Accessing ROOT and C++ functions in Spark

Jim Pivarski

Princeton University — DIANA

March 14, 2016

My work on integrating ROOT and Spark has split into:

ScaROOT: Call ROOT functions (and arbitrary C++) from Scala, and therefore Spark, Hadoop, etc.

root2avro: Bulk data flow from ROOT files to other file formats or streaming into Spark, Hadoop, etc.

For the last few weeks, root2avro has been my main focus, but I recently got ScaROOT into usable shape.

root2avro isn't a special case of ScaROOT for reasons you'll see in a moment.

Documented with unit tests and examples on the GitHub wiki page (`https://github.com/diana-hep/scaroot`).

ScaROOT

Call ROOT or arbitrary C++ code from Scala.

Motivation

ROOT is a popular framework for high energy physics data. Most "big data" frameworks, such as Hadoop and Spark, are implemented in Java or Scala. ScaROOT allows you to call ROOT functions from Scala so that ROOT can be used to perform calculations in a big data workflow.

This is for accessing ROOT library functions. See root2avro for a high-throughput data feed.

ScaROOT serves the same purpose as PyROOT, which provides access to ROOT functions in Python, though the interface differs because Python is more dynamic than the Java Virtual Machine (JVM).

Examples

ScaROOT works by linking Scala traits (abstract interfaces) to C++ code that may contain ROOT calls.

```
import org.dianahep.scaroot.RootClass

// Scala trait (abstract interface):
trait ChiSqProb {
  def apply(chi2: Double, ndof: Int): Double
}
```

Interface

PyROOT dynamically makes proxies to ROOT functions when they're called, so that working in Python is approximately the same as working in CINT/Cling (replacing "->" and "::" with ".").

Interface

PyROOT dynamically makes proxies to ROOT functions when they're called, so that working in Python is approximately the same as working in CINT/Cling (replacing "->" and "::" with ".").

Scala (and Java) are compiled languages, so field names of classes have to be declared in advance. Classes can be defined on the fly (in a private ClassLoader), but the main program can only use them if they adhere to a predefined interface (abstract class).

Interface

PyROOT dynamically makes proxies to ROOT functions when they're called, so that working in Python is approximately the same as working in CINT/Cling (replacing "->" and "::" with ".").

Scala (and Java) are compiled languages, so field names of classes have to be declared in advance. Classes can be defined on the fly (in a private ClassLoader), but the main program can only use them if they adhere to a predefined interface (abstract class).

We therefore ask the user to define a Scala interface that is satisfied by a C++ class. The C++ class can use any ROOT functions.

Example

```
import org.dianahep.scaroot.RootClass

// Scala trait (abstract class with virtual methods):
trait ChiSqProb {
  def apply(chi2: Double, ndof: Int): Double
}

// C++ class definition that satisfies the interface:
val chiSqProbClass = RootClass[ChiSqProb]("""
class ChiSqProb {
public:
  double apply(double chi2, int ndof) {
    return ROOT::Math::chisquared_cdf(chi2, ndof);
  }
};
""")

// Create an instance:
val chiSqProb = chiSqProbClass.newInstance

// And use it:
chiSqProb.apply(53.8, 50)    // or just chiSqProb(53.8, 50)
                             // because 'apply' is 'operator()'
0.6689797343068249
```

How it works

- The use of "`RootClass[ChiSqProb]`" in the code invokes a compile-time macro that generates custom hooks for the "`ChiSqProb`" trait.
- These hooks connect to ROOT through JNA (same process, no serialization, just a memory-to-memory copy).
- ROOT's `TInterpreter` is invoked to compile the code and give the JVM a direct pointer to the class.
- At execution time, the method arguments are copied from Java's memory heap to C++'s, the function is called, and the return values are copied back, all within a single process.

Who would use it

- Users doing high-throughput calculations (e.g. skimming) on Spark: the round-trip time between Spark and ROOT is minimized.
- Or a library of predefined interfaces provided to the user.

Capabilities

- The C++ code is not fixed until objects are made at runtime. This string could be dynamically generated for "just in time" flexibility and efficiency.
- `RootClass` objects are serializable (as the C++ string!) so that they can be passed to a remote Spark workflow.

Limitations

- The class must have a zero-argument constructor.
- Parameter types and return types must be primitives (numbers, strings, or an opaque `com.sun.jna.Pointer` to C++ data).
- (The appropriate version of) ROOT needs to be installed across the Spark cluster, accessible via `LD_LIBRARY_PATH` like any other natively compiled program.

Status

- Works, but has *no error checking.* Any mistake causes segmentation fault (will fix, of course).
- Has not been tested in Spark yet. (Does anyone have an application they're eager to try? Want to work with me?)

Possible extensions

- Could build a library of common functions (histograms and such) by hand.
- Could exhaustively search the space of ROOT functions by a graph traversal on `TInterpreter`. This would be a better analogy with PyROOT, but statically compiled.

Why is this not a data feed (root2avro)?

- With a primitives-only interface, each complex data structure (CMSSW, ART, Bacon, ...) would have to be adapted by hand. We want to recognize these structures automatically.

Status of root2avro

- All known data types that can be found in TTrees are handled, with a framework that can easily add new ones as they're discovered.
- Successfully examined and converted a Bacon-tuple to JSON.
- All JSON-writing functions have been supplanted with Avro-writing functions, so we should do Bacon-to-Avro soon.

What's next?

1. Convert Matteo and Christina's Bacon-tuples to Avro files, load into a Spark cluster.
2. Fix any bugs and segmentation faults along the way.
3. Copy ROOT data directly into the JVM using a technique similar to ScaROOT.
4. Wrap this up in a Spark InputRDD and/or DataFrame (they're different).

Example Avro schema generated from a Bacon-tuple

```
{"type": "record",
 "name": "Events",
 "fields": [
   {"name": "GenParticle", "type": {"type": "array", "items": {"type": "record",
    "name": "baconhep::TGenParticle",
    "fields": [
      {"name": "parent", "type": "int"},
      {"name": "pdgId", "type": "int"},
      {"name": "status", "type": "int"},
      {"name": "pt", "type": "float"},
      {"name": "eta", "type": "float"},
      {"name": "phi", "type": "float"},
      {"name": "mass", "type": "float"},
      {"name": "y", "type": "float"}
    ]
   }}},
   {"name": "LHEWeight", "type": {"type": "array", "items": {"type": "record",
    "name": "baconhep::TLHEWeight",
    "fields": [
      {"name": "id", "type": "int", "doc": "parton flavor PDG ID"},
      {"name": "weight", "type": "float", "doc": "generator-level event weight"}
    ]
   }}},
   {"name": "Electron", "type": {"type": "array", "items": {"type": "record",
    "name": "baconhep::TElectron",
    "fields": [
      {"name": "pt", "type": "float", "doc": "kinematics"},
      {"name": "eta", "type": "float", "doc": "kinematics"},
      {"name": "phi", "type": "float", "doc": "kinematics"},
      {"name": "scEt", "type": "float", "doc": "supercluster kinematics"},
      {"name": "scEta", "type": "float", "doc": "supercluster kinematics"},
      {"name": "scPhi", "type": "float", "doc": "supercluster kinematics"},
      {"name": "ecalEnergy", "type": "float", "doc": "ECAL energy"},
      ...
```