ROOT-Spark integration

Jim Pivarski

Princeton Univeristy — DIANA

March 7, 2016

Strategy

Scope

Spark is just one big data technology, but almost all of them share a common difficulty: the data backbone is implemented in the Java Virtual Machine (JVM) and ROOT is C++.

This doesn't mean that users must use Java (I use a mix of Scala and Python), but it does mean that data must be efficiently converted into a JVM-friendly format to be used at a large scale.

Methods

I've been keeping three options open:

1. Pure Java using FreeHEP (reimplementation of ROOT in Java by Tony Johnson).
2. Direct ROOT C++ $\rightarrow$ JVM connection with JNI (or an abstraction layer on top of it, such as JNA).
3. ROOT C++ as an external process, passing serialized data through a pipe.

Status

1. Pure Java using FreeHEP: works, but I have concerns about its efficiency for non-local files (e.g. xrootd). When all three options are ready, this method should be featured in a performance study.
2. Direct ROOT C++ $\rightarrow$ JVM: this now works as well. The random segmentation faults I was experiencing were related to interference between ROOT's signal handlers and Java's signal handlers.
3. ROOT C++ as an external process: of course this should work, and I've been putting my efforts here so that we at least have a working baseline. I'm using Avro to stream data from the ROOT process to the JVM process. Avro is an efficient binary format with bindings in many languages.

I'm further considering a merger of options #2 and #3: in addition to the Avro bridge, we could run ROOT as an external process with data transferred via shared memory ("off-heap," a technique used by high-frequency traders).

Status, continued

What have I actually been doing?

Consuming the entire ROOT data model and providing the appropriate translations to Avro. Required for both #2 and #3.

- Simple primitive branches (TLeaf).
- Primitive arrays, multidimensional arrays preserving substructure: e.g. "`Double32_t fMatrix[4][4]`" becomes a four-element array containing four-element arrays of numbers.
- Primitive arrays with an external counter variable, such as `fNTracks`.
- STL vectors (and maps? does ROOT use `std::map`?).
- User-defined classes, such as the `Event` example, CMSSW data with FWLite, ART data with the appropriate libraries.
- Pointers as nullable objects: e.g. "`Double32_t *fClosestDistance`" becomes "`["null", "double"]`," a value that could be null (`None` in Python) or a number.

(See `https://github.com/diana-hep/root2avro`.)

How do I expect users to use this?

This is infrastructure necessary to make an efficient Spark InputRDD. The user can use Spark directly, PySpark, or SparkR as he or she sees fit. However,

- For big jobs, like ntuple-skimming (my target use-case), they'll find Spark's JVM-to-Python bridge (Py4J) to be a source of inefficiency.

  It would be much more efficient to do

  ```
  dataset.filter(event => event.numMuons >= 2  &&
                          event.muons(2).pT > 20)
  ```

  on the Scala side before attempting heavy interactive analysis in Python. (Note the difference in syntax above.)
- If analysts really need one language for everything, it wouldn't be hard to provide a Jython skin over native Spark, to be used only for big datasets (skimming).