already have the ROOT files, generating and then converting them is awkward, especially if the dataset is large.

Fortunately, the Numpy format is extremely simple.

▶ a Numpy array is a plain C array interpreted by metadata (data type, number of elements, endianness, C vs. Fortran-style stride...)
  ▶ you can wrap any region of memory as a Numpy array

▶ a Numpy file is a literal copy of the array with a header
  ▶ you can write Numpy files with minimal code
If PyROOT gives you an `array.array`, wrap it like this:

```python
import numpy
numpy_array = numpy.frombuffer(array_from_root)
```

Now it has Numpy powers.
Examples of wrapping arrays

If PyROOT gives you an `array.array`, wrap it like this:

```python
import numpy
numpy_array = numpy.frombuffer(array_from_root)
```

Now it has Numpy powers.

As long as you perform in-place operations, like

```python
# overwrite all elements x with sin(x)
numpy.sin(numpy_array, numpy_array)
# set all values to 3.14
numpy_array[:] = 3.14
```

it will modify the same memory that ROOT is looking at.
Examples of wrapping arrays

If PyROOT gives you an `array.array`, wrap it like this:

```python
import numpy
numpy_array = numpy.frombuffer(array_from_root)
```

Now it has Numpy powers.

As long as you perform in-place operations, like

```python
# overwrite all elements x with sin(x)
numpy.sin(numpy_array, numpy_array)
# set all values to 3.14
numpy_array[:] = 3.14
```

it will modify the same memory that ROOT is looking at.

With great power comes great responsibility: if ROOT deletes this array and you continue to modify it, you will corrupt memory, causing a segmentation fault at best.
You can split a Python script into parallel processes using its built-in `multiprocessing` module. These processes can share a block of memory, which you can wrap with Numpy.

See [https://goo.gl/NPwcSL](https://goo.gl/NPwcSL) for an example.
Examples of wrapping arrays

You can split a Python script into parallel processes using its built-in `multiprocessing` module. These processes can share a block of memory, which you can wrap with Numpy.

See [https://goo.gl/NPwcSL](https://goo.gl/NPwcSL) for an example.

**Another possible use:** point Python and a C++ framework (e.g. Athena or CMSSW) to the same shared memory to transfer data between them at runtime.

Also known as a “common block.”  :)

(A good implementation would be wrapped in a thread-safe, type-safe API, of course!)
Even use non-standard allocators (this one allocates memory on Knight’s Landing MCDRAM).

```python
import ctypes
import numpy

ZILLION = 1000000

libnuma = ctypes.cdll.LoadLibrary("libnuma.so")
libnuma.numa_alloc_local.restype = ctypes.POINTER(ctypes.c_double)
ptr = libnuma.numa_alloc_local(ctypes.c_size_t(ZILLION))

ptr.__array_interface__ =
    {
        "version": 3,
        "typestr": numpy.ctypeslib._dtype(type(ptr.contents)).str,
        "data": (ctypes.addressof(ptr.contents), False),
        "shape": (ZILLION,)}

asarray = numpy.array(ptr, copy=False)
```
Writing Numpy files is easy, too

https://github.com/diana-hep/c2numpy

Pure-header C library: drop it in and write Numpy files.

```c
#include "c2numpy.h"

c2numpy_init(&writer, "output/tracks", 1000);
c2numpy_addcolumn(&writer, "pt", C2NUMPY_FLOAT64);
c2numpy_addcolumn(&writer, "eta", C2NUMPY_FLOAT64);
c2numpy_addcolumn(&writer, "phi", C2NUMPY_FLOAT64);
c2numpy_addcolumn(&writer, "dxy", C2NUMPY_FLOAT64);
c2numpy_addcolumn(&writer, "dz", C2NUMPY_FLOAT64);

...

for (auto track = tracks->cbegin(); track != tracks->end(); ++track) {
    c2numpy_float64(&writer, track->pt());
    c2numpy_float64(&writer, track->eta());
    c2numpy_float64(&writer, track->phi());
    c2numpy_float64(&writer, track->dxy());
    c2numpy_float64(&writer, track->dz());
}
```
industry standard formats

Avro/Thrift/ProtoBuf, Parquet/Feather, Arrow
What about nested structure?

Numpy isn’t appropriate (efficient) for anything but flat-flat ntuples: strictly columns of numbers, no `std::vector<double>`!
What about nested structure?

Numpy isn’t appropriate (efficient) for anything but flat-flat ntuples: strictly columns of numbers, no `std::vector<double>`!

ROOT pioneered efficient storage of nested, hierarchical data with built-in schema (TTrees), but today there are other options:

- **row-wise ("unsplit")**
  - Avro, Thrift, ProtoBuf
- **columnar ("split")**
  - Parquet, Feather
- **in-memory**
  - Arrow
Abstract type systems

All of these formats are interconvertable and accessible in dozens of programming languages because they’re all based on roughly the same abstract type systems.

Data types are

- **null**: only one possible value, usually not written explicitly
- **boolean**: true or false
- **integer**: whole numbers
- **float**: floating-point numbers (usually with specified precision)
- **string**: usually UTF-8
- **list**: arbitrary length collections of the above
- **record**: structs whose fields are any of the above
- **union**: one type or another type (tagged)

but no pointers/TRefs or class methods (functions).
Automated conversion

https://github.com/diana-hep/rootconverter

is an *unmaintained* software package that converted ROOT files, including any nested classes, into Avro format. It mapped ROOT’s TStreamerInfo onto the corresponding abstract data types.

It could be resurrected or repurposed if there’s a need: the point is that this is *possible*. 
Spark and the JVM
This was developed as part of a project to perform a CMS analysis in Apache Spark.

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

Last year, we converted all data from ROOT to Avro because Spark recognizes the Avro format (previous page).

This year, we’re using a pure Java implementation of the ROOT format to load data directly into Spark.
Root Object Browser

As an illustration of the use of the Java interface, we have built a sample application which is a simple Root Object Browser. It can be used to open any Root file and look at all the objects inside the file. If you already have Java 2 installed (JDK 1.3), you can download the root.jar file containing the application, and run it using the command:

```
java -jar root.jar
```

(on Windows you can just double-click on the root.jar file). A screen shot of the application is show below. The pane on the left shows the directory structure of the file. The object browser knows how to navigate directories (TDirectories), trees (TTrees and TBranches) and these will all be shown in the left pane. Clicking on any object in the left pane will cause the details of the object to be shown in the right pane. The right pane knows how to follow embedded pointers to other objects.
Root Object Browser

As an illustration of the use of the Java interface, we have built a sample application which is a simple Root Object Browser. It can be used to open any Root file and look at all the objects inside the file. If you already have Java 2 installed (JDK 1.3), you can download the root.jar file containing the application, and run it using the command:

```
java -jar root.jar
```

(on Windows you can just double-click on the root.jar file). A screen shot of the application is show below. The pane on the left shows the directory structure of the file. The object browser knows how to navigate directories (TDirectories), trees (TTrees and TBranches) and these will all be shown in the left pane. Clicking on any object in the left pane will cause the details of the object to be shown in the right pane. The right pane knows how to follow embedded pointers to other objects.

![Screen shot of Root Object Browser](image-url)
A fork of http://java.freehep.org/freehep-rootio/ with hooks for Spark DataFrames

Add topics

- 45 commits
- 2 branches
- 2 releases
- 2 contributors
- LGPL-2.1

Latest commit 2a7bd47 on Mar 15

- src
  - fixing issues with string and other minor updates

- .gitignore
  - updating gitignore

- DATAFORMATS.md
  - updating data format description

- LICENSE
  - Initial commit

- README.md
  - updated readme

- pom.xml
  - making hadoop as provided dependency

ROOT4J

A fork of http://java.freehep.org/freehep-rootio/
A fork of http://java.freehep.org/freehep-rootio/ with hooks for Spark DataFrames

Viktor Khristenko
University of Iowa
Example session (Spark)

Launch Spark with packages from Maven Central.

```
spark-shell --packages \
  org.diana-hep:spark-root_2.11:0.1.11
```

Read ROOT file like any other format for a DataFrame.

```
import org.dianahep.sparkroot._
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)

Launch Spark with packages from Maven Central.

```python
pyspark --packages \
org.diana-hep:spark-root_2.11:0.1.11
```

Read ROOT file like any other format for a DataFrame.

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

```python
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 (Spark)

```python
df.show()
```

<table>
<thead>
<tr>
<th>met</th>
<th>muons</th>
<th>jets</th>
</tr>
</thead>
<tbody>
<tr>
<td>55.59374</td>
<td>[[28.07075, -1.331...</td>
<td>[[194.19714, -2.65...</td>
</tr>
<tr>
<td>39.440292</td>
<td>[]</td>
<td>[[93.64958, -0.273...</td>
</tr>
<tr>
<td>2.1817229</td>
<td>[[5.523367, -0.375...</td>
<td>[[96.09923, 0.7058...</td>
</tr>
<tr>
<td>80.5822</td>
<td>[[48.910114, -0.17...</td>
<td>[[165.2686, 0.2623...</td>
</tr>
<tr>
<td>84.43806</td>
<td>[]</td>
<td>[[51.87823, 1.6442...</td>
</tr>
<tr>
<td>84.63146</td>
<td>[[33.84279, -0.062...</td>
<td>[[137.74776, -0.45...</td>
</tr>
<tr>
<td>393.8167</td>
<td>[[25.402626, -0.66...</td>
<td>[[481.8268, -1.115...</td>
</tr>
<tr>
<td>75.0873</td>
<td>[]</td>
<td>[[144.62373, -2.21...</td>
</tr>
<tr>
<td>2.6512942</td>
<td>[[6.851382, 2.3145...</td>
<td>[[72.08256, -1.713...</td>
</tr>
<tr>
<td>36.753353</td>
<td>[]</td>
<td>[[72.7172, -1.3265...</td>
</tr>
</tbody>
</table>
```

only showing top 10 rows
Example session (PySpark)

df.show()

+---------+--------------------+--------------------+
<table>
<thead>
<tr>
<th>met</th>
<th>muons</th>
<th>jets</th>
</tr>
</thead>
<tbody>
<tr>
<td>55.59374</td>
<td>[[28.07075, -1.331...</td>
<td>[[194.19714, -2.65...</td>
</tr>
<tr>
<td>39.440292</td>
<td>[]</td>
<td>[[93.64958, -0.273...</td>
</tr>
<tr>
<td>2.1817229</td>
<td>[[5.523367, -0.375...</td>
<td>[[96.09923, 0.7058...</td>
</tr>
<tr>
<td>80.5822</td>
<td>[[48.910114, -0.17...</td>
<td>[[165.2686, 0.2623...</td>
</tr>
<tr>
<td>84.43806</td>
<td>[]</td>
<td>[[51.87823, 1.6442...</td>
</tr>
<tr>
<td>84.63146</td>
<td>[[33.84279, -0.062...</td>
<td>[[137.74776, -0.45...</td>
</tr>
<tr>
<td>393.8167</td>
<td>[[25.402626, -0.66...</td>
<td>[[481.8268, -1.115...</td>
</tr>
<tr>
<td>75.0873</td>
<td>[]</td>
<td>[[144.62373, -2.21...</td>
</tr>
<tr>
<td>2.6512942</td>
<td>[[6.851382, 2.3145...</td>
<td>[[72.08256, -1.713...</td>
</tr>
<tr>
<td>36.753353</td>
<td>[]</td>
<td>[[72.7172, -1.3265...</td>
</tr>
</tbody>
</table>
+---------+--------------------+--------------------+

only showing top 10 rows
Example session (Spark)

(This is from a real CMS analysis.)

// 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()))
# 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())
spark-shell --packages \\
  org.diana-hep:spark-root_2.11:0.1.11, \\
  org.diana-hep:histogrammar_2.11:1.0.4

// 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)
pyspark --packages \
    org.diana-hep:spark-root_2.11:0.1.11, \
    org.diana-hep:histogrammar_2.11:1.0.4

# 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()
pyspark --packages \n  org.diana-hep:spark-root_2.11:0.1.11, \n  org.diana-hep:histogrammar_2.11:1.0.4

# 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)
)

# Plot the histograms
roothist = h.get("muon_pt").plot.root("muon pt")
roothist.Draw()
roothist2 = h.get("W_mt").plot.root("W mt")
roothist2.Draw()
root4j (ROOT reader) is separate from spark-root.

root4j opens the door to all the Java-based big data tools.

As far as I’m aware, it is one of only five ROOT TTree readers:

<table>
<thead>
<tr>
<th>standard ROOT</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>JsRoot</td>
<td>JavaScript</td>
</tr>
<tr>
<td>root4j</td>
<td>Java</td>
</tr>
<tr>
<td>RIO in GEANT</td>
<td>C++</td>
</tr>
<tr>
<td>go-hep</td>
<td>go</td>
</tr>
</tbody>
</table>
Perhaps you saw something here and thought,

- “I can use that to avoid my awful work-around!” or
- “I didn’t think that was possible! Now I can do something I wouldn’t have considered before,” or
- “What I want to do is possible, but it will take some work.”
Perhaps you saw something here and thought,

- “I can use that to avoid my awful work-around!” or
- “I didn’t think that was possible! Now I can do something I wouldn’t have considered before,” or
- “What I want to do is possible, but it will take some work.”

If so, contact me and I may be able to help. I know or am the author of several of these packages, and can help you get started if you need to develop something new.

pivarski@fnal.gov