Alexander and the Ragged, Jagged, Nested, Indirected, Very Awkward Arrays

Jim Pivarski, David Lange, Peter Elmer

Princeton University – IRIS-HEP

November 7, 2019

Next week, I said, I’m going to Australia.
In a data analysis, physicists need…

rich data structures

You can get that with C++ or Python, but C++ is verbose for analysis and Python is slow.
In a data analysis, physicists need...  

**rich data structures**

You can get that with C++ or Python, but C++ is verbose for analysis and Python is slow.

**interactivity**

Data-oriented DSLs like NumPy and Pandas are convenient, but only for rectangular arrays of numbers.
In a data analysis, physicists need...

**rich data structures**
You can get that with C++ or Python, but C++ is verbose for analysis and Python is slow.

**interactivity**
Data-oriented DSLs like NumPy and Pandas are convenient, but only for rectangular arrays of numbers.

**speed**
Results should pop up quickly enough that you don’t lose your train of thought.
Any data structure can be arranged as columnar arrays.

- ROOT I/O with object splitting (1997)
- Parquet file format (2015).
- Apache Arrow in-memory format (2016).
Any data structure can be arranged as columnar arrays

Clickable links to prior art:

- ROOT I/O with object splitting (1997).
- Parquet file format (2015).
- Apache Arrow in-memory format (2016).
Building columnar data structures from single-purpose parts

An array of variable-length arrays ("ragged" or "jagged") can be built out of flattened contents and offsets specifying the beginning of each subarray.

```
[[1.1, 2.2, 3.3],
 [4.4, 5.5],
 [6.6],
 [7.7, 8.8, 9.9]]
```

offsets    0,     3,   3,        5,   6,            9
content  1.1, 2.2, 3.3,    4.4, 5.5, 6.6, 7.7, 8.8, 9.9
Multiple levels of variable-length arrays ("doubly jagged") can be built by using one jagged array as the content of the other.
Building columnar data structures from single-purpose parts

Complex data structures can be built if you have enough columnar components, such as records with named fields, mixed data types, pointers, lazy loading...

```
[[Muon(31.1, -0.481, 0.882), Muon(9.76, -0.124, 0.924),
  Muon(8.18, -0.119, 0.923)],
 [Muon(5.27, 1.246, -0.991)],
 [Muon(4.72, -0.207, 0.953)],
 [Muon(8.59, -1.754, -0.264),
  Muon(8.714, 0.185, 0.629)]
```

```
 offsets  0,  3,  4,  5,  7
 pt      31.1, 9.76, 8.18, 5.27, 4.72, 8.59, 8.714
 phi     -0.481, -0.123, -0.119, 1.246, -0.207, -1.754, 0.185
 eta     0.882, 0.924, 0.923, -0.991, 0.953, -0.264, 0.629
```
We now have 1 year of user feedback and maintenance experience

pip-installs on MacOS and Windows (not batch jobs)

- numpy
- scipy
- pandas
- matplotlib
- root-numpy
- iminuit
- rootpy
- awkward
- coffea

awkward has 100 downloads/day, 2 issues/week
What I’ve learned

From feedback and tutorials:

The interface needs to be simpler: a single `awkward.Array` class to hide the `ListArray → RecordArray → NumpyArrays` structure.

Separate structural operations (e.g. cross-join) from physics (e.g. cross-product).
From feedback and tutorials:

The interface needs to be simpler: a single `awkward.Array` class to hide the `ListArray → RecordArray → NumpyArrays` structure.

Separate structural operations (e.g. cross-join) from physics (e.g. cross-product).

From maintenance:

Current implementation is entirely written in Python/Numpy.

For better flexibility, robustness, and uniformity, and in a few cases, speed, it should be rewritten in C++.
Layer 1: Python user interface: a single awkward.Array class.
Layer 1: Python user interface: a single awkward.Array class.

Layer 2: Structural classes in Python (e.g. ListArray/RecordArray).
New architecture

Layer 1: Python user interface: a single `awkward.Array` class.

Layer 2: Structural classes in Python (e.g. `ListArray/RecordArray`).

Layer 3: Array allocation and ownership; reference counting. Two languages: C++ and Numba (compiled Python).
Layer 1: Python user interface: a single awkward.Array class.

Layer 2: Structural classes in Python (e.g. ListArray/RecordArray).

Layer 3: Array allocation and ownership; reference counting. Two languages: C++ and Numba (compiled Python).

Layer 4: Implementations, where we write for loops. The only layer that needs to be optimized for speed.
First benefit: greater generality

The current implementation has some hidden limitations, such as slicing in more than two jagged dimensions.

```python
>>> import awkward       # awkward 0.x
>>> array = awkward.fromiter(
[0.0, 1.1, 2.2], [3.3, 4.4],
...
[5.5], [6.6, 7.7, 8.8, 9.9])
>>> array[::-1, ::2, 1:].tolist()
NotImplementedError: this implementation cannot slice a JaggedArray in more than two dimensions
```

But the new implementation is based on `for` loops and recursive procedures that can go arbitrarily deep.

```python
>>> import awkward1      # awkward 1.0
>>> array = awkward1.fromiter(
[0.0, 1.1, 2.2], [3.3, 4.4],
...
[5.5], [6.6, 7.7, 8.8, 9.9])
>>> awkward1.tolist(array[::-1, ::2, 1:])
[[[7.7, 8.8, 9.9]], [], [], [[1.1, 2.2], [4.4]]]
```
Full access in C++

Everything is available in C++ except the convenient syntax.

```cpp
namespace ak = awkward;

int main(int, char**) {
    std::vector<std::vector<std::vector<double>>> vector =
        {{{0.0, 1.1, 2.2}, {}, {3.3, 4.4}, {5.5}}, {},
         {{6.6, 7.7, 8.8, 9.9}}};

    ak::FillableArray builder(ak::FillableOptions(1024, 2.0));
    for (auto x : vector)
        builder.fill(x);
    std::shared_ptr<ak::Content> array = builder.snapshot();

    ak::Slice slice;
    slice.append(ak::SliceRange(ak::Slice::none(), ak::Slice::none(), -1));  // ::-1
    slice.append(ak::SliceRange(ak::Slice::none(), ak::Slice::none(), 2));   // ::2
    slice.append(ak::SliceRange(1, ak::Slice::none(), ak::Slice::none()));   // 1:

    if (array.get()->getitem(slice).get()->tojson(false, 1) !=
        "[[[7.7,8.8,9.9]],[],[[]],[[1.1,2.2],[4.4]]])")
        return -1;
    return 0;
}
```
Similarly, everything is available in functions compiled by Numba.

```python
>>> import numba

>>> @numba.jit(nopython=True)
... def compileme(a):
...     return a[::-1, ::2, 1:]
...
>>> compileme
CPUDispatcher(<function compileme at 0x7fd135e52ae8>)

>>> array = awkward1.fromiter([[[0.0, 1.1, 2.2], [], [3.3, 4.4]],
... [5.5], [], [[6.6, 7.7, 8.8, 9.9]]])

>>> awkward1.tolist(compileme(array))

[[[7.7, 8.8, 9.9]], [], [[]], [[1.1, 2.2], [4.4]]]
```
Similarly, everything is available in functions compiled by Numba.

```python
global import numba
global 
>>> @numba.jit(nopython=True)
... def compileme(a):
...     return a[::-1, ::2, 1:]
... >>> compileme
CPUDispatcher(<function compileme at 0x7fd135e52ae8>)

array = awkward1.fromiter([[0.0, 1.1, 2.2], [], [3.3, 4.4]], [5.5]], [], [6.6, 7.7, 8.8, 9.9]))

awkward1.tolist(compileme(array))

[[7.7, 8.8, 9.9]], [], [[]], [[1.1, 2.2], [4.4]]
```

Numba compiles a subset of Python; all data types must be known to the compiler.
Similarly, everything is available in functions compiled by Numba.

```python
>>> import numba
...
>>> @numba.jit(nopython=True)
... def compileme(a):
...     return a[::-1, ::2, 1:]
...
>>> compileme
CPUDispatcher(<function compileme at 0x7fd135e52ae8>)
>>> array = awkward1.fromiter([[0.0, 1.1, 2.2], [], [3.3, 4.4]],
...                             [[5.5]], [], [[6.6, 7.7, 8.8, 9.9]])
>>> awkward1.tolist(compileme(array))
[[[7.7, 8.8, 9.9]], [], [[]], [[1.1, 2.2], [4.4]]]
```

Numba compiles a subset of Python; all data types must be known to the compiler.

Data analysts can start with pure Python and only accelerate the functions that need it.
All interfaces—Python, C++, Numba—share the same implementation, a pure C library of functions on arrays.

```c
extern "C" {
    Error awkward_listarray32_getitem_next_at_64(int64_t* tocarry,
        const int32_t* fromstarts, const int32_t* fromstops,
        int64_t lenstarts, int64_t startsoffset, int64_t stopsoffset,
        int64_t at) {
        // This is only one layer with loops over array elements.
        for (int64_t i = 0; i < lenstarts; i++) {
            int64_t length = fromstops[stopsoffset + i] -
                fromstarts[startsoffset + i];

            int64_t regular_at = at;
            if (regular_at < 0) {
                regular_at += length;
            }
            if (!(0 <= regular_at && regular_at < length)) {
                return failure("index out of range", i, at);
            }
            tocarry[i] = fromstarts[startsoffset + i] + regular_at;
        }
        return success();
    }
}
```
# "For events with at least three leptons (electrons or muons) and a same-flavor opposite-sign lepton pair, find the same-flavor opposite-sign lepton pair with a mass closest to 91.2 GeV; make a histogram of the pT of the leading other lepton."

leptons = electrons union muons

cut count(leptons) >= 3 named "three_leptons" {
    Z = electrons as (lep1, lep2) union muons as (lep1, lep2)
    where lep1.charge != lep2.charge
    min by abs(mass(lep1, lep2) - 91.2)

    third = leptons except [Z.lep1, Z.lep2] max by pt
    hist third.pt by regular(100, 0, 250) named "third_pt"
}
Investigated toy languages with sets as a first-class concept

https://github.com/jpivarski/PartiQL

# "For events with at least three leptons (electrons or muons) and a same-flavor opposite-sign lepton pair, find the same-flavor opposite-sign lepton pair with a mass closest to 91.2 GeV; make a histogram of the pT of the leading other lepton."

leptons = electrons union muons

cut count(leptons) >= 3 named "three_leptons" {
  Z = electrons as (lep1, lep2) union muons as (lep1, lep2)
  where lep1.charge != lep2.charge
  min by abs(mass(lep1, lep2) - 91.2)

  third = leptons except [Z.lep1, Z.lep2] max by pt
  hist third.pt by regular(100, 0, 250) named "third_pt"
}

To define operations on sets, such as join, cross, union, and except, we need what SQL calls a surrogate-key index, so this has been added as an optional Identity array, passed through all operations in Awkward 1.0.
Converting record-oriented data into columnar data is much faster.

Uncompressed data in warm cache; ~5 GB samples in binary format; AWS r5ad.xlarge instance; measuring the wall time needed to copy from source into jagged$^N$ arrays.
With a dataset in Awkward form (from TTrees, RNtuples, Arrow...), we want to

- perform vectorized operations with a Numpy-like syntax,
- pass data to and from a C++ library,
- enter a compiled Numba function to write fast for loops,
- evaluate a declarative expression on sets of particles,

interchangeably.
Multi-paradigm arrays

With a dataset in Awkward form (from TTrees, RNtuples, Arrow...), we want to

- perform vectorized operations with a Numpy-like syntax,
- pass data to and from a C++ library,
- enter a compiled Numba function to write fast for loops,
- evaluate a declarative expression on sets of particles,

interchangeably.

Awkward 1.0 is intended as a solid foundation for that future.
When can I try it?

Nowish: it is in a testable state (for Coffea and thrill-seekers).

Will be minimally usable for physics analysis in “early 2020.”

Start an `import awkward → import awkward0`
`import awkward1 → import awkward` transition by spring.