Awkward Arrays to RDataFrame and back

Ianna Osborne, Jim Pivarski

Princeton University

Many thanks to Enrico Guiraud and Enric Tejedor from ROOT team

http://iris-hep.org

OAC-1836650

Outline

Details of the implementation exploiting JIT techniques

Examples of analysis of data stored in Awkward Arrays via a high-level interface of an RDataFrame

Examples of the column definition, applying user-defined transformations and filters written in C++, and plotting or extracting the columnar data as Awkward Arrays

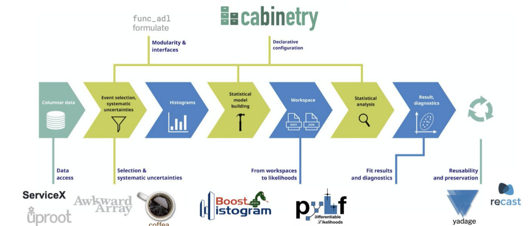
Current limitations and future plans

iris hep

PRINCETON UNIVERSITY

Awkward Arrays and RDataFrame

- Awkward Array is a library for nested, variable-sized data, including arbitrary-length lists, records, mixed types, and missing data, using NumPy-like idioms
  - An example of their use in Python eco-system
- RDataFrame - ROOT's declarative analysis interface
  - No need to go into the details - the audience knows what it is - see Enrico's talk
  - Supports many input formats
- Two very different ways of performing calculations at scale
  - Benefits of the combining both Python and C++
  - Physicists can mix analyses using Awkward Arrays, Numba, and ROOT C++ in memory, without saving to disk and without leaving their environment



iris hep

PRINCETON UNIVERSITY

From Awkward Arrays to RDataFrame: Views

The *ak.to_rdataframe* function presents a view of an Awkward Array as an RDataFrame source

- Awkward Arrays are already JIT-compiled with Numba. Here we are reusing some of the Numba implementation for C++: no performance difference

This view is generated on demand and the data is not copied

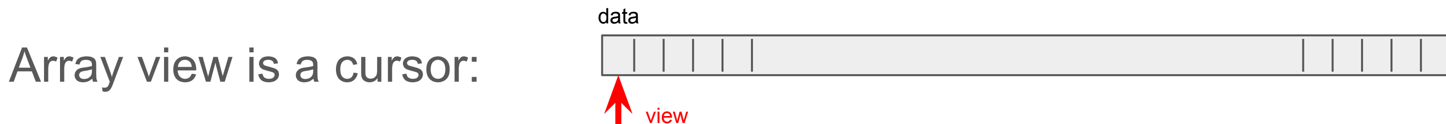
The column readers are generated based on the run-time type of the views

The readers are passed to a generated source derived from ROOT::RDF::RDataSource

From Awkward Arrays to RDataFrame: Data Source

Generated AwkwardArray RDataSource takes pointers into the original array data: a 40-byte ArrayView object is allocated on the stack

Array view is a cursor:



The large-scale array data are not copied

The views are transient, their lifetime is defined by the lifetime of their lookup Python object

C++ code generated by Awkward Array

ak.Array([[1.1, 2.2], [3.3], … []])

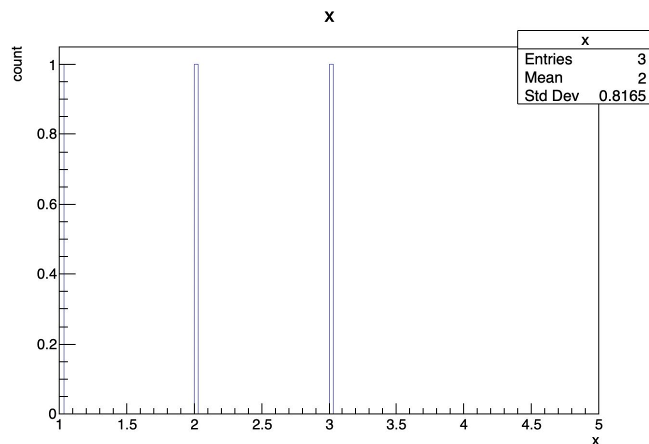
This is what is fed to Cling

```
namespace awkward {
  class ListArray_BgI9cDJVCAw: public ArrayView {
  public:
    ListArray_BgI9cDJVCAw(ssize_t start, ssize_t stop, ssize_t which, ssize_t* ptrs)
      : ArrayView(start, stop, which, ptrs) { }

    typedef NumpyArray_float64_O1I50DFDJTY value_type;

    const std::string parameter(const std::string& parameter) const noexcept {
      return "null";
    }

    value_type at(size_t at) const {
      if (at >= stop_ - start_) {
        throw std::out_of_range(std::to_string(at) + " is out of range");
      }
      else {
        return (*this)[at];
      }
    }

    value_type operator[](size_t at) const noexcept {
      ssize_t start = reinterpret_cast<int64_t*>(ptrs_[which_ + 1])[start_ + at];
      ssize_t stop = reinterpret_cast<int64_t*>(ptrs_[which_ + 2])[start_ + at];
      return value_type(start, stop, ptrs_[which_ + 3], ptrs_);
    }
  };
}
```

```
        cpp_code_declare_slots = (
            cpp_code_declare_slots
            + f"""
    ULong64_t fPtrs_{key} = 0;
    std::vector<{entry_type}>  slots_{key};
    std::vector<{entry_type}*> addrs_{key};
"""
        )

        cpp_code_define_readers = (
            cpp_code_define_readers
            + f"""
    if (name == "{key}") {{
        for (auto i : ROOT::TSeqU(fNSlots)) {{
            addrs_{key}[i] = &slots_{key}[i];
            reader.emplace_back((void *)(&addrs_{key}[i]));
        }}
    }}
"""
        )
```

Examples: Plot

Users define their analysis as a sequence of operations to be performed on the dataframe object - as usually




```
array = ak._v2.Array([
    [{"x": 1, "y": [1.1]}, {"x": 2, "y": [2.2, 0.2]}],
    [],
    [{"x": 3, "y": [3.0, 0.3, 3.3]}]])

ak_array_1 = array["x"]
ak_array_2 = array["y"]

data_frame = ak._v2.to_rdataframe(
    {"x": ak_array_1, "y": ak_array_2}
)
h = data_frame.Histo1D("x")
h.Draw()
```

Examples: Transform and Filter

User-defined transformation is passed to a compiler

Filter is applied

```
example = ak._v2.Array([1.1, 2.2, 3.3, 4.4, 5.5])
data_frame = ak._v2.to_rdataframe({"one": example})

ROOT.gInterpreter.Declare("""
template<typename T>
ROOT::RDF::RNode MyTransformation(ROOT::RDF::RNode df) {
            auto myFunc = [](T x){ return -x;};
            return df.Define("neg_one", myFunc, {"one"});
}
"""
)

data_frame_transformed = ROOT.MyTransformation[data_frame.GetColumnType("one")](
    ROOT.RDF.AsRNode(data_frame)
)
assert data_frame_transformed.Count().GetValue() == 5

data_frame2 = data_frame.Filter("one > 2.5")
data_frame2_transformed = ROOT.MyTransformation[data_frame.GetColumnType("one")](
    ROOT.RDF.AsRNode(data_frame2)
)
assert data_frame2_transformed.Count().GetValue() == 3
```

iris hep

PRINCETON UNIVERSITY

From RDataFrame to Awkward Arrays: in progress

- The *ak.from_rdataframe* function converts a selected column to native Awkward Arrays
- The conversion of the data taken out of the RDF is limited to:
  - Primitive types
  - Lists of primitive types and nested lists of primitive types
  - Awkward types
    - because Awkward Arrays are immutable no copy required
- User has an option to take the column out as a record with the column name as a tag
- By design it pulls one column at a time: it simplifies the interface and Awkward Arrays can be inexpensively joined with *ak.zip*

Example from a PyROOT tutorial: Awkward Array out

```
data_frame = ROOT.RDataFrame(1024)
coordDefineCode = """ROOT::VecOps::RVec<double> {0}(len);
                  std::transform({0}.begin(), {0}.end(), {0}.begin(), [](double){{return gRandom->Uniform(-1.0, 1.0);}});
                  return {0};"""

data_frame_x_y = (
    data_frame.Define("len", "gRandom->Uniform(0, 16)")
    .Define("x", coordDefineCode.format("x"))
    .Define("y", coordDefineCode.format("y"))
)

# Now we have in hands d, a RDataFrame with two columns, x and y, which
# hold collections of coordinates. The size of these collections vary.
# Let's now define radii out of x and y. We'll do it treating the collections
# stored in the columns without looping on the individual elements.
data_frame_x_y_r = data_frame_x_y.Define("r", "sqrt(x*x + y*y)")

array = ak._v2.from_rdataframe(data_frame_x_y_r, column="r", column_as_record=True)
assert array.layout.form == ak._v2.forms.RecordForm(
    [ak._v2.forms.ListOffsetForm("i64", ak._v2.forms.NumpyForm("float64"))], ["r"]
)
```

Awkward Array layout is described by its Form

Array: [{r: [0.813, 0.973, 0.854, 0.602]}, {...}, ..., {r: [1.27, 0.47, ..., 0.669]}]

Python type: <class 'awkward._v2.highlevel.Array'>

Array type: 1024 * {r: var * float64}

Layout form:
```
{
  "class": "RecordArray",
  "contents": {
    "r": {
      "class": "ListOffsetArray",
      "offsets": "i64",
      "content": "float64"
    }
  }
}
```

Example: Awkward Array in and Awkward Array out

```
ak_array_in = ak._v2.Array([[[1.1]], [[2.2, 3.3], [4.4]], [[5.5, 6.6], []]])

data_frame = ak._v2.to_rdataframe({"x": ak_array_in})

ak_array_out = ak._v2.from_rdataframe(
    data_frame, column="x", column_as_record=False
)

assert ak_array_in.to_list() == ak_array_out.to_list()
```

data_frame.GetColumnType("x") → awkward::ListArray_s5zeZjQHueU

Summary

Awkward Arrays and RDataFrame provide two very different ways of performing calculations at scale

By adding the ability to convert between them, users get the best of both

The Awkward-RDF bridge provides users with more flexibility in mixing different packages and languages in their analysis

It is part of Awkward version 2, which is currently a submodule '*ak._v2*', similar in spirit to ROOT's Experimental namespace.

- Versions 1 and 2 can be used side-by-side in a Python session, and version 1 will be dropped (i.e. version 2 is fully released) at the beginning of December 2022. See the timeline