Accessing ROOT from the JVM (update)

Jim Pivarski

2016-02-22

Motivation (reminder)

Data pipeline tools (such as Apache Hadoop, Spark, Storm, etc.) run on the Java Virtual Machine (JVM) and most physics data is in ROOT, so we need a bridge.

Target use-case: help physics groups move their TTree skimming jobs to Spark.

- Potentially faster for iterative studies (skim, fix bug, reskim) because intermediate datasets can be cached in-memory.
- Abstracts away file locations and transfers, focuses on data transformations.
- Consolidates many ad-hoc shell scripts into a single, programmable workflow.
- Tree of `map`/`filter`/`reduce` transformations can simplify scanning (parameter scans, cut scans, . . .).
- May require training to help physicists adopt the new paradigm, so limit scope to skimming for now.

Following two approaches:

- **FreeHEP-ROOTIO** (pure-Java reimplementation)
  - Never heard back from Tony Johnson.
  - But it works: never had any problems opening ROOT files (even old ones), and TTree interface is good.
  - However, `RootFileReader` requires a file on disk, which limits usefulness.
    - No `java.io.InputStream` constructor because of seeking.
    - `java.net.URL` constructor doesn't accept "`http://`"???

Following two approaches:

- **FreeHEP-ROOTIO** (pure-Java reimplementation)
  - Never heard back from Tony Johnson.
  - But it works: never had any problems opening ROOT files (even old ones), and TTree interface is good.
  - However, `RootFileReader` requires a file on disk, which limits usefulness.
    - No `java.io.InputStream` constructor because of seeking.
    - `java.net.URL` constructor doesn't accept "`http://`"???
- Bridge to native ROOT libraries
  - Java's built-in **JNI** requires some care (always failed for me).
  - **JNA** library works pretty well.
    - Must be adapted with intermediate C code to provide a C-like interface.
    - Intermediate `.so` file *can* be included in the deployed JAR.
    - Segmentation faults rarely (and randomly). Haven't found the cause: it's outside my code and I don't delete any pointers.
  - Newer **BridJ** library might help:
    - Intended for C++, maybe no need for intermediate `.so` file.
    - Richer interface for dealing with pointers; optimized for speed.

Status

I have working code, but I'm rapidly swapping it out as I try new things. New `scaroot` git branch for each major change.

- Built clean, fast Scala interface to TTrees using compile-time macros (next page).
- Successfully passed TTrees through Hadoop map-reduce.
- Successfully passed TTrees through a Spark workflow.

However,

- Hadoop mappers had to copy the file from HDFS to local disk before reading (fixable).
- Spark could only use the user's classes if precompiled in a JAR, not given on the commandline, dramatically changing the Spark user experience (fixable).
- Also, Spark's Kryo serialization had to be used (not a bad thing: it's 10 times faster than native Java serialization).

Scala interface

The user has to know the names and types of leaves in the TTree to define an interface. Otherwise, the interface can't be a first-class object with precompiled field accessors.

Interface could be auto-generated from a sample ROOT file and pasted into a user's project.

```
case class Dimuon(mass: Float, px: Float, py: Float, pz: Float)
```

Scala case classes:

- are immutable, lightweight data objects;
- automatically present constructor arguments as public fields;
- have a readable commandline representation;
- can be used in pattern-matching for declarative condition checking;
- are a common currency for Scala data transformation.

Scala interface

The user has to know the names and types of leaves in the TTree to define an interface. Otherwise, the interface can't be a first-class object with precompiled field accessors.

Interface could be auto-generated from a sample ROOT file and pasted into a user's project.

```
case class Dimuon(mass: Float, px: Float, py: Float, pz: Float)
{
  def momentum = Math.sqrt(px*px + py*py + pz*pz)
  def energy = Math.sqrt(mass*mass + momentum*momentum)
}
```

Scala case classes:

- are immutable, lightweight data objects;
- automatically present constructor arguments as public fields;
- have a readable commandline representation;
- can be used in pattern-matching for declarative condition checking;
- are a common currency for Scala data transformation.

Scala interface

Complete example:

```
case class Dimuon(mass: Float, px: Float, py: Float, pz: Float)
{
  def momentum = Math.sqrt(px*px + py*py + pz*pz)
  def energy = Math.sqrt(mass*mass + momentum*momentum)
}
```

Random access reader (FreeHep version and iterators are similar):

```
val dimuons = NativeRootTTreeReader[Dimuon](
  "TrackResonanceNtuple.root", "TrackResonanceNtuple/twoMuon")
```

The template resolution (`[Dimuon]` in Scala means `<Dimuon>` in Java/C++) calls a macro that creates a custom factory for the user's `Dimuon` class.

Alternatives are:

- Java runtime reflection (slower),
- putting all user operations in a sublanguage (`TTree::Draw`),
- requiring the user to set up the boilerplate.

Hadoop example

```
case class Dimuon(mass: Float, px: Float, py: Float, pz: Float) {
  def momentum = Math.sqrt(px*px + py*py + pz*pz)
  def energy = Math.sqrt(mass*mass + px*px + py*py + pz*pz)
}
class DimuonWritable extends ValueWritable[Dimuon]
class DimuonInputFormat extends RootInputFormat[
  Dimuon, DimuonWritable]("TrackResonanceNtuple/twoMuon")
```

Hadoop needs objects wrapped in `Writables` because it uses custom serialization methods. Creating this subclass invokes a macro to write them.

Hadoop example

```
case class Dimuon(mass: Float, px: Float, py: Float, pz: Float) {
  def momentum = Math.sqrt(px*px + py*py + pz*pz)
  def energy = Math.sqrt(mass*mass + px*px + py*py + pz*pz)
}
class DimuonWritable extends ValueWritable[Dimuon]
class DimuonInputFormat extends RootInputFormat[
  Dimuon, DimuonWritable]("TrackResonanceNtuple/twoMuon")
```

Hadoop needs objects wrapped in `Writables` because it uses custom serialization methods. Creating this subclass invokes a macro to write them.

```
class TestMapper extends Mapper[KeyWritable, TwoMuonWritable,
                                IntWritable, TwoMuonWritable] {
  override def map(key: KeyWritable, value: TwoMuonWritable,
                   context: Context) {

    // using pattern-matching to define "ttreeEntry" and "mass":
    val KeyWritable(ttreeEntry) = key
    val ValueWritable(TwoMuon(mass, _, _, _)) = value

    // passing to the reducer, keyed on binned mass
    context.write(new IntWritable(mass.toInt), value)
  }
}
```

Spark example

```
case class Dimuon(mass: Float, px: Float, py: Float, pz: Float) {
  def momentum = Math.sqrt(px*px + py*py + pz*pz)
  def energy = Math.sqrt(mass*mass + px*px + py*py + pz*pz)
}
```

Add a `rootRDD` method to `SparkContext` via pimp-my-library:

```
import org.dianahep.scaroot.spark._

val inputRDD = sc.rootRDD[Dimuon](
  "TrackResonanceNtuple*.root", "TrackResonanceNtuple/twoMuon")

val histogram = inputRDD.filter(_.mass > 60.0).map(_.mass.toInt).
                  countByKey()
```

- I'm internally passing my `RootInputFormat` to Spark's `sc.newAPIHadoopRDD`, but unfortunately this requires `RootInputFormat` to have a zero-argument constructor. Scala inserts a hidden constructor argument to pass data to my macro, to overcome the JVM's type erasure.
  Workaround: write a custom RDD class.
- Also, Hadoop's serialization is ignored by Spark; use Kryo.

Future direction(s)

- The pure-Java FreeHEP-ROOTIO is nice, but it can only read from a local filesystem.
  - I could alter it to add support for `FSDataInputStream` (HDFS) and **xrootd4j**, but there's no guarantee that it will handle remote seeking efficiently.
- Therefore, I want to make a native solution bug-free.
  - Testing **BridJ**, which is newer than **JNA** and is C++-aware.
  - **JNAerator** generates bindings for **JNA** or **BridJ**.
    - Attempt to convert ROOT's header files to Java: spent almost 100% of 32 CPUs for 3 hours before giving up.
  - Worst case: external process piping Avro data.
    - I've done it before, very familiar with Avro serialization.
    - By-product: we'd have a general purpose ROOT-to-Avro converter.
- Might use `TProcess::Declare` (C++ compile-time macros).
- Want to process TTrees with objects, possibly CMS's FWLite.

Attempting to use JNAerator on ROOT: