Laurelin: A ROOT I/O Implementation for Apache Spark

Andrew Melo
Vanderbilt University

May 9th, CHEP 2023 Track 6

Introduction

- Why Apache Spark
- Spark DataSources and ROOT File Format
- Laurelin
- Performance Comparisons

Why Apache Spark

- Apache Spark is a widely-used "multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters"
- Spark lazily evaluates queries by making a query plan, optimizing the plan, then finally compiling it down to JVM bytecode.
  - Externally-developed extensions enable compiling the query to vectorized x86[1,2] or CUDA[3] code
- To enable better interoperability between the burgeoning Python-based ML ecosystem, Spark supports transferring data to/from Python as Arrow-buffers,
- Recently-released Spark3.4 added a client-server Python client, meaning a single large Spark cluster can be instantiated and shared between multiple users

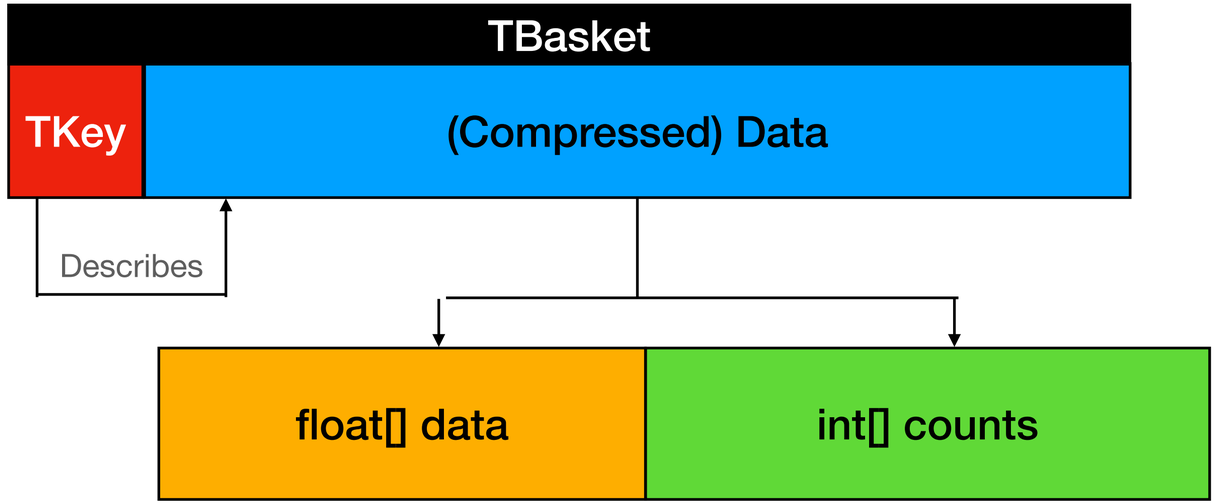
[1]https://github.com/oap-project/gazelle_plugin
[2]https://github.com/oap-project/gluten
[3]https://nvidia.github.io/spark-rapids/

Spark DataSources

- Spark has native support for many popular data sources, both file-based (e.g. Parquet) and databases (e.g. Hive, JDBC, Kafka)
- There is additionally a DataSource API which provides a (somewhat) stable interface for others to provide support for other data source/sinks
  - Support the sort of data setters/getters you expect, as well as some optional interface mixins to support various optimized "shortcuts" -- i.e. Spark can inform a DataSource that it only wants columns X and Y
- Importantly, Spark can consume and produce Arrow buffers directly

TTree to Arrow Buffers

- Data in TTrees are stored within TBranches, which eventually leads to TBaskets with the actual payload data itself
- These TBaskets consist of an array of data as well as optional array of counts (for jagged/ragged arrays)
- Conveniently, this is very close to the Arrow format (mod. endianness, etc)



Laurelin

- Laurelin is a Java-based ROOT I/O implementation to read/write TTrees to/from ROOT files
- Since TBaskets are roughly in Arrow format, and Spark has native internal support for Arrow-based columnar accesses, Laurelin "just" has to understand (de)serializing TTree and other objects to locate the Baskets
- Laurelin splits the ser/der code from the Spark code, allowing the library to be used in non-Spark JVM applications

Laurelin, one of the Two Trees of Valinor in Lord of the Rings

Image Credit

Read Path

- Reading ROOT files involves writing manual read code for enough objects (TFile, TKey, etc..) to bootstrap interpreting the Streamers, which can then be used to dynamically interpret all other classes (TTree, TBranch, TBasket..)
  - Much of Laurelin is responsible for the nuances of this deserialization
- With the Streamers loaded, we simply
  - Decompress the TBaskets into off-heap buffers
  - Perform an endianness swap
  - Construct a Spark ArrowColumnVector to wrap the data/count buffers

Write Path

Much more difficult

- Even if you know up front all of the data you want to write, you cannot write a ROOT file in one pass
    - One side effect is it's not (easily) possible to have multiple threads writing to the same ROOT file
- While reading, we can simply skip over and ignore data we're not interested in, not the case for writing
    - E.G. TBranch inherits from TAttFill, which we have to faithfully output, otherwise other implementations cannot parse the resulting file

Results

- Moving to Arrow-based off-heap buffers instead of hand-rolled on-heap buffers led to a 40% improvement in speed in some cases
  - Reduced GC pressure as well as not needing to memcopy buffers at the syscall boundary
    - Currently using a java-based LZMA decompression library, moving to that buffer would allow a true off-heap only data path
- Read performance is within 10% of Spark's Parquet implementation
- Files written by Laurelin can not currently be read using ROOT/uproot, work is continuing to fix remaining issues

Future Work

- There is a lot of work to be done on the metadata ser/der paths to speed them up
  - Using BCEL to dynamically generate Java class implementations based on streamer definitions
- RNTuple support
- Requesting/upstreaming parity between DataSource API and functionality that builtin file support has
  - e.g. when writing a file, being provided the # of rows upfront