Reading ROOT data in Java and Spark

Jim Pivarski

Princeton – DIANA

November 7, 2016

Motivation

In a physics analysis using Spark (Oliver Gutsche, Matteo Cremonesi, Cristina Mantilla), the first step is data access.

Hard to cross the boundary between ROOT's C++ runtime and Spark's Java Virtual Machine (JVM).

Motivation

In a physics analysis using Spark (Oliver Gutsche, Matteo Cremonesi, Cristina Mantilla), the first step is data access.

Hard to cross the boundary between ROOT's C++ runtime and Spark's Java Virtual Machine (JVM).

Four methods considered:

1. Convert all data from ROOT to another format (Avro).
2. Access ROOT inside the JVM via JNI.
3. Access ROOT as an external process (pipe or socket).
4. Run PyROOT in PySpark.

Motivation

In a physics analysis using Spark (Oliver Gutsche, Matteo Cremonesi, Cristina Mantilla), the first step is data access.

Hard to cross the boundary between ROOT's C++ runtime and Spark's Java Virtual Machine (JVM).

Four methods considered:

1. Convert all data from ROOT to another format (Avro).
2. Access ROOT inside the JVM via JNI.
3. Access ROOT as an external process (pipe or socket).
4. Run PyROOT in PySpark.

The 2016 analysis used option #1, but it's not ideal.

- Separate conversion step before running Spark.
- Two copies of the data: extra version control and disk usage.

Previously rejected solution

FreeHEP ROOTIO

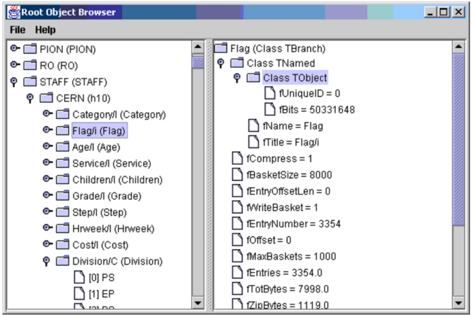
Last Published: 2013-03-01 | Version: 2.2.1

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.



Previously rejected solution

FreeHEP ROOTIO (http://java.freehep.org/freehep-rootio/)

- Re-implementation of ROOT's I/O in Java.
- Used as a backend in Java Analysis Studio (JAS).
- First talks by Tony Johnson in 2001, commits trail off ∼2010.

Previously rejected solution

FreeHEP ROOTIO (http://java.freehep.org/freehep-rootio/)

- Re-implementation of ROOT's I/O in Java.
- Used as a backend in Java Analysis Studio (JAS).
- First talks by Tony Johnson in 2001, commits trail off ∼2010.

Why not?

- Lack of documentation; high-level interface described on website no longer exists.
- Immediately failed when presented with recent ROOT file.
- Can't get in touch with Tony Johnson.

Previously rejected solution

FreeHEP ROOTIO (http://java.freehep.org/freehep-rootio/)

- Re-implementation of ROOT's I/O in Java.
- Used as a backend in Java Analysis Studio (JAS).
- First talks by Tony Johnson in 2001, commits trail off ∼2010.

Why not?

- Lack of documentation; high-level interface described on website no longer exists.
- Immediately failed when presented with recent ROOT file.
- Can't get in touch with Tony Johnson.

Why now?

- I reexamined its low-level interface (direct access to TBranches/TBaskets), and it works!
- Only 3 minor bug-fixes needed to read complex CMS AOD.

Why this is a good solution

Advantage of pure Java code:

- No intermediate files: version control and extra disk space.
- No attempt to run two large projects (ROOT and Java) in the same process, which caused hard-to-trace segmentation faults.
- No passing of data through a serialized stream (both solutions #3 and #4 on the first page).

Why this is a good solution

Advantage of pure Java code:

- No intermediate files: version control and extra disk space.
- No attempt to run two large projects (ROOT and Java) in the same process, which caused hard-to-trace segmentation faults.
- No passing of data through a serialized stream (both solutions #3 and #4 on the first page).

Advantage of this particular code:	verified?
Can read complex structures (directly via TBaskets).	$\surd$
Only small corrections are likely.	$\sim$
Well-organized; class/method names mirror ROOT.	$\surd$
JIT-compiles streamers, rather than generic functions.	$\surd$
Can write objects to ROOT files as well.	
Has an embedded XRootD client (HDFS and EOS!).	

Project: expose ROOT format in Spark

root4j

- Fork of original FreeHEP with JAS dependency and GUI components removed.
- Java, minimal dependencies, built with Maven.
- LGPL 2.1 (like original).

spark-root

- New project, depends on root4j, Hadoop, Spark.
- Presents ROOT TChain as a Spark DataFrame.
- Scala, built with SBT.
- Apache 2.0.

Project: expose ROOT format in Spark

root4j

- Fork of original FreeHEP with JAS dependency and GUI components removed.
- Java, minimal dependencies, built with Maven.
- LGPL 2.1 (like original).

spark-root

- New project, depends on root4j, Hadoop, Spark.
- Presents ROOT TChain as a Spark DataFrame.
- Scala, built with SBT.
- Apache 2.0.

User would launch Spark like this:

```
spark-shell --packages \
    org.diana-hep:spark-root_2.11:1.0.0
```

(Spark fetches JAR and its dependencies from Maven Central.)

Example session

```
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)
 |    |-- element: struct (containsNull = false)
 |    |    |-- pt: float (nullable = false)
 |    |    |-- eta: float (nullable = false)
 |    |    |-- phi: float (nullable = false)
```

```
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

```scala
case class LorentzVector(pt: Float, eta: Float)
case class Event(met: Float, muons: Seq[LorentzVector])

val rdd = df.as[Event]   // OOP-style interface

import org.dianahep.histogrammar._
import org.dianahep.histogrammar.sparksql._

// RDD plotting
val h1 = rdd.aggregate(
    Bin(100, 0.0, 20.0, {event: Event => event.met}))
    (new Increment, new Combine)

// DataFrame plotting
val h2 = df.histogrammar(Bin(100, 0.0, 20.0, $"met"))
```

Example session

```scala
case class LorentzVector(pt: Float, eta: Float)
case class Event(met: Float, muons: Seq[LorentzVector])

val rdd = df.as[Event]   // OOP-style interface

import org.dianahep.histogrammar._
import org.dianahep.histogrammar.sparksql._

// RDD plotting
val h1 = rdd.aggregate(
    Bin(100, 0.0, 20.0, {event: Event => event.met}))
    (new Increment, new Combine)

// DataFrame plotting
val h2 = df.histogrammar(Bin(100, 0.0, 20.0, $"met"))
```

In Python/PySpark,

- SQL-style access to DataFrames is as performant as Scala.
- OOP-style access isn't (but class declaration isn't necessary).

ROOT → DataFrame features

- SparkSQL examines user's query, optimizes a work plan, and provides data source with the following information:
  - which fields are required ("pruning");
  - conservative cuts to use as a prefilter ("filtering").
- Matches well with ROOT's layout: can read just the pruned branches and execute the filter at the source.

  Compare to Parquet, the leading columnar format for big data.
- Since root4j can write (untested) we should also implement

  `spark.sqlContext.save.root("filename.root")`

  to share results back to C++ ROOT.

ROOT → DataFrame features

- SparkSQL examines user's query, optimizes a work plan, and provides data source with the following information:
  - which fields are required ("pruning");
  - conservative cuts to use as a prefilter ("filtering").
- Matches well with ROOT's layout: can read just the pruned branches and execute the filter at the source.

  Compare to Parquet, the leading columnar format for big data.
- Since root4j can write (untested) we should also implement

  `spark.sqlContext.save.root("filename.root")`

  to share results back to C++ ROOT.

Viktor Khristenko (U. Iowa Ph.D. student) is developing it with guidance from me.

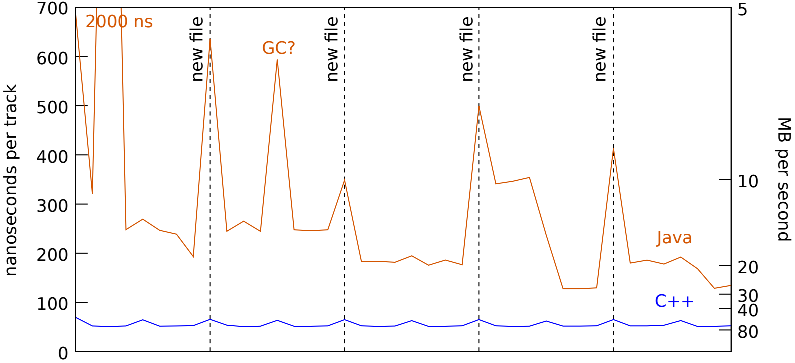
Performance?

diana**hep**

Not terrible. After a dynamic optimization phase, Java is about a factor of four slower than C++.

Reading one nested branch (χ^2 of all tracks per event), see

https://gist.github.com/jpivarski/e8b9da99152bccf70ba187cdab149563



Thanks to Philippe for help writing apples-to-apples C++ code.

Next steps

- Viktor Khristenko and I are developing a roadmap.
  - We have work-items and are determining reasonable time estimates.
  - https://github.com/diana-hep/root4j
  - https://github.com/diana-hep/spark-root
- This should someday be a feature of the CERN Spark cluster and SWAN.
  - Need to coordinate with Luca Canali, Kacper Surdy, Katarzyna Dziedziniewicz-Wojcik and Danilo and Enric.