Jagged, ragged, awkward arrays!

Jim Pivarski

Princeton University – IRIS-HEP

September 14, 2019

Institute for Research and Innovation in Software for High Energy Physics (IRIS-HEP)

Computational and data science research to enable discoveries in fundamental physics

IRIS-HEP is a software institute funded by the National Science Foundation. It aims to develop the state-of-the-art software cyberinfrastructure required for the challenges of data intensive scientific research at the High Luminosity Large Hadron Collider (HL-LHC) at CERN, and other planned HEP experiments of the 2020's. These facilities are discovery machines which aim to understand the fundamental building

Upcoming Events:

Sep 10–11, 2019 — *Fermilab*

Blueprint: Accelerated Machine Learning and Inference

Sep 12–13, 2019 — *Fermilab*

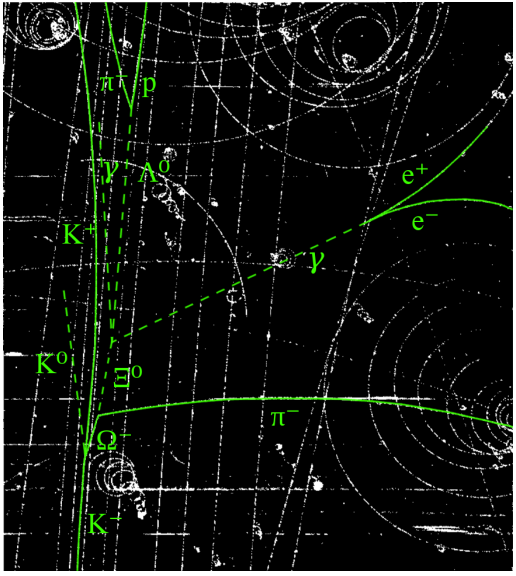
Motivation: particle physics analysis

This is a photo of subatomic particles leaving tracks in a detector.

In 1964, a group at Berkeley scanned 100,000 photos to find this one: the discovery of the Ω baryon.

Do you see it?

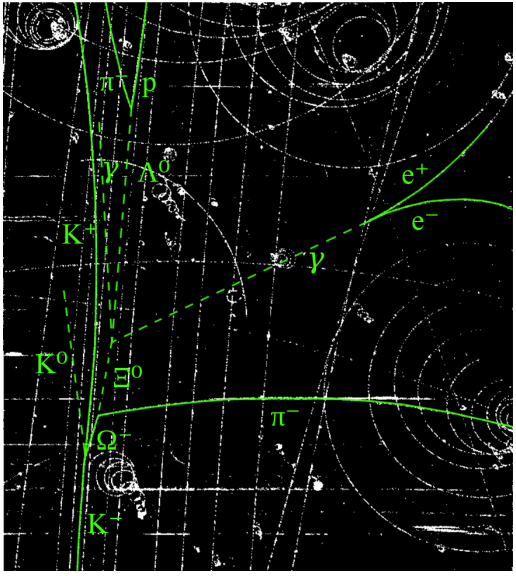
Motivation: particle physics analysis



How about now?

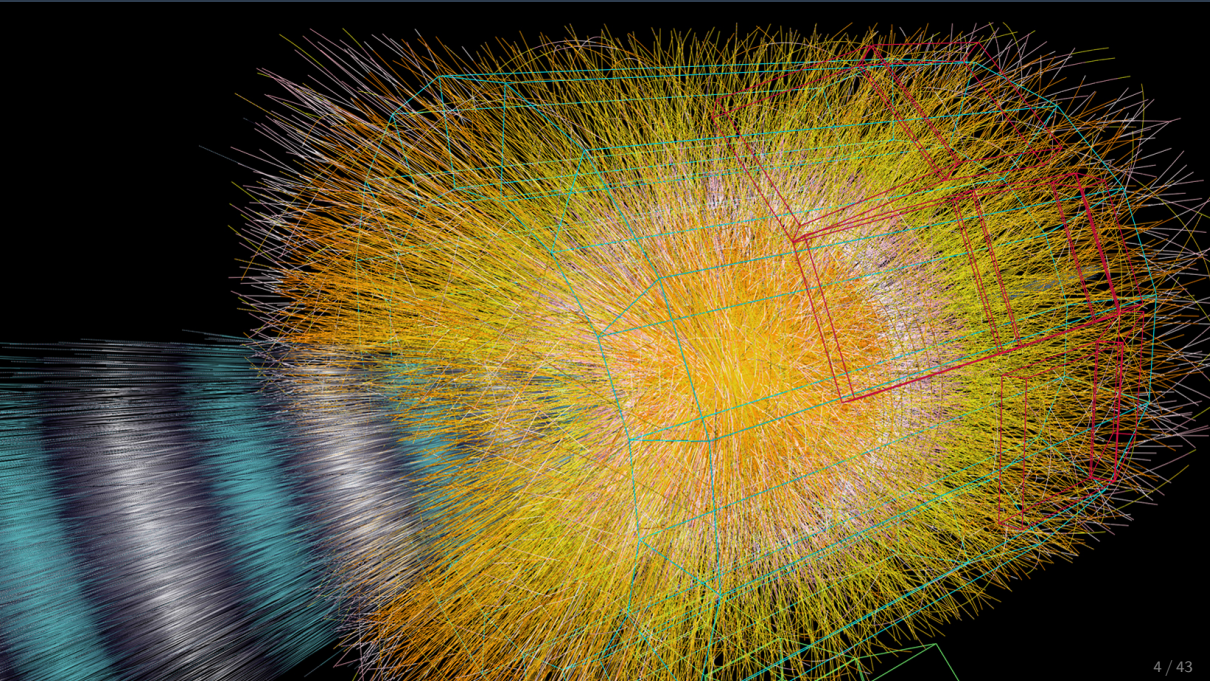
$$K^- + p \to \Omega^- + K^+ + K^0$$
$$\Omega^- \to \Xi^0 + \pi^-$$
$$\Xi^0 \to \Lambda^0 + \pi^0$$
$$\Lambda^0 \to \pi^- + p$$
$$\pi^0 \to \gamma_1 + \gamma_2$$
$$\gamma_1 \to e^+ + e^-$$
$$\gamma_2 \to e^+ + e^-$$

Motivation: particle physics analysis

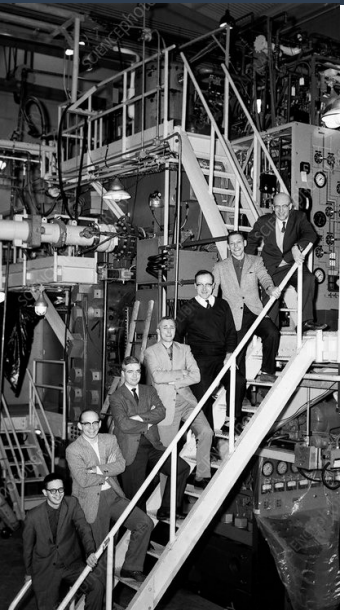


This is what particle physicists do: collide particles, produce new ones and take pictures of the decays.

However, these pictures come to us unlabeled: lots of interactions overlap the "interesting" ones.



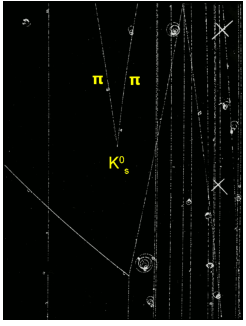
Modern particle physics scales this up

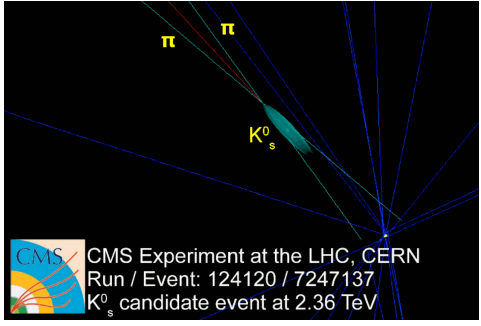


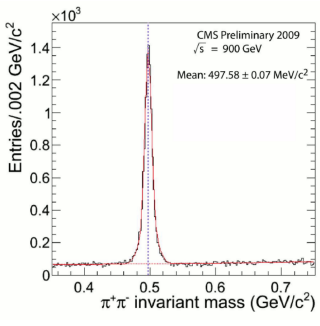
photographs	digitized signals
100,000 events	∼trillion events
manual/semi-automated scans	algorithmic searches and machine learning
30 authors	3,000 authors

Automatically identifying a particle decay

1. Loop over all pairs of particle tracks, tentatively labeling them π^+ and π^-.
2. Calculate $\mathrm{m} = \sqrt{(E_{\pi^+} + E_{\pi^-})^2 - |\vec{p}_{\pi^+} + \vec{p}_{\pi^-}|^2}$ for each pair.
3. The ones with $m \sim \mathrm{mass}(K_s^0) = 0.5\ \mathrm{GeV}/c^2$ are good candidates.

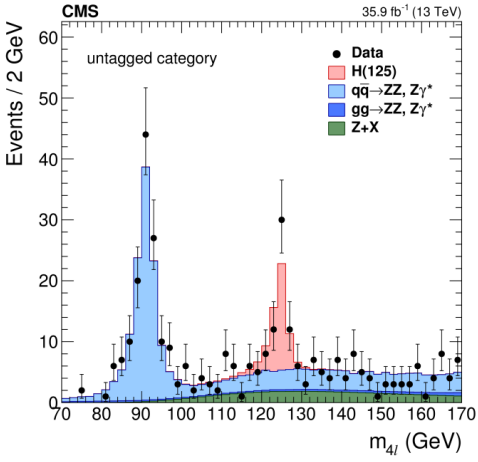


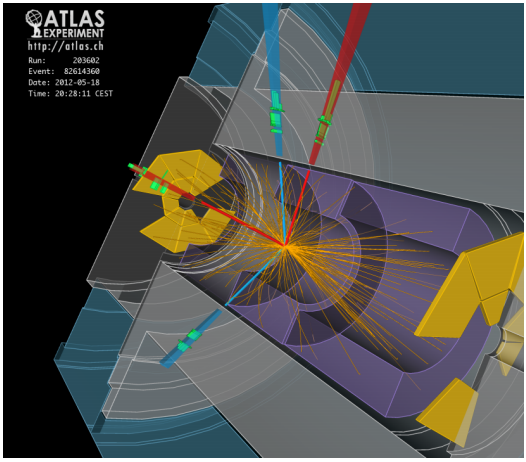




Apply this technique successively down the decay chain

$$H \to ZZ \qquad Z \to e^+e^- \qquad Z \to \mu^+\mu^-$$





From a computational point of view...

- All events are independent; no particles or decays cross from one event to the next (extremely few exceptions).

From a computational point of view...

- All events are independent; no particles or decays cross from one event to the next (extremely few exceptions).
- Detectors produce collections of different kinds of signals: tracks, energy deposition, timing, etc., each in its own variable-length collection.

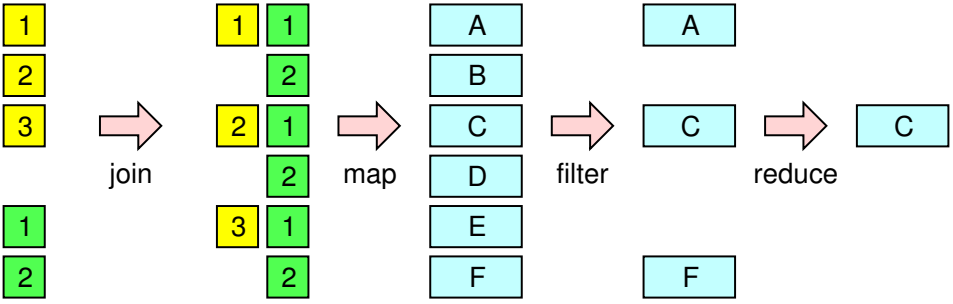
From a computational point of view...

- All events are independent; no particles or decays cross from one event to the next (extremely few exceptions).
- Detectors produce collections of different kinds of signals: tracks, energy deposition, timing, etc., each in its own variable-length collection.
- Decay candidates may be thought of as a **SELF JOIN** (single collection) or a **CROSS JOIN** (different collections) but always **ON** `t1.eventId == t2.eventId`.

From a computational point of view...

- All events are independent; no particles or decays cross from one event to the next (extremely few exceptions).
- Detectors produce collections of different kinds of signals: tracks, energy deposition, timing, etc., each in its own variable-length collection.
- Decay candidates may be thought of as a **`SELF JOIN`** (single collection) or a **`CROSS JOIN`** (different collections) but always **`ON`** `t1.eventId == t2.eventId`.
- Good candidates are identified by *filtering* (throw away the bad) and/or *reduction* (pick the best of what remains.)

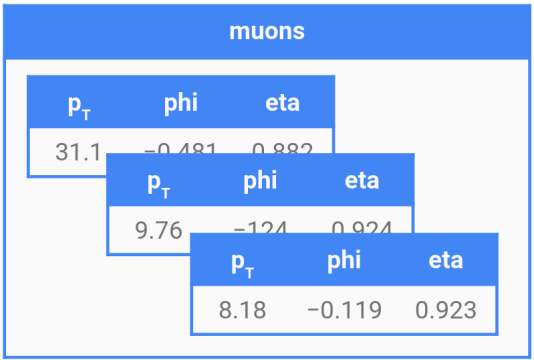
From a computational point of view. . .



. . . independently for each event.

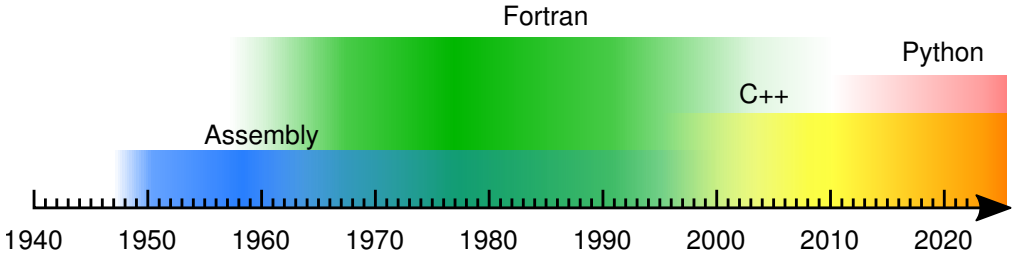
From a computational point of view. . .

- Large number of events, processed in parallel.
- Each event has an *arbitrary number* of record-like structures.
- Not reducible to a flat table without padding and/or loss.



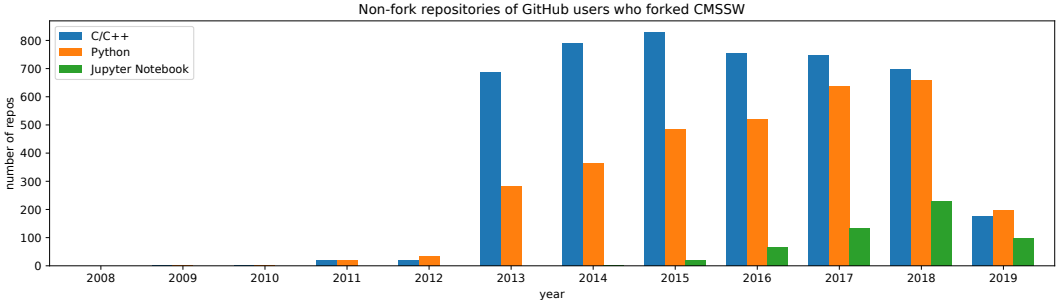
mu1 p_T	mu1 phi	mu1 eta	mu2 p_T	mu2 phi	mu2 eta
31.1	−0.481	0.882	9.76	−0.124	0.924
5.27	1.246	−0.991	n/a	n/a	n/a
4.72	−0.207	0.953	n/a	n/a	n/a
8.59	−1.754	−0.264	8.714	0.185	0.629

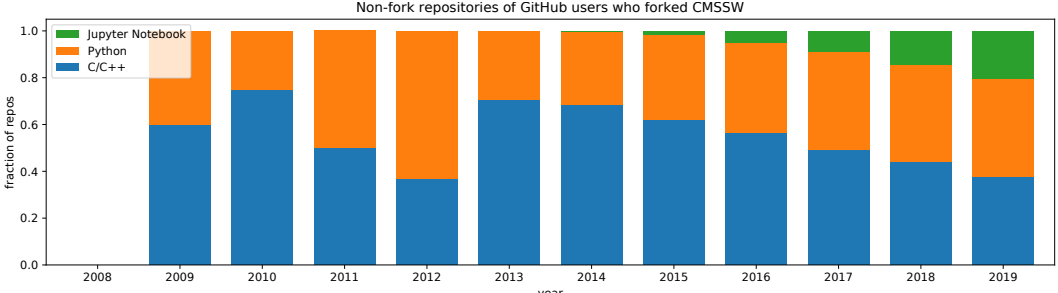
Programming languages used in particle physics



- First programmable computers were used for particle physics simulations.
- Fortran was used to analyze events soon after the language was invented; physicists wrote packages (BOS, HYDRA, ZBOOK) to add data structures.
- Major transition from Fortran to C++ in the late 1990's/early 2000's.
- Very recent adoption of Python (alongside C++).

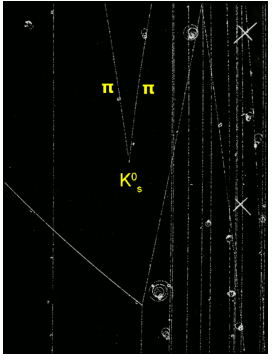
The Python transition is happening *right now*





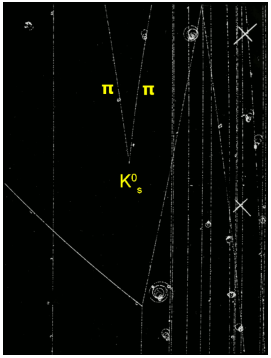
In our field, “conventional” analysis means C++

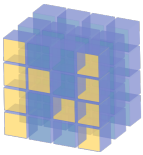
```
std::vector<Kaon*> getKaons(Event event) {
  std::vector<Track*> tracks = event.getTracks();
  std::vector<Kaon*> kaons;
  for (auto t1 = tracks.begin(); t1 != tracks.end(); ++t1) {
    for (auto t2 = t1 + 1; t2 != tracks.end(); ++t2) {
      if (t1->charge != t2->charge) {
        double m = mass(t1, t2);
        if (fabs(m - 0.5) < 0.01) {
          kaons.push_back(new Kaon(t1, t2));
        }
      }
    }
  }
  return kaons;
}
```



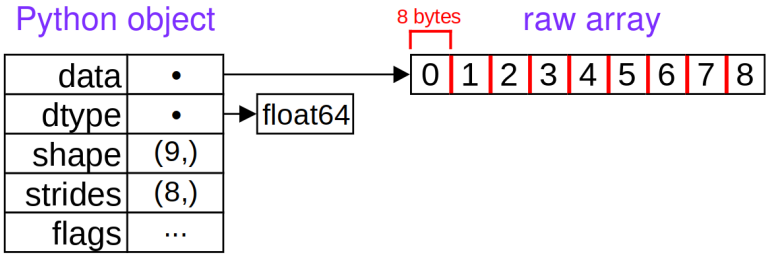
But direct translation to Python would be a performance disaster

```
def getKaons(event):
  tracks = event.getTracks()
  kaons = []
  for i, t1 in enumerate(tracks):
    for t2 in tracks[i + 1:]:
      if t1.charge != t2.charge:
        m = mass(t1, t2)
        if abs(m - 0.5) < 0.01:
          kaons.append(Kaon(t1, t2))

  return kaons
```





NumPy


Python object
8 bytes
raw array
data
dtype
shape
strides
flags
(9,)
(8,)
···
float64
0
1
2
3
4
5
6
7
8

High-performance processing in Python? Numpy!

Numpy has a suite of operations that each apply to whole arrays (compiled, maybe vectorized loop):

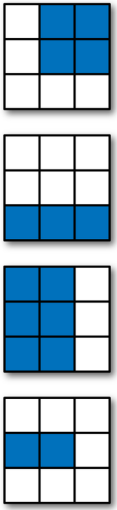
```
def calculate(pt, eta):
    pz = pt * numpy.sinh(eta)

calculate(pt_scalar, eta_scalar)
calculate(pt_array,  eta_array)
```

High-performance processing in Python? Numpy!

Numpy has a suite of operations that each apply to whole arrays (compiled, maybe vectorized loop):

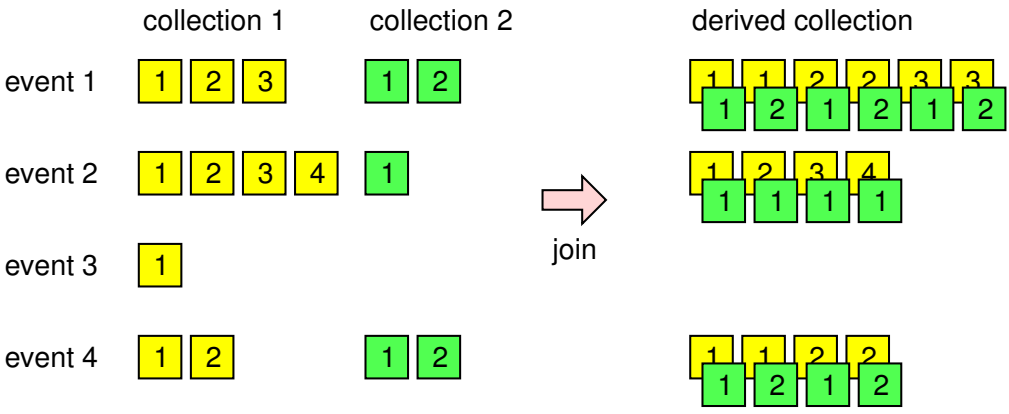
```
def calculate(pt, eta):
    pz = pt * numpy.sinh(eta)

calculate(pt_scalar, eta_scalar)
calculate(pt_array,  eta_array)
```

By manipulating the `shape` and `strides`, slices are $\mathcal{O}(1)$ and zero-copy.

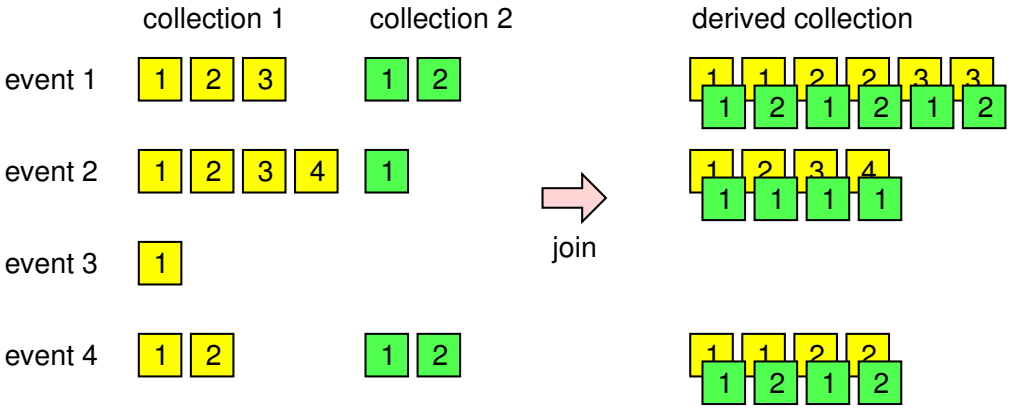


Expression	Shape
`arr[:2, 1:]`	(2, 2)
`arr[2]`	(3,)
`arr[2, :]`	(3,)
`arr[2:, :]`	(1, 3)
`arr[:, :2]`	(3, 2)
`arr[1, :2]`	(2,)
`arr[1:2, :2]`	(1, 2)

But Numpy doesn't have anything for unequal-length lists



But Numpy doesn't have anything for unequal-length lists

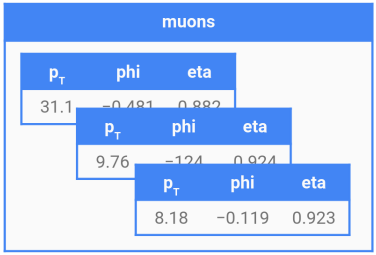


Some libraries* can represent arrays of unequal-length arrays, known as "jagged" or "ragged" arrays.

*Apache Arrow, XND, TensorFlow, Zarr (genetics), and ROOT (particle physics)

Jagged/nested data can be contiguous by field ("columnar")

```
[[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)]
]
```

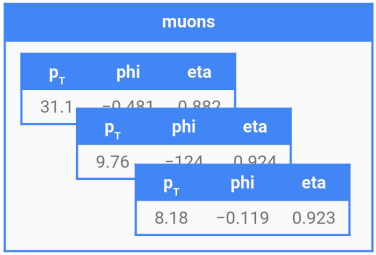


The trick: represent structure with counts or offsets or starts/stops or parents.

counts	3,			1,	1,	2	
p_T	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

Jagged/nested data can be contiguous by field ("columnar")

```
[[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)]
]
```



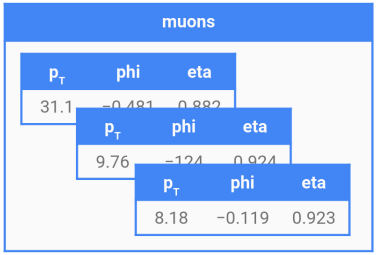
The trick: represent structure with counts or offsets or starts/stops or parents.

offsets `0, 3, 4, 5, 7`

p_T	`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`

Jagged/nested data can be contiguous by field ("columnar")

```
[[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)]
]
```

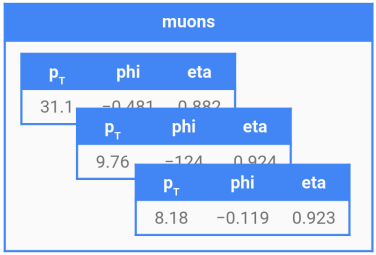


The trick: represent structure with counts or offsets or starts/stops or parents.

starts	0,			3,	4,	5	
stops	3,			4,	5,	7	
p_T	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

Jagged/nested data can be contiguous by field ("columnar")

```
[[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)]
]
```



The trick: represent structure with counts or offsets or starts/stops or parents.

parents	0,	0,	0,	1,	2,	3,	3
p_T	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

But we can go beyond *representing* columnar jagged arrays and *manipulate* them in this form

Example of a jagged operation: manipulating jagged arrays

"Remove the first muon from each event."

```
[[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)],
 ...
```

"Remove the first muon from each event."

starts	0,			3,	4,	5	
stops	3,			4,	5,	7	
p_T	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

Example of a jagged operation: manipulating jagged arrays

"Remove the first muon from each event." ⟶ **rewrite all inner lists.**

```
[[                                  Muon(9.76, -0.124, 0.924), Muon(8.18, -0.119, 0.923)],
 [                                ],
 [                                ],
 [                                   Muon(8.714, 0.185, 0.629)],
 ...
```

"Remove the first muon from each event." ⟶ **increase all starts by 1.**

starts	1,			4,	5,	6	
stops	3,			4,	5,	7	
p_T	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

Example of a jagged operation: manipulating jagged arrays

"Remove the first muon from each event." ⟶ **rewrite all inner lists.**

```
[[                                Muon(9.76, -0.124, 0.924), Muon(8.18, -0.119, 0.923)],
 [                              ],
 [                              ],
 [                                 Muon(8.714, 0.185, 0.629)],
 ...
```

"Remove the first muon from each event." ⟶ **increase all starts by 1.**

starts	1,			4,	5,	6	
stops	3,			4,	5,	7	
p_T	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 didn't need to touch any contents (i.e. read from disk/decompress them).

Can be very useful if only ∼10% of the fields are of interest!

Properties of columnar data

- Columnar data structures are *fully composable:* a jagged array's content can be another jagged array, some fields of a record structure can be jagged, others not.

Properties of columnar data

- Columnar data structures are *fully composable:* a jagged array's content can be another jagged array, some fields of a record structure can be jagged, others not.
- Other non-trivial data types may be encoded this way: heterogeneous lists, nullable types, pointers...

Properties of columnar data

- Columnar data structures are *fully composable:* a jagged array's content can be another jagged array, some fields of a record structure can be jagged, others not.
- Other non-trivial data types may be encoded this way: heterogeneous lists, nullable types, pointers...
- Most operations (beyond just slicing) can modify the structure (`counts`/`offsets`/`starts`,`stops`/`parents`) without touching the nested content.

Awkward Array: columnar data with a Numpy-like interface

```
>>> import awkward
>>> array = awkward.fromiter(
...     [[1.1, 2.2, None, 3.3, None],
...      [4.4, [5.5]],
...      [{"x": 6, "y": {"z": 7}}, None, {"x": 8, "y": {"z": 9}}]])
```

Awkward Array

Awkward Array: columnar data with a Numpy-like interface

Awkward Array

```
>>> import awkward
>>> array = awkward.fromiter(
...     [[1.1, 2.2, None, 3.3, None],
...      [4.4, [5.5]],
...      [{"x": 6, "y": {"z": 7}}, None, {"x": 8, "y": {"z": 9}}]])

>>> print(array)              # an array of columnar data
[[1.1 2.2 None 3.3 None] [4.4 [5.5]] [<Row 0> None <Row 1>]]
```

Awkward Array: columnar data with a Numpy-like interface

Awkward Array

```
>>> import awkward
>>> array = awkward.fromiter(
...     [[1.1, 2.2, None, 3.3, None],
...      [4.4, [5.5]],
...      [{"x": 6, "y": {"z": 7}}, None, {"x": 8, "y": {"z": 9}}]])

>>> print(array)              # an array of columnar data
[[1.1 2.2 None 3.3 None] [4.4 [5.5]] [<Row 0> None <Row 1>]]

>>> print(array[:, -2:])      # get whole outer list, last two of inner
[[3.3 None] [4.4 [5.5]] [None <Row 1>]]
```

Awkward Array: columnar data with a Numpy-like interface

Awkward Array

```
>>> import awkward
>>> array = awkward.fromiter(
...     [[1.1, 2.2, None, 3.3, None],
...      [4.4, [5.5]],
...      [{"x": 6, "y": {"z": 7}}, None, {"x": 8, "y": {"z": 9}}]])

>>> print(array)              # an array of columnar data
[[1.1 2.2 None 3.3 None] [4.4 [5.5]] [<Row 0> None <Row 1>]]

>>> print(array[:, -2:])      # get whole outer list, last two of inner
[[3.3 None] [4.4 [5.5]] [None <Row 1>]]

>>> (array + 100).tolist()    # element-wise function applied to arrays
                              # i.e. a Numpy ufunc (all ufuncs work)
[[101.1, 102.2, None, 103.3, None],
 [104.4, [105.5]],
 [{'x': 106, 'y': {'z': 107}}, None, {'x': 108, 'y': {'z': 109}}]]
```

Internally, it's a bunch or Numpy arrays *interpreted* as nested data

```
>>> array.layout
 layout
[            ()] JaggedArray(starts=layout[0], stops=layout[1], content=layout[2])
[             0]   ndarray(shape=3, dtype=dtype('int64'))
[             1]   ndarray(shape=3, dtype=dtype('int64'))
[             2]   IndexedMaskedArray(mask=layout[2, 0], content=layout[2, 1], maskedwhen=-1)
[          2, 0]     ndarray(shape=10, dtype=dtype('int64'))
[          2, 1]     UnionArray(tags=layout[2, 1, 0], index=layout[2, 1, 1],
                                contents=[layout[2, 1, 2], layout[2, 1, 3], layout[2, 1, 4]])
[       2, 1, 0]       ndarray(shape=7, dtype=dtype('uint8'))
[       2, 1, 1]       ndarray(shape=7, dtype=dtype('int64'))
[       2, 1, 2]       ndarray(shape=4, dtype=dtype('float64'))
[       2, 1, 3]       JaggedArray(starts=layout[2, 1, 3, 0], stops=layout[2, 1, 3, 1],
                                   content=layout[2, 1, 3, 2])
[    2, 1, 3, 0]         ndarray(shape=1, dtype=dtype('int64'))
[    2, 1, 3, 1]         ndarray(shape=1, dtype=dtype('int64'))
[    2, 1, 3, 2]         ndarray(shape=1, dtype=dtype('float64'))
[       2, 1, 4]       Table(x=layout[2, 1, 4, 0], y=layout[2, 1, 4, 1])
[    2, 1, 4, 0]         ndarray(shape=2, dtype=dtype('int64'))
[    2, 1, 4, 1]         Table(z=layout[2, 1, 4, 1, 0])
[2, 1, 4, 1, 0]           ndarray(shape=2, dtype=dtype('int64'))
```

Array indexing is a function; arrays have functional type

```
>>> # array type formatted to look like a curried function's type
>>> print(array.type)
[0, 3) -> [0, inf) -> ?((float64                |
                        [0, inf) -> float64 |
                        'x' -> int64
                        'y' -> 'z' -> int64 ))
```

How to read the above:

- First square bracket in `array[x]` takes a non-negative integer less than 3.
- Second takes a non-negative integer (unknown upper limit because it's jagged).
- What that returns is nullable (might be `None`).
- And it could be a `float`, a jagged array of floats, or a record taking `'x'` or `'y'`.
- `'x'` returns an integer; `'y'` returns another record taking `'z'` to integers.

Aside: arrays are functions; integer indexing is composition

Consider any two $\mathbb{Z}^{\geq 0} \to \mathbb{Z}^{\geq 0}$ functions,

```
def f(x):
    return x**2 - 5*x + 10
def g(y):
    return max(0, 2*y - 10) + 3
```

sample them as arrays, including the composition $f \circ g$,

```
F   = numpy.array([f(i) for i in range(10)])      # F is f at 10 elements
G   = numpy.array([g(i) for i in range(100)])     # G is g at 100 elements (to cover max(F))
GoF = numpy.array([g(f(i)) for i in range(10)])   # GoF is gof at 10 elements
```

and indexing `G` by `F` is the same as the sampled composition `GoF`:

```
print("G\u2218F =", G[F])    # integer indexing (numpy.take)
print("g\u2218f =", GoF)     # array of the composed functions returns the same thing
```

```
GoF = [13  5  3  3  5 13 25 41 61 85]
gof = [13  5  3  3  5 13 25 41 61 85]
```

The associativity of function composition $(f(g(x)) == (f \circ g)(x))$ lets us delay operations on nested arrays. (I'd love to find any literature references on this, if anyone knows where to look...)

Getting back to the main point: let's look at real data

NASA exoplanets dataset: stars may have more than one planet

```
>>> import urllib.request
>>> stars = awkward.fromiter(
...     json.load(
...         urllib.request.urlopen(
...             "http://scikit-hep.org/uproot/examples/exoplanets.json")))

>>> stars
<Table [<Row 0> <Row 1> <Row 2> ... <Row 2933> <Row 2934>]>
```

NASA exoplanets dataset: stars may have more than one planet

```
>>> import urllib.request
>>> stars = awkward.fromiter(
...     json.load(
...         urllib.request.urlopen(
...             "http://scikit-hep.org/uproot/examples/exoplanets.json")))

>>> stars[8:10].tolist()
[{'name': '24 Sex', 'ra': 155.86821, 'dec': -0.902244, 'dist': 72.21,
  'mass': 1.54, 'radius': 4.9, 'planets': [
      {'name': 'b', 'orbit': 1.333, 'eccen': 0.09, 'period': 452.8,
       'mass': 1.99, 'radius': None},
      {'name': 'c', 'orbit': 2.08, 'eccen': 0.29, 'period': 883.0,
       'mass': 0.86, 'radius': None}]},
 {'name': '2MASS J01225093-2439505', 'ra': 20.712243, 'dec': -24.664049,
  'dist': 36.0, 'mass': 0.4, 'radius': None, 'planets': [
      {'name': 'b', 'orbit': 52.0, 'eccen': None, 'period': None,
       'mass': 24.5, 'radius': None}]}
]
```

NASA exoplanets dataset: stars may have more than one planet

```
>>> import urllib.request
>>> stars = awkward.fromiter(
...     json.load(
...         urllib.request.urlopen(
...             "http://scikit-hep.org/uproot/examples/exoplanets.json")))
>>> print(stars.type)
[0, 2935) -> 'name'    -> <class 'str'>
             'ra'      -> float64
             'dec'     -> float64
             'dist'    -> ?(float64)
             'mass'    -> ?(float64)
             'radius'  -> ?(float64)
             'planets' -> [0, inf) -> 'name'   -> <class 'str'>
                                      'orbit'  -> ?(float64)
                                      'eccen'  -> ?(float64)
                                      'period' -> ?(float64)
                                      'mass'   -> ?(float64)
                                      'radius' -> ?(float64)
```

NASA exoplanets dataset: stars may have more than one planet

```
>>> import urllib.request
>>> stars = awkward.fromiter(
...     json.load(
...         urllib.request.urlopen(
...             "http://scikit-hep.org/uproot/examples/exoplanets.json")))

>>> stars["mass"]
<MaskedArray [2.7 2.78 2.2 ... 2.3 1.3 2.2]>

>>> stars["planets"]["mass"]
<JaggedArray [[19.4] [14.74] [4.8] ... [20.6] [0.6876 1.981 4.132] [2.8]]>
```

NASA exoplanets dataset: stars may have more than one planet

```
>>> import urllib.request
>>> stars = awkward.fromiter(
...     json.load(
...         urllib.request.urlopen(
...             "http://scikit-hep.org/uproot/examples/exoplanets.json")))

>>> stars["mass"]
<MaskedArray [2.7 2.78 2.2 ... 2.3 1.3 2.2]>

>>> stars["planets"]["mass"]
<JaggedArray [[19.4] [14.74] [4.8] ... [20.6] [0.6876 1.981 4.132] [2.8]]>

>>> print(stars["mass"].type)              # just selecting a column
[0, 2935) -> ?(float64)
>>> print(stars["planets"]["mass"].type)   # projecting through a column
[0, 2935) -> [0, inf) -> ?(float64)
```

This "projection" selects a column that slices through rows, but in a "jagged" way.

NASA exoplanets dataset: stars may have more than one planet

```
>>> import urllib.request
>>> stars = awkward.fromiter(
...     json.load(
...         urllib.request.urlopen(
...             "http://scikit-hep.org/uproot/examples/exoplanets.json")))

>>> stars[8]["planets"][1]["mass"]          # projecting through columns:
0.86                                        # int and str indexes commute!
>>> stars[8]["planets"]["mass"][1]
0.86
>>> stars["planets"][8][1]["mass"]
0.86
>>> stars["planets"][8]["mass"][1]
0.86
>>> stars["planets"]["mass"][8][1]
0.86
>>> stars["planets"]["mass"][8, 1]
0.86
```

NASA exoplanets dataset: stars may have more than one planet

```
>>> import urllib.request
>>> stars = awkward.fromiter(
...     json.load(
...         urllib.request.urlopen(
...             "http://scikit-hep.org/uproot/examples/exoplanets.json")))
>>> awkward.topandas(stars, flatten=True)  # put jaggedness in the index
```

		name	ra	dec	dist	mass	radius	planets					
								name	orbit	eccen	period	mass	radius
0	0	11 Com	185.179276	17.792868	93.37	2.70	19.00	b	1.290000	0.2310	326.03000	19.4000	NaN
1	0	11 UMi	229.274536	71.823898	125.72	2.78	29.79	b	1.530000	0.0800	516.21997	14.7400	NaN
2	0	14 And	352.822571	39.236198	75.59	2.20	11.00	b	0.830000	0.0000	185.84000	4.8000	NaN
3	0	14 Her	242.601303	43.817646	17.94	0.90	0.93	b	2.930000	0.3700	1773.40002	4.6600	NaN
4	0	16 Cyg B	295.466553	50.517525	21.41	1.08	1.13	b	1.660000	0.6800	798.50000	1.7800	NaN
5	0	18 Del	314.608063	10.839286	76.38	2.30	8.50	b	2.600000	0.0800	993.30000	10.3000	NaN
6	0	1RXS J16	242.376268	-21.083036	145.00	0.85	0.00	b	330.000000	NaN	NaN	8.0000	NaN
7	0	24 Boo	217.157547	49.844852	96.25	0.99	10.64	b	0.190000	0.0420	30.35060	0.9100	NaN
8	0	24 Sex	155.868210	-0.902244	72.21	1.54	4.90	b	1.333000	0.0900	452.80000	1.9900	NaN
	1	24 Sex	155.868210	-0.902244	72.21	1.54	4.90	c	2.080000	0.2900	883.00000	0.8600	NaN
...	...	...	...	...	...	...	...	...	...	...	...	...	...
2932	0	tau Gem	107.784882	30.245163	112.64	2.30	26.80	b	1.170000	0.0310	305.50000	20.6000	NaN
2933	0	ups And	24.199345	41.405460	13.41	1.30	1.56	b	0.059222	0.0215	4.61703	0.6876	NaN
	1	ups And	24.199345	41.405460	13.41	1.30	1.56	c	0.827774	0.2596	241.25800	1.9810	NaN
	2	ups And	24.199345	41.405460	13.41	1.30	1.56	d	2.513290	0.2987	1276.46000	4.1320	NaN
2934	0	xi Aql	298.562012	8.461452	56.27	2.20	12.00	b	0.680000	0.0000	136.75000	2.8000	NaN

[3933 rows x 12 columns]

Relationship to Pandas

Jaggedness → MultiIndex rows

```
>>> awkward.topandas(
...   awkward.fromiter(
...     [[[1.1, 2.2], [], [3.3]],
...      [],
...      [[4.4], [5.5, 6.6]],
...      [[7.7]],
...      [[8.8]]]
...   ), flatten=True)
```

0	0	0	1.1
		1	2.2
	2	0	3.3
2	0	0	4.4
	1	0	5.5
		1	6.6
3	0	0	7.7
4	0	0	8.8

Nested records → MultiIndex columns

```
>>> awkward.topandas(
...   awkward.fromiter([
...     {"I":
...        {"a": _, "b": {"i": _}},
...      "II":
...        {"x": {"y": {"z": _}}}}
...     for _ in range(0, 50, 10)]
...   ), flatten=True)
```

	I		II
	a	b	x
		i	y
			z
0	0	0	0
1	10	10	10
2	20	20	20
3	30	30	30
4	40	40	40

Limitation of Pandas's MultiIndex

Since jaggedness is represented by the DataFrame's index, a single DataFrame can't contain two differently jagged collections.

```
>>> array = awkward.fromiter([{"a": [1, 2, 3], "b": [1.1, 2.2]},
                              {"a":         [], "b":       [3.3]},
                              {"a":    [4, 5], "b": [4.4, 5.5]}])

                                                          #       index values
>>> awkward.topandas(array["a"], flatten=True)            # -->    0  0      1
                                                          #            1      2
                                                          #            2      3
                                                          #         2  0      4
                                                          #            1      5

>>> awkward.topandas(array["b"], flatten=True)            # -->    0  0    1.1
                                                          #            1    2.2
                                                          #         1  0    3.3
                                                          #         2  0    4.4
                                                          #            1    5.5

>>> awkward.topandas(array, flatten=True)
ValueError: this array has more than one jagged array structure
```

Relationship to other packages

- Apache Arrow: columnar representation of nested data, memory management, shared memory, streaming, and I/O.
- XND: columnar representation of nested, cross-referenced data, generalizing Numpy ufuncs, math kernels, and GPUs.
- TensorFlow: maching learning, but it includes a `RaggedTensor` type.
- Zarr: array I/O and delivery, including a `VLenArray` (ragged array) because it's useful for genetics.
- ROOT: particle physics data analysis; I/O is columnar for better compression, but in-memory data are C++ class instances.

Relationship to other packages

- Apache Arrow: columnar representation of nested data, memory management, shared memory, streaming, and I/O.
- XND: columnar representation of nested, cross-referenced data, generalizing Numpy ufuncs, math kernels, and GPUs.
- TensorFlow: maching learning, but it includes a `RaggedTensor` type.
- Zarr: array I/O and delivery, including a `VLenArray` (ragged array) because it's useful for genetics.
- ROOT: particle physics data analysis; I/O is columnar for better compression, but in-memory data are C++ class instances.

Awkward Array's focus is *not* I/O or numerical math, it's the structure manipulation that physicists regularly need to analyze particle decays.

Examples of Awkward Array's focus:

how it solves various physics problems

Problem 1: mass of $A \to X + Y$ and $B \to X + X$ candidates

Traditional for loop:

```
for event in dataset:
    event.A = []
    for x in event.X:
        for y in event.Y:
            event.A.append(mass(x, y))
    event.B = []
    for i in range(len(event.X)):
        for j in range(i + 1, len(event.X)):
            event.B.append(mass(event.X[i], event.X[j]))
```

Array manipulation:

```
dataset["A"] = dataset["X"].cross(dataset["Y"])       # cross-join
dataset["B"] = dataset["X"].choose(2)                 # self-join
```

How are `cross` and `choose` implemented? Numpy tricks!

```
def cross(self, other):
    offsets = counts2offsets(self.counts * other.counts)
    parents = offsets2parents(offsets)
    indexes = numpy.arange(offsets[-1], dtype=int)

    ocp = other.counts[parents]              # no for loops
    iop = indexes - offsets[parents]         # fully vectorizable
    iop_ocp = iop // ocp

    left = self.content[self.starts[parents] + iop_ocp]
    right = other.content[other.starts[parents] + iop - ocp * iop_ocp]

    return JaggedArray.fromoffsets(offsets, Table(left, right))

def choose(self, n):
    # vectorized as well
    # (in fact, it's n <= 5 because there's no solution to the quintic!)
    ...
```

Problem 2: filtering $A \to X + Y$ candidates

Traditional for loop:

```
dataset.byevent = []
for event in dataset:
    event.bycandidate = []
    for a in event.A:
        if quality(a) > cut:
            event.bycandidate.append(a)
    biggest = None
    for a in event.A:
        if biggest is None or quality(a) > quality(biggest):
            biggest = a
    if biggest is not None and biggest > cut:
        dataset.byevent.append(biggest)
```

Array manipulation:

```
dataset["bycandidate"] = events["A"][events["A"] > cut]
dataset["byevent"]     = events["A"][events["A"].max() > cut]
```

Masking by an array (i.e. `numpy.compress`) → jagged masking

```
>>> x
<JaggedArray [[ 1.1,   2.2,  3.3], [  4.4,   5.5], [ 6.6,  7.7,  8.8]]>

>>> mask
array(       [        True,             False,             True      ])

>>> jaggedmask
<JaggedArray [[True, False, True], [False, False], [True, True, True]]>
```

Flat array of booleans does what Numpy does; jagged booleans selects within each inner list.

```
>>> x[mask]
<JaggedArray [[1.1 2.2 3.3] [6.6 7.7 8.8]]>

>>> x[jaggedmask]
<JaggedArray [[1.1 3.3] [] [6.6 7.7 8.8]]>
```

Problem 3: Select the best $A \to X + Y$ candidate

Traditional for loop:

```
for event in dataset:
    event.best = []                  # using an empty collection as None
    for a in event.A:
        if len(event.best) == 0 or quality(a) > quality(event.best[0]):
            event.best = [a]
```

Array manipulation:

```
argbest = quality(events["A"]).argmax()    # jagged array of integers

dataset["best"] = events["A"][argbest]
```

Problem 3: Select the best $A \to X + Y$ candidate

Traditional for loop:

```
for event in dataset:
    event.best = []                  # using an empty collection as None
    for a in event.A:
        if len(event.best) == 0 or quality(a) > quality(event.best[0]):
            event.best = [a]
```

Array manipulation:

```
argbest = quality(events["A"]).argmax()    # jagged array of integers

dataset["best"] = events["A"][argbest]


# remove empty lists and concatenate singletons by dropping offsets
only_the_best = dataset["best"].flatten()
```

Indexing by an array (i.e. `numpy.take`) → jagged indexing

```
>>> x
<JaggedArray [[ 1.1, 2.2, 3.3, 4.4], [5.5, 6.6], [7.7, 8.8, 9.9]]>

>>> index
array([-1, 0, 0])

>>> jaggedindex
<JaggedArray [[0, 0, -1], [0, 0, -1], [0, 0, -1]]>
```

Flat array of integers does what Numpy does; jagged indexes takes from each inner list.

```
>>> x[index]
<JaggedArray [[7.7 8.8 9.9] [1.1 2.2 3.3 4.4] [1.1 2.2 3.3 4.4]]>

>>> x[jaggedindex]
<JaggedArray [[1.1 1.1 4.4] [5.5 5.5 6.6] [7.7 7.7 9.9]]>
```

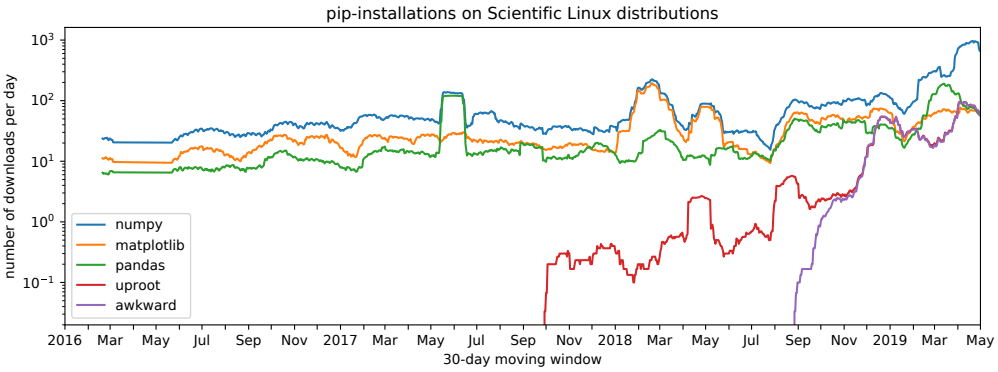
Maybe a domain-specific language is in order. . .

Problem statement (from a physicist):

"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 the mass closest to 91.2 GeV and 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"
                          titled "Leading other lepton pT"
}
```

Awkward Array project status



- Made a dependency of uproot (I/O library for particle physics) last September.
  ROOT file → Numpy or Awkward arrays, depending on structure (most are jagged).
- Several user-contributed methods, a dozen more inspired by user needs.

Current suite of nestable array types

`JaggedArray`	array containing unequal-length subarrays
`Table`	struct of arrays, presented as an array of structs
`ObjectArray`	creates Python objects on demand, such as `TLorentzVector`
`Methods`	vectorized methods, like `muons.delta_phi(muons.jet)`
`StringArray`	strings (jagged array of characters)
`IndexedArray`	"pointers" into another array (via integer indexes)
`SparseArray`	inverse of `IndexedArray`: zero everywhere except specified indexes
`MaskedArray`	marks elements as `None` with a byte array
`BitMaskedArray`	marks elements as `None` with a bit array
`IndexedMaskedArray`	indexes and masks in one array to avoid placeholders in the data array
`UnionArray`	contains multiple (enumerated) types
`ChunkedArray`	view discontiguous memory buffers as one array
`AppendableArray`	efficiently grow an array in chunks
`VirtualArray`	load data on demand

used to read ROOT files, used to read/write Arrow, used for both

lazy arrays are `ChunkedArrays` containing `VirtualArrays`

Rewrite/redesign in C and C++

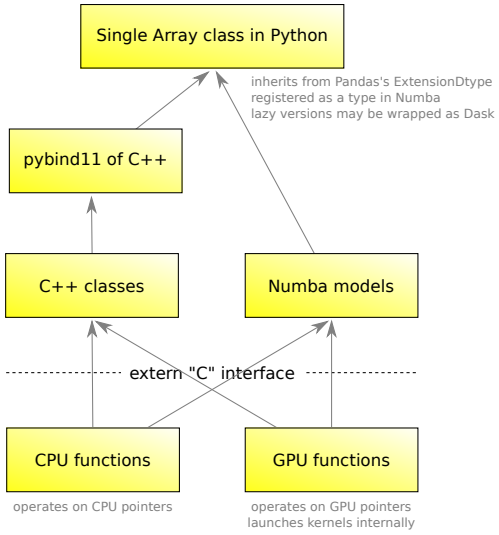
- We want versions in C++ (for interfaces) and Numba (to allow procedural programming), but keeping three versions in sync is beyond our resources.

Rewrite/redesign in C and C++

- We want versions in C++ (for interfaces) and Numba (to allow procedural programming), but keeping three versions in sync is beyond our resources.
- We also want to fix interface design flaws:
  - `A.cross(B)` versus `awkward.cross(A, B)`
  - should hide `JaggedArray`/`Table`/`MaskedArray`/etc. structure in a single `Array` class.

Rewrite/redesign in C and C++

- We want versions in C++ (for interfaces) and Numba (to allow procedural programming), but keeping three versions in sync is beyond our resources.
- We also want to fix interface design flaws:
  - `A.cross(B)` versus `awkward.cross(A, B)`
  - should hide `JaggedArray`/`Table`/`MaskedArray`/etc. structure in a single `Array` class.
- So we're rewriting it in layers $\longrightarrow \longrightarrow \longrightarrow$

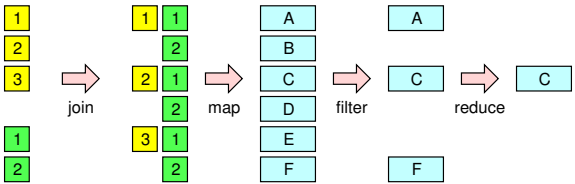


Is this only for particle physics?

The existence of Arrow and XND suggest that it's not: "nested data structures" can hardly be domain-specific.

Is this only for particle physics?

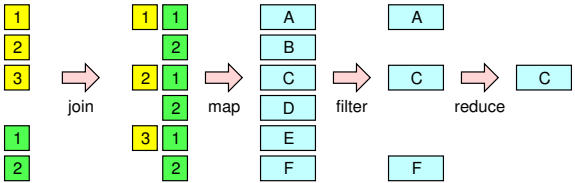
The existence of Arrow and XND suggest that it's not: "nested data structures" can hardly be domain-specific.



The join-map-filter-reduce cycle is important in genetics, too: at each site of a genetic sequence, a combinatorial number of allele candidates must be considered (join), uncertainties computed (map), and minimized (reduce).

Is this only for particle physics?

The existence of Arrow and XND suggest that it's not: "nested data structures" can hardly be domain-specific.



The join-map-filter-reduce cycle is important in genetics, too: at each site of a genetic sequence, a combinatorial number of allele candidates must be considered (join), uncertainties computed (map), and minimized (reduce).

`awkward.fromiter` could be used with `simdjson` to convert huge JSON datasets into Awkward Arrays. Do you have big JSON datasets to analyze?

Thanks!

Many people have been enormously helpful in this work:

- Jaydeep Nandi (GSoC): clever algorithms for `cross`, `pairs`, and jagged reduce
- Charles Escott (GSoC): investigation into pybind11 and C++
- The Coffea Team (Fermilab): extensive feedback from use in CMS physics
- Nick Smith (Fermilab): clever algorithms for `choose`, `argmax`, `argsort`
- Jonas Rembser (LLR-École Polytechnique): clever algorithms for `concatenate`
- Mason Proffitt (U. Washington): stress tests, bug reports, and fixes
- Henry Schreiner (Princeton): tests and feedback on HDF5 persistence
- Joosep Pata (Caltech): exploring Awkward Array on GPUs
- Alistair Miles and Gavin Band (Oxford): conversations about Zarr and genetics
- The XND Team: conversations about scope and direction of XND
- Wes McKinney and Julien Le Dem: conversations about Apache Arrow
- National Science Foundation (grants ACI-1450377 and PHY-1624356)