Laurelin: Java-native ROOT I/O for Apache Spark

Andrew Melo[1,*] and *Oksana* Shadura[2] for the CMS Collaboration

[1]Vanderbilt University, Nashville, TN 37240, USA
[2]University of Nebraska, Lincoln, NE 68588, USA

Abstract. Apache Spark[1] is one of the predominant frameworks in the big data space, providing a fully-functional query processing engine, vendor support for hardware accelerators, and performant integrations with scientific computing libraries. One difficulty in adopting conventional big data frameworks to HEP workflows is the lack of support for the ROOT file format in these frameworks. Laurelin[6] implements ROOT I/O with a pure Java library, with no bindings to the C++ ROOT[2] implementation, and is readily installable via standard Java packaging tools. It provides a performant interface enabling Spark to read (and soon write) ROOT TTrees, enabling users to process these data without a pre-processing phase converting to an intermediate format.

1 Introduction

High Energy Physics (HEP) experiments, like those performed at the Large Hadron Collider (LHC), generate an enormous amount of data which must be captured, stored, refined, and later analyzed by thousands of users. These data are produced from the collisions of billions of subatomic particles at speeds approaching the speed of light, as well as sophisticated simulations of the various expected physical processes. Each collision (either real or simulated) produces decay products which subsequently travel outwards from the interaction point and pass through finely-segmented detectors. As the particles travel outwards, they can deposit energy in the detectors which then transmit the measurements to a dedicated computing farm for storage. Once these collisions are safely on archival storage, they are replicated to a globally-distributed computing grid for further processing. Ultimately, physicists are interested in the types and properties of particles radiating from the interaction point, but the detector only provides information about energy deposits passing through itself. Therefore, reconstruction passes are performed over the initial data to ‘connect the dots’ of detector hits to determine the most probable types and trajectories of particles passing through the detector. Eventually, this process results in datasets that are apt to probe the physical processes that occurred at the interaction point.

Each collision within HEP is independent of each other. These collisions occur when Bunches, each comprising $> 10^9$ particles, are targeted at each other in the center of the detectors. Some of these particles will collide with each other and radiate decay products outwards which are detected together. The atomic unit of these datasets is therefore an *Event*, which is all the detected products emanating from a single bunch crossing. Because events are statistically independent of each other, processing datasets of events is trivially parallelized.

[*]e-mail: andrew.m.melo@accre.vanderbilt.edu

© The Authors, published by EDP Sciences. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (http://creativecommons.org/licenses/by/4.0/).

Contained within each event are the measurements that occurred while the decay products passed through the detector. In earlier stages of processing, these measurements are closely tied to the outputs of the detector, but latter stages will contain the refined and calibrated properties of the physical particles detected in the event. Conceptually, this can be likened to a table, where the rows are events and the columns are properties.

There are typically two distinct phases in the life cycle of HEP datasets. First, the raw data needs to be processed into a format usable by analyzers. This processing can take up a significant amount of the annual computing budget for an entire experiment, so it is performed infrequently. Once the analysis-appropriate data has been produced, users then examine these data for insight into the underlying physical processes that occurred in the collisions. This data exploration is an iterative process —physicists need results from the data to guide their next steps. While the initial reconstruction passes may use their CPU in one bulk task, analyzers will typically execute their smaller tasks much more frequently as they explore the data. The time it take to complete these iteration cycles directly affects the rate at which analyzers can produce results, since they effectively cannot proceed until their tasks have completed. As experiments record data at ever-increasing rates, the HEP community must explore improvements to our workflows to maintain or reduce the latency incurred by processing data.

Since the early 2000s, communities outside of HEP have dealt with a similar explosion in the amount of data stored and processed. The mid 2010s led to the proliferation and commoditization of Big Data software and techniques, designed to extract valuable information from ever-increasing data volumes at incredible speeds. These software are used in non-HEP communities to process data at impressive scales, and the widespread adoption of these tools has led to a robust ecosystem around these tools. By adopting these tools for HEP research, we would be able to both leverage the significant investments made by others and further integrate the HEP community in the broader community of data scientists.

One of the predominant big data tools is Apache Spark[1], which was originally released in 2014. Its principal abstraction is the Resilient Redundant Dataset (RDD) [3], which is an immutable dataset consisting of rows subdivided into columns. The power of RDDs [4] comes from their immutability. RDDs can only be produced by loading data from an external source or by performing a transformation on an existing RDD. This property enables fault-tolerance, natural parallelization, and faster iterative analysis. Users provide a computation graph to Spark in the form of the desired transformations they would like performed on the data. To execute this graph, Spark divides the input data into partitions which are sub-ranges of the initial dataset. These partitions can be distributed over a number of nodes that will each perform transformations over their partitions. If a node fails during the process, Spark is able to recreate the missing partitions by re-executing the graph on the subset of missing source partitions. Additionally, Spark can accelerate analyses by caching intermediate results. Subsequent requests that depend on those results will pull them from the cache instead of regenerating them from scratch.

Because many tools like Spark were developed outside of HEP, there is little native support for the ROOT file format in the Big Data ecosystem. This obviously is a significant barrier to adoption of these tools by the HEP community. ROOT is tailored to fit HEP's specific needs so migrating to a new format would mean discarding those features. Additionally, the data volumes involved mean it is impractical to maintain a 2nd copy of these data in an alternate format. To enable exploration of the potential of these novel toolkits, we need to teach them to interpret our datasets.

Laurelin [6] is a Java-based implementation of ROOT file I/O. Its Spark DataSource API [7] implementation allows Spark to access ROOT files as a first-class citizen, though there is a clean separation between the Spark API and the underlying ROOT I/O packages, which allows Laurelin to be usable in any JVM-based programs. Laurelin enables interpreting ROOT files as Spark DataFrames [8] (a type-aware extension of RDDs). It does this by mapping each row of the DataFrame to an Event in ROOT, and each column in the DataFrame to a branch in ROOT. The resulting DataFrame can then be composed in an execution graph like any other Spark DataFrame. Additionally, since ROOT files are stored column-wise on-disk (e.g. one column of multiple rows are stored in contiguous bytes), Laurelin is able to expose a column-wise view of the data to Spark. Column-wise processing of data can be significantly more efficient than row-wise processing, particularly when not all the data from each row is needed. In HEP, this is the exception rather than the rule because most analyses will focus on a small fraction of the particles from any given Event.

2 Architecture

While Laurelin is primarily intended to be a Spark plugin, its design strives to keep a strict separation between the higher-level Spark functionality and the lower-level handling of the ROOT files themselves. This enables ROOT files to be accessible from arbitrary JVM-based programs without pulling in unnecessary Spark dependencies. Since ultimately Laurelin is a compatibility library, we will briefly present the software being integrated.

2.1 ROOT file format

The ROOT file format is designed to support a near-limitless ability to serialize and deserialize arbitrary C++ objects. Since HEP experiments often exist for multiple decades, it's critical that old files can be interpreted by new software and vice-versa. To achieve these goals, ROOT files are self-describing, meaning each file also contains metadata informing the reader how to deserialize a string of bytes back into an in-memory C++ object. Importantly, these metadata (called Streamers), describe not only user-provided C++ classes stored in the files, but additionally describe all the relevant ROOT metadata classes themselves. These Streamers are very powerful, enabling not only 'schema evolution', but exposing fine-grained control of the serialization process to the users, even exposing a C++ interface to the user for advanced use-cases. These 'custom streamers' are unimplemented in most 3rd-party ROOT I/O implementations, but their use in analysis-level formats are increasingly rare due to their focus on fundamental types. For example, CMS' analysis-level NanoAOD format [9] consists exclusively of fundamental types (or arrays thereof).

ROOT files can store an arbitrary number of objects of varying types in a filesystem-like tree of directories. ROOT's columnar dataset functionality is represented by the 'TTree' class, whose columnar schema is defined by a number of (possibly nested) 'TBranch' and 'TLeaf' objects. Taken together, these three classes store all the relevant metadata of a particular columnar dataset within a file. These classes contain pointers to TBasket objects which contain all the data stored by the TTree. As data is appended to the TTree, the incoming data corresponding to each column is stored in a separate in-memory buffer. Once size or other conditions are met, the buffer is compressed, stored in a TBasket and written to disk. If the column is a fundamental type like an integer, the buffer contains an array of the big endian representation of the values for a range of rows. If, however, the column is a C++ class, the array will either contain the Streamer-encoded representation of the objects, unless the user has chosen to 'split' the column, which means each field of the class is stored in nested columns with separate baskets.

One hallmark of HEP datasets is the prevalence of variable-length columns. When collisions occur, the underlying physical processes can produce a wide distributions of decay products. For example, collisions can produce a number of electrons or photons which is different on a per-collision basis. Trying to represent these data in a strictly tabular representation by adding N additional columns to reserve enough space for N physical objects is quite wasteful, since many events will not have the maximum amount of decay products. Instead, ROOT represents these columns by storing all the values in a single buffer and maintaining a separate buffer that stores the number of values for each row. These count buffers consist of small positive integers and are therefore highly compressible.

2.2 Apache Spark DataSource API

The Spark DataSource API is a rich extension point to enable access to various data sources like files or databases. Plugins implementing this API are not accessed directly by users. Instead, a user using these plugins will create a DataFrame (an extension of the Spark RDD paradigm) and inform Spark which format they would like to use. When data is either read or written to the DataFrame, it will then delegate the actual input/output to the plugin.

Spark's DataSource API allows plugin developers to advertise what functionalities they support. This allows plugins to expose significant optimizations to the Spark query engine. For example, each step in a Spark query declares its input and output columns For example, the default DataSource interface expects that plugins provide a row-wise view of their data. Laurelin uses the *supportColumnarReads* notation to inform Spark that it can provide a columnar view of the data instead. Since ROOT's on-disk format for TBaskets [10] is very close to the in-memory format Spark uses, a decompression and metadata manipulation are all that's needed to injest these TBaskets directly into Spark. As Spark evolves, new optimizations become available, allowing data sources to prune data early, report on the statistics of the data, or expose the locality of the data within the cluster.

Spark DataSource plugins execute in two distinct environments —the driver, and the executors. When a query is executed, the driver is first responsible for dividing the input dataset into partitions. Spark then distributes these partitions to the available executors, which can be distributed across a cluster of physical nodes. Using this partitioning information, the executors then load the appropriate subsets of data and pass them upwards to Spark that then executes the user's query.

3 Components

Laurelin is divided into several interconnected components. First, the *root_io* package both handles low-level file access and re-implements ROOT Streamers [11], allowing Laurelin to parse ROOT metadata and locate the TBaskets within the files. Second, the 'interpretation' package is responsible for converting raw bytes from TBaskets into Java-intelligible values. Finally, *spark_ttree* deals with the Spark API, consuming ROOT files and exporting the data to Spark as-requested.

3.1 ROOT I/O

The re-implementation of ROOT I/O in *root_io* is, by-far, the most complicated component of Laurelin. Ultimately, ROOT files are containers for serialized objects (in the generic sense). It is relatively straightforward to find the byte-offset of any object within a ROOT file, but deserializing these objects back into an in-memory representation requires a lot of care. Because there is no formal definition of the serialization algorithm, 3rd-party implementations must reverse-engineer the correct procedure. The nearly three-decade heritage and evolution of ROOT has resulted in a number of optimizations which must be correctly handled to correctly parse these files. For example, in the 90's it was common to encode byte offsets as

32-bit integers, which have a maximum value of 4 gigabytes. As disk and data volumes increased, it became necessary to support larger files, so these offsets were increased to 64-bit integers. A conforming implementation of ROOT I/O must seamlessly handle both cases, as well as a number of other optimizations.

Fortunately, ROOT provides deserialization instructions for every class contained within a ROOT file. These instructions, known as Streamers, are fundamental to ROOT's ability to ensure forwards and backwards compatibility. The header of each ROOT file contains the offset to a TStreamerInfo [11] object, which contains a Streamer for each class stored within the file. To deserialize the object, one iterates over the TStreamerElement objects within the Streamer, reading bytes from the file and interpreting them according to the current TStreamerElement.

Without these Streamers, it would be difficult to write an independent implementation of ROOT deserialization. New revisions of ROOT can add/remove fields to its C++ classes, which changes the on-disk format. Without Streamers, downstream consumers of these files would then be required to track these changes and maintain separate deserializers for each version of ROOT. Instead, Laurelin performs deserialization by first manually bootstrapping the classes required to deserialize the Streamers. Once the Streamers have been parsed, it can then proceed with loading any other object within the file.

Though ROOT files can contain arbitrary objects, Java is strongly typed and requires that all variables have a type and all types must be defined at compilation time. Thus, it is not possible to directly create new Java classes using pure-Java constructs. There are Java libraries which can directly produce JVM bytecode to create a class at runtime, but these can be difficult to use and reference from non-dynamic classes. For example, it is not possible to access members of dynamically-constructed classes without using cumbersome reflection techniques.

Instead of exposing Java classes with an 'is-a' relationship to the underlying C++ classes, Laurelin instead populates separate 'Proxy' objects with the ROOT-deserialized data. Proxy objects store information about both the class itself and its member variables. Important ROOT classes like TTree are then implemented as standard Java classes which encapsulate a Proxy loaded from a file. The accessors for these classes then read values from the encapsulated Proxy object. Combining standard Java classes with the dynamic nature of the data loaded from ROOT files greatly decreases the complexity of the higher level interfaces, since they are completely unaware of the underlying complexities of the deserialization scheme. Additionally, these classes have no dependency on Spark internals, so they would be usable by other JVM-based projects who wanted to interact with ROOT files. Since JVM-based languages play a large role in the big data ecosystem, these classes enable additional opportunities for integration.

3.2 Interpretation

The 'interpretation' component of Laurelin is tasked with converting the raw byte-arrays stored in TBaskets into their equivalent Java representations. ROOT supports rich column types, meaning there can be a number of base datatypes with potentially multi-dimensional variable-length arrays, so properly decoding these values takes some care. Importantly, when Spark requests a batch of rows, it expects each column to start and stop at the same row. However, ROOT flushes each TBasket independently, meaning the row boundaries are often unaligned. The interpretation layer handles slicing the value and count buffers to align all the buffers in a given batch.

Additionally, ROOT encodes variable-length column types as a value buffer and a 'counts' buffer, which represents the length of the array in each row. Since the counts buffer contains small positive integers it is highly compressible, which is ideal for archival purposes. The downside to this scheme is that accessing row N requires summing up the previous $N-1$ row counts to calculate the offset into the values buffer. Therefore, it is advantageous to transform the counts buffer into an *offset* buffer, where $offset[N]$ is the offset of the first element of row N, and $offset[N+1] - offset[N]$ is the number of elements in the $Nthrow$. In order to find the values in the $Nthrow$, both the *counts* and *offsets* representations are equivalent. However, the *offset* representation can be calculated once for an entire batch of data, while the *counts* representation needs to perform the summation for every access, which is much more time consuming.

Coincidentally, this *values* and *offset* representation is a common way to store columnar data in-memory. Apache Arrow is a 'language-independent columnar memory format for flat and hierarchical data, organized for efficient analytic operations on modern hardware like CPUs and GPUs', that has been widely adopted by the big data ecosystem to interchange columnar data between frameworks. Laurelin's interpretation component returns buffers that are nearly identical to Arrow buffers (the major difference is that Arrow is little endian by default while both ROOT and Java are big endian).

3.3 Spark DataSource

The top-level component of Laurelin is *spark_ttree* which implements the Spark DataSource API. This component satisfies Spark's requests to open files, divide the files into partitions, load the partitions into memory, and finally retrieve values from the in-memory representation. The DataSource plugins are encapsulated within Spark's DataFrame objects, so the lifecycle and order of the calls are completely dependent on the Spark framework and not the user.

When a user requests a new DataFrame backed by ROOT files, Spark will greedily instantiate a 'Table' class to represent the dataset backed by those files. At this time, Laurelin spawns a parallel task to open all the ROOT input files and record their schemas that the execution engine requires to evaluate queries (Figure 1). Once a query arrives, Spark pushes down the requested columns to Laurelin that then divides the input dataset into partitions. When this happens, Laurelin stores the offset and lengths of the TBaskets, allowing the executors to simply open the files and seek to the proper position to read the data. This is an important optimization to prevent reading the metadata every time a partition is loaded. Spark then distributes these partitions to the executor processes.

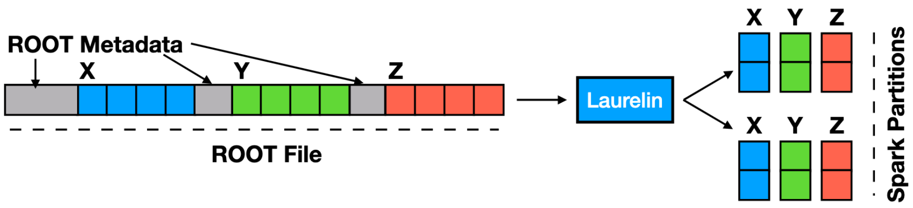
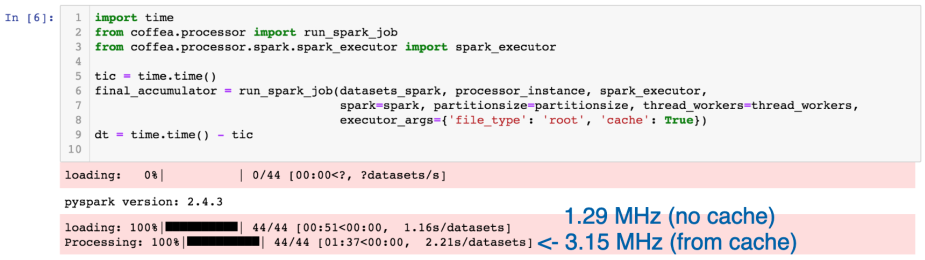


Figure 1. Schema of Laurelin ROOT file's transformation into Spark DataFrames.

During execution, Laurelin receives the partition objects from the driver process and attempts to make the data available as quickly as possible. Since the driver provided the offset and lengths of the data, the executors are able to skip any processing of the ROOT metadata and simply open the files and seek to the position of the TBaskets. Each TBasket is read, decompressed and interpreted asynchronously in a thread pool, which decreases the latency compared to a strictly serial process. Laurelin immediately returns a ColumnBatch to Spark, but all accesses are gated by Java futures tied to the asynchronous processes, ensuring further computation will block until the baskets are ready.

At this point, Laurelin has loaded the ColumnBatch into memory and, from Spark's perspective, this data is indistinguishable from data produces from other sources.

Laurelin was integrated in the python-based Coffea[5] analysis framework and is available through the wrapper for submission jobs for processing Spark datasets. In Figure 2 we would like to show how it could be used within Coffea framework, as a part of a simple benchmark, while reading 500 GB of ROOT TTree data over XRootd [12] through Laurelin.



```
In [6]:
import time
from coffea.processor import run_spark_job
from coffea.processor.spark.spark_executor import spark_executor

tic = time.time()
final_accumulator = run_spark_job(datasets_spark, processor_instance, spark_executor,
                                  spark=spark, partitionsize=partitionsize, thread_workers=thread_workers,
                                  executor_args={'file_type': 'root', 'cache': True})
dt = time.time() - tic
```

```
loading:   0%|          | 0/44 [00:00<?, ?datasets/s]

pyspark version: 2.4.3

loading: 100%|██████████| 44/44 [00:51<00:00,  1.16s/datasets]
Processing: 100%|██████████| 44/44 [01:37<00:00,  2.21s/datasets]
```

Figure 2. Evaluation of performance of Laurelin, executed in Coffea framework.

4 Conclusions and Next Steps

This paper presents Laurelin, which allows users to read ROOT TTrees directly into Apache Spark DataFrames. As one of the predominant 'Big Data' frameworks, Spark benefits from the scale of its community, and has deep integrations with both popular software libraries and hardware accelerators. By providing a way to directly ingest ROOT files into these frameworks, Laurelin enables the exploration of these technologies to accelerate the process of science within the HEP community.

The next milestone for Laurelin is to enable the ability to write ROOT files. Since ROOT is the *lingua franca* of HEP, write functionality is required to fully integrate Spark with the HEP ecosystem. In addition to write support, Laurelin needs to implement more DataSource API optimizations. For example, by providing more information of the structure of ROOT files, the Spark query engine can produced highly optimized joins of two distinct datasets. Often analyzers need to add additional columns to existing datasets. Because there is no efficient way to join multiple datasets, analyzers typically will produce new datasets which contain all the data they need. By providing a transparent and performant join functionality, experiments could lessen the disk space requirements from final analyses.

5 Acknowledgements

This work is supported in part by NSF Award PHY-1806612. Additionally, the author acknowledges the significant contributions from Viktor Khristenko for his work on spark-root [13] which inspired this project, Jim Pivarski for his implementation of much of the interpretation code and helpful assistance decoding the intricacies of ROOT deserialization, Lindsey

Gray for his valued work testing and debugging Laurelin, as well Oksana Shadura who provided valuable testing insight. Additionally, the author would like to acknowledge the assistance provided from both DIANA-HEP and IRIS-HEP. This work was conducted in part using the resources of the Advanced Computing Center for Research and Education at Vanderbilt University, Nashville, TN, with hardware supported by the Vanderbilt TIPs program.

References

[1] spark.apache.org. (n.d.). *Apache SparkTM - Unified Analytics Engine for Big Data*. [online] Available at: https://spark.apache.org.

[2] Rene Brun and Fons Rademakers, *ROOT - An Object Oriented Data Analysis Framework*, Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996, Nucl. Inst. & Meth. in Phys. Res. A **389** (1997) 81-86.

[3] spark.apache.org. (n.d.). *RDD Programming Guide - Spark 3.0.0 Documentation*. [online] Available at: https://spark.apache.org/docs/latest/rdd-programming-guide.html.

[4] Zaharia, Matei, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. *Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing.*, In 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), pp. 15-28. 2012.

[5] Smith, Nicholas, et al. *COFFEA Columnar Object Framework For Effective Analysis.*, EPJ Web of Conferences. Vol. **245**. EDP Sciences, 2020.

[6] GitHub. (2020). spark-root/laurelin. [online] Available at: https://github.com/spark-root/laurelin [Accessed 1 Mar. 2021].

[7] spark.apache.org. (n.d.). *Data Sources - Spark 3.0.2 Documentation*. [online] Available at: https://spark.apache.org/docs/latest/sql-data-sources.html [Accessed 1 Mar. 2021].

[8] spark.apache.org. (n.d.). *Spark SQL and DataFrames - Spark 2.4.5 Documentation*. [online] Available at: https://spark.apache.org/docs/latest/sql-programming-guide.html.

[9] Rizzi, Andrea, Giovanni Petrucciani, and Marco Peruzzi. *A further reduction in CMS event data for analysis: the NANOAOD format.*, EPJ Web of Conferences. **Vol. 214.**, EDP Sciences, 2019.

[10] Canal, Ph. *ROOT I/O Improvements*. Journal of Physics: Conference Series. Vol. 396. No. 5. IOP Publishing, 2012.

[11] Canal, Philippe, Brian Bockelman, and René Brun. *ROOT I/O: The fast and furious*. Journal of Physics: Conference Series. Vol. 331. No. 4. IOP Publishing, 2011.

[12] Dorigo, Alvise, et al. *XROOTD-A Highly scalable architecture for data access.*, WSEAS Transactions on Computers 1.4.3 (2005): 348-353.

[13] GitHub. (2021). diana-hep/spark-root. [online] Available at: https://github.com/diana-hep/spark-root [Accessed 1 Mar. 2021].